Vol.2014-HPC-146 No.9
2014/10/2

メタプログラミングに適した PGAS 通信クラスライブラリの検討

野瀬貴史[†1,†2]　安島雄一郎[†1,†2]　佐賀一繁[†1,†2]　志田直之[†1,†2]　住元真司[†1,†2]

Advanced Communication for Exa (ACE)プロジェクトにおいて開発している Advanced Communication Primitives (ACP)ライブラリを利用し，メタプログラミング機能により高い生産性を持つスクリプト言語である Ruby 上に PGAS モデルをサポートするクラスライブラリを実装した．実装では，リモートプロセス間コピーの最適化の余地を拡大するため，リモートの値を操作するタイミングを遅延させる機構を導入した．性能を評価した結果，言語処理系由来と見られるレイテンシが隠蔽されるケースとされないケースがあった．

Toward PGAS Communication Class Libraries Suitable for Metaprogramming

TAKAFUMI NOSE[†1,†2]　YUICHIRO AJIMA[†1,†2]
KAZUSHIGE SAGA[†1,†2]　NAOYUKI SHIDA[†1,†2]　SHINJI SUMIMOTO[†1,†2]

We implemented a PGAS communication library based on the Ruby language that has high productivity derived from its metaprogramming capability. The implementation used the Advanced Communication Primitives (ACP) library as a low-level library that was developed under the Advanced Communication for Exa (ACE) project. We utilized a mechanism that handles the remote data lazily to extend optimization opportunity of remote-to-remote data copy operation. As a result of performance evaluation, the latency that was caused by Ruby was overlapped in some cases on multiple node execution, but the case with no overlapping remained.

1. はじめに

ACE （Advanced Communication for Exa）プロジェクト[1]では，エクサスケールに向けて省メモリ・低レイテンシを両立する通信ライブラリ ACP (Advanced Communication Primitives)[2] の開発に取り組んでいる．我々は，ACP の上に構築されるプログラミング環境のあるべき姿の検討も始めている．

HPC (High Performance Computing) プログラミングの生産性を高める試みとして，PGAS (Partitioned Global Address Space) 言語が提案されている. HPC のプログラミングのデファクトスタンダードは MPI (Message Passing Interface)であるが，MPI によるプログラミングは，高い性能を出すことができる反面，並列プログラミングで考慮する必要がある要素，例えばデータの配分やその位置，通信と待ちのタイミングを詳細に記述することを要求し，計算を記述する際にもデータ分割を強く意識する必要がある．このため，生産性に問題がある．これに対して，PGAS 言語はユーザービューとして局所性を持たせるように区分された大域配列を提供し，その配列上の操作が簡潔に行えるような機能を言語仕様や標準ライブラリに持たせることで，局所性を生かしつつ生産性を高めることを狙ったものである．PGAS 言語の内，Co-Array Fortran[3], UPC (Unified Parallel C)[4], Titanium[5] は既存の高いパフォーマンスを持つ言語に新たな言語機能を追加することで並列計算の記述の生産性を高める言語である．また，Chapel[6], X10[7], Fortress[8] は最初から並列計算を記述することを考慮に入れ全く新規に設計した言語である．既存の言語を拡張するアプローチは，実績のある処理系を基盤に用いることができ，また学習コストが低いという利点はあるものの，ディレクティブ以外の新規のキーワードや構文を導入することで拡張を実現している言語では既存の言語との互換性が失われる．ディレクティブで実現している場合も，互換性は保たれるため既存のアプリケーションの並列化には適しているものの，ディレクティブは言語の文法に統合されているわけではないため，大量の#pragma が出現するなどしてソースコードの見通しが悪化する. 新規の言語は, 学習コストが高い上，言語処理系自体の安定性や最適化のノウハウが積まれるには時間がかかる．

一方，Ruby や Python といったスクリプト言語は学習しやすい文法，メタプログラミングを支援する言語機能，および豊富なライブラリを持つため生産性が高く，使用実績も十分に積まれている[9]．これらの言語は強力なメタプログラミング機能を持つため，DSL (Domain Specific Language) の記述能力が高く，言語自体の文法に拡張を加えなくても新規キーワード相当の機能や言語機能の挙動の変更を実現できる．このため，互換性を保ったまま新しい言語の概念を導入しやすい. 科学技術計算性能に関しては，これらの言語の標準的な実装では C や Fortran といった HPC で主流の言語に比べ 100 倍程度劣っている[10]が，高速な配列操作を実現する拡張モジュールの併用[11][12]お

