情報処理学会論文誌　プログラミング　Vol. 5　No. 2　1–19 (Feb. 2012)

Ruby用マルチ仮想マシンによる並列処理の実現

○笹田 耕一[†1]　卜部 昌平[†2]
松本 行弘[†2]　平木 敬[†1]

我々は，高性能な Ruby 処理系の開発を行っている．Ruby 処理系は仮想マシン（VM）を用いて実現されているが，現在の VM では，同時にたかだか 1 つの Ruby スレッドのみ実行するという制約があり，並列実行をサポートしていない．また，複数の Ruby プロセスを用いて並列実行すると，計算に必要となるオブジェクトの転送がプロセス間転送となり，オーバヘッドが大きいという問題がある．そこで，我々は 1 つのプロセスに複数の VM を並列に実行できるマルチ仮想マシン（MVM）の開発を行っている．各 VM はオブジェクト空間を独立に管理するが，各 VM が同一プロセス内にあることを活かした VM 間の高速なオブジェクト転送を行うことができる．また，これらの機能を Ruby から利用するためのプログラミングインタフェース（API）を設計した．さらに，Ruby の遠隔メソッド呼び出し機構である dRuby を MVM 上で利用できるように拡張し，MVM の利用を容易に行うことができるようにした．MVM の実装は，現在の処理系との互換性を維持するため，既存の処理系のプロセス全体で共有されるデータ，例えば C 言語のグローバル変数や I/O 資源などを，VM ごとに保持するようデータ構造を変更することで行った．本論文では，開発している MVM の設計と実装について述べ，MVM を用いるための API を説明する．そして，MVM を用いた並列処理の現在の実装での性能評価について述べる．

Parallel Processing on Multiple Virtual Machine for Ruby

○ Koichi Sasada,[†1] Shohei Urabe,[†2]
Yukihiro Matsumoto[†2] and Kei Hiraki [†1]

We are developing a high-performance Ruby interpreter. The current interpreter is a bytecode-based Virtual Machine (VM). However, the current VM does not support parallel processing because the VM is restricted to running at most one Ruby thread at time. Parallel processing using multiple processes requires additional overhead to transfer objects over some inter-process communication mechanism. Based on these challenges, we are developing a Multiple Virtual Machine (MVM) implementation which can run multiple Ruby VMs simultaneously in one process. Each individual VM manages an isolated object space. For MVM, we implemented a fast object transfer technique by taking advantage of sharing the same virtual address space. We also designed a programming interface (API) to use these features. Moreover, we extended dRuby, a remote method invocation framework, to make it easy for programmers to use MVM features. In order to keep compatibility with the current Ruby interpreter, we implemented MVM by replacing the current VM's use of process-global resources, such as C global variables and I/O resources, with the use of VM-local data structures. In this paper, we will describe the design and implementation of our MVM and its API. We will also show several performance evaluations of parallel processing in Ruby using our current MVM implementation.

1. はじめに

プログラミング言語 Ruby[20] は，その使いやすさから様々な用途で広く利用されている．現在は，Ruby on Rails といったフレームワークの充実により，とくにウェブアプリケーション開発によく利用されている．Ruby は，当初はシェルスクリプトに代表されるような，使い捨てのプログラムをさっと書く，といった用途で利用されていた．しかし，オブジェクト指向機構や例外機構など，本格的な言語機構を備え，高い生産性により短期間で開発できる，という利点から，ウェブアプリケーションに限らず本格的なソフトウェア開発に利用されている．

このように，多くの用途で Ruby は本格的に利用されている．さらに Ruby の利用範囲を広げるために，我々は Ruby 用仮想マシン（VM）の開発，Ruby から C へのコンパイラの開発[22] など，性能向上の努力を行ってきた．一方，最近はマルチコア，メニーコアといった，1 つの CPU に複数コアを搭載するアーキテクチャが一般的となっており，手軽に並列計算機環境を利用することができるようになったが，Ruby には並列処理を行うための手段が十分に整備されていない．

Java や C#などのプログラミング言語では，言語が並行処理のために用意しているスレッドを並列実行できることが多い．しかし，Ruby 処理系の最新版である Ruby 1.9 インタプリタ（以降，Ruby 1.9）では，Ruby がサポートするスレッド（以降，Ruby スレッド）の並

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 株式会社ネットワーク応用通信研究所
Network Applied Communication Laboratory, Inc.

列実行は行わず，同時に 1 つの Ruby スレッドのみを実行するように制限している．これは，過去のコードとの互換性の問題や，実際に Ruby スレッドを並列に実行して高速な実行を実現するためには多くの開発作業が必要になるためである[21]．Ruby 処理系には JRuby[12] など，いくつか派生が存在するが，その中には Ruby スレッドの並列実行をサポートするものもあるが，Ruby 1.9 との互換性の問題からそれらが利用できないことも多い．また，Ruby は副作用の伴う変更を積極的に許容する言語でもあり，適切な排他制御を伴う並列スレッドはプログラミングを困難にするという問題もある[18]．

Ruby で並列処理を行うためには，複数の Ruby インタプリタ（Ruby プロセス）を立ち上げ，Ruby プロセス間で通信を行いながら実行を進めていく，という手段がある．Ruby プロセスはそれぞれ OS により独立に実行される．Ruby プロセス間の通信は `pipe` やソケットなど，OS が提供するプロセス間通信機構（IPC）を用いて行われる．これらの低レイヤでの操作を隠蔽して遠隔メソッド呼び出し機構を提供する dRuby[19] というライブラリも存在し，これを利用することである程度手軽に並列分散プログラミングを行うことができる．しかし，複数のプロセスを起動し管理するため，スレッドの生成・管理コストに比べてオーバヘッドが大きい．また，Ruby オブジェクトの転送時にはバイナリ列へ変換，IPC 経由での転送，そしてバイナリ列からの復帰というオーバヘッドが生じる．

このように，Ruby による並列実行を行うためには，利用できる機能の制限や，Ruby オブジェクトの転送にオーバヘッドがかかるといった性能の問題があった．そこで，我々は 1 つのプロセスに複数の仮想マシン（VM）を並列に実行できるマルチ仮想マシン（MVM）を開発し，Ruby で容易に並列プログラミングができる基盤を開発している（**図 1**）．

各 VM は従来の Ruby 1.9 と同様に Ruby スレッドを同時にたかだか 1 つ実行するが，MVM 全体で見ると VM の数だけ並列に Ruby スレッドを実行する．各 VM はオブジェクト空間を独立に管理する．Ruby オブジェクトは，複数のネイティブスレッドから同時にアクセスされることはないため，スレッドを並列化する際に必要となるようなオブジェクト単位での細粒度な同期制御は不要である．VM 同士の通信は，各 VM が同一プロセス内にあることを活かした，コピーオンライトなどの仕組みを積極的に活用する高速な Ruby オブジェクト転送を行うことができる．

Ruby からこれらの機能を利用するために，アプリケーションプログラミングインターフェース（API）を設計した．主に，MVM を制御するための `RubyVM` クラスと，VM 間の Ruby オブジェクト転送を担う Channel の整備である．また，dRuby の転送層を Channel を利用した VM 間転送を用いるように拡張し，MVM の機構を意識せずに従来の dRuby よりも軽量な並列プログラミングを行うことができるようにした．

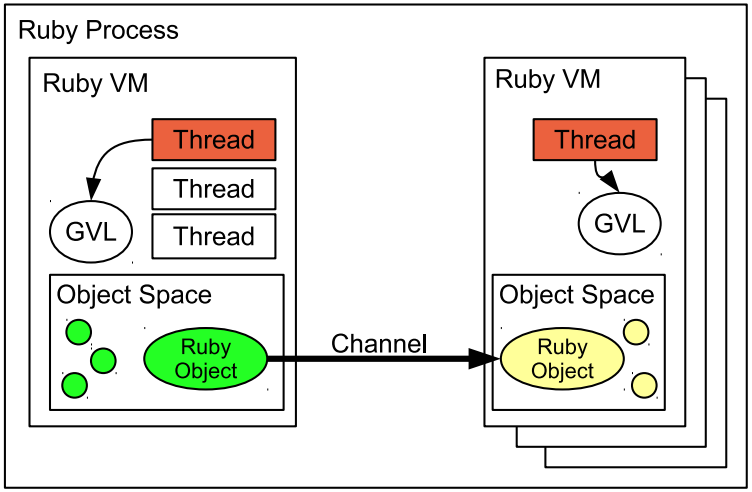


図 1　MVM の概要

MVM の実装は，現在の処理系との互換性を維持するため，Ruby 1.9 の実装をベースに，プロセス内で共有されるデータ，例えば C 言語のグローバル変数などを VM ごとに保持するようデータ構造を変更することで行った．この設計指針により，既存の Ruby プログラムとの互換性をほぼ保ちながら，MVM の機能を導入することができた．

本研究の主張点は次の 2 つである．(1) プログラミング言語機能として，Ruby に MVM という並列処理機構を設計し，実現した．(2) MVM の実装を，とくに次の 2 点について詳しく示した．(2-1) プロセス内に 1 つの VM を置くことを前提にした既存の Ruby インタプリタを，大きな互換性の問題を生じずに複数の VM を独立に置けるように拡張した．(2-2) Ruby インタプリタと同一プロセス内に VM が存在していることを活かした，Ruby オブジェクトの高速な転送方式を，Ruby VM の各データ構造を活かした形で実装した．

なお，現在の MVM の実装は開発の途上である．MVM のような機構を用いることで，他にもいくつかの性能向上の工夫，例えば起動の高速化，VM 間でメモリを共有することによる省メモリ化を行うことができるが，まだそれらは未着手である．また，6 章の評価によって，性能に問題があることがわかっている．今後，これらの改善を行っていく予定である．

以降，2 章で現状の Ruby における並列処理の記述方法について述べ，問題点を示す．3

章で MVM の概要を示し，4 章と 5 章で MVM と VM 間転送機構についての設計と実装について説明する．6 章で評価を行い，7 章で MVM に関する議論を行う．8 章で関連研究を紹介し，9 章でまとめる．

2. 現状の Ruby における並列処理

まずは，現状の Ruby でどのように並行，並列処理をサポートしているか述べ，問題点をまとめる．

2.1　並列実行スレッドを利用した Ruby の並列処理

Ruby は並行プログラミングをサポートするために言語レベルでスレッド（Ruby スレッド）を提供している．Ruby スレッドを用いることで，例えば I/O 待ちをするスレッドと計算をするスレッドというような並行処理を行うプログラムを記述することができる．

Ruby 1.9 では，Ruby スレッド 1 つにつき，OS が提供するネイティブスレッドを 1 つ用意する[21]．しかし，C 言語による既存のインタプリタの実装はスレッドセーフではないため，スレッドセーフにするために膨大な書き直しが必要となる．そのため，Ruby 1.9 では VM に 1 つ持つグローバル VM ロック（GVL）を持っている Ruby スレッドのみを実行可能としている．つまり，Ruby 1.9 では，Ruby スレッドが複数存在しても，それらが並列に実行することはない．

