Guide To Data Parallelism

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

Focusing on data instead of processes helps us create robust concurrent programs. As a programmer, you define your data together with functions that should be applied to it and then let the underlying machinery process the data. Typically, a set of concurrent tasks will be created and submitted to a thread pool for processing.

In **GPars**, the **GParsPool** and **GParsExecutorsPool** classes give you access to low-level data parallelism techniques. The **GParsPool** class relies on the **Fork/Join** implementation introduced in **JDK 7** and offers excellent functionality and performance. The **GParsExecutorsPool** is provided for those who still need to use the older Java executors.

There are three fundamental domains covered by the **GPars** low-level data parallelism:

- Processing collections concurrently
- Running functions (closures) asynchronously
- Performing **Fork/Join** (Divide/Conquer) algorithms

The API described here is based on using **GPars** with **JDK**7. It can be used with later JDKs, but **JDK8** introduced the **Streams** framework which can be used directly from **Groovy** and, in essence, replaces the **GPars** features covered here. Work is underway to provide the API described here based on the **JDK8 Streams** framework for use with **JDK8** and later to provide a simple upgrade path.

Parallel Collections

Dealing with data frequently involves manipulating collections. Lists, arrays, sets, maps, iterators, strings. A lot of other data types can be viewed as collections of items. The common pattern to process such collections is to take elements sequentially, one-by-one, and make an action for each of the items in the series.

Take, for example, the *min* function, which is supposed to return the smallest element of a collection. When you call the *min* method on a collection of numbers, a variable (*minVal* say) is created to store the smallest value seen so far, initialized to some reasonable value for the given type, so for example for integers and floating points, this may well be zero. The elements of the collection are then iterated through as each is compared to the stored value. Should a value be less than the one currently held in *minVal* then *minVal* is changed to store the newly seen smaller value.

Once all elements have been processed, the minimum value in the collection is stored in the *minVal*.

However simple, this solution is **totally wrong** on multi-core and multi-processor hardware. Running the *min* function on a dual-core chip can leverage **at most 50%** of the computing power of the chip. On a quad-core it would be only 25%. So in this latter case, this algorithm effectively wastes 75% of the computing power of the chip.

Tree-like structures prove to be more appropriate for parallel processing.

The *min* function in our example doesn't need to iterate through all the elements in row and compare their values with the *minVal* variable. What it can do, instead, is rely on the multi-core/multi-processor nature of our hardware.

A *parallel_min* function can, for example, compare pairs (or tuples of certain size) of neighboring values in the collection and promote the smallest value from the tuple into a next round of comparisons. Searching for `the minimum` in different tuples can safely happen in parallel, so tuples in the same round can be processed by different cores at the same time without races or contention among threads.

Meet Parallel Arrays

Although not part of **JDK7**, the **extra166y** library brings a very convenient abstraction called Parallel Arrays, and **GPars** has harnessed this mechanism to provide a very Groovy API.

What is `extra166y` ?

extra166y Is an implementation of Java collections supporting parallel operations using Fork-Join concurrent framework provided by JSR-166. It was never made part of the JDK — unlike the **jsr166y** library. In fact, **extra166y** was made redundant from **JDK8** onwards by the **Streams** framework. Therefore, to continue to support **JDK7, *GPars** includes a copy of **extra166y** in it so there is no external dependency.

As noted earlier, work is underway to rewrite the **GPars** API in terms of **Streams** for users of **JDK8** onwards. Of course people using **JDK8** onwards can simply use **Streams** directly from **Groovy**.

How ?

GPars leverages the `Parallel Arrays` implementation in several ways. The ***GPars*Pool** and **GParsExecutorsPool** classes provide parallel variants of the common **Groovy** iteration methods like *each* , *collect* , *findAll*, etc.

A Parallel Sample

```
 def selfPortraits = images.findAllParallel{it.contains me}.collectParallel{it.resize()}
```

It also allows for a more functional style **map/reduce** style of collections processing.

A Map/Reduce Sample

```
 def smallestSelfPortrait = images.parallel.filter{it.contains me}.map{it.resize()}.min
{it.sizeInMB}
```

GParsPool

Use of **GParsPool** — the **JSR-166y**-based concurrent collection processor

Usage of GParsPool

The **GParsPool** class enables (from **JSR-166y**), a **ParallelArray**-based concurrency DSL for collections and objects.

Examples of use:

Some Parallel Samples

```
// Summarize numbers concurrently.
GParsPool.withPool {
    final AtomicInteger result = new AtomicInteger(0)
    [1, 2, 3, 4, 5].eachParallel{result.addAndGet(it)}

    assert 15 == result
}

// Multiply numbers asynchronously.
GParsPool.withPool {
    final List result = [1, 2, 3, 4, 5].collectParallel{it * 2}

    assert ([2, 4, 6, 8, 10].equals(result))
}
```

The passed-in closure takes an instance of a **ForkJoinPool** as a parameter, which can then be freely used inside the closure.

A **ForkJoinPool** *Sample*

```
// Check whether all elements within a collection meet certain criteria.
GParsPool.withPool(5){ForkJoinPool pool ->
    assert [1, 2, 3, 4, 5].everyParallel{it > 0}

    assert ![1, 2, 3, 4, 5].everyParallel{it > 1}
}
```

The *GParsPool.withPool* method takes optional parameters for number of threads in the created pool plus an `unhandled exceptions` handler.

An Exception Handler Sample With Threads Required

```
withPool(10){...}
withPool(20, exceptionHandler){...}
```

Pool Reuse

The *GParsPool.withExistingPool* takes an already existing **ForkJoinPool** instance to reuse. The DSL is valid only within the associated block of code and only for the thread that has called the *withPool* or *withExistingPool* methods. The *withPool* method returns only after all the worker threads have finished their tasks and the pool has been destroyed, returning the resulting value of the associated block of code. The *withExistingPool* method doesn't wait for the pool threads to finish.

Alternatively, the **GParsPool** class can be statically imported as *import static groovyx.gpars.GParsPool*, which will allow omitting the **GParsPool** class name.

A Pool Sample

```
withPool {
    assert [1, 2, 3, 4, 5].everyParallel{it > 0}
    assert ![1, 2, 3, 4, 5].everyParallel{it > 1}
}
```

The following methods are currently supported on all objects in **Groovy**:

- *eachParallel*
- *eachWithIndexParallel*
- *collectParallel*
- *collectManyParallel*
- *findAllParallel*
- *findAnyParallel*
- *findParallel*
- *everyParallel*
- *anyParallel*
- *grepParallel*
- *groupByParallel*
- *foldParallel*
- *minParallel*
- *maxParallel*
- *sumParallel*
- *splitParallel*
- *countParallel*
- *foldParallel*

Meta-class Enhancer

As an alternative, you can use the **ParallelEnhancer** class to enhance meta-classes of any classes or individual instances with the parallel methods.

An Enhanced Sample

```
import groovyx.gpars.ParallelEnhancer

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
ParallelEnhancer.enhanceInstance(list)
println list.collectParallel {it * 2 }

def animals = ['dog', 'ant', 'cat', 'whale']
ParallelEnhancer.enhanceInstance animals
println (animals.anyParallel {it ==~ /ant/} ? 'Found an ant' : 'No ants found')
println (animals.everyParallel {it.contains('a')} ? 'All animals contain a' : 'Some
animals can live without an a')
```

When using the **ParallelEnhancer** class, you're not restricted to a *withPool* block when using the **GParsPool** DSLs. The enhanced classed or instances remain enhanced till they are garbage collected.

