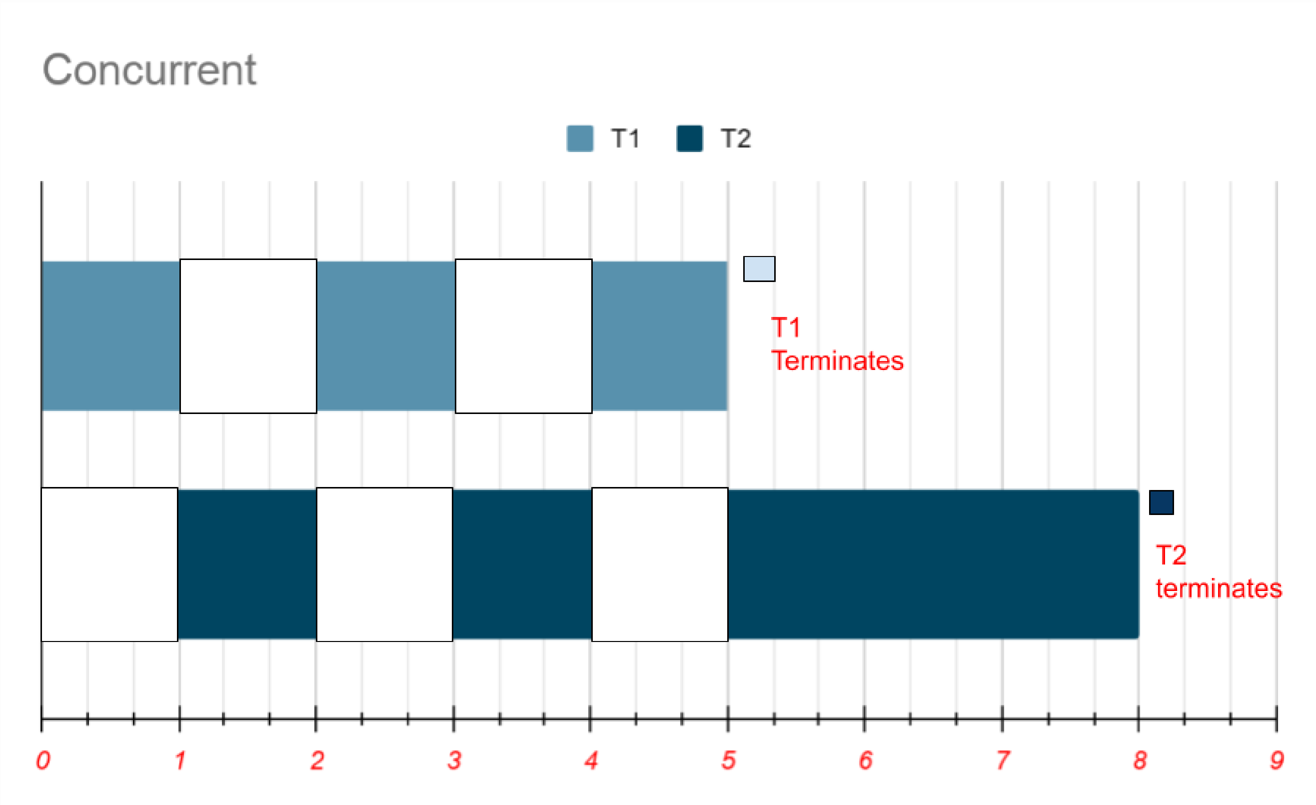
Parallelism in Ruby 3.0 with Ractors

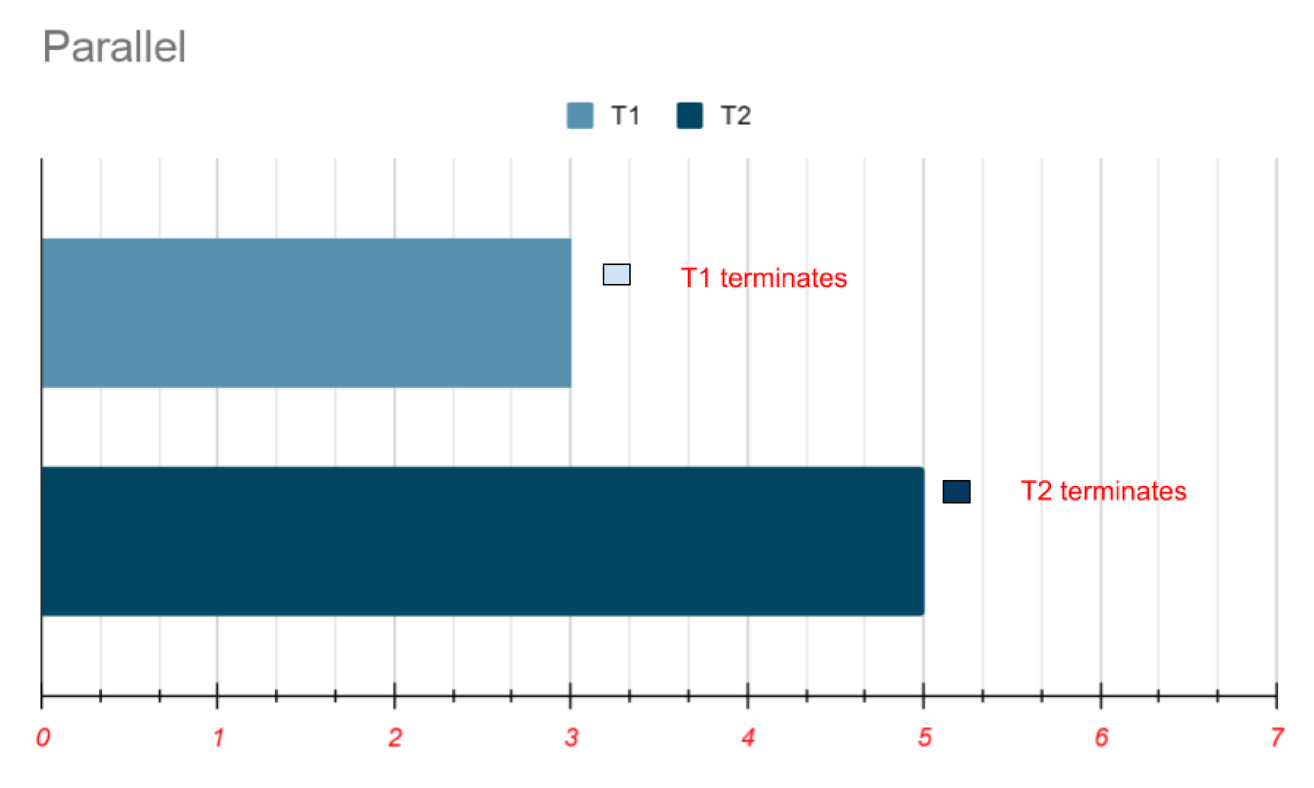
Feb 5, 2021 • Bishwa Hang Rai

With the release of *Ruby 3.0*, one of the most awaited feature was the release of **Ractor**. It is still in its experimental phase, but it shows quite big promise, especially with parallel execution of multiple Ractors (and code enclosed in it).

First, let's do a quick recap about Concurrency and Parallelism. In *Parallelism*, multiple tasks can start and finish simultaneously. Each task can be executed independently without interruptions.
In *Concurrency* , multiple task can be started together, however, in a given time instance, only one task is actually being processed(running). Meanwhile, the other tasks sits idle, and wait to get access to the resource(processor).



Concurrency



Parallelism

Let's take an example of execution of two tasks *T1* and *T2* as shown in the above pictures. In a concurrent system, both tasks start at time `t_0`, but only *T1* is actually in running state, while *T2* is in idle state, waiting for the access to resource. At `t_1`, the access switches. Now, *T2* has the access to the resource and is in running state, where as *T1* sits idle. The state again flips at `t_3` and `t_4`.

The total time taken for task `T1` is `5 seconds`, rather than `3 seconds`, because of interruptions. And, for `T2`, it's `8 seconds`, rather than `5 seconds`. Altogether, both tasks finished in `max(5, 8) = 8` seconds.

In a parallel system, both tasks *T1* and *T2* started together and ran simultaneously without interruptions. Both were always on running state, until termination. The time taken to finish `T1` is `3 seconds`, and for `T2` it is `5 seconds`. Altogether, both tasks finished in `max(3, 5) = 5` seconds.

Brief Introduction to Ractors

Ruby so far (< 3.0.0) did not support true parallelism, because of the **"Global VM Lock (GVL)"**.
GVL permits only single thread to access the `Ruby VM` at a time, restricting multiple thread to run their code simultaneously. Multiple threads take turn to get the access to "GVL" and run their code, thus making multi-threaded program concurrent, rather than parallel.

