Ractor を用いた Ruby の並列処理性能評価と Rubocop による並列コード記述支援

成蹊大学理工学部情報科学科　ソフトウェア研究室　S202148　柳澤快

1. 研究背景

Ruby は 1995 年に一般公開されたオブジェクト指向スクリプト言語である。Ruby は多くの支持を集めているが、こうした成長は Web アプリケーションフレームワーク Ruby on Rails の人気に起因している。Ruby on Rails は、開発工数を抑えられ、比較的短期間で目的のサービスやアプリ開発の実現を最大の特徴としていることから、Ruby は多くの企業で採用されている。

また、近年、コンピュータの性能はマルチコアプロセッサの普及により向上している。それに伴い、ソフトウェアの並列処理能力が重要となっている。並列計算機上で複数の処理を同時に実行するために、多くのプログラミング言語では複数スレッドを同時実行させることができる。しかし、Ruby のスレッドは GIL(Global Interpreter Lock)によって同時に実行可能なスレッドは 1 つのみとなっている。これにより、スレッドを使う限り、通常の方法では並列プログラムを Ruby で記述することはできない[1]。

Ractor の登場により異なる Ractor 間でスレッドを並列に実行することが可能になった。Ractor は開発途上であるものの積極的に改善が進行中であり、将来的には Ruby on Rails のアプリケーションサーバにおいて、リクエストを並列に処理することで更なるパフォーマンス向上などが期待されている。本研究では、このような将来性を見据えて、Ractor のより良い記述方法を明らかにし、静的解析ツール Rubocop による Ractor の記述を支援することを目指す。

2. Ractor の概要

本章では、並行・並列処理を可能にする Actor Model の抽象化である Ractor について説明する。

2.1. Ractor とは

Ractor とは、2020 年にリリースされた Ruby3.0 で導入された並行・並列処理を可能にする機構であり、スレッドセーフな並列実行を提供する Ruby の Actor Model の抽象化である。

Actor Model は 1973 年に米国マサチューセッツ工科大学の Carl Hewitt 氏によって発表された並列計算の数学的モデルの一種である。Actor Model の構成を図 1 に示す。

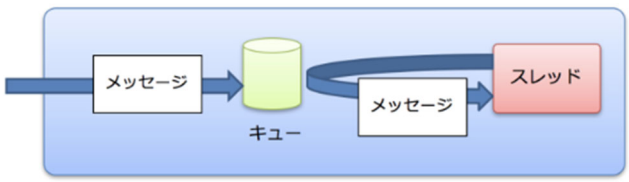

図 1：Actor Model の構成**エラー！参照元が見つかりません。**

Ruby 処理系を起動すると、1 つの Ractor を作成する(これを Main Ractor と呼ぶ)。同一 Ractor は最低 1 つのスレッドを持つ。同一 Ractor 内の複数スレッドはグローバルインタープリターロックにより同時には実行されない。

3. 実験

バブルソートを用いて Ractor の実験を行った。実験は Apple M1 チップ搭載の MacBook Air(8 コア CPU、macOS Sonoma 14.5)上の Ruby3.3.4 で実行させた。

3.1. 方法

バブルソートの並列化の方法については図 2 に示す。整数の乱数を指定の数だけ生成し、配列に格納する。

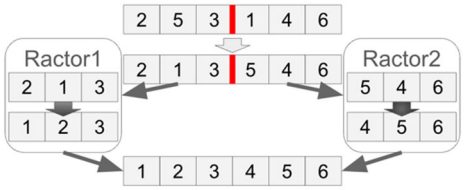

図 2：Ractor を用いたバブルソートの並列化

配列要素の中央値を基準に次の手順に従い 2 つに分割する。配列の左端から右に向かって、中央値以上の値を探し、右端から左に向かって中央値未満の値を探索する。見つけた 2 つの値を交換し、これにより、中央値未満の値が左側に、中央値以上の値が右側に移動する。この操作を左右からの探索が衝突するまで繰り返す。2 つの Ractor を生成し、各 Ractor に分割された部分配列を送信する。その後、各 Ractor で昇順にバブルソートを実施する。各 Ractor からソートされた配列を受け取り、それらを結合する。