Ruby スレッドを並列に実行する処理系として，我々の先行研究[21] や JRuby[12]，MacRuby[16] などの他の Ruby 処理系があり，これらを利用するという手段がある．

我々の先行研究[21] を有効に利用するには，処理系の膨大な書き換えが必要である．この処理系では，C で実装されたメソッド（以降，C メソッド）がスレッドセーフであると明確に宣言されていなければ，C メソッドの実行の度に GVL を取得して実行する．この手法では，徐々に C メソッドをスレッドセーフな実装に書き換えていけば複数 Ruby スレッド実行時の並列度が向上していく，という利点がある．しかし現実的にはスレッドセーフな実装への書き換えは容易ではなく，スレッドセーフな C メソッドを増やすことは難しい．そのため，スレッドセーフではない C メソッドを起動するたびに GVL の獲得，解放処理が頻発することになり，とくにシングルスレッド実行時には性能低下が起こる．また，スレッドセーフな実装にするためには細粒度な排他制御を行う必要があるが，この排他制御を確実に，しかも効率よく実装するのは一般的に困難である．

JRuby や MacRuby は，ベースとしている Java 仮想マシン，Mac OSX のネイティブスレッド機構をそのまま利用することで効率的な並列実行可能な Ruby スレッドを実現している．とくに，JRuby がベースとしている Java 仮想マシンには軽量な排他制御を行う多くの研究成果が実装されているため，効率よく Ruby スレッドが動作することを期待できる．しかし，Ruby 1.9 と完全互換であるわけではないという制限がある．

このように，Ruby スレッドを用いた Ruby での並列プログラミングを行うには，現状では問題が残る．

さらに，適切なスレッドプログラミングは，とくに Ruby においては難しいという問題がある．複数のスレッドから同時にオブジェクトへアクセスを行うため，適切な排他制御が必要になるが，これを正しく行うのは一般的に困難である．とくに，Ruby はメタプログラミングを含めて，副作用のある破壊的操作を多く活用する傾向にあり，適切な排他制御はさらに難しいと考えられる．Ruby の目標の 1 つに,「手軽なプログラミング」があるが，並列実行スレッドが導入する困難さはこの目標に反する．

なお，Ruby 1.9 のための C メソッドを記述するために必要となるプログラミングインターフェースには，GVL を解放しながら指定された関数を実行する，というものがある．このインターフェースを利用することで，時間のかかる処理，例えば多倍長整数同士のかけ算などを，GVL を解放しながら実行し他の Ruby スレッドと並列実行可能にすることができる．しかし，これは C メソッド実装の話であり，Ruby プログラミングとは言えない．また，GVL 解放中は，Ruby 処理系に関する多くの操作，たとえばオブジェクトへのアクセス，オブジェクトの生成，他のメソッド呼び出しなどが行えない，といった制限があるため，簡単に利用できるものではない．

2.2　複数プロセスを利用した Ruby の並列処理

現在 Ruby で並列プログラミングを行うためには，Ruby インタプリタ（Ruby プロセス）を複数起動し，それらが互いに通信をしながら計算を行う手法が利用できる．

複数の Ruby プロセスは，`fork` システムコールを行う `fork` メソッドや，ユーザによる外部シェルからの実行などで生成する．Ruby プロセス間の同期や情報のやりとりは，`pipe` やソケットなど，OS が提供するプロセス間通信機構（IPC）を用いて行う．

IPC を用いる場合，Ruby オブジェクトの転送は，まずバイナリデータに変換し，IPC で送受信し，最後に Ruby オブジェクトへ逆変換して行う（**図 2**）．

このような用途に利用するために，Ruby には Ruby オブジェクトをバイナリデータに変換する `Marshal` という標準モジュールが存在する．Ruby プログラム中で `str = Marshal.dump(obj)` とすることで，Ruby オブジェクト `obj` をバイナリデータに変換できる（この場合，変換結果は変数 `str` で参照できる）．`obj` が他のオブジェクトへ

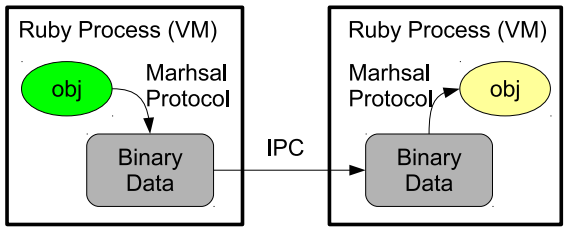


図 2　プロセス間の Ruby オブジェクトの転送

の参照を持っていた場合，参照先のオブジェクトも再帰的にバイナリデータに変換する．`Marhsla.load(str)` とすることで，逆変換を行い，バイナリデータを Ruby オブジェクトに戻すことができる．本稿では，`Marshal` クラスを利用したバイナリデータへの変換，および逆変換のことを Marshal プロトコルということにする．

一般的な複数プロセスを用いたプログラミング手法を，Ruby でほぼそのまま行うことができるため，利用のための学習コストは少ない．プロセスごとに隔離されているため，他のプロセスの影響を受けず，並列スレッドの導入に関する困難さもない．あるプロセスに障害が発生しても，そのプロセスのみ再起動するといった対処を行うことができる．プロセス間通信のプロトコルを規定すれば，Ruby インタプリタ以外のプロセスを用いた並列プログラミングも行うことができる．プロセス間通信機構としてソケットを利用している場合は，他の計算機上の Ruby プロセスとの分散計算にも拡張できる．

しかし，複数 Ruby プロセスを用いた並列プログラミングを行うには，性能上の問題がある．まず，Ruby インタプリタを複数起動する必要があるため，プロセス生成や管理のオーバヘッドが生じ，メモリなどの OS 資源も余分に必要となる．Ruby オブジェクトの転送時には，バイナリデータへの変換と復帰，そしてプロセス間通信のオーバヘッドが必要となる．これらのオーバヘッドは，プロセス数が一定で，あまり通信が行われないような場合には問題にならないが，プロセス数が頻繁に増減したり，密に Ruby プロセスが連携して並列計算を進めていく場合には問題になる．

Ruby には，IPC を利用して，遠隔メソッド呼び出しを行う dRuby[19] が存在する．dRuby は IPC と Marshal プロトコルを用いた Ruby オブジェクトの転送を自動的に行う．dRuby はプロセス間通信にソケットインターフェースを利用しているため，計算機をまたいだ通信を行うことができる．しかし，IPC を用いた Ruby オブジェクトの転送は遅い，という問題点はそのまま存在する．

3. MVM: Multiple Virtual Machine

前章で述べたとおり，現在 Ruby で並列プログラミングを行うには機能的な制限や性能上の制限がある．そこで，我々は Ruby で手軽に並列プログラミングを行うことができる仕組みであるマルチ仮想マシン（MVM）と，VM 間の Ruby オブジェクトの転送を軽量に行う Channel インターフェースの設計と実装を行っている（図 1）．

MVM は，1 つの Ruby プロセス中に複数の Ruby インタプリタ（VM）を動作させる．従来の VM と同様，各 VM は GVL を保持しており，各 VM の GVL を持つ Ruby スレッドのみ実行する．各 VM は独立して実行するため，実行可能な VM の数だけ並列度を得ることができる．

オブジェクト空間も VM ごとに保持しており，ある VM に所属する Ruby オブジェクトが他の VM の操作に影響を与えることはなく，並列スレッドで必要となる排他制御は不要である．ガーベージコレクション（以降，GC）も VM ごとに独立に実行する．MVM は従来の Ruby 1.9（Ruby 1.9.2）をベースに開発を行っているため，1VM での実行では，ほぼ互換性を維持している．

VM 間の Ruby オブジェクトの転送は，新たに設計，実装した Channel というインターフェースを用いる．Channel はキューのデータ構造であり，Ruby オブジェクトを投入し，FIFO の順で取り出すことができる．Channel は VM 間で共有することができるため，ある VM で投入した Ruby オブジェクトを，他の VM で取り出すことで，Ruby オブジェクトの転送を行うことができる．

Ruby オブジェクトの転送はコピーとして扱われる．つまり，Ruby オブジェクトを受信した VM のオブジェクト空間に新たなオブジェクトとして生成する．そのため，転送先のオブジェクトに破壊的操作を行っても，送信元のオブジェクトには影響しない．VM が同一アドレス空間内に存在することを利用して，IPC よりも，高速に Ruby オブジェクトを転送することができる．

以降，本章では MVM と Channel について，Ruby プログラムからどのように利用するか，その API について述べる．なお，本論文では，あるクラス `C` のインスタンスメソッド `foo` を，`C#foo` と表記し，シングルトンメソッド `bar`（Java などでいう静的メソッド）を `C.bar` と表記する．

3.1 VM の操作

MVM での VM 生成，実行，同期などの操作例を**図 3** に示す．図 3 には 2 つのスクリプ

```
# parentvm.rb
vm = RubyVM.new("childvm", "childvm.rb")
vm.start("a", :b, 3)
# [do something]
vm.join

# childvm.rb
RubyVM::ARGV.each{|obj|}
  p obj
}
```

図 3　VM の生成と実行，同期の例

ト parentvm.rb，childvm.rb があり，parentvm.rb を実行すると，childvm.rb で示すプログラムが子 VM として実行される，というプログラムとなっている．

Ruby プログラム中で新しく VM を生成するためには RubyVM.new メソッドを呼ぶ．このメソッドは，第 1 引数に VM の名前，第 2 引数以降に Ruby インタプリタを起動するオプション，例えば実行する Ruby プログラムが格納されたスクリプトのファイル名を指定し，渡されたオプション通りに実行する VM を生成する[⋆1]．ただし，生成直後にはまだ VM は起動しない．生成した後の VM に，実行に必要な，例えば標準入出力の設定を行う．RubyVM#start メソッドによって，生成した VM を実行する．VM は新たなネイティブスレッド上で並列に実行される．start メソッドは複数の引数を受けとり，生成された VM で RubyVM::ARGV という定数で配列として取り出すことができる．つまり，子 VM に対して初期値などを渡すことができる．図 3 では，start メソッドで渡されたオブジェクトを子 VM 側で取り出し，印字している．

生成し，実行した VM の終了を待つためには RubyVM#join メソッドを利用する．join メソッドを実行した VM は，対象となる VM の実行が終了するまでブロックする．

生成された VM は，実行が終了し，親 VM からの参照がなくなったときに，VM として確保した資源を解放する．ただし，最初に生成された VM をメイン VM として特別扱いしており，メイン VM が終了したとき，強制的に他の VM を終了させ，MVM のプロセスを終了する．

⋆1 なお，この VM 生成インターフェースは今後，より利用しやすいように改良を行っていく予定である．

3.2　VM 間の Ruby オブジェクトの転送

VM 間の Ruby オブジェクトを転送するための仕組みである Channel オブジェクトは，RubyVM::Channel.new として作成する（**図 4**）．生成した Channel オブジェクトは RubyVM#start の引数，もしくは Channel を利用して他の VM へ受け渡すことができる．

