
One Billion Rows Challenge in Elixir
From 12 minutes to 25 seconds
Raj Rajhans
Code Beam Europe 2024

Who am I?

Software Engineer at Invideo AI

Elixir, Rust & Javascript

Organizing RustMumbai

@_rajrajhans

rajrajhans.com


THE ONE BILLION ROWS
CHALLENGE

```
...
Tokyo;16.8
Cape Town;22.3
Stockholm;7.1
Marrakech;29.6
Berlin;9.7
Auckland;17.5
Bangkok;32.7
Reykjavik;3.4
Vienna;14.6
Moscow;1.9
Berlin;6.3
Hanoi;30.2
Mexico City;20.8
Cairo;33.5
Oslo;5.7
Kyoto;18.1
Miami;27.6
Berlin;11.2
Prague;15.9
Istanbul;23.4
Edinburgh;10.8
Seattle;13.7
Dubai;36.8
Queenstown;9.9
...
```

```
...
Tokyo;16.8
Cape Town;22.3
Stockholm;7.1
Marrakech;29.6
Berlin;9.7
Auckland;17.5
Bangkok;32.7
Reykjavik;3.4
Vienna;14.6
Moscow;1.9
Berlin;6.3
Hanoi;30.2
Mexico City;20.8
Cairo;33.5
Oslo;5.7
Kyoto;18.1
Miami;27.6
Berlin;11.2
Prague;15.9
Istanbul;23.4
Edinburgh;10.8
Seattle;13.7
Dubai;36.8
Queenstown;9.9
...
```



```
Auckland;17.5;17.5;17.5
Bangkok;32.7;32.7;32.7
Berlin;6.3;9.1;11.2
Cairo;33.5;33.5;33.5
Cape Town;22.3;22.3;22.3
Dubai;36.8;36.8;36.8
Edinburgh;10.8;10.8;10.8
Hanoi;30.2;30.2;30.2
Istanbul;23.4;23.4;23.4
Kyoto;18.1;18.1;18.1
Marrakech;29.6;29.6;29.6
Mexico City;20.8;20.8;20.8
Miami;27.6;27.6;27.6
Moscow;1.9;1.9;1.9
Oslo;5.7;5.7;5.7
Prague;15.9;15.9;15.9
Queenstown;9.9;9.9;9.9
Reykjavik;3.4;3.4;3.4
Seattle;13.7;13.7;13.7
Stockholm;7.1;7.1;7.1
Tokyo;16.8;16.8;16.8
Vienna;14.6;14.6;14.6
```

```
...
Nairobi;28.3
Tokyo;15.7
Berlin;7.9
Lisbon;19.4
Cairo;32.1
Vancouver;12.6
Mumbai;30.5
Stockholm;5.2
Rio de Janeiro;27.8
Moscow;-2.1
Berlin;11.3
Cape Town;22.6
Bangkok;33.9
Dublin;13.0
Sydney;24.2
Toronto;8.7
Dubai;38.4
Amsterdam;9.6
Singapore;29.8
Berlin;6.5
Seoul;18.2
Vienna;14.9
Buenos Aires;26.3
Helsinki;3.8
San Francisco;17.5
Oslo;6.1
Mexico City;21.7
Athens;23.5
...
```

```
...
Madrid;24.6
Hanoi;29.8
Queenstown;11.3
Cairo;33.5
Berlin;8.7
Kuala Lumpur;31.2
Stockholm;6.9
Brisbane;25.1
Nairobi;22.7
Mexico City;19.4
Lisbon;20.8
Anchorage;-1.5
Singapore;28.9
Warsaw;13.6
Marrakech;27.3
Berlin;11.9
Seattle;14.2
Athens;26.5
Reykjavik;3.8
Vienna;15.4
Cape Town;23.1
Moscow;2.6
Dubai;38.7
Kyoto;18.9
Berlin;5.3
Amsterdam;12.1
Buenos Aires;26.8
Toronto;9.5
...
```

```
...
Sydney;23.7
Helsinki;4.2
Havana;29.5
Johannesburg;20.1
Shanghai;18.6
Vancouver;13.9
Berlin;7.4
Cairo;32.8
Oslo;6.5
Rio de Janeiro;27.3
Mumbai;31.6
Edinburgh;11.2
Bangkok;33.9
Berlin;10.8
Reykjavik;2.7
Prague;16.3
San Francisco;17.8
Wellington;15.4
Marrakech;28.2
Taipei;25.9
Berlin;5.1
Montreal;8.3
Rome;22.6
Seoul;19.7
Lisbon;21.4
Anchorage;-0.8
Singapore;30.1
Kyoto;17.5
...
```

```
...
Tokyo;16.8
Cape Town;22.3
Dubai;38.9
Stockholm;7.1
Marrakech;29.6
Berlin;9.7
Auckland;17.5
Bangkok;32.7
Reykjavik;3.4
Vienna;14.6
Moscow;1.9
Berlin;6.3
Hanoi;30.2
Mexico City;20.8
Cairo;33.5
Oslo;5.7
Singapore;28.4
Lisbon;19.3
Vancouver;12.5
Kyoto;18.1
Miami;27.6
Berlin;11.2
Prague;15.9
Istanbul;23.4
Edinburgh;10.8
Seattle;13.7
Dubai;36.8
Queenstown;9.9
...
```

```
...
Rome;23.5
Lima;19.8
Oslo;7.2
Tokyo;18.4
Delhi;33.6
Berlin;9.1
Miami;28.7
Cairo;35.2
Bern;12.5
Seoul;20.9
Riga;8.3
Dubai;40.1
Paris;16.7
Berlin;6.8
Accra;29.4
Quito;15.3
Sofia;14.8
Lagos;31.2
Berlin;11.3
Porto;19.5
Doha;36.8
Hanoi;30.7
Minsk;9.6
Tunis;25.4
Lyon;15.9
Surat;32.1
Boston;13.7
Perth;22.8
...
```

```
...
Madrid;22.7
Bangkok;33.8
Reykjavik;4.1
Mumbai;31.5
Cairo;34.2
Vancouver;14.3
Helsinki;6.9
Rio de Janeiro;28.6
Berlin;8.5
Dubai;39.1
Prague;16.2
Sydney;24.8
Wellington;15.7
Marrakech;30.3
Anchorage;-2.1
Seoul;19.5
Berlin;10.9
Istanbul;25.1
Singapore;29.7
Lisbon;20.6
Berlin;7.3
Toronto;11.8
Athens;26.9
Oslo;5.4
San Francisco;17.9
Kyoto;18.3
Buenos Aires;26.2
Hanoi;30.8
...
```

```
...
Berlin;9.1
Barcelona;23.5
Tokyo;17.8
Dubai;38.6
Oslo;6.2
Cape Town;20.9
Bangkok;32.7
Reykjavik;3.8
Mexico City;21.3
Sydney;25.6
Marrakech;29.4
Vancouver;13.5
Berlin;7.6
Helsinki;5.7
Istanbul;25.1
Prague;15.4
Singapore;30.2
Edinburgh;11.8
Cairo;33.9
Lisbon;19.7
Kyoto;18.6
Buenos Aires;24.8
Toronto;12.1
Istanbul;26.5
Auckland;16.9
Moscow;2.4
Vancouver;14.3
Berlin;10.3
...
```

Version 1: Simple & Idiomatic Elixir

- Read the entire file into memory.
- Split it by newlines.
- Create a weather station to values map.
- Aggregate the result.

```elixir
{:ok, content} = File.read(file_path)
acc =
    content
    |> String.split("\n")
    |> Enum.map(&String.split(&1, ";"))
    |> Enum.reject(fn value -> value |> Enum.at(0) == "" end)
    |> Enum.reduce(%{}, fn [key, value], acc ->
        {val, _} = Float.parse(value)
        Map.update(acc, key, [val], fn v -> [val | v] end)
    end)
result =
    acc
    |> Enum.map(fn {key, values} ->
        min = Enum.min(values) |> round_to_single_decimal()
        max = Enum.max(values) |> round_to_single_decimal()
        mean =
          (Enum.sum(values) / length(values))
          |> round_to_single_decimal()

        {key, %{min: min, max: max, mean: mean}}
    end)
```

1BRC in Elixir: Version 1

10 Million Rows

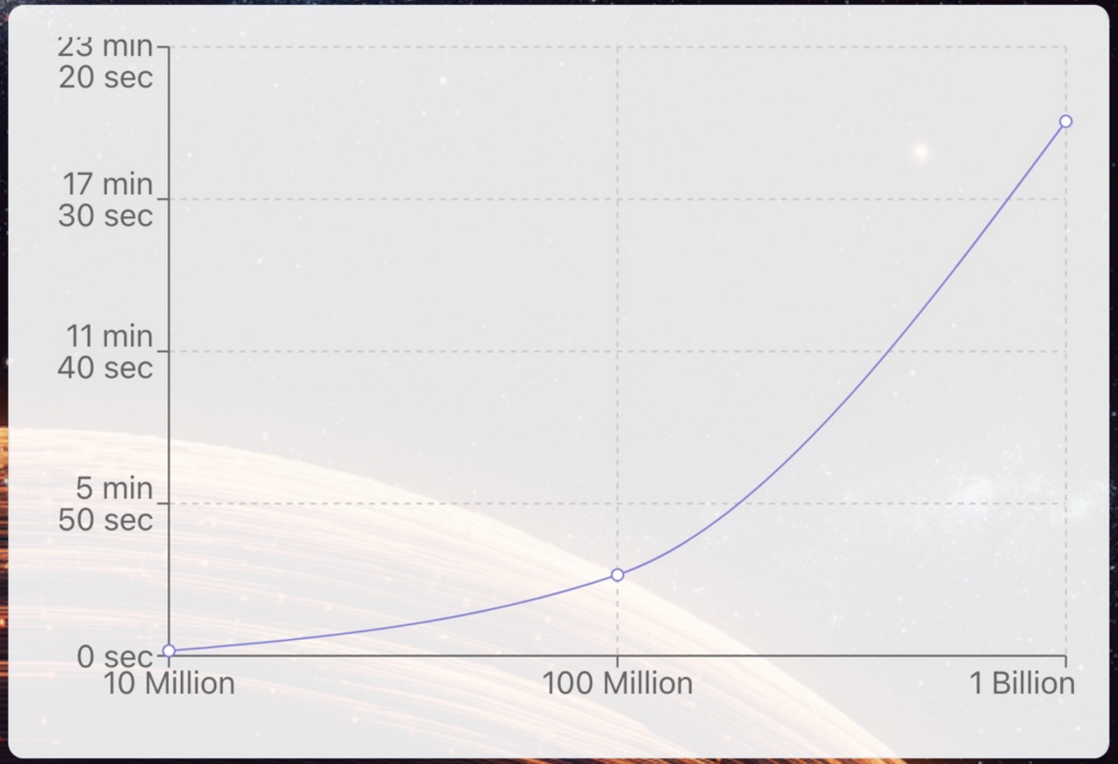
11.5 sec

100 Million Rows

3 min 6 sec

1 Billion Rows

x



Version 2: Quick wins: Streaming, Incremental Processing

- File.stream!/2 for efficient file reading
- Incremental processing: updating min, max, sum and count on-the-fly.
- Single pass over final map to calculate mean temperatures.
- Reduced memory usage by not storing all temperature values.

Version 2: Quick wins: Streaming, Incremental Processing

- File.stream!/2 for efficient file reading
- Incremental processing: updating min, max, sum and count on-the-fly.
- Single pass over final map to calculate mean temperatures.
- Reduced memory usage by not storing all temperature values.

```
result =
    File.stream!(file_path)
    |> Stream.map(&String.split(&1, ";"))
    |> Enum.reduce(%{}, fn [key, value], acc ->
    {val, _} = Float.parse(value)

    default = %{
        min: val,
        max: val,
        sum: val,
        count: 1
    }

    Map.update(acc, key, default, fn record ->
        min = if val < record.min, do: val, else: record.min
        max = if val > record.max, do: val, else: record.max
        sum = record.sum + val
        count = record.count + 1

        %{
        min: min,
        max: max,
        sum: sum,
        count: count
        }
    end)
end)
```

Version 2: Quick wins: Streaming, Incremental Processing

- File.stream!/2 for efficient file reading
- Incremental processing: updating min, max, sum and count on-the-fly.
- Single pass over final map to calculate mean temperatures.
- Reduced memory usage by not storing all temperature values.

```
result =
    result
    ▷ Enum.map(fn {key,
    %{min: min, max: max, sum: sum, count: count}} →
        mean = (sum / count) ▷ round_to_single_decimal()

        {key, %{min: min, max: max, mean: mean}}
    end)

result_txt =
    result
    ▷ Enum.sort_by(fn {key, _} → key end)
    ▷ Enum.reduce("", fn {key,
    %{min: min, max: max, mean: mean}}, acc →
    acc <> "#{key};#{min};#{mean};#{max}\n"
    end)
```

1BRC in Elixir: Version 2

10 Million Rows

7.8 sec

↘ 32.2%

100 Million Rows

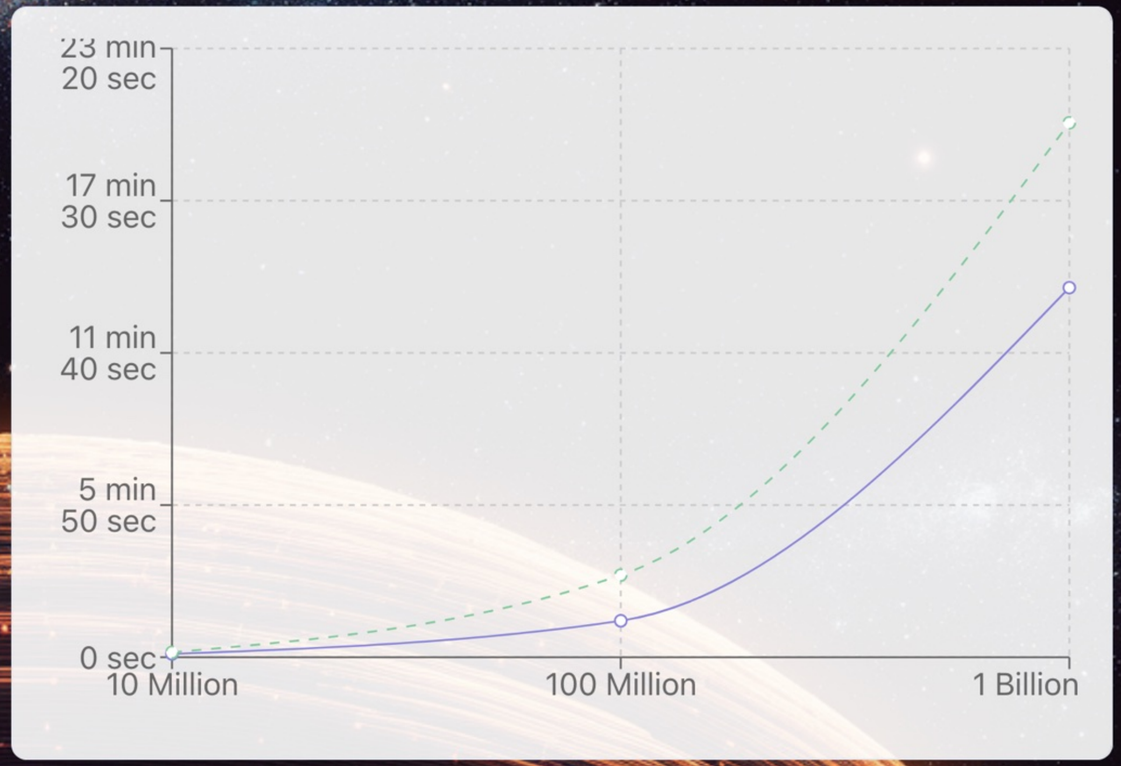
1 min 24 sec

↘ 55.6%

1 Billion Rows

14 min 10 sec

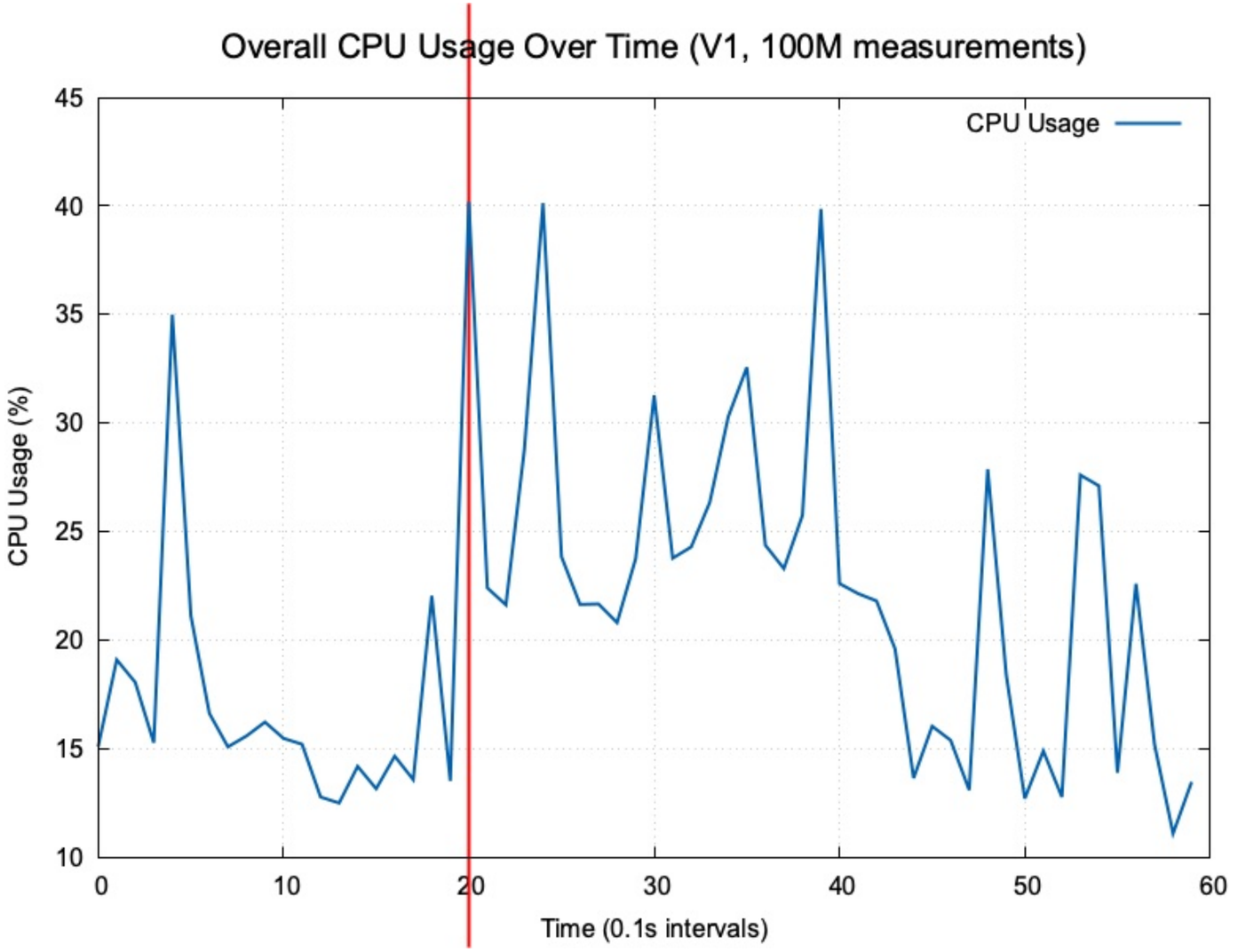
↘ 30.8%

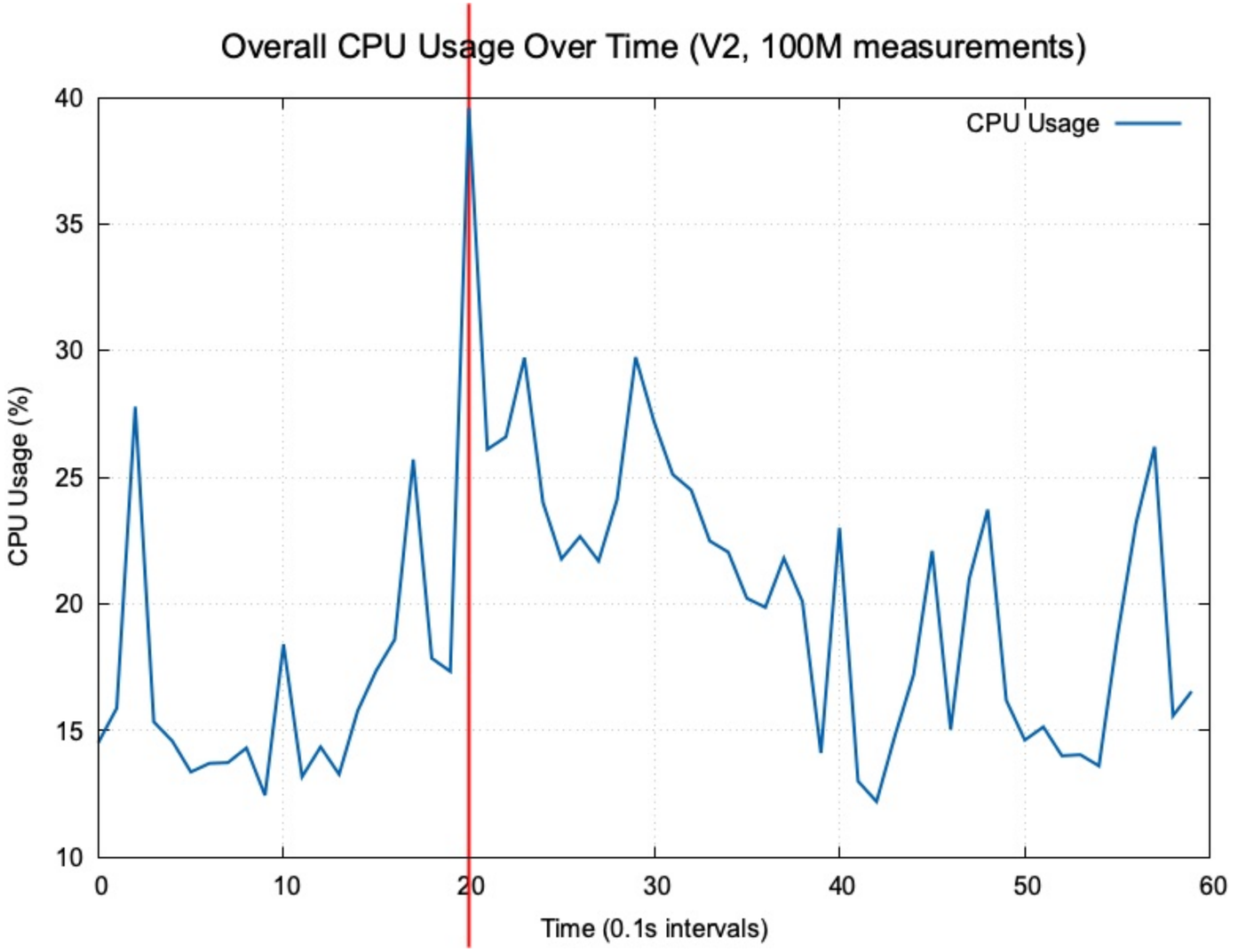


Version 2: Quick wins: Streaming, Incremental Processing

- File.stream!/2 for efficient file reading
- Incremental processing: updating min, max, sum and count on-the-fly.
- Reduced memory usage by not storing all temperature values.
- Single pass over final map to calculate mean temperatures.

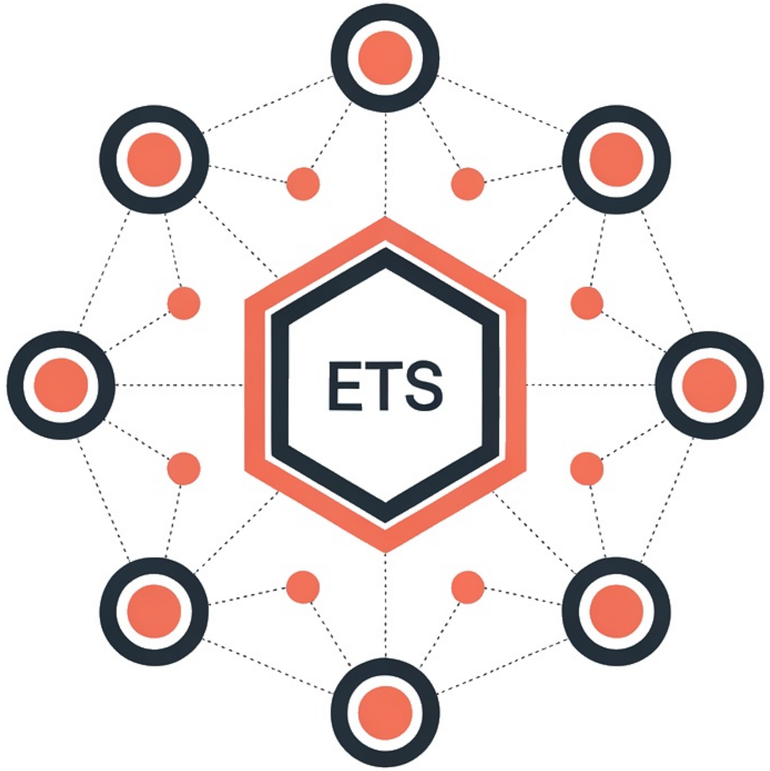
```
result =
    File.stream!(file_path)
    ▷ Stream.map(&String.split(&1, ";"))
    ▷ Enum.reduce(%{}, fn [key, value], acc →
    {val, _} = Float.parse(value)

    default = %{
        min: val,
        max: val,
        sum: val,
        count: 1
    }

    Map.update(acc, key, default, fn record →
        min = if val < record.min, do: val, else: record.min
        max = if val > record.max, do: val, else: record.max
        sum = record.sum + val
        count = record.count + 1

        %{
        min: min,
        max: max,
        sum: sum,
        count: count
        }
    end)
end)
```


Overall CPU Usage Over Time (V1, 100M measurements)
CPU Usage
CPU Usage (%)
45
40
35
30
25
20
15
10
0
10
20
30
40
50
60
Time (0.1s intervals)


Overall CPU Usage Over Time (V2, 100M measurements)
CPU Usage
CPU Usage (%)
40
35
30
25
20
15
10
0
10
20
30
40
50
60
Time (0.1s intervals)

Version 3: Introducing Concurrency

- Concurrency using Task.async_stream/3
- Using ETS table for storing intermediate results



```elixir
defp process_row([key, value], ets_table) do
    {val, _} = Float.parse(value)
    existing_record = :ets.lookup(ets_table, key)
    new_record =
      case existing_record do
        [] ->
            %{
            min: val,
            max: val,
            sum: val,
            count: 1
            }
        [{^key, record}] ->
            min = if val < record.min, do: val, else: record.min
            max = if val > record.max, do: val, else: record.max
            sum = record.sum + val
            count = record.count + 1
            %{
                min: min,
                max: max,
                sum: sum,
                count: count
            }
        end
    :ets.insert(ets_table, {key, new_record})
end
```

lib/measurements_processor.ex +9 -1

```
@@ -28,7 +28,15 @@ defmodule OneBRC.MeasurementsProcessor do
      fs
      |> Stream.map(&String.split(&1, ";"))
      |> Stream.reject(fn value -> value |> Enum.at(0) == "" end)
-     |> Stream.map(fn val -> process_row(val, ets_table) end)
+     |> Stream.chunk_every(10_000)
+     |> Task.async_stream(
+       fn val ->
+         Enum.map(val, fn row -> process_row(row, ets_table) end)
+       end,
+       max_concurrency: System.schedulers_online() * 5,
+       ordered: false,
+       timeout: :infinity
+     )
      |> Stream.run()

      t2 = System.monotonic_time(:millisecond)
```

Replacing Stream.map with Task.async_stream to add concurrency 🤡

Version 3: Introducing Concurrency

- Using ETS table for storing intermediate results
- Lesson learned: be mindful of shared state and potential race conditions with concurrent processes.