筆者は、C 言語と MPI を用いてバブルソートの並列化を行った経験があるため、Ractor でもこれを用いることにした。

3.2. 結果

実験の結果、C 言語と MPI を用いて、2 並列でバブルソートさせた時、逐次実行から約 4 倍の性能向上だったが、Ractor では最も良い場合で約 2.8 倍の性能向上となった。メモリ再配置の頻発や共通の配列要素アクセス時の排他制御などの原因が考えられ、これらの Ractor のオーバーヘッドは現状ユーザー側の記述の工夫だけで避けるのは難しいことがわかった。

これらの課題があるものの、背景でも述べた通り、Ractor は今後も改善が進められ、将来的な発展が期待される。そこで Ractor の記述を支援し、より扱いやすくするために Rubocop の活用を提案する。

4. 静的解析ツール Rubocop とは

静的解析ツールには Rubocop を用いる。Rubocop とは、Ruby の静的コード解析およびコードフォーマッターの機能を持つライブラリである。コード内で発見された問題を報告するだけでなく、Rubocop はそれらの多くを自動的に修正することも可能である[3]。

複数人で開発を行っていても書き方を統一することが可能であり、コードの品質向上に貢献することができるという利点があることから、Ruby を採用している企業では、Rubocop の導入事例は多く存在する。

5. Rubocop によるコード支援

ここでは、実際に実装した指摘の 1 例を記載する。Ractor 間の通信では、Receive 処理に対応する Send 処理が存在しない場合、Receive 処理で無限に待機状態となる。この問題のコードを図 3 に示す。

```
r = Ractor.new do
  data = Ractor.receive  # 受信先がない
end
```

図 3：問題のあるコード例

この問題のコードに対して、Receive 処理がある場

合、対応する Send 処理が存在するかをチェックする指摘について Rubocop を用いて実装した。

実装したカスタムルールについて図 4 のサンプルプログラムを用いて、動作を示す。

```
source > send_receive.rb
1  r = Ractor.new do↓
2    msg = Ractor.receive↓
3    msg↓
4  end↓
5  ↓
6  r.send 'ok'↓
7  ↓
8  p r.take # 'ok'が出力↓
```

図 4：Ractor Send/Receive サンプルコード

使用するサンプルコードについて説明する。最初に、新しい Ractor を生成する。この Ractor の中で、Receive 処理を使ってメッセージを受け取り、受け取ったメッセージをそのまま返す。次に、Send 処理を実行して、Ractor に文字列'ok'というメッセージを送信する。最後に、take メソッドを使って Ractor から返された文字列'ok'を受け取り、標準出力に返す。サンプルプログラムの実行結果を図 5 に示す。

```
$ ruby source/send_receive.rb
"ok"
```

図 5：source/send_receive.rb の実行結果

Send 処理で Ractor に送信した文字列’ok’が標準出力されていることが確認できる。この状態で CLI 上で rubocop ./source/send_receive.rb を実行した結果を図 6 に示す。

```
Inspecting 1 file
.

1 file inspected, no offenses detected
```

図 6：rubocop ./source/send_receive.rb の出力

Receive 処理に対応する Send 処理が存在するため、Rubocop での警告は出力されない。

次に、サンプルプログラム内の Send 処理(r.send ‘ok’)をコメントアウトして実行する。しかし、実行すると、強制終了しない限り、Ractor 内の Receive 処理が送信されるのを無限に待機している状態になる。Rubocop を実行すると、Receive 処理に対応する Send 処理が存在しないため、実装したカスタムルールの警告が出力される。この時の Rubocop 実行結果を図 7 に示す。

```
$ rubocop ./source/send_receive.rb
Inspecting 1 file
C

Offenses:

source/send_receive.rb:1:1: C: [Correctable] Style/RactorSendReceive: Ractor.re
ceive detected in the Ractor block but no corresponding r.send found.
r = Ractor.new do ...
^^^^^^^^^^^^^^^^

1 file inspected, 1 offense detected, 1 offense autoco
```

