Benchmarking Parallel Programs

Parallel Programming in Scala

Aleksandar Prokopec

Testing and Benchmarking

- testing – ensures that parts of the program are behaving according to the intended behavior

Testing and Benchmarking

- testing – ensures that parts of the program are behaving according to the intended behavior
- benchmarking – computes performance metrics for parts of the program

Testing and Benchmarking

- testing – ensures that parts of the program are behaving according to the intended behavior
- benchmarking – computes performance metrics for parts of the program

Typically, *testing* yields a binary output – a program or its part is either correct or it is not.

Benchmarking usually yields a continuous value, which denotes the extent to which the program is correct.

Benchmarking Parallel Programs

Why do we benchmark parallel programs?

Benchmarking Parallel Programs

Why do we benchmark parallel programs?

Performance benefits are the main reason why we are writing parallel programs in the first place.

Benchmarking parallel programs is even more important than benchmarking sequential programs.

Performance Factors

Performance (specifically, running time) is subject to many factors:

Performance Factors

Performance (specifically, running time) is subject to many factors:

- processor speed

Performance Factors

Performance (specifically, running time) is subject to many factors:

- processor speed
- number of processors

Performance Factors

Performance (specifically, running time) is subject to many factors:

- processor speed
- number of processors
- memory access latency and throughput (affects contention)

Performance Factors

Performance (specifically, running time) is subject to many factors:

- processor speed
- number of processors
- memory access latency and throughput (affects contention)
- cache behavior (e.g. false sharing, associativity effects)

Performance Factors

Performance (specifically, running time) is subject to many factors:

- processor speed
- number of processors
- memory access latency and throughput (affects contention)
- cache behavior (e.g. false sharing, associativity effects)
- runtime behavior (e.g. garbage collection, JIT compilation, thread scheduling)

Performance Factors

Performance (specifically, running time) is subject to many factors:

- processor speed
- number of processors
- memory access latency and throughput (affects contention)
- cache behavior (e.g. false sharing, associativity effects)
- runtime behavior (e.g. garbage collection, JIT compilation, thread scheduling)

To learn more, see What Every Programmer Should Know About Memory, by Ulrich Drepper.

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

- multiple repetitions

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

- multiple repetitions
- statistical treatment – computing mean and variance

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

- multiple repetitions
- statistical treatment – computing mean and variance
- eliminating outliers

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

- multiple repetitions
- statistical treatment – computing mean and variance
- eliminating outliers
- ensuring steady state (warm-up)

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

- multiple repetitions
- statistical treatment – computing mean and variance
- eliminating outliers
- ensuring steady state (warm-up)
- preventing anomalies (GC, JIT compilation, aggressive optimizations)

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

- multiple repetitions
- statistical treatment – computing mean and variance
- eliminating outliers
- ensuring steady state (warm-up)
- preventing anomalies (GC, JIT compilation, aggressive optimizations)

To learn more, see Statistically Rigorous Java Performance Evaluation, by Georges, Buytaert, and Eeckhout.

ScalaMeter

ScalaMeter is a benchmarking and performance regression testing framework for the JVM.

- performance regression testing – comparing performance of the current program run against known previous runs
- benchmarking – measuring performance of the current (part of the) program

ScalaMeter

ScalaMeter is a benchmarking and performance regression testing framework for the JVM.

- performance regression testing – comparing performance of the current program run against known previous runs
- benchmarking – measuring performance of the current (part of the) program

We will focus on benchmarking.

Using ScalaMeter

First, add ScalaMeter as a dependency.

```
libraryDependencies +=
  "com.storm-enroute" %% "scalameter-core" % "0.6"
```

Using ScalaMeter

First, add ScalaMeter as a dependency.

```
libraryDependencies +=
  ”com.storm-enroute” %% ”scalameter-core” % ”0.6”
```

Then, import the contents of the ScalaMeter package, and measure:

```
import org.scalameter._

val time = measure {
  (0 until 1000000).toArray
}

println(s”Array initialization time: $time ms”)
```

Demo

Measuring the running time.

JVM Warmup

The demo showed two very different running times on two consecutive runs of the program.

When a JVM program starts, it undergoes a period of *warmup*, after which it achieves its maximum performance.

- first, the program is *interpreted*

JVM Warmup

The demo showed two very different running times on two consecutive runs of the program.

When a JVM program starts, it undergoes a period of *warmup*, after which it achieves its maximum performance.

- first, the program is *interpreted*
- then, parts of the program are compiled into machine code

JVM Warmup

The demo showed two very different running times on two consecutive runs of the program.

When a JVM program starts, it undergoes a period of *warmup*, after which it achieves its maximum performance.

- first, the program is *interpreted*
- then, parts of the program are compiled into machine code
- later, the JVM may choose to apply additional dynamic optimizations

JVM Warmup

The demo showed two very different running times on two consecutive runs of the program.

When a JVM program starts, it undergoes a period of *warmup*, after which it achieves its maximum performance.

- first, the program is *interpreted*
- then, parts of the program are compiled into machine code
- later, the JVM may choose to apply additional dynamic optimizations
- eventually, the program reaches *steady state*

ScalaMeter Warmers

Usually, we want to measure steady state program performance.

ScalaMeter `Warmer` objects run the benchmarked code until detecting steady state.

```
import org.scalameter._

val time = withWarmer(new Warmer.Default) measure {
  (0 until 1000000).toArray
}
```

Demo

Measuring the stable running time.

ScalaMeter Configuration

ScalaMeter configuration clause allows specifying various parameters, such as the minimum and maximum number of warmup runs.

```
val time = config(
  Key.exec.minWarmupRuns -> 20,
  Key.exec.maxWarmupRuns -> 60,
  Key.verbose -> true
) withWarmer(new Warmer.Default) measure {
  (0 until 1000000).toArray
}
```

Demo

Measuring the stable running time with verbose output.

ScalaMeter Measurers

Finally, ScalaMeter can measure more than just the running time.

- `Measurer.Default` – plain running time
- `IgnoringGC` – running time without GC pauses
- `OutlierElimination` – removes statistical outliers
- `MemoryFootprint` – memory footprint of an object
- `GarbageCollectionCycles` – total number of GC pauses
- newer ScalaMeter versions can also measure method invocation counts and boxing counts

Demo

Measuring the memory footprint.