Groovy Concurrency

Alex Miller
Revellytix
GPars

- Groovy library - http://gpars.codehaus.org/
- Open-source
- Great team: Václav Pech, Dierk König, Alex Tkachman, Russel Winder, Paul King
- Integration with Griffon, Grails, etc
Concurrency Patterns

• Asynchronous processing
• Computational processing
• Managing mutable state
  • Actor messaging
  • Agents
• Dataflow variables
Asynchronous Processing

*Improve perceived performance by pushing I/O-bound work onto a different thread.*
Serialized Processing

main

Retrieve user1 tweets

Retrieve user2 tweets

Process tweets

Twitter

Twitter
Asynchronous Processing
GParsExecutorsPool

- Leverages java.util.concurrent.Executor
  - void execute(Runnable command)
- Decouple task submission from task execution
- GParsExecutorsPool.withPool() blocks add:
  - async() - takes closure, returns Future
  - callAsync() - same, but takes args
Future

• Future represents the result of an asynchronous computation
• Poll for results, block till available, etc
• Cancel, check for cancellation
GParsExecutorsPool.withPool {
    def retrieveTweets = {
        query -> recentTweets(api, query)
    }

trends.each {
    retrieveTweets.callAsync(it.query)
}
Use multiple cores to reduce the elapsed time of some CPU-bound work.
Executor Pools

Same pools used for asynch are also good for transaction processing. Leverage more cores to process the work.
Work pools and Queues
Fork/Join Pools

- DSL for new JSR 166y construct (JDK 7)
- DSL for ParallelArray over fork/join
- DSL for map/reduce and functional calls
Fork/Join Pools
Fork/Join vs Executors

- Executors (GParsExecutorsPool) tuned for:
  - **small** number of threads (4-16)
  - **medium-grained** I/O-bound or CPU-bound tasks

- Fork/join (GParsPool) tuned for:
  - **larger** number of threads (16-64)
  - **fine-grained** computational tasks, possibly with **dependencies**
Divide and Conquer

Task: Find max value in an array
Divide and Conquer
Divide and Conquer
ParallelArray

• Most business problems don’t really look like divide-and-conquer

• Common approach for working in parallel on a big data set is to use an array where each thread operates on independent chunks of the array.

• ParallelArray implements this over fork/join.
### ParallelArray

<table>
<thead>
<tr>
<th>Order</th>
<th>Overdue Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/15</td>
<td>15</td>
</tr>
<tr>
<td>1/15</td>
<td>45</td>
</tr>
<tr>
<td>2/1</td>
<td>30</td>
</tr>
</tbody>
</table>

- **Starting orders w/due dates**
- **Filter only overdue**
- **Map to days overdue**
- **Avg**

**30 days**
def isLate = { order -> order.dueDate > new Date() }  
def daysOverdue = { order -> order.daysOverdue() }  

GParsPool.withPool {  
def data = createOrders()  
  .filter(isLate)  
  .map(daysOverdue)  

  println("# overdue = " + data.size())  
  println("avg overdue by = " + (data.sum() / data.size()))  
}
Managing Mutable State
It’s the mutable state, stupid.

Make fields final unless they need to be mutable.

Immutable objects are automatically thread-safe.

Encapsulation makes it practical to manage the complexity.

Guard each mutable variable with a lock.

Guard all variables in an invariant with the same lock.

Hold locks for the duration of compound actions.

A program that accesses a mutable variable from multiple threads without synchronization is a broken program.

Don’t rely on clever reasoning about why you don’t need to synchronize.

Include thread safety in the design process - or explicitly document that your class is not thread-safe.

Document your synchronization policy.
• It’s the **mutable** state, stupid.
• Make fields final unless they need to be **mutable**.
• **Immutable** objects are automatically thread-safe.
• Encapsulation makes it practical to manage the complexity.
• Guard each **mutable** variable with a **lock**.
• Guard all variables in an invariant with the same **lock**.
• Hold **locks** for the duration of compound actions.
• A program that accesses a **mutable** variable from multiple threads without synchronization is a broken program.
• Don’t rely on clever reasoning about why you don’t need to **synchronize**.
• Include thread safety in the design process - or explicitly document that your class is not thread-safe.
• Document your **synchronization** policy.
The problem with shared state concurrency...
The problem with shared state concurrency... is the shared state.
Actors

- No shared state
- Lightweight processes
- Asynchronous, non-blocking message-passing
- Buffer messages in a mailbox
- Receive messages with pattern matching
- Concurrency model popular in Erlang, Scala, etc
Actors
Actors

start
Actors

send

start
Actors

receive
class Player extends AbstractPooledActor {
    String name
    def random = new Random()

    void act() {
        loop {
            react {
                // player replies with a random move
                reply
                    Move.values()[random.nextInt(Move.values().length)]
                }
            }
        }
    }
}
Agent

- Like Clojure Agents
- Imagine wrapping an actor around mutable state...
- Messages are:
  - functions to apply to (internal) state OR
  - just a new value
- Can always safely retrieve value of agent
Simple example...

- Use generics to specify data type of wrapped data
- Pass initial value on construction
- Send function to modify state
- Read value by accessing val

```java
def stuff = new Agent<List>([])

// thread 1
stuff.send( {it.add("pizza")} )

// thread 2
stuff.send( {it.add("nachos")} )

println stuff.val
```
```java
class BankAccount extends Agent<Long> {
  def BankAccount() { super(0) }
  private def deposit(long deposit)
  { data += deposit }
  private def withdraw(long deposit)
  { data -= deposit }
}

final BankAccount acct = new BankAccount()

final Thread atm2 = Thread.start {
  acct << { withdraw 200 }
}

final Thread atm1 = Thread.start {
  acct << { deposit 500 }
}

[atm1,atm2]*.join()
println "Final balance: ${ acct.val }"
```
Dataflow Variables

• A variable that computes its value when its’ inputs are available
• Value can be set only once
• Data flows safely between variables
• Ordering falls out automatically
• Deadlocks are deterministic
Dataflow Tasks

- Logical tasks
- Scheduled over a thread pool, like actors
- Communicate with dataflow variables
final def x = new DataFlowVariable()
final def y = new DataFlowVariable()
final def z = new DataFlowVariable()

task {
    z << x.val + y.val
    println "Result: \${z.val}"
}

task {
    x << 10
}

task {
    y << 5
}
Dataflow support

- Bind handlers - can be registered on a dataflow variable to execute on bind
- Dataflow streams - thread-safe unbound blocking queue to pass data between tasks
- Dataflow operators - additional abstraction, formalizing inputs/outputs
Sir Not- Appearing-In-This-Talk

- Caching - memoize
- CSP
- GPU processing
Alex Miller

Twitter:  @puredanger
Blog:       http://tech.puredanger.com
Code:      http://github.com/puredanger/gpars-examples
Conference: Strange Loop
           http://strangeloop2010.com