
The other day

Reverse Sort

```ruby
array = (1..1_000).to_a

array.sort do |item, other|
  other <=> item
end
```

CRuby vs JRuby

```
$ ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-linux]
$ time ruby scripts/sort.rb
real  0m0.151s
user  0m0.052s
sys   0m0.016s
$ asdf local ruby jruby-9.1.8.0
$ ruby -v
jruby 9.1.8.0 (2.3.1) 2017-03-06 90fc7ab OpenJDK 64-Bit
Server VM 25.121-b13 on 1.8.0_121-8u121-b13-
0ubuntu1.16.04.2-b13 +jit [linux-x86_64]
$ time ruby scripts/sort.rb
real  0m3.468s
user  0m8.384s
sys   0m0.232s
```

Something called benchmark!

```ruby
require 'benchmark'

Benchmark.bm do |bench|
  bench.report do
    array = (1..1_000).to_a

    array.sort do |item, other|
      other <=> item
    end
  end
end
```

```
$ asdf local ruby 2.4.1
$ ruby scripts/sort_bm_benchmark.rb
       user     system      total        real
   0.000000   0.000000   0.000000 (  0.000381)
$ asdf local ruby jruby-9.1.8.0
$ ruby scripts/sort_bm_benchmark.rb
       user     system      total        real
   0.030000   0.000000   0.030000 (  0.004920)
```


Success!

The End?


I HAVE NO
IDEA WHAT
I'M DOING

- Way too few samples
- Realistic data/multiple inputs?
- No warmup
- Non production environment
- Does creating the array matter?
- Is reverse sorting really the bottle neck?
- Setup information
- Running on battery
- Lots of applications running

A proper library

```ruby
require 'benchmark/ips'
ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "reverse sort" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end
end
```

```ruby
require 'benchmark/ips'

ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "reverse sort" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end
end
```

```ruby
require 'benchmark/ips'
ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "reverse sort" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end
end
```

```
$ asdf local ruby 2.4.1
$ ruby benchmark/reverse_sort_block.rb
Warming up
--------------------------------------
        reverse sort   169.000  i/100ms
Calculating
--------------------------------------
        reverse sort      1.688k (± 1.1%)
i/s -      8.450k in   5.005956s
$ asdf local ruby jruby-9.1.8.0
$ ruby benchmark/reverse_sort_block.rb
Warming up
--------------------------------------
        reverse sort   168.000  i/100ms
Calculating
--------------------------------------
        reverse sort      2.401k (± 4.0%)
i/s -     12.096k in   5.046038s
```

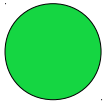
```
$ asdf local ruby 2.4.1
$ ruby benchmark/reverse_sort_block.rb
Warming up
--------------------------------------
        reverse sort   169.000  i/100ms
Calculating
--------------------------------------
        reverse sort      1.688k (± 1.1%)
i/s -      8.450k in   5.005956s
$ asdf local ruby jruby-9.1.8.0
$ ruby benchmark/reverse_sort_block.rb
Warming up
--------------------------------------
        reverse sort   168.000  i/100ms
Calculating
--------------------------------------
        reverse sort      2.401k (± 4.0%)
i/s -     12.096k in   5.046038s
```

```
$ asdf local ruby 2.4.1
$ ruby benchmark/reverse_sort_block.rb
Warming up
--------------------------------------
        reverse sort   169.000  i/100ms
Calculating
--------------------------------------
        reverse sort      1.688k (± 1.1%)
i/s -      8.450k in   5.005956s
$ asdf local ruby jruby-9.1.8.0
$ ruby benchmark/reverse_sort_block.rb
Warming up
--------------------------------------
        reverse sort   168.000  i/100ms
Calculating
--------------------------------------
        reverse sort      2.401k (± 4.0%)
i/s -     12.096k in   5.046038s
```

```ruby
require 'benchmark/ips'
ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "sort with block" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end

  bm.report ".sort.reverse" do
    ARRAY.sort.reverse
  end

  bm.report "sort_by -value" do
    ARRAY.sort_by { |value| -value }
  end

  bm.compare!
end
```

What's fastest?

```ruby
require 'benchmark/ips'
ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "sort with block" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end

  bm.report ".sort.reverse" do
    ARRAY.sort.reverse
  end

  bm.report "sort_by -value" do
    ARRAY.sort_by { |value| -value }
  end

  bm.compare!
end
```



What's fastest?

```ruby
require 'benchmark/ips'
ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "sort with block" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end

  bm.report ".sort.reverse" do
    ARRAY.sort.reverse
  end

  bm.report "sort_by -value" do
    ARRAY.sort_by { |value| -value }
  end

  bm.compare!
end
```

What's fastest?

What's fastest?

```
require 'benchmark/ips'
ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "sort with block" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end

  bm.report ".sort.reverse" do
    ARRAY.sort.reverse
  end

  bm.report "sort_by -value" do
    ARRAY.sort_by { |value| -value }
  end

  bm.compare!
end
```

```ruby
require 'benchmark/ips'
ARRAY = (1..1_000).to_a.shuffle

Benchmark.ips do |bm|
  bm.report "sort with block" do
    ARRAY.sort do |item, other|
      other <=> item
    end
  end

  bm.report ".sort.reverse" do
    ARRAY.sort.reverse
  end

  bm.report "sort_by -value" do
    ARRAY.sort_by { |value| -value }
  end

  bm.compare!
end
```

Compare!

```
$ asdf local ruby 2.4.1
$ ruby benchmark/reverse_sort.rb
Warming up --------------------------------------
     sort with block   166.000  i/100ms
       .sort.reverse     1.143k i/100ms
      sort_by -value   236.000  i/100ms
Calculating -------------------------------------
     sort with block      1.671k (± 1.9%) i/s
       .sort.reverse     11.539k (± 1.7%) i/s
      sort_by -value      2.373k (± 0.8%) i/s

Comparison:
       .sort.reverse:    11539.1 i/s
      sort_by -value:     2372.5 i/s - 4.86x  slower
     sort with block:     1671.0 i/s - 6.91x  slower
```

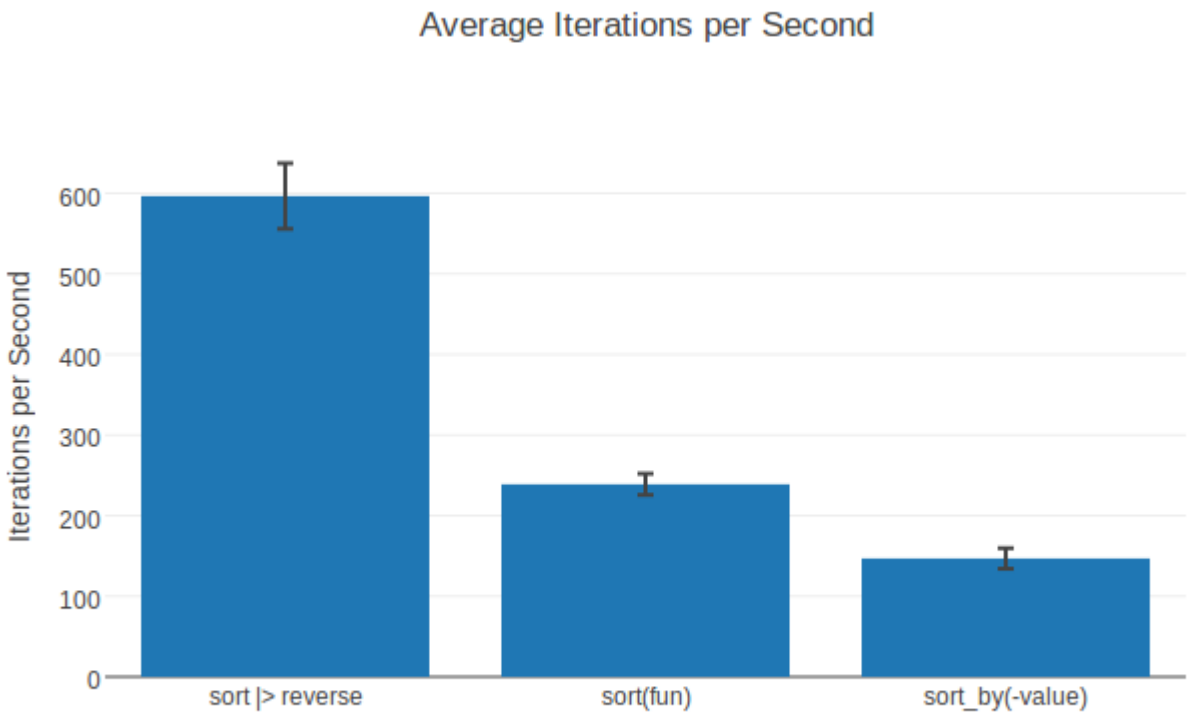
```
$ asdf local ruby 2.4.1
$ ruby benchmark/reverse_sort.rb
Warming up --------------------------------------
     sort with block   166.000  i/100ms
       .sort.reverse     1.143k i/100ms
      sort_by -value   236.000  i/100ms
Calculating -------------------------------------
     sort with block      1.671k (± 1.9%) i/s
       .sort.reverse     11.539k (± 1.7%) i/s
      sort_by -value      2.373k (± 0.8%) i/s

Comparison:
       .sort.reverse:    11539.1 i/s
      sort_by -value:     2372.5 i/s - 4.86x  slower
     sort with block:     1671.0 i/s - 6.91x  slower
```

```
$ asdf local ruby jruby-9.1.8.0
$ ruby benchmark/reverse_sort.rb
Warming up --------------------------------------
     sort with block   157.000  i/100ms
       .sort.reverse   656.000  i/100ms
      sort_by -value   305.000  i/100ms
Calculating -------------------------------------
     sort with block      2.317k (± 7.4%) i/s
       .sort.reverse      7.288k (± 1.7%) i/s
      sort_by -value      3.180k (± 1.8%) i/s

Comparison:
       .sort.reverse:     7288.0 i/s
      sort_by -value:     3180.1 i/s - 2.29x  slower
     sort with block:     2317.1 i/s - 3.15x  slower
```

benchee

```
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```

benchee




```
Name                          ips        average  deviation         median
sort |> reverse            596.54        1.68 ms     ±6.83%        1.65 ms
sort(fun)                  238.88        4.19 ms     ±5.53%        4.14 ms
sort_by(-value)            146.86        6.81 ms     ±8.68%        6.59 ms

