Vol.2016-HPC-157 No.12
2016/12/22

ACP 基本層 UDP 版におけるノード内通信性能の改善

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

概要：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 UDP 版基本層において，各ノードに複数プロセスが存在する際にノード内のプロセス間では socket インタフェースを介した UDP 通信ではなく共有メモリを介して通信を行う性能改善を実施し，さらに特定の条件でメモリコピーを通信スレッドでなくメインスレッドで行うバイパス機能を導入し，性能を評価した．評価の結果，新実装は他プロセスから別の他プロセスへのデータ転送，他プロセスから自プロセスへのデータ転送，および自プロセスから他プロセスへの遅延をおよそ 40～60%削減し，帯域をおよそ 3.2～3.4 倍向上した．自プロセスから自プロセスへのメモリコピーでは遅延を約 60%削減し，帯域はほぼ同等を実現した．

1. はじめに

エクサスケール時代の HPC システムでは，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．メニーコアと広帯域メモリの組合せでは，コアあたりのメモリ容量が減少することが課題である．エクサスケール時代では，通信ライブラリを含むシステムソフトウェアにおいてメモリ消費量の削減が重要となる．従来の多くの通信ライブラリでは，通信バッファは自動的に割当てられ，解放せずに割り当てられ続ける．これは通信バッファの解放，再割当てとも処理コストが高いためである．

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．我々は ACE プロジェクトの中核技術の一つとして，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[2].

ACP は省メモリに適した PGAS モデルでインターコネクトデバイスを抽象化した低レベル通信ライブラリである基本層[3]と，基本層を使用して様々な省メモリ通信アルゴリズムをポータブルに実装した中間層で構成される．基本層は InfiniBand，Tofu インターコネクトなどの高速なインターコネクトに加え，UDP [4]にも対応する．UDP は幅広い環境で利用できるのでアプリケーション開発環境向けに最適であり, UDP と同様に Ethernet を使用する TCP と比べるとプロトコルスタックによるメモリ使用量が小さい．

本論文では UDP 版 ACP 基本層において 1 ノードに複数プロセスがある場合，ノード内のプロセス同士が共有メモリ上のバッファを介して直接通信する方式を提案，評価する．以降では，2 章で UDP 版 ACP 基本層の概要と実装方式を説明し，3 章で課題を示すとともに改良方式を提案する．4 章で評価し，5 章では今後の課題について議論し，最後に 6 章でまとめる．

† 富士通株式会社
Fujitsu Limited

2. UDP 版 ACP 基本層

2.1 ACP 基本層のデータ転送

ACP 基本層は，実行開始時に静的割当容量を指示したメモリ，および初期化後に動的に登録したメモリに対するグローバルアドレスを提供し，グローバルアドレスを使用したデータ転送関数を提供する．データ転送関数 acp_copy() は任意のグローバルアドレスから任意のグローバルアドレスへデータを転送する．一般に低レベル通信ライブラリでは他プロセスから自プロセスにデータ転送する Get，自プロセスから他プロセスにデータ転送をする Put を提供することが多いが，ACP では acp_copy 関数 1 つで Get，Put 両方のデータ転送が可能である．さらに acp_copy 関数では他プロセスのデータを別のプロセスに転送することもできる．

acp_copy 関数は非ブロッキングであり，データ転送の開始を待たずに戻る．acp_copy 関数は返り値として GMA ハンドル（acp_handle_t 型）を返す．GMA ハンドルは，その値を返した acp_copy 関数呼び出しによるデータ転送の完了待ち合わせに使用する．acp_complete 関数は GMA ハンドルを引数に取り，GMA ハンドルを返した acp_copy 関数およびそれ以前に呼び出された関数によるデータ転送の完了を全て待ち合わせてから戻る．また，acp_copy 関数自身も他のデータ転送の完了を待ち合わせる機能を有しており，GMA ハンドルを引数に取る．連続して acp_copy 関数を呼び出す場合，直前の acp_copy 関数呼び出しの返り値を引数として与えるとデータ転送が逐次に実行され，直前の acp_copy 関数と同じ引数を与えるとデータ転送が並列に実行される．GMA ハンドルとして ACP_HANDLE_ALL 定数を指定すると，直前に返された GMA ハンドルを指定したことになる．また GMA ハンドルとして ACP_HANDLE_NULL 定数を指定すると，他のデータ転送を待ち合わせずにデータ転送を実行する．

