Ruby 3 に向けた新しい並行実行モデルの提案

○笹田 耕一[1,a)]　松本 行弘[1,b)]

概要：
マルチスレッドプログラミングは、共有するデータに対して、複数のスレッドから同時に読み込み、書き込みを行なうとデータレースなどの問題を発生させるため、排他制御などの同期を適切に行なわねばならず、難しい。この問題は、スレッドが変更可能なデータを簡単に共有できるため生じるので、そのような共有を制限する方法が必要である。
プログラミング言語 Ruby の次期メジャーバージョンである Ruby 3 では、マルチスレッドプログラミングのもつ難しさを緩和するため、“Guild” という Ruby 向けの新しい並行実行モデルを検討している。インタプリタは 1 つ以上の Guild をもつ。変更可能なすべてのオブジェクトはある 1 つの Guild に所属し、Guild 間では、変更可能なオブジェクトを共有しない。ただし、不変オブジェクトや、クラスやモジュールは共有する。Guild は 1 つ以上のスレッドをもち、異なる Guild は並列に実行する。Guild 間の通信はチャンネルを通して行ない、変更可能なオブジェクトを Guild 間で通信するにはコピーと移籍という二つの方式を提供する。移籍は、所属する Guild を変更する操作であり、単なるコピーより軽量となる。性能などの観点から、変更可能なオブジェクトを共有しなければない場合、特別なデータ構造を提供することができる。
本提案は、既存のアイデアの「いいとこ取り」を狙ったものである。本発表では、この並行実行モデルの目論見について議論する。

A proposal for a new concurrent execution model in Ruby 3

Koichi Sasada[1,a)]　Yukihiro Matsumoto[1,b)]

Abstract:
Multi-thread programming is difficult because data which shared by multiple threads requires appropriate synchronization by mutual exclusion or other techniques to avoid issues such as data race and so on. This problem is because sharing mutable data between threads easily, so that we need a way to restrict such sharing.
Ruby 3 is the next major release of programming language Ruby. Ruby 3 aims to introduce new concurrent execution model for Ruby named “Guild” to avoid multi-thread programming difficulties. An interpreter has at least one Guild. All of mutable objects belong to one guild and multiple guilds can not share mutable objects. But guilds can share immutalbe objects and class/module objects. Guild has at least one thread and threads belong to different guild can run in parallel (simultaneously). Inter guild communication is given by channel. There are two way to communicate mutable objects each other, one is copy and another is transfer membership (shortly, move). Transfering membership means changing belonging guild to another guild. It is faster than simple copying. If we need to share mutable objects because of performance, we can offer special data structure.
Our proposal is to cherry picking existing good ideas. In this presentation, we will discuss about our concurrent execution model design.

1. はじめに

プログラミング言語 Ruby は世界中で広く利用されており、とくにウェブアプリケーション開発で活用されている。ウェブアプリケーションは大規模なものもあり、インタプリタの性能が重要となっている。

我々は、Ruby を実行する Ruby 処理系（以下、CRuby）を開発しており、最新版は Ruby 2.3.1 であり、2016 年末に Ruby 2.4.0 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。

本発表は、Ruby 3 の目標の一つである、並行並列処理のサポートのために検討している *Guild* という新しい並行・並列プログラミングのための機構について報告する。

2. 背景と課題

2.1 Ruby 2 での並行並列処理のサポート

Ruby は当初よりスレッドによる並行処理をサポートしている。いわゆる「スレッド」と言われたとき、多くの人が期待するものと同様である。

CRuby が起動すると、最初に 1 つのスレッドを起動し、そのスレッドは他のスレッドを生成することができる。複数のスレッドがある場合、プリエンプションや入出力などによる処理のブロック時にコンテキストスイッチが行なわれるため、プログラマはスレッド切り替えのタイミングについて意識する必要なく、並行プログラムを Ruby で記述することができる。

