2024年度　卒業論文

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

Parallel Processing Performance Evaluation of Ruby Using Ractor and Parallel Code Writing Support Using Rubocop

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

目次

第1章 序論 1

1.1. はじめに 1

第2章 Ractorの概要 3

2.1. Ractorについて 3

2.2. Actor model 3

2.3. Ractorの仕様 5

2.3.1. インタープリタプロセス内のRactor 5

2.3.2. Ractorにおけるオブジェクト共有 5

2.3.2.1. 不変オブジェクト（Immutable Objects） 5

2.3.2.2. クラス・モジュールオブジェクト（Class/Module Objects） 6

2.3.2.3. 特別な共有可能オブジェクト（Special Shareable Objects） 6

2.3.3. Ractor間の通信プロトコル 6

2.3.3.1. プッシュ型通信（Push Type Communication） 6

2.3.3.2. プル型通信（Pull Type Communication） 7

第3章 実験 8

3.1. 概要 8

3.2. 実験環境 8

3.3. 利用するプログラム 8

3.4. 本プログラムの流れ 8

3.5. 結果 9

3.6. JITコンパイラ 11

3.6.1. YJIT(Yet Another Ruby JIT) 11

3.6.2 RJIT 11

3.6.3 実験・結果 12

第4章 Rubocop概要 15

4.1. Rubocopとは 15

4.2. 機能 15

4.2.1. コードスタイルの検証 16

4.2.2. コード品質のチェック 16

4.2.3. 自動修正 16

4.3. rubocop-ast 16

4.4. カスタムルール 17

第5章 Rubocopによるコード支援の実装 20

5.1. 方針 20

5.2. 実装 21

5.2.1. プッシュ型通信 21

5.2.2. オブジェクト共有 24

5.2.3. プル型通信 26

5.2.4. Ractor#take 28

第6章 結論 31

参考文献 33

謝辞 34

付録 35

# 序論

## はじめに

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は他の言語(Perl, Smalltalk, Eiffel, Ada, Lisp)の影響を受けて設計されている[1]。Rubyは多くの支持を集めているが、こうした成長はRubyの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SaaS企業やスタートアップ企業をはじめとして、数多くの採用例がある。Rubyにはさまざまな処理系があり、公式の処理系としてMRI(Matz’ Ruby Implementation)がある。これは、C言語で実装されたRubyの公式処理系であり、プラットフォームを問わず、動作可能である。まつもとゆきひろ氏により開発され始め、最も広く使用されている。MRI以外にも、JRuby, IronRuby, MacRuby, RubiniusなどさまざまなRuby処理系が存在する。

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はグローバルインタープリターロック（Global Interpreter Lock: GIL）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。また、RubyではGILのことを一貫してGVL(Global VM Lock)と呼ばれている。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。

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とは、スレッドの安全性を気にせずにRubyの並列実行機能を提供するように設計された機構である。RubyのフレームワークであるRuby on Railsのアプリケーションサーバーにおいても、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が期待されている。また、Rubyコミュニティでは、コード品質を維持し、可読性や保守性を向上させるためのツールとして、静的解析ツールRubocop が広く利用されている。Rubocopは、Rubyコミュニティで採用されているコーディングスタイルガイド（Ruby Style Guide）に基づき、コードのフォーマットや構文の一貫性を保つための自動チェックを提供する。これにより、開発者はスタイルの統一を図り、バグの発生を未然に防ぐことが可能となる。

現在、Rubocopは主にRuby on Railsを含むアプリケーション開発において広く活用されているが、並列処理を対象とした支援には十分な機能が備わっているとは言い難い。特に、Ractor を活用した並列処理の記述においては、新しい並列モデルに適したコーディングスタイルの確立が求められている。しかし、現状のRubocopは、Ractorの使用に特化したルールはなく、開発者が適切な記述方法を模索しながら開発を行う必要がある。

本研究では、Ractorで記述されたRubyの並列処理の性能評価を行うことにより、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を使用した記述への支援を目指す。さらに、Ractorを使用した記述時の注意点をRubocopによって警告として実装することで、デバッグ時間の短縮や、Ractorの仕様や制約を開発者にとってより分かりやすくすることを目指す。

# Ractorの概要



## Ractorについて

　Ruby 3.0 において導入された Ractor は、並行および並列処理を安全に実現するための機構であり、Ruby における Actor Model の抽象化として設計されている。従来の Ruby 環境では、GVLにより並列実行が制限されていたが、Ractor の導入により、異なる Ractor 間で GVL の制約を受けずに並列処理を実行することが可能となった。

　Ractor の設計は、スレッドセーフな並列処理を提供することを目的としており、異なる Ractor 間でのデータ共有を禁止し、明示的なメッセージパッシングを用いた通信機構を採用している。このアプローチにより、従来のスレッドプログラミングにおいて問題となる競合状態（race condition）やデッドロック（deadlock）を回避し、スレッド安全性を確保する。

　Ractor の開発は当初Guildという名称のもとで進められていたが、2020 年に正式にRactorへと名称が変更された。Ruby の処理系であるMRIが起動されると、デフォルトで 1 つの Ractor が作成され、これをメインRactorと呼ぶ[2]。メインRactor は Ruby プログラムのエントリポイントとして機能し、他の Ractor を生成・管理する役割を担う。各 Ractor は最低 1 つのスレッドを持ち、同一 Ractor 内の複数スレッドは GVL により同時実行されることはない。

　Ractor の導入により、マルチコア環境における Ruby アプリケーションのパフォーマンス向上が期待される一方で、従来のスレッドを表すクラスThreadとは異なるプログラミングモデルを採用する必要がある。これにより、並行・並列処理の設計に関する新たなベストプラクティスの確立が求められている。

## Actor model

　Actor modelは、1973年に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。このモデルは、独立したアクターと呼ばれる単位が、メッセージのやり取りを通じてそれぞれのアクターが並列に処理を行うという特性を持つ。アクター同士は互いの状態を共有せず、メッセージパッシングによってのみ通信を行うため、競合状態の発生を回避しつつ、高い並列性とスケーラビリティを実現できる。特に、スケールアウトを得意とするクラウドインフラと相性がよく、分散システムや大規模データ処理の分野において、近年再び脚光を浴びるようになった。

　このようなスケールアウト型の設計が注目を集める背景には、ムーアの法則の限界が挙げられる。ムーアの法則（Moore's Law）は、1965年にインテル社の共同創設者であるGordon Moore氏によって提唱された経験則であり、「半導体の集積度は2年ごとに2倍になる」というものである。この法則に基づき、半導体技術の進化により、コンピュータの処理能力は指数関数的に向上し、コストの削減も同時に進められてきた。ムーアの法則は、長年にわたりプロセッサの性能向上を支える基本的な指針となってきたが、近年ではその持続可能性が疑問視されている。

　スケールアウトとは、単一のプロセッサの性能向上に依存するのではなく、複数のプロセッサやノードを並列に動作させることで、全体としての処理能力を向上させる手法である。特に、クラウドコンピューティングの普及により、動的なリソースの追加・削除が容易になったことで、スケールアウトを前提としたアーキテクチャの重要性が増している。この流れを受け、ソフトウェア開発においても、並列性と分散処理を前提とした設計が求められており、Actor modelはその有力な解決策として再評価されている。

　Actor modelに基づいたプログラムは、各アクターが独立した状態を持ち、メッセージを通じて非同期に動作するため、インフラのスケールアウトに応じた拡張が容易である。これにより、リソースの追加による処理能力の向上が可能となり、システム全体の効率的な並列処理が実現される。また、アクターは状態をカプセル化し、直接的な共有を避けるため、ロック機構に依存することなく安全な並行実行が可能となる。この特性は、近年のマルチコアプロセッサの普及やクラウドネイティブな環境において、大きな利点となっている。

　Actor modelの構成は**図 1**に示す。

じょうごグラフ が含まれている画像

自動的に生成された説明

図 1 : Actor modelの構成[3]

