Vol.2015-HPC-148 No.26
2015/3/3

ACP の分散連想配列インタフェース

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

本論文では Advanced Communication for Exa (ACE)プロジェクトで開発している Advanced Communication Primitives (ACP)ライブラリへの分散連想配列インタフェースの追加を提案する．提案する分散連想配列は複数プロセスにハッシュテーブル，データを分散して 1 プロセスあたりのメモリ消費量を抑えつつ巨大な配列を実現する．提案する分散連想配列インタフェースはサーバークライアント型のインメモリ分散 KVS に比べて，ACP 基本層のデータ移動関数で実装されるため実装のポータビリティが高く，利用面では通信ライブラリであるためプログラマビリティが高いと言える．また UDP 版 ACP 基本層上で初期評価を行い，値の挿入，検索ともに 1 ミリ秒程度で実行できることを確認した．ACP 基本層を使用していることによる HPC インターコネクト上での動作の容易さを考慮すると，本提案の分散連想配列インタフェースは並列計算機上でのビッグデータ処理開発基盤として有用な技術であると言える．

1. はじめに

ポストペタスケール時代の High Performance Computing (HPC)では，メニーコア・プロセッサと広帯域の三次元積層メモリがキーテクノロジとして期待されている．しかしながら広帯域の三次元積層メモリは容量が既存のメモリモジュールと同程度の水準に留まるため，ポストペタスケール時代のシステムソフトウェアにおいてはメモリ消費量の削減が重要となる．

Advanced Communication for Exa (ACE) プロジェクト [1] ではプロセスあたりのメモリ消費量を抑制しつつ，低遅延通信を実現する通信ソフトウェア技術の創出に取り組んでいる．従来の通信ライブラリでは，処理コストの高いメモリの解放，再割当てを回避するため，動的に割り当てた通信バッファを解放せずに使い続けるが，メモリ消費量を抑制するには不要な通信バッファの解放が必要である．我々は ACE プロジェクトの目標を実現する中核技術として，利用者がメモリ消費量を意識したプログラミングが可能であるように，明示的にメモリを使用するインタフェースを備える低レベル通信ライブラリ Advanced Communication Primitives (ACP)を開発している．

ACP は低レベル通信を抽象化し，インターコネクトデバイスの違いを吸収する基本層 [2] と，基本層の上にポータブルに実装される中間層で構成される．中間層にはコミュニケーションライブラリ[3]およびデータライブラリ[4]という 2 つのサブライブラリが含まれる．コミュニケーションライブラリは既存のメッセージパッシングで記述されるアプリケーションの移行を想定したインタフェースであり，明示的な通信バッファ割当・解放が可能なチャネルインタフェースを中核とする．データライブラリは PGAS 言語ランタイム，ビッグデータ処理などの新しい用途を想定したインタフェースであり，大域的なデータ配置を最適化するための分散動的データ構造インタフェースを中核とする．

本論文では ACP のビッグデータ処理への適用可能性を広げるために，データライブラリへの分散連想配列インタフェースの追加を提案し，実装評価する．以降では，2 章で ACP について基本層とデータライブラリを中心に紹介し，3 章で分散連想配列インタフェースの設計思想と実装構造，主要なインタフェース仕様を紹介する．4 章で初期評価結果，5 章で今後の課題について記述し，最後に 6 章でまとめる．

2. Advanced Communication Primitives

2.1 ACP 基本層

ACP 基本層はポストペタスケール時代に相応しい基本通信機能を定義する．従来の通信ライブラリでは両側通信機能または片側通信機能を提供していた．これに対して ACP 基本層では転送元，宛先プロセスとも通信処理に関与しない第三者通信機能を提供する．具体的には acp_copy 関数によって任意のプロセスのメモリから任意のプロセスのメモリにデータを転送する．ACP 基本層では第三者通信機能を，片側通信機能であるリモートメモリ参照(Remote Memory Access, RMA)よりも制約の小さい通信機能と位置付け，グローバルメモリ参照(Global Memory Access , GMA)と呼ぶ．