†1 富士通株式会社　次世代テクニカルコンピューティング開発本部
Fujitsu Limited., Next Generation Technical Computing Unit
†2 独立行政法人科学技術振興機構　戦略的創造研究推進事業
Japan Science and Technology Agency (JST),
Core Research for Evolutional Science and Technology (CREST)

ⓒ2014 Information Processing Society of Japan

よび Just-In-Time コンパイル技術[13][14]や Ahead-Of-Time コンパイル技術[15]の適用によって徐々に欠点の克服がなされており，今後 HPC における適用領域が広がっていくと予想される．

本論文では，PGAS モデルによる効率的な通信と生産性の高いプログラミング環境の両立に向けて，Ruby 上に実装可能な PGAS 言語の機能とその最適化の可能性を検討し，実装の性能評価を行い，実装上の課題や Ruby の言語処理系に加えるべき機能の方向性について論ずる．

2. PGAS 拡張の実現検討

2.1 実行モデル

実行モデルとしては，比較的シンプルな仕様である Co-Array Fortran に準じたモデルを選択した．データへのアクセスがプロセスのランク番号と各ローカル配列内の添字に基づいて行われるローカルビューを持ち，プログラムの実行形態は SPMD (Single Program Multiple Data)であり，通信は片側通信によって行われる．各プロセスには均等な長さのローカル配列が確保され，あるプロセスは他プロセスに存在するローカル配列をリモートから読み書きすることができる．

図 1 に，プロセス数が 2 の場合の実行モデルとユーザーから見えるクラス構成の関係を示す．ユーザーが意識する必要があるのは CoArray と Reference の 2 つのクラスであり，このうちユーザーが明示的に作成するのは CoArray のみである．CoArray のインスタンス作成は全プロセスで同期して一斉に行われる必要がある．他プロセスが保持する配列へは CoArray クラスの at メソッドを介してアクセスする．at メソッドはランク番号を引数に取り，該当するプロセスに所属するローカル配列の先頭のグローバルアドレスを保持した Reference オブジェクトを返却する．CoArray クラス・Reference クラスに対する添字付きの参照・代入はそれぞれ自プロセス・他プロセスの該当するデータへのアクセスとなる．

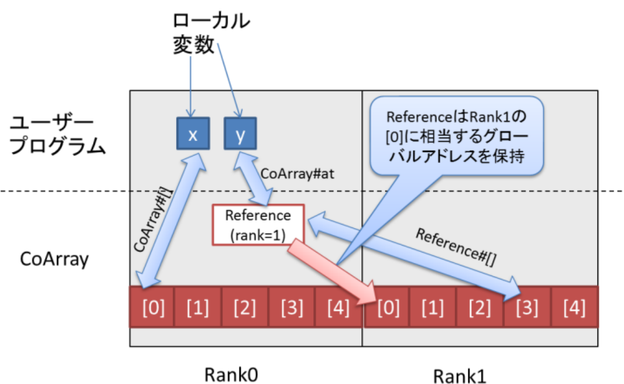

図 1 実行モデルとクラスの関係

図 2 に，具体的なソースコードの例を示す．ACP は後述する下位ライブラリをラップしたモジュールである．ACP.init, finalize, rank, procs, sync はそれぞれ初期化, 終了，自プロセスのランク番号の取得，プロセス数の取得，および全プロセスの同期を行うメソッドである．ACP のそれぞれの関数の詳しい仕様については[16]を参照されたい．LLPGAS は本論文で実装した CoArray 等のクラスを含むモジュールである．CoArray のインスタンス作成時には，要素の型を表すシンボルと，ローカルで保持する配列の長さを指定する．

```
ACP.init
myrank = ACP.rank
procs  = ACP.procs
printf "myrank = %d¥n", myrank

a = LLPGAS::CoArray.new(:int, 1)

# Assign to the local buffer
a[0] = myrank

ACP.sync

# Print remote values
procs.times {|i|
  p a.at(i)[0]
}

ACP.finalize
```

