High Performance Computing with Java

Windsor-Essex GDG DevFest 2024

- Presented by: **Abdul Habra**, @ahabra

- 2024.11.16

Introduction

- You should be experienced with Java
- Build your system to be valid and well-designed
- *Premature optimization is the root of all evil* -- Donald Knuth
- Understand the bottlenecks of your program before attempting to optimize
- Will point at different approaches to increase the performance of Java programs.
- Will NOT discuss JVM fine-tuning, or scaling.

Collections

When dealing with a List (or other collection types) of a primitive type, Java generics will use the wrapper type, for example:

```
List<Integer> ok = List.of(1, 2, 3);
List<int> notOk = List.of(1, 2, 3);  // compiler error
```

You can use the `eclipse-collections-api` library which provide primitives collections types like `IntArrayList`.

```
IntArrayList list = new IntArrayList();
list.add(1);
```

`IntArrayList` is about **4 times faster** than the JDK `List<Integer>`, and it consumes less memory.

- Supported primitives: boolean, char, byte, short, int, long, float, double
- Supported collections: list, map, set, stack.

Fused Multiply Add (FMA)

In applications which heavily use floating point calculations like in machine learning, scientific measurements, or financial applications, a common calculation happens:

1. Multiply two numbers
2. Add the product to an accumulator

Or simply calculate `a * b + c` when a, b, and c are `float` or `double`.

Two roundings can occur:

- when multiplying a and b.
- when adding the product to c.

To enhance the calculation, you can use `Math.fma(a, b, c)` which will do only one rounding.

The `fma()` method is about 15% faster than the hand-coded calculation.

JSON Parsing

There are many JSON parsing libraries, one of the fast/popular ones is `com.fasterxml.jackson.core:jackson-databind`. It is tested here.

Two approaches are tested:

1. Using `ObjectMapper`. Simple to implement. About 25 lines of code.
2. Using `JsonParser`. Harder to implement. About 150 lines of code. About 10 times faster than using `ObjectMapper`.

HttpClient

Creating an instance of `HttpClient` is expensive, reuse the instance if you can. In an example where an HTTP GET (to same URL) is called 1000 times, reusing the instance was 3 to 4 times faster and consumed about 1/3 of the memory.

Instances of `HttpClient` are not quickly garbage collected, so creating a lot of instances could crash the JVM.

RegEx

When the string to find/replace is simple, consider using `org.apache.commons.lang3.StringUtils.replace()` instead of `String.replaceAll()`, because replaceAll() takes a RegEx.

The `StringUtils.replace()` is about 50% faster than `String.replaceAll()`.

Random Numbers

Using several approaches to generate random numbers, next is a list ordered by performance, fatser is first:

1. `java.util.concurrent.ThreadLocalRandom` : this was the fastest in generating random numbers
2. `java.util.Random` : about 5 times slower than `ThreadLocalRandom`
3. `Math.random()` : about 20% slower than `java.util.Random`
4. `SecureRandom.getInstanceStrong()` : About 8 times slower than `Math.random()`
5. `new SecureRandom()` : About 30% slower than `SecureRandom.getInstanceStrong()`

In general, the fastest approach is about 75 times faster than the slowest.

Recommendation:

1. If you do not need a secure random, use `ThreadLocalRandom` .
2. If you do need a secure random, use `SecureRandom.getInstanceStrong()`
3. On some OS platforms, SecureRandom has the potential of blocking for long time. Be careful.

Hashing

To produce secure *SHA256* hashes, there are several options:

1. JDK standard MessageDigest.
2. Apache commons codec.
3. Guava.
4. SHA3_256 JDK.
5. SHA3_256 Apache commons Codec.
6. SHA3_256 Keccak. About 2 to 3 times slower than others.

The first 5 approaches perform relatively at the same scale.

If you do not need a secure hash, then there are faster and simple algorithms. Consider using `org.apache.commons.codec.digest.XXHash32` .

Reflection

Three approaches are tested:

1. Using `org.apache.commons.lang3.reflect` FieldUtils and MethodUtils. This is the slowest approach.
2. Using classic Java reflection `Class` getField and getMethod. About 4 to 5 times as fast as using apache.
3. Using the `java.lang.invoke` VarHandle and MethodHandle (added in Java 1.7). Similar to classic.

Observe that when using classic reflection or invoke package, the major performance gain is achieved because we can get a reference to the method or the field, cash it , then reuse it during the repeated invocations.

References

1. This presentation: * https://github.com/ahabra/java-perf
2. **TODO**