Exception Handling

If an exception is thrown while processing any of the passed-in closures, the first exception is re-thrown from the xxxParallel methods and the algorithm stops as soon as possible.

Exception Handling

The exception handling mechanism of **GParsPool** builds on the one built into the **Fork/Join** framework. Since **Fork/Join** algorithms are by nature hierarchical, once any part of the algorithm fails, there's usually little benefit continuing the computation, since some branches of the algorithm will never return a result.

Bear in mind that the **GParsPool** implementation doesn't give any guarantees about its behavior after a first unhandled exception occurs, beyond stopping the algorithm and re-throwing the first detected exception to the caller. This behavior, after all, is consistent with what the traditional sequential iteration methods do.

Transparently Parallel Collections

On top of adding new *xxxParallel* methods, **GPars** can also let you change the semantics of original iteration methods. For example, you may be passing a collection into a library method, which will process your collection in a sequential way, let's say, by using the *collect* method. Then by changing the semantics of the *collect* method on your collection, you can effectively parallelize the library sequential code.

A makeConcurrent() Sample

```
GParsPool.withPool {

    //The selectImportantNames() will process the name collections concurrently
    assert ['ALICE', 'JASON'] == selectImportantNames(['Joe', 'Alice', 'Dave', 'Jason']
.makeConcurrent())
}

/**
 * A function implemented using standard sequential collect() and findAll() methods.
 */
def selectImportantNames(names) {
    names.collect {it.toUpperCase()}.findAll{it.size() > 4}
}
```

The *makeSequential* method will reset the collection back to the original sequential semantics.

A Sequential Sample

```
import static groovyx.gpars.GParsPool.withPool

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

println 'Sequential: ' list.each { print it + ',' } println()

withPool {

    println 'Sequential: '
    list.each { print it + ',' }
    println()

    list.makeConcurrent()

    println 'Concurrent: '
    list.each { print it + ',' }
    println()

    list.makeSequential()

    println 'Sequential: '
    list.each { print it + ',' }
    println()
}

println 'Sequential: '
list.each { print it + ',' }
println()
```

The *asConcurrent()* convenience method allows us to specify code blocks, where the collection maintains concurrent semantics.

An asConcurrent() Sample

```
import static groovyx.gpars.GParsPool.withPool

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

println 'Sequential: '
list.each { print it + ',' }
println()

withPool {

    println 'Sequential: '
    list.each { print it + ',' }
    println()

    list.asConcurrent {
        println 'Concurrent: '
        list.each { print it + ',' }
        println()
    }

    println 'Sequential: '
    list.each { print it + ',' }
    println()
}

println 'Sequential: '
list.each { print it + ',' }
println()
```

Code Samples

Transparent parallelism, including the *makeConcurrent()* , *makeSequential()* and *asConcurrent()* methods, is also available in combination with our *ParallelEnhancer* .

A ParallelEnhancer Sample

```groovy
/**
 * A function implemented using standard sequential collect() and findAll() methods.
 */
def selectImportantNames(names) {
    names.collect {it.toUpperCase()}.findAll{it.size() > 4}
}

def names = ['Joe', 'Alice', 'Dave', 'Jason']
ParallelEnhancer.enhanceInstance(names)

//The selectImportantNames() will process the name collections concurrently
assert ['ALICE', 'JASON'] == selectImportantNames(names.makeConcurrent())
```

Another ParallelEnhancer Sample

```groovy
import groovyx.gpars.ParallelEnhancer

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

println 'Sequential: '
list.each { print it + ',' }
println()

ParallelEnhancer.enhanceInstance(list)

println 'Sequential: '
list.each { print it + ',' }
println()

list.asConcurrent {
    println 'Concurrent: '
    list.each { print it + ',' }
    println()

}
list.makeSequential()

println 'Sequential: '
list.each { print it + ',' }
println()
```

Avoid Side-Effects in Functions

We have to warn you. Since the closures that are provided to the parallel methods like *eachParallel* or *collectParallel()* may be run in parallel, you have to make sure that each of the closures is written in a thread-safe manner. The closures must hold no internal state, share data nor have side-effects beyond the boundaries of the single element that they've been invoked on. Violations of these rules will open the door for race conditions and deadlocks, the most severe enemies of a modern multi-core programmer.

Don't do this !

Concurrently Accessing a non-Thread-Safe Collection

```
def thumbnails = []
images.eachParallel {thumbnails << it.thumbnail}  //Concurrently accessing a not-thread-safe collection of thumbnails? Don't do this!
```

At least, you've been warned.

It May Not Execute The Way You Expect

Because **GParsPool** uses a **Fork/Join** pool (with work stealing), threads may not be applied to a waiting processing task even though they may appear idle.

With a work-stealing algorithm, worker threads that run out of things to do can steal tasks from other threads that are still busy.

If you use **GParsExecutorsPool** (which doesn't use **Fork/Join**), you'll get the thread allocation behavior that you would naively expect.

GParsExecutorsPool

Use of **GParsExecutorsPool** - the `Java Executors`-based concurrent collection processor -

Usage of GParsExecutorsPool

The **GParsPool** classes enable a `Java Executors`-based concurrency DSL for collections and objects.

The **GParsExecutorsPool** class can be used as a pure-JDK-based `collections parallel processor`. Unlike the **GParsPool** class, **GParsExecutorsPool** doesn't require **fork/join** thread pools but, instead, leverages the standard JDK executor services to parallelize closures to process a collection or an object iteratively.

It needs to be stated, however, that **GParsPool** typically performs much better than **GParsExecutorsPool** does.

GParsPool typically performs much better than **GParsExecutorsPool**

Examples of Use

A **GParsExecutorsPool** *Example*

```
//multiply numbers asynchronously
 GParsExecutorsPool.withPool {
     Collection<Future> result = [1, 2, 3, 4, 5].collectParallel{it * 10}

     assert new HashSet([10, 20, 30, 40, 50]) == new HashSet((Collection)result*.get())
 }

 //multiply numbers asynchronously using an asynchronous closure
 GParsExecutorsPool.withPool {
     def closure={it * 10}
     def asyncClosure=closure.async()

     Collection<Future> result = [1, 2, 3, 4, 5].collect(asyncClosure)

     assert new HashSet([10, 20, 30, 40, 50]) == new HashSet((Collection)result*.get())
 }
```

The passed-in closure takes an instance of an **ExecutorService** as a parameter, which can be then used freely inside the closure.

Another **GParsExecutorsPool** *Example*

```
//find an element meeting specified criteria
 GParsExecutorsPool.withPool(5) {ExecutorService service ->
     service.submit({performLongCalculation()} as Runnable)
 }
```

The *GParsExecutorsPool.withPool()* method takes an optional parameter declaring the number of threads in the created pool and a thread factory.

An Example Declaring Required Thread Count

```
withPool(10) {...}
withPool(20, threadFactory) {...}
```

The *GParsExecutorsPool.withExistingPool()* takes an already existing `executor service instance` to reuse. The DSL is only valid within the associated block of code and only for the thread that has called the *withPool()* or *withExistingPool()* method.

