WORKING *with* RUBY THREADS

Jesse Storimer

This book is dedicated to Sara, Inara, and the newest little one, who make it all worthwhile.

Chapter 7

How many threads is too many?

This question is relevant whether you're whipping up a quick script to scrape some websites, trying to speed up a long-running calculation, or tuning your multi-threaded webserver.

I hope you're not surprised, but the answer is: **well, that depends.**

The only way to be sure is to measure and compare. Try it out with 1 thread per CPU core, try it out with 5 threads per CPU core, compare the results, move forward.

However, there are some heuristics you can use to get started. Plus, if you're new to this, it's good to have a ballpark idea of what's sane. Is 100 threads too many? How about 10,000?

ALL the threads

How about we start by spawning ALL the threads? How many threads can we spawn?

```
1.upto(10_000) do |i|
  Thread.new { sleep }
  puts i
end
```

This code attempts to spawn 10,000 sleeping threads. If you run this on OSX you'll get about this far:

```
2042
2043
2044
2045
2046
ThreadError: can't create Thread (35)
```

On OSX, there's a hard limit on the number of threads that one process can spawn. On recent versions, that limit is somewhere around 2000 threads.

However, we were able to spawn at least that many threads without the system falling down around us. As mentioned earlier, spawning a thread is relatively cheap.

If I run this same bit of code on a Linux machine, I'm able to spawn 10,000 threads without blinking. So, **it's possible to spawn a lot of threads**, but you probably don't want to.

Context Switching

If you think about it for a minute, it should be clear why you don't want to spawn 10,000 threads on a 4-core CPU.

Even though each thread requires little memory overhead, there is overhead for the thread scheduler to manage those threads. If you have only have 4 cores, then only 4 threads can be executing instructions at any given time. You may have some threads blocking on IO, but that still leaves a lot of threads idle a lot of the time, requiring overhead to manage them.

But even in the face of increased overhead for the thread scheduler, it does sometimes make sense to spawn more threads than the number of CPU cores. Let's review the difference between IO-bound code and CPU-bound code.

IO-bound

Your code is said to be IO-bound if it is bound by the IO capabilities of your machine. Let me try that again with an example.

If your code were doing web requests to external services, and your machine magically got a faster network connection, your program would likely be sped up as well.

If your code were doing a lot of reading and writing to disk, and you got a new hard drive that could seek faster, your code would likely be sped up as well.

These are examples of IO-bound code. In these situations, your code typically spends some time waiting for a response from some IO device, and the results aren't immediate.

In this case, it *does* make sense to spawn more threads than CPU cores.

If you need to make 10 web requests, you're probably best off to make them all in parallel threads, rather than spawning 4 threads, making 4 requests, waiting, then making 4 more requests. You want to minimize that idle time spent waiting.

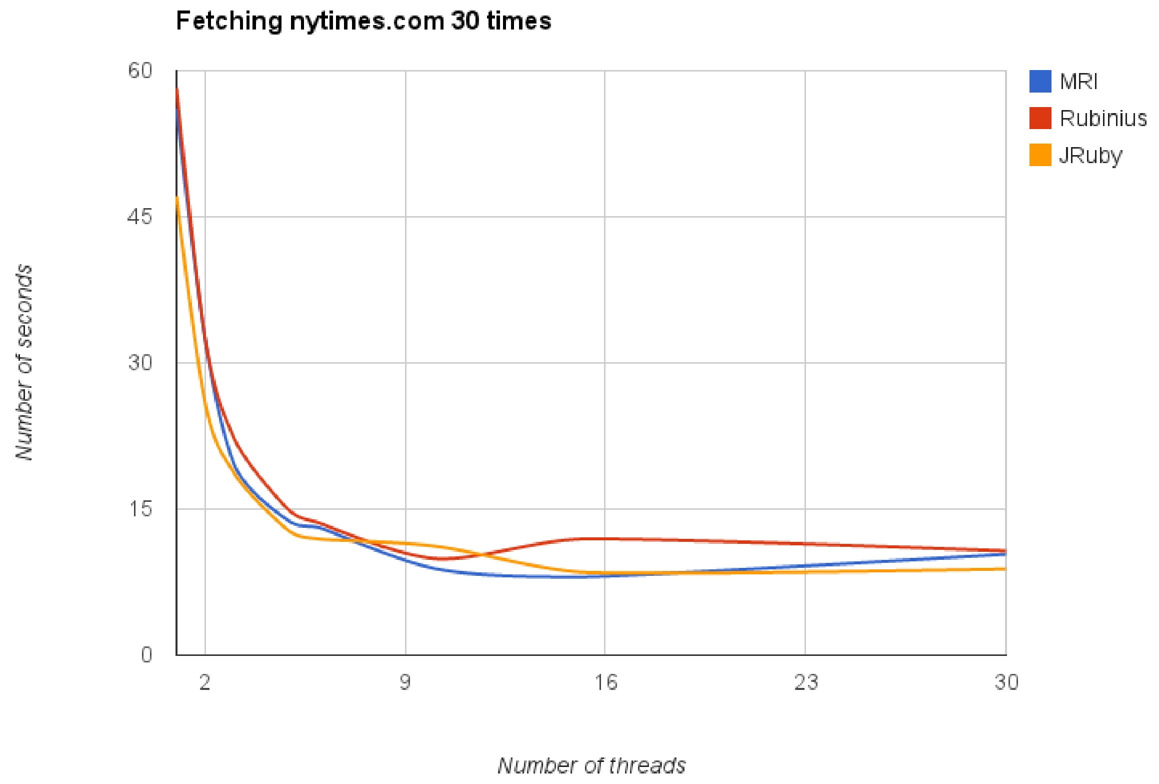
Let's make this concrete with some code.