RubyVM::Channel#send で Ruby オブジェクトの送信，RubyVM::Channel#recv で送信された Ruby オブジェクトの受信を行うことができる．もし，recv したとき，Channel に何もオブジェクトが送信されていなければ，送信されるまで待機する．send によりブロックすることはない．なお，同一 VM 内で Ruby オブジェクトを send，recv することもできるが，このとき Ruby オブジェクトはコピーが渡される．

Channel の端点がどの VM に所属するか，という指定はない．どの VM も同じく送信，受信を行うことができる．受信する VM がない場合，送信された Ruby オブジェクトは Channel の中に残る．複数の VM が 1 つの Channel オブジェクトに対して送信，受信を行うことができる．例えば，1 つの VM がデータを投入し，複数の VM が受信して処理するような，マスターワーカーモデルを 1 つの Channel オブジェクトで容易に記述できる．

図 4 の例では，親 VM がまず Channel オブジェクトを生成し，それを子 VM に start メソッドを用いて渡す．親子で共有した Channel オブジェクトを用いて，親 VM が 3 つの Ruby オブジェクトを送信し，子 VM が受信して印字する．

なお，転送にあたり，転送する Ruby オブジェクトによっては Marshal プロトコルによるバイナリ変換が発生する．現在は，true，false，nil，それから Fixnum，Symbol，Float，String クラスのオブジェクト，MVM の操作に必要となる Channel オブジェクト，および要素がこれらのインスタンスである Array オブジェクトの一部について，Marshal プロトコルによるバイナリ変換なしで転送を行う．なお，とくに，String クラスのオブジェクトは，コピーオンライト機能を利用して文字列の長さ n に対して $O(1)$ で転送することができる．Marshal プロトコルではバイナリ化できないオブジェクト，例えば Proc，File クラスのオブジェクトなどは転送できない．これは，現在の実装の制限であり，今後改善していきたい．

3.3　dRuby の MVM 拡張

遠隔メソッド呼び出し機構である dRuby は，オブジェクトの転送に IPC（具体的には TCP socket）を利用するが，MVM での VM 間 Ruby オブジェクトの転送に Channel を利用するように拡張した．この対応により，MVM の機能を直接利用せずに，dRuby を用

```
# parentvm.rb
ch = RubyVM::Channel.new
vm = RubyVM.new("childvm", "childvm.rb")
vm.start(ch)
ch.send("a")
ch.send(:b)
ch.send(3)
vm.join

# childvm.rb
ch = RubyVM::ARGV.shift
p ch.recv
p ch.recv
p ch.recv
```

図 4　VM 間の Ruby オブジェクトの転送の例

いて MVM での並列プログラミングを行うことができる．

利用するには，まず `require 'drb/mvm'` としてライブラリを読み込む．そして，VM 間をつなぐ Channel オブジェクト `ch` を転送層にする場合は，`drb+mvm://[ch の ID]` という dRuby アドレスを指定することで，`ch` を転送層として利用出来るようになる．これ以外は，これまでの dRuby と同様に利用できる．

3.4　アプリケーションから Ruby を利用するためのインターフェース

MVM では，アプリケーションプログラムが複数の Ruby VM を生成する C インターフェースを用意している．

Ruby プログラムにおける `RubyVM.new`，`RubyVM#start`，`RubyVM#join` に相当するインターフェースに加えて，現在のネイティブスレッドで実行する `ruby_vm_run()`，そして実行が終了した VM の資源を解放する `ruby_vm_destruct()` というインターフェースを用意している．

このインターフェースを利用することで，アプリケーションに Ruby インタプリタを容易に組み込むことができる．Ruby 1.9 でも不可能ではなく，実際の利用例もあるが，(1) 1 つのアプリケーション（プロセス）に 1 つのインタプリタしか存在させることができない (2) 一度生成した Ruby インタプリタは完全に初期化することができない，という制限があった．MVM ではこれらの制限がなく，より自由に利用することができる．

4.　隔離された VM の構成法

MVM では，1 つのプロセス内に複数の独立した VM を作成し，並列に実行させる．本研究では，既存の処理系との互換性を維持するために Ruby 1.9 を最低限の変更で MVM 対応させることにした．Ruby 1.9 は 1 プロセスに 1 つの VM のみサポートすることを前提に設計，実装されているため，この設計に依存した箇所を修正する必要があった．

本章では，1 つのプロセス内で独立した VM を複数実行させるために，MVM をどのように設計・実装したか，とくにプロセスに VM は 1 つしか存在しないことに依存した Ruby 1.9 をどのように変更していったかを中心に説明する．

4.1　プロセスグローバルな情報

C 言語のグローバル変数（以降，グローバル変数）は，プロセス 1 つにつき 1 つしか存在しない．VM の管理している情報をグローバル変数に格納した場合，複数の VM が 1 つのグローバル変数に対して同時にアクセスしてしまうため，問題である．このように，プロセスに 1 つしかない情報に依存した処理は再設計しなければならない．

このようなプロセスグローバルな情報はいくつか種類があるため，それぞれ説明する[⋆1]．

4.1.1　C のグローバル変数

VM の管理情報をグローバル変数に格納している場合，他の VM と競合してしまい，問題である．そこで，従来グローバル変数で管理していた情報を各 VM がもつ VM 構造体にまとめ，VM 構造体経由でアクセスするように変更した．

具体的には，クラスオブジェクト，メソッドキャッシュテーブル，オブジェクト空間などがグローバル変数で管理されていたため，これを VM 構造体経由でアクセスするようにした．なお，メソッドキャッシュテーブル，オブジェクト空間はそれぞれメソッド探索を高速化するためのテーブル，メモリ管理を行うための構造体である．

クラスオブジェクトについて，少し細かく説明する．C メソッドを定義する際，`rb_define_method()` という API を用いる．この API は，第一引数にどのクラスにメソッドを定義するかを指定するが，ここでクラスオブジェクトを利用する．このような用途

⋆1 なお，プロセスの利用可能資源の上限を設定する `ulimit()` などは未対応であり，他の VM に影響する．このような機能は，例えば，メイン VM のみで実行可能とするなどの対応を検討している．

でクラスオブジェクトはよく利用されるため，グローバル変数で参照できるようにされていた．例えば，String クラスのクラスオブジェクトは rb_cString というグローバル変数に格納されていた．しかし，クラスオブジェクトは Ruby オブジェクトであるため，各 VM でそれぞれ異なる．従って，グローバル変数を用いてクラスオブジェクトを管理することはできない．

そこで，VM ごとに独立した Ruby オブジェクトを格納する領域へのポインタを返す rb_vm_specific_ptr(key) という関数を新設した．この関数は，整数値 key に対応する，VM ごとに独立した格納領域へのポインタを返す．key は事前に割り当てを行っておく．rb_cString を直接参照している過去のプログラムとのソースコードレベルの互換性を保つため，次のようなマクロを定義している（rb_vmkey_cString は事前に定義している整数値である）．

```
#define rb_cString (*rb_vm_specific_ptr(rb_vmkey_cString))
```

key は整数値であり，VM ごとに用意した配列へのインデックスとなっている．新たにこの領域で管理したいデータが生じた場合は，rb_vm_key_create() という関数で新しいキーを生成し，利用することができる．

他にも，いくつかグローバル変数を利用している箇所があったため，この仕組みを利用して修正した．なお，このグローバル変数を発見する作業は，リンカが出力するアドレスマップが参考になった．

4.1.2　シンボルテーブル

Ruby 1.9 では，一度 Ruby のシンボルを生成すると，シンボルテーブルに登録され回収されない．同じシンボルは同じオブジェクト ID が返るようにするためである．シンボルのオブジェクト ID は処理系内部でもメソッド ID などで利用されており，グローバル変数に格納されている．

MVM では，このシンボルテーブルをプロセスグローバルなデータとして扱うことにした．つまり，各 VM 上で，同じシンボルのオブジェクト ID は同一となる．この設計としたため，シンボルのオブジェクト ID を格納しているグローバル変数はとくに変更する必要がなかった．

シンボルテーブルはハッシュテーブルとして実装されている．複数の VM から並列にアクセスされるため，シンボルの追加を排他制御を行うように変更した．シンボルのオブジェクト ID は変更することがないため，キャッシュすることが可能であり，シンボルテーブル自体へのアクセスはあまり行われないため，排他制御の導入による性能低下はほぼ無いと判断した．

4.1.3　カレントワーキングディレクトリ

プロセスがファイルを開くときなどにパス探索の起点とするカレントワーキングディレクトリ（CWD）も，OS がプロセスごとに付与する情報である．ある VM が CWD を非同期に変更し，その影響がその他の VM に及ぶとプログラミングが困難になる．そこで，CWD に依存する処理を，Solaris に導入され，Linux などにも普及している openat() などの相対位置を指定できるシステムコールを用いた処理に置き換えることにした．

openat() は，第一引数に渡されたディレクトリをさすファイルディスクリプタからの相対位置でファイルを開く．CWD を変更する Dir.chdir などのメソッドが呼ばれた場合，実際の CWD を変更するのではなく，VM ごとに保持する VM ローカル CWD に，変更後のディレクトリを指すファイルディスクリプタを保持する．open メソッドなどの処理は，VM ローカル CWD と openat() を利用して行う．

相対パスを展開するときに，VM ごとに保持する CWD を表す文字列を追加する，という手法もあるが，他のプロセスの操作のタイミングによっては従来の CWD との意味が変わってしまう[*1]ため，openat() を利用することにした．ただし，openat() などが利用できない環境では相対パスに文字列を追加する，という手段をとる．

4.1.4　タイマスレッドとシグナルの配送

Ruby 1.9 では，スレッド切り替えのタイミングを制御するためにタイマスレッドを用意し，一定の間隔で GVL を保持して実行している Ruby スレッドに対し，GVL を解放して他の Ruby スレッドへ実行権を移すよう通知を行う．また，OS から送られるシグナルを，タイマスレッドが一度受けて，シグナル処理を行うように Ruby スレッドに通知する．

MVM では，このタイマスレッドを従来どおり 1 つ用意し，各 VM に一定間隔で GVL を解放するように通知するようにした．また，シグナルも同様に，一度タイマスレッドで受けてから，全ての VM へ通知するようにした（**図 5**）．ここでの変更点は，従来は 1 つの VM のみに通知していたところを，実行中のすべての VM へ通知するようにしたことである．

なお，現在は全 VM に配送しているが，VM ごとにシグナルの配送を受ける，受けないといった設定ができるように拡張する予定である．例えば，管理 VM と計算 VM を分ける場合，シグナル配送は管理 VM だけが受ければよい，などのケースが考えられるためであ