The official doc defines `Ractor` as: *"Ractor is a Actor-model abstraction for Ruby that provides thread-safe parallel execution."* In other to achieve that, each Ractor has its own GVL lock, and its own code running context. Ractors cannot directly access each other's objects.

Ractors communicate with each other strictly via **"send/receive (push type)"**, or **"yield/take (pull type)"** method. They can use this communication protocol to share the objects with each other. Only shareable objects – Integer, frozen String (*default in Ruby 3*), ractor itself, and such – can be shared via this communication protocol[1].

```
                   Ractor r
                  +------------------------------------------+
                  | incoming                        outgoing |
                  | port                                port |
    r.send(obj) ->*->[incoming queue]     Ractor.yield(obj) ->*-> r.take
                  |                |                         |
                  |                v                         |
                  |           Ractor.receive                 |
                  +------------------------------------------+

             Fig: Ractor Sending and Receiving ports (from ruby/ractor.md)
```

```ruby
# ractor send/receive
r = Ractor.new do
  message = Ractor.receive
  puts "Message received: #{message}"
end

r.send("Hello from the other side")
# Output: 'Message received: Hello from the other side'

# ractor yield/take
r = Ractor.new do
  Ractor.yield 42
end
message = r.take
puts "Message taken: #{message}"
# Output: 'Message taken: 42'
```

With it's own "GVL", and only communicating via `send/receive` or `yield/take` with each other, multiple Ractors can run in *parallel* without blocking each other.

To learn more in details about Ractor, the Master Doc is a good place to start.

In order to play around with Ractors and test the new true *parallelism*, we will benchmark it to compute **factorial** of numbers.

Sequential Implementation

Let's start with sequential processing. We will write a method to compute factorial of any given number `n`, using the new `endless method` syntax in "Ruby 3.0.0"

```
# using Ruby 3.0.0 endless method syntax
def fact(n) = (1..n).inject(:*) || 1
puts fact(5) # => 120
```

To generate factorial of up to `n = 5000` numbers sequentially, we can simply loop `n` times and call the `fact` method.

```
TOTAL_COUNT = 5000
@result_seq = TOTAL_COUNT.times.map do |i|
  fact(i)
end.sort
```

Multi-Thread Implementation

To use mulit-thread for the factorial generation, we will use thread safe Queue data structure and its pop() method. We will first fill up the queue with the input numbers. Then, we will create `5` worker threads, that will pop the numbers from the queue (*which is thread safe*), and compute its factorial.

If the queue is `empty?`, then by default the `pop()` method will suspend the calling thread and wait until a number is pushed into the queue again. In our case, empty queue means we have finished our tasks of computing factorial of all the input numbers, and our worker threads should be terminated.

We achieve this with `queue.pop(true)`. By passing `true`, the `pop` method becomes *non-blocking* operation, and it will raise `ThreadError` when the queue is empty. We will rescue the `ThreadError`, and do nothing, which will terminate the thread gracefully.

```
queue = Queue.new
TOTAL_COUNT.times do |n|
  queue << n # fill up the queue with numbers
end

@result_threads = []
MAX_WORKERS = 5

workers = MAX_WORKERS.times.map do
  Thread.new do
    begin
      while n = queue.pop(true) # raises error when queue empty
        @result_threads << fact(n)
      end
```

```
    rescue ThreadError
      # queue has been processed, exit thread
    end
  end
end
workers.each(&:join)
@result_threads.sort!
```

Multi-Ractor Implementation

Finally, let's try Ractor based multiple workers. We will create `5` ractor workers, and pass the input number to each ractor worker via the `send` method, like we discussed in the introduction before. This will push the passed number into the ractor's incoming queue, from which the receiving `ractor` will pop the number `n`, one at a time, with `Ractor.receive` method. It will then calculate the factorial of popped number `n` with, `fact(n)`.