2.2 UDP 版 ACP 基本層の実装

UDP 版 ACP 基本層では初期化時に各プロセスで通信スレッドを 1 つ生成し, 通信処理は全て通信スレッドで行う．データ転送関数は，通信スレッドに通信指示を格納したコマンドを供給する．ここで関数を呼び出したプロセスを開

ⓒ2016 Information Processing Society of Japan

始元と呼ぶ．データ転送の宛先と送信元はグローバルアドレスで指定されるので，実際にデータ転送を実行するのは宛先もしくは送信元のプロセスの通信スレッドである．そこで，開始元の通信スレッドは通信の内容に従って，宛先もしくは送信元の通信スレッドにコマンドを転送し，実行を依頼する．コマンドを受信した通信スレッドは通信を実行し，コマンド実行の完了を開始元に通知する．

コマンドやデータは UDP データグラムで転送される．UDP はデータグラムの送達が保証されないプロトコルであるので，UDP 版 ACP 基本層自身が再送信および輻輳回避制御を実装している．受信側は受信したデータグラムに対して送達を通知する応答を返す．送信側は送達確認のタイムアウトを検出し，データグラムを再送信する．コマンド転送では順序の入れ替わりや重複を起こさないために，データグラムにシーケンス番号を付与する．受信側はシーケンス番号の抜けを検出すると送信側に通知し，送信側は送出帯域を下げる．パケット破棄は通信競合によるバッファ溢れが主な原因であるので，送出帯域を下げることでバッファに滞留するパケット数を減らし，パケット破棄の頻度を下げる．

UDP 版 ACP 基本層の初期実装では，全てのプロセス間通信において UDP を使用していた．このため，1 ノードに複数のプロセスがある場合，ノード内の複数プロセスが同時にデータグラムを送信するとパケットロスが生じ，実効帯域が低下していた．この問題を解決するため，現在の UDP 版 ACP 基本層には 1 ノードに複数プロセスがある場合, 各ノードで 1 プロセスが代表して UDP 通信を行う改良が実装されている[5]．ここで，UDP 通信を行うプロセスはゲートウェイプロセス，UDP 通信を行わないプロセスは内部プロセスと呼ばれる．送信バッファ，および受信バッファは共有メモリに配置され，各プロセスの通信スレッドから参照される，ゲートウェイプロセスの通信スレッドは送信バッファのデータグラムを UDP で送信し，UDP で受信したデータグラムを受信バッファに書き込む．

3. 課題と提案

3.1 課題

従来の UDP 版 ACP 基本層のプロトコルでは，ノード間通信，ノード内通信の区別がなく，データグラムの送信先が同一ノード内のプロセスであっても UDP で送受信する．ノード間通信とノード内通信を同じコードで実装できる点は利点であるが, UDP は socket インタフェースで送受信するためにシステムコールのオーバーヘッドがかかる点が問題である．

3.2 提案

本論文では，ノード内プロセス間通信に限定して直接共有メモリ通信方式を提案する．この方式ではノード内のプロセス間通信に UDP を使用せず, 共有メモリ上に内部通信バッファを配置する．各プロセスはノード内の他プロセスそれぞれに対する受信バッファを専用に用意する．すなわち，ノード内に n プロセスが存在する場合，各プロセスに $(n\text{-}1)$組の受信バッファが配置され，ノード内の合計では $n(n\text{-}1)$組の受信バッファが配置される．ノード間通信とノード内通信の実装をなるべく共通化するため，プロトコルはノード間，ノード内とも共通とし，内部通信バッファはデータグラムの転送に使用する．