Separating parsing from the actual calculation

lib/measurements_processor.ex +18 -8

```
32        |> Task.async_stream(
33  -       fn val ->
34  -         Enum.map(val, fn row -> process_row(row, ets_table)
      end)
35  -       end,
36          max_concurrency: System.schedulers_online() * 5,
37          ordered: false,
38          timeout: :infinity
39        )
40        |> Stream.run()
41
42        t2 = System.monotonic_time(:millisecond)
```

```
30        |> Task.async_stream(
31  +       fn val -> Enum.map(val, &parse_row/1) |>
        Enum.reject(&is_nil/1) end,
32          max_concurrency: System.schedulers_online() * 5,
33          ordered: false,
34          timeout: :infinity
35        )
36  +     |> Stream.flat_map(&elem(&1, 1))
37  +     |> Stream.map(&process_row(&1, ets_table))
38        |> Stream.run()
39
40        t2 = System.monotonic_time(:millisecond)
```

```
@@ -55,8 +53,20 @@ defmodule OneBRC.MeasurementsProcessor do
```

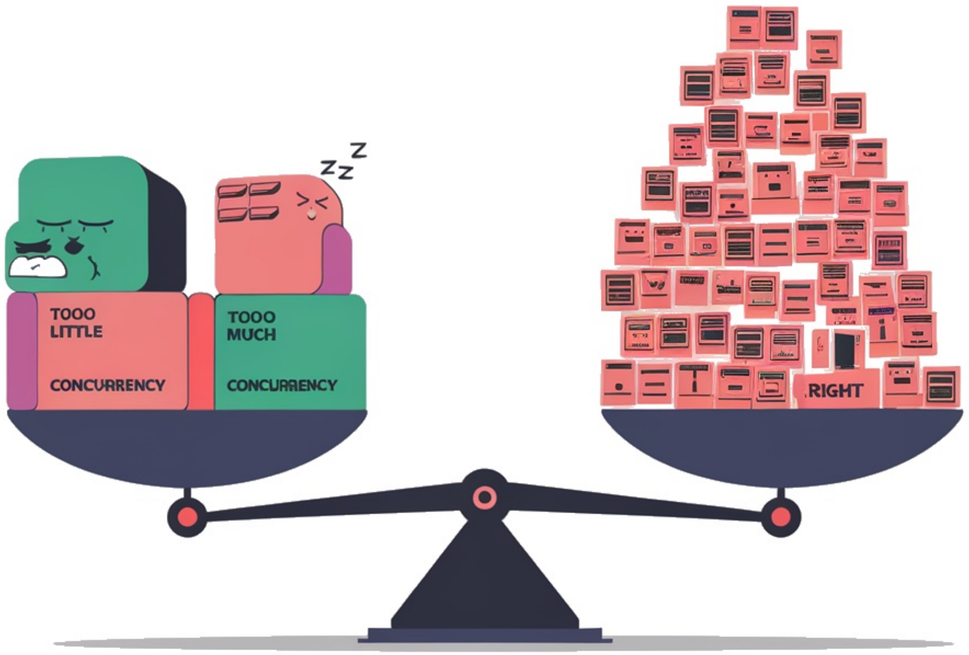
```
55        result
56      end
```

```
53        result
54      end
```

Parse rows concurrently, then process synchronously

Version 3: Introducing Concurrency

- Using ETS table for storing intermediate results
- Lesson 1: be mindful of shared state and potential race conditions with concurrent processes.
- Lesson 2: Too little concurrency underutilizes resources, while too much leads to overhead



lib/one_brc/measurements_processor.ex +1 -1

```
@@ -28,7 +28,7 @@ defmodule OneBRC.MeasurementsProcessor do
28  ets_table = :ets.new(:station_stats, [:set, :public])
29
30  fs
31 -|> Stream.chunk_every(2000)
31 +|> Stream.chunk_every(10000)
32  |> Task.async_stream(
33    fn val -> Enum.map(val, &parse_row/1) end,
34    max_concurrency: System.schedulers_online(),
```

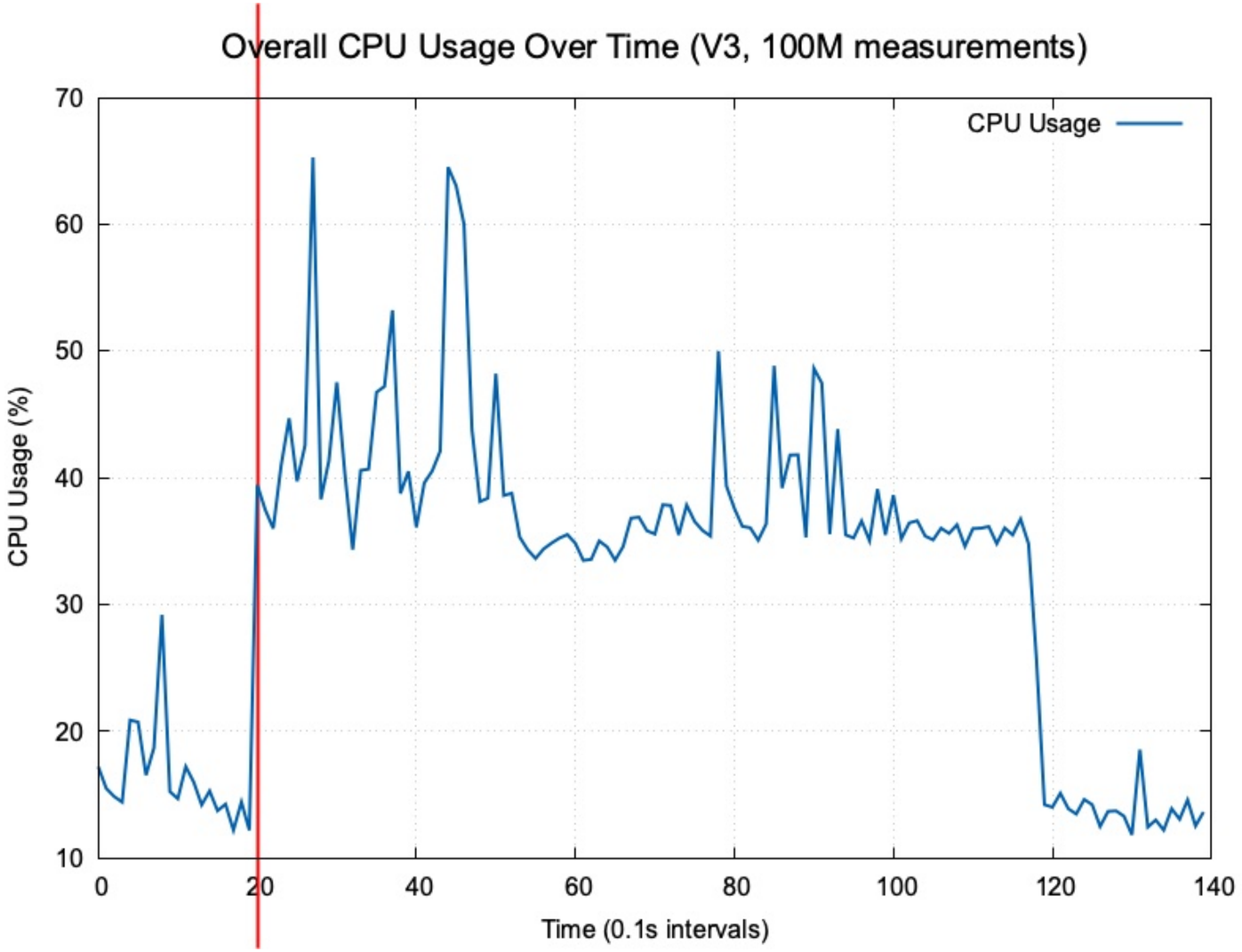
Process spawning overhead matters!

lib/measurements_processor.ex +9 -3

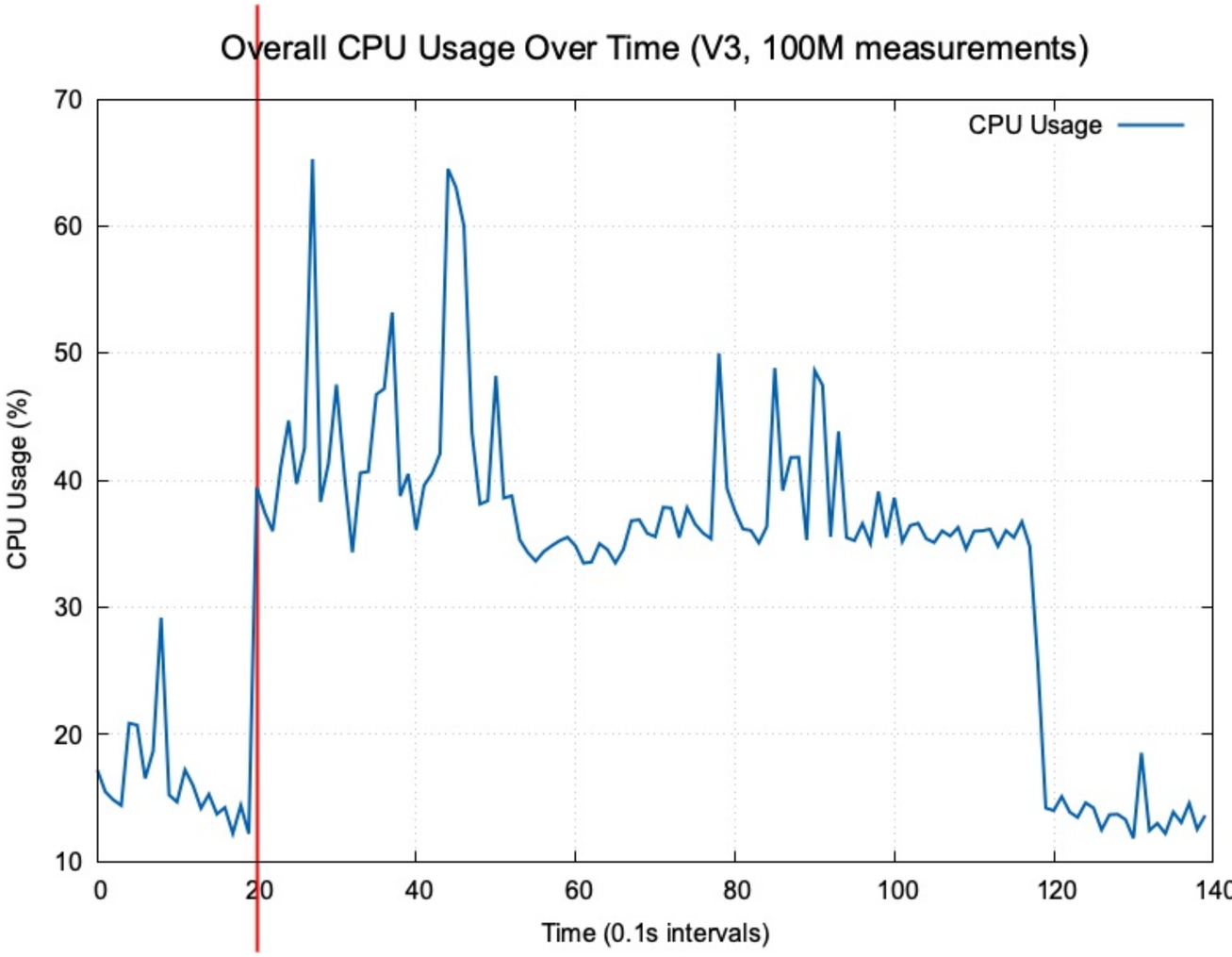
```
45 -         mean = (sum / count) |> round_to_single_decimal()
46
47 -         {key, %{min: min, max: max, mean: mean}}
48           end)
49
50         Logger.info("Processing data 1: #{t2 - t1} ms")
   @@ -60,7 +60,13 @@ defmodule OneBRC.MeasurementsProcessor do
60
61           row ->
62             [key, value] = String.split(row, ";")
63 -           parsed_value = String.to_float(value |>
   String.trim_trailing())
64             [key, parsed_value]
65         end
```

```
45 +         mean = (sum / (count * 10.0)) |>
   round_to_single_decimal()
46
47 +         {key, %{min: min / 10.0, max: max / 10.0, mean: mean}}
48           end)
49
50         Logger.info("Processing data 1: #{t2 - t1} ms")
60
61           row ->
62             [key, value] = String.split(row, ";")
63 +
64 +           parsed_value =
65 +             value |> String.trim_trailing()
66 +
67 +           [a, b] = parsed_value |> String.split(".")
68 +           parsed_value = (a <> b) |> String.to_integer()
69 +
70             [key, parsed_value]
71         end
```

Getting around float parsing



CPU usage of version 3



CPU usage of version 2 vs version 3

1BRC in Elixir: Version 3

10 Million Rows

4.7 sec

↘ 39.7%

100 Million Rows

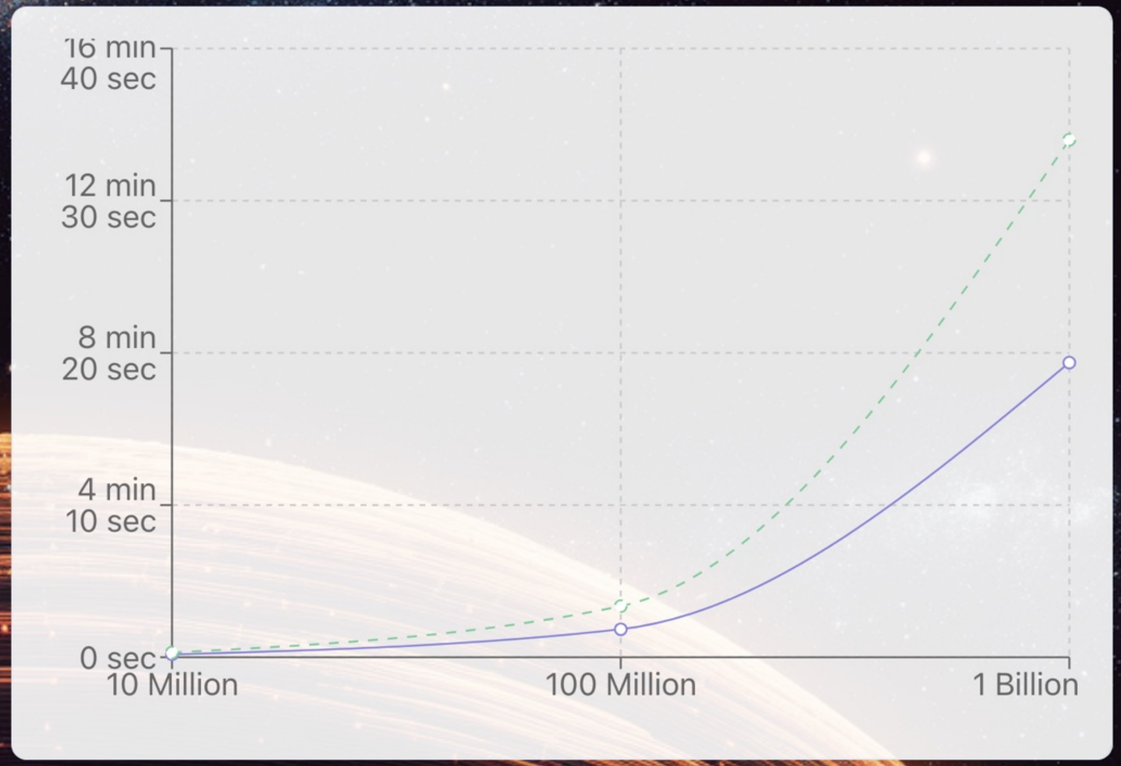
46 sec

↘ 45.2%

1 Billion Rows

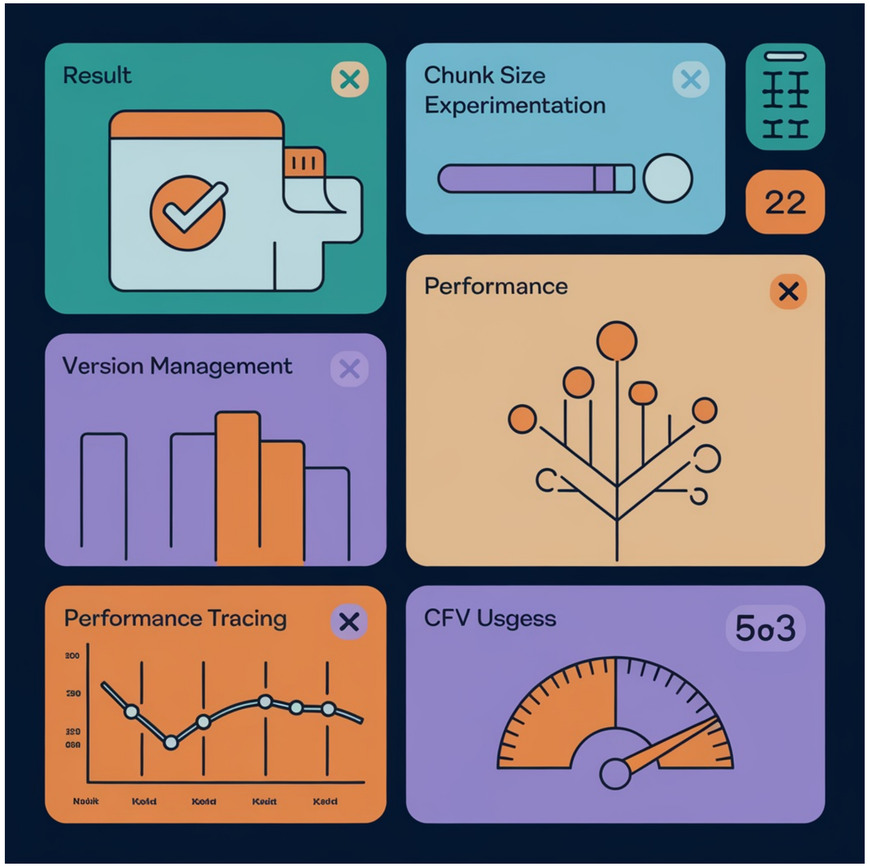
8 min 4 sec

↘ 43.1%



Need for better instrumentation

- Key needs:
  - Result correctness verification
  - Chunk size experimentation
  - Version management, performance comparison
  - Performance tracing
  - CPU usage measurement



***Demo**: maintaining & executing different versions, measuring cpu usage*

Profiling Elixir Code

- Pinpointing time-consuming code sections to detect perf bottlenecks.
- Frequency and duration of function calls, where they spend their time.
- A flame graph like visualisation would be nice.
- Measure the impact of optimisations

Profiling Elixir Code

- Pinpointing time-consuming code sections to detect perf bottlenecks.
- Frequency and duration of function calls, where they spend their time.
- A flame graph like visualisation would be nice.
- Measure the impact of optimisations
- cprof, prof, eflambe, benchee.

1. cprof

- Counts function calls, not execution time.
- Provides a quick overview of most called functions.
- Lightweight, minimal impact on code performance.

```
> run process_measurements.profile.cprof --count=1000 --version=3
Warmup ...

 ...
                                                                    CNT
Total                                                             61928
:lists                                                             6054  ⟵
  :lists.keyfind/3                                                 4013
   ...
Stream                                                             5032  ⟵
  Stream.do_resource/5                                             1002
  Stream.do_element_resource/6                                     1000
   ...
String                                                             5008  ⟵
  String.split/3                                                   2000
  String.split/2                                                   2000
  String.trim_trailing/1                                           1000
  String.do_replace/4                                                 2
   ...
String.Break                                                       5000  ⟵
  String.Break.trim_trailing/2                                     2000
   ...
OneBRC.MeasurementsProcessor.Version3                              4503  ⟵
  OneBRC.MeasurementsProcessor.Version3.process_row/2              1000
  OneBRC.MeasurementsProcessor.Version3.parse_row/1                1000
   ...
Profile done over 21915 matching functions
```

cprof output

1. cprof

- Counts function calls, not execution time.
- Provides a quick overview of most called functions.
- Lightweight, minimal impact on code performance.

2. eprof

- Counts function calls and execution time per function.
- Provides a clearer picture of time consuming functions
- Output shows percentage function calls, percentage ti per function and microseconds / call

```
> run process_measurements.profile.eprof --count=1000 --version=3
Warmup ...

...

Profile results of #PID<0.228.0>
#                                                                  CALLS     % TIME µS/CALL
Total                                                              37216 100. 3009    0.08

...

IO.each_binstream/2                                                 1001 2.82   85    0.08
Stream.do_resource/5                                                1002 2.82   85    0.08
anonymous fn/4 in Stream.chunk_while_fun/2                          1000 3.09   93    0.09
String.Chars.Float.to_string/1                                      1126 3.59  108    0.10
Enum."-map/2-lists^map/1-1-"/2                                      1188 3.82  115    0.10
OneBRC.MeasurementsProcessor.Version3.process_row/2                 1000 3.99  120    0.12
:ets.lookup/2                                                       1018 4.05  122    0.12
:prim_file.read_line_1/4                                            1002 4.12  124    0.12
:ets.insert/2                                                       1000 5.38  162    0.16
:prim_file.read_line/1                                              1001 5.95  179    0.18
```

eprof output

3. eFlambe

README Apache-2.0 license

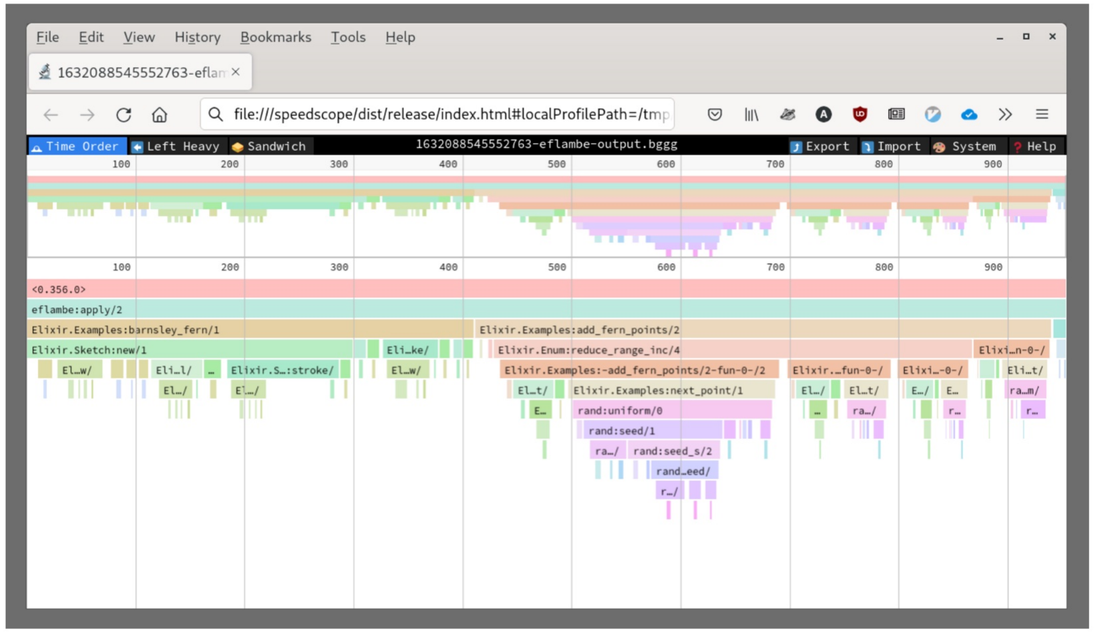
eFlambè!