　Actor modelにおいて、アクターはクラスから生成されたインスタンスであり、そのインスタンスにはスレッドとキューが関連づけられている。アクターモデルを利用する場合、キュー内のメッセージを短時間で大量に処理したいという要求が生じた場合、スケーラビリティを向上させるためにアクターを増加させることが可能である。このようなスケールアウトは非常にシンプルであり、アクターを1つ追加するだけで達成することが可能である。これは、アクター自体が内蔵するキューを持っているため、排他制御を意識する必要がないという利点がある。キューに格納されたメッセージを順次、処理する。

　Actor modelを採用している有名なプログライング言語として、ErlangやElixirがある。また、ScalaやJavaのフレームワークであるAkkaもActor modelを採用した有名な例である。

## Ractorの仕様

以下、Ractorの仕様について一部を紹介する[4]。

### インタープリタプロセス内のRactor

Rubyでは、1つのインタプリタプロセス内で複数のRactorを生成し、それらを並列に実行することが可能である。

Ractor.new { 式 } を使用すると、新しいRactorが作成され、その内部で指定された式が並列実行される。通常、RubyインタプリタはメインRactorと呼ばれる最初のRactorを起動し、プログラムの実行が進行する。

メインRactorが終了すると、スレッドと同様に、すべてのRactorに終了リクエストが送信される。これは、メインスレッド（最初に起動されたスレッド）が終了すると、Rubyインタプリタがすべての実行中スレッドに終了指示を出す仕組みと同様である。

各Ractorは1つ以上のスレッドを内部に持つ。ただし、Ractor内部のスレッドは、MRIにおけるGVLの影響を受けるため、GVLをCレベルで明示的に解放しない限り、並列実行はできない。一方、異なるRactorに属するスレッド同士は並列に実行可能である。

Ractorの生成コストは、スレッドを1つ作成する際のオーバーヘッドと同程度であるとされる。

### Ractorにおけるオブジェクト共有

RubyのRactorは、従来のスレッドとは異なり、すべてのオブジェクトを共有するわけではない。この設計により、オブジェクトの共有に起因するスレッドセーフティの問題を最小限に抑えることが可能となっている。

Ractorでは、オブジェクトは共有可能（Shareable）と共有不可（Unshareable）の2種類に分類される。通常、ほとんどのオブジェクトは共有不可とされており、その結果、異なるRactor間でデータの不整合が発生するリスクが軽減される。一方で、特定の条件を満たしたオブジェクトは共有可能となり、異なるRactor間でのやり取りが可能となる。

共有可能なオブジェクトとして、以下の3種類が挙げられる。

#### 不変オブジェクト（Immutable Objects）

不変オブジェクトとは、変更が加えられないことが保証されたオブジェクトである。具体的には、オブジェクトが freeze されており、かつ内部に共有不可オブジェクトを保持していない場合、これを不変オブジェクトとみなすことができる。

例えば、以下のオブジェクトは不変とみなされる：

* i = 123（数値は変更不可能なため、不変オブジェクト）。
* s = "str".freeze（文字列を明示的に freeze することで不変化）。

一方で、以下のようなオブジェクトは不変とはならない：

* a = [1, [2], 3].freeze（配列全体を freeze しても、内部に変更可能な要素 [2] を含むため不変オブジェクトではない）。
* h = {c: Object}.freeze（ハッシュ全体を freeze しても、内部に Symbol :c やクラス Object が含まれている場合は不変とみなされる）。

#### クラス・モジュールオブジェクト（Class/Module Objects）

　Rubyのクラスやモジュールは本質的に共有可能なオブジェクトとみなされ、Ractor間で安全に利用できる。これにより、複数のRactorが同一のクラスやモジュールを参照し、メソッドの実行が可能となる。

#### 特別な共有可能オブジェクト（Special Shareable Objects）

　Ractorオブジェクト自体や、一部の組み込みオブジェクトは、特別に設計されており、異なるRactor間で共有することが許可されている。これにより、必要最小限の情報共有を安全に行うことができる。

### Ractor間の通信プロトコル

　Ractorは、メッセージ交換による通信を通じて相互作用し、実行を同期する仕組みを持つ。この通信には2種類のプロトコルが存在し、それぞれプッシュ型（push type）とプル型（pull type）と呼ばれる。これらの通信方法を活用することで、柔軟かつ効率的な並列処理を実現している。

#### プッシュ型通信（Push Type Communication）

プッシュ型通信は、メッセージの送信と受信が非同期に行われるプロトコルである。この方法では、送信側のRactorが特定の受信側Ractorに対してメッセージを送信し、受信側がメッセージを受け取る仕組みとなっている。

* 送信と受信の仕組み  
  送信側は Ractor#send(obj) を用いて対象Ractorにメッセージを送信し、受信側は Ractor.receive を用いてメッセージを受け取る。送信側は対象のRactor（宛先）を特定するが、受信側は送信元を特定せず、どのRactorからのメッセージも受け入れる仕様となっている。
* キューを用いた非同期性  
  受信側Ractorは無限のキューを持ち、送信側はこのキューにメッセージを非同期に追加する。この仕組みにより、送信側はブロックされることなくメッセージを送信できる。なお、Ractor.receive\_if { 条件式 } を使用することで、特定の条件に一致するメッセージを選択的に受け取ることも可能である。

#### プル型通信（Pull Type Communication）

プル型通信は、送信側と受信側が明確に対となるプロトコルである。この方法では、送信側がデータを提供する準備が整った際に受信側がデータを取得し、通信が完了する。

* 送信と受信の仕組み  
  送信側は Ractor.yield(obj) を用いてデータを提供し、受信側は Ractor#take を用いてデータを取得する。プッシュ型とは異なり、送信側は宛先Ractorを知らず、受信側が送信元のRactorを特定する形となる。
* 同期的な動作  
  プル型通信では、送信側または受信側のいずれか一方が通信相手を待機していない場合、通信がブロックされる。これにより、明確なデータフロー制御が可能となる。

# 実験



## 概要

バブルソートを用いてRactorの実験を行った。筆者は、C言語とMPI(Message Passing Interface)を用いて、バブルソートの並列化を行なった経験があり、要素数Nを2並列でソートさせたときにほぼ 1/4 の実行時間で並列ソートを完了させることが出来た経験があるため、Ractorでもこれを用いることにした。

## 実験環境

|  |  |
| --- | --- |
| OS | MacOS Sonoma 14.5 |
| CPU | Apple M1　8-Core |
| Ruby Version | 3.3.4 |



## 利用するプログラム

今回利用するプログラムは付属プログラム bubble\_sort\_ractor.rbを利用する。

## 本プログラムの流れ

　また、本プログラムの流れを**図 2**に示す。

ダイアグラム

中程度の精度で自動的に生成された説明

図 2 : バブルソートの並列化

1. 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。
2. 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。
3. 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。
4. この操作を左右からの探索が衝突するまで繰り返す。
5. 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。
6. その後、各Ractorで昇順にバブルソートを実施する。
7. 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。

バブルソート（Bubble Sort）は、隣接する要素を比較しながら並び替えることでリストをソートする基本的なソートアルゴリズムである。以下に、バブルソートの基本的な手順を示す。

1. リストの先頭から末尾に向かって、隣り合う要素を比較する。
2. 比較した要素がソート条件を満たしていない場合（例：昇順ソートにおいて前の要素が後の要素より大きい場合）、これらの要素を交換する。
3. リストの末尾まで到達したら、末尾の要素は確定するため、それを除いた範囲で再び1から2の操作を繰り返す。
4. この操作をリスト全体がソート済みになるまで続ける。

## 結果

　実行時間を測定した結果について、**表 1**に示す。Rubyのみの記述による逐次実行、Ractorを1つ生成し逐次実行行った場合、そして、Ractorを2つ生成し、前項の方法で並列実行した3種類の実行時間の結果である。