RMA 通信，GMA 通信によるデータ転送の例を図 1，図 2 に示す．いずれの例でも，図中 P1 で示されるプロセス 1 で実行しているプログラムが，P2 で示されるプロセス 2 の変数を，Pn で示されるプロセス n の変数に代入している．RMA の場合，まず右辺を評価する際に P2 の変数の値を P1 の一時領域に Get 通信で転送する．そして左辺への代入では，P1 の一時領域に格納されている値を Put 通信で Pn に転送する．この実行モデルは P1 の一時領域をレジスタに見立てると，階層記憶を持つ CPU の実行モデルに近いものと解釈できる．右辺が演算を伴う式であれば，P1 が式の評価を行う際に自明な局所性があるので，RMA による階層記憶モデルの疑似は処理に適した動作と言える．

†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機能
Japan Science and Technology Agency (JST),
Core Research for Evolutional Science and Technology (CREST)

ⓒ 2015 Information Processing Society of Japan

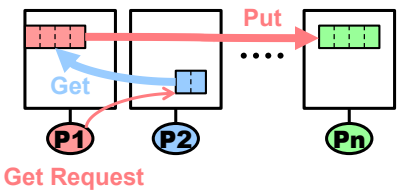

図 1　リモートメモリ参照通信によるデータ転送例
Figure 1　An example of data transfer via Remote Memory Access communication

CPU ではレジスタのデータ入出力コストが低いため，レジスタをバイパスしてメモリからメモリへ直接転送しても性能面の利得は小さい．RISC アーキテクチャに至っては，メモリ参照をレジスタへの読み出し，レジスタからの書き込みのみに限定している．しかし，分散メモリ型並列計算機において本来 1 回で済むプロセス間データ転送を 2 回行うことは冗長である．GMA は任意のプロセス間のデータ転送が可能であるので，右辺に演算が無い場合はプロセス間データ転送を 1 回で済ませることができる．通信時間が半分になって性能が上がるだけでなく，メモリ使用量，消費電力のコスト面においても節約になる．

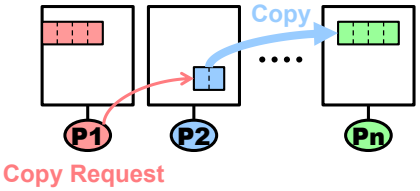

図 2　グローバルメモリ参照モデルによるデータ転送例
Figure 2　An example of data transfer via Global Memory Access

GMA と仮想共有メモリ(Virtual Shared Memory, VSM)との違いは, GMA は専用の仮想メモリ空間を扱う点に対し，VSM は CPU から直接参照可能な仮想メモリ空間を扱う点である．GMA では複数プロセスに跨って連続したメモリは定義できず，CPU が直接参照可能なのは該当プロセスで登録されたメモリすなわちローカルメモリだけであるという制約がある．GMA の扱うアドレス空間は Partitioned Global Address Space (PGAS)に分類される．また，ACP 基本層の GMA では明示的に登録されたメモリのみ参照可能であり，CPU からの直接参照を行うためのローカルアドレスと，GMA に使用するためのグローバルアドレスは明確に異なるものとして扱う．

ACP 基本層はデータ転送元で演算を行い，結果を転送する GMA 関数も用意する．演算を伴う GMA 関数は一種の並列オブジェクトモデルを実現していると言える．一般に並列オブジェクトモデルやアクティブメッセージモデルでは処理をプログラム可能であるが，ACP 基本層ではインターコネクトデバイスに実装可能な演算を想定して演算の種類を限定している．これは現代の CPU および OS では，リモートプロセスにおいてプログラムを実行するためのコンテキストスイッチのコストが高いためである．また演算の種類は，細粒度並列処理が可能な並列データ構造に関する既存の研究成果の活用を想定し，CPU のマルチスレッド処理で使用されるアトミック演算命令を参考にした取り揃えになっている．