図 2 Ruby で記述した PGAS プログラムのソースコード例

2.2 制限事項

Ruby の言語仕様上，代入演算子の再定義が不可能であるため（表 1)，Co-Array Fortran における Scalar co-array に相当する文法は実現することができない．このため，本論文の仕様には盛り込んでいない．もし同様の機能を実装する場合は，属性または添字付きの参照と代入を用いるか，代入によるアクセスではなく破壊的メソッドを提供することにより代替する必要がある．

表 1 Ruby における演算子の再定義の可否（[17]より抜粋）

再定義の可否	演算子一覧
再定義可(メソッド)	| ^ & <=> == === =~ > >= < <= << >> + - * / % ** ~ +@ -@ [] []= ` ! != !~
再定義不可(制御構造)	= ?: .. ... not && and || or ::

3. 実装

3.1 下位ライブラリ

通信を行うための下位ライブラリとして，ACP ライブラリの基本層部分[16]を使用した．ACP 基本層は，全プロセスのメモリを単一のアドレス空間で管理するグローバルメモリモデルを持つ．アドレス値は 64bit 幅であり，これをグローバルアドレスと呼称する．ACP は，データ移動を最小化するデータ転送方式であるグローバルメモリ参照の機能を持つ．従来の転送方式では，データ転送の制御を行うのは送信側か受信側のどちらかのみであったが，ACP のグローバルメモリ参照は送信側でも受信側でもない第三者がデータ転送を制御できるのが特徴である．

3.2 添字付きアクセスと通信とのマッピング

PGAS 言語を実装するにあたっては，通常は分散配列への添字を用いた参照や書き込みが図 3 と図 4 のようにそれぞれ片側通信の Get と Put に対応付けられるが，このような実装では，図 5・図 6・図 7 に示したソースコードが記述された場合のコピーの制御の自由度がないという問題がある．ソースコードの評価順を考える．まず代入文の右辺値が評価され，ランク番号 2 のデータが自プロセス（ランク番号 0）に確保した一時バッファにコピーされる．次に左辺値への代入が評価され，自プロセスに確保した一時バッファからランク番号 1 へデータが転送される（図 5）．しかし，このソースコードの意味するところはランク番号 2 からランク番号 1 へのコピーであり，自プロセスに完全なコピーを保持する必要は本来なく，コピーの制御にはいくつかのバリエーションが考えられる．例えば，ACP でサポートするようなグローバルメモリ参照の利用を行えばバッファを自ノードに持つ必要はない（図 6）．また，グローバルメモリ参照がサポートされない環境であっても，転送長が長い場合は，自プロセスで持つバッファサイズを限定し，複数回の Get と Put を組み合わせることで省メモリな転送を行うことができる（図 7）．しかし，配列の添字を単に Get と Put に対応づける方法では，リモートプロセス間のコピーは必ず単一の Get と Put の組み合わせとなってしまうため，コピーの自由度が限定される．このようなことが起こるのは，左辺値へ代入する前に右辺値が評価され，常にローカルにある値が使われるためである．静的に型が決定される言語であれば事前にこのようなパターンを検出し評価順を最適化することが可能であるが，Ruby のような動的型言語では困難である．このため，右辺値の評価を左辺値への代入を行うメソッドが呼ばれた後になるよう遅延させ，そのメソッド内でコピー方法を適宜制御する必要がある．

我々は，グローバルメモリ参照へのラッパを作成し，この問題に対処した．Reference クラスの添字参照は直接値を返却せず，該当するグローバルアドレスをラップしたオブ

