Java Experiments on MTL

From past mistakes to best practices

3/3/2015
Trevor Brown
PhD student at U of T
tabrown@cs.utoronto.ca

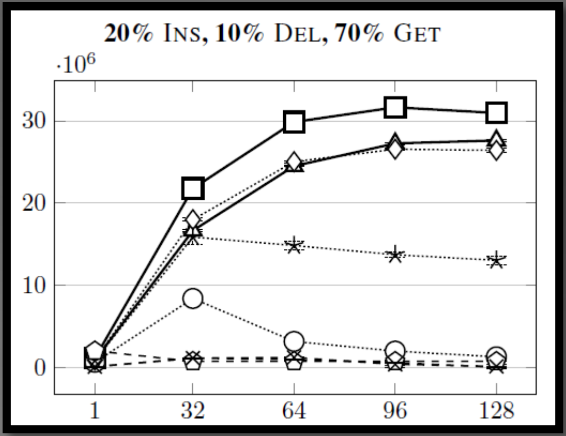
Outline

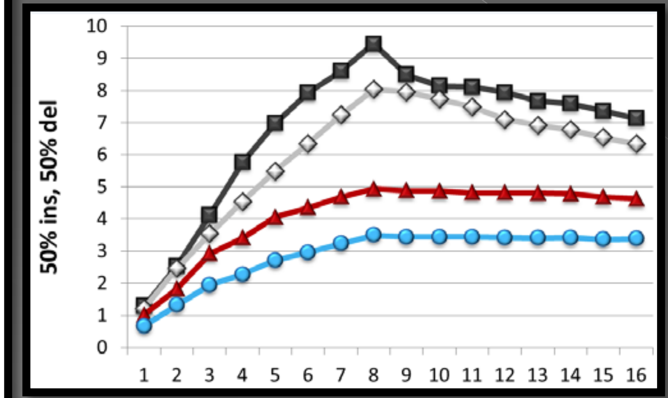
- Designing experiments
- Performing experiments in Java
- Intel’s Manycore Testing Lab

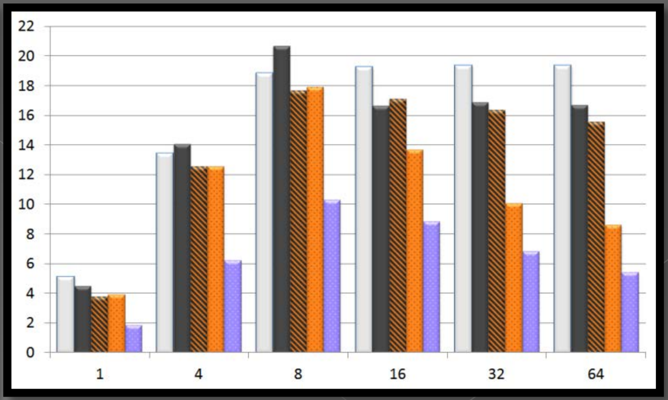
Designing experiments

Designing experiments: the goal

- High quality results that capture, e.g.,
  - How an algorithm scales
  - Which of several algorithms performs best
- Pretty graphs



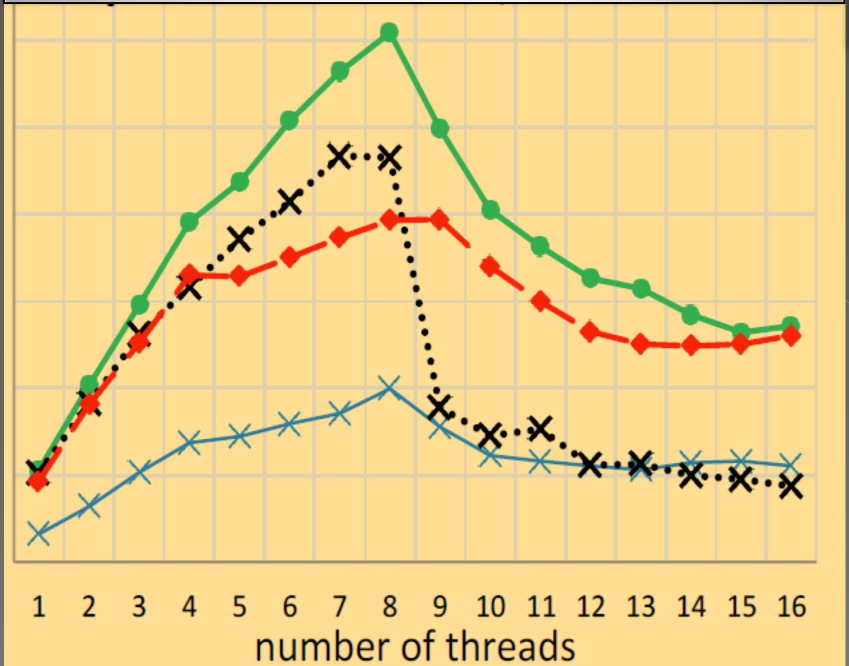




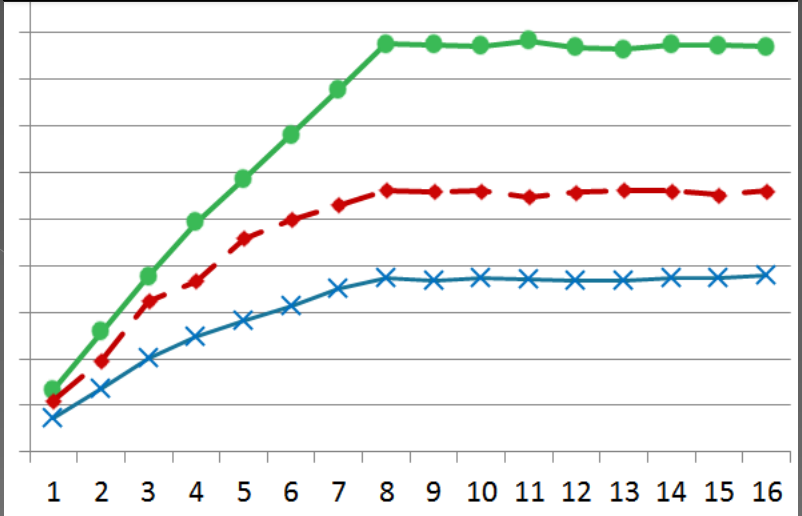
- Ability to explain behaviour on graphs

