Ruby向け並列化機構Guildの試作

○笹田 耕一[1,a)]

概要：
我々は、プログラミング言語 Ruby の次期メジャーリリースである Ruby 3 において、安全に並行・並列プログラミングを可能にする機構である Guild を開発している。スレッドプログラミングに代表される、メモリ共有型の並行プログラミングは、間違いの無いプログラムを記述することが困難であるという仮説をもとに、Guild はメモリや資源の共有をある程度制限するモデルを目指している。本発表では、この目標を実現するために、Ruby の言語仕様をどのように拡張するのか、またどのように実装すべきかについて、試作をもとに議論する。

Preliminary implementation of Guild, a parallel execution mechanism for Ruby

Koichi Sasada[1,a)]

Abstract:
We are developing Guild, a mechanism to support making safe concurrent and parallel programs for Ruby 3, the next major release of programing language Ruby. We assume that shared memory type concurrent programming model like thread programming is difficult to make correct programs. Based in this assumption, Guild will restrict sharing memory or reources. In this presentation, we will discuss how to extend the language specification of Ruby and how to implement Guild, with our preliminary implementation.

1. はじめに

本報告では、以前コンセプトを発表した Ruby 3 向けの並行処理のための抽象化である Guild [2] の試作について述べる。

我々は、Ruby を実行する Ruby 処理系（以下、CRuby）を開発している。最新版は Ruby 2.5.1 であり、2018 年末に Ruby 2.6.0 のリリースを予定している。現在、Ruby の次のメジャーバージョンである Ruby 3 の仕様について議論しており、静的解析による実行前のエラー検出、JIT コンパイルなどによる高速化、よりよい並行並列処理のサポートの三つを大きな目標としている。

並行並列処理を Ruby で記述可能にするために単純な方針は、すでに Ruby がサポートしているスレッドを並列処理可能にすることであり、我々は過去に試作を行ったこともある [3]。しかし、スレッドプログラミングは難しい。とくに、スレッド安全なプログラムを作成するのは難しい。Ruby の良さは「楽しくプログラミングできる」ことであり、並列スレッドはそのスローガンからはほど遠い。

そこで、並行並列処理のための抽象化を別に用意することで、プログラミングしやすい記述を目指している。この目標を実現するために、現在検討中の仕組みが Guild である。ユーザは Guild を複数作成し、それらは並行並列に実行される。しかし、スレッドとは異なり、共有メモリモデルをとらないため、データレースなどの問題が起こらない。ただ、完全に非共有とするのではなく、制約をつけることで共有可能なオブジェクトも導入する。

Guild の背景やコンセプトは前報告 [2] で述べているため、適宜参照されたい。本報告では読みやすさのため、前報告と多少重複する内容を記述している。本報告では、実

[1] クックパッド株式会社
[a)] ko1@cookpad.com

装を進めた上で、概念を整理した点と、いくつかの実験結果、そして残る多くの課題について述べる。

1.1 本提案の背景

現在、Ruby の安定版バージョンである Ruby 2.5 では、並行処理のために、プリエンプティブに処理を切り替えるスレッドと、アプリケーションプログラマが明示的に処理を切り替えるコルーチンをサポートしており、それぞれ `Thread` と `Fiber` クラスとして利用できる。スレッドは他の多くのプログラミング言語でのスレッド同様、共有メモリモデルで動作し、スレッド間で簡単にオブジェクトを共有することができる。

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 2.5 のスレッドは、インタプリタに 1 つ存在するジャイアントロックをもつ、たかだか 1 つのスレッドのみ実行可能となっている。そのため、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。

ジャイアントロックにより、スレッドを用いた並列処理を書けないようにしているのは、実装上の理由と言語デザイン上の理由の 2 つがある。

実装上の理由としては、処理系をスレッドセーフに記述することの困難さにある。高性能な並列処理を実現するためには、スレッド間で共有するデータについて、適切な単位で細粒度同期を必要とするが、過不足無く細粒度同期を導入するのは困難である。また、細粒度同期を工夫なく導入すると、逐次処理の性能が低下する。

言語デザイン上の理由としては、並行処理の書きやすさの問題である。これについても、2 つ理由がある。スレッドプログラミングはそもそも難しい、という点と、並列実行するスレッドプログラミングはもっと難しい、という点である。これら 2 つの点から、Ruby においてはスレッドを、並列処理を可能にするなど、より高機能にする、という方向にしないようにしている。

まず、各スレッドは様々なデータを簡単に共有することができるため、適切にプログラミングするのが一般的に困難である。共有データについて、アプリケーションプログラマが適切に同期を取る必要があるが、この同期処理についてプログラミング中に検討することは、スクリプト言語がもつ「気楽さ」とは相容れないと筆者等は考えている。何が共有データであるかを意識するのは、規模の小さなプログラムでは問題ないことが多いが、規模が大きくなると正しく意識するのは難しい。プログラムの修正が進むと、当初は共有しなかったデータが意図せず共有データとなってしまうこともあるだろう。複数人で開発していると、開発者間の意識の違いから、問題が生じることもあるだろう。スレッドによる並行並列処理は、非決定的な挙動になることが多く、デバッグが困難であると言うこともスレッドプログラミングの問題をより深刻にしている。

2 つ目の言語デザイン上の理由は、スレッドプログラミングは難しいといっても、並列処理を伴わないスレッドプログラミングは、伴う場合よりも簡単である、という理由である。Ruby 2.5 でのスレッドの実装では、ジャイアントロックの受け渡しタイミングが、プログラムの特定の箇所（具体的には、メソッドから返るタイミングや、前方ジャンプを行うタイミング）であると限定している。そのため、例えば文字列や配列の中身を破壊的に操作しているタイミングでのスレッド切り替えは考えなくてよく、その点でのスレッド安全が保障できている。本来は同期処理を導入する必要がある場合であっても、この理由により、「たまたま」動いているプログラムは、一定数あるのではないかと筆者は予想している。問題は早期に発見すべきであるためこれは欠点だ、と言うこともできるが、書き殴りのプログラムにおいて、そこそこまともに動くというのはメリットであろう。

現在の Ruby 2.5 の方針は、前報告から引用となるが、次のようにいうことができる。「このように、Ruby プログラマおよび Ruby インタプリタ開発者に優しい設計となっており、並列処理によるプログラムの性能改善を行ないたい者にとっては冷たい設計となっている。」

1.2 解決方針と目標

我々は、スレッドプログラミングは一般に共有メモリモデルであり、これが多くの問題を抱えていると考えている。プログラム中に存在する共有データを正しく把握し、適切に同期を行う必要があるからである。とくに、正しく把握することが難しい。既存研究として、ツールによる発見をサポートするアプローチや、型により陽に示すアプローチがあるが、Ruby 処理系に適用するには、実現の困難さや言語仕様との相容れなさといった問題がある。例えば、Rust のように、引数にその手の情報を記述する、といった変更は、型を記述しないことによる容易なプログラミングを目指す Ruby には受け入れられない。共有メモリモデルであっても、全てのデータは書き換え不可とする Erlang のような仕様とすれば、共有データに関する多くの問題が解決するが、Ruby に導入するのは互換性の観点から難しい。

そこで、共有メモリモデルではなく、並行実行単位で独立したメモリ空間をもつモデルを目指す。各並行実行単位は専用のチャンネルを通じ通信することでプログラムを進める。この並行実行単位を Guild とする。

共有メモリモデルを用いないため、基本的には Guild 間ではオブジェクトは共有しない。ただし、共有が許される特殊なオブジェクトを用意することで、プログラミングのしやすさも実現する。

まとめると、Guild は Ruby が実現してきたプログラム

の容易性（「気楽なプログラミング」の実現）と、現実的なインタプリタの開発・維持コストを実現するため、下記を目標としている。

(1) Ruby 2 との互換性の維持
(2) 並列処理のサポート
(3) 共有メモリモデルを用いない安全なプログラミングの実現
(4) 高速な Guild 間通信の実現
(5) 現実的な処理系開発・維持コストの実現

感覚的に述べると、Guild は「Ruby らしい楽しく簡単な並行プログラミング」の実現を目指している。

次章では、この目標を達成するための Guild の現在の設計について議論する。

2. Guild の設計

本章では、並行・並列処理を可能にする新しい抽象化である Guild の設計について議論する。

2.1 Guild の生成

Ruby 処理系を起動すると、1 つの Guild を作成する（これをメイン Guild とする）。Guild は最低 1 つのスレッドを持つ。同一 Guild 内の複数のスレッドは、Ruby 2 と同様に並行処理を行うが、ジャイアントロックによって同時に実行はされない（Ruby 2 と完全に互換となる）。Guild が異なるスレッドは、ジャイアントロックによって排他制御を行わないため、並列に実行することができる。並行処理を記述する場合は、複数 Guild を作成することで行う。