表 1 : 3種類の実行時間の結果

テーブル

自動的に生成された説明

　バブルソートの計算量はである。並列化した際に最速になる場合を考えると、要素数が半分のであるため、計算量は となる。つまり、逐次実行時よりも4倍の速度向上が予測される。しかし、結果としてはデータ数が50,000の時の約2.87倍が並列化効率の最もよい場合であった。データ数が少ない場合では、Ractorのオーバーヘッドやデータ分割処理により、実行時間が逐次実行に比べて並列実行は4.3倍ほど遅くなった。

　次に、Ractorのどの箇所でオーバーヘッドがあるのかをRactorの各処理における実行時間を測定することで調べた。前実験のデータ数50,000の並列化について各Ractorの処理の実行時間を計測した結果を**表 2**に示す。

表 2 : Ractorの各処理における実行時間

テーブル

自動的に生成された説明

　結果として、実行時間の約99%をバブルソートに費やしている。さらに、ractor1の生成時間が明らかにractor2よりもかかっていることがわかる。さらにRactorを2つ生成し、計4つの各Ractorの生成時間を計測したが、ユーザーが生成する1つ目に生成されるRactorは他のRactorの生成に比べて生成時間がかかることがわかった。

　また、データ数50,000の並列化が最速になる場合、各Ractorに25,000のデータが送られバブルソートされることになる。試しに、データ数25,000でRactorが1つのときの逐次実行を計測した結果、約46[sec]になることがわかった。さらに、2つのRactorによる並列実行について、一方のRactorのバブルソート処理を行わないようにすると約46[sec]の実行時間が得られた。このことから、複数のRactor間で配列の各要素に対してスワップ処理を行う部分では、各Ractorで処理時間が遅くなることがわかる。

　この要因としては次の可能性が考えられるが引き続き調査が必要である。Rubyの配列は可変長配列であるため、ソートする際のスワップ処理を大量に繰り返すことでメモリの再配置が頻繁に発生する。ガベージコレクションが頻発することによる速度低下である。これについてはガベージコレクションを停止して実行しても効果は得られなかった。さらに、Ractorはスレッド安全を保証するために、共通の配列要素へのアクセス時に排他制御が働き、結果として並列に実行されない可能性も考えられる。この場合、排他制御のオーバーヘッドが発生し、並列処理の性能向上が妨げられている可能性がある。

## JITコンパイラ

　RubyはJIT(Just-In-Time)コンパイラの機能を備えている。有効化することにより、実行時に機械語を生成し、最適化が行われ、実行が高速になる[5]。Ruby3.3にはYJITとRJITの2つのJITコンパイラがあるが、どちらもデフォルトでは無効になっている。そのため、JITコンパイラを有効化して再度同様の実験を行った。

### YJIT(Yet Another Ruby JIT)

　特にRubyの実行速度を大幅に向上させることを目指して開発された。YJITは、Rustで書かれているため、起動時にRustコンパイラが存在していないと有効化することはできない。アプリケーションのコードに変更を加える必要がないまま、コンパイラの最適化技術を用いてパフォーマンスを向上させるため、多くの開発現場で注目されている。特に、Webアプリケーション開発で広く使われるRuby on Railsにおいては、YJITを利用することで、実行速度が最大40%向上するケースも報告されている。

### RJIT

　RJITは、JITコンパイラの最適化実験を支援する環境を提供することを主な目的としている。そのため、本番環境では主にYJITの導入が推奨されている。Rubyで書かれたJITコンパイラを提供している。Ruby 3.2では、MJITという別のJITコンパイラが存在していたが、これに置き換わる形でRJITはRuby 3.3に導入された。MJITはRuby 2.0に比べて3倍の高速化を達成したが、その性能を圧倒するYJITの登場により導入例が減少してしまった。しかし、MJITはRubyで書かれていたため記述が容易であり、YJITに比べて実装コストが低く、モンキーパッチすることにより、独自のJITコンパイラを実装することができるという利点があった。しかし、実行時にCコンパイラを起動する仕様であり、その仕様から特殊な実装がされていたため、保守が難化してしまう問題があった。その問題を解決し、MJITの利点を受け継ぐRJITがYJIT開発を助けることを目的に開発された。

### 実験・結果

　上記で述べた通り、YJITは本番環境に推奨されており、より性能が高いことから、今回はYJITを使用して実験を行った。

　まず、実験では、YJITを有効化するためにRust環境を構築した。Rustの環境構築を完了し、YJITが有効化されているかの確認を行った。確認の結果を図 **3**に示す。Rust環境が正常に導入されているが、Ruby側でRustコンパイラの認識ができていなかった。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 3 : Rust versionとYJITの有効化確認

　Rubyは主に、rbenvによって管理されている。rbenvとは、Rubyのバージョン管理ツールである。プロジェクトごとに異なるバージョンを簡単に切り替えられ、シンプルで軽量な設計であることから、広く使用されている。このrbenvを使用し、Ruby 3.3.4の再インストールを行うことにより、YJITの有効化を確認した。YJITの有効化の確認結果を**図 4**に示す。



図 4 : YJITの有効化確認

　YJITの有効化したため、これを使用して実験を行った。実験は、バブルソートを逐次実行と2つのRactorによる並列実行するプログラムを使用し、YJITを無効化した場合と有効化した場合のそれぞれ4通り行った。実験の結果を**表 3**に示す。

表 3 : YJITの有無によるバブルソート実行時間

テーブル

AI によって生成されたコンテンツは間違っている可能性があります。

　データ数が50個の場合は、YJITが無効化されていた方が実行速度が速いことが確認された。これは、プログラムの実行時にバイトコードをネイティブコードに変換し、このプロセスがオーバーヘッドとなっているため、データ数が小さい場合は、Rubyインタブリタをそのまま実行する方がYJITを挟むよりも効率的であると考察される。データ数が500個以上の場合は、YJITを有効化した方が、高速であることが確認された。また、YJITが有効になっている場合、無効化されている場合に比べて、データ数が多くなるにしたがい、速度性能の向上倍率が大きいことがわかった。さらに、YJITの有無による並列実行と逐次実行の性能倍率について図 **5**に示す。

グラフ, 折れ線グラフ

AI によって生成されたコンテンツは間違っている可能性があります。

図 5 : YJITの有無による並列実行と逐次実行の性能倍率



# Rubocop概要

## Rubocopとは

　プログラミングにおけるコード品質の向上は保守性や開発効率を高めるための重要な課題である。特に、動的型付け言語であるRubyは、その柔軟性ゆえにコードスタイルやエラーの検出が開発者間で一貫しない場合が多い。この問題を解決するために登場したのが、Ruby向けの静的解析ツールであるRubocopである。

　Rubocopは、Rubyで記述されたコードの静的解析ツールであり、コードの品質やスタイルガイドの遵守をサポートするためのライブラリである。Rubyコミュニティで広く使用されており、Rubyのコーディング規約であるRuby Style Guideを基に、コードが適切に書かれているかを自動的に解析する。また、コードのフォーマットや構文上の問題を検出・自動修正する機能を提供する。Rubocopは主に、以下の目的で使用される：

* コードスタイルの一貫性の維持：開発チーム間でスタイルの統一を図る。
* 潜在的なバグの発見：コードの非推奨な構文や誤りを事前に検出する。
* コードの可読性向上：規定されたスタイルに従うことで、コードを直感的に理解しやすく、プロジェクト全体で記述を統一する。

　また、Rubocopはgemという形式で公開されている。gemはRubyGemsと呼ばれるRuby専用のパッケージ管理システムによって管理されている。Ruby 1.9以降に組み込まれている。Rubyのライブラリは主にRubyGems.orgにgemとして置かれている[6]。そのため、Rubyの開発ではライブラリ指す際にgemと呼ぶことがある。Rubyのエコシステムを支える中核的な存在であり、効率的な開発やコミュニティの活性化を実現する重要な仕組みである。