2.2　ACP データライブラリ

ACP データライブラリは ACP 基本層の上に構築される中間層のライブラリであり，分散データ構造インタフェースを中核とする．分散動的データ構造インタフェースは，データ構造を操作するアルゴリズム自体を変えずに，配置指定の変更だけでグローバルなデータ配置の最適化を可能にすることを目標とする．そのため，データ生成時に配置を明示的に制御することで，データ構造を複数プロセスに分散させる．データの生成，操作，破棄は非同期に，配置するプロセスと同期せずに行う．さらに，データがローカルに配置されている場合，通常のローカルなデータ構造の操作と比較して遜色のない性能を目指す．

ACP データライブラリは，分散データ構造インタフェースの共通基盤としてグローバルメモリアロケータを実装する．グローバルメモリアロケータでは acp_malloc 関数で任意のプロセスのメモリを割当て，acp_free 関数で解放する．散データ構造インタフェースは GMA 関数とグローバルメモリアロケータ関数を組合せて実装される．

ACP データライブラリの分散データ構造インタフェースは，上位層の言語処理系などでラップされることを想定して，データ構造間でインタフェースの直交性が高い仕様を目標とする．データ構造の型は C++言語の標準テンプレートライブラリ(Standard Template Library, STL) [5]を参考にしており，これまでにベクタおよびリストを導入した[4].

ベクタは可変長一次元配列であり，任意のプロセスに生成し，任意のプロセスから要素の追加が可能である．Pn が P2 にベクタを生成する例を図 3 に, P2 のベクタに P1 が要素を追加する例を図 4 に示す．ベクタは同一プロセス内で連続した領域として生成され，複数プロセスに跨ったデータ配置はできない．

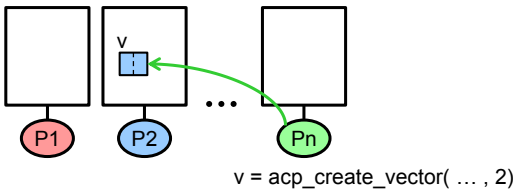

図 3　ベクタ生成の例
Figure 3　An example of creation of a vector

ⓒ 2015 Information Processing Society of Japan

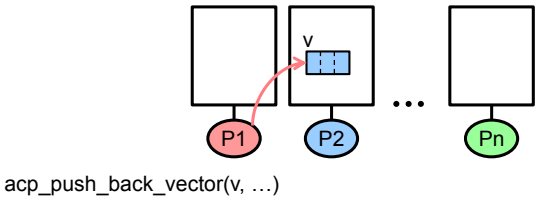

図 4　ベクタへの要素追加の例
Figure 4　An example of adding an element to a vector

リストは双方向リンクのリストであり，任意のプロセスに制御情報を生成し，追加する要素も任意のプロセスに配置することができる．P1 が P2 に空リストを生成する例を図 5 に，P1 が P2 の空リストに P1 に配置する要素を追加する例を図 6 に，P1 が P2 のリストに Pn に配置する要素を追加する例を図 7 に示す．