Explaining behaviour is important

Graph from a published paper



Graph produced by re-running the authors' experiments



- Authors did not explain negative scaling
- There was a bug in their test setup

Common experimental setups

- Standardized benchmarks
  - Example: SPECjvm2008 – measuring the performance of Java runtime environments
  - Only exist for some problems
- Macrobenchmarks
  - Replace an algorithm or data structure in a large software package
  - Perform experiments on the result
- *Microbenchmarks: apply randomized workloads to an algorithm or data structure*

Performing microbenchmarks

- Goal: show how algorithms scale and/or perform relative to one another
- Make graphs that plot performance versus number of concurrent threads
- Each graph shows the result of one **experiment**
- Each data point on a graph is an average of a set of randomized **trials**

A typical **trial** for a **data structure**

- Each thread:
  - Runs for a fixed length of time
  - Performs random operations according to some predefined probability distribution
    - E.g., 25% insertion, 25% deletion and 50% searching in a list
  - Records the number of operations completed in the allotted time

A poorly designed **trial**

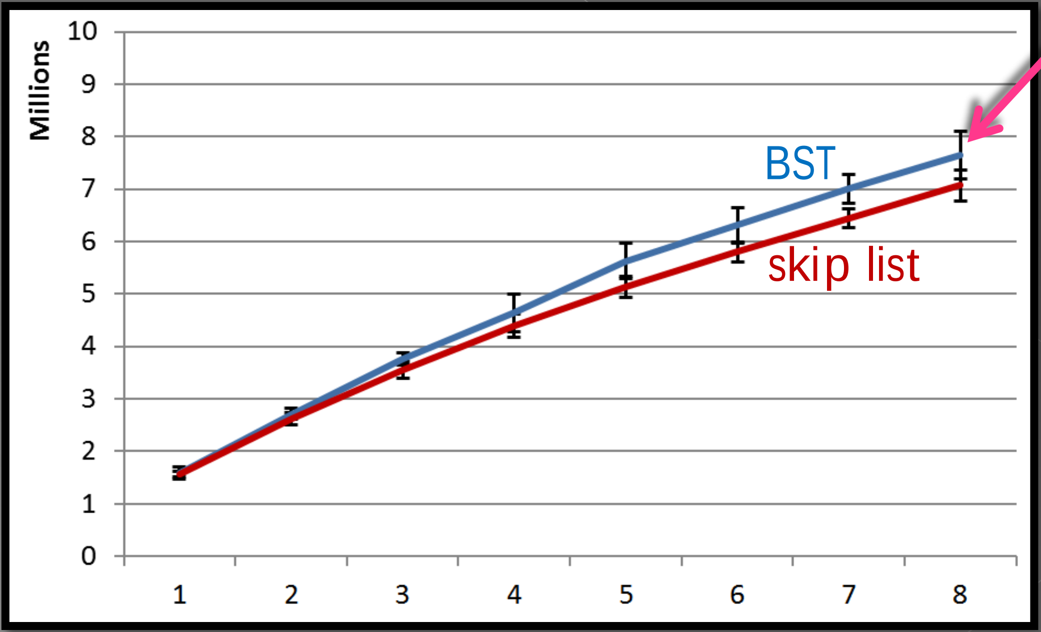
- Each thread:
  - Performs **M** operations and then stop
  - Records the time when the last thread stops
- In this type of trial, one thread can finish long after the others
- This can make it seem like much more time is needed to perform **M** operations

Steady states

- Some algorithms and data structures will reach a **steady state** for some workloads
- Example: consider a binary search tree T that stores keys in the range [0, 1000)
- Suppose the expected workload is 50% insertion and 50% deletion
- In expectation, T will contain 500 keys after it has been in use for a long time

Steady states are important

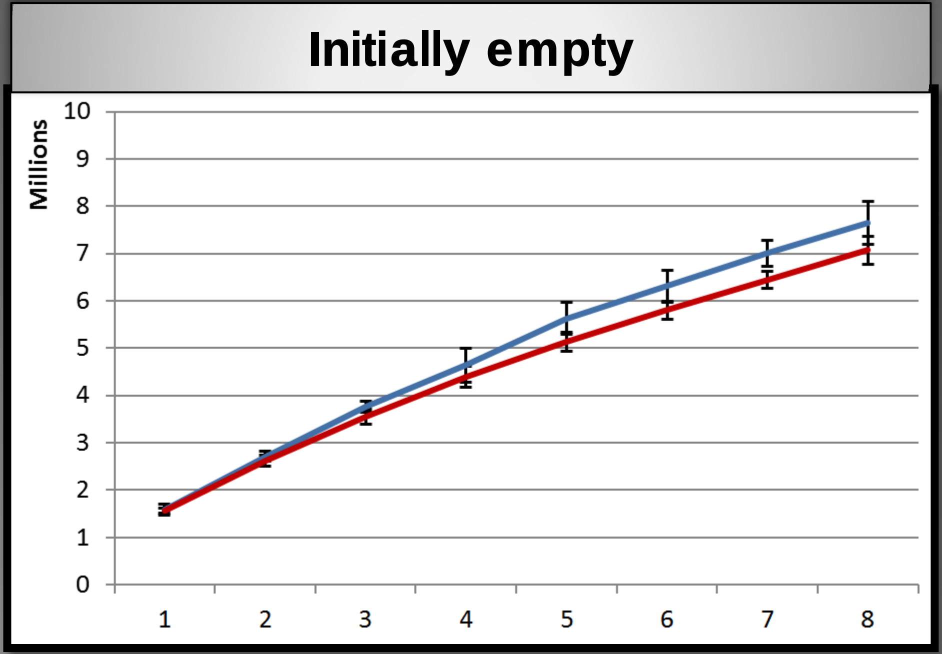
- Example: consider a **BST** and a **skip list** that store keys in the range [0, 1000000)
- Running 1 second trials with 50% insertions and 50% deletions, we get:

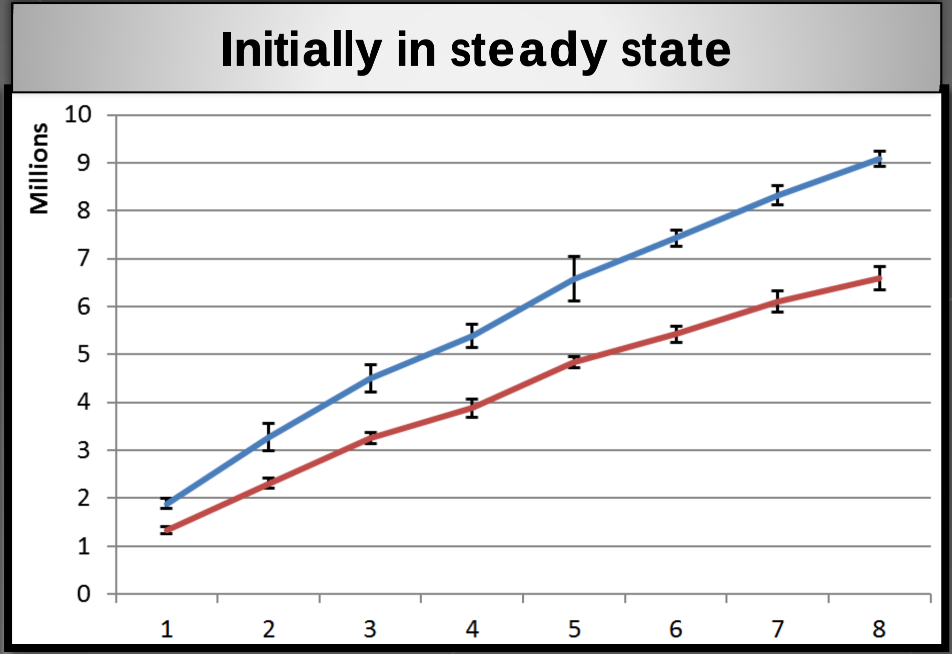


95% confidence interval

Steady states are important

- Suppose we prefill the data structures to their steady states (~500,000 keys)
- Then, running 1 second trials with 50% insertions and 50% deletions, we get:


Initially empty


Initially in steady state

Memory allocation affects the steady state

- Consider an algorithm that:
  - Allocates a large amount of memory in the early stages of its execution
  - Then, simply reuses that memory (never allocating any more)
- If trials are very short, then memory allocation overhead is large
- However, in an infinite execution, the amortized cost of allocation is zero

Other steady state factors

- Memory reclamation and deallocation
- Processor cache occupancy
- Experimental design
  - E.g., choice of workload (75% insertion & 25% deletion has a different steady state than 50% insertion & 50% deletion)
- Properties of algorithms / data structures

Reaching a steady state

- May have to run trials for a very long time to reach a steady state
- Time constraints might prevent this
- Goal: find a way to run short trials that give the same answers as long trials
  - Always sanity check by running some very long trials to see if the results are different

Performing experiments in Java

Running an experiment – attempt 1

Main thread

- **state** = pending
- Create & start threads
- Wait on a barrier **b**
- **state** = running
- Sleep() for S seconds
- **state** = done
- Print **#ops / S**

Every other thread

- Perform initialization
- Wait on barrier **b**
- Wait until **state** == running
- Loop
    - Perform random op
    - If **state** == done, then terminate

- Problem: Sleep() might sleep for longer than S seconds

Running an experiment – attempt 2

Main thread

- **start** = null
- Create & start threads
- Wait on a barrier **b**
- **start** = System.nanoTime()
- Sleep() for S seconds
- **end** = System.nanoTime()
- Print **#ops** / ((**end**-**start**)/10^9)

Every other thread

- Perform initialization
- Wait on barrier **b**
- Wait until **start ≠ null**
- Loop
  - Perform random op
  - If **end ≠ null** then halt

- Problem: if the main thread is context switched out after reading the current time, but before writing to **start**, then threads are timed while waiting **start** to be written

Running an experiment – attempt 3

Main thread

- **start** = null
- **state** = pending
- Create & start threads
- Wait on a barrier **b**
- **state** = running
- Wait for threads to halt
- **end** = max{$\mathbf{end}_p$}
- Print **#ops** / ((**end** - **start**)/10^9)

Every other thread p

- Perform initialization
- Wait on barrier **b**
- Wait until **state** ≠ pending
- $\mathbf{start}_p$ = System.nanoTime()
- CAS(**start**, null, $\mathbf{start}_p$)
- Loop
  - Perform random operation
  - $\mathbf{end}_p$ = System.nanoTime()
  - if $\mathbf{end}_p$ – **start** > $10^9 \cdot S$ then halt

- Problem: $\mathbf{end}_p$ can be *much* more than **S** seconds after **start** if p sleeps just before calling nanoTime

Running an experiment – attempt 4

Good idea to call
System.gc() here

java.util.concurrent.CyclicBarrier

Main thread

- **start** = null
- **state** = pending
- Create & start threads
- Wait on a barrier **b**
- **state** = running
- Wait for threads to halt
- **end** = max{$\mathbf{end_p}$}
- Print **#ops** /
  ((**end** - **start**)/10^9)