A tool for rapid profiling of Erlang and Elixir applications

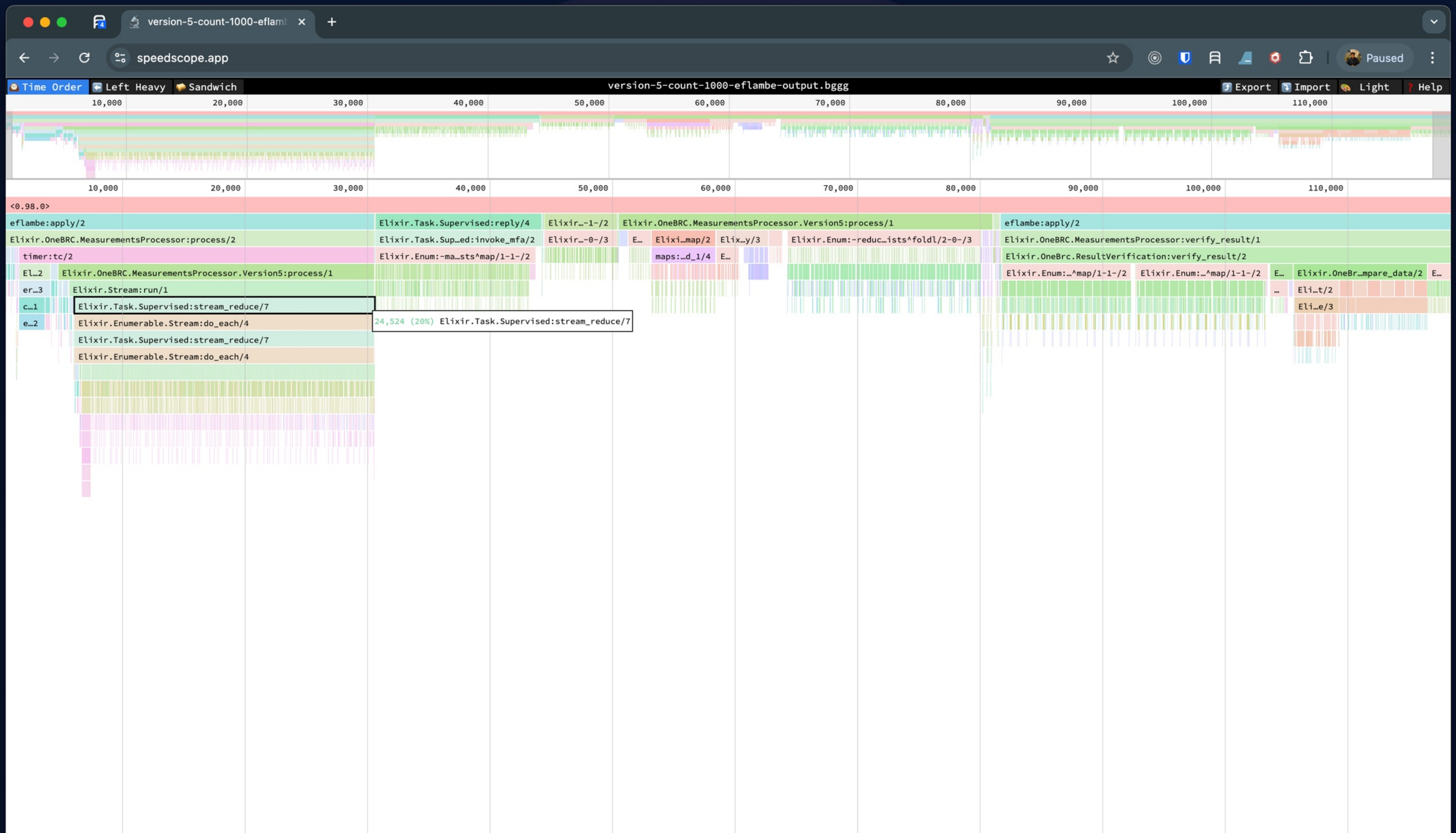
CI failing hex v0.3.1 documentation

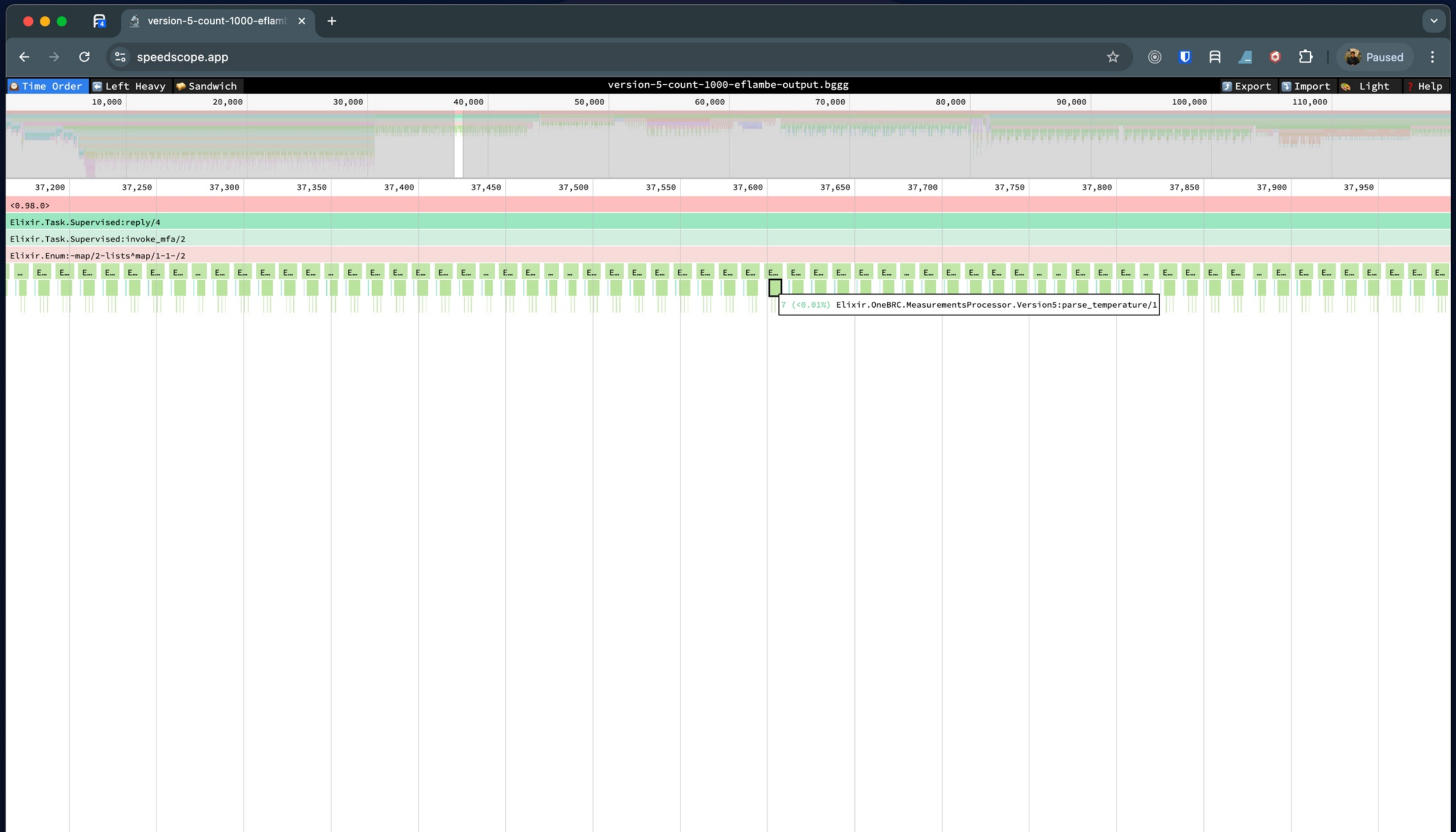
- Designed for rapid generation of flamegraphs - no need to edit source code or recompile!
- Generates output in several common formats - perf and Brendan Gregg's
- Interface inspired by recon_trace to automatically stop flamegraph generation
- Can automatically open data in your flamegraph viewer of choice

Output Viewed in Speedscope



Watch a quick demo here


version-5-count-1000-eflamt
speedscope.app
Paused
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
<0.98.0>
eflambe:apply/2
Elixir.Task.Supervised:reply/4
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.OneBRC.MeasurementsProcessor:process/2
Elixir.Task.Sup…ed:invoke_mfa/2
Elixir.OneBRC.MeasurementsProcessor:verify_result/1
timer:tc/2
Elixir.Enum:-ma…sts^map/1-1-/2
Elixir.OneBrc.ResultVerification:verify_result/2
El…2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Enum:…^map/1-1-/2
er…3
Elixir.Stream:run/1
c…1
Elixir.Task.Supervised:stream_reduce/7
e…2
Elixir.Enumerable.Stream:do_each/4
24,524 (20%) Elixir.Task.Supervised:stream_reduce/7
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4


version-5-count-1000-eflamt
speedscope.app
Paused
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
37,200
37,250
37,300
37,350
37,400
37,450
37,500
37,550
37,600
37,650
37,700
37,750
37,800
37,850
37,900
37,950
<0.98.0>
Elixir.Task.Supervised:reply/4
Elixir.Task.Supervised:invoke_mfa/2
Elixir.Enum:-map/2-lists^map/1-1-/2
7 (<0.01%) Elixir.OneBRC.MeasurementsProcessor.Version5:parse_temperature/1

4. Benchee

- Library for easy benchmarking in Elixir.
- Compares the performance of different pieces of code.
- Runs each of your functions for a given amount of time after an initial warmup.
- Measures execution time, memory consumption.
- Provides a plethora of statistics - avg, ips, median, 99p.
- Has different visualisation plugins for creating reports - HTML, Markdown, JSON.

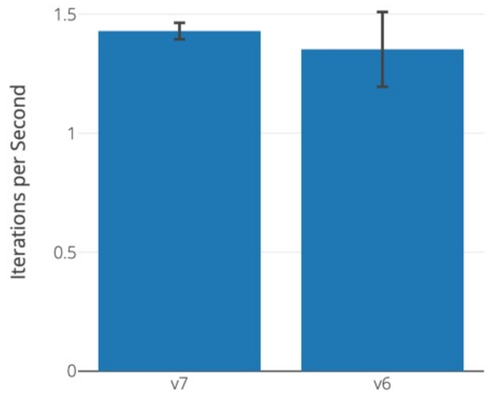
benchee report

System info

Run Time Comparison

Name	Iterations per Second	Average	Deviation	Median	Mode	Minimum	Maximum	Sample size
v7	1.43	699.57 ms	±2.40%	707.20 ms	none	674.78 ms	714.78 ms	8
v6	1.35	739.26 ms	±11.60%	780.37 ms	none	577.56 ms	799.37 ms	7

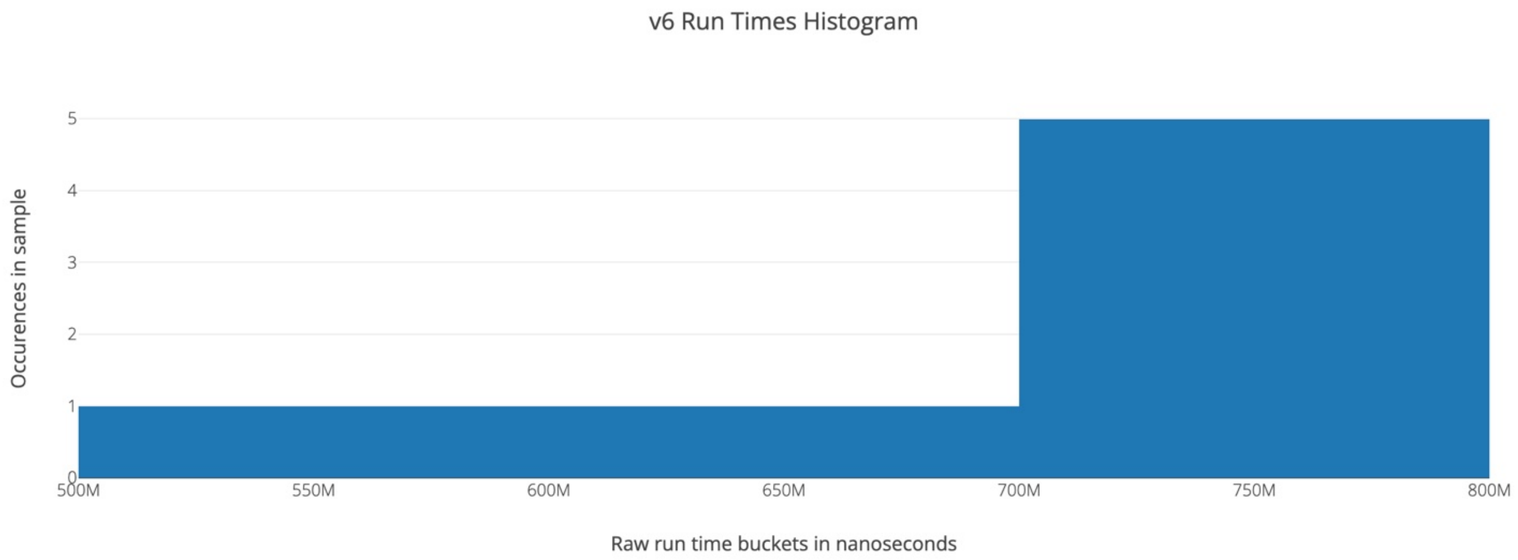
Average Iterations per Second

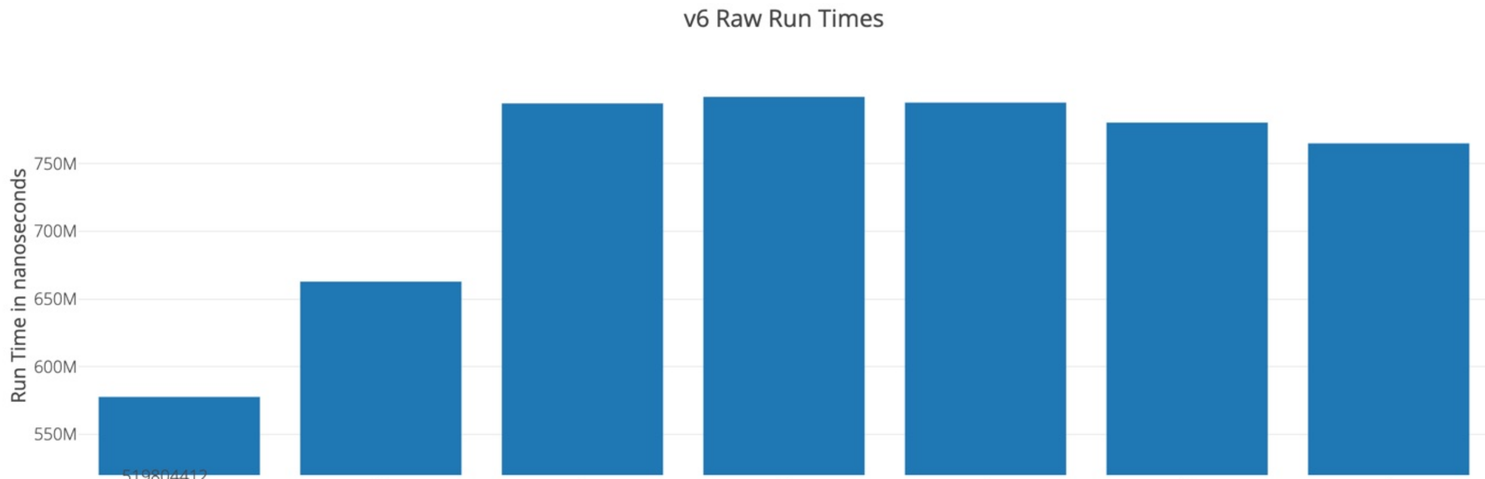


Run Time Boxplot

v6 Run Time

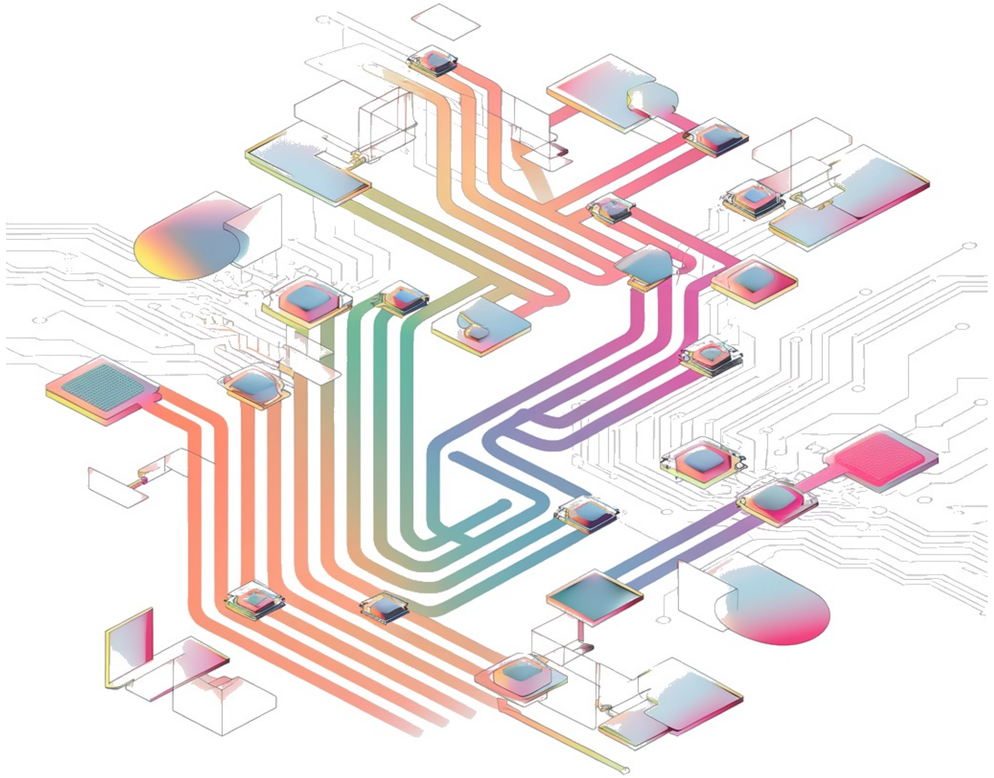
Name	Iterations per Second	Average	Deviation	Median	Mode	Minimum	Maximum	Sample size
v6	1.35	739.26 ms	±11.60%	780.37 ms	none	577.56 ms	799.37 ms	7





Version 4: Making processing rows concurrent

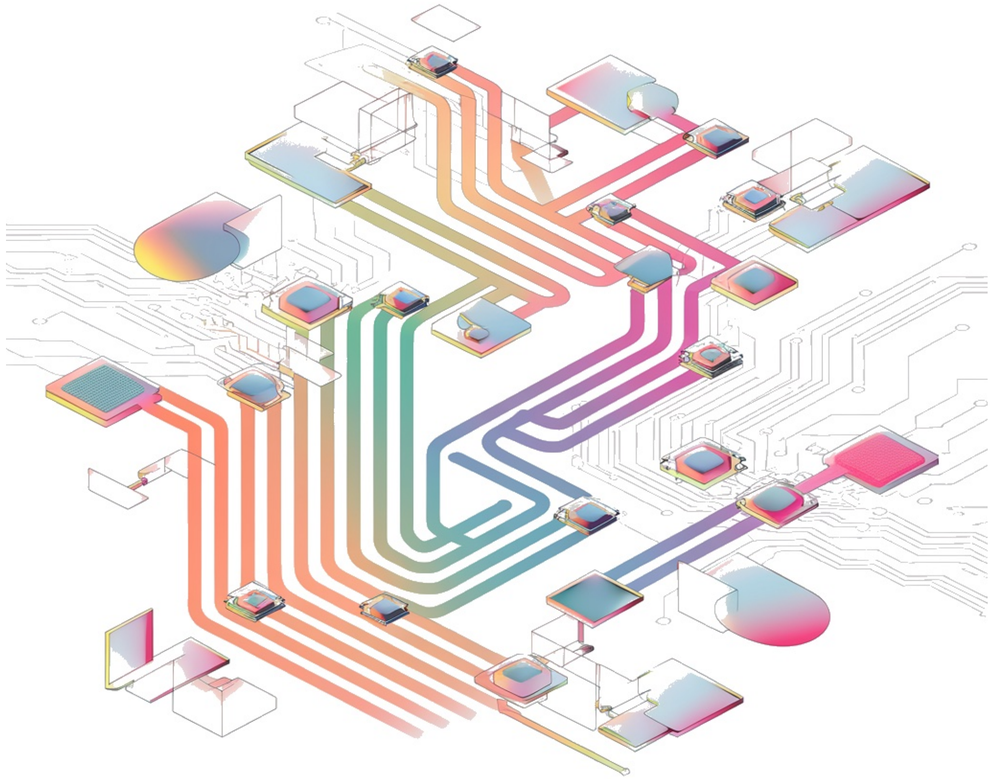
- Concurrency currently limited to the parsing step.
- Processing step was sequential due to shared state.



```elixir
defp process_row([key, value], ets_table) do
    {val, _} = Float.parse(value)
    existing_record = :ets.lookup(ets_table, key)
    new_record =
      case existing_record do
        [] ->
            %{
            min: val,
            max: val,
            sum: val,
            count: 1
            }
        [{^key, record}] ->
            min = if val < record.min, do: val, else: record.min
            max = if val > record.max, do: val, else: record.max
            sum = record.sum + val
            count = record.count + 1
            %{
                min: min,
                max: max,
                sum: sum,
                count: count
            }
        end
    :ets.insert(ets_table, {key, new_record})
end
```

Version 4: Making processing rows concurrent

- Concurrency currently limited to the parsing step.
- Processing step was sequential due to shared state.
- Proposed solution: Divide data into chunks.
- Each chunk produces intermediate results independently.
- A single pass at the end combines intermediate results for the final output.



lib/one_brc/measurements_processor.ex +79 -17

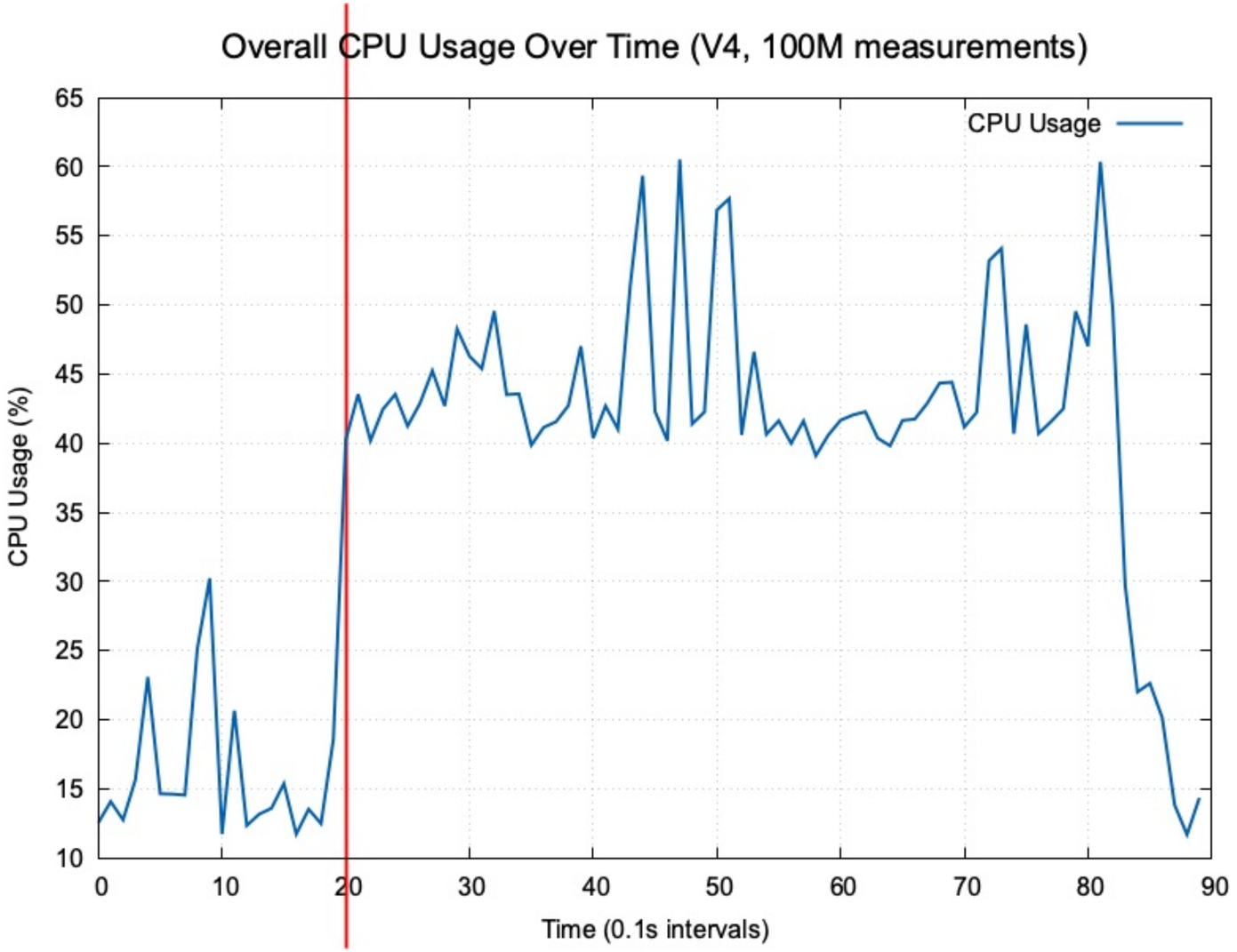
```
-  defp process_row([key, val], ets_table) do
-    existing_record = :ets.lookup(ets_table, key)

     new_record =
       case existing_record do
-        [] ->
           %{
             min: val,
             max: val,
             mean: val,
             count: 1
           }

-        [{^key, record}] ->
-          min = if val < record.min, do: val, else: record.min
-          max = if val > record.max, do: val, else: record.max
-          mean = (record.mean * record.count + val) / (record.count + 1)

           %{
             min: min,
             max: max,
             mean: mean,
-            count: record.count + 1
+            count: count + 1
           }
       end

-    :ets.insert(ets_table, {key, new_record})
+  defp process_row([key, val], acc) do
+    existing_record = Map.get(acc, key, nil)

     new_record =
       case existing_record do
+        nil ->
           %{
             min: val,
             max: val,
             mean: val,
             count: 1
           }

+        %{count: count, min: min, max: max, mean: mean} ->
+          min = if val < min, do: val, else: min
+          max = if val > max, do: val, else: max
+
+          mean = (mean * count + val) / (count + 1)

           %{
             min: min,
             max: max,
             mean: mean,
+            count: count + 1
           }
       end

+    Map.put(acc, key, new_record)
```

lib/one_brc/measurements_processor.ex +79 -17

```
30        |> Task.async_stream(
31          fn val -> Enum.map(val, &parse_row/1) end,
32  -       max_concurrency: System.schedulers_online() * 5,
33          ordered: false,
34          timeout: :infinity
35        )
36  -     |> Stream.flat_map(&elem(&1, 1))
37  -     |> Stream.map(&process_row(&1, ets_table))
38        |> Stream.run()
39
```

```
30        |> Task.async_stream(
31          fn val -> Enum.map(val, &parse_row/1) end,
32  +       max_concurrency: System.schedulers_online(),
33  +       ordered: false,
34  +       timeout: :infinity
35  +     )
36  +     |> Stream.with_index()
37  +     |> Task.async_stream(
38  +       fn {{:ok, parsed_rows}, row_index} ->
39  +         interim_records =
40  +           Enum.reduce(parsed_rows, %{}, fn row, acc ->
41  +             process_row(row, acc)
42  +           end)
43  +
44  +         :ets.insert(ets_table, {row_index, interim_records})
45  +       end,
46  +       max_concurrency: System.schedulers_online(),
47          ordered: false,
48          timeout: :infinity
49        )
50        |> Stream.run()
51
```

```
initialize final accumulator
for each entry in ETS table do
  intermediate accumulator = entry.value
  for each key (city) in intermediate accumulator do
    if key exists in final accumulator then
      merge records:
      update min to the lesser of the two min values.
      update max to the greater of the two max values.
      calculate the new sum by adding both sums.
      calculate the new count by adding both counts.
    else
      add record to final accumulator
    end
  end
end
```



