Guide To Getting Started

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

A Few Assumptions

Let's set out a few assumptions before we start :

- You know and use **Groovy** and/or **Java** : otherwise you'd not be investing your valuable time studying a concurrency and parallelism library for **Groovy** and/or **Java**.
- You definitely want to write code employing concurrency and parallelism concepts.
- If you are not using **Groovy**, you are prepared to pay the inevitable verbosity tax of using **Java**.
- You target multi-core hardware with your code.
- You appreciate that in concurrent and parallel code things can happen at any time, in any order, and, more likely, with more than one thing happening at once.

Ready ?

With those assumptions in place, we can get started.

It’s becoming more and more obvious that dealing with concurrency and parallelism at the thread/synchronized/lock level, as provided by the JVM, is far too low a level to be safe and comfortable.

Many high-level concepts, such as **actors** and **dataflow** have been around for quite some time. Parallel-chipped computers have been in use, at least in data centres if not on the desktop, long before multi-core chips hit the hardware mainstream.

So now is the time to adopt these higher-level abstractions into the mainstream software industry.

This is what **GPars** enables for the **Groovy** and **Java** languages, allowing these hackers to use higher-level abstractions and, therefore, make development of concurrent and parallel software easier and less error prone.

The concepts available in **GPars** can be categorized into three groups:

- *Code-level helpers* - Constructs that can be applied to small parts of your code-base, such as an individual algorithms or data structure, without any major changes in the overall project architecture
  - Parallel Collections
  - Asynchronous Processing
  - Fork/Join (Divide/Conquer)
- *Architecture-level concepts* - Constructs that need to be taken into account when designing the project structure
  - Actors
  - Communicating Sequential Processes (CSP)
  - Dataflow
  - Data Parallelism
- *Shared Mutable State Protection* - More than 95% of the current use of shared mutable states can be avoided using proper abstractions. Good abstractions are still necessary for the remaining 5% of those use cases, i.e. when shared mutable state cannot be avoided.
  - Agents
  - Software Transactional Memory (not fully implemented in **GPars** as yet)

Downloading and Installing

GPars is now distributed as part of **Groovy**. So if you have a **Groovy** installation, you should already have **GPars**. Your exact version of **GPars** will, of course, depend on which version of **Groovy** you use.

If you don't already have **GPars**, and you do have **Groovy**, then perhaps you should upgrade your **Groovy** !

If you need to download it, you can download **Groovy** from here.

If you don't have a **Groovy** installation, but use **Groovy** by using dependencies or perhaps, just having the **groovy-all** artifact, then you will need to get **GPars**. Also if you want to use a different version of **GPars** to the one bundled witnh **Groovy**, or have an old **GPars**-free **Groovy** that you cannot upgrade, you will need to get **GPars**. The ways to download **GPars** are:

- Download the artifact from a repository and add it and all the transitive dependencies manually.
- Specify a dependency in Gradle, **Maven**, or Ivy (or Gant, or Ant) build files.
- Use Grapes (especially useful for **Groovy** scripts).
- Download and install it from here.

If you're building a Grails or a Griffon application, you can use the appropriate plugins to fetch our jar files for you.

The GPars Artifact

As noted above, **GPars** is now distributed as standard with **Groovy**. If however, you have to manage this dependency manually, the **GPars** artifact is in the main **Maven** repository and was in the Codehaus main and snapshots repositories before they closed.

Release versions can be found in the **Maven** main repositories but, for now, the current development version (SNAPSHOT) was in the Codehaus snapshots repository. We're moving it to another location.

To use **GPars** from **Gradle** or Grapes, use the specification:

A **Gradle** *Example*

```
"org.codehaus.gpars:gpars:1.2.0"
```

You may need to add our snapshot repository manually to the search list in this latter case. Using **Maven** the dependency is:

A Sample **Maven** *Declaration*

```
<dependency>
    <groupId>org.codehaus.gpars</groupId>
    <artifactId>gpars</artifactId>
    <version>1.3.0</version>
</dependency>
```

Transitive Dependencies

GPars as a library depends on **Groovy** versions later than 2.2.1. Also, the **Fork/Join** concurrency library must be available. This comes as standard with **Java** 7.

GPars 2.0 will depend on **Java 8** and will only be usable with **Groovy 3.0** and later.