通信スレッドはメインスレッドからのコマンドを受け付けるコマンドキュー, 自プロセスの UDP 受信バッファに加え，内部通信バッファにおける自プロセスの受信バッファをポーリングする．ゲートウェイプロセスの通信スレッドはそれに加え，UDP 送信バッファをポーリングして UDP でデータグラムを送信するとともに, データグラムを UDP 受信する．すなわち，旧実装の通信スレッドは 1 個のコマンドキューをポーリングしつつ, UDP 送受信を行っていた．新実装のゲートウェイプロセスの通信スレッドはこれに加え，n-1 個の送信バッファと，n-1 個の内部通信バッファのポーリングを行う．新実装の内部プロセスの通信スレッドは 1 個のコマンドキュー，1 個の受信バッファと n-1 個の内部通信バッファをポーリングし, UDP 送受信は行わない．

新実装では通信スレッドのプロトコル処理負荷が増加するが, UDP 通信のオーバーヘッドに比べると小さいことが期待される．ただし，送信元，宛先とも自プロセスの場合は, 通信スレッドでメモリコピーし UDP 通信を行わないためオーバーヘッド増加が問題となると予想される. そこで, 送信元，宛先とも自プロセスの場合は通信スレッドではなくメインスレッドでメモリコピーを実施するバイパス機能を追加する．バイパス機能の実装には，順序制御を含めたコマンドの実行制御を通信スレッドとメインスレッドで協調して行う必要がある．このため，コマンドキューの制御変数だけでなく，コマンドの実行制御変数もメインスレッドと通信スレッドの共有変数とし，排他制御の対象とする変更を加えた．

4. 評価

本章では UDP 版 ACP 基本層の新旧実装によるノード内プロセス間通信のデータ転送遅延，スループットを評価する．評価では 1 ノードで 3 プロセスを起動し，他プロセスから別の他プロセスにデータを転送する remote-to-remote, 他プロセスから自プロセスにデータを転送する remote-to-local，自プロセスから他プロセスにデータを転送する local-to-remote, 自プロセスから自プロセスにデータを転送する local-to-local で行う．Local-to-local については通信スレッドをバイパスする場合の性能も評価する．評価は 1 ノード実行であるのでゲートウェイプロセスはなく，3

プロセスとも内部プロセスとして動作する．

4.1 評価環境

評価は PC クラスタで実施した．表 1 に評価環境を示す．プロセッサのハイパースレッディングはオンになっており，ノードあたりのスレッド数はコア数の 2 倍の 16 であった．インターコネクトは Gigabit Ethernet を使用して評価した．評価では 4 バイトから 4MiB までのデータを繰り返し転送し，データ転送回数が 1000 に達するか，データ転送量が合計 40MiB に達するまでの時間を計測し，データ転送 1 回あたりの平均時間を算出した．また，プロトコル処理全体のオーバーヘッドを遅延として測定するため，データ転送 1 回毎に acp_complete 関数で待ち合わせを行い，データ転送のオーバーラップが起きないようにした．

表 1 評価環境
Table 1 Evaluation environment

Node	Fujitsu PRIMERGY RX200 S5
CPU	Intel Xeon E5520 (4 cores, 2.27 GHz), 2 sockets
Memory	DDR3 SDRAM 48GB, 51.2 GB/s
Network	Gigabit Ethernet (125 Mbyte/sec)

4.2 Remote-to-remote データ転送の評価結果

図 1 は新旧実装の remote-to-remote データ転送の遅延評価結果である．旧実装では約 43 μ 秒，新実装では約 26 μ 秒の遅延が測定された．新実装は remote-to-remote データ転送の遅延を約 40%削減した．