*1 例えば，CWD の名前を，他のプロセスで変更した場合，Ruby プロセス自体はその名前変更を知ることができないため，不整合が生じる．

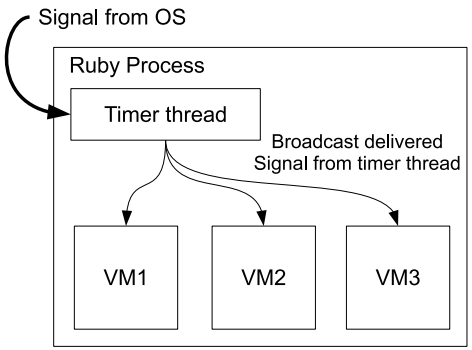


図 5　シグナルの配送

る．さらに，タイマスレッドと VM 間の通知機構を，任意の VM 間で通知が行えるように拡張することを検討している．この仕組みを用いることで，例えばマスタ VM からワーカ VM へシグナルの配送を行い，ワーカ VM の動作を終了させるといった VM の管理が可能になる．

4.2　VM 構造体

VM の状態を表す VM 構造体は，Ruby スレッド情報（VM 実行に必要なスタックの情報，プログラムカウンタなど）の集合など，Ruby プログラムの実行に必要な情報を，既存の VM と同様に保持する．

MVM のために，VM の状態（実行前，実行中，終了処理中，実行終了）や，VM 開始時に渡された値（`RubyVM::ARGV` で取得できる値）への参照，そして前節で述べた，グローバル変数で管理していた情報を格納する領域を設けている．メソッドキャッシュやオブジェクト空間への参照は，VM 構造体に新規のフィールドを追加して対応した．その他，クラスオブジェクトなどの情報は `rb_vm_specific_ptr()` 用に確保した領域に保存することにした．新規のフィールドとするか，`rb_vm_specific_ptr()` を利用するかは，その情報の参照頻度によって決定した．

実行中，VM 構造体へのポインタはネイティブスレッド環境がサポートするスレッドローカルストレージ（TLS）から辿るようにした．正確には，実行中のスレッド管理用の構造体へのポインタを TLS に格納し，そこから VM 構造体へのポインタを辿る．例えば，POSIX スレッドでは `pthread_getspecific()` などで利用できる．Linux 上の GCC コンパイラを用いる環境では，GCC 拡張である `__thread` キーワードを用いてグローバル変数を TLS として扱うことができる．

VM 構造体を TLS から辿ることで，VM 構造体が必要になる処理が発生したとき，VM 構造体へのポインタを持ち回るように，API を変更することなく VM 構造体へアクセスすることができる．

4.3　VM の操作

`RubyVM.new` を用いて VM を生成すると，VM 構造体を生成し，渡された引数を解析し，実行可能状態とする．`RubyVM#start` を実行すると，ネイティブスレッドを生成し，その上でこれまでの Ruby 1.9 と同様に Ruby プログラムの処理を進める．`RubyVM#join` を実行すると，条件変数を利用して待機対象の VM が終了するのを待つ．

生成，実行，待機，終了処理は Ruby スレッドをネイティブスレッドで実行するときに行う処理[21] とあまり変わらない．ただし，初期化，および資源の解放については従来インタプリタの起動時，終了時に行われていた処理を行う．

4.4　拡張ライブラリ，および C メソッド構築方法の変更

C メソッドなどを追加し，Ruby 処理系の機能を拡張するための仕組みである拡張ライブラリは，OS の動的ロードの機構を用いて実現されている．MVM で対応するために，拡張ライブラリの仕組みを若干変更した．

Ruby 1.9 では，拡張ライブラリの読み込み時，その初期化コードを実行し，C メソッドの定義を行う．例えば，`foo` という拡張ライブラリを読み込むと，拡張ライブラリに定義されている `Init_foo()` という関数を呼び出す．

MVM に対応した拡張ライブラリとするには，4.1 節で述べたような，プロセスグローバルな情報を排除するための処理を行わなければならない．例えば，グローバル変数を利用している場合，`rb_vm_key_create()` と `rb_vm_specific_ptr()` を組み合わせて VM ローカルなデータとして保持するように変更しなければならない．もし各 VM で共有するデータを扱う場合，適切な排他制御を行いスレッドセーフにする必要がある．

未対応な拡張ライブラリを MVM 環境でロードすることを避けるため，対応した拡張ライブラリは，例えば `foo` という名前であれば，`Init_foo()` に加え，`InitVM_foo()` という関数を必要とした．`Init_foo()` は読み込み時に一度だけ呼び出され，`InitVM_foo()` は VM で要求されたとき（Ruby で `require` されたとき）に呼ばれる．前者は `rb_vm_key_create()` のような，プロセスで 1 度だけ行えば良い処理を記述し，後者はメソッド定義など，VM ごとに行わなければならない処理を記述する．もし，`InitVM_foo()` 関数が定義されていなかった場合，MVM 未対応と判断できるため，読み込み時に例外を発生する．

ただし，未対応の拡張ライブラリが利用できないと，従来のプログラムが MVM 対応 Ruby では動作させることができない，という互換性の問題が生じる．そこで，最初に起動するメイン VM は，MVM に未対応な拡張ライブラリも利用可能とした．そのため，1VM のみ利用する場合は従来の Ruby 1.9 とほぼ互換である．MVM 未対応の拡張ライブラリを利用する処理はメイン VM に，それ以外はその他の VM で実行する，といったプログラムの構成が可能である．

5. Channel の設計と実装

本章では，各 VM 間で Ruby オブジェクトを転送するための Channel の設計と実装について述べる．

Channel オブジェクトはスレッドセーフな操作に対応したキューである．転送時には，一般的には Marshal プロトコルでバイナリフォーマットに変換して VM 間でコピーを行うが，オブジェクトの型に応じては Marshal プロトコルによる変換を行わず高速である．とくに，文字列オブジェクトについてはコピーも行わずに $O(1)$ の転送コストで済む．

5.1 基 本 設 計

Channel オブジェクトのデータ構造はスレッドセーフなキューである．キューにはデータを投入し，FIFO の順で取り出すことができる．

Channel は複数の VM から同時にアクセスされるため，Channel オブジェクトごとに，この投入と取り出し操作を排他制御し，スレッドセーフにしている．現在は POSIX Thread の条件変数とロックを利用した単純な実装としている．今後，ロックフリーデータ構造のような，より高速な構造に置き換える予定である．その際には，適切なメモリバリアを挿入し，プロセッサ間のキャッシュのコンシステンシについて考慮する必要がある．

Channel オブジェクトは，Channel オブジェクトの実体（キュー）への参照である．Channel オブジェクトの実体の寿命は参照カウントで管理しており，その値は各 VM にあるその実体を参照している Channel オブジェクトの数，および Channel オブジェクトを Channel で送信し，まだ受信されていない状態の数の和である．Channel オブジェクトがゴミ集めによって回収された，もしくは Channel オブジェクトが受信されたとき，参照カウントを減らしていき，0 になった時点で Channel オブジェクトの実体自体を回収する．ただし，参照カウントで管理していることから，循環参照となるような場合に回収されない．これは，現在の実装の制限であり，今後改善していく．

5.2 Ruby オブジェクトの送信

Ruby オブジェクトの送信は，基本的に次の順番で行う．まず，(S1) Marhsal プロトコルによりデータをバイナリ列に変換する（文字列として保持する）．そして，(S2) Channel オブジェクトに登録する．受信は次の順番で行う．(R1) Channel オブジェクトからバイナリ列に変換された文字列を取り出す．そして，(R2) Marshal プロトコルにより Ruby オブジェクトに変換する．

ただし，転送する Ruby オブジェクトが次節で述べるデータ型であった場合，(S1)，(R2) での変換処理を省略，もしくはより軽量な処理を行う．どのような処理を行うかは次節で説明する．

このように型によって復帰方法が異なるため，どのような型で送ったか，という情報も転送する．つまり，(S2) で Channel に投入するデータは，(R2) での復帰に必要になる 2 つの情報（バイナリデータ，もしくはそれに代わるもの，およびそのデータ型）である．

特定の型の場合は Marshal プロトコルが不要，もしくはより軽量な処理で置き換えるため，変換処理のオーバヘッドが少ない．また，各 VM はプロセスの仮想アドレス空間を共有するため，プロセス間通信が不要である．これらの特長から，プロセスをまたがる転送と比較して，高速に Ruby オブジェクトの転送を行うことができる．

5.3 データ型による高速化

Marshal プロトコルは，Ruby オブジェクトをファイルに保存したり，別の計算機に転送することを目的に設計されているため，同一プロセス内の VM 間の Ruby オブジェクトの通信では冗長な部分が多い．そのため，転送する Ruby オブジェクトのデータ型によって，可能であれば Marshal プロトコルを省略，もしくはより軽量な処理を行うようにした．

ここでいう型とは，Ruby 1.9 でのオブジェクトの表現方法の違いを示すものであり，オブジェクトのクラスとは若干異なる．例えば `String` クラスのオブジェクトは String 型であるが，`String` クラスを継承したクラスのインスタンスも `String` クラスのオブジェクトと同様に表現するため，同じく String 型である．

なお，現状では，まだすべての型に対して検討できているとは言えないため，今後より多くの型に対応していきたい．

本節では，対応するデータ型ごとに，どのように転送するかについて示す．

5.3.1 即　値　型

Ruby 1.9 では，`true`，`false`，`nil` オブジェクト，`Fixnum` クラスは特殊な即値として表現されている．これをここでは即値型と呼ぶ．これらの即値は各 VM で同一の表現が可能

であるため，とくに変換などを行わず，そのまま転送する．

`Symbol` クラスのオブジェクトも即値（シンボルオブジェクトの ID に関連する値）で表現される．前章で述べたとおりシンボルテーブルを全 VM で共有するため，転送元 VM のシンボルのオブジェクト ID を転送すれば，転送先の VM でも利用できるため，他の即値型と同様に即値のみ転送する．

5.3.2　Float　型

浮動小数点数を扱う Float クラスのオブジェクトは，C の `double` 型の値を持つオブジェクトである．Marshal プロトコルではマシン間での浮動小数点数のバイナリ表現の差を吸収するため，浮動小数点数を文字列表現に変換する．しかし，VM 間の通信ではこのような配慮は不要であるため，転送はこの `dobule` 型の値のみを直接転送する．

5.3.3　String 型

文字列を表現する String 型は，Ruby 1.9 がすでに実装しているコピーオンライトの機能を利用して，文字列の長さに関係無く $O(1)$ のコストで転送できるようにした．

Ruby 1.9 で String 型を表現する構造体を `RString` という．`RString` は，クラスやエンコーディングなどの基本的な情報と，文字列への実体へのポインタを持つ（**図 6** の (a)．この例では文字列`"Hello"`を持つ場合を示す[⋆1]）．文字列の実体は `malloc()` 関数によって確保されたヒープ上の領域に格納される．