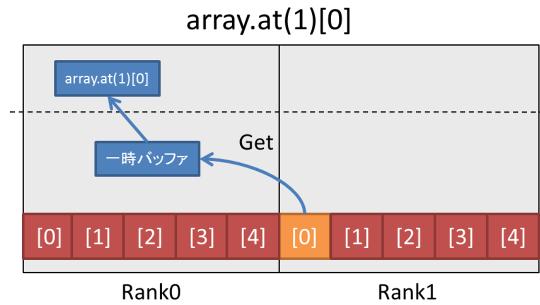

図 3 添字参照と Get の対応付け

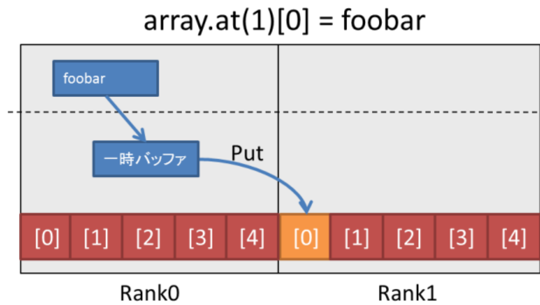

図 4 添字代入と Put の対応付け

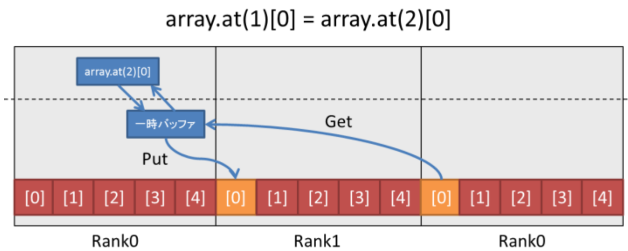

図 5 リモートプロセス間コピーがローカルを経由する例

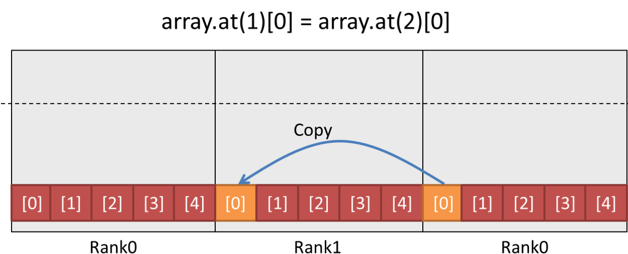

図 6 ACP によるリモートプロセス間コピー

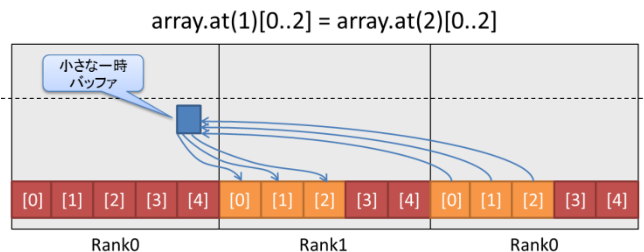

図 7 Put と Get のみ存在する環境下で，長メッセージ長のデータコピーのメモリ使用量を削減した例

ⓒ2014 Information Processing Society of Japan

```
def respond_to?(name, priv=false)
  if name == :__acp_proxy_ga ||
     name == :__acp_proxy_datatype ||
     name == : __acp_proxy_maybe_copy then
    return true
  else
    self.__acp_proxy_maybe_copy
    return @target.respond_to?(name, priv)
  end
end

def method_missing(name, *args, &block)
  self.__acp_proxy_maybe_copy
  @target.__send__(name, *args, &block)
end
```

図 8 プロクシオブジェクトの実装の抜粋

```
# array は CoArray のインスタンスであり，
# rank=1 の先頭に 42 が格納されているものとする
p array.at(1)[0] # => "42"
```

図 9 リモートの値をローカルで表示するコード

```
# 自プロセスのランク番号は 0 とする
array.at(1)[0] = array.at(2)[0]
```

図 10 リモートプロセス間コピーを行うコード

ジェクトを返却する．このオブジェクトは，プロクシオブジェクトとして動作する．オブジェクトが作成された当初は，内部で保持するインスタンス変数@target は nil であり，必要になった時に初めてリモートからのコピーが行われ，値が格納される．プロクシオブジェクトが受信するほとんどのメッセージは@target の send メソッドを用いて転送されるため，このオブジェクトを利用するプログラムからは通常の値との区別はつかない．値がコピーされるタイミングは，プロクシオブジェクトに存在しないメソッドが method_missing に転送された後，@target にメソッド呼び出しが転送される直前である．ただし，唯一，メソッドの存在確認を行う respond_to?メソッドが上書きされており，プロクシオブジェクト特有のメソッドの存在確認が行われた場合は値のコピーが行われないようになっている．respond_to?メソッドを使用することで，配列に代入されようとしている値がローカルの値なのか，ラップされたリモートの値なのかが判別することができるため，ラップされたリモートの値である場合はグローバルアドレスを取得し，適宜最適化された転送を行うことができる．図 8 に，respond_to?と method_missing の実装を示す．__acp_proxy_ という接頭辞が付いたメソッドは，プロクシオブジェクト固有のものである．__acp_proxy_maybe_copy メソッドは，@target が未初期化状態であればデータのコピーを行う．

評価の結果，リモート=ローカル間コピーが発生する例として，図 9 のソースコードを実行した際の条件判定，メッセージ，あるいはデータの流れを図 11 に示す．ソース上の"p"は Ruby の Kernel モジュールのメソッドであり，引数れた時に呼ばれる method_missing メソッドが，@target にに渡されたオブジェクトの inspect メソッドを呼び出し，その結果と改行を連結した結果を標準出力に表示するものである．プロクシオブジェクトは inspect メソッドが存在しない BasicObject を継承しているため，inspect メソッドは method_missing メソッドに転送される．method_missing では@target が取得済みかを確かめ，取得されていない場合はリモートよりデータの取得を行う．取得が完了次第，inspect メソッドが@target に転送され，文字列 "42" が返却される．

次に，リモート間コピーが発生する例として，図 10 のソースコードを実行した時の流れを図 12 に示す．ランク番号 1 の Reference に対する添字付き代入は， []=メソッドの呼び出しとなる．右辺値が評価された後，[]=メソッドの引数には key として 0 が，value としてプロクシオブジェクトが設定される．[]=メソッド内では value がプロクシオブジェクトであるかどうかの判定が行われ，この結果が真であればグローバルアドレスの取り出しが行われ，ランク番号 2 番からランク番号 1 番へのリモートプロセス間コピーが呼び出される．

このようにして，プロクシオブジェクトの導入によりデータの評価順を遅延させることで，コピーの自由度を増すことができる．

4. 性能評価

4.1 評価環境

下位ライブラリに ACP の UDP 版を使用し，本クラスライブラリの性能評価を行った．評価環境を表 2 に示す．評価対象はソフトウェアレイテンシの影響が大きいメッセージ長である 4 バイトのリモートツーローカル，ローカルツーリモート，およびリモートプロセス間コピーである．比較のため，Ruby で記述したテストプログラムと同様のものを C 言語でも作成した．C 言語版のコンパイルオプションは-O3 であった．コピーの試行は各テストで 1000 回行った．

4.2 評価結果

評価結果を表 3 および表 4 に示す．各項目の数値の単位はマイクロ秒である．表 3 は全プロセスをシングルノードで実行した時の，また，表 4 は各プロセスに 1 ノードを割り当てて実行した際の性能を示している．プロセス数はどちらも 3 プロセスである．ローカルツーリモートのコピー方向はランク 0 からランク 1，リモートツーローカルのコピ

