Parallel programming in Ruby3 with Guild

Koichi Sasada
Cookpad Inc.
<ko1@cookpad.com>
Today’s talk

• Ruby 2.6 updates of mine
• Introduction of Guild
  • Design
  • Discussion
  • Implementation
  • Preliminary demonstration
Koichi Sasada
http://atdot.net/~ko1/

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

is a father of the youngest attendee of Rails Girls Tokyo 10th @Cookpad Tokyo office
My 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).

• **Introduce Transient heap** [Bug #14858]
Transient heap

• Manage heap for young memories
  • vs. malloc heap
  • malloc() / free() is heavy operation and introduce memory fragmentation issue and theap solves it.
• Using Generational copy GC algorithm w/ MRI specific hack

• Array, Object (user defined class), Struct and small Hash objects use theap now
  • Support String is desired, but too difficult
Transient heap
Array creation (loop{Array.new(n)})

Array with 0..3 elements don’t use the heap
Transient heap
Small hash creation (loop\{h = \{\ldots\}\})

Speedup ratio (upper is better)

# of Hash elements

>8 Hash objects don't use theap

x1.5

x1.0
Ruby 2.6: Transient heap

Summary

• Transient heap is new memory hack
  • Generational Copy GC technique
  • MRI specific hack to keep compatibility

• Your application can improve performance on Ruby 2.6
  • Microbenchmarks show good improvements.
    • x1.5 - x2.0 faster for creation & collection
  • Unfortunately, discourse rails benchmark doesn’t show clear perf. improvements
Parallel programming in Ruby3 with Guild

Koichi Sasada

Cookpad Inc.
<ko1@cookpad.com>
TL;DR

- Guild is new concurrent abstraction to force no-sharing mutable objs for Ruby 3
- Guild specification is not fixed yet
- Guild implementation is not mature (PoC)
- Your comments are highly welcome!!
Background of Guild
Motivation

Productivity (most important for Ruby)

• Thread programming is too difficult because sharing mutable objects
• Correct/safe concurrent programs easily is important

Performance by Parallel execution

• Utilizing Multi/many CPU cores is important for performance
RubyKaigi2016 (and RubyConf 2016) 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*
Design of Guild

Not fixed yet.
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)

G1:T1

Acquire GGL

G1:T2

Acquire GGL

G2:T3

Acquire GGL
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 on Guilds.
Design “shareable” and “non-sharable”

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 sharable objects
Design “shareable” and “non-shareable”

- **Non-shareable** objects == most of objects
  - Most of mutable objects (String, Array, …)
  - They are **member of only one Guild**
  - If you use only 1 Guild, it compatible with Ruby 2

![Diagram showing two guilds with objects that cannot be accessed from another guild](image)
Design “Shareable” and “non-sharable”

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

- **Important invariant**
  - Sharable objects only refer to sharable objects
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

- Example of immutable objects
  - Numeric objects, symbols, true, false, nil are immutable
  - 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:
    • Easy Implementation and good communication performance
    • 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 protocol** 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)
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
FYI: Other languages using similar ideas

• Similar to Guild
  • Racket: Place (imm. or special mut. values)
  • Kotlin/Native: Worker (check ownership)
• Almost isolated
  • Shell script: Process (copy byte stream)
  • JavaScript: Worker
• Everything immutable
  • Erlang, Elxir: Process
Inter-Guild communication API

• **Actor model**, send/receive semantics
  • **Not fixed yet** (discuss later)
• 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

```ruby
# create Isolated Proc
n = Guild.receive
r = fib(n)
Guild.parent << r

# or
g1.send(30)

p Guild.receive #=> 1346269
```
Sending shareable objects

$$g2 \ll o1$$

Guild1: g1

$$o1 = \text{Guild.receive}$$

Guild2: g2

O2: Data

O3: Data
Sending non-shareable objects
(1) Send by Copy

g2 <-- o1

Guild1

Guild2

o1 = Guild.receive

COPY

channel

o1

o2

o3

O2:Data

O3:Data

O2:Data

O3:Data

COPY
Sending non-shareable objects

(2) Send by **Move**

$g2.move(o1)$

$o1 = Guild.receive$

**Guild1**

$o1$

$o2$

$o3$

**Guild2**

$o2:Data$

$o3:Data$
Sending non-shareable objects

(2) Send by **Move**

\[ g2.move(o1) \]

From Guild1 perspective, sent objects are invalidated

\[ o1 = \text{Guild.receive} \]
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 objects (send request I/O to workers)
Summary of sharable/non-sharable objects with copy/move operations

• 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

• Shareable objects
  • Several types of shareable objects
  • They require special synchronization protocol to mutate them
  • We can share them between Guilds by sending references

Mutable objs are NOT shared accidentally as Thread
→ Safe concurrent programming
Discussion:
How to represent communication channel?