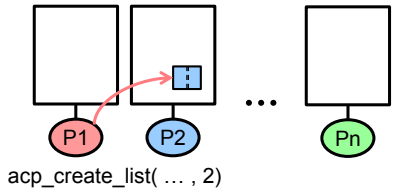

図 5　空リスト生成の例
Figure 5　An example of creation of an empty list

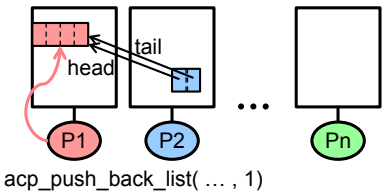

図 6　空リストへの要素追加の例
Figure 6　An example of adding an element to an empty list

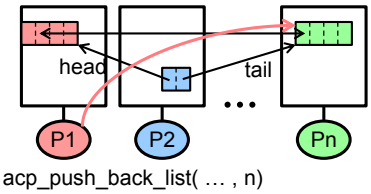

図 7　リストへの要素追加の例
Figure 7　An example of adding an element to an list

3. 分散連想配列インタフェースの提案

本論文では分散データ構造インタフェースへの分散連想配列の追加を提案する．連想配列は多くのプログラミング言語で採用されているデータ構造であり，幅広い応用範囲が期待できる．インメモリ分散連想配列として想定される競合はサーバークライアント型のインメモリ分散 Key Value Store (KVS)である．ACP データライブラリの分散連想配列は ACP 基本層のデータ移動関数で実装されるため実装のポータビリティが高く，利用面では通信ライブラリであるためプログラマビリティが高い点が利点と言える．

ACP では STL の map および unordered_map を参考として，連想配列をマップ型のデータ構造と名付ける．STL の map は平衡二分木による実装を想定しており，要素がソートされて格納されていることを前提としたインタフェースになっている．一方 unordered_map はチェインハッシュによる実装が想定されており，その名の通り要素は順不同に格納される．平衡二分木の操作における計算量は O(*logn*)でハッシュの O(*1*)と比べるとメモリ参照回数が多い．ACP のマップは，要素がソートされていることの利点よりもメモリ参照回数の少なさを重視して，ハッシュによる実装を想定したインタフェースとする．ハッシュによる連想配列実装はインメモリ分散 KVS サーバでも採用されており，分散メモリをスケールする NoSQL データベースとして利用する際には一般的な方法である．

なお，ACP ではノードの動的な追加，削除を想定していないため，コンシステントハッシュは採用しない．ハッシュテーブルを複数ノードに分散し，要素を追加する際は，ハッシュテーブルの断片が配置されたプロセスに要素を配置する．これまでに提案してきたベクタ，リストでは要素の配置先プロセスを利用者が明示的に指定したが，マップではキーの値によって暗黙に要素の配置先プロセスが決定する点が異なる．ただし，配置先プロセスの候補はマップを生成する際に指定する．

3.1 実装構造

マップではハッシュテーブルを複数プロセスに分散する．プロセスあたりのスロット数はインタフェースで明示的に指定する．マップ全体でのスロット数はプロセス数×プロセスあたりのランク数になる．ハッシュ関数は生成多項式 0x42F0E1EBA9EA3693 の 64 ビット CRC を 16 ビット右シフトしたものを使用する．CRC は計算コストが高いが，プロセス間通信時間に比べると計算時間は短い．

P1 が n 個のプロセスに跨る空マップを生成する例を図 8 に示す．各プロセスに同じスロット数のハッシュテーブル断片を生成し，ハッシュテーブル断片へのポインタ配列を P2 に配置する．ハッシュテーブルの各スロットには空のリストが格納される．

ⓒ 2015 Information Processing Society of Japan

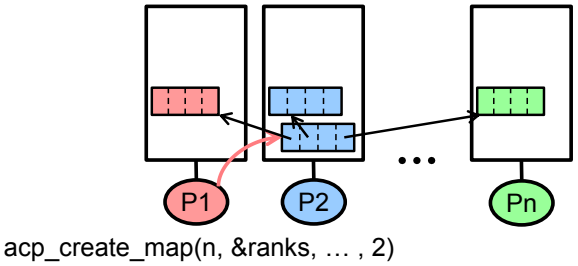

図 8　空マップ生成の例
Figure 8　An example of creation of an empty map

P1 が空マップに要素を追加する例を図 9 に示す．まず P1 ではキーのハッシュ値を計算する．次に P2 に配置されているハッシュテーブル断片へのポインタ配列を参照し，要素を追加すべきスロットのグローバルアドレスを取得する．最後に，該当スロットのリストに要素を追加する．

