
Parallelism and Concurrency

Final Exam

Monday, August 10, 2020

Manage your time All points are not equal. We do not think that all exercises have the same difficulty, even if they have the same number of points.

Follow instructions The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, otherwise you will lose points.

Refer to the API The last pages of this exam are a small API. Please consult it before you reinvent the wheel. Feel free to detach it. You are free to use methods that are *not* part of this API provided they exist in the standard library.

Allowed materials This is a purely pen and paper exam. You are only allowed to bring your pen and other writing materials. You may **not** use any notes, books, calculators, or any electronic devices.

Extra paper sheets This exam comes with 4 pages of scrap paper (papier brouillon) that you can find at the end of the exam. Please do not ask for extra scrap paper or use your own.

Exam duration 2 hours and 40 minutes

Exercise	Points	Points Achieved
1	20	
2	20	
3	20	
4	20	
Total	80	

Exercise 1: Parallel computation (20 points)

In this exercise you will implement the logic of the `count` and `countIn` methods defined in a `Matrix` trait. The exercise will require you to use the `parallel` method to split computations into parallel parts.

```
trait Matrix[T] {

  /** Number of lines */
  def lines: Int

  /** Number of columns */
  def columns: Int

  /** Get the element at position 'n' ('0 <= n < lines') and 'm' ('0 <= m < columns')
   * Throws if indices are not in bounds.
   */
  def elem(n: Int, m: Int): T

  /** Counts the number of elements in the matrix that satisfy a predicate 'p'.
   * The predicate operation is assumed to be computationally expensive.
   * The predicate is evaluated in parallel on all elements of the matrix.
   */
  def count(p: T => Boolean): Int = {
    ... // TODO: Question 1.1
  }

  /** Counts the number of elements in a non-empty block of the matrix
   * which satisfy a predicate 'p'.
   * The predicate operation is assumed to be computationally expensive.
   * The predicate is evaluated in parallel on all elements of the matrix.
   * The block of the matrix is bounded by the rectangle starting at the
   * point ('startN', 'startM') and ending at ('endN-1', 'endM-1').
   */
  def countIn(p: T => Boolean, startN: Int, endN: Int, startM: Int, endM: Int): Int = {
    assert(startN < endN && startM < endM)
    ... // TODO: Question 1.2
  }
}
```

To parallelize this operation we can use a divide and conquer strategy. On an array, this is done by dividing the array into two sections, then compute the partial result of each in parallel and finally combine the results. This approach can be generalized on matrices by reducing on one axis at a time.

Read carefully the documentation of `count` and `countIn`.

Question 1.1 (5 points)

Implement `count` in terms of `countIn`.

```
def count(p: T => Boolean): Int =
```

Question 1.2 (15 points)

Implement `countIn` making sure that each `p` is evaluated in parallel using the `parallel` operation. Parallelizing all each `p` operation implies that the sequential threshold is of 1 element.

```
def countIn(p: T => Boolean, startN: Int, endN: Int, startM: Int, endM: Int): Int = {  
  assert(startN < endN && startM < endM)
```

```
}
```

Exercise 2: Memory model (20 points)

Each of the following 10 pieces of code runs code on some shared state in multiple threads, then prints a result. Your task is to determine whether the printed output is deterministic, that is: whether it always be the same regardless of how the threads are scheduled and assuming that the **Java Memory Model** is followed.

A correct answer gives 2 points, a wrong answer gives -1 point, no answer gives 0 points, if your total is negative you get 0 points for the exercise.

Question 2.1: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  var value: Int = 0
  def plus(x: Int): Unit = {
    value += x
  }
  def minus(x: Int): Unit = {
    value -= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.plus(1)
  myInt.minus(2)
}
val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}

t1.join()
t2.join()

println(myInt.value)
```

Question 2.2: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  @volatile var value: Int = 0
  def plus(x: Int): Unit = {
    value += x
  }
  def minus(x: Int): Unit = {
    value -= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.plus(1)
  myInt.minus(2)
}
t1.join()

val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}
t2.join()

println(myInt.value)
```

Question 2.3: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  @volatile var value: Int = 0
  def plus(x: Int): Unit = {
    value += x
  }
  def minus(x: Int): Unit = {
    value -= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.plus(1)
  myInt.minus(2)
}
val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}
t1.join()
t2.join()
println(myInt.value)
```

Question 2.4: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  @volatile var value: Int = 0
  def plus(x: Int): Unit = {
    value += x
  }
  def minus(x: Int): Unit = {
    value -= x
  }
  def times(x: Int): Unit = {
    value *= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.plus(1)
  myInt.minus(2)
  myInt.times(2)
}
val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}
t1.join()
t2.join()
println(myInt.value)
```

Question 2.5: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  var value: Int = 0
  def plus(x: Int): Unit = synchronized {
    value += x
  }
  def minus(x: Int): Unit = synchronized {
    value -= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.plus(1)
  myInt.minus(2)
}
val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}
t1.join()
t2.join()
println(myInt.value)
```

Question 2.6: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  var value: Int = 0
  def plus(x: Int): Unit = synchronized {
    value += x
  }
  def minus(x: Int): Unit = synchronized {
    value -= x
  }
  def times(x: Int): Unit = synchronized {
    value *= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.times(2)
  myInt.plus(1)
  myInt.minus(2)
}
val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}
t1.join()
t2.join()
println(myInt.value)
```

Question 2.7: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  val value: AtomicInteger =
    new AtomicInteger(0)

  def plus(x: Int): Unit = {
    val cur = value.get
    val retry =
      !value.compareAndSet(cur, cur + x)
    if (retry) this.plus(x)
  }
  def minus(x: Int): Unit = {
    val cur = value.get
    val retry =
      !value.compareAndSet(cur, cur - x)
    if (retry) this.minus(x)
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.plus(1)
  myInt.minus(2)
}
val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}
t1.join()
t2.join()
println(myInt.value.get)
```

Question 2.8: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  val value: AtomicInteger =
    new AtomicInteger(0)

  def plus(x: Int): Unit = {
    val cur = value.get
    val retry =
      !value.compareAndSet(cur, cur + x)
    if (retry) this.plus(x)
  }
  def minus(x: Int): Unit = {
    val cur = value.get
    val retry =
      !value.compareAndSet(cur, cur - x)
    if (retry) this.minus(x)
  }
  def times(x: Int): Unit = {
    val cur = value.get
    val retry =
      !value.compareAndSet(cur, cur * x)
    if (retry) this.times(x)
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.times(2)
  myInt.plus(1)
  myInt.minus(2)
}
val t2 = task {
  myInt.minus(4)
  myInt.plus(3)
}
t1.join()
t2.join()
println(myInt.value.get)
```

Question 2.9: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  var value: Int = 0
  def plus(x: Int): Unit = {
    value += x
  }
  def minus(x: Int): Unit = {
    value -= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.synchronized {
    myInt.plus(1)
    myInt.minus(2)
  }
}
val t2 = task {
  myInt.synchronized {
    myInt.minus(4)
    myInt.plus(3)
  }
}

t1.join()
t2.join()

println(myInt.value)
```

Question 2.10: In the code below, is the printed output deterministic?

Yes No

```
class MyInt {
  var value: Int = 0
  def plus(x: Int): Unit = {
    value += x
  }
  def minus(x: Int): Unit = {
    value -= x
  }
  def times(x: Int): Unit = {
    value *= x
  }
}
val myInt = new MyInt
val t1 = task {
  myInt.synchronized {
    myInt.times(2)
    myInt.plus(1)
    myInt.minus(2)
  }
}
val t2 = task {
  myInt.synchronized {
    myInt.minus(4)
    myInt.plus(3)
  }
}

t1.join()
t2.join()

println(myInt.value)
```

Exercise 3: Futures (20 points)

Question 3.1 (8 points)

Implement the `sequence` function that transforms a `List[Future[A]]` into a `Future[List[A]]`. If any of the input futures fail with exceptions, the returned future should fail with the exception of the first failed future in the list. When all the input futures f_1, f_2, \dots, f_n succeed with values v_1, v_2, \dots, v_n , the returned future should succeed with the list containing v_1, v_2, \dots, v_n .

Your implementation can use any method of `List` and `Future` (see Appendix) **except** `Future.sequence` and `Future.traverse`.

Hints: Decompose the input list `fs` using pattern matching. Use `for` comprehension to combine futures.

```
def sequence[A](fs: List[Future[A]]): Future[List[A]] =
```

Question 3.2 (4 points)

Using the `sequence` function defined in Question 1, Implement the `traverse` function that transforms a `List[A]` into a `Future[List[B]]` using the provided function `f`. This is useful for performing operations in parallel. For example, to apply a function to all items of a list in parallel: `traverse(myList)(x => Future(myFunc(x)))`. The result of `traverse` should contain results of applying `f` to the input list `xs`.

```
def traverse[A, B](xs: List[A])(f: A => Future[B]): Future[List[B]] =
```

Question 3.3 (8 points)

Given the following lemma about `traverse`:

Lemma: For all types A, B, C and values xs of type $List[A]$, f of type $A \Rightarrow Future[B]$, and g of type $B \Rightarrow Future[C]$, the following expressions are equivalent:

1. `traverse(xs)(f).flatMap(ys => traverse(ys)(g))`
2. `traverse(xs)(a => f(a).flatMap(g))`

Prove the following theorem about `sequence`:

Theorem: For all types A and values ffs of type $List[Future[Future[A]]]$, the following expressions are equivalent:

1. `sequence(ffs).flatMap(fs => sequence(fs))`
2. `sequence(ffs.map(ff => ff.flatten))`

In your proof you can use the fact that `Future` and `List` are monads, that is, the following properties hold for $F = Future$ and $F = List$

- a. Monad associativity: for all types A, B, C and valued ma of type $F[A]$, f of type $A \Rightarrow F[B]$, and g of type $B \Rightarrow F[C]$:

$$ma.flatMap(f).flatMap(g) == ma.flatMap(x => f(x).flatMap(g))$$

- b. Monad flatten: for all types A and values ffa of type $F[F[A]]$:

$$ffa.flatten == ffa.flatMap(x => x)$$

- c. Functor composition: for all types A, B, C and values fa of type $F[A]$, f of type $A \Rightarrow B$, and g of type $B \Rightarrow C$:

$$fa.map(f).map(g) == fa.map(x => g(f(x)))$$

- d. Functor identity: for all types A and values fa of type $F[A]$:

$$fa.map(x => x) == fa$$

Exercise 4: Actors (20 points)

In this exercise, you will design an actor system that implements a simplified version of *Tor*. *Tor* is an Internet protocol designed to anonymize the data relayed across it. For the purpose of this exercise, we will assume that *Tor* a network consists of several `TorNode` actors which are all capable of emitting new requests, relaying requests, and performing outgoing requests to the Internet.

To ensure anonymity, every request must be relayed a fixed number of times inside the *Tor* network before being sent to the Internet (determined by the `RELAY_COUNT` constructor argument to the `TorNode` class). We call *exit node* a node that performs an outgoing request. Responses from the Internet should be relayed back through the *Tor* network by going through the same relays in reverse, this time from the exit node all the way back to the initial request emitter.

For example, with `RELAY_COUNT = 3`, if node `N_A` sends a new request to the *Tor* network, here is a possible sequence of events to complete the request (here `N_x -> N_y` indicates that `N_x` sends a message to `N_y`):

```
N_A.emitRequest("http://perdu.com") // entry point
Request relayed through 3 nodes: N_A -> N_B -> N_C -> N_D
Node N_D calls performHttpRequest("http://perdu.com")
Response relayed back to N_A: N_D -> N_C -> N_B -> N_A
Node N_A calls displayResponse("<html><head><title>Vous Etes Perdu ?...")
```

Nodes should pick relays at random out of a set of actors called `adjacentRelays`, passed as a constructor argument to the `TorNode` class. You can assume that by construction randomly picking relays from `adjacentRelays` will never result in a loop, i.e. the nodes `N_A`, `N_B`, `N_C` and, `N_D` are distinct in the above example.

A node should maintain local state to allow responses to be routed through the same paths that their corresponding requests.

Your implementation should use the following `Request` and `Response` classes as messages exchanged between the actors of the network:

```
import java.util.UUID // UUID stands for Universally Unique Identifier

case class Request(url: String, remainingRelays: Int, id: UUID)
case class Response(payload: String, requestId: UUID)
```

Recall the Actor's own `ActorRef` is available as `self` and the current message's sender as `sender()`.

Fill in the implementation of the `TorNode` class given on the next page.

```

import akka.actor._
import scala.util.Random

class TorNode(RELAY_COUNT: Int, adjacentRelays: List[ActorRef]) extends Actor {
  // TODO: Add state for response routing

  def emitRequest(url: String): Unit = {
    // Randomly generate a new UUID for the request
    val id: UUID = UUID.randomUUID()

    // TODO: Create a new Request and send it to a relay picked at random

  }

  def displayResponse(payload: String): Unit = {
    println(payload)
  }

  // Synchronously performs an HTTP requests to the Internet and
  // returns the resulting payload.
  def performHttpRequest(url: String): String = { "..." }

  def receive = {
    // TODO: handle Request and Response messages

  }
}

```


Appendix

Static methods

- `def parallel[A, B](op1: => A, op2: => B): (A, B)`: Executes the two given computations in parallel and returns a pair with the results.
- `def task[A](f: => A): Task[A]`: Create a task that executes the computation passed as argument in a separate thread.

AnyRef

Relevant API for AnyRef:

- `def synchronized[A](op: => A): A`: Executes a computation within a synchronized block.

Task

Relevant API for Task[T]:

- `def join(): T`: Blocks until the task is completed and returns the computed value.

AtomicInteger

Contains an `Int` value that may be updated atomically.

Relevant API for `AtomicInteger`:

- `new AtomicInteger(initialValue: Int)`: Creates a new `AtomicInteger` with the given initial value.
- `def get: Int`: Gets the current value.
- `def compareAndSet(expectedValue: Int, updatedValue: Int): Boolean`: Atomically sets the value to the given `updatedValue` if the current value == `expectedValue`.

Future

Relevant API for `Future[T]`:

- `def onComplete(callback: Try[T] => Unit): Unit`: When this future is completed, either through an exception, or a value, apply the provided function.
- `def flatMap[S](f: T => Future[S]): Future[S]`: Creates a new future by applying a function to the successful result of this future, and returns the result of the function as the new future.
- `def filter(p: T => Boolean): Future[T]`: Creates a new future by filtering the value of the current future with a predicate.
- `def map[S](f: T => S): Future[S]`: Creates a new future by applying a function to the successful result of this future.
- `def foreach(f: T => Unit): Unit`: Asynchronously processes the value in the future once the value becomes available.

Relevant API for the `Future` object:

- `def apply[T](body: => T): Future[T]`: Starts an asynchronous computation and returns a `Future` instance with the result of that computation.
- `def successful[T](result: T): Future[T]`: Creates an already completed `Future` with the specified result.

A `Future` represents a value which may or may not **currently** be available, but will be available at some point, or an exception if that value could not be made available.

Random

- `def nextInt(n: Int): Int`: Returns a pseudorandom, uniformly distributed int value between zero (inclusive) and the specified value (exclusive).

Option

Relevant API for `Option[+T]`:

- `def isEmpty: Boolean`: Returns `true` if the option is `None`, `false` otherwise.
- `def get: T`: Returns the option's value. Throws an exception in case of `None`.
- `def orElse[R >: T](alternative: => Option[R]): Option[R]`: Returns `this Option` if it is nonempty, otherwise returns the result of evaluating `alternative`.

Constructors for `Option[+T]`:

- `case class Some[+T](value: T) extends Option[T]`: Represents existing values of type `T`.
- `case object None extends Option[Nothing]`: Represents non-existent values.

List

Relevant API for List[+T]:

- `def ::[B >: A](elem: B): List[B]`: Adds an element at the beginning of this list.
- `def :::[B >: A](prefix: List[B]): List[B]`: Adds the elements of a given list in front of this list.
- `def filter(p: (A) => Boolean): List[A]`: Selects all elements of this list which satisfy a predicate.
- `def flatMap[B](f: (A) => List[B]): List[B]`: Builds a new list by applying a function to all elements of this list and using the elements of the resulting lists.
- `def foldLeft[B](z: B)(op: (B, A) => B): B`: Applies a binary operator to a start value and all elements of this sequence, going left to right.
- `def foldRight[B](z: B)(op: (A, B) => B): B`: Applies a binary operator to all elements of this list and a start value, going right to left.
- `def foreach[U](f: (A) => U): Unit`: Apply f to each element for its side effects Note: [U] parameter needed to help scalac's type inference.
- `def head: A`: Selects the first element.
- `def isEmpty: Boolean`: Tests whether the list is empty.
- `def length: Int`: The length (number of elements) of the list.
- `def map[B](f: (A) => B): List[B]`: Builds a new list by applying a function to all elements of this list.
- `def reverse: List[A]`: Returns new list with elements in reversed order.
- `def tail: List[A]`: The rest of the list without its first element.

Set

Relevant API for Set[T]:

- `def +(elem: T): Set[T]`: Adds an element to the set.
- `def -(elem: T): Set[T]`: Removes an element from the set.
- `def foreach(action: T => Unit): Unit`: Executes an action for each element of the set.

Relevant API for the Set object:

- `def apply[T](elems: T*): Set[T]`: Creates a set with the specified elements.
- `def empty[T]: Set[T]`: Returns an empty set.

Map

Relevant API for Map[K, +V]:

- `def +(kv: (K, V)): Map[K, V]`: Creates a new map obtained by updating this map with a given key/value pair.
- `def -(key: K): Map[K, V]`: Removes a key from this map, returning a new map.
- `def get(key: K): Option[V]`: Optionally returns the value associated with a key.

Relevant API for the Map object:

- `def apply[K, V](elems: (K, V)*): Map[K, V]`: Creates a map with the specified elements.
- `def empty[K, V]: Map[K, V]`: Returns an empty map.