CPU usage of version 4

1BRC in Elixir: Version 4

10 Million Rows

3.3 sec

↘ 29.8%

100 Million Rows

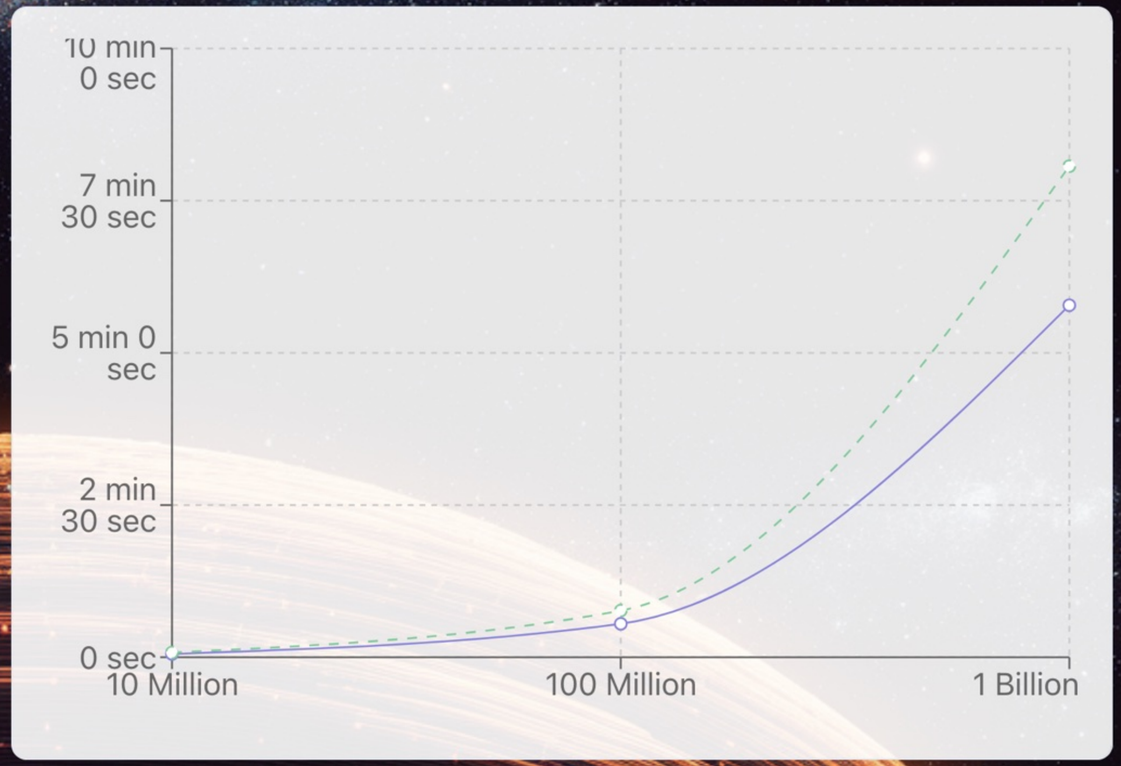
33 sec

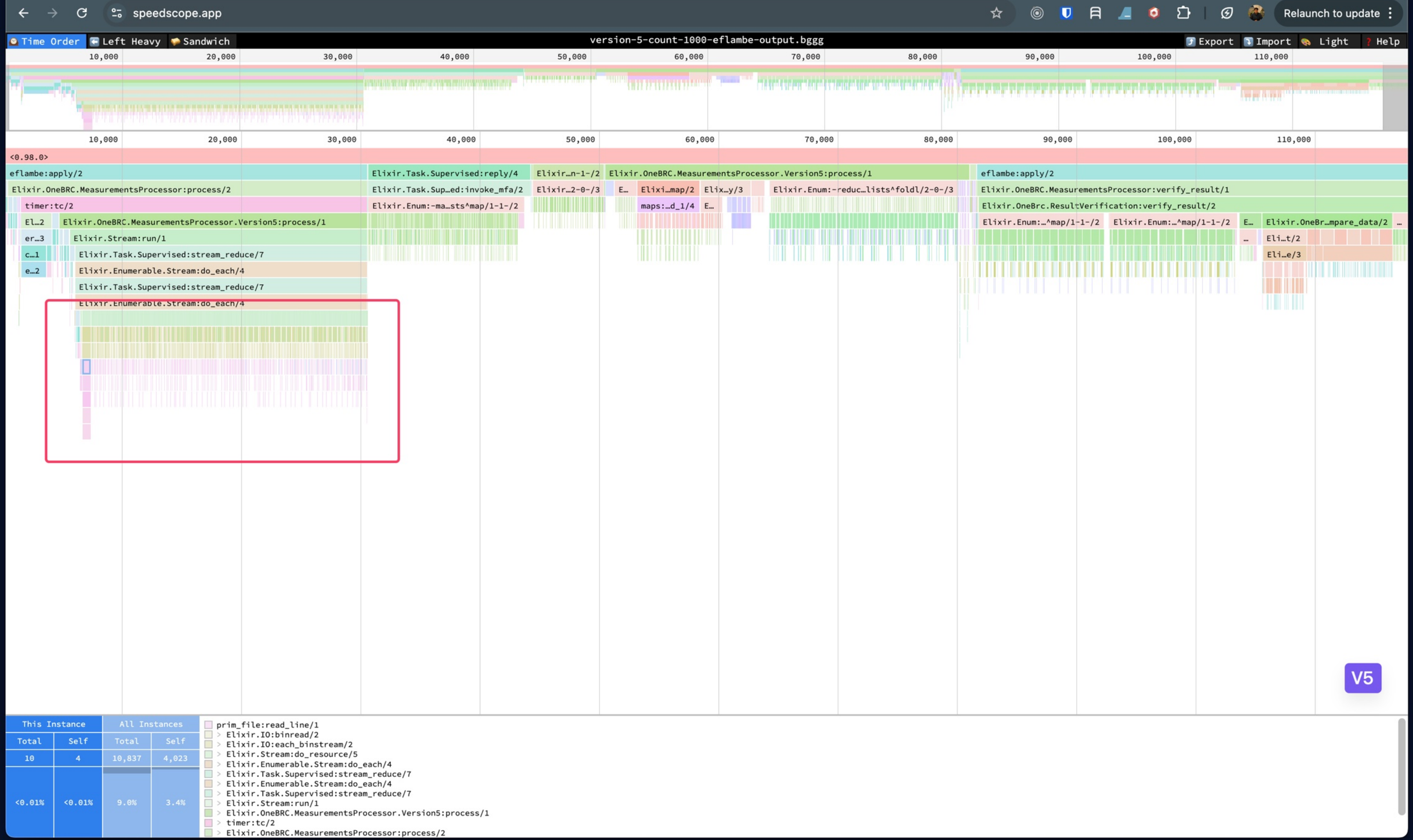
↘ 28.3%

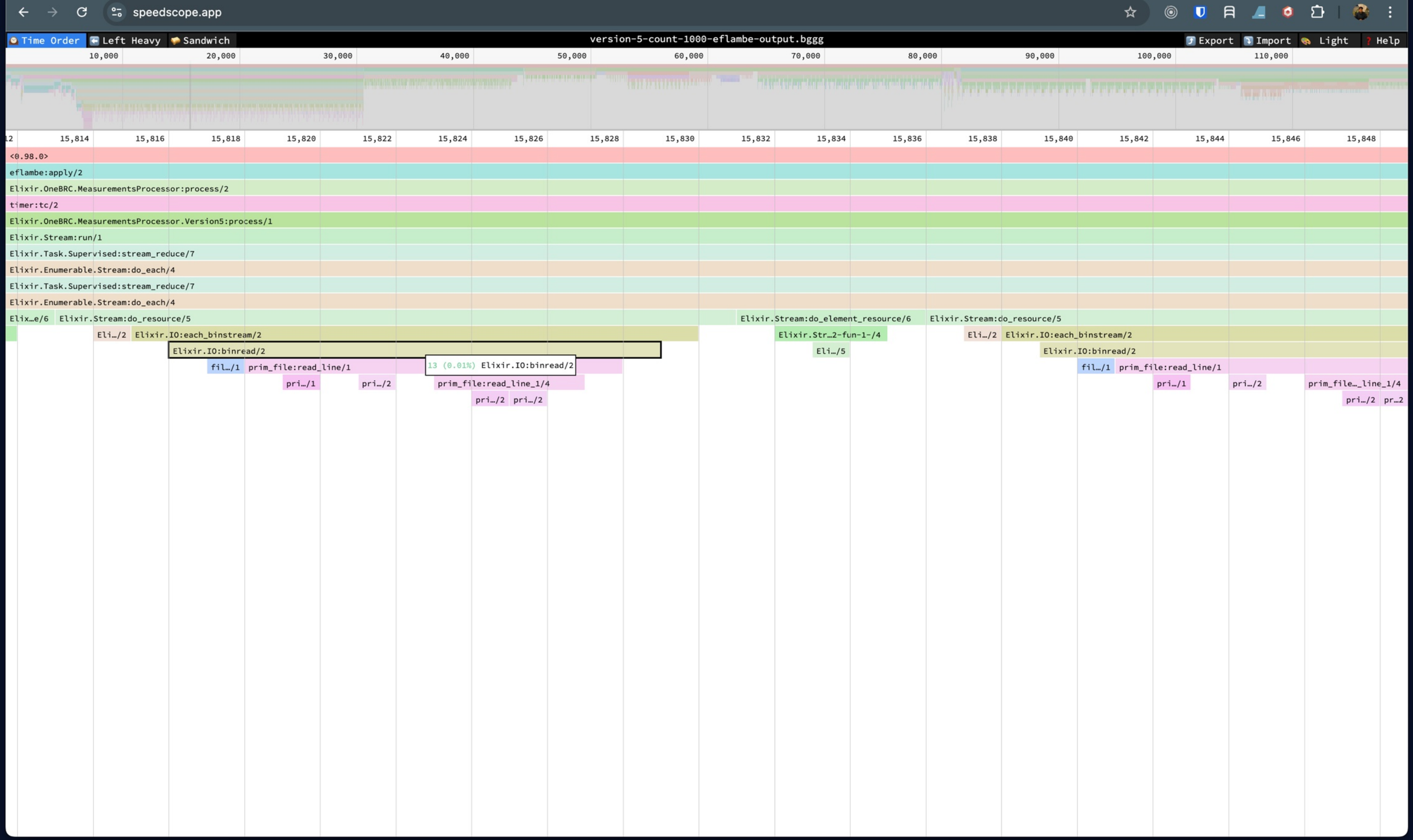
1 Billion Rows

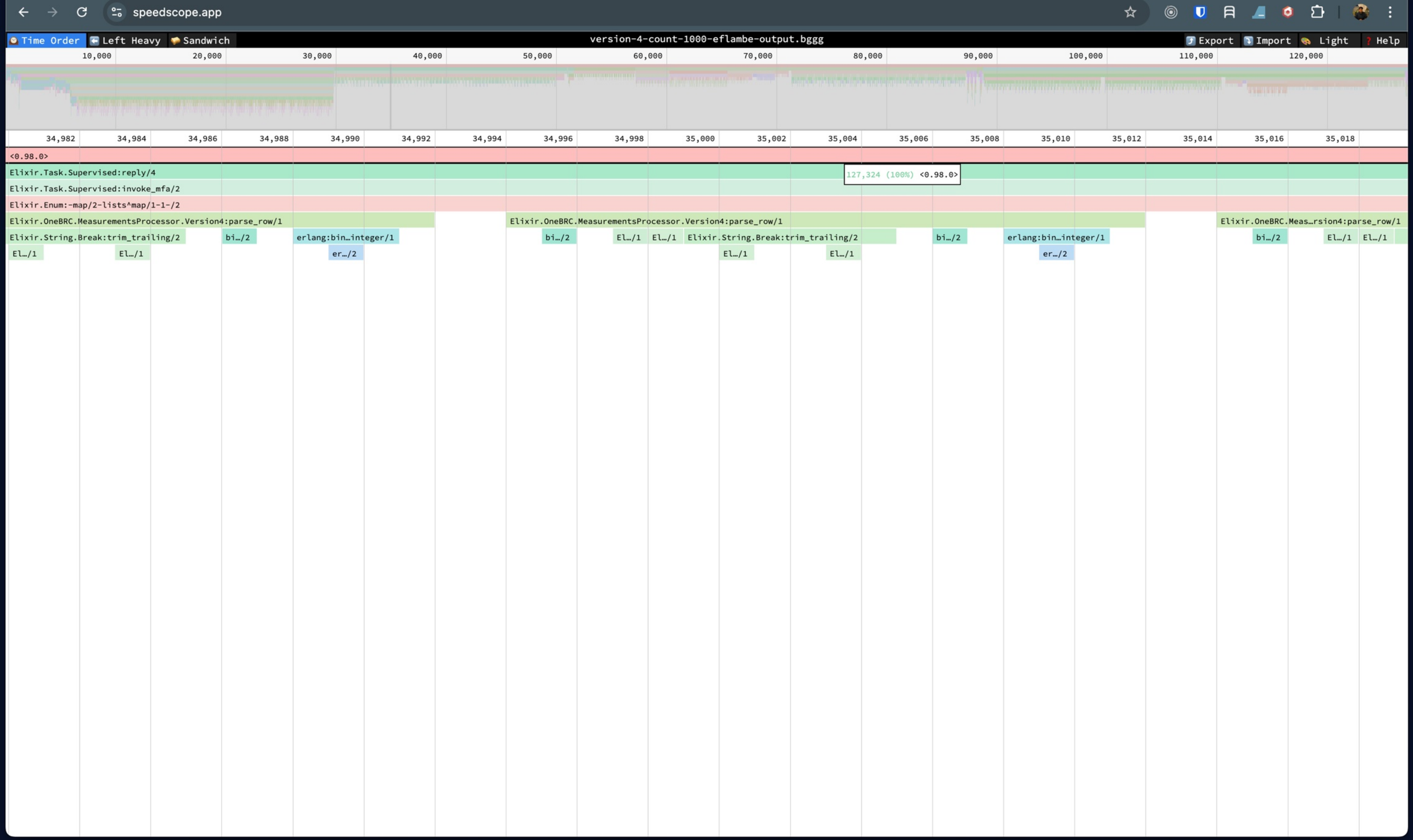
5 min 47 sec

↘ 28.3%



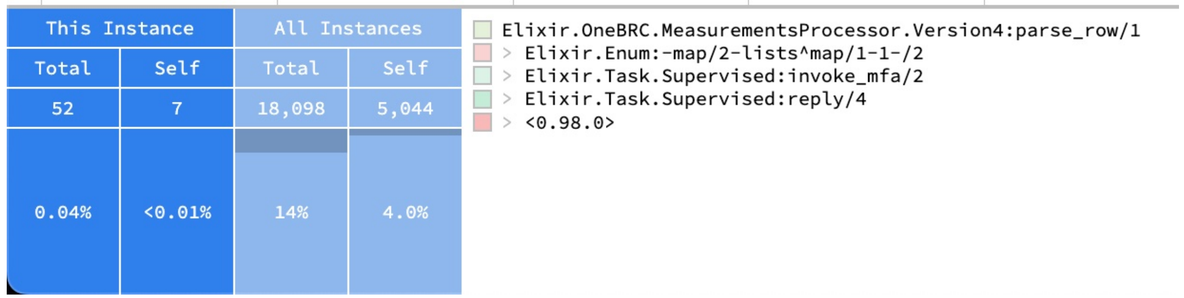

speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
<0.98.0>
eflambe:apply/2
Elixir.Task.Supervised:reply/4
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.OneBRC.MeasurementsProcessor:process/2
Elixir.Task.Supervised:invoke_mfa/2
Elixir.Enum:-reduc…lists^foldl/2-0-/3
Elixir.OneBRC.MeasurementsProcessor:verify_result/1
timer:tc/2
Elixir.Enum:-ma…sts^map/1-1-/2
Elixir.OneBrc.ResultVerification:verify_result/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Enum:…^map/1-1-/2
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
V5
This Instance
All Instances
Total
Self
10
4
10,837
4,023
<0.01%
<0.01%
9.0%
3.4%
prim_file:read_line/1
Elixir.IO:binread/2
Elixir.IO:each_binstream/2
Elixir.Stream:do_resource/5
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Stream:run/1
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor:process/2


speedscope.app
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
15,814
15,816
15,818
15,820
15,822
15,824
15,826
15,828
15,830
15,832
15,834
15,836
15,838
15,840
15,842
15,844
15,846
15,848
<0.98.0>
eflambe:apply/2
Elixir.OneBRC.MeasurementsProcessor:process/2
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elix…e/6
Elixir.Stream:do_resource/5
Elixir.Stream:do_element_resource/6
Elixir.Stream:do_resource/5
Eli…/2
Elixir.IO:each_binstream/2
Elixir.Str…2-fun-1-/4
Eli…/2
Elixir.IO:each_binstream/2
Elixir.IO:binread/2
Eli…/5
Elixir.IO:binread/2
fil…/1
prim_file:read_line/1
13 (0.01%) Elixir.IO:binread/2
fil…/1
prim_file:read_line/1
pri…/1
pri…/2
prim_file:read_line_1/4
pri…/1
pri…/2
prim_file…_line_1/4
pri…/2
pri…/2
pri…/2
pr…2


speedscope.app
Time Order
Left Heavy
Sandwich
version-4-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
120,000
34,982
34,984
34,986
34,988
34,990
34,992
34,994
34,996
34,998
35,000
35,002
35,004
35,006
35,008
35,010
35,012
35,014
35,016
35,018
<0.98.0>
Elixir.Task.Supervised:reply/4
127,324 (100%) <0.98.0>
Elixir.Task.Supervised:invoke_mfa/2
Elixir.Enum:-map/2-lists^map/1-1-/2
Elixir.OneBRC.MeasurementsProcessor.Version4:parse_row/1
Elixir.OneBRC.MeasurementsProcessor.Version4:parse_row/1
Elixir.OneBRC.Meas…rsion4:parse_row/1
Elixir.String.Break:trim_trailing/2
bi…/2
erlang:bin…integer/1
Elixir.String.Break:trim_trailing/2
bi…/2
erlang:bin…integer/1
bi…/2
El…/1
El…/1
El…/1
El…/1
El…/1
El…/1
El…/1
El…/1
er…/2
er…/2

Version 5: Some micro-optimisations to parsing

- String.split/3, String.trim_trailing/1 are being called a lot of times.
- Pattern matching to the rescue!



```
# sample row: `Mumbai;30.5\n`

# BEFORE:

[key, t_value] = :binary.split(row, “;")

[a, b] = t_value ▷ String.trim_trailing() ▷ :binary.split(".")

parsed_temp = (a <> b) ▷ String.to_integer()
```

Code Blame 3177 lines (2388 loc) · 94.1 KB Raw

```
2949
2950        String.to_integer("invalid data")
2951        ** (ArgumentError) argument error
2952
2953    """
2954    @spec to_integer(String.t()) :: integer
2955    def to_integer(string) when is_binary(string) do
2956      :erlang.binary_to_integer(string)
2957    end
2958
2959    @doc """
2960    Returns an integer whose text representation is `string` in base `base`.
2961
2962    Inlined by the compiler.
2963
2964    ## Examples
2965
2966        iex> String.to_integer("3FF", 16)
2967        1023
2968
2969    """
2970    @spec to_integer(String.t(), 2..36) :: integer
2971    def to_integer(string, base) when is_binary(string) and is_integer(base) do
2972      :erlang.binary_to_integer(string, base)
2973    end
2974
```

Code Blame 12723 lines (10733 loc) · 445 KB Raw

```
991   as  list_to_integer/1 .
992
993   Failure: `badarg` if `Binary` contains a bad representation of an integer.
994   """.
995   -doc(#{since => <<"OTP R16B">>}).
996   -doc #{ group => terms }.
997   -spec binary_to_integer(Binary) -> integer() when
998         Binary :: binary().
999   binary_to_integer(Binary) ->
1000      case erts_internal:binary_to_integer(Binary, 10) of
1001          N when erlang:is_integer(N) ->
1002              N;
1003          big ->
1004              case big_binary_to_int(Binary, 10) of
1005                  N when erlang:is_integer(N) ->
1006                      N;
1007                  Reason ->
1008                      error_with_info(Reason, [Binary])
1009              end;
1010          badarg ->
1011              badarg_with_info([Binary])
1012      end.
1013
1014  %% binary_to_integer/2
1015  -doc """
1016  Returns an integer whose text representation in base `Base` is `Binary`.
1017
1018  For example:
```

Code Blame 3244 lines (2895 loc) · 70.4 KB Raw

```
3118    }
3119
3120    BIF_RETTYPE erts_internal_binary_to_integer_2(BIF_ALIST_2)
3121    {
3122        const byte *temp_alloc = NULL, *bytes;
3123        Uint size;
3124        Uint base;
3125        Eterm res;
3126
3127        if (!is_small(BIF_ARG_2)) {
3128            BIF_RET(am_badarg);
3129        }
3130
3131        base = (Uint)signed_val(BIF_ARG_2);
3132
3133        if (base < 2 || base > 36) {
3134            BIF_RET(am_badarg);
3135        }
3136
3137        bytes = erts_get_aligned_binary_bytes(BIF_ARG_1, &size, &temp_alloc);
3138        if (bytes == NULL) {
3139            BIF_RET(am_badarg);
3140        }
3141
3142        res = chars_to_integer(bytes, size, base);
3143        erts_free_aligned_binary_bytes(temp_alloc);
3144        BIF_RET(res);
3145    }
```

Code Blame 3244 lines (2895 loc) · 70.4 KB Raw

```
3051  static Eterm chars_to_integer(const byte *bytes, Uint size, const Uint base)
3052  {
3053      Sint i = 0;
3054      int neg = 0;
3055
3056      if (size == 0) {
3057          return am_badarg;
3058      }
3059
3060      if (bytes[0] == '-') {
3061          neg = 1;
3062          bytes++;
3063          size--;
3064      } else if (bytes[0] == '+') {
3065          bytes++;
3066          size--;
3067      }
3068
3069      if (size == 0) {
3070          return am_badarg;
3071      }
3072
3073      /* Trim leading zeroes */
3074      while (*bytes == '0') {
3075          bytes++;
3076          size--;
3077          if (size == 0) {
3078              /* All zero! */
```

Code Blame 3244 lines (2895 loc) · 70.4 KB Raw

```
3091             * case.
3092             */
3093            while (size--) {
3094                Uint digit = *bytes++ - '0';
3095                if (digit >= base) {
3096                    return am_badarg;
3097                }
3098                i = i * base + digit;
3099            }
3100        } else {
3101            while (size) {
3102                byte b = *bytes++;
3103                size--;
3104
3105                if (c2int_is_invalid_char(b, base)) {
3106                    return am_badarg;
3107                }
3108
3109                i = i * base + c2int_digit_from_base(b);
3110            }
3111        }
3112
3113        if (neg) {
3114            i = -i;
3115        }
3116        ASSERT(IS_SSMALL(i));
3117        return make_small(i);
3118    }
```

Code Blame 3244 lines (2895 loc) · 70.4 KB Raw

```
2956            return (ch >= '0' && ch < ('0' + base));
2957        else
2958            return (ch >= '0' && ch <= '9')
2959                || (ch >= 'A' && ch < ('A' + base - 10))
2960                || (ch >= 'a' && ch < ('a' + base - 10));
2961    }
2962
2963    static ERTS_INLINE int c2int_is_invalid_char(byte ch, int base) {
2964        return !c2int_is_valid_char(ch, base);
2965    }
2966
2967    static ERTS_INLINE byte c2int_digit_from_base(byte ch) {
2968        return ch <= '9' ? ch - '0'
2969                : (10 + (ch <= 'Z' ? ch - 'A' : ch - 'a'));
2970    }
2971
2972    /*
2973     * How many bits are needed to store 1 digit of given base in binary
2974     * Wo.Alpha formula: Table [log2[n], {n,2,36}]
2975     */
2976    static const double lg2_lookup[36-1] = {
2977        1.0, 1.58496, 2.0, 2.32193, 2.58496, 2.80735, 3.0, 3.16993, 3.32193,
2978        3.45943, 3.58496, 3.70044, 3.80735, 3.90689, 4.0, 4.08746, 4.16993, 4.24793,
2979        4.32193, 4.39232, 4.45943, 4.52356, 4.58496, 4.64386, 4.70044, 4.75489,
2980        4.80735, 4.85798, 4.90689, 4.9542, 5.0, 5.04439, 5.08746, 5.12928, 5.16993
2981    };
2982
2983    /*
```

```
iex(1)> ?A
65

iex(2)> ?B
66

iex(3)> ?0
48

iex(4)> ?1
49

iex(5)> ?2
50

iex(6)> ?3
51
```

```
iex(1)> ?5 - ?0
5
```

- One digit before the decimal (`'4'`)
- Decimal point (`'.'`)
- One digit after the decimal (`'5'`)

4.5

- One digit before the decimal (`'4'`)
- Decimal point (`'.'`)
- One digit after the decimal (`'5'`)
- `<<d1, ?., d2, _::binary>>`
  - *Also eliminates the need for String.trim_trailing*

4.5

```elixir
defp parse_temperature(<<d1, ?., d2, _::binary>>) do
    char_to_num(d1) * 10 + char_to_num(d2)
end
```

```
# sample row: `Mumbai;30.5\n`
# AFTER:

parsed_temp = t_value |> parse_temperature()
...
# ex: -4.5
defp parse_temperature(<<?-, d1, ?., d2, _::binary>>) do
    -(char_to_num(d1) * 10 + char_to_num(d2))
end

# ex: 4.5
defp parse_temperature(<<d1, ?., d2, _::binary>>) do
    char_to_num(d1) * 10 + char_to_num(d2)
end

# ex: -45.3
defp parse_temperature(<<?-, d1, d2, ?., d3, _::binary>>) do
    -(char_to_num(d1) * 100 + char_to_num(d2) * 10 + char_to_num(d3))
end

# ex: 45.3
defp parse_temperature(<<d1, d2, ?., d3, _::binary>>) do
    char_to_num(d1) * 100 + char_to_num(d2) * 10 + char_to_num(d3)
end

defp char_to_num(char) do
    char - ?0
end
```

