情報処理学会論文誌　プログラミング　Vol.5　No.4　1–16 (Sep. 2012)

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

中川 博貴[1,†1,a)]　笹田 耕一[1,†2]

受付日 2012年2月13日, 採録日 2012年5月18日

概要：我々はオブジェクト指向スクリプト言語 Ruby において，マルチコアプロセッサによる並列プログラミングを効率的，かつユーザが扱いやすい形で実現することを目指し，Ruby オブジェクトのプロセス間転送・共有を行う Teleporter ライブラリを開発した．既存のプロセス間通信機構でオブジェクトを転送するには，オブジェクトをバイナリ列に変換してから送信し，受信側でそれをオブジェクトに復元する必要があり，そのオーバヘッドが問題となっていた．そこで Teleporter では通信路にプロセス間共有メモリを利用することで，オブジェクトを従来の機構よりも軽量なシリアライズ，もしくはシリアライズを行わずに転送するようにし，オーバヘッドの少ない通信を実現した．さらにプロセス間で Ruby オブジェクトを安全に共有する仕組みについて設計し，実装を行った．Teleporter はユーザが手軽に利用できることを目指すため，Ruby 処理系の改修を行わずに Ruby の拡張ライブラリとして実装した．本稿では，開発した Teleporter の設計と実装，そして API について詳細に述べる．また，Teleporter を用いた評価結果を示し，その考察を示す．

キーワード：Ruby，プロセス間オブジェクト転送，プロセス間オブジェクト共有，並列プログラミング

Design and Implementation of Efficient Interprocess Transfer and Sharing Mechanism for Ruby Objects

Hiroki Nakagawa[1,†1,a)]　Koichi Sasada[1,†2]

Received: February 13, 2012, Accepted: May 18, 2012

Abstract: We developed a Ruby library “Teleporter”, for an interprocess transfer and sharing mechanism for Ruby objects to make easy and efficient parallel programs on multi-core processors. In traditional interprocess communication mechanisms, objects must be serialized and deserialized. This overhead has become a problem. Teleporter uses an interprocess shared memory as a communication channel and it enables to transfer objects with lightweight serialization or without serialization, so communication overhead is lower than traditional mechanisms. Also, we designed and implemented a mechanism to safely share Ruby objects among processes. Teleporter is developed as an extension library for Ruby without modification of the Ruby interpreter. In this paper, we will show the design and implementation of the Teleporter and its API in detail. In addition, we will show the results of performance evaluation using Teleporter.

Keywords: Ruby, interprocess object transfer, interprocess object sharing, parallel programming

1. はじめに

消費電力やクロック速度の限界によってシングルコアでの性能向上が困難になったため，単一コアの性能を上げるのではなく，コア数を増やすことで全体のスループット向上を目指したマルチコア，メニーコアプロセッサの利用が

[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, Bunkyo, Tokyo 113–8656, Japan
[†1] 現在，グーグル株式会社
Presently with Google Inc.
[†2] 現在，Heroku, Inc.
Presently with Heroku, Inc.
[a)] sternheller@gmail.com

さかんに行われるようになった．そのため，今後は多数のコアプロセッサを効率的に並列実行するためのプログラミング環境がますます重要となる．

オブジェクト指向スクリプト言語 Ruby [1] は分かりやすい文法や実行の手軽さなどから世界中で広く利用されている．Ruby[*1]には並行並列処理を行うための仕組みはあるが，しかし不十分である．まず，言語レベルでスレッド（Ruby スレッド）をサポートしているが並列に動作しない．また，複数プロセスで並列処理を行うことが可能であるが，データをやりとりするにはオーバヘッドの大きいプロセス間通信機構を利用する必要がある．このように，現在の Ruby における並列プログラミング環境は十分に整備されているとはいえず，より効率的でかつユーザが扱いやすい形で並列プログラミングを支援する仕組みが必要である．

そこで我々は効率的なオブジェクト転送や共有をプロセスで可能にする Teleporter ライブラリを開発している．Teleporter は Ruby プロセス間でオブジェクトの転送や共有を効率的に行う仕組みである．Teleporter では通信路にプロセス間共有メモリを利用することで，オブジェクトを従来よりも軽量なシリアライズ，もしくはシリアライズを行わずに転送する．これにより時間的オーバヘッドの少ない通信を実現することができる．また，共有メモリの特徴を生かし，既存のプロセス間通信機構では実現できなかったオブジェクトのプロセス間共有をサポートし，これをユーザが安全に利用するための API を設計した．ただし，Teleporter で通信可能なのは共有メモリで通信することができる同一ノード内のプロセス間のみである．Teleporter はユーザが手軽に利用できるようにするため，Ruby 処理系の改修を行わずに Ruby の拡張ライブラリとして実装した．

本研究の貢献は主に次の 3 つである．1 つ目が，Ruby における並列処理環境の現状を分析・議論し，既存のプロセス間通信の仕組みよりも効率的なプロセス間オブジェクト転送機構を実現したことである．2 つ目が，共有メモリを利用したプロセス間通信機構を実現するうえで問題となる点を詳細にまとめたことである．そして 3 つ目が，プロセス間オブジェクト転送機構を拡張し，オブジェクトの共有を実現したことである．本機構では不変オブジェクトに限らず可変オブジェクトの共有に関しても議論し，これを実現している．またユーザが安全に共有されたオブジェクトを扱えるよう，API や排他制御モデルについて議論し，得られた知見をまとめている．

本稿の構成は次のとおりである．2 章では Ruby における並列プログラミング環境の現状について紹介し，それらをふまえて 3 章で Teleporter の提案と概要について述べる．4 章と 5 章では Teleporter が提供するプロセス間オブジェクト転送機構とオブジェクト共有機構の設計と実装について述べる．6 章では Teleporter の評価結果と考察を示し，7 章で議論を行う．そして，8 章で関連研究を紹介し，9 章でまとめを述べる．

[*1] 以後，“Ruby” や “Ruby 処理系” と表記した場合は，C 言語で実装された CRuby 処理系のことを指す．

2. Ruby における並列プログラミングの現状

本章では，現在の Ruby における並列プログラミングの環境について紹介し，既存の仕組みが持つ特徴や課題についてまとめる．本章で述べる内容は，すべて C 言語で実装された CRuby 処理系に関することである．以降，Ruby クラス（たとえば `String` クラス）のインスタンスメソッド `foo` を `String#foo`，クラスメソッド `bar` を `String.bar` と表記する．

2.1 スレッドによる並列処理

Ruby では並行処理を行うための仕組みとして Ruby スレッドを提供している．スレッドはメモリ空間を共有するため，オブジェクトを複数のスレッドで共有するような処理を自然に書くことができる．また排他制御を行う仕組みとして `Mutex` クラスを提供している．

Ruby スレッドは一部例外を除いて並列には動作せず，GVL（Giant VM Lock）と呼ばれるロックを獲得したスレッドのみが動作する．これは，スレッドセーフに実装されていない現在の Ruby 処理系や拡張ライブラリを保護するためである．文献 [2] ではスレッドを並列に動作させたが，スレッドセーフではない C 言語で実装されたメソッドを呼び出すたびに GVL の取得と解放を行う必要があり，シングルスレッドで実行した場合に特に性能低下が起きることが分かっている．CRuby 処理系以外では，Objective-C 実装である MacRuby [3] 処理系や Java 実装である JRuby [4] 処理系などが GVL の問題を解決し，スレッドを並列に動作させることができる．

スレッドによる並列処理では複数のスレッドによる競合状態が発生しないようにユーザがロック処理を行う必要がある．しかし，ロック処理を適切に行うことは一般的に困難であり，デバッグの難しいバグの原因となりやすい．我々はユーザが排他制御を行う負担を減らす必要があると考えている．

2.2 プロセスによる並列処理

Ruby ではプロセスを用いて並列処理することができる．スレッドとは異なり，プロセスは処理系の制約を受けずに並列に動作する．しかし，プロセス間でオブジェクトをやりとりするにはプロセス間通信機構を利用する必要がある．プロセス間通信機構には OS が提供するパイプやソケット，共有メモリ機構などがあるが，Ruby ではこのうちパイプやソケットを操作する仕組みを提供している．

パイプやソケットはバイトストリーム型の通信路である

ため，オブジェクトのように構造を持ったデータをそのまま送ることができない．そこで，オブジェクトをバイナリ列に変換するシリアライズを行ってから通信する．Rubyではシリアライズを行う `Marshal` モジュールが標準で付属されており，`Marshal.dump` によってオブジェクトをバイナリ列へ変換し，`Marshal.load` によってバイナリ列からオブジェクトの復元を行うことができる．

プロセス間共有メモリは各プロセスからアクセスできる共有されたメモリ領域を提供する．Linuxではプロセス間共有メモリのAPIとしてPOSIX共有メモリを提供しており，共有メモリの確保や解放，複数プロセスをまたいだ排他制御の仕組みが用意されている．しかし，パイプやソケットとは異なり，共有メモリはRubyレベルでは提供されていない．

Rubyではパイプやソケットをラップした並列処理機構がいくつか提供されている．dRuby [5] は分散オブジェクトを扱うためのライブラリで，Rubyのメソッド呼び出しの仕組みを拡張してネットワーク越しにメソッド呼び出しを行うことができる．オブジェクト共有機構Rinda [5] は協調言語Linda [6] のRuby実装で，共有空間であるTuple Spaceに対してオブジェクトを出し入れすることでオブジェクトのやりとりを行う．Tuple Spaceは概念上の共有メモリであるため，実際にはオブジェクトのコピーが行われる．parallel [7] はプロセスによる並列処理を隠蔽してマスタ・ワーカ型のタスクを簡単に書けるようにするRubyライブラリである．これらは通信路にパイプやソケットを用いるため，シリアライズのオーバヘッドが生じる．