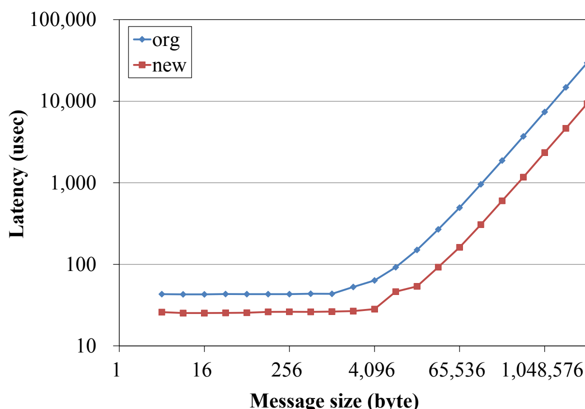

図 1 Remote-to-remote データ転送のレイテンシ
Figure 1 Latency results of remote-to-remote data transfer

図 2 は新旧実装の remote-to-remote データ転送の帯域評価結果である．旧実装では約 140 MB/s，新実装では約 450 MB/s の帯域が測定された．新実装は remote-to-remote データ転送の帯域を約 3.2 倍に向上した．

新実装は遅延，帯域とも大きく性能を向上し，特に帯域の向上が大きいことが分かった．

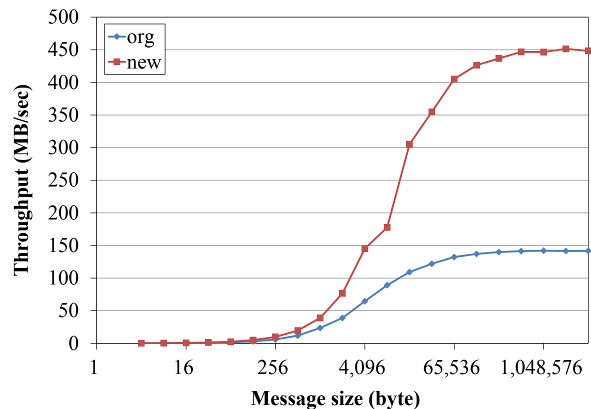

図 2 Remote-to-remote データ転送のスループット
Figure 2 Throughput results of remote-to-remote data transfer.

4.3 Remote-to-local データ転送の評価結果

図 3 は新旧実装の remote-to-local データ転送の遅延評価結果である．旧実装では約 13 μ 秒，新実装では約 5 μ 秒の遅延が測定された．新実装は remote-to- local データ転送の遅延を約 60%削減した．

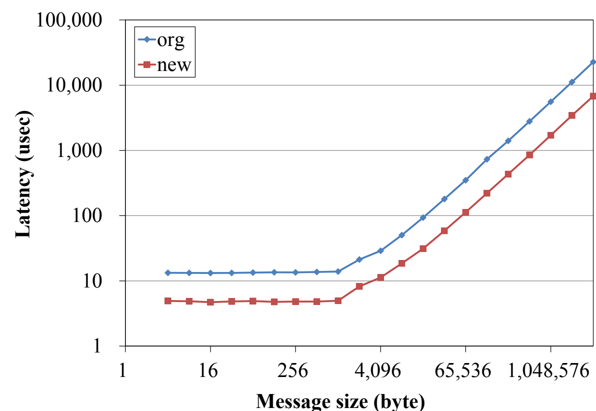

図 3 Remote-to-local データ転送のレイテンシ
Figure 3 Latency results of remote-to-local data transfer

図 4 は新旧実装の remote-to-local データ転送の帯域評価結果である．旧実装では約 190 MB/s，新実装では約 620 MB/s の帯域が測定された．新実装は remote-to-local データ転送の帯域を約 3.4 倍に向上した．

Remote-to-local データ転送では remote-to-remote データ転送と同様に，新実装は遅延，帯域とも大きく性能を向上し，特に帯域の向上が大きいことが分かった．

ⓒ2016 Information Processing Society of Japan

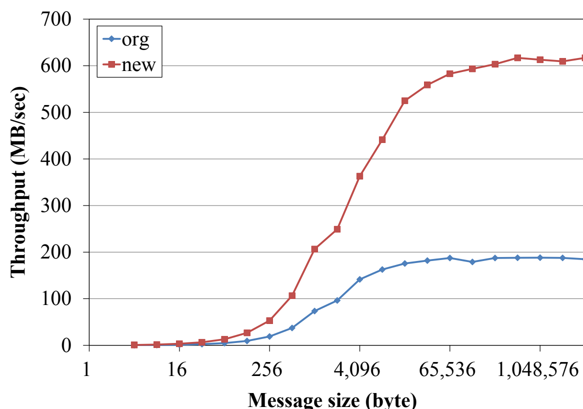

図 4 Remote-to-local データ転送のスループット

Figure 4 Throughput results of remote-to-local data transfer.

