# GParsPool

Russell Winder

Version 1.0, 2015-10-01

# Table of Contents

# GParsPool Concurrent collection processing

Dealing with data frequently involves manipulating collections. Lists, arrays, sets, maps, iterators, strings and 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 row.

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, the caller thread will create an *accumulator* or *so-far-the-smallest-value* initialized to the minimum value of the given type, let say to zero. And then the thread will iterate through the elements of the collection and compare them with the value in the *accumulator* . Once all elements have been processed, the minimum value is stored in the *accumulator* .

This algorithm, however simple, is **totally wrong** on multi-core 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%.

this algorithm effectively **wastes 75% of the computing power** !

Correct! this algorithm effectively **wastes 75% of the computing power** of the chip.

Tree-like structures proved 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 *accumulator* . What it can do instead is relying on the multi-core nature of your hardware.

A *parallel_min()* function could, 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 comparison.

Searching for minimum in different tuples can safely happen in parallel and so tuples in the same round can be processed by different cores at the same time without races or contention among threads.

The *GParsPool* class enables a **ParallelArray**-based (from JSR-166y) DSL on collections.

Examples of use:

*GParsPoolwithPool Sample*

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

    //a map-reduce functional style
    def smallestSelfPortrait = images.parallel
        .filter{it.contains me}
        .map{it.resize()}
        .min{it.sizeInMB}
}
```

The *GParsExecutorsPool* class provides similar functionality to the *GParsPool* class, however uses JDK thread pools instead of the more efficient **ParallelArray**-based (from JSR-166y) implementation.

Examples of use:

*GParsExecutorsPool.withPool*

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

See the Parallel Collection section in the User Guide for more information.