• Actor model
  • Destination is specified by a Guild
  • `guild << obj`
  • Erlang/Elixir, …

• CSP model
  • Destination is specified by a channel
  • `ch << obj`
  • Go, JavaScript, Kotlin/native, Racket, …

• They have advantages and disadvantages…
Retrieve multiple channels

• Sometimes we need to manipulate with multiple channel
  • Data channel and control channel
  • Monitoring channel for child Guilds
• How to provide APIs to support it?
Go language goroutine and channels

# https://tour.golang.org/concurrency/5

```go
select {
case c <- x:
    x, y = y, x+y
case <-quit:
    fmt.Println("quit")
return
}
```
Erlang/Elixir Process

# https://elixir-lang.org/getting-started/processes.html

iex> receive do
  ...
  ...> {:hello, msg} -> msg
  ...
  ...> {:world, msg} -> "won't match"
  ...
  ...> end
JavaScript Worker and MessageChannel

var channel = new MessageChannel();
var output = document.querySelector('.output');
var iframe = document.querySelector('iframe');

// Wait for the iframe to load
iframe.addEventListener("load", onLoad);

function onLoad() {
    // Listen for messages on port1
    channel.port1.onmessage = onMessage;

    // Transfer port2 to the iframe
    iframe.contentWindow.postMessage('Hello from the main page!', '*', [channel.port2]);
}

// Handle messages received on port1
function onMessage(e) {
    output.innerHTML = e.data;
}
Racket place

; https://docs.racket-lang.org/reference/sync.html#%28def._%28%28quote._~23~25kernel%29._handle(evt)%29%29
> (define msg-ch (make-channel))
> (define exit-ch (make-channel))
> (thread
  (λ ()
    (let loop ([val 0])
      (printf "val = ~a~n" val)
      (sync (handle-evt
               msg-ch
               (λ (val) (loop val)))
             (handle-evt
              exit-ch
              (λ (val) (displayln val))))))))
Multiple channels for Actor model

• Support “Tag” (shows channel)
  • `guild.send_to(:data, obj)`
  • `guild.send(obj)` send to the default tag

• Receive with multiple tag
  • `Guild.receive(tag1, tag2, ...) { |tag, obj| ... }`

```ruby
g2 = Guild.new{
  cont = true
  while cont
    Guild.receive(:data, :ctrl){ |tag, obj| ... } case tag
    when :data
      calc(obj)
    when :ctrl
      case obj
        when :exit
          cont = false
        else
          raise "unknown"
      end
    end
  end
}
```
Multiple channels for CSP model

• Making channels explicitly
• Send to a channel
  • `ch << obj`
• Receive with multiple channels
  • `Guild::Channel.receive(ch1, ch2, ...) { |ch, obj| ... }

```ruby
g2 = Guild.new(data_ch, cntl_c){ |d_ch, c_ch|
  cont = true
  while cont
    # wait for multiple channel
    Guild::Channel.receive(d_ch, c_ch){ |ch, obj|
      case ch
      when d_ch
        calc(obj)
      when c_ch
        case obj
        when :exit
          cont = false
        else
          raise "unknown ctrl: #{obj}"
        end
      end
    end
  end
}
```
Guild Implementation

Preliminary implementation includes many bugs, performance issues.

https://github.com/ko1/ruby/tree/guild
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 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

![Diagram showing Garbage Collection process]
Implementation is not completed

• Features
  • Fix GC bug
  • **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

• They introduced several optimization techniques to reduce synchronizations
Naming of “Guild”
Why “Guild”? 

• Prefix should be different from “P” (Process), “T” (Therad) 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
Demonstrations on the current PoW implementation.
Demonstration (on 40 vCPUs)

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

• Workload
  • Calculate \( \text{fib}(23) \times 100\_000 \) times
    • Serial version: \( 100\_000.\text{times}\{ \text{fib}(23) \} \)
    • 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
    requests = []
  end
end

while true
  cmd, sender_guild, data = *Guild.receive
  case cmd
  when :req
    if g = guilds.pop
      g << data
    else
      requests << data
    end
  when :ans
    Guild.parent << data
    if req = requests.pop
      sender_guild << req
    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

# of Guilds

Speedup ratio (compare with serial execution)
Demonstration (on 40 vCPUs)

• Workload
  • Calculate \( \text{fib}(n) \times 100\_000 \) times \( 0 \leq n \leq 30 \)
    • Serial version: \( 100\_000.\text{times}\{ \text{fib}(23) \} \)
    • Guild version: 40 Guilds
Execution time (sec) of $\text{fib}(n) \times 100\,000$ with 40 Guilds on 40 vCPUs

- **real (sec)**: 12,896.23 sec
  - $\approx 3.5$ hours
- **real-serial (sec)**: 791.2541 sec
  - $\approx 13$ minutes
fib(n) with 40 Guilds on 40 vCPUs

Speedup ratio compare with serial execution

fib(9): about 100 recursive calls

Slower

Faster
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)

```ruby
def word_count file
  r = File.read(file).b.upcase.spli...
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)

We can change # of workers
It is **SLOW** with multiple Guilds because GC/object allocation require naïve global locking (current implementation limitation) and huge contentions.
Today’s talk

• Ruby 2.6 updates of mine
• Introduction of Guild
  • Design
  • Discussion
  • Implementation
  • Preliminary demonstration
Thank you for your attention

Parallel programming
in Ruby3
with Guild

Koichi Sasada
Cookpad Inc.
<ko1@cookpad.com>
<table>
<thead>
<tr>
<th>Available</th>
<th>Process</th>
<th>Guild</th>
<th>Thread</th>
<th>Auto-Fiber</th>
<th>Fiber</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
<td>Specify</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No (on MRI)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>N/A</td>
<td>(mostly) N/A</td>
<td>Everything</td>
<td>Everything</td>
<td>N/A</td>
</tr>
<tr>
<td>Hard</td>
<td>Maybe Easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Easy</td>
<td>Easy</td>
<td>Difficult</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Easy?</td>
<td>Maybe Easy</td>
<td>Hard</td>
<td>Maybe hard</td>
<td>Easy</td>
<td></td>
</tr>
</tbody>
</table>