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) Since day one Java and the JVM have given us the basic tools for concurrent/multi-threaded programming: Thread, Runnable, synchronized, final, volatile, etc.

3) 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.

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

5) 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, not synchronizing by default, et alia.

6) 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) 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.

3) 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.

4) 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.

5) Instead of focusing on performance at first, focus on getting the architecture of a concurrent program correct. Groovy can help remove boilerplate and repetitive code which tends to obfuscate what your program is actually doing. GPars provides several frameworks

Groovy Helps for Multi-Threaded Programming

1) There are lots of areas in which the Java language provides a less than optimal solution for writing threaded/concurrent code. The following is a sample of areas where Groovy makes it easier to write correct and more easily understood multi-threaded code.

2) Immutability. It's hard to overestimate the advantages of writing properly immutable code in a multi-threaded/concurrent application. But, Java variables are by default mutable. Groovy variables (unlike Scala and Clojure variables) are also by default mutable. However, the powerful @Immutable transformation can guarantee that your code is properly immutable. Sample code: DemoValues.groovy and ImmutableList.groovy

3) Thread control. Starting a thread is relatively easy in Java (though it's even easier in Groovy). Convincing a thread in Java to shutdown cleanly is tricky and hard to do in a generic way. Groovy closures, lexical scoping, and automatic interface implementation provide a way to manage thread shutdown in a clean and generic way. Sample code: Interrupts.groovy and DemoInterrupts.groovy.

4) Locking and lock management. Java provides the synchronized keyword to manage native monitors cleanly. But, it doesn't provide the same facilities for managing the new JDK 5 locks such as ReentrantLock and ReentrantReadWriteLock. Groovy can make using these new locks as easy as using the synchronized keyword. Sample code: DemoLocking.groovy

GPars Helps for Multi-Threaded Programming

1) GPars provides thread pooling and management facilities. These facilities are used throughout the various GPars frameworks, and can also be used independently of them. Sample code: Email.groovy.

2) Actors. Probably the best known concurrent framework. The main idea in actors is that each actor can maintain its own state without worrying about synchronization. Code communicates to actors, and actors communicate to each other by means of message passing. The messages are serialized in the actors message queue which is why synchronization is not needed inside actors. It is important to remember that there is no 1:1 relationship between actors and threads. Sample code: FileManager.groovy, DemoEachFileParallel.groovy, DemoWebServer.groovy.

3) Dataflow. In my opinion, this is one of the most useful and easiest to use framework in GPars. The dataflow framework builds on the “futures” concept. It also decouples JVM threads from units of work, called tasks. However, the nicest part is that Dataflow provides a way of managing dependencies between tasks which makes it much easier to develop complete workflows that are multi-threaded. Even better, deadlocks are usually deterministic in dataflow programs. This is nice because complex workflows are especially prone to deadlocks. Sample code: DemoWebCrawling.groovy

4) Fork/Join. Fork/Join is a huge topic and I won't spend a long time on it. Fork/Join is trying to solve a very specific problem. When you are writing concurrent code you generally want to parallelize either I/O or computation. It's generally pretty easy to speed up I/O by running operations in parallel. But, computation is much harder to speed up using generic threading frameworks. This is because things like thread swapping, synchronization, and frameworks overhead can easily swamp the time spend doing actual computation. It's fairly easy to make compute intensive code go **slower** in parallel. Fork/Join tries to address all of these concerns. Fork/Join pools made it into JDK 7, but most of the Fork/Join framework did not because Java still lacks lambdas/closures. However, since Groovy has closures right now, you can use the full Fork/Join framework today with Groovy and GPars. Sample code: DemoForkJoin.groovy.

5) Parallel Collections/Arrays. Uses Fork/Join to provide multi-threaded support for basic computational tasks like sorting and searching. One warning about parallel collections, good performance only happens when hotspot kicks in. Sample code: DemoParallelArrays.groovy

6) 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.