Fork/Join Pool

Table of Contents

Concepts

ForkJoinPool

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%. 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.

Usage

Parallel collection processing

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

- eachParallel()
- eachWithIndexParallel()
- collectParallel()
- findAllParallel()
- findParallel()
- everyParallel()
- anyParallel()
- grepParallel()
- groupByParallel()
- foldParallel()
- minParallel()
- maxParallel()
- sumParallel()

```
// Summarize numbers concurrently.
ForkJoinPool.withPool{
    final AtomicInteger result = new AtomicInteger(0)
    [1, 2, 3, 4, 5].eachParallel{result.addAndGet(it)}
    assert 15 == result
}

// Multiply numbers asynchronously.
ForkJoinPool.withPool{
    final List result = [1, 2, 3, 4, 5].collectParallel{it * 2}
    assert ([2, 4, 6, 8, 10].equals(result))
}
```

Meta-class enhancer

```
import groovyx.gpars.ParallelEnhancer

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

Transparently parallel collections

```
ForkJoinPool.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}
}
```

Map/Reduce

Available methods:

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

The *collection* property will return all elements wrapped in a Groovy collection instance.

```
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()
```