Scala Concurrency Notes

Parallelism and Concurrency

Background

- Modern CPUS are getting faster by increasing the number of cores instead of increasing clock speed
    - This is largely because power dissipation requirements of a chip increase dramatically with clock speed, known as the “power wall”
- Simultaneous computation by separate/independent physical processing elements is called “Parallel computation”
- Concurrent computation refers to time-multiplexed computation
    - Concurrent processing is generally focused on improving modularity/maintainability and perhaps response latency
    - Parallel is about using the available hardware to complete the work sooner
- Scala collections are not intrinsically threadsafe, so avoid concurrent writes to mutable versions (immutable ones are only ever read, so are safe!)
    - Never write to a collection that’s being concurrently traversed
    - Never read from a collection that’s being concurrently modified
    - Immutable collections, and Java concurrent collections may be used from multiple threads

Parallel computation forms

- Parallel computation forms include:
    - ***Bit parallel*** (4, 8, 16, 32, 64 bit words)

    - ***Instruction parallel*** (independent operations in a sequence run at the same time)
    - ***Task parallelism*** (separate “threads” making progress on related sub-tasks that contribute progress to the overall task)
    - ***Data parallel*** (the same computations performed in parallel on subsets of an overall dataset)

Data parallelism

- The “data parallel” approach depends on the ability to subdivide data, process the subsets independently, and then merge the results back.
    - Splitting and rejoining are the essence of the “map-reduce” concept.
    - Data parallel may be constructed in an ad-hoc fashion but is supported directly by Scala’s parallel collections
    - The operations must (generally) be associative and independent, since the order of operations is unpredictable, and data cannot easily be shared among processing threads
    - Preparing a parallel collection from a non-parallel one involves compute costs. For List, this is expensive, for arrays and trees the cost is much lower.
- Parallel trees:
    - Can be used in a purely functional fashion (they’re immutable) so threadsafety is good.
    - Typically exhibit poor “locality” which means they behave poorly with respect to the CPU cache.)
    - Involve lots of small objects to construct a large tree. This can behave poorly with respect to garbage collection
    - Combine efficiently

- Arrays:
    - Individual cells must be protected from concurrent modifications
    - Have excellent index-based access
    - Have good locality
    - Are expensive to concatenate
- Scala provides several structures for parallel computation
    - ParIterable, ParSet, ParMap, ParSeq
    - There are also “parent” traits that are agnostic regarding parallelism, these are GenIterable, GenSet, GenMap, GenSeq
    - Monadic operations (`flatMap`, `map`, `withFilter`, etc.) on parallel collections shard the data and run in parallel. Because of this, for expressions built on parallel collections also run parallel automatically
- Manual splitting / combining
    - Divide source data into groups, and provide “thread-private” destination structures (because these are thread-private, they can be mutable)
    - Recombine the partial results after processing all the individual sub-groups
    - The processing requirements for splitting and recombination are crucial to good performance
    - The `Splitter` trait (which extends `Iterator`) allows dividing an `Iterator` into sub-chunks
    - The `Combiner`, which extends `Builder`, is useful for merging things back together.

General concurrency problems

- Three basic problems must be addressed in concurrent or parallel computation
    - Visibility

    - "Transactional" correctness
    - Timing
- In addition, performance and responsiveness can be lost from other problems, such as:
    - Deadlock
    - Memory Bottleneck (trivial computations may be memory bound)
    - Context switch overhead
- Overview of JVM memory model
    - Platform independent rules defined using "Happens-Before" relationships
    - If hb(a,b) and b observes an effect of a, then b will see the effect of a
    - Must not make assumptions about implementation
    - Happens-before can be quite difficult to reason about in any but very simple cases--avoid temptation to be clever
    - ***CRITICALLY IMPORTANT Happens-before describes "visibility of effect", not actual order of occurrence.***
- 8 Language defined relationships:
    1. Lines of code ***in a single thread*** have happens-before relationships consistent with the meaning of sequence, iteration, and selection in the source language.
    2. Happens before relationships are transitive; that is if hb(a,b) and hb(b,c) then hb(a,c) -- *this rule can propagate relationships across threads and is therefore very important*
    3. Start a thread by one thread happens before the target thread begins
    4. End of a thread happens before another thread determines the thread ended

