A Guide To Tips

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

General GPars Tips

Grouping

High-level concurrency concepts, like **Agents, Actors** or **Dataflow** tasks and operators can be grouped around shared thread pools. The *PGroup* class and its sub-classes represent convenient **GPars** wrappers around thread pools. Objects created using the group's factory methods will share the group's thread pool.

A **xxxPGroup** *Sample*

```
def group1 = new DefaultPGroup()
def group2 = new NonDaemonPGroup()

group1.with {
    task {...}
    task {...}
    def op = operator(...) {...}
    def actor = actor{...}
    def anotherActor = group2.actor{...}  //will belong to group2
    def agent = safe(0)
}
```

Groups For Thread Pools

When customizing the thread pools for groups, consider using the existing **GPars** implementations - the *DefaultPool* or *ResizeablePool* classes. Or you may wish to create your own implementation of the *groovyx.gpars.scheduler.Pool* interface to pass to the *DefaultPGroup* or *NonDaemonPGroup* constructors.

Java API

Much of **GPars** functionality can be used from **Java** just as well as from **Groovy**. Checkout the `2.6 Java API - Using GPars from Java` section of our `User Guide`. Then experiment with the Maven-based stand-alone Java demo applications.

Take **GPars** with you wherever you go!

Performance

Your code in **Groovy** can be just as fast as code written in **Java**, **Scala** or any other programing language. This should not be surprising, since **GPars** is technically a solid tasty Java-made cake with a Groovy DSL frosting on it.

Unlike **Java**, however, with **GPars**, as well as with other DSL-friendly languages, you are very likely to experience a useful code speed-up for free. This speed-up comes from a better and cleaner design of your application.

Coding with a concurrency DSL will give you smaller code-base with code using the concurrency primitives as language constructs. So it's much easier to build robust concurrent applications, identify potential bottle-necks or errors and then eliminate them.

While this whole `User Guide` is describing how to use **Groovy** and **GPars** to create beautiful and robust concurrent code, we wanted to use some of these tips to highlight a few places where some code tuning or minor design compromises could give you interesting performance gains.

Parallel Collections

Methods like parallel collection processing, like *eachParallel()* , *collectParallel()* and such-like, use *Parallel Array* , an efficient tree-like data structure behind the scenes. This data structure has to be built from the original collection each time you call any of the parallel collection methods. Thus when chaining parallel method calls, you might consider using the *map/reduce* API instead or alternatively, use the *ParallelArray* API directly to avoid the *Parallel Array* creation overhead.

A Sample of Parallel Finds

```
import groovyx.gpars.GParsPool;
GParsPool.withPool {
    people.findAllParallel{it.isMale()}.collectParallel{it.name}.any{it == 'Joe'}
    people.parallel.filter{it.isMale()}.map{it.name}.filter{it == 'Joe'}.size() > 0
    people.parallelArray.withFilter({it.isMale()} as Predicate).withMapping({it.name} as
Mapper).any{it == 'Joe'} != null
}
```

In many scenarios, changing the pool size from the default value can give you performance benefits. Especially if your tasks perform IO operations, such as file or database access, networking, etc. So increasing the number of threads in the pool is likely to help performance.

To Boost Performance

```
import groovyx.gpars.GParsPool;
GParsPool.withPool(50) {
    ...
}
```

Since the closures you provide to the parallel collection processing methods are executed frequently, and concurrently, you may further slightly benefit from turning them into Java.

Actors

GPars actors are fast. *DynamicDispatchActors* and *ReactiveActors* are about twice as fast as the *DefaultActors* , since they don't have to maintain an implicit state between subsequent message arrivals. The *DefaultActors* are, in fact, on a par in performance with actors from **Scala** , which you rarely hear of as being slow.

If top performance is what you're looking for then identify patterns in your code

---

If top performance is what you're looking for, a good start is to identify the following patterns in your actor code:

A Pattern To Look For

```
actor {
    loop {
        react {msg ->
            switch(msg) {
                case String:...
                case Integer:...
            }
        }
    }
}
```

A Better Replacement : DynamicDispatchActor :

```
messageHandler {
    when{String msg -> ...}
    when{Integer msg -> ...}
}
```

The *loop* and *react* methods are rather costly to call.

Defining a *DynamicDispatchActor* or *ReactiveActor* as classes instead of using the *messageHandler* and *reactor* factory methods will also give you some more speed:

A Dynamic Sample

```
class MyHandler extends DynamicDispatchActor {
    public void handleMessage(String msg) {
        ...
    }

    public void handleMessage(Integer msg) {
        ...
    }
}
```

Now, convert that *MyHandler* class to Java to squeeze the last bit of performance from **GPars**.

Pool Adjustment

GPars allows you to group actors around thread pools, giving you the freedom to organize actors any way you like. It's always worthwhile to experiment with the actor pool size and type.

FJPool usually gives better characteristics that *DefaultPool* , but seems to be more sensitive to the number of threads in the pool. Sometimes using a *ResizeablePool* or *ResizeableFJPool* could help performance by automatic eliminating unneeded threads.

A Sample

```
def attackerGroup = new DefaultPGroup(new ResizeableFJPool(10))
def defenderGroup = new DefaultPGroup(new DefaultPool(5))

def attacker = attackerGroup.actor {...}
def defender = defenderGroup.messageHandler {...}
...
```

Agents

GPars Agents are even a bit faster in processing messages than **actors**. The advice to group **agents** wisely around thread pools and then tune the pool sizes and types applies to **agents** as well as **actors**. With **agents**, you may also benefit from submitting Java-written closures as messages.

Share Your Experience

The more we hear about **GPars** uses in the wild, the better we can adapt it for the future. Let us know how you use **GPars** and how it performs. Send us your benchmarks, performance comparisons or profiling reports to help us tune **GPars** for you. See this page for more details.

Hosted Environment

Hosted environments, such as *Google App Engine*, can impose additional restrictions on threading. For **GPars** to better integrate with these environments, the default thread factory and timer factory can be customized.

Hosted environments like *Google App Engine* impose restrictions on threading

The **GPars_Config** class provides static initialization methods allowing third parties to register their own implementations of the *PoolFactory* and *TimerFactory* interfaces. These can then be used to create default pools and timers for **Actors**, **Dataflow** and **PGroups**.

Some Static Methods To Initialize Objects

```
public final class GParsConfig {
    private static volatile PoolFactory poolFactory;
    private static volatile TimerFactory timerFactory;

    public static void setPoolFactory(final PoolFactory pool)

    public static PoolFactory getPoolFactory()

    public static Pool retrieveDefaultPool()

    public static void setTimerFactory(final TimerFactory timerFactory)

    public static TimerFactory getTimerFactory()

    public static GeneralTimer retrieveDefaultTimer(final String name, final boolean
daemon)

    public static void shutdown()
}
```

The custom factories should be registered immediately after application startup in order for **Actors** and **Dataflow** to be able to use them for their default groups.

Shutdown

The *GParsConfig.shutdown()* method can be used in managed environments to properly shutdown all

asynchronously running timers and free up the memory from all thread-local variables.

After the call to this method, the **GPars** library can no longer provide the declared services.

Compatibility

Some further compatibility issues can occur when running **GPars** in a hosted environment. The most noticeable one is probably the lack of **ForkJoinThreadPool** support in **GAE**. Functionality such as **Fork/Join** and **GParsPool** may not be available on some services as a result. However, **GParsExecutorsPool, Dataflow, Actors, Agents** and **Stm** should work normally even when using managed non-Java SE thread pools.