Aalto University
School of Science

CS-E4110
Concurrent Programming

Week 5 – Exercise session

2021-12-03

14:15 – 16:00

Jaakko Harjuhahto

Today

- Futures
- Parallel collections
- Dicussion: Reactor Task-A solutions
- General Q&A

Futures

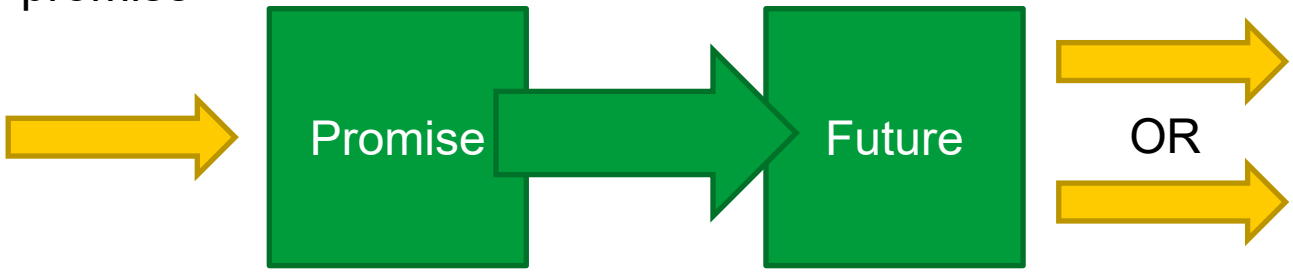
Futures and Promises

- Conceptually: work with data that will become available in the future
- Decouple the results of a computation from the mechanism of performing the computation
- Futures become truly useful when the computations take a long time, or there are multiple concurrent computations going on
  - Do other things while waiting for the results
- No need for explicit threading and synchronization!

Simplified View

Producer completes the promise

Consumer waits for results



Either by:
- Succeeding (writes result data to the promise)
- Failing (stores the reason e.g. Exception)

Consumer specifies what to do when the result becomes available

Execution

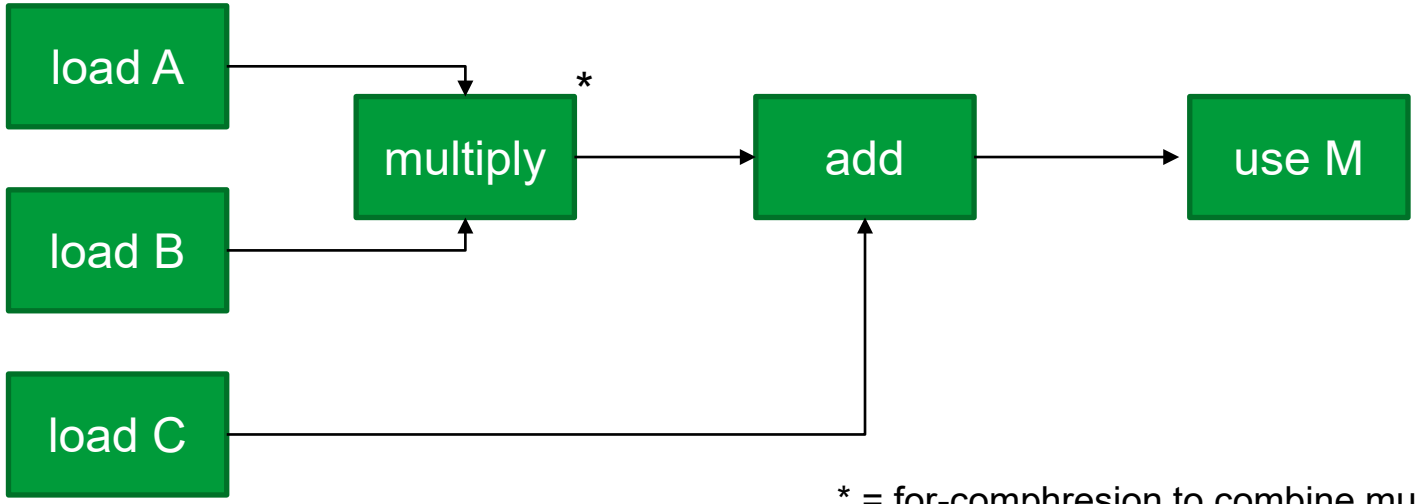
- How are the computations executed?
- In Scala, the default is an implicit `ExecutionContext`
  - The default `ExecutionContext` is a pool of threads
  - When creating futures, the computation code is given to a thread in the executor pool
  - Once the computation result is avaible (a success or failure), the executor completes the associated promise
  - The thread that ran the computation returns to the pool to wait for more work
- The execution context can also be defined explicitly when creating the future