The *withPool()* method returns control only after all the worker threads have finished their tasks and the executor service has been destroyed, returning the resulting value of the associated block of code.

Did you know the *withExistingPool()* method doesn't wait for `executor service threads` to finish ?

Statically import the **GParsExecutorsPool** class as *import static groovyx.gpars.GParsExecutorsPool.`` to omit the *GParsExecutorsPool* class name.

A FindParallel Example

```
withPool {
     def result = [1, 2, 3, 4, 5].findParallel{Number number -> number > 2}
     assert result in [3, 4, 5]
 }
```

The following methods are currently supported on all objects that support iterations in **Groovy** :

- eachParallel()
- eachWithIndexParallel()

- collectParallel()
- findAllParallel()
- findParallel()
- allParallel()
- anyParallel()
- grepParallel()
- groupByParallel()

Meta-class Enhancer

As an alternative, you can use the *GParsExecutorsPoolEnhancer* class to enhance meta-classes for any classes or individual instances having asynchronous methods.

Enhancing Your Code

```
import groovyx.gpars.GParsExecutorsPoolEnhancer

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
GParsExecutorsPoolEnhancer.enhanceInstance(list)
println list.collectParallel {it * 2 }

def animals = ['dog', 'ant', 'cat', 'whale']
GParsExecutorsPoolEnhancer.enhanceInstance animals

println (animals.anyParallel {it ==~ /ant/} ? 'Found an ant' : 'No ants found')
println (animals.allParallel {it.contains('a')} ? 'All animals contain a' : 'Some animals
can live without an a')
```

When using the *GParsExecutorsPoolEnhancer* class, you're not restricted to a *withPool()* block with the use of the `GParsExecutorsPool DSLs`. The enhanced classes or instances remain enhanced until they are garbage collected.

Exception Handling

Exceptions can be thrown while processing any of the passed-in closures. An instance of the *AsyncException* method will wrap any/all of the original exceptions re-thrown from the xxxParallel methods.

Avoid Side-effects in Functions

Once again we need to warn you about using closures with side-effects. Please avoid logic that affects objects beyond the scope of the single, currently processed element. Please avoid logic or closures that keep state. Don’t do that! It’s dangerous to pass them to any of the *xxxParallel()* methods.

Memoize

The *memoize* function enables caching of a function’s return values. Repeated calls to the memoized function with the same argument values will, instead of invoking the calculation encoded in the original function, retrieve the resulting value from an internal, transparent cache.

Provided the calculation is considerably slower than retrieving a cached value from the cache, developers can trade-off memory for performance.

Checkout out the example, where we attempt to scan multiple websites for particular content:

The **memoize** functionality of **GPars** was donated to **Groovy** for version 1.8 and if you run on **Groovy** 1.8 or later, we recommend you use the **Groovy** functionality.

Memoize, in **GPars**, is almost identical, except that it searches the memoized caches concurrently using the surrounding thread pool. This may give performance benefits in some scenarios.

Memoize Me Up, Scotty

The **GPars memoize** functionality has been renamed to avoid future conflicts with the **memoize** functionality in **Groovy**.

GPars now calls these methods with a preceding letter *g* , such as *gmemoize()*.

Examples Of Use

A **GParsPool** *Example With gmemoize()*

```
GParsPool.withPool {
    def urls = ['http://www.dzone.com', 'http://www.theserverside.com',
'http://www.infoq.com']

    Closure download = {url ->
        println "Downloading $url"
        url.toURL().text.toUpperCase()
    }

    Closure cachingDownload = download.gmemoize()

    println 'Groovy sites today: ' + urls.findAllParallel {url -> cachingDownload(url)
.contains('GROOVY')}
    println 'Grails sites today: ' + urls.findAllParallel {url -> cachingDownload(url)
.contains('GRAILS')}
    println 'Griffon sites today: ' + urls.findAllParallel {url -> cachingDownload(url)
.contains('GRIFFON')}
    println 'Gradle sites today: ' + urls.findAllParallel {url -> cachingDownload(url)
.contains('GRADLE')}
    println 'Concurrency sites today: ' + urls.findAllParallel {url -> cachingDownload
(url).contains('CONCURRENCY')}
    println 'GPars sites today: ' + urls.findAllParallel {url -> cachingDownload(url)
.contains('GPARS')}
}
```

Notice how closures are enhanced inside the *GParsPool.withPool()* blocks with a *memoize()* function. This returns a new closure wrapping the original closure as a cache entry.

In the previous example, we're calling the *cachingDownload* function in several places in the code, however, each unique url is downloaded only once - the first time it's needed. The values are then cached and available for subsequent calls. Additionally, these values are also available to all threads, no matter which thread originally came first with a download request for that particular url and had to handle the actual calculation/download.

So, to wrap up, a **memoize** call shields a function by using a cache of past return values.

However, *memoize* can do even more! In some algorithms, adding a little memory may have a dramatic impact on the computational complexity of the calculation. Let's look at a classical example of `Fibonacci` numbers.

Fibonacci Example

A purely functional, recursive implementation that follows the definition of Fibonacci numbers is exponentially complex:

A Fibonacci Example

```
Closure fib = {n -> n > 1 ? call(n - 1) + call(n - 2) : n}
```

Try calling the *fib* function with numbers around 30 and you'll see how slow it is.

Now with a little twist and an added **memoize** cache, the algorithm magically turns into a linearly complex one:

A Better Version of the Fibonacci Example

```
Closure fib
fib = {n -> n > 1 ? fib(n - 1) + fib(n - 2) : n}.gmemoize()
```

The extra memory we added has now cut off all but one recursive branch of the calculation. And all subsequent calls to the same *fib* function will also benefit from the cached values.

Look below to see how the *memoizeAtMost* variant can reduce memory consumption in our example, yet preserve the linear complexity of the algorithm.

Available Variants

Memoize

The basic variant keeps values in the internal cache for the whole lifetime of the memoized function. It provides the best performance characteristics of all the variants.

memoizeAtMost

Allows us to set a hard limit on number of items cached. Once the limit has been reached, all subsequently added values will eliminate the oldest value from the cache using the **LRU** (`Last Recently Used`) strategy.

So for our Fibonacci number example, we could safely reduce the cache size to two items:

A Cached Fibonacci Example

```
Closure fib
fib = {n -> n > 1 ? fib(n - 1) + fib(n - 2) : n}.memoizeAtMost(2)
```

Setting an upper limit on the cache size serves two purposes:

- Keeps the memory footprint of the cache within defined boundaries

- Preserves desired performance characteristics of the function. Too large a cache increases the time to retrieve a cached value, compared to the time it would have taken to calculate the result directly.

memoizeAtLeast

Allows unlimited growth of the internal cache until the JVM’s garbage collector decides to step in and evict a `SoftReferences` entry (used by our implementation) from the memory.

The single parameter to the *memoizeAtLeast()* method indicates the minimum number of cached items that should be protected from gc eviction. The cache will never shrink below the specified number of entries. The cache ensures it only protects the most recently used items from eviction using the LRU (`Last Recently Used`) strategy.

memoizeBetween

Combines the **memoizeAtLeast** and **memoizeAtMost** methods to allow the cache to grow and shrink in the range between the two parameter values depending on available memory and the gc activity.

The cache size will never exceed the upper size limit to preserve desired performance characteristics of the cache.