4.4 Local-to-remote データ転送の評価結果

図 5 は新旧実装の local-to-remote データ転送の遅延評価結果である．旧実装では約 14μ 秒，新実装では約 6μ 秒の遅延が測定された．新実装は local-to-remote データ転送の遅延を約 50%削減した．

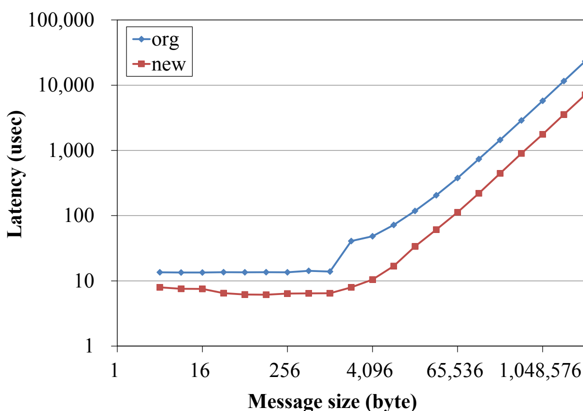

図 5 Local-to-remote データ転送のレイテンシ

Figure 5 Latency results of local-to-remote data transfer

図 6 は新旧実装の local-to-remote データ転送の帯域評価結果である．旧実装では約 180 MB/s，新実装では約 590 MB/s の帯域が測定された．新実装は local-to-remote データ転送の帯域を約 3.3 倍に向上した．

Local-to-remote データ転送では remote-to-local データ転送，remote-to-remote データ転送と同様に，新実装は遅延，帯域とも大きく性能を向上し，特に帯域の向上が大きいことが分かった．

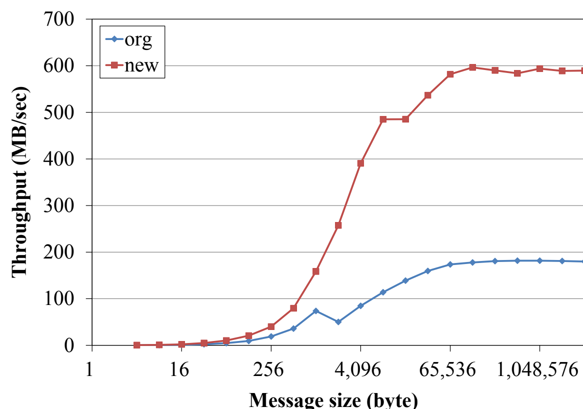

図 6 Local-to-remote データ転送のスループット

Figure 6 Throughput results of local-to-remote data transfer.

4.5 Local-to-local データ転送の評価結果

図 7 は新旧実装の local-to-local データ転送の遅延評価結果である．旧実装では約 1.3μ 秒，新実装では約 4.9μ 秒の遅延が測定された．新実装は local-to-remote データ転送の遅延を約 3.8 倍に増加した．これに対して新実装に通信スレッドバイパスを加えた場合の遅延は約 0.5μ 秒であり，旧実装から遅延を約 60%削減している．

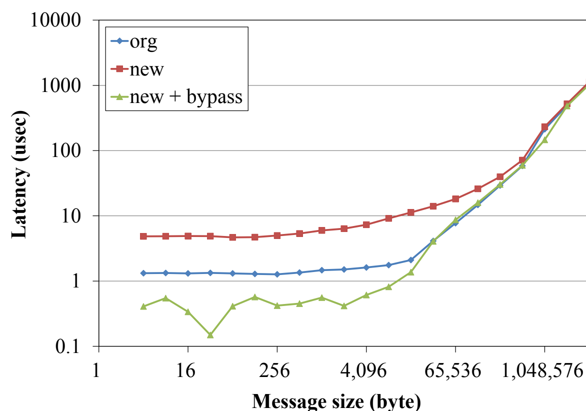

図 7 Local-to-local データ転送のレイテンシ

Figure 7 Latency results of local-to-local data transfer.