1BRC in Elixir: Version 5

10 Million Rows

3.1 sec

↘ 6.1%

100 Million Rows

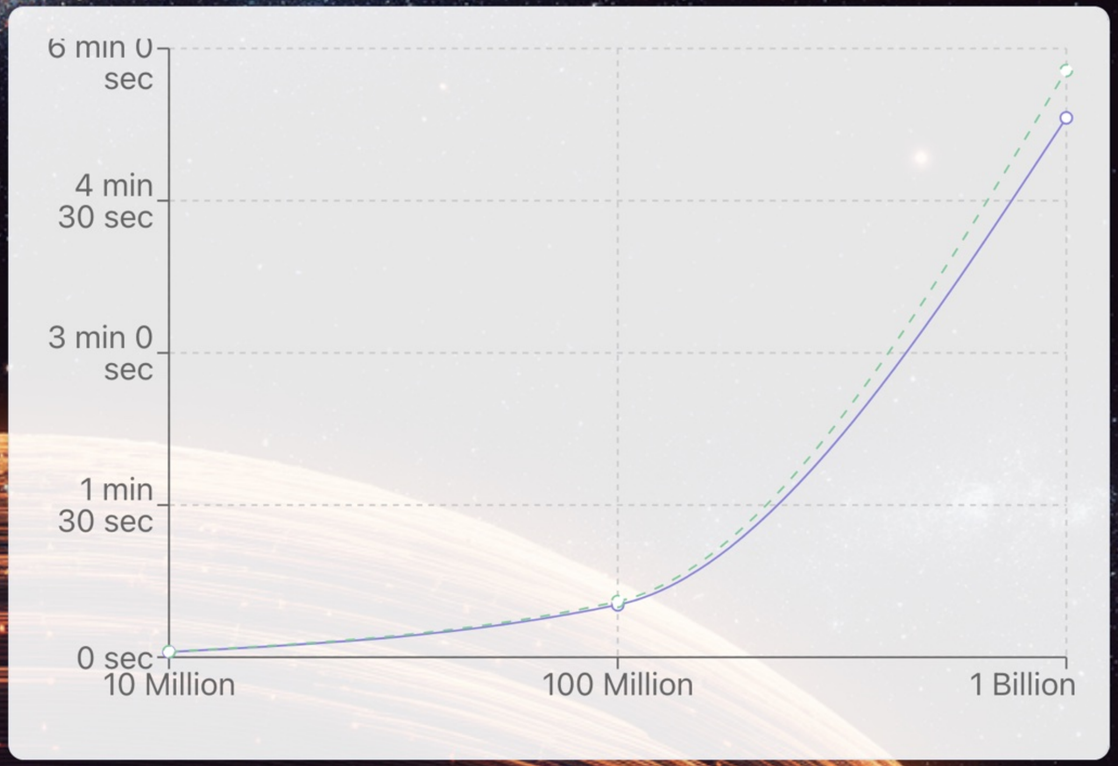
31 sec

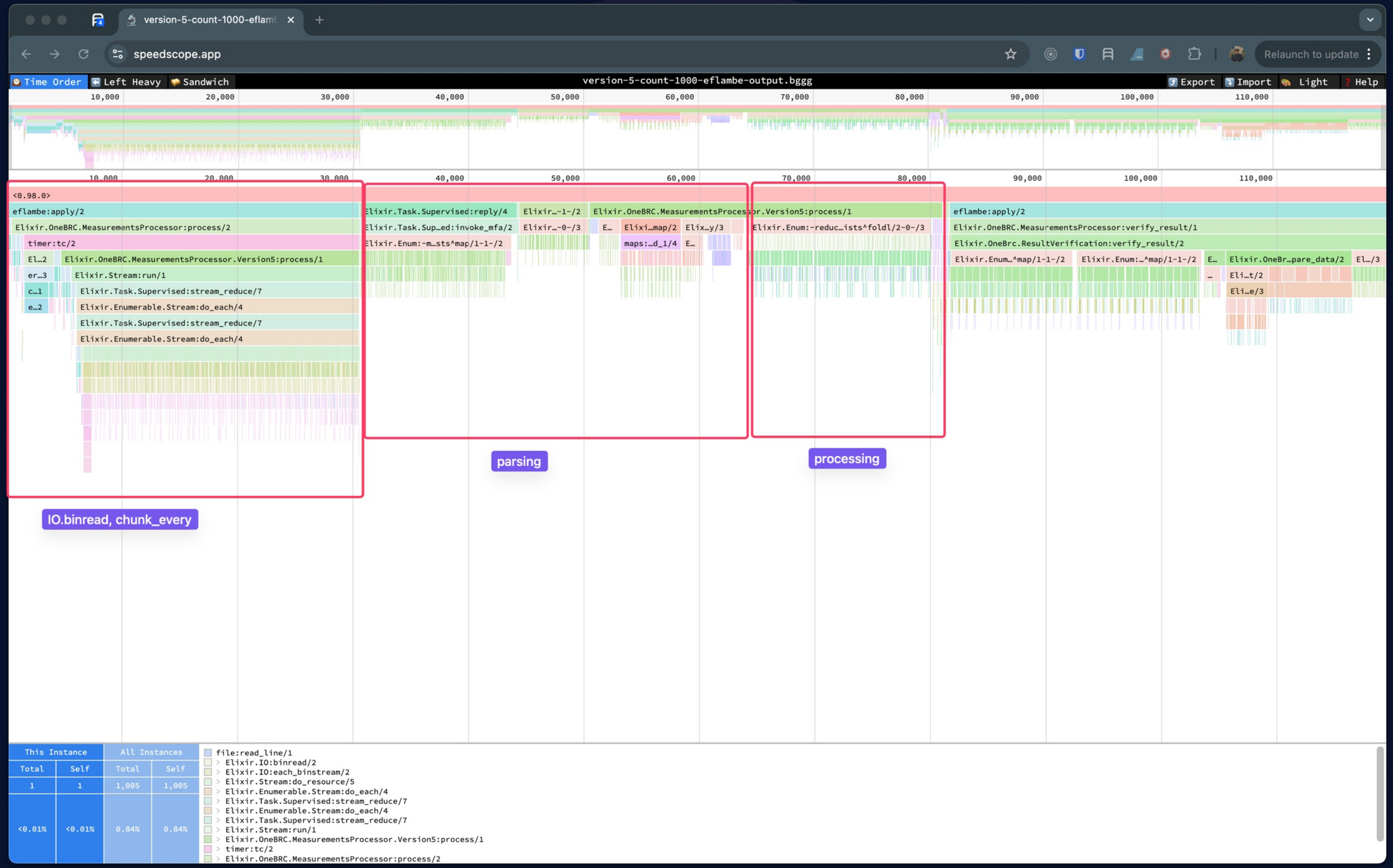
↘ 6.1%

1 Billion Rows

5 min 19 sec

↘ 8.1%




version-5-count-1000-eflam
speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
<0.98.0>
eflambe:apply/2
Elixir.OneBRC.MeasurementsProcessor:process/2
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:reply/4
Elixir.Task.Supervised:invoke_mfa/2
Elixir.Enum:-m…sts^map/1-1-/2
Elixir.Enum:-reduc…ists^foldl/2-0-/3
Elixir.OneBRC.MeasurementsProcessor:verify_result/1
Elixir.OneBrc.ResultVerification:verify_result/2
Elixir.Enum…^map/1-1-/2
Elixir.Enum:…^map/1-1-/2
Elixir.OneBr…pare_data/2
parsing
processing
IO.binread, chunk_every
This Instance
All Instances
Total
Self
1
1,005
<0.01%
0.84%
file:read_line/1
Elixir.IO:binread/2
Elixir.IO:each_binstream/2
Elixir.Stream:do_resource/5
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Stream:run/1
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor:process/2

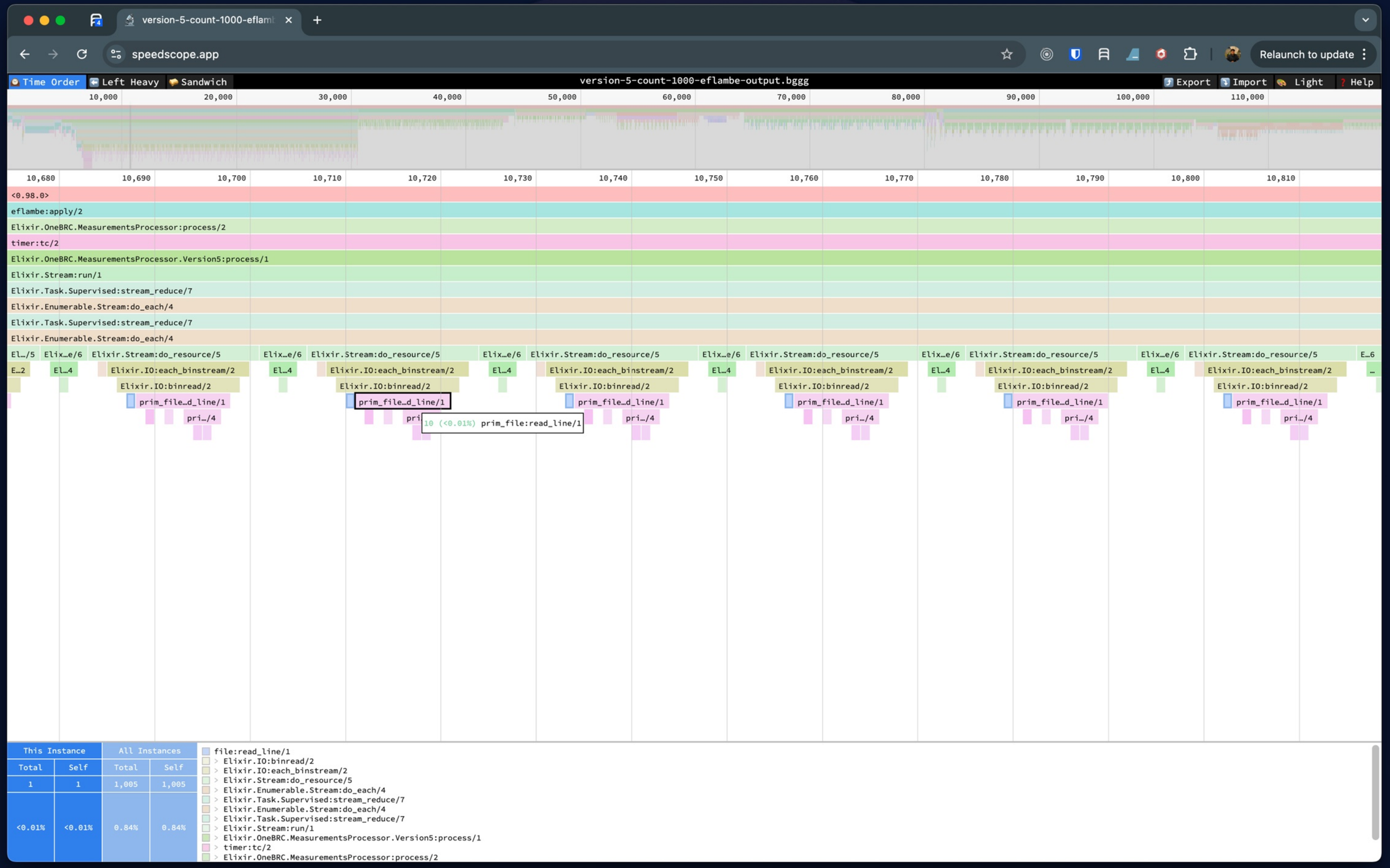
`stream!(path, line_or_bytes, modes)`

```
@spec stream!(Path.t(), :line | pos_integer(), [stream_mode()]) :: File.Stream.t()
```

Returns a `File.Stream` for the given `path` with the given `modes`.

The stream implements both `Enumerable` and `Collectable` protocols, which means it can be used both for read and write.

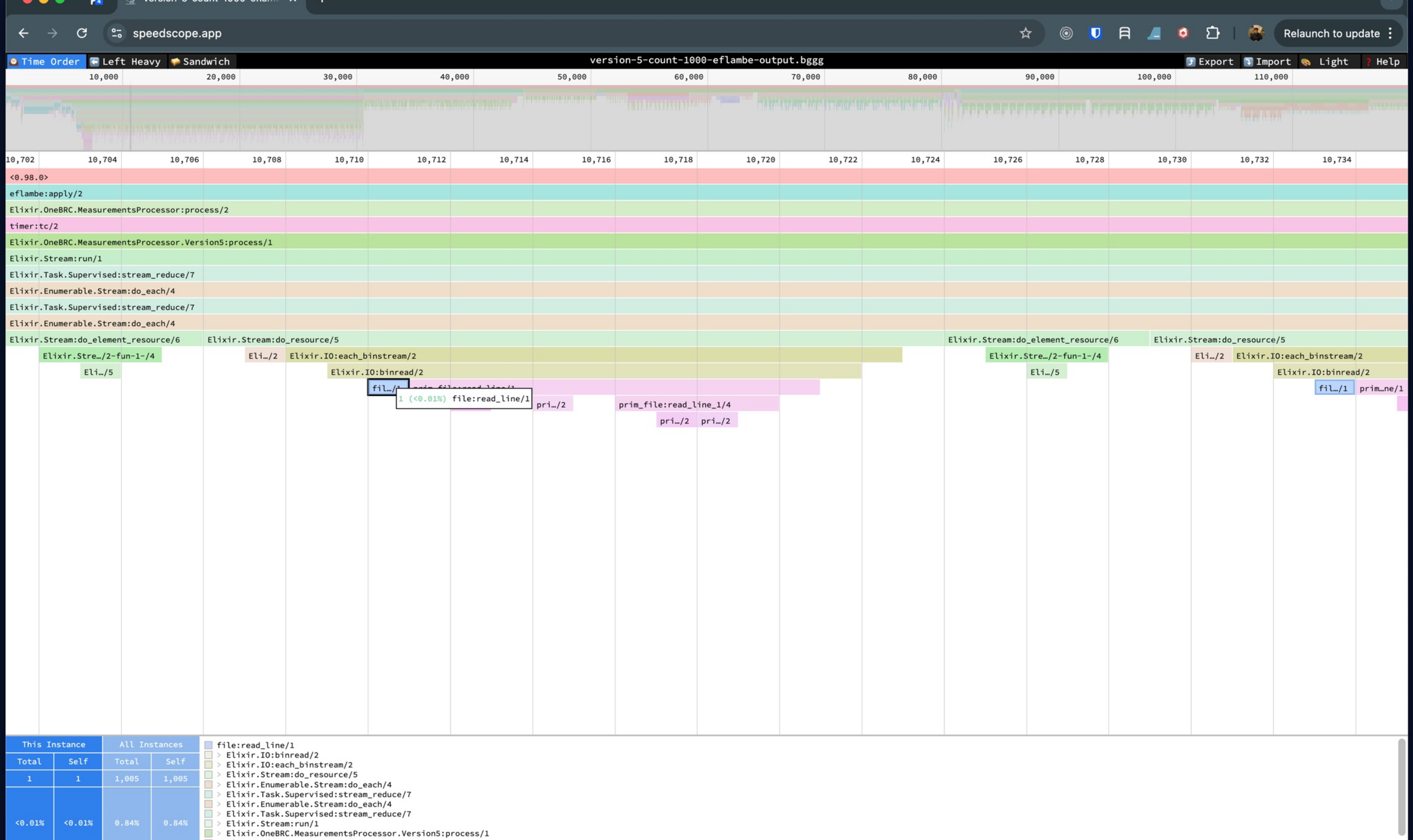
The `line_or_bytes` argument configures how the file is read when streaming, by `:line` (default) or by a given number of bytes. When using the `:line` option, CRLF line breaks (`"\r\n"`) are normalized to LF (`"\n"`).

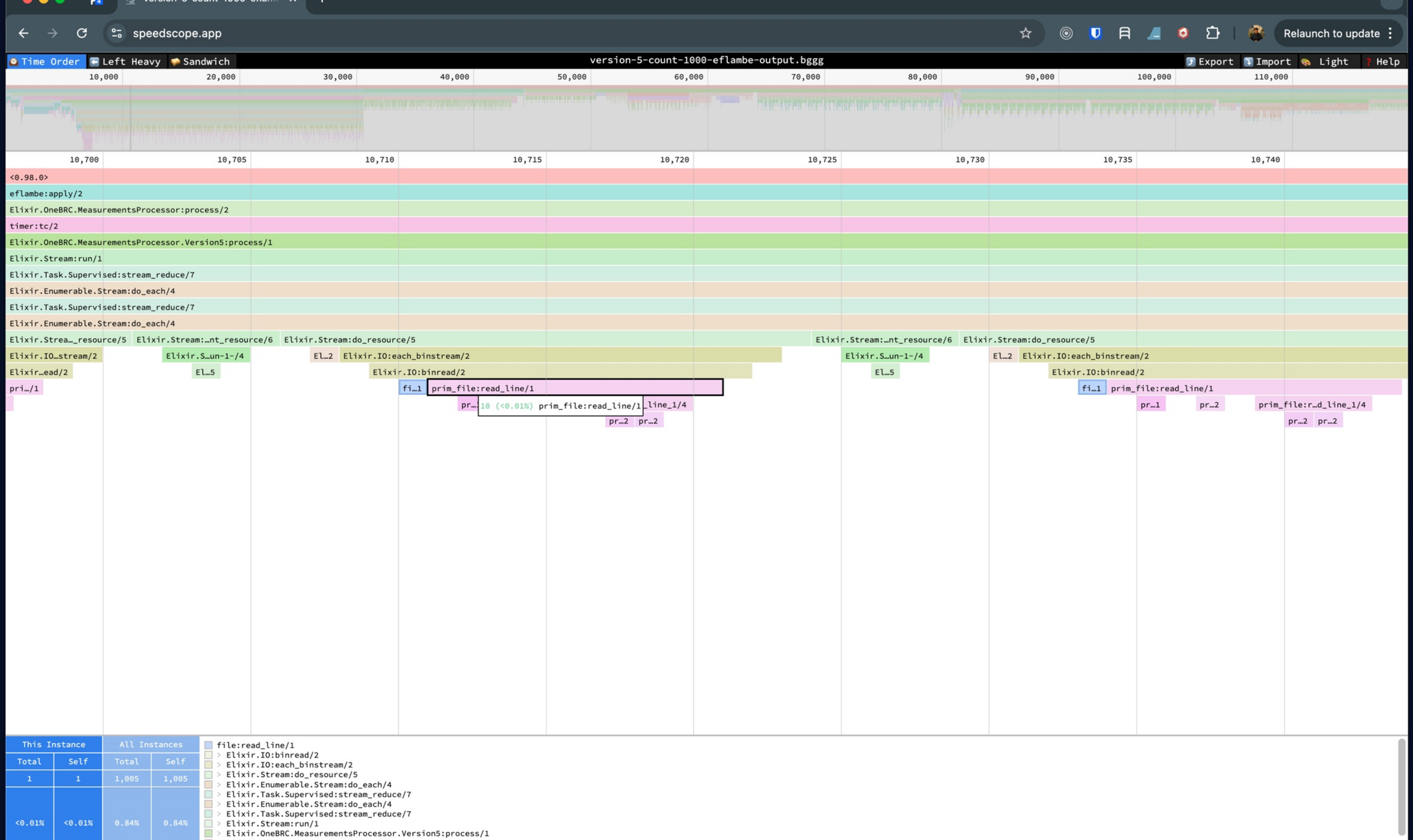

version-5-count-1000-eflam
speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
10,680
10,690
10,700
10,710
10,720
10,730
10,740
10,750
10,760
10,770
10,780
10,790
10,800
10,810
<0.98.0>
eflambe:apply/2
Elixir.OneBRC.MeasurementsProcessor:process/2
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Stream:do_resource/5
Elixir.IO:each_binstream/2
Elixir.IO:binread/2
prim_file…d_line/1
10 (<0.01%) prim_file:read_line/1
This Instance
All Instances
Total
Self
1
1,005
<0.01%
0.84%
file:read_line/1
Elixir.IO:binread/2
Elixir.IO:each_binstream/2
Elixir.Stream:do_resource/5
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Stream:run/1
timer:tc/2

```
 ...
Tokyo;16.8\nCape Town;22.3\nStockholm;7.1\n
Marrakech;29.6\nBerlin;9.7\nAuckland;17.5\n
Bangkok;32.7\nReykjavik;3.4\nVienna;14.6\n
Moscow;1.9\nBerlin;6.3\nHanoi;30.2\n
Cairo;33.5\nOslo;5.7\nKyoto;18.1\n
Miami;27.6\nBerlin;11.2\nPrague;15.9\n
Istanbul;23.4\nEdinburgh;10.8\nSeattle;13.7\n
Dubai;36.8\nQueenstown;9.9\n
 ...
```


...
Tokyo;16.8\nCape Town;22.3\nStockholm;7.1\n
Marrakech;29.6\nBerlin;9.7\nAuckland;17.5\n
Bangkok;32.7\nReykjavik;3.4\nVienna;14.6\n
Moscow;1.9\nBerlin;6.3\nHanoi;30.2\n
Cairo;33.5\nOslo;5.7\nKyoto;18.1\n
Miami;27.6\nBerlin;11.2\nPrague;15.9\n
Istanbul;23.4\nEdinburgh;10.8\nSeattle;13.7\n
Dubai;36.8\nQueenstown;9.9\n
...

```
 ...
Tokyo;16.8\nCape Town;22.3\nStockholm;7.1\n
Marrakech;29.6\nBerlin;9.7\nAuckland;17.5\n
Bangkok;32.7\nReykjavik;3.4\nVienna;14.6\n
Moscow;1.9\nBerlin;6.3\nHanoi;30.2\n
Cairo;33.5\nOslo;5.7\nKyoto;18.1\n
Miami;27.6\nBerlin;11.2\nPrague;15.9\n
Istanbul;23.4\nEdinburgh;10.8\nSeattle;13.7\n
Dubai;36.8\nQueenstown;9.9\n
 ...
```


speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
10,702
10,704
10,706
10,708
10,710
10,712
10,714
10,716
10,718
10,720
10,722
10,724
10,726
10,728
10,730
10,732
10,734
<0.98.0>
eflambe:apply/2
Elixir.OneBRC.MeasurementsProcessor:process/2
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Stream:do_element_resource/6
Elixir.Stream:do_resource/5
Elixir.Stream:do_element_resource/6
Elixir.Stream:do_resource/5
Elixir.Stre…/2-fun-1-/4
Eli…/2
Elixir.IO:each_binstream/2
Eli…/5
Elixir.IO:binread/2
fil…/1
1 (<0.01%) file:read_line/1
pri…/2
prim_file:read_line_1/4
pri…/2
pri…/2
Elixir.Stre…/2-fun-1-/4
Eli…/2
Elixir.IO:each_binstream/2
Eli…/5
Elixir.IO:binread/2
fil…/1
prim…ne/1
This Instance
All Instances
Total
Self
Total
Self
1
1
1,005
1,005
<0.01%
<0.01%
0.84%
0.84%
file:read_line/1
Elixir.IO:binread/2
Elixir.IO:each_binstream/2
Elixir.Stream:do_resource/5
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Stream:run/1
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1


speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
10,700
10,705
10,710
10,715
10,720
10,725
10,730
10,735
10,740
<0.98.0>
eflambe:apply/2
Elixir.OneBRC.MeasurementsProcessor:process/2
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Strea…_resource/5
Elixir.Stream:…nt_resource/6
Elixir.Stream:do_resource/5
Elixir.Stream:…nt_resource/6
Elixir.Stream:do_resource/5
Elixir.IO…stream/2
Elixir.S…un-1-/4
El…2
Elixir.IO:each_binstream/2
Elixir.S…un-1-/4
El…2
Elixir.IO:each_binstream/2
Elixir…ead/2
El…5
Elixir.IO:binread/2
El…5
Elixir.IO:binread/2
pri…/1
fi…1
prim_file:read_line/1
fi…1
prim_file:read_line/1
pr…
10 (<0.01%) prim_file:read_line/1
_line_1/4
pr…1
pr…2
prim_file:r…d_line_1/4
pr…2
pr…2
pr…2
pr…2
This Instance
All Instances
Total
Self
Total
Self
1
1
1,005
1,005
<0.01%
<0.01%
0.84%
0.84%
file:read_line/1
Elixir.IO:binread/2
Elixir.IO:each_binstream/2
Elixir.Stream:do_resource/5
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Stream:run/1
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1

nato Aug/2023

I swapped out some file:read_file and file:write_files today, for some prim_file ones, and I noticed an astounding speedup. Other than the the `don't use this module' standard reply, can I get some context on this module, and why it's there (along with the other erts prim modules)?

Appreciate the info!!

4

629 views 18 likes 2 links 7 users




mikpe Aug/2023

`prim_file` is a building block for the so-called "file server". The proper question to ask is why does Erlang need the file server. (I don't *know* the answer to that, and I prefer not to speculate.)

1

Prim_file speedup

Questions / Help | prim_file

Log In

1

raimo Erlang Core Team Aug/2023

The purpose of this indirection is to be able to run on a diskless machine, so a node can be configured to run with a remote file server. Therefore `file` calls a file server process for all its operations. The file server uses `prim_file` as backend to its local file system.

Applications then does not need to know on which machine the file system is, as long as they are using `file` to access it.

There is a `raw` file mode option that can be used for most file operations, e.g. for `file:write_file/3`. Try it and see if the performance is close enough to `prim_file:write_file/3`.

Unfortunately; `file:read_file/1` does not today have an arity that takes a `Mode` argument…

2 Replies

5

mikpe raimo Aug/2023

It's one thing to have an indirection to add flexibility, it's another to indirect through a single (*)

Aug 2023

2 of 13
Aug 2023

Aug/2023

Code Blame 893 lines (790 loc) · 27.9 KB Raw

```
203  read_line(Fd) ->
204      try
205          #{ handle := FRef,
206             r_ahead_size := RASz,
207             r_buffer := RBuf } = get_fd_data(Fd),
208          SearchResult = prim_buffer:find_byte_index(RBuf, $\n),
209          LineSize = max(?MIN_READLINE_SIZE, RASz),
210          read_line_1(FRef, RBuf, SearchResult, LineSize)
211      catch
212          error:badarg -> {error, badarg}
213      end.
214
215  -spec read_line_1(FRef, RBuf, SearchResult, LineSize) -> Result when
216        FRef :: prim_file_ref(),
217        RBuf :: term(),
218        SearchResult :: not_found | {ok, non_neg_integer()},
219        LineSize :: non_neg_integer(),
220        Result :: eof | {ok, binary()} | {error, Reason :: atom()}.
221  read_line_1(FRef, RBuf, not_found, LineSize) ->
222      case read_nif(FRef, LineSize) of
223          {ok, Data} ->
224              prim_buffer:write(RBuf, [Data]),
225              SearchResult = prim_buffer:find_byte_index(RBuf, $\n),
226              read_line_1(FRef, RBuf, SearchResult, LineSize);
227          eof ->
228              case prim_buffer:size(RBuf) of
229                  Size when Size > 0 -> {ok, prim_buffer:read(RBuf, Size)};
230                  Size when Size =:= 0 -> eof
```

:prim_file

- **open/2**: `open(Name, Modes)`

  Opens a file and returns a file descriptor.

- **read_line/1**: `read_line(Fd)`

  Reads a line from a file descriptor until a newline (\n) character is encountered.

- **read/2**: `read(Fd, Size)`

  Reads a specified number of bytes from an open file descriptor.

```
{ok, Fd} = prim_file:open("file.txt", [read, binary]),
% Do something with Fd
prim_file:close(Fd).
```

:prim_file

- **open/2**: `open(Name, Modes)`

  Opens a file and returns a file descriptor.

- **read_line/1**: `read_line(Fd)`

  Reads a line from a file descriptor until a newline (\n) character is encountered.

- **read/2**: `read(Fd, Size)`

  Reads a specified number of bytes from an open file descriptor.

```
{ok, Fd} = prim_file:open("file.txt", [read, binary]),
case prim_file:read_line(Fd) of
    {ok, Line} → io:format("Read line: ~p~n", [Line]);
    eof → io:format("End of file reached~n")
end,
prim_file:close(Fd).
```

:prim_file

- **open/2**: `open(Name, Modes)`

  Opens a file and returns a file descriptor.

- **read_line/1**: `read_line(Fd)`

  Reads a line from a file descriptor until a newline (\n) character is encountered.

- **read/2**: `read(Fd, Size)`

  Reads a specified number of bytes from an open file descriptor.

```
{ok, Fd} = prim_file:open("file.txt", [read, binary]),
{ok, Data} = prim_file:read(Fd, 100),
prim_file:close(Fd).
```

:prim_file

- **open/2**: `open(Name, Modes)`

  Opens a file and returns a file descriptor.

- **read_line/1**: `read_line(Fd)`

  Reads a line from a file descriptor until a newline (\n) character is encountered.