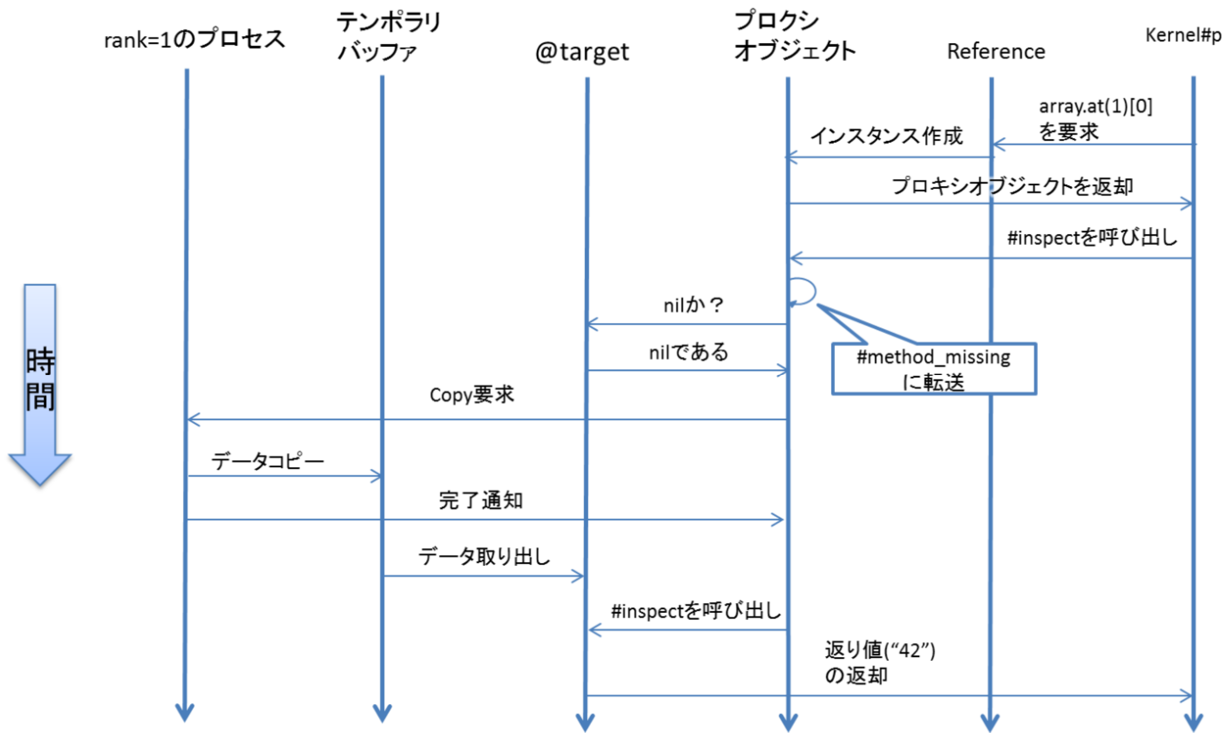

図 11 リモートの値を表示するまでの流れ

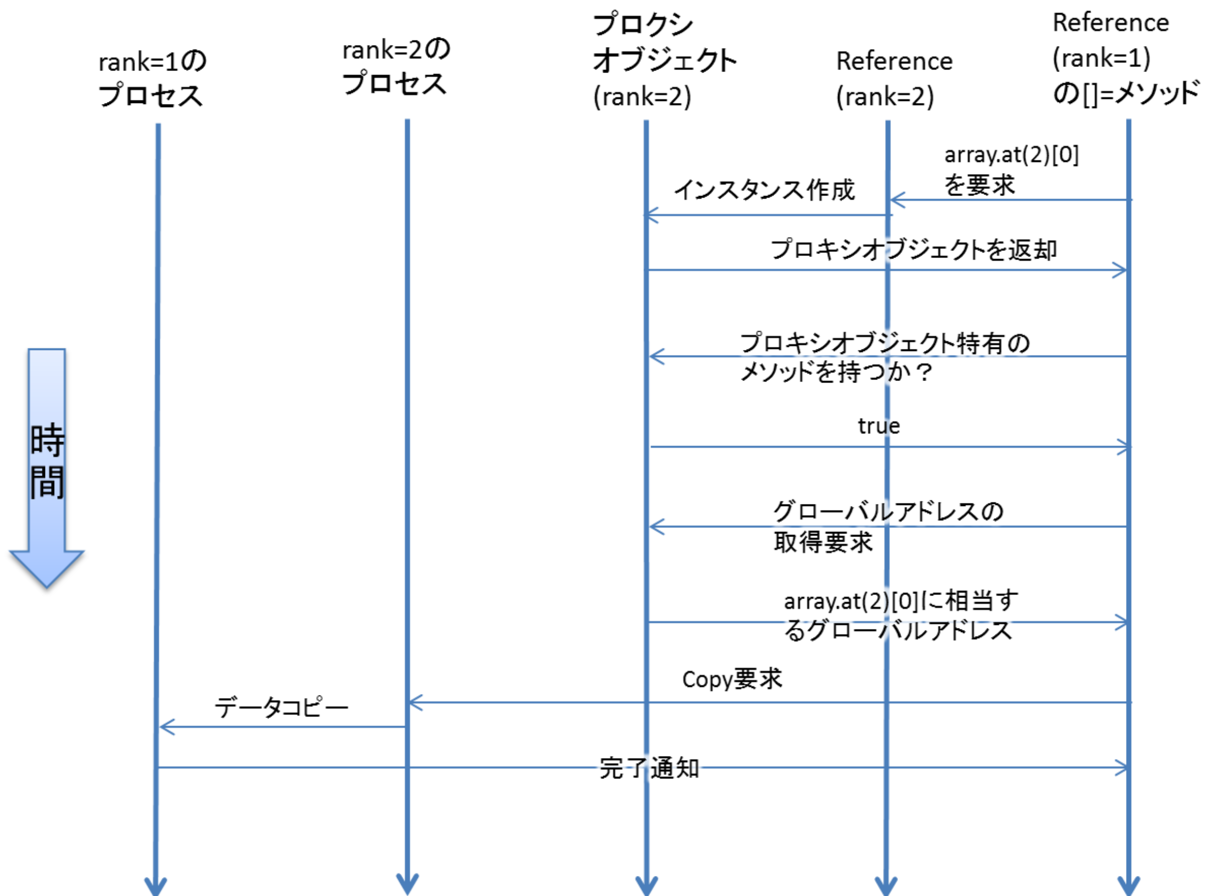

図 12 リモートプロセス間コピーの流れ

ⓒ2014 Information Processing Society of Japan

表 2 評価環境

CPU	Intel Xeon E5520 (4 cores, 2.27 GHz), 2 sockets
メモリ	48GB, DDR3 1066MHz
ネットワーク	Gigabit Ethernet (125 Mbyte/sec)
OS	Linux version 2.6.32
Ruby	1.9.3p194 (2012-04-20 revision 35410)
コンパイラ	GCC 4.1.2

表 3 シングルノード実行時の性能（単位：マイクロ秒）

	C	Ruby
Local to remote	12.7	22.1
Remote to local	13.2	27.9
Remote to remote	35.2	50.1

表 4 複数ノード実行時の性能（単位：マイクロ秒）

	C	Ruby
Local to remote	50.9	52.5
Remote to local	50.9	52.6
Remote to remote	101.8	117.0

一方向はランク 1 からランク 0，リモートツーリモートのコピー方向はランク 2 からランク 1 である．

4.3 考察

表 3 はシングルノード実行であるため，インターコネクトで発生するコストを含んでいない．Ruby 版は C 版と比較してどのオペレーションも 10 から 15 マイクロ秒程度のコストが掛かっており，これは言語処理系自体のオーバーヘッドと考えられる．一方，表 4 の複数ノード実行では，リモートツーローカルおよびローカルツーリモート時の実装間の性能差は 2 マイクロ秒程度に縮小しているのに対し，リモートツーリモート時の性能差は縮小が見られない．リモートツーローカルおよびローカルツーリモート時の性能差は，I/O 待ちにより言語処理系由来の遅延が隠蔽されていることによると考えられる．リモートツーリモートでは，プロクシオブジェクトにより上書きされた独自実装の respond_to?に対する呼び出しが発生することが他のオペレーションに比べ異なっているが，この呼び出しはシングルノード実行でも発生するため，関連は不明瞭であり，さらなる詳細なコスト分析が必要である．

5. 関連研究

PyGAS[18]は Python による GASNet のラッパであり，プロクシオブジェクトを利用した手法を実装している．しかし，プロクシオブジェクトの利用目的は，GASNet のアクティブメッセージ機構を利用してリモートに存在するオブジェクトに対するメソッド呼び出しを行うことであり，本論文の目的とは異なる．

Global Arrays[19]の Python バインディングは NumPy 互換のモジュールが実装されており，高いユーザビリティを備えている．NumPy 同様，配列の添字はコピーではなくスライスを返却する点が本論文と類似している．添字経由での参照は集団的に行う必要があり，片側通信を行うには配列に対して get メソッドを呼びコピーを明示的に行う必要がある点が異なる．代入は Global Arrays の Scatter に対応付けられている．

[20]は X10 と意味論に互換性のある DSL を Ruby をベースに構築し，ソースツーソーツ変換器によって X10 のソースコードを出力することで，Ruby による高い生産性と X10 の性能の両立を目指したものである．本提案における DSL は，最終的には純粋な X10 のプログラムになるため，実行時に Ruby 仮想マシンを要求する動的評価が行えない．Ruby ユーザーが言語内 DSL を構築する時，動的評価はしばしば使われるテクニックであり[21]，動的評価の欠如は生産性に影響がある．

6. 今後の課題

今回の実装では配列に対する操作の一貫性が問題となる可能性がある．例えば，リモートの値を参照するプロクシオブジェクトが生成された後に該当のアドレスを書き換えることが可能であり，その後にプロクシオブジェクトに対するメソッドが呼び出されると，ユーザーは書き換え前の値を取得することを期待していたにも関わらず，書き換え後の値が取得されることになり，一貫性が失われる．プロクシオブジェクトが指し示すアドレスに対し自プロセスが書き込みを行うことを検知する，あるいはそのアドレスが属するランクに対する何らかの書き込みを自プロセスが検知することで，書き込みの直前に値を取得し一貫性を保証することもできるが，コストが高い．ユーザー責任で一貫性をあえて保証しない，あるいは言語処理系の側でリモートツーリモートコピーが本当に必要な時を分析し，必要でない場合にはプロクシオブジェクトの値評価を遅延させないようにする等の対策が考えられる．

性能評価の結果，ある程度レイテンシの隠蔽が認められたが，今回の評価環境は Ethernet 上の UDP 実装であり，Tofu インターコネクト[22]のようなレイテンシの小さいデバイス上では言語処理系由来のレイテンシの隠蔽が不十分になり，言語処理系の高速化が必要になる可能性がある．

7. まとめ

メタプログラミング機能により優れた DSL 作成能力を持つプログラミング言語である Ruby 上で，リモートプロセス間コピー操作をサポートする PGAS 言語を，ベースとなる言語処理系の仕様やランタイム実装に変更を加えること

ⓒ2014 Information Processing Society of Japan

なく実現した．また，実装のオーバーヘッドを測定した．ノード間にまたがるローカルツーリモートおよびリモートツーローカル間コピーでは Ruby によるオーバーヘッドが隠蔽される傾向が見られたが，リモートツーリモートでは隠蔽されなかった．隠蔽されなかったケースについては詳細なコスト分析が必要である．

参考文献

1) ACE Project, http://ace-project.kyushu-u.ac.jp/main/jp/index.html
2) 住元真司, 安島雄一郎, 佐賀一繁, 野瀬貴史, 三浦健一, 南里豪志: エクサスケール通信向け ACP スタックの設計思想, 情報処理学会研究会報告 2014-HPC-143-8(2014).
3) Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. ACM SIGPLAN Fortran Forum Archive, 17:1–31 (1998)
4) The UPC Consortium: UPC Language Specifications Version 1.3 (2013)
5) P. Hilfinger, et al.: Titanium Language Reference Manual, U.C. Berkeley Tech Report, UCB/EECS-2005-15 (2005)
6) Cray Inc: Chapel Language Specification Version 0.95 (2014)
7) Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove: X10 Language Specification Version 2.4 (2014)
8) Eric Allen, et al: The Fortress Language Specification Version 1.0 (2008)
9) TIOBE Index for August 2014, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
10) 野瀬 貴史, 泊 久信, 平木 敬: 多種言語処理系性能の評価に適したベンチマークプログラム, 情報処理学会研究会報告 2011-HPC-130-2 (2011).
11) NumPy, http://www.numpy.org/
12) Numerical Ruby NArray, http://masa16.github.io/narray/index.ja.html
13) Armin Rigo and Samuele Pedroni.: PyPy's approach to virtual machine construction, OOPSLA 2006, pp. 944-953 (2006)
14) Topaz, http://topaz.readthedocs.org/en/latest/
15) 中村晃一, 野瀬貴史, 稲葉真理, 平木敬: HPC Ruby: 静的解析に基づく Ruby の高度最適化コンパイラ, 情報処理学会研究会報告 2011-HPC-130-63 (2011).
16) 安島雄一郎, 佐賀一繁, 野瀬貴史, 三浦健一, 住元真司: ACP 基本層の設計思想とインタフェース, 情報処理学会研究会報告 2014-HPC-143-9 (2014)
17) Ruby 1.9.3 リファレンスマニュアル > 演算子式, http://docs.ruby-lang.org/ja/1.9.3/doc/spec=2foperator.html
18) Michael Driscoll, et al: PyGAS: A Partitioned Global Address Space Extension for Python, PGAS2012 (2012)
19) Global Arrays Toolkit: http://hpc.pnl.gov/globalarrays/
20) Tetsu Soh: Design and Implementation of a DSL based on Ruby for Parallel Programming, Master's thesis, The University of Tokyo (2011)
21) Paolo Perrotta: メタプログラミング Ruby, アスキー・メディアワークス (2010)
22) Yuichiro Ajima, et al: Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers, Computer, 42(11), pp. 36-40 (2009)

ⓒ2014 Information Processing Society of Japan