Fire and Forget Concurrency

1) Oftentimes concurrency is just a matter of running lots of independent tasks in parallel. If you do not need to coordinate tasks or synchronize data, multi-threading becomes a lot easier.

2) You can "fire and forget" in Groovy quite easily because Thread.start will run a closure asynchronously without much work.

3) However, if you are doing this often, you want to consider using a thread pool. GParsExecutorsPool.withPool and GParsExecutorsPool.withExistingPool can help you do this easily.

4) Relevant Code: DemoThreadCreate.groovy and DemoAsynchronous.groovy

Dataflow Variables

1) Are a simplification and significant enhancement of Java Futures

2) Are excellent for when tasks have complex dependencies, but still need to be parallelized. They can really help keep code robust. Hard coding dependencies leads to brittle and hard to manage code. It's better to code things as simple tasks and let the dataflow framework work out the dependencies automatically.

4) As an added benefit, deadlocks with dataflow variables are almost always deterministic, meaning they are repeatable in your development environment. This is actually very helpful because complex dependencies are prone to deadlock, but since they are deterministic, they are easy to spot and fix in development.

5) The main idea is that dataflow variables can only be written to once, but read from many times. All reads block until the variable is written to.

6) Relevant Code: DemoIntegrate.groovy and DemoWebCrawling.groovy

Actors

1) Are great at managing tasks independently of threads. For example, you might have many times more actors being run than there are threads in the underlying pool.

2) Are excellent at modelling problems that can be thought of as Master/Slave or Manager/Workers problems.

3) Provide isolated mutability by forcing actors to interact with each other by passing messages. Each actor can maintain as much mutable state as they want, but that mutable state should not be shared with other actors, unless done by message passing.

4) Messages passed in actor frameworks should ideally be immutable.

5) Messages passed don't have to be immutable, but each actor should striclty enforce a "hands off" policy once a message is sent that contains mutable data. The actors framework does provide enough synchronization to guarantee that the Java Memory Model is not violated.

6) Beware of chatty protocols. Each message sent may result in a context switch which is costly.

7) In a Manager/Worker scenario, the Manager should maintain all state.

8) Recursive problems can be modelled using actors using a Manager/Worker design where the master maintains a stack for managing state.

9) Note for GUI programmers. Manager/Worker designs look a lot like how most GUI frameworks manage state and concurrency. The event thread or event loop is the manager. Long running tasks are executed by worker threads, and the main event thread is notified about completed tasks by some sort of message passing or event. All state is managed by the event thread/loop, just like the manager does in a manager/worker design. Swing works this way.

10) Relevant Code: DemoEachFileParallel.groovy and DemoWebServer.groovy

Fork/Join

1) Newest concurrency framework for Java. The Fork/Join framework was developed as part of the JSR 166y project. However, only part of the 166y code made it into JDK 7. The reason for this is that a lot of the Fork/Join framework is awkward and verbose without lambdas, and lambdas didn't ship with JDK 7, therefore a lot didn't make it in.

2) However, since Groovy has closures, the complete 166y framework makes sense inside of Groovy right now. Because of this, the GPars people included the full 166y framework inside of GPars.

3) Most concurrency frameworks are built around speeding up course grained tasks that have lots of I/O. Because most of the time is spent in I/O, there isn't much problem with threading and synchronization overhead swamping run times. The previous JDK 5 threading improvements, such as Executors and Futures, work really well in this kind of environment.

4) However, compute intensive operations can very easily be swamped by threading and concurrency overhead in concurrency frameworks designed with I/O and long running tasks in mind. Fork/Join is built to minimize overhead for these types of compute intensive tasks.

5) Fork/Join is useful when the following hold for your code:
   a. You are dealing with large datasets
   b. The design constraints of Fork/Join are kept (no I/O, no synchronization)
   c. The code is on a hot path in your application (otherwise hotspot will not speed it up)
   d. The code is run in server mode (client mode usually disables hotspot)

6) Relevant Code: DemoForkJoin.groovy

Parallel Collections/Arrays

1) Are probably the simplest to understand of all the concurrency frameworks inside GPars.

2) However, the parallel operations are not a panacea. Under real world conditions, it's easily possible that parallel algorithms will perform worse than serial algorithms. For example, if you are trying to

sort a list with a couple thousand items in it, don't even bother with parallel collections. By the time a parallel algorithm can set up its machinery and divide up the work, the serial algorithm will be done.

3) If performance is your primary consideration in using parallel collections/arrays, you will probably need a good understanding of your problem domain and a good understanding of the underlying Fork/Join machinery to get optimal performance.

4) The timing experiments I have done seem to show that the convenience parallel collection classes (such as findAllParallel) don't buy you much, and may even slow down your computations. Still, they are easy enough to use and if your measurements indicate that they give you a performance boost, use them.

5) However, if you really have a need for speed, you will most likely need to work with parallel arrays. This is still very easy to do inside of groovy, the only real difference being the need to use more traditional functional programming names for the functionality you need (map and reduce instead of collect and inject, etc.).

6) eachParallel is still useful as a way of executing tasks simultaneously while blocking on all results. This is much easier than starting threads and joining them or even using ExecutorService and Futures. If you are doing I/O or synchronizing, use GParsExecutorsPool instead of GParsPool.

7) Relevant Code: DemoParallelEach.groovy and DemoParallelArrays.groovy