Please visit the page Integration on the **GPars** website for more details.

A Hello World Example

Once you're setup, try the following **Groovy** script to confirm your setup is functioning properly.

A **Groovy** *Sample*

```
import static groovyx.gpars.actor.Actors.actor

/**
 * A demo showing two cooperating actors. The decryptor decrypts received messages
 * and replies them back.  The console actor sends a message to decrypt, prints out
 * the reply and terminates both actors.  The main thread waits on both actors to
 * finish using the join() method to prevent premature exit, since both actors use
 * the default actor group, which uses a daemon thread pool.
 * @author Dierk Koenig, Vaclav Pech
 */

def decryptor = actor {
    loop {
        react { message ->
            if (message instanceof String) reply message.reverse()
            else stop()
        }
    }
}

def console = actor {
    decryptor.send 'lellarap si yvoorG'
    react {
        println 'Decrypted message: ' + it
        decryptor.send false
    }
}

[decryptor, console]*.join()
```

You receive a message "Decrypted message: **Groovy** is parallel" printed on the console.

Java API

GPars has been designed primarily for use with the **Groovy** programming language. Of course all **Java** and **Groovy** programs are just bytecodes running on the JVM, so **GPars** can be used with **Java** source.

Despite being aimed at **Groovy**, the solid technical foundation, plus the good performance characteristics of **GPars** makes it an excellent library for **Java** programs too. In fact most of **GPars** is written in **Java**, so there is no performance penalty for **Java** applications using **GPars**.

For details please refer to the **Java API** section.

To quick-test **GPars** with the **Java API**, compile and run the following **Java** code:

Another **Java** *Sample*

```
import groovyx.gpars.MessagingRunnable;
import groovyx.gpars.actor.DynamicDispatchActor;

public class StatelessActorDemo {

    public static void main(String[] args) throws InterruptedException {
        final MyStatelessActor actor = new MyStatelessActor();
        actor.start();
        actor.send("Hello");
        actor.sendAndWait(10);

        actor.sendAndContinue(10.0, new MessagingRunnable<String>() {
            @Override protected void doRun(final String s) {
                System.out.println("Received a reply " + s);
            }
        });
    }
}

class MyStatelessActor extends DynamicDispatchActor {
    public void onMessage(final String msg) {
        System.out.println("Received " + msg);
        replyIfExists("Thank you");
    }

    public void onMessage(final Integer msg) {
        System.out.println("Received a number " + msg);
        replyIfExists("Thank you");
    }

    public void onMessage(final Object msg) {
        System.out.println("Received an object " + msg);
        replyIfExists("Thank you");
    }
}
```

Artifacts maybe needed

Remember though that you will almost certainly have to add the **Groovy** artifact to the build as well as the **GPars** artifact. **GPars** may well work at **Java** speeds with **Java** applications, but it still has some compilation dependencies on **Groovy**.

Code Conventions

We follow certain conventions in tour code samples. Understanding these conventions may help you read and comprehend **GPars** code samples better.

- The *leftShift* operator **'<<'** has been overloaded on **actors**, **agents** and **dataflow** expressions (both variables and streams) to mean *send* a message or *assign* a value.

Using The leftShift Operator

```
myActor << 'message'

myAgent << {account -> account.add('5 USD')}

myDataflowVariable << 120332
```

- On **actors** and **agents**, the default *call()* method has been also overloaded to mean *send* . So sending a message to an actor or agent may look like a regular method call.

```
myActor "message"

myAgent {house -> house.repair()}
```

- The *rightShift* operator **'>>'** in **GPars** has the *when bound* meaning. So

```
 myDataflowVariable >> {value -> doSomethingWith(value)}
```

will schedule the closure to run only after *myDataflowVariable* is bound to a value, with the value as a parameter.

Usage

In samples, we tend to statically import frequently used factory methods:

- GParsPool.withPool()
- GParsPool.withExistingPool()
- GParsExecutorsPool.withPool()
- GParsExecutorsPool.withExistingPool()
- Actors.actor()
- Actors.reactor()

- Actors.fairReactor()
- Actors.messageHandler()
- Actors.fairMessageHandler()
- Agent.agent()
- Agent.fairAgent()
- Dataflow.task()
- Dataflow.operator()

It’s more a matter of style preferences and personal taste, but we think static imports make the code more compact and readable.