すべてのスレッドはすべてのオブジェクトを共有するため、スレッド間の通信は、グローバル変数を利用する、共有する環境のローカル変数を経由する、といった様々な方法で行なうことができる。複数のスレッドから、同じオブジェクトを参照することができるため、可変オブジェクトは、その読み書き時に適切に排他制御などの同期を行なう必要がある。同期のための仕組みとして、`Mutex` や `Queue` などがある。

ここからは、実装固有の事情となる。CRuby では、スレッドの実現に OS などが提供するネイティブスレッド（例えば、POSIX 環境では Pthread）を利用している。同時実行可能なプロセッサコアが複数ある場合（そして、現状そうではない計算機を探す方が困難であろう）でも、スレッドは同時に Ruby プログラムを実行することはない。インタプリタが唯一持つ GVL（Global VM Lock、Python などでは GIL: Global Interpreter Lock）を獲得したスレッドのみが Ruby プログラムを進めることができ、コンテキストスイッチのタイミングで解放し、他のスレッドが獲得し、そのスレッドが実行を継続する。

同時並列なスレッドの動作を許さない理由は、主には同時並列に動作するプログラムを適切に記述するのが困難だからである。すでにスレッドによる並行プログラミングをサポートしているため、本質的に同種の難しさを持ってはいるが、しかしプリエンプションのタイミングをメソッドの終了時などに限定しているため、程度問題としてだが、完全に同時並列にスレッドが動作するよりも簡単に Ruby によるスレッドプログラミングが可能である。

また、インタプリタ開発も、並列プログラミングに特有の、メモリオーダリングの考慮といった難しさから解放され、格段に容易となる。とくに、CRuby は予期しない終了（例えば、セグメンテーションフォールトによる強制終了）をできる限り許さないという方針を立てている。これを実現するためには、排他制御など細粒度同期処理を適切に配置しなければならないが、これは困難である。また、必要となる細粒度同期処理のオーバヘッドを低減する必要があり、様々な提案が行なわれている。これらは、手数があれば（おそらくは）解決可能な問題だが、CRuby 開発にかけられる工数には限りがあるため、現実的ではない。

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。

なお、C で記述したプログラムを、スレッドセーフであると宣言し、並列に実行する仕組みは存在し、たとえば入出力処理や大きな多倍長整数の演算、`zlib` などの圧縮、展開処理などは並列に実行される。

JRuby や Rubinius といった、CRuby 以外の Ruby 処理系では、スレッドの並列実行を許しているが、Ruby プログラマが適切な同期を行なうことを期待している。例えば、複数のスレッドが同時に Ruby オブジェクトに読み書きしようとすると、処理系が異常終了する、といったことが起こる。また、JIT コンパイルとメモリオーダリングの問題から、逐次実行していれば起こりえないような問題が発生しており、現在、Java のようにメモリモデルを策定しようという提案が行なわれている [*1]。

2.2 課題

Ruby 3 では、主に (1) 並列処理のサポート (2) より使いやすい並行・並列プログラミングのサポートを実現したいと考えている。

なお、実現においては、過去の Ruby プログラムとの互換性を保つことが求められる。これは、既存の Ruby プロ

[1] Heroku, Inc.
[a)] ko1@heroku.com
[b)] matz@heroku.com

[*1] [Feature #12020] Documenting Ruby memory model ¡https://bugs.ruby-lang.org/issues/12020¿

グラム、および Ruby 用に記述された C 言語による拡張モジュール（C 拡張）が、Ruby 3 でも問題無く動くことを意味する。

2.2.1 並列処理のサポート

現在の MRI では、Ruby で並列処理を記述するためには、複数プロセスを用いるしか方法がない。しかし、複数プロセスを利用すると、メモリ消費が増大する、プロセス間通信により、プログラミングが面倒になる、といった問題がある。スレッドを並列実行する、という、JRuby などと同様の解決策もあるが、処理系開発に大きな負担がかかる。

2.2.2 より使いやすい並行・並列プログラミング

スレッドプログラミングは難しい。複数のスレッドが同時に一つのオブジェクトへ読み書きを行なうために生じるデータレースやレースコンディションはよく知られている。これらの問題はタイミングによって非決定的に生じるため、再現することが難しく、発生箇所の特定も困難で、デバッグが難しい。また、処理順序によってはデッドロックやライブロックを引き起こす（ただし、本報告では、デッドロック・ライブロックの排除については議論しない。というのも、この問題は、問題の発現時の状況把握がある程度可能であり、解決が他の問題よりも容易と考えられるからである）。

このような課題に対し、スレッドプログラミングを容易にするため、いろいろな手法が提案されてきた。(1) より使いやすい排他制御、同期機構の提案、並列処理に適したデータ構造の提案、(2) スレッドデバッグ手法の提案などである。

しかし、(1) の提案は適切に利用することが求められ、不用意に用いると意図しない結果が引き起こされるという点は変わらない。また、プログラムが煩雑になる。例えば、Java ではスレッドセーフなデータ構造と、そうでないデータ構造を適切に選択する必要があるが、その選択を適切に行なうのは難しい作業である。(2) はツールなどの不完全さから完全に危険を除去するのは困難であり、また実装も難しい。

プログラミング言語レベルで並行・並列処理に対応しようとするアプローチもある。排他制御などを不要にするため、スレッドなどの並行実行単位が、同一オブジェクトに対して同時に読み書きしないように制限するのがよくある手法でる。

例えば、プログラミング言語 Erlang では、データ型を読み込み専用にし、変更不可にすることで複数の並行実行単位（Erlang ではプロセス）がデータレースなどを起こさないようになっている。プログラミング言語 Clojure では、基本的には変更不可であるが、変更可能なオブジェクトの読み書きを、排他制御を強制するデータ構造（具体的にはソフトウェアトランザクショナルメモリなど）を使うように強制する。静的型システムがある言語では、複数の並行実行単位が同時にアクセスするようなことがないよう、型システムを構築するものもあり、これらはすべて実行前に解決される。

これらのアプローチを Ruby に適用可能か検討する。Ruby は変更可能なオブジェクトを操作しながら進めるプログラミング言語なので、変更を禁止するのは現実的ではない。同様に、Clojure のような、変更処理に特殊な制限を入れることも難しい。また、Ruby には型を明示的に示す方法が無く、Ruby 3 でも導入する予定はないので、これも困難である。

このように、スレッドをベースに拡張するのは、これ以上困難であるため、スレッド以外の仕組みを提供する必要がある。

解決したい課題を以下のように整理する。

(1) Ruby 2 で動作するプログラムがそのまま動作する。
(2) スレッドとは異なる（もしくは、拡張した）並行実行単位を提供する。
(3) 並行実行単位は、可能なら同時並列に動作する。
(4) 同一のオブジェクトに対して同時に読み書きを行なわないように制限することで、同期について気にせずプログラムを行なうことができる。
(5) 並行実行単位間での通信は高速に行なうことができる。
(6) 処理系開発に大きな負担はかからず、維持が容易である。

Ruby 3 では、これらの課題の解決を目標としている。

3. Guild による並行・並列制御

Ruby 3 にて、並行・並列実行をサポートする機構として、スレッドを拡張する *Guild* を提案する。Guild を用いることで、スレッドプログラミング特有の難しさを排除し、Ruby による並列プログラミングを実現する。

3.1 概要

Guild はスレッドの上位に存在する概念となる。プログラムは最低 1 つの Guild を持ち、Guild には 1 つ以上のスレッドが所属する。ある Guild に所属するスレッドは、これまでと同様に並行に実行されるが、並列には実行しない。

通常の書き換え可能なオブジェクト（以下、可変オブジェクト）は、すべて、ある一つの Guild に所属する。所属する Guild 以外のスレッドからは、そのオブジェクトへアクセスすることはできない。そのため、他の Guild に所属するスレッド同士は、一つの可変オブジェクトに対して同時に読み書きを行なうことはない。そのため、所属する Guild が異なるスレッドは、同時並列に実行することができる。現在の GVL を、Guild ごとに持つようなイメージとなる。