5. Write to a volatile variable happens before a subsequent read of that variable
6. Release of a monitor (exiting a synchronized region) happens before a subsequent acquisition of that monitor (entry to a region synchronized on the same object)
7. Interrupting a thread by one thread happens before the target thread notices it has been interrupted
8. End of a constructor happens before start of finalization

- In Scala the JVM's notion of volatile variables and monitor locks are provided through:
  - `@volatile` annotation
  - `AnyRef.synchronized` method
- In general, using the primitive mechanisms for interacting with the Java Memory Model is complex and error prone. It's almost always better to use higher level library and API features instead.

Actor Model

- A simple, well-established model for parallel and concurrent computation is known variously as the pipeline, producer-consumer, or actor model.
- Earlier versions of Scala provided an actor implementation as part of the core libraries, but these have been deprecated, removed from the API, and instead the Akka framework is recommended.
- An actor receives messages through a queue, which solves the three major data sharing problems of visibility, timing, and transactional correctness
  - When the message is taken from the queue the infrastructure guarantees that all the contents of the message are visible to the receiver

    - The message contents must not be mutated when any other actor might be looking at it. In general, all messages should be entirely immutable as this avoids the risk of error. Either way, it’s fairly easy to ensure that mutation never happens while another actor has access to an object, and with that, the transactional problems are avoided.
    - Messages sent by one source to one receiver are received in the order they were sent. However, messages from other sources may be interleaved in unpredictable ways
    - Messages are received after they are written, and no CPU time is spent polling for them. This provides a solution to the timing problem
- Akka’s actor implementation uses a callback type mechanism for processing messages in actors. This scales better than creating many threads

Configuring Akka

- To include Akka into an SBT project add these lines:

```
lazy val akkaVersion = "2.5.3"

libraryDependencies ++= Seq("com.typesafe.akka"
%% "akka-actor" % akkaVersion)
```

Elements of Akka’s API

- Ensure the necessary imports are available

```
import akka.actor.{Actor, ActorRef, ActorSystem,
Props}
```

- Configure an Actor infrastructure

```
val system = ActorSystem("MyActors")
```

- Define the actor’s class as a subclass of Actor, and give it a receive method that is a partial function that processes the messages that the actor expects

```
class MyActor extends Actor {
  override def receive = {
    case "Hello" => sender() ! "What!?"
    case x => println(s"received $x")
  }
}
```

- Use the actor system to create and start the actor, storing an actor reference to use for sending messages

```
val act = system.actorOf(Props[MyActor], "act")
```

- Then use the exclamation point operator to send a message to the actor through its reference

```
act ! (oven, "Hello")
```

- Note: this snippet doesn’t show the sender receiving the response that the actor above would return
- The Akka system provides a far more complex and complete API than shown here, but the core behavior is message passing
- In particular, Akka can run in a distributed fashion, passing messages between machines as well as simply passing them locally

The Future and Promise API

- A promise, or promise pipeline, is a monad-like concept that handles asynchrony allowing the programmer to specify a

series of transformation that are applied to data after they have become available
- The internal implementation of promise libraries generally use a callback mechanism, thereby avoiding the creation of many threads

Scala Promise API

- Future is a monadic structure that allows a computation to be performed in a separate thread (taken from a pool) and the result of that operation to be passed forward along the monad structure
- The execution pool is normally provided using implicits via the import:

```
scala.concurrent
 .ExecutionContext.Implicits.global
```

- Future factory with behavior argument
  - isCompleted and value attributes
- To handle callback type situations, use a Promise
  - The promise is used to create and control a future
  - Upon completion of the callback/asynchronous task, inject the result data into the pipeline using any one of `complete(Try)`, `success(value)`, and `failure(exception)`
- Monad-like behaviors for handling problems include `failed`, `fallBackTo`, `recover`, `recoverWith`