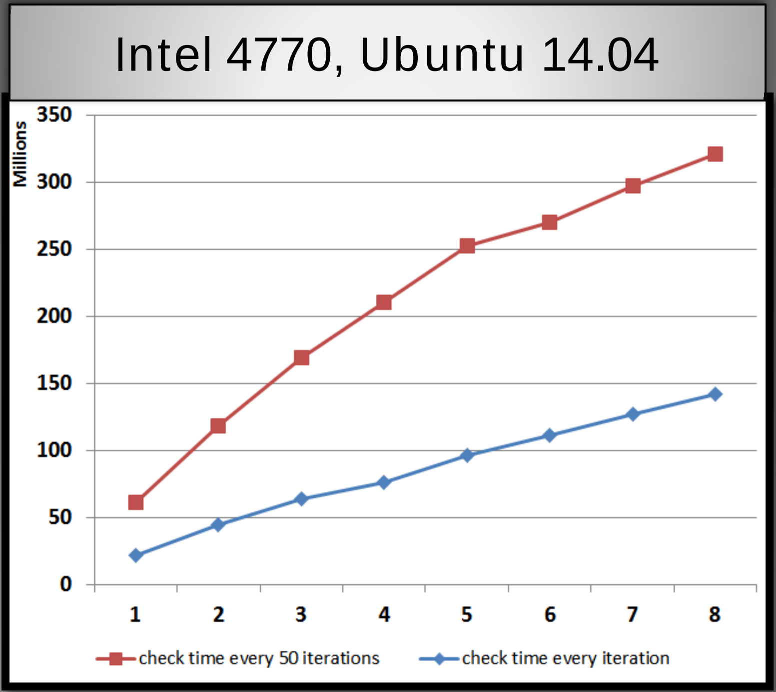
Eve

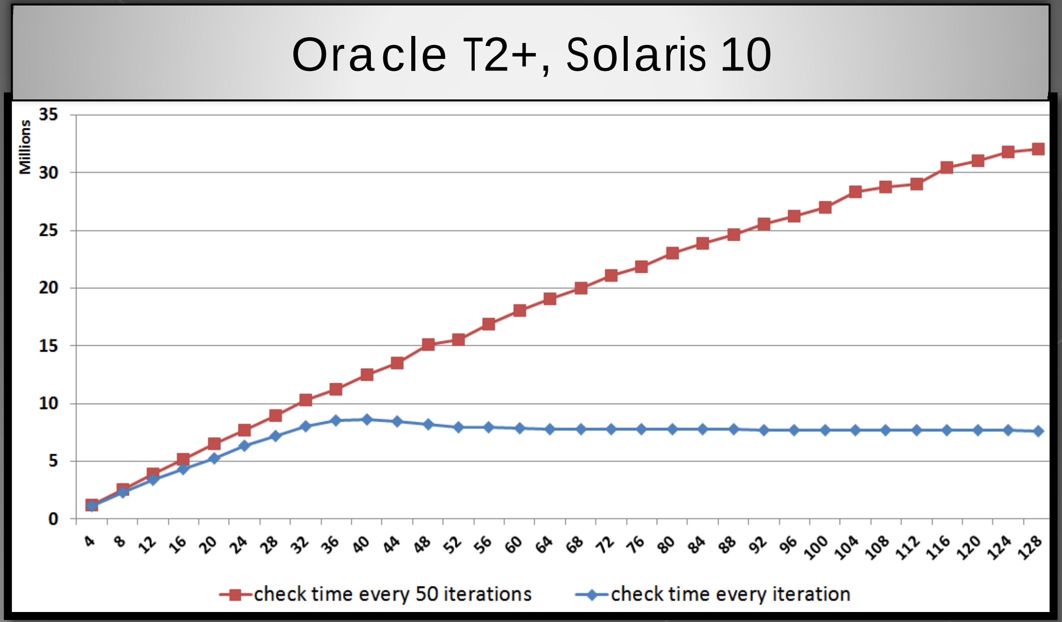
- Perform initialization
- Wait on barrier **b**
- Wait until **state** ≠ pending
- $\mathbf{start_p}$ = System.nanoTime()
- CAS(**start**, null, $\mathbf{start_p}$)
- Loop
  - Perform random operation
  - **t** = System.nanoTime()
  - if **t** – **start** > $10^9 \cdot \mathbf{S}$ then halt
    else $\mathbf{end_p}$ = **t**

- Lemma: $\mathbf{end_p}$ is at most **S** seconds after **start**, and is captured between **p**’s last two operations.

Cost of nanoTime varies

- On Solaris, nanoTime performs a CAS, which can *severely* limit scaling
- On Ubuntu, nanoTime has significant overhead, but affects scaling less


