Toward Ractor local GC

Koichi Sasada
STORES, Inc.

Summary

- Introducing Ractor local GC is promising but challenging
  - Cross-Ractor references make this difficult
- We propose a conservative approach to enable Ractor local GC by keeping shareable objects as root objects
  - Since the number of sharable objects is small enough, it is feasible
  - At least, the behavior is equivalent to the current implementation (Global GC every time)
- We can observe **significant** improvements in micro-benchmarks

Acknowledgement

- This work is a collaboration with **Rohit Menon**
  - He is a Google Summer of Code developer in 2022
  - I mentored him on the project "Optimizing Garbage Collection for Ruby Ractors“
- Contributions
  - I provided basic idea and discuss him
  - Rohit continued developing his own implementation extending my idea over 3 years
  - I learned from his experience and developed my implementation (this talk)

Koichi Sasada

- Ruby interpreter developer employed by **STORES, Inc.** (2023~) 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~)
  - …
- Ruby Association Director (2012~)
- An owner of @.bookstore at 2nd floor

Off-topic: Ractor::Port proposal https://bugs.ruby-lang.org/issues/21262

- Proposal to change the Ractor API
  - Introduce: `Ractor::Port` for communication
  - Deprecate: `Ractor.yield` and `Ractor#takd` pair
- `Ractor::Port` is a small enhancement of the mailbox concept in the Actor model

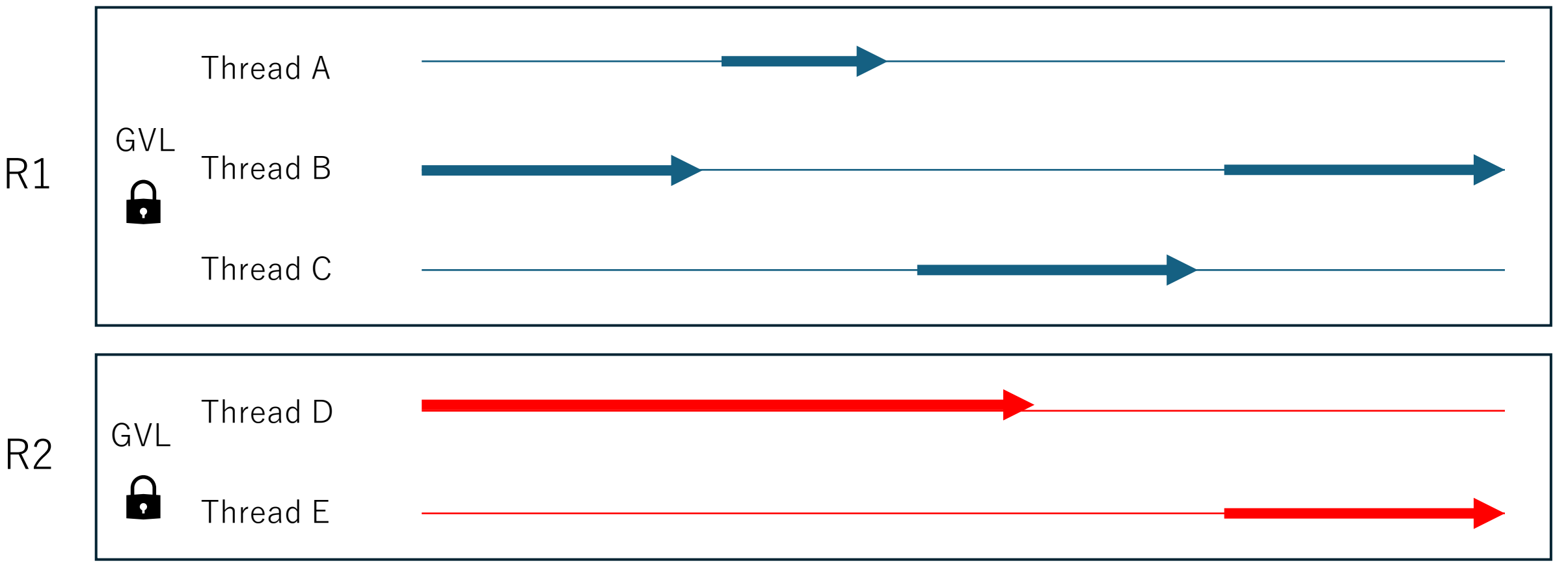
Background

“Ractor” is

- Introduced in Ruby 3.0
- Designed to enable
  - 😄 parallel computing on Ruby for more performance on multi-core CPUs
    - Enables faster applications
  - 😄 robust concurrent programming
    - Prevents bugs caused by shared mutable state

What is parallel?

- Threads of different Ractors can run in **parallel**
  - They can utilize CPU cores on your machine



Ractor as an Isolated Object Space

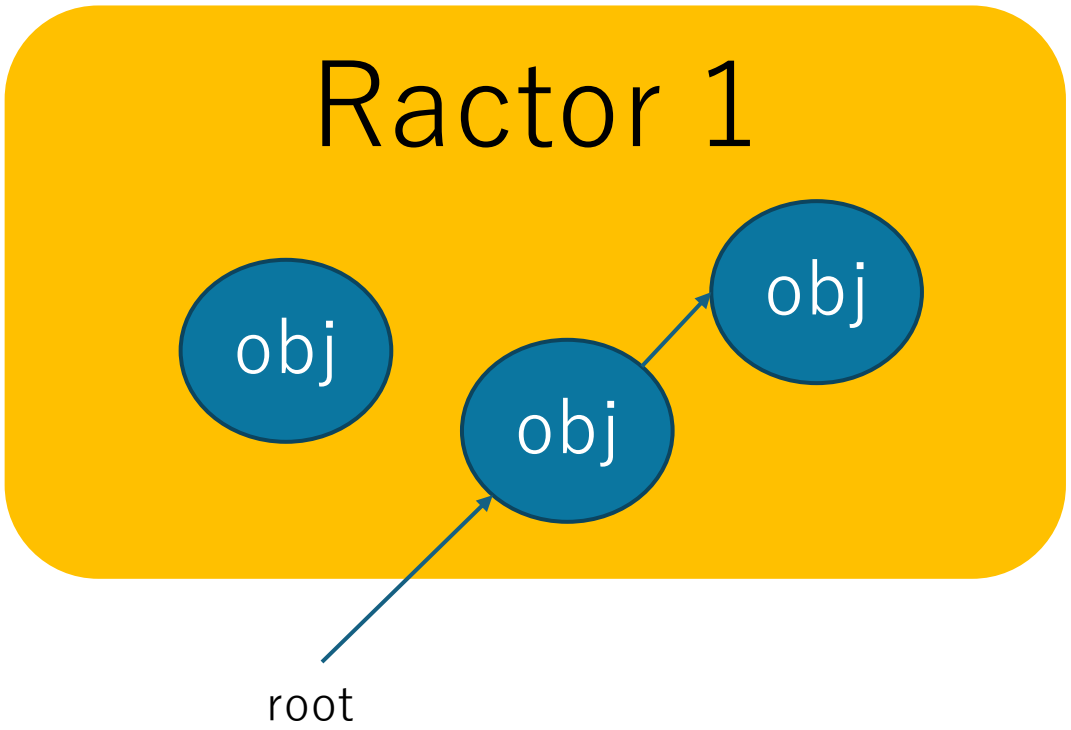
- Each Ractor has its own isolated object space (objspace)
  - Every object (obj) belongs to exactly one Ractor