　前述した通り、RubyにはこのRubyGemsを扱うためのライブラリが組み込まれており、これを扱うためにgemコマンドが提供されている。Rubocopの導入には、このgemの仕組みを活用する。Ruby環境があれば、gem install ‘gemパッケージ名’を実行することで、すぐにrubocopを使用することが可能である。CLIでrubocopを実行することにより、プロジェクト内のコード解析を行うことが可能である。また、ローカル環境の実行だけにとどまらず、大規模なRubyプロジェクトにおいて、RubocopをCI/CDパイプラインに組み込むことで、自動的にコードスタイルをチェックするという使用方法も存在する。

## 機能

　Rubocopの機能は大きく以下の3つに分類される。さらに、その他の解析ルールについてもgemとして多数、存在している。その代表例として、WebアプリケーションフレームワークであるRuby on Railsに関するルールをRubocopとして提供するruboocop-railsやテストフレームワークであるRSpecのルールを提供するrubocop-rspecなどがある。

### コードスタイルの検証

Rubocopは、コードがスタイルガイドに準拠しているかをチェックするための多くのルールを提供している。簡単な例として、以下のようなルールが含まれる：

* インデント：スペースの数やタブの使用方法の確認。
* 行の長さ：1行あたりの文字数制限（デフォルトでは120文字）。
* 変数名やメソッド名の命名規則：スネークケースやキャメルケースの適用。

### コード品質のチェック

　Rubocopは、潜在的なバグや非効率なコードパターンを検出する機能も提供する。以下のような例があがる：

* 未使用の変数やメソッド。
* 冗長なコードや非推奨構文の使用。
* 性能に影響を与える可能性があるコード。

### 自動修正

　Rubocopは検出した問題を自動的に修正する機能を提供する。rubocopコマンドのautocorrectオプションを使用することで使用可能である。この機能は、フォーマットに関連する問題を効率的に解決するため、開発者の負担を大幅に軽減する。

## rubocop-ast

　rubocop-astとはRubocopのバックエンドライブラリであり、gemとして提供されている。Rubyコードを抽象構文木(AST: Abstract Syntax Tree)し、その解析を可能にする。ASTは、プログラムの構文を木構造で表現したもので、構文解析や静的コード解析の基盤として広く利用されている。Rubocopはrubocop-astを使用し、効率的にコード解析を行えるようにしている。

　また、Rubyコードを解析可能なデータ構造に変換するだけでなく、Rubocop上での解析を用意にするための拡張機能を提供している。通常のASTと比較して、rubocop-astは次のような追加機能を持っている。

* メソッド群  
  ASTノードを操作するためのメソッドが豊富に提供されている。メソッドの例を、**図 6**に示す。メソッド呼び出しのノードに対して、レシーバ(メソッドの呼び出し元)や引数を簡単に取得することが可能である。

|  |
| --- |
| node.receiver # メソッド呼び出しのレシーバを取得  node.arguments # メソッド呼び出しの引数を取得 |

図 6 : ASTノードの操作メソッド

* パターンマッチング機能  
  ノードの種類や構造を簡潔に判定するためのDSL（ドメイン固有言語）が利用できる。これにより、解析対象を正確に絞り込むことが可能である。パターンマッチングの例を**図 7**に示す。

|  |
| --- |
| node.match?('(send nil? :puts ...)') |

図 7 : ASTのパターンマッチング

match?はrubocop-astによるASTパターンマッチングの機能である。このメソッドを使用すると、特定の構文パターンにノードが一致するかを確認することが可能である。'(send nil? :puts ...)' は、AST ノードのパターンを表現する パターン文字列 である。それぞれの部分は以下のように分解できる。(send ...)ノードは send ノード（メソッド呼び出し）を表す。Ruby のコードにおけるメソッド呼び出しは AST では send ノードとして表現される。nil?ではレシーバが nil であることを意味する。Ruby では、レシーバが省略されたメソッド呼び出しは内部的にレシーバが nil として扱われる。putsは呼び出されたメソッド名を示す。この場合は puts メソッドが呼び出されていることを意味する。...は可変長引数を表す。この部分は「0 個以上の引数」を意味する。

## カスタムルール

　Rubocopは、特定のルールを担当しているモジュールがある。例として、以下のカテゴリがある。

* Style：スタイルに関するルールを検証。
* Lint：潜在的なバグや不具合を検出。
* Metrics：複雑度やコード量を測定。

他にも多くのルールが提供されているが、プロジェクト固有のスタイルや特別な要件を反映するために、既存のルールでは不十分な場合がある。このような場合、Rubocopは独自のルールを定義するカスタムルールの作成を可能にしている。これは、Rubocopの既存ルールに加えて、独自のルールを定義するための拡張機能である。これにより、Rubocopは標準ガイドラインに加えて、柔軟にカスタマイズされたコード検証を提供する。

　カスタムルールを作成するには、Rubocopの内部構造とrubocop-astによるAST解析を理解する必要がある。カスタムルールは、Rubocopの内部で用いられるASTを解析し、特定の条件に一致するノードを検出する仕組みで動作する。ASTは、Rubyのコードを構造化した木構造として表現したもので、各ノードはプログラムの構文要素であるメソッド呼び出し、条件分岐、変数の代入などを表す。

　まず、作成するルールの目的を定義する。どのような問題を検出し、どのようなスタイルを強制するかを明確にする。これにより、ルールの実装方針を定める。

　カスタムルールは、RuboCop::Cop::Baseを継承したクラスとして実装する。このクラスでは、on\_<node\_type>という形式のメソッドを定義し、メソッド呼び出しや変数定義などの特定のノードタイプを解析する[7]。カスタムルールの実装方法を図 **8**に示す。

|  |
| --- |
| module RuboCop  module Cop  module Custom  class MethodName < Base  MSG = ' Do not prefix reader method names with number.'  def on\_send(node)  return unless bad\_reader\_name?(node)  add\_offense(node, message: MSG)  end  private  def bad\_reader\_name?(node)  node.method\_name.to\_s.start\_with?(/\d/)  end  end  end  end  end |

図 8 : カスタムルールの実装例

　このルールは、数字で始まるメソッド名を検出し、警告を出力するものである。カスタムルールの実装はRubocopのCopモジュール内に新しいサブクラスを定義する。この例では、RuboCop::Cop::Custom::MethodNameとして実装している。この構造により、Rubocopの他のルールと一貫性を保ちながら拡張可能な形でカスタムルールを構築することができる。

　MSGには違反が検出された際に出力される警告メッセージを指定する。本例では’Do not prefix reader method names with number.’がメッセージとして設定されている。これにより、開発者が違反内容を明確に理解することができる。

　また、Rubocopでは、RubyコードをASTとして解析する。on\_sendメソッドは、メソッド呼び出しノードを処理するために利用される。このメソッド内で、検出対象ノードを判別するための条件を設定する。このノードタイプメソッドは、rubocop-astにて、あらかじめ提供されている[8]。また、on\_sendメソッド内のnodeは解析中のASTノードを表し、add\_offenseメソッドで違反箇所を記録し、警告メッセージを表示するためのメソッドである。

　bad\_reader\_name?メソッドは違反条件を定義している。違反条件はRubocopで特別提供されているものではないため、自ら実装する必要がある。本例では、node.method\_nameにてメソッド名を取得し、to\_s\_star\_with? (/\d/)でメソッド名が数字で始まるかを判定する。return unless bad\_reader\_name?(node)にて、bad\_reader\_name?メソッドの返り値がfalseの場合、早期にメソッドを終了させる。これを一般にガード節と呼ぶ。

　以上のような流れでカスタムルールを作成することが可能である。

# Rubocopによるコード支援の実装

## 方針

　Rubyにおける並列処理のための新しいモデルとして導入されたRactorは、スレッドセーフな並列処理を実現するための強力な仕組みを提供する。しかし、Ractorは新しい構文や制約を持ち、これを適切に理解する学習コストが必要であり、逐次処理を記述するのに比べて難しい。具体的には、以下のような特徴と課題がある。