Map-Reduce

The Parallel Collection Map/Reduce DSL gives **GPars** a more functional flavor. In general, the Map/Reduce DSL may be used for the same purpose as the *xxxParallel()* family of methods and has very similar semantics. On the other hand, **Map/Reduce** can perform considerably faster, if you need to chain multiple methods together to process a single collection in multiple steps:

A **Map-Reduce** *Example*

```
    println 'Number of occurrences of the word GROOVY today: ' + urls.parallel
            .map {it.toURL().text.toUpperCase()}
            .filter {it.contains('GROOVY')}
            .map{it.split()}
            .map{it.findAll{word -> word.contains 'GROOVY'}.size()}
            .sum()
```

The *xxxParallel()* methods must follow the same contract as their non-parallel peers. So a *collectParallel()* method must return a legal collection of items, which you can treat as a **Groovy** collection.

Internally, the *parallel collect method* builds an efficient parallel structure, called a parallel array. It then performs the required operation concurrently. Before returning, it destroys the *Parallel Array* as it builds a collection of results to return to you. A potential call to, for example, *findAllParallel()* on the resulting collection would repeat the whole process of construction and destruction of a Parallel Array instance under the covers.

With **Map/Reduce**, you turn your collection into a Parallel Array and back again only a single time. The **Map/Reduce** family of methods do not return **Groovy** collections, but can freely pass along the internal Parallel Arrays directly.

Invoking the *parallel* property of a collection will build a Parallel Array for the collection and then return a thin wrapper around the Parallel Array instance. Then you can chain any of these methods together to get an answer :

- map()
- reduce()
- filter()
- size()
- sum()
- min()
- max()
- sort()

- groupBy()
- combine()

Returning a plain **Groovy** collection instance is always just a matter of retrieving the *collection* property.

A **Map-Reduce** *Example*

```
def myNumbers = (1..1000).parallel.filter{it % 2 == 0}.map{Math.sqrt it}.collection
```

Avoid Side-effects in Functions

Once again we need to warn you. To avoid nasty surprises, please, keep any closures you pass to the **Map/Reduce** functions, stateless and clean from side-effects.

❗ To avoid nasty surprises keep your closures stateless

Availability

This feature is only available when using in the **Fork/Join**-based **GParsPool** , not in the **GParsExecutorsPool** method.

Classical Example

A classical example, inspired by thevery, counts occurrences of words in a string:

A Telling Example

```
import static groovyx.gpars.GParsPool.withPool

def words = "This is just plain text to count words in"
print count(words)

def count(arg) {

  withPool {

    return arg.parallel
      .map{[it, 1]}
      .groupBy{it[0]}.getParallel()
      .map {it.value=it.value.size();it}
      .sort{-it.value}.collection
  }

}
```

The same example can be implemented with the more general *combine* operation:

A Combine Example

```
def words = "This is just plain text to count words in"
print count(words)

def count(arg) {

  withPool {
    return arg.parallel
      .map{[it, 1]}
      .combine(0) {sum, value -> sum + value}.getParallel()
      .sort{-it.value}.collection
  }

}
```

Combine

The *combine* operation expects an input list of tuples (two-element lists), often considered to be key-value pairs (such as [[key1, value1], [key2, value2], [key1, value3], [key3, value4] …]). These might have potentially repeating keys.

When invoked, the *combine* method merges the values of identical keys using the provided accumulator function. This produces a map of the original (unique) keys and their (now) accumulated values.

E.g. will be combined into [a : b+e, c : d+f]. Some logic like the '+' operation for the values will need to be provided as the accumulation closure logic.

The *accumulation function* argument needs to specify a function to use when combining (accumulating) values belonging to the same key. An *initial accumulator value* needs to be provided as well.

Since the *combine* method processes items in parallel, the *initial accumulator value* will be reused multiple times. Thus the provided value must allow for reuse.

It should either be a **cloneable** (or **immutable**) value or a **closure** returning a fresh initial accumulator each time it's requested. Good combinations of accumulator functions and reusable initial values include:

Some Examples of a Combining-Accumulator Function and Reusable Initial Value

```
accumulator = {List acc, value -> acc << value} initialValue = []
accumulator = {List acc, value -> acc << value} initialValue = {-> []}
accumulator = {int sum, int value -> acc + value} initialValue = 0
accumulator = {int sum, int value -> sum + value} initialValue = {-> 0}
accumulator = {ShoppingCart cart, Item value -> cart.addItem(value)} initialValue = {->
new ShoppingCart()}
```

The return type is a map.

E.g. [['he', 1], ['she', 2], ['he', 2], ['me', 1], ['she', 5], ['he', 1]] with an initial value of zero will combine into ['he' : 4, 'she' : 7, 'me' : 1]

Comparion Logic

The keys will be mutually compared using their **equals** and **hashCode** methods. Consider using |*@Canonical* or |*@EqualsAndHashCode* annotations to annotate objects you use as keys.

As with all hash maps in **Groovy**, be sure you're using a **String** not a **GString** as a key!

For more involved scenarios when you *combine()* complex objects, a good strategy here is to have a complete class to use as a key for common use cases and to apply different keys for uncommon cases.

```
import groovy.transform.ToString
import groovy.transform.TupleConstructor

import static groovyx.gpars.GParsPool.withPool

// declare a complete class to use in combination processing
@TupleConstructor @ToString
class PricedCar implements Cloneable {  // either Clonable or Immutable
    String model
    String color
    Double price

    // declare a way to resolve comparison logic
    boolean equals(final o) {
        if (this.is(o)) return true
        if (getClass() != o.class) return false

        final PricedCar pricedCar = (PricedCar) o

        if (color != pricedCar.color) return false
        if (model != pricedCar.model) return false

        return true
    }

    int hashCode() {
        int result
        result = (model != null ? model.hashCode() : 0)
        result = 31 * result + (color != null ? color.hashCode() : 0)
        return result
    }

    @Override
    protected Object clone() {
        return super.clone()
    }
}

// some data
def cars = [new PricedCar('F550', 'blue', 2342.223),
        new PricedCar('F550', 'red', 234.234),
        new PricedCar('Da', 'white', 2222.2),
        new PricedCar('Da', 'white', 1111.1)]


withPool {
    //Combine by model
    def result =
        cars.parallel.map {
            [it.model, it]
        }.combine(new PricedCar('', 'N/A', 0.0)) {sum, value ->
            sum.model = value.model
            sum.price += value.price
            sum
        }.values()

    println result


    //Combine by model and color (using the PricedCar's equals and hashCode))
    result =
        cars.parallel.map {
            [it, it]
        }.combine(new PricedCar('', 'N/A', 0.0)) {sum, value ->
            sum.model = value.model
            sum.color = value.color
            sum.price += value.price
            sum
        }.values()

    println result
}
```

Parallel Arrays

As an alternative, the efficient tree-based data structures defined in **JSR-166y** - Java Concurrency can be used directly. The *parallelArray* property on any collection or object will return a *ParallelArray* instance holding the elements of the original collection. These then can be manipulated through the **jsr166y** API.

Please refer to **jsr166y** documentation for API details.

