Parallel Programming with Ractors

Koichi Sasada

STORES

https://www.st.inc/

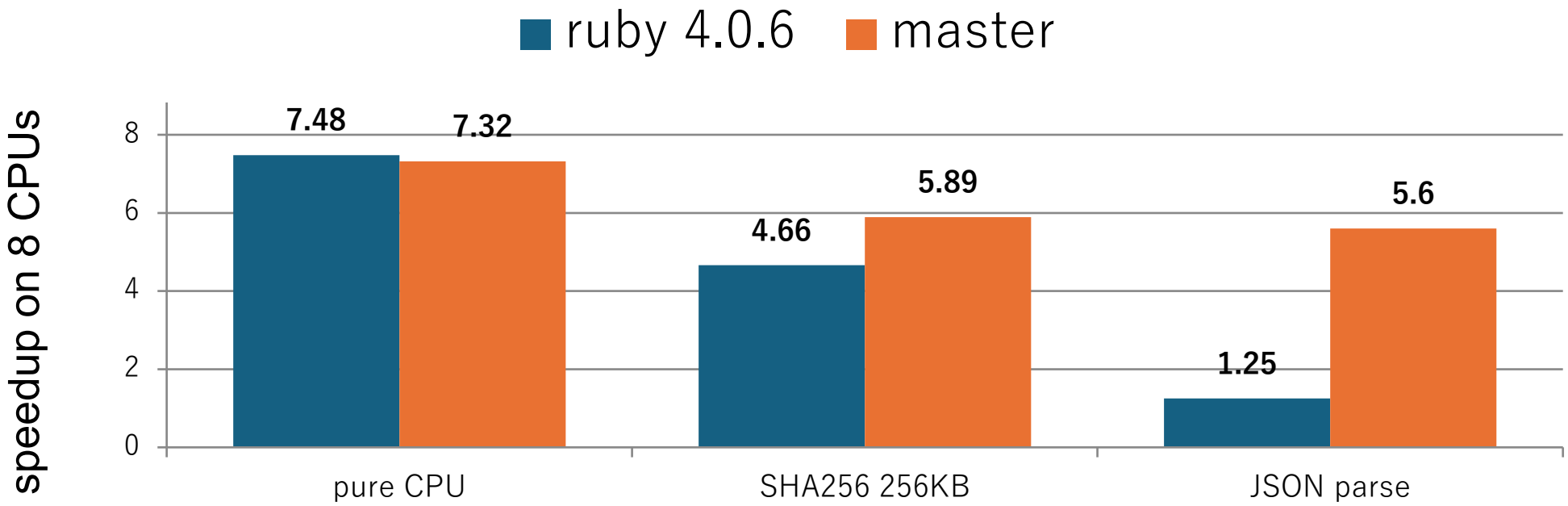
EURUKO 2026 (2026/Sep/17-18)

Today's topic

No internal details

Ask for details at the RHC workshop

1. Ractor on Ruby HEAD is very good now — Let me brag a little.



2. Why do we need Ractor on Ruby?

Koichi Sasada

5th EuRuKo talk

- Ruby interpreter developer at STORES, Inc. (since 2023), working with @mametter
  - YARV (Ruby 1.9~)
  - Generational/Incremental GC (Ruby 2.1~)
  - Ractor (Ruby 3.0~)
  - debug.gem (Ruby 3.1~)
  - M:N Thread scheduler (Ruby 3.3~)
  - …
- Director of the Ruby Association (since 2012)

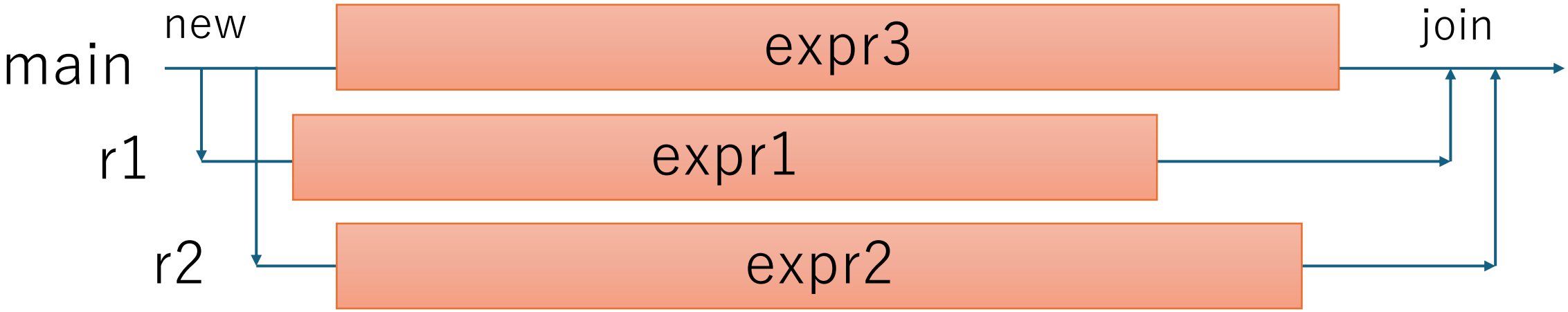
What is the Ractor? (1)

Run expressions in parallel

```
r1 = Ractor.new{ expr1 }
r2 = Ractor.new{ expr2 }
expr3; r1.join; r2.join
```

✅ Like threads

👍 Parallel without GVL

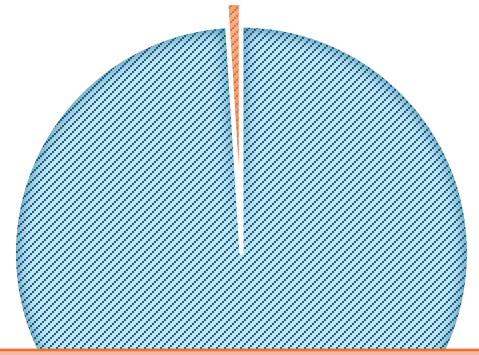


expr1, 2, 3 runs in parallel (simultaneously)

What is the Ractor? (2)

Isolate shared states

```
ARY = [1, 2, 3]
Ractor.new{ ARY << 4 }
#=> Ractor::IsolationError
```



Most objects are unshareable

- Sharing mutable objects is restricted (unshareable objects)
- Shareable objects
  - Classes/Modules
  - Frozen shareable objects (frozen and only refer to shareable objects)
  - Special objects (Ractor, Ractor::Port, Ractor::TVar, …)

What is the Ractor? (3)

Communicate with Ports

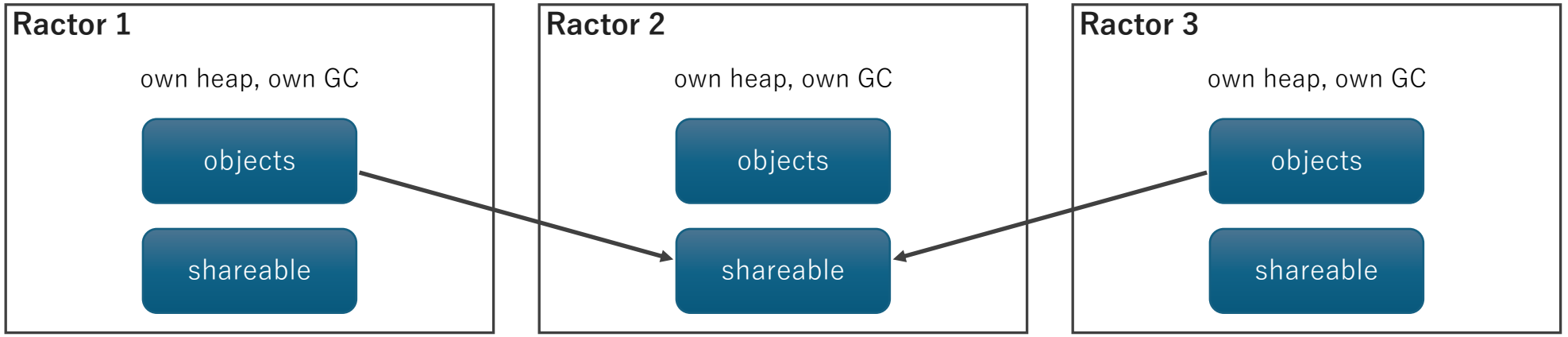
```
port = Ractor::Port.new
Ractor.new(port) do |port|
  port << answer()
end
port.receive #=> 42 (wait for the answer)
```

- Port is a primitive for synchronization between Ractors
  - Like Erlang's mailbox (labeled mailbox)
  - `Ractor#join` is implemented on the Port
  - We can check system-level blocking by observing ports
- Objects are copied or moved on sending

Part 1
Recent improvements

The two changes

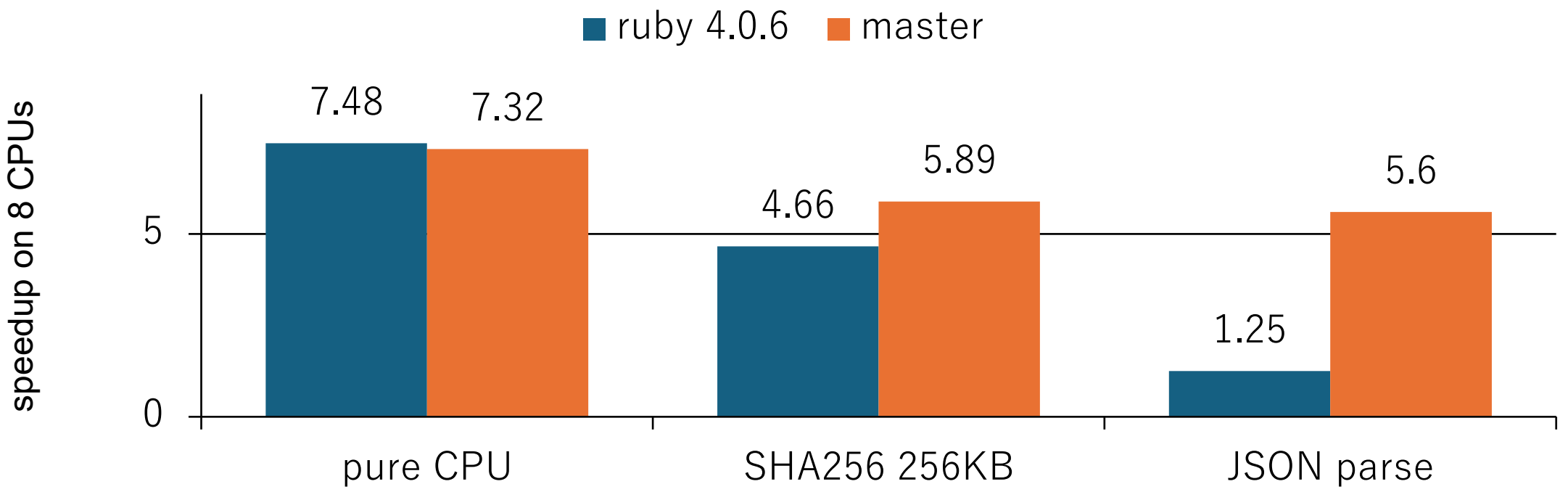
Ractor local GC with per-Ractor objspace



- **Run GC in parallel** for Ractor local Objects
  - **Feature #22227** https://bugs.ruby-lang.org/issues/22227
- Shareable objects live until Global GC
  - Stop all Ractors and collect them same as Ruby 4.0

Measurements on Ruby HEAD

Allocation heavy Ractors

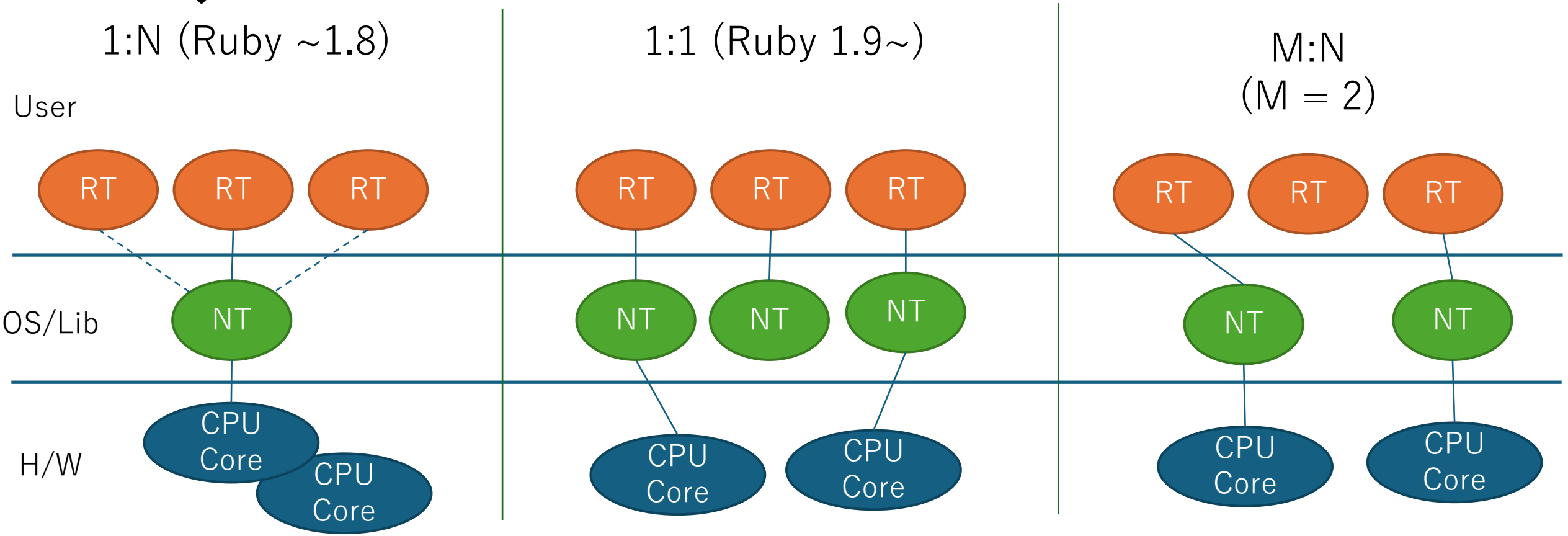


- pure CPU makes **0 objects** for each item. SHA256 makes **21**. JSON parse makes **2,405**.
- On the 8 CPUs

Introduced from Ruby 3.3
Improved on Ruby HEAD

The two changes

Improve the M:N thread scheduler

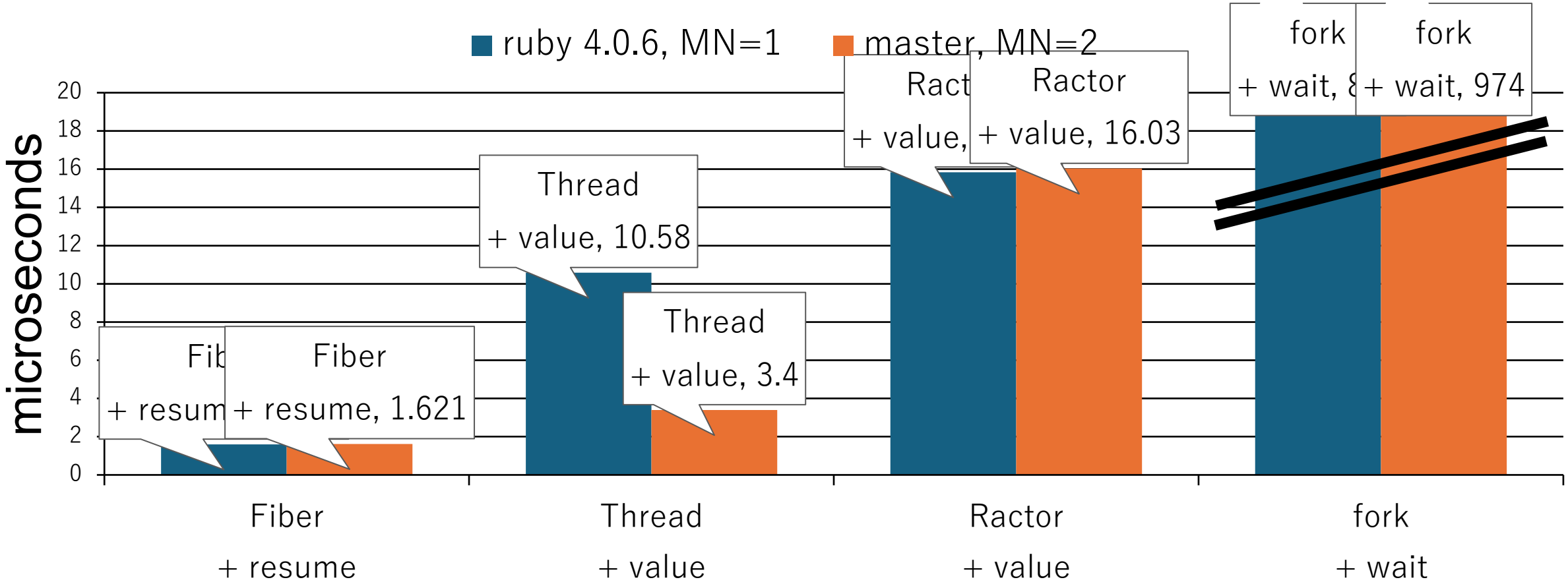


✅ Remove O(n) logics, ✅ Supports IO#wait_readable/writable
✅ Add “all” M:N option (RUBY_MN_THREADS=2)

RUBY_MN_THREADS=2 is introduced

Measurements on the Ruby HEAD

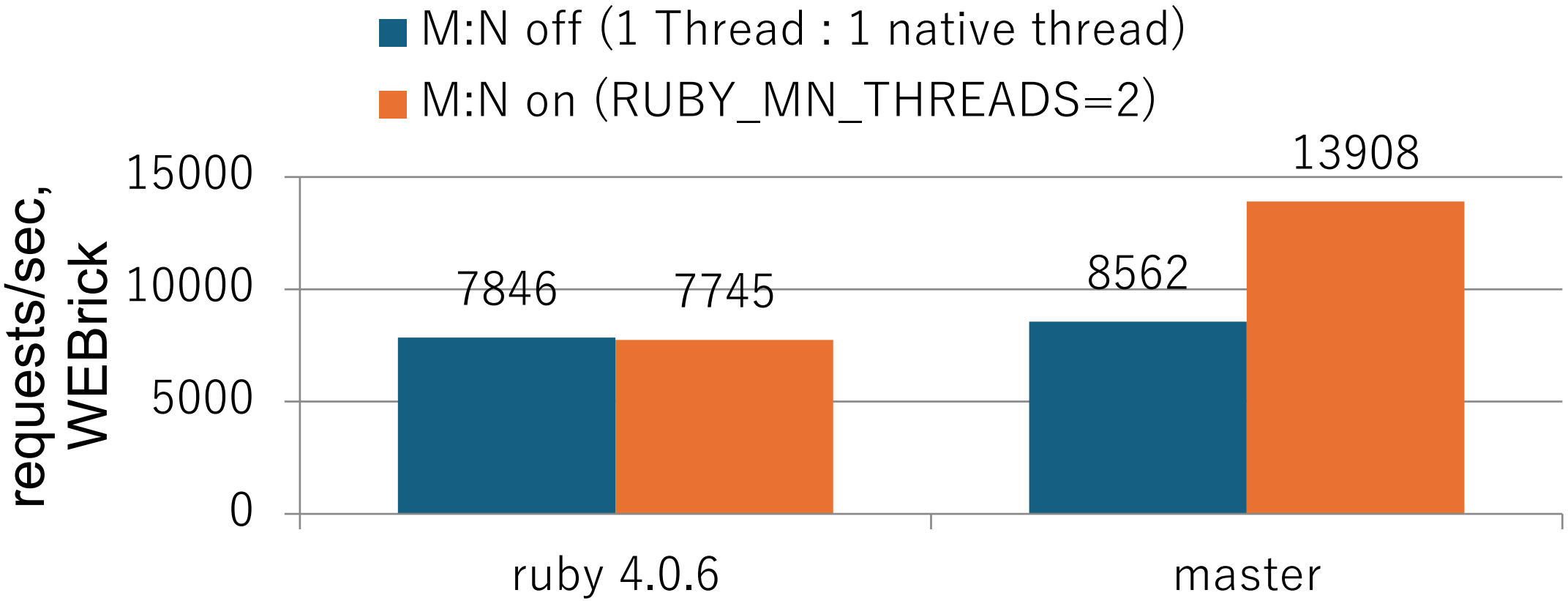
What one unit of concurrency costs



Thread + value: 10.6 us -> 3.4 us. Each version at its best setting; 4.0.6 has no `MN=2`.
MN is `RUBY_MN_THREADS`. `Ractor` and `fork` are slightly slower on master. Median of 3 runs on 8 cores.

Measurements on Ruby HEAD

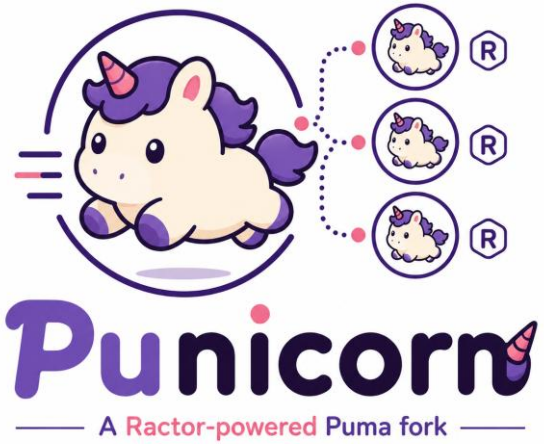
M:N scheduler improves 1 Thread per Conn



Native threads: 259 -> 3. 4.0.6 with M:N on still holds 68-85. **CPU per request: 152 us -> 72 us.**
WEBrick 1.9.2, 256 keep-alive connections, `wrk -t8 -c256` from another machine. `TCP_NODELAY` on, or the delayed-ACK timer caps every build at the same 6,100 rps.

Measurements on Ruby HEAD
punions: two experimental Rack servers

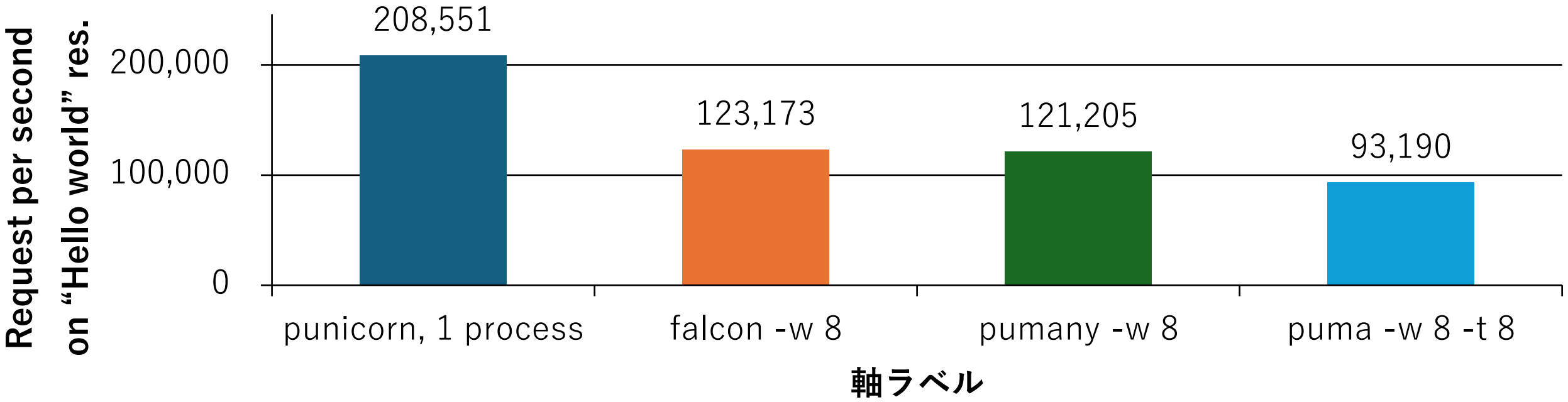
- Puma based servers
  https://github.com/ko1/punions
  - To measure HTTP servers
  - Only connection handling is changed
- PuMaNy
  - 1 Thread per connection, 595 lines diff.
  - Measure for M:N scheduler
- Punicorn
  - 1 Ractor per connection, 1,810 lines diff.
  - Like “unicorn” model (1 process per conn.)
  - A Rack app must be Ractor-aware



ℹ️ Just ask Claude Code to make them
ℹ️ Logos are made by ChatGPT

Measurements on Ruby HEAD

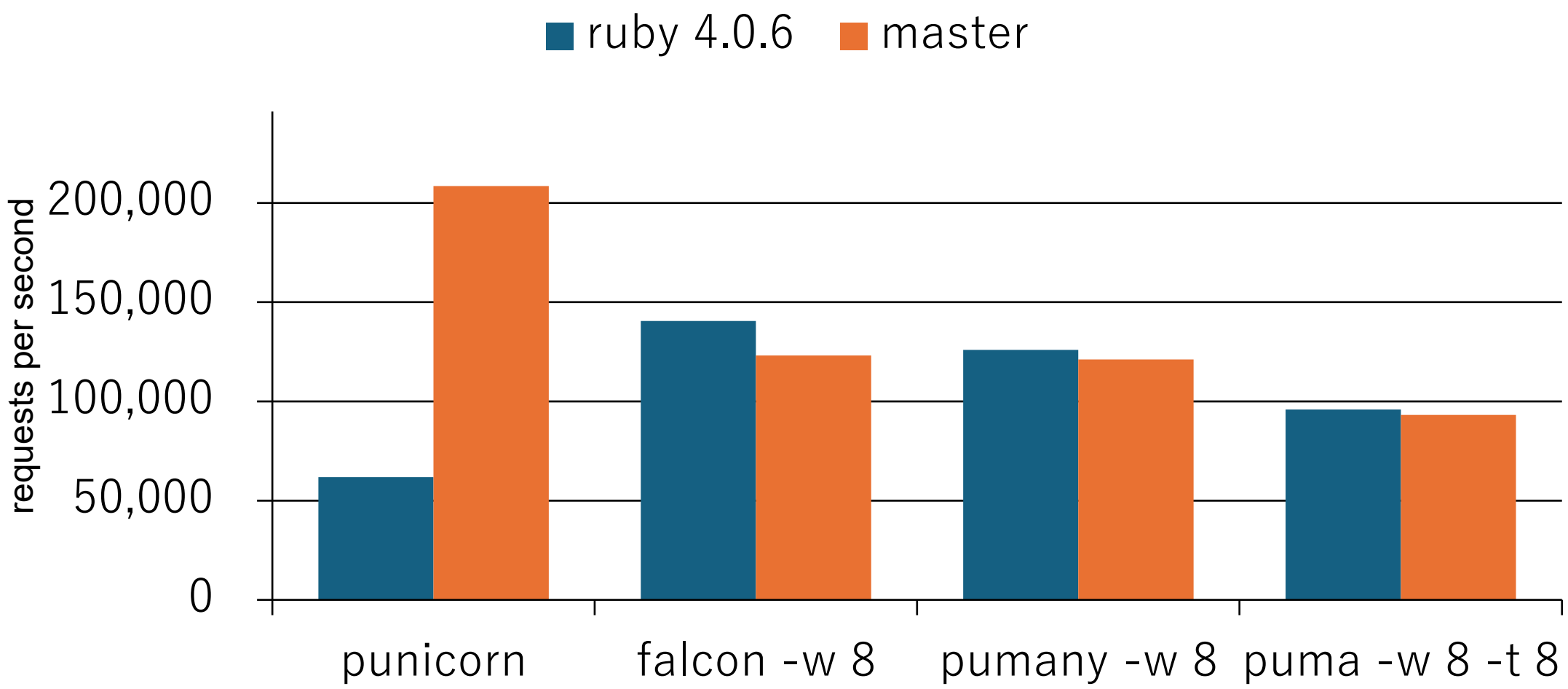
Rack servers with a simple handler



- punicorn is **1.69 times faster than the best of the other three.**
- Each request costs 35.0 us of CPU, against falcon's 60.8 us.
- Server on 4 physical cores, wrk on the other 4, keep-alive on.

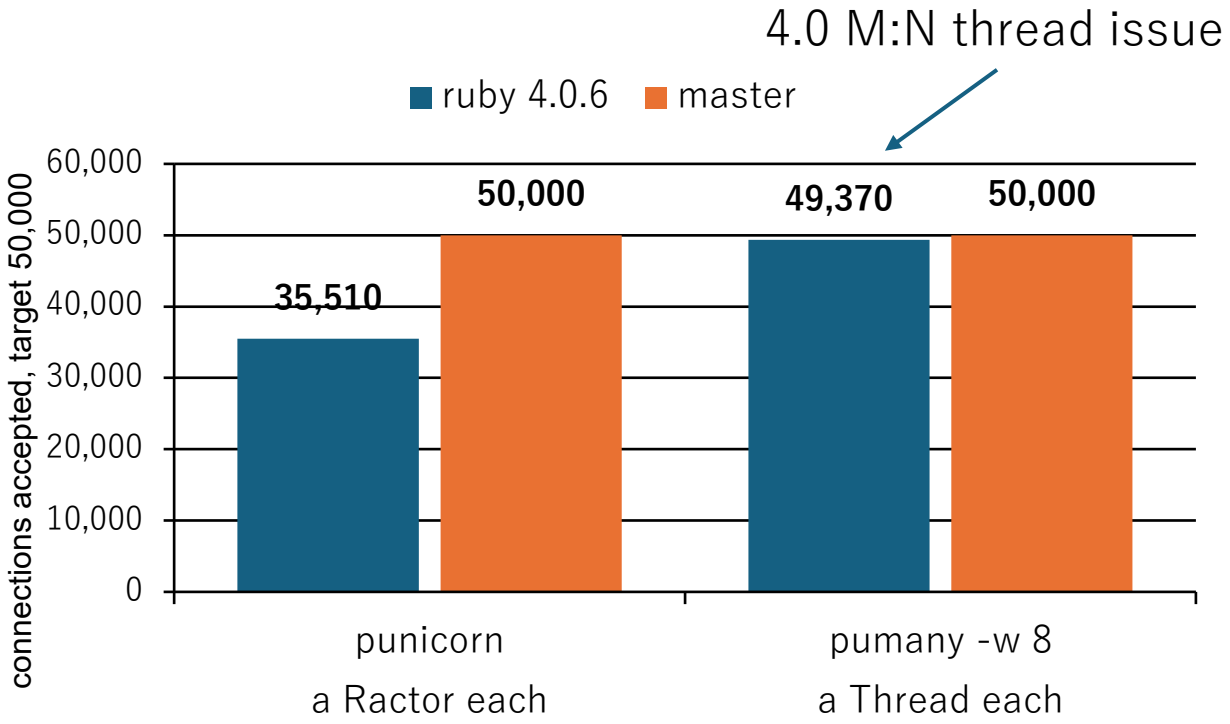
Measurements on Ruby HEAD

Between 4.0 and Ruby HEAD



Measurements on Ruby HEAD

WebSocket: 50,000 connections on one box



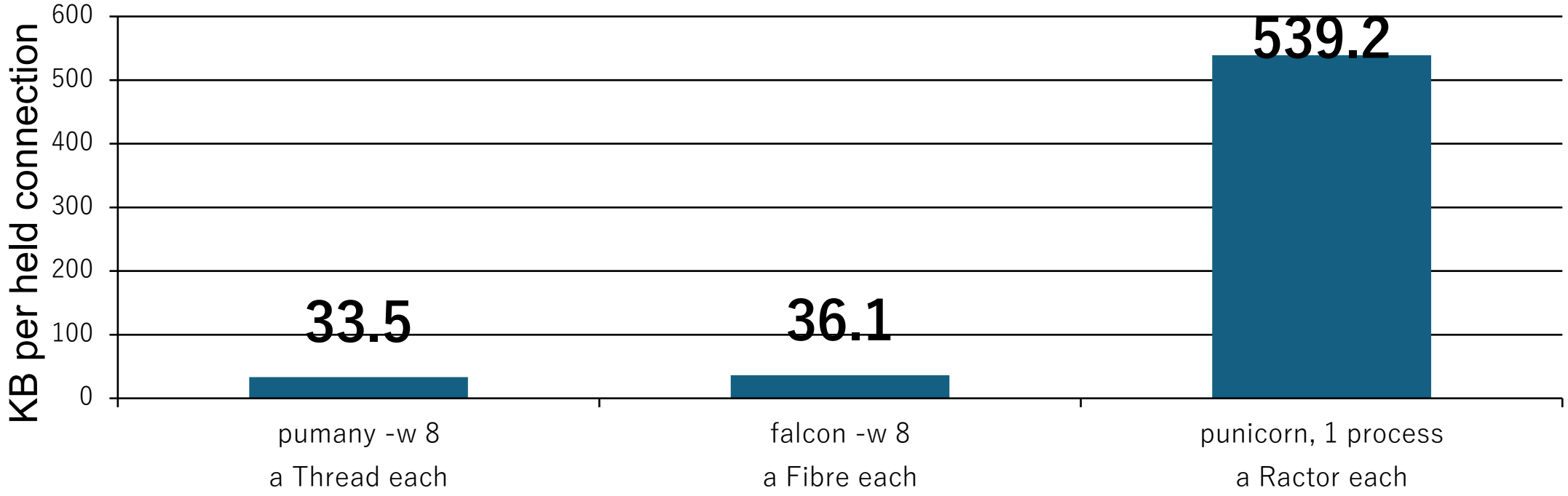
master	p50	p99
pumany -w 8	41-59 ms	84-137 ms
falcon -w 8	44-78 ms	102-207 ms
punicorn	**29-190 ms**	**55-1,835 ms**

Latency on master.
punicorn has both
the fastest and the slowest.
A global GC issue. We are investigating.

- Try to hold 50,000 WebSocket connections on one box, 10,000 messages a second.
- Both on master reach 50,000 and answer every message. punicorn on 4.0.6 stops at 35,510.

Measurements on Ruby HEAD

WebSocket: 50,000 connections on one box



Holding 50,000 idle connections: **1.88 GB**, 2.10 GB, **26.99 GB**. **539.2 KB for one Ractor**, against 33.5 KB for a Thread. It is flat from 1,000 connections to 50,000, so it is a floor for a Ractor, not a leak. **This heap is what gets master to 50,000** - and it is the number I want to reduce next.