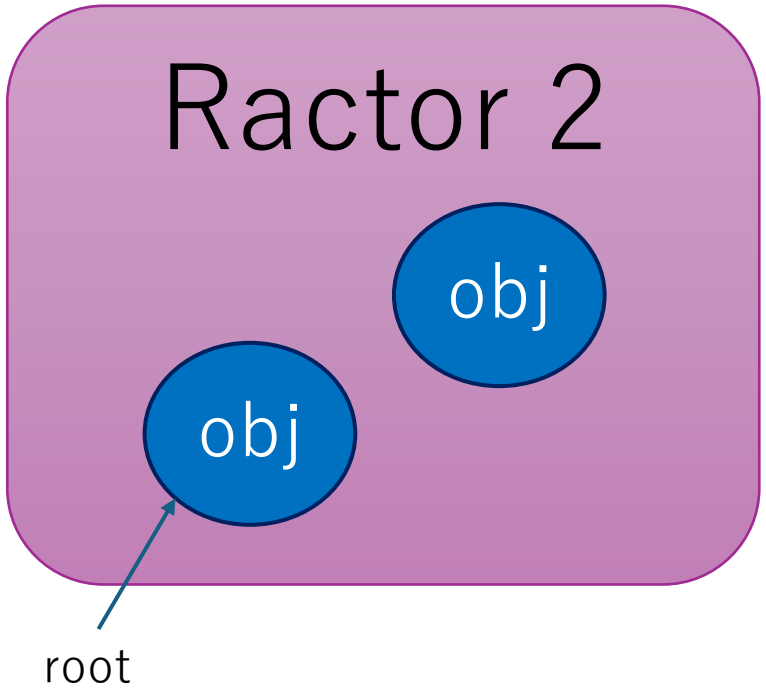
Ractor as an Isolated Object Space

- Cannot access objects in other Ractors

“Ractor local GC”
It seems natural to run GC simultaneously

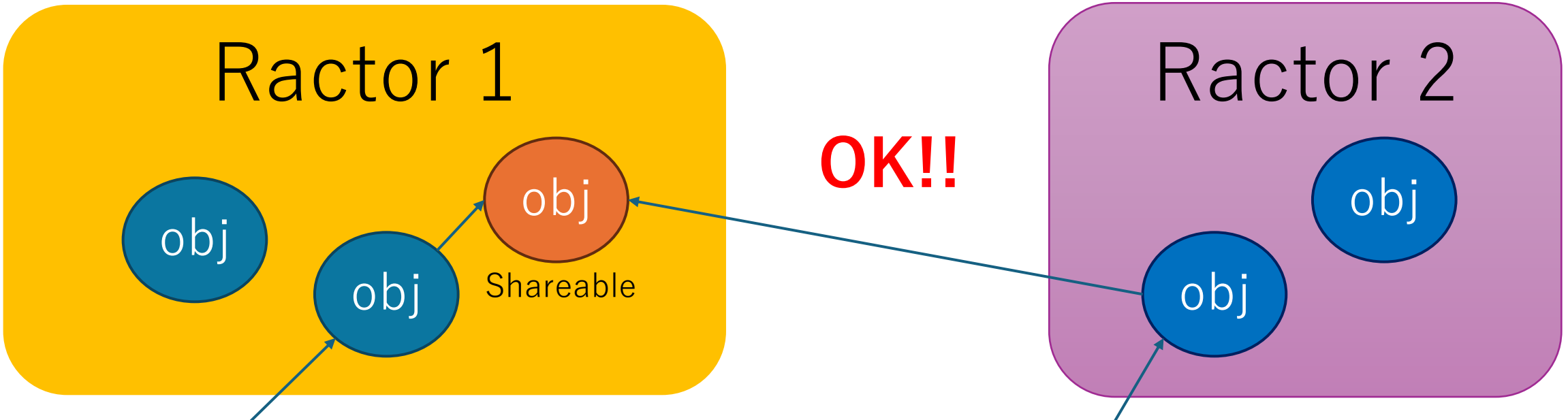
- Run Ractors in parallel
- Run GC independently in parallel in each Ractor objspace





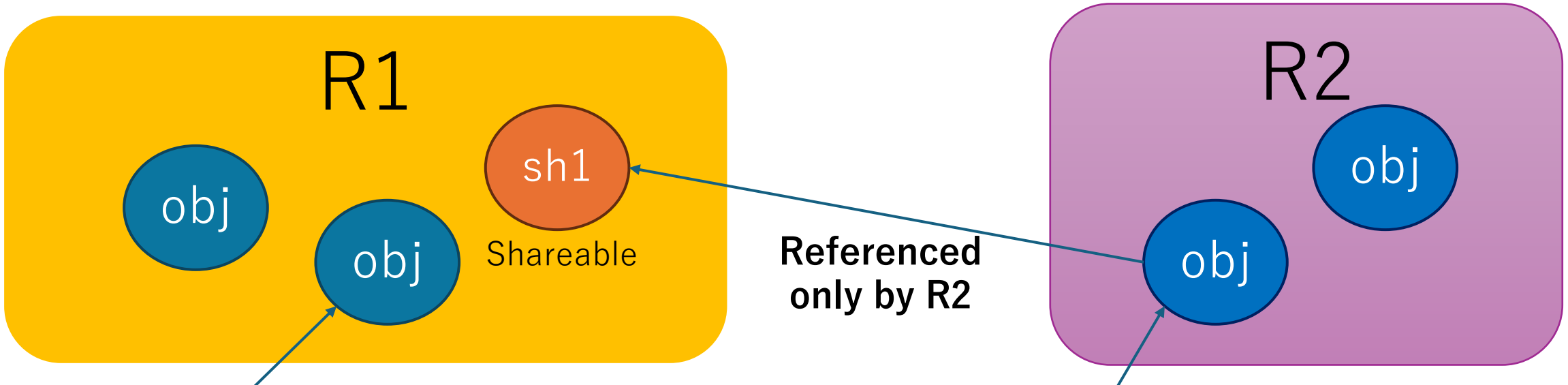
Ractors can share “Shareable objects”

- Shareable objects (shobj):
  - Classes and Modules
  - Immutable objects (frozen and only reference shareable objects)
  - Special objects (Ractor objects and so on)
- Ractors can hold to shareable objects



Question: How should Ractor-Local GC handle sharable objects?