* オブジェクト共有の制約  
  Ractorは並列処理においてデータの一貫性を維持するため、オブジェクトの共有に厳格な制約を設けている。この制約を守るためのルールは複雑であり、コード記述において頻繁にエラーを引き起こす可能性がある。
* 新しいAPIと構文の学習コスト  
  Ractorは従来のRubyコードとは異なるAPIや構文を用いるため、既存の開発者にとっては新しい知識の習得が必要となる。これにより、初学者や既存のプロジェクトへの適用が難しくなることがある。
* 並列処理のバグ発生リスク  
  並列処理には競合状態やデッドロックといった問題が起こり得る。Ractorはこれらのリスクを軽減するよう設計されているが、プログラム全体のロジックが複雑になることで、バグのリスクは依然として存在する。

以上の背景を踏まえ、Ractorを用いたコードの記述を支援するための手法として、Rubocopを活用することを提案する。

　Ractorの文法規則やベストプラクティスに基づいた静的解析ルールをRubocopに実装することで、開発者が効率的かつ正確に並列処理を記述できるよう支援する。具体的には以下の3つを目指す。

* 文法エラーの早期検出  
  Ractorに固有の文法規則（例: 分離性の要件）をRuboCopのカスタムルールとして実装することで、開発者がコード記述時に即座に問題点を把握できるようにする。
* ベストプラクティスの普及  
  Ractorの効率的な利用方法や推奨される設計パターンをルール化し、コードレビューを自動化することで、開発者が高品質なコードを書く習慣を身に付けられるよう支援する。
* 学習支援  
  開発者がRactorに関する知識を深められるよう、ルール違反の警告メッセージに適切な修正方法を提供する。

## 実装

### プッシュ型通信

　Ractor は、明示的なメッセージパッシング (send/receive) を用いてデータをやり取りする。この仕組みによりスレッドセーフな並行処理が可能となるが、開発時には以下のような問題が発生しやすい。

* send と receive の対応関係が不明確になる  
  Ractor#send が呼び出された箇所と、それに対応する Ractor.receive が記述された箇所の整合性を手動で確認するのは困難である。
* エラーチェックが複雑化する  
  対応関係が誤っている場合、プログラムの実行時にエラーや無限に待機してしまう状態が発生する可能性がある。

これらの問題を解決するため、Rubocop のカスタムルールを実装し、send と receive の対応関係を静的解析で検証する仕組みを構築した。

　通常の Rubocop の機能では、AST ノード内の Ractor#send と Ractor.receive の対応を確認することは不可能であるが、本実装では RactorChecker クラスを別途作成することで、これを実現した。RactorCheckerクラスの実装について、付属プログラム ractor\_checker.rbに示す。具体的には、Ractor 名をキーとしたマッピングを構築し、それに基づいて対応関係を検証する。また、対応関係が正しくない場合には警告を生成するだけでなく、必要に応じてコードを自動修正する機能も提供する。

　実装の要点としては、まず RactorChecker クラスがコードファイルを解析して AST を生成し、Ractor.new ブロック内での send および receive の呼び出しを検出する。検出したノードは、それぞれ Ractor.receive または Ractor#send のリストに分類される。次に、これらのリストを比較することで対応関係を検証し、対応する Ractor#send または Ractor.receive が不足している場合には警告を生成する。

　このカスタムルールは、以下の 2 つの Cop を中心に構成される。ひとつは Style::RactorSendReceive クラスであり、Ractor.receive が存在するにもかかわらず対応する Ractor#send が見つからない場合に警告を出す。Style::RactorSendReceiveクラスの実装については、付属プログラム ractor\_send\_receive.rbに示す。もうひとつは Style::RactorReceiveSend クラスであり、Ractor#send が存在するにもかかわらず対応する Ractor.receive が見つからない場合に警告を出す。Style::RactorReceiveSendクラスの実装については、付属プログラム ractor\_receive\_send.rbに示す。これらの Cop はそれぞれ RactorChecker を利用して解析を行い、静的解析の結果を基にコードの修正提案を行う。

　例えば、Ractor.receive に対応する Ractor#send が不足している場合、Style::RactorSendReceive クラスは警告を生成し、オートコレクト機能によって不足している Ractor#send を自動的に挿入する。同様に、Ractor#send に対応する Ractor.receive が不足している場合には、Style::RactorReceiveSend クラスが警告を生成し、不足している Ractor.receive を自動的に挿入する。このオートコレクト機能により、開発者は手動で修正箇所を特定する必要がなくなり、コードの品質と開発効率が向上する。

　このカスタムルールについて図 **9**のサンプルコードを用いて、動作を示す。

|  |
| --- |
| r = Ractor.new do  msg = Ractor.receive  msg  end  r.send 'ok'  p r.take |

図 9 : Ractor Send/Receive サンプルコード

　使用するサンプルコードの流れについて説明する。まず、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メソッドを使用してRactorから返された文字列’ok’を受け取り、標準出力に出力する。

　実際にサンプルプログラムの実行結果を図 **10**に示す。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 10 : 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 **11**に示す。

グラフィカル ユーザー インターフェイス が含まれている画像

AI によって生成されたコンテンツは間違っている可能性があります。

図 11 : 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してみる。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 **12**に示す。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 12 : 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 **13**に示す。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 13 : 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 **14**に示す。8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。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 14 : 自動修正機能の実行

### オブジェクト共有

　次のカスタムルールとして、Ractor ブロック外の変数を誤って参照している箇所を指摘するものである。Ractor を活用した並行処理では、Ractor ブロック外の変数を誤って参照しようとするとエラーが発生する仕様になっている。この問題を防ぐために、Rubocop のカスタムルールを実装し、Ractor ブロック内での外部変数の参照を検出する仕組みを提供した。

　本カスタムルールでは、Style::RactorSharingVariables クラスを中心に、Ractor ブロック内で使用される変数がブロック外のスコープに属していないかを検証する。Style::RactorSharingVariablesクラスの実装については、付属プログラム ractor\_sharing\_variables.rbに示す。このルールは、ASTを解析し、Ractor.new ブロック内での変数参照をチェックすることで動作する。

　このルールの実装において重要な役割を果たすのが、RactorExternalReferencesChecker クラスである。RactorExternalReferencesCheckerクラスの実装については、付属プログラム ractor\_external\_references\_checker.rbに示す。このクラスは、コード全体の AST を解析し、Ractor.new ブロック内で定義されていない変数が参照されているかどうかを検出する。具体的には、以下のような手順で解析を行う。

1. コードファイルを読み込み、AST を生成する。
2. AST ノードを深さ優先探索し、変数の定義箇所と参照箇所を特定する。
3. Ractor.new ブロック内で参照されている変数が、ブロック外で定義されている場合に警告を生成する。

　Style::RactorSharingVariables クラスは、Rubocop の Cop を継承し、AST ノード内での変数参照を検出する。具体的には、on\_lvar メソッドをオーバーライドし、ローカル変数参照ノードを検出した際に、その変数が Ractor ブロック外部のスコープに属しているかどうかを RactorExternalReferencesChecker を通じて検証する。検証の結果、外部変数が参照されている場合には警告を出力する。

　通常の Rubocop の機能では Ractor ブロック内での外部変数参照を検出することは困難である。しかし、本実装では RactorExternalReferencesChecker を用いることで、AST の解析を通じてこれを実現している。また、検出した問題に対して自動修正機能は提供しない設計となっている。これは、外部変数の参照を解消するには通常、コード全体の設計変更が必要となるためである。

　このカスタムルールについて図 **15**のサンプルコードを用いて、動作を示す。

|  |
| --- |
| s = 'hello'  r = Ractor.new do  s << 'world'  end  p r.take |

図 15 : RactorSharingVariables検証用サンプルコード

　使用するサンプルコードについて解説する。このコードは、変数sに文字列’hello’を代入する。その後、Ractorを生成し、その中で変数sに格納された’hello’に’world’を追加する。そして、Ractorの結果をr.takeで取得し、標準出力に出力している。しかし、Ractorでは安全な並列処理のため、可変なオブジェクトを複数のRactor間で同時に共有して操作しようとすることはできない仕様になっている。そのため、Ractor内で変数sを操作しようとした時点でエラーになってしまう。