Getting Set Up In An IDE

Adding the **GPars** jar files to your project or defining the appropriate dependencies in pom.xml should be enough to get you started with **GPars** in your IDE.

GPars DSL recognition

IntelliJ IDEA in both the free *Community Edition* and the commercial *Ultimate Edition* will recognize the **GPars** domain specific languages, complete methods like *eachParallel()* , *reduce()* or *callAsync()* and validate them. **GPars** uses the **Groovy** DSL mechanism, which teaches IntelliJ IDEA the DSLs as soon as the **GPars** jar file is added to the project.

Applicability of Concepts

GPars provides a lot of concepts to pick from. We're continuously building and updating our documents to help users choose the right level of abstraction for their tasks at hands. Please, refer to the Concepts Compared page for details.

To briefly summarize the suggestions, here are some basic guide-lines:

- You're looking at a collection, which needs to be **iterated** or processed using one of the many beautiful **Groovy** collection methods, like *each()* , *collect()* , *find()* etc.. Suppose that processing each element of the collection is independent of the other items, then using **GPars parallel collections** can be appropriate.
- If you have a **long-lasting calculation** , which may safely run in the background, use the **asynchronous invocation support** in **GPars**. Since **GPars** asynchronous functions can be composed, you can quickly parallelize tyhese complex functional calculations without having to mark independent calculations explicitly.
- Say you need to **parallelize** an algorithm. You can identify a set of **tasks** with their mutual dependencies. The tasks typically do not need to share data, but instead some tasks may need to wait for other tasks to finish before starting. Now you're ready to express these dependencies explicitly in code. With **GPars dataflow tasks**, you create internally sequential tasks, each of which can run concurrently with the others. **Dataflow** variables and channels provide the tasks with the capability to declare their dependencies and to exchange data safely.
- Perhaps you can't avoid using **shared mutable state** in your logic. Multiple threads will be accessing shared data and (some of them) modifying it. A traditional locking and synchronized approach feels too risky or unfamiliar? Then go for **agents** to wrap your data and serialize all access to it.
- You're building a system with high concurrency demands. Tweaking a data structure here or task there won't cut it. You need to build the architecture from the ground up with concurrency in mind. **Message-passing** might be the way to go. Your choices could include :
  - **Groovy CSP** to give you highly deterministic and composable models for concurrent processes. A model is organized around the concept of **calculations** or **processes**, which run concurrently and communicate through synchronous channels.
  - If you're trying to solve a complex data-processing problem, consider **GPars dataflow operators** to build a data flow network. The concept is organized around event-driven transformations wired into pipelines using asynchronous channels.
  - **Actors** and **Active Objects** will shine if you need to build a general-purpose, highly concurrent and scalable architecture following the object-oriented paradigm.

Now you may have a better idea of what concepts to use on your current project. Go check out more details on them in our **User Guide**.

What’s New

The new **GPars 1.3.0** release introduces several enhancements and improvements on top of the previous release, mainly in the dataflow area.

Check out the JIRA release notes.

Project changes

Breaking Changes

See the Breaking Changes listing for the list of breaking changes.

Asynchronous functions

TBD

Parallel collections

TBD

Fork / Join

TBD

Actors

- Remote actors
- Exception propagation from active objects

Dataflow

- Remote dataflow variables and channels
- Dataflow operators accepting variable number arguments
- Select made @CompileStatic compatible

Agent

- Remote agents

Stm

TBD

Other

- Raised the JDK dependency to version 1.7
- Raised the **Groovy** dependency to version 2.2
- Replaced the **jsr-177y fork-join** pool implementation with the one from JDK 1.7
- Removed the dependency on **jsr-166y**

Renaming Hints

Java API – Using GPars from Java

Using **GPars** is very addictive, I guarantee. Once you get hooked you won’t be able to code without it. If the world forces you to write code in **Java**, you will still be able to benefit from most of **GPars** features.

Java API specifics

Some parts of **GPars** are irrelevant in **Java** and it is better to use the underlying **Java** libraries directly: * Parallel Collection – use jsr-166y library’s **Parallel Array** directly until **GPars 1.3.0** becomes available * Fork/Join – use **jsr-166y** library’s **Fork/Join** support directly until **GPars 1.3.0** becomes available * Asynchronous functions – use **Java** executor services directly

The other parts of **GPars** can be used from **Java** just like from **Groovy**, although most will miss the **Groovy** DSL capabilities.

