Vol.2014-HPC-146 No.18
2014/10/3

ACP の分散動的データ構造インタフェース

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)ライブラリの分散動的データ構造インタフェースの設計思想，構成方法，主要なインタフェース仕様，初期評価結果を紹介した．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．またデータの生成，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．これらを実現するために，グローバルメモリアロケータを導入する．本論文では分散動的データ構造インタフェースのサポートするデータ型のうち，ベクタ型とリスト型について主要なインタフェースの仕様を示す．また初期評価として，Tofu インターコネクト版および UDP 版の ACP 基本層上でグローバルメモリアロケータ関数の実行時間を評価する．acp_malloc および acp_free 関数の平均実行時間は割当先 local の場合，Tofu 版で 9 及び 19μ秒，UDP 版で 16 および 26μ秒，割当先 remote の場合，Tofu 版で 34 及び 68μ秒，UDP 版で 2.0 および 7.9 ミリ秒となった．分散動的データ構造インタフェースは実用上，Tofu インターコネクトのような低遅延のインターコネクトの使用が必須であるといえる．

1. はじめに

エクサスケールに向けたポストペタスケール時代では，メニーコア・プロセッサと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．三次元積層メモリには新しいインタフェースで広帯域を実現するものと，既存のインタフェースとパッケージのまま大容量を実現するものの 2 種類がある．High Performance Computing (HPC)においては主記憶の帯域が重要であるため，広帯域型の三次元積層メモリが主流になると予想される．しかしながら広帯域型の三次元積層メモリの容量は既存のメモリモジュールと同程度の水準に留まるため，今後のメニーコア化の進展に伴ってコアあたりのメモリ容量が減少することが課題である．

ポストペタスケール時代ではメモリ消費量の削減がより重要となるため，通信ライブラリにおいても連続的改良だけでなく，革新的進歩が必要となっている．従来の通信ライブラリで動的に割り当てられた通信バッファは，処理コストの高い解放，再割当てを回避するため，解放せずに割り当てられ続ける．しかしながらメモリ使用量の削減が求められる将来の通信ライブラリでは，アプリケーション性能を劣化させることなく通信バッファを解放，再割当てしなければならない．この最適化にはアプリケーションの通信パターンの性質に対する深い理解が必要であり，自動最適化には限界がある．将来の通信ライブラリは，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必要がある．

Advanced Communication for Exa (ACE) プロジェクト [1] ではプロセッサのメニーコア化が進むエクサスケールの時代に向けて，プロセスあたりの消費メモリ量を抑制しつつ，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．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，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ており，中間層にはコミュニケーションライブラリおよびデータライブラリという 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ストリーム転送の使用メモリを最適化するためのチャネルインタフェースを中核とし，データライブラリは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．

本論文では分散データ構造インタフェースの設計思想と構成方法，プロトタイプ実装と初期性能評価結果を紹介する．以降では，2 章で分散データ構造インタフェースを構築する基盤となる ACP 基本層について紹介し，3 章で分散データ構造インタフェースの設計思想と内部構成について述べる． 4 章で主要なインタフェース仕様を紹介し，5 章で初期評価結果を示す．6 章で関連研究につい記述し，最後に 7 章でまとめる．

2. ACP 基本層

分散データ構造インタフェースは ACP 基本層の上に構築される．基本層はエクサスケール時代に相応しいインターコネクトデバイス抽象化層の機能を定義する．

従来のインターコネクトデバイスは自プロセスと他プロセス間でデータを転送する片側通信機能を有するが，片側通信では読出し側もしくは書き込み側のプロセスがデータ転送を制御する必要がある．基本層では無駄なデータ移動や無駄な同期を削減するために，送信元でも宛先でもないプロセスがプロセス間のデータ転送を制御する関数 acp_copy を導入する．acp_copy では送信元，宛先の引数に ACP が提供するグローバルアドレスを使用する．

†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)