In this example, the task is to fetch `http://nytimes.com` 30 times. I tested this using different numbers of threads. With one thread, 30 fetches will be performed serially. With two threads, each thread will need to perform 15 fetches, etc.

I tested different numbers of threads in order to find the sweet spot. Make no mistake, **there will be a sweet spot between utilizing available resources and context switching overhead.**

```ruby
# ./code/benchmarks/io_bound.rb

require 'benchmark'
require 'open-uri'

URL = 'http://www.nytimes.com/'
ITERATIONS = 30

def fetch_url(thread_count)
  threads = []

  thread_count.times do
    threads << Thread.new do
      fetches_per_thread = ITERATIONS / thread_count
      fetches_per_thread.times do
        open(URL)
      end
    end
  end

  threads.each(&:join)
end

Benchmark.bm(20) do |bm|
  [1, 2, 3, 5, 6, 10, 15, 30].each do |thread_count|
    bm.report("with #{thread_count} threads") do
      fetch_url(thread_count)
    end
  end
end
```

I plotted the results across the studied Ruby implementations to get this graph illustrating the results:



I ran this benchmark on my 4-core Macbook Pro.

As expected, all of the implementations exhibited similar behaviour with respect to concurrent IO.

The sweet spot for code that is fully IO-bound is right around 10 threads on my machine, even though I tested up to 30 threads. Using more than 10 threads no longer improves performance in this case, just uses more resources. For some cases, it actually made performance worse.

If the latency of the IO operations were higher, I might need more threads to hit that sweet spot, because more threads would be blocked waiting for responses more of the time. Conversely, if the latency were lower, I might need less threads to hit the sweet spot because there would be less time spent waiting, each thread would be freed up more quickly.

Finding the sweet spot is really important. Once you've done this a few times, you can probably start to make good guesses about how many threads to use, but the sweet spot will be different with different IO workloads, or different hardware.

CPU-bound

The flip side of IO-bound code is CPU-bound code. Your code is CPU-bound if its execution is bound by the CPU capabilities of your machine. Let's try another example.

If your code needed to calculate millions of cryptographic hashes, or perform really complex mathematical calculations, these things would be bound by how much throughput your CPU could muster in terms of CPU instructions.

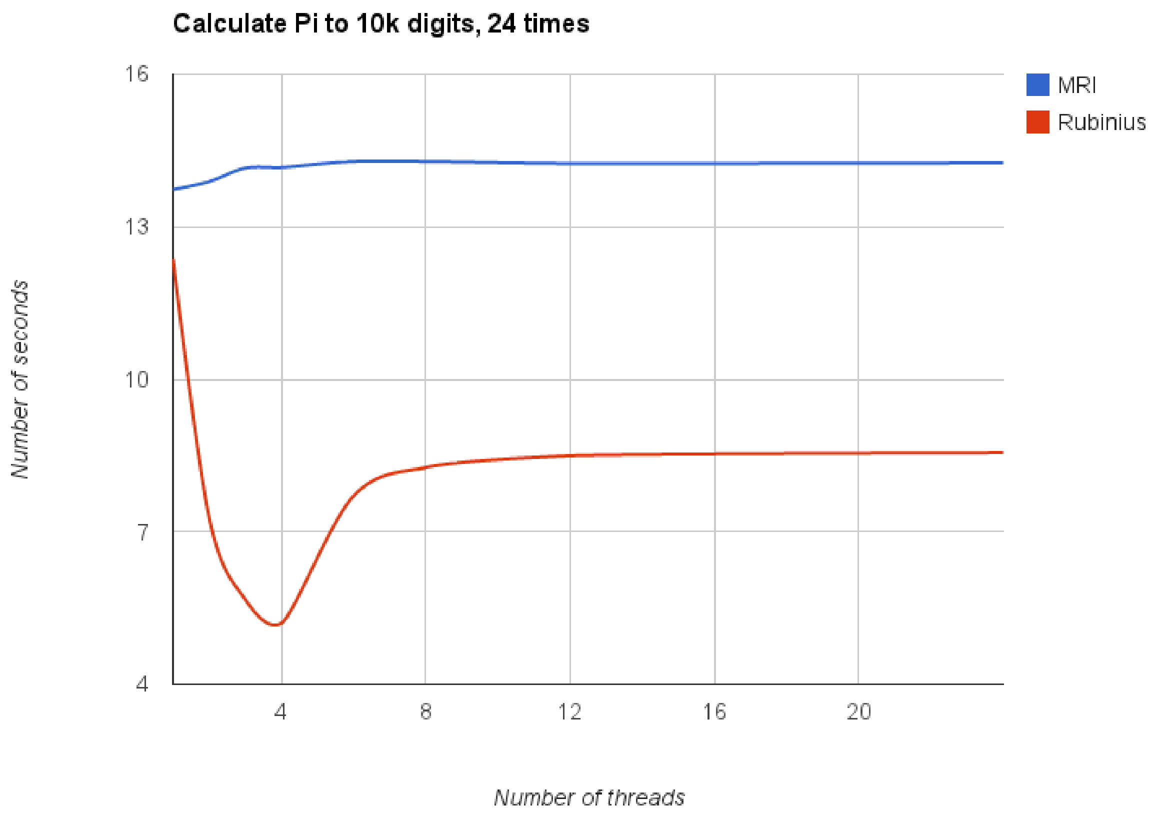
If you upgraded to a CPU with a faster clock speed, your code would be able to do these calculations in a shorter time period.

Let's re-use the example from the last chapter that calculated digits of pi. That's certainly a CPU-bound bit of code. Here's the benchmark code:

```ruby
# ./code/benchmarks/cpu_bound.rb

require 'benchmark'

require 'bigdecimal'
require 'bigdecimal/math'

DIGITS = 10_000
ITERATIONS = 24

def calculate_pi(thread_count)
  threads = []

  thread_count.times do
    threads << Thread.new do
      iterations_per_thread = ITERATIONS / thread_count

      iterations_per_thread.times do
        BigMath.PI(DIGITS)
      end
    end
  end

  threads.each(&:join)
```

```
end

Benchmark.bm(20) do |bm|
  [1, 2, 3, 4, 6, 8, 12, 24].each do |thread_count|
    bm.report("with #{thread_count} threads") do
      calculate_pi(thread_count)
    end
  end
end
```

This graph illustrates the results:


Calculate Pi to 10k digits, 24 times
16
13
10
7
4
4
8
12
16
20
Number of seconds
Number of threads
🟦 MRI
🟥 Rubinius

Note: JRuby seems to have an issue with calculating that many digits of pi. Its results weren't comparable, so I left them out.

This graph has a bit of a different shape than the last one. Again, the absolute values aren't important here; the shape is the important part.

For MRI, there's no curve this time. Performance isn't impacted with the introduction of more threads. This is a direct result of the GIL. Since the GIL is a lock around the execution of Ruby code, and in this case we're benchmarking execution of Ruby code, more threads has no effect.

For Rubinius, the the curve was lowest when running with 4 threads. I ran this example locally on my 4-core CPU. That's no coincidence.

CPU-bound code is inherently bound by the rate at which the CPU can execute instructions. So when I have my code running in parallel across 4 threads, I'm utilizing all of the power of my 4-core CPU without any overhead.

Once you introduce more than 4 threads, it's still possible for you to fully utilize all of your CPU resources, but now there's more overhead. That's why we see the curve going back up, the thread scheduler is incurring overhead for each extra thread.

The takeaway from all this is that each thread you spawn incurs overhead to manage it. **More threads isn't always necessarily faster.** On the other hand, introducing more threads improved performance in these two examples by anywhere between 100% and 600%. Finding that sweet spot is certainly worth it.

So... how many should you use for your code?

I showed some heuristics for code that is fully IO-bound or fully CPU-bound. In reality, your application is probably not so clear cut. Your app may be IO-bound in some places, and CPU-bound in other places. Your app may not be bound by CPU or IO, it may be memory-bound, or simply not maximizing resources in any way.

A Rails application is a prime example of this. Between communicating with the database, communicating with the client, and calling out to external services, there's lots of chances for it to be IO-bound. On the other hand, there are things that will tax the CPU, like rendering HTML templates, or transforming data to JSON documents.

So, as I said at the beginning of this chapter, **the only way to a surefire answer is to measure.** Run your code with different thread counts, measure the results, and then decide. Without measuring, you may never find the 'right' answer.