GPars Closures in Java API

To overcome the lack of closures as a language element in **Java** and to avoid forcing users to use **Groovy** closures directly through the **Java** API, a few handy wrapper classes have been provided to help you define callbacks, **actor** body or **dataflow** tasks. * groovyx.gpars.MessagingRunnable - used for single-argument callbacks or **actor** body * groovyx.gpars.ReactorMessagingRunnable - used for **ReactiveActor** body * groovyx.gpars.DataflowMessagingRunnable - used for **dataflow** operators' body

These classes can be used in places where the **GPars API** expects a **Groovy** closure.

Actors

The *DynamicDispatchActor* as well as the *ReactiveActor* classes can be used just like in **Groovy**:

A **DynamicDispatchActor** *Sample*

```java
import groovyx.gpars.MessagingRunnable;
import groovyx.gpars.actor.DynamicDispatchActor;

public class StatelessActorDemo {
    public static void main(String[] args) throws InterruptedException {
        final MyStatelessActor actor = new MyStatelessActor();
        actor.start();
        actor.send("Hello");
        actor.sendAndWait(10);
        actor.sendAndContinue(10.0, new MessagingRunnable<String>() {
            @Override protected void doRun(final String s) {
                System.out.println("Received a reply " + s);
            }
        });
    }
}

class MyStatelessActor extends DynamicDispatchActor {
    public void onMessage(final String msg) {
        System.out.println("Received " + msg);
        replyIfExists("Thank you");
    }

    public void onMessage(final Integer msg) {
        System.out.println("Received a number " + msg);
        replyIfExists("Thank you");
    }

    public void onMessage(final Object msg) {
        System.out.println("Received an object " + msg);
        replyIfExists("Thank you");
    }
}
```

There are few differences between **Groovy** and **Java** for **GPars** use, but notice the callbacks instantiating the *MessagingRunnable* class in place of a **Groovy** closure.

A **MessagingRunnable** *Sample*

```
import groovy.lang.Closure;
import groovyx.gpars.ReactorMessagingRunnable;
import groovyx.gpars.actor.Actor;
import groovyx.gpars.actor.ReactiveActor;

public class ReactorDemo {
    public static void main(final String[] args) throws InterruptedException {

        final Closure handler = new ReactorMessagingRunnable<Integer, Integer>() {
            @Override protected Integer doRun(final Integer integer) {
                return integer * 2;
            }
        };
        final Actor actor = new ReactiveActor(handler);
        actor.start();

        System.out.println("Result: " +  actor.sendAndWait(1));
        System.out.println("Result: " +  actor.sendAndWait(2));
        System.out.println("Result: " +  actor.sendAndWait(3));
    }
}
```

Convenience Factory Methods

Obviously, all the essential factory methods to build actors quickly are available where you'd expect them.

Factory Samples

```
import groovy.lang.Closure;
import groovyx.gpars.ReactorMessagingRunnable;
import groovyx.gpars.actor.Actor;
import groovyx.gpars.actor.Actors;

public class ReactorDemo {
    public static void main(final String[] args) throws InterruptedException {
        final Closure handler = new ReactorMessagingRunnable<Integer, Integer>() {
            @Override protected Integer doRun(final Integer integer) {
                return integer * 2;
            }
        };
        final Actor actor = Actors.reactor(handler);

        System.out.println("Result: " +  actor.sendAndWait(1));
        System.out.println("Result: " +  actor.sendAndWait(2));
        System.out.println("Result: " +  actor.sendAndWait(3));
    }
}
```

Agents

Agent Samples

```
 import groovyx.gpars.MessagingRunnable;
 import groovyx.gpars.agent.Agent;

 public class AgentDemo {

     public static void main(final String[] args) throws InterruptedException {

         final Agent counter = new Agent<Integer>(0);
         counter.send(10);
         System.out.println("Current value: " + counter.getVal());
         counter.send(new MessagingRunnable<Integer>() {
             @Override protected void doRun(final Integer integer) {
                 counter.updateValue(integer + 1);
             }
         });

         System.out.println("Current value: " + counter.getVal());
     }
 }
```

Dataflow Concurrency

Both *DataflowVariables* and *DataflowQueues* can be used from **Java** without any hiccups. Just avoid the handy overloaded operators and go straight to the methods, like *bind* , *whenBound*, *getVal* and other.