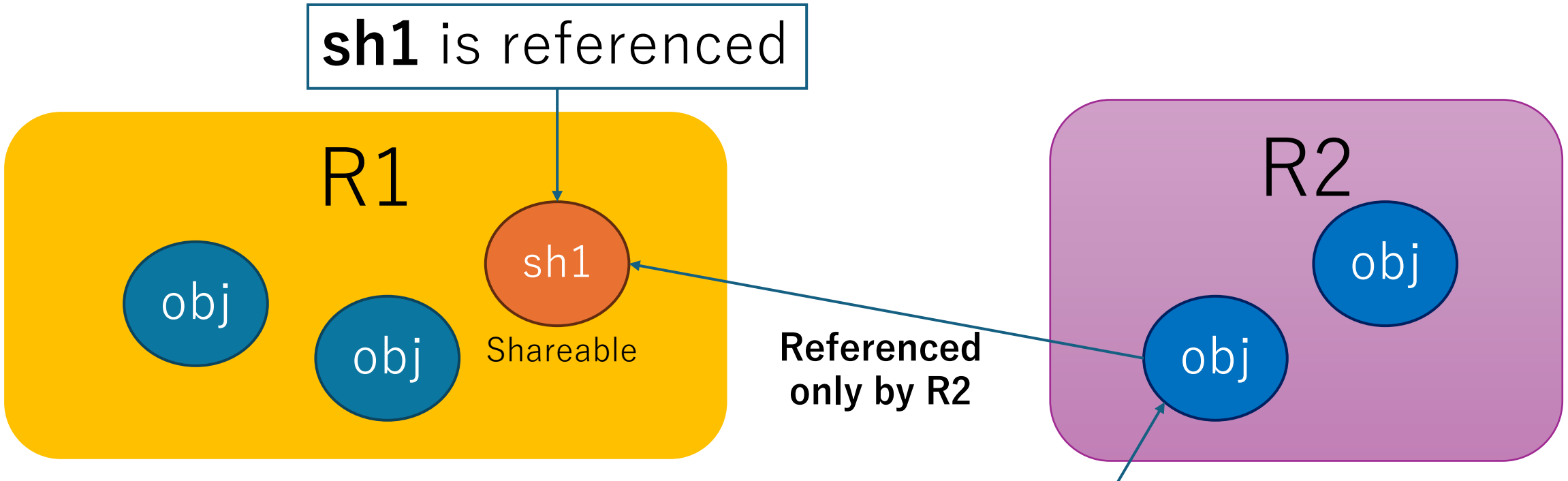
1. Ractor R1 has a shareable object (**sh1**)
2. Ractor R2 holds a reference to **sh1**
3. R1 run GC and **sh1** is collected because no refenrece to sh1 (even though R2 does)
4. R2 tries to use **sh1**, but it has already been freed → CRASH!



Challenges

Solution (1)
Tracking references to shareable objects

- If R1 knows that **sh1** is referenced by another Ractor, its GC can take this into account and avoid collecting **sh1**



Problem with Solution (1)
Tracking shareable references is Difficult

- References can be updated at any time
- All Ractors execute in parallel

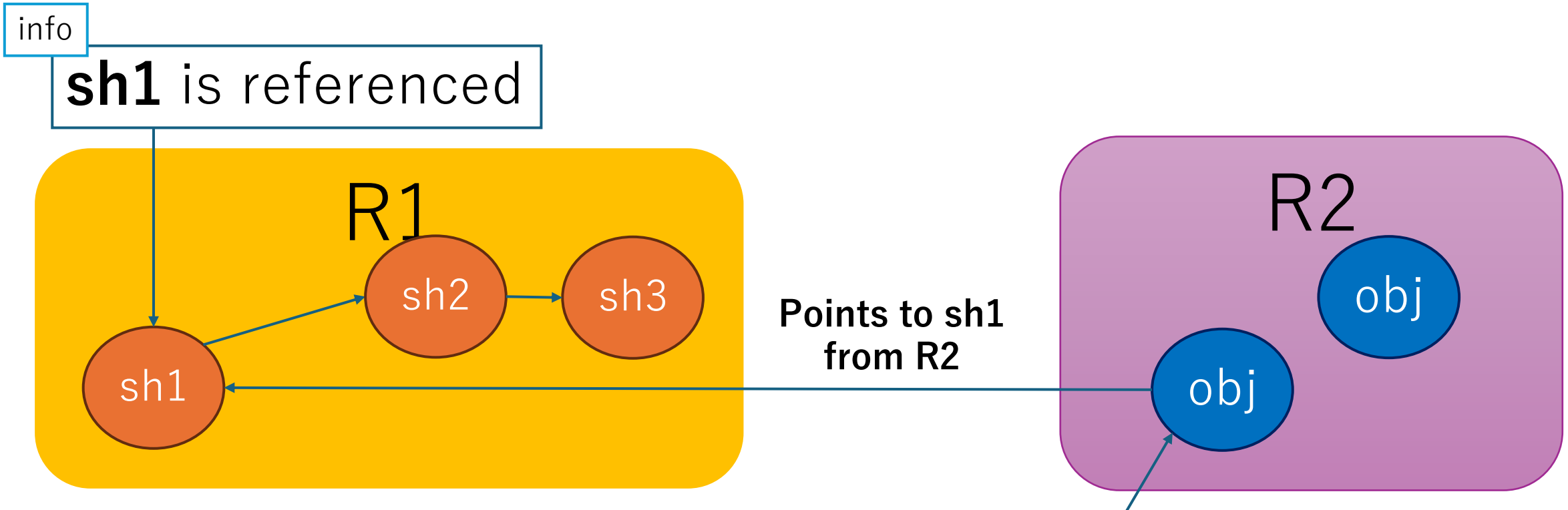
→ Difficult to keep reference states accurate across Ractors

😥 Incorrect information can introduce marking miss

- Performance concerns
  - Additional overhead from tracking inter-Ractor references
  - Additional memory usage

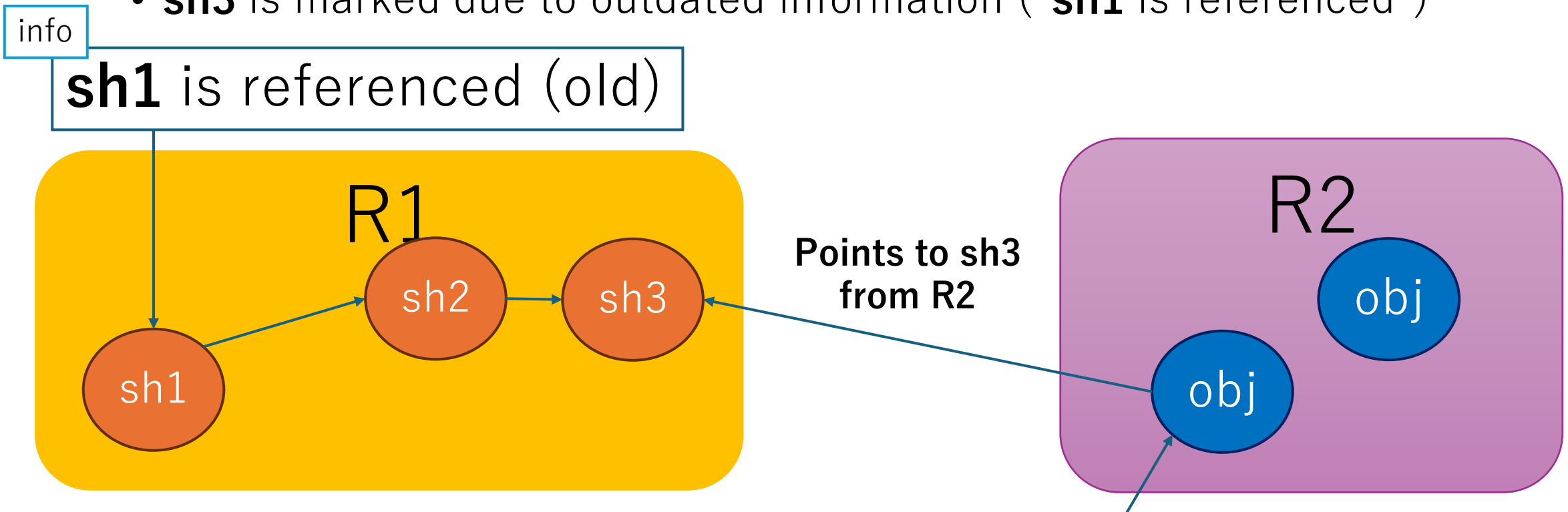
Sample scenario

1. **sh1** is sent from R1 to R2 and R2 holds a reference to **sh1**
2. The runtime detects “**sh1** is referenced”