また、書き込みが出来ないように制限した不変オブジェク

トは、同時に読み書きを行なわれる恐れがないため、Guild 間で共有可能である。

いくつかの特殊なオブジェクトは、書き換え可能でありながら、Guild 間で共有できる。それらのオブジェクトへのアクセスは、同期を行なうために十分に制限される。これらのオブジェクトを特殊共有オブジェクトと呼ぶ。クラスやモジュール、Guild 間の通信に利用するオブジェクトや、Guild を表現する Guild オブジェクト自身がこれにあたる。

まとめると、Guild においては、すべてのオブジェクトは次の 3 通りに分類される。

可変オブジェクト　Guild 間で共有**不可**
不変オブジェクト　Guild 間で共有**可能**
特殊共有オブジェクト　Guild 間で共有**可能**

可変オブジェクトは Guild 間で共有しない、という制限により、多くのオブジェクトへのアクセスについて、同期処理に関する考慮を不要とする。インタプリタ性能を落とすこと無く、また維持も容易になる。

3.2 不変オブジェクト

不変オブジェクトは、そのオブジェクトに対して書き込みができないため、Guild 間で共有可能である。

不変オブジェクトとは、そのオブジェクトへの書き込みができず、またそのオブジェクトから辿って参照可能なすべての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必要がある。

Ruby には `freeze` というメソッドがあり、これが呼ばれたオブジェクトは書き換え不可となるが、そのオブジェクトから参照されるオブジェクトが書き換え可能である場合があるため、単に `freeze` されたオブジェクトは不変オブジェクトとはならない。例えば、`a1 = [1, 2].freeze` として生成した配列オブジェクト `a1` は、書き換えができず、参照しているオブジェクトが不変オブジェクトである整数値であるため、不変オブジェクトと言える。しかし、`a2 = [[3], 4].freeze` として生成される配列オブジェクト `a2` は、書き換え不可だが、最初の要素が書き換え可能な配列であるため、不変オブジェクトではない。

数値やシンボルオブジェクトは、他の参照を持たず、生成時から `freeze` として生成されているため、典型的な不変オブジェクトということができる。文字列は他の参照を持たないことが多いが、書き換え可能（`freeze` ではない）ため、不変オブジェクトではない。不変オブジェクトとするためには明示的な `freeze` が必要である。

不変オブジェクトを他の Guild に渡す場合、参照を渡すのみで良いため軽量である。

3.3 Guild 間の通信

可変オブジェクトがただ一つの Guild に所属する、という性質を維持するため、オブジェクトの参照を制限する必要がある。そこで、特別な通信路を用意し、これのみで通信するようにする。

3.3.1 チャンネルを用いた通信

Guild 間の通信は、チャンネルによって行い、`Guild::Channel` として提供される。チャンネルは、スレッドセーフな FIFO キューとして実装され、オブジェクトの参照を Guild を跨いだスレッド間で送り合うことができる。

オブジェクトの転送は、不変オブジェクト、および特殊共有オブジェクトについては問題無いが、可変オブジェクトは Guild 間で共有を許可しないため、そのまま転送できない。そこで、(1) コピーと (2) 移籍という、2 種類の方法を用意する。

転送時におけるコピーは直感的に理解可能な方法である。転送されるオブジェクトへの参照は、コピーされたものであるため、複数の Guild 間で共有されることはない。コピーするのはそのオブジェクトだけではなく、そのオブジェクトから辿ることができるすべての可変オブジェクトに対してコピーを行なう。コピーに際し、適切に Copy on Write の仕組みを利用することで、コピーのオーバヘッドを削減できる可能性がある。

移籍は、所属する Guild を変更する操作である。送り元 Guild からはアクセスできなくし、送り先でのみアクセス可能とする。コピーよりも高速に転送することが可能である。コピーと同様に、再帰的に移籍操作を行なう。送り元 Guild にて、転送済みオブジェクトへアクセスすると、例外が発生する。例外により、プログラマは転送済みオブジェクトへアクセスしたことを確認することができる。より詳細に述べると、移籍は、送り元の所属 Guild から離籍し、送り先に加入する、という操作になる。

なお、移籍により転送したオブジェクトの一意性は保証しない。つまり、各オブジェクトに一意に振られる `object_id` が、移籍前と移籍後に異なる可能性がある。

移籍の実現手法については後述する。

3.3.2 その他の通信

Guild 間で配列や連想配列を共有するのが自然な場合がある。現在、具体的にどの程度提供するかは決めていないが、特殊共有オブジェクトとして提供することは可能である。それらのオブジェクトへの読み書きは、特定のプロトコルを前提とすることで、適切な同期や排他制御行を強制することができる。例えば、ソフトウェアトランザクショナルメモリ的なデータ構造を提供したり、内部的にはロックフリーデータ構造を用いたコンテナデータ構造を提供する、といったことが考えられる。

また、外部のトランザクション処理の可能なデータベース（典型的には RDB）を用いて、Guild 間データを共有する、といったことも可能である。

3.4 言語仕様の拡張

可変オブジェクトは単一の Guild に所属する、という条件を満たすために、Ruby の言語仕様を拡張する必要がある。まだ決定は行なわれていないため、現在必要だろうと検討している変更点について述べる。

まず、グローバル変数は一つの Guild ローカルな変数とする。プロセスでグローバルな変数は導入しない。これにより、`$i+=1` といった、単純なカウンタ操作において、同期を行なわずに値がずれる、といったバグを排除できる。ただし、`$LOAD_PATH` といった、特殊な設定のために利用されるグローバル変数については、判断を保留する。

クラスやモジュールは Guild 間で共有する。定数の定義はクラスやモジュールに対する書き込みであるため、Guild 間で共有されることになる。また、クラス変数や、クラスやモジュールを含む特殊共有オブジェクトのインスタンス変数も、Guild 間で共有される変数となる。これらについては、(1) 共有可能なオブジェクトのみ、代入を許可する (2) 代入した Guild でのみ、アクセスを許可し、その他の Guild からのアクセスを禁止する、(3) メイン Guild（起動時に生成された Guild）からのみアクセスが可能、という 3 つの方法が考えられる。互換性を考えると、(2) もしくは (3) が候補となるが、今後検討していく。

クロージャを表現する `Proc` オブジェクトや、ローカル変数などの環境を表現する `Binding` オブジェクトをどのように扱うかも問題になる。転送時にコピーを行なうのが自然であるが、例えば、下記のように生成した `Proc` オブジェクトについて考える。

```
a = 1
pr = Proc.new{ a = 2 }
```

ここで `pr` を他の Guild に転送し、実行しても、転送元のローカル変数 `a` には変更がないことになる。このような挙動が、許容されるかは、これから議論が必要である。なお、移籍はより困難であると思われる。環境への書き込みが無い場合、これらのデータを不変データとして扱う方法を提供するのも 1 案である。

4. 実装

Guild の実装について紹介する。実際に実装したわけではなく、まだ構想段階であるため、不備が残っている可能性が十分にある。

4.1 インタプリタの並行制御

インタプリタで唯一管理していた GVL を、各 Guild が管理するようにする。これにより、異なる Guild に所属するスレッド同士では、同時並列に動作することが可能になる。

マークアンドスイープにより実装されているガーベージコレクション（GC）は並列制御の対象となる。現在は、すべてのスレッドを止め、一斉にマークを行なうよう検討している。スイープ処理は並列に行なっても良いような実装となっている。[5] で提案した手法がそのまま利用できる見通しである。

カレントワーキングディレクトリは、プロセスグローバルな 1 つを用いるか、スレッドごとに異なるワーキングディレクトリを用意するか、検討を要する。現状はプロセスグローバルであるため、互換性について留意する必要がある。

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 [5] 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、[6] 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。

4.2 Guild 間のオブジェクトの移籍

移籍は、オブジェクトの送り元からの離籍と、送り先への加入、という 2 つの操作によって行なわれる。

離籍は、次の 3 つのステップで行なわれる。(1) 離籍時に新規オブジェクトヘッダを生成し、(2) ヘッダの中身だけをコピーして、(3) 元オブジェクトをアクセス不能オブジェクトにする。つまり、離籍時には新規オブジェクトを生成するが、ヘッダの確保、および設定のみで済むため、単なるコピーよりも高速である。オブジェクトヘッダを新規生成するため、転送前と後のオブジェクトのアイデンティティ（`object_id`）の同一性は保証しない。

(3) のアクセス不能オブジェクトへの変換は、クラスを差し替えることで実現可能である。つまり、転送後、転送元でどのようなメソッドを呼ぼうとも、例外などが発生するように差し替える。

	実行時間（秒）
Single-Guild	19.45
Multi-Guild	10.45

表 1 フィボナッチ数を求めるアプリケーションの実行時間
（2 仮想ハードウェアスレッド上）

	(A) 実行時間（秒）	(B) 実行時間/opt（秒）
Single-Guild	1	1
Multi/ref	0.64	0.64
Multi/move	4.29	0.64
Multi/copy	5.16	N/A

表 2 配列の要素を足し合わせるアプリケーション
（2 仮想ハードウェアスレッド上）

5. 予備評価

まだ実装を行なっていないが、チャンネルによる通信性能を計測するため、予備評価を行なった。評価環境は Intel(R) Core(TM) i5-3380M CPU@ 2.90GHz 上で動作する Windows 7 上で動作する VirtualBox 上で動作する Ubuntu 14.04.1 LTS で行なった。対象とする Ruby 処理系は ruby 2.4.0dev [x86_64-linux] を用いた。CPU は 4 ハードウェアスレッドをサポートするが、VirtualBox では 2 ハードウェアスレッドを仮想的に提供する、という環境である。

予備評価は、2 つのアプリケーションを用いて行なった。フィボナッチ数を求める関数を実行する 4 つのワーカー Guild で実行させるアプリケーションと、整数を含めた配列を足し合わせる 4 つのワーカー Guild で実行させるアプリケーションの 2 つである。

なお、まだ Ruby プログラムを他の Guild で利用することはできないため、ワーカー Guild の処理は C で記述した。つまり、Ruby の処理におけるマルチ Guild 化のオーバヘッドを計測するものではない。このモデルにおける通信のオーバヘッドを見積もるための予備評価となっている。

5.1 フィボナッチ数を求めるアプリケーション

このアプリケーションでは、メイン Guild から整数値を渡し、各 Guild で処理を行なう。実験の結果、**表 1** に示すとおり、整数値の受け渡しはオーバヘッドがほぼなく、ほぼ 2 倍の性能向上という、理想的な評価結果を得た。

5.2 配列の要素を足し合わせるアプリケーション

10 万要素の配列をメイン Guild から 4 つのワーカ Guild に渡し、各ワーカー Guild で和を計算する、というアプリケーションである。

結果を**表 2** に示す。Multi/ref というのは、本来は許さないが、参考のため、可変オブジェクトの参照をそのまま送る、マルチスレッドを模した場合、Multi/mvoe は移籍を利用した場合、Multi/copy はコピーを行なった場合となっている。

表 2 の (A) を見ると解るとおり、Multi/ref では、単一 Guild を用いる場合に比べて 1.6 倍ほど高速となっており、並列実行の効果を確認できるが、Multi/move、Multi/copy は 4 倍、5 倍程度、逆に遅くなっている。これは、10 万要素の配列について、前者は移籍のため、後者はコピーのために処理が入るため、そのオーバヘッドにより遅くなっている。移籍の場合、すべての要素について、共有可能なオブジェクトであるかどうかをチェックするため、その走査のために遅くなっている。ただし、Multi/move のほうが Multi/copy に比べ 1.2 倍ほど高速である。