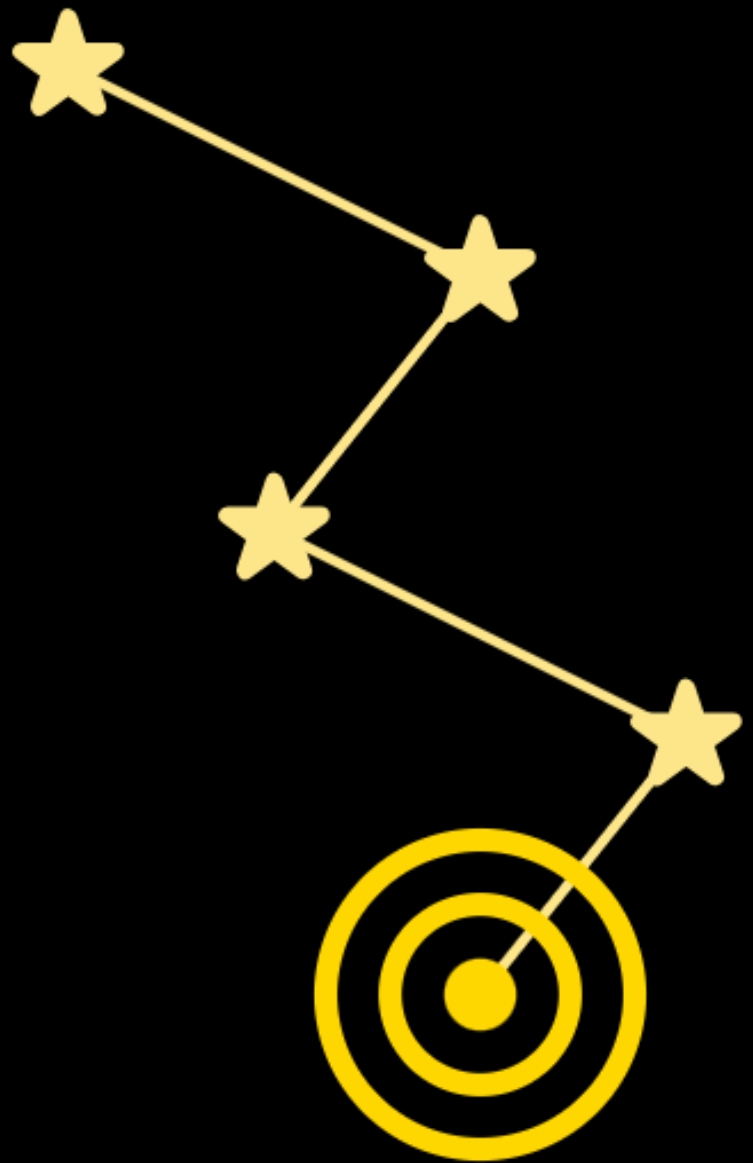
Ractors on Ruby HEAD improved better

Please try!

Intrude

Another AOT Ruby compiler

https://github.com/ko1/astro/



ASTro

AST-based

Reusable Optimization

Framework

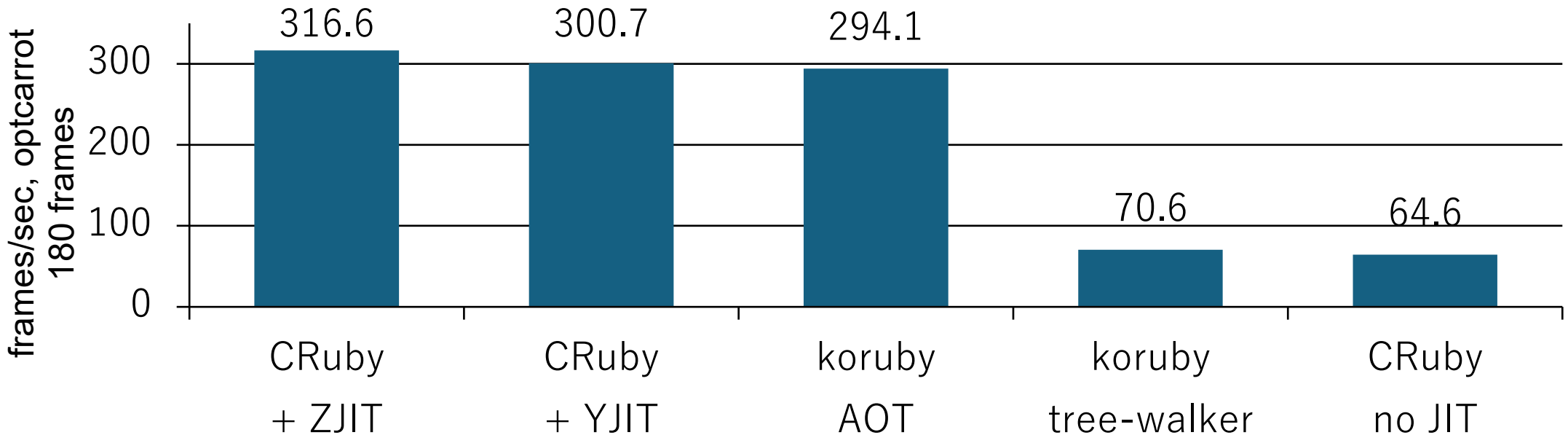
ASTro-based Ruby interpreter

koruby (kind of Ruby)

- Goal: Drop-in replacement faster Ruby
  - Built on ASTro: Writing AST interpreter and get AOT/JIT compilers for free.
  - Better GC (Moving GC)
  - rubyspec: core + language + library ⇒ 98.14%

Measurements

Optcarrot on koruby



The JITs warm up at run time; koruby bakes ahead of time.

The tree-walker alone beats the CRuby VM, with no specialisation at all.

optcarrot 180 frames, median of 5 runs, 16 cores, machine to itself. All CRuby modes are one build, ruby 4.1.0dev of 2026-08-28 +PRISM. **ZJIT is still in development - this is that build, not ZJIT today.** Baking costs 27 s once per program, not counted here.

Numbers of ASTro

23/29

Languages / Interpreters

6167

Commits
Since 2026/Apr

0/3

PRs / Contributors
(ko1, Claude, Codex)

Resources of ASTro

RubyKaigi 2026

The AST Galaxy to the Virtual Machine Blues

https://rubykaigi.org/2026/presentations/ko1.html

Documents on ko1/astro

https://github.com/ko1/astro/

Many documents written in Japanese,
but machine translation will help you.

Part 2
Why do we need Ractor on Ruby?

Ractor is for Happy Programming

- Concurrent and parallel programming is hard
- Ractor tries to reduce the difficulties and makes **parallel programming on Ruby happy**

Ractor is for Happy Programming

Why not “parallel Threads”?

- Threads share all objects by default
  - Example: `ARY = [1, 2, 3]`
    And any thread can mutate `ARY`
  - Protection or synchronization (lock, etc.) is needed for concurrent access
- Difficult to

  Know they are needed
    - Such issues are difficult to reproduce
    - E.g. ThreadSanitizer for detection, but not perfect

  Use them correctly
    - Each synchronization tool has its own rules
    - E.g. Deadlock with multiple locks

Ractor is for Happy Programming

The goal (1): keep Ruby enjoyable

- The main idea
  Avoid “sharing mutable states” issues by making sharing mutable states explicit
  - By “Shareable object rules”
  - 😿 Isolation errors in some cases
- It means building a **new language** on Ruby
  - Happy usual Ruby programming
  - Happy healthy parallel Ruby programming on Ractors

Ractor is for Happy Programming

The goal (2): make Ruby faster/safer

- Fine-grained locks are not needed with isolation
  - String, Array, Hash, … don't need fine-grained locks for safe parallel programming
    - GVL protects us from harder issues
    - E.g. Research on Java VMs, Python removing GIL (PEP-703)

(1) Makes the interpreter **FASTER**

(2) Makes the interpreter **SAFER**

- "**Parallel programming is difficult**" is the same for interpreter developers

Ractor is for Happy Programming

The cost: Makes Ruby smaller on Ractors

- Introduced **some** restrictions on Ractors
  - Need to consider “shareable” and “unshareable”
  - Shareable Proc has several restrictions
  - Constants/class variables/class instance variables have several restrictions
    - Only the creator (owner) ractor can set and only shareable values can be read by other Ractors
    - Some more …
  - Global variables are not allowed on sub-Ractors

Ractor is for Happy Programming

Design: Sharing mutable state safely

- Well-synchronized sharing state mechanisms
  - Transactional variables (e.g. Haskell, Clojure, …)
  - Process-wide Hash table (e.g. ETS on Erlang, DB, …)
  - Active Object pattern (e.g. Erlang GenServer, Akka actors, Ruby's Celluloid)
- Not provided officially
  - https://github.com/ko1/ractor-sharing

Fast on no-conflicts!

Sharing mutable state safely

Transactional variables

- Optimistic locking (e.g. RDBMS)
  - Rollback (re-do the atomically block on confliction)

```
C = [Ractor::TVar.new(0),
     Ractor::TVar.new(0)].freeze
Ractor.atomically {
  C[0].value += 1
  C[1].value += 2
} # Guarantee “C[1] == C[0] * 2”
```

Fast on no-conflicts!

Sharing mutable state safely

Shareable Hash (like) table

- (mostly) Lock-free key/value table

```
Cache = Ractor::KeyLockHash.new
page = Cache.store_if_absent(:foo){
  render_page(:foo)
}
```

⚠ **No way to update multiple keys atomically**

Sharing mutable state safely

Active Object pattern

- Close mutable states in a (server-like) Ractor
  - Send requests to the Ractor and get response

```
class People < Ractor::ActiveObject
  def initialize = @db = {}
  async def add(name, age) = @db[name] = age   # does not wait
  sync  def find(name)     = @db[name]  # waits for the answer
end

people = People.new
people.add("ada", 36)
Ractor.new(people) { |p| p.find("ada") }.value   #=> 36
```

Ractors for Scripting

DSL: Ractor::Pipeline

- Ractor DSL like Shell's pipe
  https://github.com/ko1/ractor-pipeline
  - Each block runs in its own Ractor
  - "lanes:" generates multiple consumer Ractors

```
require "ractor/pipeline"; include Ractor::Pipeline
# cat | grep ERROR | jq …
stream(File.foreach("app.log"))
  .filter_pipe(lanes: 8) { it.include?("ERROR") }
  .pipe(lanes: 8)        { JSON.parse(it) }
  .to_a
```

Additional Part

My Ten years with Ractors

~2017

2017~2023

2016 I showed the design at RubyKaigi. It was called **Guild.**
2017
2018 A prototype ran. It was slow.
2019
2020 Ruby 3.0 shipped it as an experiment. The name became **Ractor.**
2021 I started a debugger for Ractors.
2022 **M:N threads**. A Ractor stopped needing its own native thread.
2023 I asked people to try Ractor (a chicken and egg problem)
2024 I made ordinary things work inside a Ractor. require. timeout.
2025 I proposed **R** :Port API
2026 Ractor loca nprovements.

2023~

STORES

AI Coding

STORES

Question

Does Ractor matter in the AI coding era?

- The goal: **“Ractor is for Happy Programming”**
  - For humans. That was the point in 2016.
  - … but today, how much of the code do humans write?
  - AI seems to write good parallel code and to do smart debugging.

- Question: Does Ractor make AI happy too?

Question

Does Ractor matter in the AI coding era?

- My answer: YES (A position talk, of course) (at least now)
- Writing code became cheap. Checking it did not.
  - AI writes good local code. A rule that is not in the code is expensive to check. For a person, and for AI.
  - A data race is that kind of rule.
    - Sometimes. Only on some runs.
    - Somewhere else. Not where it crashed.
    - Later. Two weeks in production.

Question

Does Ractor matter in the AI coding era?

- Ractor puts the rule in the interpreter.
  That kind of race cannot happen; an error happens instead.
  - Always. Ractor::IsolationError raises on every run.
  - Here. At the line that broke the rule.
  - Now. Inside the loop that wrote it, not two weeks later.
- AI can debug. But the cheapest bug is the one that cannot happen.

Question

Does Ractor matter in the AI coding era?

- Ractor's restrictions are a boundary the interpreter checks, for humans and for AI.

- Just luck, not foresight.

What is left

- Performance
  - Memory consumption
  - Global GC tuning
- Ecosystem
- Removing the "experimental" mark.

Acknowledgements

- Contributors
  - Rohit Menon. Ractor local GC started as his Google Summer of Code project in 2022, and we have discussed on it together since.
  - The Shopify Ruby team, for trying Ractor and improving it.
  - And other contributors
- Supporters
  - Heroku/Salesforce, Cookpad, STORES
  - GitHub sponsors https://github.com/sponsors/ko1/
- The AI agents that moved this forward
  - "Claude for Open Source" by Anthropic

Wrap-up
Today's topic

Ask for detail at
the RHC workshop

1. Ractor on Ruby HEAD is very good now
2. Why do we need Ractor on Ruby?

Please try Ractor on Ruby HEAD.
It is a different thing now.

Thank you!