
The other day

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
iex(3)> list = Enum.to_list(1..100)
[...]
iex(4)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{165, :ok}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
iex(3)> list = Enum.to_list(1..100)
[...]
iex(4)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{165, :ok}
iex(5)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{170, :ok}
iex(6)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{184, :ok}
```


Success!

The End?

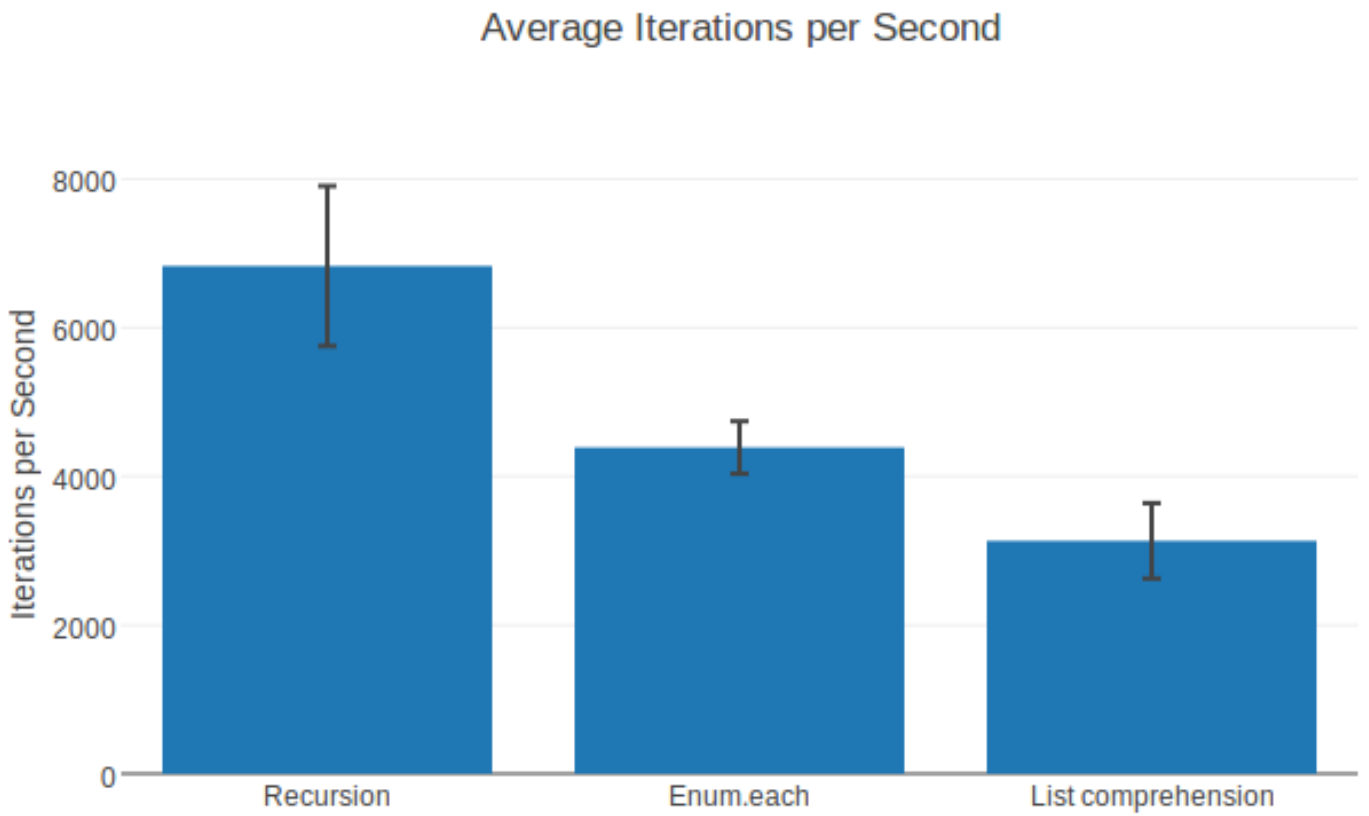

I HAVE NO
IDEA WHAT
I'M DOING

How many **atrocities** have we just committed?

Atrocities

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

```elixir
n      = 10_000
range = 1..n
list  = Enum.to_list range
fun   = fn -> 0 end

Benchee.run %{
  "Enum.each" =>
    fn -> Enum.each(list, fn(_) -> fun.() end) end,
  "List comprehension" =>
    fn -> for _ <- list, do: fun.() end,
  "Recursion" =>
    fn -> RepeatN.repeat_n(fun, n) end
}
```




```
Name                           ips        average  deviation         median
Recursion                   6.83 K      146.41 µs    ±15.76%      139.00 µs
Enum.each                   4.39 K      227.86 µs     ±8.05%      224.00 µs
List comprehension          3.13 K      319.22 µs    ±16.20%      323.00 µs

Comparison:
Recursion                   6.83 K
Enum.each                   4.39 K - 1.56x slower
List comprehension          3.13 K - 2.18x slower
```

How fast is it really?

Benchmarking in Practice

Tobias Pfeiffer
@PragTob
pragtob.info

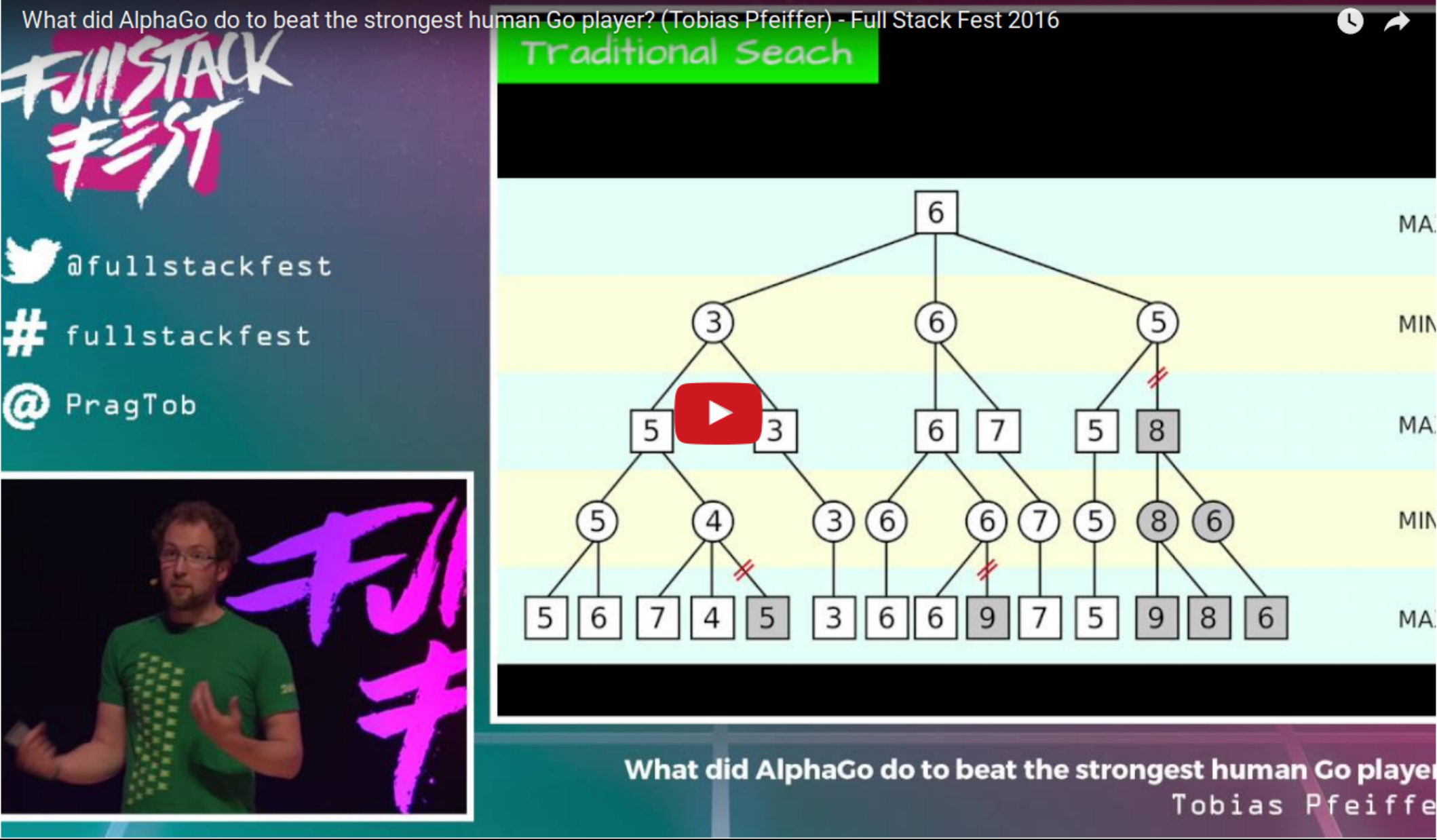
LIEFERY

How fast is it really?

Benchmarking in Practice

Tobias Pfeiffer
@PragTob
pragtob.info

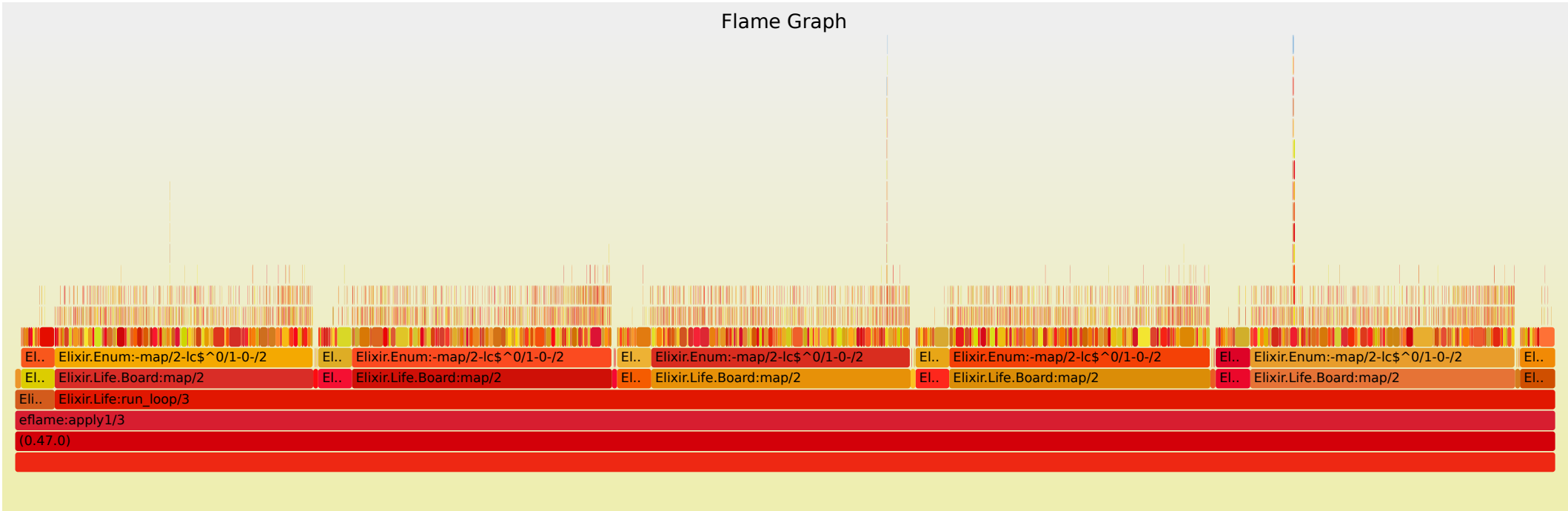
What did AlphaGo do to beat the strongest human Go player? (Tobias Pfeiffer) - Full Stack Fest 2016



Concept vs Tool Usage

Ruby?

Profiling vs. Benchmarking



http://learningelixir.joekain.com/profiling-elixir-2/

What to benchmark?

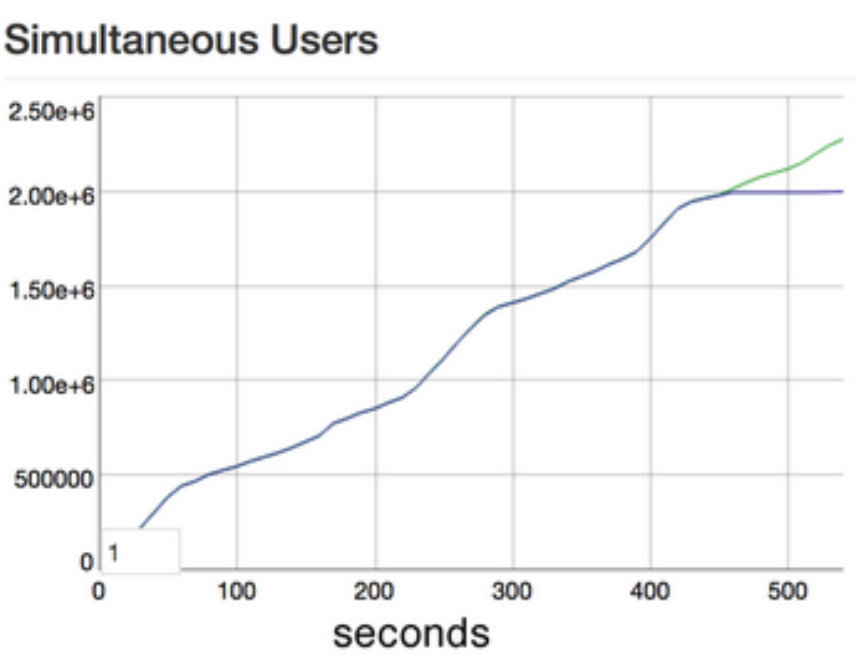
What to measure?

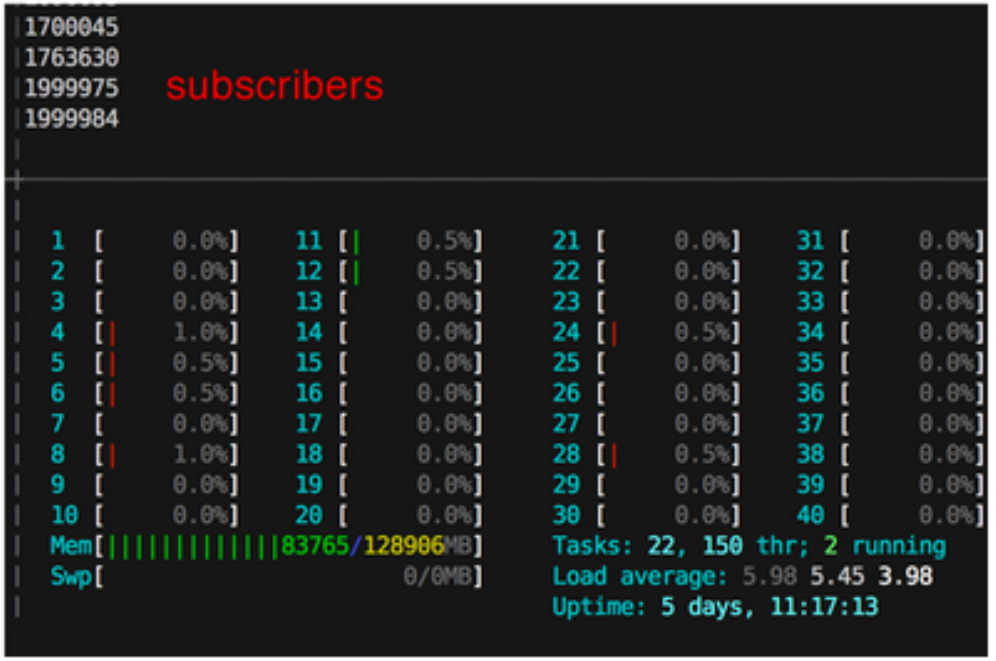
- Runtime?
- Memory?
- Throughput?
- Custom?

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?

Enum.sort/1 performance

```
Name             ips        average  deviation         median
10k           595.62        1.68 ms     ±8.77%        1.61 ms
100k           43.29       23.10 ms    ±13.21%       21.50 ms
1M              3.26      306.53 ms     ±9.82%      291.05 ms
5M              0.53     1899.00 ms     ±7.94%     1834.97 ms

Comparison:
10k           595.62
100k           43.29 - 13.76x slower
1M              3.26 - 182.58x slower
5M              0.53 - 1131.09x slower
```

Enum.sort performance

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
10k	595.62	1678.91 µs	±8.77%	1614.00 µs	1539.00 µs	3240.00 µs	2976
100k	43.29	23102.24 µs	±13.21%	21500.00 µs	20692.00 µs	30785.00 µs	217
1M	3.26	306527.88 µs	±9.82%	291052.00 µs	277885.00 µs	375008.00 µs	17
5M	0.53	1899004.67 µs	±7.94%	1834967.00 µs	1754852.00 µs	2107195.00 µs	3

Enum.sort performance

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
10k	595.62	1678.91 µs	±8.77%	1614.00 µs	1539.00 µs	3240.00 µs	2976
100k	43.29	23102.24 µs	±13.21%	21500.00 µs	20692.00 µs	30785.00 µs	217
1M	3.26	306527.88 µs	±9.82%	291052.00 µs	277885.00 µs	375008.00 µs	17
5M	0.53	1899004.67 µs	±7.94%	1834967.00 µs	1754852.00 µs	2107195.00 µs	3

Did we make it **faster**?

"Isn't that the root of all evil?"


"Programing Bumper Sticker"
REAL
STICK'R

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

"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)

80 / 20

What is **critical**?

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)

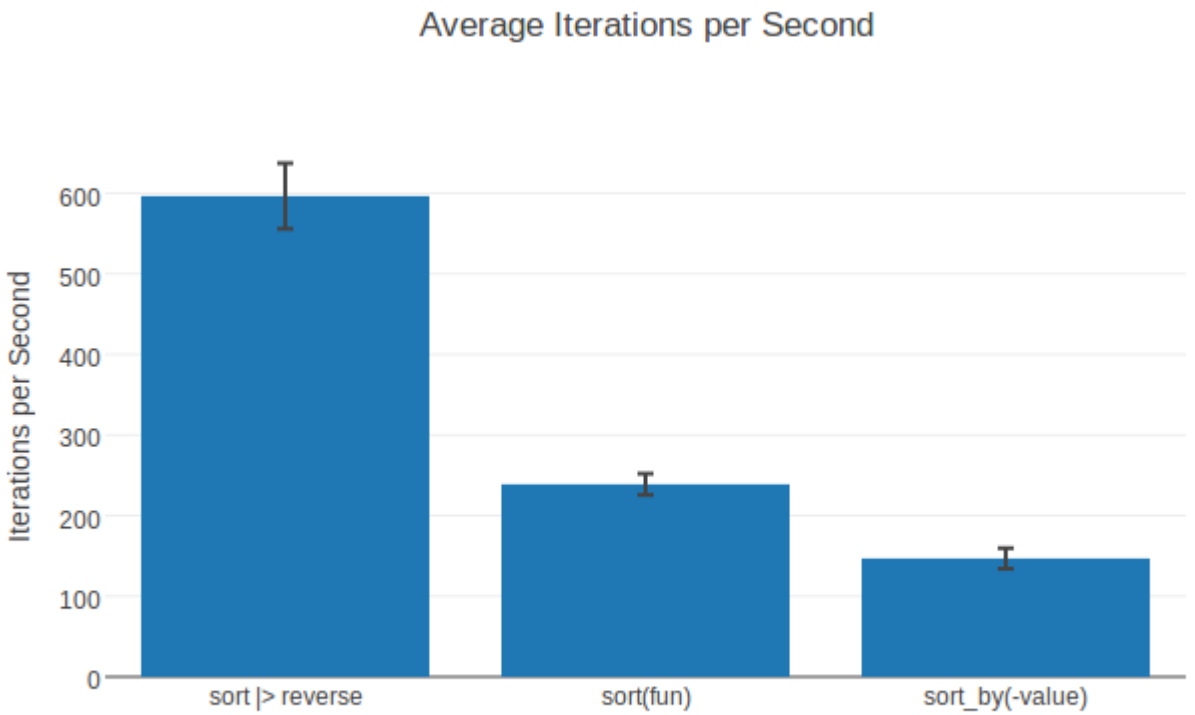
What's the **fastest** way to sort a list of numbers largest to smallest?

```elixir
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
}
```

```elixir
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
}
```

```elixir
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
}
```

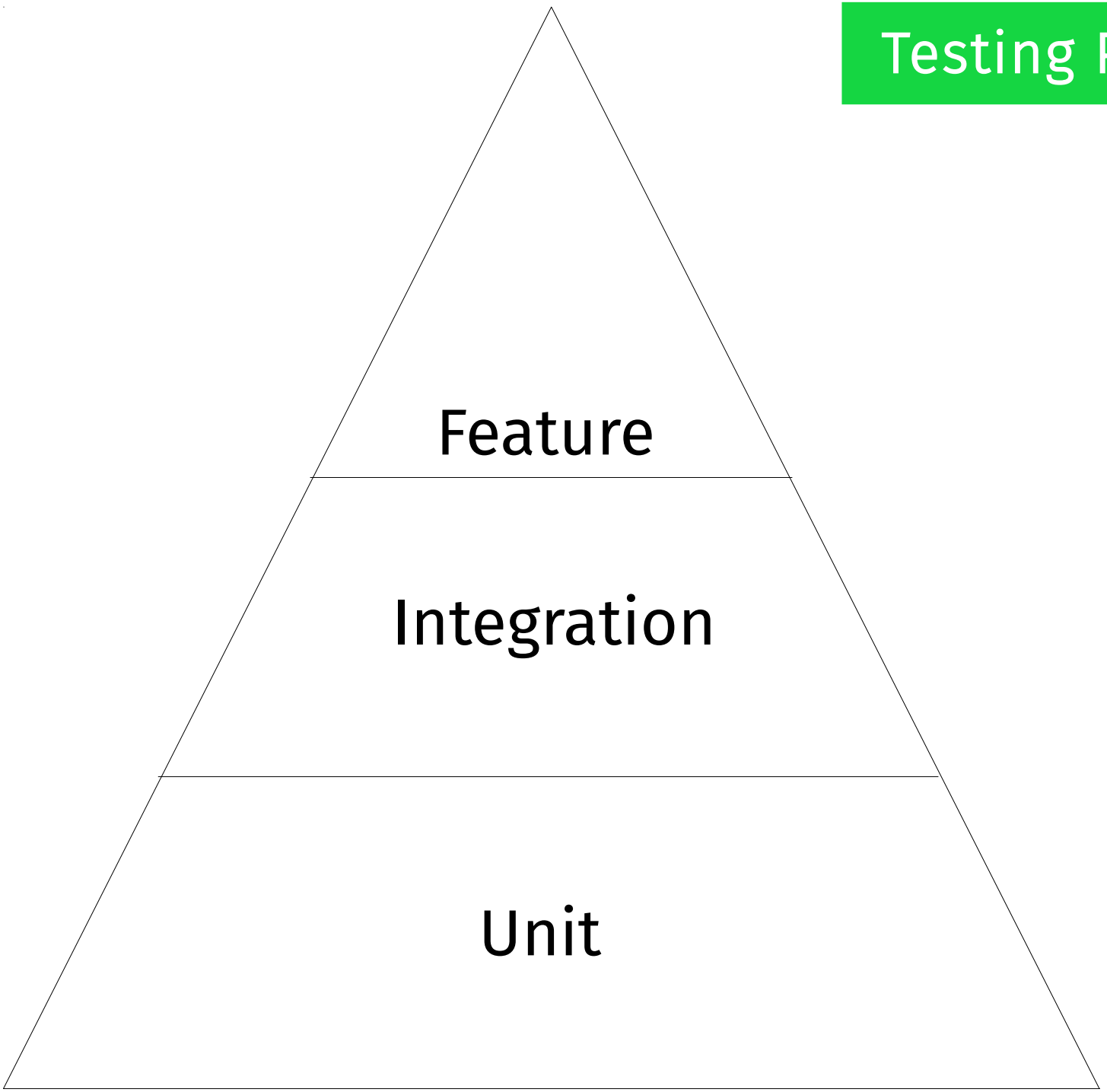
```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

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
}
```



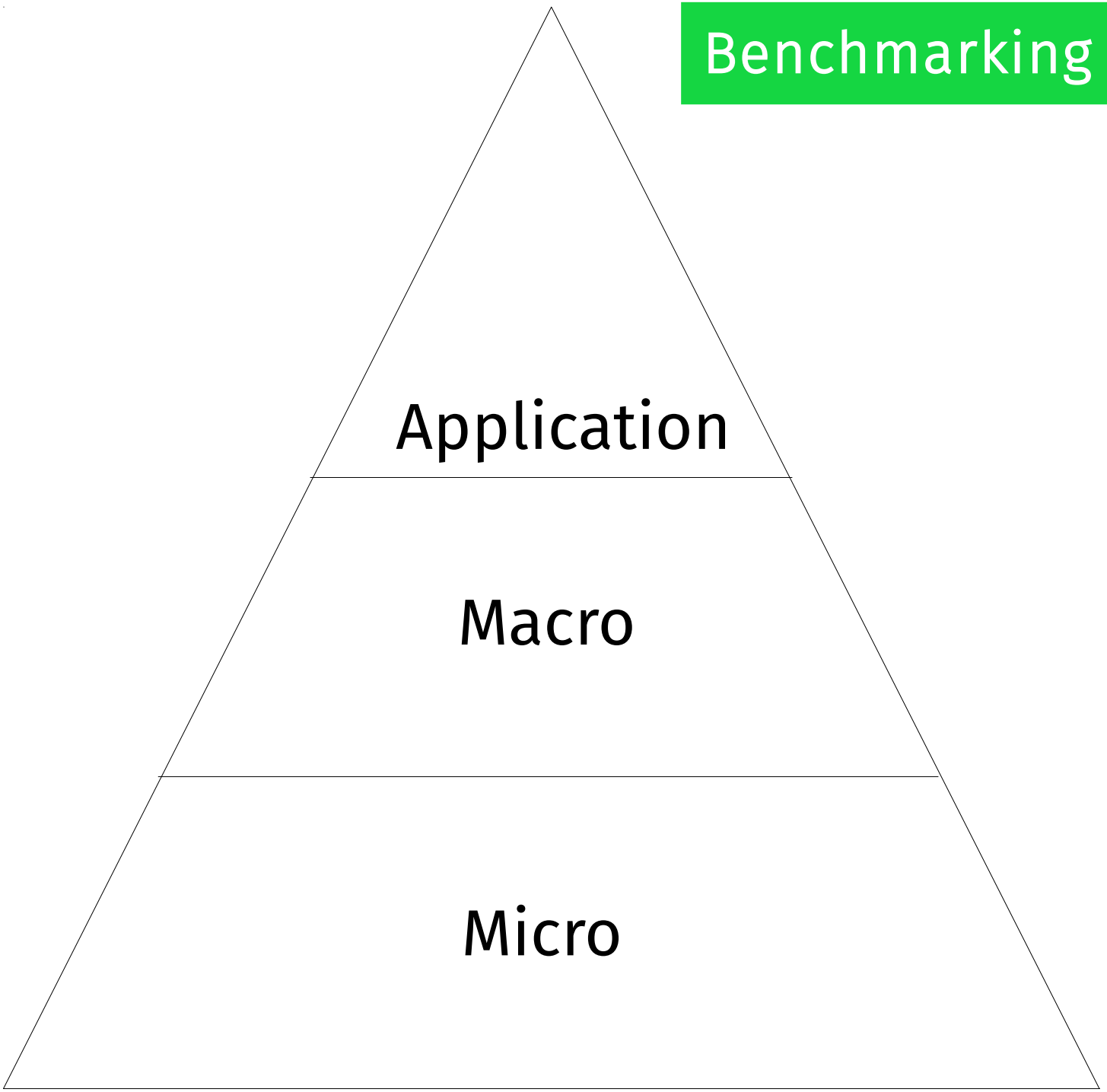

```
Name                          ips        average  deviation         median
sort |> reverse            596.54        1.68 ms     ±6.83%        1.65 ms
sort(fun)                  238.88        4.19 ms     ±5.53%        4.14 ms
sort_by(-value)            146.86        6.81 ms     ±8.68%        6.59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

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

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)

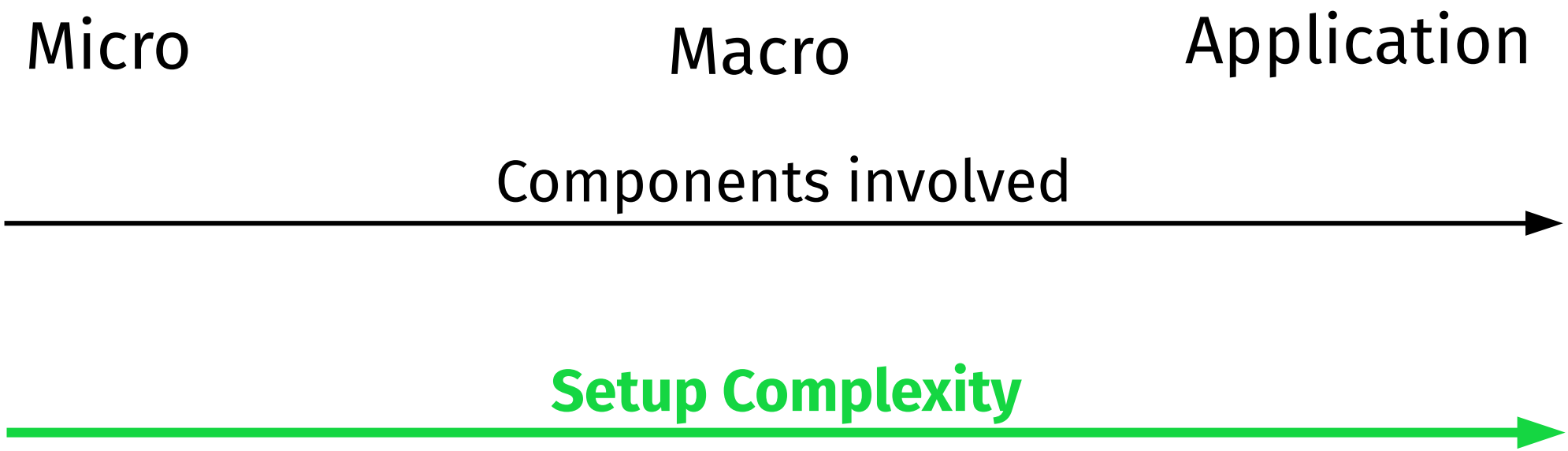
Framework	Performance (higher is better)	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA	Errors
fasthttp-postgresql-	3,050 100.0%	Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea	0
wt-postgres	2,945 96.6%	Ful	C++	Non	Non	Lin	Pg	Lin	Ful	Rea	0
revenj.jvm	2,873 94.2%	Ful	Jav	Svt	Res	Lin	Pg	Lin	Ful	Rea	0
express-mongodb	2,841 93.1%	Mcr	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

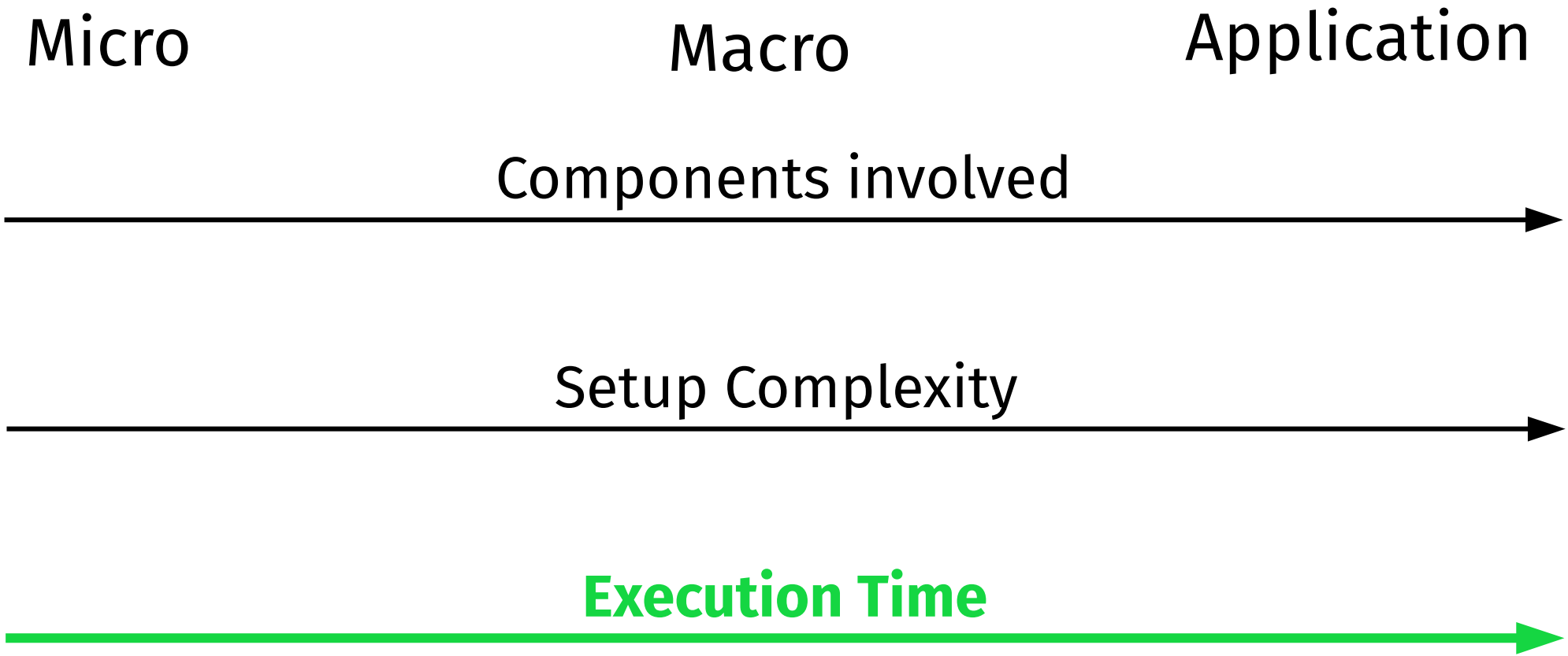
Micro

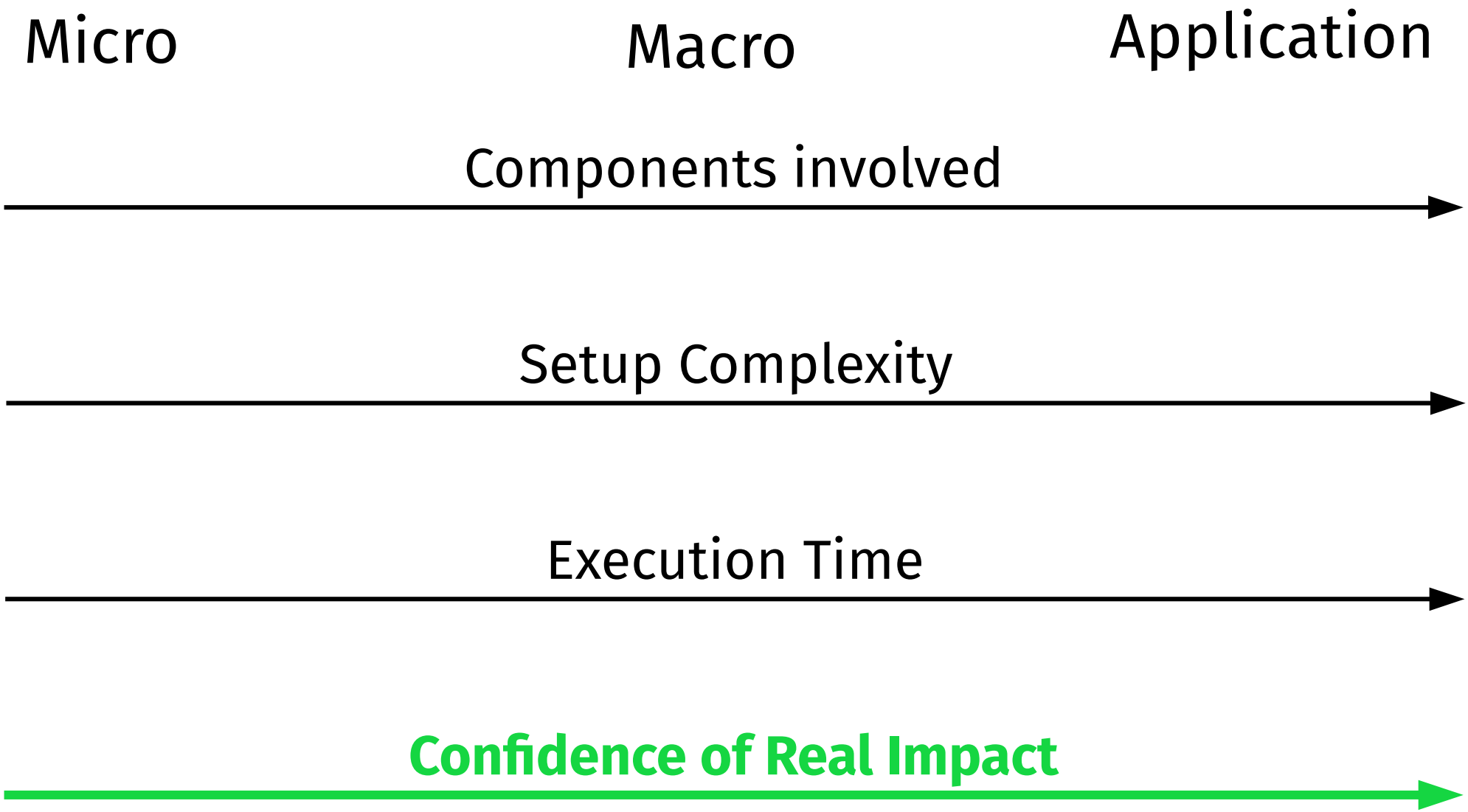
Macro

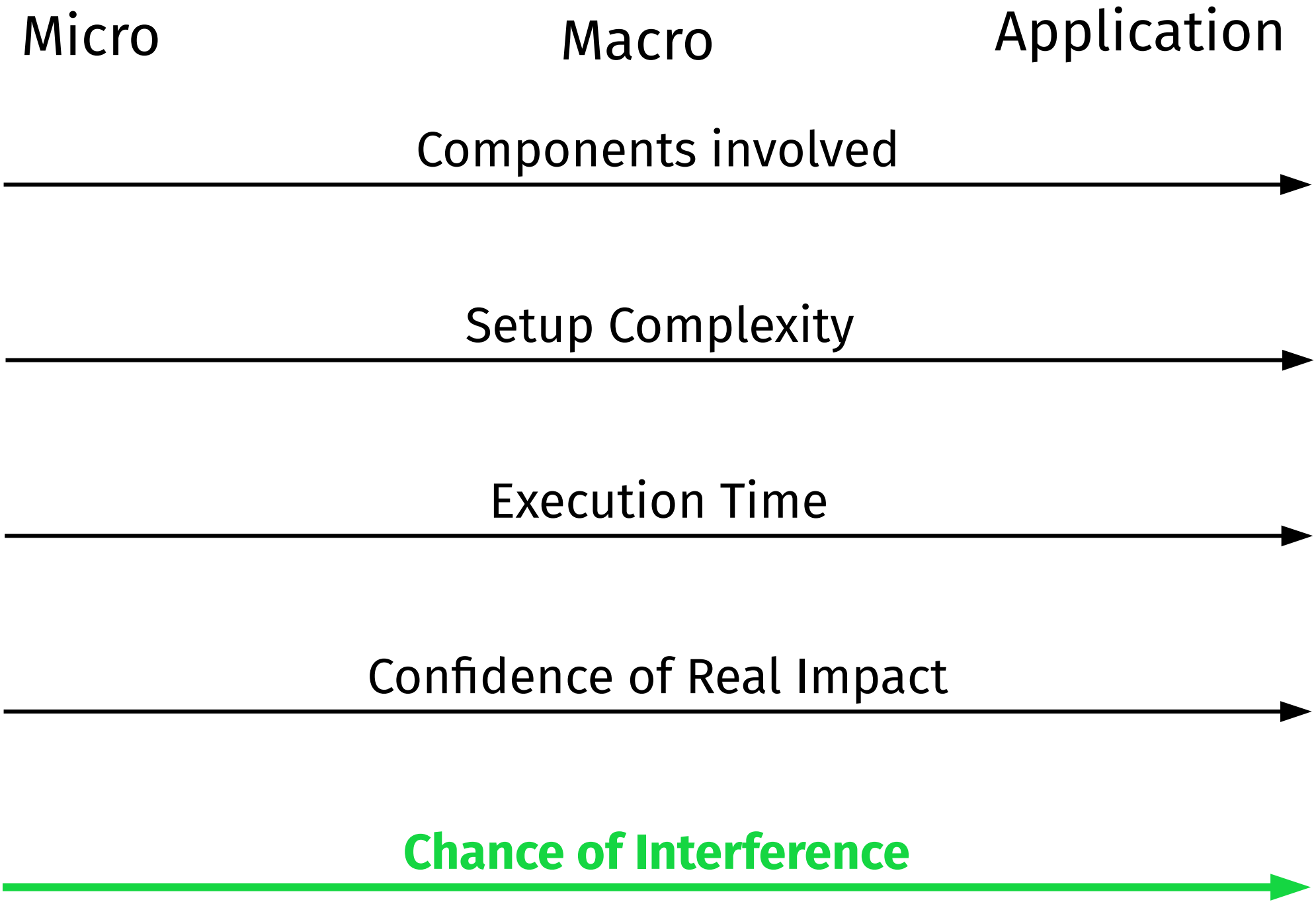
Application

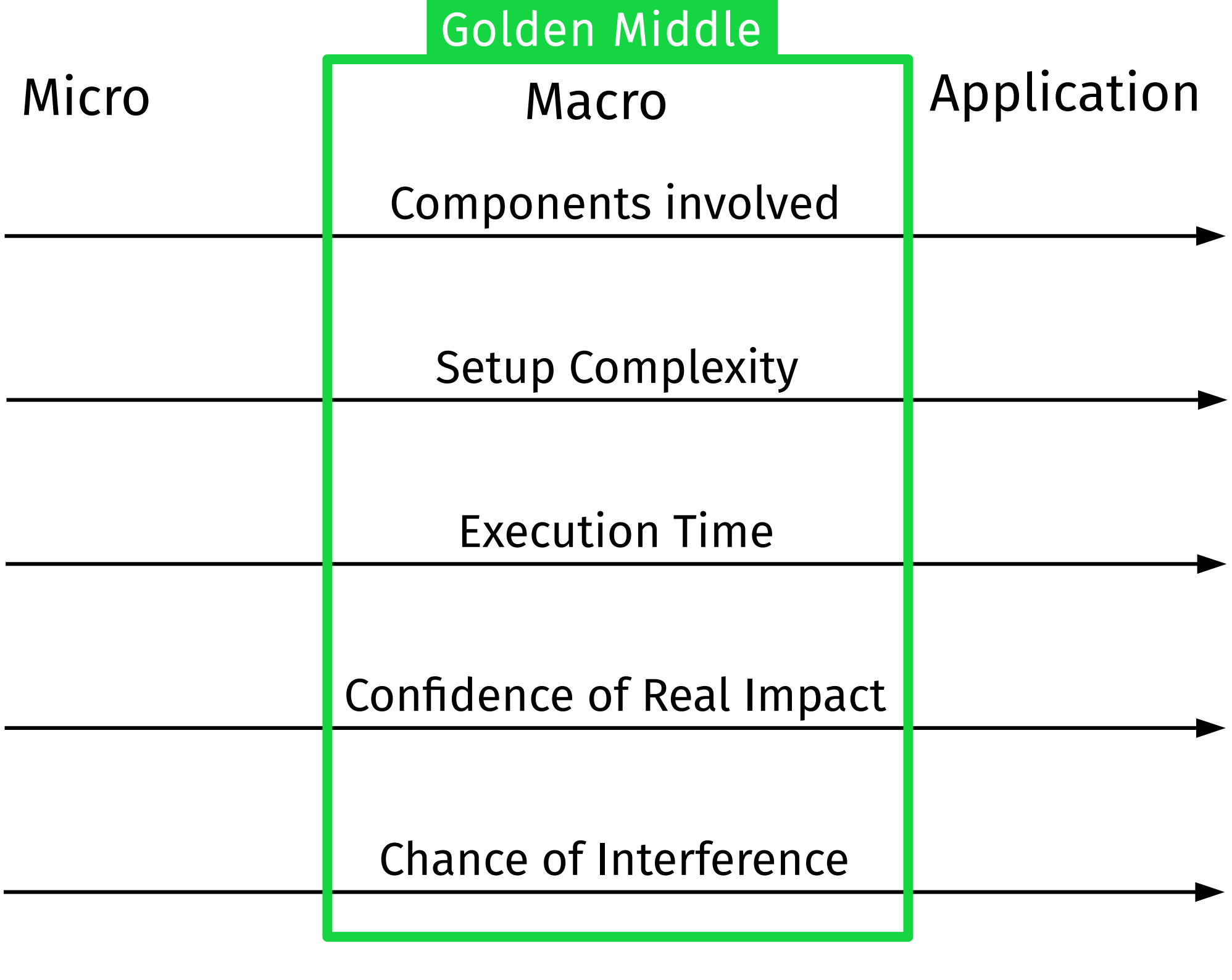

Micro
Macro
Application
Components involved

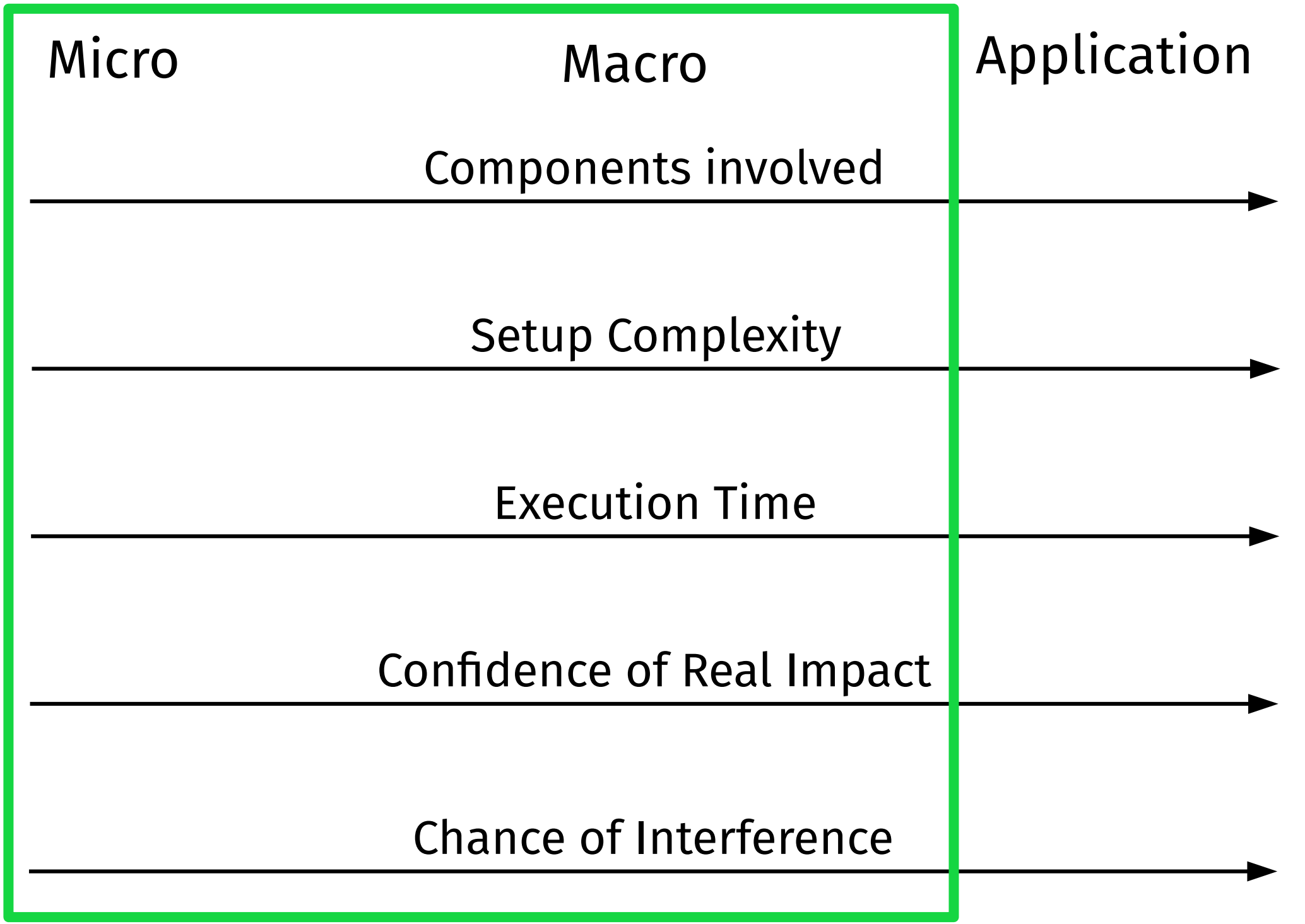

Micro
Macro
Application
Components involved
Setup Complexity


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

Good Benchmarking

What are you benchmarking for?

Overly **specific** benchmarks &
exaggerated results

System Specification

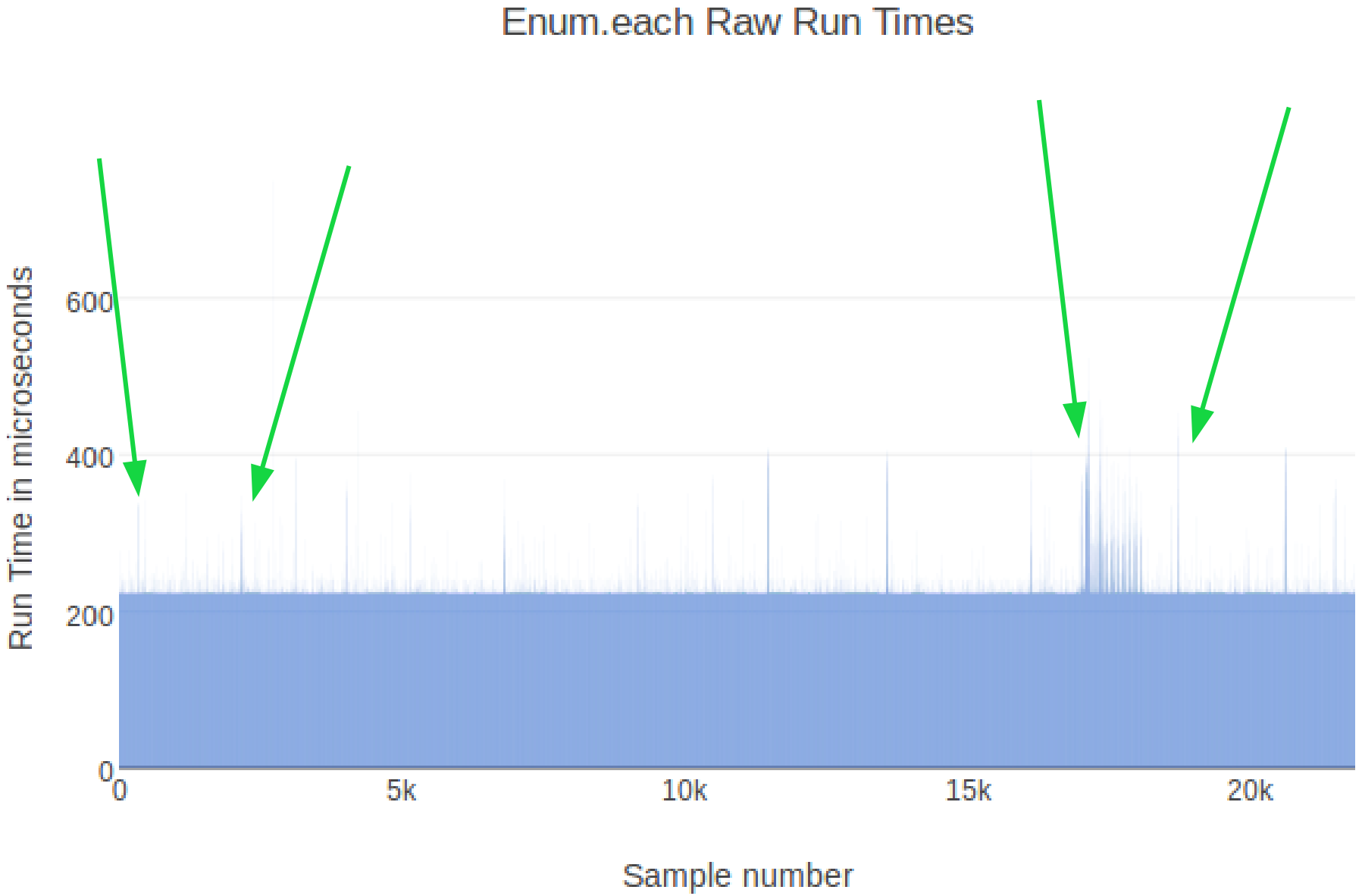
- Elixir 1.3.4
- Erlang 19.1
- i5-7200U – 2 x 2.5GHz (Up to 3.10GHz)
- 8GB RAM
- Linux Mint 18 - 64 bit (Ubuntu 16.04 base)
- Linux Kernel 4.4.0-51


Interference free Environment

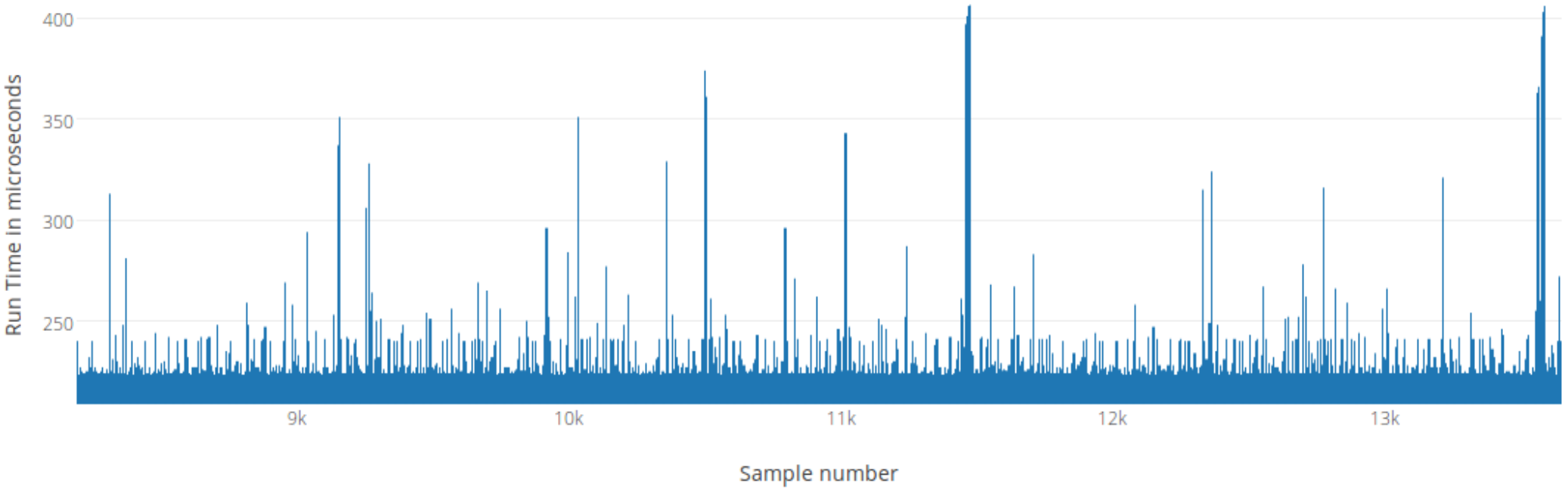
Logging & Friends

```
[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}}]
```


Garbage Collection


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

Zoom in


Run Time in microseconds
400
350
300
250
9k
10k
11k
12k
13k
Sample number


Correct & Meaningful Setup


Warmup


Inputs matter!