何らかの最適化が必要であるが、ここで配列のすべての要素が共有可能オブジェクトであることを既知と仮定して移籍を行なった結果が表 2 の (B) である。この結果では、移籍は参照をそのまま送る場合と同様の実行時間となった。このとことから、例えば配列の各要素が不変オブジェクトである、などの情報を備える配列を特別に用意する、もしくは特殊共有オブジェクトとして特別なデータ構造を用意する、といった工夫が有望であると考えられる。

6. 関連研究

我々は、Ruby に並列化を導入するために、いくつかの先行研究を行なってきた。

まずは、[5] では、素直に Ruby のスレッドを並列に実行する処理系の実装方法について論じている。しかし、スレッドをそのまま利用することになり、その難しさを引き継ぐことになり、使いやすい並行・並列プログラミングを支援することができない。

[7] では、複数プロセスでの並列処理を、共有メモリを用いて支援するためのシステムを提案した。本報告の移籍のアイデアは、この論文で提案している `move` と同様であるが、この論文では任意のデータについての実現方法を十分に示しておらず、またあるタイミングでは `move` でも可変状態が複数のプロセスで共有してしまうという問題があることが判明した。そのため、本報告で示したような実現手法を提案した。

[6] では、空間を完全にわける、いわゆるサンドボックスのような VM 環境を一つのインタプリタプロセス中に作成する方法について論じた。VM 間の通信については、CoW を活用することで一部のデータについて高速に転送できることを示した。Guild と違い、オブジェクトスペースは完全に分離している。なお、プログラミング言語 Racket における Place[4] が同様の提案を行なっている。

このように、プロセス、MVM、スレッドと、並行実行単位の粒度が異なる並行・並列処理機構について検討してきたが、プロセスや MVM では、各単位が分離しすぎており共有が難しく、スレッドでは共有しすぎてしまい、ス

レッドプログラミングの困難さが生じる。Guild は、これらの中間に位置し、不変オブジェクトや、特別に考慮している特殊共有オブジェクトは共有できるが、通常の可変オブジェクトは共有させない、というモデルとなっている。ほとんどのプログラムは、このモデルで問題無く記述できると想定しているが、例えば共有するクラスのロードや変更が、他の Guild が動作と並行に起こった場合、本当に使いやすいかは議論する必要がある。

実装の面から考えると、GC については、オブジェクトの空間を完全に分離したほうが、一度に対象とするオブジェクトの数は減るため性能が良くなる。また、メソッド呼び出しにおけるインラインキャッシュなど、実行に必要な可変データを共有しないよう分離するほうが、性能は向上するが、オブジェクトのコピーが行なわれるためメモリ効率は低下する。Guild は、いくらかの共有オブジェクトやデータを各 Guild が共有するためメモリ効率は高いが、同期のためのオーバヘッドが必要になる。GC はすべての Guild のオブジェクトを管理するため、性能は低くなる。これは、メモリ効率を優先し、性能を犠牲にする選択だと言える。

移籍は、オブジェクト指向における所有権・エイリアスの管理手法の一手法と捉えることもで、この分野は多くの研究がある [1]。しかし、多くの研究では、型システムを拡張することによって実現し、実行前に解析を終了することで、実行時オーバヘッドを生じない。本研究では、静的な型を持たない動的なプログラミング言語 Ruby において、どのように実現するか議論している点で、これらの研究とは異なる。

