An Introduction to
Java Microbenchmark Harness (JMH)

IN3030

Michael Kirkedal Thomsen

michakt@ifi.uio.no

Spring, Mar 8 2023

Me!

- Associate Professor, arrived in June 2021
- Danish
- Previously at University of Copenhagen and University of Bremen
- Background in programming languages, domain specific languages, logic design
- Works with
  - security and reliability in decentralised systems and models
  - security and communication in the maritime sector
  - energy of computations

What is Java Microbenchmark Harness (JMH)?

- A framework for building, running and analysing benchmarks written in Java
- Microbenchmarks are performance metrics on the lowest level.

You can compare them to unit tests, which means they invoke single methods or execute small pieces of business logic without “more (cross-cutting) stuff” around.

JHM have many similarities to JUnit

Creating your benchmark

Easy with Maven...

```
mvn archetype:generate \
-DinteractiveMode=false \
-DarchetypeGroupId=org.openjdk.jmh \
-DarchetypeArtifactId=jmh-java-benchmark-archetype \
-DgroupId=com.benchmark \
-DartifactId=myBenchmark \
-Dversion=1.0
```

To build run `mvn clean install` -- You can omit the `clean`

To run `java -jar target/benchmarks.jar`

What!!!!!

```
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: com.benchmark.MyBenchmark.testMethod

# Run progress: 0.00% complete, ETA 00:08:20
# Fork: 1 of 5
# Warmup Iteration   1: 2508409993.172 ops/s
# Warmup Iteration   2: 2446156719.550 ops/s
...
```

Runs 5 tests, where each test runs 5 times with 5 warmups. Waiting 10 seconds in between...

Benchmark Setup

Add `@Benchmark` before your method to make it a benchmark. The alternative to JUnit `@Test`.

- `@Fork(1)` - number of runs
- `@Warmup(iterations = 2)` - warmup iterations
- `@Measurement(iterations = 3)` - number of measurements in each run

Why warm-up runs?

Why warm-up runs?

First runs can by erroneous.

Make sure

- the program is in memory / cache,
- that JIT compiler have done its optimization,
- that data have been read from disk - Virtual Memory can swap it,
- that data have been though memory / cache.

Blackholes

Use `Blackhole` class of JMH can avoid deal code elimination by JVM.

- If you pass the calculated results to `blackhole.consume()`, it would trick the JVM. JVM will never drop the code, thinking that `consume()` method uses the result.
- Alternatively, your benchmarks can return the value

What!!!! -- a Result

```
Result "com.benchmark.MyBenchmark.testMethod":
  2449662660.070 ±(99.9%) 59785993.182 ops/s [Average]
  (min, avg, max) = (2108656264.402, 2449662660.070, 2498643339.455), stdev = 79812590.406
  CI (99.9%): [2389876666.888, 2509448653.251] (assumes normal distribution)


# Run complete. Total time: 00:08:21

....


Benchmark                  Mode  Cnt           Score          Error  Units
MyBenchmark.testMethod    thrpt   25  2449662660.070 ± 59785993.182  ops/s
```

- 2.4 GHz Quad-Core (hyper threaded) Intel Core i5.
- 8 GB RAM

What does this all mean?

Benchmarks are only interesting when we consider and discuss the result.

Benchmark Modes

- `Throughput("thrpt", "Throughput, ops/time")`
  - It will calculate the number of times your method can be executed with in a second
- `AverageTime("avgt", "Average time, time/op")`
  - It will calculate the average time in seconds to execute the test method
- `SampleTime("sample", "Sampling time")`
  - It randomly samples the time spent in the test method calls
- `SingleShotTime("ss", "Single shot invocation time")`
  - It works on single invocation of the method and is useful in calculating cold performance
- `All("all", "All benchmark modes")`
  - Calculates all the above

What about mean time?

- Warm-up runs help removing artificial outliers
- Mean time can be a selective method.
  - Less scientific as we should not remove bad data...
- These are deterministic programs. How about non-determinism?
- We are still reporting the standard derivation.

NB!

With simple (home-made) testing mean time can be more correct. With the right setup of a harness average time is better.

Time Measurement

Use `@OutputTimeUnit` annotation to set the preferred time from `java.util.concurrent.TimeUnit`

```
@OutputTimeUnit(TimeUnit.SECONDS)
```

The TimeUnit enum has the following values:

```
NANOSECONDS
MICROSECONDS
MILLISECONDS
SECONDS
MINUTES
HOURS
DAYS
```

Let's test Fib

```
    @Benchmark
    @Fork(1)
    @Warmup(iterations = 1)
    @Measurement(iterations = 3)
    public int testfib() {
        return fib(25);
    }


    public int fib(int n) {
        if (n < 2) {
            return 0;
        }
        else {
            return fib(n-1) + fib(n-2);
        }
    }
```

How do we parallelise Fib?

- Methods?
    - Spawn for each of the two recursions

What is the result?

```
# Benchmark: com.benchmark.MyBenchmark.testMethodFib

Iteration   1: 2.482 ops/s
Iteration   2: 2.491 ops/s
Iteration   3: 2.167 ops/s

Result "com.benchmark.MyBenchmark.testMethodFib":
  2.380 ±(99.9%) 3.364 ops/s [Average]
  (min, avg, max) = (2.167, 2.380, 2.491), stdev = 0.184
  CI (99.9%): [≈ 0, 5.744] (assumes normal distribution)


# Benchmark: com.benchmark.MyBenchmark.testMethodPar

Iteration   1: 5.297 ops/s
Iteration   2: 5.708 ops/s
Iteration   3: 5.706 ops/s


Result "com.benchmark.MyBenchmark.testMethodPar":
  5.570 ±(99.9%) 4.314 ops/s [Average]
  (min, avg, max) = (5.297, 5.570, 5.708), stdev = 0.236
  CI (99.9%): [1.256, 9.885] (assumes normal distribution)


# Run complete. Total time: 00:01:21

Benchmark                     Mode  Cnt  Score   Error  Units
MyBenchmark.testMethodFib  thrpt    3  2.380 ± 3.364  ops/s
MyBenchmark.testMethodPar  thrpt    3  5.570 ± 4.314  ops/s
```

Running Matrix Mult

```
Benchmark                 Mode  Cnt  Score   Error  Units
MyBenchmark.test_ijk  avgt    2  2.053          s/op
MyBenchmark.test_ikj  avgt    2  0.228          s/op
MyBenchmark.test_kji  avgt    2  9.866          s/op
```

perf

perf: it can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing). It is capable of lightweight profiling. It is also included in the Linux kernel, under tools/perf, and is frequently updated and enhanced.

To run with cache `java -jar target/benchmarks.jar -prof perf`

Resources

[1] http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples

[2] https://dzone.com/articles/java-microbenchmark-harness-jmh

[3] https://perf.wiki.kernel.org/index.php/Main_Page