Malformed inputs

Where are your inputs

```
n    = 10_000
fun = fn -> 0 end

Benchee.run %{
  "Enum.each" => fn ->
     Enum.each(Enum.to_list(1..n), fn(_) -> fun.() end)
     end,
  "List comprehension" => fn ->
      for _ <- Enum.to_list(1..n), do: fun.()
    end,
  "Recursion" => fn -> RepeatN.repeat_n(fun, n) end
}
```

Executed every time

```
n    = 10_000
fun = fn -> 0 end

Benchee.run %{
  "Enum.each" => fn ->
     Enum.each(Enum.to_list(1..n), fn(_) -> fun.() end)
     end,
  "List comprehension" => fn ->
      for _ <- Enum.to_list(1..n), do: fun.()
    end,
  "Recursion" => fn -> RepeatN.repeat_n(fun, n) end
}
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end

  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

TCO

```elixir
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end

  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

TCO

```
alias Benchee.Formatters.{Console, HTML}

map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"   => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"       => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"    => Enum.to_list(1..5_000_000)
}

Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end
}, time: 20, warmup: 10, inputs: inputs,
   formatters: [&Console.output/1, &HTML.output/1],
   html: [file: "bench/output/tco_small_sample.html"]
```

```
alias Benchee.Formatters.{Console, HTML}

map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"   => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"       => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"    => Enum.to_list(1..5_000_000)
}

Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end
}, time: 20, warmup: 10, inputs: inputs,
   formatters: [&Console.output/1, &HTML.output/1],
   html: [file: "bench/output/tco_small_sample.html"]
```

TCO

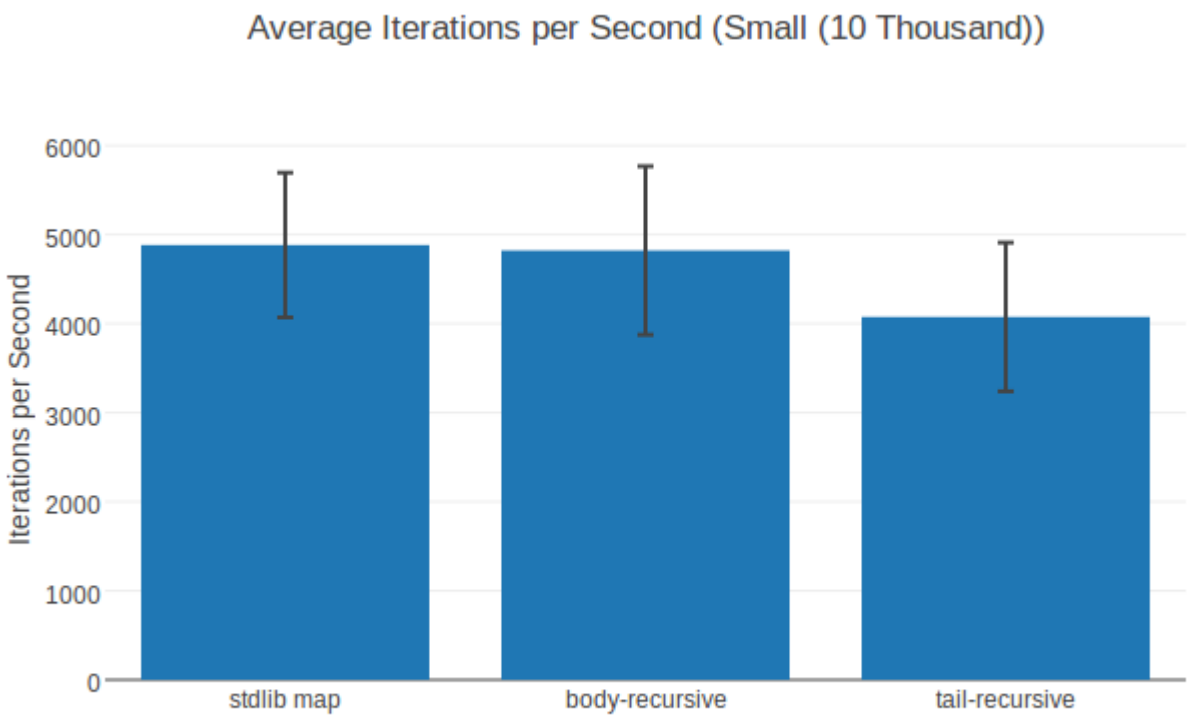
TCO

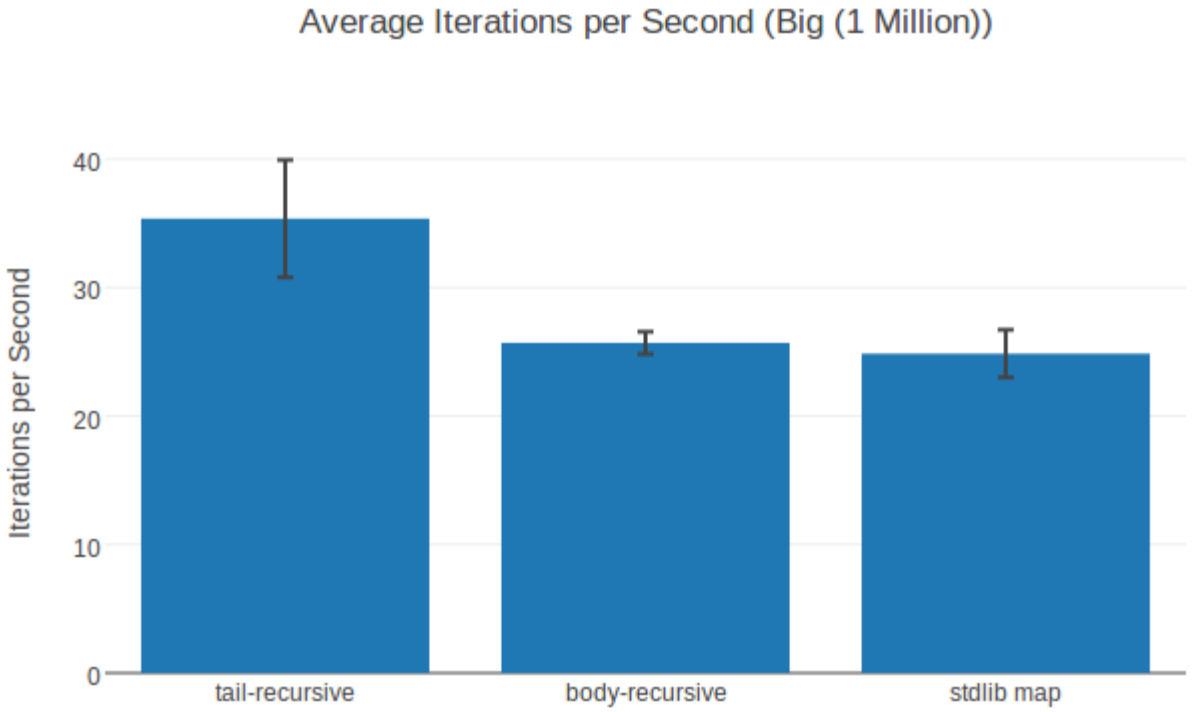
```
##### With input Small (10 Thousand) #####
Comparison:
body-recursive         5.12 K
stdlib map             5.07 K - 1.01x slower
tail-recursive         4.38 K - 1.17x slower

##### With input Middle (100 Thousand) #####
Comparison:
body-recursive         491.16
stdlib map             488.45 - 1.01x slower
tail-recursive         399.08 - 1.23x slower

##### With input Big (1 Million) #####
Comparison:
tail-recursive          35.36
body-recursive          25.69 - 1.38x slower
stdlib map              24.85 - 1.42x slower

##### With input Bigger (5 Million) #####
Comparison:
tail-recursive           6.93
body-recursive           4.92 - 1.41x slower
stdlib map               4.87 - 1.42x slower
```

TCO

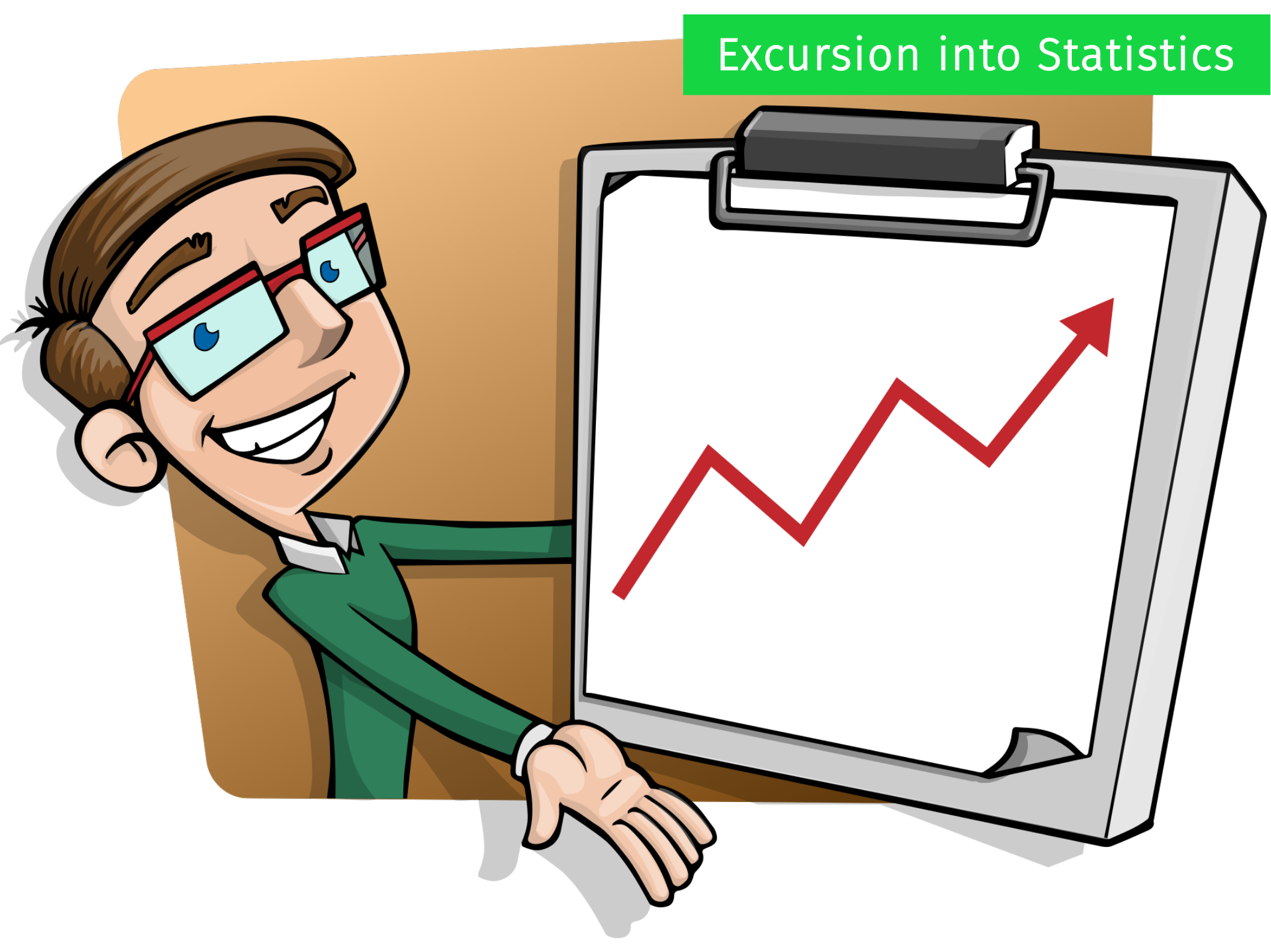
```
##### With input Small (10 Thousand) #####
Comparison:
body-recursive          5.12 K
stdlib map              5.07 K - 1.01x slower
tail-recursive          4.38 K - 1.17x slower

##### With input Middle (100 Thousand) #####
Comparison:
body-recursive          491.16
stdlib map              488.45 - 1.01x slower
tail-recursive          399.08 - 1.23x slower

##### With input Big (1 Million) #####
Comparison:
tail-recursive           35.36
body-recursive           25.69 - 1.38x slower
stdlib map               24.85 - 1.42x slower

##### With input Bigger (5 Million) #####
Comparison:
tail-recursive            6.93
body-recursive            4.92 - 1.41x slower
stdlib map                4.87 - 1.42x slower
```

TCO


Average Iterations per Second (Small (10 Thousand))
Iterations per Second
6000
5000
4000
3000
2000
1000
0
stdlib map
body-recursive
tail-recursive


Average Iterations per Second (Big (1 Million))
Iterations per Second
40
30
20
10
0
tail-recursive
body-recursive
stdlib map

Excursion into Statistics



Average

```
average = total_time / iterations
```

Why not just take the **average**?

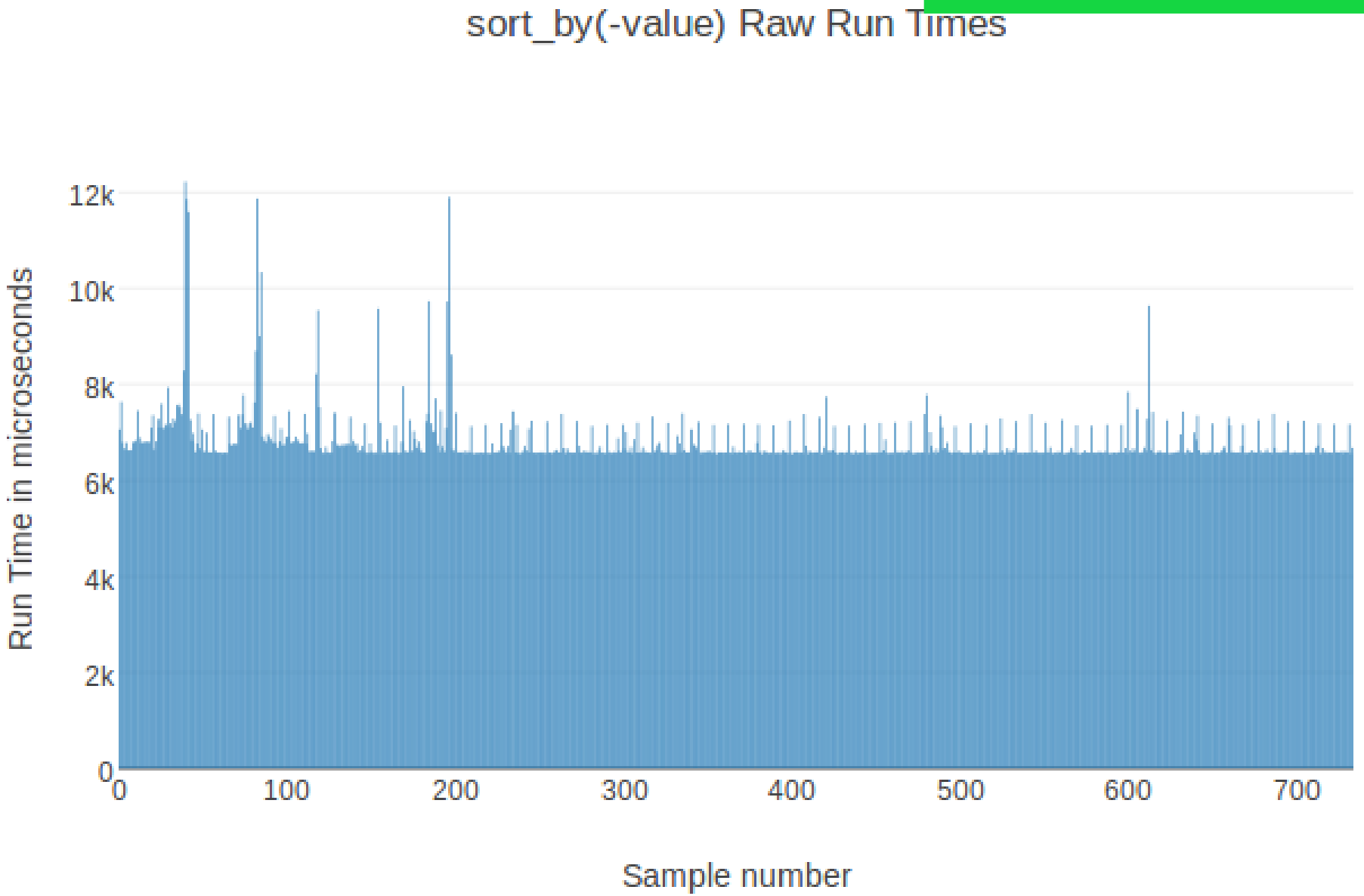
Standard Deviation

```
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
```

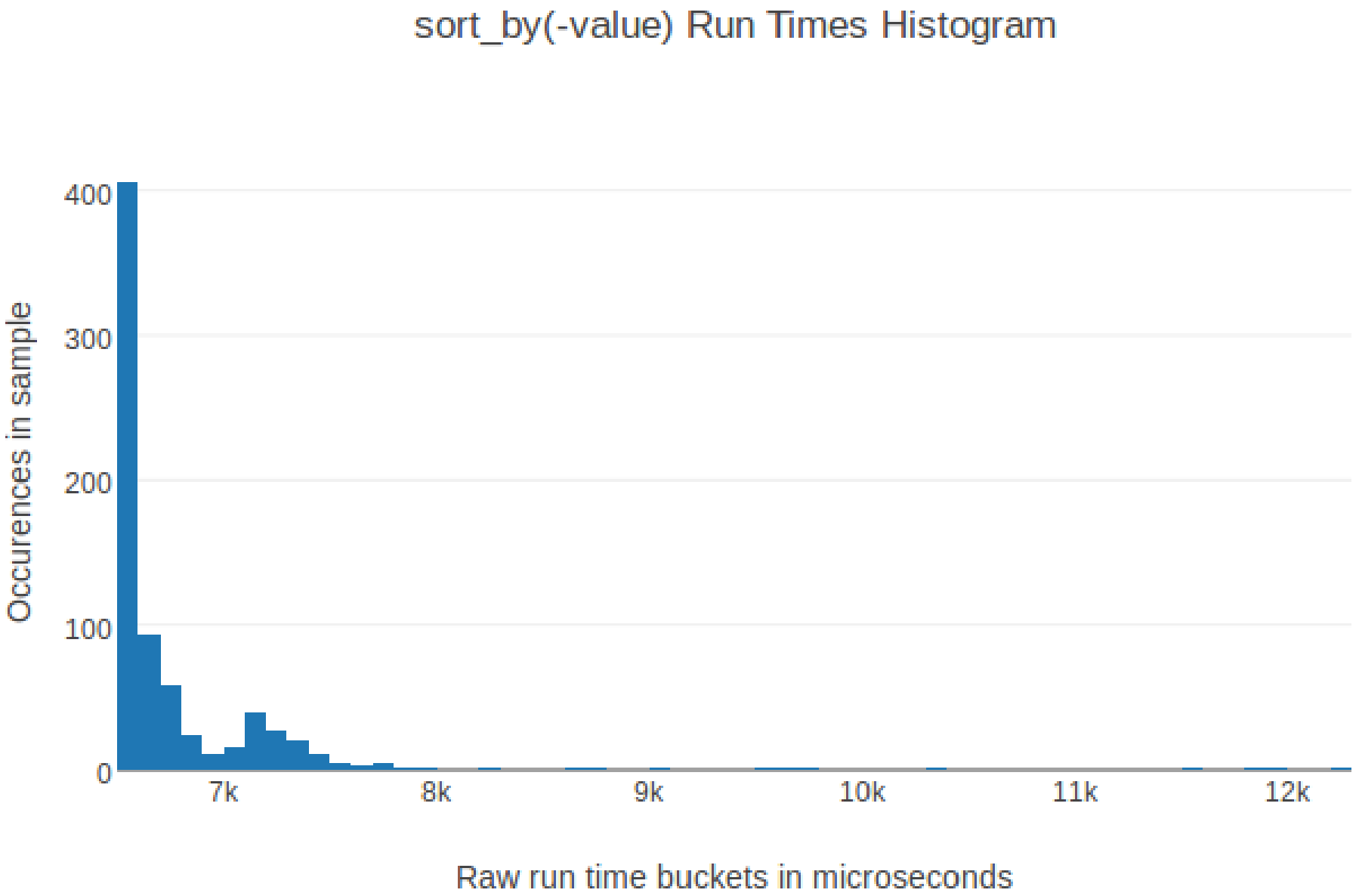
Spread of Values

```
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
```

