Towards a Data Analytics API targeted to FastFlow

Guy Tremblay
Professeur
Département d'informatique

UQAM
http://www.labunix.uqam.ca/~tremblay

20 juin 2016

Presentation outline

1. PRuby : A Ruby *gem* for parallel programming
2. An example illustrating various frameworks :
   The «Hello world» of Big Data
3. Some additional constructs
4. Next steps

PRuby : A Ruby *gem* for parallel programming

The syntax used in the examples = `PRuby::Stream`

PRuby

A small Ruby *gem* (library) that I wrote for my parallel programming courses

PRuby supports various parallel programming style/constructs

- Task parallelism
  - `pcall`
  - `future`
- Loop parallelism «*à la OpenMP*»
  - `peach`
  - `peach_index`
- Data parallelism (`Array` based)
  - `pmap`
  - `preduce`
- Flow parallelism «*à la Go*»
  - `pipeline_source`, `|`, `pipeline_sink`
  - `each`, `<<`, `close`

PRuby supports various parallel programming style/constructs

- Task parallelism
  - `pcall`
  - `future`
- Loop parallelism «*à la OpenMP*»
  - `peach`
  - `peach_index`
- Data parallelism (`Array` based)
  - `pmap`
  - `preduce`
- Flow parallelism «*à la Go*»
  - `pipeline_source`, `|`, `pipeline_sink`
  - `each`, `<<`, `close`
- Flow parallelism «*à la Java 8.0* `Stream`»
  - …

The syntax used in the examples = `PRuby::Stream`

PRuby : :Stream

- Started as an API mimicking Java 8.0 `Stream`
- Extended with additionnal nodes :
  - `stateful`
  - `ff_node`
  - `go`
- Recently extended (last week !) with additional operations from Spark, Flink, FlumeJava to compare them

The current implementation of `PRuby::Stream` is simple/naive

- Each node is an independant thread
- The thread is activated as soon the node is created
- The stream elements are evaluated eagerly
- Bounded buffers are used $\Rightarrow$ Producer is delayed if too fast

The syntax used in the examples = Pure `Ruby`

- Blocks and lambda-expressions

```
map { |x| 2 * x }            # Block

map( ->(x) { 2 * x } )



map( &:fst )                 # Implicit block

map( ->(x) { x.fst } )
```

The syntax used in the examples = Pure `Ruby`

- Blocks and lambda-expressions
  Instead of lambda-expressions only

```
map { |x| 2 * x }         # Block

map( ->(x) { 2 * x } ) # Lambda expr.
```

```
map( &:fst )              # Implicit block

map( ->(x) { x.fst } ) # Lambda expr.
```

The syntax used in the examples = Pure `Ruby`

- Keyword arguments

```
stateful( initial_state: 0 )
        { |s, x| ...foo... }

stateful( initial_state: 0,
          at_eos: -> (s) { s } )
        { |s, x| ...foo... }


stateful( 0, ->(s, x) { ...foo... } )

stateful( 0, ->(s) { s },
             ->(s, x) { ...foo... } )
```

The syntax used in the examples = Pure Ruby

- Keyword arguments
  Instead of optional arguments or fluent interface

```
stateful( initial_state: 0 )
        { |s, x| ...foo... }

stateful( initial_state: 0,
          at_eos: -> (s) { s } )
        { |s, x| ...foo... }


stateful( 0, ->(s, x) { ...foo... } )

stateful( 0, ->(s) { s },
             ->(s, x) { ...foo... } )
```

The syntax used in the examples = Pure `Ruby`

- Implicit pair = `Array` of size 2

```
map { |x| [x, 1] }        # Array of size 2


map { |x| Pair.new(x, 1) } # Explicit Pair
# or
map { |x| Pair[x, 1] }     # Explicit Pair
```

The syntax used in the examples = Pure `Ruby`

- Implicit pair = `Array` of size 2
  Instead of explicit `Pair` objects

```
map { |x| [x, 1] }         # Array of size 2


map { |x| Pair.new(x, 1) } # Explicit Pair
# or
map { |x| Pair[x, 1] }     # Explicit Pair
```

