

# Guild Prototype

Koichi Sasada

Cookpad Inc.  
<ko1@cookpad.com>



**cookpad**

RubyKaigi 2016 talk

# **A proposal of new concurrency model for Ruby 3**

Koichi Sasada  
ko1@heroku.com



RubyKaigi 2016 talk

# “Guild”

New concurrency model for Ruby 3

# TL;DR

- We can run **parallel programs** with Guilds
  - Current implementation is too preliminary and so buggy (easy to halt)
- **“Thread programming” is too hard for human beings.**

# Today's talk

- Background of Guild
- Demonstrations
- Guild specifications (ideas)
- Guild implementations (w/ future plan)
  - Synchronizations
  - Performance optimizations

# Koichi Sasada

<http://atdot.net/~ko1/>

- A programmer
  - 2006-2012 Faculty
  - 2012-2017 Heroku, Inc.
  - 2017- Cookpad Inc.
- Job: MRI development
  - Core parts
    - VM, Threads, GC, etc



**cookpad**

# Recent achievements for Ruby 2.6

- **Speedup `Proc#call`** ... **x1.4** improvements [Bug #10212].
- **Speedup `block.call`** where **`block`** is **passed** block parameter. [Feature #14330] (**x2.62**).

**Free with  
Ruby 2.6!**

# Advertisement

## Cookpad booth events

### 【Day 2】


15:20~15:50 Global Office Hours

### 【Day 3】

12:00~13:00 Q&A with @wyhaines

15:20~15:50 Ruby interpreter developme  
live by @ko1 & @mame

### **Cookpad X RubyKaigi 2018: Day 2 Party**

 June 1st, 19:30 - 21:30 (opens 19:00)

 Free (Registration required)

**Show up to this booth at 18:40 if you want to head with us!**



# cookpad



# Background of Guild

# Motivation

## Productivity (most important for Ruby)

- Thread programming is **too difficult**
- Making **correct/safe** concurrent programs easily

## Performance by Parallel execution

- Making parallel programs
- Threads can make concurrent programs, but can't run them in parallel on MRI (CRuby)
- People want to utilize Multi/many CPU cores

# RubyKaigi2016 Proposal

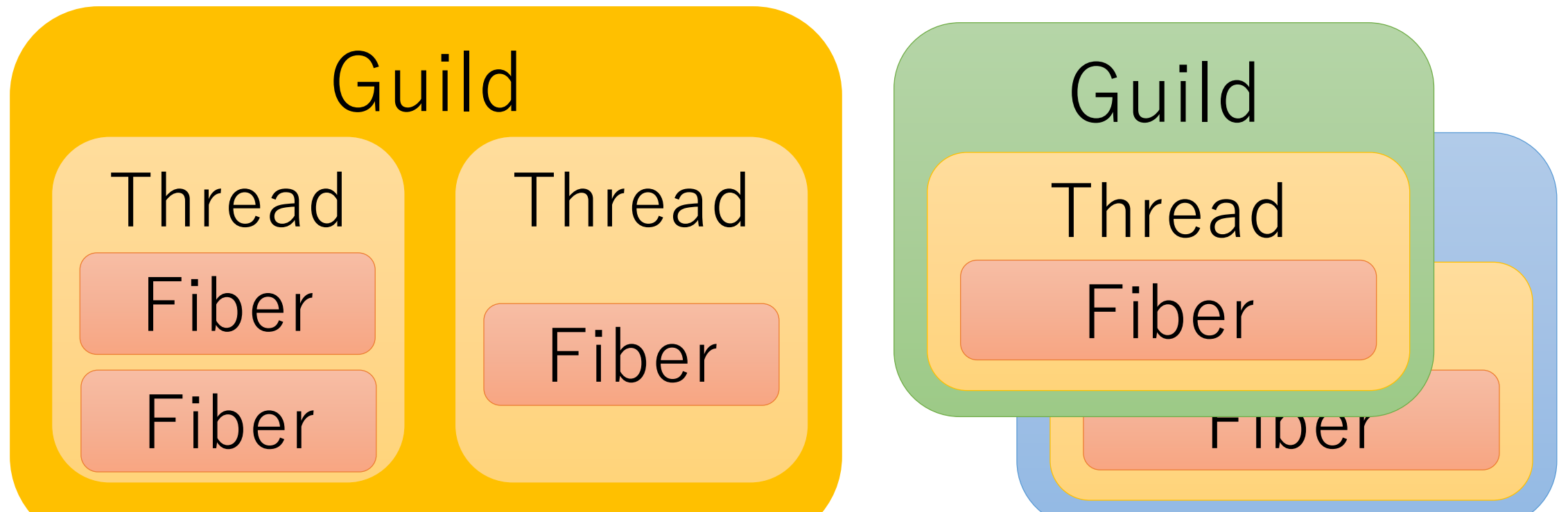
**Guild:** new concurrency abstraction for Ruby 3

- Idea: **DO NOT SHARE** mutable objects between Guilds  
→ No data races, no race conditions

*Replace Threads to Guilds*

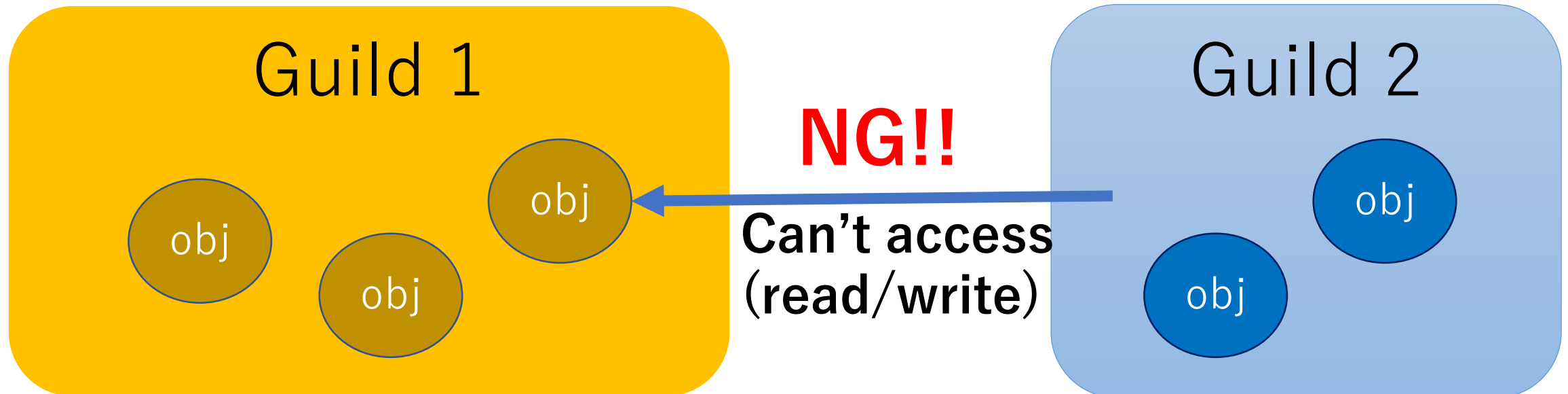
# Guilds, Threads and Fibers

- Guild has at least one thread (and a thread has at least one fiber)
- Threads in different Guilds can run in parallel



# Design “Shareable” and “non-sharable”

- **Non-shareable** objects
  - (normal) Mutable objects (String, Array, ...)
  - They are **members of only one Guild**
  - Using only 1 Guild, it compatible with Ruby 2



Demonstrations

# Demonstration (on 40 vCPUs)

- CPU 40 virtual CPUs (2 x 10 x 2)
  - x 2 CPUs
  - Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
    - x10 cores
    - x2 hyper threading
- Ubuntu 16.10
  - Already EOL ☹️

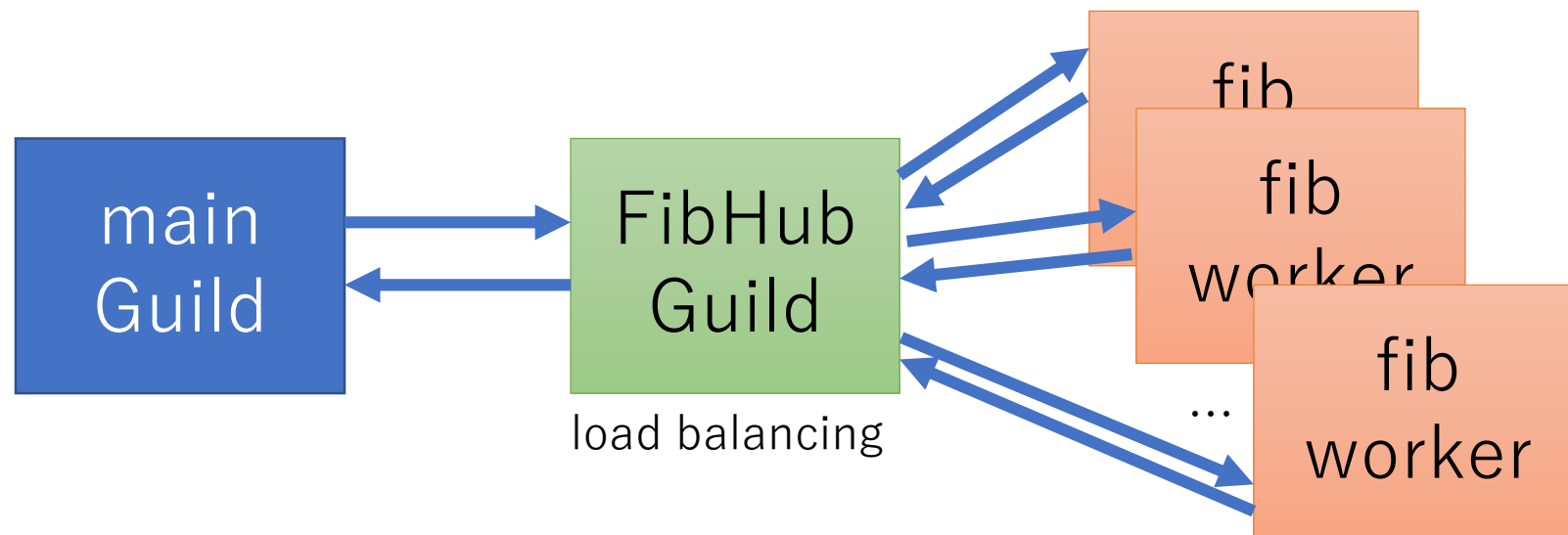
# Demonstration (on 40 vCPUs)

- Workload

- Calculate **fib(23)** x 100\_000 times

- Serial version: `100_000.times{ fib(23) }`

- Guild version:



We can change  
# of workers



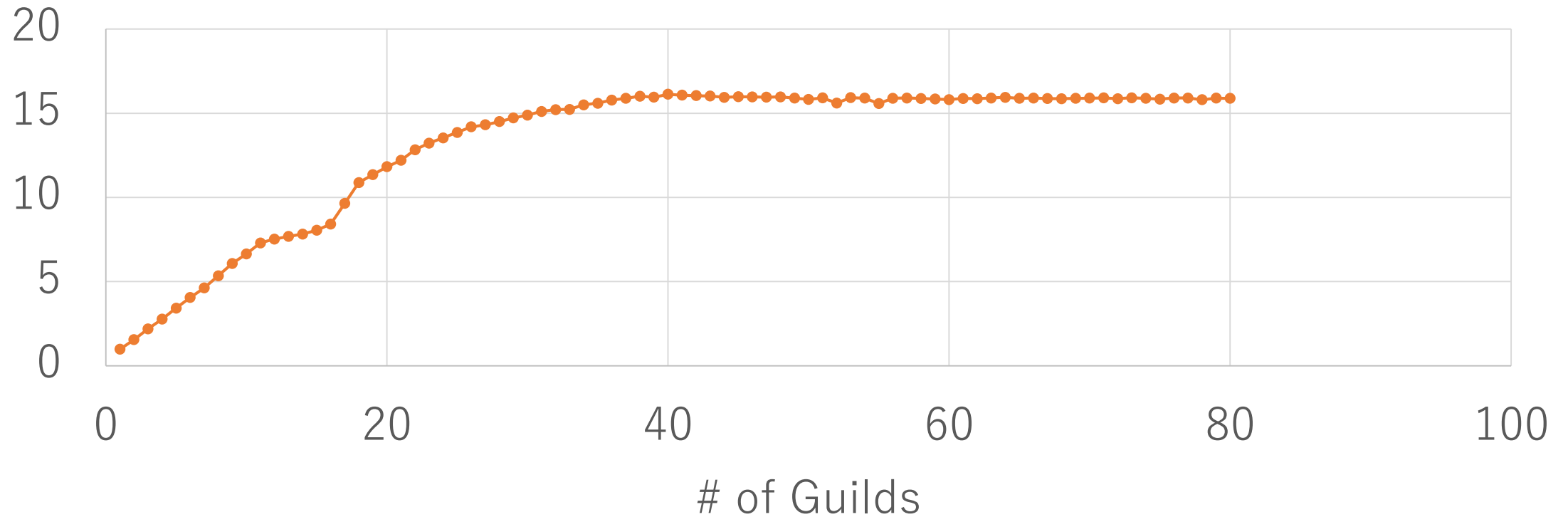
```
FIBHUB = make_worker_hub do |n|
  [n, fib(n)]
end
# library
def make_worker_hub n_workers = WN, &worker_proc
  pp WN: n_workers if $VERBOSE

  Guild.new(n_workers, worker_proc) do |nw, wp|
    guilds = nw.times.map do
      Guild.new do
        while data = Guild.receive
          result = wp.call(data)
          Guild.parent << [:ans, Guild.current, result]
        end
      end
    end
  end
  requests = []
```

```
while true
  cmd, sender_guild, data = *Guild.receive
  case cmd
  when :req
    # Receive a request from master
    if g = guilds.pop
      # Send a task
      g << data
      # if an idle worker is available
    else
      requests << data
    end
  when :ans
    # Receive an answers from workers
    Guild.parent << data
    # Send an answers to master
    if req = requests.pop
      sender_guild << req
      # Send a remaining task
      # to the worker if exists
    else
      guilds << sender_guild
    end
  end
end
```

**# You don't need to write such common code**  
**# but we provide some kind of a framework**

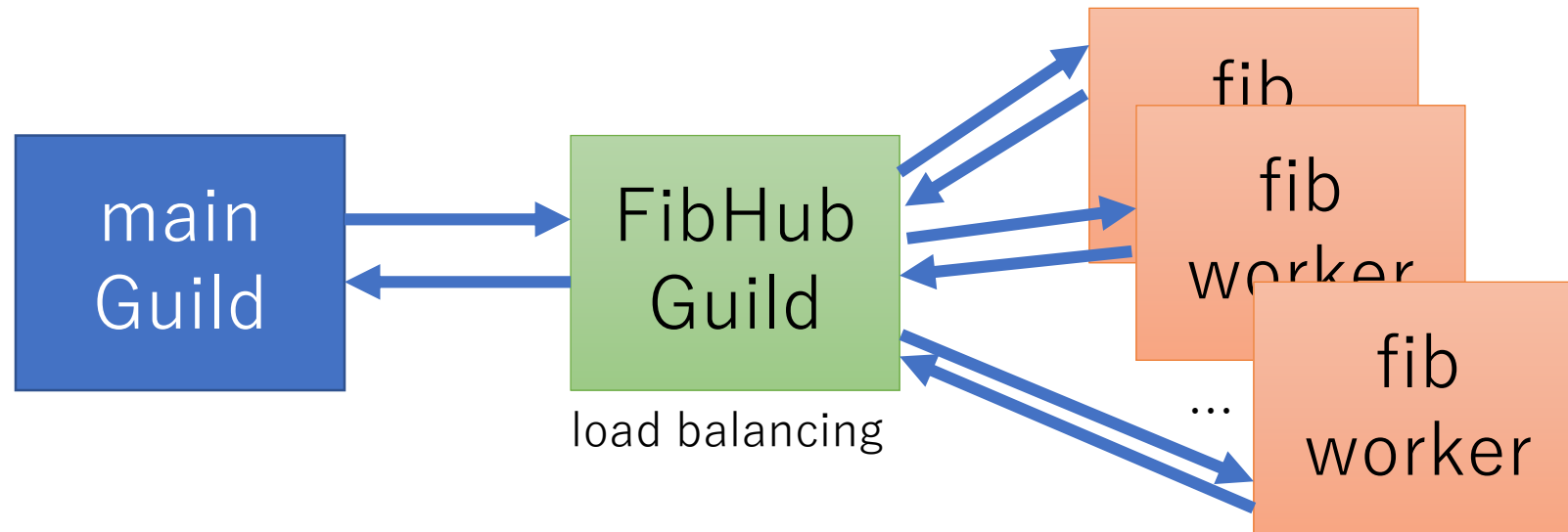
# fib(23) with # of Guilds on 40 vCPUs



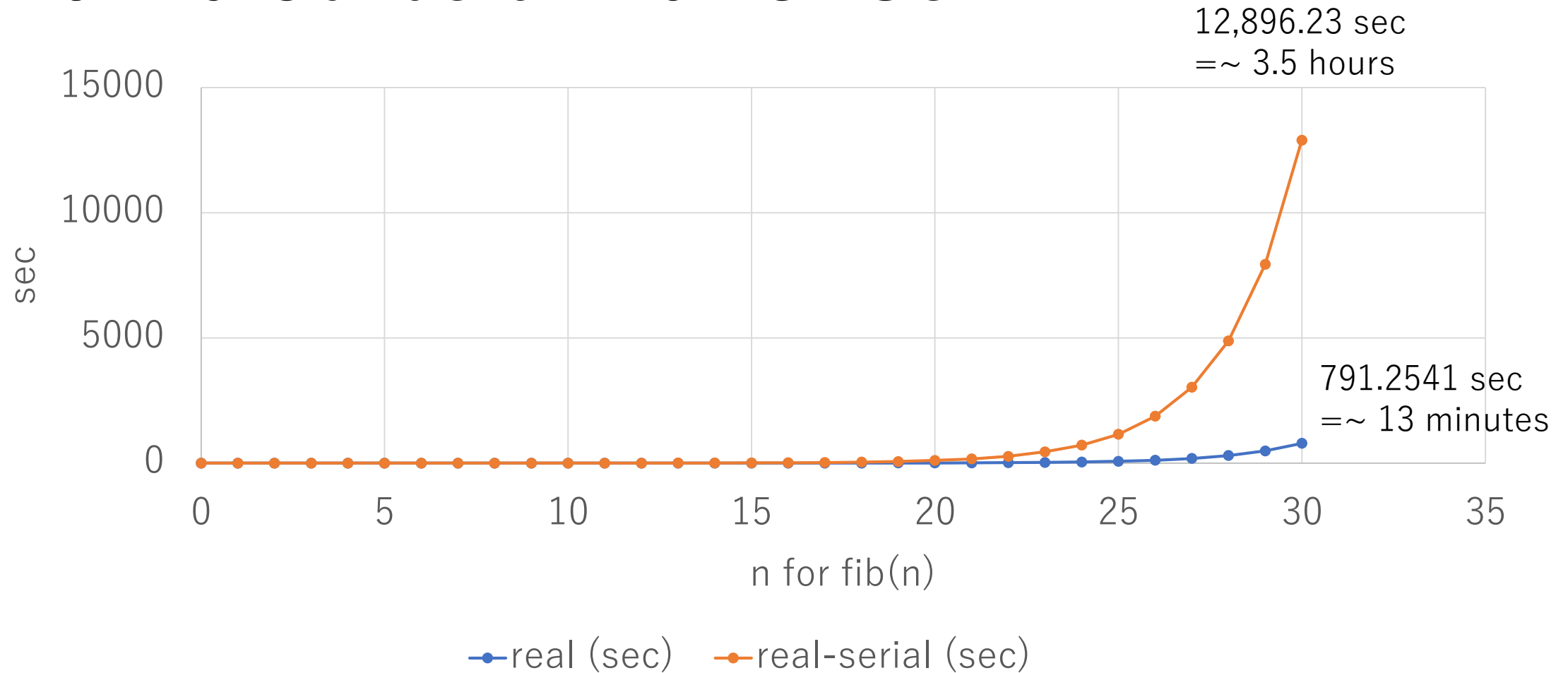
—•— Speedup ratio (compare with serial execution)

# Demonstration (on 40 vCPUs)

- Workload
  - Calculate **fib(n)** x 100\_000 times ( $0 \leq n \leq 30$ )
    - Serial version: 100\_000.times{ fib(23) }
    - Guild version: 40 Guilds

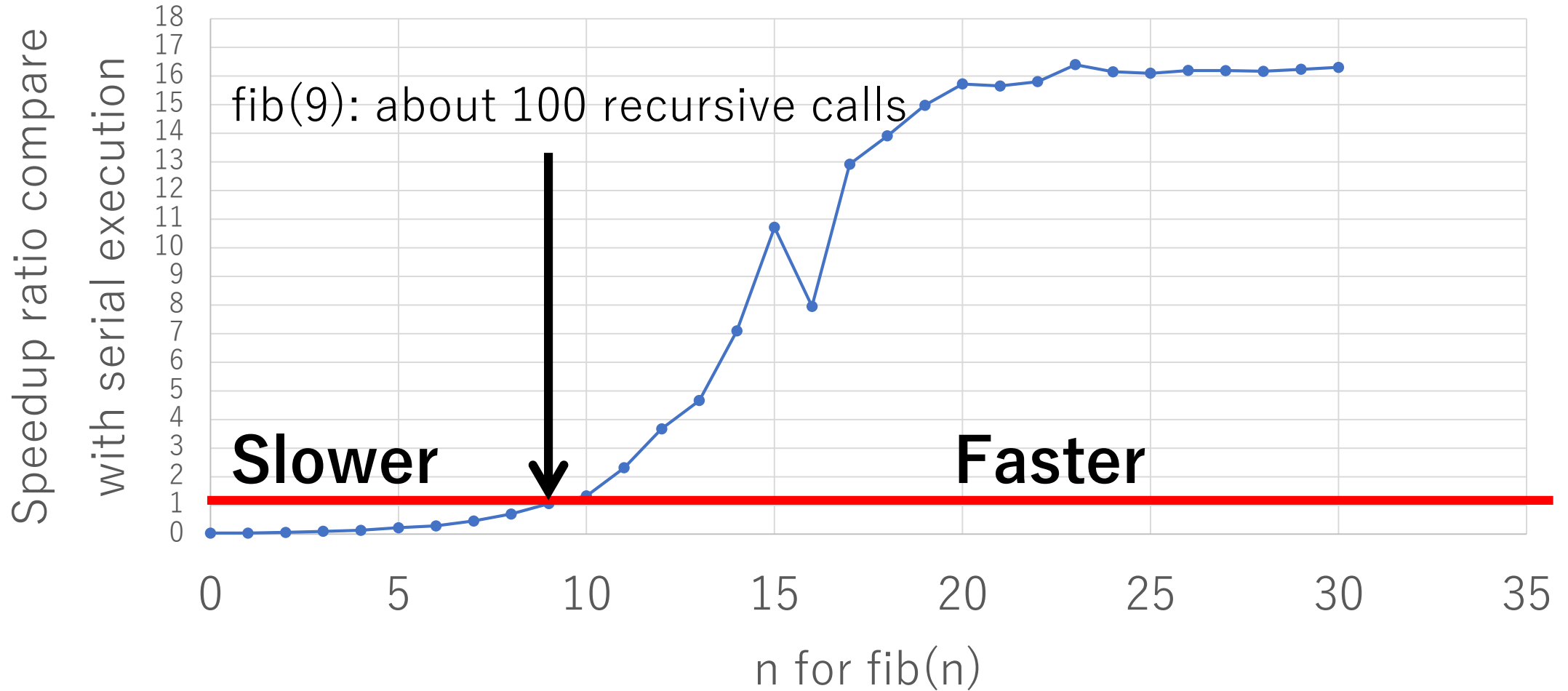


# Execution time (sec) of $\text{fib}(n) \times 100\_000$ with 40 Guilds on 40 vCPUs



**Order of  $\text{fib}(n)$  is " $O(2^n)$ "**

# fib(n) with 40 Guilds on 40 vCPUs



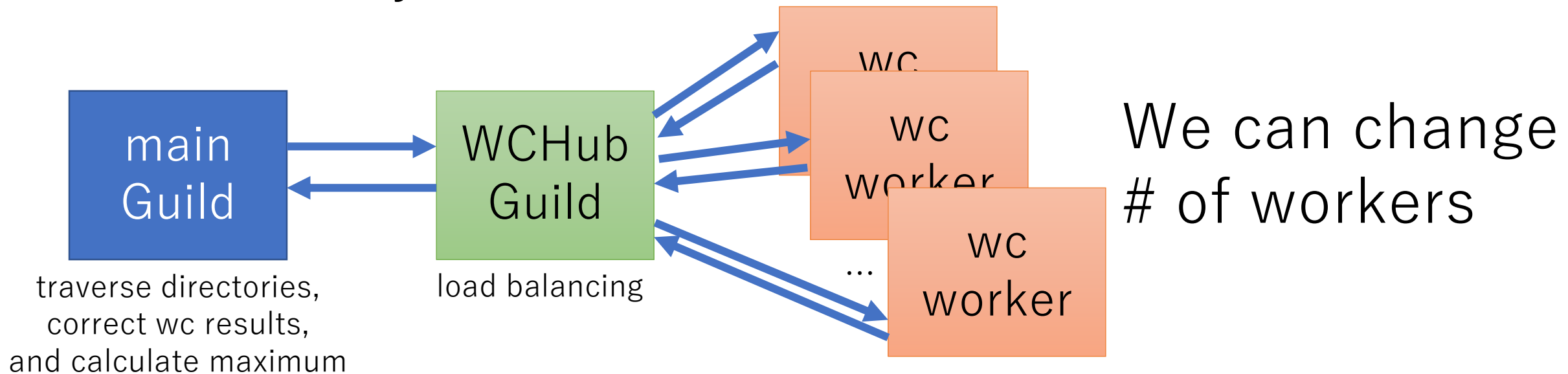
# Demonstration (on 40 virtual CPU)

- Workload
  - Calculate wordcount for files and find a file which contains maximum number of words.
    - on “ruby/test/\*\*/\*” files (1,108 files)

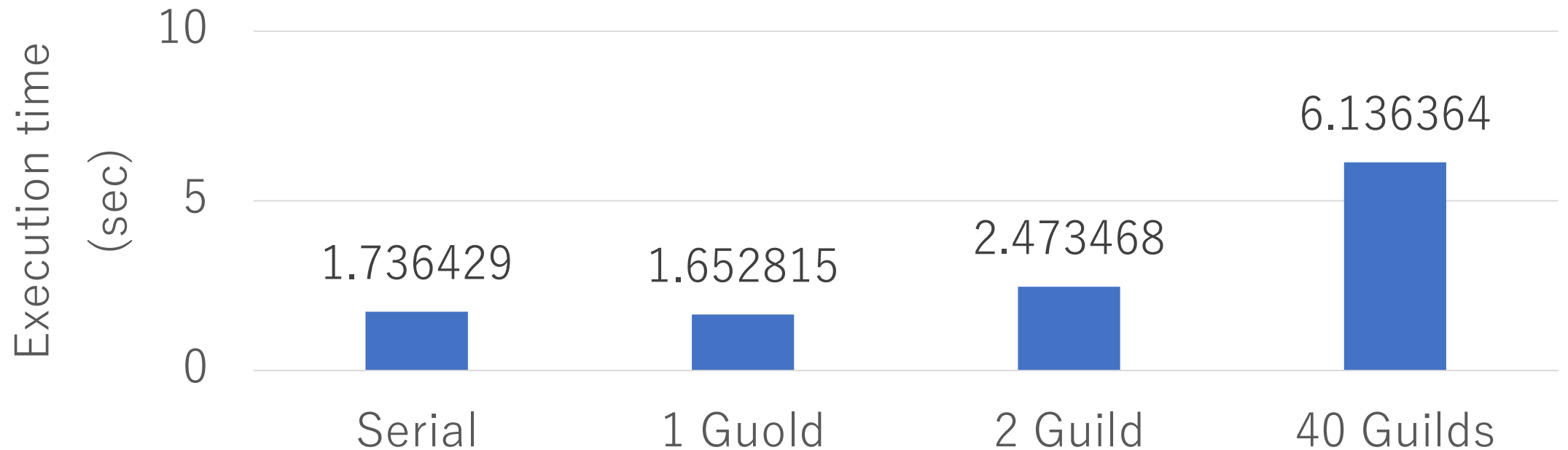
```
def word_count file
  r = File.read(file).b.upcase.split(/¥W/).uniq.size
end
```

# Demonstration (on 40 virtual CPU)

- Workload
  - Calculate wordcount for files and find a file which contains maximum number of words.
    - on “ruby/test/\*\*/\*” files (1,108 files)



# Demonstration (on 40 virtual CPU)



It is **SLOW** with multiple Guilds because GC/object allocation require naïve global locking (current implementation limitation) and huge contentions.

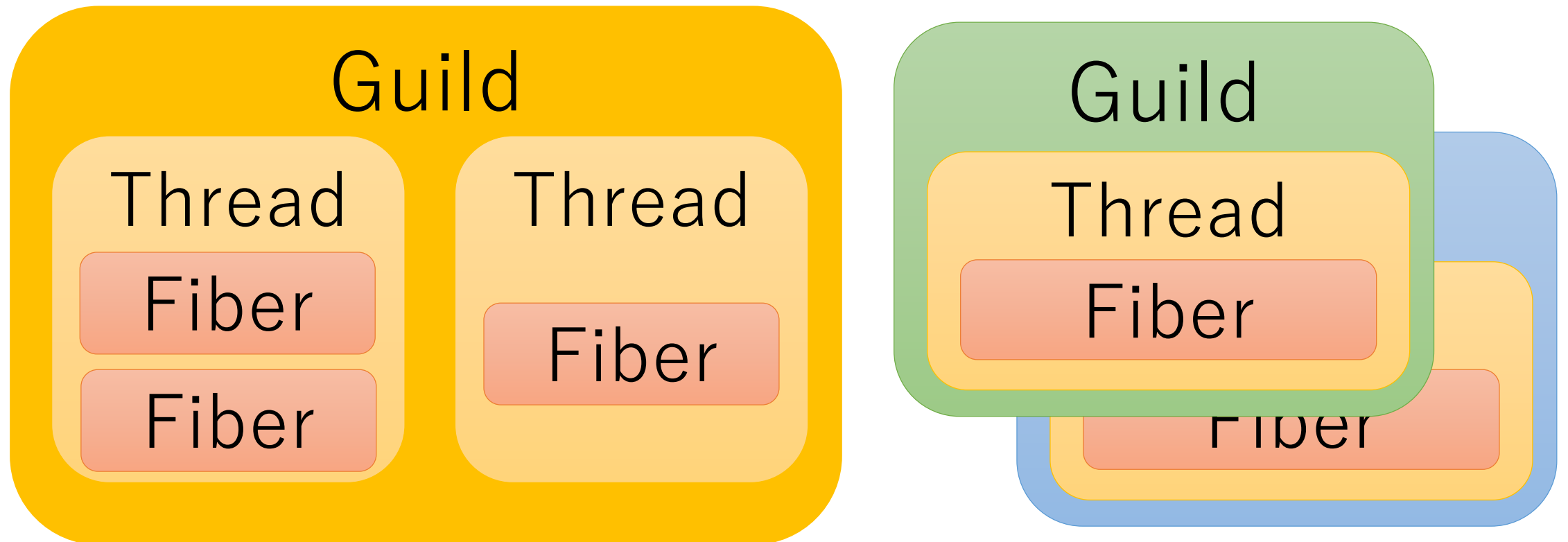


# Guild Specification

Temporary one.

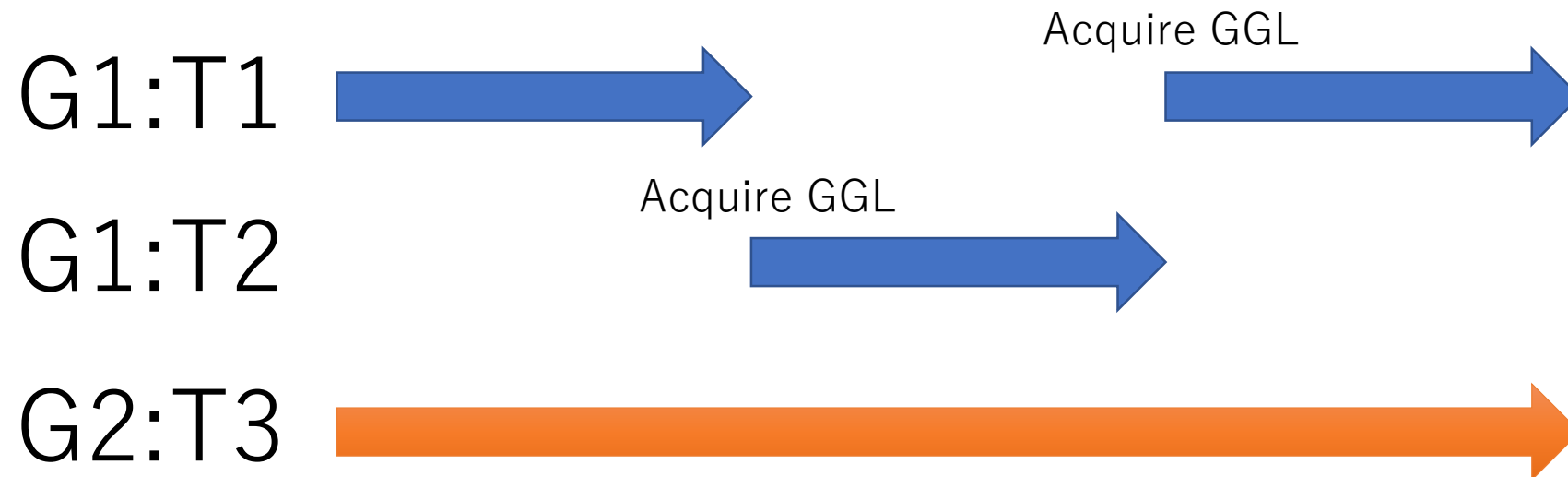
# Guilds, Threads and Fibers

- Guild has at least one thread (and a thread has at least one fiber)



# Threads in different guilds can run in PARALLEL

- Threads in different guilds can run in parallel
- Threads in a same guild can not run in parallel because of GVL (or GGL: Giant Guild Lock)



# Making Guilds

```
g1 = Guild.new do  
  expr1
```

```
end
```

```
g2 = Guild.new do  
  expr2
```

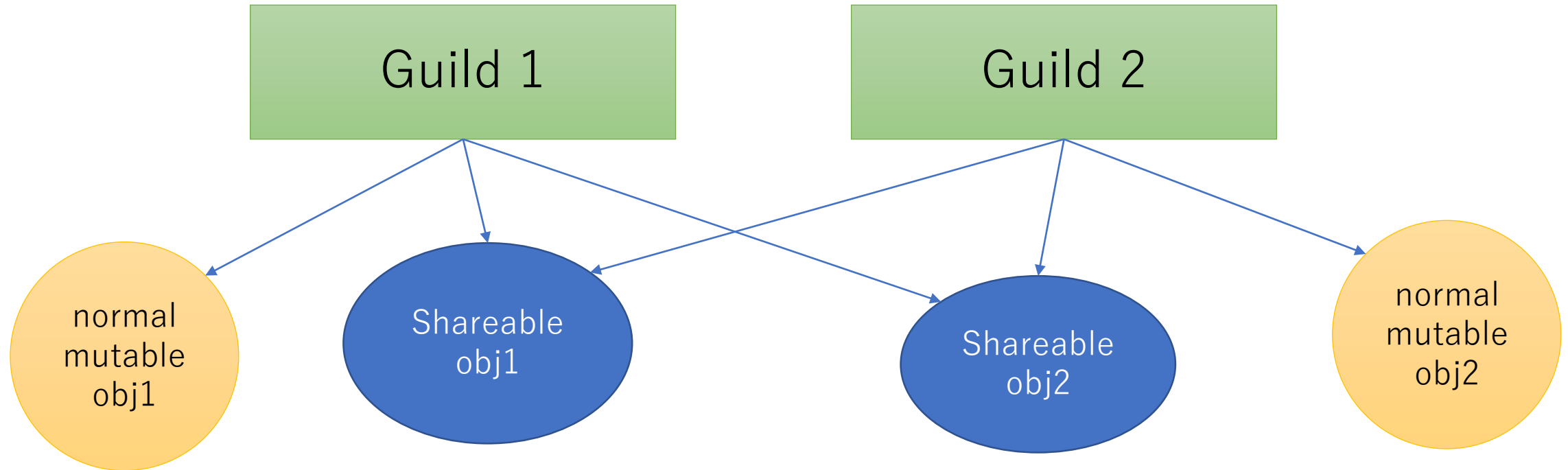
```
end
```

```
# Two new Guilds and Threads are created
```

```
# expr1 and expr2 can run in parallel
```

# Inter-Guild communication

## Share only “shareable” objects

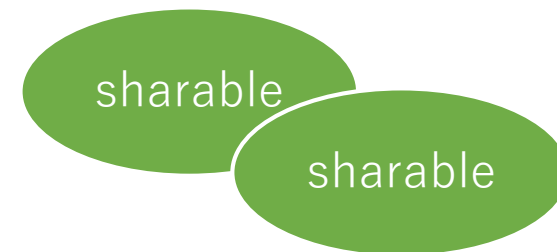
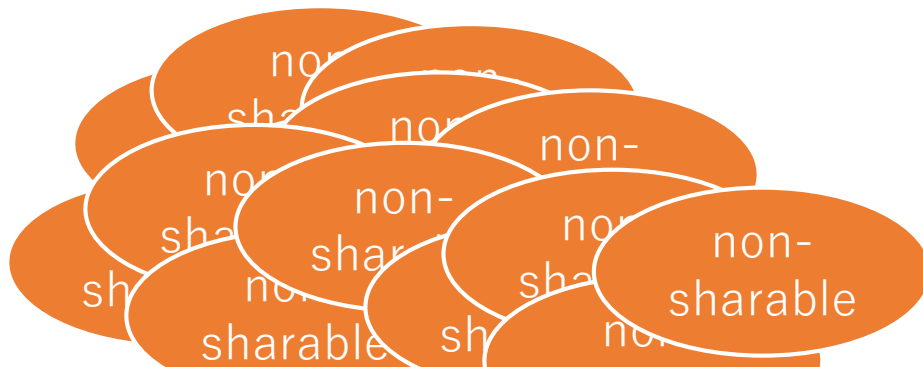


# Design “Shareable” and “non-sharable”

- You can enjoy usual mutating programming without any thread-safe concerns.
- Because we can't share mutable objects between Guilds. They are “non-sharable”.
- In other words, **you can't make thread-unsafe** (data-racy) programs with Guilds.

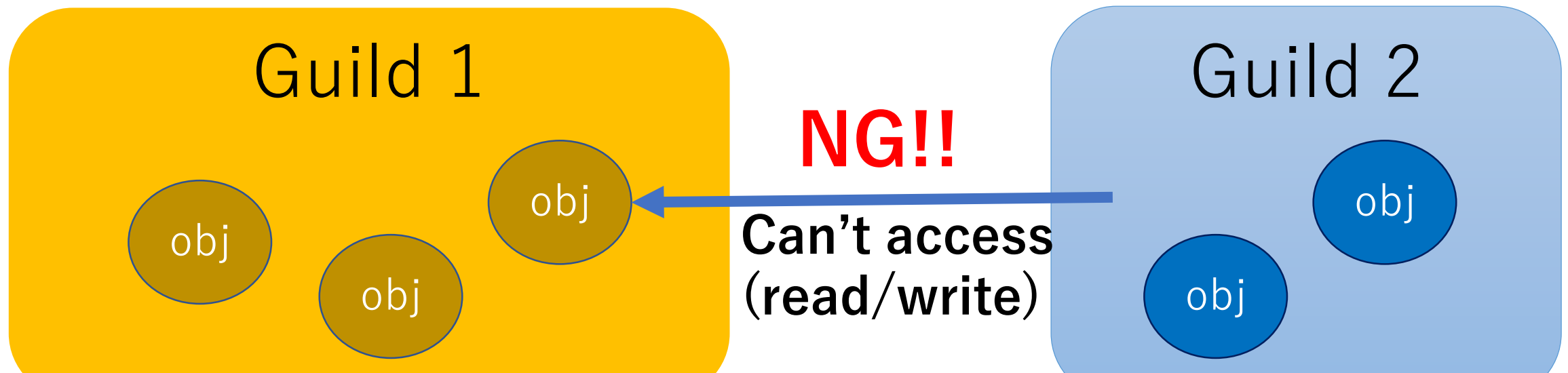
# Design “Shareable” and “non-sharable”

- You need to use “Sharable” objects to share objects between Guilds
- On concurrent programs, most of objects are not shared (thread-local)
  - **Tons** of local objects and **a few** sharing objects
  - You only need to care about a few objects



# Design “Shareable” and “non-sharable”

- **Non-shareable** objects
  - (normal) Mutable objects (String, Array, ...)
  - They are **member of only one Guild**
  - Using only 1 Guild, it compatible with Ruby 2





# Design “Shareable” and “non-sharable”

- **Shareable** objects

- (1) Immutable objects (Numeric, Symbol, ...)
- (2) Class/Module objects
- (3) Special mutable objects
- (4) Isolated Proc

# Shareable objects

## (1) Immutable objects

- **Immutable objects** can be shared with any guilds
  - Because no mutable operations for them
- “Immutable” != “Frozen”
  - `a1 = [1, 2, 3].freeze`: `a1` is Immutable
  - `a2 = [1, Object.new, 3].freeze`: `a2` is not Immutable
  - Maybe we will introduce deep freeze feature
- Numeric objects, symbols, true, false, nil are immutable (from Ruby 2.0, 2.1, 2.2)
- Frozen string objects are immutable (if they don't have instance variables)

# Shareable objects

## (2) Class/Module objects

- All objects (including any sharable objects) point to own classes
  - Good: Sharing class/module objects makes program easier
  - Bad: They can point to other mutable objects with Constants, @@class\_variable and @instance\_variables

```
class C
  Const = [1, 2, 3] # Const points a mutable
  array
end
# We will introduce special protocol for them
```

# Shareable objects

## (3) Special mutable objects

- Introduce shared/concurrent data structure
  - Shared hash, array, ...
  - Software transactional memory (from Clojure, ...), ...
  - Guild objects and so on
- They require special process to force synchronization explicitly
  - They can't mutate without synchronizations.
  - Easy to make correct concurrent programs
- Compared with normal Array, Hash, ... they should require special synchronization protocol to access

# Shareable objects

## (4) Isolated Proc

- Normal Proc can point to mutable objects with outer local variable (free-variables)

- `a = []; Proc.new{p a}.call`

- Introduce Isolated Proc (made by `Proc#isolate`) which is prohibited to access outer variables

- `a = []; Proc.new{p a}.isolate.call`  
`#=> RuntimeError (can't access a)`

- (there are more details but skip)

# Shareable objects

## (4) Isolated Proc

```
# Initial block for Guild is isolated
proc
  g1 = Guild.new do
    expr1 # Make isolated block and invoke
  end
  g2 = Guild.new do
    p g1 #=> RuntimeError (can't access
    "g1")
    # because block is isolated
  end
end
```

# Inter-Guild communication API

- **Actor model**, send/receive semantics
- Destination addresses are represented by Guild itself like Erlang/Elixir processes
- Sending shareable objects means sending only references to the objects (lightweight)
- Two methods to send non-shareable objects
  - (1) COPY
  - (2) MOVE

# Sending objects between Guilds

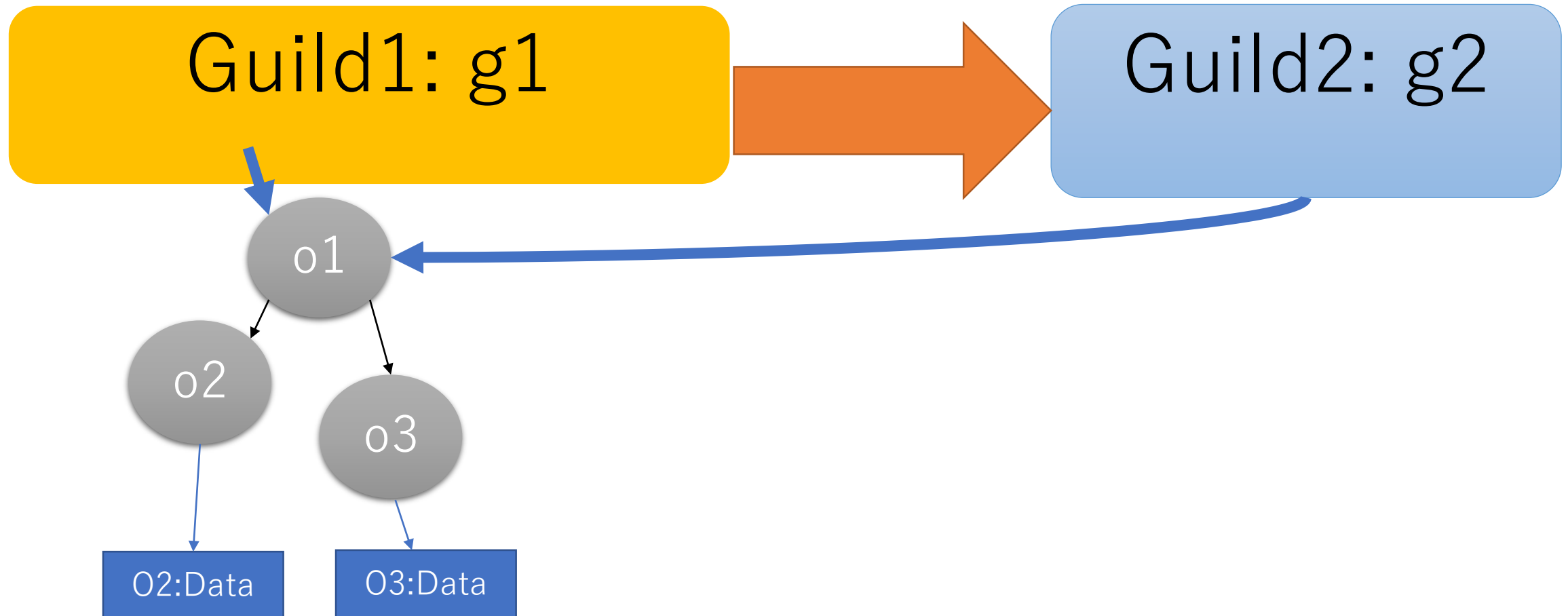
```
g1 = Guild.new do # create Isolated Proc
  n = Guild.receive
  r = fib(n)
  Guild.parent << r
end
g1 << 30
p Guild.receive #=> 1346269
```



# Sending shareable objects

**g2 << o1**

**o1 = Guild.receive**

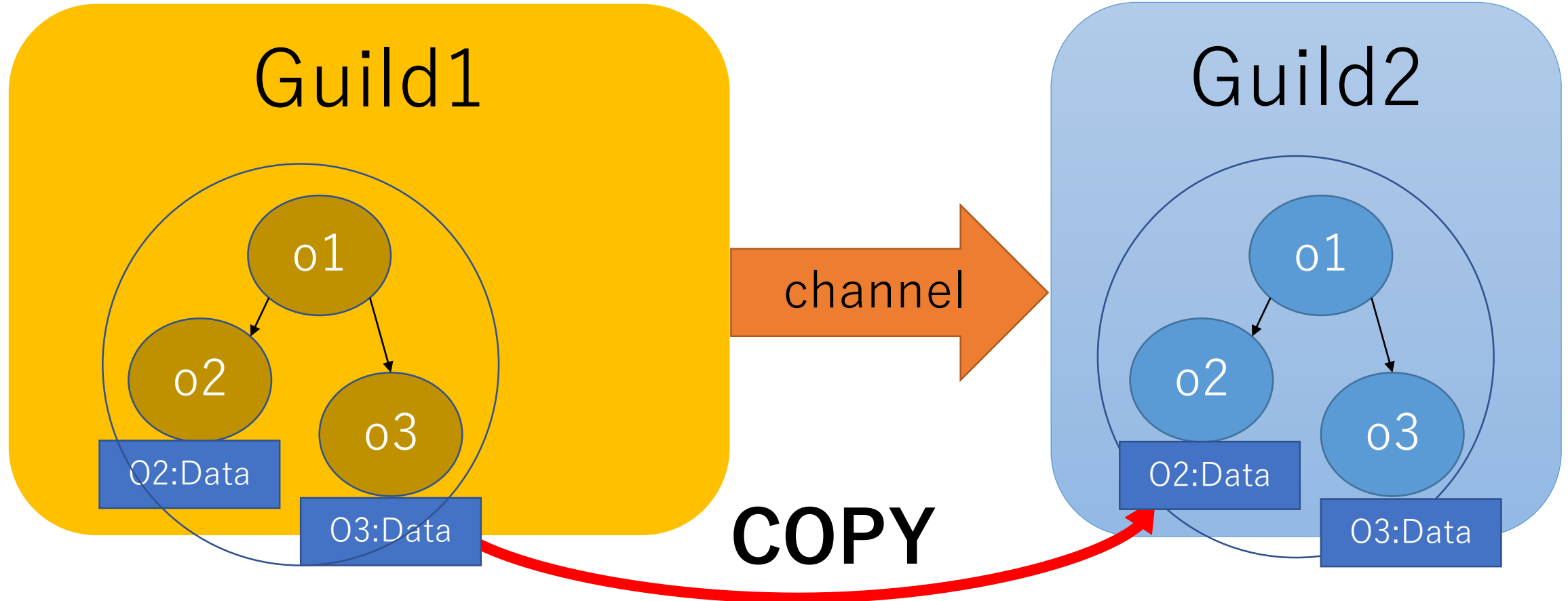


# Sending non-shareable objects

## (1) Send by **Copy**

**g2 << o1**

**o1 = Guild.receive**

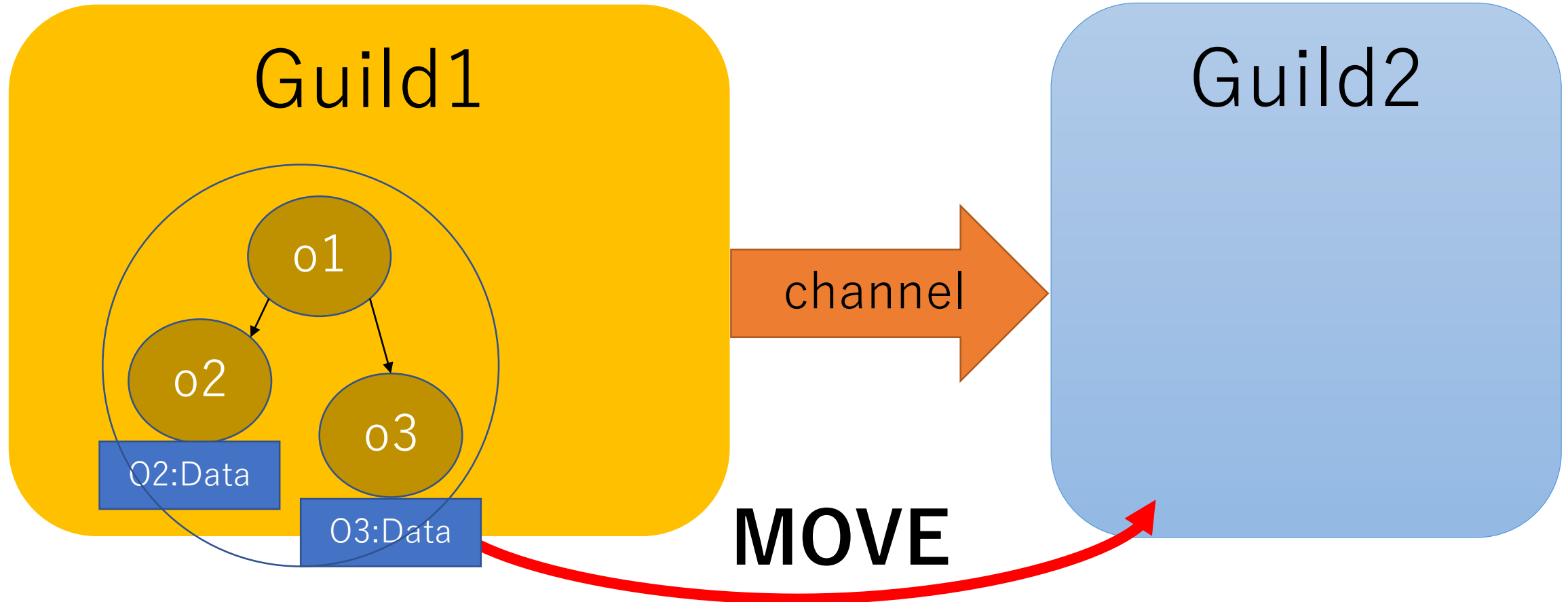


# Sending non-shareable objects

## (2) Send by Move

**g2.move(o1)**

**o1 = Guild.receive**

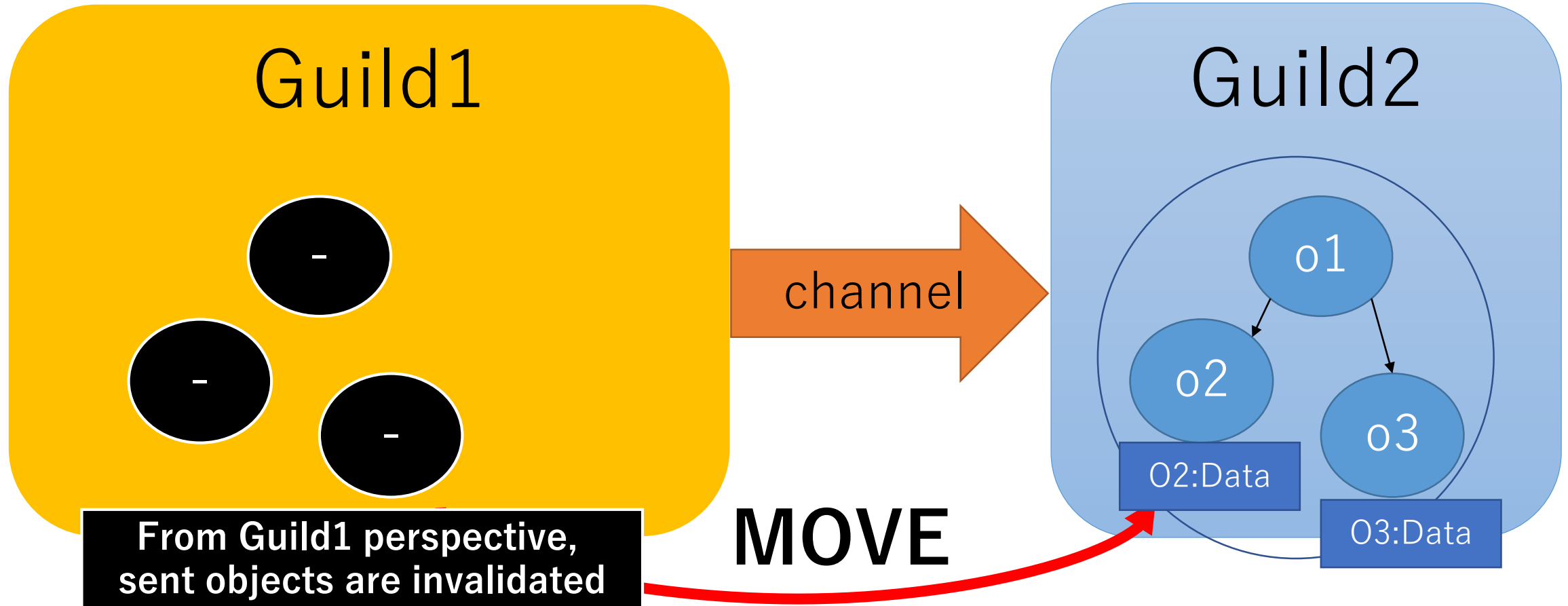


# Sending non-shareable objects

## (2) Send by Move

**g2.move(o1)**

**o1 = Guild.receive**



# Sending non-shareable objects

## (2) Send by **Move**

- If we don't access sent objects after sending them (and there are many such cases), we can send them faster
- Examples
  - Huge string data
  - I/O operation (send request I/O to workers)

# Summary of object sharing/non-sharing

- Shareable objects
  - Several types of shareable objects
  - We can share them between Guilds
  - They requires special synchronization protocol to mutate them
- Non-sharable objects
  - Normal mutable objects (like String, Array, ...)
  - Only one Guild can access such objects == membership
  - We can send them by COPY or MOVE
- Mutable objects are NOT shared accidentally as Thread programming → Correct concurrent Prog.

# Guild Implementation

Preliminary implementation includes many bugs, performance issues.

# Guild context

- Before Guild
  - VM -> \*Threads -> \*Fibers
- After Guild
  - VM -> \*Guilds -> \*Threads -> \*Fibers
  - Introduce `rb_guild_t`.

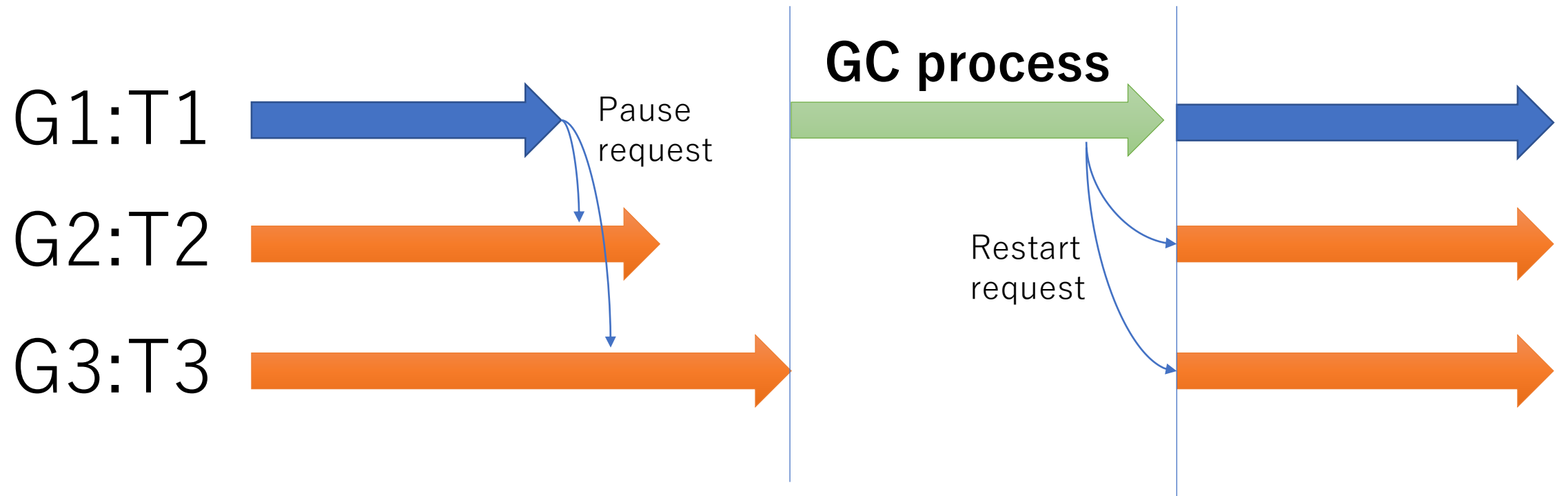


# Introduce synchronizations

- Before Guild
  - Multiple threads cannot run simultaneously
- After Guild
  - Run (native) threads in parallel
- Need to introduce so many synchronizations
  - Introduce VM-wide locks for VM-wide resources
  - It is the multi-thread programming!!

# Garbage collection

- Stop all Guilds (threads) at GC process



# Future work and optimizations

- Features
  - **Prohibit** sharing non-sharable objects
  - **Introduce synchronizations** to protect VM-wide resources (process-global)
  - **Introduce “sharable” object** protocols
- Performance
  - **Reduce synchronizations**
  - **Per Guild** Garbage collection
  - **Introduce new “C API” to reduce TLS access**

# Future optimizations

- Koichi Sasada, et.al. : An Implementation of Parallel Threads for YARV: Yet Another RubyVM (2007)
  - They introduced several optimization techniques to reduce synchronizations

## Ruby 用仮想マシン YARV における並列実行スレッドの実装

笹田 耕一<sup>1</sup> 松本 行弘<sup>2</sup>  
前田 敦司<sup>3</sup> 並木 美太郎<sup>4</sup>

本論文ではスクリプト言語 Ruby 用仮想マシン YARV: *Yet Another RubyVM* における並列実行スレッド処理機構の実装について述べる。Ruby はその使いやすさから世界中で広く利用されているプログラム言語である。Ruby の特徴のひとつにマルチスレッドプログラミングに対応しているという点があるが、現在広く利用されている Ruby 処理系は移植性を高めるため、すべてユーザレベルでスレッド制御を行っている。しかし、このスレッド実現手法では、実行がブロックしてしまう処理が C 言語レベルで記述できない、並列計算機において複数スレッドの並列実行による性能向上ができないなどの問題がある。そこで、現在筆者らが開発中の Ruby 処理系 YARV において、OS やライブラリなどによって提供されるネイティブスレッドを利用するスレッド処理機構を実装し、複数スレッドの並列実行を実現した。並列化にあたっては、適切な同期の追加が必要であるが、特に並列実行を考慮しない C 言語で記述した Ruby 用拡張ライブラリを安全に実行するための仕組みが必要であった。また、同期の回数を減らす工夫についても検討した。本論文では、これらの仕組みと実装についての詳細を述べ、スレッドの並列実行によって得られた性能向上について評価した結果を述べる。

## An Implementation of Parallel Threads for YARV: Yet Another RubyVM

KOICHI SASADA<sup>1</sup>, YUKIHIRO MATSUMOTO<sup>2</sup>, ATSUSHI MAEDA<sup>3</sup>  
and MITARO NAMIKI<sup>4</sup>

In this paper, we describe an implementation of parallel threads for YARV: Yet Another RubyVM. The Ruby language is used worldwide because of its ease of use. Ruby also supports multi-threaded programming. The current Ruby interpreter controls all threads only in user-level to achieve high portability. However, this user-level implementation can not support blocking task and can not improve performance on parallel computers. To solve these problems, we implement parallel threads using native threads provided by systems software on YARV: *Yet Another RubyVM* what we are developing as another Ruby interpreter. To achieve parallel execution, correct synchronizations are needed. Especially, C extension libraries for Ruby which are implemented without consideration about parallel execution need a particular scheme for running in parallel. And we also try to reduce a number of times of synchronization. In this paper, we show implementations of these schemes and results of performance improvement on parallel threads execution.

One more thing...

Naming of “Guild”

# Why “Guild”?

- Prefix should be different from “P” (Process), “T” (Thread) and “F” (Fiber).
- Ownership can be explained with the word “Membership”.
  - All (normal) objects belong to one Guild.
  - Easy to explain “Move” semantics

# Any problem?

- “Move” operation is not so popular operation (most of case “copy” is enough)
- No other languages use this terminology
- Naming is important
- Just now “Guild” is a code name of this project

# Today's talk

- Background of Guild
- Demonstrations
- Guild specifications (ideas)
- Guild implementations (w/ future plan)
  - Synchronizations
  - Performance optimizations



Thank you for your attention

# Guild Prototype

Koichi Sasada

Cookpad Inc.

<ko1@cookpad.com>



**cookpad**

# Pros./Cons. Matrix

	Process	Guild	Thread	Auto-Fiber	Fiber
Available	Yes	<b>No</b>	Yes	<b>No</b>	Yes
Switch on time	Yes	Yes	Yes	<b>No</b>	<b>No</b>
Switch on I/O	Auto	Auto	Auto	<b>Auto</b>	No
Next target	Auto	Auto	Auto	Auto	<b>Specify</b>
Parallel run	<b>Yes</b>	<b>Yes</b>	No (on MRI)	No	No
Shared data	<b>N/A</b>	<b>(mostly) N/A</b>	Everything	Everything	Everything
Comm.	Hard	<b>Maybe Easy</b>	<b>Easy</b>	<b>Easy</b>	<b>Easy</b>
Programming difficulty	Hard	<b>Easy</b>	Difficult	<b>Easy</b>	<b>Easy</b>
Debugging difficulty	Easy?	<b>Maybe Easy</b>	Hard	Maybe hard	<b>Easy</b>