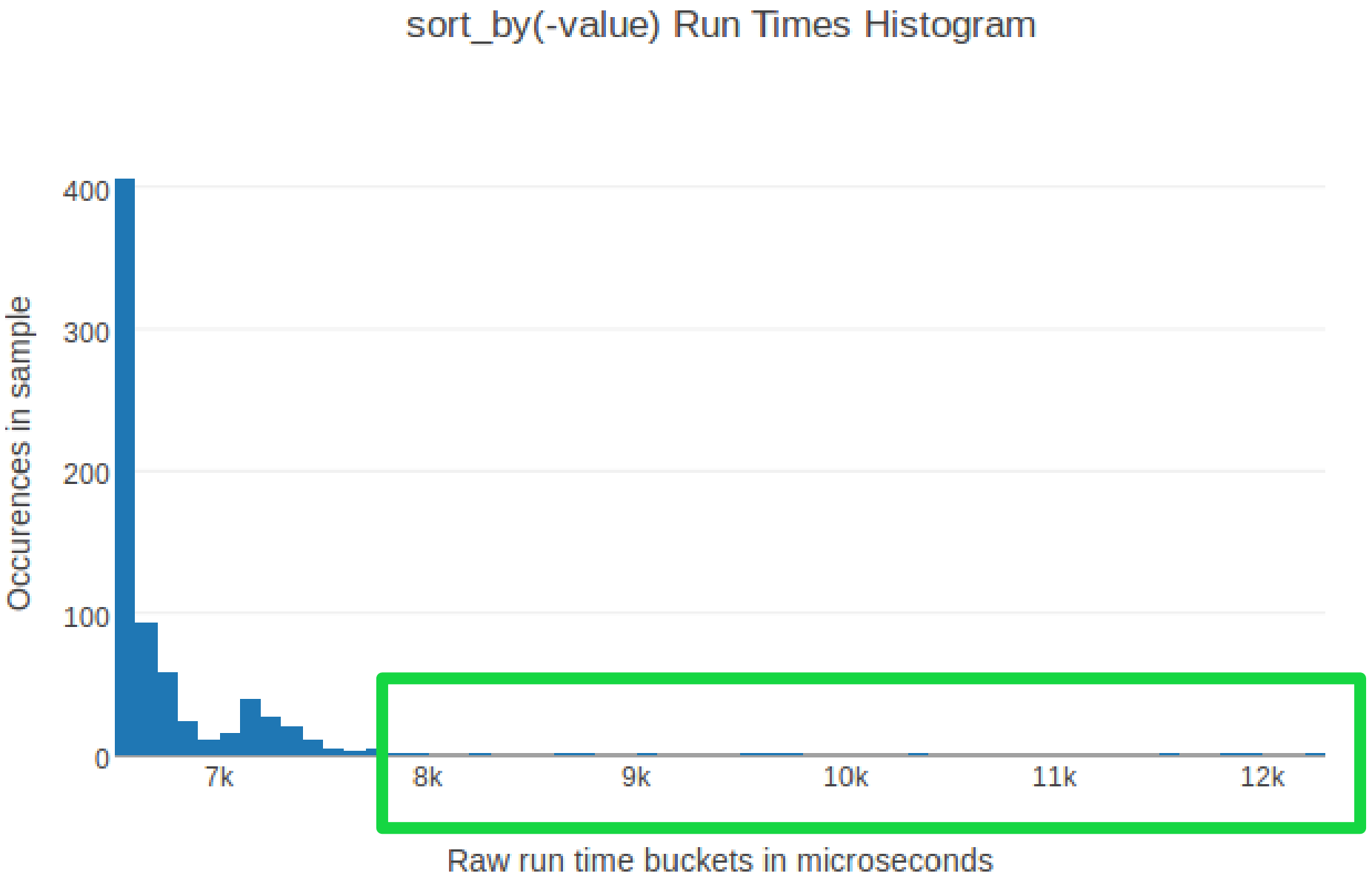
Raw Run Times



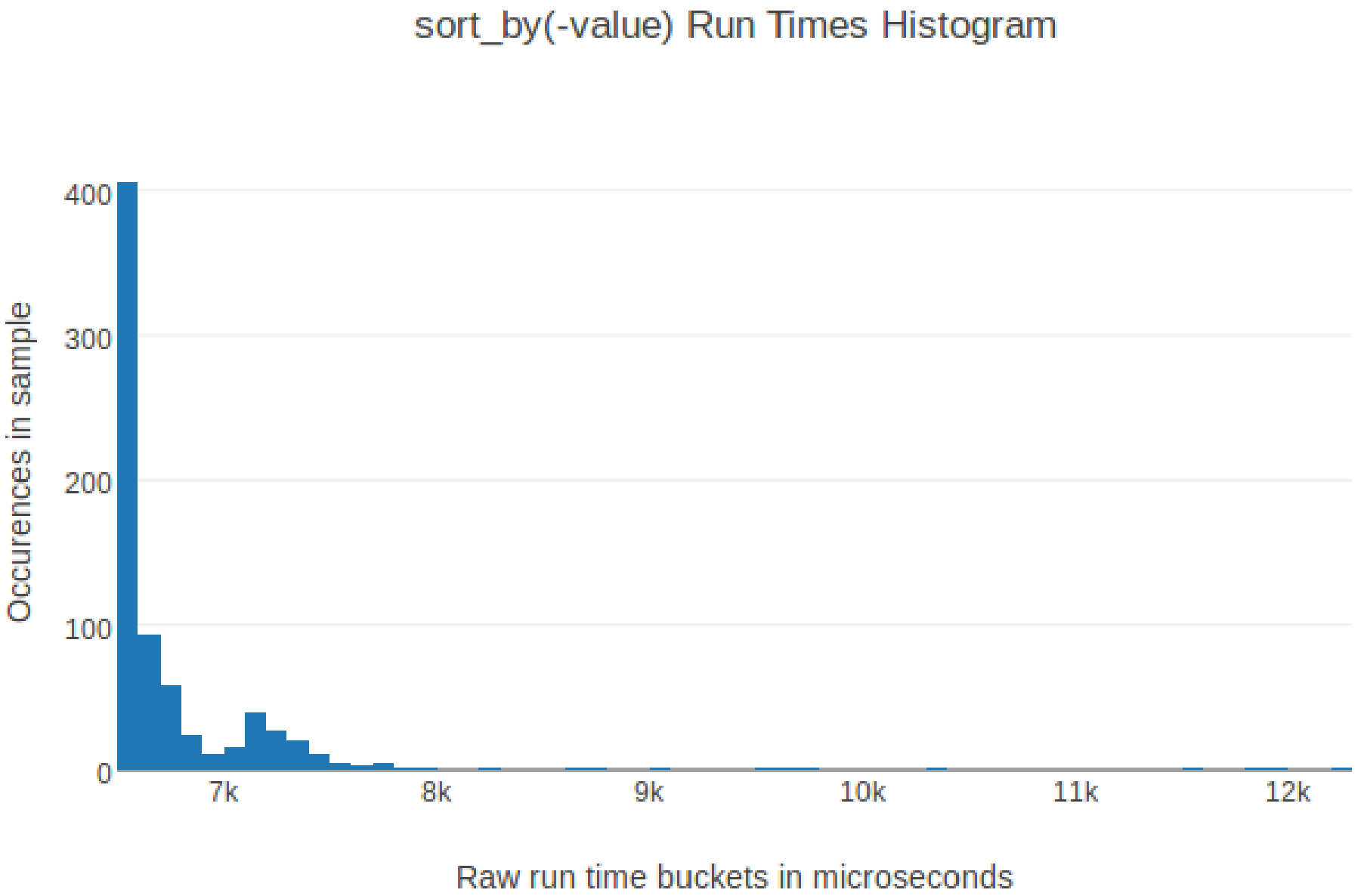
Histogram

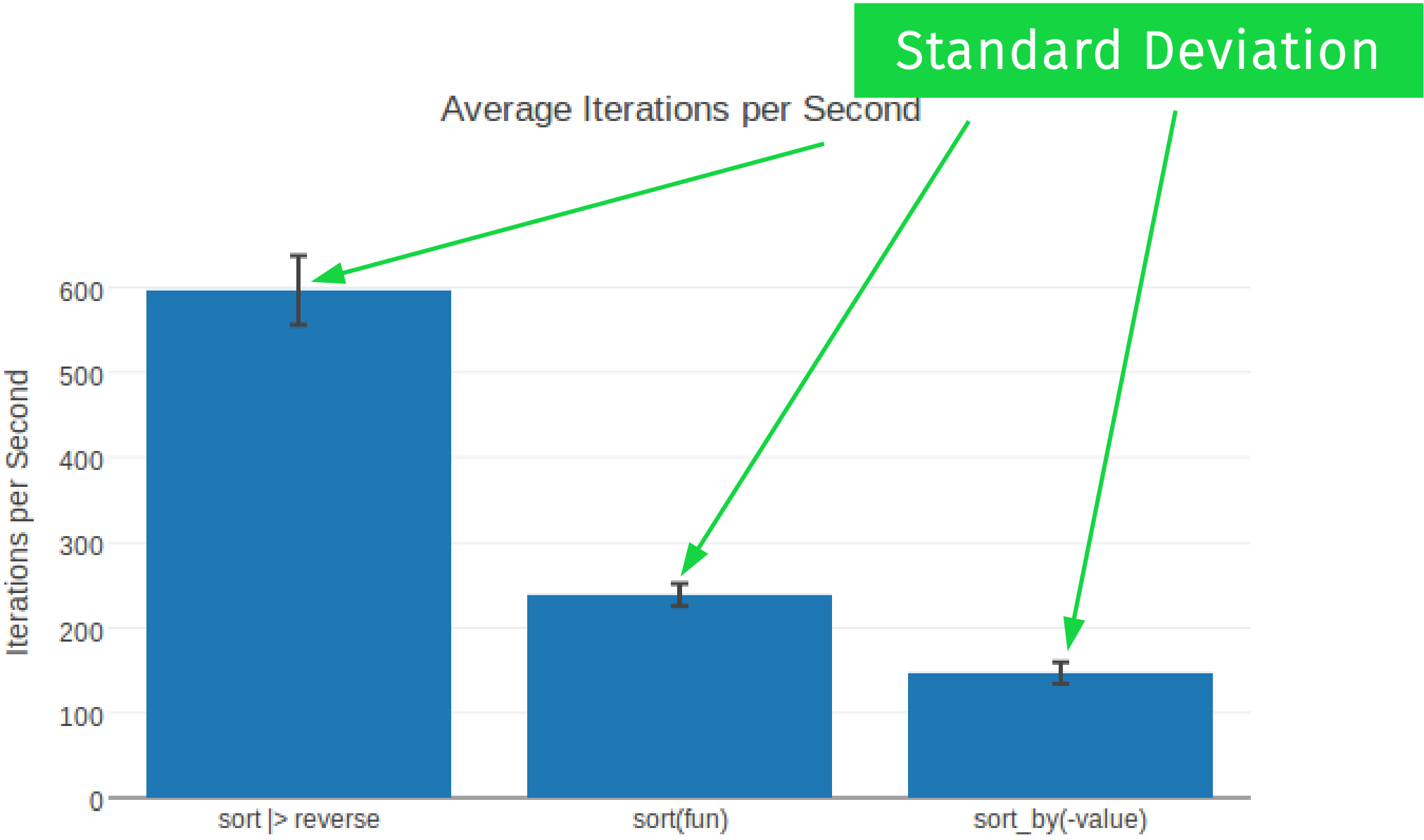

sort_by(-value) Run Times Histogram
Occurences in sample
400
300
200
100
0
7k
8k
9k
10k
11k
12k
Raw run time buckets in microseconds

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

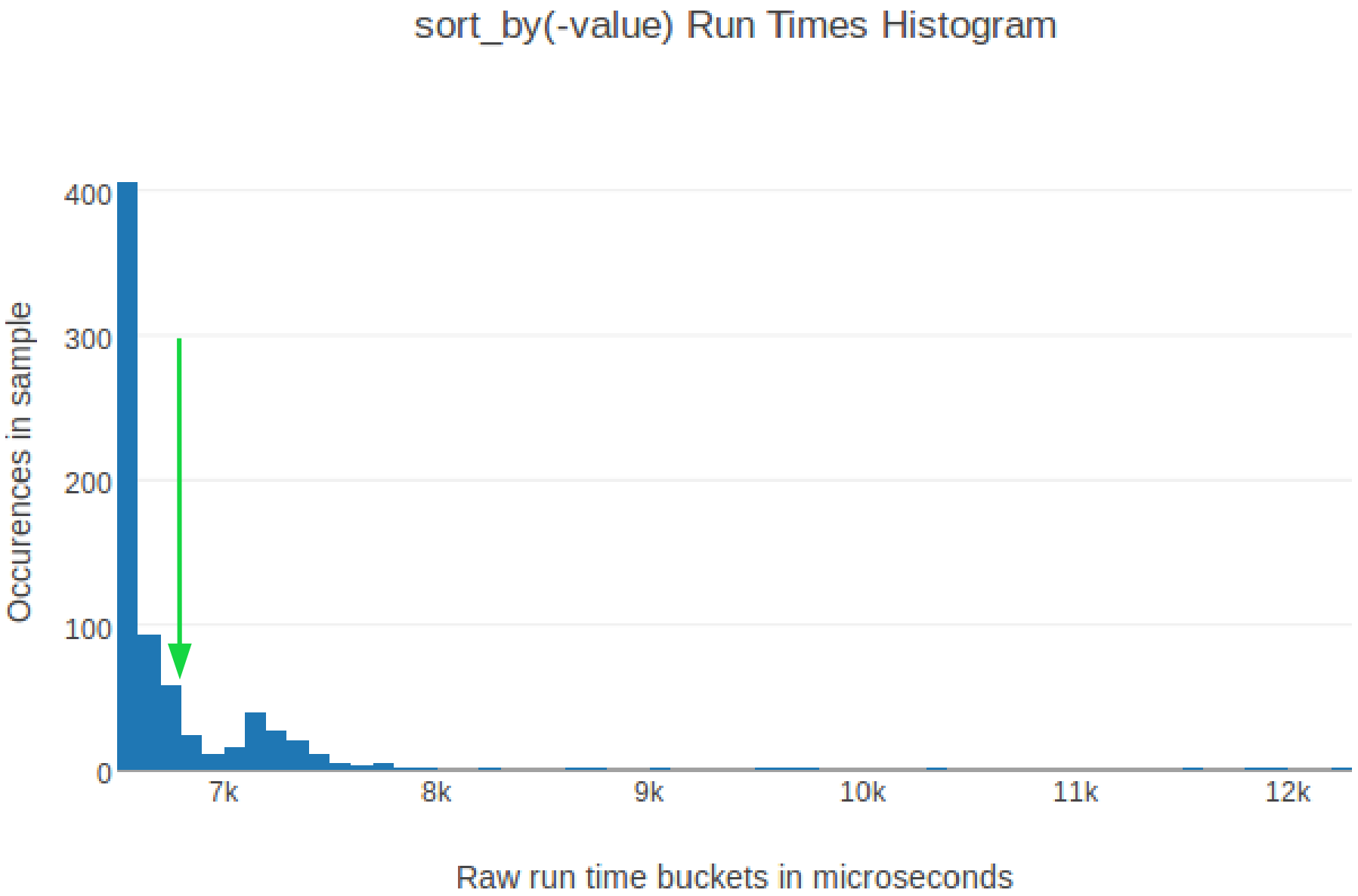



Standard Deviation
Average Iterations per Second
Iterations per Second
600
500
400
300
200
100
0
sort |> reverse
sort(fun)
sort_by(-value)

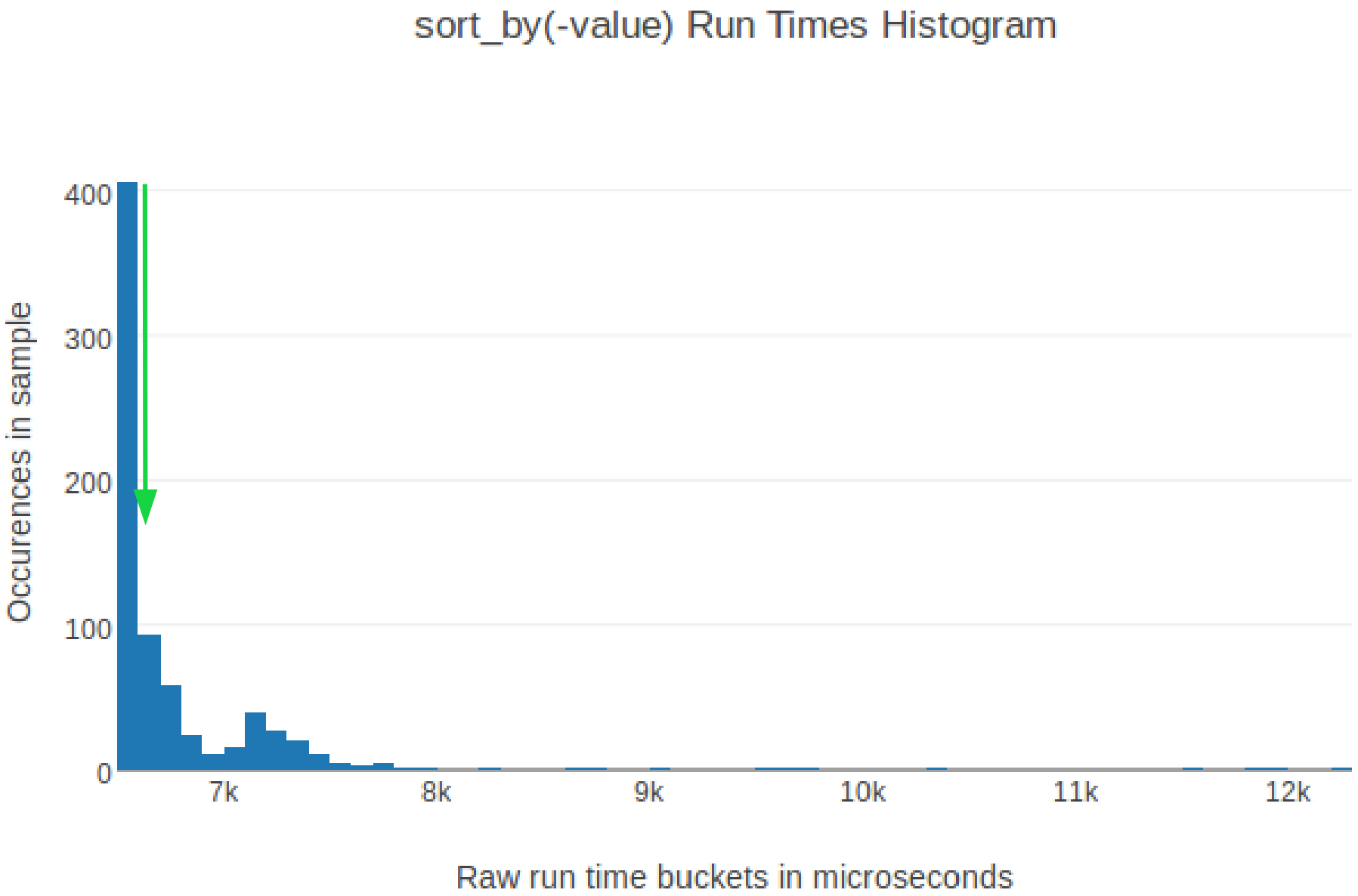
Median

```
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
```

