Ruby用マルチ仮想マシンによる並列処理の実現

○笹田 耕一 (東京大学),
卜部 昌平, 松本 行弘 (NaCL),
平木 敬 (東京大学)

東京大学
情報理工学系研究科
創造情報学専攻

この発表について

- 背景：CRubyの並列処理primitiveの欠如
- 提案：マルチ仮想マシン（MVM）で，並列処理
  - RubyをMVM（1プロセス中に複数VM）に対応
  - VM間通信機構Channel→高速なオブジェクトの転送
- 設計と実装
  - MVMの設計と実装（グローバル変数，I/O，etc）
  - Channelの設計と実装
    - データ構造
    - CoWによるStringのO(1)転送
- 評価，関連研究，まとめ

背景

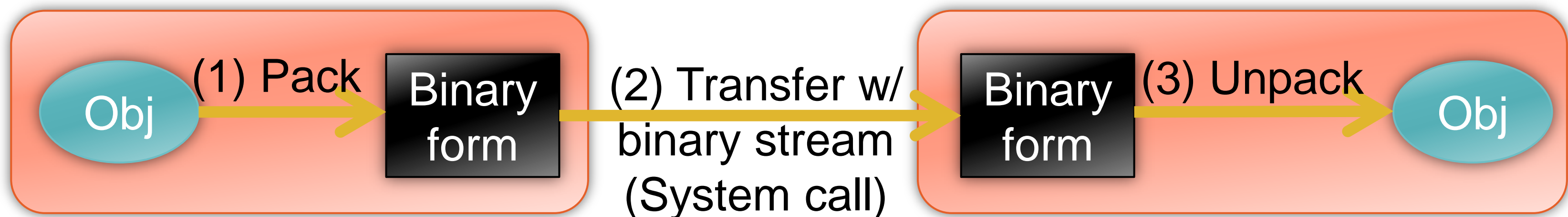
- オブジェクト指向プログラミング言語Ruby
  - ウェブアプリケーション（RoR）で多く利用
  - 利用範囲の拡大（HPC，組込み，etc…）
  - Ruby処理系（Ruby 1.9）
    - CRuby（MRI）：今回の研究・開発対象
    - 他にも JRuby，MacRuby，Rubinius，MagLev など
  - 2012/2月 Ruby 2.0 リリース予定（？）
- マルチ/メニーコア化→Rubyでも並列計算したい
- Ruby には，並列処理の決定版ができていない
  - 案1：スレッドで並列（JRubyやMacRubyなどは対応）
  - 案2：プロセスで並列
  - （ノードで分散，は今回対象外）

背景：スレッド並列

- 議論
  - 利点
    - よく知られた並列計算モデル（共有メモリ）
    - オブジェクトを共有でき，密な並列処理が書きやすい
  - 欠点
    - よく知られた悲劇
    - オブジェクトを共有でき，変更に細粒度ロックが必須
    - シングルスレッドより遅い，高速な細粒度ロックの実装は大変
- 現状
  - CRubyはGiant VM Lock（GVL）をもつスレッドのみ実行（逐次）
    - 処理系に細粒度排他制御を持たせる必要がない → ○メンテナンス，○逐次性能
  - [笹田2007]，JRuby，MacRuby，Rubinius（次の版）で並列Thread
  - 処理系（ライブラリ）をバグなく軽量に並列対応するのは困難[笹田2007]
  - **簡単に書けるスクリプト言語で「難しい」スレッド処理を書かせる必要があるのか？　とくに，Rubyはなんでも弄れる（e.g. オープンクラス）ので適切な同期を判断するのは困難では？**

背景：プロセス並列

- OSが提供するIPC / Socket 機能を提供
  - File (pipe, etc), Socket (TCP/IP, etc) 等，システムコール
  - dRuby（分散Ruby. RMIを実現）
- 利点：Rubyオブジェクトの転送はコピーであるため，排他制御不要（w/o Threadの悲劇）
- 問題：直列化（RubyではMarshal）と復帰，コピー（複数）のため，Rubyオブジェクトの転送が遅い

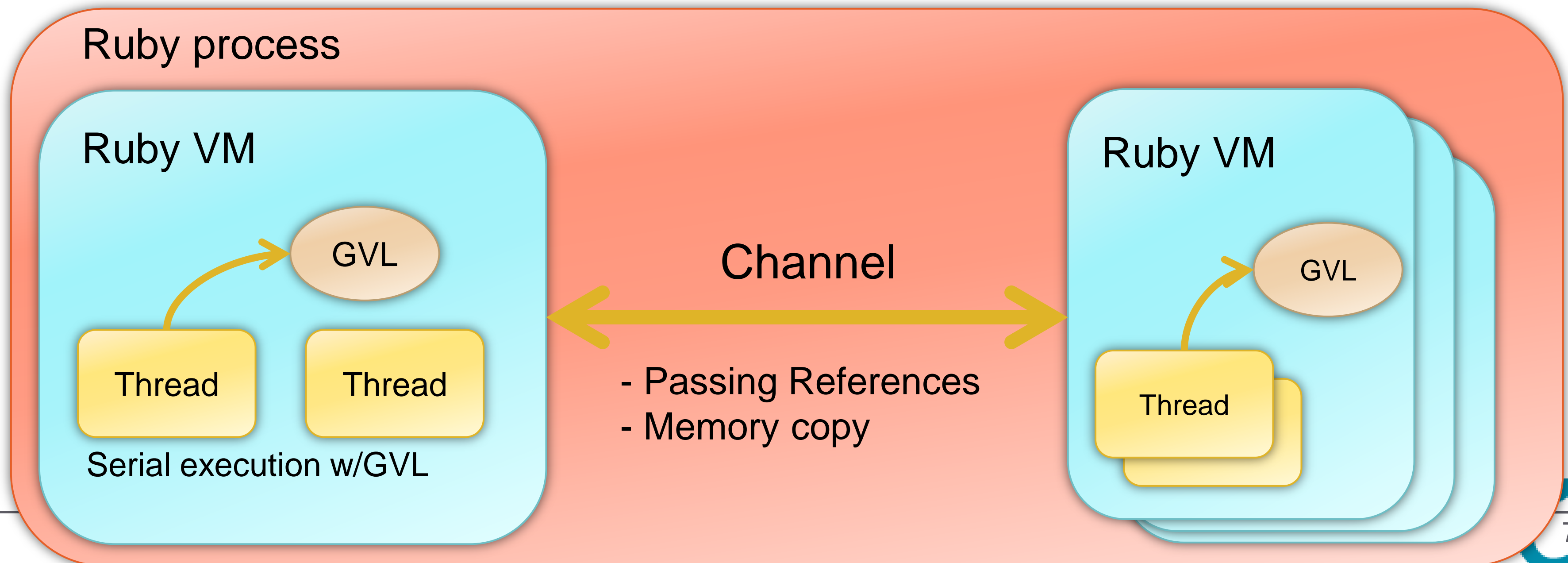


問題のまとめ

- スレッド並列は大変
  - Ruby プログラミング大変
  - 処理系開発大変
- プロセスはオーバヘッド大
  - Rubyオブジェクトのプロセス間通信
  - 直列化のオーバヘッド，コピーのオーバヘッド

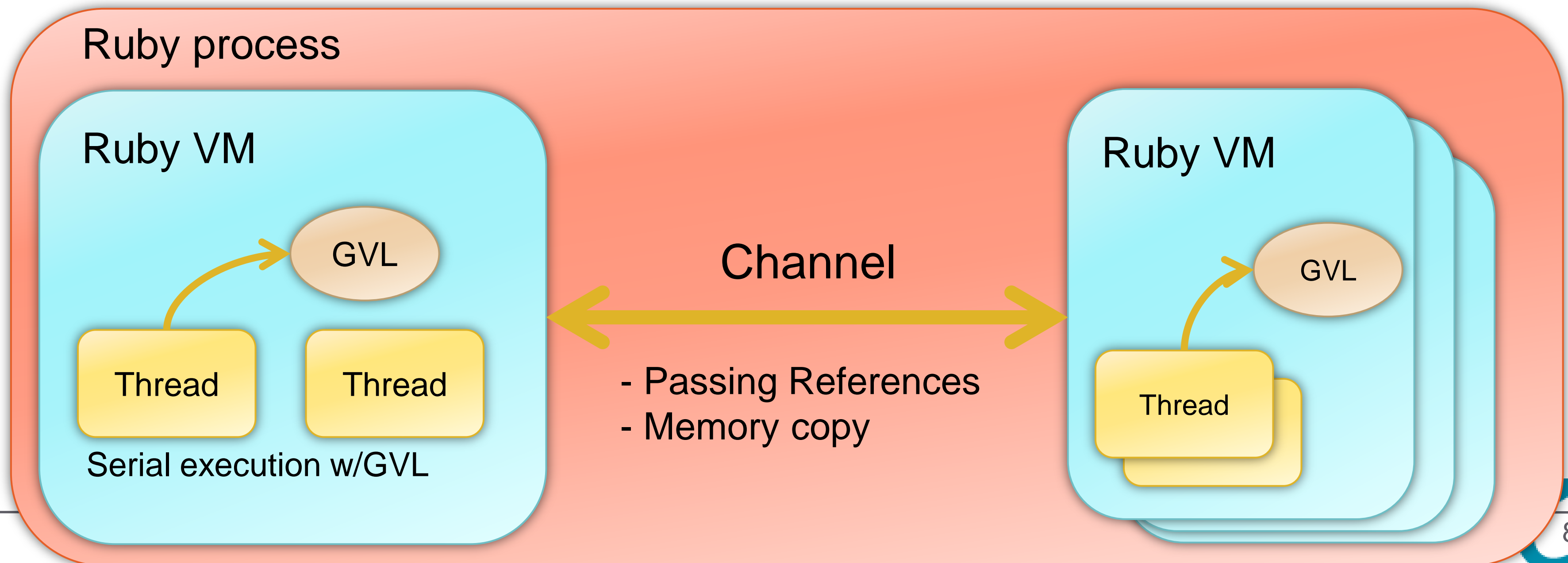
提案：マルチ仮想マシン（MVM）による並列処理 (1)

- マルチ仮想マシン（MVM）
  - **1つのプロセス**に複数のVMを実行
  - 1つのVMはこれまで通り**逐次処理**（GVL）
  - 各VMは独立に，**並列に実行**
    - ヒープは各VMに存在し，GCもVMごとに並列に実行
    - 細粒度同期は（ほぼ）不要：○プログラム簡単，○シングル性能も良い



提案：マルチ仮想マシン（MVM）による並列処理 (2)

- ChannelによるRubyオブジェクトの転送・同期
  - Rubyオブジェクトを転送
  - キューになっており，受信時に待ち合わせ可能
  - 同一アドレス空間であることを利用した高速な転送
    - 例：CoWによるO(1)の文字列転送



MVMの使い方（API）

- 生成：vm = RubyVM.new(“ruby”, “[arg]”)
  - [arg] は Ruby インタプリタに渡す引数と同じ
  - プログラムはファイル，もしくは文字列（-e option）
- 開始：vm.start([arg]+)
  - [arg] で渡したオブジェクトは，子VM側でRubyVM::ARGVでアクセス可能（コピー）
  - 他のVMと一切データを共有しない
    - コードも共有しないのはちょっと不便
      →今後の課題（できれば，fork みたいなことがやりたい）
- 待ち合わせ：vm.join

- Ruby API と同様にC 関数としても同様の API を用意

Channelの使い方（API）

- 生成：ch = RubyVM::Channel.new
- オブジェクト送信：ch.send(obj)
  - Marshal できるオブジェクトを送信可能（現在の制限）
  - ch 自体も ch で送信可能
- オブジェクト受信：obj = ch.recv
  - 送信されていなければ待つ（ブロック）
  - 受信されたオブジェクトは複製（Channel以外）
- 送信・受信するVMは問わない
  - ChannelはVMとは独立
  - ch.send(obj).recv() が可能（意味ないが）
  - 複数のVMが ch.recv() でブロックしていた場合，どれか1つのVMがオブジェクトを recv()

MVM, Channel の利用例 1 Master → 10 Worker

```
# parent.rb
ch = RubyVM::Channel.new
vms = (1..10).map{|i|
  vm = RubyVM.new("vm#{i}", 'child.rb')
  vm.start(ch)
}
tasks.each{|task| ch.send(task)} # worker に仕事送信
vms.each{ch.send(nil)}
vms.each{|vm| vm.join} # 終了を待つ

# child.rb
ch = RubyVM::ARGV.shift
loop{break if task = ch.recv; work(task)}
```