A Parallel Array Example

```
import groovyx.gpars.extra166y.Ops

groovyx.gpars.GParsPool.withPool {

    assert 15 == [1, 2, 3, 4, 5].parallelArray.reduce({a, b -> a + b} as Ops.Reducer, 0)
//summarize

    assert 55 == [1, 2, 3, 4, 5].parallelArray.withMapping({it ** 2} as Ops.Op).reduce({
a, b -> a + b} as Ops.Reducer, 0)       //summarize squares

    assert 20 == [1, 2, 3, 4, 5].parallelArray.withFilter({it % 2 == 0} as Ops.Predicate)
//summarize squares of even numbers
            .withMapping({it ** 2} as Ops.Op)
            .reduce({a, b -> a + b} as Ops.Reducer, 0)

    assert 'aa:bb:cc:dd:ee' == 'abcde'.parallelArray
//concatenate duplicated characters with separator
            .withMapping({it * 2} as Ops.Op)
            .reduce({a, b -> "$a:$b"} as Ops.Reducer, "")
```

Asynchronous Invocations

Long running background tasks happen a lot in most systems.

Typically, a main thread of execution wants to initialize a few calculations, start downloads, do searches, etc. even when the results may not be needed immediately.

GPars gives the developers the tools to schedule asynchronous activities for background processing and collect the results later, when they’re needed.

Usage of GParsPool and GParsExecutorsPool Asynchronous Processing Facilities

Both **GParsPool** and **GParsExecutorsPool** methods provide nearly identical services while leveraging different underlying machinery.

Closures Enhancements

The following methods are added to closures inside the *GPars(Executors)Pool.withPool()* blocks:

- async() - To create an asynchronous variant of the supplied closure which, when invoked, returns a **future** object for the potential return value
- callAsync() - Calls a closure in a separate thread while supplying the given arguments, returning a **future** object for the potential return value,

An **async()** *Example*

```
GParsPool.withPool() {
    Closure longLastingCalculation = {calculate()}
    Closure fastCalculation = longLastingCalculation.async()  //create a new closure,
which starts the original closure on a thread pool

    Future result=fastCalculation()                           //returns almost
immediately

    //do stuff while calculation performs ...
    println result.get()
}
```

A **callAsync()** *Example*

```
GParsPool.withPool() {
    /**
     * The callAsync() method is an asynchronous variant of the default call() method to
invoke a closure.
     * It will return a Future for the result value.
     */
    assert 6 == {it * 2}.call(3)
    assert 6 == {it * 2}.callAsync(3).get()
}
```

Timeouts

The *callTimeoutAsync()* methods, taking either a long value or a **Duration** instance, provides a timer mechanism.

A Timed Example

```
{->
    while(true) {
        Thread.sleep 1000  //Simulate a bit of interesting calculation
        if (Thread.currentThread().isInterrupted()) break;  //We've been cancelled
    }
}.callTimeoutAsync(2000)
```

To allow cancellation, our asynchronously running code must keep checking the *interrupted* flag of it's own thread and stop calculating when/if the flag is set to true.

Executor Service Enhancements

The **ExecutorService** and **ForkJoinPool** classes are enhanced with the '<<' (leftShift) operator to submit tasks to the pool and return a *Future* for the result.

A Convenient Example Using [red]'<<'

```
GParsExecutorsPool.withPool {ExecutorService executorService ->
    executorService << {println 'Inside parallel task'}
}
```

Running Functions (closures) in Parallel

The **GParsPool** and **GParsExecutorsPool** classes also provide handy methods *executeAsync()* and *executeAsyncAndWait()* to easily run multiple closures asynchronously.

Example:

An Example

```
GParsPool.withPool {
    assert [10, 20] == GParsPool.executeAsyncAndWait({calculateA()}, {calculateB()}
//waits for results
    assert [10, 20] == GParsPool.executeAsync({calculateA()}, {calculateB()})*.get()
//returns Futures instead and doesn't wait for results to be calculated
}
```

Composable Asynchronous Functions

Functions are to be composed. In fact, composing side-effect-free functions is very easy. Much easier and more reliable than composing objects, for example.

Given the same input, functions always return the same result, they never change their behavior unexpectedly nor they break when multiple threads call them at the same time.

Functions in Groovy

We can treat **Groovy** closures as functions. They take arguments, do their calculation and return a value. Provided you don't let your closures touch anything outside their scope, your closures are well-behaved, just like pure functions. Functions that you can combine for a higher good.

A Higher Good Example

```
def sum = (0..100000).inject(0, {a, b -> a + b})
```

For this example, by combining a function adding two numbers `{a,b}` with the *inject* function, which iterates through the whole collection, you can quickly summarize all items. Then, replacing the *adding* function with a *comparison* function immediately gives you a combined function to calculate maximums.

Find The Maximums

```
def max = myNumbers.inject(0, {a, b -> a>b?a:b})
```

You see, functional programming is popular for a reason.

Are We Concurrent Yet?

This all works just fine until you realize you're not using the full power of your expensive hardware. These functions are just plain sequential! No parallelism is used! All but one processor core is doing nothing, they're idle, totally wasted!

A Generic Way to Use Asynchronous Functions

Those paying attention might decide to use the *Parallel Collection* techniques described earlier and they would certainly be correct.

For our scenario described here, where we process a collection, using those *parallel* methods would be the best choice. However, we're now looking for a generic way to create and combine asynchronous functions. This would help us, not only for collections processing, but mostly in other, more generic, cases like the one right below.

All but one processor core is doing nothing! They're idle! Totally wasted!

To make things more obvious, here's an example of combining four functions, which are supposed to check whether a particular web page matches the contents of a local file. We need to download the page, load the file, calculate hashes of both and finally compare the resulting numbers.

An Example

```
Closure download = {String url ->
    url.toURL().text
}

Closure loadFile = {String fileName ->
    ...  //load the file here
}

Closure hash = {s -> s.hashCode()}

Closure compare = {int first, int second ->
    first == second
}

def result = compare(hash(download('http://www.gpars.org')), hash(loadFile(
'/coolStuff/gpars/website/index.html')))
println "The result of comparison: " + result
```

We need to download the page, load up the file, calculate hashes of both and finally compare the resulting numbers. Each of the functions is responsible for one particular job. One function downloads the content, a second loads the file, and a third calculates the hashes and finally the fourth one will do the comparison.

Combining the functions is as simple as nesting their calls.

Making It All Asynchronous

The downside of our code is that we haven't leveraged the independence of the *download()* and the *loadFile()* functions. Neither have we allowed the two hashes to be run concurrently. They could well run in parallel, but our approach to combine functions restricts parallelism.

Obviously not all of the functions **can** run concurrently. Some functions depend on results of others. They cannot start before the other function finishes. We need to block them until their parameters are available. The *hash()* functions needs a string to work on. The *compare()* function needs two numbers to compare.

So we can only take parallelism so far, while blocking parallelism of others. Seems like a challenging task.

Things Are Bright in the Functional World

Luckily, the dependencies between functions are already expressed implicitly in the code. There's no need to duplicate that dependency information. If one functions takes parameters and the parameters need to be calculated first by another function, we implicitly have a dependency here.

The *hash()* function depends on *loadFile()* as well as on the *download()* functions in our example. The *inject* function in our earlier example depended on the results of the *addition* functions gradually invoked on all elements of the collection.