　実際に作成したカスタムルールを実行した結果を図 **16**に示す。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 16 : Rubocop RactorSharingVariables 実行結果

　本カスタムルールにより、Ractor を用いた並行処理コードの品質向上が期待できる。このルールを活用することで、開発者はエラーの原因となる外部変数参照を事前に防止できる。

### プル型通信

次にpull型の通信についてのルールを実装した。RactorはRactor.yield と Ractor#take を用いてデータをやり取りすることが可能である。しかし、Ractor.yield が記述されていても、それに対応する Ractor#take が記述されていない場合、プログラムの挙動が意図しないものになる可能性がある。このような不整合を防ぐため、Rubocop のカスタムルールを実装し、Ractor.yield と Ractor#take の対応関係を静的に検証する仕組みを構築した。

　本カスタムルールは、Style::RactorYieldTake クラスを中心に、Ractor.yield が適切に Ractor#take と対応付けられているかを検証する。Style::RactorYieldTakeクラスの実装については、付属プログラム ractor\_yield\_take.rbに示す。このルールは、ASTを解析し、Ractor.yield の呼び出しが存在する場合に、その Ractor 内で対応する Ractor#take が存在するかを確認する。通常の Rubocop の機能では、このようなペアリングの検証は不可能であるが、本実装では RactorYieldTakeChecker クラスを利用することでこれを実現している。RactorYieldTakeCheckerクラスの実装については、付属プログラム ractor\_yield\_take\_checker.rbに示す。

　RactorYieldTakeChecker クラスは、コード全体の AST を解析し、Ractor.yield および Ractor#take の呼び出し箇所を特定する。解析の手順は以下の通りである。

1. コードファイルを読み込み、AST を生成する。
2. AST ノードを探索し、Ractor.yield と Ractor#take の呼び出しを検出する。
3. 検出したノードを Ractor 名をキーとしてマッピングし、それに基づいて対応関係を検証する。

　Style::RactorYieldTake クラスは、Rubocop の Cop を継承し、on\_send メソッドをオーバーライドすることで、Ractor.yield の呼び出しを検出する。この際、Ractor.new ブロック内における Ractor.yield の呼び出しを対象とし、対応する Ractor#take の有無を RactorYieldTakeChecker を通じて検証する。対応する Ractor#take が見つからない場合には警告を生成する。

　このカスタムルールについて図 **17**のサンプルコードを用いて、動作を示す。

|  |
| --- |
| r = Ractor.new do  Ractor.yield 'ok'  end  p r.take |

図 17 : RactorYieldTake 検証用サンプルコード

　使用するサンプルコードについて解説する。まず、Ractorを生成し、Ractor内でRactor.yieldによりRactor#takeで値を取得するまで待ち、文字列’ok’を返す。その後、Ractor#takeで受け取った文字列’ok’を標準出力に出力する。つまり、Ractor.yieldがあり、それに対応するRactor#takeがない場合はRactor.yieldで返した値を取得する手段がなくなってしまう。

　実際に作成したカスタムルールをサンプルコードで実行した結果を図 **18**に示す。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 18 : Rubocop RactorYieldTake 実行結果

　このルールを活用することで、開発者は Ractor.yield と Ractor#take の対応関係を明確にし、意図しない動作を防止できる。

### Ractor#take

　Ractor#takeに関してのルールをもう一つ実装した。Ractor#take を適切に使用しない場合、期待するデータの受け渡しが行われず、プログラムの動作に問題を引き起こす可能性がある。この課題を解決するために、Ractor.take の適切な使用を検証するカスタムルールを実装した。

　本カスタムルールは、Style::RactorTakeクラスを中心に、Ractor.newでインスタンスが生成されている際に、Ractor#takeが使用されているかを検証する。Style::RactorTakeクラスの実装については、付属プログラム ractor\_take.rbに示す。このルールは、ASTを解析してRactor.newの呼び出しを検出し、対応するRactor#takeが存在するかを確認する。これにより、Ractor間でのデータやり取りが正しく行われるように保証する。

　RactorTakeクラスの動作は以下の通りである。

1. ASTノード内でRactor.newブロックを検出する。
2. Ractor.newと対応付いたRactor#take呼び出しを解析する。
3. 対応するRactor#takeが存在しない場合には警告を出力し、問題を開発者に通知する。

　このルールの実装にはRactorTakeCheckerクラスが対応するRactor#takeを検出する役割を果たしている。RactorTakeCheckerクラスの実装については、付属プログラム ractor\_take\_checker.rbに示す。このクラスは、コードファイル全体のASTを解析し、Ractor#takeの呼び出し箇所を特定する。また、Ractorインスタンスに対応するRactor#takeが存在するかを判定する。

　このカスタムルールについて**図 19**のサンプルコードを用いて、動作を示す。

|  |
| --- |
| r = Ractor.new do  sleep 10  p 'a'  end  r.take |

図 19 : Rubocop RactorTake検証用サンプルコード

　使用するサンプルコードについて解説する。まず、新しいRactorを生成する。このRactorの中で、10秒プログラムの実行を停止する。その後、文字’a’を標準出力に出力する。最後に、r.takeでRactorの結果を受け取る。このr.takeがない場合、Ractor内の処理が完了するのを待たずに終了してしまう。それでは意図した動作にはならないため、takeメソッドを使用してRactorの結果を受け取ることが必要とされる。

　実際にr.takeをコメントアウトし、作成したカスタムルールを実行した結果を**図 20**に示す。rubocop ./source/take.rbを実行するとtake.rbファイル内のRactor.newによりRactorが生成されている場合、対応するRactor#takeが存在しなければ、Rubocopによる警告が出力される。

テキスト

AI によって生成されたコンテンツは間違っている可能性があります。

図 20 : Rubocop RactorTake 実行結果

　このように、Ractorインスタンスに対応するRactor#takeを検出することで前述したサンプルコードのような、意図しない挙動を実行せずに検出することができる。

# 結論



　本研究では、Rubyの並列処理機構であるRactorの性能評価を通じて、その利点と課題を明らかにし、静的解析ツールRubocopを用いたコード支援の可能性について検討した。Ractorは、並列処理における安全性と効率性を両立するために設計された仕組みであり、GVLの制約を超えて、異なるRactor間で並列実行が可能であるという特長を持つ。しかし、その利用には特有の制約があり、開発者にとっては新しいプログラミングモデルの理解と適応が求められる。これにより、Ractorを活用する際の学習コストや記述時のミスが課題として浮かび上がる。

　本研究で注力したのは、Rubocopを活用してこれらの課題を解決するための方法を模索することであった。Rubocopはコードの静的解析を行い、スタイルガイドに基づいた記述の一貫性を保つための強力なツールであり、本研究ではこれにRactor専用のカスタムルールを実装することで、並列処理コードの記述を支援した。その結果、いくつかの効果が確認された。まず、カスタムルールを用いることで、プログラム実行前に潜在的なエラーを検出することが可能となった。これにより、従来は実行中に発覚していた問題を、記述段階で修正することができ、デバッグに要する時間を削減できた。また、Ractorの特有の制約を静的解析によって明示することで、開発者がRactorを効率的に活用しやすくなり、結果としてコードの可読性や保守性が向上した。

　加えて、本研究で実装したカスタムルールは、Ractorの利用に不慣れな開発者にも有用であることが示された。たとえば、RactorにおけるSend/Receiveの対応関係を検証するルールでは、対応するメソッドが不足している場合に警告を表示するだけでなく、必要に応じて自動修正を提供する機能も備えている。このような機能により、開発者はRactor特有の仕様を自然に学びながらコードを記述できる環境が整った。また、これらのルールはコードレビューの負担を軽減するだけでなく、ベストプラクティスを普及させる役割も果たす。