An example illustrating various frameworks :
The «Hello world» of Big Data

A common example used to illustrate the «API style» of the various frameworks

The «Hello World» of Big Data

A common example used to illustrate the «API style» of the various frameworks

The «Hello World» of Big Data

- Word counting

The frameworks that are illustrated

- MapReduce
- Spark
- Flink
- Java 8.0 `Stream` API
- FlumeJava

The frameworks that are illustrated

- MapReduce
- Spark
- Flink
- Java 8.0 `Stream` API
- FlumeJava (aka. Google Dataflow, Apache Beam)

MapReduce

MapReduce «à la Hadoop»

Expressed using files, with line-by-line mode

```
mapper = lambda do |offset, line, output|
  line.split( ' ' ).each do |word|
    output.emit [word, 1]
  end
end

reducer = lambda do |word, occs, output|
  output.emit [word, occs.reduce(:+)]
end

MapReduce.new( mapper, reducer )
         .run( input_file, output_file )
```

MapReduce «à la Hadoop»

Expressed using **lines**, to make it similar to other examples

```
mapper = lambda do |line, output|
  line.split( ' ' ).each do |word|
    output.emit [word, 1]
  end
end

reducer = lambda do |word, occs, output|
  output.emit [word, occs.reduce(:+)]
end

MapReduce.new( mapper, reducer )
         .run( lines )
         .to_a
```

Spark

Using `group_by_key`

Must use an explicit collection of pairs

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by_key # Input = collection of pairs
  .map { |w, occs| [w, occs.reduce(&:+)] }
  .to_a
```

Using `group_by_key`

Must use an explicit collection of pairs

```
lines = ["abc def ghi", "abc def", "abc"]

puts Stream.source(lines)
     .flat_map { |line| line.split( ' ' ) }
     .map { |word| [word, 1] }
     .group_by_key
     .peek { |k, v| puts "'#{k}' => #{v}" }
     .map { |w, occs| [w, occs.reduce(&:+)] }
     .to_a
-----------------------------------
'abc' => [1, 1, 1]
'def' => [1, 1]
'ghi' => [1]

[["abc", 3], ["def", 2], ["ghi", 1]]
```

Using `reduce_by_key`

Combine grouping with reduction

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .reduce_by_key { |x, y| x + y }
  .to_a
```

Flink

Using `group_by`

Can group by any property, but the whole object is used, not just the value

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by(&:fst) # Input not necessarily pairs
  .map { |w, occs| [w, occs.map(&:snd)] }
  .map { |w, occs| [w, occs.reduce(&:+)] }
  .to_a
```

Using `group_by`

Can group by any property, but the whole object is used, not just the value

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by(&:fst)
  .peek { |k, v| puts "'#{k}' => #{v}" }
  .map { |w, occs| [w, occs.map(&:snd)] }
  .peek { |k, v| puts "'#{k}' => #{v}" }
  .map { |w, occs| [w, occs.reduce(&:+)] }
  .to_a
--------------------------------
'abc' => [["abc", 1], ["abc", 1], ["abc", 1]]
'def' => [["def", 1], ["def", 1]]
'ghi' => [["ghi" , 1]]

'abc' => [1, 1, 1]
'def' => [1, 1]
'ghi' => [1]
```

Using `group_by`

Can group by any property, but the whole object is used, not just the value

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by(&:fst)
  .map { |w, occs| [w, occs.map(&:snd)] }
  .map { |w, occs| [w, occs.reduce(&:+)] }
  .to_a
```

Using `group_by` and more complex `reduce`

Can group by any property, but the whole object is used, not just the value

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by(&:fst)
  .map do |w, occs|
        [w, occs.reduce(0) { |a, x| a+x.snd }]
      end
  .to_a
```

Using `group_by` and value mapping

Can group by any property, but the whole object is used, not just the value, unless value mapping function is provided

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by( map_value: ->(x) { x.snd }, &:fst )
  .map { |w, occs| [w, occs.reduce(&:+)] }
  .to_a
```

Using `group_by` and value mapping

Can group by any property, but the whole object is used, not just the value, unless value mapping function is provided

```
lines = ["abc def ghi", "abc def", "abc"]

Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by( map_value: ->(x) { x.snd }, &:fst )
  .peek { |k, v| puts "'#{k}' => #{v}" }
  .map { |w, occs| [w, occs.reduce(&:+)] }
  .to_a
-----------------------------------
'abc' => [1, 1, 1]
'def' => [1, 1]
'ghi' => [1]
```

Using `group_by` and `sum_by_key`

