Computer Science 3MI3 – 2020 homework 10

Using concurrency to break up large problems

Mark Armstrong

December 4, 2020

Contents

Introduction

This homework presents a somewhat contrived problem involving a computation over a slightly large dataset in order to justify the use of concurrency.

This problem is based on the one presented on day 1 of the Advent of Code programming challenge for 2020.

The homework provides a solution to the problem in Clojure, Ruby and Scala, and asks you to modify the solutions to take advantage of concurency.

Boilerplate

Submission procedures

Submission method

Homework should be submitted to your McMaster CAS Gitlab respository in the `cs3mi3-fall2020` project.

Ensure that you have **pushed** the commits to the remote repository in time for the deadline, and not just committed to your local copy.

Naming requirements

Place all files for the homework inside a folder titled `hn`, where `n` is the number of the homework. So, for homework 1, use the folder `h1`, for homework 2 the folder `h2`, etc. Ensure you do not capitalise the `h`.

Unless otherwise instructed in the homework questions, place all of your code for the homework in a single file in the `hn` folder named `hn.ext`, where `ext` is the appropriate extension for the language used according to this list:

- For Scala, `ext` is `sc`.
- For Prolog, `ext` is `pl`.
- For Ruby, `ext` is `rb`.
- For Clojure, `ext` is `clj`.

If multiple languages are used in the homework, submit a `hn.ext` file for each language.

If the language supports multiple different file extensions, you must still follow the extension conventions above.

Incorrect naming of files may result in up to a 10% deduction in your grade.

Do not submit testing or diagnostic code

Unless you are instructed to do so in the homework questions, **you should not submit testing code with your homework submission**.

This includes

- any `main` function,
- any `print` statements which output information **that is not directly requested as console output in the homework questions**.

If you do not wish to remove diagnostic print statements manually, you will have to find a way to ensure that they disabled in your final submission.

For instance, by using a wrapper on the print function or macros.

Due date and allowance for technical difficulties

Homework is due on the second Sunday following its release, by the end of the day (midnight). Submissions past 00:00 may not be considered.

If you experience technical difficulties leading up to the submission time, please contact Mark **ASAP** with the details of the problem and, if possible, attach the current state of your homework to the communication. This information will help ensure we are able to accept your submission once the technical difficulties are resolved.

Proper conduct for coursework

Individual work

Unless explicitely stated in the homework questions, all homework in this course is intended to be *individually completed.*

You are welcome to discuss the content of the homework in the public forum of the class Microsoft Teams team homework channel, though obviously solutions or partial solutions should not be posted or described.

Private discussions about the homework cannot reasonably be forbidden, but such discussions should follow the same guidelines as public discussions.

Inappopriate collaboration via private discussions which is later discovered by course staff may be considered academic dishonesty.

When in doubt, make the discussion private, or report its contents to the course staff by making a note of it in your homework.

To clarify what is considered appropriate discussions of homework content, here are some examples:

1. Discussing the language features introduced or needed for the homework.
    - Such as relevant builtin datatypes and datatype definition methods and their general use.
    - Code snippets that are not partial solutions to the homework are welcome and encouraged.
2. Questions of the form "What is meant by `x`?", "Does `x` really mean `y`?" or "Is there a mistake with `x`?"
    - Of course, questions of those form which would be answered by partial solutions are not considered appropriate.
3. Questions or advice about errors that may be encountered.
    - Such as "If you see a `scala.MatchError` you should probably add a catch-all `_` case to your `match` expressions."

Language library resources

Unless explicitely stated in the questions, it is not expected that you will use any language library resources in the homeworks.

Possible exceptions to this rule include implementations of datatypes we discuss in this course, such as lists or options/maybes, if they are included in a standard library instead of being builtin.

Basic operations on such types would also be allowed.

- For instance, `head`, `tail`, `append`, etc. on lists would not require explicit permission to be used.
- More complex operations such as sorting procedures would require permission before you used them.

Additionally, the standard *higher-order* operations including `map`, `reduce`, `flatten`, and `filter` are permitted generally, unless the task is to implement such a higher-order operator.

Part 0: The problem

You are provided with a collection of integers for this homework. What we want to compute on this collection is: Given an integer `sum`, find all possible pairs of (two of) those integers whose sum is less than or equal to `sum`, and return a sequence of those pairs.

(The pairing of one of the integers with itself is assumed not to count, unless that integer appears twice in the provided collection.)

Code is provided which solves this problem in Clojure, Ruby and Scala. Your task is then to improve the performance of this code using concurrency.

Note that we will not test for the exact ordering of the sequence, and it is not required to be in a specific order. We will instead test for membership within the list.

The collection is available here in plaintext, and below as a defined sequence in each language.

Part 1: Clojure [15 points]

The given collection is available in the file collection.clj, which defines it as a Clojure array `input`.

Below, you will see the function `summingPairs` implemented in Clojure to solve our given problem.

Implement a version of summingPairs which takes advantage of the future form to run parts of the computation in separate threads. You may choose how many threads to make use of. Try to improve the performance of summingPairs through this process.

Here is the function:

```
(defn summingPairs [xs sum]
  (letfn [(summingPairsHelper [xs the_pairs]
            ;; If `xs` is empty, we're done.
            (if (empty? xs) the_pairs
                ;; Otherwise, decompose `xs` into the `fst`
                ↪  element
                ;; and the `rest`.
                (let [[fst & rest] xs]
                  ;; We use the `recur` form to make the
                  ↪  recursive call.
                  ;; This ensures tail call optimisation
                  (recur
                   rest
                   ;; Concatenate `the_pairs` we have so far
                   ↪  with the sequence
                   ;; of every `[fst snd]` where `snd` is in
                   ↪  `rest` with
                   ;; `fst + snd <= sum`. The `doall` outside
                   ↪  the `concat`
                   ;; forces it to be calculated immediately;
                   ↪  without this,
                   ;; we get a (lazy) buildup of `concat`'s
                   ↪  which may
                   ;; cause a stack overflow when looking at
                   ↪  the result.
                   (doall
                    (concat the_pairs
                            (for [snd rest ;; For each `snd`
                             ↪  in `rest`...
                                  :when (<= (+ fst snd) sum)]
                              ;;... put `[fst snd]` into this
                              ↪  sequence.
                              [fst snd])))))))]
    (summingPairsHelper xs [])))
```

It can be run over the `input` using the following code, which also shows the starting and ending time of the computation, so you may judge its performance.

```
(load-file "./collection.clj")

(println (str
          "Starting at:   "
          (.getSecond (java.time.LocalDateTime/now))
          " seconds, "
          (.getNano (java.time.LocalDateTime/now))
          " nanoseconds"))
(println (summingPairs input 2020))
(println (str
          "Ending at:     "
          (.getSecond (java.time.LocalDateTime/now))
          " seconds, "
          (.getNano (java.time.LocalDateTime/now))
          " nanoseconds"))
```

Part 2: Ruby [15 points]

Repeat part 1 in Ruby, potentially using the below method `summingPairs` as a starting point.

The `input` collection is defined as a Ruby array in the file collection.rb.

The method:

```
Pair = Struct.new(:fst,:snd)

def summingPairs(xs, sum)
  the_pairs = []
  len = xs.length

  for i in 0..(len-1)
    for j in (i+1)..(len-1)
      if xs[i] + xs[j] <= sum
        the_pairs.push(Pair.new(xs[i],xs[j]))
      end
    end
  end
  return the_pairs
end
```

And code to make use of it on the `input` array:

```
require 'date'
require_relative 'collection'

puts "Starting at:   #{DateTime.now.sec} seconds,
 ↪  #{DateTime.now.strftime("%9N")} nanoseconds"
puts summingPairs(INPUT,2020)
puts "Ending at:     #{DateTime.now.sec} seconds,
 ↪  #{DateTime.now.strftime("%9N")} nanoseconds"
```

Part 3: Scala [15 points]

Once more, repeat parts 1 and 2, this time in Scala. You may make use of the below method `summingPairs` if you like.

This time, the `input` collection is defined as a Scala array in collection.sc. Note this is an array, not a list as we have usually made use of in Scala. The size of the collection necessitated avoiding the (linked) list type.

The method:

```
def summingPairs(xs: Vector[Int], sum: Int):
 ↪  Vector[Tuple2[Int,Int]] = {
  def summingPairsHelper(xs: Vector[Int],
                         the_pairs: Vector[Tuple2[Int,Int]]):
                          ↪  Vector[Tuple2[Int,Int]] =
    xs match {
      case fst +: rest =>
        // Search through `rest` for numbers `snd` such that
         ↪   `fst + snd` is the `sum`.
        val pairs_here = rest.collect({case snd if fst + snd
         ↪  <= sum => (fst,snd)})
        // Make the recursive call, adding in the pairs we
         ↪  just found.
        summingPairsHelper(rest, the_pairs ++ pairs_here)
      case _ => the_pairs // If there's no head element, the
       ↪  vector is empty.
    }
  summingPairsHelper(xs,Vector())
}
```

And code to make use of it on the `input` array:

```
import java.time.LocalDateTime
import $file.collection, collection._

println(s"Starting at:   ${LocalDateTime.now.getSecond}
↪   seconds, ${LocalDateTime.now.getNano} nanoseconds")
println(summingPairs(input,2020))
println(s"Ending at:     ${LocalDateTime.now.getSecond}
↪   seconds, ${LocalDateTime.now.getNano} nanoseconds")
```

Part 4: Prolog [5 bonus points]

For bonus marks, implement a `summingPairs` predicate in Prolog, making use of the `concurrent` predicate.

Testing

:TODO: