Concepts Compared

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

People love variety but frequently fail to make appropriate choices. Since **GPars** brings several concurrency paradigms to the table, we decided to build a simplified guide helping users pick the right concept for their task.

A Short Guide

Data decomposition

- \--→ Geometric decomposition (Parallel collections)
- \--→ Recursive (Fork/Join)

Tasks decomposition

- \--→ Independent tasks (Parallel collections, dataflow tasks)
- \--→ Recursively dependent tasks (Fork/Join)
- \--→ Tasks with mutual dependencies (Composable asynchronous functions, Dataflow tasks, CSP)
- \--→ Tasks cooperating on the same data (Stm, agents)

Streamed Data Decomposition

- \--→ Pipeline (Dataflow channels and operators)
- \--→ Event-based (Actors, Active objects)

A Detailed Guide

Processing A Collection in Parallel

Whenever you come across a collection that takes a while to process, consider using parallel collection methods. Although enabling collections for parallel processing imposes overhead, it frequently outweights the ineffectiveness of sequential processing. **GPars** gives you two options here:

- *GParsPool*, which leverages the efficient Fork/Join algorithm using a Fork/Join thread pool
- *GParsExecutorsPool*, which builds on plain old Java 5 executors

Processing A Collection in Parallel and Chaining Methods

The parallel collection methods, such as *eachParallel*, *findAllParallel*, etc., discussed above provide an easy migration path from sequential to concurrent code. However, when chaining multiple collection-processing methods it is more effective to use the *map/reduce* principle instead. Use *GParsPool*-based *map/reduce* operations to avoid the overhead of creating and destroying parallel collection for each parallel method in the chain. The conversion will be done only once - during the call to retrieve the *parallel* property of a collection. Since then the parallel tree-like data structure will be reused by all subsequent calls. The *map/reduce* approach should be preferred for chained parallel method calls.

Processing A Hierarchical Data Structure - `divide-n-conquer` Recursive Algorithm

Trees or hierarchies are naturally parallel data structures. *Fork/Join* algorithms process hierarchical data or problems concurrently. Use *GParsPool* and its *Fork/Join* convenience layer was designed to created *Fork/Join* calculations easily.

Creating Asynchronous Functions for Background Execution

Long-lasting calculations can be run in the background with very little syntactic and performance overhead. Use asynchronous functions within either *GParsPool* or *GParsExecutorsPool*

- *callAsync()* to invoke a closure asynchronously
- *asyncFun()* to create an asynchronous closure out of the original one.
- Asynchronous closures can then be combined just like the original sequential ones

Building Concurrent Systems for Large Data Mining

Use dataflow channels and operators to build networks of independent, asynchronous, event-driven calculations that process data. Dataflow networks are typically used for data or image processing, data mining or computer simulations.

Protecting Shared Pieces of Data from Concurrent Access

In some scenarios shared mutable state models the problem domain better than other paradigms. Imagine, for example, a shopping cart accessed concurrently by multiple user requests. To protect such a shared resource from concurrent modifications use *Agents*. They will wrap the data and guard access to it.

Protecting Multiple Shared Data Elements from Concurrent Access to Preserve Mutual Consistency

When the number of shared mutable resources grows and multiple of them need to be touched without intervention from other threads, use *Software Transactional Memory* to demarcate ACI(D) transactions for concurrent blocks of code that access the data. STM will give you the illusion that threads can access data without care about the other threads. The STM engine will resolve all conflicts and automatically re-try aborted transactions.

Share One-shot Values Among Threads/tasks, e.g. To Test Asynchronously Calculated Results

Use *DataflowVariables* with the thread-safe single-write multiple-read semantics. They ensure the reader cannot continue before a value is safely written to the variable by another thread. Also, they allow for callbacks to be registered and invoked as soon as a value gets bound.

Splitting Algorithms Explicitly Into Independent Asynchronous Objects with Direct Addressing

Use *Actors* in one of it’s many flavors. This is exactly their domain - independent active objects exchanging messages asynchronously.

Wrapping Actors With A POJO Facade

Active Objects offer OO interface to actors. You get POJOs, whose methods are asynchronous and whose state is protected under the same guarantees as with actors.

Splitting Algorithms Explicitly Into Independent Concurrent Processes With Indirect Addressing

Use *Dataflow tasks/processes* communicating through *dataflow channels* or *Groovy CSP*. Unlike with *Actors*, you get deterministic behavior allowing for re-use and composability. Additionally, you may also combine asynchronous and synchronous communication channels to limit the number of unprocessed messages in the network.

The ability to address parties indirectly through channels loosens the coupling between components of the algorithm and makes tasks such as load-balancing or broadcasting easier to implement.