> However difficult it may seem at first, our task is, in fact, very simple. We only need to teach our functions to return a *promise* of their future results. And we need to teach the other functions to accept those *promises* as parameters so that they will wait for the real values before they start their work.
>
> And if we convince the functions to release the threads they hold, while waiting for the values, we get directly to where the magic can happen.

In the best traditions of **GPars**, we've made it very straightforward for you to convince any function to believe in the **promises** of other functions. Call the *asyncFun()* function on a closure and you're asynchronous !

Promises, Promises

```
withPool {
    def maxPromise = numbers.inject(0, {a, b -> a>b?a:b}.asyncFun())

    println "Look Ma, I can talk to the user while the math is being done for me!"
    println maxPromise.get()
}
```

The *inject* function doesn't really care what objects are returned from the *addition* function, maybe it's a little surprised each call to the *addition* function returns so fast, but doesn't moan much, keeps iterating and finally returns the overall result we expect.

Now is the time you should stand behind what you say and do what you want others to do. Don't frown at the result and just accept that you got back just a **promise**. A **promise** to get the answer delivered as soon as the calculation is complete. The extra heat from your laptop is an indication that the calculation exploits natural parallelism in your functions and makes its best effort to deliver the result to you quickly.

A Promise Is A Promise

The *promise* is a good old *DataflowVariable*, so you can query its status, register some notification hooks or even make it an input to a **Dataflow** algorithm !

An Promising Example

```
withPool {
    def sumPromise = (0..100000).inject(0, {a, b -> a + b}.asyncFun())

    println "Are we done yet? " + sumPromise.bound

    sumPromise.whenBound {sum -> println sum}
}
```

Do You Need A Timeout ?

The *get()* method has also a variant with a timeout parameter, if you want to avoid the risk of waiting indefinitely.

Can Things Go Wrong?

Sure. But you'll get an exception thrown from the **promise** *get()* method.

An Exceptional Example

```
try {
    sumPromise.get()

} catch (MyCalculationException e) {
    println "Guess, things are not ideal today."
}
```

This Is All Fine, But What Functions Can Really Be Combined?

There are no limits to your ambitions. Take any sequential functions you need to combine and you should be able to combine their asynchronous variants as well.

Review our initial example comparing the content of a file with a web page. We simply make all the functions asynchronous by calling the *asyncFun()* method on them and we are ready to set off.

Using The asyncFun() Example

```
    Closure download = {String url ->
        url.toURL().text
    }.asyncFun()

    Closure loadFile = {String fileName ->
        ...  //load the file here
    }.asyncFun()

    Closure hash = {s -> s.hashCode()}.asyncFun()

    Closure compare = {int first, int second ->
        first == second
    }.asyncFun()

    def result = compare(hash(download('http://www.gpars.org')), hash(loadFile(
'/coolStuff/gpars/website/index.html')))

    println 'Allowed to do something else now'
    println "The result of comparison: " + result.get()
```

Calling Asynchronous Functions from Within Asynchronous Functions

Another very valuable attribute of asynchronous functions is that `promises` can be combined.

Promises can be combined !

An Asynchronous Function Within Another

```
import static groovyx.gpars.GParsPool.withPool

  withPool {
      Closure plus = {Integer a, Integer b ->
          sleep 3000
          println 'Adding numbers'
          a + b
      }.asyncFun(); // ok, here's one func

      Closure multiply = {Integer a, Integer b ->
          sleep 2000
          a * b
      }.asyncFun()  // and second one

      Closure measureTime = {->
          sleep 3000
          4
      }.asyncFun(); // and another

      // declare a function within a function
      Closure distance = {Integer initialDistance, Integer velocity, Integer time ->
          plus(initialDistance, multiply(velocity, time))
      }.asyncFun(); // and another


      Closure chattyDistance = {Integer initialDistance, Integer velocity, Integer time
->
          println 'All parameters are now ready - starting'
          println 'About to call another asynchronous function'
          def innerResultPromise = plus(initialDistance, multiply(velocity, time))
          println 'Returning the promise for the inner calculation as my own result'
          return innerResultPromise
      }.asyncFun(); // and declare (but not run) a final asynch.function

      // fine, now let's execute those previous asynch. functions
      println "Distance = " + distance(100, 20, measureTime()).get() + ' m'
      println "ChattyDistance = " + chattyDistance(100, 20, measureTime()).get() + ' m'
  }
```

If an asynchronous function (e.g. like the *distance* function in this example) in its body calls another asynchronous function (e.g. *plus*) and returns the the promise of the invoked function, the inner function’s (*plus*) resulting promise will combine with the outer function’s (*distance*) results promise.

The inner function (*plus*) will now bind its result to the outer function’s (*distance*) promise, once the inner function (plus) finishes its calculation. This ability of promises to combine logic allows functions to cease their calculation without blocking a thread. This happens not only when waiting for parameters, but also whenever they call another asynchronous function anywhere in their code body.

Methods as Asynchronous Functions

Methods can be referred to as closures using the .& operator. These closures can then be transformed using the *asyncFun* method into composable asynchronous functions just like ordinary closures.

An Example

```
class DownloadHelper {

    String download(String url) {
        url.toURL().text
    }

    int scanFor(String word, String text) {
        text.findAll(word).size()
    }

    String lower(s) {
        s.toLowerCase()
    }
}

//now we'll make the methods asynchronous
withPool {
    final DownloadHelper d = new DownloadHelper()
    Closure download = d.&download.asyncFun()   // notice the .& syntax
    Closure scanFor = d.&scanFor.asyncFun()     // and here
    Closure lower = d.&lower.asyncFun()         // and here

    //asynchronous processing
    def result = scanFor('groovy', lower(download('http://www.infoq.com')))
    println 'Doing something else for now'
    println result.get()
}
```

Using Annotations to Create Asynchronous Functions

Instead of calling the *asyncFun()* function, the *@AsyncFun* annotation can be used to annotate Closuretyped fields. The fields have to be initialized in-place and the containing class needs to be instantiated within a *withPool* block.

An Annotation Example

```
import static groovyx.gpars.GParsPool.withPool
import groovyx.gpars.AsyncFun

class DownloadingSearch {
    @AsyncFun Closure download = {String url ->
        url.toURL().text
    }

    @AsyncFun Closure scanFor = {String word, String text ->
        text.findAll(word).size()
    }

    @AsyncFun Closure lower = {s -> s.toLowerCase()}

    void scan() {
        def result = scanFor('groovy', lower(download('http://www.infoq.com')))
//synchronous processing

        println 'Allowed to do something else now'
        println result.get()
    }
}

withPool {
    new DownloadingSearch().scan()
}
```

Alternative Pools

The *AsyncFun* annotation, by default, uses an instance of **GParsPool** from the wrapping `withPool` block. You may, however, specify the type of pool explicitly:

A Explicit Example

```
@AsyncFun(GParsExecutorsPoolUtil) def sum6 = {a, b -> a + b }
```

Blocking Functions Through Annotations

The *AsyncFun* method also allows us to specify, whether the resulting function should allow blocking (true) or non-blocking (false - default) semantics.

An Example of Blocking Semantics

```
@AsyncFun(blocking = true)
def sum = {a, b -> a + b }
```

Explicit and Delayed Pool Assignment

When using the *GPars(Executors)PoolUtil.asyncFun()* function directly to create an asynchronous function, you have two additional ways to assign a thread pool to the function.