グローバルアドレスは対象となるメモリ領域を ACP ライブラリに登録することで発行される．リモートプロセスに動的にデータを生成するためにはリモートプロセスのメモリを確保する必要がある．基本層のインタフェースでグローバルアドレスを取得するために登録できるメモリはローカメモリだけである．

3. 設計思想と構成方法

3.1 設計思想

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．

3.2 グローバルメモリアロケータ

基本層のインタフェースでグローバルアドレスを取得するために登録できるメモリはローカルプロセスのメモリだけである．しかしリモートプロセスに動的にデータを生成するためにはリモートプロセスのメモリを確保する必要がある．

この要求を満たす最小限の解は非同期グローバルヒープ [3] である．非同期グローバルヒープとは各ノードが空きメモリを登録しておき，他プロセスが端から順に空き領域を取得できるようにしたデータ構造である．非同期グローバルヒープの割当済み領域と空き領域の境界はブレークポインタで示される．ブレークポインタの操作を排他制御することにより，複数プロセスが同時にメモリを取得しようとする際の競合を解決する．しかし，1 つの非同期グローバルヒープから複数プロセスがメモリを取得した場合，メモリの解放が困難になる問題がある．

そこで分散動的データ構造インタフェースでは，より高度なメモリ管理アルゴリズムの実装を想定した，グローバルメモリアロケータを導入する．グローバルメモリアロケータではブレークポインタの操作でメモリ割当，解放を行うのではなく，メモリ割当と解放に別のインタフェース，具体的には acp_malloc および acp_free 関数を用意する．割り当てたメモリ量などの管理情報は利用者からは隠されて管理されており，割り当てたメモリを解放する際は該当のグローバルアドレスを指定して acp_free 関数を呼び出すだけで良い．

3.3 データ構造とインタフェース

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)を参考に，可変長一次元配列 vector, 双方向リンクトリスト list, 双方向キューdeque, 連想配列 map，集合 set をサポートする．また，インタフェースも同様に STL を参考とするが, ACP ライブラリは C 言語のライブラリであるので，演算子は全て関数として実装され，イテレータは元のデータ構造への参照も引数として求められる場合があるなど，やや複雑なインタフェースとなっている．ただし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くなるように考慮される．

4. インタフェース仕様

本章では分散動的データ構造インタフェースの主要なインタフェース仕様を紹介する．各データ型のインタフェースに関しては特にベクタ(vector)型，およびリスト(list)型について説明する．

4.1 グローバルメモリアロケータ関数

グローバルメモリアロケータ関数の仕様を表 1 に示す. グローバルメモリ割当関数 acp_malloc およびグローバルメモリ解放関数 acp_free の定義はほぼ C 言語の標準 C ライブラリの malloc，free 関数と同様であるが，acp_malloc 関数にはメモリ割り当てを行うプロセス番号の指定がある点が異なる．

4.2 ベクタ型関数

ベクタ型の主要関数仕様を表 2 に示す. ベクタ生成関数 acp_create_vector では要素数，要素サイズ，プロセス番号を指定する．要素数を明示的に指定する理由は，分散動的データ構造インタフェースではデータ型をサポートせず，上位層での実装に任せるためである．プロセス番号は生成するベクタ型データを配置するプロセスを指定する．ベクタ生成関数以外にベクタ複製関数 acp_duplicate_vector もベクタの生成を伴うため，プロセス番号の指定を伴う．ベクタ型のデー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．

ベクタの各要素にアクセスするためにはまずベクタ先頭イテレータ取得関数 acp_begin_vector などでイテレータを取得し，イテレータを引数として指定した上でベクタ間接参照関数 acp_dereference_vector を呼び出す必要がある．ベクタ型のイテレータの実体はインデックスを格納する整数であり，即値からの型変換やイテレータに対する演算も可能である．

また，ベクタ型は可変長であり，ベクタ末尾要素追加関数 acp_push_back_vector や末尾要素削除関数 acp_pop_back_vector などにより, 要素数は動的に増減する．ベクタ型データの要素数が変わる関数の中には，呼び出し以前のイテレータを無効化する副作用を持つものがある．