```
Stream.source(lines)
  .flat_map { |line| line.split( ' ' ) }
  .map { |word| [word, 1] }
  .group_by(&:fst)
  .sum_by_key(&:snd)
  .to_a
```

Java 8.0 `Stream` API

Java 8.0 : Streams vs. Collections

[S]treams differ from collections in several ways :

- No storage[:] they carry values from a source [. . .] through a pipeline of computational steps
- Functional in nature
- Laziness-seeking
- Bounds optional [$\Rightarrow$ allow for potentially] infinite streams

`http://www.drdobbs.com/jvm/lambdas-and-streams-in-java-8-libraries/240166818`

What streams are not good for

Streams should be used with high caution when processing intensive computation tasks. In particular, by default, all streams will use the same `ForkJoinPool`*, configured to use as many threads as there are cores in the computer on which the program is running.*

If evaluation of one parallel stream results in a very long running task, this may be split into as many long running sub-tasks that will be distributed to each thread in the pool. From there, no other parallel stream can be processed because all threads will be occupied.

https://dzone.com/articles/whats-wrong-java-8-part-iii

Grouping by key is done through `collect` operations

The `collect` operation turns a `Stream` into a regular—non-parallel—collection

```
Stream.source(lines)
  .flat_map { |line| line.split( ’ ’ ) }
  .map { |word| [word, 1] }
  .collect_grouping_by { |w, c| w }
  .stream  # Turn back collection into a Stream!
  .map do |w, occs|
        [w, occs.reduce(0) { |a, x| a+x.snd }]
      end
  .to_a
```

FlumeJava — aka. Google Dataflow/Apache Beam

Must use `parallel_do` with explicit `emit` operation

Mapping of elements for reduction is implicit

```
Stream.source(lines)
  .parallel_do do |line, emitter_fn|
    line.split(' ').each { |l| emitter_fn.emit l }
   end
  .parallel_do do |word, emitter_fn|
    emitter_fn.emit [word, 1]
   end
  .group_by_key
  .combine_values { |x, y| x + y }
  .to_a
```

Some additional constructs

Simple state manipulation—using functional style

Scan-like operation = All intermediate sums : $\sum_{i=0}^{1} x_i, \sum_{i=0}^{2} x_i, \ldots, \sum_{i=0}^{n-1} x_i, \sum_{i=0}^{n} x_i$

```
cumulate_total = lambda do |total, x|
  [ total + x, total + x ]
end

Stream.source( [10, 20, 30, 40] )
  .stateful( initial_state: 0,
             &cumulate_total )
  .to_a
  .must_equal [10, 30, 60, 100]
```

Scan-like operation = All intermediate sums, with 0^{th} : $\sum_{i=0}^{0} x_i, \sum_{i=0}^{1} x_i, \sum_{i=0}^{2} x_i, \ldots, \sum_{i=0}^{n-1} x_i, \sum_{i=0}^{n} x_i$

```
cumulate_total = lambda do |total, x|
  [ total + x, total ]
end

Stream.source( [10, 20, 30, 40] )
  .stateful( initial_state: 0,
             at_eos: -> total { total },
             &cumulate_total )
  .to_a
  .must_equal [0, 10, 30, 60, 100]
```

FastFlow style node

Simple filtering node

```
Stream.source( [1, 2, 3, 4] )
  .ff_node do |x|
    if x % 2 == 0
      x
    else
      PRuby::GO_ON
    end
   end
  .to_a
  .must_equal [2, 4]
```

Simple generating node with use of «send_out»