You may also continue to use **dataflow** tasks passing them instances of *Runnable* or *Callable* just like groovy closures.

Dataflow Samples

```java
import groovyx.gpars.MessagingRunnable;
import groovyx.gpars.dataflow.DataflowVariable;
import groovyx.gpars.group.DefaultPGroup;

import java.util.concurrent.Callable;

public class DataflowTaskDemo {

    public static void main(final String[] args) throws InterruptedException {
        final DefaultPGroup group = new DefaultPGroup(10);

        final DataflowVariable a = new DataflowVariable();

        group.task(new Runnable() {
            public void run() {
                a.bind(10);
            }
        });

        final Promise result = group.task(new Callable() {
            public Object call() throws Exception {
                return (Integer)a.getVal() + 10;
            }
        });

        result.whenBound(new MessagingRunnable<Integer>() {
            @Override protected void doRun(final Integer integer) {
                System.out.println("arguments = " + integer);
            }
        });

        System.out.println("result = " + result.getVal());
    }
}
```

Dataflow Operators

The sample below should illustrate the main differences between **Groovy** and **Java** APIs for dataflow operators.

- Use the convenience factory methods when accepting lists of channels to create operators or selectors
- Use *DataflowMessagingRunnable* to specify the operator body
- Call *getOwningProcessor()* to get hold of the operator from within the body in order to e.g. bind output values

More Dataflow Samples

```
import groovyx.gpars.DataflowMessagingRunnable;
import groovyx.gpars.dataflow.Dataflow;
import groovyx.gpars.dataflow.DataflowQueue;
import groovyx.gpars.dataflow.operator.DataflowProcessor;

import java.util.Arrays;
import java.util.List;

public class DataflowOperatorDemo {

    public static void main(final String[] args) throws InterruptedException {
        final DataflowQueue stream1 = new DataflowQueue();
        final DataflowQueue stream2 = new DataflowQueue();
        final DataflowQueue stream3 = new DataflowQueue();
        final DataflowQueue stream4 = new DataflowQueue();

        final DataflowProcessor op1 = Dataflow.selector(Arrays.asList(stream1), Arrays
.asList(stream2), new DataflowMessagingRunnable(1) {
            @Override protected void doRun(final Object... objects) {
                getOwningProcessor().bindOutput(2*(Integer)objects[0]);
            }
        });

        final List secondOperatorInput = Arrays.asList(stream2, stream3);

        final DataflowProcessor op2 = Dataflow.operator(secondOperatorInput, Arrays
.asList(stream4), new DataflowMessagingRunnable(2) {
            @Override protected void doRun(final Object... objects) {
                getOwningProcessor().bindOutput((Integer) objects[0] + (Integer) objects
[1]);
            }
        });
        stream1.bind(1);
        stream1.bind(2);
        stream1.bind(3);
        stream3.bind(100);
        stream3.bind(100);
        stream3.bind(100);
        System.out.println("Result: " + stream4.getVal());
        System.out.println("Result: " + stream4.getVal());
        System.out.println("Result: " + stream4.getVal());
        op1.stop();
        op2.stop();
    }
}
```

Performance

In general, **GPars** overhead is identical irrespective of whether you use it from **Groovy** or **Java** and it tends to be very low anyway. **GPars actors**, for example, can compete head-to-head with other JVM **actor** options, like **Scala actors**.

Since **Groovy** code, in general, runs a little slower than **Java** code, due to dynamic method invocations, you might consider writing your code in **Java** to improve performance.

Typically numeric operations or frequent fine-grained method calls within a task or **actor** body may benefit from a rewrite into **Java**.

Prerequisites

All the **GPars** integration rules apply equally to **Java** projects and **Groovy** projects. You only need to include the **Groovy** distribution jar file in your project and your are good-to-go.

You may also want to check out our sample **Java-Maven** project for tips on how to integrate **GPars** into a **Maven**-based pure **Java** application – Sample **Java Maven** Project