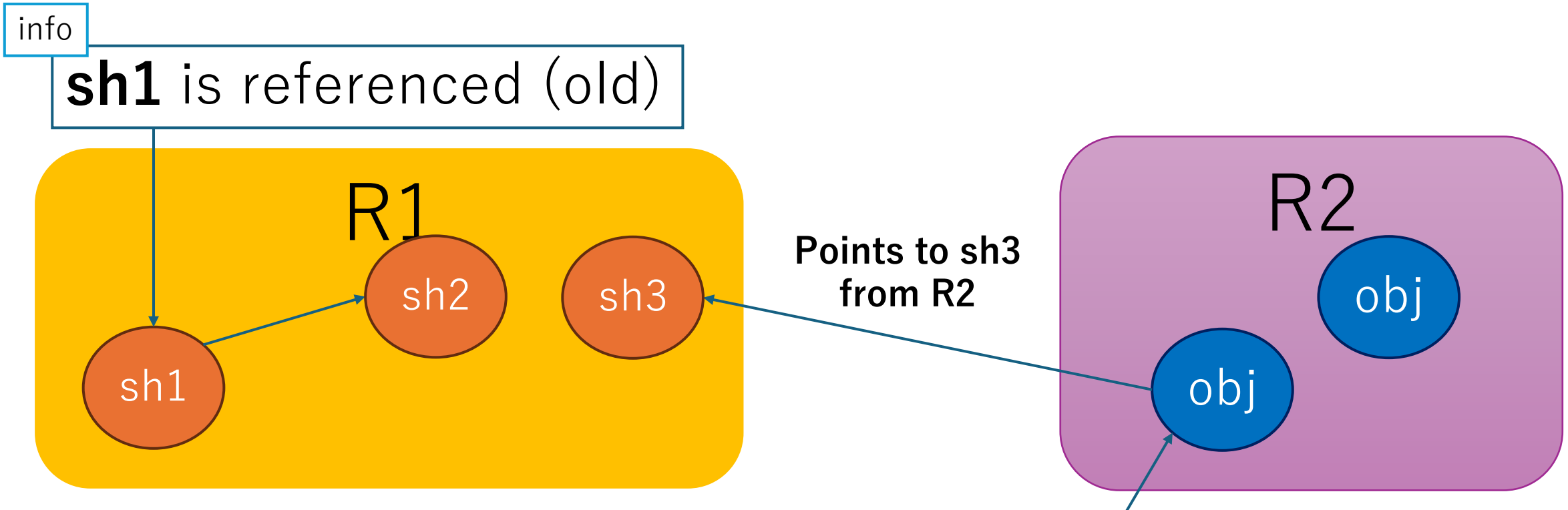
Sample scenario

3. R2 references **sh3** (because it's reachable from **sh1**)
   - The reference information becomes stale, but cannot be updated immediately because no mechanism to that (or too slow)
   - **sh3** is marked due to outdated information ("**sh1** is referenced")



Sample scenario

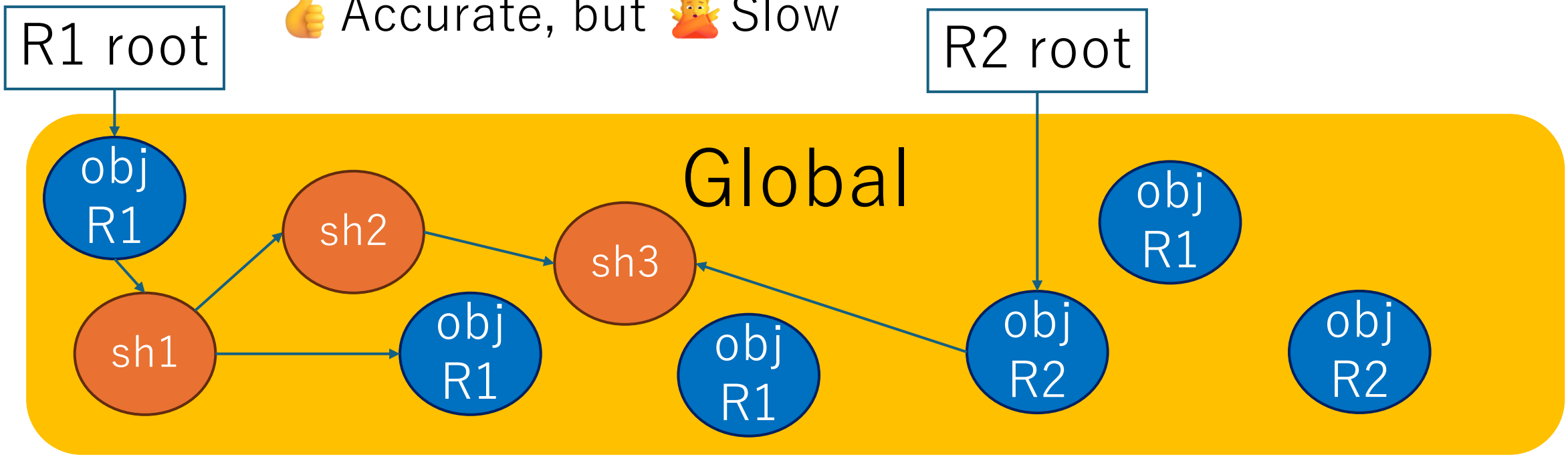
4. **sh2** removes the references to **sh3**, making **sh3** unreachable
5. As a result, **sh3** is collected by R1's local GC



Solution (2) Global Object Space

- The current implementation uses a single “global object space”
- No per-Ractor heap: objects are managed in one shared space
- GC **stop all Ractors** and marks objects correctly

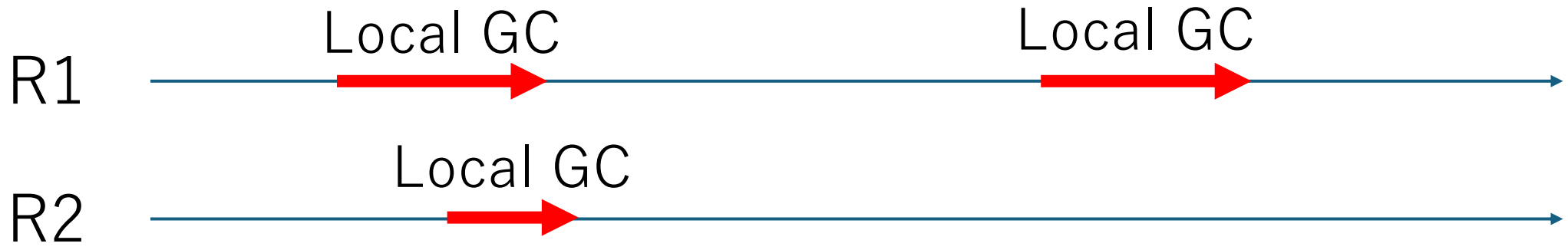
👍 Accurate, but 🫠 Slow



GC timeline

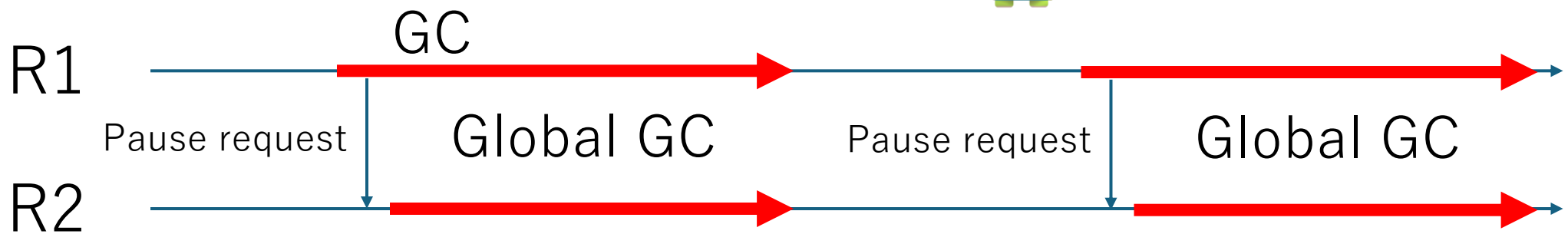
Solution (1) Local GC

👍 Only Ractor's objects
👍 Runs in Parallel
😢 Difficult to implement



Solution (2) Current Global GC

👍 Easy to make
👍 Space efficient
🐢 Slow



The Goal and the Challenge

- Goal: Introduce Local GC to improve performance
  - Track only Ractor local objects
  - Run Local GC in parallel, independently (like Erlang/Elixir)
- Difficult to implement with shareable objects
  - Tracking shareable objects across Ractors is hard
  - (This is why Ractor local GC hasn't been introduced for years)
- Challenge: Design a safe algorithm for Ractor local GC

Proposal

Proposal: Do not collect Shareable objects

- The problem is “hard to detect liveness of shareable objects”

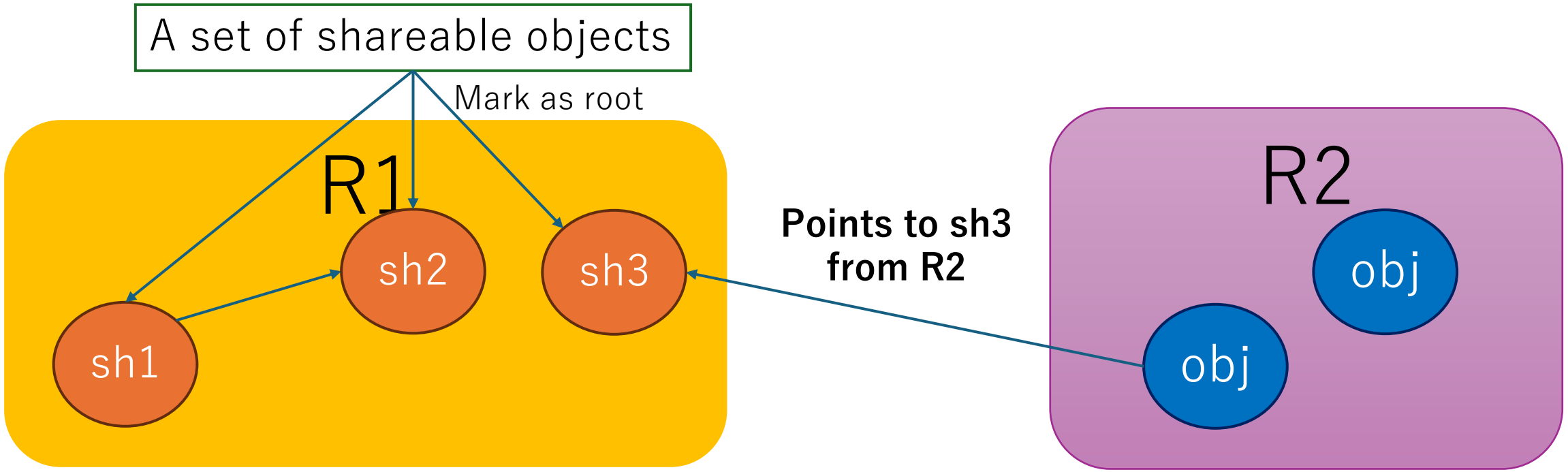
→ **Give up detecting liveness; assume they are alive**

  - The number of shareable objects is limited
  - A conservative but safe approach

- On Local GC, Mark local shareable objects as root objects
- Use Global GC to reclaim shareable objects correctly
  - Stop all Ractors and check cross-Ractor references
  - Run less frequently than the current Global GC

Shareable objects as Root objects

- 👍 Accurate
- ❓ Shareable objects that no longer reachable are never reclaimed until Global GC



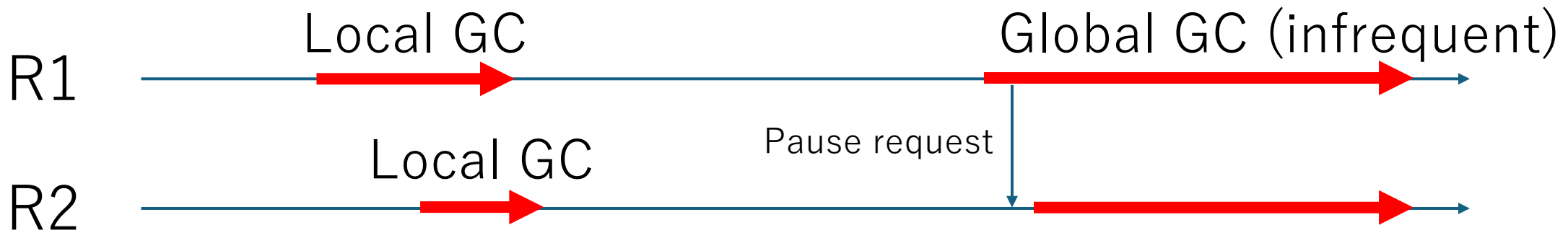
Lifetime of Shareable objects

- Classes, Modules, methods and bytecode (ISeqs)
  - Typically, long-lived objects
- Ractors
  - Depends on applications, but most are not short-lived now
- Immutable objects (marked as shareable)
  - Depends on the application
  - Immutable objects assigned to constants tend to be long-lived

Few shareable objects are short-lived

→ Conservative approach is acceptable

GC timeline (proposed approach)



- 👍 Only Ractor's objects on local GC
- 👍 Runs in Parallel
- 👍 Easy to implement

Key insight:
Done is better than Perfect

- Solution (1) is ideal, but difficult to implement
  - We've discussed this for 3 years but haven't completed yet
  - And we couldn't deliver any improvements
- Proposed conservative approach may not be perfect, but it's likely better than the current implementation (solution (2))
- Let's adopt a practical and effective compromise
  - As long as the number of shareable objects remains small, this approach should work well
  - It can serve as a baseline for future algorithm improvements

Implementation

1. Introduce a verifier
   - Shareable objects should only refer to shareable objects
     - There are some exceptions though
   - Implement in `GC.verify_internal_consistency`
   - With this verifier, sharable states become correct
2. Separate Object Space per Ractor
   - All Ractor has own heap
   - Introduce Global Object Space as well
3. Track a set of sharable objects as roots
   - A sharable object is appeared, add it to the shareable set
   - Note: Sharable objects never become unshareable
4. Make a Local GC that uses the set as roots
5. Make a Global GC that rebuilds a set of sharable objects
6. Implement logic to choose between Local GC or Global GC

Techniques

- A set is implemented using arrays
  - GenGC reduces the marking time, so managing with an array is efficient (and allows marking exceptional reference (e.g. sh → unsh))
  - However, if one sharable object is added, all sharable objects in the array is marked by the write-barrier and remember-set
  - To mitigate the overhead, use array list (`[prev_set, sh1, sh2, …]`) and only the last array is marked at write-barrier
- Rebuilding an array during Global GC
  - Creating new arrays during GC is not allowed
  - So, collect sharable object references across Ractors with `st_table`, and rebuild the set (as arrays) at the end of global GC
- Objects belong to the creating Ractor
  - When a Ractor is created, the Ractor object belongs to the Ractor that created it.
    - Marking functions should be separated into “Ractor-local” and “from outer Ractor”
  - At the same time, a Thread object is also created and belongs to the creating Ractor.
    - Replace the newly created Thread object in created Ractor during Ractor bootstrap

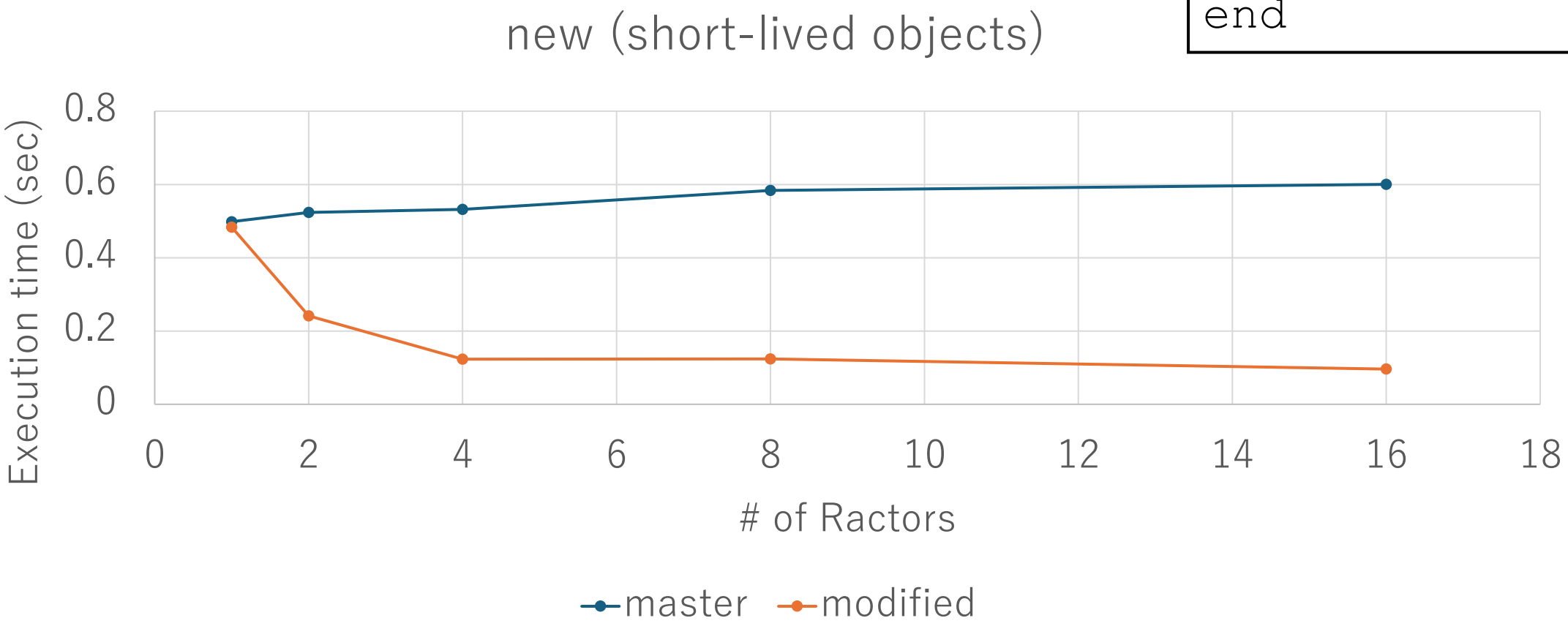
Evaluation

Benchmark setup

- Environment
  - CPU
    - 13th Gen Intel(R) Core(TM) i7-13700H
    - 14 Cores (6 performance cores with HT), 8 efficient cores)
  - Ubuntu 22.04.5 LTS
  - gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
  - Ruby versions
    - master: ruby 3.5.0dev (2025-03-17T17:19:43Z master 52f6563422) +PRISM [x86_64-linux]
    - modified: this proposal is patched on top of the above version
- Microbenchmarks
  - Repeats simple tasks N times
    - Each Ractor perform TN tasks
    - TN = N / RN, where RN is the number of Ractors
    - https://gist.github.com/ko1/5e8cb732a8d8ecfe49605674c5a01193

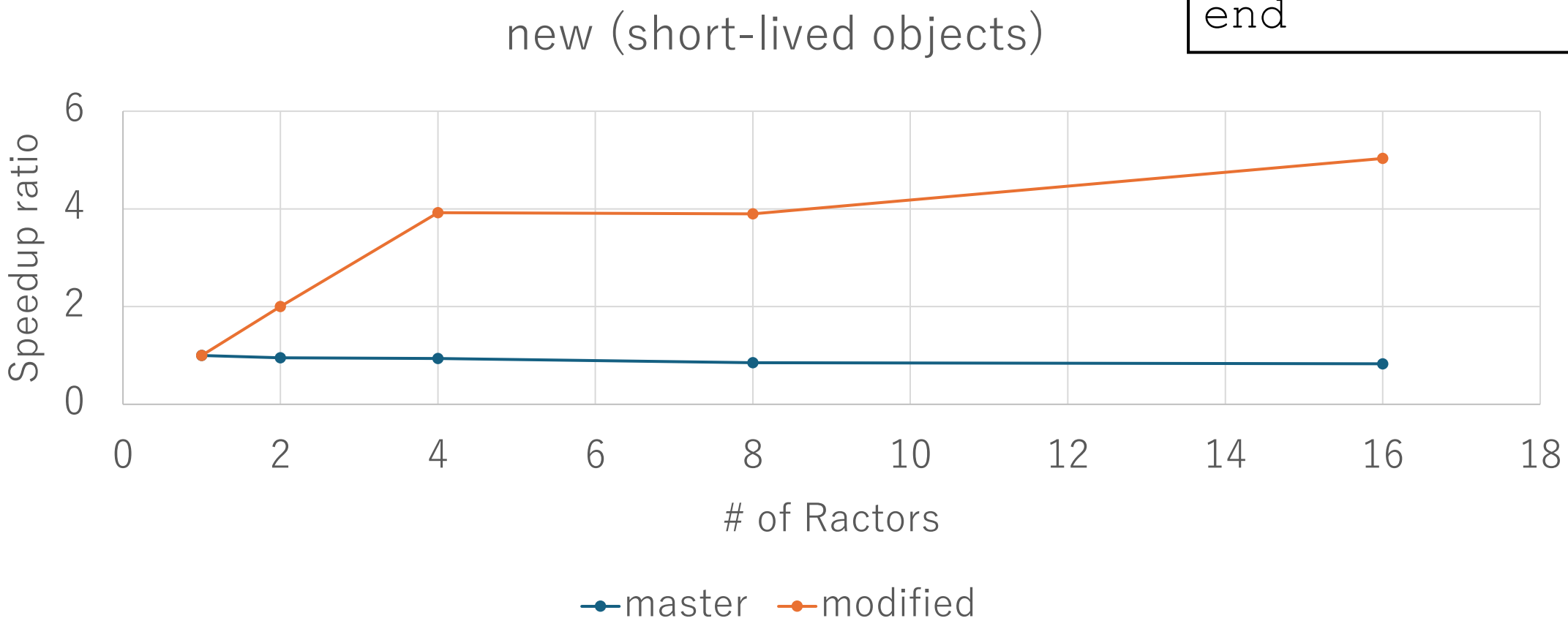
Short lived objects
Execution time (sec)

```
def task
  TN.times{
    ''
  }
end
```



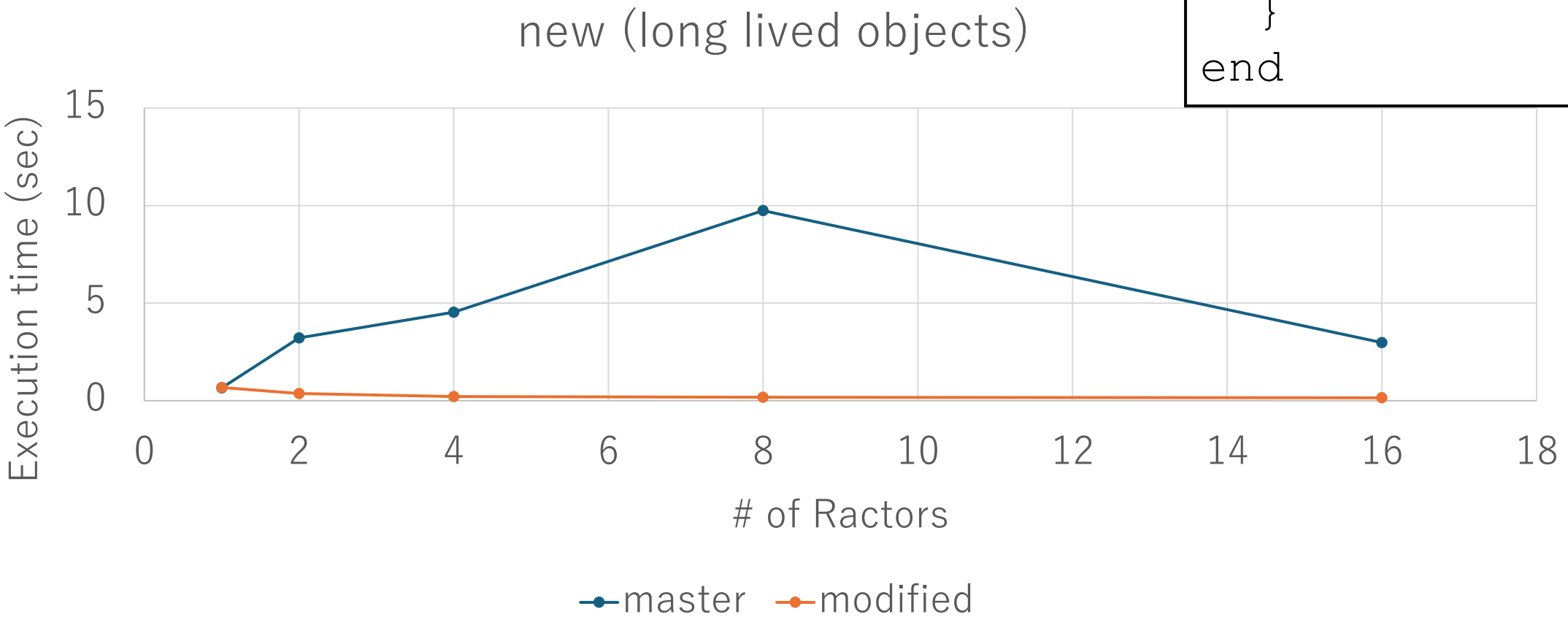
Short lived objects Speedup Ratio

```
def task
  TN.times{
    ''
  }
end
```



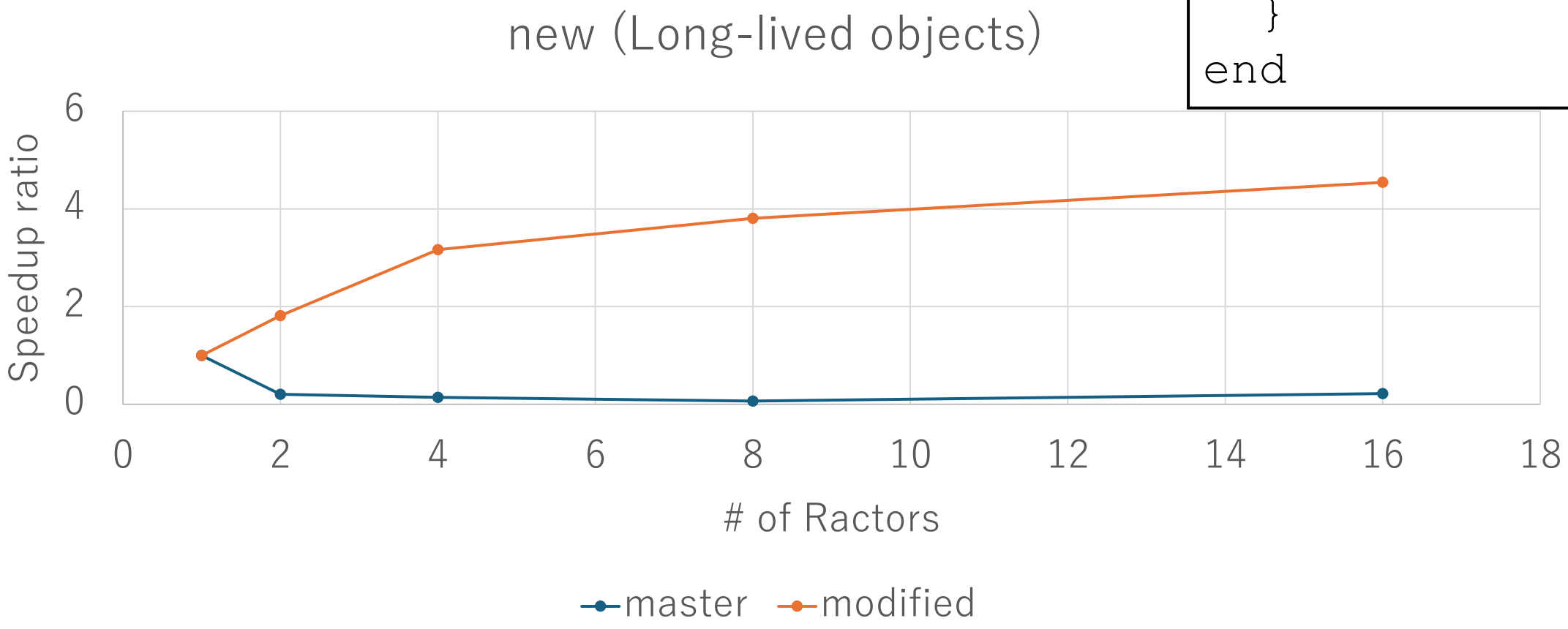
Long lived objects
Execution time (sec)

```
def task
  a = []
  TN.times{
    a = [a]
  }
end
```