```
Stream.source([10])
  .ff_node do |n, out_channel|
     for k in 1..n
       out_channel << k    # ff_send_out
     end
     PRuby::EOS
   end
  .ff_node { |x| x * 10 }
  .to_a
  .must_equal (1..10).map { |x| x * 10 }
```

Simple node with internal state—using functional style

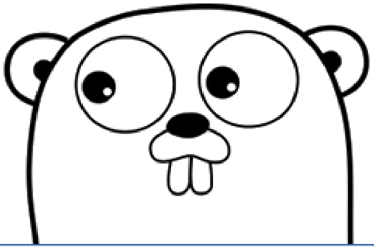
```
Stream.source( [10, 20, 30, 40] )
  .ff_node_with_state do |state, x|
     if state.nil?
       [x, PRuby::GO_ON]  # Wait for 2nd number
     else
       [nil, state + x]
     end
   end
  .to_a
  .must_equal [30, 70]
```

Go style node

Go

https://golang.org/

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.



Download Go

Binary distributions available for
Linux, Mac OS X, Windows, and more.

Go

[C]reated at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson.

[A] compiled, statically typed language in the tradition of Algol and C, with garbage collection, limited structural typing, memory safety features and CSP-style concurrent programming features added.

https://en.wikipedia.org/wiki/Go_(programming_language)

The Go Language : Channels and basic operations

- `c chan int`

- `<- c`

- `c <- v`

The Go Language : Channels and basic operations

- `c chan int` — # declare channel `c`
- `<- c` — # read from channel `c`
- `c <- v` — # write `v` to channel `c`

Sum of two consecutive elements

```
Stream.source( [10, 20, 30, 40] )
  .go do |cin, cout|
    while (v1 = cin.get) != PRuby::EOS
      cout << v1 + cin.get
    end
   end
  .to_a
  .must_equal [30, 70]
```

User-defined stream transformations

User-defined stream transformations using explicit `apply` operation

```
plus_2 = lambda do |s|
  s.map { |x| x + 1 }.map { |x| x + 1 }
end
take_2 = lambda do |s|
  s.take(2)
end

Stream.source([10, 20, 30])
  .apply( &plus_2 )
  .apply( &take_2 )
  .to_a
  .must_equal [12, 22]
```

User-defined stream transformations using pipe-like operation

```
plus_2 = lambda do |s|
  s.map { |x| x + 1 }.map { |x| x + 1 }
end
take_2 = lambda do |s|
  s.take(2)
end

(Stream.source([10, 20, 30]) >> plus_2 >> take_2)
  .to_a
  .must_equal [12, 22]
```

Stream joining

A (standard) `join` that processes streams of pairs : the joining key is implicit = first element of the pair

```
s0 = Stream.source( [[1, 2], [3, 4], [3, 6]] )
s1 = Stream.source( [[3, 9]] )

s0.join( s1 )
  .to_a
  .must_equal [[3, [4, 9]], [3, [6, 9]]]
```

A `join` that processes arbitrary streams : the joining key is explicit = lambda argument

```
s0 = Stream.source( [[2, 1], [4, 3], [6, 3]] )
s1 = Stream.source( [[9, 3]] )

s0.join( s1,
         key: ->(s){ s.snd } )
  .to_a
  .must_equal [[3, [[4, 3], [9, 3]]],
               [3, [[6, 3], [9, 3]]]]
```

A `join` that processes arbitrary streams : the joining key is explicit = lambda argument

And the values are simplified using `map_value`

```
s0 = Stream.source( [[2, 1], [4, 3], [6, 3]] )
s1 = Stream.source( [[9, 3]] )

s0.join( s1,
         key: ->(s){ s.snd },
         map_value: ->(s){ s.fst } )
  .to_a
  .must_equal [[3, [4, 9]],
               [3, [6, 9]]]
```

A `tee` operation ?

A `tee` operation that duplicates a stream

```
s1, s2 = Stream.source( [10, 20, 30] )
           .tee

s1.map { |x| x / 10 }
  .to_a
  .must_equal [1, 2, 3]

s2.map { |x| 2 * x }
  .to_a
  .must_equal [20, 40, 60]
```

A `tee` operation that duplicates a stream

```
s1, s2, s3 = Stream.source( [10, 20, 30] )
                   .tee(nb_outputs: 3)

s1.map { |x| x / 10 }
  .to_a
  .must_equal [1, 2, 3]

s2.map { |x| 2 * x }
  .to_a
  .must_equal [20, 40, 60]

s3.map { |x| x + 1 }
  .to_a
  .must_equal [11, 21, 31]
```

Next steps

Next steps

- Look at some other binary operators :
  - `union`, `union`
  - `cogroup`
  - Another style for `fork` ?
- Agree on the key elements of API
- Rewrite the API and examples in Ruby/C++ style
- Define the C++ API