
Parallelism and Concurrency

Midterm Exam

Wednesday, April 10, 2019

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 page of this exam is 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.

Exercise	Points	Points Achieved
1	20	
2	20	
3	20	
Total	60	

Exercise 1: Parallel Top-K (20 points)

Given an array of integers, your task in this exercise is to compute the k -largest elements *in descending order*. The function to implement has the following signature:

```
def topk(xs: Array[Int], k: Int): List[Int]
```

For instance, for a five-element array and $k = 2$ the function will return the two largest elements:

```
topk(Array(5, 7, 0, 11, 3), 2) == List(11, 7)
```

Note that in the special cases where $k = 1$ and $k = \text{xs.length}$, `topk(xs, k)` corresponds to computing the maximum element of `xs` and sorting `xs`, respectively:

```
topk(Array(5, 7, 0, 11, 3), 1) == List(11)
topk(Array(5, 7, 0, 11, 3), 5) == List(11, 7, 5, 3, 0)
```

Requirements

Your implementation should be *parallelized*, i.e., break down the computation into smaller chunks and use the `parallel` construct seen in class. For chunks of size k or smaller you should simply compute the solution sequentially (in other words, choose k as your threshold). Finally, the running time of your implementation assuming unbounded parallelism, i.e., the *depth*, should be in $O(k \cdot \log_2 n)$, where n is the length of the array passed to `topk`.

We strongly suggest you split your implementation into a separate function that will compute the k -largest elements for a given chunk of the overall array:

```
def topk(xs: Array[Int], k: Int): List[Int] =
  topkWithin(xs, k, 0, xs.length)
```

Helper Functions

You may assume the existence of a helper function that *slices* a part of an array and transforms it into a list:

```
def slice[T](xs: Array[T], from: Int, until: Int): List[T] = ...
// Example: slice(Array('a', 'b', 'c', 'd', 'e'), 1, 3) == List('b', 'c')
```

You may assume this function runs in time linear in the size of the returned slice. You may also want to consult the API at the end of this exam for useful methods on lists.

Implement `topkWithin` and any other auxiliary functions you might need on the following page.

```
def topk(xs: Array[Int], k: Int): List[Int] =  
  topkWithin(xs, k, 0, xs.length)  
  
def topkWithin(xs: Array[Int], k: Int, from: Int, until: Int): List[Int] =
```

Exercise 2: Aggregaddon (20 points)

The year is 2030, and a brave little robot named Will-Is has been sent to an (amazingly flat) asteroid threatening to destroy the Earth. The robot's mission is to reach a very specific location on the asteroid and plant a thermonuclear bomb that will, according to all estimates, destroy it. Since landing, all visual contact with the robot has been lost. Your goal in this exercise is to locate the robot so that this critical mission can continue.

Positions

The asteroid being completely flat, the position of the robot can be modeled as a simple two-dimensional vector.

```
case class Vector2D(x: BigInt, y: BigInt) {  
  def +(that: Vector2D) = Vector2D(this.x + that.x, this.y + that.y)  
}
```

Directions

The robot records and sends back to Earth all the moves it performs. Each move consists of advancing by one meter towards either North, East, South or West on the asteroid.

```
sealed abstract class Direction(val toVector: Vector2D)  
case object North extends Direction(Vector2D(0, 1))  
case object East extends Direction(Vector2D(1, 0))  
case object South extends Direction(Vector2D(0, -1))  
case object West extends Direction(Vector2D(-1, 0))
```

Figure 1 shows an example path taken by the robot. In this case, the sequence of directions taken by the robot would be North, North, North, East, East, South, South, East, South, South, South, East.

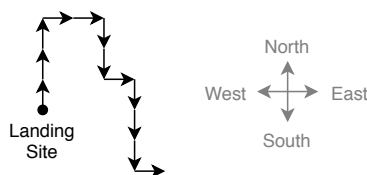


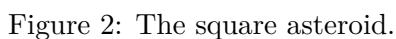
Figure 1: Landing site, path and directions.

Question 2a (8 points)

Using `aggregate`, implement a function that efficiently computes the final coordinates of the robot given the list of moves performed and the coordinates of the landing site. For this question, you should assume that the asteroid is infinite in all directions.

```
def getCoordinates(moves: ParSeq[Direction], start: Vector2D): Vector2D =
```

Unfortunately (or fortunately) the asteroid is not of infinite dimensions. The asteroid is shaped as a square with sides of size S meters (see figure 2). (What a weird object!)



If, at any point in time, the robot leaves the boundaries of the square, we consider that the robot fell into space and thus will not be able to complete its mission. A point (x, y) is considered within the square if and only if both x and y are between 0 and S (inclusive).

Implement a function that, given the list of moves performed, the coordinates of the landing site and the size S of the sides of the square, returns the coordinates of the robot wrapped in `Some` if it remained at all times on the asteroid, or `None` if the robot fell into space. Your function should be reasonably efficient and should use `aggregate`.

You may find it useful to construct a new datatype representing both a vector and a boundary box which keeps track of the min and max coordinate values on the path. It could look something like:

Think *very carefully* about how to add a `Direction` to such a vector and about how to add two such vectors together.

6

```
def getCoordinates(moves: ParSeq[Direction], start: Vector2D, size: BigInt): Option[Vector2D] =
```

Exercise 3: Concurrency (20 points)

EPFL has recently acquired a new priceless super-computer: the HelveticaBot™. EPFL is feeling generous and has decided that everyone should be able to use this new machine as long as they write Scala code, so they provide the following simple API:

```
object HelveticaBot {  
  /** Execute 'f()' on HelveticaBot.  
   *  
   * @return 'true' if the job was successfully completed, 'false' otherwise.  
   */  
  def runJob(f: () => Unit): Boolean  
}
```

At the morning launch event for HelveticaBot everything seems to be going well, but while you're enjoying your café-croissant you get an alert message on your phone: HelveticaBot is running out of resources because too many people are trying to run jobs at the same time! Since you know both Scala and concurrency well, you've been tasked with fixing this. You quickly turn off access to the machine by marking `object HelveticaBot` as `private` and come up with a replacement public API:

```
object HelveticaBotAccess {  
  def runJob(f: () => Unit): Boolean = synchronized {  
    HelveticaBot.runJob(f)  
  }  
}
```

This solves the immediate resource exhaustion problem but means that two jobs can never be run at the same time which is clearly not perfect. After some analysis you find that a limit of **10 concurrent jobs** would be optimal for this machine. `synchronized` isn't going to be enough here, you need a concurrency mechanism that can keep track of a non-negative integer: *semaphores* are such a mechanism.

```
/** Stores a non-negative integer initialized to 'initialValue'.  
 * The methods in this class are thread-safe.  
 */  
class Semaphore(initialValue: Int) {  
  assert(initialValue >= 0)  
  
  /** If the stored value is positive, decrement it and return.  
   * Otherwise, block until some other thread increments this semaphore,  
   * then decrement it and return.  
   *  
   * @return The new stored value.  
   */  
  def decrement(): Int  
  
  /** Increment the stored value and return.  
   *  
   * @return The new stored value.  
   */  
  def increment(): Int  
}
```

Note that if multiple threads call `decrement()` on the same semaphore instance with a stored value of 0 they will all block, and if another thread later calls `increment()` then only one thread will be chosen (arbitrarily) among the blocked threads to be unblocked, this guarantees that the stored value is always ≥ 0 .

A semaphore whose initial value is 1 is called a *binary semaphore*. A `synchronized` block that does not contain calls to `wait()`, `notify()` or `notifyAll()` can always be replaced by a binary semaphore, for example:

```
object Test {
  def hello(): Unit = synchronized {
    println("hello")
  }
}
```

is functionally equivalent to:

```
object Test {
  private val lock = new Semaphore(1)
  def hello(): Unit = {
    lock.decrement()
    println("hello")
    lock.increment()
  }
}
```

Question 3a (5 points)

Redo the implementation of `runJob` to allow up to 10 concurrent jobs using a semaphore. Your solution must not use `synchronized` or any concurrency API other than the `Semaphore` class seen above. Your solution must not change the public API of `HelveticaBotAccess`. You may assume all jobs terminate normally after a finite amount of time.

Hints: You will need to add an extra `private val` to `HelveticaBotAccess`. You will not need to use the values returned by `decrement` and `increment`.

Fill in the blanks below.

```
object HelveticaBotAccess {
  private val maxClients = 10

  def runJob(f: () => Unit): Boolean = {

    val success = HelveticaBot.runJob(f)

    success
  }
}
```

Question 3b (10 points)

The solution you implemented for Question 3a worked well! However, sometimes it is necessary to run a job with no other jobs running on the machine at the same time. A new method `runExclusiveJob` should be added to `HelveticaBotAccess` to enable this use-case. The precise requirements are:

- When a thread calls `runExclusiveJob`, the job won't be run as long as any other job is running on the machine.
- Once an exclusive job has started running, no other job will run until it has finished running.
- If the bot is issued a finite number of jobs, all jobs should eventually be run.

You do not need to worry about the order in which jobs will be executed, in particular your solution is not expected to be fair: it is acceptable not to schedule an exclusive job as long as other jobs are issued.

Like in Question 3a you are not allowed to use `synchronized`.

Hints: This question requires at least one more semaphore than the previous one, the extra semaphore(s) you will need should be binary semaphore(s). Feel free to use the values returned by `decrement` and `increment` if you find them useful.

Fill in the blanks in the next page.

```

object HelveticaBotAccess {
  private val maxClients = 10

  def runJob(f: () => Unit): Boolean = {

    val success = HelveticaBot.runJob(f)

    success
  }

  def runExclusiveJob(f: () => Unit): Boolean = {

    val success = HelveticaBot.runJob(f)

    success
  }
}

```

Question 3c (5 points)

Now that you know how to use semaphores, you should be able to implement them! Using `synchronized`, `wait()` and either `notify()` or `notifyAll()` complete the implementation of the `Semaphore` class by filling in the blanks:

```

/** Stores a non-negative integer initialized to 'initialValue'.
 * The methods in this class are thread-safe.
 */
class Semaphore(initialValue: Int) {
  assert(initialValue >= 0)
  private var value = initialValue

  /** If the stored value is positive, decrement it and return.
   * Otherwise, block until some other thread increments this semaphore,
   * then decrement it and return.
   *
   * @return The new stored value.
   */
  def decrement(): Int =

  /** Increment the stored value and return.
   *
   * @return The new stored value.
   */
  def increment(): Int =

}

```

Appendix

Parallel & Task

```
def parallel[A, B](x: => A, y: => B): (A, B)
```

Evaluate both arguments in parallel and return the pair of results.

```
def task[A](f: => A): Task[A]
```

Create a task that executes the computation passed as argument in a separate thread.

Methods of Task[A]

```
def join: A
```

Block until the task is completed and returns the computed value.

List

Methods of List[Int]:

```
def isEmpty: Boolean
```

Check whether this list is empty.

```
def head: Int
```

Return the first element of this list.

```
def tail: List[Int]
```

Select all elements except the first.

```
def sorted: List[Int]
```

Return a copy of this list with elements sorted *in ascending order* (i.e., from smallest to largest). You can assume this method runs in time $O(n \cdot \log_2 n)$, where n is the size of the list.

```
def reverse: List[Int]
```

Reverse the list. You can assume this method runs in time $O(n)$, where n is the size of the list.

ParSeq

Methods of ParSeq[A]:

```
def aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

Aggregate the results of applying an operator to the elements.

More on the next page.

BigInt

Methods of `BigInt`:

def `+(that: BigInt): BigInt`

Return the sum of this integer and that integer.

def `min(that: BigInt): BigInt`

Return the minimum between this integer and that integer.

def `max(that: BigInt): BigInt`

Return the maximum between this integer and that integer.

Monitor

Methods of `AnyRef`:

def `synchronized[A](x: => A): A`

Execute a computation within a synchronized block.

def `wait(): Unit`

Block the thread until notified.

def `notify(): Unit`

Wake up a single thread that is waiting on this object's monitor.

def `notifyAll(): Unit`

Wake up all threads that are waiting on this object's monitor.