1. The thread pool to be used by the function can be specified explicitly as an additional argument at creation time
2. The implicit thread pool can be obtained from the surrounding scope at invocation-time rather at creation time

When specifying the thread pool explicitly, the call doesn’t need to be wrapped in a *withPool()* block:

To Specify Thread Pools Explicitly

```
Closure sPlus = {Integer a, Integer b ->
    a + b
}

Closure sMultiply = {Integer a, Integer b ->
    sleep 2000
    a * b
}

println "Synchronous result: " + sMultiply(sPlus(10, 30), 100)

final pool = new FJPool();

Closure aPlus = GParsPoolUtil.asyncFun(sPlus, pool)
Closure aMultiply = GParsPoolUtil.asyncFun(sMultiply, pool)

def result = aMultiply(aPlus(10, 30), 100)

println "Time to do something else while the calculation is running"
println "Asynchronous result: " + result.get()
```

With a delayed pool assignment, only the function invocation must be surrounded with a *withPool()* block:

A Delayed Pool Assignment Example

```
Closure aPlus = GParsPoolUtil.asyncFun(sPlus)
Closure aMultiply = GParsPoolUtil.asyncFun(sMultiply)

withPool {
    def result = aMultiply(aPlus(10, 30), 100)

    println "Time to do something else while the calculation is running"
    println "Asynchronous result: " + result.get()
}
```

For us, this is a very interesting domain to explore. So any comments, questions or suggestions are welcome on combining asynchronous functions or hints about its limits.

Fork-Join

Fork/Join or *Divide-and-Conquer*, is a very powerful abstraction to solve hierarchical problems.

The Abstraction

When talking about hierarchical problems, think about quick sort, merge sort, file system or general tree navigation problems.

- **Fork/Join** algorithms essentially split a problem into several smaller sub-problems and then recursively applies the same algorithm to each of the sub-problems.
- Once the sub-problem is small enough, it is solved directly.
- The solutions of all sub-problems are combined to solve their parent problem, which in turn helps solve its' own grand-parent problem.

A Picture Is Worth A Thousand Words

Check out the fancy Interactive **Fork/Join** visualization demo. It shows you how threads co-operate to solve a common divide-and-conquer algorithm.

The mighty **JSR-166y** library co-ordinates **Fork/Join** orchestration rather nicely, but leaves a few rough edges, which can hurt you, if you don’t pay enough attention. You must still deal with threads, pools and/or synchronization barriers.

The GPars Abstraction Convenience Layer

GPars can hide the complexities of dealing with threads, pools and recursive tasks from you, yet let you leverage the powerful **Fork/Join** implementation in **jsr166y**.

A Complex Example to Walk A File Directory

```
import static groovyx.gpars.GParsPool.runForkJoin
import static groovyx.gpars.GParsPool.withPool

withPool() {
    println """Number of files: ${

        runForkJoin(new File("./src")) {file ->
            long count = 0
            file.eachFile {
                if (it.isDirectory()) {
                    println "Forking a child task for $it"
                    forkOffChild(it)           //fork a child task

                } else {
                    count++
                }
            }
            return count + (childrenResults.sum(0))
            //use results of children tasks to calculate and store own result
        }

    }""".toString();
}
```

The *runForkJoin()* factory method uses the supplied recursive code together with the provided values to build a hierarchical **Fork/Join** calculation. The number of values passed to the *runForkJoin()* method must match the number of expected parameters of the closure. This must equal the same number of arguments passed to the *forkOffChild()* or *runChildDirectly()* methods.

```
def quicksort(numbers) {

    withPool {

        runForkJoin(0, numbers) {index, list ->

            def groups = list.groupBy {it <=> list[list.size().intdiv(2)]}

            if ((list.size() < 2) || (groups.size() == 1)) {
                return [index: index, list: list.clone()]
            }

            (-1..1).each {forkOffChild(it, groups[it] ?: [])}

            return [index: index, list: childrenResults.sort {it.index}.sum {it.list}]

        }.list
    }
}
```

It's Asynchronous, Mate !

The important piece of the puzzle to note here is that *forkOffChild()* doesn't wait for the child to run. It merely schedules it for execution at a future time. If a child task throws an exception, don't expect the exception to be fired from the *forkOffChild()* method itself. The exception will have happened long after the parent has called *forkOffChild()*.

It's the *getChildrenResults()* method that will re-throw any child sub-task exceptions back to the parent.

Alternative Approach

Alternatively, the underlying mechanism of nested **Fork/Join** worker tasks can be used directly. Custom-tailored workers can eliminate the performance overhead associated with parameter spreading imposed when using the generic workers.

Also, custom workers can be implemented in **Java** for further increases in performance.

A Custom Worker

```
public final class FileCounter extends AbstractForkJoinWorker<Long> {
    private final File file;

    def FileCounter(final File file) {
        this.file = file
    }

    @Override
    protected Long computeTask() {
        long count = 0;

        file.eachFile {
            if (it.isDirectory()) {
                println "Forking a thread for $it"
                forkOffChild(new FileCounter(it))           //fork a child task

            } else {
                count++
            }
        }
        return count + ((childrenResults)?.sum() ?: 0)  //use results of children tasks
to calculate and store own result
    }
}

withPool(1) {pool ->  //feel free to experiment with the number of fork/join threads in
the pool
    println "Number of files: ${runForkJoin(new FileCounter(new File("..")))}"
}
```

The **AbstractForkJoinWorker** subclasses can be written in both **Java** and **Groovy**. Either choicr lets you optimize for execution speed, if low performance of the worker becomes a bottleneck.

Fork / Join Saves Your Resources

Fork/Join operations can safely br run with small numbers of threads thanks to internal use of the **TaskBarrier** class to synchronize the threads.

While a thread is blocked inside an algorithm waiting for its sub-problems to be calculated, the thread is silently returned to it's pool to take on any other available sub-problems from the task queue and process them. Although the algorithm creates as many tasks as there are sub-directories and tasks wait for the sub-directory tasks to complete, often as few as a single thread is enough to keep the computation going and eventually calculate a valid result.

Mergesort Example

Come on Punk, Merge my day !

```
import static groovyx.gpars.GParsPool.runForkJoin
import static groovyx.gpars.GParsPool.withPool

/**
 * Splits a list of numbers in half
 */
def split(List<Integer> list) {
    int listSize = list.size()
    int middleIndex = listSize / 2
    def list1 = list[0..<middleIndex]
    def list2 = list[middleIndex..listSize - 1]
    return [list1, list2]
}

/**
 * Merges two sorted lists into one
 */
List<Integer> merge(List<Integer> a, List<Integer> b) {
    int i = 0, j = 0
    final int newSize = a.size() + b.size()
    List<Integer> result = new ArrayList<Integer>(newSize)

    while ((i < a.size()) && (j < b.size())) {
        if (a[i] <= b[j]) result << a[i++]
        else result << b[j++]
    }

    if (i < a.size()) result.addAll(a[i..-1])
    else result.addAll(b[j..-1])
    return result
}

final def numbers = [1, 5, 2, 4, 3, 8, 6, 7, 3, 4, 5, 2, 2, 9, 8, 7, 6, 7, 8, 1, 4, 1, 7,
5, 8, 2, 3, 9, 5, 7, 4, 3]

withPool(3) {  //feel free to experiment with the number of fork/join threads in the pool
    println """Sorted numbers: ${
        runForkJoin(numbers) {nums ->
            println "Thread ${Thread.currentThread().name[-1]}: Sorting $nums"
            switch (nums.size()) {
                case 0..1:
                    return nums                                   //store own result
                case 2:
                    if (nums[0] <= nums[1]) return nums     //store own result
                    else return nums[-1..0]                       //store own result
                default:
                    def splitList = split(nums)
                    [splitList[0], splitList[1]].each {forkOffChild it}  //fork a child
task
                    return merge(* childrenResults)      //use results of children tasks
to calculate and store own result
            }
        }
    }"""
}
```

