Reproducibility of Experiments in Concurrent Programming

Vincent Gramoli
Roadmap

• Why reproducibility?

• Towards the reproducibility of an experiment?

• Concurrent experiments cannot be reproduced

• How to do better?

• Understanding the impact of the workload

• Hands-on Session
Corroboration

- Science moves forward by corroboration
Science advances faster when people waste less time pursuing false leads. [...] There is growing alarm about results that cannot be reproduced.

Explanations include increased levels of scrutiny, complexity of experiments and statistics, and pressures on researchers.

Journals, scientists, institutions and funders all have a part in tackling reproducibility.

http://www.nature.com/news/reproducibility-1.17552
## Software and System Evaluation

<table>
<thead>
<tr>
<th>Conference</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOPSLA</td>
<td>2013</td>
</tr>
<tr>
<td>ECOOP</td>
<td>2013</td>
</tr>
<tr>
<td>POPL</td>
<td>2014</td>
</tr>
<tr>
<td>PLDI</td>
<td>2014</td>
</tr>
<tr>
<td>SAS</td>
<td>2014</td>
</tr>
<tr>
<td>PPoPP</td>
<td>2015</td>
</tr>
<tr>
<td>CGO</td>
<td>2015</td>
</tr>
<tr>
<td>CAV</td>
<td>2015</td>
</tr>
<tr>
<td>ADAPT</td>
<td>2015</td>
</tr>
</tbody>
</table>

*The ‘Since’ column indicates the year the conference is held*
Evaluated Artifact

1. **Consistent** with the paper: does the artifact substantiate and help reproduce the claims of the paper

2. **Complete**. What is the fraction of the results that can be reproduced

3. **Well documented**. Does the artifact describe and demonstrate how to apply the presented method to a new input?

4. **Easy to reuse**. How easy is it to reuse the provided artifact?
Evaluated Artifact
Ok, I want my experiments to be reproducible, but how can I do?
Experimenting

Computing

Visualizing

Aggregating
Experimenting cont’d

benchmark.java ➔ Makefile ➔ run.sh ➔ log.dat

graph.pdf ← plot.gp ← stats.dat ← process.sh
List experimental settings

- Compile flags:
  - gcc -O3 -std=gnu11

- Environment variables:
  - LD_PRELOAD: what memory allocator is called?
  - JAVA_HOME: what version of Java?

- Runtime options (JVM HotSpot):
  - -server
  - -XX:+UseBoundThreads (Bind user threads to Solaris kernel threads)
  - -Xoptimize (Use the optimizing JIT compiler)

- Median, average, min, max, standard deviation, 99%-ile
Example: Script `run.sh`

```bash
#!/bin/bash
DIR=..
OUT=../log
BIN=../bin
OPTS=-server

# settings
THREADS="1 2 4 8 16 32 64"
SIZES="1024 4096 16384 65536"
UPDATES="0 10 20 30 40 50 60 70 80 90 100"
LENGTH="5000"
ITERATIONS=5
WARMUP=2
DATE=`date`
COMP=`uname -a`

# benchmarks
SYNCS="Sequential LockBased LockFree Transactional"
ALGOS="Tree Queue SkipList HashTable Array LinkedList"
TYPES="Map Set Collection"
MAINCLASS=contention.benchmark.Test

for sync in ${SYNCS}; do
    for algo in ${ALGOS}; do
        for type in ${TYPES}; do
            for update in ${UPDATES}; do
                for thread in ${THREADS}; do
                    for size in ${SIZES}; do
                        r=`echo "2*${i}" | bc`
                        out=${output}/log/${bench}-lockfree-i${i}-u${write}-t${t}.log
                        bench=${sync}${algo}${type}
                        echo "# " ${TIME} "-" ${COMP} "-" ${JAVA_HOME} "-" ${PATH} > $out
                        for (( j=1; j<=${ITERATIONS}; j++ )); do
                            java ${OPTS} -cp ${CP} ${MAINCLASS} -W ${WARMUP} \
                                -u ${[update]} -d ${[i]} -t ${[t]} -i ${[i]} \ 
                                -r ${[r]} -b ${[bench]} 2>&1 >> $out
                        done
                        done
                    done
                done
            done
        done
    done
done
```

12
Backup everything

- Output:
  - Keep the output logs even though you are interested in the mean/median/min/max

- Aggregation:
  - Keep the intermediary data (after aggregation)

- Visualization:
  - Keep the visualization scripts (gnuplot)

- Use version management system
  - git, svn, blockchain, etc.
Take existing benchmarks