Guild の作成は、`Thread` クラスを用いたスレッドの生成と同様、ブロックを渡すことで実現する。

```
g1 = Guild.new do
  expr1
end
g2 = Guild.new do
  expr2
end
```

この例では、`g1`、`g2` という 2 つの Guild を生成している。Guild を生成すると、暗黙にスレッドも生成されるため、各コード片はそれぞれ並行に処理される。そして、それぞれ別 Guild に所属するスレッドであるため、並列処理することができる。

このとき、`Guild.new` に渡したブロック（無名関数）は自由変数などのブロック外の情報にアクセスできない隔離ブロックという特殊なブロックとして扱われる。

2.2 共有可能オブジェクトと共有不可オブジェクト

共有メモリモデルでは、スレッド間で共有するオブジェクトの発見が困難である、と前章で述べた。そこで、Guild では、基本的にオブジェクトは共有されないこととする。つまり、これまで通りの逐次プログラムを記述すると、そこで生成されるオブジェクトは Guild 間で共有されることはない共有不可オブジェクトである。どのオブジェクトが共有されるのか、精査しながら並行プログラムを記述する必要がなくなり、並行プログラミングの難しさの 1 つが解消される。

しかし、Guild 間で共有するのが自然な情報もある。例えば、複数の Guild 間で結果を集約したい場合に、共有のスコアボードを用いたいような場合である。このようなプログラミングを可能にするため、共有可能オブジェクトを用意する。共有可能オブジェクトの操作には排他制御処理を必須とすることで、スレッド安全性を高める。つまり、共有される可能性のあるオブジェクトは、言語仕様として同期処理の記述を強制されることになる。[*1]

スレッドプログラムの挙動を観察すると、多くの場合、ほとんどのオブジェクトはスレッド間で共有されない[*2]。そのため、スレッド間で共有するオブジェクトは例外的な存在であると考えられる。

共有するオブジェクトは例外的であるため、出現する場所も少ないことが期待できる。共有可能オブジェクトの利用は手間がかかるが、そもそも出現箇所も少ないため、現実的なコストで Guild を利用することができるのではないかと期待している。

例えば、逐次プログラムをスレッドで並行処理する、といったとき、コード中に共有する可能性のあるデータを全て洗い出し、それらへのアクセスに排他制御等を導入する必要がある。数十行といった小さい規模のプログラムでは難しくないが、大規模なプログラムを並行処理しようとすると、難しい。Guild では、オブジェクトの共有ができないため、必要な箇所を共有可能オブジェクトに書き換えなければ実行することができず、フェイルセーフな設計といえる。

共有可能オブジェクトは、他の共有オブジェクトへの参照しか持たないようにする。つまり、共有可能オブジェクトを通じて共有不可オブジェクトを誤って Guild 間で共有するといった事態は起こらない。

共有可能オブジェクトは次の 5 種類である。

- 共有コンテナオブジェクト
- 不変オブジェクト
- クラス・モジュールオブジェクト
- 隔離 `Proc` オブジェクト
- Guild 制御オブジェクト

[*1] Clojure 言語における書き換え可能オブジェクトと同様の考え方である。ただし、Clojure では、すべての（読み書き可能オブジェクトでの）読み書きについて、このような同期プロトコルの利用が必須であるが、Ruby における Guild では、共有可能オブジェクトでの読み書きにのみ適用されるのが異なる。

[*2] この特性を利用して、あるオブジェクトがスレッドローカルである、と解析し、その情報を最適化に利用するという研究が多数存在する。

これら以外のオブジェクトはすべて共有不可オブジェクトであり、ある唯一の Guild に所属する。

実装コストの観点からいうと、共有不可オブジェクトについては、Ruby 2 から実装をあまり変えなくてよい、という利点がある。ただし、インタプリタグローバルな情報を利用している実装（例えば、C のグローバル変数を用いた実装）は、Guild ローカルにするために変更が必要である。性能的には、細粒度排他制御を行わないため、逐次処理の性能は変らないと期待できる。

2.2.1 共有コンテナオブジェクト

複数の Guild 間で読み書き可能な状態を共有したいこともあるため、共有可能なコンテナオブジェクトを用意する。

共有コンテナオブジェクトが格納するのは、すべて共有可能オブジェクトとする。共有可能オブジェクトの読み書き時には、特定のプロトコルを用いることを必須として、一貫性の維持を強制させる。

特定のプロトコルの、もっとも原始的なアイディアは、読み書き中にロックを要求するものだろう。

```
s = SharedStorage.new
s.lock do # ブロック実行中ロックされる
  s[:key] = val
  s[:n] += 1
end

val = s[:key] # この時、s はロックされていない
              # ので、エラーになる
```

このコードでは、SharedStorage クラスを、Ruby の Hash クラス（辞書構造）のデータと似たようなものとしている。lock に渡したブロックでは、ロックを獲得しているため、s への読み書きを安全に行うことができる。ロックを獲得しないときにアクセスしようとすると、排他制御を行っていないため、例外を発生させる。このように、Guild 間での一貫性の維持をプログラマに強制させる。

アクセス時にトランザクション制御を用いる、いわゆる、ソフトウェアトランザクショナルメモリ（STM）を用いるオブジェクトについても検討している。

上記では、Hash クラスのようなデータ構造を用意したが、どのようなデータ構造が必要であるかは課題として残る。簡単なプログラムでは、キーと値に数値や文字列、シンボルといったプリミティブなデータ型しか許さない、制限されたデータ構造でも十分だろう。では、木構造のようなデータや、配列、オブジェクトとフィールドのような記述は可能にするべきだろうか。許せばプログラムの自由度を増すが、複雑性を増すため、ミスが混入する可能性が増える。

例えば、共有データが複数あるとき、データをまたがって一貫性が求められるような場合、トランザクションを適切に制御することが必要になるが、それを誤らずに制御可能に誘導するインターフェースが重要である。すべての共有コンテナオブジェクトを STM にすることで、複数の共有データにまたがったトランザクションを、ロックの順番によるデッドロックの心配なく実現することができるので、STM は有力な手法である。

しかし、一貫しなければならない処理に対して、複数トランザクションを実行してしまうようなプログラムミスを検出できない。

```
# 間違ったコード
n = s.transaction{ s[:n] }
s.transaction{ s[:n] = n+1 }

# 正しいコード
s.transaction{ s[:n] = s[:n] +1 }
```

上記コードは非常に単純な例なので、間違いようがないと思うが、例えば、これらの取得、設定メソッドを設け、メソッド内でトランザクションを独立に構成している場合、容易に間違いそうなプログラムを記述してしまう。

```
# 間違いそうな例
def get_num(s)
  s.transaction{ s[:n] }
end
def set_num s, n
  s.transaction{ s[:n] = n}
end

# 間違い
set_num( s, get_num(s) + 1 )

# 正しいコード
s.transaction do
  set_num( s, get_num(s) + 1 )
end
```

対症療法として、トランザクション内のログを取る、といった手法が考えられる。共有メモリモデルよりも範囲を絞ることができるので、実現しやすい手法である。今のところ、この問題を回避しやすい API デザインは思いついていないため、今後の課題である。

2.2.2 不変オブジェクト

不変オブジェクトは複数の Guild で共有しても、同時に読み書きが行われないため、スレッド安全である。そのため、不変オブジェクトは Guild 間で共有が可能である。

Ruby には、あるオブジェクトの書き換えを禁止する freeze というメソッドがあるが、書き換え禁止にしただけでは不変オブジェクトとはならない。書き換え禁止オブジェクト a が書き換え可能オブジェクト b への参照を持っていた場合、a を共有したとき b も同時に共有されうるためである。

```
b = [1]  # 書き換え可能な配列オブジェクト
a = [b]
a.freeze # a は書き換え禁止だが、
         # a が参照する b は書き換え可能なので
         # a は不変オブジェクトではない
c = []
c.freeze # c は不変オブジェクト
```

不変オブジェクトであるとは、そのオブジェクトが書き換え不可オブジェクトであり、そのオブジェクトから参照されるオブジェクトが共有可能オブジェクトのみであるとき、不変オブジェクトと呼ぶ。数値リテラルやシンボルオブジェクト、freeze された文字列オブジェクトなどは不変オブジェクトである。例えば、ary=[1, 2].freeze とした配列 ary は不変オブジェクトである。

2.2.3 クラス・モジュールオブジェクト

全てのオブジェクトは、オブジェクトの継承関係を管理するためのクラスやモジュールオブジェクトへの参照を保持している。共有可能オブジェクトも同様である。そのため、クラス・モジュールオブジェクトも共有可能オブジェクトとする。

ただ、クラス・モジュールオブジェクトは書き換え可能であるため、特別な対処が必要である。クラスやモジュールオブジェクトがもつ主な書き換え可能な情報は、メソッド定義、定数、クラス変数である。

メソッド定義や定数については、基本的には初期化時に設定されるため、共有しても比較的問題がない。ただし、定数が共有不可オブジェクトを参照することがあるため、特殊な取り決めが必要である。

```
class C
  Const = [] # 配列は共有不可オブジェクト
end

Guild.new do
  C::Const
  #=> 共有不可オブジェクト（配列）を取り出そ
  #   うとするが、セットした Guild と違うた
  #   め実行時例外
end

C::Const
#=> 共有不可オブジェクト（配列）を取り出す
```

そこで、Ruby 2 との互換性のため、共有不可オブジェクトを定数に代入したときは、セットした Guild 以外でアクセスしようとすると実行時例外をあげることにする[*3]。つまり、単一 Guild で実行する分には問題ないが、複数 Guild で利用しようとすると、実行時例外により停止する複数 Guild で利用する場合は、例えば上記の例では単なる配列ではなく、それに相当する共有可能オブジェクトを利用するようにプログラムを変更しなければならない。既存のプログラムは動かなくなるが、気づかずにオブジェクトを共有してしまい、同期漏れを起こす、といった事故を防ぐことができる。

クラス変数は、読み書き可能な情報であるため、定数と同様の仕組みが必要である。

先ほど、「メソッド定義や定数については、基本的には初期化時に設定されるため、共有しても比較的問題がない」と述べたが、初期化時以外に行われれば、一貫性の問題が生じる可能性がある。

この問題を回避するため、クラス定義は Guild 生成自にコピーし、変更は各 Guild で閉じる、といった仕様も検討したが、例えば Guild 間で共有オブジェクトを共有しているとき、あるメソッドは特定の Guild でしか定義されていない、もしくはあるメソッド定義が Guild 間で異なる、といった問題を生じる（なお、これは分散 Ruby（dRuby）を利用したときに生じる問題と同様である）。

利点欠点を検討した結果、現在は Guild 間で定義を共有し、変更が他の Guild に影響することを許容するモデルを採用している[*4]。ただし、クラス・モジュールへの変更をすべてトレースする機構を導入することで、問題発見を可能にする。開発のプラクティスとしては、複数 Guild 実行時は、定義を変更しないようにすることだろう（トレース機構によって、これを検出できる）。

2.2.4 隔離 Proc オブジェクト

Ruby でクロージャを表現する Proc オブジェクトは、ブロックという文法で与えたコード片と、その時のローカル変数等の環境をまとめたものである。環境は自由変数（ブロックの外側にある変数）の参照をもつため、Proc オブジェクトをそのまま共有可能オブジェクトとすると、自由変数経由で共有不可オブジェクトを誤って共有してしまう可能性がある。しかし、Guild 間で Proc を用いて手続きを手軽に共有したい、というニーズは大きい。

そこで、自由変数などの環境を共有しない隔離 Proc オブジェクトをサポートする。隔離 Proc オブジェクトは、Proc#isolate メソッドを用いて作成される。自由変数などへの参照がある場合、例外を発生させ、アプリケーションプログラマにこの自由変数などへのアクセスは許されていないことを示す。「自由変数など」とあるのは、自由変数以外にも環境は情報をもっており、それらへのアクセスをすべて禁止するためである。

現在の実装では、ローカル変数アクセス時、自由変数へのアクセスであれば例外を発生させるという、簡単な手法を用いている。しかし、Proc#isolate 実行時に与えられ

[*3] ただし、この仕様はまだ検討中であり、実際に利用しながら仕様を固めていく。

[*4] 別案として、メソッド定義にトランザクション制御を設ける、ということも検討している。

たコード片が自由変数アクセスを行うかどうか、判別可能であるので、Proc#isolate 呼び出し時に例外を発生させ、問題を早期発見できるようにする予定である。

Guild.new に渡したブロックは、暗黙的に Proc#isolate メソッドにより生成された隔離 Proc オブジェクトが利用される。

なお、Proc#isolate の呼び出しではなく、文法レベルで隔離 Proc オブジェクトを生成するということも検討しているが、まだ良い文法を発見できていない。

2.2.5 Guild 制御オブジェクト

Guild を制御するための Guild オブジェクト等は、Guild 間で共有できるようにする。Guild 間通信などに用いるが、それについては次節で説明する。

2.2.6 共有不可オブジェクト

上述した共有可能オブジェクト以外は、すべて共有不可オブジェクトである。

Ruby プログラムを記述すると、文字列や配列といったオブジェクトが大半を占める。それらは共有不可オブジェクトであるため、これらの操作にスレッドの並列処理に必要な細粒度同期が不要となり、不必要な細粒度同期による逐次実行の速度低下をふせぐことができる。