Mergesort Example Using A Custom-tailored Worker Class

An Example

```
public final class SortWorker extends AbstractForkJoinWorker<List<Integer>> {
    private final List numbers

    def SortWorker(final List<Integer> numbers) {
        this.numbers = numbers.asImmutable()
    }

    /**
     * Splits a list of numbers in half
     */
    def split(List<Integer> list) {
        int listSize = list.size()
        int middleIndex = listSize / 2
        def list1 = list[0..<middleIndex]
        def list2 = list[middleIndex..listSize - 1]
        return [list1, list2]
    }

    /**
     * Merges two sorted lists into one
     */
    List<Integer> merge(List<Integer> a, List<Integer> b) {
        int i = 0, j = 0
        final int newSize = a.size() + b.size()

        List<Integer> result = new ArrayList<Integer>(newSize)

        while ((i < a.size()) && (j < b.size())) {
            if (a[i] <= b[j]) result << a[i++]
            else result << b[j++]
        }

        if (i < a.size()) result.addAll(a[i..-1])
        else result.addAll(b[j..-1])
        return result
    }

    /**
     * Sorts a small list or delegates to two children, if the list contains more than
two elements.
     */
    @Override
    protected List<Integer> computeTask() {
        println "Thread ${Thread.currentThread().name[-1]}: Sorting $numbers"

        switch (numbers.size()) {
            case 0..1:
                return numbers                                  //store own result

            case 2:
                if (numbers[0] <= numbers[1]) return numbers     //store own result
                else return numbers[-1..0]                       //store own result

            default:
                def splitList = split(numbers)
                [new SortWorker(splitList[0]), new SortWorker(splitList[1])].each
{forkOffChild it}  //fork a child task
                return merge(* childrenResults)      //use results of children tasks to
calculate and store own result
        }
    }
}

final def numbers = [1, 5, 2, 4, 3, 8, 6, 7, 3, 4, 5, 2, 2, 9, 8, 7, 6, 7, 8, 1, 4, 1, 7,
5, 8, 2, 3, 9, 5, 7, 4, 3]

withPool(1) {  //feel free to experiment with the number of fork/join threads in the pool
    println "Sorted numbers: ${runForkJoin(new SortWorker(numbers))}"
}
```

Running Child Tasks Directly

The *forkOffChild* method has a sibling — called the *runChildDirectly* method. This method will run the child task directly and immediately within the current thread instead of scheduling the child task for asynchronous processing on the thread pool. Typically you'd call *forkOffChild* on every sub-task but the last, which you invoke directly without the scheduling overhead.

A Fork-In-Time-Saves-Nine

```
Closure fib = {number ->
    if (number <= 2) {
        return 1
    }

    forkOffChild(number - 1)                            //  This task will run
asynchronously, probably in a different thread
    final def result = runChildDirectly(number - 2)     //  This task is run directly
within the current thread
    return (Integer) getChildrenResults().sum() + result
}

withPool {
    assert 55 == runForkJoin(10, fib)
}
```

Availability

This feature is only available when using in the **Fork/Join**-based **GParsPool** , but not **GParsExecutorsPool** .

Parallel Speculations

With processor cores having become plentiful, some algorithms might benefit from brutal-force parallel duplication. Instead of deciding up-front about how to solve a problem, what algorithm to use or which location to connect to, you run all potential solutions in parallel.

Parallel Speculations

Imagine you need to perform a task like e.g. calculate an expensive function or read data from a file, database or internet. Luckily, you know several good ways (e.g. functions or urls) to reach your goal. However, all are not equal.

Although they return the same (as far as your needs are concerned) result, the elapsed time of each will differ and some may even fail (e.g. network issues). What's worse, no-one's going to tell you which choice gives you the single best solution nor which paths might lead to no solution at all.

1. Shall I run *quick sort* or *merge sort* on my list?
2. Which url will work best?
3. Is this service available at its primary location or should I use the backup one?

GPars Speculations give you the option to try all the available alternatives in parallel and receive the result from the fastest functional path, silently ignoring the slow or broken ones.

This is what the *speculate* methods on **GParsPool** and **GParsExecutorsPool** can do for you.

A Sort Example

```
def numbers = ...
def quickSort = ...
def mergeSort = ...
def sortedNumbers = speculate(quickSort, mergeSort)
```

So we're performing both a *quick sort* and a *merge sort* at the same time (concurrently), while getting the result of the faster one.

Given the parallel resources available these days on mainstream hardware, running the two functions in parallel will not have a dramatic impact on speed of calculation of either one, and thus we get the results of both in about the same time as if we ran only ran the faster of the two calculations. And also, the result arrives sooner than when running the slower one. Yet we didn't have to know up-front, which of the two sorting algorithms would perform better on our data. Thus we speculated (guessed).

Similarly, downloading a document from several sources with different speeds and/or reliability might look like this:

A Dpcument DownLoad Example

```
import static groovyx.gpars.GParsPool.speculate
import static groovyx.gpars.GParsPool.withPool

def alternative1 = {
    'http://www.dzone.com/links/index.html'.toURL().text
}

def alternative2 = {
    'http://www.dzone.com/'.toURL().text
}

def alternative3 = {
    'http://www.dzzzzzone.com/'.toURL().text  //wrong url
}

def alternative4 = {
    'http://dzone.com/'.toURL().text
}

withPool(4){
    println speculate([alternative1, alternative2, alternative3, alternative4]).contains
('groovy')
}
```

Thread Starvation

Make sure the surrounding thread pool has enough threads to process all alternatives in parallel. The size of the pool should match the number of closures supplied.

Alternatives Using Dataflow Variables and Streams

In some use cases, we can ignore failing alternatives, so **Dataflow** variables or **Streams** may be used to obtain the results of the winning speculation.

See the User Guide's topic on Dataflow Concurrency

Please refer to the **Dataflow Concurrency** section of the **User Guide** for details on **Dataflow Variables** and streams.

An Example

```
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.Dataflow.task

def alternative1 = {
    'http://www.dzone.com/links/index.html'.toURL().text
}

def alternative2 = {
    'http://www.dzone.com/'.toURL().text
}

def alternative3 = {
    'http://www.dzzzzzone.com/'.toURL().text  //will fail due to wrong url
}

def alternative4 = {
    'http://dzone.com/'.toURL().text
}

//Pick either one of the following, both will work:
final def result = new DataflowQueue()
//  final def result = new DataflowVariable()

[alternative1, alternative2, alternative3, alternative4].each{code ->
    task{
        try {
            result << code()
        }
        catch (ignore) { }  // We deliberately ignore unsuccessful urls.
    }
}

println result.val.contains('groovy')
```