- **read/2**: `read(Fd, Size)`

  Reads a specified number of bytes from an open file descriptor.

```
{ok, Fd} = prim_file:open("file.txt", [read, binary]),
{ok, Data} = prim_file:read(Fd, 100),
prim_file:close(Fd).
```

- Taking the middle road:

  - use read/2 to take in a fixed chunk of bytes

  - then use read_line/1 to finish till the next \n

Code Blame 194 lines (156 loc) · 5.22 KB Raw

```
1   defmodule OneBRC.MeasurementsProcessor.Version5 do
13    def process(count) do
16      fs = File.stream!(file_path)
17
18      ets_table = :ets.new(:station_stats, [:set, :public])
19
20      fs
21      |> Stream.chunk_every(10000)
22      |> Task.async_stream(
23        fn val -> Enum.map(val, &parse_row/1) end,
24        max_concurrency: System.schedulers_online(),
25        ordered: false,
26        timeout: :infinity
27      )
28      |> Stream.with_index()
29      |> Task.async_stream(
30        fn {{:ok, parsed_rows}, row_index} ->
31          interim_records =
32            Enum.reduce(parsed_rows, %{}, fn row, acc ->
33              process_row(row, acc)
34            end)
35
36          :ets.insert(ets_table, {row_index, interim_records})
37        end,
38        max_concurrency: System.schedulers_online(),
39        ordered: false,
40        timeout: :infinity
41      )
```

Code Blame 239 lines (190 loc) · 6.13 KB Raw

```elixir
107
108   defp read_and_process(file, ets_table, tasks) do
109     chunk_size = 1024 * 1024 * 1
110
111     data =
112       case :prim_file.read(file, chunk_size) do
113         :eof ->
114           nil
115
116         {:ok, data} ->
117           case :prim_file.read_line(file) do
118             {:ok, line} ->
119               <<data::binary, line::binary>>
120
121             :eof ->
122               data
123           end
124       end
125
126     if is_nil(data) do
127       tasks
128     else
129       task = Task.async(fn -> process_chunk(data, ets_table) end)
130
131       read_and_process(file, ets_table, [task | tasks])
132     end
133   end
134
```

Code Blame 239 lines (190 loc) · 6.13 KB Raw

```elixir
    require Logger

    def process(count) do
      t1 = System.monotonic_time(:millisecond)
      file_path = measurements_file(count)

      {:ok, file} = :prim_file.open(file_path, [:raw, :binary, :read])

      ets_table = :ets.new(:station_stats, [:duplicate_bag, :public])

      tasks = read_and_process(file, ets_table, [])

      Task.await_many(tasks, :infinity)
      # old way ->
      # fs
      # |> Stream.chunk_every(10000)
      # |> Task.async_stream(
      #   fn val -> Enum.map(val, &parse_row/1) end,
      #   max_concurrency: System.schedulers_online(),
      #   ordered: false,
      #   timeout: :infinity
      # )
      # |> Stream.with_index()
      # |> Task.async_stream(
      #   fn {{:ok, parsed_rows}, row_index} ->
      #     interim_records =
      #       Enum.reduce(parsed_rows, %{}, fn row, acc ->
      #         process_row(row, acc)
```

Code Blame 239 lines (190 loc) · 6.13 KB Raw

```
13    def process(count) do
51      result =
52        :ets.tab2list(ets_table)
53        |> Enum.reduce([], fn {_, m}, acc -> [acc | Enum.into(m, [])] end)
54        |> List.flatten()
55        |> Enum.reduce(%{}, fn {key, val}, acc ->
56          existing_record = Map.get(acc, key, nil)
57
58          new_record =
59            case existing_record do
60              nil ->
61                val
62
63              %{count: count, min: min, max: max, mean: mean} ->
64                min = if val.min < min, do: val.min, else: min
65                max = if val.max > max, do: val.max, else: max
66                new_c = count + val.count
67
68                mean = (mean * count + val.mean * val.count) / new_c
69
70                %{
71                  min: min,
72                  max: max,
73                  mean: mean,
74                  count: new_c
75                }
76            end
77
```

1BRC in Elixir: Version 6

10 Million Rows

0.6 sec

↘ 80.6%

100 Million Rows

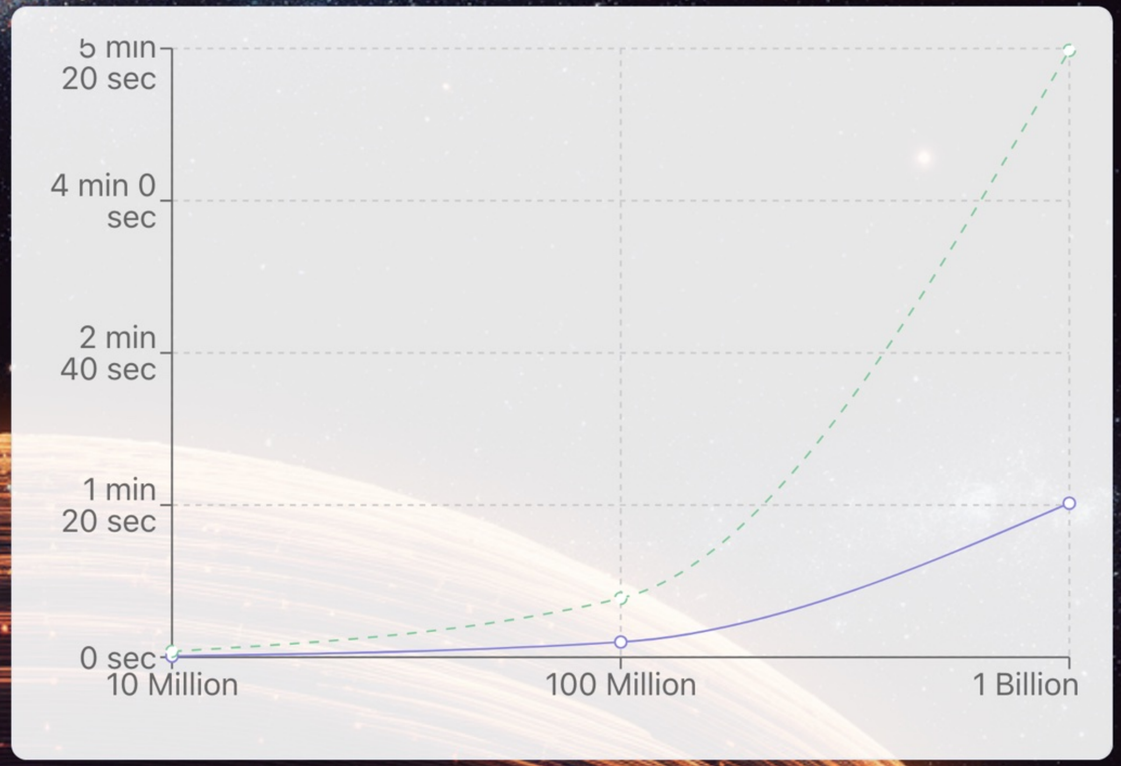
8 sec

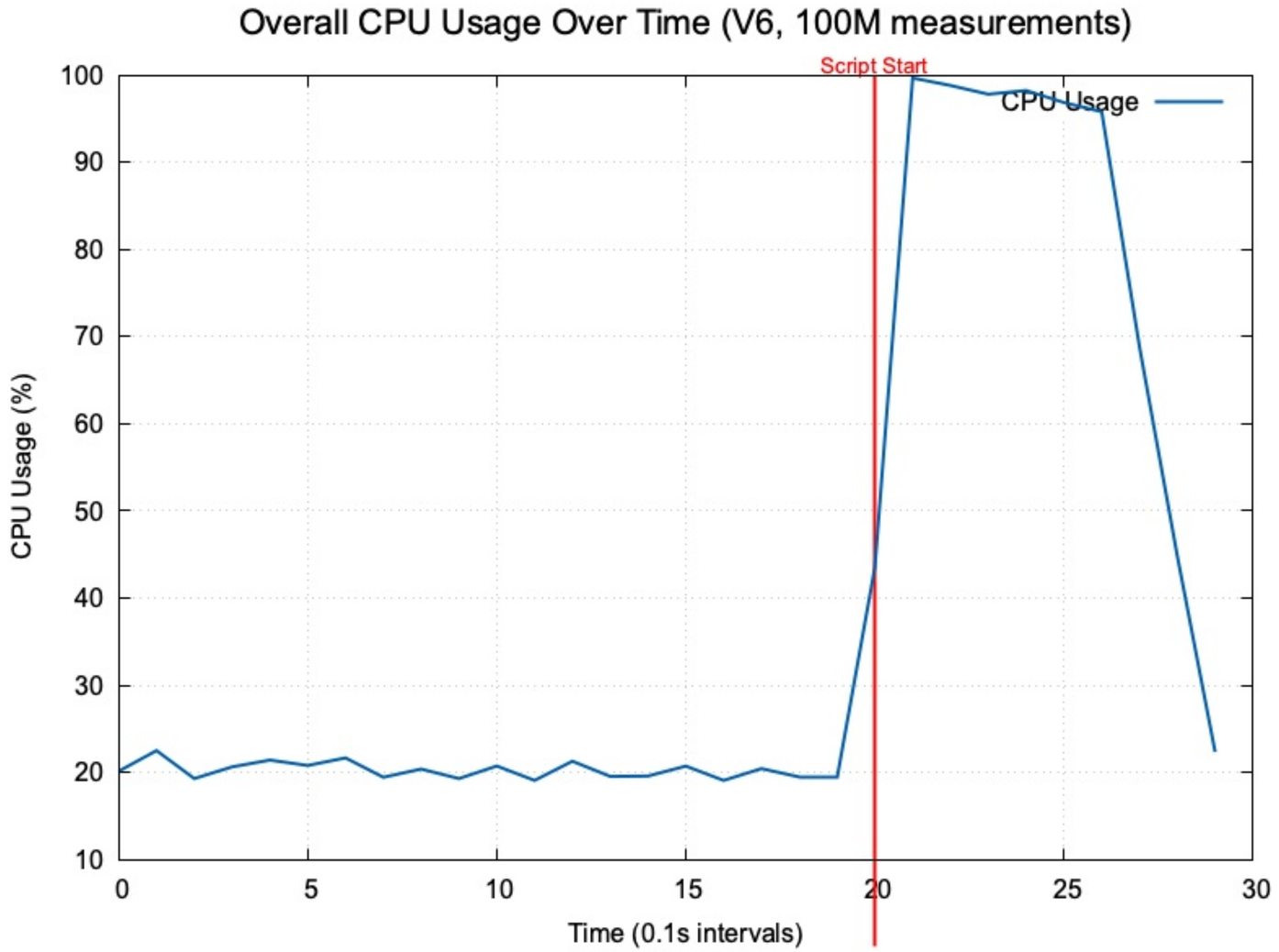
↘ 74.2%

1 Billion Rows

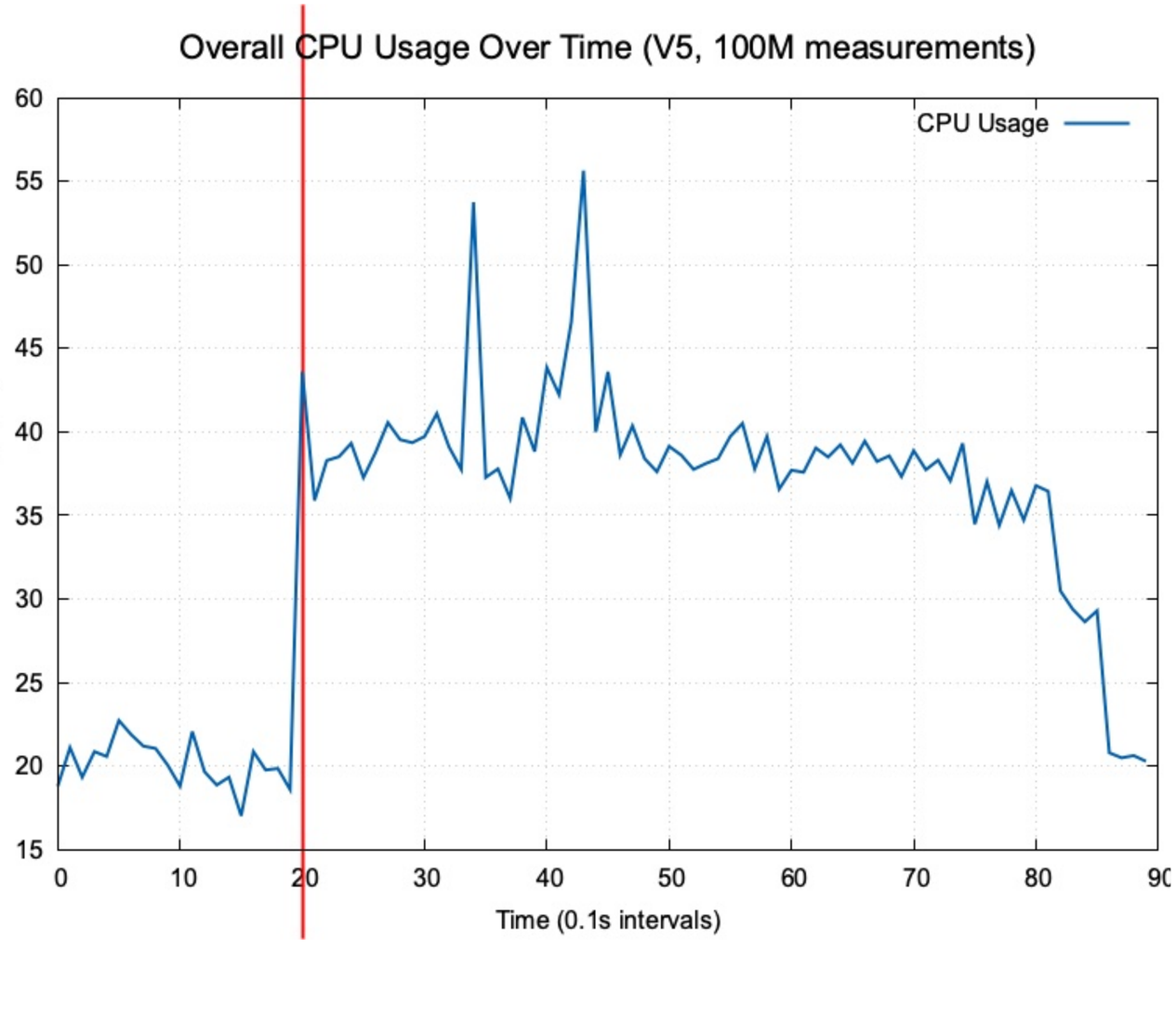
1 min 21 sec

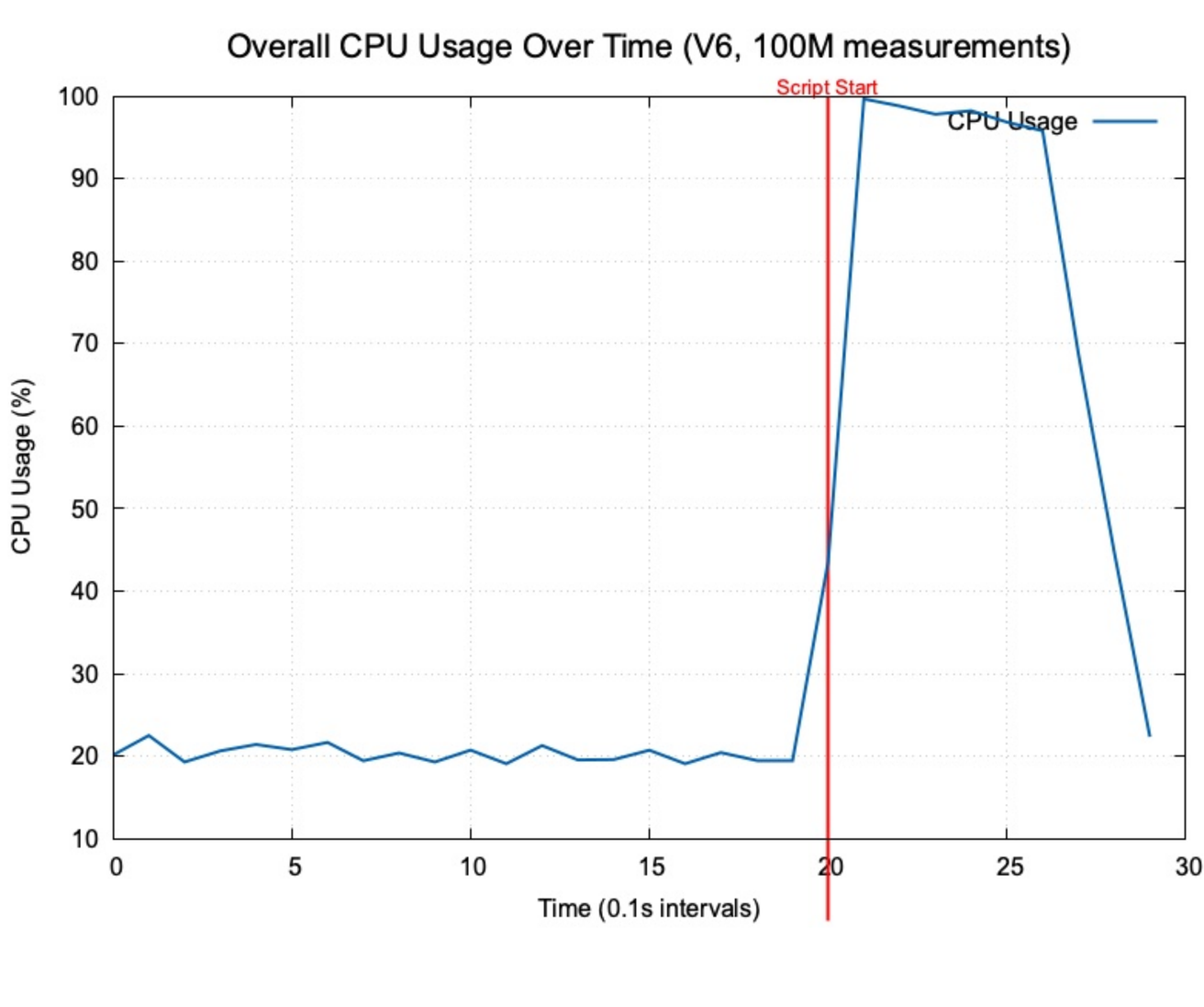
↘ 74.6%





CPU usage of version 6





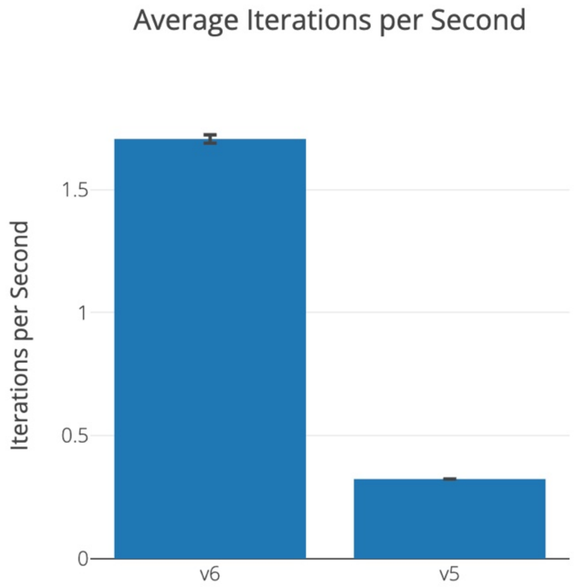
CPU usage of version 5 vs version 6

benchee report

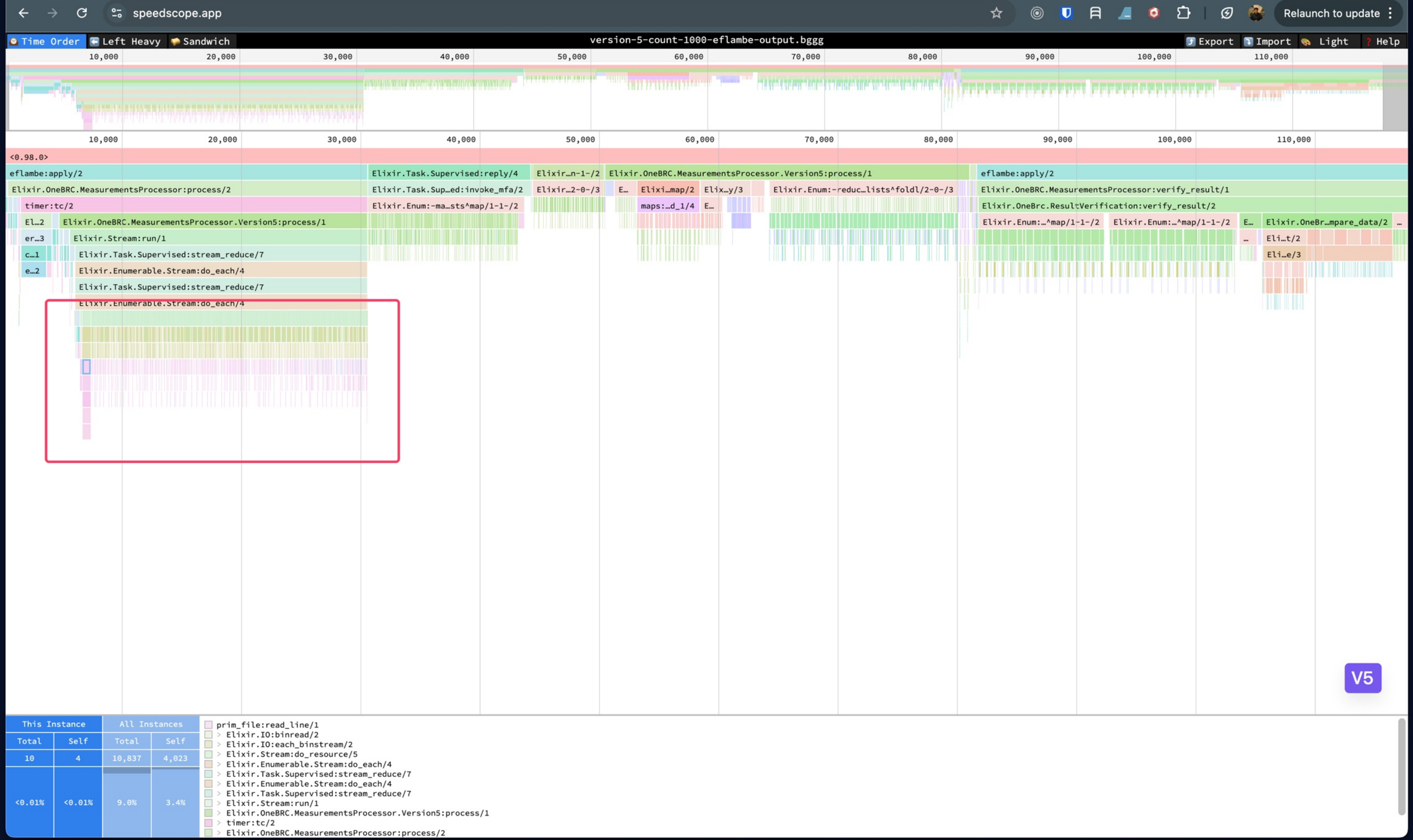
System info

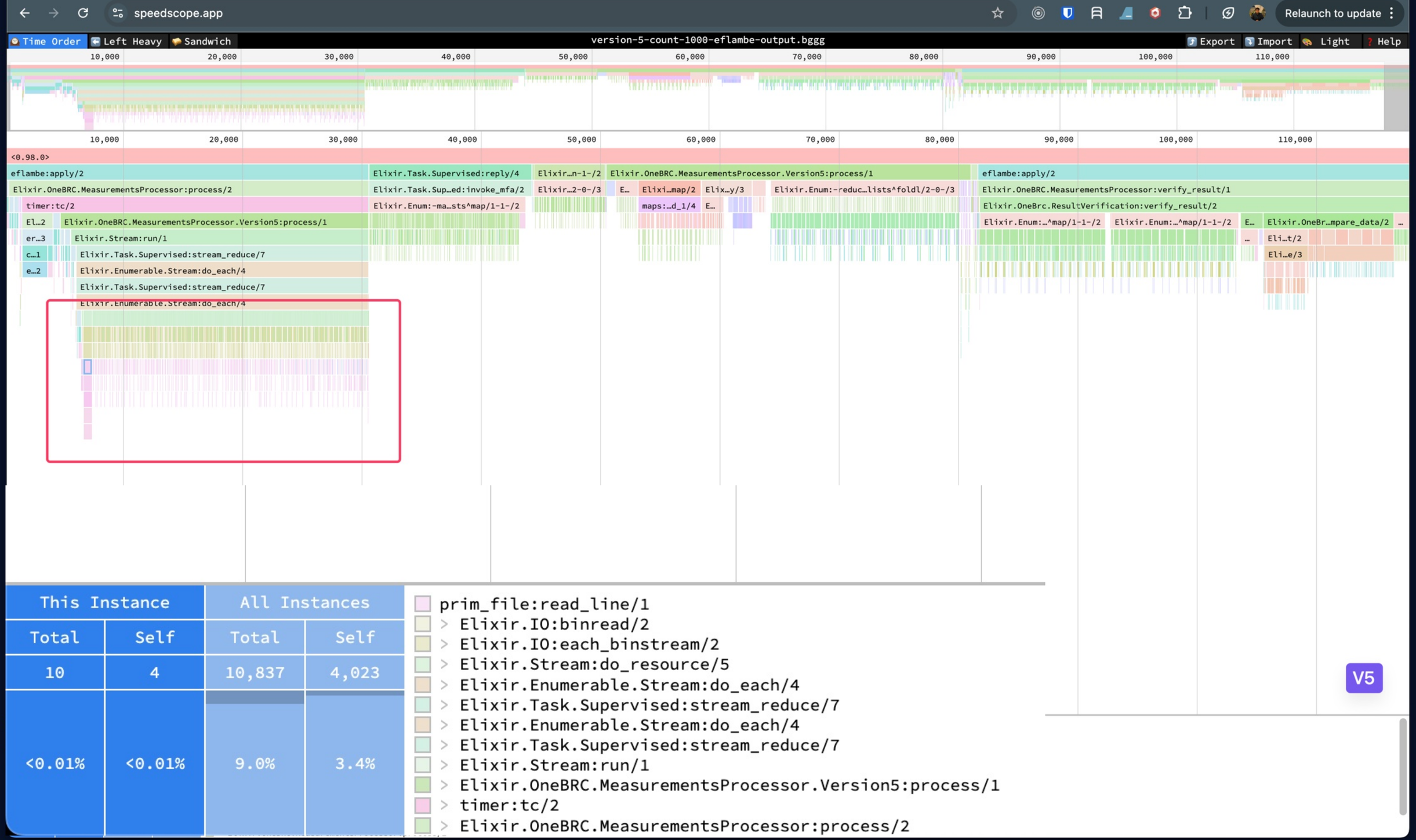
Run Time Comparison

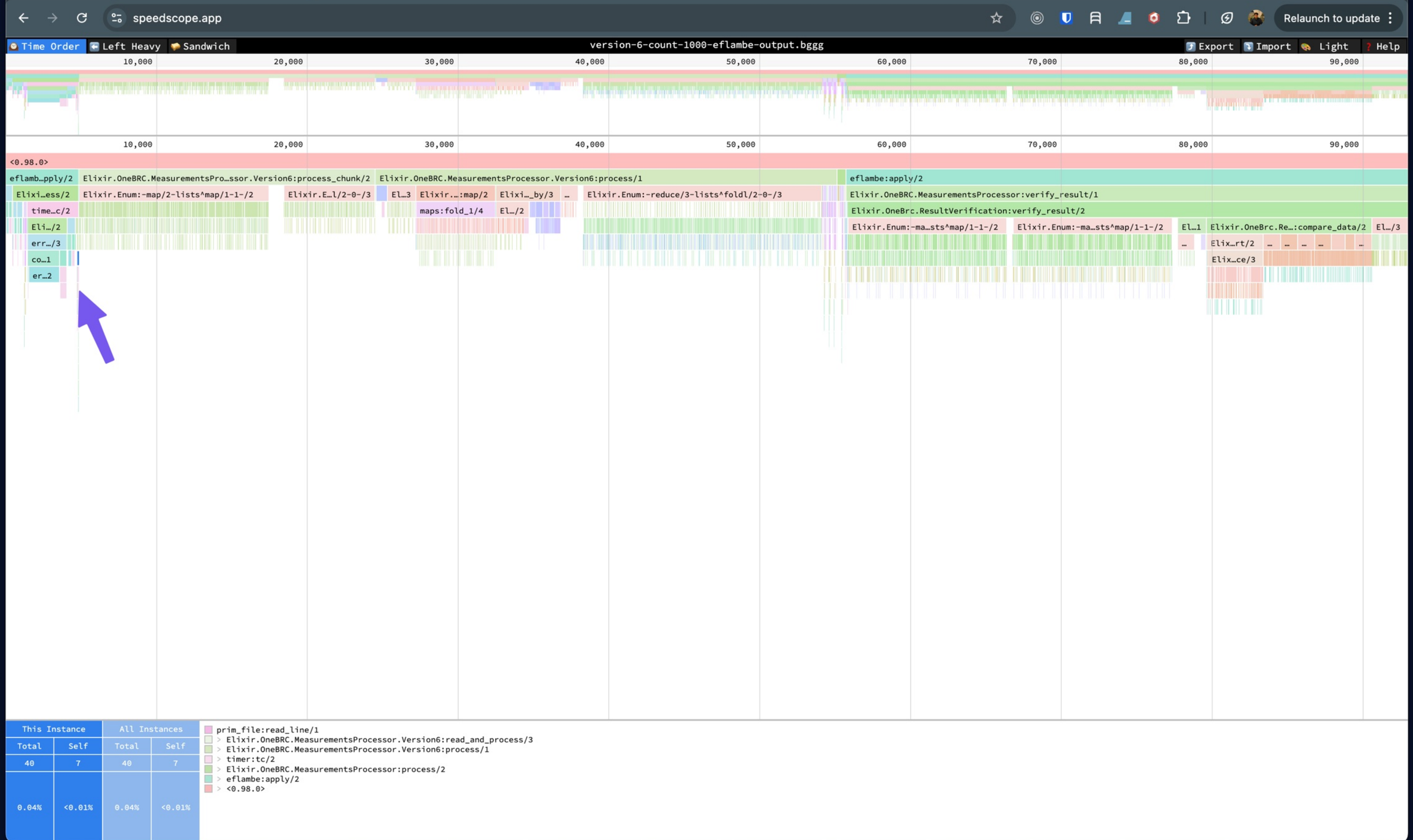
Name	Iterations per Second	Average	Deviation	Median	Mode	Minimum	Maximum	Sample size
v6	1.71	0.59 s	±1.00%	0.59 s	none	0.58 s	0.60 s	9
v5	0.32	3.08 s	±0.01%	3.08 s	none	3.08 s	3.08 s	2