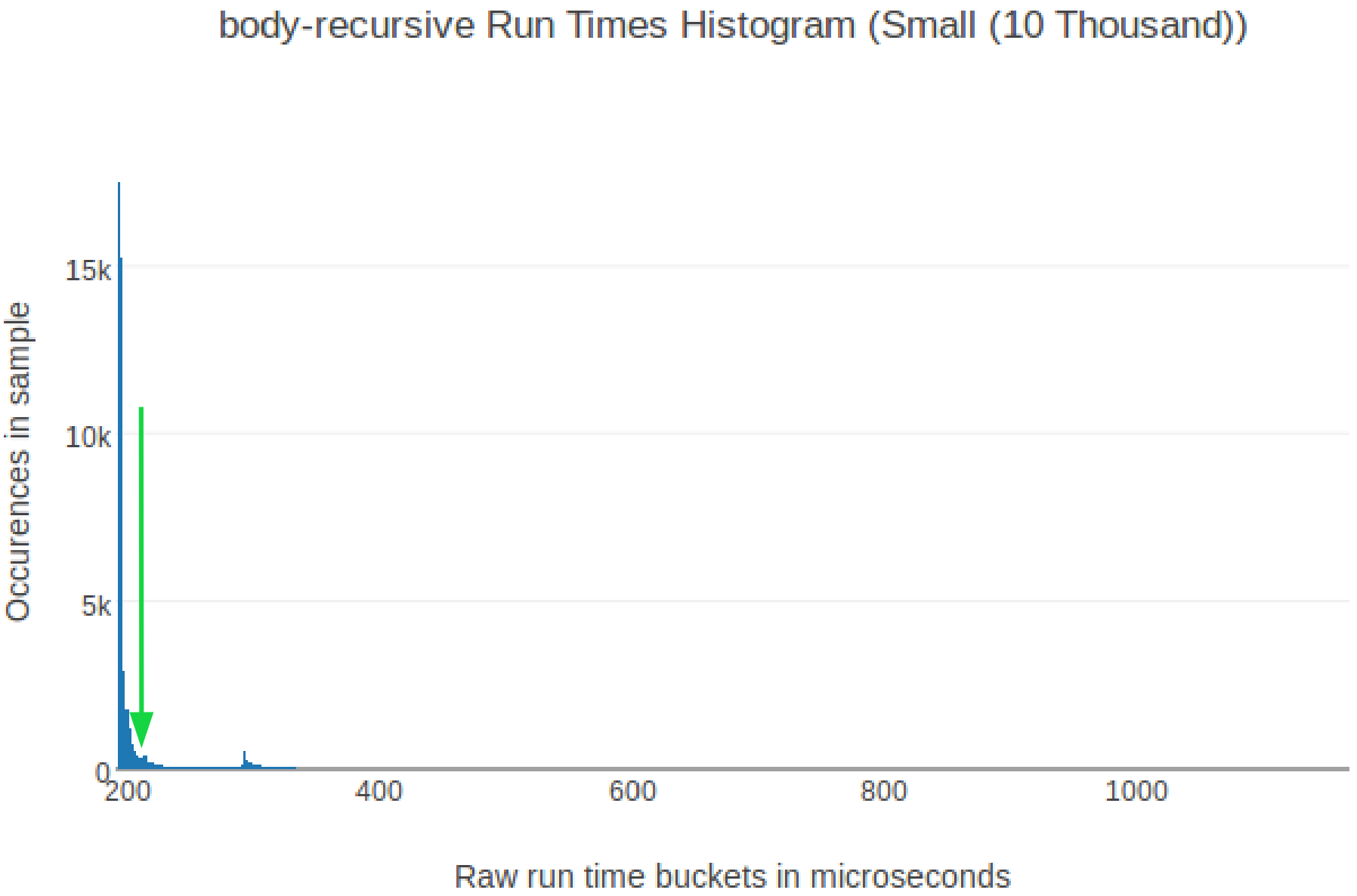
Average

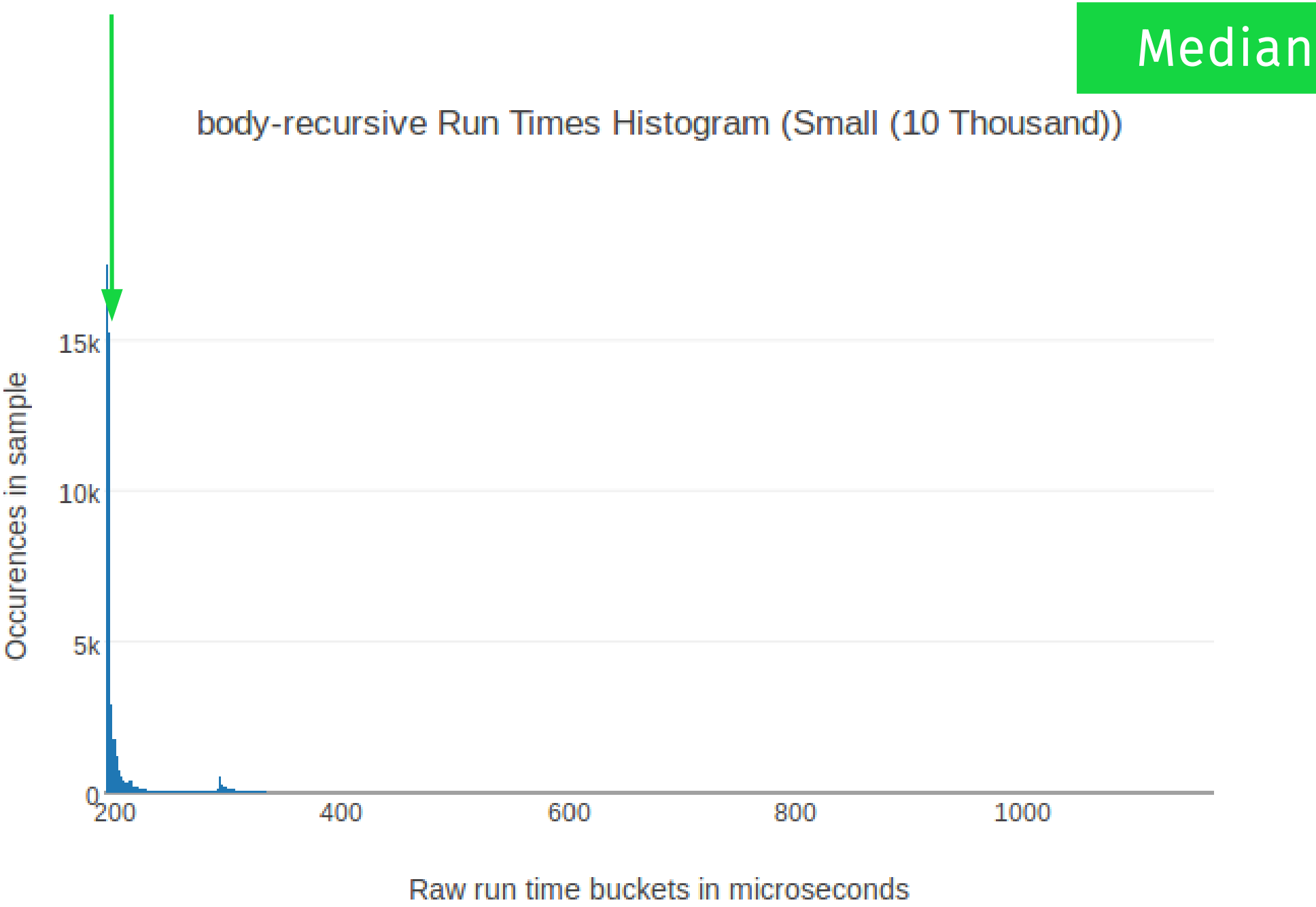

sort_by(-value) Run Times Histogram
Occurences in sample
400
300
200
100
0
7k
8k
9k
10k
11k
12k
Raw run time buckets in microseconds

Median


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

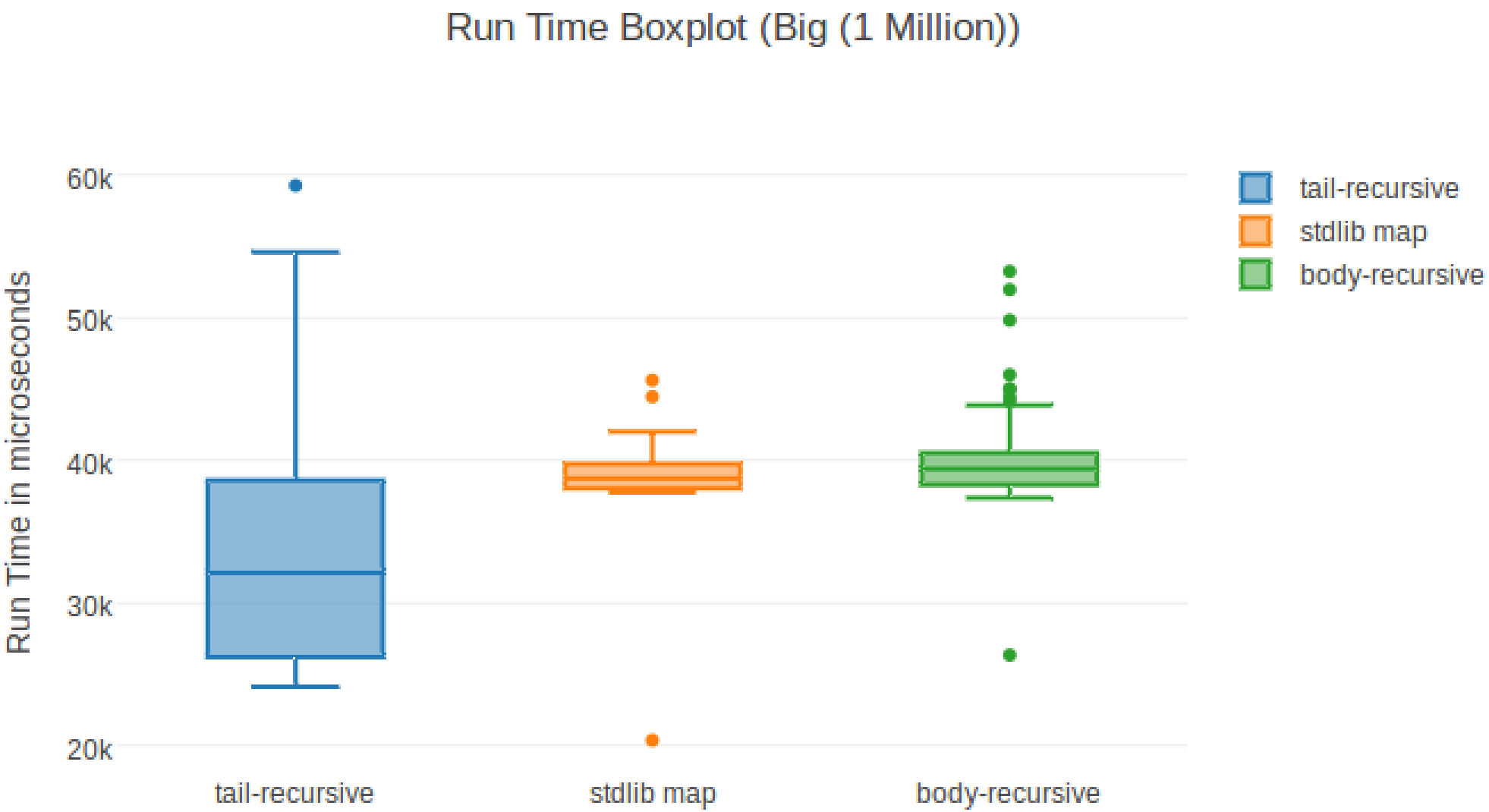
Average


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

Minimum & Maximum

Boxplot

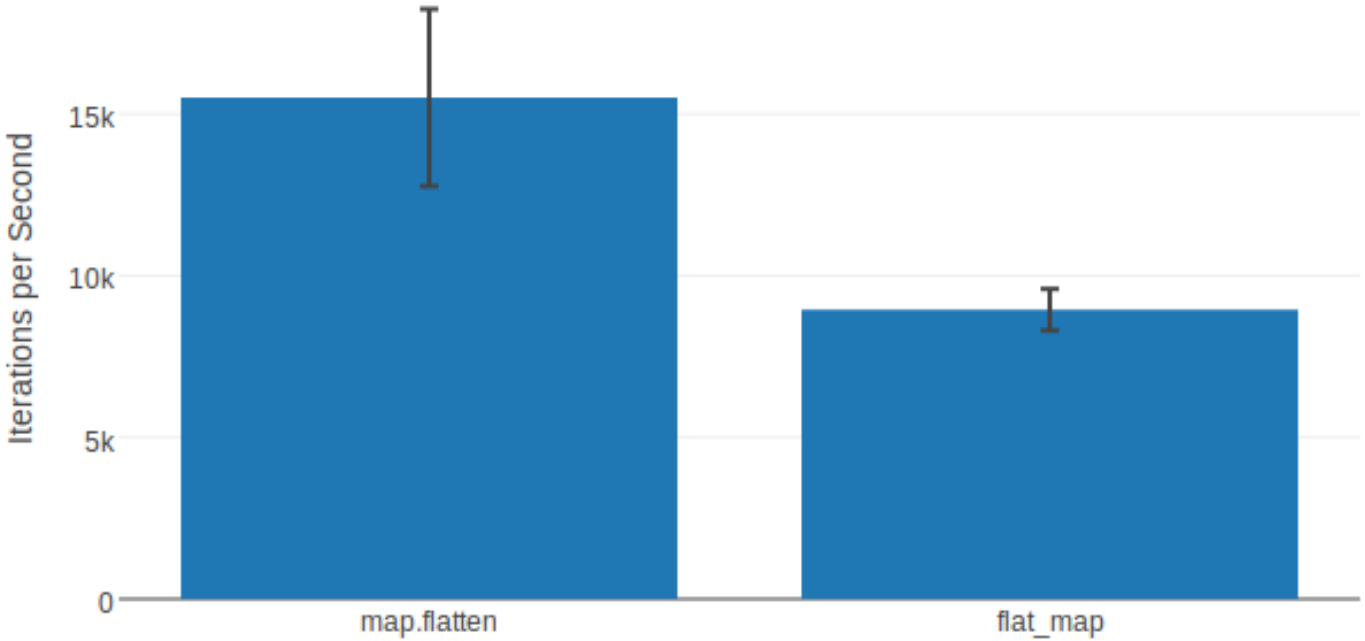

Run Time Boxplot (Big (1 Million))
Run Time in microseconds
60k
50k
40k
30k
20k
tail-recursive
stdlib map
body-recursive
tail-recursive
stdlib map
body-recursive

Surprise findings

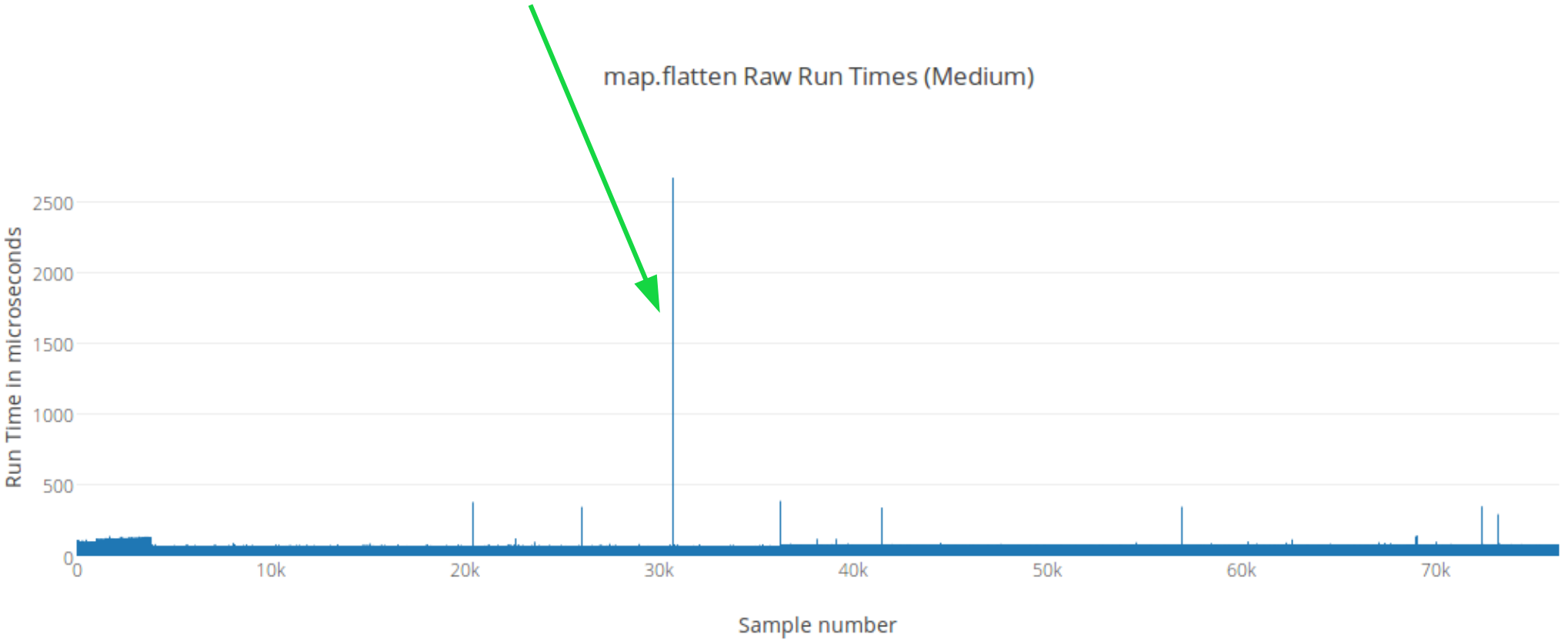
flat_map

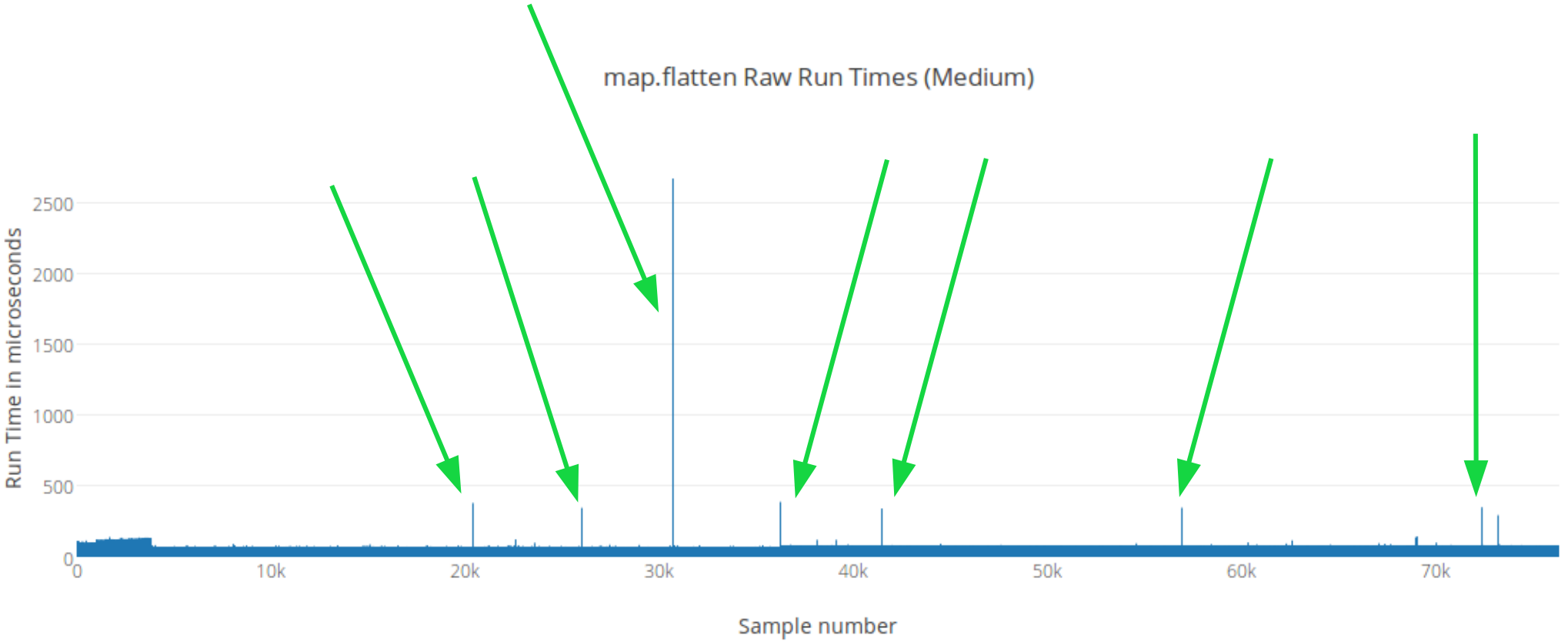
```
alias Benchee.Formatters.{Console, HTML}
map_fun = fn(i) -> [i, i * i] end

inputs = %{
  "Small"  => Enum.to_list(1..200),
  "Medium" => Enum.to_list(1..1000),
  "Bigger" => Enum.to_list(1..10_000)
}

Benchee.run(%{
  "flat_map" =>
    fn(list) -> Enum.flat_map(list, map_fun) end,
  "map.flatten" => fn(list) ->
    list
    |> Enum.map(map_fun)
    |> List.flatten
  end
}, inputs: inputs,
   formatters: [&Console.output/1, &HTML.output/1],
   html: [file: "bench/output/flat_map.html"])
```

flat_map




```
##### With input Medium #####
Name                     ips        average  deviation         median
map.flatten          15.51 K       64.48 µs    ±17.66%       63.00 µs
flat_map              8.95 K      111.76 µs     ±7.18%      112.00 µs

Comparison:
map.flatten          15.51 K
flat_map              8.95 K - 1.73x slower
```


map.flatten Raw Run Times (Medium)
Run Time in microseconds
2500
2000
1500
1000
500
0
0
10k
20k
30k
40k
50k
60k
70k
Sample number


map.flatten Raw Run Times (Medium)
Run Time in microseconds
2500
2000
1500
1000
500
0
0
10k
20k
30k
40k
50k
60k
70k
Sample number

merge/2 vs merge/3

```
base_map = (0..50)
           |> Enum.zip(300..350)
           |> Enum.into(%{})

# deep maps with 6 top level conflicts
orig = Map.merge base_map, some_deep_map
new = Map.merge base_map, some_deep_map_2

simple = fn(_key, _base, override) -> override end

Benchee.run %{
  "Map.merge/2" => fn -> Map.merge orig, new end,
  "Map.merge/3" =>
    fn -> Map.merge orig, new, simple end,
}, formatters: [&Benchee.Formatters.Console.output/1,
                &Benchee.Formatters.HTML.output/1],
   html: %{file: "bench/output/merge_3.html"}
```

merge/2 vs merge/3

```
base_map = (0..50)
           |> Enum.zip(300..350)
           |> Enum.into(%{})

# deep maps with 6 top level conflicts
orig = Map.merge base_map, some_deep_map
new = Map.merge base_map, some_deep_map_2

simple = fn(_key, _base, override) -> override end

Benchee.run %{
  "Map.merge/2" => fn -> Map.merge orig, new end,
  "Map.merge/3" =>
    fn -> Map.merge orig, new, simple end,
}, formatters: [&Benchee.Formatters.Console.output/1,
                &Benchee.Formatters.HTML.output/1],
   html: %{file: "bench/output/merge_3.html"}
```