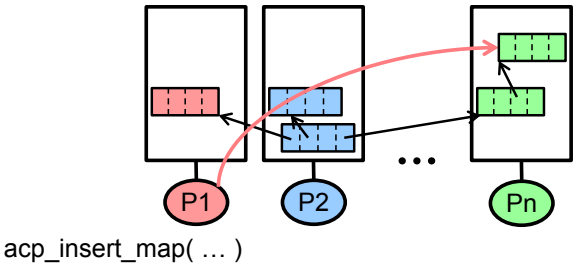

図 9　空マップへの要素追加の例
Figure 9　An example of adding an element to a map

3.2 マップ型の主要関数

マップ型の主要関数仕様を表 1 に示す．

マップ生成関数 acp_create_map ではプロセス数，プロセスランク配列，スロット数，プロセスランクを指定する．スロット数を明示的に指定する理由は，現在のマップ型関数はリハッシュ機能を搭載していないため，利用者側でリハッシュを行うなどのスロット数制御を行うためである．プロセス数にはハッシュテーブルを分散するプロセスの数，プロセスランク配列にはハッシュテーブルを分散するプロセスのランク番号を指定する．プロセスランクはハッシュテーブル断片へのポインタ配列を配置するプロセスのランク番号を指定する．

マップ要素挿入関数 acp_insert_map ではマップ，キー，値を指定する．ここでキーおよび値は可変長であり，キーと値の格納先ポインタとサイズをそれぞれ指定する．マップでは要素の挿入位置をキーのハッシュ値で自動的に決定するため，要素挿入関数はイテレータを引数に取らない．

マップ要素検索関数 acp_find_map ではマップ，キーを指定する．返り値としてキーを発見したかどうかを表す真偽値と，キーを発見した場合の要素を参照するイテレータを返す．

3.3 リスト関数仕様の変更

現在のマップ型関数実装では，内部的にリスト型関数を使用する．このために以前に提案したリスト型関数の仕様を変更した．更新したリスト型の主要関数仕様を表 2 に示す．

まず，要素を固定長から可変長に変更した．この変更により，リスト生成関数 acp_create_list からサイズの指定がなくなった．その代り，acp_push_back_list，acp_insert_list などの要素を追加する関数にサイズの指定が追加された．

次に，追加する値を指すポインタをグローバルアドレスからローカルアドレスに変更した．値を単純に GMA でコピーするだけであればグローバルアドレスで十分だが，キーと値を組にした要素を作ってリストに登録したり，キーからハッシュ値を計算したりするには，ローカルアドレスの方が適するためである．

さらに，イテレータ内部にリストを格納し，イテレータ指定時にリストを引数に取る必要を無くした．これはマップ型の内部で使用するためというよりも，利用者の利便性を向上するための変更である．

表 1　マップ型主要関数
Table 1　Map data type major functions

名称	定義
マップ生成	acp_map_t **acp_create_map**(int *num_ranks*, const int* *ranks*, int *num_slots*, int *rank*);
マップ全要素消去	void **acp_clear_map**(acp_map_t *map*);
マップ破棄	void **acp_destroy_map**(acp_map_t *map*);
マップ要素挿入	acp_map_it_t **acp_insert_map**(acp_map_t *map*, void* *key*, size_t *size_key*, void* *value*, size_t *size_value*);
マップ要素検索	acp_map_ib_t **acp_find_map**(acp_map_t *map*, void* *key*, size_t *size_key*);

ⓒ 2015 Information Processing Society of Japan

表 2　リスト型主要関数(更新)
Table 2　List data type major functions (updated)

名称	定義
リスト生成	acp_list_t **acp_create_list**(int *rank*);
リスト末尾要素追加	void **acp_push_back_list**(acp_list_t *list*, const void* *ptr*, size_t *size*, int *rank*);
リスト要素挿入	acp_list_it_t **acp_insert_list**(acp_list_it_t *it*, const void* *ptr*, size_t *size*, int *rank*);
リスト要素削除	acp_list_it_t **acp_erase_list**(acp_list_it_t *it*);

4. 初期評価

4.1 評価環境

初期評価は PC クラスタで実施した．表 3 に評価環境の詳細を示す．インターコネクトは Gigabit Ethernet であり，UDP [6] 版の ACP 基本層を使用して評価した．テストプログラムは 4 ノードを使用し，ノードあたり 1 プロセス，合計 4 プロセスで実行した．

表 3　評価環境
Table 3　Evaluation environment1: PC Cluster

Node	Fujitsu PRIMERGY RX200 S5
CPU	Intel Xeon E5520 (4 cores, 2.27 GHz), 2 sockets
Memory	48GB, DDR3 1066MHz
Network	Gigabit Ethernet (125 Mbyte/sec)
OS	Linux version 2.6.32
RTE	ACP 1.0 / acprun

評価実験ではマップ要素挿入関数 acp_insert_map およびマップ要素検索関数 acp_find_map の実行時間を計測した．また比較対象としてグローバルメモリ割当関数 acp_malloc および解放関数 acp_free の実行時間も計測した．

acp_insert_map の評価ではキー，値ともに 32 バイトの要素を 1024 個挿入する時間を計測した．キーの内容は乱数で生成した．ハッシュテーブルを配置するプロセス数は 1 と 4 とし，1 プロセスの場合はハッシュテーブルを自プロセスに配置する場合(local)と他プロセスに配置する場合(remote)のケースを計測した．4 プロセスの場合は 1 プロセスが自プロセス，3 プロセスが他プロセスとした．スロット数は合計で 128 とし，1 プロセスの場合はプロセスあたり 128，4 プロセスの場合はプロセスあたり 32 を指定した．

acp_find_map の評価ではキー，値ともに 32 バイトの要素を 1024 個格納したマップに対し，32 バイトのキーによる検索を 1024 回実行する時間を計測した．検索するキーはランダムで生成するが，50%の確率でマップに格納されているキーを使用した．

acp_malloc については 1024 回連続で呼び出し，平均実行時間を求めた．割当サイズは 1 バイト以上 32768 バイト以下の乱数を使用し，毎回変更した．割当先のプロセス番号は 1024 回とも同じプロセスを指定した．ただし，インターコネクトを介した通信が必要なプロセス(remote)と自プロセス(local)の 2 通りの評価を行った．

割当てた 1024 個のグローバルメモリは acp_free で解放し，その平均実行時間も求めた．解放する順番は割当て順ではなく，ランダムに定めた．

4.2 評価結果

図 10 にマップ型関数とグローバルメモリアロケータ関数の平均実行時間評価結果を示す．ハッシュテーブルが自プロセスに配置されている場合，要素挿入，検索とも 56 μ 秒で実行された．これに対してハッシュテーブルが他プロセスに配置されている場合は要素の挿入に 1031 μ 秒，検索に 776 μ 秒かかった．また， 1/4 の確率で自プロセスに要素が配置される 4 プロセスの場合，挿入は 811 μ 秒，検索は 618 μ 秒に短縮された．

acp_malloc の平均実行時間は local で 13 μ 秒，remote で 413 μ 秒であった．これに対して acp_free の平均実行時間は local で 137 μ 秒，remote で 4771 μ 秒となった．

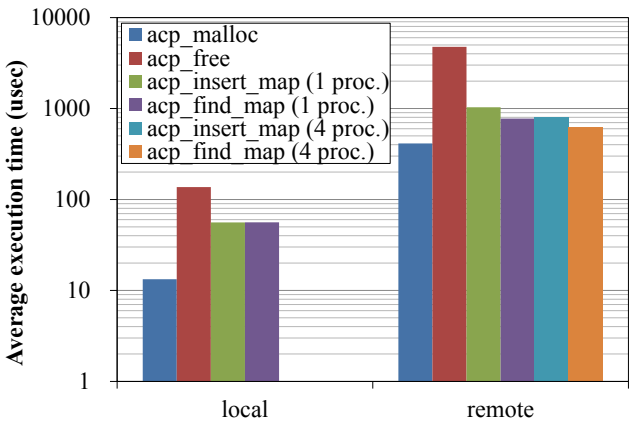

図 10　連想配列型関数の平均実行時間評価結果
Figure 10　Evaluation results of average execution time of global memory allocator functions

4.3 考察

local の場合でも関数実行時間は 2 桁 μ 秒以上に達している．これは現状の UDP 版 ACP 基本層の実装では，例えば acp_copy の転送元，宛先ともに自プロセスであった場合でも ACP の通信スレッドがデータ転送を処理するためである．メインスレッドと通信スレッド間でのコマンド指示と応答が 1 往復で 5 μ 秒程度のオーバーヘッドを引き起こしており，改善する余地がある．

ⓒ 2015 Information Processing Society of Japan

remote の場合のマップへの要素の挿入，検索の関数実行時間は，ともに 1 ミリ秒以下であった．これはサーバークライアント型のインメモリ分散 KVS とほぼ同等である[7]. ACP 基本層を使用しているので HPC インターコネクト上での動作が容易であることを考慮すると，本提案の分散連想配列インタフェースは並列計算機上でのビッグデータ処理開発基盤として有用な技術であると言える．

acp_malloc 関数より acp_free 関数の実行時間が長いのは，現在のグローバルメモリアロケータがソートされた片方向リストで空き領域を管理する Kernighan and Richie (K & R) のアルゴリズムを採用しているためである．K & R 方式ではメモリを解放する際，空き領域のリストに挿入する位置を調べるために O(n)の計算量がかかる．

acp_malloc および acp_free 関数の平均実行時間は文献[4]でのプロトタイプ版 ACP 基本層を使用した評価と比べて半分程度に短縮されている．これは，今回の評価ではランタイム環境が整備された ACP v1.0 の基本層を使用しているためと考えられる．

5. 今後の課題

本提案の分散連想配列インタフェースは排他制御を行わないので，並列データ構造として利用するためには外部で排他制御を行わなければならない．しかし外部で排他制御を行うと排他制御の粒度が大きいため性能がスケールしない．将来的にはマップ型関数の中で細粒度の排他制御を導入する必要がある．

local 処理でも関数実行時間が長いのは ACP 基本層実装の最適化が不十分であることが原因である．ACP データライブラリの実用性を高めるためには，ACP 基本層の性能最適化を進めなくてはならない．

6. まとめ

本論文では ACE プロジェクトで開発している ACP ライブラリへの分散連想配列インタフェース，マップの追加を提案した．マップは複数プロセスにハッシュテーブル，データを分散して 1 プロセスあたりのメモリ消費量を抑えつつ巨大な連想配列を実現する．マップはサーバークライアント型の分散 KVS サービスに比べて，ACP 基本層のデータ移動関数で実装されるため実装のポータビリティが高く，利用面では通信ライブラリであるためプログラマビリティが高いと言える．また UDP 版 ACP 基本層上で初期評価を行い，値の挿入，検索ともに 1 ミリ秒程度で実行できることを確認した. ACP 基本層を使用していることによる HPC インターコネクト上での動作の容易さを考慮すると，本提案のマップデータ型は並列計算機上でのビッグデータ処理開発基盤として有用な技術であると言える．

参考文献

1) ACE Project, http://ace-project.kyushu-u.ac.jp/index.html
2) 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)
3) Advanced Communication Primitives (ACP) Library version 1.0 Tutorial, http://ace-project.kyushu-u.ac.jp/main/en/07_document/acp-1.0-tutorial-2014nov.pdf
4) 安島雄一郎, 野瀬貴史, 佐賀一繁, 志田直之, 住元真司: ACP の分散動的データ構造インタフェース, 情報処理学会研究会報告 2014-HPC-146-18 (2014)
5) Containers library, http://en.cppreference.com/w/cpp/container
6) Jonathan B. Postel (editor): User Datagram Protocol, RFC 768 (1980)
7) Apache Cassandra NoSQL Performance Benchmarks, http://planetcassandra.org/nosql-performance-benchmarks/#EndPoint

ⓒ 2015 Information Processing Society of Japan