In the end, we collect the results (factorial of all input numbers), with the method `Ractor.select`. The `select()` method accepts list of `ractors` as an argument and listens to the each `ractor's` outgoing port. It returns any first ractor that has something in it's outgoing port, defined by `_r` in our example code, and along with it the object yielded by the same returned ractor, defined by `result` in our example code. We are interested in the `result` part, which we collect and then sort.

```
workers = []

workers = MAX_WORKERS.times.map do
  Ractor.new do
    while n = Ractor.receive
      Ractor.yield fact(n)
    end
  end
end

TOTAL_COUNT.times do |n|
  workers[n % MAX_WORKERS].send(n)
end

@result_ractors = TOTAL_COUNT.times.map do
  _r, result = Ractor.select(*workers)
  result
end.sort
```

Benchmarking

Putting all the above implementation together (see below for full code), and running the benchmark, produced following results.

```
Rehearsal -------------------------------------------------------
Sequential           10.883678   0.327006  11.210684 ( 11.222835)
Threads (5-workers)  11.272800   0.249546  11.522346 ( 11.533432)
Ractors (5 workers) <internal:ractor>:267: warning: Ractor is experimental, and the
 17.856913   5.290015  23.146928 (  7.181732)
--------------------------------------------- total: 45.879958sec

                          user     system      total        real
Sequential           12.184099   0.452136  12.636235 ( 12.648889)
Threads (5-workers)  12.506612   0.606289  13.112901 ( 13.124167)
Ractors (5 workers)  18.147833   4.863485  23.011318 (  7.221413)
```

The sequential implementation finished in `12.648899` seconds, which is a bit faster (`1.037` times) than multi-threaded implementation, which finished in `13.124167` seconds. The multi-threaded implementation is slower than sequential because of the contention for the `GVL` between the worker threads, during the execution (concurrent).

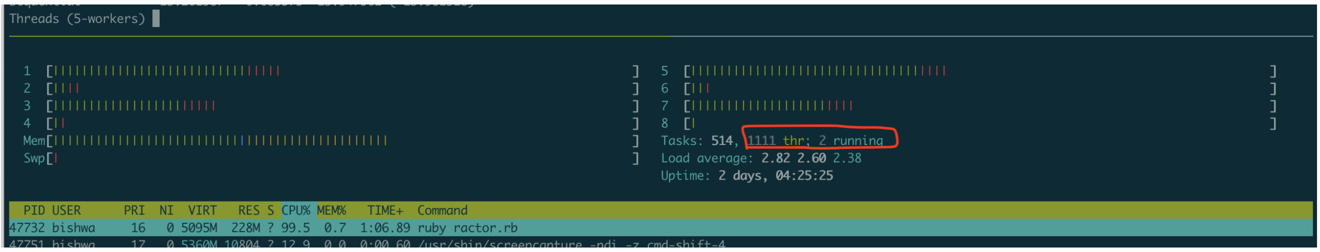
The clear winner, by far, is `ractor` based implementation. It took only `7.221413` seconds, which is `1.75` times faster than the sequential implementation, and `1.81` times faster than the multi-thread implementation.

One interesting observation that we can see is that, in case of `Ractor implementation`, the total CPU time, `user (18.147833) + system (4.863485) = 23.011318 seconds`, is way higher than the real clock time `7.221413 seconds`. I was curious about it and wanted to know the **Why?**.

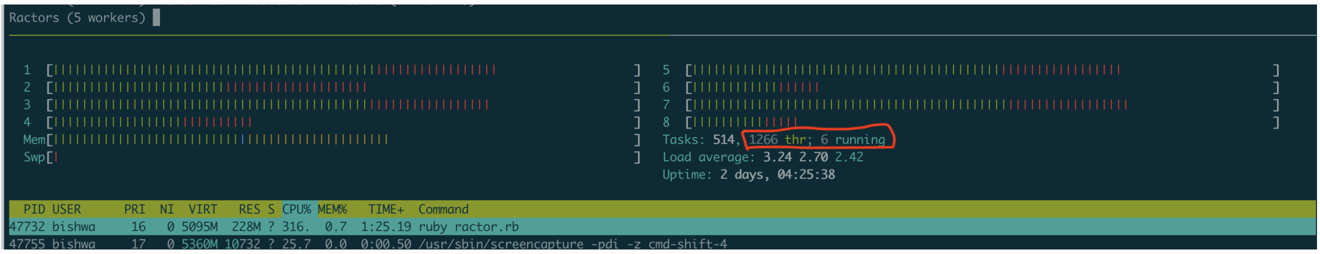
I asked the Internet and found some answers that said, if the `real time < user time`, the process took advantage of multi-cores and parallel execution [2]. And also, when the ratio of `CPU time / real time` is greater than 1, the number symbolized (roughly) the number of usage of cores in multi-core processor[3]. The ratio (`CPU time / real time`) in case of the *Ractor implementation* is `3.18 (23.011318 / 7.221413)`, which means probably `3 CPU core` was used during the execution of process. My machine is Quad-Core. ¯\(ツ)/¯