Ruby 1.9 では，例えば `String#dup` などによって文字列がコピーされた場合，文字列の実体を複数の文字列オブジェクトで共有する仕組みを備えている．実体を共有している文字列に破壊的操作を行う場合は，まず文字列の実体を複製し，複製した実体に対して操作をする，いわゆるコピーオンライト（CoW）処理を行う．

図 6 の (b) では，str1 が複製元，str2 が複製された文字列オブジェクト，str0 が複製のために作成される内部オブジェクトである．str1 を複製するとき，複製先の str2 と内部オブジェクト str0 を作成する．これらはすべて同じ文字列の実体を参照するが，str1，str2 の文字列共有フラグを真に，str0 の変更不可能を示すフラグを真にしておく．str1，str2 は str0 への参照を保持する．String 型のオブジェクトの表現としては，図 6 の (a) と変わらないため，str1，str2 の参照は問題無く行うことができる．str1，str2 のどちらかに破壊的操作を行う場合，文字列実体のコピーを行い，コピーされた実体に対して破壊的操作を行い，str0 への参照を解除する．str0 への str1，str2 からの参照が切れたとき，すなわち両

⋆1 厳密には，Ruby 1.9 は文字列の長さが一定以下の場合は特殊な表現を用いるが，本稿では扱わない．

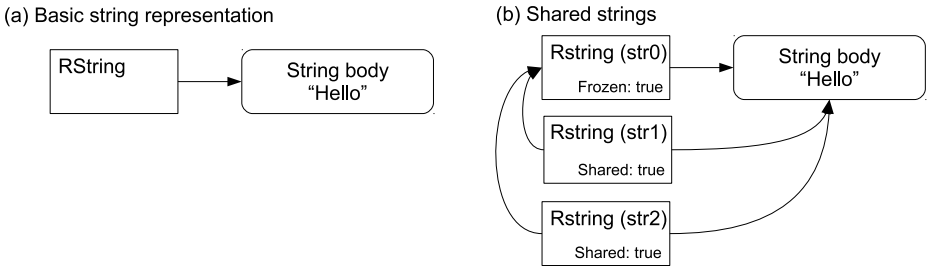


図 6　Ruby 1.9 での文字列の取り扱い

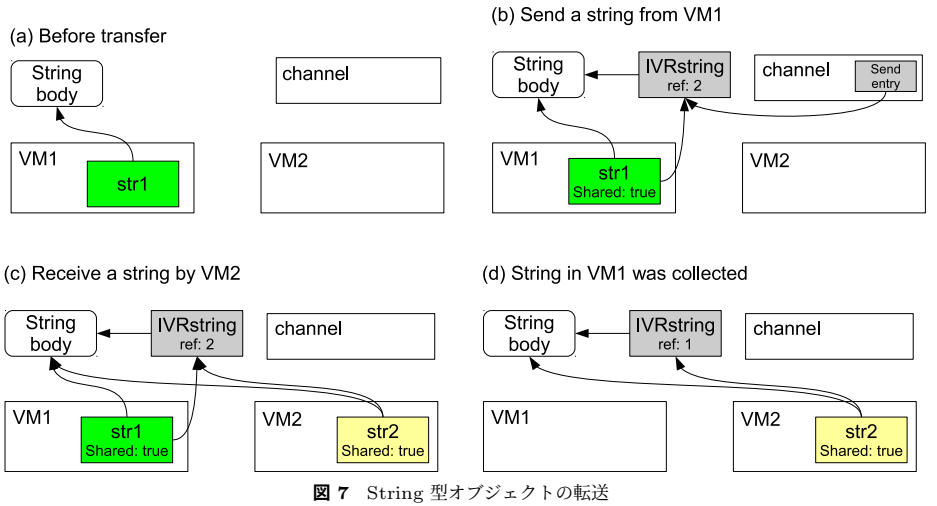


図 7　String 型オブジェクトの転送

文字列に破壊的操作が行われたとき，もしくは両オブジェクトがゴミ集めにより回収されたとき，str0 がゴミ集めにより回収される．

Channel での String 型の Ruby オブジェクトの転送は，この文字列の CoW 対応を利用して，VM 間で CoW を行うことで，文字列の実体を転送せず，$O(1)$ のコストでの転送を可能にしている．VM 間の文字列の転送を**図 7** の (a) 〜(c) に示す．そして，VM1 で転送した文字列がゴミ集めにより回収された様子を (d) に示す．

図 7(a) では VM1 に文字列 str1 があり，実体は String body であることを示す．str1 を

VM1 と VM2 で共有されている Channel オブジェクトである channel に送った後のデータ構造を図示する．前述の CoW 対応で必要になった内部オブジェクトに代わる，VM 間で共有する IVRString という管理用のデータを生成する．str1 と IVRString は文字列の実体を参照し，str1 は IVRString を参照する．channel には IVRString への参照を格納する（Send entry）．IVRString を適切に後始末するため，IVRString は参照カウンタを持っており，(b) では str1 と Send entry から参照されているので，参照カウンタに 2 が設定される．VM2 が channel から文字列を受信した状態を (c) に示す．受信して生成した文字列 str2 は，str1 と同様に文字列への実体と IVRString への参照をもつ．str1，str2 から参照されているため，IVRString の参照カウンタは 2 である．VM1 の文字列 str1 がゴミ集めにより回収されたときの図を (d) に示す．str1 回収時，IVRString の参照カウンタを 1 減らす．str2 が回収されたときなど，参照カウンタが 0 になったとき，IVRString を解放する．IVRString の参照カウンタは複数の VM から同時にアクセスされる可能性があるため，参照カウンタの増減は不可分な操作として実装している．

プロセス間通信では，文字列オブジェクトは (1) Marshal プロトコルによるバイナリデータへの変換，(2) 変換したバイナリデータの IPC を用いた転送，(3) バイナリデータから Marshal プロトコルを用いた文字列への復帰，といった手順が必要であった．しかし，MVM では主にポインタ操作のみで VM 間の文字列の転送が可能である．

なお，本節で述べたデータ型以外は，Marshal プロトコルによって Ruby オブジェクトをバイナリデータに変換するが，送信はこの文字列型のオブジェクトの転送を用いる．つまり，Marshal プロトコルによる変換オーバヘッドはかかるが，IPC のオーバヘッドは不要である．

5.3.4　Channel 型

Channel オブジェクト自身を Channel で送ることができるようにした．例えば VM1 と VM2，VM2 と VM3 がすでに Channel で接続されていた場合，VM2 経由で VM1 から VM3 へ Channel オブジェクトを送ることで，VM1 と VM3 が直接通信することを可能にする．

5.3.5　Array 型

Array 型で表現される，Ruby の配列オブジェクトを VM1 から VM2 へ送るとき，配列オブジェクトの全要素をチェックして，全ての型が本節で述べる型だった場合，単に配列のコピーを生成し，必要なら文字列などを IVRString へ変換したものに置き換えて転送する．受信側では，必要なら文字列などを IVRString から文字列オブジェクトへ変換し，VM2 での配列オブジェクトとする．

この工夫により，例えば送信する配列オブジェクトの各要素がすべて整数型であった場合，Marshal プロトコルではなく，メモリ領域のコピーのみで済ませることができる．

表 1　シングル実行の結果（実行時間，実行時間比）

	Ruby 1.9（秒）	MVM（秒）	実行時間比（1.9/MVM）
mandelbrot	8.51	13.85	0.61
tarai	1.14	1.57	0.72
pentomino	28.95	39.59	0.73
regexp	1.97	2.09	0.94

6. 評　　価

本章では，開発した MVM を評価するために，いくつかのベンチマークを行う．まずは基本的な性能を示し，次に MVM での利用が想定されるアプリケーションでのベンチマーク結果を示す．

6.1　評 価 環 境

評価環境は，Intel Xeon E7450 プロセッサ（2.40GHz）上で動作する Linux 2.6.26-2-amd64 上で行った．利用したコンパイラは gcc version 4.3.2 であり，最適化オプションは `-O3` を利用している．比較する Ruby 処理系は，MVM の開発ベースである ruby 1.9.2dev (2010-08-16 revision 29016) を用いた．

6.2　マイクロベンチマーク

6.2.1　単一実行時の性能

まずは，MVM 対応を行った VM で，単一実行の場合の性能について，Ruby 1.9 に付属するベンチマークを実行し，確認した．結果をいくつか選び，**表 1** に示す．評価は各ベンチマークを 5 回試行し，最小の時間となるものを選択した．選んだベンチマークは，マンデルブロ集合を求めるプログラム，たらい回し関数，ペントミノパズルを解くプログラム，そして正規表現検索を繰り返すプログラムである．

結果を見ると，MVM 版の Ruby インタプリタは，通常の Ruby 1.9 よりも遅くなっていることがわかる．この理由は，Ruby 1.9 ではグローバル変数に格納していたスレッド構造体や，その他管理用オブジェクトに，TLS を経由してアクセスするようにしたためである．

現在の Linux では，TLS を実現するために，いくつかの手法を併用しているが，動的リンクライブラリ中のプログラムが TLS を利用する場合，`__tls_get_addr()` という関数が実行される[9]．動的リンクライブラリではないプログラムでは，x86-64 環境ではセグメン

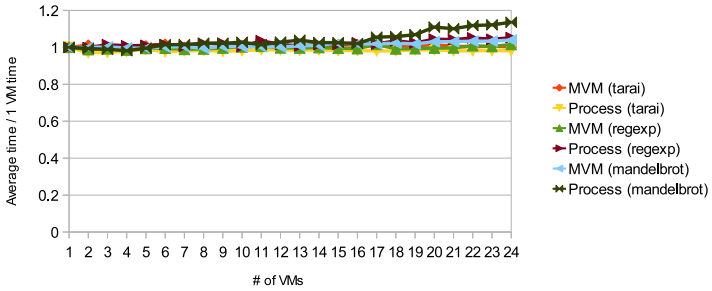


図 8　並列実行時の単体性能の評価

トレジスタを利用するなど，より高速に利用できる．Ruby インタプリタはビルド時に，インタプリタ自体を動的リンクライブラリとするように指定できる（`--enable-shared` オプション）．そこで，このオプションを用いずに，動的リンクライブラリとしないようにビルドしたところ，1 割程度の実行時間低下で済むことがわかった．

Ruby は一般的に動的リンクライブラリの形で配布・利用されるため，TLS を用いることによる速度低下は問題である．改善策としては，TLS へのアクセスを減らすため，VM 構造体，もしくはスレッド構造体を引数で持ち回るように内部構造を変更することが考えられる．また，プログラムを実行時に書き換えることで TLS アクセスを高速化する手法[14] を用いることが考えられる．

6.2.2　並列実行時の性能

前項で利用したベンチマークのうち，tarai，mandelbrot，regexp を用いて，MVM，もしくはプロセスで並列実行させたとき，VM の単体性能が変化するかを調査した．各ベンチマークについて，n 個のベンチマークを同時に実行し，各実行時間の平均を，$n = 1$ のときの実行時間で割った値を求めた．結果を**図 8** に示す．なお，計測した時間には VM もしくはプロセス起動時間は含まないように行った．

結果は，どのベンチマークもほぼ 1 倍と，あまり変わらない．とくに，MVM では複数の VM を同時実行しても各 VM の性能が低下しないことがわかった．ただし，Process (mandelbrot) のみ，並列度が上がるにつれ，若干遅くなっていることがわかった．mandelbrot ベンチマークは，一時オブジェクトを多く作成するため GC が多く起こる．複数プロセスでメモリアクセスが多いことが理由と考えられるが，詳細な理由は調査中である．

表 2　生成・終了・待ち合わせを 500 回繰り返した結果

	実行時間（秒）
スレッド	0.03
プロセス（spawn）	4.45
プロセス（fork）	1.21
VM	4.34

表 3　500VM 生成した場合のメモリ利用量

	利用メモリ（KB）	利用メモリを 500 で割った値（KB）
MVM	614,822	1,348
プロセス（spawn）	916,542	1,833
プロセス（fork）	125,004	250

6.2.3　VM 制御の性能

VM 制御の性能を調べるため，空のプログラムを実行する VM の生成，終了，そして待ち合わせを 500 回繰り返す時間を計測した．比較対象として，Ruby スレッド，Ruby インタプリタプロセスについても，同様の処理を行った．Ruby プロセスの生成は，`spawn` により Ruby インタプリタを起動する方法と，`fork` により，現在実行中のプロセスを複製する方法の 2 通りを行った．結果を**表 2** に示す．

結果を見ると，圧倒的にスレッドが高速である．初期化処理が毎回必要となるプロセス（`spawn`）と VM がとくに遅い．`fork` 版は生成処理がない分高速である．ただし，スレッドに比較すると，2 桁の差がある．これらの結果から，スレッドのように，手軽に生成・削除を行うようなプログラムは，プロセスでも VM でも速度的な問題があることが確認できた．VM の生成は，初期化済みの VM を事前に用意しておき，`fork` と同じようなコピーオンライトの機構を用意することで，さらに高速化することを検討している．

6.2.4　VM のメモリ使用量

MVM が VM を作成するにあたり，どの程度のメモリ消費があるか評価を行った．評価は，起動後に `sleep` するだけの VM を，MVM による子 VM 生成，`spwan` によるプロセス生成，`fork` によるプロセス生成の 3 種類によって，それぞれ 500 個作成し，作成前と作成後のシステムの総メモリ使用量を `free` コマンドによって測定して行った．測定にあたって，何度か繰り返すことでページキャッシュが十分あるようにし，結果が安定していることを確認している．結果を**表 3** に示す．なお，1VM あたりの利用メモリは，単純に利用メモリ量を 500 で割って小数点以下を切り捨てたものである．

結果を見ると，`fork` によってプロセスを生成した場合がもっとも小さい．`fork` によるプ

表 4　オブジェクトの転送速度

転送に利用したオブジェクト	pipe（秒）	Channel（秒）	Tunnel（秒）
1	0.41	0.35	0.32
1.1	0.52	0.34	0.34
2^{100} (Bignum)	0.45	0.56	0.34
Object.new	0.50	0.58	0.34

ロセス生成は，親プロセスのクラス定義等がそのまま受け渡るので，他の方法とは条件が違うことに注意する必要がある[*1]．新規プロセスを立ち上げる場合は，MVM が `spawn` によって新しく VM を生成した場合にくらべ，約 0.75 倍のメモリ消費となっている．いくつか理由が考えられるが，タイマスレッドを共有する点，データ領域や BSS 領域を共有する点などで，MVM のほうがメモリ消費が少ないと考えられる．なお，同じ実行ファイルであれば，テキスト領域など読み込みのみの領域は，`spawn` でも共有されることが期待できる．

6.2.5　VM 間の通信性能

最後に，Channel の通信速度について調べた．2 つの VM が Channel を通していくつかの種類のオブジェクトを転送し，それをそのまま返す（ping pong）という処理を 1 万回繰り返し，その実行時間を計測した．比較対象として，同様の処理を，2 つの Ruby プロセスを生成し，pipe を用いて同様の処理を行った場合の実行時間を計測した．pipe を用いる場合，Marshal プロトコルによってバイナリ列にして転送した．また，共有メモリを用いて，Ruby オブジェクトを転送する機構である Teleporter[23] の一部である Tunnel も評価に利用した．Tunnel は，共有メモリを用いてプロセス間でオブジェクトのコピーを行う．Channel と同様に，`Fixnum` 型などは，Marshal プロトコルを利用せず，独自のコピープロトコルを持つ．ただし，Tunnel の実装はまだ不具合を含んでおり，いくつかの評価をとることができなかった．結果を**表 4**，**図 9** に示す．

pipe を用いる場合に比べて，Channel では 1 や 0.1 といった，転送に Marshal プロトコルを用いないオブジェクトの転送がより高速であることが確認できた．また，pipe を用いる場合，システムコールを発行する必要があるが，Channel はユーザランドで処理を完結できるため，高速であると考えられる．ただし，Marshal を用いる 2^{100} を表現する Bignum オブジェクトや，`Object.new` で生成したオブジェクト転送する場合，若干遅くなることが確認できた．これは，Marshal を用いるのは pipe も Channel も同様であるが，Channel

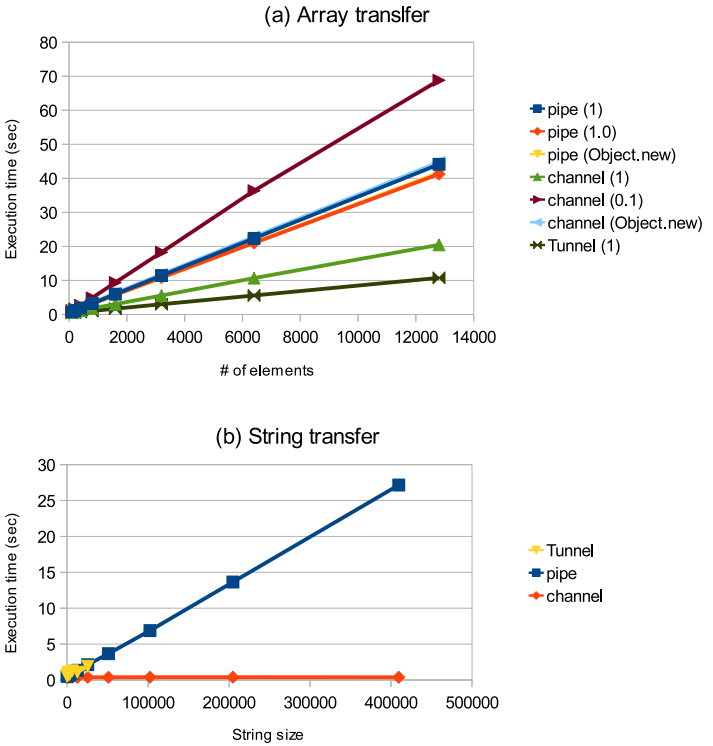


図 9　文字列と配列の転送速度

*1 なお，POSIX では，`fork` システムコール後は速やかに `exec` を行うべき（非同期シグナルセーフな操作以外は保証しない），としているため，POSIX に従うならば，`fork` 後に子プロセスで任意の Ruby プログラムを実行してはいけない．

の場合，Marshal が必要であるかのチェックルーチンが必要になり，これが pipe に比べてオーバヘッドになると考えられる．Tunnel は，共有メモリを用いる以外は，ほぼ Channel と同様の動作を行うが，若干高速である．これは，Channel の実装に非効率な点があるからだと思われる．

配列の転送は，各要素が 1，0.1，もしくは `Object.new` で生成したオブジェクトの 3 通りについて，長さを変えて Channel で転送した場合，pipe で転送した場合，Tunnel で転送した場合について計測した（図 9 (a)）．ただし，Tunnel は，実装に不具合があり，各要素が 1 の場合のみの実験となった．要素が 1（Fixnum）である場合は，Channel で転送すると Marshal 処理が不要であるため，pipe に比べて高速に実行することができた．しかし，`Object.new` の転送は，pipe を利用した転送に比べて性能は同程度であることがわかった．さらに，0.1 を要素とする配列を転送する場合，遅くなることがわかった．これは，Float 型のオブジェクトの転送の特殊化のための処理が悪影響を与えていると思われる．Channel を用いるよりもさらに高速であるため，今後この性能を目標にしていきたい[⋆1]．

Channel を用いた場合，文字列オブジェクトの転送は長さに関係無く一定時間で行うことができることが確認できた（図 9 (b)）．Tunnel を用いた評価では，実装の不具合のためすべてのデータをとることが出来なかった．pipe と比較すれば高速だが，長さに比例した転送時間がかかっていることがわかる．

6.3　アプリケーションでのベンチマーク

本節では，MVM を利用して並列化した，いくつかのアプリケーションの実行結果を示す．

6.3.1　HTML 生成アプリケーション

Ruby はウェブアプリケーションを実装する用途で使われている場合が多い．そこで，本稿では 2 つの関連アプリケーションの評価を行う．まずは，HTML テキストの生成をワーカ VM で行うプログラムである．

このプログラムは，1 つのマスタ VM および 1 つ以上のワーカ VM を用意し，マスタ VM が文字列を Channel へ送ると，ワーカ VM の 1 つがそれを受信し，文字列と eRuby テンプレート[1)] を用いた HTML 文字列の生成を行い，それを結果転送用の Channel を用いてマスタに返す．マスタが転送する文字列は，日本郵政の提供する郵便番号一覧（112,992 行）を行ごとに分割した文字列であり，ワーカはこの行を CSV フォーマットとしてパースし，

⋆1 ただし，Tunnel は共有メモリの管理に不具合があることがわかっており，単に必要な処理をしていないため高速に実行できている可能性がある．

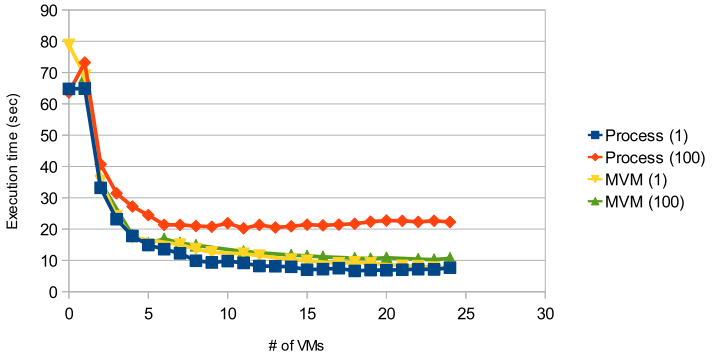


図 10　HTML 生成アプリケーションの結果

HTML テキストを生成する．生成する HTML は，1 つの `div` 要素を持つもの，および 100 個の `div` 要素を持つものの 2 種類で行った．

比較対象として，複数のプロセスを用いる例を用意した．通信には pipe を用いる．pipe を用いた場合，1 つの pipe を複数のプロセスで共有できないため，ワーカプロセス数だけ pipe を用意し，マスタがラウンドロビンで pipe を選択し，送信するようにした．

結果を**図 10** に示す．(1) が 1 つ，(100) が 100 個の `div` 要素を持つ場合である．なお，MVM 版でデータがない箇所は，処理系の不具合によりベンチマークが終了しなかった点である．

結果を見ると，複数プロセス，MVM ともに性能がスケールしていることがわかる．ただし，Process (100) は，返す HTML テキストが大きいため，性能向上が頭打ちになっている．MVM は，文字列転送のオーバヘッドがないため，(1) も (100) も同様に VM 数に応じて性能がスケールしている．

6.3.2　DB アプリケーション

今回は，複数のフロントエンド VM（Front VM）と 1 つのデータベース VM（DB VM）があり，Front VM から DB VM へ DB リクエスト送り，DB VM はリクエストに応じたレスポンスを返す，というプログラムを開発した．この実験では，データベースとして日本郵政の提供する郵便番号一覧を利用し，Front VM が郵便番号を渡すと，DB VM がその住所を返す，という処理とした．

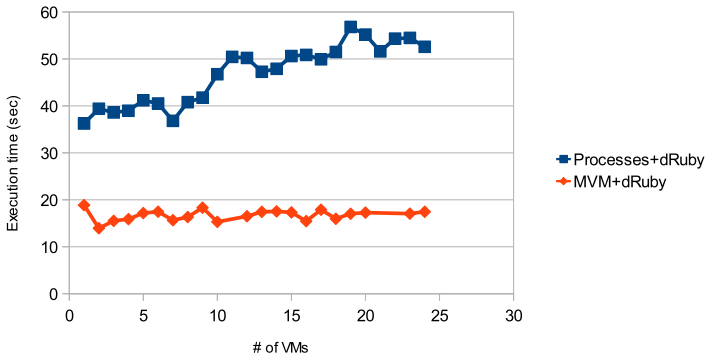


図 11　DB アプリケーションの結果

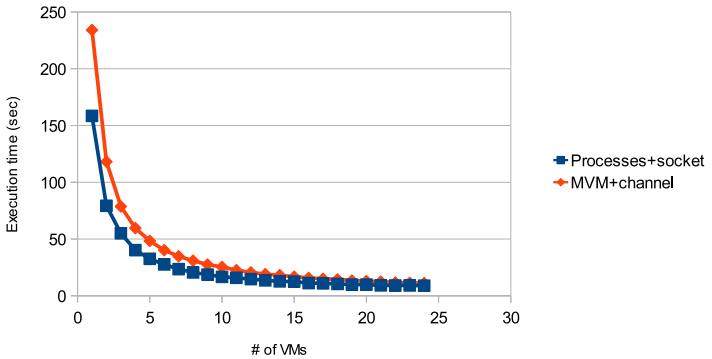


図 12　AO Bench の結果

複数プロセスを TCP socket による dRuby で通信を行い計算する版と，MVM 対応した dRuby を用いる版を用意し，比較を行った．結果を**図 11** に示す．

いずれの場合も並列度が上昇しても全体の処理クエリ数には影響を与えていないのが分かる．これは，処理のボトルネックが DB VM にあるからと考えられる．ただし通信路のオーバーヘッドは提案手法のほうが少ないことから、スループットの面では提案手法が有利であることが分かる．TCP Socket の場合に並列度が上がるにつれて効率が落ちているのが観測されたが，そのような振る舞いは提案手法には見られなかった．

6.3.3　AO Bench

最後に，数値計算の例として AO Bench を挙げる．AO Bench は三次元画像のレンダリングである．与えられたシーンのオブジェクトとその配置から，環境光遮蔽（Ambient Occlusion）を計算する．シーンの中のオブジェクトのとある点が，どの程度環境に対して暴露しているかをある程度近似的に計算している．

AO Bench では，各点ごとの計算はその隣接する点の計算とは無関係（計算結果は類似の値となるが，依存関係はない）であるので，各点を並列に計算が可能である．今回は，いくつかの点をまとめて，まとまりごとに並列化するという手法をとった．

実験結果を**図 12** に示す．比較対象として，TCP Socket を用いた従来手法での並列化と比較した．このアプリケーションでは並列化がうまく機能しており，ほぼ線形にスケールしているのが見て取れる．前節で述べたとおり，MVM には単体性能の問題があり，従来手法より速度低下が見られるが，並列化により，ほぼ遜色ない結果を得ることができた．

7.　議　　論

本章では，MVM の設計と実装について議論を行う．

7.1　MVM の現在の実装と課題

Ruby 1.9 は，スレッドプログラミングがまだ一般的ではなかった 1993 年の開発当時の，マルチスレッドに非対応なソースコードを多く含む．例えば，C 言語のグローバル変数に依存したプログラム，および拡張ライブラリの仕様である．本研究では，このようなソースコードに対して，変更を最小に，互換性を失わないように設計し，実装した点がとくに困難であった．とくに，クラスを示す値（例えば，`rb_cString`）がグローバル変数となっており，多くの箇所で参照されている．本研究では `rb_vm_specific_ptr(key)` という API を用いることで，グローバル変数から VM ローカルな領域に変更して対応した．ただし，この処理には毎回 TLS へのアクセスが必要になるため，性能の問題を生じることになった．今後，より効率的な処理へと変更していきたい．

現在の実装では，起動時に通常の Ruby 1.9 とほぼ同じ初期化処理を行う．Ruby 1.9 は，初期化処理として (0) プロセスの起動（OS による処理のため，0 と表記），(1) オブジェクトスペースの構築，(2) VM，およびスレッド構造体の構築，(3) タイマスレッドの構築，(4) 組込みクラス，メソッドのセットアップを行う．MVM では，新しい子 VM を作成する

とき，(0) の代わりに (0') 新 VM 用のネイティブスレッドの作成を行い，(3) は行わない（親 VM と共有する）．Linux のスレッド処理機構を用いる場合，(0) と (0') はほぼ実行コストが変わらず，(1)，(2)，(4) は同じく必須であるため，現状では (3) の省略を行っている．(1)，(2) は必須であるが，(4) の結果は各 VM で共有することができる．

(4) の処理は，主にクラス（モジュール）定義であり，どのクラス（モジュール）がどのようなメソッドを持っており，そのメソッドがどのような挙動（C メソッド）であるかどうかを設定する．現在，起動時には約 200 のクラス（モジュール）があり，約 1000 メソッドの定義を行う．ここで定義するクラス定義は，どの VM も持つことになるため，共有することで初期化処理を省略することができる．ただし，Ruby はプログラム中でクラス（モジュール）の定義を変更することが可能であるため，コピーオンライトのように，最初は定義を共有するが，モジュールやメソッドの定義が変更された時点で初めてコピーを作成する，といった工夫を行う必要がある．コピーオンライトを実現するためには，変更を検出するためのチェックコストが必要となるが，モジュールやメソッドの定義の変更は希であるため，ほぼ問題にならないと考えられる．

7.2　MVM の応用範囲

本稿では，MVM を Ruby に並列処理を導入するための機構として説明したが，MVM 自体には他にも様々な利用用途があると考えている．

複数の VM が同じメモリイメージを利用していたとき，VM 間でそのメモリイメージを共有することにより重複排除により省メモリ化を行うことができる．例えば，同じ Ruby スクリプトから生成されたバイトコードなどは，共有できる可能性が高いと思われる．また，初期化済みの VM を先にいくつか用意しておくなどの工夫により，起動速度の向上といったことも考えられる．今後，このような方向にも研究開発を進めていきたい．

3.4 節でも述べた通り，アプリケーションに Ruby インタプリタを埋め込み利用する，といったことが容易になる．例えば，アプリケーションの挙動をユーザプログラマに制御させるための DSL として利用するということが考えられる．現在の Ruby 1.9 では 1 つの Ruby インタプリタしか利用することができず，例えば複数の独立したインタプリタが利用したい，といった用途に利用することができない．また，1 度起動した Ruby インタプリタはプロセス終了まで初期化することができないため，状態を完全に元に戻して利用する，ということができなかった．MVM では，初期化された状態のインタプリタを用意するには，単に新しい VM を作るだけでよい．MVM を用いることで，Ruby インタプリタをより広い範囲で利用することができるようになる．

MVM はユーザレベルのみで完結することから，プロセス，もしくは同様の機能のない OS に対して，隔離された環境を提供することができる．この特長は，RubyOS[24] などの研究に応用することができると考えている．

7.3　制　　限

本稿で述べた MVM の設計と実装は，既存の Ruby 1.9 と最大限の互換性を備えることを念頭に開発を行った．そのため，TLS に VM 構造体へのポインタを格納するという設計にした．この設計により，1 つのネイティブスレッドに 1 つの VM しか同時に実行できない，という制約が生じた．もちろん，TLS に格納せず，すべての処理に VM 構造体へのポインタを引数として渡していけば，この制限は生じない．実際，Lua[10] などの言語ではそのような設計としている．しかし，このような設計とするためには，既存のコードベースを大幅に書き換える必要がある．この制限が問題となるケースは少ないと判断したため，現在の設計とした．

既存の拡張ライブラリには対応を行わなければ MVM で利用できないという制限もある．ただし，対応しなければならないのは，主にグローバル変数などの利用を行わないようにすることであり，特別難しいものではない．また，メイン VM は対応を行っていない拡張ライブラリを利用可能であるため，これまで利用できていた Ruby プログラムが利用できなくなるといった問題はない．

複数の VM を同一のプロセス内で実行するため，例えば VM にバグがあり，強制終了してしまうような場合は，他の VM も同時に強制終了してしまうという問題がある．現在の実装では例えば C メソッド中で，VM 終了時にメモリやファイルディスクリプタなどの資源を適切に解放しないように書いてしまった場合，資源がリークしてしまう，という問題がある．従来は OS のプロセス終了時の資源解放により問題とならなかったが，現在の実装ではこれらの資源を解放することができない．これらの問題は MVM の品質によるものであり，利用用途に応じてこのリスクを勘案する必要がある．

8. 関 連 研 究

MVM という考え方は，他のプログラミング言語でも用いられており，例えば Java では JSR121 Application Isolation[3),7),8)] という規格があり，また実装手法が提案されている．JSR121 では，1 つの JVM に isolation という実行単位を複数独立に存在させることができる．Link によって isolation 間の通信が可能である．isolation と Link は，それぞれ本研究における VM と Channel に相当する．しかし，JSR121 の目的は複数のアプリケーショ