2.3　MVMによる並列処理

スレッドやプロセスによる並列処理とは異なったアプローチで並列性能を獲得する試みも行われている．文献 [8] では1つのRubyプロセス内で複数のRubyVMを並列実行させるMVM（Multiple Virtual Machine）を導入した．MVMでは各VMがそれぞれ独自のGVLを保持するため，起動したVMの数だけRubyスレッドを並列に動作させることが可能となる．また，すべてのVMはメモリ空間を共有しているため，その特徴を生かしてVM間のRubyオブジェクト転送を軽量に行う通信路（Channel）を提供している．しかし，MVMは現在のRuby処理系に大幅な修正を加える必要があるため，利用するには処理系ごと入れ替える必要がある，また，単体性能が低下することも分かっている．

2.4　現在のRubyにおける並列プログラミングの課題

本章では，Rubyにおける既存の並列処理機構やプロセス間通信機構について紹介し，その特徴や課題について述べた．本節では現在のRubyで並列プログラミングを行ううえでの課題をまとめる．

スレッドはメモリ空間を共有できるため，スレッド間のオブジェクトの転送や共有を容易に行えるが，並列に動作するのはごく限られたときのみである．スレッドを並列動作させる研究 [2] も行われているが，実装や性能の問題から導入されていない．また，たとえスレッドが並列に動作したとしても，ユーザが適切に排他制御を行う必要があるためプログラミングが難しくなる．MVM [8] は並列に動作させることが可能であるが．処理系の改修コストや単体性能の低下の問題がある．プロセスは処理系の制約を受けずに並列に動作することができる．しかし，プロセス間でオブジェクトを共有することができず，オブジェクトをやりとりするにはプロセス間通信機構を利用する必要がある．プロセス間通信機構はオブジェクトをバイナリ列に変換・逆変換する処理を行う必要があり，そのオーバヘッドは大きい．

このように現在のRubyにおける並列プログラミング環境は十分に整備されているとはいえず，より効率的でかつユーザが扱いやすい形で並列プログラミングを支援する仕組みを実現する必要がある．

3.　Teleporterの提案と概要

前章でまとめた問題点をふまえ，我々はプロセスによる並列処理がかかえる問題を解決することで，効率的で扱いやすい並列プログラミング環境を実現するTeleporterを提案する．本章ではTeleporterの概要について述べる．

3.1　提案手法

Teleporterは複数のRubyプロセス間でオブジェクトの転送・共有を効率的に行うための機能を提供し，Rubyにおける並列プログラミングをサポートするライブラリである．また，プロセスを用いた並列処理で問題となっていたプロセス間通信のオーバヘッドを削減し，プロセス間のオブジェクト共有を可能にする．

Teleporterではプロセス間共有メモリを通信路として用いることで通常よりも軽量なシリアライズ，もしくはシリアライズを行わずにオブジェクトの転送を行えるようにした．また，共有メモリの特徴を生かし，既存の機構では実現できなかったオブジェクトの共有をサポートした．オブジェクト共有機能ではデータが変更されない不変オブジェクトに限らず，データが変更可能な可変オブジェクトの共有も可能である．Teleporterで通信可能なのは共有メモリで通信することができる同一ノード内のプロセスのみである．これらをまとめたのが**表 1** である．Teleporterはユーザが手軽に利用できるように，Ruby処理系を改造せずに拡張ライブラリとして実装した．これによりユーザはTeleporterライブラリを読み込むだけで本機構を使用することができる．

表 1　機能比較
Table 1　Feature comparison.

機能（通信路）	マルチスレッド（アドレス空間の共有）	マルチプロセス（pipe，socket）	マルチプロセス（Teleporter）
in-node	△（GVL）	○	○
out-node	×	○	×
シリアライズ	不要	必要	必要（軽量，一部不要）
転送速度	○	△	○
オブジェクト共有	○	×	△（専用クラスのみ）

```
require "Teleporter"
tunnel = Teleporter::Tunnel.new

Process.fork do
  tunnel.send "Hello"
end

puts tunnel.recv   # Hello
```

図 1　オブジェクト転送の利用例
Fig. 1　How to use object transfer.

3.2　機能と使い方

Teleporter はプロセス間でオブジェクトを転送する機能とオブジェクトを共有する機能を提供する．本節ではオブジェクト転送・共有機能の特徴と使い方について述べる．

3.2.1　オブジェクト転送機能

Teleporter はパイプに似た API を持つプロセス間オブジェクト転送の機能を提供する．我々はオブジェクト転送機能やそれを使うための API 群を Tunnel と命名した．

オブジェクト転送機能の使い方を**図 1** に示す．まず，ユーザはオブジェクト転送を行うための API となる `Tunnel` オブジェクトを生成する．次に `Process.fork` によって新しくプロセスを生成し，`Tunnel` オブジェクトを継承することで，親子関係のあるプロセス間で通信路を確立することができる．オブジェクトの送信は `Tunnel#send` で行い，送信された順番に `Tunnel#recv` でオブジェクトを受信することができる．`Tunnel` は送信されたオブジェクトをバッファリングするため，送受信側プロセスで非同期にオブジェクトの読み書きを行うことができる．また，`Tunnel` の端点はプロセスセーフに実装されているため複数のプロセスで共有して読み書きすることができ，**図 2** のように `Tunnel` をタスクキューとしたマスタ・ワーカ型の並列処理を書くことができる．

3.2.2　オブジェクト共有機能

Teleporter はプロセス間でオブジェクトを共有する機能を提供する．本機能では専用クラスのオブジェクトである共有オブジェクトのみ共有可能で，現在は `String` クラス相当の `SharedString` クラスを提供している．共有処理のためのルーチンを加えた専用クラスを記述することで共有可能なクラスを増やすことができる．オブジェクト共有の機能は，オブジェクト転送機構に共有処理を行う機構を追加することで実現されており，オブジェクト転送と同様の API で共有処理を行うことができる．共有可能なオブジェクトは不変オブジェクトに限らず，可変オブジェクトも共有することができる．オブジェクト共有機能は共有対象のオブジェクトを変更不能にするか，変更可能にするかによって呼び出すメソッドが異なる．

```
WORKER_NUM.times do
  # Worker processes.
  Process.fork do
    task = tunnel.recv
    exec task
  end
end

# Master process.
loop do
  tunnel.send(Task.generate)
end
```

図 2　マスタ・ワーカ型並列処理の記述例
Fig. 2　Master-Workers processing with Teleporter.

```
require "Teleporter"
tunnel = Teleporter::Tunnel.new

Process.fork do
  shobj = SharedString.new "Hello"
  tunnel.share_send shobj
  puts shobj     # Hello
  shobj.mutate!  # Exception occurs.
end

shobj = tunnel.share_recv
puts shobj       # Hello
shobj.mutate!    # Exception occurs.
```

図 3　不変オブジェクトの共有の利用例
Fig. 3　How to use immutable object sharing.

まず，不変オブジェクトを共有する場合の使い方を**図 3** に示す．基本的な使い方はオブジェクト転送の場合と同様で，送受信メソッドを `Tunnel#share_send` と `Tunnel#share_recv` にすることでオブジェクトの共有を行う．これにより，送受信側それぞれのオブジェクトは共

```
require "Teleporter"
tunnel = Teleporter::Tunnel.new

Process.fork do
  shobj = SharedString.new "Hello"
  tunnel.move_send shobj
  puts shobj    # Exception occurs.
end

shobj = tunnel.move_recv
puts shobj      # Hello
shobj.mutate!
```

図 4　可変オブジェクトの共有の利用例
Fig. 4　How to use mutable object sharing.

有メモリ上の同じデータを指し，プロセス間で共有された状態になる．また，共有したときに不変オブジェクトに変わり，破壊的な操作を加えようとすると例外を発生する．これにより，ユーザが共有オブジェクトを誤って変更するのを防ぐことができる．

次に，可変オブジェクトを共有する場合の使い方を図 4 に示す．送受信時のメソッドを `Tunnel#move_send` と `Tunnel#move_recv` にすることでオブジェクトの共有を行う．受信側のオブジェクトは変更可能なので自由に読み書きできる．一方，送信側のオブジェクトは無効化され，オブジェクトを読み書きしようとすると例外を発生する．このように `Tunnel#move_send` でオブジェクトを送信した後は，送信元となったオブジェクトにアクセスすることはできなくなる．これは，複数プロセスが同時に可変オブジェクトを読み書きするのを防ぐためであり，可変オブジェクトに対してアクセス可能なのはつねにたかだか 1 つのプロセスに制限される．

図 3 と図 4 では `SharedString` オブジェクトを生成し，それを共有する例を示したが，共有オブジェクトではない `String` オブジェクトを共有しようとした場合は自動的に `SharedString` オブジェクトに変換されて共有が行われる．

3.3　全体構成

Teleporter はオブジェクトの転送機能や共有機能を実現するために，図 5 に示す 3 つの機構，(1) オブジェクト転送路 Tunnel，(2) プロセス間オブジェクトスペース，(3) 共有メモリ管理機構によって構成される．以降では，各機構が担う機能について述べる．

3.3.1　オブジェクト転送路 Tunnel

オブジェクトの転送・共有のための通信路を構築し，その Ruby API を提供する役割を持つ．プロセス間通信路の実体は共有メモリ上に構築されたキューである．キューはプロセスセーフに実装されており，複数プロセスからの処理要求を適切に処理する．API によってユーザから処理要求が出されると，オブジェクトを転送するのか共有するのか，共有する場合は変更可能にするのか変更不能にするのか，指定されたセマンティクスに応じてオブジェクトに前処理を行うルーチンを呼び出し，その結果をキューに入れる．一方，受信側ではキューから取り出した結果に対し，オブジェクトの復元処理を行ってそれを返す．

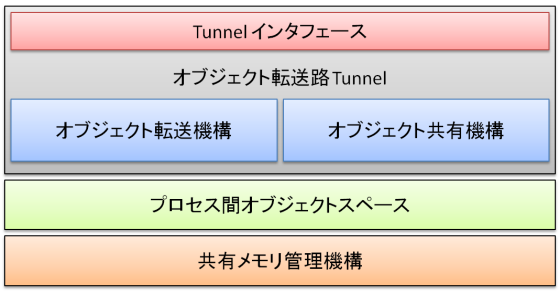

図 5　Telepoter の構成図
Fig. 5　Teleporter architecture.

3.3.2　プロセス間オブジェクトスペース

オブジェクトを転送するためには，オブジェクトをいったん共有メモリ上に配置し，送り先のプロセスからも見えるようにする必要がある．この処理を抽象的に扱えるようにしたのがプロセス間オブジェクトスペースである．また，共有メモリ上に置かれたデータからオブジェクトを構築する仕組みも提供する．

3.3.3　共有メモリ管理機構

現在の Ruby 処理系には共有メモリを扱うための仕組みが存在しないため，共有メモリの割当てや回収処理を行う共有メモリ管理機構を新たに開発した．共有メモリ管理機構は，Teleporter ライブラリ読み込み時に POSIX 共有メモリを確保し，本機構が持つメモリアロケータによってその割当てと解放処理を行う．複数のプロセスからアクセスできるように，アロケータが使用する管理データはすべて共有メモリ上に配置する．割り当てた共有メモリは Ruby 処理系が管理しているメモリと同じように読み書きすることができるが，管理する機構が異なるため両者を明確に区別して扱う必要がある．たとえば，Ruby 処理系のメモリ解放の仕組みで共有メモリを解放したり，共有メモリの仕組みで Ruby 処理系が割り当てたメモリを解放したりすることはできない．

4.　プロセス間オブジェクト転送機構の設計と実装

共有メモリを使ったオブジェクト転送機構を実現するためには，(a) 共有メモリ上に通信路を構築し，オブジェクトの送受信要求を処理する仕組みであるオブジェクト転送路 Tunnel，(b) オブジェクトを共有メモリ上に配置し，送り先のプロセスからもアクセス可能な状態にしたり，配置したデータからオブジェクトを復元したりする仕組みであるプロセス間オブジェクトスペース，(c) 共有メモリの割

当てや解放処理を行う仕組みである共有メモリ管理機構，が必要となる．

オブジェクトの転送処理の流れを図 6 に示す．まず，Tunnel オブジェクトを介してユーザから送信要求が出されると，(1) プロセス間オブジェクトスペースの機能を用いてオブジェクトを共有メモリ上に配置する．このとき，Ruby オブジェクトの実体である構造体をそのまま共有メモリ上に配置するのではなく，Teleporter 独自のフォーマットに変換して配置する．変換されたデータにはそれぞれ固有の識別子が割り振られ，データをプロセス間で一意に特定することができる．次に，(2) オブジェクト転送路 Tunnel が持つキューに先ほどの識別子を入れる．転送処理が進むと，(3) 受信側プロセスで先ほどの識別子が取り出される．(4) 取り出された識別子をもとにプロセス間オブジェクトスペース上のデータにアクセスし，オブジェクトを復元する．以上の流れにより，オブジェクトの転送処理は完了する．

4.1　プロセス間オブジェクトスペースの設計と実装

プロセス間オブジェクトスペースは共有メモリ上にオブジェクトを配置する処理を抽象化し，その C API を提供する．プロセス間オブジェクトスペース上に置かれるのは Ruby オブジェクトの構造体ではなく，独自の構造体に変換されたダンプデータである．

4.1.1　プロセス間オブジェクトスペースの機能

プロセス間オブジェクトスペースが提供する機能は，(1) オブジェクトをプロセス間オブジェクトスペースに置く処理，(2) プロセス間オブジェクトスペースに置かれたオブジェクトを読み出す処理の 2 つである．読み出す処理は，

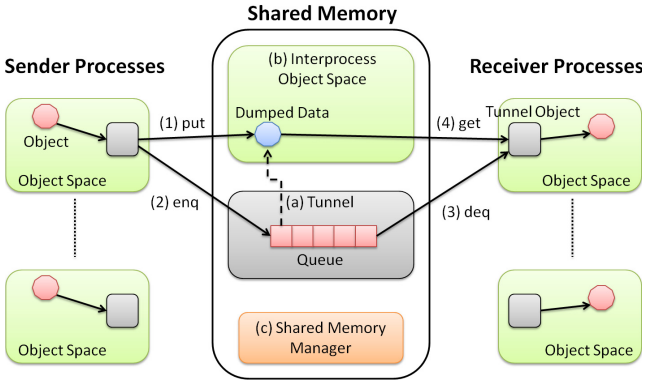

図 6　オブジェクト転送の流れ
Fig. 6　Object transfer processing.

読み出したオブジェクトを (2a) プライベートオブジェクトとするか，それとも (2b) 共有オブジェクトとするかによってさらに 2 種類の操作に分かれる．プライベートオブジェクトはプロセス固有のメモリ空間に置かれたオブジェクト，共有オブジェクトは共有メモリ空間に置かれたオブジェクトと定義する．これらの C API を表 2 に示す．

関数 put_objspace() は引数で与えられたオブジェクトをダンプデータに変換し，それを共有メモリ上に置く．また，返り値としてダンプデータを一意に特定する識別子を返す．識別子は sid_t 型で表される．これは現在の実装ではダンプデータの先頭アドレスを用いている．ダンプデータの形式はオブジェクトのクラスによって異なるため，引数で渡されたオブジェクトのクラスに応じたダンプ形式への変換関数を呼び出す．関数 get_objspace() は識別子に対応するダンプデータからオブジェクトの復元を行う．復元方法もクラスによって異なるため，ダンプデータが示すクラスに応じた復元関数を呼び出す．一方，関数 refer_objspace() はオブジェクトの共有処理で用いられ，ダンプデータから共有オブジェクトとして復元を行う．オブジェクトの共有については次章で述べる．

4.1.2　ダンプデータ

プロセス間オブジェクトスペースには Ruby オブジェクトの構造体を直接置くのではなく，Teleporter 独自の構造体に変換したデータを置く．これは，Ruby オブジェクトの構造体を直接置くだけでは受信側で復元できないクラス（Symbol など）への対応と，参照カウントによるデータの寿命管理を行うためである．ダンプ形式は各クラスごとに個別に定義されているが，どのダンプ形式も第 1 メンバとして，ダンプの種類（クラス）やダンプデータの共有状態を表すフラグ，そしてダンプデータの参照カウントを記録する構造体を持つ．

以降では，Teleporter が対応するクラスのダンプ形式を示し，オブジェクトをダンプ形式へと変換する手順やその逆変換の手順について述べる．

埋め込みオブジェクト　Ruby では，固定長整数クラス Fixnum や真偽値を表す true，false，そして空を意味する nil など，頻繁に扱うクラスをポインタに埋め込んだ即値として扱うことで性能向上を図っている．これらは構造を持たないため，ダンプせずに値をそのまま共有メモリ上に配置する．

Float オブジェクト　浮動小数点クラス Float のオブジェ

表 2　プロセス間オブジェクトスペースの C API
Table 2　C API of the interprocess object space.

操作名	内容
sid_t put_objspace(VALUE obj)	オブジェクトのダンプデータを置き，その識別子を返す．
VALUE get_objspace(sid_t sid)	ダンプデータからプライベートオブジェクトを生成する．
VALUE refer_objspace(sid_t sid)	ダンプデータから共有オブジェクトを生成する．

表 3　ダンプデータ処理のインタフェース（String クラスの場合）
Table 3　Interfaces of dump data processing (String class).

操作名	内容
`sid_t w_dumped_string(VALUE obj)`	オブジェクトをダンプデータとして書き出す．
`VALUE r_dumped_string(dump *d)`	ダンプデータからプライベートオブジェクトを生成する．
`VALUE c_dumped_string(dump *d)`	ダンプデータから共有オブジェクトを生成する．
`void d_dumped_string(dump *d)`	ダンプデータの解放処理を行う．

表 4　Tunnel API
Table 4　Tunnel API.

操作名	内容
`Tunnel#send(obj)`，`Tunnel#recv`	オブジェクトを copy 方式で送受信する．
`Tunnel#share_send(obj)`，`Tunnel#share_recv`	オブジェクトを share 方式で送受信する．
`Tunnel#move_send(obj)`，`Tunnel#move_recv`	オブジェクトを move 方式で送受信する．

クトの実体は `double` 型の値であり，ダンプデータではこれを保持する．一方，ダンプデータからオブジェクトを復元する場合は，`double` 型の値から `Float` オブジェクトを生成する関数を Ruby 処理系が提供しているため，これを用いる．

`Complex`, `Rational`, `Bignum` オブジェクト　複素数クラス `Complex` のオブジェクトは実部と虚部を表すオブジェクトへのポインタを持つため，ダンプデータへと変換する場合はまず実部と虚部に入るオブジェクトをダンプし，得られた識別子を保持する．一方，ダンプデータからオブジェクトを復元する場合は，実部と虚部のオブジェクトを復元し，`Complex` オブジェクトの生成を行う．なお，有理数クラス `Rational` や `Bignum` もほぼ同様の方法でダンプを行う．

`String` オブジェクト　文字列クラス `String` のダンプデータは，共有メモリ上に配置された文字列へのポインタと文字列の文字コード名を保持する．Ruby では処理系内部で統一して扱われる文字コードは存在せず，各文字列オブジェクトがそれぞれ文字コード情報を保持する仕組みになっている [9]．そのため，ダンプデータでも元の `String` オブジェクトの文字コードを保持する必要がある．なお，Ruby 処理系内では文字コードは C 言語の配列で管理されており，その添字によって文字コードを指定することができる．しかし，この配列は処理系のバージョンによって異なったり，ユーザが任意の文字コードを追加できたりするため，送受信元で添字と文字コードの組が一致しない可能性がある．そこで，文字コードの名前を送ることで同じ文字コードで `String` オブジェクトが復元されるようにしている．

`Symbol` オブジェクト　`Symbol` オブジェクトの実体は名前を示す整数値である．この整数値は `Symbol` オブジェクトを生成したときに適当な値が割り振られる．同一の Ruby プロセス内であれば，同じシンボル文字列には必ず同じ整数値が割り振られることになっているが，プロセス間で整数値をやりとりしたとしても，その値が同じ名前を指すとは限らない．そこで，ダンプデータでは整数値ではなくシンボル文字列を保持する．オブジェクトを復元するときはシンボル文字列から `Symbol` オブジェクトを生成する．

`Array` オブジェクト　`Array` オブジェクトが持つ各要素は Ruby オブジェクトである．つまり，`Array` オブジェクトをダンプするときは，各要素も再帰的にダンプ形式に変換する必要がある．`Array` オブジェクトのダンプデータでは，各要素をダンプ形式に変換した際に得られた識別子を配列で管理する．

4.1.3　ダンプデータのインタフェース

ダンプ形式はクラスごとに違うため，与えられたオブジェクトのクラスに応じてダンプ形式への変換方法を使い分けなくてはならない．また，オブジェクトの内部にオブジェクトを保持している場合のように，変換処理の間で別のクラスの変換処理を呼び出さなくてはならないときもある．そこで Teleporter ではダンプ形式への変換やダンプ形式からの復元，そしてダンプ形式を共有メモリ上から削除するためのインタフェースを定義した（**表 3**）．新しく転送に対応したクラスを増やしたい場合は，このインタフェースに従った処理を記述し，共有オブジェクトスペースの C API 内で呼び出すようにすれば対応できる．

4.2　オブジェクト転送路 Tunnel の設計と実装

Tunnel は転送されたオブジェクトを格納するプロセスセーフなキューを持ち，転送処理のための Ruby API を提供する．Tunnel が提供する Ruby API を**表 4** に示す．使用する API に応じて Tunnel はオブジェクトの転送方法を切り替える．本章ではオブジェクトの転送を行う `Tunnel#send`，`Tunnel#recv` について述べる．

Tunnel のオブジェクト転送処理は次のようになる．オブジェクトの転送要求が出されると，Tunnel は

put_objspace() を呼び出してオブジェクトをプロセス間オブジェクトスペース上に配置し，得られた識別子を Tunnel が持つキューに対して入れる．一方，オブジェクトの受信要求が出されるとキューから識別子を取り出し，それをもとにプロセス間オブジェクトスペース上からオブジェクトを生成する．これは，プロセス間オブジェクトスペースが提供する get_objspace() を呼び出すことで行う．

Tunnel の持つキューは複数のプロセスが同時に読み書きする可能性があるので，投入・取り出し時にプロセスをまたいだロックをかけることでプロセスセーフとしている．キューが満杯のときに識別子を投入しようとしたり，空のときに取り出そうとしたりすると，そのプロセスはブロックされる．また，Tunnel の端点に対して複数のプロセスが読み書きを行うことができ，図 2 のように Tunnel をタスクキューとしたマスタ・ワーカ型の処理を簡潔に書くことができる．

4.3　共有メモリ管理機構の設計と実装

共有メモリ管理機構は，Teleporter ライブラリ読み込み時に共有メモリを確保し，本機構が持つアロケータによってその割当てと解放処理を行う．共有メモリアロケータを複数のプロセスから利用できるように管理データはすべて共有メモリ上に配置する．また，プロセス間で共有するロックを用いてプロセスセーフに参照できるようにしている．

Teleporter を使って通信するプロセスは，すべて同じ仮想アドレスに共有メモリをマップする必要がある．これは共有オブジェクトの識別子としてダンプデータへのアドレスを用いているため，送信側と受信側で同じアドレスに同じダンプデータが置かれている必要があるからである．Teleporter ではライブラリを読み込むときに共有メモリをマップし，その後ユーザが Process.fork を実行することでこれを実現している．fork システムコールは実行時にメモリマッピングを親プロセスから子プロセスへと継承するため，必ず同じ仮想アドレスに共有メモリをマップすることができる[*2]．しかし，この方法では通信できるのが親子関係のあるプロセスに限られ，かつ子プロセスを生成する前に共有メモリをマップしておく必要がある．これは現在の実装での制限である．

5.　プロセス間オブジェクト共有機構の設計と実装

本章では，前章で説明したオブジェクト転送機構を拡張してプロセス間でオブジェクト共有を行う方法について述べる．オブジェクト共有機能を実現するためには，まず，(1) オブジェクト共有モデル（排他制御，共有処理），(2) オブジェクト共有のための Ruby 用 API の設計，(3) オブジェクト共有機能の設計と実装，について検討する必要がある．次にこれらをふまえて共有機構の設計と実装について述べる．

5.1　オブジェクト共有モデルに関する議論

可変オブジェクトの同時アクセスを許している場合，排他制御を適切に行わないと競合状態が生じてデバッグ困難なバグの原因となる．そこでまず共有オブジェクトの排他制御について議論し，Teleporter で採用したモデルについて述べる．次に，排他制御モデルに適したオブジェクトの共有処理について検討する．

5.1.1　共有オブジェクトの排他制御

ロックを用いた排他制御ではユーザが柔軟に排他制御することができる反面，競合状態を防ぐためにユーザ自身が間違いなくロックをかける必要がある．

我々はユーザがより手軽に共有オブジェクトを扱えるように，所有権と不変オブジェクトを合わせたモデル [10] を導入した．所有権モデルとは，任意の時点で共有オブジェクトの所有者をたかだか 1 人に制限し，その所有権を他者へ譲渡していくことでオブジェクトのやりとりを行うモデルである．このモデルでは，共有オブジェクトへの処理を終えた人は所有権を手放す．所有権を手放した人は以後その共有オブジェクトを読み書きすることはできない．読み込みも禁止する理由は書き込み時の不定状態を読み込ませないようにするためである．所有権を受け取った人は共有オブジェクトを自由に読み書きすることができるようになる．一方，不変オブジェクトは排他制御を行う必要がないため，複数人が同時に読むことができる．

Teleporter では所有権の譲渡を move 操作，オブジェクトを不変にして共有可能にする操作を share 操作と呼び，所有権を持つのはプロセスとなる．move 操作は Tunnel を介して所有権を別のプロセスへ送り，送り元のオブジェクトを無効化する．share 操作は不変状態にしたオブジェクトを Tunnel を介して共有する．

本モデルの利点は，ユーザが共有オブジェクトに対してロックをかけなくてよい点である．これによりロックのかけ忘れや操作ミスによる競合を防ぐことができる．一方，本モデルの欠点は，所有権を譲渡してもらわないと書き込み処理を行えないため，ある共有オブジェクトを複数のプロセスが能動的に書き換えるような処理を書きにくいことである．しかし，我々はユーザが安全に共有オブジェクトを扱えることを重視し，本モデルを採用することにした．

5.1.2　Teleporter の共有モデルによる並列プログラム

本項では，Teleporter の共有モデルによる並列プログラムの記述性について検討する．Teleporter の通信路である Tunnel は複数のプロセスが同時に読み書き可能で，非同期

[*2] mmap システムコールは共有メモリをマップする場所を自由に指定することができるが，確実に成功する保証はない．

のタスクキューとして利用することができるため，共有オブジェクトに対するマスタワーカ型の処理を簡潔に記述できる．また，今後 SharedArray を導入することで大きな配列オブジェクトを共有し，部分ごとに各プロセスで処理を行うデータ並列プログラムも実現できる．

従来のロックを用いた処理ではプロセスが目的の共有オブジェクトのロックを能動的に取得してアクセスしていたのに対し，Teleporter では渡されたオブジェクトに対して読み書きを行うことができる受動的なモデルとなる．そのため，次々と送られてくるオブジェクトを処理し，それを次のプロセスに渡していくパイプライン・グラフ型の処理などに適している．また，ノード間でコピーレスに可変データのやりとりを行うことができるため，通信コストの削減が見込める．たとえば，MapReduce フレームワークに Teleporter を利用することが可能である．

一方，Teleporter では受信するオブジェクトを受信側で選ぶことができないため，特定の共有オブジェクトを選択して処理するようなプログラムを書く場合には工夫が必要となる．また，共有オブジェクトに書き込めるのは move で受け取ったただ 1 つのプロセスに制限されるため，1 つの共有オブジェクトに対して複数のプロセスが書き込みを行うような場合も工夫が必要となる．このようなアプリケーションを実現するには特定の共有オブジェクトを管理するマネージャプロセスを用意し，マネージャに対してオブジェクトの転送要求や書き込み要求を行うように実装する方法があげられる．これによりタプルスペースのような並列プログラムを記述することができる．ロックを取得してから自分で共有データへの書き込みを行う場合に対し，マネージャを介した処理はメッセージ転送のコストがかかるが，誤ったロック操作によって生じる共有オブジェクトの一貫性の問題を回避することができる．

5.1.3　オブジェクトの共有処理

オブジェクトの共有処理を検討するためには，共有対象のオブジェクトがプライベートオブジェクトなのか共有オブジェクトなのか，送受信側でどのような共有セマンティクスを指定したかが重要となる．以降では，共有対象のオブジェクトの種類や指定された共有セマンティクスごとに共有処理がどのように行われるのか述べる．なお，今後は share 操作によって共有されたオブジェクトを不変共有オブジェクト，move 操作によって共有されたオブジェクトを可変共有オブジェクトと呼ぶ．

プライベートオブジェクトの場合　プライベートオブジェクトを共有することはできないため，新たに共有オブジェクトを生成し，それを共有する．このため，share と move どちらの方式を選択しても必ずプロセス間オブジェクトスペース上へのデータコピーが発生する．share や move で送信したオブジェクトを share で受信すると不変共有オブジェクトとして受信することができる．一方，move で送信したオブジェクトを move で受信すると可変共有オブジェクトとして受信することができるが，share で送信したオブジェクトを move で受信しようとした場合は例外を発生させる．

不変共有オブジェクトの場合　不変共有オブジェクトはプロセス間オブジェクトスペース上にすでに実体が存在するため，share と move どちらの方式を選択したとしてもデータコピーは発生しない．share で送信したオブジェクトは share でのみ受信することができる．move で受信しようとした場合は例外を発生させ，変更可能な状態で読み出されるのを防ぐ．同様に move で送信しようとした場合も例外を発生させる．

可変共有オブジェクトの場合　可変共有オブジェクトはプロセス間オブジェクトスペース上にすでに実体が存在するため，share と move どちらの方式を選択したとしてもデータコピーは発生しない．share で送信したオブジェクトは share でのみ受信することができる．move で受信しようとした場合は例外を発生させ，変更可能な状態で読み出されるのを防ぐ．move で送信した場合は share と move どちらでも受信することができる．また，送信時に送信元オブジェクトの所有権を無効化し，move 後は送信元の共有オブジェクトからデータにアクセスできなくする．

なお，共有オブジェクトの実体であるダンプデータは参照カウント方式によって管理される．参照カウントは新たに共有オブジェクトを生成した場合や共有オブジェクトが Ruby の GC によって回収されるときに増減し，参照数が 0 になるとダンプデータが削除される．また，共有オブジェクトは共有メモリ管理機構によって管理されたメモリ領域を使用しているため，そのままプライベートオブジェクトに戻すことはできない．ただし，共有オブジェクトが持つデータをもとに，新たにプライベートオブジェクトを生成することは可能である．

5.2　オブジェクト共有のための Ruby API 設計

本節ではオブジェクト共有機構を利用するための Ruby API について述べる．

5.2.1　API に関する検討

所有権モデルでは，オブジェクトの送受信時に対象オブジェクトの所有権をどう扱うかユーザが指定できる必要がある．そこで，不変共有オブジェクトをやりとりする `Tunnel#share_send`，`Tunnel#share_recv` と，可変共有オブジェクトをやりとりする `Tunnel#move_send`，`Tunnel#move_recv` を新たに設けた（表 4 参照）．これらの API を利用したオブジェクト共有の例を図 3 と図 4 に示す．なお，`Tunnel#share_recv` や `Tunnel#move_recv` で受け取る共有オブジェクトは共有オブジェクト専用クラスと呼ばれる特殊なクラスとなる．

5.2.2　共有オブジェクト専用クラス

プロセス間でオブジェクトを共有するためには2つの問題を解決する必要がある．我々は共有オブジェクトとして扱えるのを共有オブジェクト専用クラスのオブジェクトのみに制限することでこの問題を解決した．

1つ目が，排他制御に関する問題である．Teleporterでは排他制御を行うために，所有権と共有状態を示すフラグとそれらをチェックするルーチンをメソッドに加える必要がある．たとえば，共有オブジェクトが自身のデータにアクセスするときはinvalidフラグが立っていないか確認する必要がある．invalidフラグとは所有権を失った共有オブジェクトに立てられるフラグで，共有オブジェクトのヘッダ部分に保持する．invalidフラグが立っているときは共有データへアクセスすることができないので例外を発生させる．また，自身のデータを破壊するようなアクセスを行うときはshareフラグが立っていないか確認する．shareフラグとは不変共有オブジェクトのときに立てられるフラグで，ダンプデータのヘッダ部分に保持する．shareフラグが立っている場合は不変共有オブジェクトなので，破壊的な操作を行わずに例外を発生させる．これら処理を共有オブジェクト専用クラス内で行うことで，適切に排他制御を行うことが可能になる．

2つ目が，メモリ管理機構の違いに関する問題である．従来のRubyオブジェクトはRuby独自のメモリ管理機構によって管理されているが，共有メモリ上のオブジェクトはTeleporter独自の共有メモリ管理機構で管理されており，これらを区別することができない．そこで共有オブジェクト専用クラスを導入することで共有メモリの割当てや解放をすべてTeleporterが提供する共有メモリアロケータで行うようにすることができる．

このほかに，共有オブジェクト専用クラスのオブジェクトは共有オブジェクト専用クラスのオブジェクトしか持てないように制限する．これは共有オブジェクトがプライベートオブジェクトへの参照を持たないようにするためである．また，ツリーのような複雑な構造を持ったオブジェクト群の一部を取り出されたとしても，共有メモリのルールに従って適切にオブジェクトの運用が行われることを保証することができる．

以上のように，共有オブジェクト専用クラスのオブジェクトはオブジェクト共有に関わる問題を解決することができる．しかし，既存のクラスとは別に共有メモリ専用のクラスを実装する必要があり，実装コストがかかってしまう．ただし，大半のクラスは既存クラスのメモリ割当てと解放処理を共有メモリ管理機構で行うように修正し，データアクセスの前にフラグチェックのルーチンを入れるだけであり，クラスが持つロジックの部分はほぼそのまま流用することができる．

5.2.3　コンテナ型オブジェクトの共有に関する検討

現在コンテナ型のオブジェクトを共有することはできないが，そのようなオブジェクトを共有する方法について検討し，今後どのように導入していくか述べる．ここでは`Hash`の共有オブジェクト版である`SharedHash`について検討する．

`SharedHash`は`Hash`と同様の内部構造によってキーと値を保持する．`Hash`と異なるのは，保持する値へのアクセス方法が`SharedHash`オブジェクトの可変状態によって制限される点である．コンテナオブジェクトの共有では，コンテナオブジェクト自体の可変状態と，それが値として持つオブジェクトの可変状態をどのように扱うかが重要となる．我々は個々の値が様々な状態になりうるのはユーザがプログラムを記述するのを難しくすると考え，コンテナオブジェクトを受信した時点ではすべての値がコンテナオブジェクトの可変状態に一致するように設計を行っている．つまり，`Tunnel#share_recv`された`SharedHash`オブジェクトが参照するオブジェクトはすべて不変で，`Tunnel#move_recv`された`SharedHash`オブジェクトが参照するオブジェクトはすべて可変となるようにする．

不変状態にある`SharedHash`オブジェクトからは自由に値を読むことができるが，値を変更したり追加削除したりすることはできないようにする．また，保持する値はすべて不変共有オブジェクトに制限することで，値の一部が書き換えられることによって生じる一貫性の問題を防ぐ．この制限はshareを行った際に値をすべて不変共有オブジェクトに変更することで満たすことができる．

一方，可変状態にある`SharedHash`オブジェクトは値の読み書きや追加削除を自由に行うことができ，値として持てるオブジェクトに制限はない．可変`SharedHash`オブジェクトをshareする場合は，値をすべて不変共有オブジェクトに変更してから共有を行う．moveする場合は，値をすべて可変共有オブジェクトに変更してから共有を行うが，不変共有オブジェクトが含まれる場合はその値のみ新たに可変共有オブジェクトを生成して共有する．可変`SharedHash`オブジェクトは不変共有オブジェクトを持てないようにしたり，不変共有オブジェクトを持とうとしたときに可変共有オブジェクトに変換したりすることなども検討したが，ユーザの利便性や変換処理の発生タイミングの分かりやすさなどから，このような設計を行った．

以上の操作により，受信側では`SharedHash`オブジェクトとそれが持つ値オブジェクトの変更可能性が一致する．なお，現在共有メモリ専用クラスを持たないオブジェクトについては共有操作時にコピーすることを検討している．またコンテナ型のオブジェクトは循環参照の問題が生じるため，参照カウンタによるダンプデータの回収が適切に実行されない可能性があり，これを解決する必要がある．

5.3　オブジェクト共有機構の設計と実装

本節ではオブジェクト転送機構上にオブジェクト共有機能を組み込むための設計と実装について述べる．なお，オブジェクト転送機構に組み込むうえで拡張を加えなくてはならないのは，プロセス間オブジェクトスペースとオブジェクト転送路 Tunnel の 2 つである．

5.3.1　プロセス間オブジェクトスペースの拡張

オブジェクトを共有するためには，(1) 共有メモリ上にオブジェクトが持つデータを配置し，(2) それを参照する形で共有オブジェクトを生成する処理が必要となる．(1) はプロセス間オブジェクトスペースが提供する `put_objspace()` によってすでに提供されているが，(2) はまだ実現されていない．そこで，ダンプデータを参照するオブジェクトとして読み出す `refer_objspace()` を新たに実装した（表 2 参照）．`refer_objspace()` は引数で与えられた識別子に対応するダンプデータからオブジェクトの復元を行う．オブジェクトは共有オブジェクトとして，つまりダンプデータを参照する共有オブジェクト専用クラスのオブジェクトとして復元される．

