Extracted from:

The dRuby Book

Distributed and Parallel Computing with Ruby

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The dRuby Book

Distributed and Parallel Computing with Ruby

Masatoshi Seki

Translated by Makoto Inoue

Foreword by Yukihiro “Matz” Matsumoto

Edited by Susannah Davidson Pfalzer

The Facets of Ruby Series

The team that produced this book includes:

Susannah Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Original Japanese edition:
“dRuby niyoru Bunsan Web Programming” by Masatoshi Seki
Copyright © 2005. Published by Ohmsha, Ltd

Printed in the United States of America.
ISBN-13: 978-1-934356-93-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

8.2 Concurrency in rinda_eval

You can gain two benefits by splitting tasks into multiple processes. First, each task can have its own address space. Second (especially for MRI—the Matz Ruby Interpreter in Ruby), you can use the power of multicore CPUs. Let's see the latter benefit with some simple examples.

Rinda::rinda_eval and Thread

ruby-1.9 uses an OS-native thread that itself can make use of multicore CPUs. However, there is a limitation in which only one thread can run at a time, so you can't make use of multicore by just using Thread.

In the following example, we'll run a Fibonacci function as an example of a CPU-heavy task. Sure, you can run this code faster by using techniques such as memoization or by passing around an n-1 value, but we'll use a simple implementation because our intention here is to create a long-running task.

rinda_bench1.rb

```
require 'benchmark'
def fib(n)
  n < 2 ? n : fib(n - 2) + fib(n - 1)
end
def task(n)
  puts "fib(#{n}) = #{fib(n)}"
end

puts Benchmark.measure{
  [30, 30, 30].each{|x| task x}
}
```

First, let's call fibonacci three times with an argument of 30 and then measure the time it took to run.

```
$ ruby rinda_bench1.rb
fib(30) = 832040
fib(30) = 832040
fib(30) = 832040
  0.720000   0.000000   0.720000 (  0.737842)
```

Next, let's rewrite the code using Thread. I ran this on two-core machine (the result shouldn't change even if you run this on single-core machine).

rinda_bench2.rb

```
require 'benchmark'
def fib(n)
  n < 2 ? n : fib(n - 2) + fib(n - 1)
end
def task(n)
```

```
  puts "fib(#{n}) = #{fib(n)}"
end

puts Benchmark.measure{
  [30, 30, 30].map{|x| Thread.new{task x}}.map{|y| y.join}
}
```

```
$ ruby rinda_bench2.rb
fib(30) = 832040
fib(30) = 832040
fib(30) = 832040
  0.750000   0.010000   0.760000 (  0.767103)
```

The execution times were about the same. How disappointing. This is because of the limitation I mentioned earlier, in which a Ruby thread can run only one interpreter at a time. In Ruby 1.8, this is because Thread is implemented virtually at the user level (called *Green Thread*). Ruby 1.9 uses the native threading of the CPU, but it is restricted by the Global Interpreter Lock (GIL), and multiple threads can't run at the same time.

Next, let's rewrite this to the rinda_eval version. This time, we should be able to leverage the multicore.

rinda_bench3.rb

```
require 'benchmark'
require 'rinda/tuplespace'
require 'rinda/eval'
def fib(n)
  n < 2 ? n : fib(n - 2) + fib(n - 1)
end
def task(n)
  puts "fib(#{n}) = #{fib(n)}"
end

place = Rinda::TupleSpace.new
DRb.start_service

puts Benchmark.measure{
  [30, 30, 30].each {|x|
    Rinda::rinda_eval(place) {|ts| [:result, x, task(x)]}
  }.each {|x|
    place.take([:result, x, nil])
  }
}
```

```
$ ruby rinda_bench3.rb
fib(30) = 832040
fib(30) = 832040
fib(30) = 832040
  0.000000   0.000000   0.530000 (  0.477666)
```

The code became slightly bigger than the Thread version, but you should be able to experience a speed increase because this version creates a new process for each task and runs in multiple CPU cores.

By the way, you can work with multicores using Thread if you use JRuby, which runs on the Java Virtual Machine (JVM). Let's run the Thread example with JRuby.

```
$ jruby rinda_bench2.rb
fib(30) = 832040
fib(30) = 832040
fib(30) = 832040
  0.640000   0.000000   0.640000 (  0.492000)
```

The preceding code is an example of running the Thread version with JRuby. Notice the same speed increase as with the rinda_eval version. You can use just JRuby if your project allows it. If your Ruby environment depends on MRI, then rinda_eval can be a handy way to do concurrent computing, and you may want to add this into your toolbox.

By the way, there is a common misunderstanding about GIL. Some people may think that operations like read, write, and sleep will block other native threads and therefore Ruby threads can't switch operations from one thread to the other. The truth is that it is only the calling Ruby thread that gets blocked during the long system calls, and other Ruby threads will run without any problems (you need to be aware that there are actually some extension libraries that do block all the running threads). Therefore, Ruby threading is very effective for I/O and network operations that interact with external resources, such as web crawling.