Scala Implicits

- Scala mechanism for automatically selecting arguments to pass to functions and classes
  - ”If the code needs a variable or function for X, make this Y available as an option”
  - Here: futures need an execution context and will look for an implicitly defined one that meets the type requirements
- Common for Scala libraries to use implicits this way
  - Can be hard to debug, since it may not be immediately clear where the implicit definition came from, or which one was used

Chaining

- Create multi-phase processes by chaining `onSuccess` and `onFailure` callbacks
- Example: Matrix multiply-add for large matrices
  - Classic AxB+C



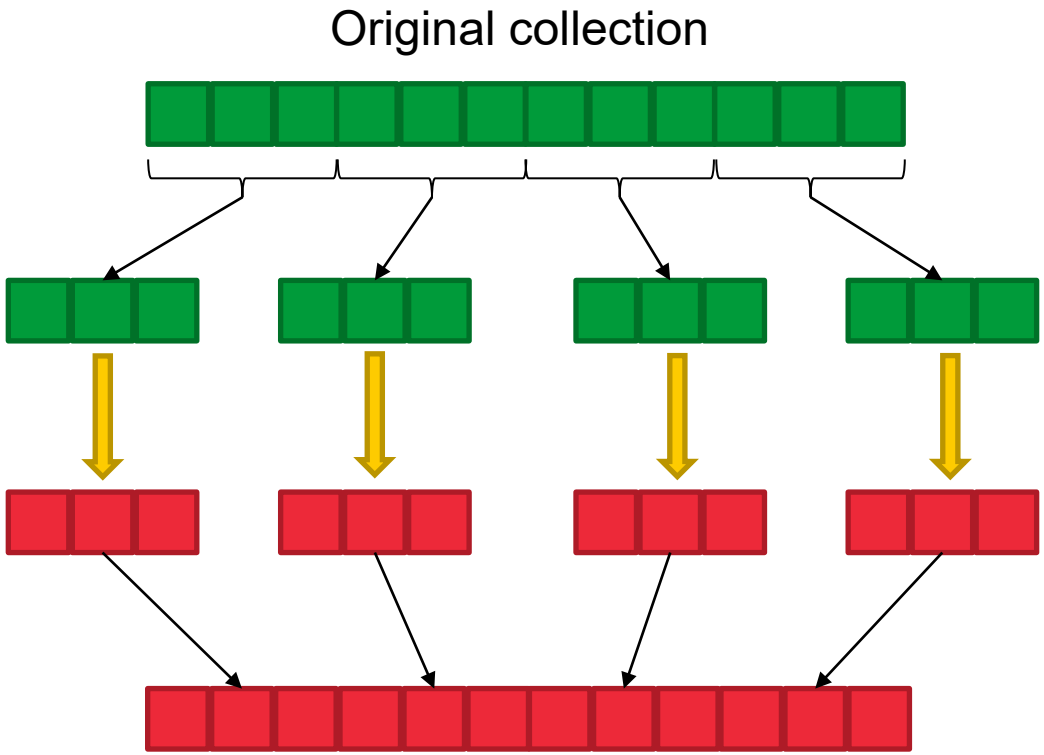
* = for-comphresion to combine multiple future

Parallel Collections

Scala Parallel Collections

- Perform an operation in parallel to a potentially large collection of data elements is a standard problem
  - Can be done with threads and manual synchronization, but...
  - Standard problems benefit from standard solutions
  - Scala offers a library for parallel operations on collections, without explicitly using threads and synchronization
- One more example of building more expressive abstractions on top of the elementary building blocks we have studied on this couse: threads and synchronization

Splitters and Combiners



This image represents a map operation. The same idea applies to other patterns, such as fold/reduce, for-each, etc

Beyond Scala

- The same concept of separating the dataset, operations to run on the dataset and the execution engine or environment is common
- For very large datasets the execution engine can be distributed over a cluster of machines
- Variations of similar ideas:
  - C# Parallel LINQ queries
  - Apache Spark data processing engine
  - Hadoop (and MapReduce) framework

Using the Parallel Collections

- The collections are not shipped as a part of the base Scala library and must be added to SBT dependencies

```
libraryDependencies += “org.scala-lang.modules” %% “scala-parallel-collections” % “1.0.4”
```

- Option-A: Import the adapters to extend the standard collections and convert an existing collection:

```
import scala.collection.parallel.CollectionConverters._
myCollectionInstance.par
```

- Option-B: Create a parallel collection directly:

```
import scala.collection.parallel.immutable_
val myCollectionInstance = new ParVector[type]
```

- Beware non-transitive and non-commutative operations
  - The order of operations is not necessarily the same as when iterating over the collection sequentiallly

Discussion: Reactor Task-A Solutions

Q&A