Motivations for Multi Threaded/Concurrent Programs

1) Does anyone have a single processor system anymore?

2) The free lunch is over. Processors have traditionally gotten faster because of higher clock speeds, exotic optimizations (branch prediction, instruction reordering, pipelining, etc.), and bigger on die caches. Of those three, only bigger on die caches will likely continue to improve.

3) But, Moore's law is still active and transistor counts continue to improve...and the engineers need to stick them somewhere. This is what leads to higher and higher core counts.

4) A lot of this processing power can be utilized by email servers, web servers, and database servers. We can run our code inside those environments and take advantage of concurrency there. However, these only provide a coarse grained level of concurrency and force a particular execution model onto an application. Code must execute in response to an email, a web request, or a query. If an application needs finer grained concurrency or a different execution model, you need to roll your own.

Multi-Threading on the JVM

1) Java has had a threading story since Java 1.0 and was really ahead of its time at that point.

2) However, the original designers did make a few mistakes at the time such as making every object a lock, making Runnable return void, broken memory model, synchronizing too many classes by default, etc. Still, it was better than anything else out there.

3) The original threading libraries provided basic tools, but not much in the way of guidance.

4) Things gradually began to improve over time and eventually JDK 5 fixed most of the original problems with Java threading. JDK 5 added better locks, atomics, Callable, futures, good memory model, and not synchronizing by default.

5) Groovy makes it even easier to use JDK 5 features. GPars builds on JDK 5 to provide a rich and easy to use concurrency framework.

General Notes

1) Immutability is a huge win for multi-threaded and concurrent programming. Anything that can be made immutable should be made immutable.

2) Even things that may not appear immutable, may be amenable to being made immutable. In my initial version of the Integrate class, I used a mutable variable that caused a nasty race condition in the code. Thinking a little harder, it was possible to replace the mutable variable with a computation done in each loop. Had I made everything immutable, I would have never seen this problem and I would have saved some serious debugging time.

3) Multi-threaded programming is very susceptible to Heisenbugs, bugs that disappear once you add instrumentation or attach a debugger. Be prepared to do more thinking and less exploring with multi-threaded programming.

4) Read Java Concurrency in Practice by Brian Goetz. Then read it again. Knowing the Java Memory Model dispels a lot of the mystery in best practices for concurrent Java programs.

5) Performance in multi-threaded programming can be frustrating. Often serial programs can perform better than well-designed parallel programs in real world situations. Your intuitions may be very wrong and threading overhead can add up quickly if you are not careful.

6) Because of this, using something like GPars should probably mostly be motivated by considerations of good program design and allowing for clean and maintainable code.

7) The Clojure and Scala people have done a lot thinking about concurrency, leverage as many of their ideas and as much of their know-how has you can.

Plain Vanilla Groovy Helps for Multi-Threaded Programming

1) There isn't anything in Groovy that is dedicated to helping multi-threaded programming specifically. But, a lot of the groovyness of groovy can really help with multi-threaded programming.

2) Closures: By far the biggest thing holding Java programmers back from writing multi-threaded code much more easily is lack of lambdas (due to arrive in JDK 8, but don't hold your breath, they were supposed to arrive in JDK 7). Since Groovy has closures, and closures are lambdas, this is simply not a problem for Groovy.

3) Lexical scoping: The ability to access lexically scoped variables inside of closures can be both a source of power and of subtle bugs in multi-threaded programming.

4) Groovy will automatically synthesize classes that implement single method interfaces. Since Runnable and Callable are both single method interfaces, this can really simplify lots of multi-threaded code.

5) Currying adapts argument lists of closures and methods. This allows for delayed invocation and can simplify the use of Futures.

6) AST Transformations can implement lots of error prone and boiler plate code in an elegant way. Several AST Transformations directly target the needs of multi-threaded programmers.

7) MetaClass hacking: Groovy can fill in missing functionality, correct bad functionality, and provide abstractions not available in Java.

Code Samples Illustrating These Principles

1) Timing.groovy (closures)

2) DemoThreadCreate.groovy (closures and uses interface implementation under the hood)

3) Interrupts.groovy and DemoInterrupts (closures and interface implementation)

4) DemoValues.groovy and ImmutableList.groovy (AST Transformations)

5) DemoLocking.groovy (AST Transformations)

6) DemoParallelCollect.groovy (closures and currying)

7) DemoLockEnhancements (closures and metaclass hacking)

Modern Threading (Moving Towards GPars)

1) Creating and managing threads is error prone. Threads should be managed by a pool and the pool should be managed by the framework.

2) Manual Synchronization leads to race conditions, data races, deadlocks, performance problems, etc. Using a framework which handles synchronization for you can alleviate, but not eliminate, many of these problems.

3) Immutability is highly encouraged. In the functional world, this is the default. While this is probably not realistic for a language like Groovy (and truthfully it may not be realistic for any language), the less that mutates, the easier it is to write correct multi-threaded code.

4) Having multiple threads synchronize access to shared data leads to deadlocks and performance issues. It is also really hard to get right. If immutability is not possible, try to isolate mutability to a single thread.

5) Nice YouTube video (from JAX 2011) on Modern Java Concurrency: https://www.youtube.com/watch?v=qrCUy9H76IA They give 3 libraries to watch as being the future of Java concurrency, and one of them is GPars.