図 8 は新旧実装の local-to-local データ転送の帯域評価結果である．旧実装ではメッセージサイズが小さい方から 512KiB まで帯域が向上し，約 9.0 GB/s に達した．これに対して新実装でもメッセージサイズが増加すると帯域も増加し，512KiB で約 7.3 GB/s の帯域が計測された．この差は新旧実装の通信スレッドの負荷の違いに起因すると考えられる．新実装に通信スレッドバイパスを加えた場合は旧実装とほぼ同等の約 8.9 GB/s の帯域が計測されたことも，これを裏付けている．メッセージサイズが 1MiB 以上に増加すると，どの実装でも帯域が 4 GB/s 近辺に低下する．これは L2 キャッシュのミスによると推測される．また，新実装にバイパスを加えた場合はメッセージサイズ 16KiB で帯域が

ⓒ2016 Information Processing Society of Japan

約 12 GB/s に達している．これは通信スレッドバイパスによりメインスレッドがメモリコピーを行うため，L1 キャッシュのスループットが反映されていると考えられる．

Local-to-local データ転送では新実装の通信スレッド負荷増加が遅延，帯域を低下させるが，通信スレッドバイパスにより旧実装から遅延を約 60%削減し，帯域は同等を達成した．

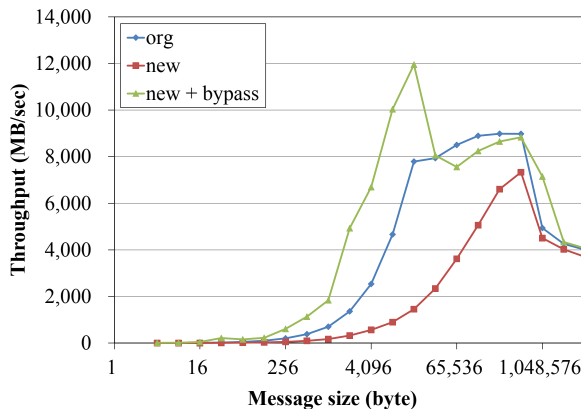

図 8　Local-to-local データ転送のレイテンシ

Figure 8　Throughput results of local-to-local data transfer.

5.　今後の課題

共有メモリ上の全対全の内部通信バッファによるノード内プロセス間通信は，多数のバッファをポーリングしなければならないためオーバーヘッドが大きい．この対策としては２つのアプローチが考えられる．

1 つ目はプロトコル処理の回数を減らすことである．新実装ではノード間通信でもノード内通信でも 1 つのデータグラムに格納できる最大のデータ長を 1,408 バイトとしているが，これは Ethernet フレーム長の制約によるものであり，共有メモリ通信ではデータ長を拡大することが可能である．ただしこのアプローチはメモリ使用量を増加させることが懸念される．

2 つ目はバッファの数を減らすことである．送受信相手が 1 対 1 で固定された全対全の通信バッファではなく，プロセスあたり 1 つの受信バッファに複数のプロセスが同時にアクセスすれば，各プロセスは 1 つの受信バッファをポーリングすればよい．このアプローチの懸念事項は，複数のプロセスが受信バッファの制御変数にアクセスするため，排他制御がオーバーヘッドになることである．