Comparison:
sort |> reverse            596.54
sort(fun)                  238.88 - 2.50x slower
sort_by(-value)            146.86 - 4.06x slower
```

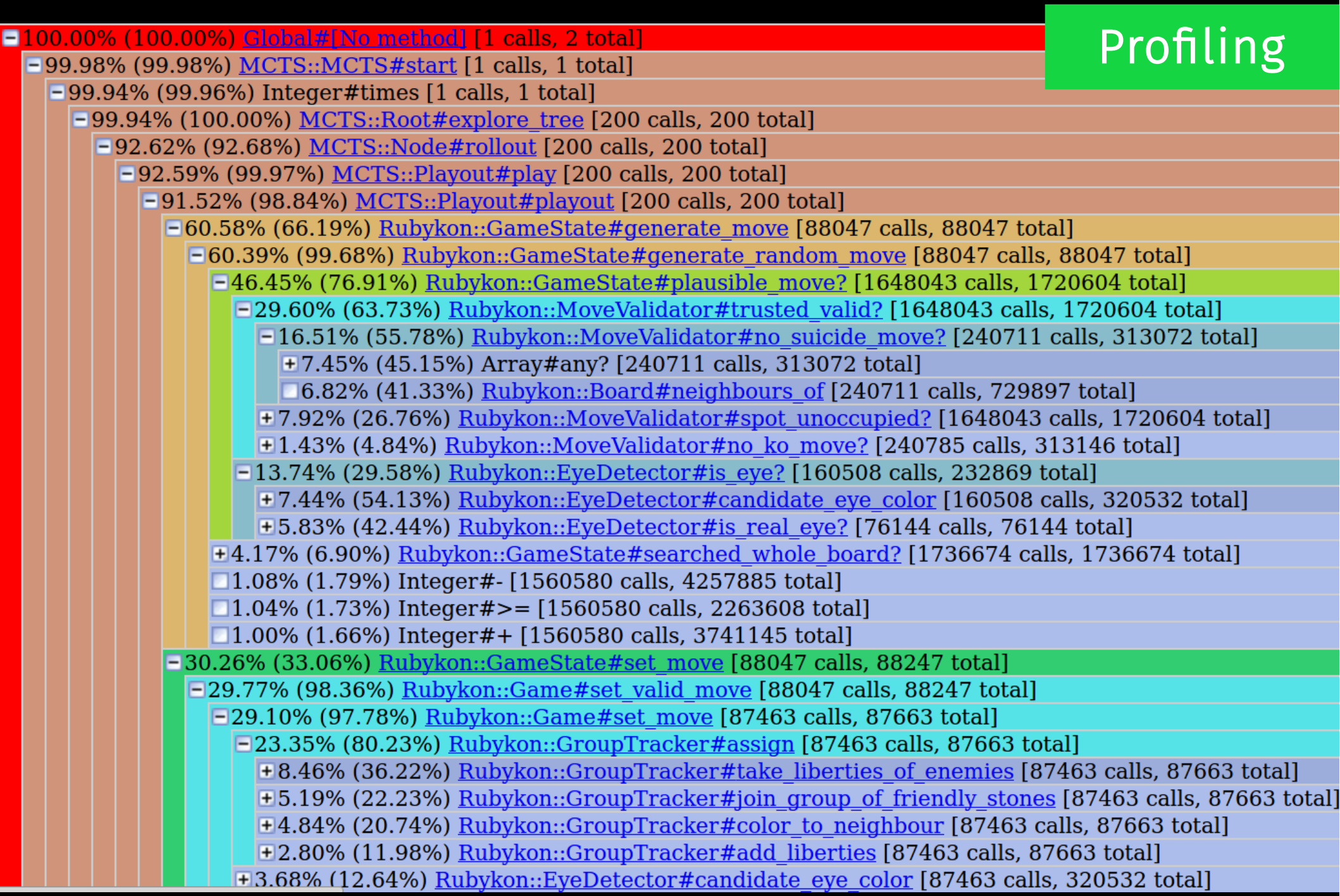
How fast is it really?

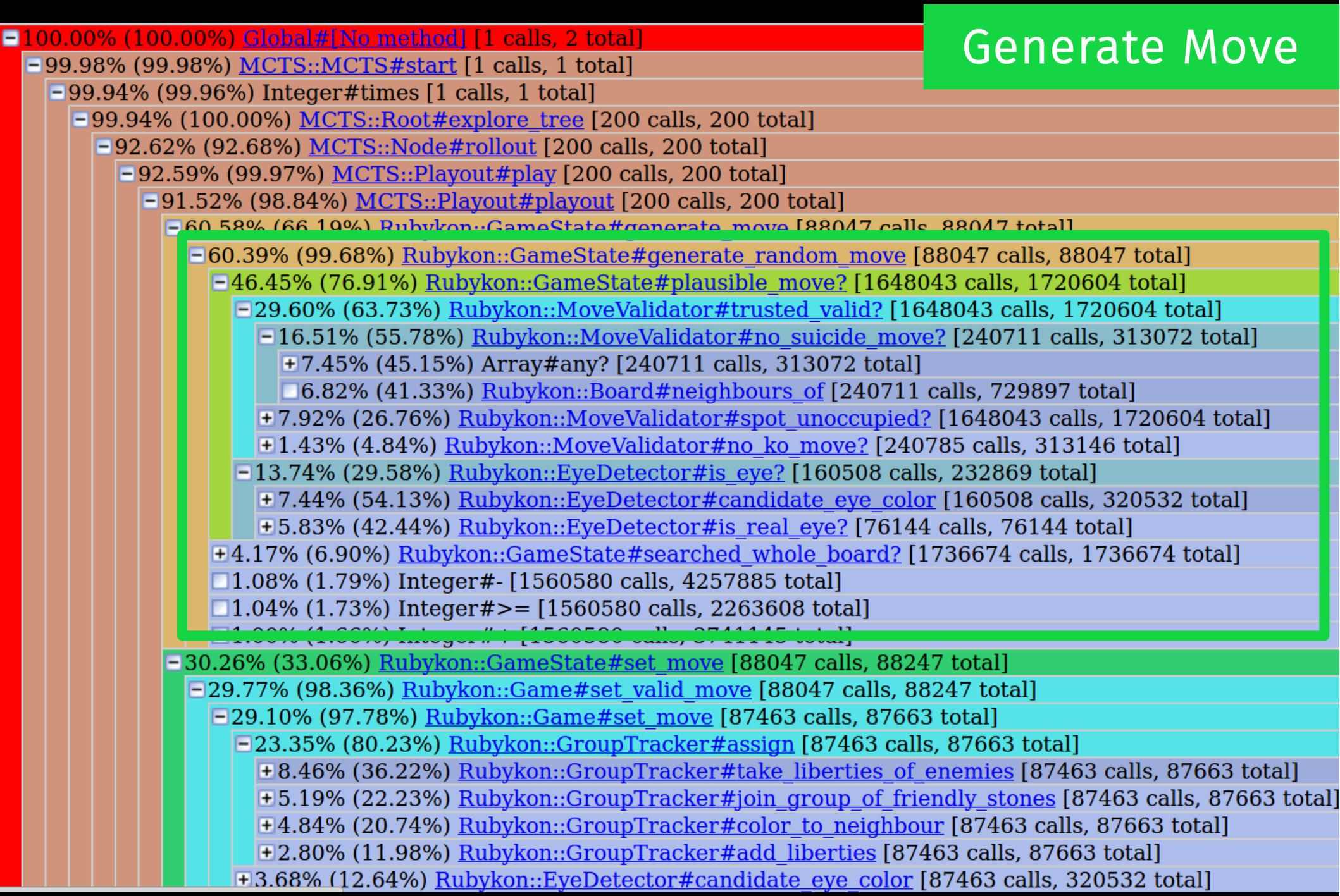
Benchmarking in Practice

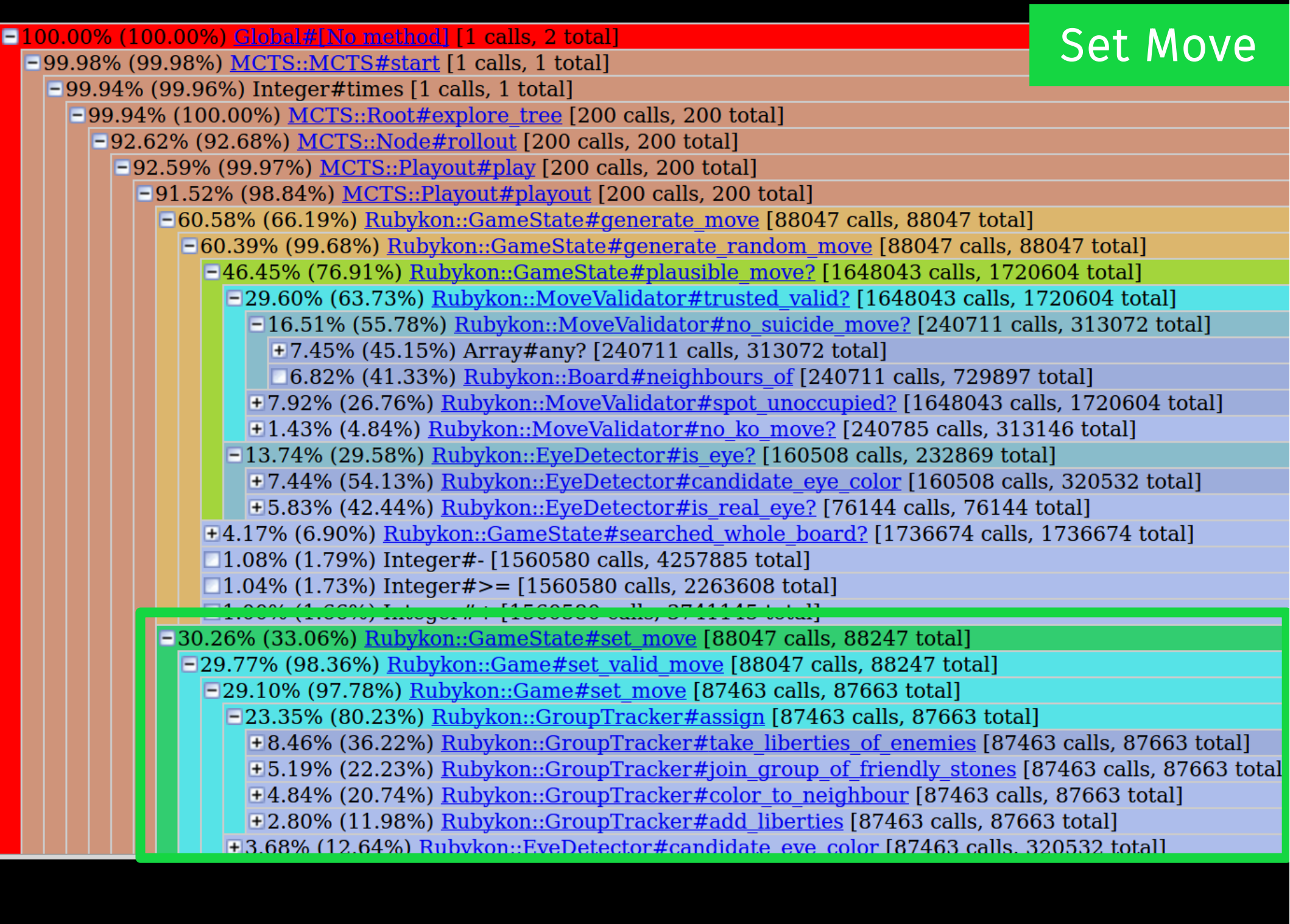
Tobias Pfeiffer
@PragTob
pragtob.info

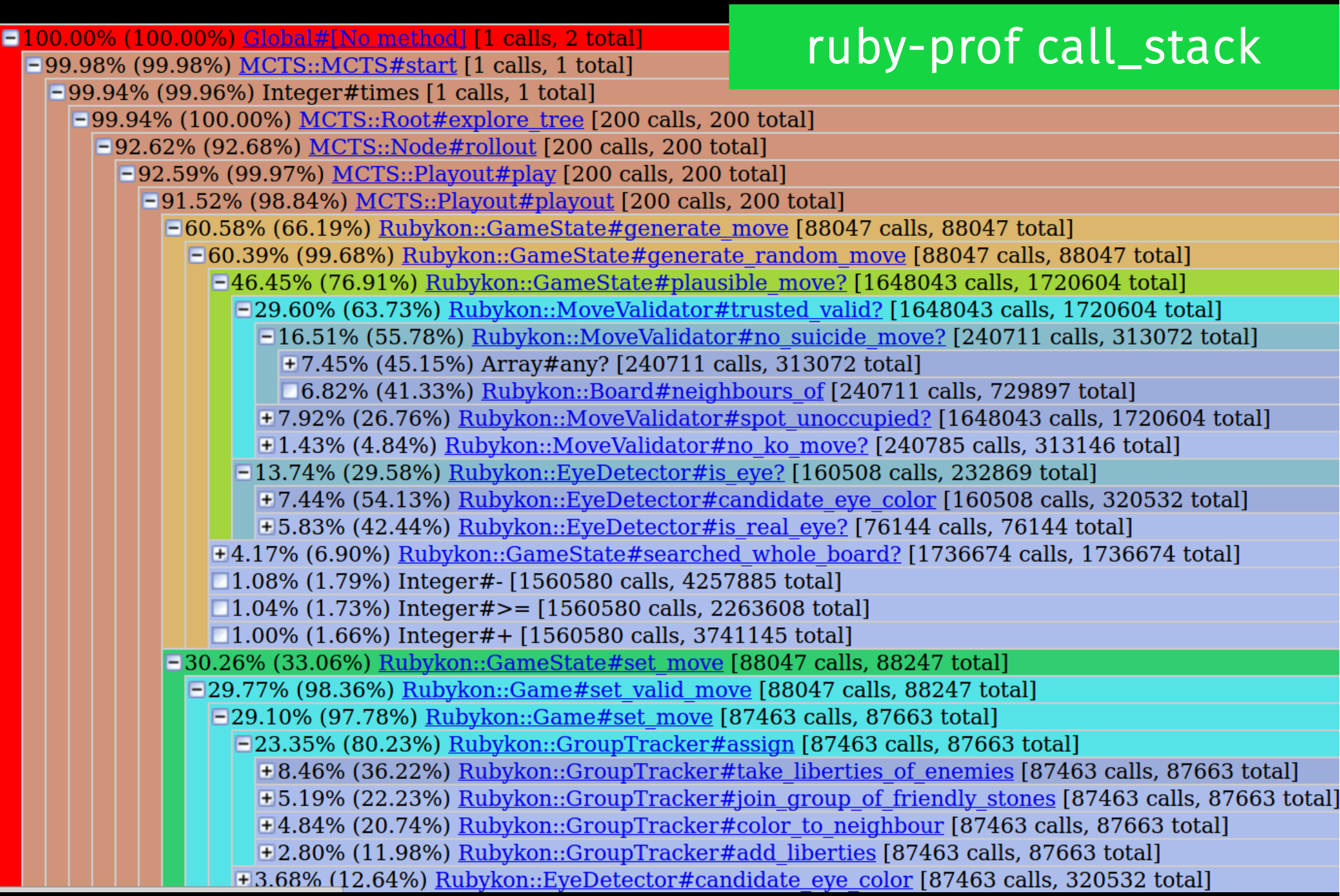
Concept vs Tool Usage

Profiling vs. Benchmarking


Profiling
100.00% (100.00%) Global#[No method] [1 calls, 2 total]
99.98% (99.98%) MCTS::MCTS#start [1 calls, 1 total]
99.94% (99.96%) Integer#times [1 calls, 1 total]
99.94% (100.00%) MCTS::Root#explore_tree [200 calls, 200 total]
92.62% (92.68%) MCTS::Node#rollout [200 calls, 200 total]
92.59% (99.97%) MCTS::Playout#play [200 calls, 200 total]
91.52% (98.84%) MCTS::Playout#playout [200 calls, 200 total]
60.58% (66.19%) Rubykon::GameState#generate_move [88047 calls, 88047 total]
60.39% (99.68%) Rubykon::GameState#generate_random_move [88047 calls, 88047 total]
46.45% (76.91%) Rubykon::GameState#plausible_move? [1648043 calls, 1720604 total]
29.60% (63.73%) Rubykon::MoveValidator#trusted_valid? [1648043 calls, 1720604 total]
16.51% (55.78%) Rubykon::MoveValidator#no_suicide_move? [240711 calls, 313072 total]
7.45% (45.15%) Array#any? [240711 calls, 313072 total]
6.82% (41.33%) Rubykon::Board#neighbours_of [240711 calls, 729897 total]
7.92% (26.76%) Rubykon::MoveValidator#spot_unoccupied? [1648043 calls, 1720604 total]
1.43% (4.84%) Rubykon::MoveValidator#no_ko_move? [240785 calls, 313146 total]
13.74% (29.58%) Rubykon::EyeDetector#is_eye? [160508 calls, 232869 total]
7.44% (54.13%) Rubykon::EyeDetector#candidate_eye_color [160508 calls, 320532 total]
5.83% (42.44%) Rubykon::EyeDetector#is_real_eye? [76144 calls, 76144 total]
4.17% (6.90%) Rubykon::GameState#searched_whole_board? [1736674 calls, 1736674 total]
1.08% (1.79%) Integer#- [1560580 calls, 4257885 total]
1.04% (1.73%) Integer#>= [1560580 calls, 2263608 total]
1.00% (1.66%) Integer#+ [1560580 calls, 3741145 total]
30.26% (33.06%) Rubykon::GameState#set_move [88047 calls, 88247 total]
29.77% (98.36%) Rubykon::Game#set_valid_move [88047 calls, 88247 total]
29.10% (97.78%) Rubykon::Game#set_move [87463 calls, 87663 total]
23.35% (80.23%) Rubykon::GroupTracker#assign [87463 calls, 87663 total]
8.46% (36.22%) Rubykon::GroupTracker#take_liberties_of_enemies [87463 calls, 87663 total]
5.19% (22.23%) Rubykon::GroupTracker#join_group_of_friendly_stones [87463 calls, 87663 total]
4.84% (20.74%) Rubykon::GroupTracker#color_to_neighbour [87463 calls, 87663 total]
2.80% (11.98%) Rubykon::GroupTracker#add_liberties [87463 calls, 87663 total]
3.68% (12.64%) Rubykon::EyeDetector#candidate_eye_color [87463 calls, 320532 total]


Generate Move
100.00% (100.00%) Global#[No method] [1 calls, 2 total]
99.98% (99.98%) MCTS::MCTS#start [1 calls, 1 total]
99.94% (99.96%) Integer#times [1 calls, 1 total]
99.94% (100.00%) MCTS::Root#explore_tree [200 calls, 200 total]
92.62% (92.68%) MCTS::Node#rollout [200 calls, 200 total]
92.59% (99.97%) MCTS::Playout#play [200 calls, 200 total]
91.52% (98.84%) MCTS::Playout#playout [200 calls, 200 total]
60.39% (99.68%) Rubykon::GameState#generate_random_move [88047 calls, 88047 total]
46.45% (76.91%) Rubykon::GameState#plausible_move? [1648043 calls, 1720604 total]
29.60% (63.73%) Rubykon::MoveValidator#trusted_valid? [1648043 calls, 1720604 total]
16.51% (55.78%) Rubykon::MoveValidator#no_suicide_move? [240711 calls, 313072 total]
7.45% (45.15%) Array#any? [240711 calls, 313072 total]
6.82% (41.33%) Rubykon::Board#neighbours_of [240711 calls, 729897 total]
7.92% (26.76%) Rubykon::MoveValidator#spot_unoccupied? [1648043 calls, 1720604 total]
1.43% (4.84%) Rubykon::MoveValidator#no_ko_move? [240785 calls, 313146 total]
13.74% (29.58%) Rubykon::EyeDetector#is_eye? [160508 calls, 232869 total]
7.44% (54.13%) Rubykon::EyeDetector#candidate_eye_color [160508 calls, 320532 total]
5.83% (42.44%) Rubykon::EyeDetector#is_real_eye? [76144 calls, 76144 total]
4.17% (6.90%) Rubykon::GameState#searched_whole_board? [1736674 calls, 1736674 total]
1.08% (1.79%) Integer#- [1560580 calls, 4257885 total]
1.04% (1.73%) Integer#>= [1560580 calls, 2263608 total]
30.26% (33.06%) Rubykon::GameState#set_move [88047 calls, 88247 total]
29.77% (98.36%) Rubykon::Game#set_valid_move [88047 calls, 88247 total]
29.10% (97.78%) Rubykon::Game#set_move [87463 calls, 87663 total]
23.35% (80.23%) Rubykon::GroupTracker#assign [87463 calls, 87663 total]
8.46% (36.22%) Rubykon::GroupTracker#take_liberties_of_enemies [87463 calls, 87663 total]
5.19% (22.23%) Rubykon::GroupTracker#join_group_of_friendly_stones [87463 calls, 87663 total]
4.84% (20.74%) Rubykon::GroupTracker#color_to_neighbour [87463 calls, 87663 total]
2.80% (11.98%) Rubykon::GroupTracker#add_liberties [87463 calls, 87663 total]
3.68% (12.64%) Rubykon::EyeDetector#candidate_eye_color [87463 calls, 320532 total]


Set Move
100.00% (100.00%) Global#[No method] [1 calls, 2 total]
99.98% (99.98%) MCTS::MCTS#start [1 calls, 1 total]
99.94% (99.96%) Integer#times [1 calls, 1 total]
99.94% (100.00%) MCTS::Root#explore_tree [200 calls, 200 total]
92.62% (92.68%) MCTS::Node#rollout [200 calls, 200 total]
92.59% (99.97%) MCTS::Playout#play [200 calls, 200 total]
91.52% (98.84%) MCTS::Playout#playout [200 calls, 200 total]
60.58% (66.19%) Rubykon::GameState#generate_move [88047 calls, 88047 total]
60.39% (99.68%) Rubykon::GameState#generate_random_move [88047 calls, 88047 total]
46.45% (76.91%) Rubykon::GameState#plausible_move? [1648043 calls, 1720604 total]
29.60% (63.73%) Rubykon::MoveValidator#trusted_valid? [1648043 calls, 1720604 total]
16.51% (55.78%) Rubykon::MoveValidator#no_suicide_move? [240711 calls, 313072 total]
7.45% (45.15%) Array#any? [240711 calls, 313072 total]
6.82% (41.33%) Rubykon::Board#neighbours_of [240711 calls, 729897 total]
7.92% (26.76%) Rubykon::MoveValidator#spot_unoccupied? [1648043 calls, 1720604 total]
1.43% (4.84%) Rubykon::MoveValidator#no_ko_move? [240785 calls, 313146 total]
13.74% (29.58%) Rubykon::EyeDetector#is_eye? [160508 calls, 232869 total]
7.44% (54.13%) Rubykon::EyeDetector#candidate_eye_color [160508 calls, 320532 total]
5.83% (42.44%) Rubykon::EyeDetector#is_real_eye? [76144 calls, 76144 total]
4.17% (6.90%) Rubykon::GameState#searched_whole_board? [1736674 calls, 1736674 total]
1.08% (1.79%) Integer#- [1560580 calls, 4257885 total]
1.04% (1.73%) Integer#>= [1560580 calls, 2263608 total]
30.26% (33.06%) Rubykon::GameState#set_move [88047 calls, 88247 total]
29.77% (98.36%) Rubykon::Game#set_valid_move [88047 calls, 88247 total]
29.10% (97.78%) Rubykon::Game#set_move [87463 calls, 87663 total]
23.35% (80.23%) Rubykon::GroupTracker#assign [87463 calls, 87663 total]
8.46% (36.22%) Rubykon::GroupTracker#take_liberties_of_enemies [87463 calls, 87663 total]
5.19% (22.23%) Rubykon::GroupTracker#join_group_of_friendly_stones [87463 calls, 87663 total
4.84% (20.74%) Rubykon::GroupTracker#color_to_neighbour [87463 calls, 87663 total]
2.80% (11.98%) Rubykon::GroupTracker#add_liberties [87463 calls, 87663 total]
3.68% (12.64%) Rubykon::EyeDetector#candidate_eye_color [87463 calls, 320532 total]


ruby-prof call_stack
100.00% (100.00%) Global#[No method] [1 calls, 2 total]
99.98% (99.98%) MCTS::MCTS#start [1 calls, 1 total]
99.94% (99.96%) Integer#times [1 calls, 1 total]
99.94% (100.00%) MCTS::Root#explore_tree [200 calls, 200 total]
92.62% (92.68%) MCTS::Node#rollout [200 calls, 200 total]
92.59% (99.97%) MCTS::Playout#play [200 calls, 200 total]
91.52% (98.84%) MCTS::Playout#playout [200 calls, 200 total]
60.58% (66.19%) Rubykon::GameState#generate_move [88047 calls, 88047 total]
60.39% (99.68%) Rubykon::GameState#generate_random_move [88047 calls, 88047 total]
46.45% (76.91%) Rubykon::GameState#plausible_move? [1648043 calls, 1720604 total]
29.60% (63.73%) Rubykon::MoveValidator#trusted_valid? [1648043 calls, 1720604 total]
16.51% (55.78%) Rubykon::MoveValidator#no_suicide_move? [240711 calls, 313072 total]
7.45% (45.15%) Array#any? [240711 calls, 313072 total]
6.82% (41.33%) Rubykon::Board#neighbours_of [240711 calls, 729897 total]
7.92% (26.76%) Rubykon::MoveValidator#spot_unoccupied? [1648043 calls, 1720604 total]
1.43% (4.84%) Rubykon::MoveValidator#no_ko_move? [240785 calls, 313146 total]
13.74% (29.58%) Rubykon::EyeDetector#is_eye? [160508 calls, 232869 total]
7.44% (54.13%) Rubykon::EyeDetector#candidate_eye_color [160508 calls, 320532 total]
5.83% (42.44%) Rubykon::EyeDetector#is_real_eye? [76144 calls, 76144 total]
4.17% (6.90%) Rubykon::GameState#searched_whole_board? [1736674 calls, 1736674 total]
1.08% (1.79%) Integer#- [1560580 calls, 4257885 total]
1.04% (1.73%) Integer#>= [1560580 calls, 2263608 total]
1.00% (1.66%) Integer#+ [1560580 calls, 3741145 total]
30.26% (33.06%) Rubykon::GameState#set_move [88047 calls, 88247 total]
29.77% (98.36%) Rubykon::Game#set_valid_move [88047 calls, 88247 total]
29.10% (97.78%) Rubykon::Game#set_move [87463 calls, 87663 total]
23.35% (80.23%) Rubykon::GroupTracker#assign [87463 calls, 87663 total]
8.46% (36.22%) Rubykon::GroupTracker#take_liberties_of_enemies [87463 calls, 87663 total]
5.19% (22.23%) Rubykon::GroupTracker#join_group_of_friendly_stones [87463 calls, 87663 total]
4.84% (20.74%) Rubykon::GroupTracker#color_to_neighbour [87463 calls, 87663 total]
2.80% (11.98%) Rubykon::GroupTracker#add_liberties [87463 calls, 87663 total]
3.68% (12.64%) Rubykon::EyeDetector#candidate_eye_color [87463 calls, 320532 total]

Application Performance Monitoring

What to benchmark?

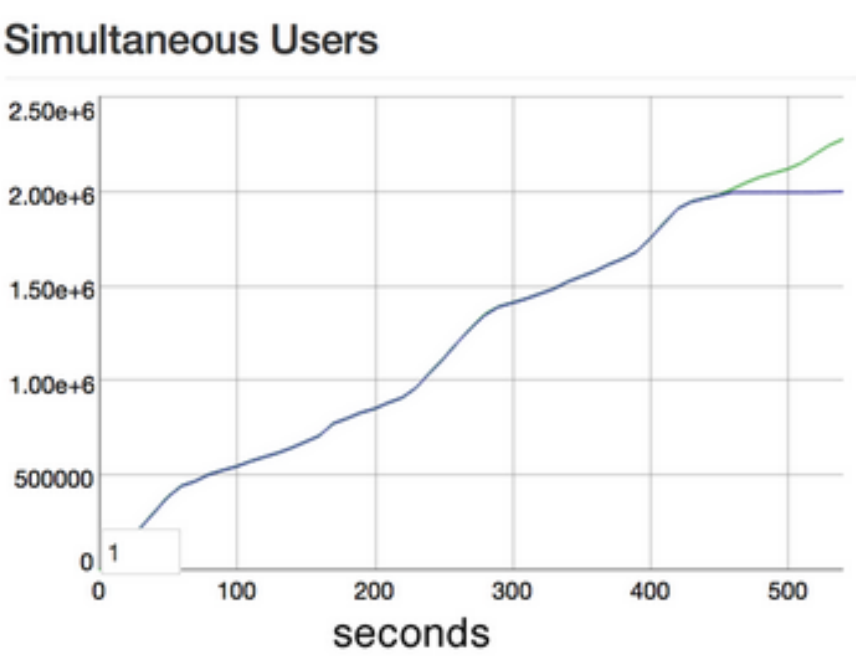
What to measure?

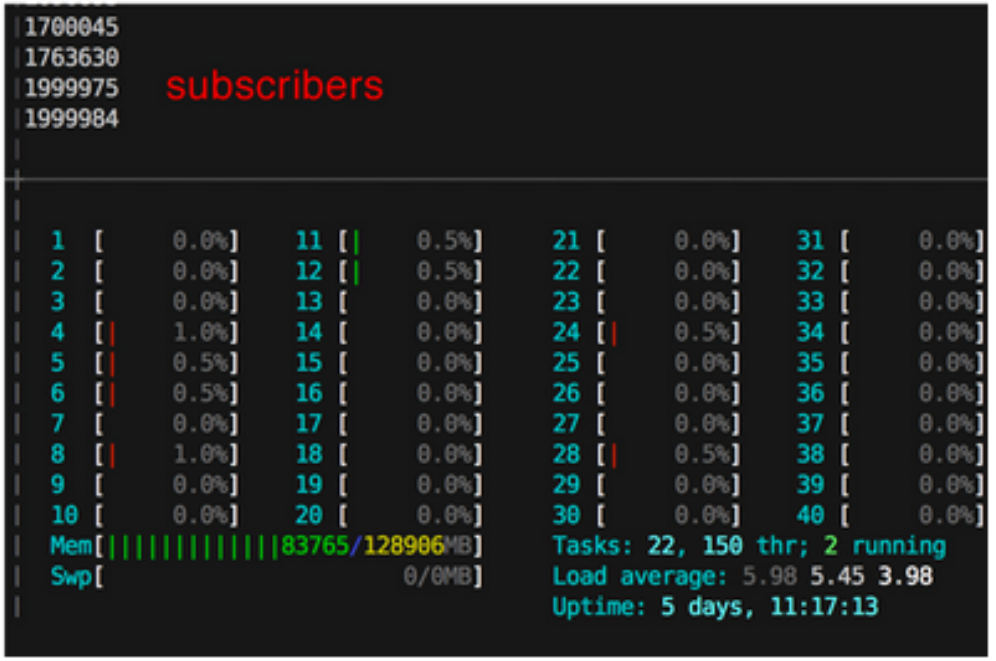
- Runtime?
- Memory?
- Throughput?
- Custom?

The famous post

The Road to 2 Million Websocket Connections in Phoenix

By Gary Rennie · about a year ago · v1.0.0





If you have been paying attention on Twitter recently, you have likely seen some increasing numbers regarding the number of simultaneous connections the Phoenix web framework can handle. This post documents some of the techniques used to perform the benchmarks.

What to measure?

- **Runtime!**
- Memory?
- Throughput?
- Custom?

But, **why?**

What's **fastest**?

How **long** will this take?

Did we make it **faster**?

“Isn’t that the root of all evil?”

More likely, not reading the sources is the source of all evil

Me, just now

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Donald Knuth, 1974

(Computing Surveys, Vol 6, No 4, December 1974)

The very next sentence

"Yet we should not pass up our opportunities in that critical 3%.
A good programmer (...) will be wise to look carefully at the critical code but only after that code has been identified."
Donald Knuth, 1974
(Computing Surveys, Vol 6, No 4, December 1974)

Prior Paragraph

"In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering."

Donald Knuth, 1974

(Computing Surveys, Vol 6, No 4, December 1974)

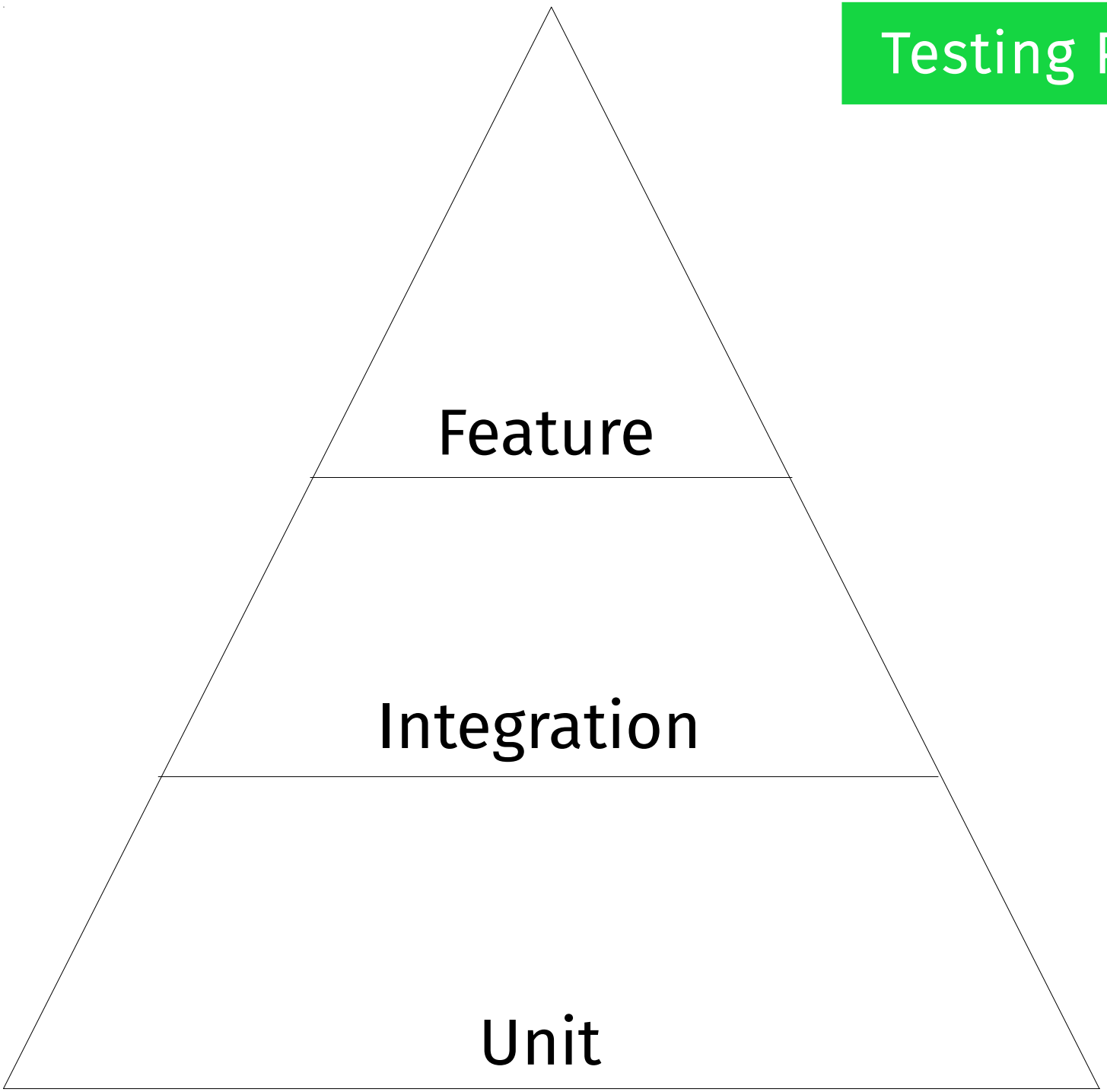
*"It is often a **mistake to make a priori judgments** about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that **their intuitive guesses fail.**"*

Donald Knuth, 1974

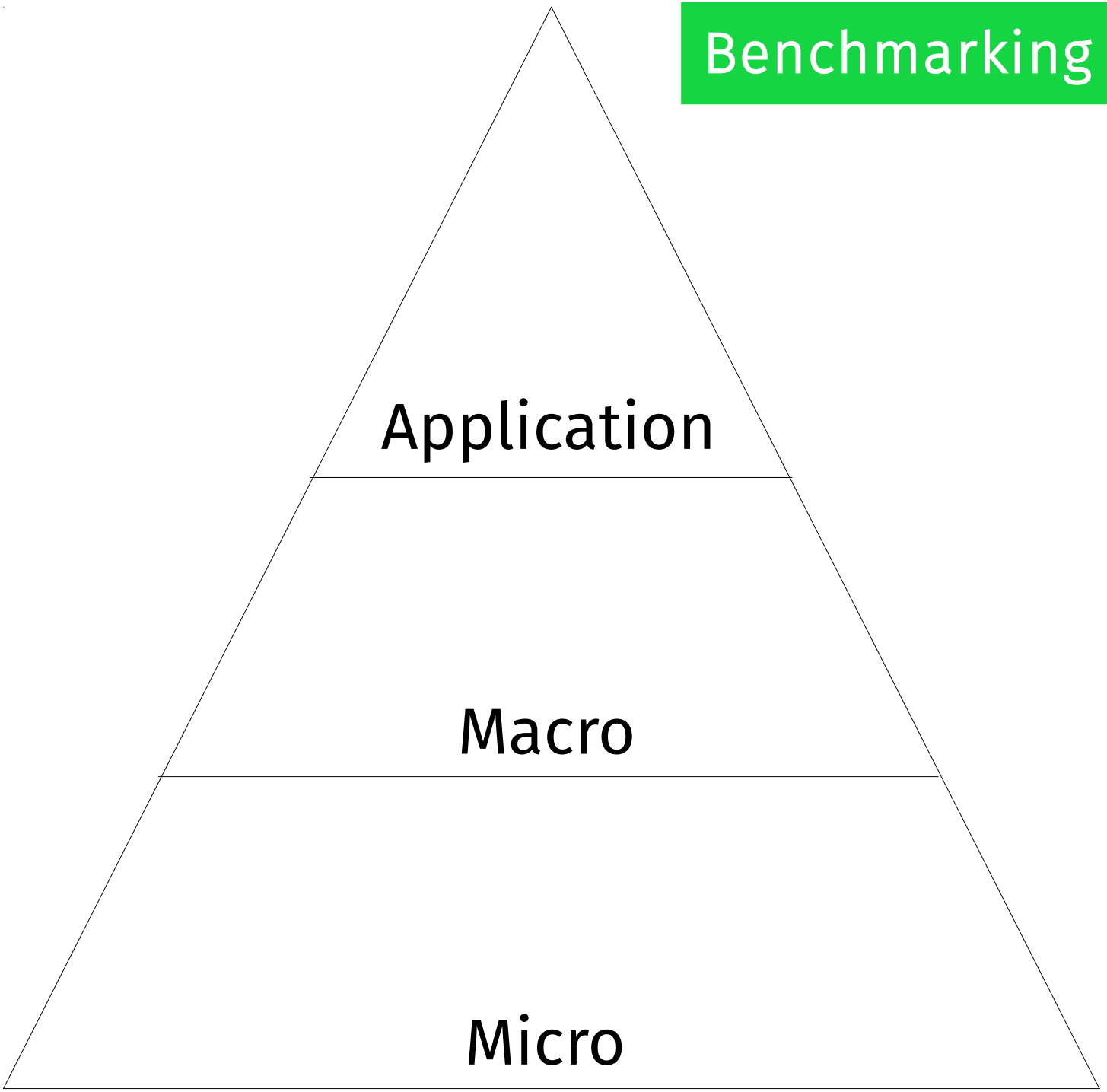
(Computing Surveys, Vol 6, No 4, December 1974)

Different **types** of benchmarks

Testing Pyramid



Benchmarking Pyramid



Results

20-updates (bar) | Data table | Latency | Framework overhead

Responses per second at 20 updates per request, Dell servers at ServerCentral (179 tests)

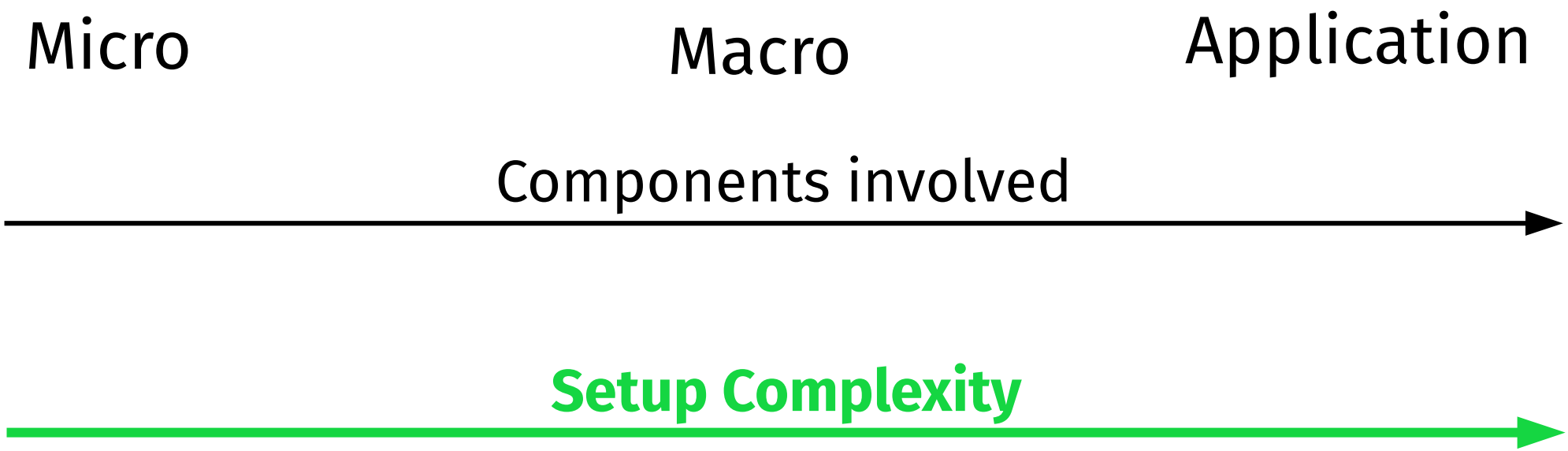
Framework	Performance (higher is better)	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA	Errors
fasthttp-postgresql-	3,050 100.0%	Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea	0
wt-postgres	2,945 96.6%	Ful	C++	Non	Non	Lin	Pg	Lin	Ful	Rea	0
revenj.jvm	2,873 94.2%	Ful	Jav	Svt	Res	Lin	Pg	Lin	Ful	Rea	0
express-mongodb	2,841 93.1%	Mcr	JS	Non	Non	Lin	Mo	Lin	Ful	Rea	0
cutelyst-pf-pg-raw	2,735 89.7%	Plt	C++	Qt	Non	Lin	Pg	Lin	Raw	Rea	0
cutelyst-uwsgi-nginx	2,717 89.1%	Plt	C++	Qt	ngx	Lin	Pg	Lin	Raw	Rea	0
cutelyst-thread-pg-r	2,699 88.5%	Plt	C++	Qt	Non	Lin	Pg	Lin	Raw	Rea	0
mojolicious	2,479 81.3%	Ful	Prl	Non	Hyp	Lin	Pg	Lin	Raw	Rea	72
fasthttp	2,455 80.5%	Plt	Go	Non	Non	Lin	My	Lin	Raw	Rea	0
wt	2,341 76.8%	Ful	C++	Non	Non	Lin	My	Lin	Ful	Rea	0
ulib-mysql	2,317 76.0%	Plt	C++	Non	ULi	Lin	My	Lin	Mcr	Rea	0
fasthttp-mysql-prefo	2,280 74.8%	Plt	Go	Non	Non	Lin	My	Lin	Raw	Rea	0
cutelyst-pf-mysql-ra	1,996 65.4%	Plt	C++	Qt	Non	Lin	My	Lin	Raw	Rea	0
nodejs	1,982 65.0%	Plt	JS	njs	Non	Lin	My	Lin	Raw	Rea	0
aspnetcore-mvc-raw	1,976 64.8%	Ful	C#	Net	Non	Lin	Pg	Lin	Raw	Rea	0
cutelyst-thread-mysq	1,975 64.8%	Plt	C++	Qt	Non	Lin	My	Lin	Raw	Rea	0
aspnetcore-middlewar	1,955 64.1%	Mcr	C#	Net	Non	Lin	Pg	Lin	Raw	Rea	0
cutelyst-uwsgi-nginx	1,933 63.4%	Plt	C++	Qt	ngx	Lin	My	Lin	Raw	Rea	0
phoenix	1,915 62.8%	Mcr	Eli	Cow	Non	Lin	Pg	Lin	Ful	Rea	0
redstone-postgresql	1,857 60.9%	Mcr	Dar	Non	Non	Lin	Pg	Lin	Mcr	Rea	0

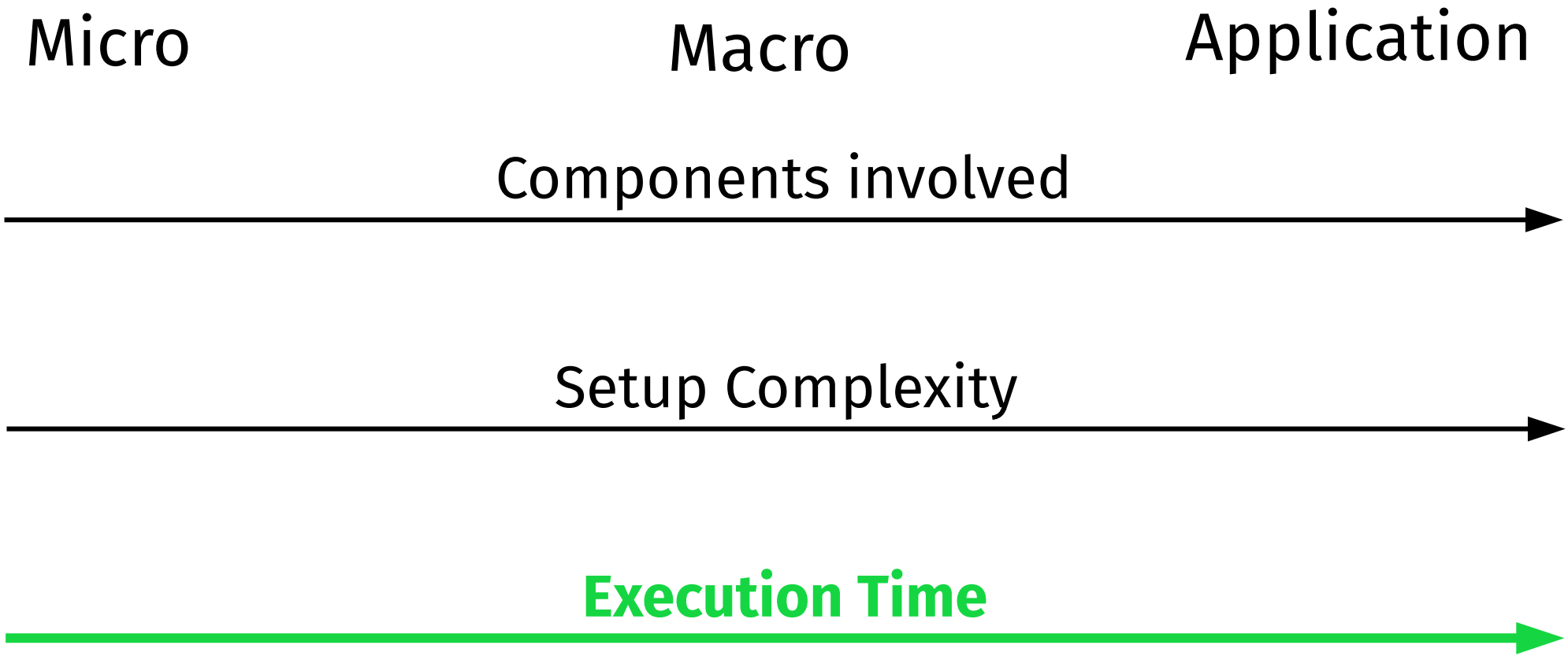
Micro

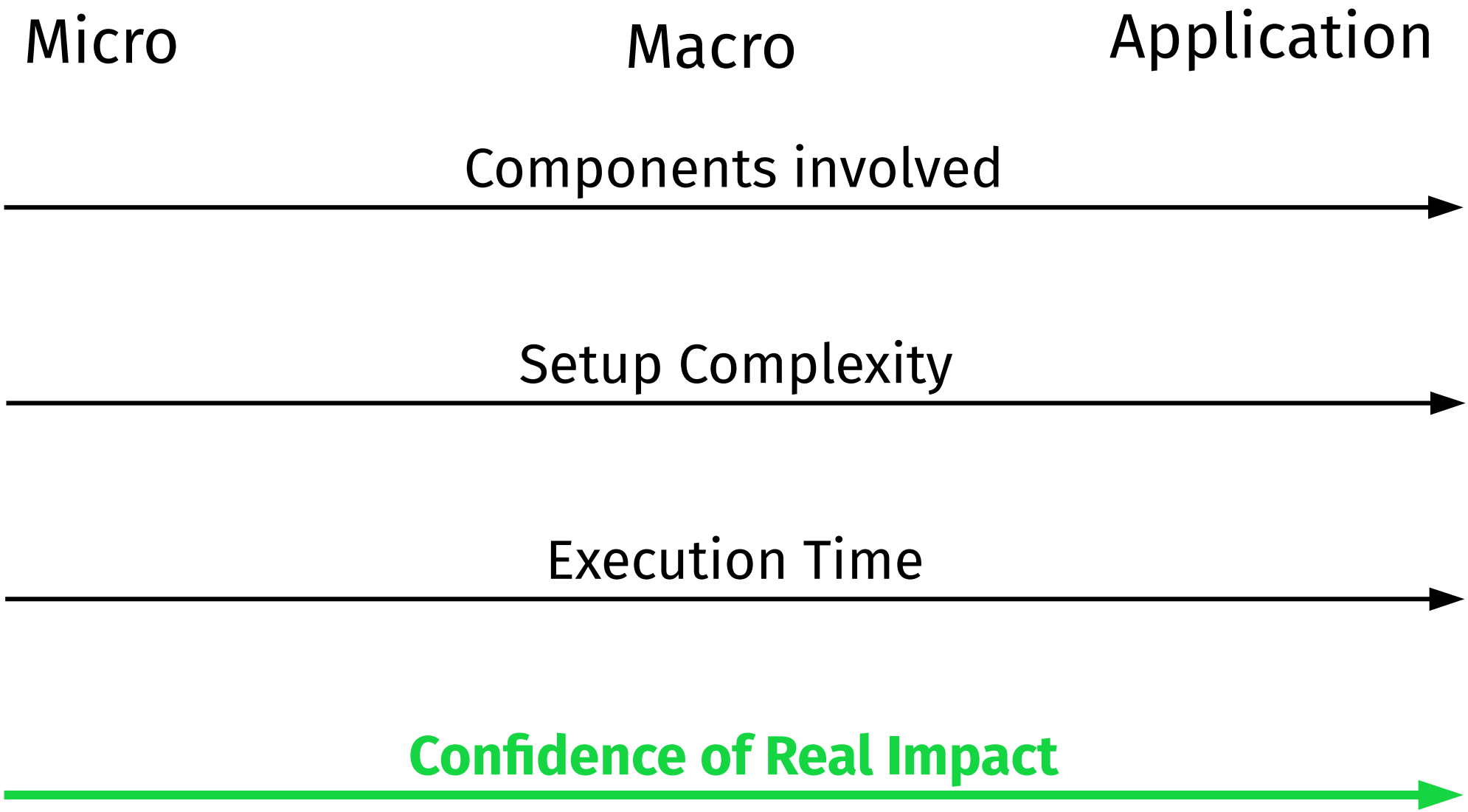
Macro

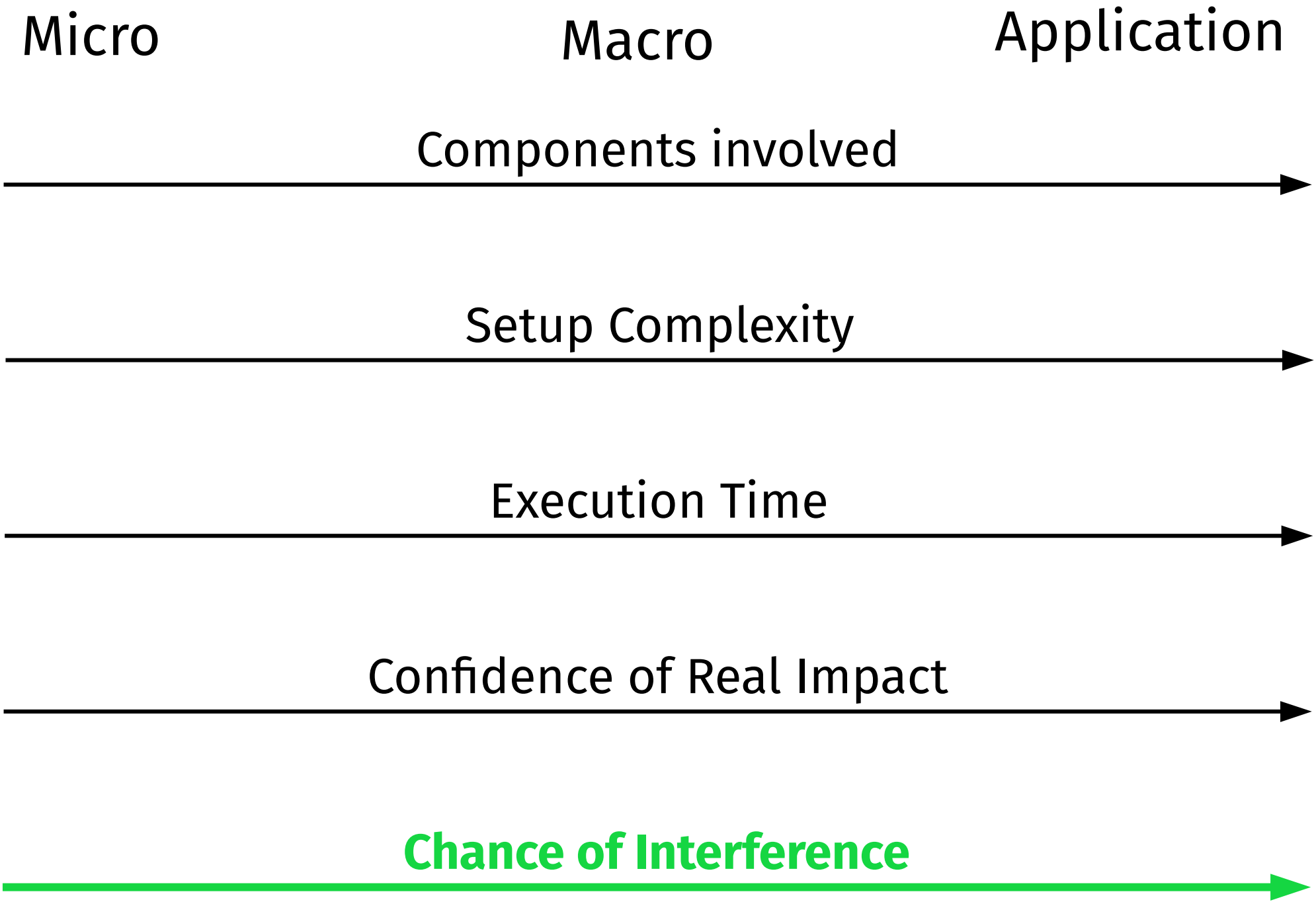
Application

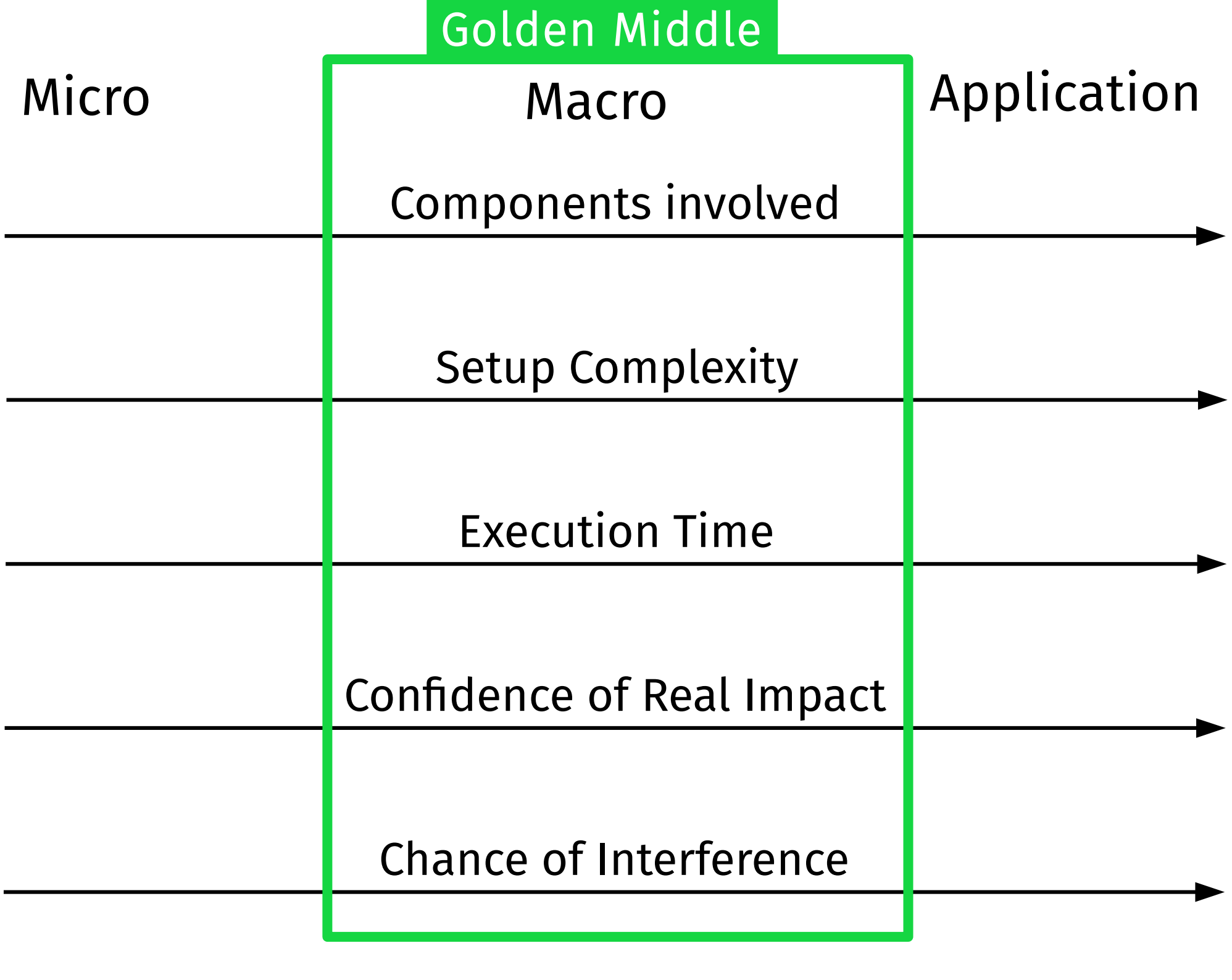

Micro
Macro
Application
Components involved

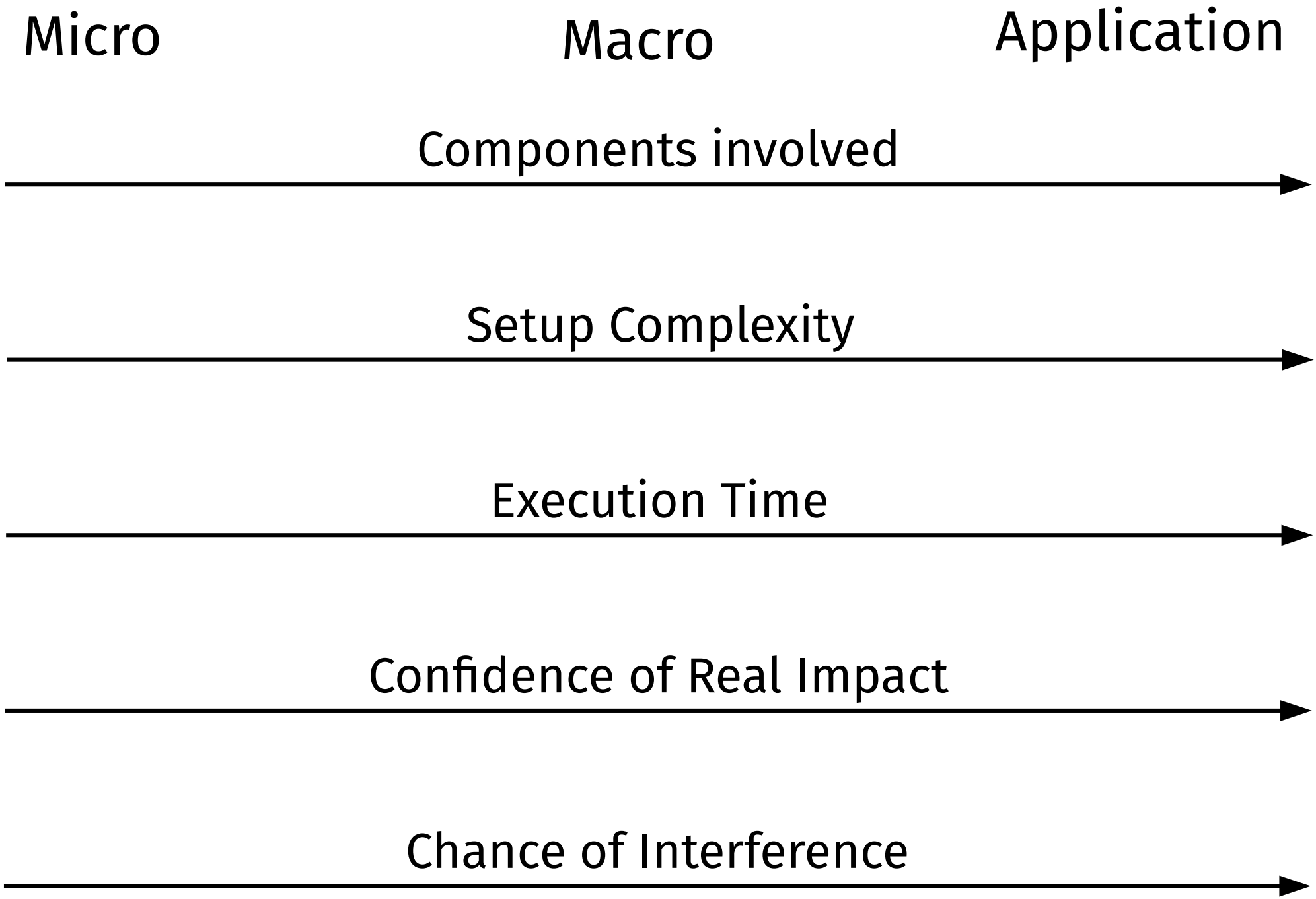

Micro
Macro
Application
Components involved
Setup Complexity


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference


Golden Middle
Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference

micro: tr vs gsub

```ruby
# frozen_string_literal: true
require 'benchmark/ips'

BASE_STRING =
  "Some arbitrary string that we want to manipulate"

Benchmark.ips do |bm|
  bm.report("gsub") do
    BASE_STRING.gsub(" ", "_")
  end

  bm.report("tr") do
    BASE_STRING.tr(" ", "_")
  end

  bm.compare!
end
```

micro: tr vs gsub

```
                gsub    619.337k (± 1.5%) i/s
                  tr      2.460M (± 1.6%) i/s

Comparison:
                  tr:  2460218.8 i/s
                gsub:   619336.7 i/s - 3.97x  slower
```

micro: scoring

```ruby
Benchmark.ips do |benchmark|
  game_19 = playout_for(19).game_state.game
  scorer  = Rubykon::GameScorer.new

  benchmark.report '19x19 scoring' do
    scorer.score game_19
  end
end
```

macro: playout

```
Benchmark.ips do |benchmark|
  benchmark.report '19x19 playout' do
    game       = Rubykon::Game.new(19)
    game_state = Rubykon::GameState.new(game)
    mcts       = MCTS::Playout.new(game_state)
    mcts.playout
  end
end
```

Application: tree search

```ruby
Benchmark.avg do |benchmark|
  game_19       = Rubykon::Game.new(19)
  game_state_19 = Rubykon::GameState.new game_19
  mcts          = MCTS::MCTS.new

  benchmark.config warmup: 180, time: 180

  benchmark.report "19x19 1_000 iterations" do
    mcts.start game_state_19, 1_000
  end
end
```

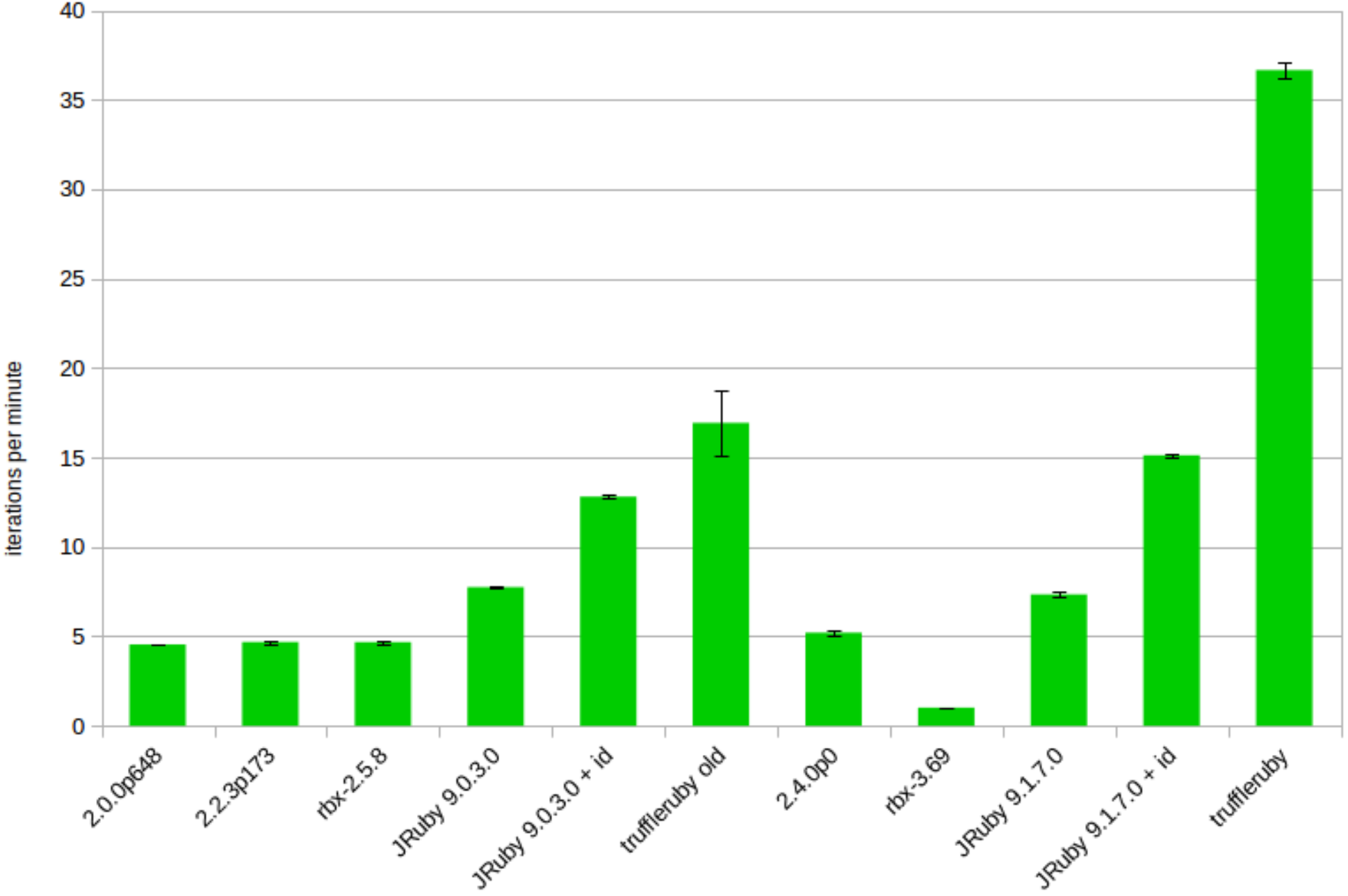
Application: tree search

```ruby
Benchmark.avg do |benchmark|
  game_19          = Rubykon::Game.new(19)
  game_state_19 = Rubykon::GameState.new game_19
  mcts             = MCTS::MCTS.new

  benchmark.config warmup: 180, time: 180

  benchmark.report "19x19 1_000 iterations" do
    mcts.start game_state_19, 1_000
  end
end
```

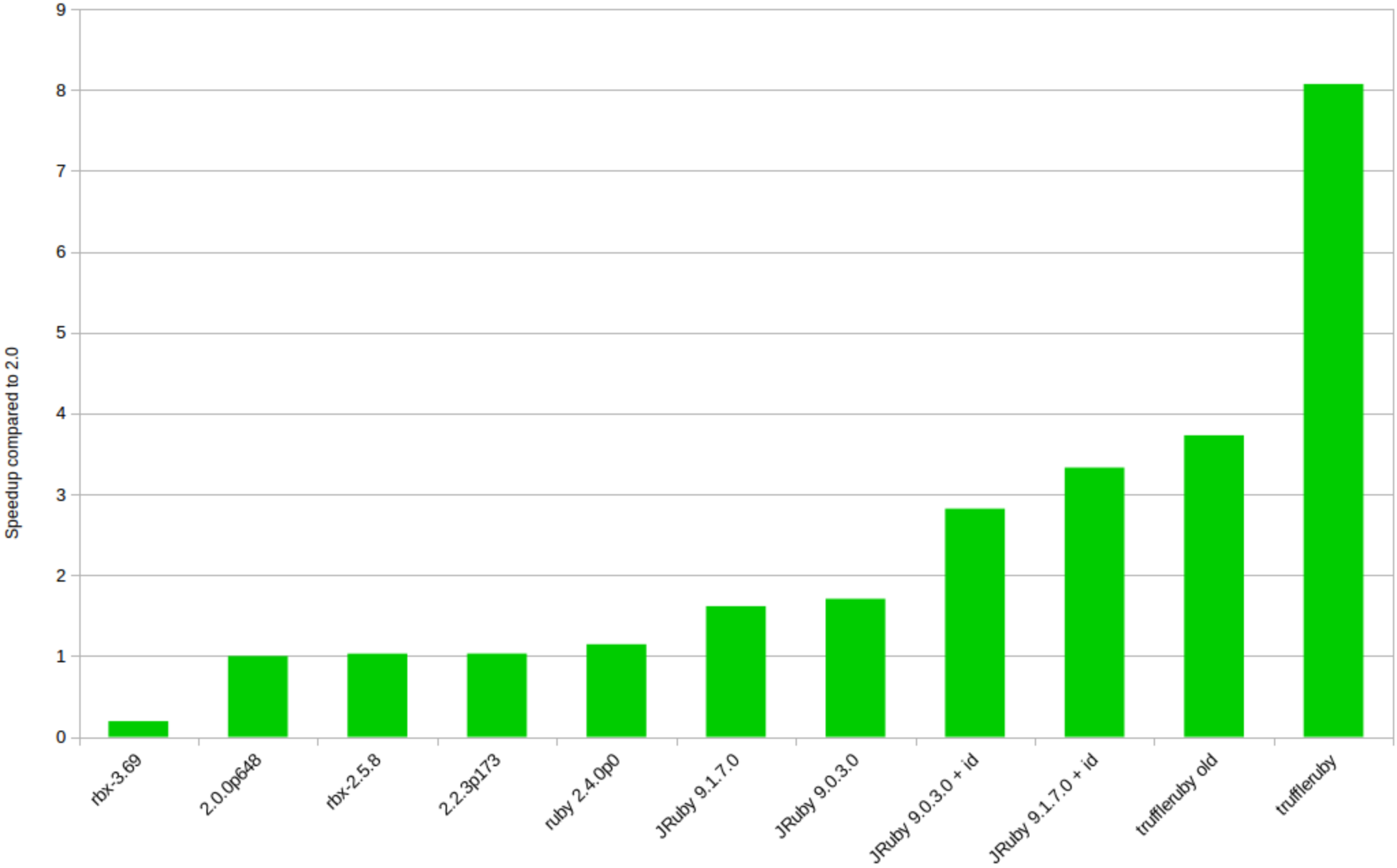
Application: tree search

```ruby
Benchmark.avg do |benchmark|
  game_19       = Rubykon::Game.new(19)
  game_state_19 = Rubykon::GameState.new game_19
  mcts          = MCTS::MCTS.new

  benchmark.config warmup: 180, time: 180

  benchmark.report "19x19 1_000 iterations" do
    mcts.start game_state_19, 1_000
  end
end
```

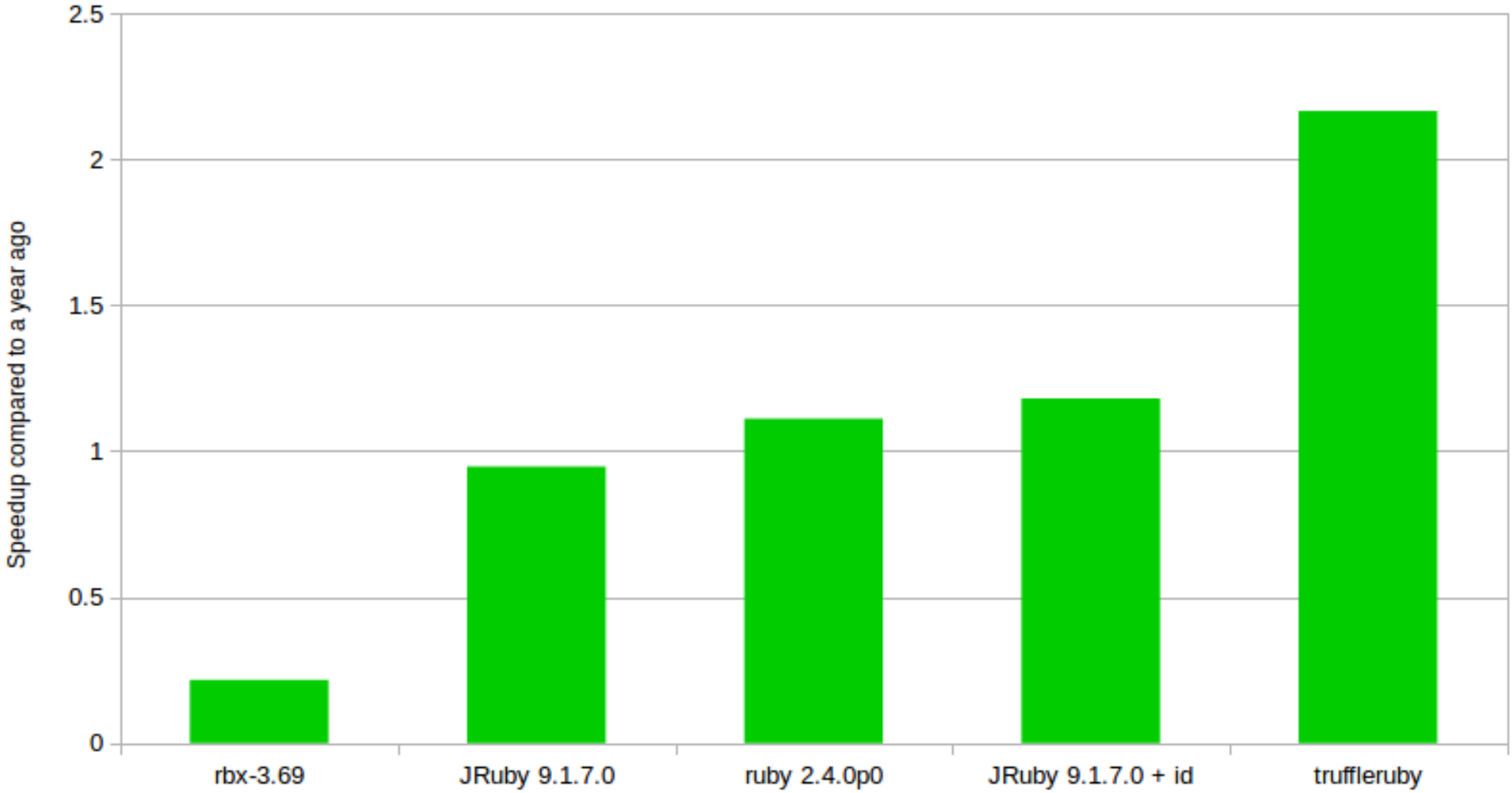
Great ruby rumble



Speedup relative to 2.0



Relative to a year ago



Good Benchmarking

What are you benchmarking for?

System Specification

- Ruby 2.4.1 / JRuby 9.1.8.0 on OpenJDK 8
- Elixir 1.3.4
- Erlang 19.1
- i5-7200U – 2 x 2.5GHz (Up to 3.10GHz)
- 8GB RAM
- Linux Mint 18.1 - 64 bit (Ubuntu 16.04 base)
- Linux Kernel 4.8.0


Interference free Environment


Correct & Meaningful Setup

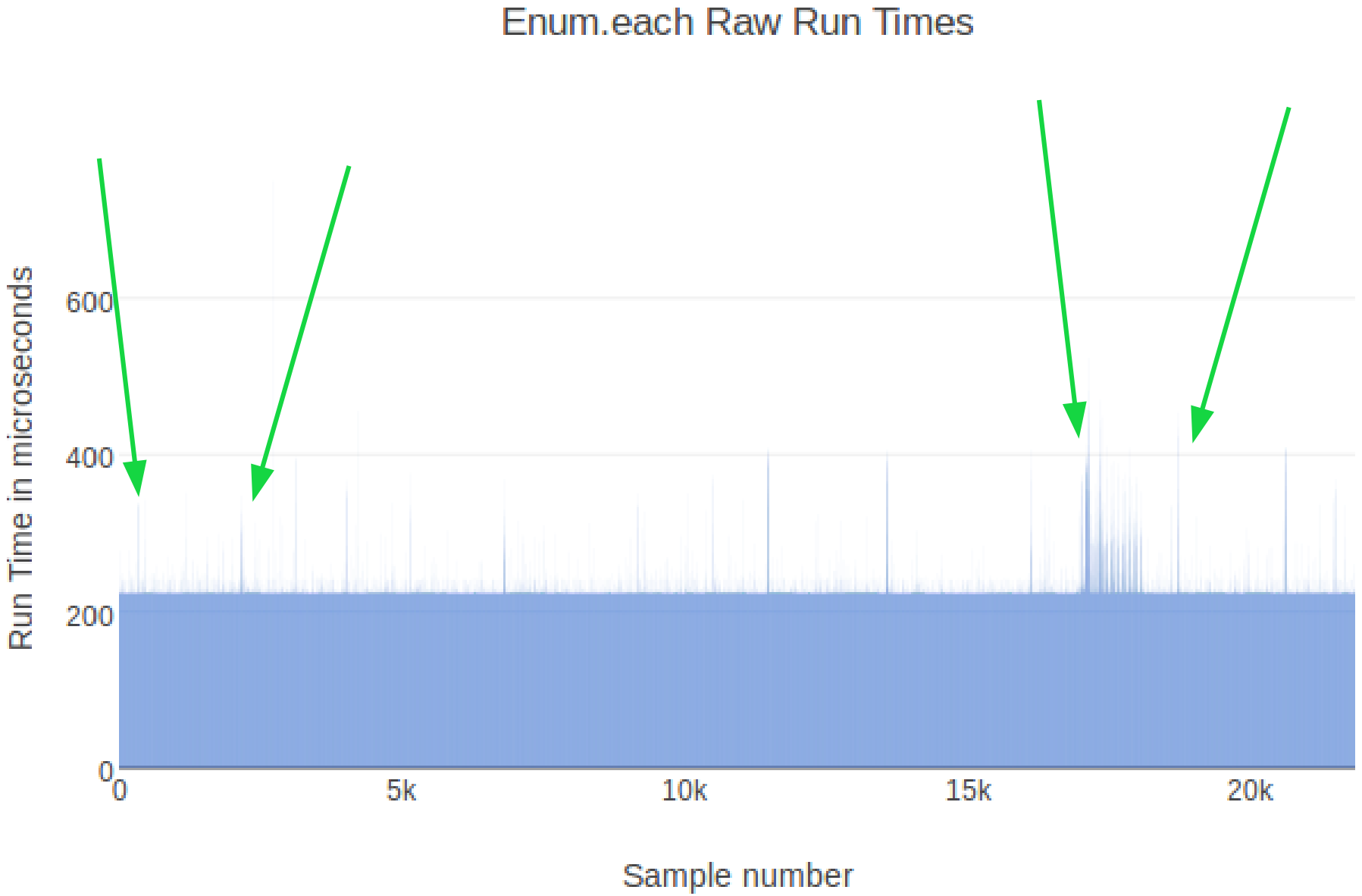

RAILS_ENV=performance

Logging & Friends

```
[info] GET /
[debug] Processing by Rumbl.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 46ms
[info] GET /sessions/new
[debug] Processing by Rumbl.SessionController.new/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 5ms
[info] GET /users/new
[debug] Processing by Rumbl.UserController.new/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 7ms
[info] POST /users
[debug] Processing by Rumbl.UserController.create/2
  Parameters: %{"_csrf_token" =>
"NUEUdRMNAiBfIHEeNwZkfA05PgAOJgAAf0ACXJqCjl7YojW+trdjdg==", "_utf8" => "✓", "user" =>
%{"name" => "asdasd", "password" => "[FILTERED]", "username" => "Homer"}}
  Pipelines: [:browser]
[debug] QUERY OK db=0.1ms
begin []
[debug] QUERY OK db=0.9ms
INSERT INTO "users" ("name","password_hash","username","inserted_at","updated_at") VALUES
($1,$2,$3,$4,$5) RETURNING "id" ["asdasd",
"$2b$12$.qY/kpo0Dec7vMK1ClJoC.Lw77c3oGllX7uieZILMlFh2hFpJ3F.C", "Homer", {{2016, 12, 2}
{14, 10, 28, 0}}, {{2016, 12, 2}, {14, 10, 28, 0}}]
```


Warmup


Garbage Collection


Enum.each Raw Run Times
Run Time in microseconds
600
400
200
0
0
5k
10k
15k
20k
Sample number


Inputs matter!

```elixir
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with_courier_ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

A real case

```
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with_courier_ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

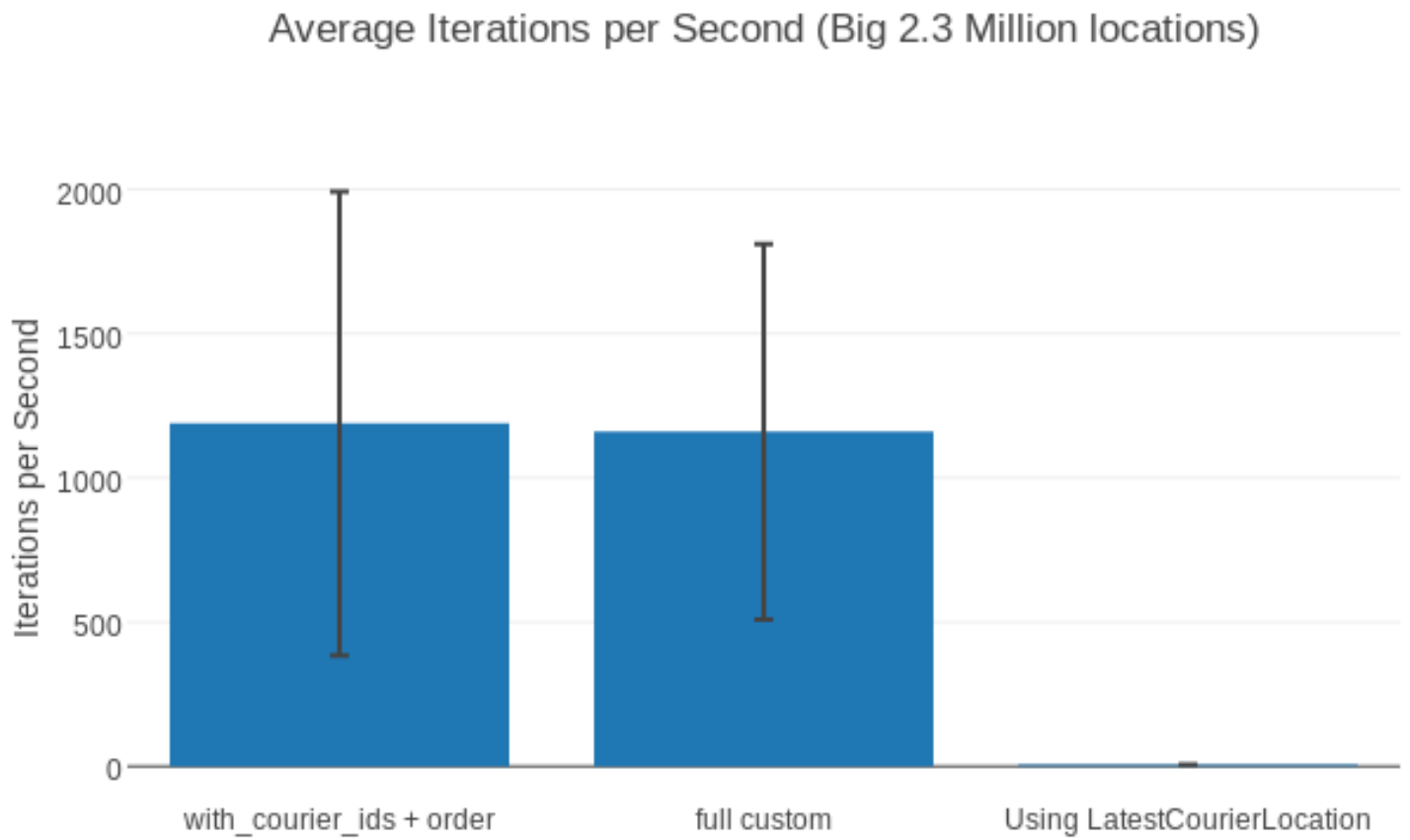
Database View

```
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with_courier_ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

Only difference

```
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with courier_ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

Another job well done?




```
Name                                  ips        average  deviation         median
with_courier_ids + order           1.19 K      841.44 µs    ±67.64%      675.00 µs
full custom                        1.16 K      862.36 µs    ±56.06%      737.00 µs
Using LatestCourierLocation     0.00603 K   165897.47 µs     ±2.33%   165570.00 µs

Comparison:
with_courier_ids + order           1.19 K
full custom                        1.16 K - 1.02x slower
Using LatestCourierLocation     0.00603 K - 197.16x slower
```

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}, inputs: inputs, time: 25, warmup: 5
```

Old

```
##### With input Big 2.3 Million locations #####
Comparison:
with_courier_ids + order           1.19 K
full custom                        1.16 K - 1.02x slower
Using LatestCourierLocation     0.00603 K - 197.16x slower

##### With input ~200k locations #####
Comparison:
Using LatestCourierLocation         3.66
full custom                        0.133 - 27.57x slower
with_courier_ids + order           0.132 - 27.63x slower

##### With input ~20k locations #####
Comparison:
Using LatestCourierLocation         38.12
full custom                        0.122 - 312.44x slower
with_courier_ids + order           0.122 - 313.33x slower

##### With input No locations #####
Comparison:
Using LatestCourierLocation       2967.48
full custom                          0.114 - 25970.57x slower
with_courier_ids + order             0.114 - 26046.06x slower
```

Old

```
##### With input Big 2.3 Million locations #####
Comparison:
with_courier_ids + order            1.19 K
full_custom                         1.16 K - 1.02x slower
Using LatestCourierLocation     0.00603 K - 197.16x slower

##### With input ~200k locations #####
Comparison:
Using LatestCourierLocation          3.66
full_custom                         0.133 - 27.57x slower
with_courier_ids + order            0.132 - 27.63x slower

##### With input ~20k locations #####
Comparison:
Using LatestCourierLocation          38.12
full_custom                         0.122 - 312.44x slower
with_courier_ids + order            0.122 - 313.33x slower

##### With input No locations #####
Comparison:
Using LatestCourierLocation        2967.48
full_custom                         0.114 - 25970.57x slower
with_courier_ids + order            0.114 - 26046.06x slower
```

Old

```
##### With input Big 2.3 Million locations #####
Comparison:
with_courier_ids + order            1.19 K
full custom                         1.16 K - 1.02x slower
Using LatestCourierLocation     0.00603 K - 197.16x slower

##### With input ~200k locations #####
Comparison:
Using LatestCourierLocation          3.66
full custom                         0.133 - 27.57x slower
with_courier_ids + order            0.132 - 27.63x slower

##### With input ~20k locations #####
Comparison:
Using LatestCourierLocation         38.12
full custom                         0.122 - 312.44x slower
with_courier_ids + order            0.122 - 313.33x slower

##### With input No locations #####
Comparison:
Using LatestCourierLocation       2967.48
full custom                         0.114 - 25970.57x slower
with_courier_ids + order            0.114 - 26046.06x slower
```

Combined Index

```
##### With input Big 2.3 Million locations #####
Comparison:
full custom                          3921.12
with_courier_ids + order               23.05 - 170.09x slower
Using LatestCourierLocation             5.98 - 655.74x slower

##### With input ~200k locations #####
Comparison:
full custom                          4272.84
with_courier_ids + order               14.20 - 300.91x slower
Using LatestCourierLocation             3.80 - 1125.59x slower

##### With input ~20k locations #####
Comparison:
full custom                          3792.97
with_courier_ids + order               78.93 - 48.06x slower
Using LatestCourierLocation            35.62 - 106.47x slower

##### With input No locations #####
Comparison:
full custom                           5.14 K
with_courier_ids + order              3.87 K - 1.33x slower
Using LatestCourierLocation           3.29 K - 1.56x slower
```

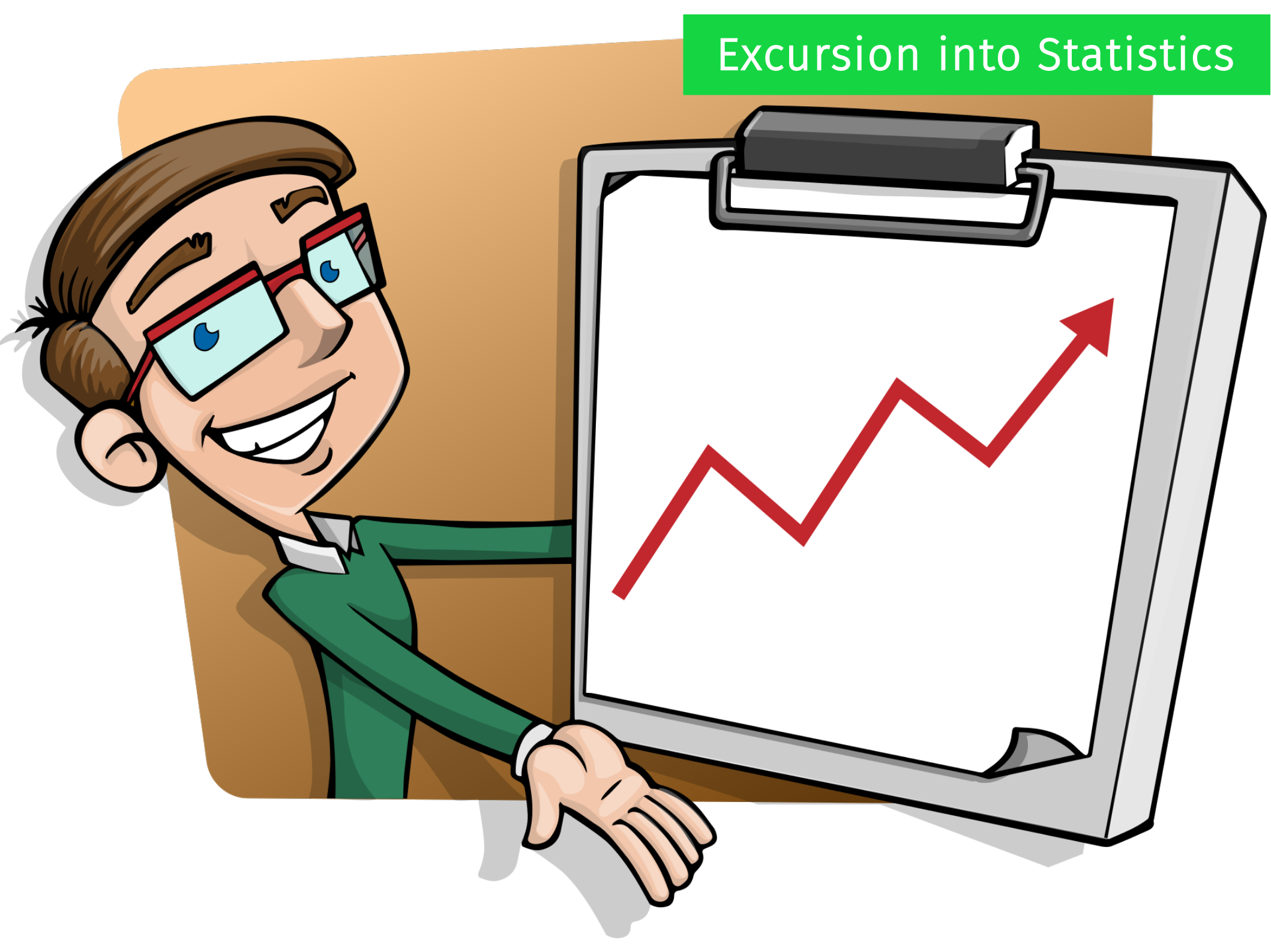
Combined Index

```
##### With input Big 2.3 Million locations #####
Comparison:
full custom                                3921.12
with_courier_ids + order                     23.05 - 170.09x slower
Using LatestCourierLocation                   5.98 - 655.74x slower

##### With input ~200k locations #####
Comparison:
full custom                                4272.84
with_courier_ids + order                     14.20 - 300.91x slower
Using LatestCourierLocation                   3.80 - 1125.59x slower

##### With input ~20k locations #####
Comparison:
full custom                                3792.97
with_courier_ids + order                     78.93 - 48.06x slower
Using LatestCourierLocation                  35.62 - 106.47x slower

##### With input No locations #####
Comparison:
full custom                                 5.14 K
with_courier_ids + order                    3.87 K - 1.33x slower
Using LatestCourierLocation                 3.29 K - 1.56x slower
```

Insertion Time

```
# with an index on courier_id and one on time

Name                           ips        average  deviation         median
Updating a location         366.90        2.73 ms    ±36.35%        2.29 ms

# with a combined index on courier_id and time

Name                           ips        average  deviation         median
Updating a location         283.41        3.53 ms    ±52.18%        2.77 ms
```

Excursion into Statistics



Average

```
average = total_time / iterations
```

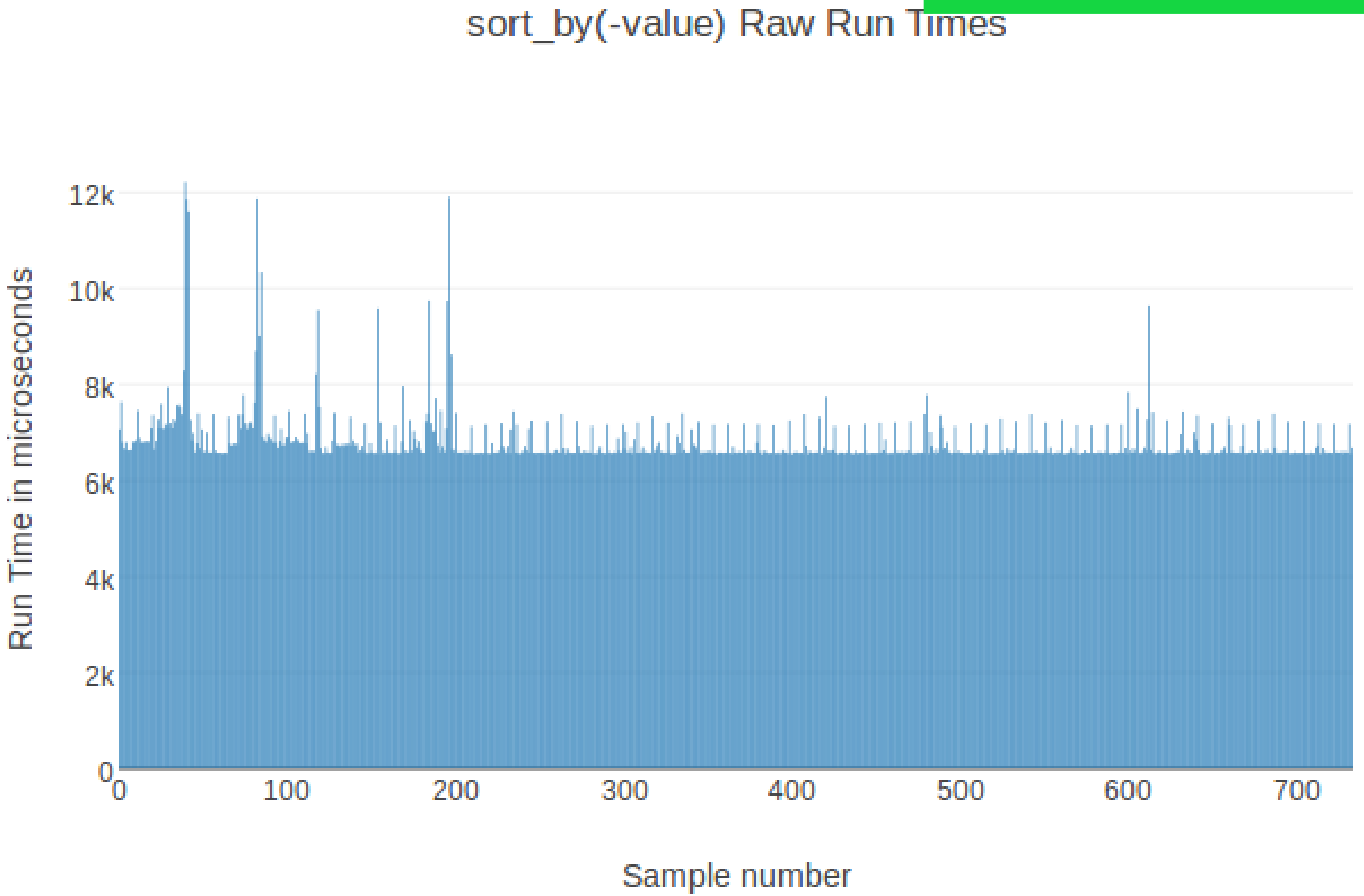
Standard Deviation

```
defp standard_deviation(samples, average, iterations) do
  total_variance = Enum.reduce samples, 0,  fn(sample, total) ->
    total + :math.pow((sample - average), 2)
  end
  variance = total_variance / iterations
  :math.sqrt variance
end
```

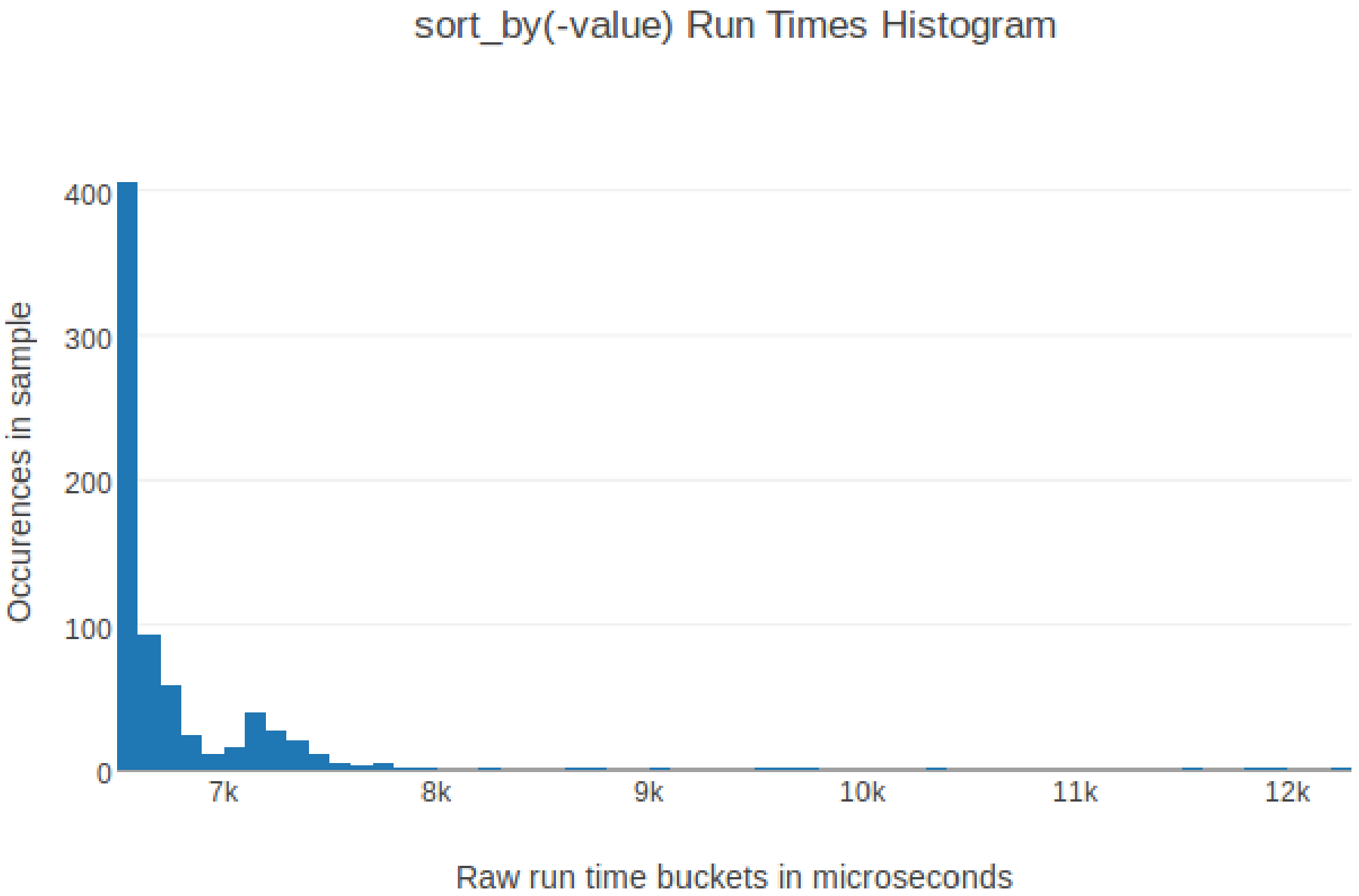
Spread of Values

```
defp standard_deviation(samples, average, iterations) do
  total_variance = Enum.reduce samples, 0,  fn(sample, total) ->
    total + :math.pow((sample - average), 2)
  end
  variance = total_variance / iterations
  :math.sqrt variance
end
```

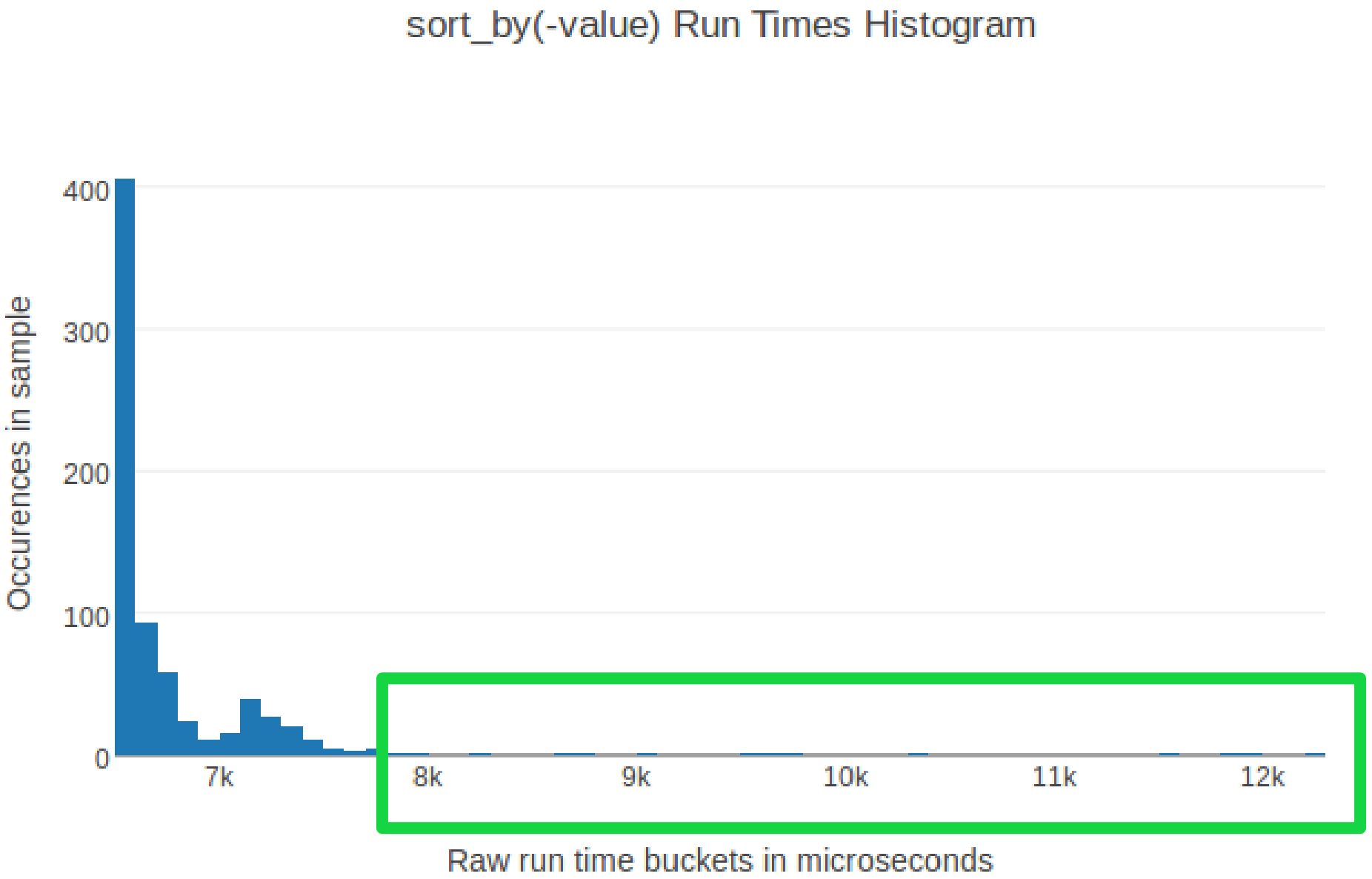
Raw Run Times



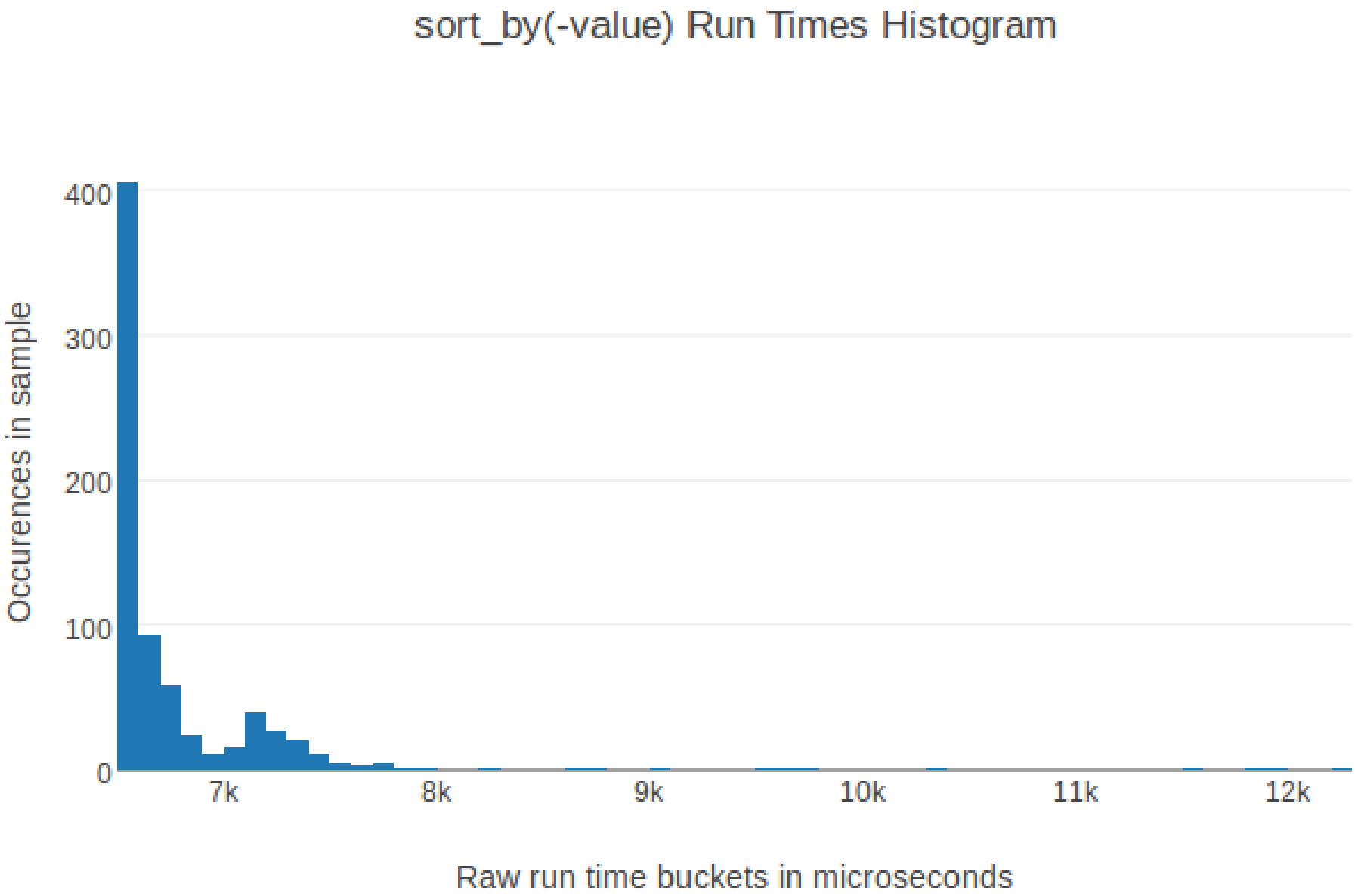
Histogram

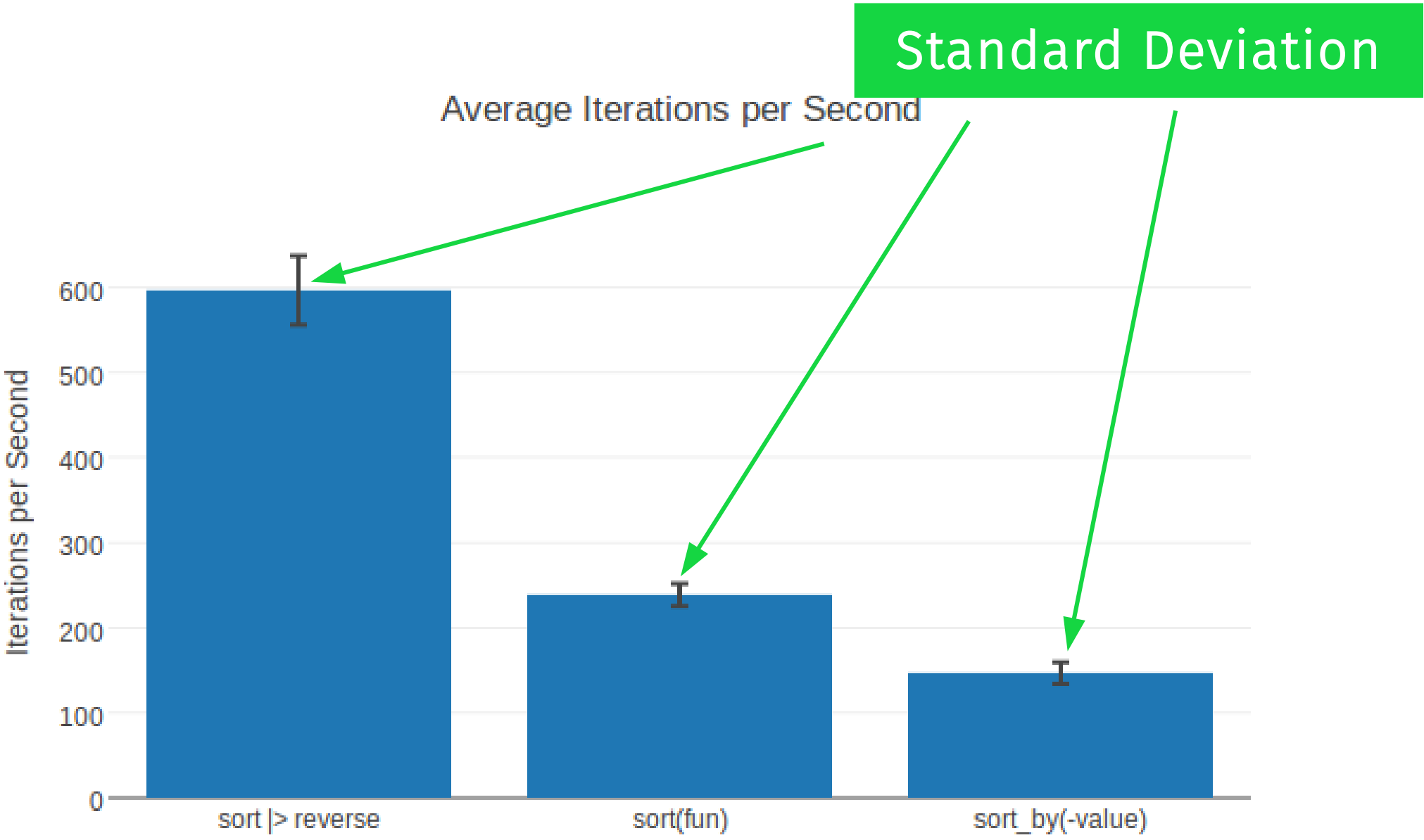

sort_by(-value) Run Times Histogram
Occurences in sample
400
300
200
100
0
7k
8k
9k
10k
11k
12k
Raw run time buckets in microseconds

Outliers


sort_by(-value) Run Times Histogram
Occurences in sample
400
300
200
100
0
7k
8k
9k
10k
11k
12k
Raw run time buckets in microseconds

Low Standard Deviation

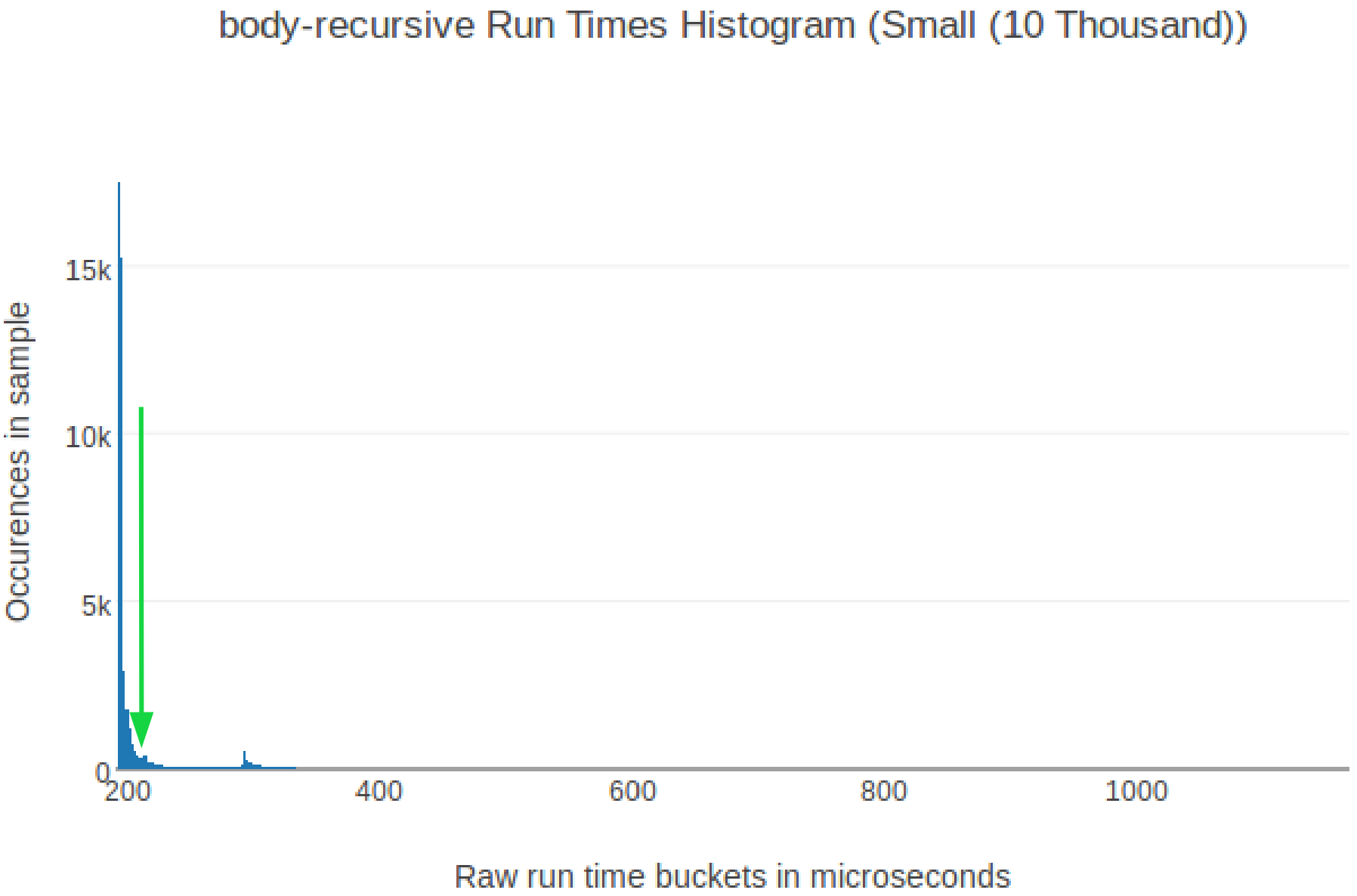


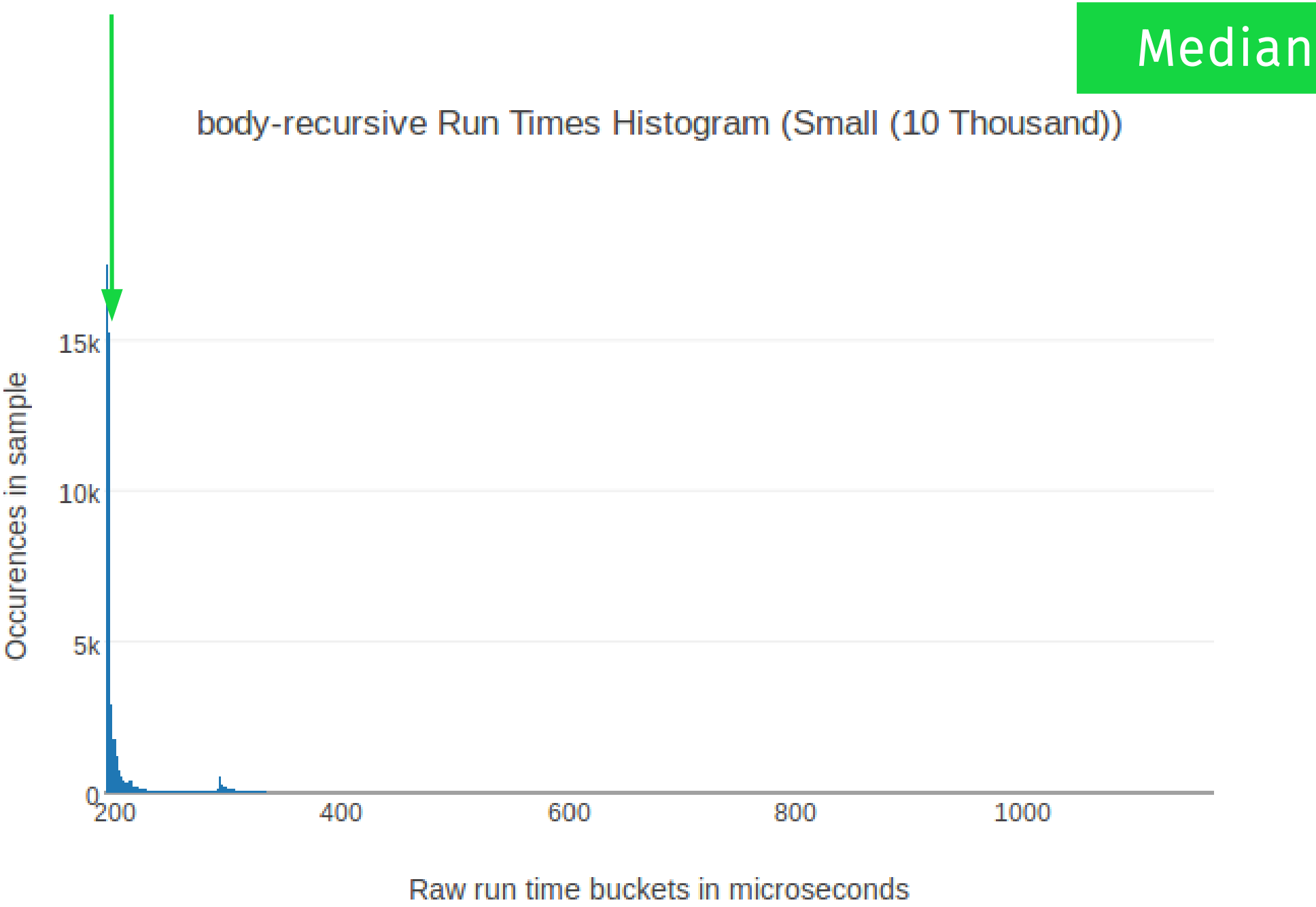

Standard Deviation
Average Iterations per Second
Iterations per Second
600
500
400
300
200
100
0
sort |> reverse
sort(fun)
sort_by(-value)

Median

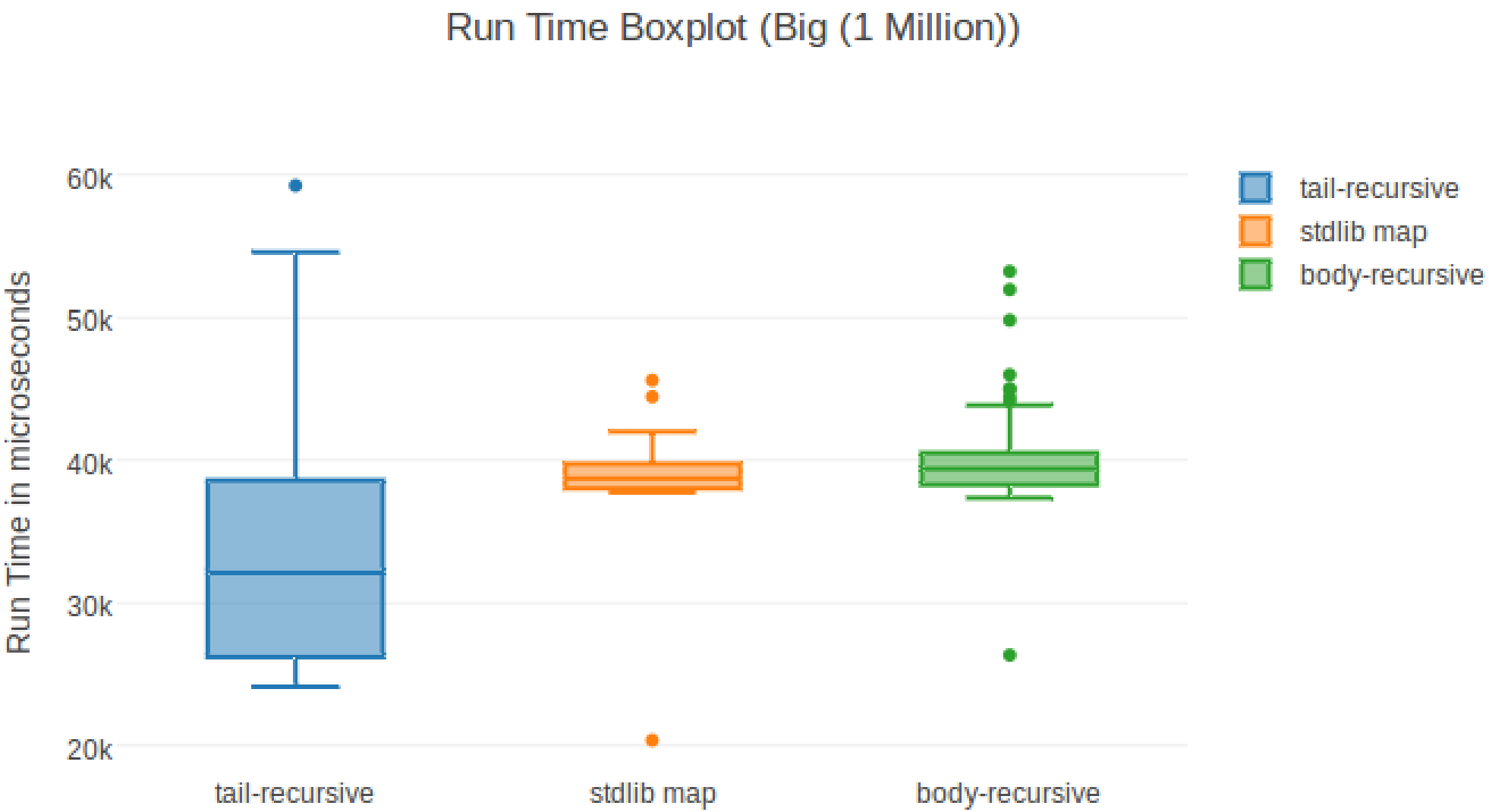
```
defp compute_median(run_times, iterations) do
  sorted = Enum.sort(run_times)
  middle = div(iterations, 2)

  if Integer.is_odd(iterations) do
    sorted |> Enum.at(middle) |> to_float
  else
    (Enum.at(sorted, middle) +
      Enum.at(sorted, middle - 1)) / 2
  end
end
```

Average


body-recursive Run Times Histogram (Small (10 Thousand))
Occurences in sample
15k
10k
5k
0
200
400
600
800
1000
Raw run time buckets in microseconds


Median
body-recursive Run Times Histogram (Small (10 Thousand))
15k
10k
5k
0
Occurences in sample
200
400
600
800
1000
Raw run time buckets in microseconds

Boxplot


Run Time Boxplot (Big (1 Million))
Run Time in microseconds
60k
50k
40k
30k
20k
tail-recursive
stdlib map
body-recursive
tail-recursive
stdlib map
body-recursive

A transformation of inputs

```
config
|> Benchee.init
|> Benchee.system
|> Benchee.benchmark("job", fn -> magic end)
|> Benchee.measure
|> Benchee.statistics
|> Benchee.Formatters.Console.output
|> Benchee.Formatters.HTML.output
```

Enjoy Benchmarking! ❤

Tobias Pfeiffer
@PragTob
pragtob.info
github.com/evanphx/benchmark-ips
github.com/PragTob/benchee

LIEFERY