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

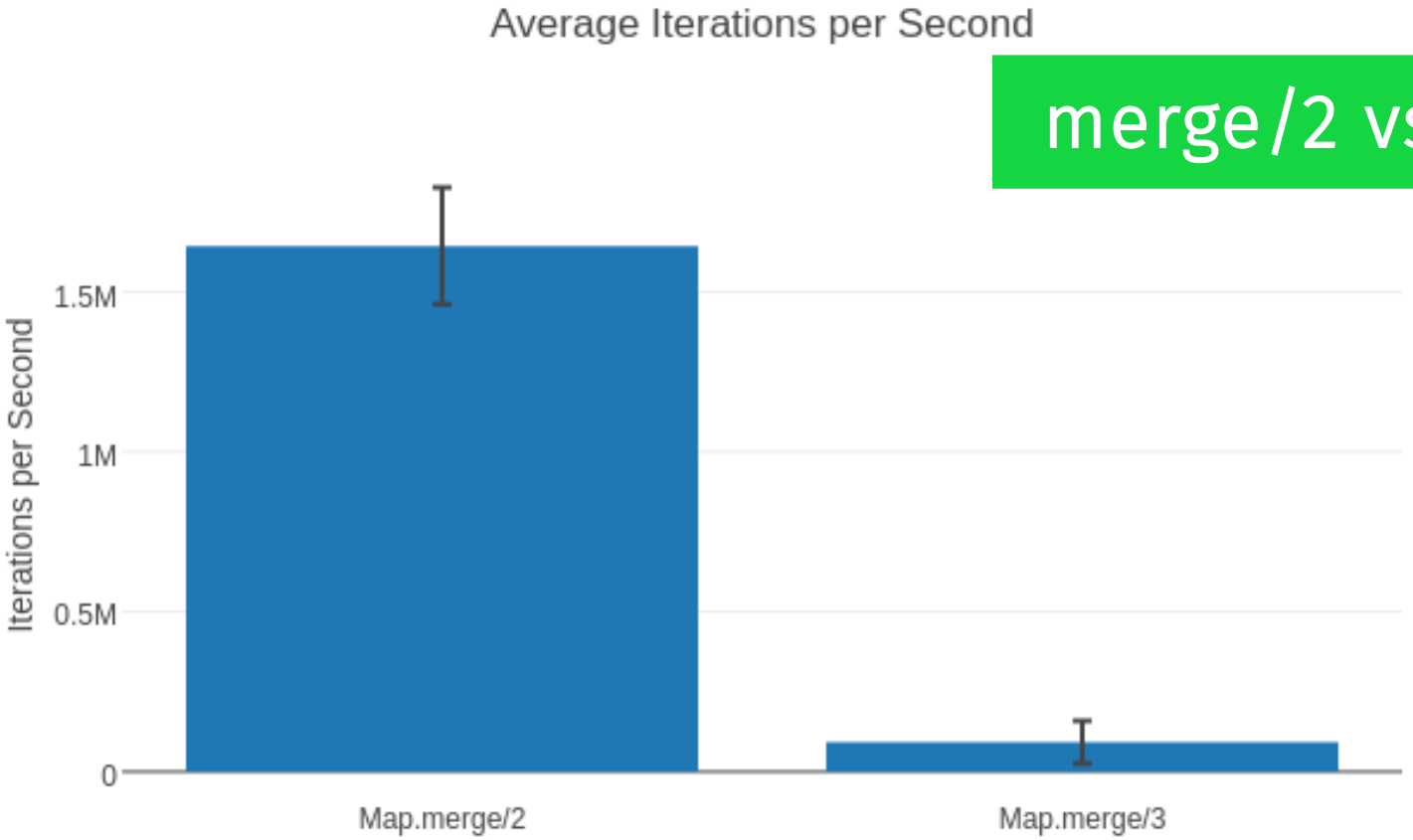
- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3




```
Name                  ips        average  deviation         median
Map.merge/2         1.64 M        0.61 µs    ±11.12%        0.61 µs
Map.merge/3       0.0921 M       10.86 µs    ±72.22%       10.00 µs

Comparison:
Map.merge/2         1.64 M
Map.merge/3       0.0921 M - 17.85x slower
```

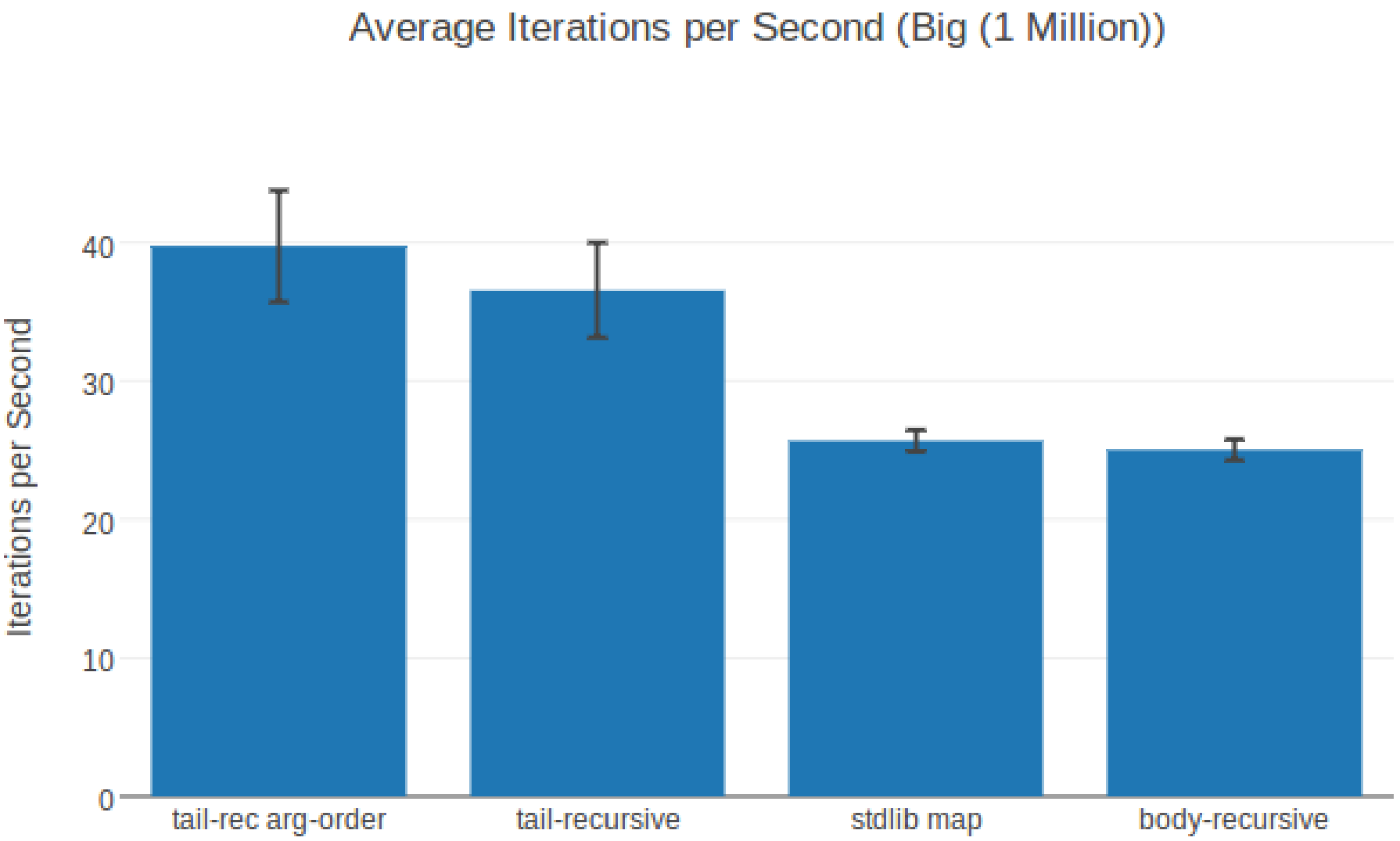
```elixir
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], function) do
    do_map_tco([function.(head) | acc], tail, function)
  end

  def map_tco_arg_order(list, function) do
    Enum.reverse do_map_tco_arg_order(list, function, [])
  end

  defp do_map_tco_arg_order([], _function, acc) do
    acc
  end
  defp do_map_tco_arg_order([head | tail], func, acc) do
    do_map_tco_arg_order(tail, func, [func.(head) | acc])
  end
end
```

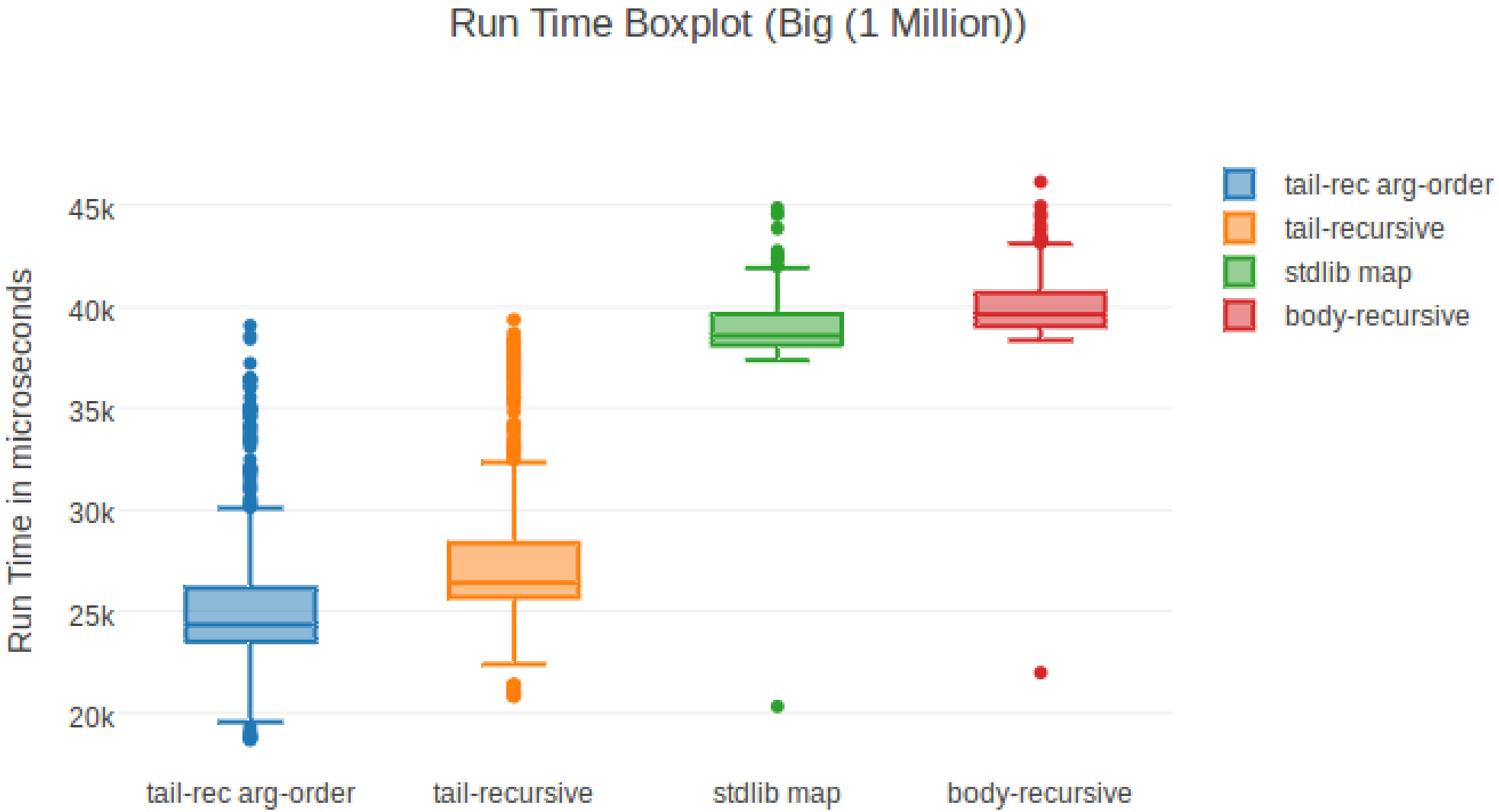
```elixir
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], function) do
    do_map_tco([function.(head) | acc], tail, function)
  end

  def map_tco_arg_order(list, function) do
    Enum.reverse do_map_tco_arg_order(list, function, [])
  end

  defp do_map_tco_arg_order([], _function, acc) do
    acc
  end
  defp do_map_tco_arg_order([head | tail], func, acc) do
    do_map_tco_arg_order(tail, func, [func.(head) | acc])
  end
end
```

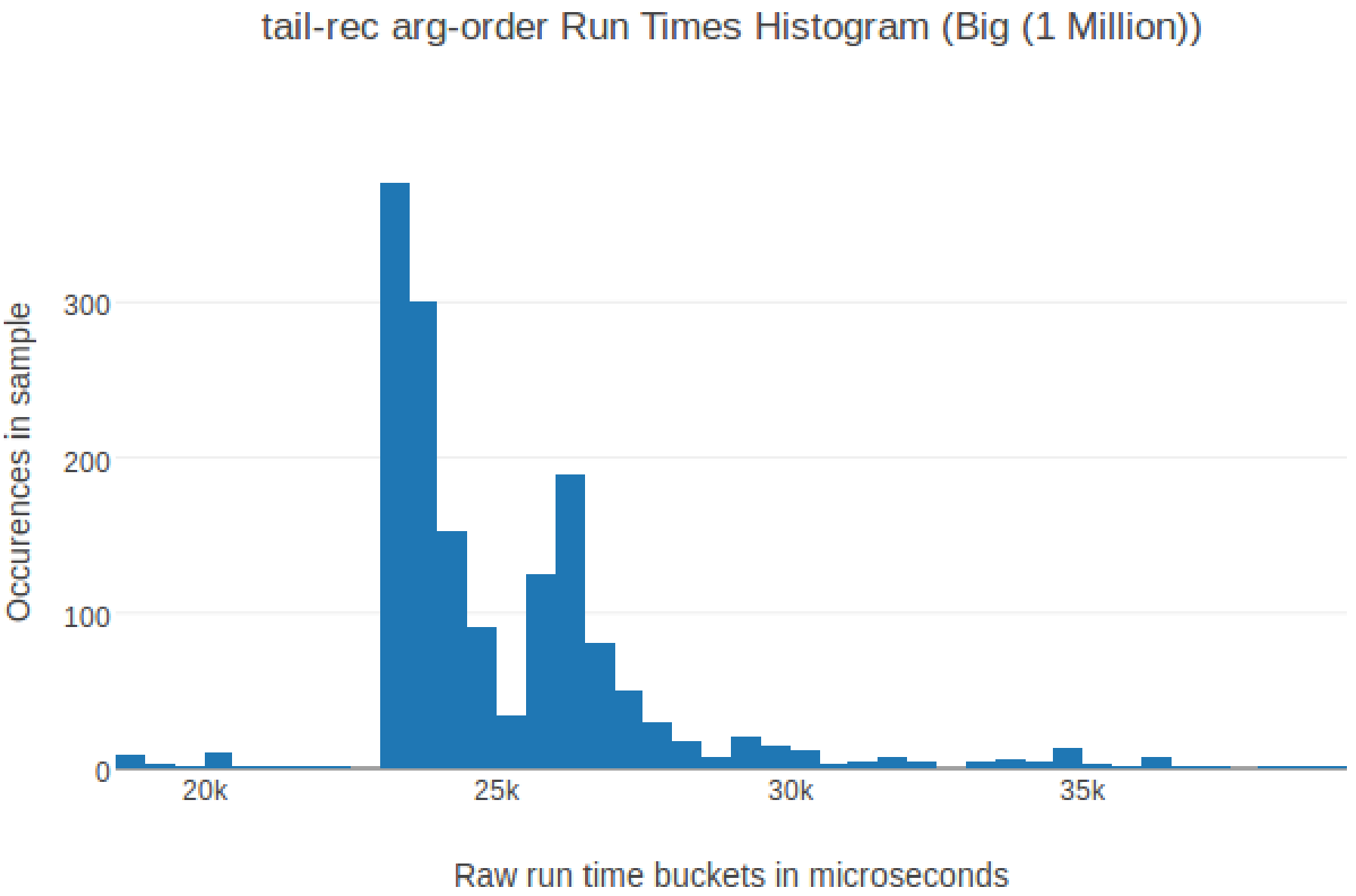
Does **argument order** make a difference?

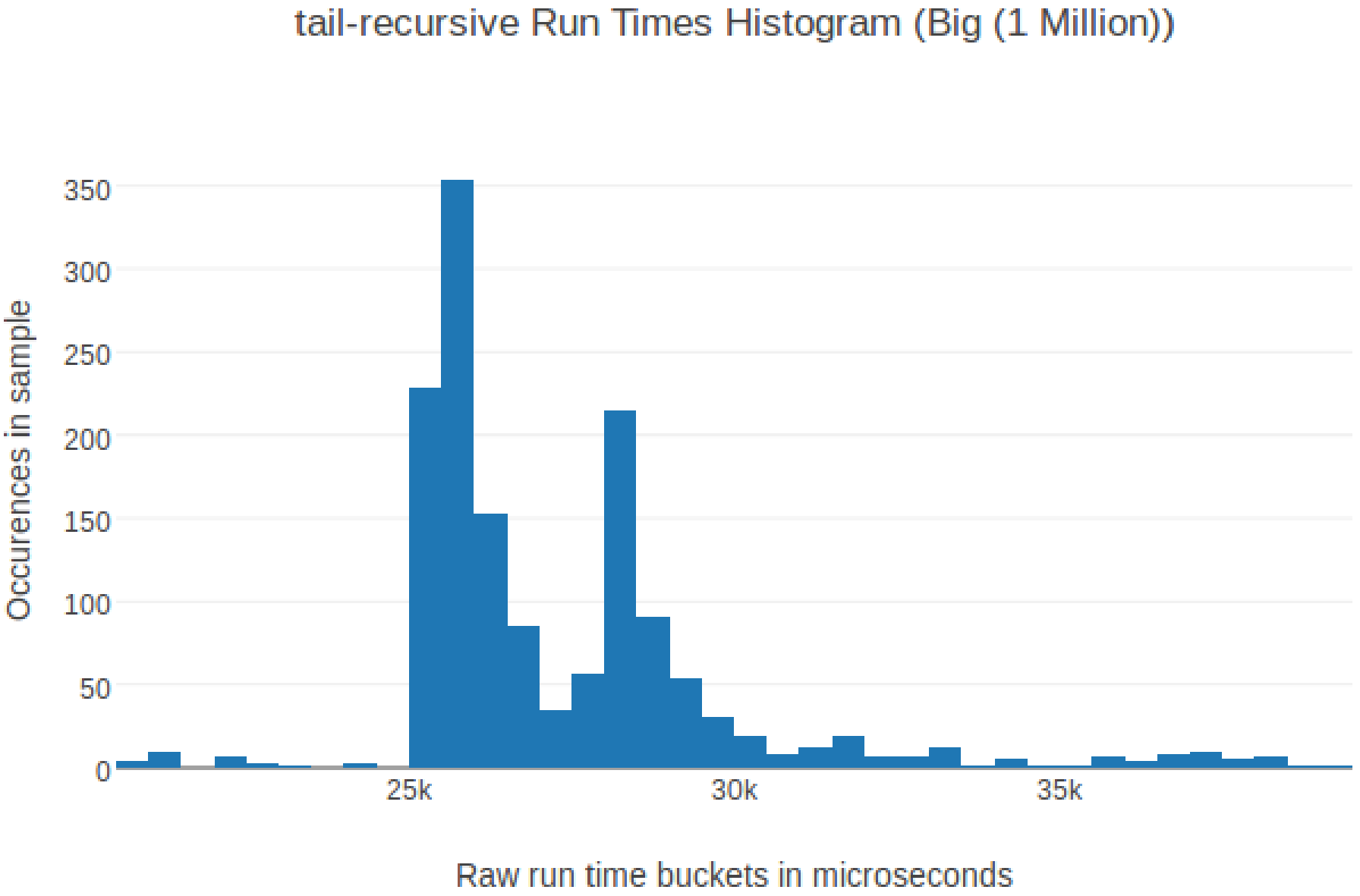
```
##### With input Middle (100 Thousand) #####
Name                          ips        average  deviation         median
stdlib map                 490.02        2.04 ms     ±7.76%        2.07 ms
body-recursive             467.51        2.14 ms     ±7.34%        2.17 ms
tail-rec arg-order         439.04        2.28 ms    ±17.96%        2.25 ms
tail-recursive             402.56        2.48 ms    ±16.00%        2.46 ms

Comparison:
stdlib map                 490.02
body-recursive             467.51 - 1.05x slower
tail-rec arg-order         439.04 - 1.12x slower
tail-recursive             402.56 - 1.22x slower

##### With input Big (1 Million) #####
Name                          ips        average  deviation         median
tail-rec arg-order           39.76       25.15 ms    ±10.14%       24.33 ms
tail-recursive               36.58       27.34 ms     ±9.38%       26.41 ms
stdlib map                   25.70       38.91 ms     ±3.05%       38.58 ms
body-recursive               25.04       39.94 ms     ±3.04%       39.64 ms

Comparison:
tail-rec arg-order           39.76
tail-recursive               36.58 - 1.09x slower
stdlib map                   25.70 - 1.55x slower
body-recursive               25.04 - 1.59x slower
```

```
##### With input Middle (100 Thousand) #####
Name                          ips        average  deviation         median
stdlib map                 490.02        2.04 ms     ±7.76%        2.07 ms
body-recursive             467.51        2.14 ms     ±7.34%        2.17 ms
tail-rec arg-order         439.04        2.28 ms    ±17.96%        2.25 ms
tail-recursive             402.56        2.48 ms    ±16.00%        2.46 ms

Comparison:
stdlib map                 490.02
body-recursive             467.51 - 1.05x slower
tail-rec arg-order         439.04 - 1.12x slower
tail-recursive             402.56 - 1.22x slower

##### With input Big (1 Million) #####
Name                          ips        average  deviation         median
tail-rec arg-order           39.76       25.15 ms    ±10.14%       24.33 ms
tail-recursive               36.58       27.34 ms     ±9.38%       26.41 ms
stdlib map                   25.70       38.91 ms     ±3.05%       38.58 ms
body-recursive               25.04       39.94 ms     ±3.04%       39.64 ms

Comparison:
tail-rec arg-order           39.76
tail-recursive               36.58 - 1.09x slower
stdlib map                   25.70 - 1.55x slower
body-recursive               25.04 - 1.59x slower
```


Average Iterations per Second (Big (1 Million))
Iterations per Second
40
30
20
10
0
tail-rec arg-order
tail-recursive
stdlib map
body-recursive


Run Time Boxplot (Big (1 Million))
Run Time in microseconds
45k
40k
35k
30k
25k
20k
tail-rec arg-order
tail-recursive
stdlib map
body-recursive
tail-rec arg-order
tail-recursive
stdlib map
body-recursive


tail-rec arg-order Run Times Histogram (Big (1 Million))
Occurences in sample
300
200
100
0
20k
25k
30k
35k
Raw run time buckets in microseconds


tail-recursive Run Times Histogram (Big (1 Million))
Occurences in sample
350
300
250
200
150
100
50
0
25k
30k
35k
Raw run time buckets in microseconds


But...
it can not be!

A wild José appears!

"The order of arguments will likely matter when we generate the branching code. The order of arguments will specially matter if performing binary matching."

José Valim, 2016

(Comment Section of my blog!)

A transformation of inputs

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

benchee report Big (1 Million)

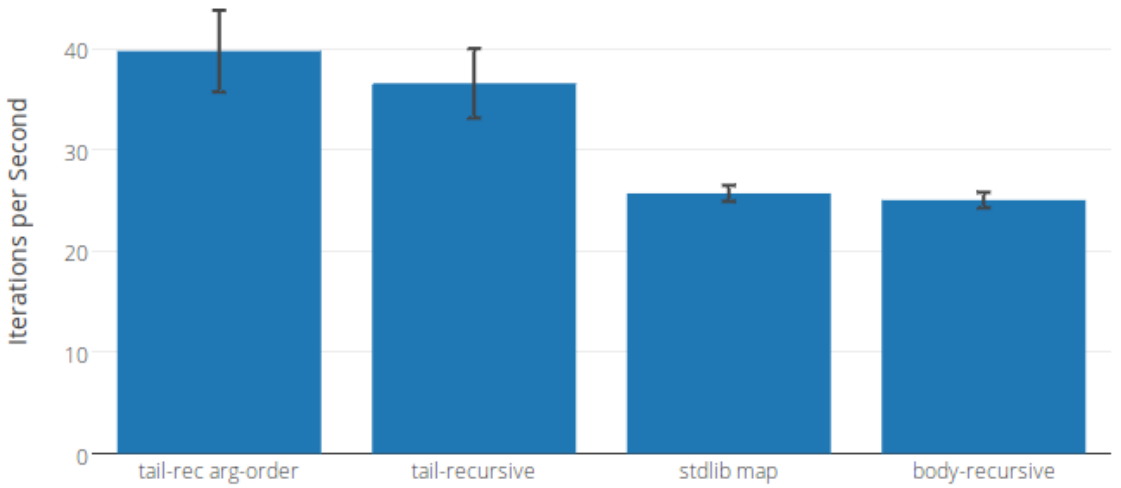
Elixir 1.3.4, Erlang 19.1

Comparison

Data Table

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
tail-rec arg-order	39.76	25151.35 µs	±10.14%	24331.50 µs	18677.00 µs	39087.00 µs	1590
tail-recursive	36.58	27338.88 µs	±9.38%	26405.00 µs	20819.00 µs	39377.00 µs	1463
stdlib map	25.70	38907.66 µs	±3.05%	38582.00 µs	20335.00 µs	44868.00 µs	1028
body-recursive	25.04	39936.11 µs	±3.04%	39635.50 µs	22004.00 µs	46155.00 µs	1002

Average Iterations per Second (Big (1 Million))



Always do **your own benchmarks**!

```elixir
alias Benchee.Formatters.{Console, HTML}
map_fun = fn(i) -> [i, i * i] end

inputs = %{
  "Small"  => Enum.to_list(1..200),
  "Medium" => Enum.to_list(1..1000),
  "Bigger" => Enum.to_list(1..10_000)
}

Benchee.run(%{
  "flat_map" =>
    fn(list) -> Enum.flat_map(list, map_fun) end,
  "map.flatten" => fn(list) ->
    list
    |> Enum.map(map_fun)
    |> List.flatten
  end
}, inputs: inputs,
   formatters: [&Console.output/1, &HTML.output/1],
   html: [file: "bench/output/flat_map.html"])
```

Remember?

Mhm Upgrades

- Elixir **1.4.0-rc.0**
- Erlang 19.1
- i5-7200U – 2 x 2.5GHz (Up to 3.10GHz)
- 8GB RAM
- Linux Mint 18 - 64 bit (Ubuntu 16.04 base)
- Linux Kernel 4.4.0-51

flat_map

```
alias Benchee.Formatters.{Console, HTML}
map_fun = fn(i) -> [i, i * i] end

inputs = %{
  "Small"  => Enum.to_list(1..200),
  "Medium" => Enum.to_list(1..1000),
  "Bigger" => Enum.to_list(1..10_000)
}

Benchee.run(%{
  "flat_map" =>
    fn(list) -> Enum.flat_map(list, map_fun) end,
  "map.flatten" => fn(list) ->
    list
    |> Enum.map(map_fun)
    |> List.flatten
  end
}, inputs: inputs,
   formatters: [&Console.output/1, &HTML.output/1],
   html: [file: "bench/output/flat_map.html"])
```

flat_map

```
alias Benchee.Formatters.{Console, HTML}
map_fun = fn(i) -> [i, i * i] end

inputs = %{
  "Small"  => Enum.to_list(1..200),
  "Medium" => Enum.to_list(1..1000),
  "Bigger" => Enum.to_list(1..10_000)
}

Benchee.run(%{
  "flat_map" =>
    fn(list) -> Enum.flat_map(list, map_fun) end,
  "map.flatten" => fn(list) ->
    list
    |> Enum.map(map_fun)
    |> List.flatten
  end
}, inputs: inputs,
   formatters: [&Console.output/1, &HTML.output/1],
   html: [file: "bench/output/flat_map.html"])
```

flat_map

```
Erlang/OTP 19 [erts-8.1] [source] [64-bit] [smp:4:4] [async-
threads:10] [hipe] [kernel-poll:false]
Elixir 1.4.0-rc.0
Benchmark suite executing with the following configuration:
warmup: 2.0s
time: 5.0s
parallel: 1
inputs: Bigger, Medium, Small
Estimated total run time: 42.0s


Benchmarking with input Bigger:
Benchmarking flat_map...
Benchmarking map.flatten...

Benchmarking with input Medium:
Benchmarking flat_map...
Benchmarking map.flatten...

Benchmarking with input Small:
Benchmarking flat_map...
Benchmarking map.flatten...
```

flat_map

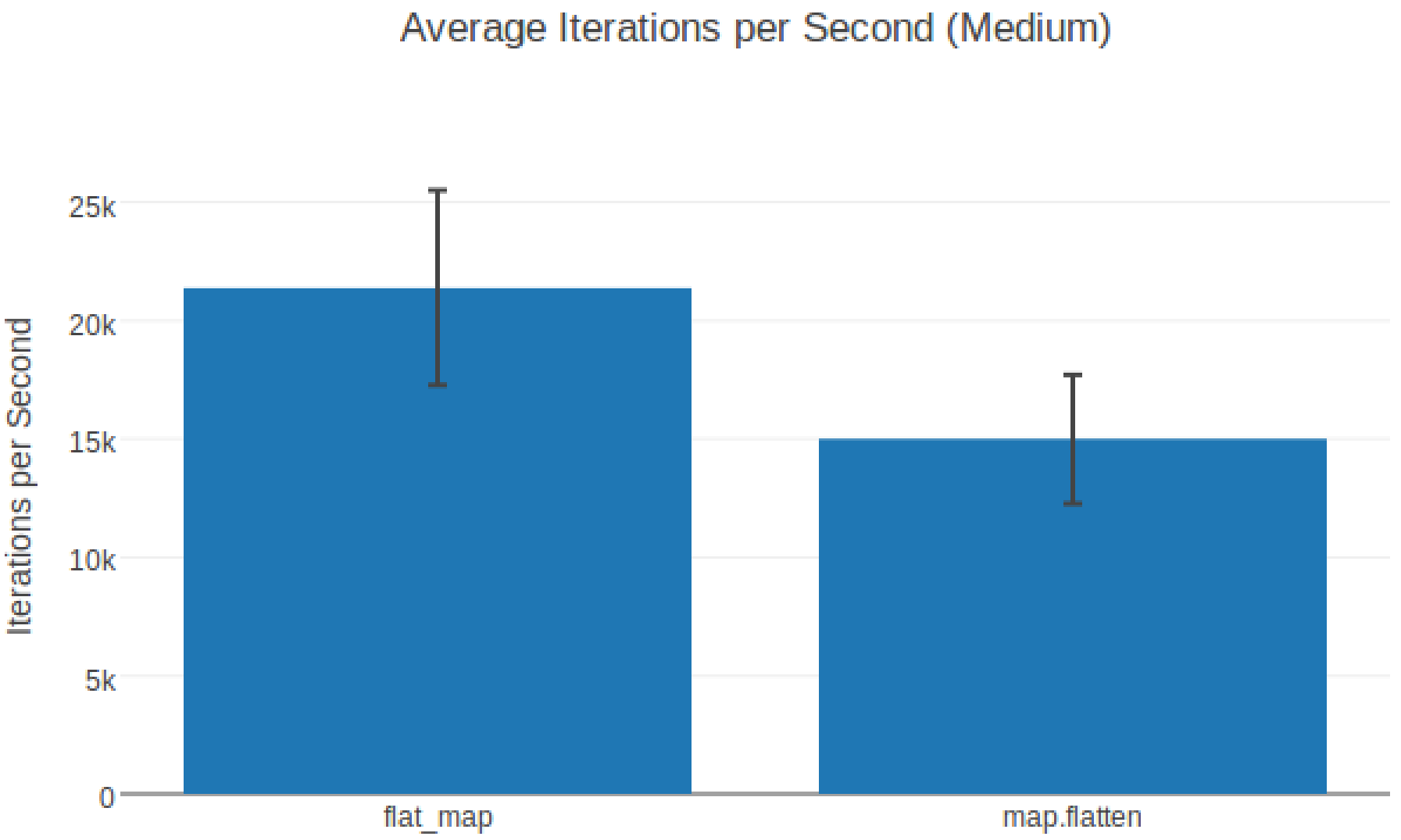
```
Erlang/OTP 19 [erts-8.1] [source] [64-bit] [smp:4:4] [async-
threads:10] [hipe] [kernel-poll:false]
Elixir 1.4.0-rc.0
Benchmark suite executing with the following configuration:
warmup: 2.0s
time: 5.0s
parallel: 1
inputs: Bigger, Medium, Small
Estimated total run time: 42.0s


Benchmarking with input Bigger:
Benchmarking flat_map...
Benchmarking map.flatten...

Benchmarking with input Medium:
Benchmarking flat_map...
Benchmarking map.flatten...

Benchmarking with input Small:
Benchmarking flat_map...
Benchmarking map.flatten...
```

```
##### With input Bigger #####
Name                    ips        average  deviation         median
flat_map              1.76 K      569.47 µs    ±26.95%      512.00 µs
map.flatten           1.02 K      982.57 µs    ±25.06%      901.00 µs

Comparison:
flat_map              1.76 K
map.flatten           1.02 K - 1.73x slower

##### With input Medium #####
Name                    ips        average  deviation         median
flat_map             21.39 K       46.76 µs    ±19.24%       48.00 µs
map.flatten          14.99 K       66.71 µs    ±18.13%       65.00 µs

Comparison:
flat_map             21.39 K
map.flatten          14.99 K - 1.43x slower

##### With input Small #####
Name                    ips        average  deviation         median
flat_map            118.66 K        8.43 µs   ±180.99%        8.00 µs
map.flatten          79.25 K       12.62 µs    ±97.97%       12.00 µs

Comparison:
flat_map            118.66 K
map.flatten          79.25 K - 1.50x slower
```

The tables have turned!

```
##### With input Bigger #####
Name                    ips        average  deviation         median
flat_map              1.76 K      569.47 µs    ±26.95%      512.00 µs
map.flatten           1.02 K      982.57 µs    ±25.06%      901.00 µs

Comparison:
flat_map              1.76 K
map.flatten           1.02 K - 1.73x slower

##### With input Medium #####
Name                    ips        average  deviation         median
flat_map             21.39 K       46.76 µs    ±19.24%       48.00 µs
map.flatten          14.99 K       66.71 µs    ±18.13%       65.00 µs

Comparison:
flat_map             21.39 K
map.flatten          14.99 K - 1.43x slower

##### With input Small #####
Name                    ips        average  deviation         median
flat_map            118.66 K        8.43 µs   ±180.99%        8.00 µs
map.flatten          79.25 K       12.62 µs    ±97.97%       12.00 µs

Comparison:
flat_map            118.66 K
map.flatten          79.25 K - 1.50x slower
```

> 2x faster


Average Iterations per Second (Medium)
Iterations per Second
25k
20k
15k
10k
5k
0
flat_map
map.flatten

How did that **happen**?

flat_map is slower than map |> flatten (and erlangs :lists.flatmap) #5082

Closed **PragTob** opened this issue on Aug 2 · 5 comments

PragTob commented on Aug 2 Contributor

(this is not strictly a bug but a somewhat surprising performance degradation I found, hope the issue tracker is still the appropriate place :))

18 minutes later...

josevalim commented on Aug 2 • edited

Member

So Enum.flat_map never calls `++` and uses a reversal, that should theoretically consume less memory. `flat_map` also accepts any enumerable to be flattened while `:lists` can afford to work only on lists. In any case, can you please try these versions?

```
# v1
def flat_map(enumerable, fun) when is_function(fun, 1) do
  Enum.reduce(enumerable, [], fn(entry, acc) ->
    case fun.(entry) do
      list when is_list(list) -> :lists.reverse(list, acc)
      other -> Enum.reduce(other, acc, &[&1 | &2])
    end
  end) |> :lists.reverse
end
```

```
# v2
def flat_map_list([h | t], fun) do
  case fun.(h) do
    list when is_list(list) -> list ++ flat_map_list(t, fun)
    other -> Enum.to_list(other) ++ flat_map_list(t, fun)
  end
end
```

edit: forgot to mention, both versions are significantly faster for all use cases and v2 is even the fastest which is 🚀 🚀 🚀 - which is amazing :D It could have taken you at most 18 minutes or so :)

🎉 3

josevalim added a commit that closed this issue on Aug 2

`Optimize flat_map, closes #5082` 2dfdb36

josevalim commented on Aug 2

Thank you for benchmarking! ❤️

Enjoy Benchmarking! ❤

Tobias Pfeiffer
@PragTob
pragtob.info
github.com/PragTob/benchee

LIEFERY