Ractorを用いたRubyの並列処理性能評価と静的解析ツールによるコード支援

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で書かれたソフトウェア、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、SaaS企業やスタートアップ企業をはじめとして、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のスレッドはグローバルインタープリターロック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。

　Ractorの登場により、複数の独立した実行単位 (Ractor) 間でスレッドを並列に実行することが可能になった。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の最適な記述方法を明らかにし、静的解析ツールによるRactorで記述されたRubyの並列処理への支援を目指す。

1. 静的解析ツールRubocopについて

　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。

　本研究では、Ractorの性能評価を行い、最適な記述方法を分析した結果をRubocopへカスタムルールとして反映することにより、Ractorを使用したRubyコードに関して支援を行う。これにより、ユーザーはコードの品質を向上することができ、早い段階でエラーを見つけることができるためバグ修正のコストを削減することができると考えている。

1. 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。

* 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。Ractorも同じ構成をとっている。

ダイアグラム 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[2]

　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。

1. 実験

　バブルソートを用いてRactorの実験を行った。これは、RubocopによるRactorで記述されたRubyの並列処理への指摘事項を明確にするためである。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。

* 1. 方法

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

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

　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。

* 1. 結果と考察

　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。

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、要素数が半分になるため、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。これはC言語の分散メモリ型のマルチプロセス処理を実現するMPI(Message Passing Interface)というライブラリを用いて、約4倍の性能向上を確認している。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。

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。また、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。

　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。1つ目は、配列の参照時にRactor間で共有するため、ロックを行うが、このロックが競合することによる速度低下である。2つ目は、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。

　以上の結果から、これらのオーバーヘッドは避けられないことがわかった。

1. Rubocopによるコード支援

　実験の結果から、Ractorによるバブルソートの並列化では配列の分割後の要素数が元の配列要素数の半分に近い状態で行えているかが、ユーザー側で考慮可能な点であると考えた。

　よって、Rubocopでは、次のような場合にRactorによるRubyの並列処理に指摘を行う方針である。Ractorによるバブルソートでの並列化を行う際に、分割された要素数がから大きく逸脱する場合である。これにより、ユーザーがRactorによるRubyの並列処理を記述する際に、最適な記述の実現を支援することが可能である。

1. まとめ

　本報告では、Ractorの性能評価に関する進捗とコード支援による今後の方針を述べた。Ractorの性能評価については、未知の部分が多いため、引き続き性能評価を通して最適な記述方法を調査していく。そして、最適な記述方法をさらに調査していき、エラーやバグが起こると予測される箇所にRubocopを用いてコード支援の実装を行なっていく方針である。

参考文献

1. “Ruby Programming Language”: <https://www.ruby-lang.org/ja/> (2024年7月27日参照可)
2. SIOS Tech.Lab ”「アクターモデル」による並列処理プログラミング入門”: <https://tech-lab.sios.jp/archives/8738> (2024年7月27日参照可)
3. GitHub “rubocop/rubocop”: <https://github.com/rubocop/rubocop> (2024年7月27日参照可)