4.3 リスト型関数

リスト型の主要関数仕様を表 3 に示す. リスト生成関数

acp_create_list では要素サイズとプロセス番号だけを指定する．これは生成直後のリスト型データは要素数 0 の空であるためである．プロセス番号は管理情報を配置するプロセスを指定する. 先頭要素追加関数 acp_push_front_list や末尾要素追加関数 acp_push_back_list など，リストに要素を追加する関数にはプロセス番号の指定がある．すなわち，リスト型データは要素ごとに配置するプロセスを指定することが出来る. リスト型イテレータの実体はポインタであり，ベクタ型イテレータのように演算することはできない．

表 1　グローバルメモリアロケータ関数
Table 1　Global memory allocator functions

名称	定義
割当	acp_ga_t **acp_malloc**(size_t *size*, int *rank*);
解放	void **acp_free**(acp_ga_t *ga*);

表 2　ベクタ型主要関数
Table 2　Vector data type major functions

名称	定義
ベクタ生成	acp_vector_t **acp_create_vector**(size_t *nelem*, size_t *elsize*, int *rank*);
ベクタ破棄	void **acp_destroy_vector**(acp_vector_t *vector*);
ベクタ間接参照	acp_ga_t **acp_dereference_vector**(acp_vector_t *vector*, acp_vector_it_t *it*);
ベクタ先頭イテレータ取得	acp_vector_it_t **acp_begin_vector**(acp_vector_t *vector*);
ベクタ末尾イテレータ取得	acp_vector_it_t **acp_end_vector**(acp_vector_t *vector*);
ベクタ型イテレータ増加	acp_vector_it_t **acp_increment_vector_it**(acp_vector_it_t *it*);
ベクタ型イテレータ減少	acp_vector_it_t **acp_decrement_vector_it**(acp_vector_it_t *it*);
ベクタ充填	void **acp_fill_vector**(acp_vector_t *vector*, size_t *nelem*, acp_ga_t *ga*);
ベクタ代入	void **acp_assign_vector**(acp_vector_t *vector1*, acp_vector_t *vector2*, acp_vector_it_t *it1*, acp_vector_it_t *it2*);
ベクタ末尾要素追加	void **acp_push_back_vector**(acp_vector_t *vector*, acp_ga_t *ga*);
ベクタ末尾要素削除	void **acp_pop_back_vector**(acp_vector_t *vector*);
ベクタ要素挿入	acp_vector_it_t **acp_insert_vector**(acp_vector_t *vector*, acp_vector_it_t *it*, acp_ga_t *ga*);
ベクタ要素削除	acp_vector_it_t **acp_erase_vector**(acp_vector_t *vector*, acp_vector_it_t *it*);
ベクタ全要素交換	void **acp_swap_vector**(acp_vector_t *vector1*, acp_vector_t *vector2*);
ベクタ全要素消去	void **acp_clear_vector**(acp_vector_t *vector*);
ベクタ複製	acp_vector_t **acp_duplicate_vector**(acp_vector_t *vector*, int *rank*);