I did an easy check by monitoring my machine cores during the program execution with `htop`, and I could see that the cores were being used way more during `ractor workers` execution, than during `threads workers`. And the number of threads running during `Threads implementation` were `2`, where as during `Ractors implementation`, it were `6`. My guess, `5` threads for each ractor workers, as defined in our code, and `1` for the main program? ¯\(ツ)/¯

I would love to hear from others, if they know better explanation about `real time` vs `CPU time`, or any pointers where I could look into.



htop during thread execution



htop during ractor execution

Remarks

Ractors is clearly faster and brings true parallelism to Ruby. However, we need to wait for its final stable release, along with its adoption in all of the popularly used gems (e.g., rails, sidekiq, sequel), before we can take full benefit of its speed from parallelism. And well, that could take some time. :-)

1. The benchmarking was done with **"RUBY_VERSION = 3.0.0"** on an **iMac** (Processor: *"4 GHz Quad-Core Intel Core i7"*), Memory: *"32 GB 1867 MHz DDR3"*)
2. Ractor is not yet stable. It crashed sometimes, when I first created large number of independent threads (`Thread.new`), before creating ractors (`Ractor.new`). Screenshot
3. We didn't look into **Fiber** and **Process fork**.

Full Implementation

```ruby
#!/usr/bin/env ruby

require 'benchmark'

def fact(n) = (1..n).inject(:*) || 1
TOTAL_COUNT = 5000
MAX_WORKERS = 5

Benchmark.bmbm do |x|
  x.report("Sequential") do
    @result_seq = TOTAL_COUNT.times.map do |i|
      fact(i)
```

```ruby
    end.sort
  end

  x.report("Threads (#{MAX_WORKERS}-workers)") do
    queue = Queue.new
    TOTAL_COUNT.times do |n|
      queue << n # fill up the queue with numbers
    end

    @result_threads = []

    workers = MAX_WORKERS.times.map do
      Thread.new do
        begin
          while n = queue.pop(true) # raises error when queue empty
            @result_threads << fact(n)
          end
        rescue ThreadError
          # queue has been processed, exit thread
        end
      end
    end
    workers.each(&:join)
    @result_threads.sort!
  end

  x.report("Ractors (#{MAX_WORKERS} workers)") do
    workers = []

    workers = MAX_WORKERS.times.map do
      Ractor.new do
        while n = Ractor.receive
          Ractor.yield fact(n)
        end
      end
    end

    TOTAL_COUNT.times do |n|
      workers[n % MAX_WORKERS].send(n)
    end

    @result_ractors = TOTAL_COUNT.times.map do
      _r, result = Ractor.select(*workers)
      result
    end.sort
  end
end
```

```ruby
puts "*" * 10
p @result_seq.take(6) # => [1, 1, 2, 6, 24, 120]
p @result_threads.take(6) # => [1, 1, 2, 6, 24, 120]
p @result_ractors.take(6) # => [1, 1, 2, 6, 24, 120]
p [@result_ractors, @result_threads, @result_ractors]
    .each_cons(2).map {|a, b| a == b}.all? # => true
```

Ractors Crash

```
Ractors (worker pool)         <internal:ractor>:267: warning: Ractor is experimental,
ractor.rb:10: [BUG] Segmentation fault at 0x0000000000000000
ruby 3.0.0p0 (2020-12-25 revision 95aff21468) [x86_64-darwin20]

-- Crash Report log information --------------------------------------------
   See Crash Report log file under the one of following:
     * ~/Library/Logs/DiagnosticReports
     * /Library/Logs/DiagnosticReports
   for more details.
Don't forget to include the above Crash Report log file in bug reports.

-- Control frame information -----------------------------------------------
SEGV received in SEGV handler
Abort trap: 6
```

Ractor Crash

Footnotes

1. Limited sharing between multiple ractors ↩
2. Why real time can be lower than user time ↩
3. Where’s your bottleneck? CPU time vs wallclock time ↩