Concurrent Programming With Groovy

Who Am I: **David Clark**

Where Do I Work: **Principal Engineer, Research Now**

Where Do I Code: https://github.com/dwclark

Where Is This Presentation: https://github.com/dwclark/concurrent-groovy

Why Is This Presentation So Boring And On A White Background?:

Because this is a really hard topic to present on. I went with the "wall of text" anti-pattern so that you can review this info later. I'm hoping that for the most part we have a discussion, please ask questions and share experiences whenever. Concurrency is more art than science so it's hard to give a clear cut series of practices to follow.

Let’s Start With a Rant

- Concurrent/Parallel programming is really hard.
- I’ve done concurrent programming in C++, I’ve heard it’s better now with C++14, but I still wouldn’t go back unless paid big bucks.
- Most dynamic languages have a bad threading story because of lack of VM support, lack of memory model, presence of interpreter locks, etc.
- Perl 6 was supposed to provide an interesting answer for this, but everyone forgot about Perl 6...including the Perl people.
- Scala sort of does immutability...except when it doesn’t.
- Clojure has an interesting story for certain classes of concurrency problems, but I have my doubts that it can solve the hardest types of concurrency problems.

Sure, But Can’t You Rant About Groovy?

- Yes, I can: Groovy has been an interesting mix of behind the curve, about average, and ahead of the curve at different points in it’s history.
- Groovy came with anonymous functions (closures) from day one (ahead).
- GPars gave us a way to do Fork/Join and parallel collection processing years before anyone in the plain Java world could (ahead).
- But, Groovy closures have performance issues which means using them in GPars or with Java 8 streams doesn’t work for performance sensitive applications (behind).
- GPars has not been updated to use newer Java threading features so we are stuck using JSR 166y parallel arrays and thread pools (behind).
- But since it is Groovy we can always use Java directly, use @CompileStatic, and write ugly-ish code if necessary (about average).

And Now For the Bad News (End Rant)

- It would be nice to just ignore concurrency but we can’t because...
- Computer clock speeds are not getting any faster.
- Exotic CPU optimizations are basically maxed out (pipelining, branch prediction, out of order instruction execution).
- The only real improvement that makes all programs faster is larger L2 caches.
- Some programs are also sped up by SSD drives (but only I/O bound ones) .
- That’s all folks!
- The only thing we are left with is the fact that computers sprout cores like mushrooms. If you want to do more or go faster you have to use them.
- The worst news: Using all your cores ranges from “about as hard as regular programming” to “here be dragons!”

The Good News

- It’s possible to do this and you can be the hero on your team.
- There are some simple practices that reduce most/all concurrency issues
    1) Use a friendly language and platform. Because you are using Groovy on the JVM you have this covered.
    2) The more immutable your code, the better.
    3) Use synchronization primitives as little as possible, preferably never
    4) Use thread safe libraries and understand what your responsibilities are when using code that is not explicitly thread safe (which are most libraries).
    5) Understand the execution model of the libraries you are using.
    6) Match the execution model to the problem you are solving. For example I/O heavy code doesn’t play well with Java Streams. Fine grained computationally intensive code doesn’t play well with GPars actors.

Immutability

- I can’t emphasize this enough, the less you mutate, the easier your life will be.
- Every use of **final** is at least one less bug you will have
- If all of your fields are final you basically don’t have any problems with concurrency (yes, for the experts out there this is an oversimplification).
- But if your all of your fields are final, you are basically programming in clojure with Groovy syntax.
- Controlled mutation is what allows for full flexibility and performance in a multi-threaded process.

`import groovy.transform.*`

- `@Immutable`: Make a class thread safe by making mutations impossible. Use this as often as possible in concurrent code.
- `@Synchronized`: A safer alternative than using `synchronized` keyword at the method level. Use sparingly, you should avoid locks as much as possible.
- `@WithReadLock`: Convenience annotation for wrapping methods with read locks. Use even more sparingly, read/write locks are surprisingly hard to get right.
- `@WithWriteLock`: Same as `@WithReadLock`, same warnings apply

Decisions To Make When Doing Concurrency

1. Do I have dedicated tasks that run for the life of my application or are they ephemeral, in other words does it make sense to name each task or not?
2. What kind of pool do I need?
3. How computationally intense are my parallel tasks?
4. How much shared state do I have and how complex/dependent are the tasks generating these states?

For each combination of the above questions, there tends to be a relatively good/safe choice at least as a first attempt at solving the problem.

Decision #1, Dedicated or Ephemeral Tasks?