Let's change the task from fibonacci to sleep. This swaps a CPU-intensive task with an external resource call.

rinda_bench4.rb

```
require 'benchmark'
def task(n)
  sleep(n * 0.1)
end
puts Benchmark.measure{
  [30,30,30].map{|x| Thread.new{task x}}.map{|y| y.join}
}
```

```
$ ruby rinda_bench4.rb
  0.010000   0.010000   0.020000 (  3.000341)
```

In the preceding example, we replaced fib(n) with sleep(n * 0.1). You can see that it finishes in three seconds. This will be the same if we replace fib with read or write. In the preceding example, multiple Ruby threads are dealing with multiple I/O, which is similar to how you handle multiple clients with one native thread in C with select and asynchronous read and write. Using Ruby Thread, you can easily write logic to receive chunks of packets from a TCP stream and return them to the higher layer. In fact, dRuby uses asynchronous I/O to handle multiple Ruby threads. Having said that, this model is useful for handling up to dozens of client connections with one process. If the number of clients increases to thousands or tens of thousands, then you may want to use other solutions, such as EventMachine.

Service with Rinda::rinda_eval

In the previous example, rinda_eval processed only one task at a time, but there is a better way. Not only can you communicate between processes one at a time through the end result of the rinda_eval block, but you also can directly communicate between parent and child processes, or even among child processes. This way, you can create a long-running service that behaves like the Actor model, explained in the next section.

Here's an example of a service to return a Fibonacci calculation:

rinda_eval_service.rb

```
require 'rinda/tuplespace'
require 'rinda/eval'

def fib(n)
  n < 2 ? n : fib(n - 2) + fib(n - 1)
end

def fib_daemon(place)
  Rinda::rinda_eval(place) do |ts|
    begin
      while true
        _, n = ts.take([:fib, Integer])
        ts.write([:fib_ans, n, fib(n)])
      end
      [:done]  # not reached
    rescue DRb::DRbConnError
      exit(0)
    end
  end
end

place = Rinda::TupleSpace.new
DRb.start_service

2.times { fib_daemon(place) }
```

```
[30, 20, 10, 30].each {|x|
  place.write([:fib, x])
}.each {|x|
  p place.take([:fib_ans, x, nil])
}
```

[:fib, Integer] represents a request for a new Fibonacci, and [:fib_ans, n, fib(n)] represents a response for the Fibonacci calculation result. The fib_daemon method generates a new process. Inside the rinda_eval block, a loop continues taking :fib and writing :fib_ans. Not only can the child processes return a single result, but they also keep processing multiple requests. If you treat :fib as an address and Integer as a message, then it looks similar to the *Actor model*.

Rinda::rinda_eval and the Actor Model

There is a concurrent computation model called the Actor model. The essence of this idea is that each process coordinates with one another by sending one-way messages. The primitive type of message passing goes only one way; it's not a "request and response" two-way cycle. When you send a message, you send an "address" and "message body" and send to the destination without caring about the state of the other end. The receiver takes messages one at a time when possible. You can avoid lots of shared resource–related problems in multithreaded programming if processes don't share objects or memory and each process reads its own message only when its own resource is not in a critical state. This exchange of messages in one direction is similar to Linda's process coordination model.

The key of the Actor model is the message passing and nonshared state. Erlang provides both functionalities as a pure Actor model, but implementations in most other languages are an afterthought. You can write it easily, even in Ruby. It's also easy to write in a message-passing style using a library, but it's difficult to create nonshared state.

If you really want to have nonshared state, you can leverage the power of the OS by just dividing actors into multiple processes. rinda_eval comes in handy in this situation because you can create real processes easily. You can have the entire copy of the object space of the parent process, such as the class definition and binding of the preprocessing state. At the same time, you can have completely nonshared space. You can use Rinda's tuplespace as message-passing middleware.

Looking Inside rinda_eval

So far, you've seen the power of rinda_eval; it's a handy way to create processes for concurrent computing. To see how this method is implemented, let's look inside the implementation of rinda_eval. [3]

```
require 'drb/drb'
require 'rinda/rinda'

module Rinda
  module_function
  def rinda_eval(ts)
    Thread.pass
    ts = DRbObject.new(ts) unless DRbObject === ts
    pid = fork do
      Thread.current['DRb'] = nil
      DRb.stop_service
      DRb.start_service
      place = TupleSpaceProxy.new(ts)
      tuple = yield(place)
      place.write(tuple) rescue nil
    end
    Process.detach(pid)
  end
end
```

Wow, it has fewer than twenty lines of code. The code looks complicated, but there are two key things in the code. It uses fork to generate a child process, and it passes a reference of TupleSpace into its child process.

fork

fork is a Unix system call that creates a new process by copying the memory space and resources of its parent process. Like Unix's fork, Ruby's fork method carries over the entire Ruby object space into the child process. This is useful to set up the initial state of the child process, by passing the state of the parent process right at the time of the fork method call. But how do you send information to a child process once fork is called? In traditional Unix programming, you use pipe or socketpair. Both functions use a stream between parent and child processes. rinda_eval uses TupleSpace to exchange information between processes.