Intel 4770, Ubuntu 14.04

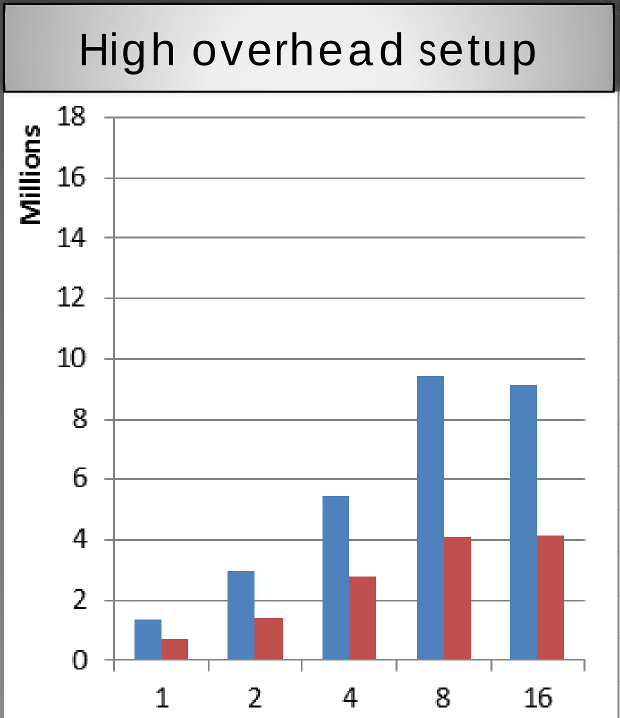

Oracle T2+, Solaris 10

Experimental setup overhead

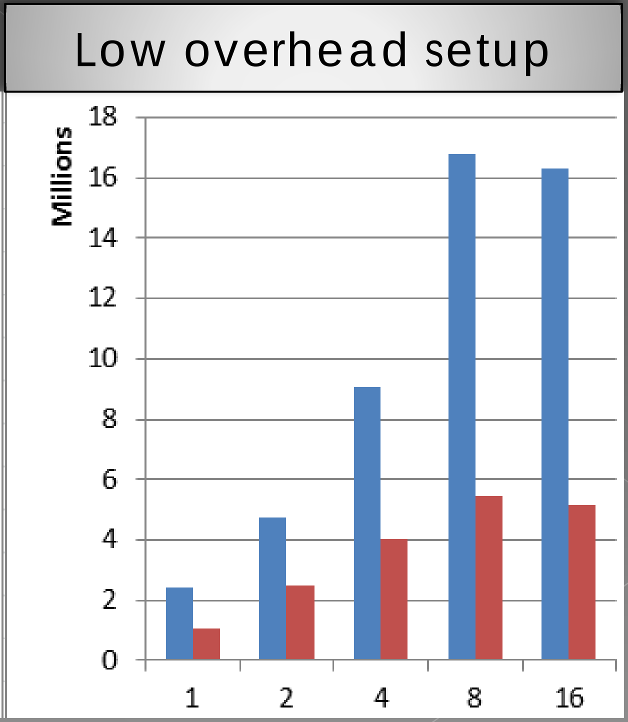
- Create a dummy data structure with operations that do nothing
- Measure its performance to check the overhead of your test harness
- A low overhead test harness is vital when testing short, simple operations

Why overhead matters

- If an algorithm's performance is limited by the overhead of your experimental setup, it cannot be evaluated fairly!



High overhead setup

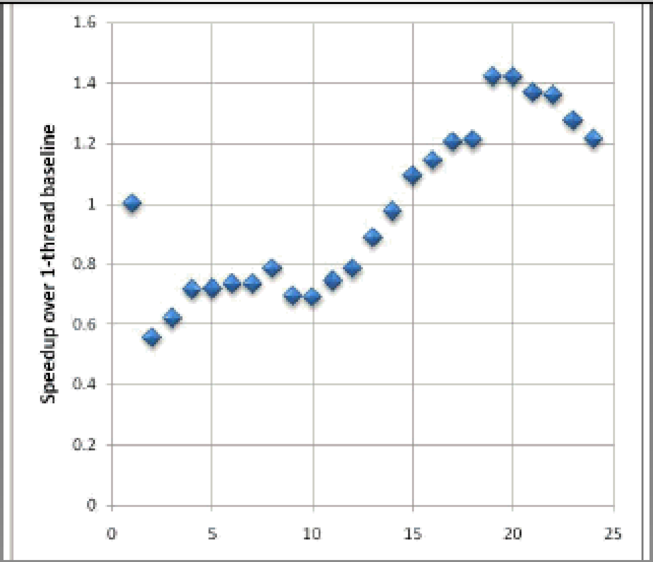


Low overhead setup

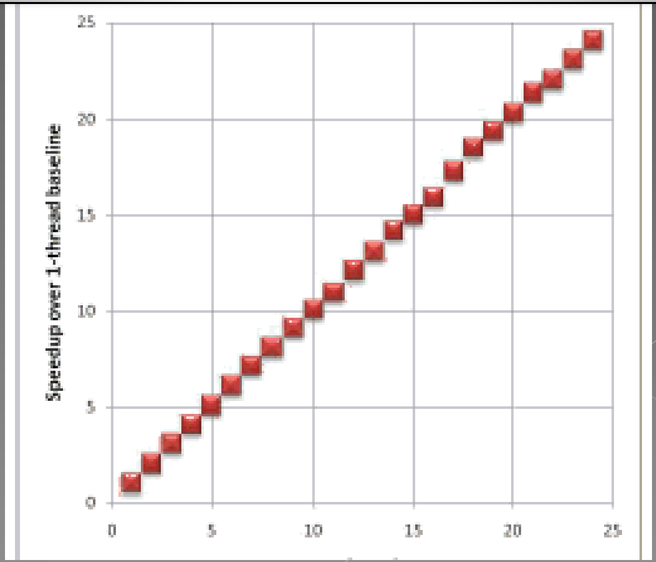
False sharing

- Trivial parallel algorithm:
  - divide a random matrix into equal parts, one for each thread
  - each thread counts odd entries in its part

Counters: `long count[n];`
Thread i does: `++count[i];`

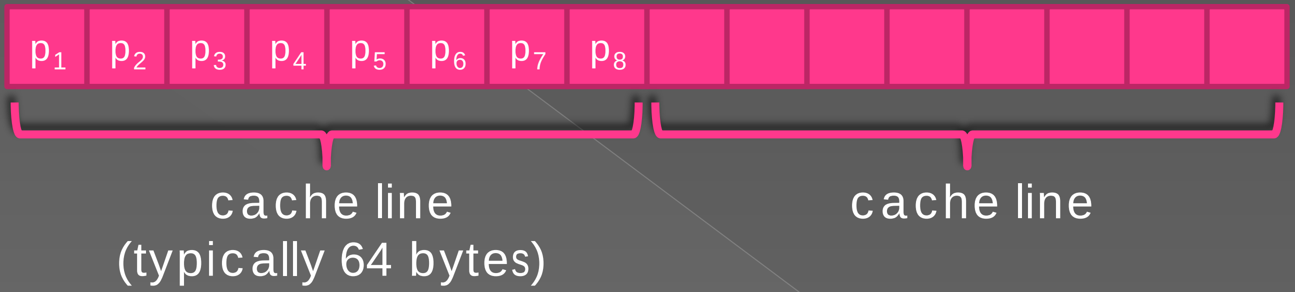


Counters: `long count[n*8];`
Thread i does: `++count[i*8];`



False sharing

- Naïve arrays with one slot of private data per thread can cause contention!



p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8

cache line
(typically 64 bytes)

cache line

- When p_1 writes to its slot, it invalidates the entire cache line on all CPU cores
- Solution: only one slot per cache line

p_1 p_2 p_3 …

  - Requires much more space

Be careful with auto boxing

- Consider the following toy Java class:

```
class SingleCell<K> {
    K value;
    boolean set(K key) { value = key; }
}
```

- How much memory is used by this code?

```
    SingleCell<Integer> cell = new SingleCell<>();
    for (int i = 0; ; ++i) cell.set(i % 100);
```

- When i%100 is passed to set(), an Integer object with the value i%100 is created
  - This is called auto boxing

What is this type?