また，共有メモリ上のバッファを介するプロセス間通信では，送信元のプロセスがバッファにデータをコピーし，宛先のプロセスがバッファからデータをコピーするので，2 回のメモリコピーが帯域を制約している．ここで Linux Kernel 3.2 以降に含まれている Cross Memory Attachment (CMA)を使用すると，ノード内のプロセス間で直接メモリコピーを行える．CMA はメモリコピーが 1 回であるので，2 回以上メモリコピーを行う共有メモリバッファや UDP データグラム転送に比べ，データ転送帯域の向上に役立つと考えられる．

ここで，CMA に関しては 2 つ懸念事項がある．1 つ目はデータ転送のオーバーヘッドである．CMA はシステムコールであるのでオーバーヘッドが大きいと予想される．2 つ目はアドレス変換のオーバーヘッドである．CMA を使用する場合，グローバルアドレスをアドレス変換し，目標のプロセスにおける論理アドレスを得ることが必要である．そのためにはノード内の全プロセスがそれぞれのアドレス変換テーブルを共有メモリに置き，他のプロセスに参照させる必要がある．これにはアドレス変換テーブルに関する制御変数もまた共有変数にしなければならず，アドレス変換の際に排他制御のオーバーヘッドがかかる．

CMA に関する 2 つの懸念事項は，静的割当メモリに関しては，共有メモリに配置することで解決できると考えられる．具体的には，初期化時に各プロセスは自プロセスの静的割当メモリを無名 mmap 上に確保し，ノード内の他プロセスと共有する．各プロセスは初期化時にさらに，共有メモリ上の他プロセスの静的割当メモリの自プロセスの論理アドレス空間内の位置をアドレス変換テーブルに記録しておく．この仕組みにより，グローバルアドレスがノード内他プロセスの静的割当メモリを指している場合，CMA を使用しなくてもプロセッサが直接メモリコピーを行うことができる．

InfiniBand や Tofu インターコネクトなど RDMA プロトコルをハードウェアで実行するデバイスでは，おそらくノード間通信をノードあたり 1 プロセスに制約する必要はない．それに対して全対全の内部通信バッファは RDMA が使用できる場合でも帯域で上回る可能性があり，将来 CMA が導入されるとさらにその可能性が高まる．また，通信スレッドをバイパスできる場合は RDMA よりも低遅延，高帯域であることが期待でき，将来静的割当メモリのノード内共有が導入されるとバイパスできるケースがより多くなると期待できる．UDP 版に限らず他の ACP 基本層実装にもノード内プロセス間の共有メモリ通信の適用を拡大することが今後の課題である．

6.　まとめ

本論文では ACE プロジェクトで開発している ACP ライブラリの UDP 版基本層において，各ノードに複数プロセスが存在する際にノード内のプロセス間では socket インタフェースを介した UDP 通信ではなく共有メモリを介して通信を行う性能改善を実施し，性能を評価した．評価の結果，新実装は他プロセスから別の他プロセスへのデータ転送，他プロセスから自プロセスへのデータ転送，および自

ⓒ2016 Information Processing Society of Japan

プロセスから他プロセスへの遅延をおよそ40～60%削減し，帯域をおよそ3.2～3.4倍向上することが分かった．しかし，自プロセスから自プロセスへのメモリコピーでは帯域はほぼ同等である一方，遅延には約3.8倍の増加が見られた．これは通信バッファの数が増えることによる通信プロセスのオーバーヘッド増加が，UDP通信に対しては十分小さいものの，CPUによるメモリコピーに対しては大きいことが原因である．そこでメモリコピーを通信スレッドでなくメインスレッドで行うバイパス機能を導入し，遅延を旧実装から約60%削減した．

謝辞　本研究は，JST， CRESTの支援を受けたものである．

参考文献

[1] ACE Project, http://ace-project.kyushu-u.ac.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] 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)
[4] Jonathan B. Postel (editor): User Datagram Protocol, RFC 768 (1980)
[5] 安島雄一郎, 野瀬貴史, 佐賀一繁, 志田直之, 住元真司: ACP基本層UDP版におけるノード内複数プロセス時のノード間通信性能の改善, 情報処理学会研究会報告 2016-HPC-156-4 (2016)

ⓒ2016 Information Processing Society of Japan