図 7：Send 処理がない場合の rubocop 実行結果

また、VSCode であれば、作成したカスタムルールを読み込み、コード上に警告が出力され、Rubocop を手動で実行する手間を省くことが可能である。この様子を図 8 に示す。

```
source > send_receive.rb
1  r = Ractor.new do↓
Ractor.receive detected in the Ractor block but no
corresponding `r`.send found.
(convention:Style/RactorSendReceive)
問題の表示 (⌥F8)   利用できるクイックフィックスはありません
7  ↓
8  p r.take # 'ok'が出力↓
```

図 8： コード上での Rubocop 警告

さらに、実装したカスタムルールを自動修正機能に対応させた。実際に自動修正機能を実行してみると Ractor の処理の後に Send 処理が追加される。自動修正機能を使用する場合は rubocop コマンドを-A オプション付きで実行する。開発者は、自動で挿入された Send 処理に引数を記述するのみでよくなる。実際に、自動修正機能を実行した結果を図 9 に示す。6 行目に Send 処理が自動で追加され、図下部の出力では自動修正機能を実行し、修正されたことが出力されている。

```
source > send_receive.rb
1  r = Ractor.new do↓
2    msg = Ractor.receive↓
3    msg↓
4  end↓
5  ↓
6  r.send()↓
7  ↓
8  # r.send 'ok'↓
9  ↓
10 p r.take # 'ok'が出力↓

Offenses:

source/send_receive.rb:1:1: C: [Corre
cted] Style/RactorSendReceive: Ractor
.receive detected in the Ractor block
 but no corresponding r.send found.
r = Ractor.new do ...
^^^^^^^^^^^^^^^^

1 file inspected, 1 offense detected,
 1 offense corrected

yanagisawakai at kym in ~/college_res
earch (main●●)
```

図 9： 自動修正機能の実行

他にも Rubocop による Ractor の記述に対するカスタムルールの実装を行なった。これらのカスタムルールの実装により、開発者がデバッグに費やす時間を短縮することができる。また、Ractor の使用に不慣れな開発者でも、制約を自然に学びながらコードを書くことが可能となる。

6. まとめ

本研究では、Ruby の並列処理機構 Ractor を性能評価し、その利点と課題を明らかにした。実験では、メモリ再配置や排他制御の影響による性能低下も確認され、これらの課題はあるものの、Ractor は今後も改善が進められ、将来的な発展が期待される。そこで Ractor の記述を支援し、より扱いやすくするために Rubocop の活用を提案した。Ractor は並列処理の安全性と効率性を両立し、GIL の制約を超えた並列実行を実現するが、新たなプログラミングモデルの理解が求められるため、学習コストや記述ミスが課題となる。これに対応するため、Rubocop に Ractor 専用のカスタムルールを実装し、潜在的なエラーの検出や自動修正を可能にすることで、デバッグ時間の削減や効率的なコード記述を支援した。本研究で実装したルールは、初心者にも有用であり、Send/Receive の対応関係などの検証機能や自動修正機能を通じて、Ractor 特有の仕様を学びながら利用できる環境を整備した。一方で、さらなるルールの追加や解析速度の最適化が課題として残り、特に大規模コードに対する効率的な解析が求められる。本研究は、Ractor をより扱いやすくするものであり、Rubocop の拡張を通じて Ractor 以外の Ruby プログラムへの支援も期待される。

参考文献

[1] 笹田耕一：「Ruby 向け並列化機構 Guild の試作」. 情報処理学会プログラミング研究会. 2018

[2] “「アクターモデル」による並列処理プログラミング入門”. SIOS Tech Lab. https://tech-lab.sios.jp/archives/8738, (参照 2025-2-2)

[3] “rubocop/robocop”. GitHub. https://github.com/rubocop/rubocop, (参照 2025-2-2)