Which type does set() require?

Boxing is expensive

- Running this code for 3 seconds produces more than **15GB** of garbage **Integer** objects

```
[GC (System.gc())  235929K->20378K(15073280K), 0.0165442 secs]
[Full GC (System.gc())  20378K->20182K(15073280K), 0.1702617 secs]
starting trial...
[GC (Allocation Failure)  3952342K->20382K(15073280K), 0.0010818 secs]
[GC (Allocation Failure)  3952542K->20286K(15073280K), 0.0006483 secs]
[GC (Allocation Failure)  3952446K->20318K(15073280K), 0.0005037 secs]
[GC (Allocation Failure)  3952478K->20350K(15073280K), 0.0006393 secs]
finished trial...
```

- Garbage collection notifications like this can (and should) be printed by running `java` **`-XX:+PrintGC`** `MyProgram`
- Save it to a file instead with **`-Xloggc:my.log`**

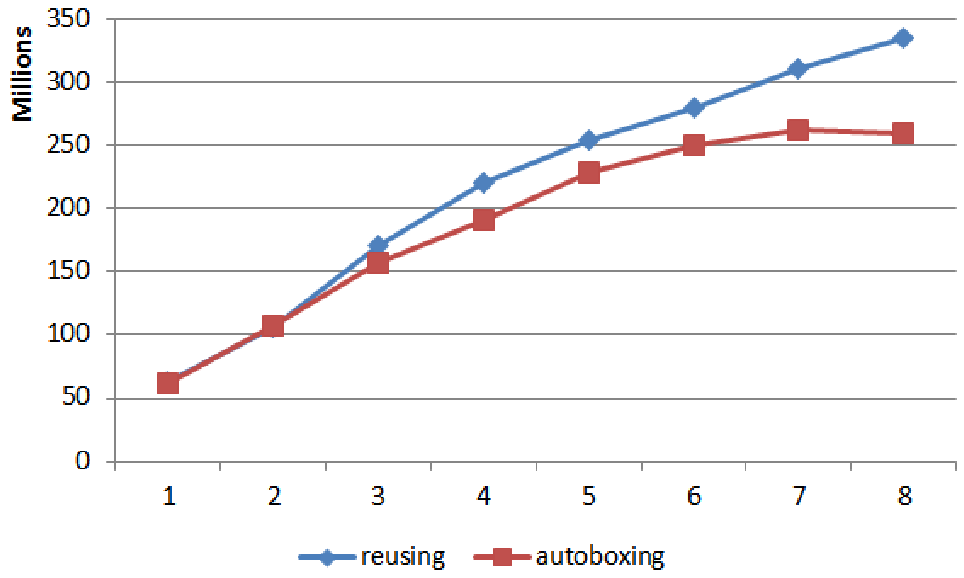
A space efficient alternative

- Instead of creating a new Integer object each time an integer in the range [0, 99) is passed to set, we can reuse Integer objects

```
Integer[] reuse = new Integer[100];
for (int i = 0; i < 100; ++i) reuse[i] = i;
SingleCell<Integer> cell = new SingleCell<>();
for (int i = 0; ; ++i) cell.set(reuse[i % 100]);
```

The impact on performance

- The following graph shows how reusing Integer objects improves performance
- The JVM heap is 256MB
  - A smaller heap makes auto boxing slower

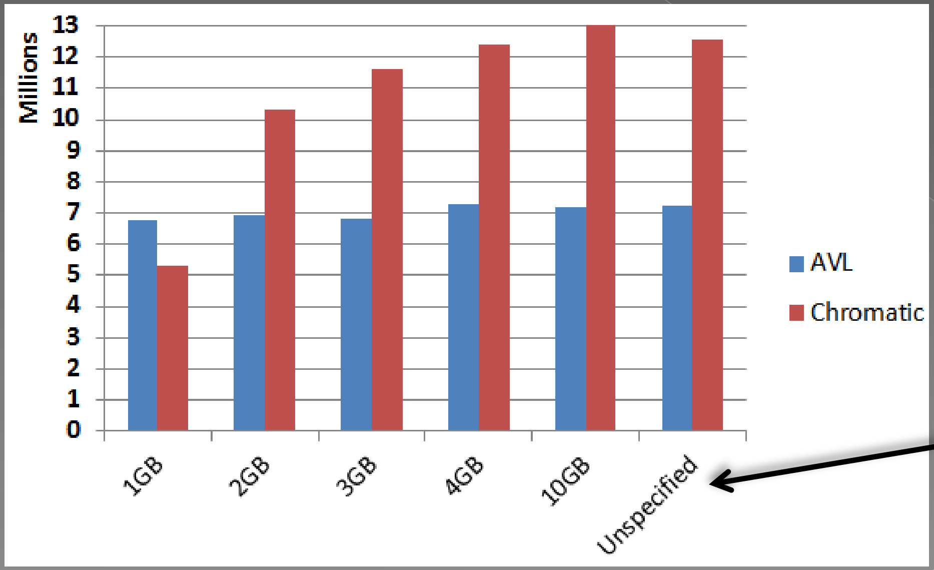


The JVM heap and resizing

- The JVM accepts two arguments, –Xms and –Xmx, which specify minimum and maximum heap sizes, respectively
- If these parameters are not specified, the JVM can resize the heap
  - In practice, JVMs frequently resize the heap
  - Since this may occur in some trials, and not in others, it is best to control this variable