　一方で、本研究ではいくつかの課題も明らかとなった。第一に、Ractorに関連するさらなるカスタムルールの追加が求められる。現時点では、Ractorの利用に関する基礎的なサポートは実現したが、実用的なシナリオにおいてはより詳細な解析が求められる場合が多い。また、カスタムルールの実行速度に関する問題も課題として浮上した。本研究で実装した解析機能では、Send/Receiveの対応関係を検証する際に対象ファイルを2回解析する仕組みを採用したが、このプロセスがコード量の増加に比例して性能の低下を引き起こす可能性がある。この問題に対しては、解析アルゴリズムの改善が必要であり、効率的なデータ構造や手法の採用が今後の課題となる。

　さらに、Ractorを用いた実験では、Rubyのガベージコレクションが性能に与える影響が一部確認された。Ractor間でのデータのスワップ処理が頻発する場合、可変長配列を用いることによるメモリの再配置が発生し、これが性能の低下につながることが示唆された。さらに、排他制御の影響で並列に実行されない可能性もあることが考察された。

　本研究の成果は、RubyコミュニティにおいてRactorの普及を促進し、並列処理に関する課題解決の一助となるものである。特に、Rubocopを用いた静的解析ルールの拡張は、Ractorを初めて利用する開発者にとってもわかりやすく、効率的な支援を提供することができる。今後の展望としては、さらなるルールの拡充や、Rubocopにおける解析機能の最適化を図り、より実用的な解析ツールを開発することが挙げられる。また、本研究で得られた知見をもとに、Rubocopを用いて、Ractor以外のRubyプログラムの支援を提供することが期待される。

# 参考文献

1. “Rubyとは“. Ruby A PROGRAMMER’S BEST FRIEND. <https://www.ruby-lang.org/ja/about/>, (参照 2025-1-21)
2. 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
3. “「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 2025-1-22)
4. “Ractor – Ruby’s Actor-like concurrent abstraction”. docs.ruby-lang.org. <https://docs.ruby-lang.org/en/master/ractor_md.html>, (参照 2025-1-24)
5. 国分崇志. “Ruby 3.3 YJITのメモリ管理とRJIT”. gikyo.jp. 2024-1-22. <https://gihyo.jp/article/2024/01/ruby3.3-jit>, (参照 2025-1-24)
6. “ライブラリ“. Ruby A PROGRAMMER’S BEST FRIEND. <https://www.ruby-lang.org/ja/libraries/>, (参照 2025-1-24)
7. “Development”. docs.rubocop.org. <https://docs.rubocop.org/rubocop/development.html>, (参照 2025-1-24)
8. “Node Types”. docs.rubocop.org. <https://docs.rubocop.org/rubocop-ast/node_types.html>, (参照 2025-1-25)

# 謝辞

本研究を進めるにあたり、熱心なご指導を賜りました甲斐宗徳教授をはじめ、協力していただいた皆様に感謝いたします。

# 付録

1. 付属プログラム bubble\_sort\_ractor.rb

require 'benchmark'

def bubble\_sort(array)

size = array.size

(size - 1).times do

(size - 1).times do |j|

array[j], array[j + 1] = array[j + 1], array[j] if array[j] > array[j + 1]

end

end

end

def bubble\_sort\_parallel(array)

# divide using pivot(MAX/2)

from\_first = 0

from\_last = NUM - 1

while from\_first < from\_last

from\_first += 1 while array[from\_first] < MAX / 2

from\_last -= 1 while array[from\_last] >= MAX / 2

next unless from\_first < from\_last

array\_from\_first = array[from\_first]

array[from\_first] = array[from\_last]

array[from\_last] = array\_from\_first

end

branch\_point = from\_first

# parallel bubble sort

Ractor.make\_shareable(array)

r1 = Ractor.new do

smaller\_part = Ractor.receive

bubble\_sort(smaller\_part)

smaller\_part

end

r2 = Ractor.new do

larger\_part = Ractor.receive

bubble\_sort(larger\_part)

larger\_part

end

r1.send(array[...branch\_point])

r2.send(array[branch\_point..])

smaller\_part = r1.take

larger\_part = r2.take

smaller\_part.concat(larger\_part)

end

def print\_array(array, max\_display = 10)

n = array.size

n.times do |i|

if i < max\_display

print "#{array[i].to\_s.rjust(5)}#{(i + 1) % 15 == 0 ? "\n" : ' '}"

elsif i == max\_display

puts "\n \*\*\*\*\*\*\*\*"

end

if i >= n - max\_display

print "#{array[i].to\_s.rjust(5)}#{(i + 1) % 15 == 0 ? "\n" : ' '}"

end

end

puts "\n"

end

raise 'Usage: ruby script.rb MAX NUM' if ARGV.size != 2

MAX = ARGV[0].to\_i

NUM = ARGV[1].to\_i

data = Array.new(NUM) { rand(MAX) }

puts '----- Before sort -----'

print\_array(data)

time = Benchmark.realtime do

data = bubble\_sort\_parallel(data)

end

puts "\n----- After Sort --#{time}sec---"

print\_array(data)

1. 付属プログラム ractor\_checker.rb

require 'parser/current'

class RactorChecker

attr\_reader :ractor\_receives, :ractor\_sends

def initialize(file\_path)

@file\_path = file\_path

@ractor\_receives = []

@ractor\_sends = []

end

def check

buffer = Parser::Source::Buffer.new(@file\_path)

buffer.source = File.read(@file\_path)

parser = Parser::CurrentRuby.new

ast = parser.parse(buffer)

analyze\_ast(ast)

end

def receive\_paired\_with\_send?(node)

exist\_receive = false

@ractor\_receives.each do |receive|

next unless node.children[0] == receive[:ractor]

exist\_receive = true

@ractor\_sends.each do |send|

return true if receive[:ractor] == send[:ractor]

end

end

return false if exist\_receive

true

end

def send\_paired\_with\_receive?(node)

@ractor\_sends.each do |send|

next unless node.children[0].children[0] == send[:ractor]

@ractor\_receives.each do |receive|

return true if send[:ractor] == receive[:ractor]

end

end

false

end

private

def analyze\_ast(node, current\_ractor = nil)

return unless node.is\_a?(Parser::AST::Node)

case node.type

when :lvasgn

return unless node.to\_json.include?('(const nil :Ractor) :receive') && node.to\_json.include?('(const nil :Ractor) :new')

current\_ractor = node.children[0]

@ractor\_receives << { ractor: current\_ractor, node: node }

when :send

return unless node.children[1] == :send

@ractor\_sends << { ractor: node.to\_json.scan(/\(lvar :(\w+)\) :send/)[0]&.first&.to\_sym, node: node }

else

node.children.each { |child| analyze\_ast(child, current\_ractor) }

end

end

1. 付属プログラム ractor\_send\_receive.rb

require 'rubocop'

require\_relative './ractor\_checker'

module RuboCop

module Cop

module Style

class RactorSendReceive < Base

extend AutoCorrector

MSG = 'Ractor.receive detected in the Ractor block '\

'but no corresponding `%<ractor>s`.send found.'.freeze

def\_node\_search :ractor\_new?, <<~PATTERN

(lvasgn $\_ractor\_name

(block

(send (const nil? :Ractor) :new)

...

)

)

PATTERN

def on\_lvasgn(node)

return unless ractor\_new?(node)

file\_path = processed\_source.file\_path

checker = RactorChecker.new(file\_path)

checker.check

return if checker.receive\_paired\_with\_send?(node)

message = message(node)

add\_offense(node, message: message) do |corrector|

corrector.insert\_after(node, "\n\n#{node.children[0]}.send()")

end

end

private

def message(node)

format(MSG, ractor: node.children[0])

end

end

end

end

end

1. 付属プログラム ractor\_receive\_send.rb

require 'rubocop'

require\_relative './ractor\_checker'

module RuboCop

module Cop

module Style

class RactorReceiveSend < Base

extend AutoCorrector

MSG = 'Found `%<ractor>s`.send ' \

'but no corresponding Ractor.receive in ractor block found.'.freeze

def\_node\_search :ractor\_new\_block?, <<~PATTERN

(lvasgn $\_ractor\_name

(block

(send (const nil? :Ractor) :new)

...