1. Write README.md file, INSTALL file, doc, HOWTOs...

2. Make a screencast available

3. Package into virtualized environment (virtual box, docker), with OS, libraries, benchmarks...
Experiments of Concurrent Programs cannot be Reproduced
Reproducing Concurrency

- Multi-threading is non-deterministic
  - Replaying an execution leads to a different interleaving
- The notion of “Thread” is tied to the OS
  - Lightweight processes, Java threads, POSIX threads...
- The notion of core/memory is tied to the architecture
  - single-/multi-socket, memory controllers...
- Have you tried to reproduce concurrency bugs?
Reproducing Concurrency

- Multi-threading is non-deterministic.
- Replaying an execution leads to different interleaving.
- The notion of "Thread" is tied to the OS.
- Lightweight processes, Java threads, POSIX threads...
- The notion of core/memory is tied to the architecture.
- Single-/multi-socket, memory controllers...
- Have you tried to reproduce concurrency bugs?

Identify the causes of your results.

Not an excuse!
The List-Based Set

- Sorted linked list with sentinel head “H” and tail “T”

- Implements a integer set (insert/remove/contains)

```
H -> 1 -> 3 -> 5 -> T
```

- A value belongs to the set if it is the key of a node reachable from the head, e.g., \{1, 3, 5\}
The List-Based Set (con’t)

- Multiple threads execute operations concurrently

- Intuitively, modifications by concurrent threads of non-adjacent nodes should not conflict
Lack of documentation

List-based set, Java8, 10% updates, avg 10 runs of 10s

Throughput (ops / ms) vs Number of threads
List-based *set*, Java8, 10% *updates*, avg 10 runs of 10s

![Graph showing throughput (ops / ms) vs. Number of threads]
Lack of documentation

List-based set, Java8, 10% updates, avg 10 runs of 10s

what if insert(v) fails updating because v ∈ set?
Lack of documentation

List-based set, Java8, 10% updates, avg 10 runs of 10s

same JVM iterations?
Lack of documentation

List-based set, **Java 8**, 10% updates, avg 10 runs of 10s

how much time to warmup the JIT compiler?

Throughput (ops/ms)

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Lack of documentation

List-based set, Java8, 10% updates, avg 10 runs of 10s

![Graph showing throughput (ops / ms) vs number of threads]

When does hyperthreading kicks in? At 2 or 5?
Lack of documentation

List-based set, Java 8, 10% updates, avg 10 runs of 10s

how does it compare to sequential code performance?
How can we do better?
Take existing benchmarks

1. Do not reinvent the wheel
2. List the parameters
3. Explain the experimental settings
Synchrobench [PPoPP’15]
Synchrobench (con’t)

1. Written in **Java** and **C/C++**

2. Run on **x86(-64), SPARC, Power8, Tilera...**

3. Compare **CAS, TM, Mutex, CopyOnWrite, RCU, Spinlock, no synchronization...**

4. Benchmark w/ **trees, skip lists, linked lists, queues, arrays, hash tables...**
Synchrobench (con’t)

Main parameters:

- **b**: benchmark to run
- **u**: update ratio (%) ($\#\text{insert} + \#\text{delete})/\#\text{contains}$
- **f**: are updates effective?
- **i**: initial size
- **t**: #threads
- **d**: duration (ms) of the benchmark
- **n**: #iterations in one JVM instance
- **W**: warmup (seconds)
Architecture Impact

- **AMD** Turbo Core
- **Intel** Turbo Boost
Architecture Impact

Report the throughput with 1 thread in different JVM

1. without Turbo Boost
2. with Turbo Boost

Throughput (op/s)

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>w/o TB</th>
<th>w/ TB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6,000,000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7,500,000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9,000,000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10,500,000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12,000,000</td>
<td></td>
</tr>
</tbody>
</table>

2x8-HT-Intel-Xeon—W0-t1-d5000-u40-i0-r50-bVersionedListSet
Architecture Impact

**Simultaneous Multi-threading**: allows to have threads sharing pipeline, CPU, and caches and execute multiple instructions per cycle on a processor.
Architecture Impact

Simultaneous Multi-threading [CACM’11]

Performance of hyperthreading over no hyperthreading on Intel Xeon CPUs
Operating System Impact

Thread can be bound to Solaris lightweight processes (LWP) on the HotSpot JVM. When a LWP for thread is created, the kernel assigns a thread to a locality group (or lgroup)

- In Solaris 10 threads balanced over dies, then over cores, then over pipelines
- In Solaris 11 threads are grouped on a unique process on the same lgroup until the workload exhausts half of the resources of this lgroup.

https://blogs.oracle.com/dave/entry/thread_placement_policies_on_numa
Operating System Impact

• Thread pinning \[\text{[PPoPP'15]}\]

- **AMD**
  - Params: u10-f1-l5000-s0-a0-i[16384..65536]-r[32768..132072]-W20-

- **COMP**: compact pinning policy (same NUMA-node first)
- **SCAT**: scatter pinning policy (different NUMA node first)
To compare-and-set two values atomically:

- sun.misc.Unsafe (Java 8–)

- Java AtomicMarkableReference stores the mark and reference separately, requiring an extra read to return the reference.