This is a simple example of using fork:

fork.rb

```
result = 0
pid = fork do
```

3. https:/github.com/seki/MoreRinda/blob/master/lib/rinda/eval.rb

```
result += 1
end
Process::waitpid(pid)
p result
```

The result of the preceding example returns 0. The result variable is redefined in the child process, so it won't change the state of the parent process.

Let's change the code to use the tuplespace to exchange values:

rinda_fork.rb

```
require 'drb'
proc = Proc.new {|x| x + 1}
parent = Rinda::TupleSpace.new
DRb.start_service
child = DRbObject.new(parent)
result = 0
pid = fork do
 DRb.stop_service
 child.write([:result, proc[result]])
end
Process.detach(pid)
_, result = parent.take([:result, nil])
p result
```

The result of the preceding example returns 1. You can pass the logic of the calculation using Proc because a child process inherits the entire context of its parent process. The difference between the two is shown in Figure 43, *Difference between passing the result via fork and via TupleSpace*, on page 12. Note that you can use Proc inside the fork block, because the child process has the entire context of the parent process. To run Ruby's block inside a different process, you need to pass the entire binding of the object space, not just the statement you want to execute. If your statement doesn't depend on the external environment, you can just pass the statement as a string and execute Ruby's eval method.

Remember, we created a distributed factorial service in Figure 33, *Expressing the factorial request tuple and the result tuple as a service*, on page ?. The downside of the service is that you had to define the algorithm of the computation (factorial logic) on the server side in advance. Using fork and TupleSpace, you can define anything in a parent process, distribute the computation into its child processes, and then receive the result afterward. rinda_eval abstracted the logic into twenty lines of code. The downside of rinda_eval compared to the distributed factorial service is that you can't distribute to different machines, because rinda_eval depends on fork.

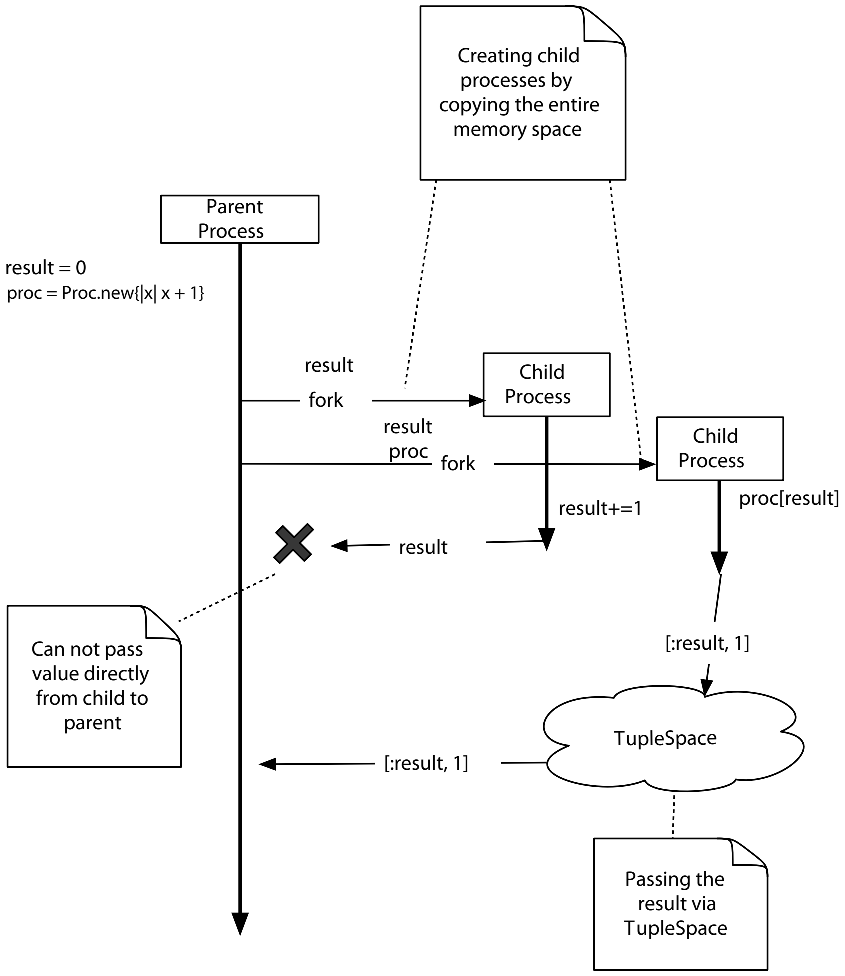
Creating child processes by copying the entire memory space
Parent Process
result = 0
proc = Proc.new{|x| x + 1}
result
fork
Child Process
result
proc
fork
Child Process
proc[result]
result+=1
result
Can not pass value directly from child to parent
[:result, 1]
TupleSpace
[:result, 1]
Passing the result via TupleSpace

Figure 43—Difference between passing the result via fork and via TupleSpace