- I have dedicated tasks
  - I have tasks that run continuously
    - Use `Thread.start(task)` at the start of your application
  - I have tasks that run intermittently
    - Create a `ScheduledExecutorService`
    - Schedule your tasks using `scheduleAtFixedRate` or `scheduleWithFixedDelay`
  - Skip to decision #3
- I don’t have dedicated tasks, go to decision #2

Decision #2, What Kind of Pool?

- My computation involves I/O (files, sockets, db connections, etc.)
  - Create a growable/shrinkable `ThreadPoolExecutor` with a bounded queue
  - Size your pool with a max size of 5x the number of cores you have
- My computation does not involve I/O
  - My computation does not involve heavy use of floating point operations
    - Use the built in `ForkJoinPool.commonPool()` to schedule your tasks
  - My computation involves heavy use of floating point operations
    - You need GPU support. Use something like deeplearning4j which will expose parallelism via `ParallelWrapper` and `CudaEnvironment`.
- Move to decision #3

Decision #3, What Kind of Code?

- My computation involves I/O (files, sockets, db connections, etc.)
  - Use plain vanilla Groovy or `@TypeChecked` Groovy
  - Move to Decision #4
- My computation is CPU intense and uses integral types (`short`, `int`, `long`, etc.)
  - Use `@CompileStatic` Groovy or Java
  - Move to Decision #4
- My computation involves floating point operations (machine learning, neural nets)
  - Use Fortran libraries (Blas, Linpack, Lapack)
  - You're done, good luck, hire a lot of Ph.D's, be prepared for lots of results that are useless, hard to interpret, baffling...and rarely...a real breakthrough.

Decision #4, Linearly Dependent State

- Your code is executing a series of operations where the next state depends on the previous state, the number of operations is irrelevant here.
- Each operation takes a long time and you are not executing operations on millions of items stored in collections.
- Option #1: Choose Java 8 `CompletableFuture`
- Option #2: Choose GPars `Promise/Dataflow` operations
- Examples: *DownloadWebsites.groovy* and *BasicPipeline.groovy*

Decision #4, Parallel Collections

- Your code is processing a large number of items in collections. Each operation is independent of each other except for possibly a final reduction phase.
- Choose: Choose Java 8 streams (or GPars Parallel Arrays if you are stuck on 7).
- Gotcha #1: It’s harder than you think to process collections in parallel; measure your actual performance and avoid parallel collections if they are slower.
- Gotcha #2: Groovy casting and boxing/unboxing can dominate your execution. If you are doing either of these, you are probably doing it wrong.
- Gotcha #3: Memory allocation/GC can also dominate your processing. Immutable reductions are a common culprit here. Take a look at mutable reductions using `Colllector`/`Collectors` in `java.util.stream`.
- Example: *ParallelCollections.groovy*

Decision #4, Independent States

- Your code has a large number of independent states that must be maintained.
- Choose: GPars Actors
- Actors isolate all state decisions. Only one thread at a time can ever touch the internal state of an actor.
- The only way to change the state of an actor is to send a message to the actor. In GPars actors maintain queues which are used to store/process messages.
- A very large number of actors can be serviced by a small number of threads. For performance related actors should be attached to the same thread pool.
- Example: *FilingCabinet.groovy*

Decision #4, Independent Operations

- Your code has a large number of operations that must all operate on the same shared state.
- Choose: GPars Agents
- Agents are actors in reverse. Actors define operations and accept messages. Agents define shared state and accept operations.
- Only one operation at a time will operate on the shared state. Agents really shine when standard OO encapsulation is unnatural and your code is better modelled as a series of functions.
- Example: *ShoppingCart.groovy*

Decision #4, Complex State Dependencies

- Your code has a large number of states that have a complex set of dependencies.
- Choose: GPars Dataflow
- Dataflow allows you to define complex networks of state dependencies.
- A small number of concepts can be used to model these dependencies: Tasks, DataflowVariables, DataflowQueues, DataflowChannels, Selectors, Operators.
- Example: *BasicComputation.groovy*

What’s the *Future<T>* of Concurrent Programming With Groovy?

- Groovy will continue to take advantage of JVM and Java library improvements.
- Groovy friendly platforms such as Grails and Ratpack.
- Need #1: Groovy really needs something like Java lambdas to take advantage of the processing power of Java 8 streams (there are workarounds that work today).
- Need #2: GPars is showing its age.
  - GPars needs to use Java 8 features
  - GPars needs to get away from forcing usage of dynamic Groovy. Modern Groovy is more and more supporting both static and dynamic Groovy.
  - But, this is all open source and free, so this is not a criticism of the GPars team.
- Anything we can imagine that doesn’t break too many physical laws, Groovy is a great platform, let’s push the envelope and make great concurrent stuff.