所有権の管理を、型チェックを用いずに、ランタイムでチェックを行なう研究も存在する [2], [3]。これらは、参照のたびに所有権を確認することで実現している。そして、その動的なオーバヘッドをいかに削減するか、というのが重要な要素技術として提案されている。本報告で提案した移籍は、参照時にチェックする必要がなく、これらの手法に比べその点で有利である。ただし、移籍のたびに参照先も含めてすべて可変オブジェクトであるかのチェックを行ない、新規オブジェクトを生成するという点で負荷がかかる。また、移籍前後でオブジェクトのアイデンティティを保つことができないという欠点がある。しかし、転送・移籍の回数は、オブジェクトの参照よりも十分に少なく、またオブジェクトのアイデンティティに依存したプログラムはほぼ無いだろうとの予測から、この方式を提案した。また、いくつかの場合（例えば C 拡張）において、このチェックでは可変オブジェクトの意図しない共有を回避することが難しい（回避するためには大きなオーバヘッドを生じる）ことがわかったため、この手法を採らなかった。

7. 今後の課題

まだ諸々の実装を終えていないため、まずは実装することが急務である。設計は示したが、実際のプログラミングを通じて、可変オブジェクトが複数の Guild で意図せず共有してしまうといったミスがないかなど、問題をあぶり出し、使い勝手を検証していく必要がある。

言語仕様の検討としては、クロージャの受け渡しをどのようにするかが問題である。例えば、他の Guild に行なって欲しい手続きとして、クロージャを渡すことは自然の要求である。しかし、クロージャは読み書き可能な可変オブジェクトであるため、そのままでは転送できず、コピーを行なわなければならない。この問題は、Guiild を作成し、その Guild で最初に起動されるスレッドの処理をどのように指定するか、という問題にも通じる。Java のように、Guild ごとにクラスを指定するような方法も考えられるが、毎回そのためにクラスを作成、指定せねばならず、煩雑である。Ruby におけるスレッドのように、ブロックを指定し（クロージャを渡して）Guild の最初に実行する手続きを指定したいが、可変オブジェクトの受け渡しとなる。例えば、クロージャを不変オブジェクトとして成立させるような新しい文法を導入する必要があるかもしれない。

[6] 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 [*2]。

8. おわりに

本報告では、現在 Ruby 3 のために検討中の、新しい並行・並列処理を記述するための機構である Guild について紹介した。まだ、実装を十分に行なっておらず、構想段階であるため、開発を進めながら、より良いものに育てていきたい。皆様の忌憚ないご意見を頂くことを期待している。

なお、本報告には図を用いた説明、利用方法や、想定している具体的なプログラム例などの詳細が記述されていないが、発表スライドにそれらを含めるので、参考にされたい。本報告の最新版、および発表スライドは `http://www.atdot.net/~ko1/activities/` にて公開予

[*2] 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。

定である。

参考文献

[1] Clarke, D., Noble, J. and Wrigstad, T.(eds.): *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Lecture Notes in Computer Science, Vol. 7850, Springer (2013).

[2] Claudel, B., Sabah, Q. and Stefani, J.-B.: Simple Isolation for an Actor Abstract Machine, *Formal Techniques for Distributed Objects, Components, and Systems* (Graf, S. and Viswanathan, M., eds.), Lecture Notes in Computer Science, Vol. 9039, Springer, pp. 213–227 (2015).

[3] Gordon, D. and Noble, J.: Dynamic Ownership in a Dynamic Language, *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, New York, NY, USA, ACM, pp. 41–52 (online), DOI: 10.1145/1297081.1297090 (2007).

[4] Tew, K., Swaine, J., Flatt, M., Findler, R. B. and pdinda@northwestern.edu, P. D.: Places: Adding Message-passing Parallelism to Racket, *SIGPLAN Not.*, Vol. 47, No. 2, pp. 85–96 (online), DOI: 10.1145/2168696.2047860 (2011).

[5] 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).

[6] 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).

[7] 中川博貴，笹田耕一：Ruby オブジェクトの効率的なプロセス間転送・共有機構の設計と実装，情報処理学会論文誌プログラミング（PRO），Vol. 5, No. 4, pp. 1–16 (2012).