MVMの設計と実装

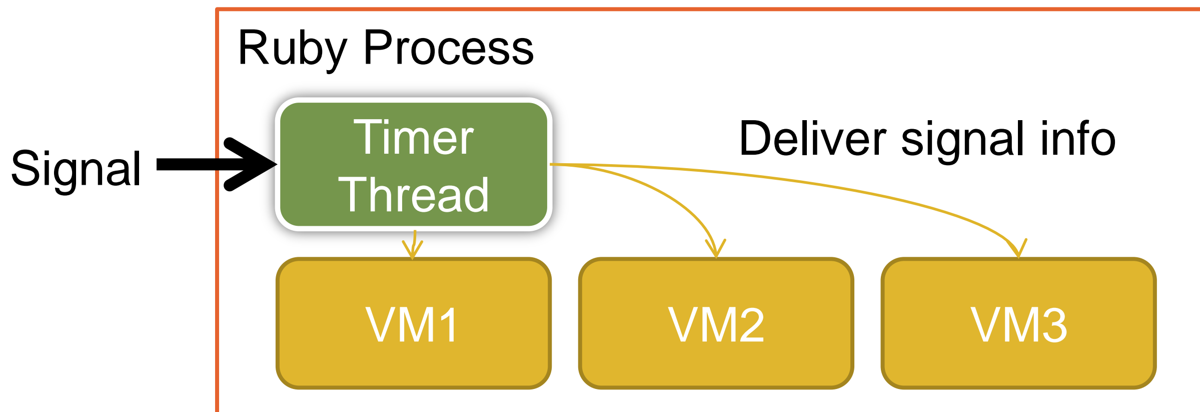
- CRubyのコードを利用（Ruby 1.9.2）
  - 過去のプログラム資産（Ruby，C）との互換性堅持
    - 全部書き換えることができれば，簡単な話なんですが…
- VMは専用のネイティブスレッドを1つ以上持つ
  - VMの操作はThread操作とほぼ同様
- <u>プロセスグローバルな情報をVMローカルに</u>
  - CRubyはプロセスに1VMであることを前提とした設計
  - Cのグローバル変数（各VMごと持つように分離）
  - シンボル表（VM間で共有）
  - カレントワーキングディレクトリ（CWD）（分離）
  - シグナル（dispatcher を用意）
- 拡張ライブラリの仕様を拡張
  - でも，互換性は維持する→メインVMは変更不要

Cのグローバル変数

- CRubyはグローバル変数を多用
- グローバル変数 → TLSに格納
  - TLS: Thread Local Storage
    - pthread_getspecific(), gcc の __thread など
  - VM構造体へのポインタをTLSに格納
  - VM構造体に格納
    - 専用メンバを追加
      - GCなど
    - 汎用フィールドを追加
      - key = rb_vm_key_create(); // 汎用フィールドの領域を確保しindexを返す
      - rb_vm_specific_ptr(key); // 汎用フィールドの領域を返す
    - rb_cString など，よく利用されているグローバル変数はマクロで汎用フィールドにアクセスするように定義
      - #define rb_cString (*rb_vm_specific_ptr(rb_vmkey_cString))

MVMの設計と実装：プロセスグローバルな情報をVMローカルに
CWD，シグナル

- CWDはVMごとに持つように変更
  - openat() などのシステムコールを利用（Solaris 由来）
  - 対応して無ければ，パス名に追加（不十分）
- シグナル
  - プロセスに1つシグナルを受けるタイマスレッドを用意
  - 全VMにそこからシグナル配送



MVMの設計と実装
拡張ライブラリ

- Ruby処理系と同様にグローバル変数を利用
  → 対応が必要
- 拡張ライブラリ仕様の拡張
  - 従来：ロード時，初期化関数 Init_foo() が呼ばれる
  - 拡張：InitVM_foo() が定義されていればMVM対応
    - Init_foo() はプロセスにロード時に1度呼ばれる
      - rb_vm_key_create() など，必要な処理を行う
    - InitVM_foo() はVMにロードされる度に呼ばれる
  - 対応は，たいていグローバル変数の置き換え
- 非対応拡張ライブラリは，メインVMのみ利用可能
  - メインVM：最初に起動したVM
  - 1VM利用時には完全互換
  - 最悪，メインVMに処理を依頼するような対処が可能

Channelの設計と実装

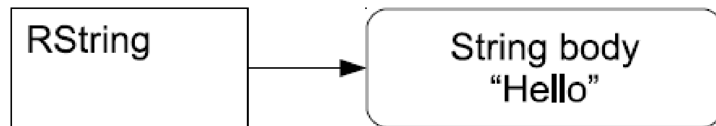
- 排他制御するキューとして実装
  - （さぼって）pthread_mutex_lock() で同期
  - recv() 時，send() されていなければ block する
- 参照カウントで管理
  - 参照カウントが 0 になったら解放
  - 制限：ch 自分自身を送りrecvされないと循環参照→解放されず
- 転送
  - Marshalによる直列化と復元
  - 型によって，最適化（とくに文字列は O(1) で転送）

Channelの設計と実装
Rubyオブジェクトの転送方法

- 型によって異なる転送方法
  - <u>文字列：CoWによるO(1)の転送</u>
  - 即値型（Fixnum，true, false, nil，Symbol）：そのまま
  - Float型：実際の double 値のみ転送（Marshal不要）
  - 配列：中身が上記型だった場合，
  - それ以外：Marshal して転送（文字列）