2.3 Guild 間通信

共有メモリモデルをとらないため、Guild 間で協調するためには、特別な Guild 間をつなぐ通信路が必要になる。通信路によって、情報の伝達および同期（待ち合わせ）が可能となる。そこで、Guild 間でオブジェクトを伝達する手法を提供する。

2.3.1 通信路の抽象化：メールボックスとチャンネル

通信路の抽象化の方法として、Guild ごとにメールボックスを持たせる方法と、Guild 間で共有したチャンネルを用いる方法が考えられる。前者は Erlang 言語などが利用し、後者は Go 言語などがサポートしている。また、パイプやソケットなども後者の例である。

どちらも、同期キューとしてモデル化できる。利用方法に応じて、得意不得意がある。いくつかの点について議論する。

- Guild 間で 2 種類以上の通信路が必要になった場合（例えば、データ通信路と制御通信路が欲しい場合）、チャンネルの場合は、単に 2 つのチャンネルを用意すればよい。メールボックスではユーザが、送るデータを工夫して区別可能にする必要がある（もしくは、区別可能なように、送信時に付加情報を必須とする、という選択肢もある）。また、受信するメッセージのフィルタ機能が必要である。通信路ごとに固定の名前を付ける場合、名前の衝突が起こる可能性がある。
- 送信先不在の検出がメールボックスでは容易である。送信先の Guild がエラーなどで終了している場合、メールボックスでは送信時に異常検知ができるが、単純に双方向通信を行うチャンネルオブジェクトを用いると、これはできない。片方向通信チャンネルを用いるパイプやソケットは SIGPIPE によって送信先の不在を知ることができるが、入出力の端点を Guild 間で適切に管理しようとすると、API が煩雑になる。
- 生産者消費者モデルにおいて、それぞれに対応する Guild が複数あるとき、手の空いている消費者がタスクを受け取りたい、という場合において、チャンネルでは、生産者と消費者が 1 つのチャンネルへ送信、受信を行うことで実現できる。メールボックスでは、Guild 間の仲介を行うハブ的な仕組みを別途用意する必要がある。
- ある Guild がシステムからの通知を非同期で受け取る場合（例えば、監視している Guild が異常終了した、など。Erlang でいう link(pid)）、メールボックスであれば通知先がメールボックス 1 つで済むが、チャンネルの場合、通知用チャンネルを用意しなければならない。

他にも様々な点があるだろう（指摘頂きたい）。チャンネルは柔軟で、メールボックスは単純でミスがしづらい、という印象をもっているため、現状ではメールボックスをサポートしている。具体的には、Guild#send によって、その Guild へメッセージを送信する。Guild.receive によって、自 Guild へ届いたメッセージを受信する。

```
# ping-pong の例

child_g = Guild.new(Guild.current) do |parent_g|
  obj = Guild.receive
  parent_g.send obj
end

child_g.send 1
p Guild.receive #=> 1
```

この例では、親から子 Guild へ、オブジェクト（‘1‘）を送り、それを親へ送り返している。‘Guild.new‘ に渡したパラメータは、通信路を経由してブロックパラメータとして新しく生成した Guild に渡される。

まだ、フィルタして受信する機能を設計、搭載していないため、複数の通信路をシミュレートするには、アプリケーション開発者側で記述しなければならない。

2.3.2 メッセージの送信：複製と移籍

共有可能オブジェクトを送るのは、参照を送るのみでよく、軽量である。

共有不可オブジェクトは、共有不可であるため、参照を送ってはならない。Guild では、共有不可オブジェクトの送信のために、複製と移籍の 2 つをサポートする。

複製は、送信するオブジェクトの複製を生成し、別の

Guild へ渡す。送信元と送信先で別のオブジェクトになるため、状態を共有しない。ただし、送信するオブジェクトだけでなく、そのオブジェクトから辿ることができるすべての共有不可オブジェクトを複製する必要がある。オブジェクト間の参照は循環構造になる可能性があるため注意が必要である。

移籍は、送信するオブジェクトの所属を、送信先 Guild へ移すものであり、送信元からはアクセスを禁止する。複製のように、すべてをコピーする必要は無いため軽量である。例えば、タスクを送る、結果を返すといった場合、送信元では送信するオブジェクトは、以降不要になることが多く、利用するシーンはあると考える。とくに、大きな文字列データ、ソケットオブジェクトなどの移籍を想定している。なお、移籍の前後で、オブジェクト ID の同一性は維持しない。