- Bitmask in C/C++:
  - ref = word & ~(uintptr_t) 1
  - mark = word & (uintptr_t) 1
Prog. Lang. Impact

Harris linked list (Algo.21) [DISC’01]

• Use Java version using AtomicMarkableReference

• C/C++ version using bitmask

2x8-HT-Intel-Xeon—d5000-u50-f1-i65536-r132768-bNonBlockingLinkedListSet
Workload Impact
Lazy list-based set

Grab 2 fine-grained locks per update operation [OPODIS’05]

- The lazy linked list grabs no lock during traversal

- But its insertion (resp. deletion) grabs locks before detecting that an element is present (resp. absent)

- This rejects schedules in read-only executions
Lazy list-based set

Traverse; lock; validate; try-update; unlock

1. Find the position to insert/delete

2. lock the pred and current node

3. validate:
   - check neither pred nor curr are deleted
   - check that pred points to curr

4. try inserting or deleting the node

5. unlock
Lazy list-based set

\textit{insert(4): traverse}

![Diagram showing the process of inserting 4 into a list-based set]
Lazy list-based set

`insert(4): traverse; lock`
Lazy list-based set

\texttt{insert(4): traverse; lock; validate}

![Diagram showing a linked list with nodes 3 and 5, and questions about whether they are still linked and deleted.](image)
Lazy list-based set

\[\text{insert}(4): \text{traverse;} \text{ lock;} \text{ validate;} \text{ try-update}\]
Lazy list-based set

insert(4): traverse; lock; validate; try-update; unlock
Lazy list-based set

Why is the validation necessary after locking?
Lazy list-based set

Why is the validation necessary after locking?

Between the time pred/curr are found and the time they are locked...

...some other threads could:

- delete pred,
- delete curr or
- insert between pred and curr
Lazy list-based set

Why is the validation necessary after locking?

First value > 4
Lazy list-based set

Why is the validation necessary after locking?
Lazy list-based set

Why is the validation necessary after locking?
Lazy list-based set

Why is the validation necessary after locking?
Lazy list-based set

What happen if an update returns unsuccessfully?
Lazy list-based set

What happen if an update returns unsuccessfully?
Lazy list-based set

What happen if an update returns unsuccessfully?
Lazy list-based set

What happens if an update returns unsuccessfully?

Lost of stalls: reduced concurrency
Lazy list-based set

What happen if an update returns unsuccessfully?

Is this necessary to lock without effective update?
Lazy list-based set

Traverse; lock; validate; try-update; unlock

1. Find the position to insert/delete
2. lock the pred and current node
3. validate:
   - check neither pred nor curr are deleted
   - check that pred points to curr
4. try inserting or deleting a node
5. unlock
Lazy list-based set

Traverse; lock; validate; try-update; unlock

1. Find the position to insert/delete

3. validate:
   - check neither pred nor curr are deleted
   - check that pred points to curr

2. lock the pred and current node

4. try inserting or deleting a node

5. unlock
Lazy list-based set

Traverse; lock; validate; try-update; unlock

1. Find the position to insert/delete

3. validate:
   - check neither pred nor curr are deleted
   - check that pred points to curr

2. lock the pred and current node

4. try inserting or deleting a node

5. unlock

Cannot work with traditional locks

Valid!
Versioned list-based set

• The versioned list uses a pre-locking validation [DISC’15]

• It uses a versioned try-lock \langle\text{ver, lock}\rangle per node with ver = \#acquirements

• Traversals do not acquire versioned try-locks

• Once the position is found:
  • The update validates by reading the version
  • An insert(v) (resp. delete(v)) grabs the lock only if v is absent (resp. present) and version is unchanged
Versioned list-based set

\text{insert}(4)

First value \(> 4\)
Versioned list-based set

\texttt{insert(4)}

Record version: 0
Versioned list-based set

insert(4)

value != 4?

version of pred: 0
Versioned list-based set

$\text{insert}(4)$

Not deleted?

Still linked?

version of pred: 0
Versioned list-based set

\textbf{insert(4)}

version of pred: 0
Versioned list-based set

insert(4)

compare...

version of pred: 0
Versioned list-based set

$\text{insert(4)}$

...and swap
Versioned list-based set

\text{insert}(4)
Versioned list-based set

Traverse; validate; try-lock; update; unlock

1. Find the position to insert/delete
2. record version of pred
3. validate:
   - check neither pred nor curr are deleted
   - check that pred points to curr
4. lock the pred and current node
5. try inserting or deleting a node
6. unlock
Versioned list-based set

What happen if an update returns unsuccessfully?
Versioned list-based set

What happen if an update returns unsuccessfully?
Versioned list-based set

What happens if an update returns unsuccessfully?

value $\neq 4$?
Versioned list-based set

What happen if an update returns unsuccessfully?

No, value=4
Effective Updates

In the lazy list:

All attempted updates grab locks

This is necessary to guarantee correctness

In the versioned list:

Only effective updates grab locks

The versioned list is concurrency-optimal [DISC'15]: There is no list that accepts more correct schedules of memory accesses
Hands-on Session
Install Synchrobench

1. wget https://github.com/gramoli/synchrobench/archive/master.zip
2. unzip master.zip
3. cd synchrobench-master/java
4. ant jarbench
Compute

• Run the following code 5 times

```
java -server -cp bin\
contention.benchmark.Test -W 0 -t 2 -d 5000 -u 40 -i 0 -r 50 -b \linkedlists.lockbased.VersionedListSet -n 0
```

• Run the following code once with -n 5

```
java -server -cp bin\
contention.benchmark.Test -W 0 -t 2 -d 5000 -u 40 -i 0 -r 50 -b \linkedlists.lockbased.VersionedListSet -n 5
```

• What do you observe, regarding the ‘Throughput’
JIT optimizations

Report the throughput with 2 threads & 5 iterations:

1. as part of the same JVM instance (-n 5)
2. as part of separate instances

Throughput (op/s)

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 x java-n0</td>
<td>5,100,000</td>
<td>5,135,000</td>
<td>5,170,000</td>
<td>5,205,000</td>
<td>5,240,000</td>
</tr>
<tr>
<td>1 x java-n5</td>
<td>5,100,000</td>
<td>5,135,000</td>
<td>5,170,000</td>
<td>5,205,000</td>
<td>5,240,000</td>
</tr>
</tbody>
</table>

2x8-HT-Intel-Xeon—W0-t2-d5000-u40-i0-r50-bVersionedListSet
• Run the following code with different thread counts 1, 2, 4, 8.

java -server -cp lib/compositional-deucestm-0.1.jar \ contention.benchmark.Test -W 0 -t 1 -d 5000 -u 40 -i 0 -r 50 -b \ linkedlists.lockbased.LazyLinkedListSortedSet

java -server -cp lib/compositional-deucestm-0.1.jar \ contention.benchmark.Test -W 0 -t 1 -d 5000 -u 40 -i 0 -r 50 -b \ linkedlists.lockbased.VersionedListSet
 Aggregate

- Report the “Throughput (ops/s): 1731921.1783” in a table where columns are <VersionedList, LazyList> and rows are <1,2,4,8> threads

<table>
<thead>
<tr>
<th>#threads</th>
<th>Versioned List</th>
<th>Lazy List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4523706.846</td>
<td>2940885.842</td>
</tr>
<tr>
<td>2</td>
<td>4895072.451</td>
<td>2415960.232</td>
</tr>
<tr>
<td>4</td>
<td>5661667.679</td>
<td>2266188.053</td>
</tr>
<tr>
<td>8</td>
<td>6837793.548</td>
<td>2026012.515</td>
</tr>
</tbody>
</table>
Visualize

• Using gnuplot, excel, open office or google sheets, draw a graph to compare the performance

• What do you observe? Why?
Results

• The lazy list is overly conservative while the versioned list is concurrency-optimal [DISC’15]

• Difference in performance is exacerbated with the contention (4x16-core 2.4GHz AMD Proc. 6378, Ubuntu 14.04.3 LTS, Java 1.7.0_95 OpenJDK 64-Bit Server) => documenting workloads is crucial
The Slide to Remember

1. Use well-documented benchmarking tools

2. Gather your sources (code, scripts, logs and aggregated dataset), list the settings (OS, architecture, environment, runtime observations, workloads)

3. Package everything, share your code and submit your artifact
See you in Sydney
References