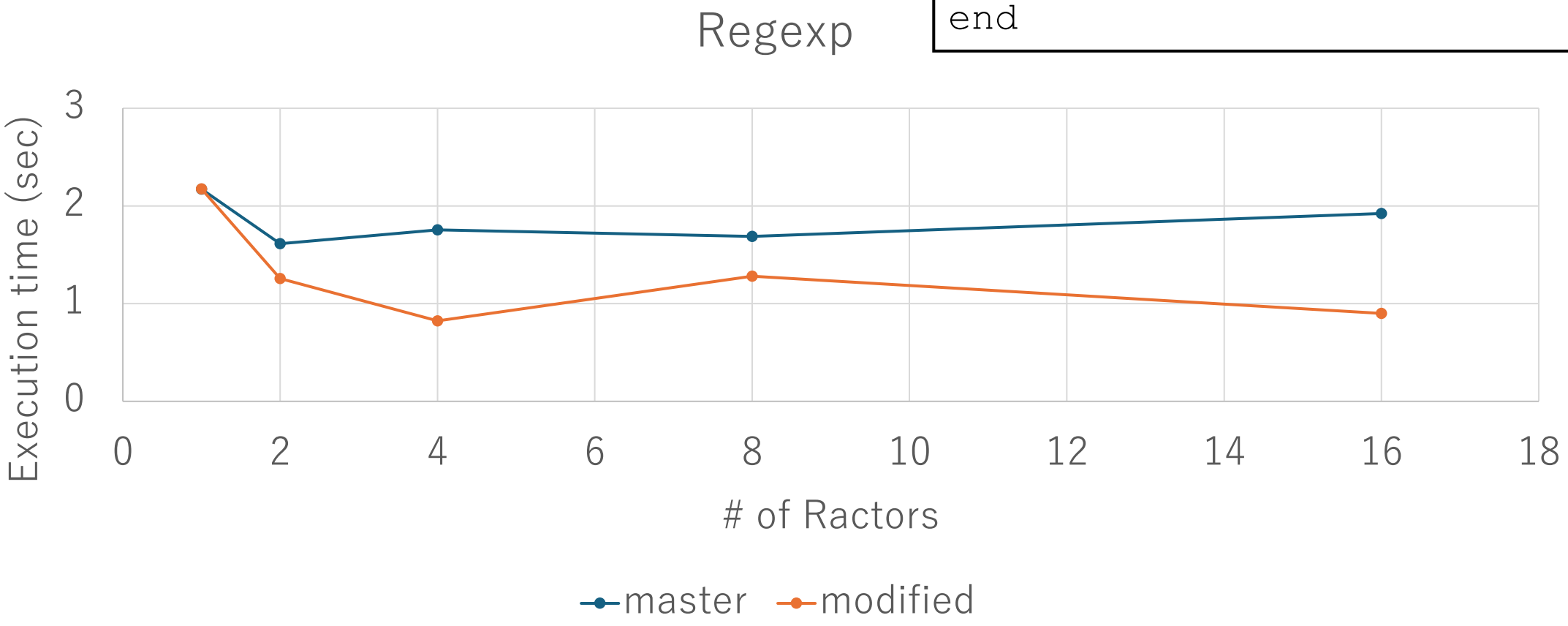
Long lived objects Speedup Ratio

```
def task
  a = []
  TN.times{
    a = [a]
  }
end
```



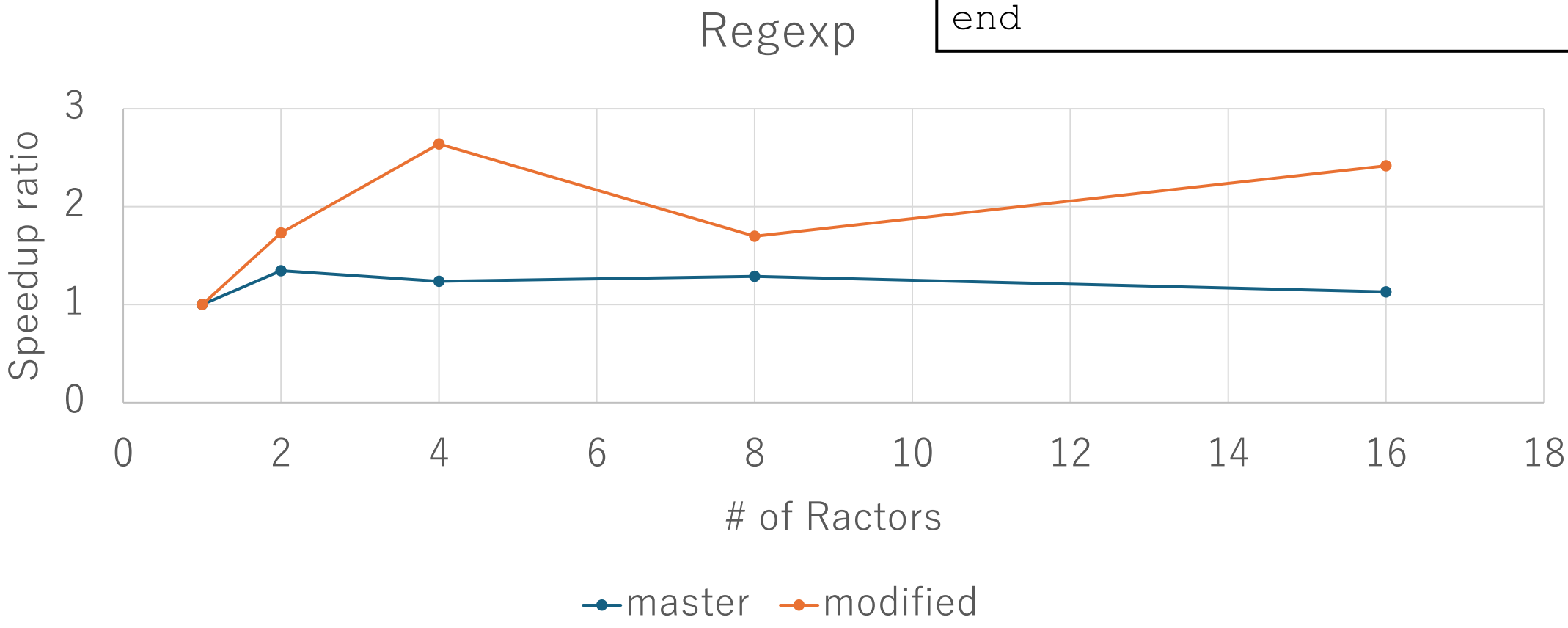
Regexp Execution time (sec)

```
def task
  str = 'hello world'
  TN.times{
    /(h)(e)(l)(l)(o)/ =~ str
  }
end
```



Regexp Speedup Ratio

```
def task
  str = 'hello world'
  TN.times{
    /(h)(e)(l)(l)(o)/ =~ str
  }
end
```



Wrap-up

Future work

- Complete implementation for Ruby 3.5 (Dec 2025)
  - Hopefully, only minor issues remain
  - Need to introduce a lightweight way to send an object
  - Need to reduce sharable objects like callcache
  - And more details chages…
- Parallel marking on the Global GC
  - Enables per-Ractor heap marking in parallel, improving global GC efficiency
- Try the ideal but difficult solution (1) for more performance
  - Rohit said he might have found a good algorithm
  - Let’s look forward to his future report

Summary

- Introducing Ractor local GC is promising but challenging
  - Cross-Ractor references make this difficult
- We propose a conservative approach to enable Ractor local GC by keeping shareable objects as root objects
  - Since the number of sharable objects is small enough, it is feasible
  - At least, the behavior is equivalent to the current implementation (Global GC every time)
- We can observe significant improvements in micro-benchmarks
- NOTE: This work is a collaboration with **Rohit Menon**

Questions and feedbacks are very welcome!

Thank you for listening!