表 3　リスト型主要関数
Table 3　List data type major functions

名称	定義
リスト生成	acp_list_t **acp_create_list**(size_t *elsize*, int *rank*);
リスト破棄	void **acp_destroy_list**(acp_list_t *list*);
リスト間接参照	acp_ga_t **acp_dereference_list**(acp_list_t *list*, acp_list_it_t *it*);
リスト先頭イテレータ取得	acp_list_it_t **acp_begin_list**(acp_list_t *list*);
リスト末尾イテレータ取得	acp_list_it_t **acp_end_list**(acp_list_t *list*);
リスト充填	void **acp_fill_list**(acp_list_t *list*, size_t *nelem*, acp_ga_t *ga*, int rank);
リスト代入	void **acp_assign_list**(acp_list_t *list1*, acp_list_t *list2*, acp_list_it_t *it1*, acp_list_it_t *it*, int rank);
リスト先頭要素追加	void **acp_push_front_list**(acp_list_t *list*, acp_ga_t *ga*, int *rank*);
リスト先頭要素削除	void **acp_pop_front_list**(acp_list_t *list*);
リスト末尾要素追加	void **acp_push_back_list**(acp_list_t *list*, acp_ga_t *ga*, int *rank*);
リスト末尾要素削除	void **acp_pop_back_list**(acp_list_t *list*);
リスト要素挿入	acp_list_it_t **acp_insert_list**(acp_list_t *list*, acp_list_it_t *it*, acp_ga_t *ga*, int *rank*);
リスト要素削除	acp_list_it_t **acp_erase_list**(acp_list_t *list*, acp_list_it_t *it*);
リスト型イテレータ増加	acp_list_it_t **acp_increment_list_it**(acp_list_it_t *it*);
リスト型イテレータ減少	acp_list_it_t **acp_decrement_list_it**(acp_list_it_t *it*);
リスト全要素交換	void **acp_swap_list**(acp_list_t *list1*, acp_list_t *list2*);
リスト全要素消去	void **acp_clear_list**(acp_list_t *list*);
リストソート	void **acp_sort_list**(bool (**compare*)(const acp_ga_t *ga1*, const acp_ga_t *ga2*));

5. 初期評価

5.1 評価環境

初期評価は PC クラスタとスーパーコンピュータで実施した．表 4 および表 5 に評価環境の詳細を示す．PC クラスタのインターコネクトは Gigabit Ethernet であり，UDP [4] 版の ACP 基本層を使用して評価した．

表 4　評価環境 1: PC クラスタ
Table 4　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

スーパーコンピュータのインターコネクトは Tofu インターコネクト [5][6] であり，Tofu 版の ACP 基本層を使用して評価した．

表 5　評価環境 2: スーパーコンピュータ
Table 5　Evaluation environment2: supercomputer

Node	Fujitsu Supercomputer PRIMEHPC FX10
CPU	Fujitsu SPARC64TM IXfx (16 cores, 1.848 GHz)
Memory	64GB, DDR3 1333MHz
Network	Tofu interconnect (5.0 Gbyte/sec)
OS	Linux version 2.6.52.8

以上の評価環境において，グローバルメモリ割当関数 acp_malloc および解放関数 acp_free の実行時間を計測した．

acp_malloc については 100 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 100 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 100 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．

5.2 評価結果

図 1 にグローバルメモリアロケータ関数の実行時間評価結果を示す．Tofu 版と UDP 版の ACP 基本層，acp_malloc 関数と acp_free 関数，local と remote の割当先の違いで，計 8 種類の評価結果がグラフに示されている．

割当先 local の場合，acp_malloc および acp_free 関数の平均実行時間は Tofu 版で 9 及び 19 μ 秒，UDP 版で 16 および 26 μ 秒となった．この場合インターコネクトを介した通信は行っていないため，実行時間の差は ACP 基本層の実装方式の違いおよびプロセッサの特性に由来する．Tofu 版も UDP 版も通信処理専用にスレッドを起動し，自プロセスが宛先の場合でも通信スレッドが処理する．そのため，主にスレッド間通信のオーバヘッドが実行時間に表れている．また，acp_malloc 関数に比べて 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．一方，メモリの割当てはフラグメンテーションが進まない限りほぼ O(1)の計算量である．

割当先 remote の場合，acp_malloc および acp_free 関数の平均実行時間は Tofu 版で 34 及び 68 μ 秒，UDP 版で 2.0 および 7.9 ミリ秒となった．