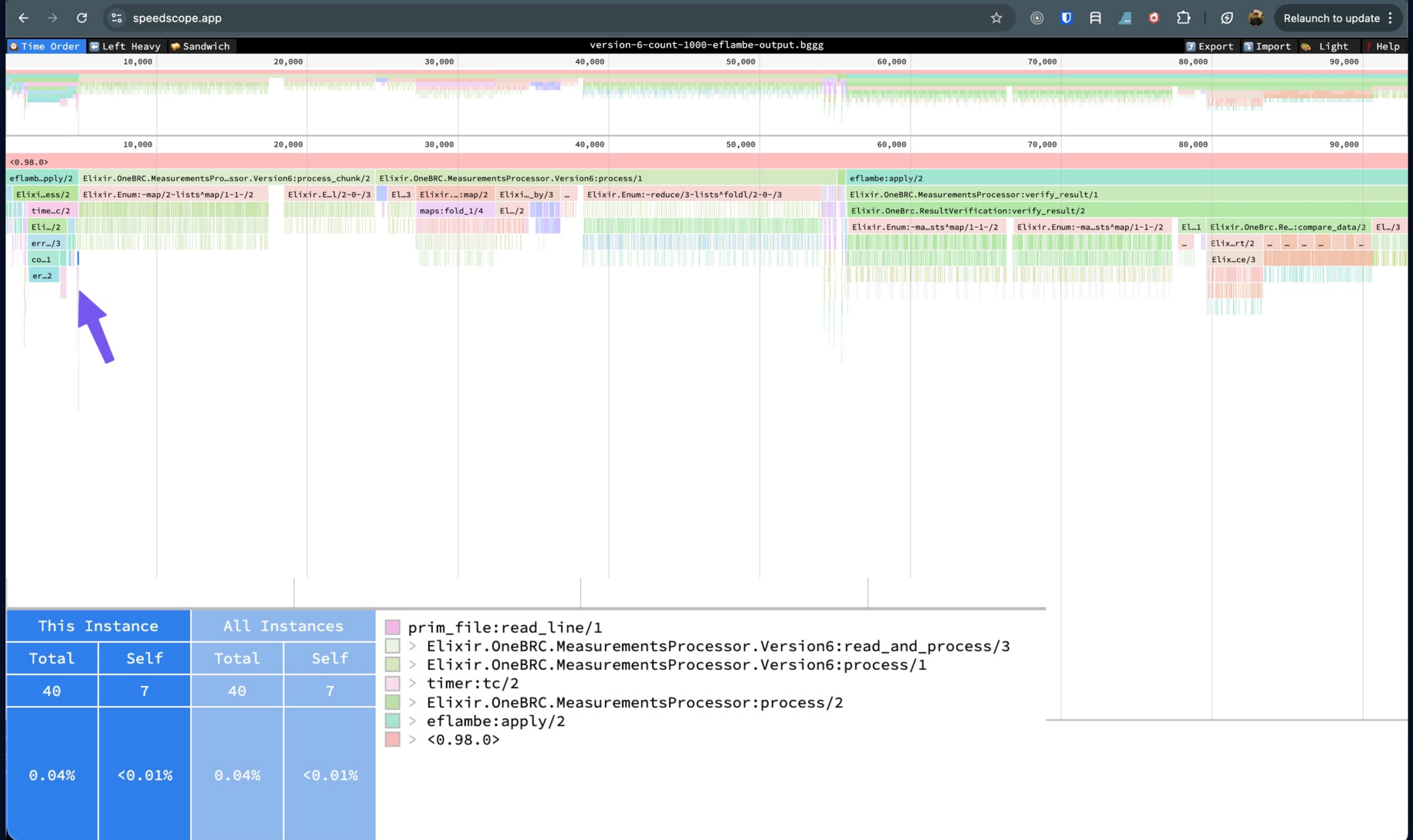


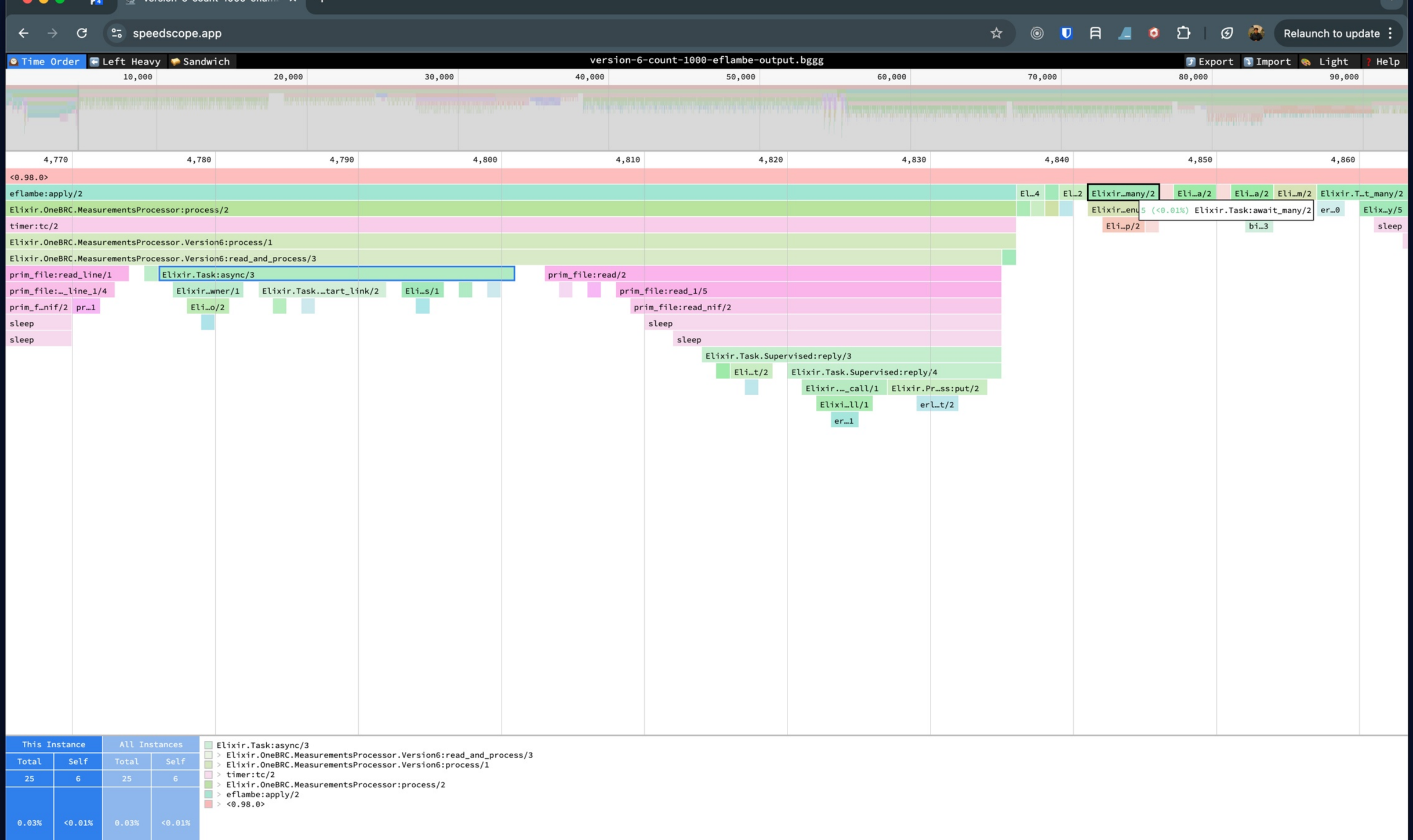
Benchee comparison of version 5 vs version 6


speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
<0.98.0>
eflambe:apply/2
Elixir.Task.Supervised:reply/4
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.OneBRC.MeasurementsProcessor:process/2
Elixir.Task.Supervised:invoke_mfa/2
Elixir.Enum:-reduc…lists^foldl/2-0-/3
Elixir.OneBRC.MeasurementsProcessor:verify_result/1
timer:tc/2
Elixir.Enum:-ma…sts^map/1-1-/2
Elixir.OneBrc.ResultVerification:verify_result/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Enum:…^map/1-1-/2
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
V5
This Instance
All Instances
Total
Self
10
4
10,837
4,023
<0.01%
<0.01%
9.0%
3.4%
prim_file:read_line/1
Elixir.IO:binread/2
Elixir.IO:each_binstream/2
Elixir.Stream:do_resource/5
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Stream:run/1
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor:process/2


speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-5-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000
110,000
<0.98.0>
eflambe:apply/2
Elixir.Task.Supervised:reply/4
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.OneBRC.MeasurementsProcessor:process/2
Elixir.Task.Supervised:invoke_mfa/2
timer:tc/2
Elixir.Enum:-ma…sts^map/1-1-/2
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
Elixir.Stream:run/1
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.OneBRC.MeasurementsProcessor:verify_result/1
Elixir.OneBrc.ResultVerification:verify_result/2
Elixir.Enum:…^map/1-1-/2
Elixir.Enum:…^map/1-1-/2
Elixir.OneBr…mpare_data/2
This Instance
All Instances
Total
Self
Total
Self
10
4
10,837
4,023
<0.01%
<0.01%
9.0%
3.4%
prim_file:read_line/1
Elixir.IO:binread/2
Elixir.IO:each_binstream/2
Elixir.Stream:do_resource/5
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Enumerable.Stream:do_each/4
Elixir.Task.Supervised:stream_reduce/7
Elixir.Stream:run/1
Elixir.OneBRC.MeasurementsProcessor.Version5:process/1
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor:process/2
V5


speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-6-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
<0.98.0>
eflamb…pply/2
Elixir.OneBRC.MeasurementsPro…ssor.Version6:process_chunk/2
Elixir.OneBRC.MeasurementsProcessor.Version6:process/1
eflambe:apply/2
Elixi…ess/2
Elixir.Enum:-map/2-lists^map/1-1-/2
Elixir.E…l/2-0-/3
El…3
Elixir.…:map/2
Elixi…_by/3
Elixir.Enum:-reduce/3-lists^foldl/2-0-/3
Elixir.OneBRC.MeasurementsProcessor:verify_result/1
time…c/2
maps:fold_1/4
El…/2
Elixir.OneBrc.ResultVerification:verify_result/2
Eli…/2
Elixir.Enum:-ma…sts^map/1-1-/2
Elixir.Enum:-ma…sts^map/1-1-/2
El…1
Elixir.OneBrc.Re…:compare_data/2
El…/3
err…/3
Elix…rt/2
co…1
Elix…ce/3
er…2
This Instance
All Instances
Total
Self
Total
Self
40
7
40
7
0.04%
<0.01%
0.04%
<0.01%
prim_file:read_line/1
Elixir.OneBRC.MeasurementsProcessor.Version6:read_and_process/3
Elixir.OneBRC.MeasurementsProcessor.Version6:process/1
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor:process/2
eflambe:apply/2
<0.98.0>

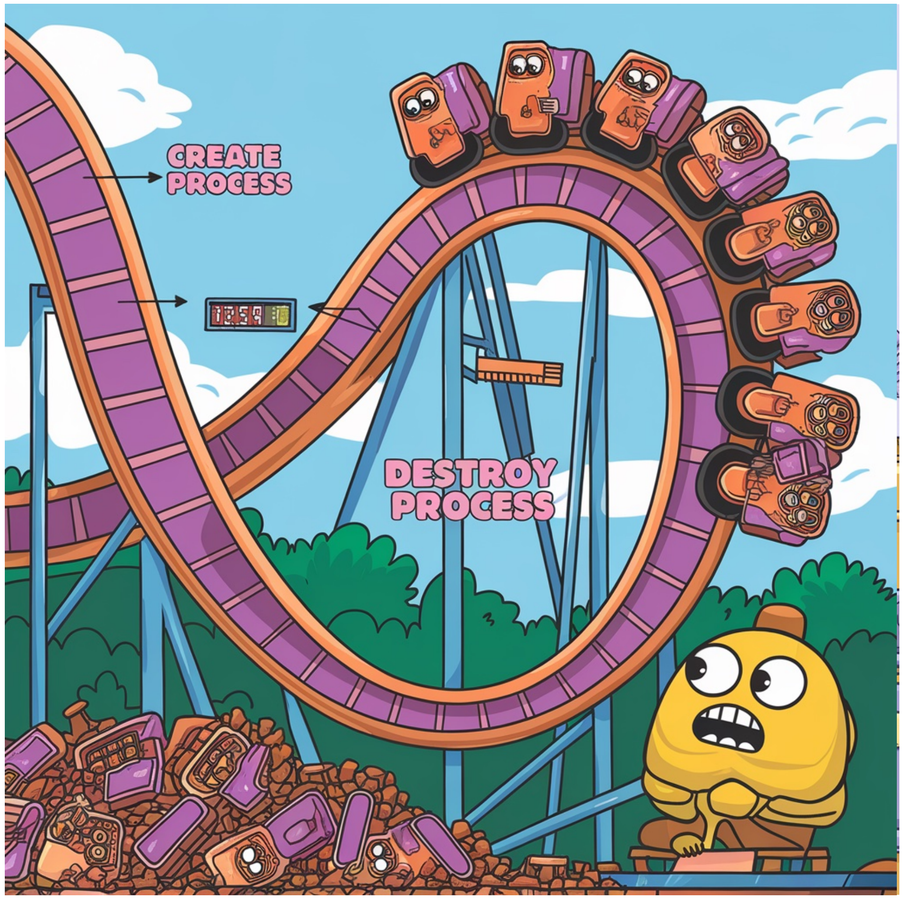

speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-6-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
<0.98.0>
eflamb…pply/2
Elixir.OneBRC.MeasurementsPro…ssor.Version6:process_chunk/2
Elixir.OneBRC.MeasurementsProcessor.Version6:process/1
eflambe:apply/2
Elixi…ess/2
Elixir.Enum:-map/2-lists^map/1-1-/2
Elixir.E…l/2-0-/3
El…3
Elixir.…:map/2
Elixi…_by/3
Elixir.Enum:-reduce/3-lists^foldl/2-0-/3
Elixir.OneBRC.MeasurementsProcessor:verify_result/1
time…c/2
maps:fold_1/4
El…/2
Elixir.OneBrc.ResultVerification:verify_result/2
Eli…/2
Elixir.Enum:-ma…sts^map/1-1-/2
Elixir.Enum:-ma…sts^map/1-1-/2
El…1
Elixir.OneBrc.Re…:compare_data/2
El…/3
err…/3
Elix…rt/2
co…1
Elix…ce/3
er…2
This Instance
All Instances
Total
Self
Total
Self
40
7
40
7
0.04%
<0.01%
0.04%
<0.01%
prim_file:read_line/1
> Elixir.OneBRC.MeasurementsProcessor.Version6:read_and_process/3
> Elixir.OneBRC.MeasurementsProcessor.Version6:process/1
> timer:tc/2
> Elixir.OneBRC.MeasurementsProcessor:process/2
> eflambe:apply/2
> <0.98.0>


speedscope.app
Relaunch to update
Time Order
Left Heavy
Sandwich
version-6-count-1000-eflambe-output.bggg
Export
Import
Light
Help
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
4,770
4,780
4,790
4,800
4,810
4,820
4,830
4,840
4,850
4,860
<0.98.0>
eflambe:apply/2
Elixir.OneBRC.MeasurementsProcessor:process/2
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor.Version6:process/1
Elixir.OneBRC.MeasurementsProcessor.Version6:read_and_process/3
prim_file:read_line/1
Elixir.Task:async/3
prim_file:read/2
prim_file:read_1/5
prim_file:read_nif/2
sleep
Elixir.Task.Supervised:reply/3
Elixir.Task.Supervised:reply/4
Elixir.Task:await_many/2
This Instance
All Instances
Total
Self
25
6
0.03%
<0.01%
Elixir.Task:async/3
Elixir.OneBRC.MeasurementsProcessor.Version6:read_and_process/3
Elixir.OneBRC.MeasurementsProcessor.Version6:process/1
timer:tc/2
Elixir.OneBRC.MeasurementsProcessor:process/2
eflambe:apply/2
<0.98.0>

Optimising Task overhead

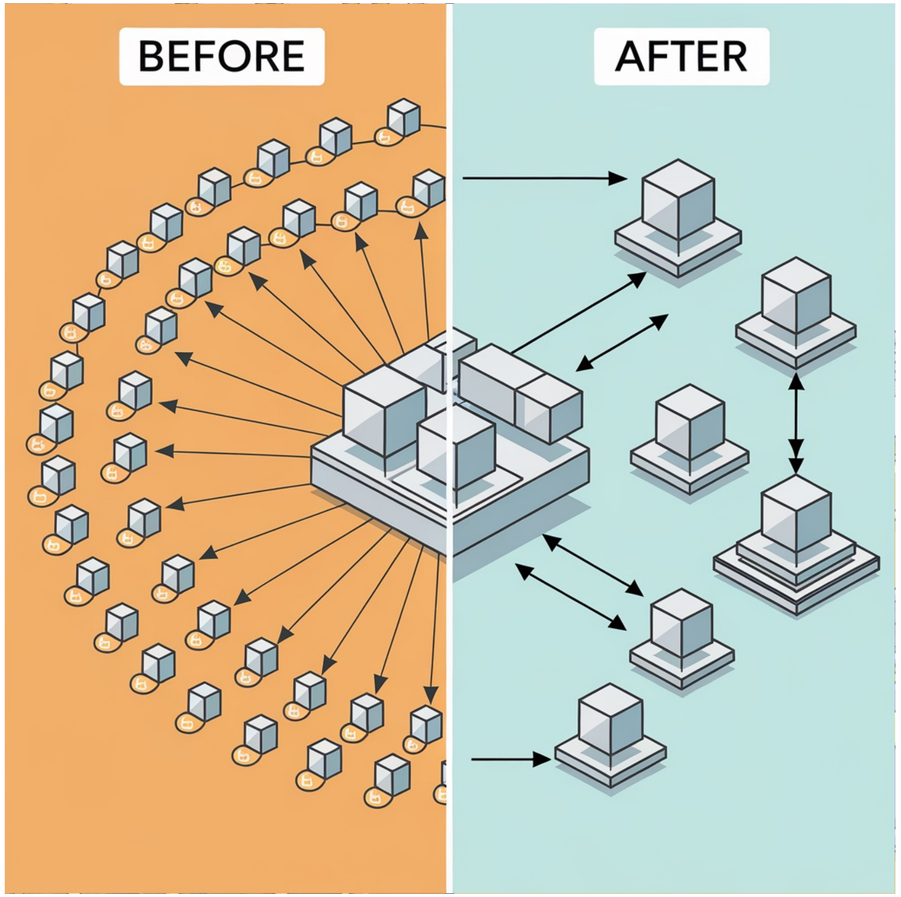
- Task.async for each chunk and then Task.await_many helped 🚀 CPU utilisation

- Task.async creates a process, runs our code, destroys the process.



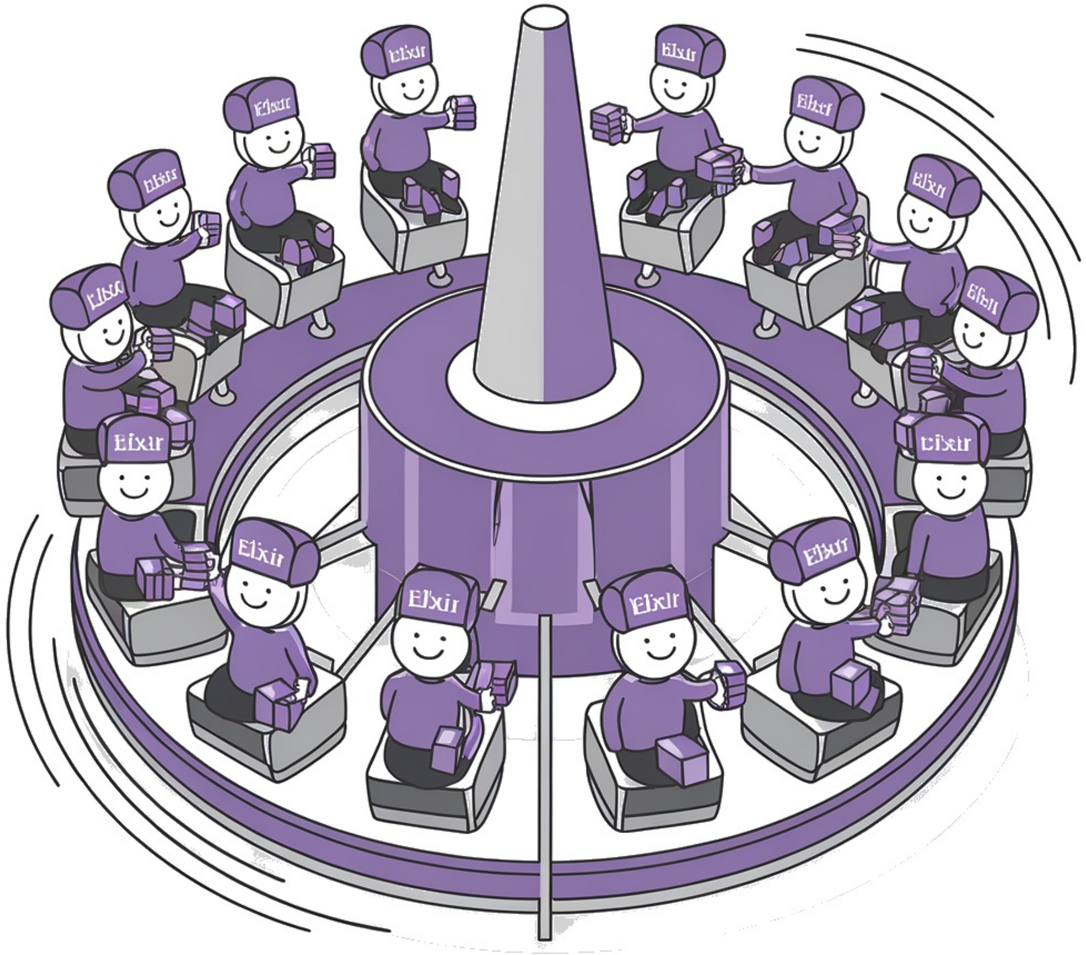
Optimising Task overhead

- Task.async for each chunk and then Task.await_many helped 🚀 CPU utilisation
- Task.async creates a process, runs our code, destroys the process.
- Process creation overhead can be avoided.
- We can use a fixed worker pool for handling jobs.
- Reuse workers, minimise overhead.



Reusable Worker pool of processes

- Parent process spawns worker processes
- Each worker process sends give_work message to it’s parent
- Parent process, when received a give_work message from worker, sends a chunk for the worker to process
- Worker process processes the chunk, writes result to it’s dict, and sends give_work message to parent again
- Repeat
- At the end, parent collects intermediate results from each worker.

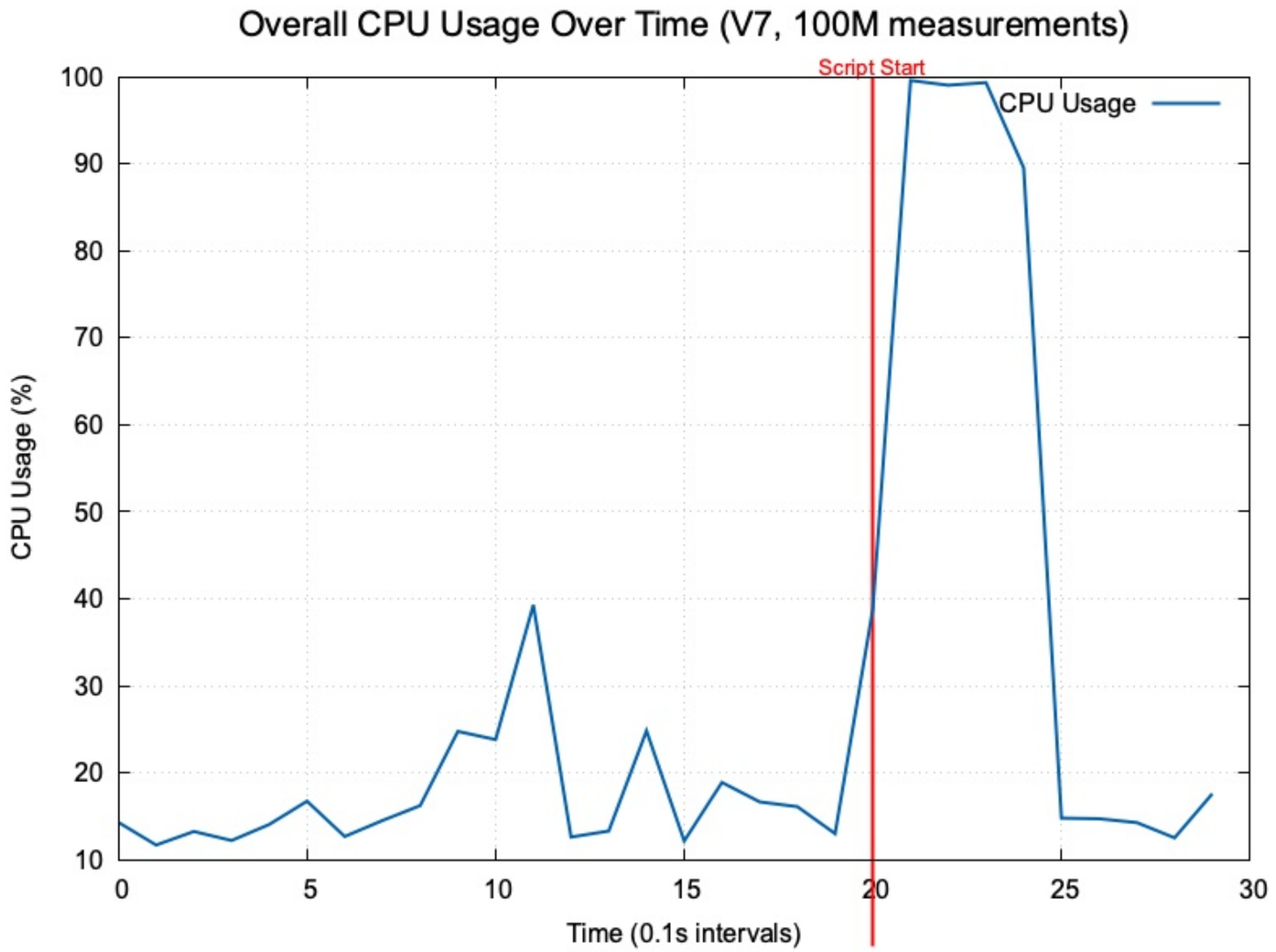


```elixir
defmodule Worker do
  def run(parent_pid) do
    send(parent_pid, {:give_work, self()})

    receive do
      {:do_work, chunk} ->
        process_chunk(chunk)
        run(parent_pid)

      :result ->
        send(parent_pid, {:result, :erlang.get()})
        # die
    end
  end

  defp process_chunk(bin) do
    :binary.split(bin, "\n", [:global])
    |> Enum.map(&parse_row/1)
    |> Enum.map(fn row ->
      process_row(row)
    end)
...
```

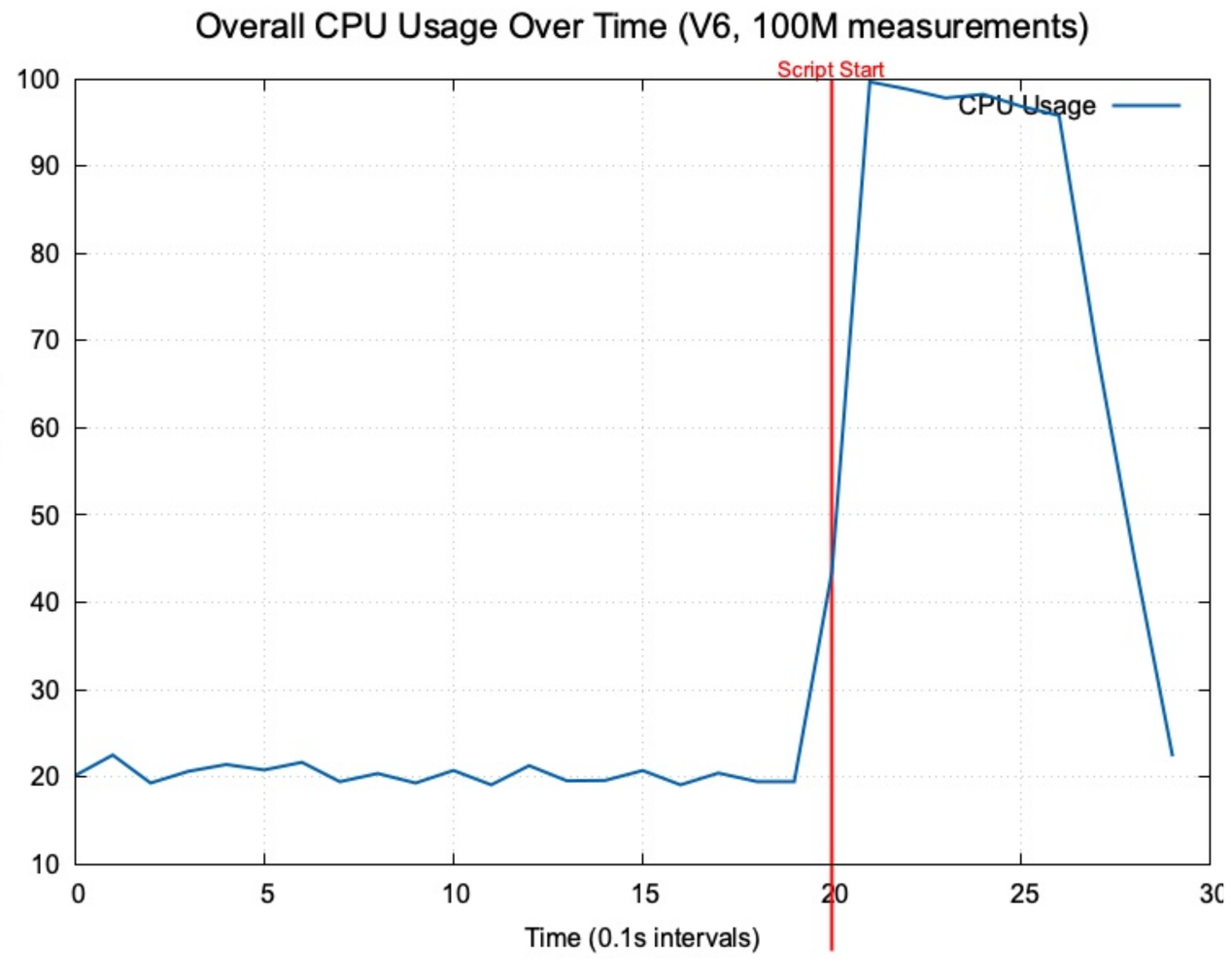
```
worker_count = System.schedulers_online() * 2
# boot up workers
parent = self()

wpids =
    Enum.map(1..worker_count, fn _ ->
      spawn_link(fn ->
          Worker.run(parent)
      end)
    end)

{:ok, file} = :prim_file.open(file_path, [:raw, :binary, :read])

:ok = read_and_process(file)

…
```

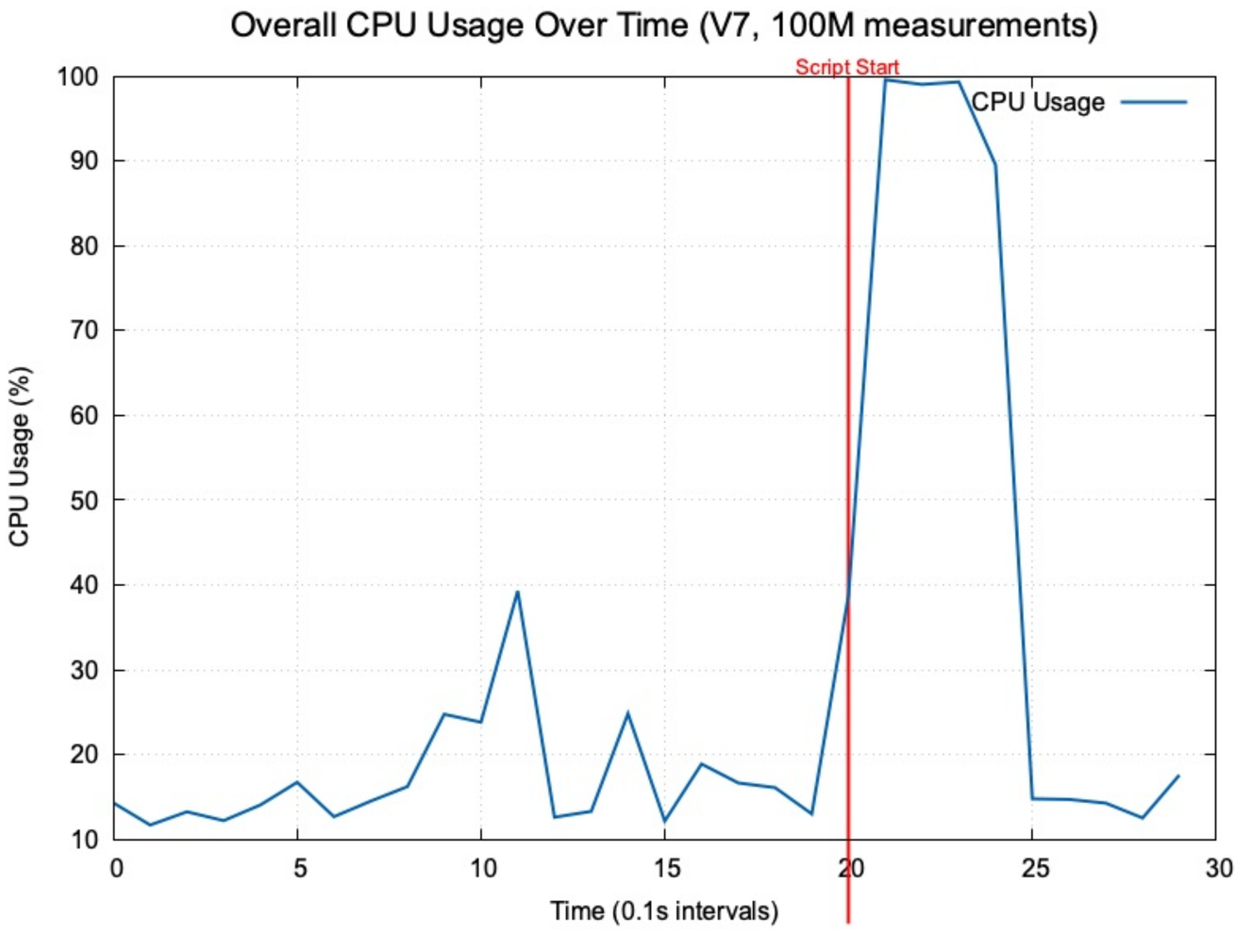
```
defp read_and_process(file) do
    chunk_size = 1024 * 1024 * 1
    data =
      case :prim_file.read(file, chunk_size) do
        :eof ->
          nil

        {:ok, data} ->
          case :prim_file.read_line(file) do
            {:ok, line} ->
              <<data::binary, line::binary>>

            :eof ->
              data
          end
      end

    if !is_nil(data) do
      receive do
        {:give_work, worker_pid} ->
          send(worker_pid, {:do_work, data})
      end
      read_and_process(file)
    else
      :ok
    end
  end
```

```elixir
# wait for all workers to finish
Enum.map(1..worker_count, fn _ ->
    receive do
      {:give_work, _worker_pid} ->
          :ok
    end
end)

results =
    wpids
    |> Enum.map(fn wpid ->
      send(wpid, :result)
    end)
    |> Enum.map(fn _ ->
      receive do
          {:result, result} ->
          result
      end
    end)

:prim_file.close(file)
```



CPU usage of version 7





CPU usage of version 6 vs version 7

1BRC in Elixir: Version 7

10 Million Rows

0.3 sec

↘ 50.0%

100 Million Rows

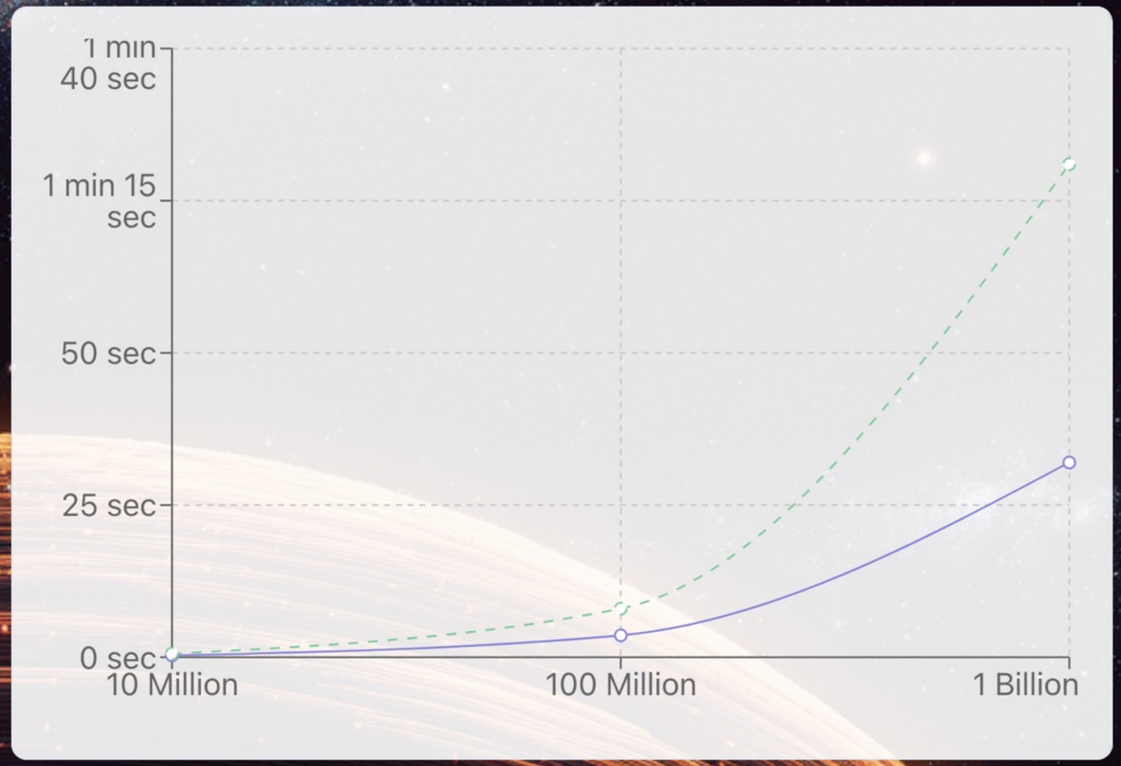
3.6 sec

↘ 55.0%

1 Million Rows

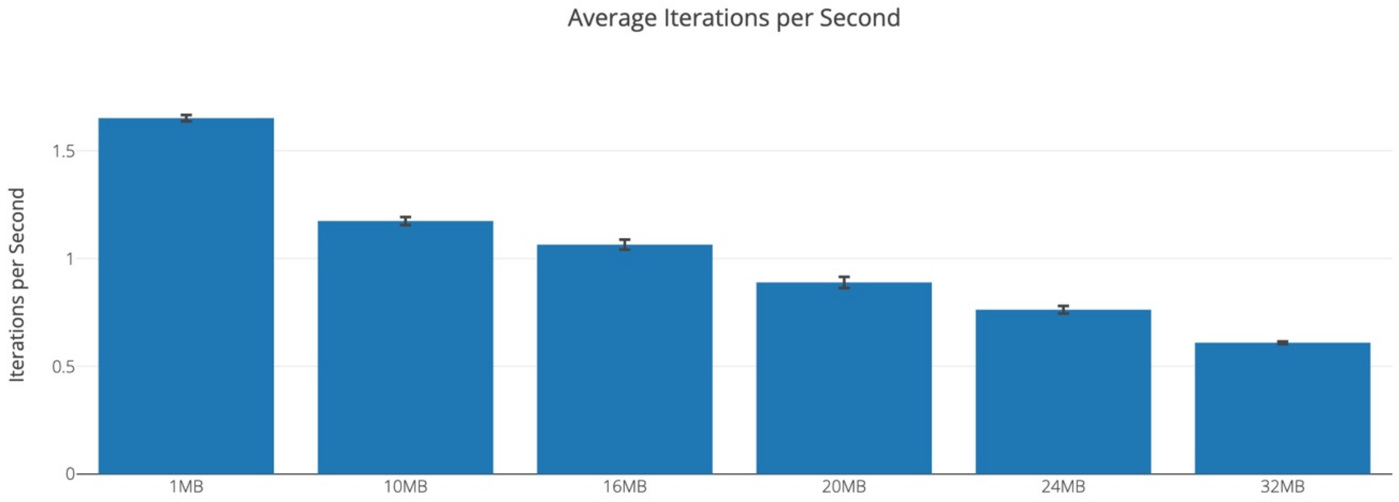
32 sec

↘ 60.5%



Run Time Comparison

Name	Iterations per Second	Average	Deviation	Median	Mode	Minimum	Maximum	Sample size
1MB	1.65	0.61 s	±0.88%	0.61 s	none	0.60 s	0.61 s	9
10MB	1.17	0.85 s	±1.58%	0.85 s	none	0.83 s	0.87 s	6
16MB	1.06	0.94 s	±2.16%	0.94 s	none	0.91 s	0.96 s	6
20MB	0.89	1.12 s	±2.91%	1.13 s	none	1.09 s	1.17 s	5
24MB	0.76	1.31 s	±2.24%	1.32 s	none	1.27 s	1.34 s	4
32MB	0.61	1.64 s	±0.98%	1.63 s	none	1.63 s	1.66 s	4



Run Time Boxplot

Using Benchee to find the optimal chunk size

```elixir
defmodule Worker do
  def run(parent_pid) do
    send(parent_pid, {:give_work, self()})

    receive do
      {:do_work, chunk} ->
        process_chunk(chunk)
        run(parent_pid)

      :result ->
        send(parent_pid, {:result, :erlang.get()})
        # die
    end
  end

  defp process_chunk(bin) do
    :binary.split(bin, "\n", [:global])
    |> Enum.map(&parse_row/1)
    |> Enum.map(fn row ->
      process_row(row)
    end)
...
```

```elixir
# ex: -4.5
defp parse_temp(<<?-, d1, ?., d2, "\n", rest::binary>>, key) do
    temp = -(char_to_num(d1) * 10 + char_to_num(d2))
    process_row(key, temp)
    process_chunk_lines(rest)
end

# ex: 4.5
defp parse_temp(<<d1, ?., d2, "\n", rest::binary>>, key) do
    temp = char_to_num(d1) * 10 + char_to_num(d2)
    process_row(key, temp)
    process_chunk_lines(rest)
end

# ex: -45.3
defp parse_temp(<<?-, d1, d2, ?., d3, "\n", rest::binary>>, key) do
    temp = -(char_to_num(d1) * 100 + char_to_num(d2) * 10 + char_to_num(d3))
    process_row(key, temp)
    process_chunk_lines(rest)
end

# ex: 45.3
defp parse_temp(<<d1, d2, ?., d3, "\n", rest::binary>>, key) do
    temp = char_to_num(d1) * 100 + char_to_num(d2) * 10 + char_to_num(d3)
    process_row(key, temp)
    process_chunk_lines(rest)
end
```

```elixir
defp process_chunk_lines(<<>>) do
  :ok
end

defp process_chunk_lines(bin) do
  parse_weather_station(bin, bin, 0)
end

# ↓ found ";" - split and parse
defp parse_weather_station(bin, <<";", _rest::binary>>, count) do
  <<key::binary-size(count), ";", temp_bin::binary>> = bin
  parse_temp(temp_bin, key)
end

# → no ";" yet - move to next char
defp parse_weather_station(bin, <<_c, rest::binary>>, count) do
  parse_weather_station(bin, rest, count + 1)
end
```

```
iex(1)> h <<>>

defmacro <<args>>

...

For binaries, the default is the size of the binary.

    iex> <<name::binary-size(5), " the ", species::binary>> = <<"Frank the Walrus">>
    "Frank the Walrus"
    iex> {name, species}
    {"Frank", "Walrus"}

The size can be a variable or any valid guard expression:

    iex> name_size = 5
    iex> <<name::binary-size(^name_size), " the ", species::binary>> = <<"Frank the Walrus">>
    iex> {name, species}
    {"Frank", “Walrus"}

...
```

Output of `h <<>>` in iex

1BRC in Elixir: Version 8

10 Million Rows

0.25 sec

↘ 16.7%

100 Million Rows

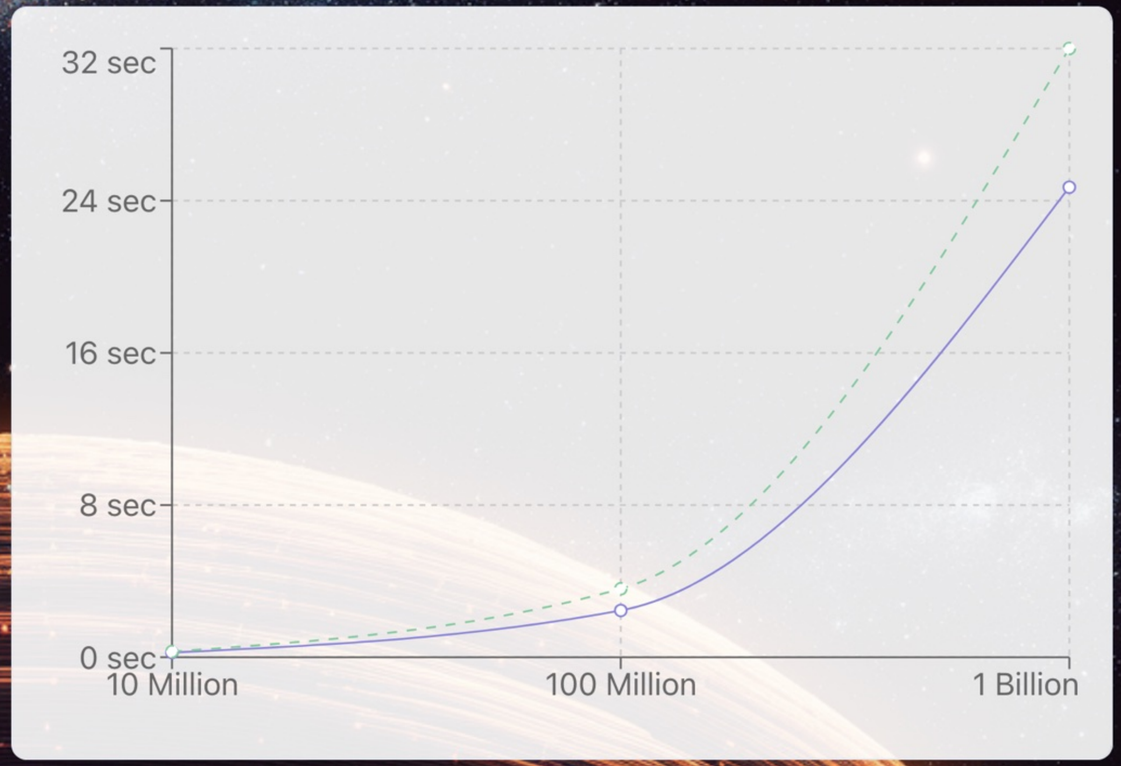
2.46 sec

↘ 31.7%

1 Billion Rows

24.71 sec

↘ 22.8%



1BRC in Elixir: From Version 1 to Version 8

10 Million Rows

0.25 sec

↘ 97.8%

100 Million Rows

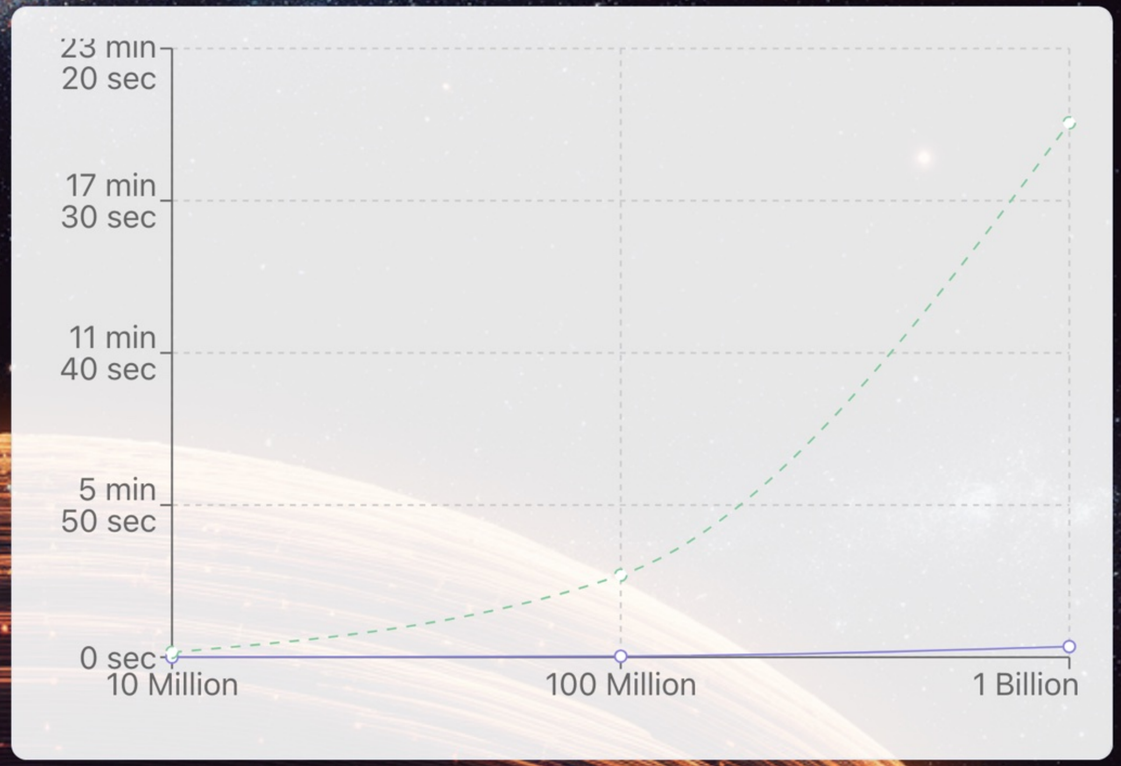
2.46 sec

↘ 98.7%

1 Billion Rows

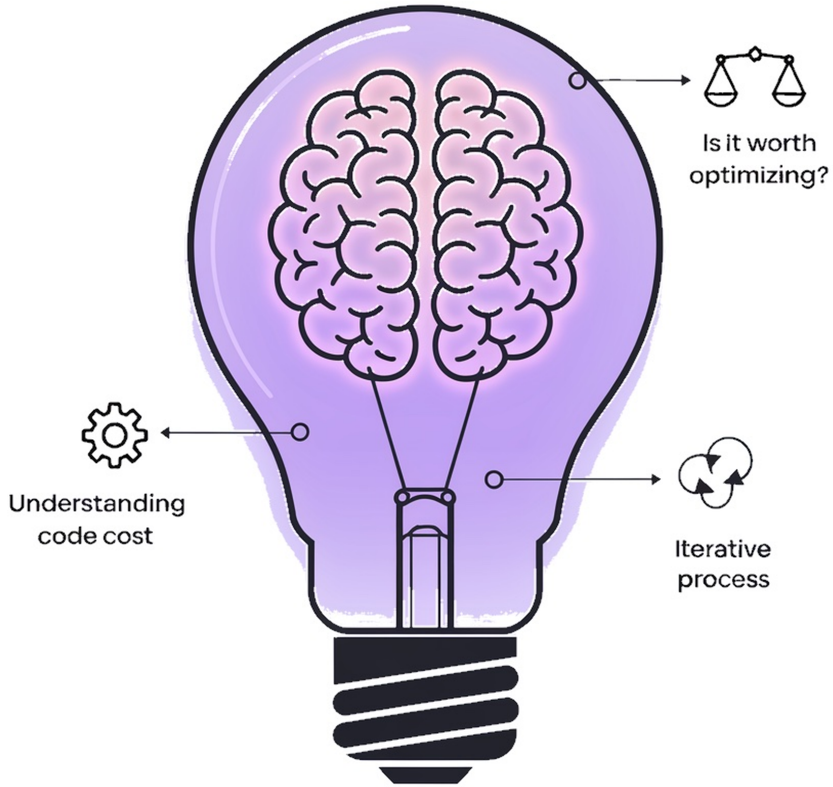
24.71 sec

↘ 98.0%



Some takeaways

- Developing a performance mindset.
- Understanding the true cost of code and it’s performance implications.
- Ask: “is this code worth optimising”?
- Optimisation is a iterative process.
- Understanding trade-offs
- Using profiling tools effectively



Thanks!

rajrajhans/
elixir_1brc

The One Billion Rows Challenge in Elixir

@_rajrajhans

Thanks!

rajrajhans/
elixir_1brc

The One Billion Rows Challenge in Elixir

@_rajrajhans