Channelの設計と実装 Stringオブジェクトの転送（1）

- Rubyの文字列処理はCopy on Write (CoW) を実装
  - 複製作成時，文字列の実体はコピーせず，破壊的操作が起こったとき，初めてコピー



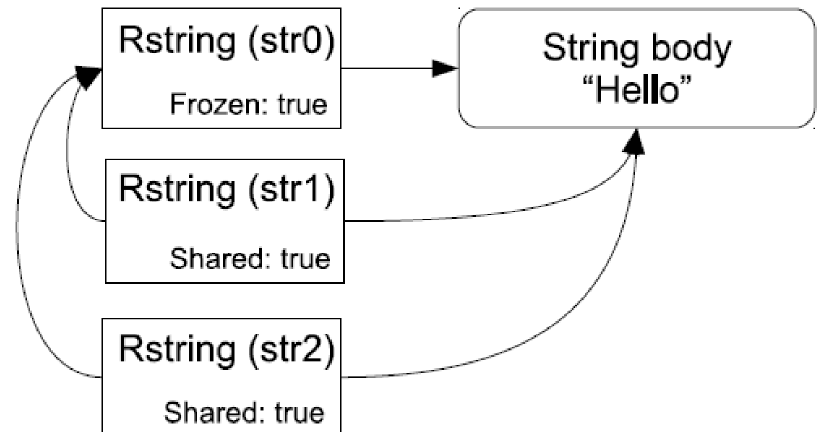


図 6　Ruby 1.9 での文字列の取り扱い

Channelの設計と実装 Stringオブジェクトの転送（2）

- Channelによる転送時，CoWの機能をそのまま利用して，文字列の実体をコピーせず，参照のみ転送

→ 文字列の長さによらず，O(1) で転送

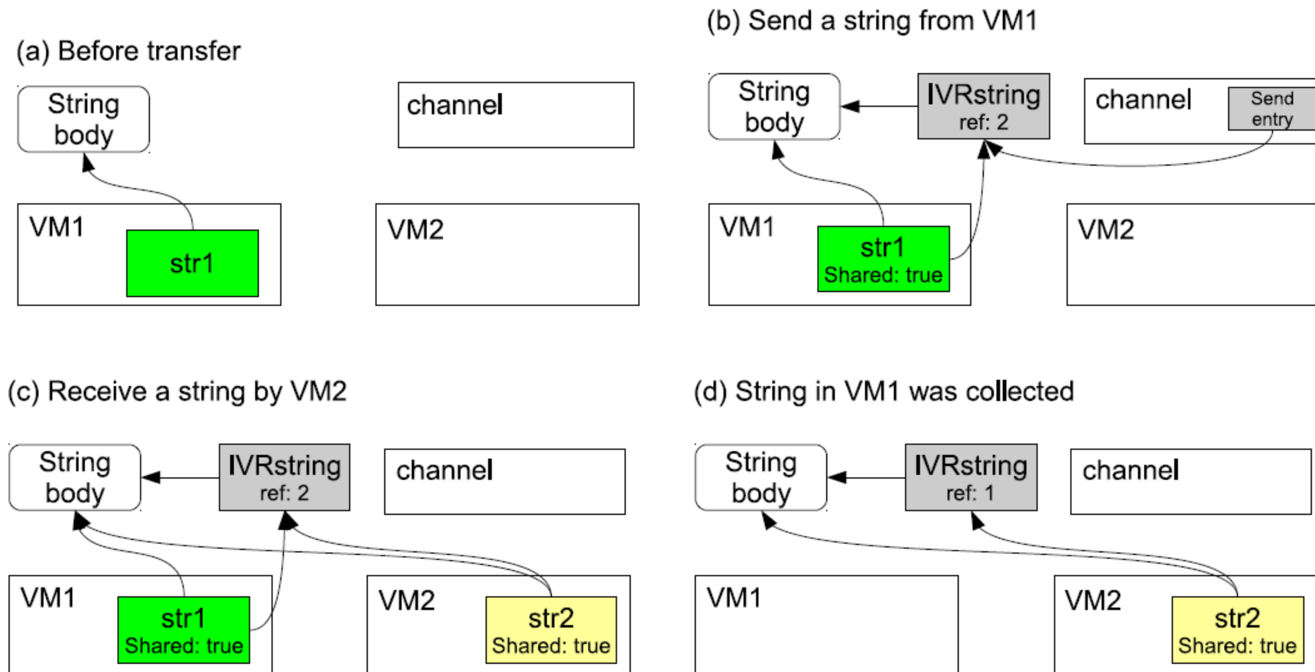


図 7　String 型オブジェクトの転送

Channelの設計と実装
dRubyバックエンドに対応

- dRuby（分散Ruby）はリモートノードでの計算に対応するために TCP/IP をバックエンドに持つ
- MVMでのVM間転送の場合，TCP/IPは無駄
  → 転送のバックエンドを Channel に対応
- URIで drb+mvm://… と指定
- あとはこれまで通りの dRuby 操作が可能

評価

- 評価環境
  - Intel Xeon E7450 4 cores x 6 processors = 24 cores
  - Linux 2.6.26-2-amd
  - gcc 4.3.2（-O3）
  - 比較対称のRuby : ruby 1.9.2dev（2010-08-16）

評価
シングル実行

	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

- 遅い！
  - TLSが予想外に遅い
  - 調査すると，shared library（shlib）中のTLSアクセスはヘルパ関数を利用しており，それが予想外に遅い

  (処理系の大部分は shlib で提供されることが多い)

→ **今後，TLSアクセスの削減が必須**

VM構造体へのポインタを持ち回るとか・・・

評価
生成・終了・待ち合わせ

	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

（500回の実行時間）

- **現状はプロセス生成と同程度**
  - 現在はVMの生成処理を従来と同じように実行しているため．今後は共有できる部分，省略できる部分を省略していく必要がある

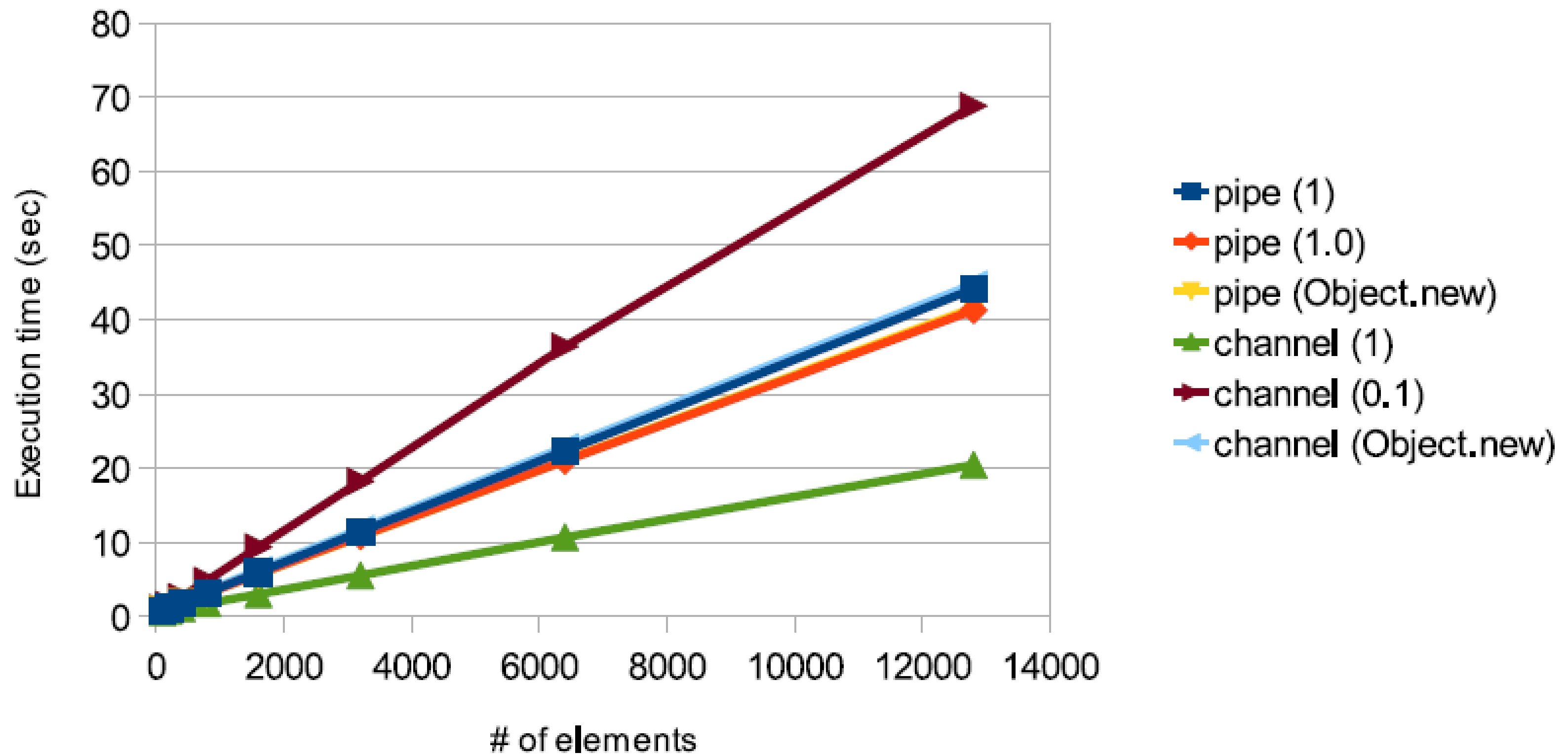
評価
オブジェクトの転送速度

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

1万回 ping pong した結果

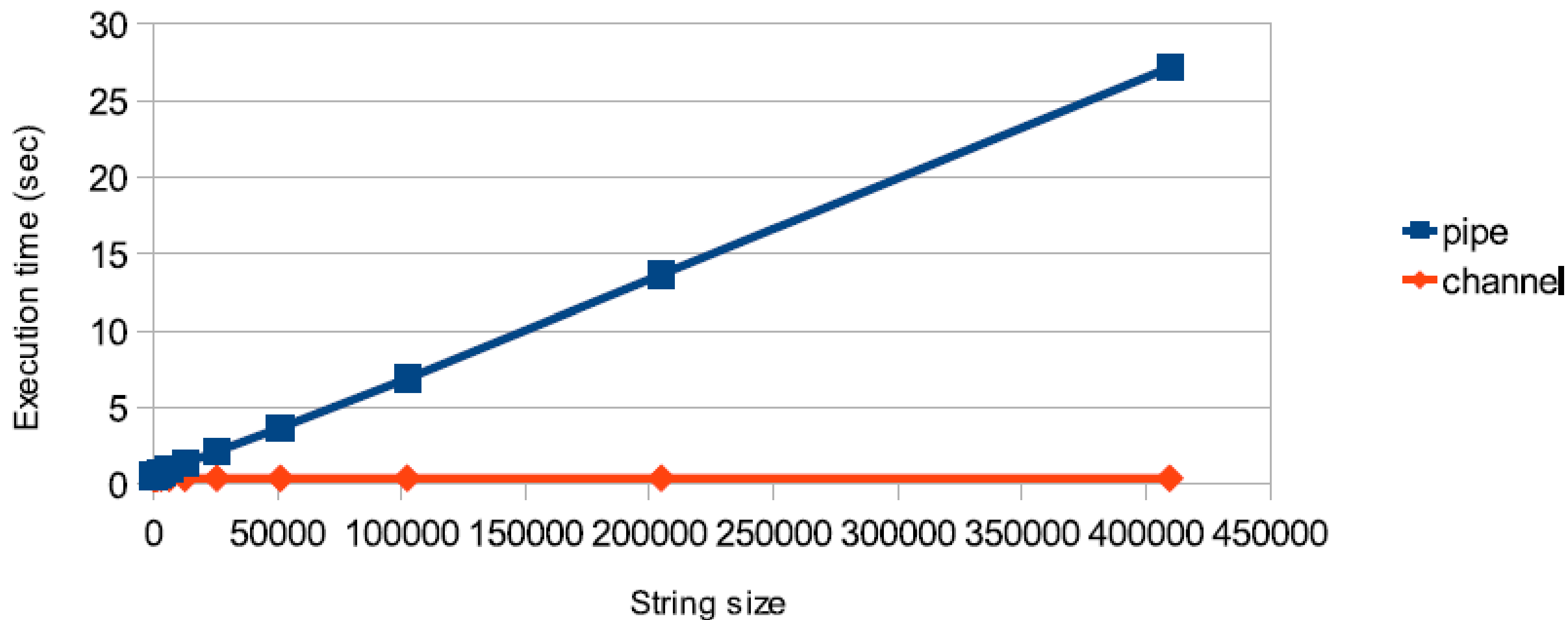
- 1, 1.1 などは pipe より若干速い
- Marshal が必要な Bignum，Object は若干遅い（原因調査中）

評価
オブジェクトの転送速度（Array）



- Marshalが必要ない場合（Fixnum），それなりに速い
- Floatが異様に遅い → 要再検討

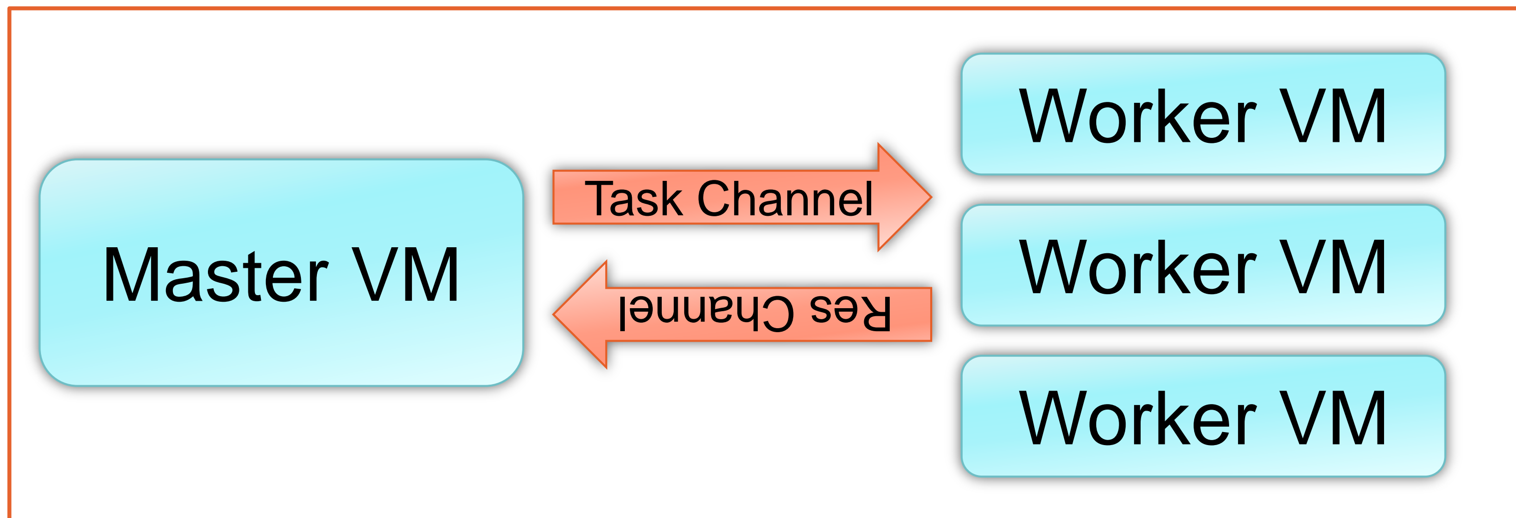
評価
オブジェクトの転送速度（String）



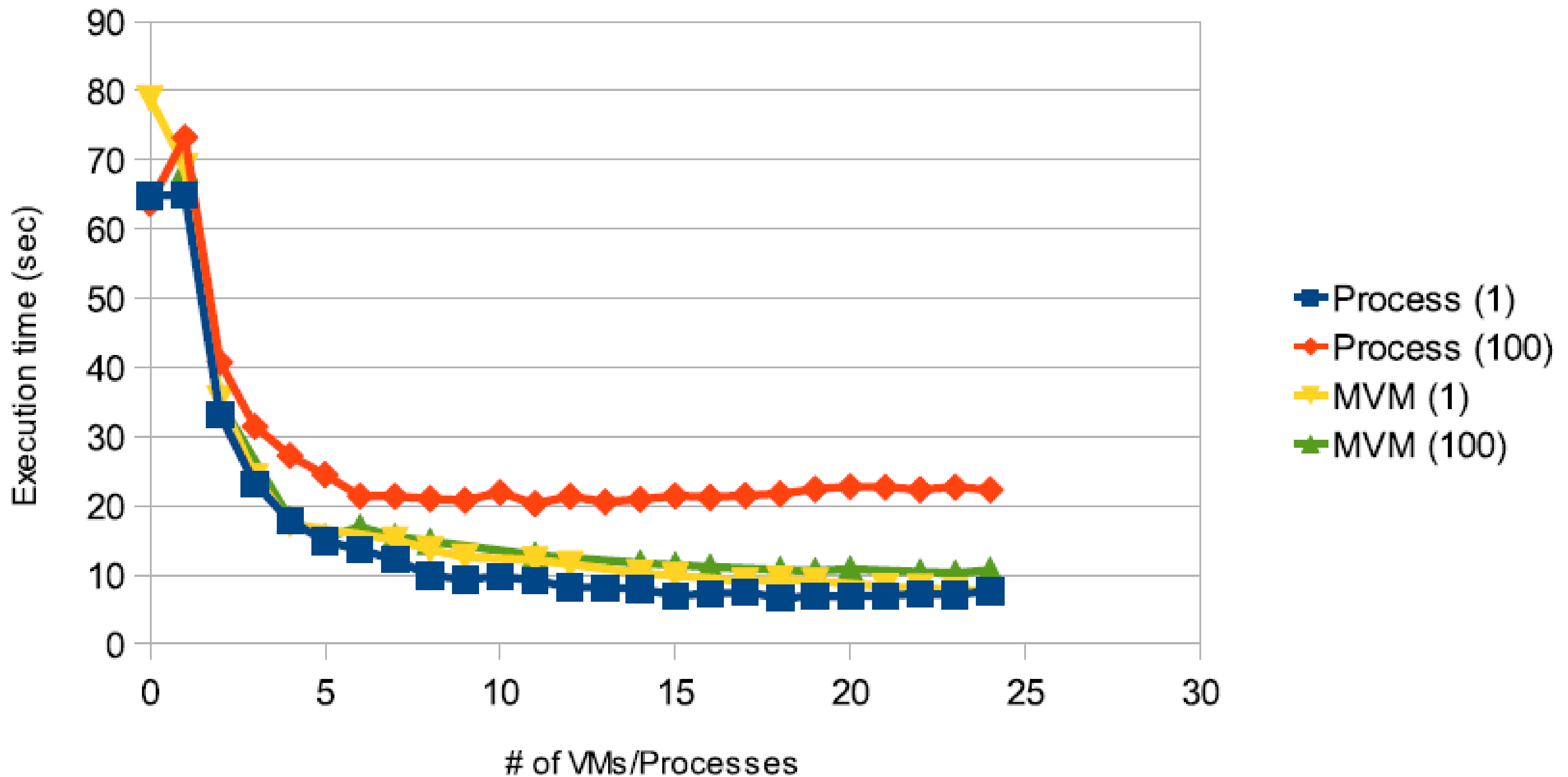
- 文字列の長さによらず一定であることを確認

評価
HTML生成アプリケーション

- Master VM が Worker VM に処理を依頼
- Worker VMは ERB（HTML template）でHTMLを生成
  - Channelは2本（requestとresponse）
  - 全 WorkerVM は request チャンネルで処理を待つ
  - 返すHTMLは <div> 要素が1つの場合と100の場合
    - 要するに，返す HTML（文字列）のサイズが違う

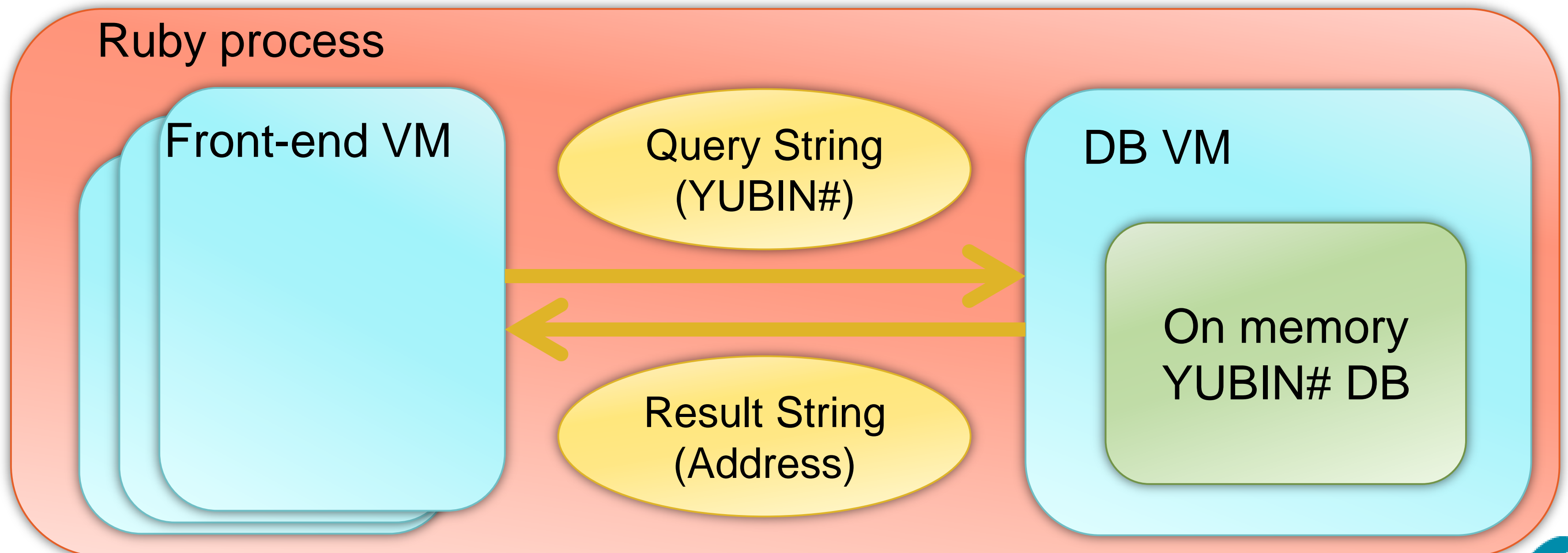


評価
HTML生成アプリケーション（結果）

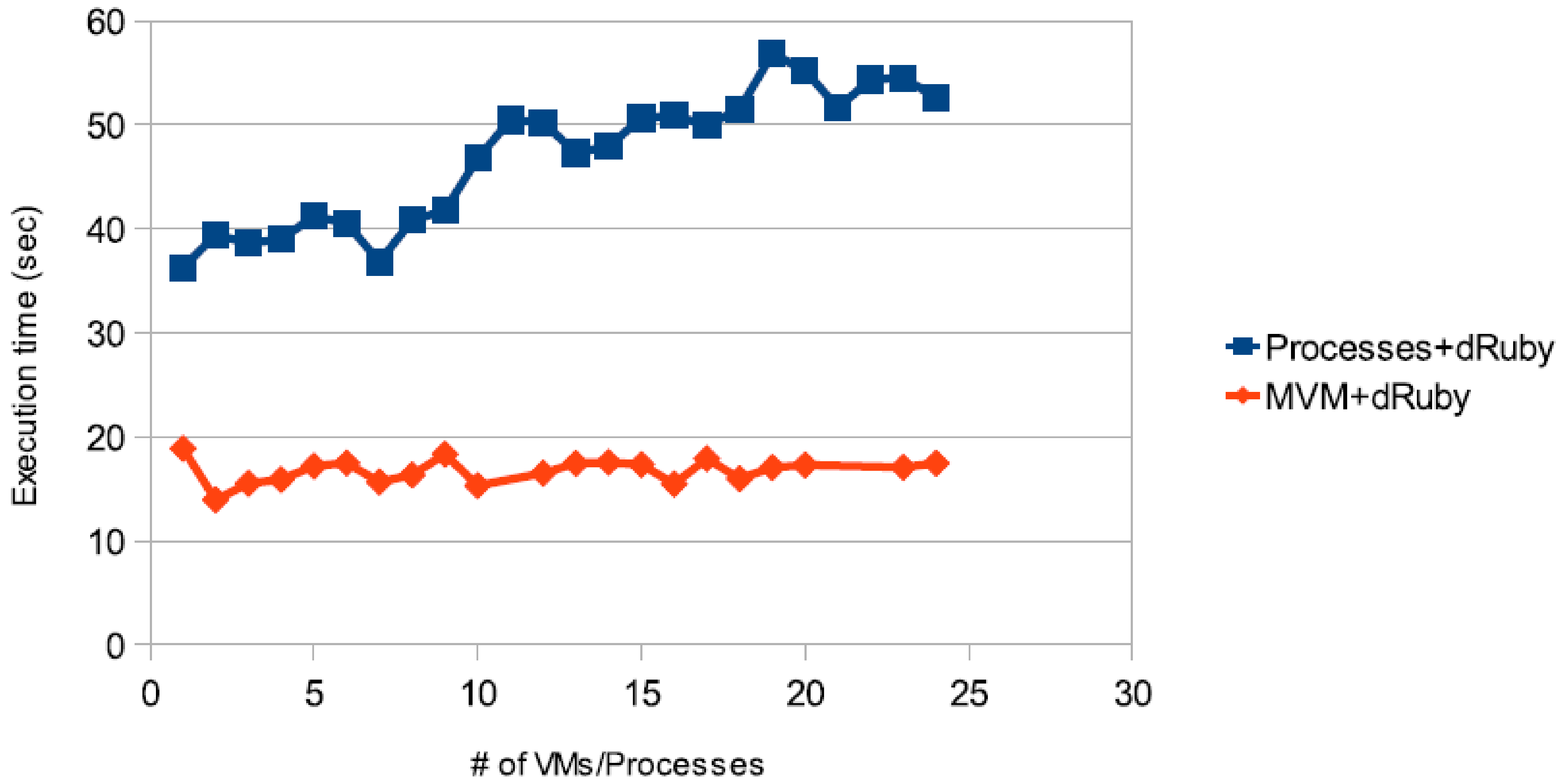


評価
DBアプリケーション

- 郵便番号から住所をひくアプリ
  - DB VMはオンメモリにデータを保持
- 通信には，dRuby（on Channel or TCP/IP）を利用



評価
DBアプリケーション（結果）

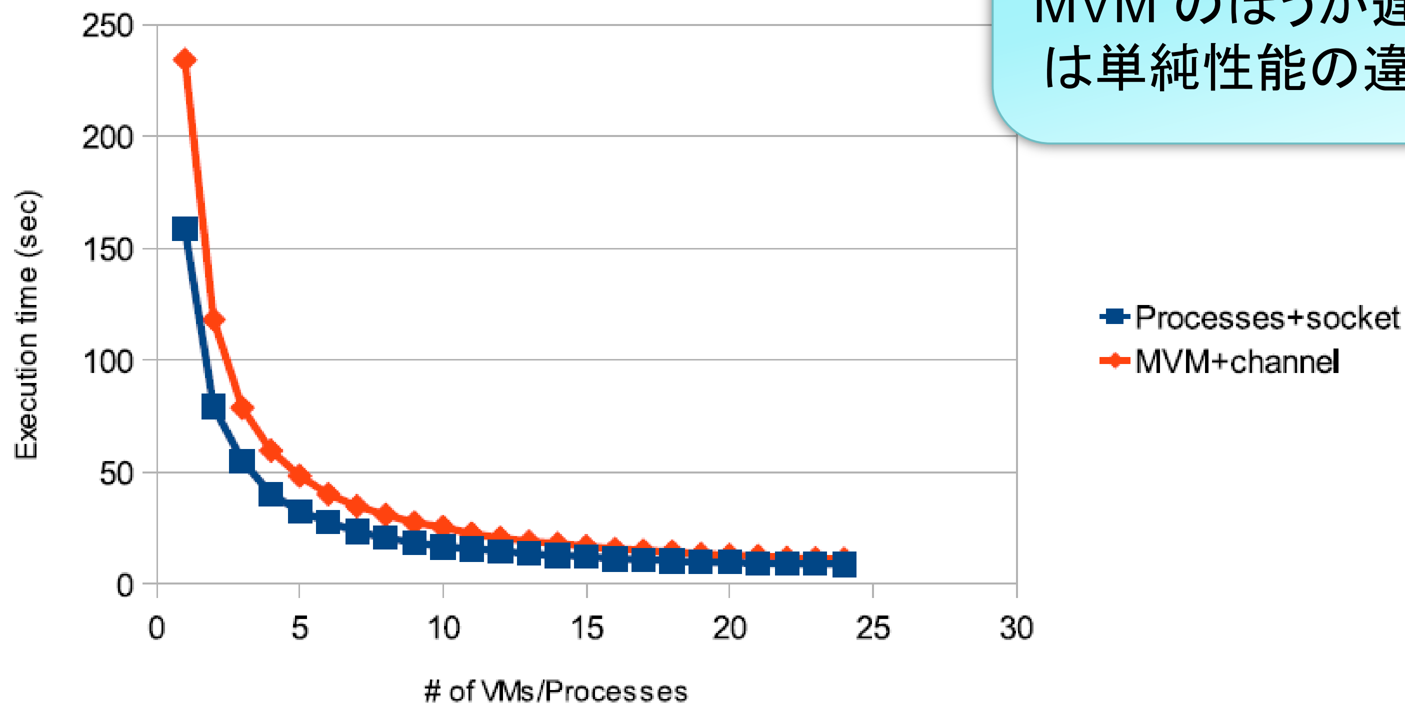


評価
AOベンチ

- レイトレベンチ
  - タスクは各行ごとに処理

両方綺麗にスケール

MVM のほうが遅いのは単純性能の違いか



議論

- 応用範囲（並列処理以外）
  - ○アプリケーション組込み
  - ○複数のRubyVMで資源共有し，**省資源，起動高速化**
  - RubyOSの基盤（？）
- 制限
  - ネイティブスレッドに1つしかVMが持てない→多分問題無い
  - 拡張ライブラリの対応が必要 → メインVMはOK
  - 1つのVMが強制終了すると，他のVMを道連れ．資源解放など，プロセス終了に任せていた部分の対応が不十分 → MVMの品質

- fork でいいじゃん → fork のない環境だってあるんだ！

関連研究
JSR121 Application Isolation

- MVMの元ネタ（Multi-tasking VM）
- 本研究とは目的が違う（資源共有による省資源化）
- 用語：VMをIsolation，ChannelをLinkという
- Linkの仕様が異なる（目的が違うから）
  - 転送可能オブジェクトに制限
  - 送信側もブロック（ランデブー）
- 論文いろいろ

関連研究
並列計算モデル

- 共有モデル v.s. **<u>メッセージパッシング</u>**
  - Erlang：オブジェクトが全部 Immutable なので，そもそも共有とか関係無い
  - Go，Scala – Channel，Actor などでメッセージパッシングをサポート．しかし，共有も可能
- Rubyは環境から何から簡単に弄れてしまう（例えばクラスの再定義）ので，適切な同期は困難

→完全に**環境を隔離**できたほうが良い → MVM

（多分，そっちのほうが気楽）

関連研究
プロセス間通信

- 共有メモリを用いたプロセス間Rubyオブジェクト転送機構Tunnel [中川2011]
  - MVMは同一アドレス空間のため，転送が高速
    - ただ，プロセスで切った方が管理がしやすいのはたしか
    - 資源の共有は，fork との競争
  - 使い勝手の比較などはこれから検討
  - 多分，どちらも進めていく

関連研究
Racket（PLT Scheme）の Place

- Tew, Swaine, Flatt, Findler, and Dinda: Places: Adding Message-Passing Parallelism to Racket , DLS 2011 (2011-10)
- 問題意識も内容も本研究と殆ど同じだった！
  - 用語：VM→Place，Channel→Place channel
  - 親PlaceのImmutableなものは共有
  - 共有できるデータ構造（ベクタ）を用意
  - TLSの問題について検討しており，JITでVM構造体相当のポインタを埋め込んでしまう
  - CWDやシグナル，拡張ライブラリなどには言及なし
  - 対象言語の違い（Scheme，Ruby），実装の違い，転送方法の違い（メモリ管理方法の違い）

今後の課題

- TLS 問題の解決
- Channel の転送の改善
  - データ構造を効率的なものに
  - ファイルが送れるように
  - 手続き（Proc）が送れるように
  - CoW を使った軽量な転送を文字列以外にも適用（例：NArray）
  - （安全に）共有可能なデータ構造が送れるように
  - VMの状態変更を送れるように（現状は待つしかできない）
- MVMでのプログラミングパターンの発見
- 資源の共有による省資源化，高速起動

  - Ruby 3.0 くらいに入れられるといいなぁ

Ruby用マルチ仮想マシンによる並列処理の実現
まとめ

- 背景：Rubyの並列処理primitiveの欠如
- 提案：マルチ仮想マシン（MVM）で，並列処理
  - RubyをMVM（1プロセス中に複数VM）に対応
  - VM間通信機構Channel→高速なオブジェクトの転送
- 設計と実装
  - MVMの設計と実装（グローバル変数，I/O，etc）
  - Channelの設計と実装
    - データ構造
    - CoWによるStringのO(1)転送
- 評価：並列処理による性能向上を確認
- 今後の課題：いろいろ

DB ベンチマーク pipe v.s. TCP/IP socket v.s. channel