ンを資源を共有しながら存在させることを目的としているため，並列処理を目的とする本研究とは目指している点が異なる．例えば，Link もバッファリングせずに送り側が受信されるまでブロックするようになっており，isolation 間の同期を第一目的としている．本研究での Channel は，キューによりバッファリングを行うため，受信者が受信しなくてもブロックすることはなく，例えば生産者・消費者のような並列計算プログラムを容易に記述できる．JSR121 を拡張して，軽量な遠隔メソッド呼び出しを実現する提案もある[13]．本研究では RMI よりも基本的なレイヤである Channel について実現手法について検討している．文献 11) では，OS の提供する `fork` の仕組みを用いて JSR121 と互換の API を提供している．本研究は単一プロセスのみを用いる点が異なる．

オブジェクトのメッセージパッシングにより並行，並列計算を行う，という考え方は，もちろん多くのプログラミング言語に存在する．例えば，Erlang[6] では，言語モデルとしてオブジェクトが変更不能であり，そのオブジェクトを Erlang のプロセス間で通信しながら並行・並列にプログラムを実行していく（Actor モデル）．Scala[5] も同様の Actor モデルの仕組みを備える．Go 言語[2] では，まさにオブジェクトを転送するためのチャンネルという仕組みを備えている．Go におけるチャンネルは，型とともに宣言し，その特定の型のみを通信可能にする．

Erlang は，オブジェクトの破壊的な操作を禁止している点が，破壊的な操作を多様する Ruby とは根本的に異なる．MVM では，単一 VM 中では従来通りの副作用のある操作を用いたプログラミングを行うことができる．Scala や Go は，変更可能なオブジェクトを複数の並列実行単位で共有可能であり，必要であれば排他制御を行わなければならない．それを怠り他の並列実行単位の操作が意図せず影響してしまう，ということが考えられる．とくに，Ruby では処理系全体で影響のあるメタプログラミングなどを行うことができるため，問題である．MVM では，オブジェクトは完全に独立に存在するため，VM 間での状態や，オブジェクトの排他制御を考える必要はない．

Ruby は破壊的な操作を多用してプログラムを作成することが多いため，オブジェクトを並列実行単位で共有するモデルの上で並列処理を記述すると，多くの排他制御が必要になり，プログラミングが困難となる．そのため，Scala や go のように並列実行単位間でオブジェクトを共有可能にすることはせず，完全に独立するようにした．MVM を用いることで，これまでの Ruby プログラムのように記述しながら，オブジェクトの共有による意図しない変更のために生じるバグを起こすことなく，気軽に並列プログラミングを行うことができると考えている．

Place[17] は，Scheme 処理系の Racket に導入された並列処理のための機構である．Place が本研究における VM にあたり，Place 間を Place Channel という通信機構で接続する．問題意識とアプローチは本研究とほぼ同様であるが，実装方法が異なる．GC の方式が異なるため，Channel の実装がメモリページを意識した実装となっている．TLS を用い，それが単体実行性能の低下を引き起こしている点も同じだが，彼らは JIT コンパイラで TLS に極力アクセスしないようなコードを生成することで解決を図っている．Ruby 処理系は JIT コンパイラを持っていないため，この解決方法を用いることはできない．なお，文献 17) では，C のグローバル変数の除去については言及しているが，本研究で述べたカレントワーキングディレクトリなどの取り扱いにかんする説明はない．

プロセス間通信を用いて並列プログラムを記述する手法はよく知られている．また，Ruby 向け分散オブジェクトフレームワークである dRuby や，共有メモリを介した Ruby オブジェクトの転送機構 Teleporter[23] や，Python の Multi-processing[4] など，複数プロセス間の連携をサポートする手法を提案されている．とくに，Teleporter は，Channel とほぼ同様のインターフェースを持つ．MVM では，複数のプロセスではなく，単一プロセス内で実行するため，同一アドレス空間であることを利用した高速なオブジェクトの転送や，より積極的な資源の共有による利用資源の削減を行うことができる．ただし，これらの手法は競合するものではないため，どちらの機構も提供し，状況に応じて使いわけるとよいと思われる．

9. ま と め

本稿では，プログラミング言語 Ruby に並列プログラミング言語の機能を導入するため，1 つのプロセスに複数の Ruby 仮想マシンを並列に動作させるマルチ仮想マシン（MVM）と，VM 間の Ruby オブジェクトの転送インターフェースである Channel を Ruby 1.9 に実装した．MVM と Channel を利用することで，Ruby プログラムで気軽に並列プログラミングを行うことができるようになる．本稿では，これらの利用方法から設計，実装の詳細について述べた．評価によって，とくに文字列オブジェクトの VM 間転送は，文字列の長さに関わらず一定時間で送ることができることを確認した．

本研究には，課題が多く残っている．まず，7.1 章で述べた MVM の実装の課題がある．より，効率的な実装とするため，VM 間でのメモリの共有機構などに取り組んでいきたい．転送については，Channel を用いて転送可能なオブジェクトが Marhsal プロトコルで転送できるものに制限されている問題がある．例えば入出力オブジェクトを VM 間で転送できれば，並列プログラミングの幅が広がる．メソッドやクロージャなどの転送にも対応してい

きたい．また，コピーオンライトで高速な転送を実現できるのは現状では文字列オブジェクトのみであるが，これを他の型にも適用できるようにしたい．1 つの例として，行列演算に利用する NArray[15] に適用できると，複数の VM で行列を共有しながら並列に計算するといったことができるようになる．これらの課題は，実際に MVM で並列プログラミングを行い知見を深め，設計と実装を進めていきたい．

謝　　辞

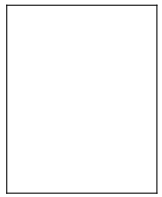
本研究は，Ruby 開発コミュニティの方々から助言を頂きました．感謝致します．

本研究の一部（高速な VM 間通信機構）は，日本学術振興会科学研究費補助金基盤研究 (S)，課題番号 21220001 の助成を得て行いました．

参 考 文 献

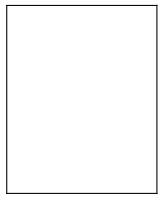
1) 4 hello, eruby. http://www.druby.org/ilikeruby/d204.html.
2) The go programming language. http://golang.org/.
3) JSR-000121 application isolation api specification 1.0 final release. http://download.oracle.com/otndocs/jcp/app_isolation-1.0-final-oth-JSpec/.
4) multiprocessing - process-based parallelism. http://docs.python.org/py3k/library/multiprocessing.html.
5) The Scala Programming Language. http://www.scala-lang.org/.
6) Joe Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pp. 6–1–6–26, New York, NY, USA, 2007. ACM.
7) Grzegorz Czajkowski. Application isolation in the java virtual machine. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pp. 354–366, New York, NY, USA, 2000. ACM.
8) Grzegorz Czajkowski and Laurent Daynés. Multitasking without comprimise: a virtual machine evolution. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pp. 125–138, New York, NY, USA, 2001. ACM.
9) Ulrich Drepper. Elf handling for thread-local storage. http://people.redhat.com/drepper/tls.pdf.
10) Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pp. 2–1–2–26, New York, NY, USA, 2007. ACM.
11) Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Cloneable jvm: a new approach to start isolated java applications faster. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pp. 1–11, New York, NY, USA, 2007. ACM.
12) CharlesOliver Nutter and ThomasE Enebo. Jruby java powered ruby implementation. http://jruby.org/.
13) Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynas. Incommunicado: efficient communication for isolates. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pp. 262–274, New York, NY, USA, 2002. ACM.
14) Y. Shinjo and C. Pu. Achieving efficiency and portability in systems software: a case study on posix-compliant multithreaded programs. *Software Engineering, IEEE Transactions on*, Vol.31, No.9, pp. 785 – 800, sept. 2005.
15) Masahiro Tanaka. Numerical ruby narray. http://narray.rubyforge.org/index.html.ja.
16) MacRuby team. Macruby. http://www.macruby.org/.
17) Kevin Tew, James Swaine, Matthew Flatt, RobertBruce Findler, and Peter Dinda. Places: adding message-passing parallelism to racket. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pp. 85–96, New York, NY, USA, 2011. ACM.
18) Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pp. 1–8, Berkeley, CA, USA, 2010. USENIX Association.
19) 関将俊. dRuby による分散オブジェクトプログラミング. アスキー, 2001.
20) 松本行弘. Ruby の真実. 情報処理, Vol.44, No.5, pp. 515–521, 2003.
21) 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.
22) 芝哲史, 笹田耕一, 卜部昌平, 松本行弘, 稲葉真理, 平木敬. 実用的な ruby 用 aot コンパイラ. 情報処理学会論文誌 (PRO), Vol.4, No.1, pp. 90–108, 3 2011.
23) 笹田耕一中川博貴. Teleporter: Ruby オブジェクトの効率的なプロセス間転送・共有機構. ソフトウェア科学会 第 28 回大会 講演論文集, 9 2011.
24) 吉原陽香, 笹田耕一, 並木美太郎. Ruby による os 構成法の提案とその実行基盤の試作. 情報処理学会研究報告. [システムソフトウェアとオペレーティング・システム], Vol. 2010, No.4, pp. 1–9, 2010-04-14.

(平成 23 年 10 月 1 日受付)
(平成 23 年 12 月 31 日採録)



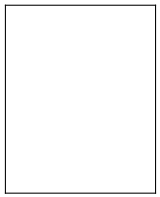
笹田　耕一（正会員）

2004 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．2006 年同大学院工学教育部博士後期課程電子情報工学専攻退学．博士（情報理工学）（東京大学情報理工学系研究科 2007 年）．2006 年東京大学情報理工学系研究科助手，2008 年同講師（現職）．システムソフトウェア，とくに並列処理システム，言語処理系に関する研究に興味を持つ．



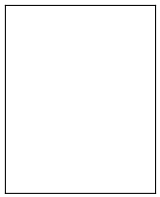
卜部　昌平

2008 年電気通信大学大学院電気通信工学研究科博士前期課程情報通信工学専攻修了．修 (工)．2005 年 (株) トランス・ニュー・テクノロジー入社．2009 年 (株) ネットワーク応用通信研究所入社 (現職)．オープンソースソフトウェアの開発，教育に従事．



松本　行弘（正会員）

1990 年筑波大学第三学群情報学類卒業．同年 (株) 日本タイムシェア入社．1994 年トヨタケーラム (株) 入社．1997 年 (株) ネットワーク応用通信研究所入社．2007 年より (株) 楽天 技術研究所 フェローおよび合同会社 Ruby アソシエーション理事長も兼務．プログラミング言語の設計と実装に興味を持つ．ACM 会員．



平木　　敬（正会員）

東京大学理学部物理学科，東京大学理学系研究科物理学専門課程博士課程退学，理学博士．工業技術院電子技術総合研究所，米国 IBM 社 T.J.Watson 研究センターをへて現在東京大学大学院情報理工学系研究科勤務．数式処理計算機 FLATS，データフロースーパーコンピュータ SIGMA-1，大規模共有メモリ計算機 JUMP-1 など多くのコンピュータシステムの研究開発に従事，現在は超高速ネットワークを用いる遠隔データ共有システム Data Reservoir システムの研究，超高速計算システム GRAPE-DR の研究を行っている．