5.3.2　オブジェクト転送路 Tunnel の拡張

オブジェクトの共有処理は，共有オブジェクトを送受信することで実現するように設計されているため，多くの処理はオブジェクト転送機構のものをそのまま利用することができる．オブジェクトの転送処理で Tunnel が行っていたのは，オブジェクトをプロセス間オブジェクトスペースに配置して識別子をキューに入れる処理と，キューから取り出された識別子をもとにプロセス間オブジェクトスペースからプライベートオブジェクトを復元する処理の 2 つであった．これに対し，オブジェクト共有の場合は識別子をキューに入れる前に共有オブジェクトの share フラグと invalid フラグのチェック処理を行う点と，受信側でオブジェクトを復元するときに共有オブジェクトとして復元し，share フラグをセットする必要がある点が異なる．

まず share 操作について説明する．share 操作の流れを**図 7** に示す．(1) 共有対象のオブジェクトが共有オブジェクトの場合は invalid フラグを確認し，フラグが立っている場合は例外を発生させて転送処理を終了する．(2) フラグが立っていない場合は `put_objspace()` によってオブジェクトをプロセス間オブジェクトスペースに配置し，ダンプデータの識別子を得る．そして，(3) ダンプデータに share フラグを立てて不変オブジェクトとし，(4) 識別子をキューに入れる．受信側では (5) 識別子をキューから取り出し，識別子に対応するダンプデータに share フラグを立てる．次に (6) `refer_objspace()` によって，ダンプデータが表すクラスに対応した共有オブジェクト専用クラスの共有オブジェクトを生成する．この共有オブジェクトは share フラグが立っているので不変オブジェクトになる．

次に move 操作について説明する．共有対象のオブジェクトが共有オブジェクトの場合は invalid フラグを確認し，フラグが立っている場合は例外を発生させて転送処理を終了する．フラグが立っていない場合は `put_objspace()` によってオブジェクトをプロセス間オブジェクトスペースに配置し，ダンプデータの識別子を得る．次にダンプデータの share フラグが立っていないか確認する．share フラグが立っている場合は不変共有オブジェクトなので，例外を発生させて転送処理を終了する．フラグが立っていない場合は送信元共有オブジェクトに invalid フラグを立て，識別子をキューに入れる．受信側では識別子をキューから取り出し，識別子に対応するダンプデータに share フラグが立っていないか確認する．share フラグが立っている場合は，不変共有オブジェクトの所有権を持とうとしているので例外を投げる．フラグが立っていない場合は，`refer_objspace()` によって，ダンプデータの種類に応じた共有オブジェクト専用クラスの可変共有オブジェクトを生成する．

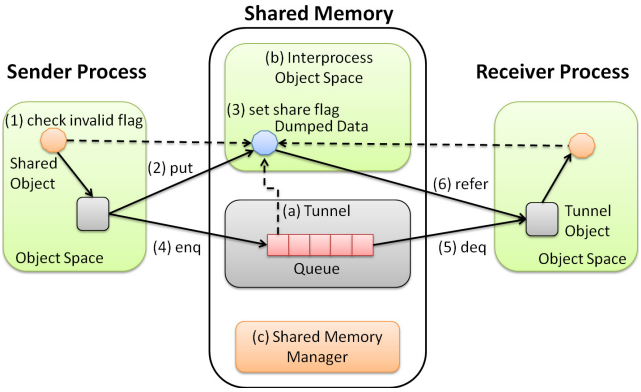

図 7　オブジェクト共有の流れ（不変共有オブジェクトの場合）
Fig. 7　Object sharing processing (immutable shared object).

6.　評価と考察

本章では，開発した Teleporter ライブラリの評価結果を示し，その考察を述べる．実験には**表 5** に示すマシンを使用し，オブジェクトの転送処理に関する性能評価とオブジェクトの共有処理に関する性能評価を行った．

6.1　オブジェクト転送の性能評価

本節では，オブジェクト転送機構の性能を評価する．実験では，受信したオブジェクトをすぐに送り返すピンポンと呼ばれる処理を行い，その処理時間を測定した．2 つのプロセスの間で 2 本の転送路を用意し，それぞれを送信用と返信用としてピンポン処理を行う．また，比較対象としてパイプ，ソケット，そして MVM [8] でも同様の評価を行う．パイプやソケットでは送受信の前後で `Marshal` によるシリアライズを行う．実験に使用したのは表 5 の Machine 1 である．

まず，**表 6** に示すオブジェクトの転送時間を測定した．

表 5　評価環境
Table 5　Evaluation environments.

	Machine 1	Machine 2
OS	GNU/Linux 2.6.35 - 64 bit	GNU/Linux 2.6.26 - 64 bit
CPU	Intel Core 2 Quad 2.66 GHz (2core×2)	Intel Xeon E7450 2.4 GHz (4core×6)
Memory	4 GB	16 GB
Compiler	GCC 4.5.1	GCC 4.3.2

表 6　オブジェクトの転送内容
Table 6　Object conditions.

Name	Contents
`Fixnum`	`1000`
`Float`	`0.1`
`Bignum`	`10**100`
`Complex` (`Fixnum`)	`Complex(3, 2)`
`Complex` (`Float`)	`Complex(0.1, 0.1)`
`Complex` (`Bignum`)	`Complex(10**100, 10**100)`
`Rational` (`Fixnum`)	`Rational(3, 2)`
`Rational` (`Bignum`)	`Rational(10**100+1, 10**100)`
`String` (100)	`'a'*100`

各オブジェクトを 1 万回ピンポンする処理を 5 回ずつ測定し，その中で最も実行時間が短かったものを評価に用いた．測定結果を**図 8** に示す．結果より，シリアライズのオーバヘッドを削減することでパイプやソケットよりも高速にオブジェクト転送を行えることが確認できた．MVM と比較すると，ポインタ埋め込みオブジェクトや `Float` といった MVM が高速化の工夫をしている転送がほぼ同じ転送時間で，特に `String` は MVM の方が高速に転送していることが分かる．

次に `Array` オブジェクトの転送時間を測定した．`Array` では `Fixnum`，`Float`，`Bignum` オブジェクトを要素として持つ配列を生成し，ピンポン処理にかかる時間を測定した．この結果を**図 9**，**図 10**，**図 11** に示す．どの場合もパイプやソケットに比べて高速であることが確認できた．MVM との比較では，`Array` (`Float`) の転送で MVM の方が高速に転送を行えた．一方，MVM は `Bignum` オブジェクトを `Marshal` で転送するため，`Array` (`Bignum`) の転送では Teleporter の方が高速に転送できた．

`String` オブジェクトでは文字列長を変化させた場合の転送時間を測定した．結果を**図 12** に示す．Teleporter よりも MVM の方が高速であることが分かる．MVM は `String` オブジェクトの転送に特別な最適化（コピーオンライト）処理を加えており，受信側でオブジェクトに対し破壊的な操作を加えるまで受信元のオブジェクトを参照し，コピーを遅延させる．これにより文字列長にかかわらず一定の時間で転送処理を行うことができるため MVM (Channel, CoW) は非常に高速である．一方，受信側で破壊的処理を加えて，強制的に文字列をコピーさせた場合が MVM (Channel, CoW) である．MVM はローカルのメモリ空間を共有しているため，転送時のメモリコピーの回数が Teleporter の場合と比べて少なく済み，より高速に転送できる．ただし，実験に使用した実装では 350,000（Byte）以上の文字列を持つオブジェクトを送ろうとすると急激に性能が悪化し，適切に測定することができなかった．

6.2　アプリケーションを用いた並列処理の性能評価

本節では，並列処理を行うアプリケーションのベンチマークの結果を示し，Teleporter の並列処理性能について評価する．実験に用いたのは住所とテンプレートから HTML テキストを生成するアプリケーションで，文献 [8] で使われているものを Teleporter 用に書きなおしたものである．本アプリケーションでは．マスタプロセスが住所を表す文字列を送り，ワーカプロセスがそれを受け取って HTML を表す文字列を生成，返送用の通信路を使って結果を送り返す．住所は日本郵政が提供する郵便番号一覧を行ごとに分割した文字列である．返送されるテキストのサイズによる性能変動を見るために，生成した HTML テキストを 100 個連結した場合も別途測定する．実験に使用したのは表 5 の Machine 2 である．同様の評価をパイプと MVM でも行った．なお，パイプは端点を複数のプロセスで共有することができないため，ワーカプロセスの数だけパイプオブジェクトを用意し，それらに対して順番にタスク割当てを行うようになっている．

測定結果を**図 13** に示す．各系列名の末尾に付く (1) が HTML テキストをそのまま返す場合，(100) が 100 個連結した HTML テキストを返す場合である．横軸がワーカプロセスの数を表しており，0 の場合はマスタプロセスで HTML テキストを生成した場合を意味する．結果より，(1) と (100) の場合ともにワーカプロセス数が増えるにつれてスケールしていることが分かる．しかし，(100) の場合は送信するテキストのサイズが大きいため，どの機構でも途中で性能が頭打ちになってしまう．パイプや MVM はその後横ばいであるが，Tunnel では 17 プロセスあたりから性能が低下し始める．これは複数のワーカプロセスが共有メモリアロケータを奪い合うことで生じている．

6.3　オブジェクト共有の性能評価

本節では Ruby オブジェクトの共有にかかる時間を測定

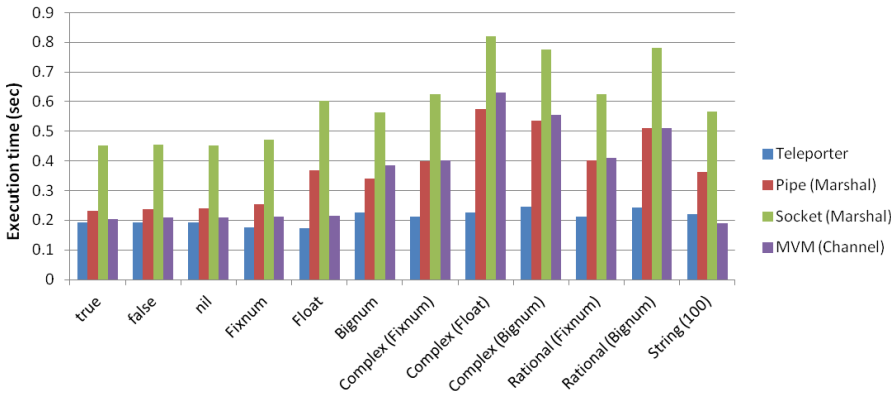

図 8　オブジェクトの転送時間
Fig. 8　Results of transferring objects.