)

)

PATTERN

def\_node\_search :ractor\_send?, <<~PATTERN

(send (lvar \_) :send ...)

PATTERN

def on\_send(node)

return unless ractor\_send?(node)

file\_path = processed\_source.file\_path

checker = RactorChecker.new(file\_path)

checker.check

return if checker.send\_paired\_with\_receive?(node)

message = message(node)

add\_offense(node, message: message) do |corrector|

corrector.insert\_before(find\_ractor\_new\_block(node).children[1].children[2], "Ractor.receive\n")

end

end

private

def find\_ractor\_new\_block(node)

node.each\_ancestor.find { |ancestor| ractor\_new\_block?(ancestor) }

.each\_child\_node.find { |ancestor\_node| ractor\_new\_block?(ancestor\_node) }

end

def message(node)

format(MSG, ractor: node.children[0].children[0])

end

end

end

end

end

1. 付属プログラム ractor\_sharing\_variables.rb

require 'rubocop'

require\_relative './ractor\_external\_references\_checker'

module RuboCop

module Cop

module Style

class RactorSharingVariables < Base

MSG = 'This may be referencing variables outside of the ractor block,' \

'which will result in an error.'.freeze

def\_node\_matcher :ractor\_new?, <<~PATTERN

(block

(send (const nil? :Ractor) :new)

...

)

PATTERN

def on\_lvar(node)

return unless ractor\_new?(node.ancestors[1])

file\_path = processed\_source.file\_path

checker = RactorExternalReferencesChecker.new(file\_path)

checker.check

return unless checker.reference\_external\_variables?(node.children[0])

add\_offense(node, message: MSG)

end

end

end

end

end

1. 付属プログラム ractor\_external\_references\_checker.rb

require 'parser/current'

class RactorExternalReferencesChecker

attr\_reader :variables\_in\_ractor, :ractor\_external\_variables

def initialize(file\_path)

@file\_path = file\_path

@variables\_in\_ractor = []

@ractor\_external\_variables = []

end

def check

buffer = Parser::Source::Buffer.new(@file\_path)

buffer.source = File.read(@file\_path)

parser = Parser::CurrentRuby.new

ast = parser.parse(buffer)

analyze\_ast(ast)

end

def reference\_external\_variables?(variable\_name)

@ractor\_external\_variables.any? { |var| var[:name] == variable\_name }

end

private

def analyze\_ast(node)

return unless node.is\_a?(Parser::AST::Node)

case node.type

when :lvasgn

if node.to\_json.include?('(const nil :Ractor) :new')

find\_lvar(node)

else

@ractor\_external\_variables << { name: node.children[0] } unless @ractor\_external\_variables.any? { |var| var[:name] == node.children[0] }

end

else

node.children.each { |child| analyze\_ast(child) }

end

end

def find\_lvar(node)

return unless node.is\_a?(Parser::AST::Node)

case node.type

when :lvar

@variables\_in\_ractor << { name: node.children[0] } unless @variables\_in\_ractor.any? { |var| var[:name] == node.children[0] }

else

node.children.each { |child| find\_lvar(child) }

end

end

end

1. 付属プログラム ractor\_yield\_take.rb

require 'rubocop'

require\_relative 'ractor\_yield\_take\_checker'

module RuboCop

module Cop

module Style

class RactorYieldTake < Base

MSG = 'No take found that corresponds to yield'.freeze

def\_node\_search :ractor\_yield?, <<~PATTERN

(send (const nil? :Ractor) :yield ...)

PATTERN

def\_node\_search :ractor\_new\_block?, <<~PATTERN

(lvasgn $\_ractor\_name

(block

(send (const nil? :Ractor) :new)

...

)

)

PATTERN

def on\_send(node)

return unless ractor\_yield?(node)

file\_path = processed\_source.file\_path

checker = RactorYieldTakeChecker.new(file\_path)

checker.check

ractor\_name = node.each\_ancestor.find { |ancestor| ractor\_new\_block?(ancestor) }.children[0]

return if checker.yield\_paired\_with\_take?(ractor\_name)

message = message(node)

add\_offense(node, message: message)

end

private

def message(node)

format(MSG, ractor: node.children[0].children[0])

end

end

end

end

end

1. 付属プログラム ractor\_yield\_take\_checker.rb

require 'parser/current'

class RactorYieldTakeChecker

attr\_reader :ractor\_yields, :ractor\_takes

def initialize(file\_path)

@file\_path = file\_path

@ractor\_yields = []

@ractor\_takes = []

end

def check

buffer = Parser::Source::Buffer.new(@file\_path)

buffer.source = File.read(@file\_path)

parser = Parser::CurrentRuby.new

ast = parser.parse(buffer)

analyze\_ast(ast)

end

def yield\_paired\_with\_take?(ractor\_name)

@ractor\_takes.each { |ractor\_take| return true if ractor\_take[:ractor] == ractor\_name }

false

end

private

def analyze\_ast(node, current\_ractor = nil)

return unless node.is\_a?(Parser::AST::Node)

case node.type

when :lvasgn

return unless node.to\_json.include?('(const nil :Ractor) :yield')

current\_ractor = node.children[0]

@ractor\_yields << { ractor: current\_ractor, node: node }

when :send

return unless node.to\_json.scan(/\(lvar :(\w+)\) :take\)/).any?

@ractor\_takes << { ractor: node.to\_json.scan(/\(lvar :(\w+)\) :take\)/)[0]&.first&.to\_sym, node: node }

else

node.children.each { |child| analyze\_ast(child, current\_ractor) }

end

end

end

1. 付属プログラム ractor\_take.rb

require 'rubocop'

require\_relative 'ractor\_take\_checker'

module RuboCop

module Cop

module Style

class RactorTake < Base

MSG = 'No `%<ractor>s`.take found that corresponds to yield'.freeze

def\_node\_search :ractor\_new\_block?, <<~PATTERN

(lvasgn $\_ractor\_name

(block

(send (const nil? :Ractor) :new)

...

)

)

PATTERN

def on\_lvasgn(node)

return unless ractor\_new\_block?(node)

file\_path = processed\_source.file\_path

checker = RactorYieldTakeChecker.new(file\_path)

checker.check

return if checker.paired\_with\_take?(node.children[0])

message = message(node)

add\_offense(node, message: message)

end

private

def message(node)

format(MSG, ractor: node.children[0])

end

end

end

end

end

1. 付属プログラム ractor\_take\_checker.rb

require 'parser/current'

class RactorYieldTakeChecker

attr\_reader :ractor\_takes

def initialize(file\_path)

@file\_path = file\_path

@ractor\_takes = []

end

def check

buffer = Parser::Source::Buffer.new(@file\_path)

buffer.source = File.read(@file\_path)

parser = Parser::CurrentRuby.new

ast = parser.parse(buffer)

analyze\_ast(ast)

end

def paired\_with\_take?(ractor\_name)

return false if @ractor\_takes.empty?

@ractor\_takes.each { |ractor\_take| return true if ractor\_take[:ractor] == ractor\_name }

false

end

private

def analyze\_ast(node, current\_ractor = nil)

return unless node.is\_a?(Parser::AST::Node)

case node.type

when :send

return unless node.to\_json.scan(/\(lvar :(\w+)\) :take\)/).any?

@ractor\_takes << { ractor: node.to\_json.scan(/\(lvar :(\w+)\) :take\)/)[0]&.first&.to\_sym, node: node }

else

node.children.each { |child| analyze\_ast(child, current\_ractor) }

end

end

end