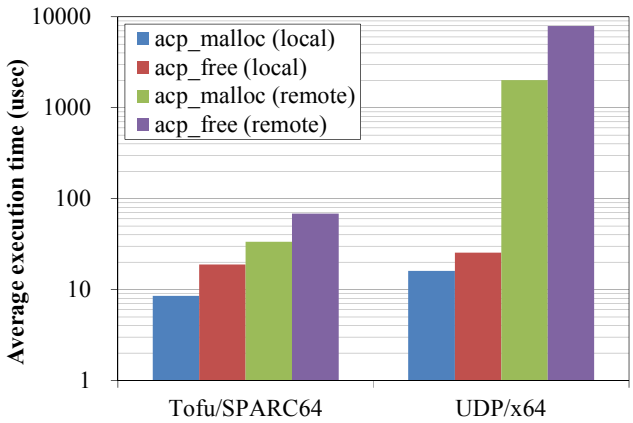

図 1　グローバルメモリアロケータ関数の平均実行時間評価結果
Figure 1　Evaluation results of average execution time of global memory allocator functions

5.3 考察

Tofu 版では local と remote で実行時間は 3〜4 倍しか変わらないが，UDP 版では 100〜300 倍遅くなる．分散動的データ構造インタフェースは従来のメッセージパッシング通信に比べ，より高い頻度，より細かい粒度で通信が行われることを想定しており，実用上は Tofu インターコネクトのような低遅延のインターコネクトの使用が必須であるといえる．

著者らは非同期グローバルヒープの提案 [3] において，RDMA とアトミック通信のリモート側での順序制御が可能であれば通信遅延の隠ぺいにより，ヒープ操作関数の実行時間を半減できることを示した．しかし，現在のグローバルメモリアロケータ実装ではリモート側での順序制御は行っていない．これは ACP 基本層の仕様としてはリモート側順序制御を定義してあるが，現在の実装は Tofu 版，UDP 版とも送信元で順序を保証する簡易実装になっているためである．今後 Tofu 版，UDP 版 ACP 基本層でリモート側順

序制御が実装されれば，グローバルメモリアロケータ関数の実行時間が半減することが期待される．また，FX10 後継機向け Tofu インターコネクト 2 のように RDMA とアトミック通信のリモート側順序制御をハードウェアでサポートするインターコネクトを使用する場合，ACP 基本層のリモート側順序制御実装が簡単になると期待される．

6. 関連研究

本節では既存の，上位層に並列ランタイムを想定した低レベル通信ミドルウェアを紹介する．

6.1 ARMCI

Aggregate Remote Memory Copy Interface (ARMCI) [7] は米国パシフィックノースウェスト国立研究所で開発されている低レベル通信ライブラリである．ARMCI は上位の通信ミドルウェアである Global Arrays Toolkit で使用されるために設計されている．ARMCI は RDMA Put/Get 通信やアトミック通信, 集団通信をサポートする. ARMCI は RDMA やアトミック通信のために，MPI と併用してを利用されることがある．

6.2 GASNet

Global-Address Space Networking (GASNet) [8] は米国ローレンスバークレー国立研究所およびカリフォルニア大学バークレー校が開発した低レベル通信インタフェースである．GASNet はグローバルアドレス空間を有する言語処理系のランタイムシステムの実装に使用されることを想定されている．GASNet は迅速なプロトタイピングと新しいシステムへの移植の簡易さを目指したアクティブメッセージ・パラダイムに基づいたコア API を有する．GASNet の拡張 API は Put および Get データ転送機能とバリア同期機能を有する．

6.3 UCCS

Universal Common Communication Substrate (UCCS) [9] は米国オークリッジ国立研究所で開発されたネットワークインタフェース抽象化ライブラリである．UCCS は近年，通信ミドルウェア OpenSHMEM によってサポートされた．従来の OpenSHMEM は GASNet 上に実装されていた．UCCS はアクティブメッセージ，RDMA Put/Get 通信，アトミック通信，集団通信に対応し，全ての通信 API は非ブロッキングである. UCCS は集団通信に Cheetah フレームワーク [10] を使用している．適切なプロトコルを明示的に使用するために，メッセージサイズ別に異なる RDMA Put/Get API を備える．