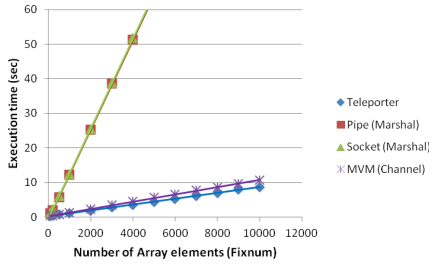

図 9　`Array` オブジェクト（`Fixnum`）の転送時間
Fig. 9　Results of transferring `Array` objects (`Fixnum`).

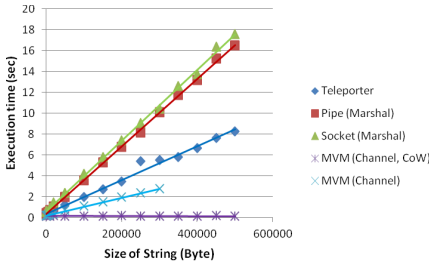

図 12　`String` オブジェクトの転送時間
Fig. 12　Results of transferring `String` objects.

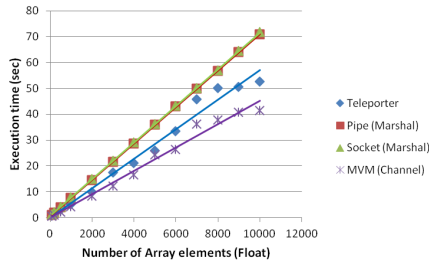

図 10　`Array` オブジェクト（`Float`）の転送時間
Fig. 10　Results of transferring `Array` objects (`Float`).

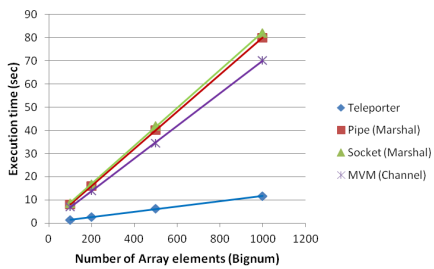

図 11　`Array` オブジェクト（`Bignum`）の転送時間
Fig. 11　Results of transferring `Array` objects (`Bignum`).

することでオブジェクト共有機構の性能を評価する．Teleporter (copy，share，move) と MVM で `String` オブジェクトのピンポン処理を 1 万回行い，その処理時間を測定した．実験に使用したのは表 5 の Machine 1 である．測定結果を**図 14** に示す．Teleporter (copy) や MVM (Channel) では文字列が長くなるにつれて線形に処理時間が増えているのに対し，Teleporter (share) や Teleporter (move) では文字列が長くなっても処理時間はほぼ一定であることが確認できた．前者ではピンポン処理を行うたびに文字列のコピーが生じるが，後者で文字列コピーが発生するのはプライベートオブジェクトをプロセス間オブジェクトスペースに配置する最初の 1 回のみで，後は参照の受け渡しだけで転送が完了するからである．MVM (Channel, CoW) は文字列のコピーが発生しないため，文字列長に依存せずに高速に共有することができる．

7.　議論

本章では，評価結果をふまえた議論を行い，今後の課題についてまとめる．

評価結果より，パイプやソケットよりも高速にオブジェクトの転送が行えることが確認できた．一方，MVM との比較では `String` の転送が MVM よりも処理時間が長いという結果になった．Teleporter ではプライベートオブジェクトを必ず 1 度共有メモリ空間にコピーする必要があるのに対し，MVM はローカルのメモリ空間を共有するため，共有メモリ空間へのコピーを行う必要がないからである．Teleporter でもオブジェクトを初めから共有メモリ上に割り当てることで，このコピーコストを削減する必要がある．

6.2 節の並列処理ベンチマークの結果より，プロセス間共有メモリを通信路として利用する場合は共有メモリアロケータの並列性能が重要であることが分かった．今後さら

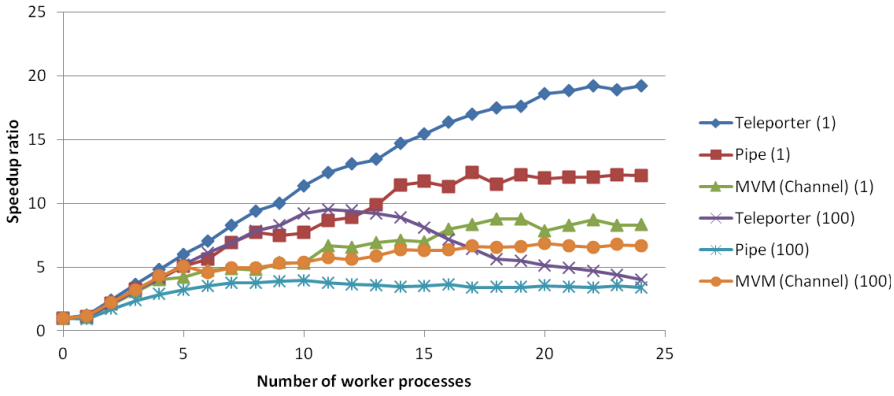

図 13　並列処理のアプリケーションベンチマークの実行結果
Fig. 13　Results of parallel processing application benchmark.

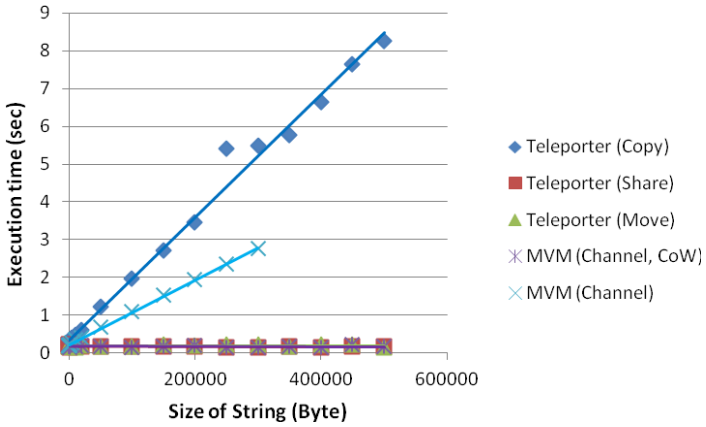

図 14　`String` オブジェクトの共有処理時間の測定結果
Fig. 14　Results of sharing `String` objects.

にコア数が増えたときもスケールするようにアロケータの並列性能を向上させていくことが重要な課題である．我々のメモリアロケータで問題となっているのは，複数のプロセスがメモリアロケータを奪い合うことによるロック待ちのコストである．そこで，プロセスごとに別の領域から共有メモリを確保するようにし，なるべく競合を減らすようにする必要がある [11], [12]．また，ロックフリーなデータ構造を用いてロック取得にかかるコストの削減も行いたい [13]．

現在の実装では送信先プロセスに送信オブジェクトのクラス定義がすでに存在することを前提としており，クラス定義が存在しない場合は例外が発生する．そこで，クラス定義を送れるようにすることで，`Process.fork` 後に定義されたクラスのオブジェクトも転送できるようにすることを検討している．また，コンパイル済みのバイトコードを共有することでコンパイル時間の削減といった使い方も考えられる．

Teleporter は非同期型のタスクキューとして利用することを想定している．同様に非同期型のタスクキュー機能を提供するメッセージキューシステム RabbitMQ [14] では，タスクキュー以外の使い方として出版–購読（Publish-Scribe）モデル [15] をサポートしている．出版–購読モデルを使うことでタスクキュー型の仕組みでは書きにくかった複数のプロセスに同じオブジェクトを転送・共有するブロードキャスト処理を簡潔に記述できる．また，オブジェクトの種類に応じて特定のプロセスにのみ送るマルチキャスト処理や，受信側が受け取りたいオブジェクトを指定できるフィルタリング処理なども導入していきたいと考えている．

現在の Teleporter は親子関係のあるプロセス間でしか通信できないという制約があるが，この制限を外すために後から共有メモリをマップしても適切に共有データにアクセスできるように改良することを考えている．具体的には共有メモリへのアクセスを（マップした共有メモリの先頭アドレス＋オフセット）という間接アドレスの形式に書き換える．これにより，共有メモリをマップしたアドレスに依存しなくなるため，後から共有メモリをマップしても適切に処理することができる．しかし，間接アドレスによるアクセスはアドレス計算のオーバヘッドが増えるため，性能低下を引き起こす可能性がある．今後，間接アドレスによってアクセスする実装を試作し，両者の性能を比較して導入すべきか検討していく．

8. 関連研究

Ruby 処理系を並列動作させる試みはいくつか存在する．MacRuby [3] や JRuby [4] は GVL の問題を解決し，スレッドを並列に動作させることができる．CRuby でも同様の研究が行われているが実装上の問題などにより実用化されていない [2]．スレッド並列ではユーザが適切に排他制御を行う必要があるが，我々は所有権モデルを導入することで安全に共有オブジェクトを扱えるようにしている．

CRuby では 1 プロセス中に複数の VM を動作させることで並列性を獲得する MVM [8] の研究が行われている．同様に，文献 [16] では Scheme 処理系で GVL の問題を解決するために MVM の導入を行っている．MVM の導入による並列性能の獲得は，我々が主張するプロセス並列による並列処理と競合するものではなく，状況に応じて両者を組み合わせていくことが今後重要になると考えている．

スクリプト言語 Python の C 言語実装である CPython [17] にも GVL と同様の問題があり，現在の実装ではスレッドは並列に動作しない．Python では代わりにプロセス並列プログラミングを可能にする multiprocessing [18] モジュールを提供している．multiprocessing はパイプやキューによるプロセス間通信をサポートしているが，転送時にシリアライズが必要である．また共有メモリによるプロセス間データ共有もサポートしているが，共有できるのはプリミティブなデータ型に制限されていて，かつユーザが共有メモリへのアクセスをロック制御しなくてはならない．本研究はシリアライズを軽量に行うことでオブジェクト転送を高速化しており，またユーザが排他制御を行わずに安全にオブジェクト共有を行える機構を持つ点が異なる．

オブジェクト共有に関する研究は分散共有メモリの分野で広く行われてきた．DiSOM [19] や Orca [20], [21] はオブジェクト単位で共有を行う分散共有メモリである．DiSOM はユーザがロックによる排他制御を行う必要があるが，Orca は共有データとその操作をカプセル化したデータオブジェクトというものを共有することで，ユーザが明示的に排他制御を行わなくて済む．分散共有メモリはローカルのメモリ空間にキャッシュを作ることでアクセスの高速化を図るため，このキャッシュ間のコヒーレンシをどう解決するかが問題となる．一方，Teleporter はキャッシュを作らずただ 1 つの実体へとアクセスするため，キャッシュコヒーレンシの問題は生じない．

プロセス間共有メモリを用いたオブジェクト共有を行う研究に XMem [22] がある．XMem は HotSpot JVM [23] 上に共有メモリを用いたプロセス間オブジェクト共有機構を実現する．クラス定義をプロセス間で共有するための様々な工夫が施されており，共有オブジェクトを通常のプライベートオブジェクトと区別せずに扱うことができる．XMem では明示的な排他制御を行う必要がある点が Teleporter と異なるが，クラス定義の共有や共有メモリの扱いなどは我々の研究の参考になると考えている．

オブジェクトの不変性や所有権の概念を型システムに取り込み，排他制御を行う研究は広く行われている [10], [24], [25]．ユーザが不変性や所有権を表すアノテーションを記述することで，排他制御を行うためのコードを自動的に生成する．これらの研究ではスレッド並列下における競合を扱っているが，我々はプロセス並列における所有権について扱っていて，かつアノテーションを記述しなくてよい点が異なる．文献 [26] はデータの所有権を静的に解析することで，通常の転送処理を自動的に Teleporter の move 操作にあたる処理に置き換え，メッセージパッシングのコピーオーバヘッドを抑える研究を行っている．このようにソースコードを解析して，転送処理を move や share 操作に自動的に置き換えることは今後検討していきたい．なお，この研究では不変オブジェクトの共有には対応していない．

我々の研究ではより高速なプロセス間通信路を用いることで高速化を実現したが，プロセス間通信や I/O 処理を OS レベルで改善することで高速化を目指す研究も広く行われている．ソケット通信やディスク I/O では，受信したデータをいったんカーネル空間にバッファしてからユーザ空間へコピーする．このメモリコピーのコストを削減するため，文献 [27] はユーザ・カーネル空間でバッファを共有してゼロコピー化を行うライブラリを設計，実装している．ゼロコピー化によって転送時オーバヘッドを削減する手法は広く知られており，Teleporter では move や share 操作によってこれを実現している．

9. おわりに

本稿では，効率的なオブジェクト転送や共有をプロセスで実現する Teleporter ライブラリを提案した．既存のプロセス間通信機構ではシリアライズのオーバヘッドが問題となっていたが，Teleporter では通信路にプロセス間共有メモリを用いることで通常よりも軽量なシリアライズによってオブジェクトの転送を行うことができる．評価より，パイプやソケットよりも高速にオブジェクト転送できていることが確認できた．またプロセス間でオブジェクトを共有する機構を設計し，実装した．共有機構では不変オブジェクトに限らず可変オブジェクトも共有することができる．また共有オブジェクトをユーザが安全に扱えるようオブジェクト共有のモデルについて議論した．

7 章で述べたとおり，解決すべき課題も多く残されている．6.2 節の並列処理ベンチマークの結果より，共有メモリアロケータの並列性能に問題があることが分かった．今後は並列性能を高めるために既存アロケータの知見 [11], [12], [13] を生かして共有メモリアロケータの性能

向上を行っていく．また，親子関係のないプロセス間でも通信を行えるように本機構を改良し，ユーザのプログラミングの幅が広がるようにしていきたい．

参考文献

[1] オブジェクト指向スクリプト言語 Ruby，入手先 ⟨http://www.ruby-lang.org/⟩.

[2] 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：プログラミング，Vol.48, No.10, pp.1–16 (2007).

[3] MacRuby, available from ⟨http://www.macruby.org/⟩.

[4] JRuby, available from ⟨http://jruby.org/⟩.

[5] Seki, M.: dRuby and Rinda: implementation and application of distributed Ruby and its parallel coordination mechanism, *Int. J. Parallel Program.*, Vol.37, pp.37–57 (2009).

[6] Gelernter, D. and Bernstein, A.J.: Distributed communication via global buffer, *Proc. 1st ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, *PODC '82*, pp.10–18, ACM (1982).

[7] Ruby: parallel processing made simple and fast — grosser/parallel — GitHub, available from ⟨https://github.com/grosser/parallel⟩.

[8] 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，情報処理学会第 86 回プログラミング研究会 (2011).

[9] 松本行弘：Ruby における実用的な多言語処理の実装，情報処理学会論文誌 プログラミング，Vol.2, No.2, pp.27–36 (2009).

[10] Boyapati, C. and Rinard, M.: A parameterized type system for race-free Java programs, *Proc. 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, *OOPSLA '01*, pp.56–69, ACM (2001).

[11] Evans, J.: A Scalable Concurrent malloc (3) Implementation for FreeBSD, *Proc. BSDCan*, No.3, pp.1–14 (2006).

[12] Wolfram Gloger's malloc homepage, available from ⟨http://www.malloc.de/en/⟩.

[13] Michael, M.M.: Scalable lock-free dynamic memory allocation, *Proc. ACM SIGPLAN 2004 conference on Programming language design and implementation*, *PLDI '04*, pp.35–46, ACM (2004).

[14] RabbitMQ — Messaging that just works, available from ⟨http://www.rabbitmq.com/⟩.

[15] Eugster, P.T., Felber, P.A., Guerraoui, R. and Kermarrec, A.-M.: The many faces of publish/subscribe, *ACM Comput. Surv.*, Vol.35, pp.114–131 (2003).

[16] Tew, K., Swaine, J., Flatt, M., Findler, R.B. and pdinda@northwestern.edu, P. D.: Places: adding message-passing parallelism to racket, *Proc. 7th symposium on Dynamic languages*, *DLS '11*, pp.85–96, ACM (2011).

[17] Python Programming Language — Official Website, available from ⟨http://python.org/⟩.

[18] multiprocessing — Process-based parallelism, available from ⟨http://docs.python.org/py3k/library/multiprocessing.html⟩.

[19] Castro, M., Guedes, P., Sequeira, M. and Costa, M.: Efficient and Flexible Object Sharing, *Proc. 1996 Int. Conf. on Parallel Processing*, Vol.I, pp.128–137 (1995).

[20] Bal, H.E., Kaashoek, M.F. and Tanenbaum, A.S.: Orca: A Language for Parallel Programming of Distributed Systems, *IEEE Trans. Softw. Eng.*, Vol.18, pp.190–205 (1992).

[21] Bal, H.E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Rühl, T. and Kaashoek, M.F.: Performance evaluation of the Orca shared-object system, *ACM Trans. Comput. Syst.*, Vol.16, pp.1–40 (1998).

[22] Wegiel, M. and Krintz, C.: XMem: type-safe, transparent, shared memory for cross-runtime communication and coordination, *Proc. 2008 ACM SIGPLAN conference on Programming language design and implementation*, *PLDI '08*, pp.327–338, ACM (2008).

[23] Java SE HotSpot at a Glance, available from ⟨http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html⟩.

[24] Birka, A. and Ernst, M.D.: A practical type system and language for reference immutability, *Proc. 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, *OOPSLA '04*, pp.35–49, ACM (2004).

[25] Kerfoot, E., McKeever, S. and Torshizi, F.: Deadlock freedom through object ownership, *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pp.3:1–3:8, ACM (2009).

[26] Negara, S., Karmani, R.K. and Agha, G.: Inferring ownership transfer for efficient message passing, *Proc. 16th ACM symposium on Principles and practice of parallel programming*, *PPoPP '11*, pp.81–90, ACM (2011).

[27] Khalidi, Y.A. and Thadani, M.N.: An Efficient Zero-Copy I/O Framework for UNIX, Technical report, Mountain View, CA, USA (1995).

中川 博貴

1986 年生まれ．2010 年慶應義塾大学理工学部情報工学科卒業．2012 年東京大学大学院情報理工学系研究科創造情報学専攻修了．同年グーグル株式会社入社（現職）．

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，2012 年 Heroku, Inc. にて Ruby 処理系の開発に従事（現職）．システムソフトウェア，特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．