Results depend on heap size

- When comparing algorithms that use lots of memory, heap size matters
- It is important to think about whether comparisons should include or exclude memory reclamation cost



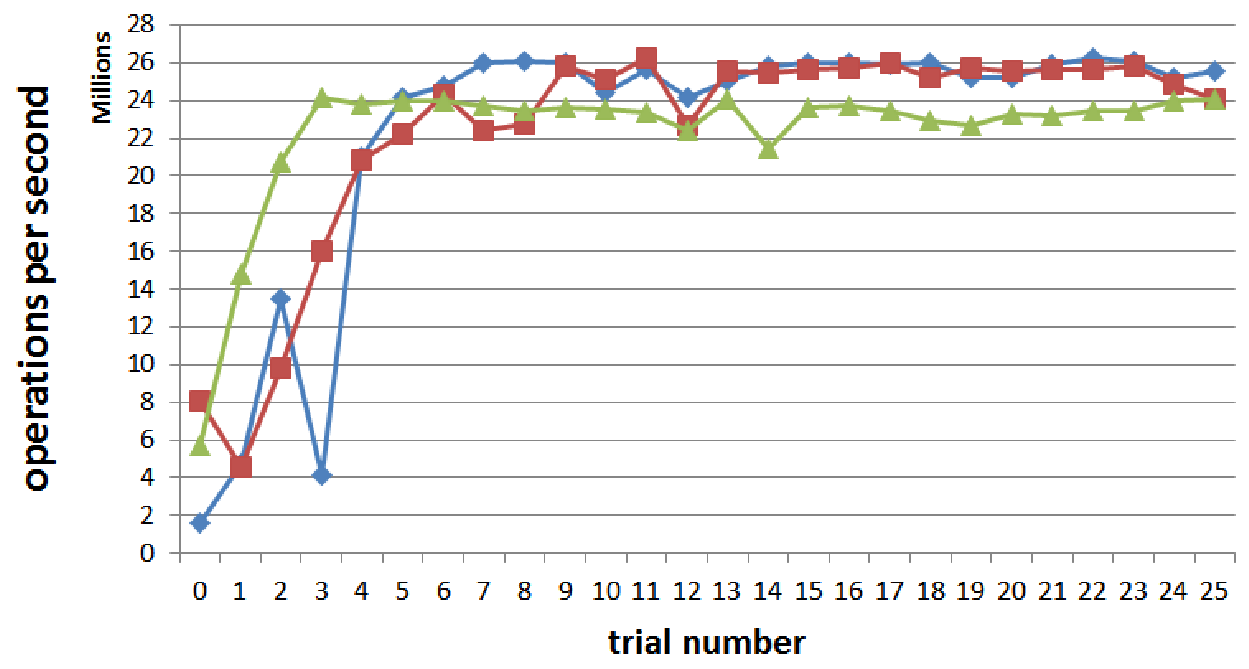
Minimum and maximum heap size

Hot-spot compilation

- Java compilation occurs throughout an execution (but mostly in the first few seconds)
- This is important when comparing algorithms
  - Some algorithms take longer to compile, and stay in a slow, interpreted state for longer
  - This reduces their measured performance compared to faster compiling algorithms
- One solution is to discard the first few trials of each experiment

Hot-spot compilation - 2

- For example, the following graph shows how the throughput for three data structures changes as they are compiled

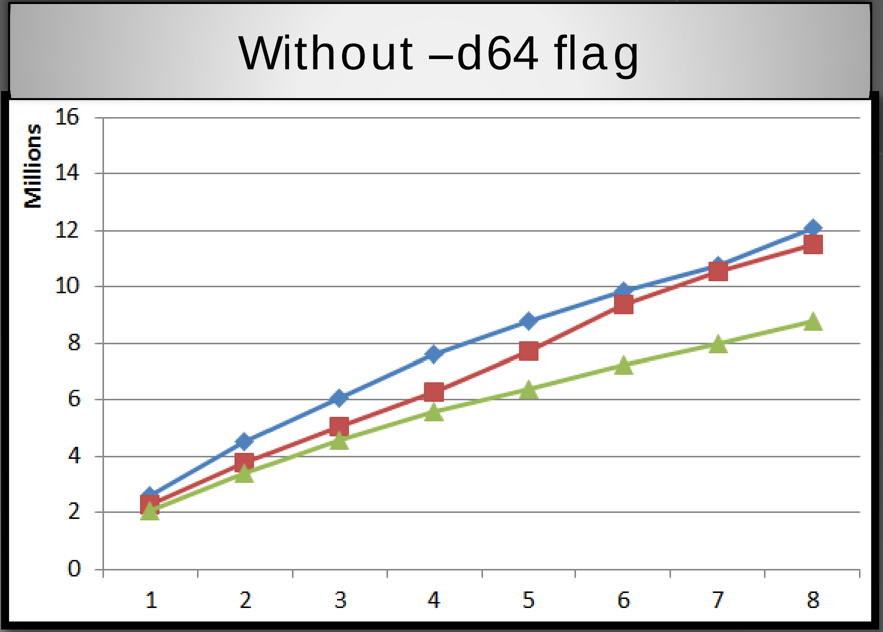


- Should discard trials 0-4 (maybe even 0-14)

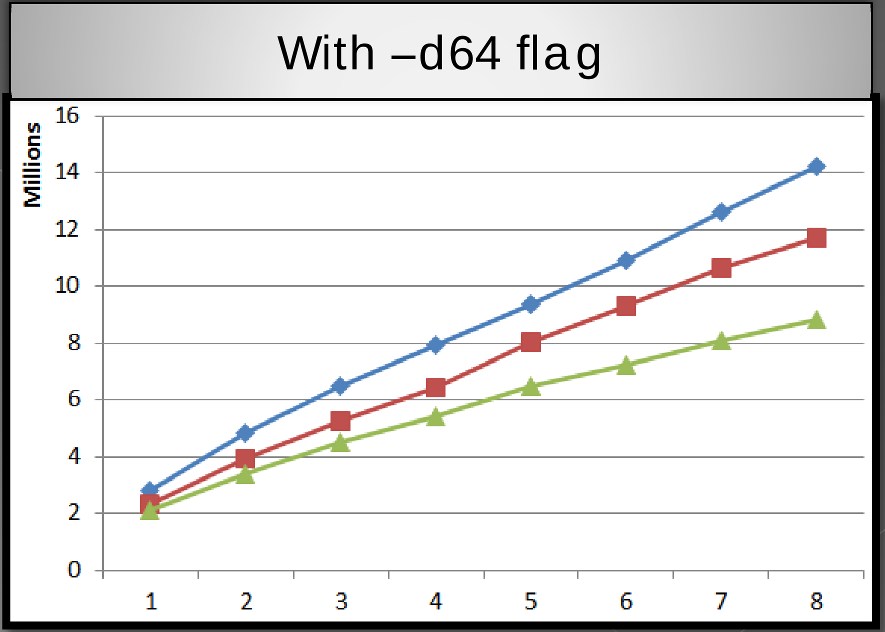
Runtime flags for the JVM

- Use a 64-bit JVM on a 64-bit machine
- Use the -d64 and -server JVM flags
  - `java -d64 -server MyProgram`
  - The former enables 64-bit execution
  - The latter enables aggressive optimizations
- These flags can change performance measurements significantly

Impact of the –d64 flag



Without –d64 flag



With –d64 flag

Run each data point in a separate JVM

- Multiple experiments run in the same JVM are **not** statistically independent
  - See “Statistically Rigorous Java Performance Evaluation” by Georges et al.
- It is not enough to simply run garbage collection between each pair of trials
- The internal state of the memory allocator, garbage collector and Hotspot compiler are largely inaccessible

Recording data in a trial
(e.g., number of operations completed)

- Collect data on a per-thread basis to avoid synchronization
  - Create a private ThreadData object for each thread, containing private counters
  - Aggregate (Sum/Average/Min/Max) the data in these objects after a trial has ended

Output for each trial

- Output all per-thread data, and any useful debugging information (as long as this does not affect performance)
- The extra output helps with debugging
- Use Bash scripts to prune unwanted info

Example output file for a trial:

perf-i10-d10-k1000000-n2-t3000-trial4.csv

```
PREFILL op# 1000000 sz=316497 expectedSize=500000
PREFILL op# 2000000 sz=432205 expectedSize=500000
PREFILL op# 3000000 sz=474889 expectedSize=500000
finished prefilling to size 485001 for expected size 500000
main thread: starting timer...
main thread: attempting to join thread 0
tid= 0: op# 1000000
tid= 1: op# 1000000
tid= 1: op# 2000000
tid= 0: op# 2000000
   ...
main thread: joined thread 0
main thread: attempting to join thread 1
main thread: joined thread 1
total insert succ              : 1095616
total insert retry             : 1
total erase succ               : 1095312
total erase retry              : 2
total find succ                : 8761970
total find retry               : 0
total succ insert+erase+find   : 10952898
throughput (succ ops/sec)      : 3650966
elapsed milliseconds           : 3000
```

Using Bash to get results from trial file: perf-i10-d10-k1000000-n2-t3000-trial4.csv

```
   ...
total succ insert+erase+find   : 10952898
throughput (succ ops/sec)      : 3650966
elapsed milliseconds           : 3000
```

- Suppose $file contains the trial’s filename
- For example, we can extract the throughput, using **grep**, **cut** and **tr:**

  -
    ```
    x=` grep "throughput" $file | cut –d":" –f2 | tr –d " " `
    ```
- We can also extract, e.g., the number of threads from the filename:

  -
    ```
    nthreads=` echo $file | cut -d"-" -f5 | tr -d "n" `
    ```

Source of randomness

- Avoid java.util.Random, which uses locks
- Alternative Random implementation:

```
public class Random
    private int seed;
    public Random (int seed) { this.seed = seed; }
    public int nextInt() {
        seed ^= seed << 6;
        seed ^= seed >>> 21;
        seed ^= seed << 7;
        return seed;
    }
}
```

- Create an instance of Random for *each thread* (with *different* seed values from, e.g., https://www.random.org/)

Intel’s Multicore Testing Lab

Logging in to MTL

- `ssh indigo`
- `ssh yufb-s##@207.108.8.131`
- (You must go through indigo, because all other IPs are rejected by MTL)

Copying files to MTL

- Copy MyFolder to your MTL home directory
  - `scp -r MyFolder yufb-s##@207.108.8.131:`
- Copy MyFile to your MTL home directory
  - `scp MyFile yufb-s##@@207.108.8.131:`

Path to `java` and `javac`

- `javac -version`
  - `Eclipse Java Compiler v_677_R32x, 3.2.1 release, Copyright IBM Corp 2000, 2006. All rights reserved.`
- This is quite old, so make sure you use the versions of `javac` and `java` located in `/opt/java/latest/bin/`
- `/opt/java/latest/bin/javac -version`
  - `javac 1.7.0_01`

Do not spawn too many threads

- The number of threads each user can spawn is limited
- If you spawn too many, Java will experience an internal error, and, in my experience, will refuse to terminate
- Since you have exhausted your supply of threads, you will be unable to log in again or execute **kill** to stop your runaway JVM
- After 24 hours your JVM will be auto-killed

Premeditated killing

- If you do not want to experience this, you can first “reserve” a victim process
  - I run an extra SCP connection to MTL
  - If I need to free up a process, I terminate my SCP connection, which freeing up a process I can then use to run **kill**
  - The following command does the trick:

```
for i in {1..9999}; do kill -9 $i; done
```

Miscellanea

- Control the set of processors that your application will use with `taskset`, e.g.,
  - `taskset 1-16 MyBenchmarkScript`
  - MyBenchmarkScript will use only CPUs 1-16
- Text editor on MTL: `nano`
- SCP for Windows: [WinSCP]
- SSH for Windows: [PuTTY]
- Check who else is running on MTL: `top`