なお、複製と同じように、当該オブジェクトから辿ることができるすべての共有不可オブジェクトに対して移籍を行う必要がある。そのため、意図せず、参照されているオブジェクトが送信元 Guild で参照不可となり、アクセスして例外になる、という場合があり得るが、フェイルセーフを実現しているとも言える。

移籍の実装は、オブジェクトのヘッダのコピーのみ生成し、オブジェクトの実体への参照を付け替える、という手法で実現できる。

2.4 その他

Ruby にはグローバル変数など、インタプリタ内で情報を共有する仕組みがある。共有不可オブジェクトをグローバル変数などでどの箇所からも参照されるとまずいため、言語仕様に色々と手を加える必要がある。仕様変更においては、単一 Guild で実行する場合、Ruby 2 と完全に互換であることを目指す。

グローバル変数は、Guild ローカルの変数とする。ただし、`$LOAD_PATH` といった特殊なグローバル変数については判断を保留する。

3. 実装

本章では Guild の実装の方針について述べる。

インタプリタプロセスに唯一であったグローバルロックを、Guild ごとに持たせる。すでにスレッドはネイティブスレッドで実装しているため、並列計算機上では並列処理が可能である。

ガーベージコレクションは、全 Guild を停止し、マークを行う。スイープは各 Guild で遅延スイープを行う。今後、Guild ローカルな GC を行うことも検討しているが、共有可能オブジェクトをどのように扱うかが問題になる。おそらく、分散 GC の仕組みの導入が必要になる。

以降の段落は前報告からの引用である。

クラスやモジュールは特殊共有可能変数であるため、メソッドテーブルなど、これらが管理するデータは同期処理が必要になる。そのため、テーブルアクセス時にはロックなどを導入する。同様に、シンボルテーブルなどの、インタプリタでグローバルに管理しなければならないテーブルも、同期処理を挿入する必要がある。

その他、内部的に Guild 間で同時に読み書きされるデータについても、すべて適切な同期処理を挟む必要がある。例えば、インラインメソッドキャッシュがそれにあたるが、これも [3] で提案したように、更新時は新しいエントリを作るようにすると良いと思われる。

C 拡張には、並列に動作することができない、マルチスレッドセーフでないものが含まれていることが当然考えられる。そこで、既存の C 拡張はメイン Guild 以外では動作しないようにし、マルチスレッドセーフであると宣言した C 拡張のみ、その他の Guild で実行できるとする。これは、[4] で提案した手法である。

各オブジェクトには、`freeze` されたオブジェクトであるかどうかを示すフラグがあるが、参照しているオブジェクトが不変オブジェクト（もしくは特殊共有オブジェクト）である、不変オブジェクトを示すフラグがないため、これを付与する必要がある。

4. 実験

試作を用いていくつかの実験を行った。実験は、Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz（10 コア、HT により 20 CPU スレッド）を 2 つ詰む、合計 40 CPU スレッドをもつマシン上で実行させた。

まず、生成時間について調査した。比較対象として、Proc、Fiber（コルーチン）、スレッド、Unix プロセス（fork）を用いた。それぞれ、生成し、値を渡し、その値を呼び出し元へ返す、という一連の処理を、10 万回繰り返した時間を計測した（fork の場合は、1 万回繰り返した時間を 10 倍した）。fork の場合は、パイプを用いて値のやりとりを行った。

結果は、Proc: 0.09 秒、Fiber: 0.24 秒、スレッド: 3.23 秒、Guild: 6.48 秒、fork: 63.40 秒となった。スレッドと Guild はほぼ同一となるかと思ったが、約 2 倍遅い。Guild の生成と解放時、スレッド＋α の生成と解放が必要になるため、遅くなったのではないかと思われる。プロセス生成が必要な fork に比べ、Guild は 10 倍程度高速である。

次に、ある数値 n に対するフィボナッチ数を求めるワーカー Guild を複数生成し、マスター Guild からリクエストを送る、単純なプログラムを作成する実験を行った。マスターは負荷分散用 Hub へリクエストを送信し、ワーカーはハブからリクエストを取得し、結果は逆の順序で送り返す、というものである。

結果を図 **1** に示す。X 軸がワーカー Guild の数（並列度）で、Y 軸は逐次実行と比較しての速度向上率である。

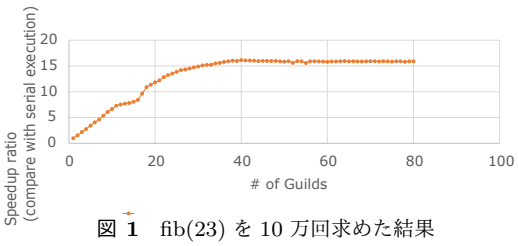

図 1 fib(23) を 10 万回求めた結果

並列度が上がると、最大 15 倍ほどの性能向上を達成できている。

現状、GC の実装が不完全で、多くの Ruby プログラムが実行できないが、例えば複数ファイルから文字列を検索するプログラムを実装すると、逐次実行と比べ 2〜6 倍遅くなった。これは、オブジェクト生成時にグローバルロックを取得するナイーブな実装を用いているためであり、オブジェクト生成をある程度 Guild ローカルで行うことにすれば徐々に改善するのではないかと思われる。

5. 関連研究

関連研究については、前報告を参照されたい。

1 点補足する。メールボックスを用いた Guild をアクターモデルと称して良いか、という点である。Koster 等の分類 [1] では、アクターモデルを 4 つに分類したうちの「プロセス」に含まれると思われるため、称しても良いと思われる。ただし、彼らの分類では、アクターモデルが共有メモリモデルを用いていても良い（アクターモデルと称する必須事項ではない）としているため、本報告ではアクターモデルという用語を用いなかった。

6. 今後の課題

課題は多く残っているが、少なくとも下記が未解決である。

Guild の終了時やエラー時の挙動の意味が不十分である。Guild がエラーによる異常終了を行ったとき、何をどのように通知するか検討が必要である。Ruby のスレッドでは、異常終了しても単に無視する、もしくは例外情報を出力する、というのがデフォルトの挙動である。他の選択肢としては、例外を親（もしくはメイン Guild）の Guild に伝搬させる、という手法がある（Go 言語の goroutine での例外の伝搬）。異常終了のキャッチを迅速に行うデザインである。Erlang では、監視しているプロセスが異常終了したとき、その通知をメールボックスに受け取る。

共有可能オブジェクトの設計が十分に進んでいない。STM を用いると述べたが、具体的にどのようなインターフェースとなるか、まだ検討していない。また、不変オブジェクトが共有不可オブジェクトを参照していないことを確認する手法について、未解決である。

（前報告からの引用）[4] で示したとおり、現在の MRI の設計（C 言語レベル）では、最大限の性能を引き出すことができないため、コンテキストへのポインタを第一引数に必ず指定するような方法が有効である。そのためには、MRI のソースコードの設計を大幅に改め、また C 拡張を記述するための Ruby C API の仕様を刷新する必要がある（ただし、互換背のために、現在利用している C API は、互換レイヤとして残す必要がある）。この変更は、決めてしまえば機械的に行なうことができる。しかし、C API の prefix である `rb_` を、別の prefix を与えねばならず、その名前付けがとくに困難である[*5]。

7. おわりに

本報告では、Ruby 3 に向けて開発している並行処理の抽象化である Guild について、その進捗を述べた。報告にあるとおり、未だ検討段階である部分が多いため、多くのご意見を頂けるとありがたい。今後、実装を進め、また報告できればと思っている。

なお、実装は https://github.com/ko1/ruby/tree/guild から取得できる。

参考文献

[1] De Koster, J., Van Cutsem, T. and De Meuter, W.: 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties, *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, New York, NY, USA, ACM, pp. 31–40 (online), DOI: 10.1145/3001886.3001890 (2016).

[2] 笹田耕一，松本行弘：Ruby 3 に向けた新しい並行実行モデルの提案，情報処理学会論文誌プログラミング（PRO），Vol. 10, No. 3, pp. 16–16（オンライン），入手先 ⟨https://ci.nii.ac.jp/naid/170000148644/⟩ (2017).

[3] 笹田耕一，松本行弘，前田敦司，並木美太郎：Ruby 用仮想マシン YARV における並列実行スレッドの実装，情報処理学会論文誌 (PRO)，Vol. 48, No. SIG 10(PRO33), pp. 1–16 (2007).

[4] 笹田耕一，卜部昌平，松本行弘，　平木敬：Ruby 用マルチ仮想マシンによる並列処理の実現，情報処理学会論文誌プログラミング（PRO），Vol. 5, No. 2, pp. 25–42 (2012).

[*5] Ruby コミュニティでは、メソッドやデータ構造の名前付けに多くの苦労が払われている。Guild という名前も、数年越しで検討した結果提案した名前であるが、すでに多くの異論を頂いている。