6.4 PAMI

Parallel Active Messaging Interface (PAMI) [11][12] は IBM Blue Gene/Q [13] のために開発された低レベル通信ライブラリである．ARMCI，MPI，UPC のような複数の上位層インタフェースの同時使用を可能にするため，PAMI ライブラリはクライアント方式を導入している．PAMI を使用するプログラムは最初に PAMI_Client_create サブルーチンを呼んでクライアント識別子を取得し，以降 PAMI サブルーチンを呼ぶ際は引数としてクライアント識別子を指定する．PAMI サブルーチンの引数は基本的にクライアント識別子とパラメータ構造体へのポインタで構成される．PAMI の通信機能は Send，Put，Get，Read-modify-write (Rmw)，集団通信，そして Active Message である．PAMI は通信プロトコルをバックグラウンドで進めるための通信スレッド機能を有する．

6.5 uGNI

user-level General Network Interface (uGNI) [14] は Cray Gemini および Aries インターコネクトのために開発された低レベル通信インタフェースである．uGNI では通信は論理エンドポイント間で行われる．どのエンドポイント間でもデータグラムの交換は可能である．RDMA や Fast Memory Access (FMA)を使用するには，エンドポイントは他のエンドポイントにバインドされていなければならない．FMA は Put，Get，Atomic Memory Operation (AMO)を含む短メッセージを扱う．Gemini および Aries インターコネクトは AMO キャッシュを備え, AMO のスループットを向上する．しかし，AMO キャッシュとプロセッサのキャッシュはキャッシュコヒーレントではない．

7. まとめ

本論文では ACE プロジェクトで開発している ACP ライブラリの分散動的データ構造インタフェースの設計思想，構成方法，主要なインタフェース仕様，初期評価結果を紹介した．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．またデータの生成，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．これらを実現するために，グローバルメモリアロケータを導入する．本論文では分散動的データ構造インタフェースのサポートするデータ型のうち，ベクタ型とリスト型について主要なインタフェースの仕様を示した．また，Tofu インターコネクト版および UDP 版の ACP 基本層上でグローバルメモリアロケータ関数の実行時間を評価した．acp_malloc および acp_free 関数の平均実行時間は割当先 local の場合，Tofu 版で 9 及び 19 μ 秒，UDP 版で 16 および 26 μ 秒，割当先 remote の場合，Tofu 版で 34 及び 68 μ 秒, UDP 版で 2.0 および 7.9 ミリ秒となった．分散動的データ構造インタフェースは実用上，Tofu インターコネクトのような低遅延のインターコネクトの使用が必須であるといえる．

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) 安島 雄一郎, 秋元 秀行, 岡本 高幸, 三浦 健一, 住元 真司: 非同期グローバルヒープの提案と初期検討, 情報処理学会研究会報告 2013-HPC-138-10 (2013)
4) Jonathan B. Postel (editor): User Datagram Protocol, RFC 768 (1980)
5) Yuichiro Ajima, et al: Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers, Computer, 42(11), pp. 36-40 (2009)
6) Yuichiro Ajima, et al.: The Tofu Interconnect, Micro, 32(1), pp. 21-31 (2012)
7) ARMCI - Aggregate Remote Memory Copy Interface, http://hpc.pnl.gov/armci/documentation.htm
8) GASNet Communication System, http://gasnet.lbl.gov/
9) UCCS - Universal Common Communication Substrate, http://uccs.github.io/uccs/
10) Richard L. Graham, Pavel Shamis, et al.: Cheetah: A Framework for Scalable Hierarchical Collective Operations, IEEE/ACM CCGrid 2011, pp. 73-83 (2011)
11) Sameer Kumar, Amith R. Mamidala, el al.: PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer, IEEE 26th IPDPS, pp. 764-774 (2012)
12) PAMI Programming Guide Version 1 Release 1.0, http://publib.boulder.ibm.com/epubs/pdf/a2322730.pdf
13) The IBM Blue Gene Team: The Blue Gene/Q Compute Chip, HOT CHIPS 23 (2011)
14) Using the GNI and DMAPP APIs, http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf