新しいタスクモデルによるメニーコア環境に適した MPI ノード内通信の実装

島田 明男[1,†1,a)] 堀 敦史[1,b)] 石川 裕[1,c)]

受付日 2014年10月18日, 採録日 2015年2月3日

概要：近年，HPC システムを構成するノード1台あたりのコア数は飛躍的に増加してきている．一方で，1コアあたりのメモリ量は減少する傾向にある．Partitioned Virtual Address Space（PVAS）は，このようなメニーコア環境において，効率的に並列処理を行うための新たなタスクモデルである．PVAS タスクモデルを用いると，並列処理を行うノード内のプロセス群を同一アドレス空間で動作させることが可能になるため，アドレス空間越しにデータを送受信するためのコストを払うことなく，プロセス間でノード内通信を実行することができる．本研究では，PVAS タスクモデルを Message Passing Interface（MPI）に適用することで，高速かつメモリ消費量の少ない，よりメニーコア環境に適した MPI ノード内通信を実現できることを示す．PVAS タスクモデルを利用した MPI ノード内通信を実装し，NAS Parallel Benchmarks により評価したところ，既存の MPI ノード内通信の実装と比べて最大で約 18%，実行性能が改善された．また，Intel MPI Benchmarks によって，MPI ノード内通信のメモリ消費量を測定したところ，最大で約 264 MB，メモリ消費量を削減することができた．

キーワード：HPC，メニーコア，OS，MPI，ノード内通信

Implementing Many-core Friendly MPI Intra-node Communication with New Task Model

Akio Shimada[1,†1,a)] Atsushi Hori[1,b)] Yutaka Ishikawa[1,c)]

Received: October 18, 2014, Accepted: February 3, 2015

Abstract: Recently, number of cores in an HPC system node grows rapidly. Meanwhile, the amount of per-core memory resources seems to be limited. Partitioned Virtual Address Space (PVAS) is a new task model, which enables efficient parallel processing in such many-core environments. The PVAS task model allows multiple processes to run in the same address space, which means that processes running on the PVAS task model can conduct intra-node communication without incurring extra costs when crossing address space boundaries. In this paper, the PVAS task models is applied to Message Passing Interface (MPI), and many-core friendly MPI intra-node communication, which can conduct fast communication with small memory footprint, is proposed. The benchmark results show that the MPI intra-node communication proposed in this paper improves MPI application performance by up to 18%. Moreover, the memory footprint for MPI intra-node communication is decreased by up to 264 MB.

Keywords: HPC, many-core, OS, MPI, intra-node communication

1. はじめに

近年では，電力効率の観点から，コア単体の処理性能を高めるよりも1プロセッサあたりのコア数を増加させることで高い処理能力を実現する CPU アーキテクチャが一般的になっており，HPC システムを構成するノード1台あ

[1] 理化学研究所計算科学研究機構
RIKEN AICS, Kobe, Hyogo 640–0047, Japan
[†1] 現在，株式会社日立製作所
Presently with Hitachi Ltd.
[a)] a-shimada@riken.jp
[b)] ahori@riken.jp
[c)] yutaka.ishikawa@riken.jp

© 2015 Information Processing Society of Japan

表 1　Xeon Phi の製品使用
Table 1　Specifications of Xeon Phi.

型番	7100 番台	5100 番台	3100 番台
コア数*1	61	60	57
論理コア数	244	240	228
コア周波数	1.238 GHz	1.053 GHz	1.1 GHz
メモリサイズ	16 GB	8 GB	6 GB

たりのコア数は飛躍的に増加してきている．ノード 1 台あたりのコア数が増加する一方で，1 コアあたりのメモリ量は減少する傾向にある [7]．次世代 HPC システムでの採用が期待される Hybrid Memory Cube [13] のような高性能メモリは容量が少なく，多数のコアが存在する場合，1 コアに割り振ることができるメモリ量は厳しく制限される．また，PCI カードデバイスとして提供される coprocessor 型のメニーコアプロセッサは，搭載できるメモリ量に制限がある．たとえば，Intel 社のメニーコア製品である Intel Xeon Phi は**表 1** に示すような製品仕様となっており，下位モデルでは，1 論理コアあたりのメモリ量は約 27 MB となる．

Partitioned Virtual Address Space（PVAS）[27], [28], [31] は，このようなメニーコア環境において効率的に並列処理を行うために，本研究の著者らが提案，開発した新たなタスクモデルである．PVAS タスクモデルは並列処理を行うノード内のプロセス群を同一アドレス空間で動作させることを可能にする．既存のタスクモデルでは並列処理を行うノード内の各プロセスが個別のアドレス空間で動作するため，互いのメモリに直接アクセスすることができない．よって，ノード内通信を行う際に，通信遅延の増加やメモリ消費量の増加といった，アドレス空間越しにデータを送受信するためのコストが発生する．対して，PVAS タスクモデルによって同一アドレス空間で動作するプロセスどうしは互いのメモリに直接アクセスすることができるため，アドレス空間越しにデータを送受信するためのコストを払うことなく，ノード内通信を実行することができる．PVAS タスクモデルは Message Passing Interface（MPI）[20] や Partitioned Global Address Space（PGAS）[2], [6], [17], [19] のような，ノード内通信が行われる並列化モデルに幅広く適用することができる．本研究では，PVAS タスクモデルを MPI に適用し，よりメニーコア環境に適した MPI ノード内通信の実装を行った．

MPI は並列計算処理のための通信規格であり，並列アプリケーションの開発に広く用いられている．MPI は複数のプロセスを起動し，並列処理を実行させるプログラミングモデルになっている．並列処理を行うプロセスどうしは，並列処理に必要なデータを MPI ライブラリの提供する通信 API を用いて，互いに送受信することができる．MPI を利用して並列処理を行うプロセスを MPI プロセスと呼ぶ．MPI を利用して実装されたアプリケーションを HPC システム上で実行する場合，システムの並列計算処理能力を効率的に利用するため，利用可能なコア数に応じて MPI プロセスの数を調整するのが一般的になっている．多数のコアを利用可能な環境では，より多くの MPI プロセスが動作することになる．

MPI アプリケーションの実行性能は MPI 通信の性能に大きな影響を受ける．MPI 通信は，異なるノード上で動作する MPI プロセスどうしの通信である MPI ノード間通信と，同一ノード上で動作する MPI プロセスどうしの通信である MPI ノード内通信に分けられる．ノード 1 台あたりのコア数が増加するメニーコア環境では多数の MPI プロセスが同一ノード上で動作する．よって，アプリケーション実行時に発生する MPI ノード内通信の回数が従来よりも増加する．MPI ノード間通信はもちろんのこと，MPI ノード内通信の性能がアプリケーションの実行性能に従来よりも大きな影響を与えることになり，高速な MPI ノード内通信が求められる．MPI ノード内通信の性能を向上させることが，1 ノードあたりの実行性能だけではなく，システム全体での実行性能向上につながる．

また，メニーコア環境では，MPI ノード内通信のメモリ消費量の低減も求められる．MPI ノード内通信によるメモリ消費量は，通信を行う同一ノード内の MPI プロセスの数にともない増加する．しかし，メニーコア環境では，同一ノード内の MPI プロセスの数が従来よりも増加する反面，コアあたりのメモリ量は少なくなる傾向にある．このような環境でアプリケーションにより多くのメモリを割り当てるためには，MPI ノード内通信のメモリ消費量を低減させる必要がある．

MPI ノード内通信を行うためには通信制御情報や通信メッセージといったデータを MPI プロセスどうしで送受信する必要がある．MPI は複数のプロセスに並列処理を実行させるプログラミングモデルであり，各 MPI プロセスは個別のアドレス空間で動作する．既存の MPI ノード内通信の実装では MPI ノード内通信の実行に必要なデータの送受信を行う際，アドレス空間越しにデータを送受信するためのコストが発生する．このコストは MPI ノード内通信の通信遅延の増加やメモリ消費量の増加といったかたちで顕在化される．

そこで本研究では，PVAS タスクモデルを用いて，並列処理を行うノード内の MPI プロセスを同一アドレス空間で実行し，MPI ノード内通信からアドレス空間超しにデータを送受信するためのコストを排除することを提案する．アドレス空間越しにデータを送受信するためのコストを排除することで，高速かつメモリ消費量の少ない，よりメニーコア環境に適した MPI ノード内通信を実現する．

PVAS タスクモデルを利用した MPI ノード内通信を主要

*1　1 コアあたり 4 つのハードウェアスレッドをサポート．

© 2015 Information Processing Society of Japan

な MPI ライブラリの 1 つである OpenMPI [22] に実装し，NAS Parallel Benchmarks [5] を用いて評価したところ，既存の MPI ノード内通信の実装と比べて最大で約 18%，実行性能が向上した．また，Intel MPI Benchmark [15] を用いて PVAS タスクモデルを利用した MPI ノード内通信のメモリ消費量を測定したところ，既存の MPI ノード内通信の実装と比べて最大で約 264 MB，メモリ消費量を削減することができた．

本論文の構成は以下のとおりである．次章で既存の MPI ノード内通信の実装とその問題点について述べる．3 章で PVAS タスクモデルの概要と PVAS タスクモデルを用いた MPI ノード内通信の実装について説明する．4 章で評価について述べ，5 章で関連研究について述べる．6 章で本研究について議論し，最後に本研究の結論を述べる．

2. 背景

OpenMPI [22] や MPICH [21] といった主要な MPI ライブラリでは，低遅延かつ高スループットな実装として，共有メモリを用いた MPI ノード内通信をサポートしている．本章では，共有メモリを用いた MPI ノード内通信の実装，およびその問題点について述べる．主要な MPI ライブラリの 1 つである OpenMPI の実装を例にして説明するが，他の MPI ライブラリでも実装方式に大きな差はない．

2.1 共有メモリを用いた MPI ノード内通信の実装

OpenMPI の構造はモジュール化されており，通信アルゴリズムやメッセージの送受信方式を柔軟に追加・変更することができる．OpenMPI では，Byte Transfer Layer（BTL）の sm BTL コンポーネントにおいて，共有メモリによる MPI ノード内通信をサポートしている．

MPI 通信を実行するためには，送信リクエスト等の通信制御情報と通信メッセージそのもの（ペイロード）を MPI プロセス間で送受信する必要がある．sm BTL をはじめとする共有メモリを用いた MPI ノード内通信の実装では，通信を行う MPI プロセスの双方が利用可能な共有メモリを作成し，両プロセスのアドレス空間にマッピングする．マッピングした共有メモリにデータの読み書きを行うことで，異なるアドレス空間で動作する MPI プロセスどうしが通信制御情報と通信メッセージの送受信を行う．

2.1.1 通信制御情報の送受信

sm BTL では，送信リクエスト等の通信制御情報の送受信を共有メモリ上に確保した通信キューを用いて行う．通信制御情報を送信する MPI プロセスが，通信先のプロセスのキューに通信制御情報をコピーすることで，通信制御情報の送受信が実行される．通信キューは共有メモリ上に存在するため，通信を行う MPI プロセスの双方からデータを読み書きすることができる．

各 MPI プロセスの通信キューを格納する共有メモリは，

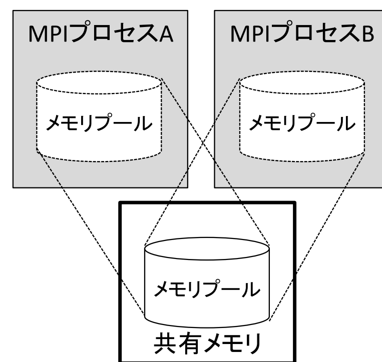

図 1　sm BTL におけるメモリプール
Fig. 1　Memory pool of sm BTL.

MPI ライブラリの初期化時に作成されたメモリプールから取得する．MPI プログラムの実行時，最初に起動した MPI プロセスがある程度大きなサイズの共有メモリを作成する．他の MPI プロセスは，**図 1** に示すように，その共有メモリを自身のアドレス空間にマッピングし，メモリプールとして利用する．共有メモリの作成およびマッピングは `mmap` を用いて行う．メモリプールから取得されたメモリは，共有メモリとして同一ノード内の全 MPI プロセスのアドレス空間にマッピングされるため，どの MPI プロセスからでもデータの読み書きを行うことができる．

仮想記憶をサポートする近年の OS カーネルはメモリをページという単位に区切り管理している．デマンドページングをサポートする OS カーネルでは，`mmap` を実行した時点では，実際にはメモリの割当ては行わない．プロセスが `mmap` を実行したアドレス領域に実際にアクセスした際に，当該アドレスに対してメモリページの割当てを行う．よって，ある程度大きなサイズのメモリプールを作成したとしても，作成した時点でメモリ消費量が大きく増加することはない．メモリプールのうち，実際に MPI ノード内通信のために使用された領域にのみ，メモリが共有メモリとして割り当てられ，その分だけメモリ消費量が大きくなる．

2.1.2 Eager 通信

MPI 通信では，通信制御情報のほかに，通信メッセージそのものを送受信する必要がある．主要な MPI ライブラリでは，メッセージの送受信において，Eager 通信と Rendezvous 通信の 2 つの通信プロトコルをサポートしている．Eager 通信は，通信を行うプロセス間で同期なしに通信を行う方式である．一方，Rendezvous 通信は送信プロセスと受信プロセス双方の準備が完了した時点で通信を開始する方式である．まず，sm BTL の Eager 通信の実装について述べる．

Eager 通信では，送信メッセージを通信バッファにバッファリングすることで，非同期にメッセージの送受信を行う．sm BTL の Eager 通信では，まず送信プロセスが前項で述べたメモリプールから Eager 通信用のバッファを取得する．この Eager 通信用バッファは共有メモリ上に存在す

© 2015 Information Processing Society of Japan

るため，送信プロセスと受信プロセスの双方がデータの読み書きを行うことができる．次に送信プロセスは，送信するメッセージを自身の送信バッファから Eager 通信用バッファにコピーする．そして，メッセージをコピーした当該バッファのアドレス（実際にはメモリプール上でのオフセット値）を受信プロセスに通知して送信処理を完了する．受信プロセスは受信バッファの準備が完了した時点で，当該 Eager 通信用バッファからメッセージを自身の受信バッファにコピーして受信処理を完了する．Eager 通信では，受信プロセスの受信準備が完了するのを待つことなく，送信側が通信を開始，完了することができるので，通信遅延が小さくなる．

一度使用された Eager 通信用バッファは，以降の Eager 通信で再利用される．sm BTL では，再利用する Eager 通信用バッファを free list によって管理している．free list を参照する際の競合を避けるため，各 MPI プロセスは，個別の free list で，自身の利用する Eager 通信用バッファを管理する．Eager 通信の際，送信プロセスは自身の free list から使用していない Eager 通信用バッファを取得して Eager 通信を行う．受信プロセスはメッセージのコピーを終えた後，当該 Eager 通信用バッファを送信プロセスの free list に返却する処理を行う．free list に利用できるバッファがなかった場合，送信プロセスはメモリプールから新たに Eager 通信用バッファを取得して，free list に追加する．集団通信のように短期間に多数の通信を行う処理が実行された場合，一度に多数の Eager 通信用バッファがメモリプールから取得され，メモリ消費量が大きくなる．

メッセージサイズが大きくなると次項で述べる理由により，Rendezvous 通信の方が Eager 通信よりも通信遅延が小さくなる．sm BTL では Eager 通信と Rendezvous 通信を切り替えるメッセージサイズを eager limit というパラメータで調整することができる．sm BTL では eager limit のデフォルト値は 4 KB である．送信するメッセージと付随するメタデータの合計サイズが eager limit 以下の場合，そのメッセージは Eager 通信で送信される．Eager 通信用バッファの再利用を可能にするため，Eager 通信用バッファのサイズは eager limit に設定されたサイズとなる．

2.1.3　Rendezvous 通信

Rendezvous 通信では，送信プロセスと受信プロセス双方の準備が完了した時点でメッセージの送受信を行う．通信を行う双方のプロセスは個別のアドレス空間で動作するため，送信プロセスの送信バッファから受信プロセスの受信バッファに直接メッセージをコピーすることはできない．よって，メッセージの送受信は，共有メモリ上に確保した中間バッファを経由して行う．以下に sm BTL における Rendezvous 通信の実装について述べる．

sm BTL の Rendezvous 通信では，まず送信プロセスがメモリプールから Rendezvous 通信用の中間バッファを取得する．この中間バッファは共有メモリ上に存在するため，送信プロセスと受信プロセスの双方からデータを読み書きすることができる．次に送信プロセスは，送信するメッセージを固定サイズのブロックに分割する．そして分割したブロックを図 2 に示すように，先頭から順に中間バッファにコピーしていく．受信プロセスは，送信プロセスからのコピーが完了したブロックを順々に，中間バッファから自身の受信バッファにコピーしていく．このように，中間バッファを経由したパイプライン転送を行うことで，送信プロセス側のメモリコピー（送信バッファから中間バッファ）と受信プロセス側のメモリコピー（中間バッファから受信バッファ）をオーバラップさせることが可能になる．このため，図 3 に示すように，単純にメッセージを中間バッファを経由してコピーするよりも通信遅延が小さくなる．

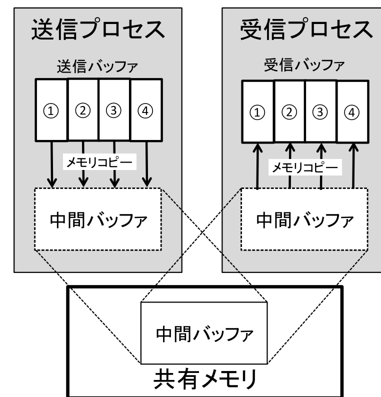

図 2　中間バッファによる sm BTL の Rendezvous 通信
Fig. 2　Rendezvous communication of sm BTL.

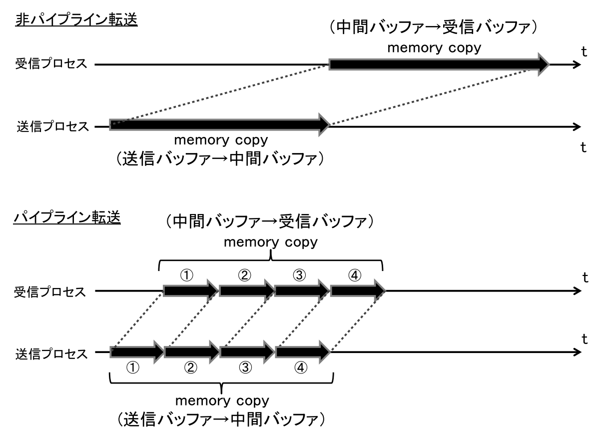

図 3　パイプライン転送
Fig. 3　Pipeline transmission.

Eager 通信では，送信プロセスと受信プロセスが非同期に通信を行う．よって，Rendezvous 通信のように，パイプライン転送でメッセージを送受信することができない．メッセージを送受信する際に，メモリコピーをオーバラップさせることができないので，ある程度メッセージサイズが大きくなると，Eager 通信の方が Rendezvous 通信より

© 2015 Information Processing Society of Japan

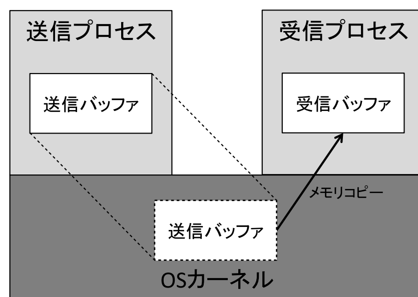

図 4　OS カーネルの支援による sm BTL の Rendezvous 通信
Fig. 4　Rendezvous communication with OS kernel support.

も通信遅延が大きくなる．

sm BTL では，パイプライン転送のブロックサイズのデフォルト値は 32 KB である．パイプライン転送の途中で中間バッファに空きがなくなった場合，新たにメモリプールから共有メモリを取得し，中間バッファの拡張を行う．一度確保された中間バッファは以降の通信で再利用される．中間バッファを利用する際の競合を避けるため，各 MPI プロセスは個別の中間バッファをそれぞれ保持する．

sm BTL では共有メモリ上の中間バッファを経由したパイプライン転送のほかに，OS カーネルの支援による Rendezvous 通信もサポートしている．この方式では，まず送信プロセスが送信バッファのアドレスを受信プロセスに通知する．受信プロセスはシステムコールを実行し，通信メッセージのコピーを OS カーネルに依頼する．この際，送信バッファのアドレスをシステムコールの引数として OS カーネルに通知する．依頼を受けた OS カーネルは，送信バッファが格納されているメモリを OS カーネルが使用しているアドレス空間にマッピングする．そして，マッピングした領域からメッセージを受信プロセスの受信バッファにコピーする．メッセージのコピー終了後，マッピングしたメモリはアンマッピングされる．この方式では，**図 4** に示すように，一度のメモリコピーでメッセージの送受信を終えることができるが，通信のたびにシステムコールが発生してしまう．OS カーネルの支援によるメッセージの送受信をサポートするための Linux カーネルモジュールとして，KNEM [10] や LiMIC [16] がリリースされており，sm BTL では，KNEM を用いた Rendezvous 通信をサポートしている．また，MPICH においても OpenMPI と同様に，KNEM を用いた Rendezvous 通信をサポートしている．

2.2　共有メモリによる MPI ノード内通信の問題点

本節では，共有メモリによる MPI ノード内通信の問題点について述べる．前節で説明した OpenMPI の sm BTL の実装を前提として問題点を述べるが，他の MPI ライブラリの MPI ノード内通信の実装と sm BTL の実装に大きな差はなく，ここで述べる問題点は共有メモリを用いた MPI ノード内通信の実装全般に共通する問題となる．

2.2.1　Rendezvous 通信の通信遅延増加

Rendezvous 通信において，共有メモリを用いた MPI ノード内通信の実装では，共有メモリ上の中間バッファを経由したメモリコピーでメッセージを転送するか，OS カーネルの支援によるメモリコピーでメッセージを転送していた．これらの方式では，通信の際に以下で述べるオーバヘッドが生じ，通信遅延が増加してしまう．

共有メモリ上の中間バッファを経由したメモリコピーによってメッセージを転送する場合，メッセージを固定サイズのブロックに分割し，パイプライン転送する．よって，メッセージサイズが大きくなると，多数のメモリコピーを実行する必要がある．メモリコピー用の関数を呼び出す回数が増え，そのオーバヘッドにより通信遅延が増加してしまう．ブロックサイズを大きくすることでメモリコピーの回数を減らすことができるが，その場合，パイプライン転送の段数が少なくなるため，パイプライン転送の効果が小さくなってしまう．

OS カーネルの支援によるメモリコピーでメッセージを転送する場合は，一度のメモリコピーで通信を完了することができる．しかし，通信のたびにシステムコールを実行する必要があり，そのオーバヘッドにより通信遅延が増加してしまう．

2.2.2　ページテーブルの肥大化によるメモリ消費量増加

すでに述べたとおり，仮想記憶をサポートする近年の OS では，メモリをページという単位に区切り管理している．各プロセスがどのメモリページを自身のアドレス空間のどの位置にマッピングしているかという情報は，ページテーブルという OS カーネル内の管理テーブルに記録される．CPU はこのページテーブルを参照し，現在実行しているプロセスが使用しているメモリのアドレスを物理メモリ上のアドレスに変換する．

共有メモリを用いた MPI ノード内通信では，通信を行う双方の MPI プロセスのアドレス空間に共有メモリをマッピングし，マッピングした共有メモリを経由してデータの送受信を行っていた．MPI プロセスがマッピングする共有メモリが増加する場合，マッピング情報を記録するページテーブル内のエントリ（ページテーブルエントリ）の数もあわせて増加するためにページテーブルが肥大化し，ページテーブルによるメモリ消費量が増加する．共有メモリを用いた MPI ノード内通信では，以下のデータを格納する共有メモリを，各 MPI プロセスが自身のアドレス空間にマッピングする必要がある．

- 通信先の通信キュー
- 通信先の Eager 通信用バッファ
- 通信先の Rendezvous 通信用の中間バッファ

同一ノード内で動作する N 個の MPI プロセスが全対全の通信を行う場合，N 個のプロセスが $(N-1)$ 個の通信先を持つことになる．N 個のプロセスが，$(N-1)$ 個の通信

© 2015 Information Processing Society of Japan

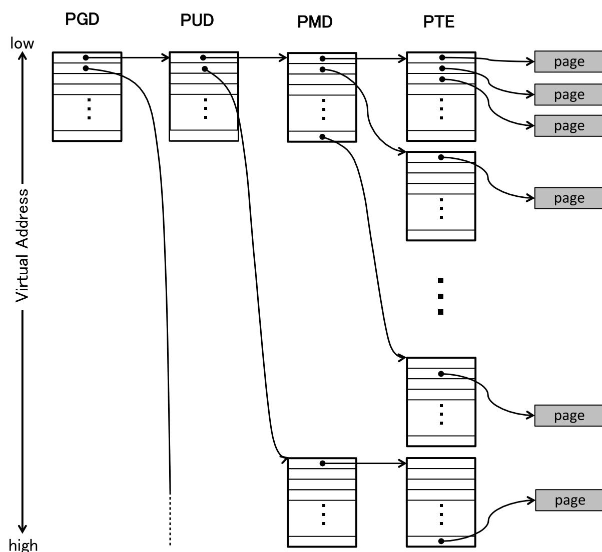

図 5　x86_64 アーキテクチャのページテーブル
Fig. 5　x86_64 page table architecture.

先が使用している共有メモリをマッピングするので，システム全体としては，N^2 のオーダのメモリマッピングが生成される．よって，システム全体のページテーブルによるメモリ消費量も N^2 のオーダで増加する．1 ノードあたりのコア数および MPI プロセス数は今後も増加していくことが予想されるため，N^2 のオーダで増加するページテーブルのメモリ消費量は，メニーコア環境では大きな問題となる．

ページテーブルがフラットな構造をとる場合，メモリページをマッピングしていないアドレス領域についても，ページテーブルエントリを持つ必要があり，ページテーブルサイズは必要以上に大きくなってしまう．そこで多くの CPU アーキテクチャでは，階層構造のページテーブルをサポートしている．たとえば，x86_64 アーキテクチャのページテーブルは，図 **5** に示すとおり，4 段階（PGD/PUD/PMD/PTE）の階層構造となっている．各階層のページテーブルは 512 のエントリを持っており，256 TB のアドレス空間を取り扱うことができる．4 階層目のページテーブルである PTE は，1 つで最高 512 のメモリページをマッピングすることができる．連続したアドレス領域にメモリページをマッピングする場合は，必要な PTE の数は少なくなるが，メモリページをマッピングすべきアドレスが点在する場合は，必要な PTE の数が多くなる．ページサイズが 4 KB のとき，2 MB のメモリをマッピングするには合計で 512 のエントリが必要となる．メモリページをマッピングするアドレス領域が連続している場合，1 つの PTE が 512 のエントリを持つため，必要な PTE の数は最低 1 つでよい．しかし，メモリページをマッピングするアドレスが点在している場合，最悪のケースでは 1 つの PTE が 1 つのメモリページをマッピングすることになり，最大で 512 の PTE が必要となる．PTE のサイズは 4 KB なので，この場合は 2 MB のメモリをマッピングするために 2 MB のメモリがページテーブルのために消費されてしまう．

同一ノード内で動作する 100 個の MPI プロセスが全対全の通信を行うとき，1 つの通信先についてマッピングしなければならない共有メモリのサイズが 2 MB だとする．この場合，システム全体で必要な PTE の個数は最低で 9,900（$1 \times 99 \times 100$）個，最高で 5,068,800（$512 \times 99 \times 100$）個となる．これをメモリ消費量に換算すると，最低で約 39 MB，最高で約 19.3 GB ものメモリが消費されてしまう．

MPI ライブラリ自体のメモリ消費量を削減するための研究 [1], [11] はすでに行われている．しかし，MPI ノード内通信によるページテーブルサイズの増加に着目した研究は，まだ行われていない．

3. 提案

前章で述べたとおり，共有メモリを用いた MPI ノード内通信には，Rendezvous 通信の通信遅延増加とページテーブルサイズの肥大化によるメモリ消費量の増加という問題があった．この通知遅延の増加とメモリ消費量の増加はアドレス空間越しにデータを送受信するためのコストと定義することができる．MPI プロセスがそれぞれ個別のアドレス空間で動作するため，Rendezvous 通信において，送信プロセスの受信バッファから送信プロセスのバッファに直接メッセージをコピーすることができず，通信遅延が増加していた．また，MPI プロセスがそれぞれ個別のアドレス空間で動作するため，通信先のプロセスが使用している共有メモリを，各 MPI プロセスが自身のアドレス空間にマッピングしてデータの送受信を行う必要があり，ページテーブルサイズの肥大化によるメモリ消費量増加を招いていた．

もし，MPI プロセスが同一アドレス空間で動作するならば，アドレス空間越しにデータの送受信を行う必要がなくなり，MPI ノード内通信から，これらのコストを排除することができるといえる．そこで本研究では，MPI プロセスを同一アドレス空間で実行することで，MPI ノード内通信からアドレス空間越しにデータを送受信するためのコストを排除することを提案する．アドレス空間越しにデータを送受信するためのコストを排除することで，よりメニーコア環境に適した，高速かつメモリ消費量の少ない MPI ノード内通信を実現する．本章では，MPI プロセスを同一アドレス空間で実行することを可能にする PVAS タスクモデルについて説明し，PVAS タスクモデルを利用した MPI ノード内通信の実装について述べる．

3.1　PVAS タスクモデルの概要

PVAS タスクモデルは，並列処理を行うノード内のプロセスを同一アドレス空間で動作させることを可能にする．本節では PVAS タスクモデルの概要について述べる．

3.1.1　アドレス空間レイアウト

図 **6** は，既存のタスクモデルと PVAS タスクモデルのア

© 2015 Information Processing Society of Japan

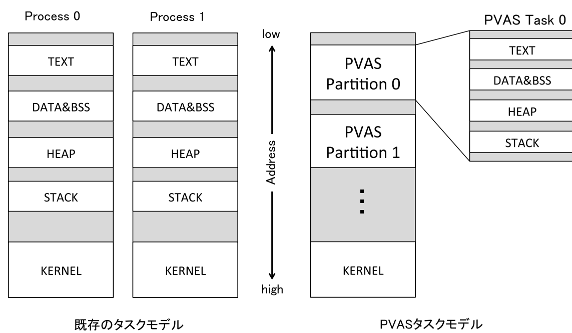

図 6　アドレス空間レイアウト
Fig. 6　Address space layout.

ドレス空間レイアウトを示している．図に示すように，既存のタスクモデルでは，各プロセスが個別のアドレス空間で動作する．TEXT/BSS/DATA/HEAP/STACK といったメモリセグメントは個別のアドレス空間にマッピングされる．

それに対し，PVAS タスクモデルでは，複数のプロセスが同一アドレス空間で動作することを可能にする．PVAS タスクモデルでは，1 つのアドレス空間を PVAS パーティションと呼ぶ領域に分割し，同一アドレス空間で動作させる各プロセスに割り当てる．PVAS タスクモデル上で実行されるプロセスを，通常のプロセスと区別するため，PVAS タスクと呼ぶ．各 PVAS タスクは TEXT/BSS/HEAP/STACK といったメモリセグメントを，各自個別の PVAS パーティション内にマッピングする．通常のプロセスと同様に，TEXT セグメントのメモリは同一バイナリを実行する PVAS タスク間で共有される仕様となっている．

各 PVAS タスクは，自身の PVAS パーティションを自身が利用可能なアドレス領域として認識する．各 PVAS タスクは，`mmap` や `munmap` といった仮想メモリ操作を，自身の PVAS パーティション内のアドレス領域を対象に実行することはできるが，他の PVAS タスクに割り当てられた PVAS パーティション内のアドレス領域を対象にしては実行することができない仕様となっている．

同一アドレス空間内で動作可能な PVAS タスクの数は，PVAS パーティションのサイズに依存する．x86_64 アーキテクチャでは，256 TB のアドレス空間を扱うことができる．この場合，PVAS パーティションのサイズが 64 GB のときは，最大 4,096 の PVAS タスクが，同一アドレス空間で動作することが可能である．

3.1.2　PVAS タスク間の通信

既存のタスクモデルでは，各プロセスが異なるアドレス空間で動作するため，各プロセスが個別のページテーブルでメモリのマッピング情報を管理する．よって，他のプロセスが使用しているメモリに直接アクセスすることができず，通信遅延の増加やメモリ消費量の増加といった，アドレス空間越しにデータを送受信するためのコストが，ノード内通信の際に発生する．

対して，同一アドレス空間で動作する PVAS タスク群は，同一ページテーブルを使用してメモリのマッピング情報を管理する．よって，他の PVAS タスクが使用しているメモリに load/store 命令で直接アクセスすることが可能になり，アドレス空間越しにデータを送受信するためのコストを払うことなく，ノード内通信を実行することができる．

同一ページテーブルを使用するため，ページテーブルの更新時には，排他制御を行う必要がある．排他制御によるページテーブルの更新速度低下を回避するため，排他制御の粒度を細分化する最適化を，PVAS タスクモデルでは行っている．ページテーブルの更新方法については，文献 [28] に詳しい．

3.1.3　プログラムの実行

PVAS タスクと通常のプロセスの違いは，他のプロセスとアドレス空間を共有しているか否かのみである．PVAS タスクは，通常のプロセスと同様，独自のメモリセグメント（TEXT/BSS/HEAP/STACK），ファイルディスクリプタ，プロセス ID，シグナルハンドラ等を持つ．よって，ソースコードへの改変なしに，既存のプログラムを PVAS タスクとして実行することができる．

PVAS タスクモデルの機能をユーザプログラムが利用するために，3 つシステムコールが用意されている．`pvas_create` は，PVAS タスクが動作するアドレス空間を作成するために用いられる．このシステムコールは，作成したアドレス空間の識別子を，ユーザプログラムに返す．`pvas_spawn` は引数で指定されたプログラムを PVAS タスクとして実行する．PVAS タスクが実行されるアドレス空間は，`pvas_create` が返す識別子によって指定することができる．`pvas_destroy` は，不要になったアドレス空間を削除するために用いられる．削除するアドレス空間は，`pvas_create` が返す識別子によって指定する．

Intel がリリースする Xeon Phi 用の Linux カーネル [14] に上記のシステムコールを追加し，PVAS タスクモデルを実装した．実装の詳細については，文献 [28] に詳しい．

3.2　PVAS タスクモデルを用いた MPI ノード内通信

MPI プログラムを PVAS タスクとして実行することで，PVAS タスクとして動作する MPI プロセスを起動することができる．PVAS タスクとして同一アドレス空間で動作する MPI プロセスどうしは，互いのメモリにアクセスすることができる．PVAS タスクモデルを用いた MPI ノード内通信を OpenMPI に実装した．本節では，OpenMPI への実装の詳細について述べる．

3.2.1　プロセスマネージャ

MPI ライブラリは，MPI プロセスの生成と管理を行うプロセスマネージャと通信 API を提供する通信ライブラリによって構成される．通常，OpenMPI のプロセスマネー

© 2015 Information Processing Society of Japan

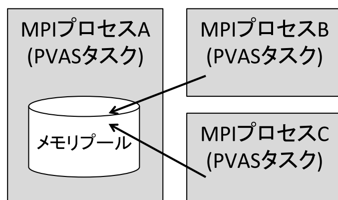

図 7　PVAS BTL におけるメモリプール
Fig. 7　Memory pool of PVAS BTL.

ジャは，fork によって，ユーザが指定した数だけプロセスを生成する．生成されたプロセスは，ユーザに指定された MPI プログラムを execve によって実行し，MPI プロセスとなる．

対して，PVAS タスクモデルを用いた MPI ノード内通信をサポートする OpenMPI のプロセスマネージャは，まず，pvas_create によって，PVAS タスクを生成するアドレス空間を作成する．そして，pvas_spawn を呼び出して，ユーザが指定した数だけ当該アドレス空間に PVAS タスクを生成する．生成された PVAS タスクは，pvas_spawn の引数で指定された MPI プログラムを実行し，MPI プロセスとなる．PVAS タスクとして動作する全 MPI プロセスの実行が終了したら，プロセスマネージャは，pvas_destroy を呼び出し，不要になったアドレス空間を削除する．

3.2.2　PVAS BTL

PVAS タスクモデルを用いた MPI ノード内通信をサポートする BTL コンポーネントを，sm BTL をベースに実装した．このコンポーネントを PVAS BTL とする．

まず，PVAS BTL における通信制御情報の送受信と Eager 通信によるメッセージの送受信について述べる．PVAS BTL では，**図 7** に示すように，通信制御情報を送受信するための通信キューと Eager 通信用のバッファを取得するメモリプールを，最初に起動した MPI プロセスが malloc によって作成する．並列処理を実行する各 MPI プロセスは，このメモリプールから，自身が用いる通信キューと Eager 通信用バッファを取得する．PVAS BTL では，並列処理を行う MPI プロセス群が同一アドレス空間内の PVAS タスクとして動作している．MPI プロセスどうしは，互いのメモリにアクセスすることができるので，どの MPI プロセスも，このメモリプールからメモリを取得することができる．また，メモリプールから取得したメモリにはどの MPI プロセスからでもアクセス可能である．よって，このメモリプールから取得した通信キューと Eager 通信用バッファを用い，sm BTL と同様の方法で，通信制御情報の送受信と Eager 通信によるメッセージの送受信を行うことができる．メモリプールの作成方法以外は sm BTL の実装を流用するため，通信制御情報の送受信と Eager 通信によるメッセージの送受信の性能は，sm BTL と同等となる．

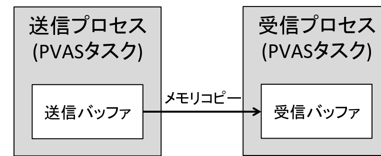

図 8　PVAS BTL における Rendezvous 通信
Fig. 8　Rendezvous communication of PVAS BTL.

次に，Rendezvous 通信によるメッセージの送受信について述べる．PVAS BTL では，並列処理を行う MPI プロセスどうしが互いのメモリにアクセスすることができる．よって，PVAS BTL における Rendezvous 通信では，受信プロセスが，送信プロセスの送信バッファから自身の受信バッファにメッセージを直接コピーすることで通信を行う．送信プロセスが送信リクエストとともに送信バッファのアドレスを受信プロセスに通知し，受信プロセスは通知されたアドレスから，メッセージを受信バッファにコピーする．**図 8** に示すように，一度のメモリコピーで通信を完了することができる．2 章で述べたように，共有メモリを用いた MPI ノード内通信の実装では，メモリコピーの回数増加によるオーバヘッド，またはシステムコールの実行によるオーバヘッドで通信遅延が増加していた．PVAS BTL の Rendezvous 通信では，これらのオーバヘッドは発生しないため，より高速なメッセージの送受信を実現することができる．

共有メモリを用いた MPI ノード内通信では，通信先のプロセスが使用している共有メモリを通信元のプロセスが自身のアドレス空間にマッピングして，MPI ノード内通信を実行するために必要なデータの送受信を行っていた．よって，共有メモリのマッピングによりページテーブルが肥大化し，メモリ消費量が増加していた．PVAS BTL では，並列処理を行う MPI プロセスどうしが，互いのメモリにアクセスすることができるため，共有メモリを用いなくても，MPI ノード内通信の実行に必要なデータの送受信を行うことができる．したがって，共有メモリのマッピングによるページテーブルの肥大化を回避し，その分のメモリ消費量を削減することができる．

4.　評価

PVAS を用いた MPI ノード内通信の性能とメモリ消費量を，ベンチマークソフトによって評価した．評価環境を **表 2** に示す．

4.1　性能評価

4.1.1　Intel MPI Benchmarks

まず，PVAS BTL と sm BTL の通信性能を，Intel MPI Benchmarks（IMB）を用いて比較した．IMB には，MPI 通信の性能を評価するための種々のベンチマークが含まれている．本評価では，MPI_Send/Recv を用いて ping-pong

© 2015 Information Processing Society of Japan

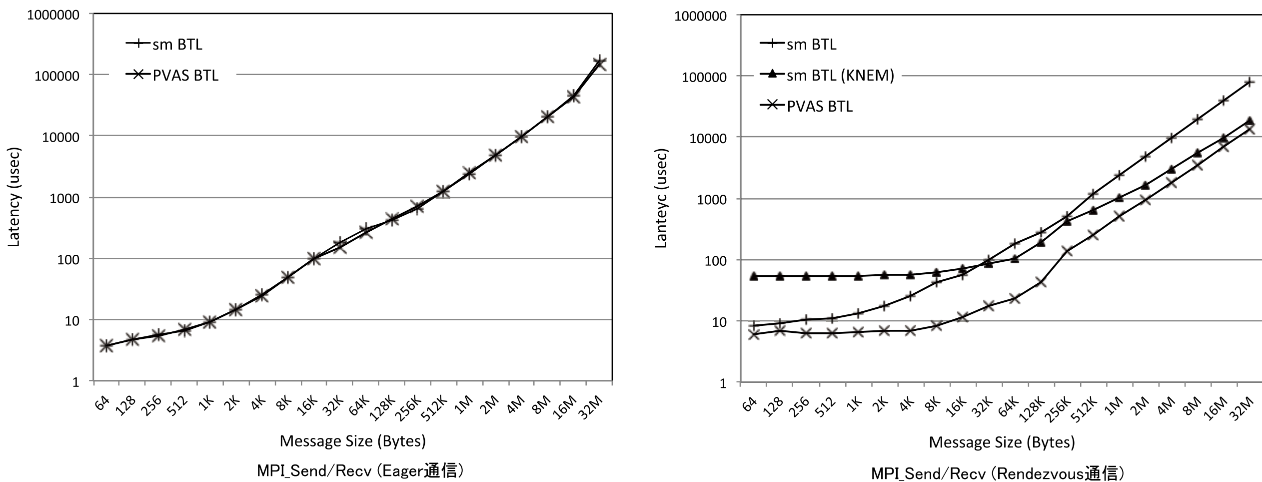

図 9 Point-to-Point 通信の通信遅延

Fig. 9 Latency for Point-to-Point communication.

表 2 評価環境

Table 2 Evaluation environment.

プロセッサ	Intel Xeon Phi coprocessor 5110P ・物理コア数 60，論理コア数 240 ・32 KB L1 キャッシュ，512 KB L2 キャッシュ ・8 GB メインメモリ
OS カーネル	Linux カーネル (MPSS 3.1.2) + PVAS
MPI	OpenMPI version 1.8 + PVAS BTL

通信を行うベンチマークによって Point-to-Point 通信の性能を測定した．また，`MPI_Bcast` によるブロードキャスト通信を行うベンチマークと，`MPI_Allreduce` によるリダクション通信を行うベンチマークによって，集団通信の性能を測定した．集団通信の測定を行う際の MPI プロセス数は 240 プロセスとした．測定は，すべてのメッセージの送受信を Eager 通信で行う場合と Rendezvous 通信によって行う場合について行った．sm BTL の Rendezvous 通信においては，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う場合と，KNEM カーネルモジュールを用いて OS カーネルの支援によりメッセージの送受信を行う場合の 2 通りを測定した．共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う際のブロックサイズは，sm BTL のデフォルト値である 32 KB とした．

Point-to-Point 通信の通信遅延の測定結果を**図 9** に示す．Rendezvous 通信において，KNEM カーネルモジュールを用いて OS カーネルの支援によるメッセージの送受信を行う場合の sm BTL は，sm BTL (KNEM) と記述した．

Eager 通信においては，PVAS BTL と sm BTL は，ほぼ同等の通信性能を示している．3 章で述べたとおり，PVAS BTL の Eager 通信は sm BTL の実装を流用している．Eager 通信用バッファを格納するメモリが共有メモリか通常のメモリかという違いしかないため，通信性能はほぼ同等となる．

Rendezvous 通信においては，PVAS BTL が最も通信遅延が小さい．sm BTL の Rendezvous 通信では，共有メモリ上の中間バッファを経由してメッセージのパイプライン転送を行う．メッセージサイズがパイプライン転送のブロックサイズである 32 KB より小さい場合は，送信プロセス側のメモリコピーと受信プロセス側のメモリコピーのオーバラップを行うことができず，一度のメモリコピーで通信が終わる PVAS BTL よりも通信遅延が大きくなる．メッセージサイズがブロックサイズより大きい場合は，パイプライン転送により，送信プロセス側のメモリコピーと受信プロセス側のメモリコピーをオーバラップさせることが可能になるが，メモリコピーの実行回数自体は増加する．メモリコピー用の関数を呼び出す回数が増え，そのオーバヘッドにより，PVAS BTL よりも通信遅延が大きくなる．KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合は，PVAS BTL と同様に一度のメモリコピーで通信が完了するが，通信のたびにシステムコールが実行されるため，そのオーバヘッドにより，通信遅延が PVAS BTL よりも大きくなる．

sm BTL において，共有メモリ上の中間バッファを経由してパイプライン転送を行う場合と KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合を比較すると，メッセージサイズが小さい場合はシステムコール実行のオーバヘッドにより，OS カーネルの支援によるメッセージの送受信の方が，通信遅延が大きくなる．メッセージサイズが大きくなると，メモリコピーの回数が増加し，そのオーバヘッドがシステムコール実行のオーバヘッドを上回り，中間バッファを経由したパイプライン転送の方が通信遅延が大きくなる．

図 10 に集団通信の通信遅延の測定結果を示す．集団通信を行う場合はプロセス数が増加し，各 MPI プロセスが持つ通信バッファによってシステム全体のメモリの消費量が大きくなる．表 2 の測定環境において，ベンチマークを実行

© 2015 Information Processing Society of Japan

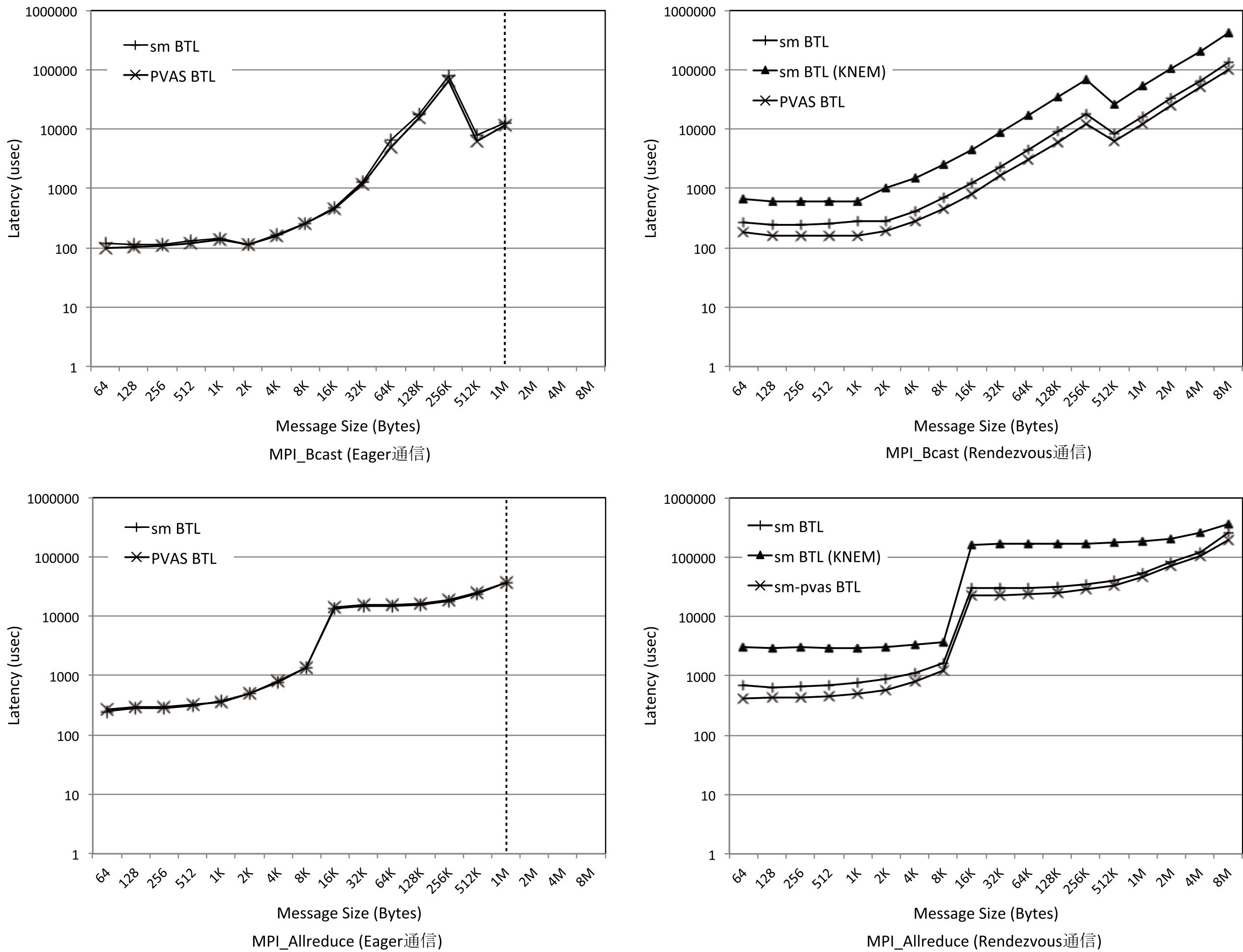

図 10　集団通信の通信遅延

Fig. 10　Latency for collective communication.

可能な最大メッセージサイズは Rendezvous 通信では 8 MB となった．また，集団通信を Eager 通信で行うと一度の通信で多数の Eager 通信用バッファが必要となり，さらにメモリを消費する．このため，Eager 通信においてベンチマークを実行可能な最大メッセージサイズは 1 MB となった．

Point-to-Point 通信のときと同様の理由で，Eager 通信は，PVAS BTL と sm BTL で，ほぼ同等の通信性能となった．Rendezvous 通信においては，Point-to-Point 通信のときと同様の理由で，PVAS BTL が最も高い通信性能を示している．集団通信では，メッセージサイズが大きくなっても，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合の方が，共有メモリ上の中間バッファを用いてパイプライン転送する場合よりも，通信遅延が大きくなっている．OpenMPI の MPI_Bcast，MPI_Allreduce の実装では，通信のバンド幅を改善するため，図 11 に示すように，メッセージを細かいチャンクに分割して，各プロセスが協調して並列に通信を行う通信アルゴリズムとなっている．実際に各 MPI プロセスが送受信するメッセージサイズは小さくなるため，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う方が，通信遅延が大きくなってしまう．

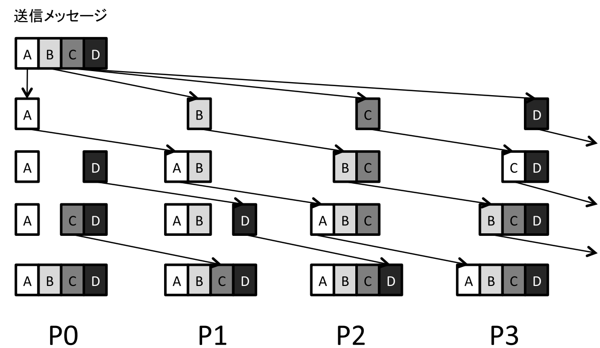

図 11　MPI_Bcast/Allreduce の通信アルゴリズム

Fig. 11　Algorithm for MPI_Bcast/Allreduce.

4.1.2　NAS Parallel Benchmarks

次に，NAS Parallel Benchmarks（NPB）を用いて，PVAS BTL と sm BTL の性能を評価した．NPB には，MPI を用いた並列計算の処理能力を評価するベンチマークが含まれている．IMB が通信性能のみを評価するベンチマークであるのに対し，NPB は，通信と計算処理を含めた総合的な実行性能の評価を行う．NPB に含まれるベンチマークを

© 2015 Information Processing Society of Japan

表 3　NAS Parallel Benchmarks
Table 3　NAS Parallel Benchmarks.

	実行する並列計算処理
EP	乗算合同法による一様乱数，正規乱数の生成
MG	簡略化されたマルチグリッド法のカーネル
CG	正値対称な大規模疎行列の最小固有値を求めるための共役勾配法
FT	FFT を用いた 3 次元偏微分方程式の解法
IS	大規模整数ソート
LU	Symmetric SOR iteration による CFD アプリケーション
SP	Scalar ADI iteration による CFD アプリケーション
BT	5 × 5 block size ADI iteration による CFD アプリケーション

表 3 に示す．

本評価では，通信が発生しない EP ベンチマークを除く，7 つのベンチマークで性能測定を行った．それぞれのベンチマークに対し，問題サイズが異なる 7 つのクラス（S < W < A < B < C < D < E）が用意されている．性能測定では，標準的な問題サイズとされるクラス A，B，C を用いた．ただし，FT ベンチマークのみ，メモリ不足により，クラス C では測定を行うことができなかった．MG，CG，FT，IS，LU ベンチマークにおいては，プロセス数が 2 の累乗でなければならないという制限があるため，プロセス数を 128 プロセスとしてベンチマークを実行した．SP，BT ベンチマークにおいては，プロセス数が任意の数の累乗でなければならないという制限があるため，プロセス数を 225 プロセスとしてベンチマークを実行した．

測定は，eager limit を sm BTL のデフォルト値である 4 KB に設定して行った場合と，すべてのメッセージの送受信を Rendezvous 通信によって行う場合について行った．sm BTL については，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う場合と，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合の 2 通りを評価した．共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う際のブロックサイズは，sm BTL のデフォルト値である 32 KB とした．

eager limit を 4 KB に設定したときの実行結果を**図 12** に示す．Rendezvous 通信において，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う場合の sm BTL の実行性能を基準とし，相対的な性能差をグラフ中に示した．

ベンチマークの実行時間のうち，通信処理の時間よりも計算処理の割合が大きい場合や，通信の大部分を Eager 通信が占める場合をのぞき，PVAS BTL が他と比べて高い実行性能を示している．これは前節で述べたとおり，PVAS BTL では，他と比べて高速な Rendezvous 通信を実現可能なためである．eager limit を 4 KB に設定したケースでは，最大で約 11%の性能向上を実現することができた（SP ベンチマークのクラス B）．すべての通信が Rendezvous 通信で行われる場合，**図 13** に示すように，性能差はより顕著になる．この場合，最大で約 18%の性能向上を実現することができた（SP ベンチマークのクラス B）．

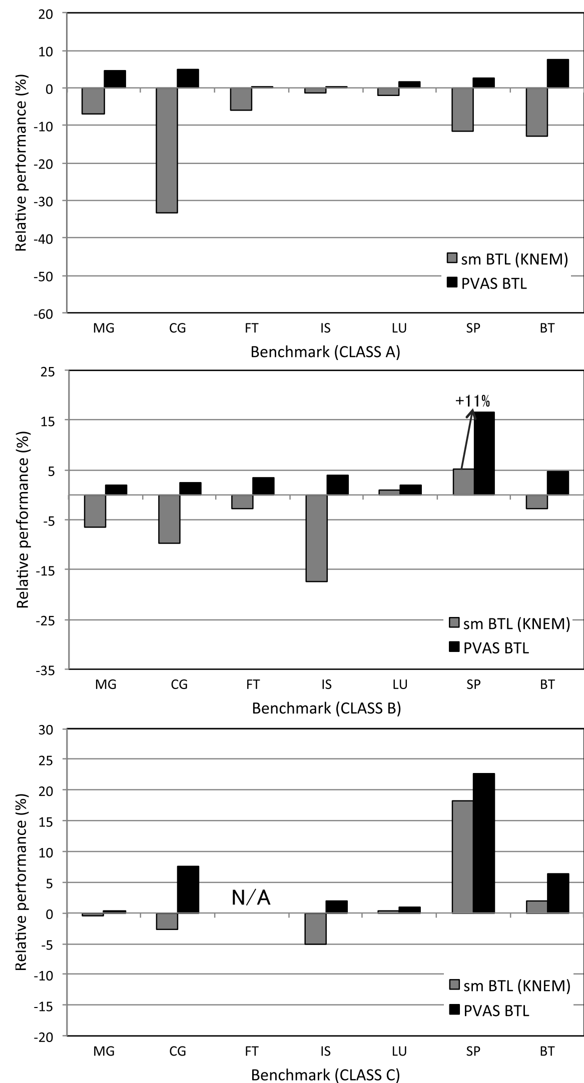

図 12　NPB の実行結果（eager limit = 4 KB）
Fig. 12　NPB results (eager limit = 4 KB).

sm BTL において，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う場合よりも全体的に実行性能が低下している．すべての通信を Rendezvous 通信で行う場合，性能低下の比率はより大きくなる．これは，ほとんどのベンチマークで，送受信するメッセージのサイズが小さいためである．メッセージサイズが小さい場合は，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行った方が，Rendezvous 通信の通信遅延が大きくなる．送受信するメッセージのサイズが大きい SP，BT ベンチマークのクラス C では，逆に実行性能が増加している．

4.2　メモリ消費量の評価

4.2.1　Intel MPI Benchmarks

まず，IMB を用いて，sm BTL と PVAS BTL の MPI

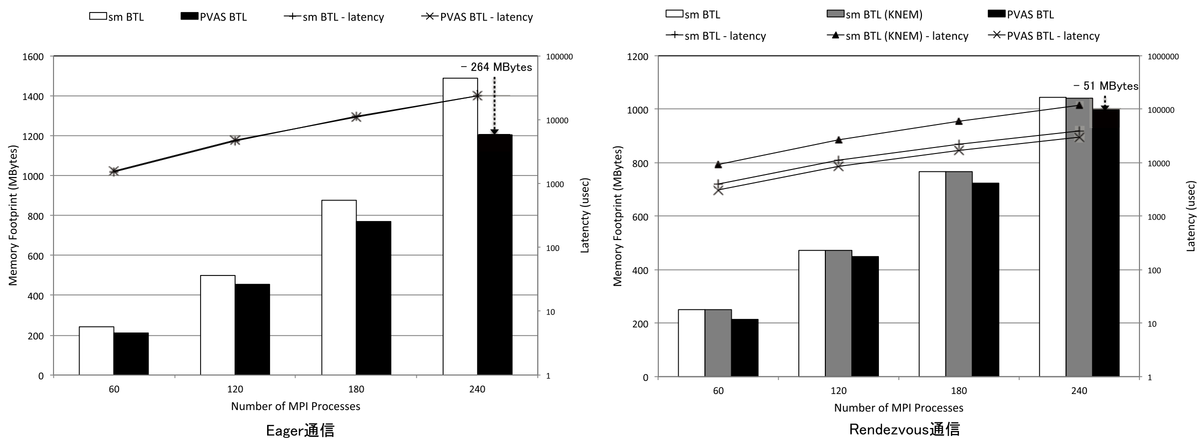

図 14　ベンチマークと MPI ライブラリのメモリ消費量（MPI_Alltoall）
Fig. 14　Memory footprint for executing MPI library (MPI_Alltoall).

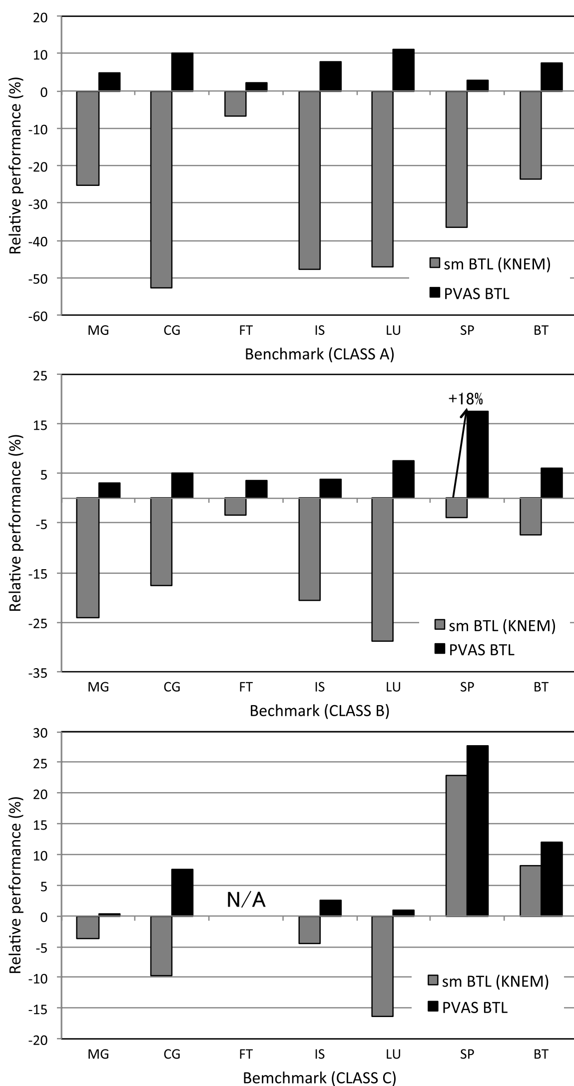

図 13　NPB の実行結果（Rendezvous 通信）
Fig. 13　NPB results (Rendezvous).

ノード内通信のメモリ消費量の比較を行った．測定方法は以下のとおりである．

まず，ベンチマークを実行する前のシステム全体のメモリ消費量を `free` コマンドで測定しておく．次に，ベンチマークを実行する全 MPI プロセスで同期をとり，全 MPI プロセスの処理が `MPI_Finalize` を実行する直前に到達したところで，`free` コマンドによりシステム全体のメモリ消費量を測定する．そして，両測定値の差分をとり，ベンチマークと MPI ライブラリによるメモリ消費量を概算する．sm BTL を用いた場合のメモリ消費量と PVAS BTL を用いた場合のメモリ消費量を比較することで，両 BTL における MPI ノード内通信のメモリ消費量の差を確認することができる．

`MPI_Alltoall` による全対全の通信を行うベンチマークで測定を行った．測定は，メッセージの送受信をすべて Eager 通信で行う場合と，すべて Rendezvous 通信で行う場合の 2 通りで行った．Eager 通信の測定では，eager limit を sm BTL のデフォルト値である 4 KB に設定し，メッセージサイズを 2 KB に固定することで，メッセージの送受信をすべて Eager 通信で行うようにした．Rendezvous 通信の測定では，メッセージサイズを eager limit のデフォルト値より大きい 16 KB に固定してベンチマークを実行した．sm BTL の Rendezvous 通信の測定では，共有メモリ上の中間バッファを経由してメッセージの送受信を行う場合と，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合の双方で測定を行った．共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う際のブロックサイズは，sm BTL のデフォルト値である 32 KB とした．ベンチマークでの通信の実行回数は 1,000 とした．

プロセス数が 60 から 240 のときにおける，ベンチマークと MPI ライブラリのメモリ消費量を図 **14** に示す．また，グラフ中には，ベンチマークを実行したときの通信性能も示した．PVAS BTL と sm BTL を比較すると，PVAS BTL は sm BTL と同等以上の通信性能を維持しながら，メモリ消費量は sm BTL と比べて少なくなっている．240 プロセスで Eager 通信を行う場合は約 264 MB，Rendezvous 通信を行う場合は約 51 MB，メモリ消費量が低減されてい

© 2015 Information Processing Society of Japan

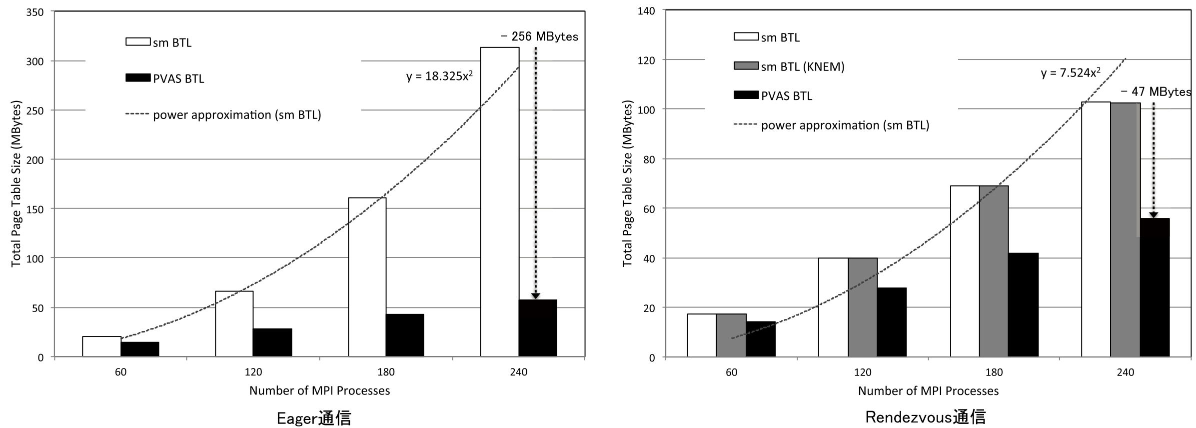

図 15　ベンチマークの実行によるシステム全体のページテーブルサイズの増加量（MPI_Alltoall）
Fig. 15　Memory footprint for page tables.

る．このメモリ消費量の差は，共有メモリのマッピングによるページテーブルサイズの増加に起因すると考えられる．

これを確認するため，MPI ノード内通信によるページテーブルサイズの増加量を比較した．ベンチマークを実行する前のシステム全体のページテーブルサイズと，ベンチマークを実行する全 MPI プロセスが `MPI_Finalize` の直前で停止したときのシステム全体のページテーブルサイズを，*/proc/meminfo* ファイルを参照して確認し，その差分をとることで，ベンチマークの実行によって増加したシステム全体のページテーブルサイズを概算する．sm BTL を用いた場合のページテーブルサイズの増加量と PVAS BTL を用いた場合のページテーブルサイズの増加量を比較することで，MPI ノード内通信を実行したときのページテーブルサイズの増加量の差を確認することができる．

プロセス数が 60 から 240 のときにおいて，ベンチマークの実行により増加したページテーブルサイズを**図 15** に示す．240 プロセスで Eager 通信を行う場合，PVAS BTL と sm BTL を比べると，ページテーブルサイズの増加量は，PVAS BTL の方が約 256 MB 少なくなっている．これは，図 14 のグラフで示したメモリ消費量の差にほぼあてはまる．Rendezvous 通信のときも同様に，ページテーブルサイズの増加量の差が，図 14 のグラフで示したメモリ消費量の差にほぼあてはまっている．以上の結果から，PVAS BTL では，共有メモリのマッピングによるページテーブルサイズの増加が抑制され，MPI ノード内通信によるメモリ消費量が削減されたことが分かる．2 章で述べたとおり，共有メモリによる MPI ノード内通信を用いて全対全の通信を行う場合，プロセス数の 2 乗のオーダに従って，ページテーブルによるメモリ消費量が増加する．これは，図 15 のグラフ中に示した累乗近似曲線からも確認することができる．今後も 1 ノードあたりのコア数と MPI プロセス数は増加していくことが予想されるため，ページテーブルによるメモリ消費量の増加は，メニーコア環境では，より大きな問題になると考えられる．

集団通信を Eager 通信で行う場合，一度に多数の Eager 通信用バッファが使用されるため，Rendezvous 通信の場合よりもメモリ消費量が大きくなっている．また，sm BTL では，通信先の Eager 通信用バッファが格納されている共有メモリを各 MPI プロセスが，自身のアドレス空間にマッピングするため，ページテーブルサイズの増加量も，Rendezvous 通信の場合と比べて大きくなっている．

sm BTL の Rendezvous 通信において，共有メモリ上の中間バッファを経由したパイプライン転送を行う場合と KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合では，メモリ消費量に大きな差はない．sm BTL は，共有メモリ上の中間バッファを経由したパイプライン転送を行うか否かにかかわらず，MPI ライブラリの初期化時に，パイプライン転送用の中間バッファを作成してしまう．よって，この 2 つの方式で，メモリ消費量に大きな差が出ることはない．

4.2.2　NAS Parallel Benchmarks

次に，NPB を用いて MPI ノード内通信のメモリ消費量の比較を行った．測定方法については，IMB によるメモリ消費量の測定方法と同じである．

ベンチマークには，`MPI_Alltoall` を実行する IS ベンチマークのクラス A を用いた．プロセス数は 2 の累乗である必要があるため，128 プロセスとした．測定は，eager limit を sm BTL のデフォルト値である 4 KB に設定した場合と，全通信を Rendezvous 通信で行う場合の 2 通りで行った．sm BTL の Rendezvous 通信において，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行う際のブロックサイズは，デフォルト値である 32 KB とした．eager limit を 4 KB に設定したときのベンチマークのメモリ消費量（MPI ライブラリのメモリ消費量を含む）とベンチマークの実行によって増加したページテーブルサイズを**表 4** に示す．また，すべての通信を Rendezvous 通信で行った場合のベンチマークのメモリ消費量（MPI ライブラリのメモリ消費量を含む）とベンチマークの実行によっ

© 2015 Information Processing Society of Japan

表 4　IS ベンチマークでのメモリ消費量（eager limit = 4 KB）
Table 4　Memory footprint for executing IS benchmark (eager limit = 4 KB).

	sm BTL	PVAS BTL
ベンチマークのメモリ消費量（MB）	547	517
増加したページテーブルサイズ（MB）	51	31
実行性能（Mop/s total）	432.6	434.02

表 5　IS ベンチマークでのメモリ消費量（Rendezvous 通信）
Table 5　Memory footprint for executing IS benchmark (Rendezvous).

	sm BTL	PVAS BTL
ベンチマークのメモリ消費量（MB）	524	503
増加したページテーブルサイズ（MB）	41	31
実行性能（Mop/s total）	377.18	406.54

て増加したページテーブルサイズを**表 5** に示す．参考に，4.1.2 項で測定した，ベンチマークの実行性能も併記した．なお，sm BTL の Rendezvous 通信において，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合については，共有メモリ上の中間バッファを用いてパイプライン転送を行う場合と実行結果に有意な差が見られなかったため，記述を省略した．

eager limit を 4 KB に設定した場合，Eager 通信が行われ，Eager 通信用バッファが使用される．よって，すべての通信を Rendezvous 通信で行う場合に比べて，メモリ消費量が大きくなっている．また，通信先の Eager 通信用バッファが格納されている共有メモリを各 MPI プロセスが，自身のアドレス空間にマッピングするため，ページテーブルサイズの増加量も，Rendezvous 通信の場合と比べて大きくなっている．

sm BTL と PVAS BTL を比較すると，eager limit を 4 KB に設定したときのベンチマークのメモリ消費量の差は 30 MB である．このうち 20 MB が，ページテーブルサイズの増加量の差である．残りの 10 MB は，パイプライン転送用の中間バッファによるものと考えられる．PVAS BTL では，共有メモリのマッピングによるページテーブルサイズの増加が抑制され，MPI ノード内通信のメモリ消費量が削減されたことが分かる．すでに述べたとおり，共有メモリを用いた MPI ノード内通信では，プロセス数の 2 乗のオーダで，ページテーブルによるメモリ消費量が増加する．本測定では，IS ベンチマークの仕様により，128 プロセスでしかベンチマークを実行することができなかったが，より多数のプロセスで動作可能なアプリケーションを実行した場合は，ページテーブルサイズの増加量の差はより顕著になる．

IMB では，120 プロセスで Eager 通信の `MPI_Alltoall` を実行したときに，sm BTL と PVAS BTL のページテーブルサイズの増加量の差は約 38 MB（図 15）であった．IS ベンチマークでは，ほぼ同数のプロセスで Eager 通信の `MPI_Alltoall` を実行しているにもかかわらず，ページテーブルサイズの増加量の差は 20 MB にとどまっている．これは各ベンチマークが実行する通信の回数の差に起因すると考えられる．各 MPI プロセスは，Eager 通信に用いる Eager 通信用バッファを free list で管理しており，同じ通信先に毎回同じ Eager 通信用バッファが使用されるとは限らない．よって，通信回数が増えると，通信先の MPI プロセスがアクセスする Eager 通信用バッファの数が増加する．sm BTL の場合は，通信先の MPI プロセスが自身のアドレス空間にマッピングしなければならない共有メモリが増え，ページテーブルサイズが増加してしまう．IMB のベンチマークでは，1,000 回通信を実行していたが，IS ベンチマークでは，10 回しか通信を実行しないため，ページテーブルサイズの増加量が IMB のときと比べて少なくなっている．

5. 関連研究

5.1 MPI のスレッド実装

通常 MPI は，1 プロセスを 1 MPI プロセスとして並列計算処理を実行させるプログラミングモデルになっている．しかし，MPI ライブラリの実装の中には，1 スレッドを 1 MPI プロセスとして並列計算処理を実行させるものも存在する [12], [24], [29]．同一ノード内に存在するスレッドは，同一アドレス空間で動作するため，共有メモリを用いなくても，MPI ノード内通信に必要なデータの送受信を行うことができる．よって，PVAS タスクモデルを用いる場合と同様に，高速かつメモリ消費量の少ない MPI ノード内通信を実現することができる．しかし，1 スレッドを 1 MPI プロセスとしてプログラミングを行う場合，MPI プロセス間でグローバル変数が共有されてしまい，既存の MPI のプログラミングモデルを大きく逸脱してしまうという問題がある．

5.2 メモリ共有方式

5.2.1 XPMEM

共有メモリとは異なるメモリ共有の仕組みとして，XP-MEM [30] があげられる．XPMEM は，Linux のカーネルモジュールで，プロセスが他のプロセスの使用しているメモリを，自身のアドレス空間にマッピングすることを可能にする．共有メモリの場合は，あるプロセスが明示的に共有メモリを作成し，その共有メモリを他のプロセスが自身のアドレス空間にマッピングする．対して XPMEM では，すでに他のプロセスが使用しているメモリを，通信を行いたいプロセスが自身のアドレス空間にマッピングする点が，共有メモリとは異なっている．

通信先のプロセスが使用しているメモリを，通信元のプロセスが自身のアドレス空間にマップすることで，MPI

© 2015 Information Processing Society of Japan

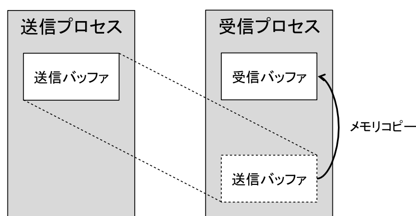

図 16　XPMEM を用いた Rendezvous 通信
Fig. 16　Rendezvous communication with XPMEM.

ノード内通信に必要なデータの送受信を高速に行うことができる．特に Rendezvous 通信においては，送信プロセスの送信バッファを，受信プロセスが自身のアドレス空間にマッピングすることで，1 回のメモリコピーで通信を完了できるようになるため（**図 16** 参照），共有メモリ上の中間バッファを経由してメッセージの送受信を行うよりも通信遅延が小さくなる．また，同じ通信バッファを用いて繰り返し通信を行う場合，メモリのマッピングを行うためのシステムコールを実行するのは，最初の通信のときだけでよい．よってこのケースでは，Rendezvous 通信の性能は，PVAS タスクモデルを用いたときと同等となる．

XPMEM を用いても，プロセス間で共有したいメモリを相互にマッピングしなければならない点は共有メモリを用いたときと同じである．よって，XPMEM を用いて MPI ノード内通信を実装しても，共有メモリを用いたときと同様に，メモリマッピングによりシステム全体のページテーブルサイズが増加し，メモリ消費量が大きくなってしまう．

5.2.2　SMARTMAP

SMARTMAP [3] は，プロセスが他のプロセスのメモリを自身のアドレス空間にマッピングする機能を提供する．SMARTMAP は，x86_64 アーキテクチャのページテーブルの構造を利用して実装されている．x86_64 のページテーブルは，図 5 に示すように，4 段階の階層構造になっている．SMARTMAP を利用するプロセスは，第 1 階層のページテーブルである PGD の 512 エントリのうち，最初のエントリのみを使用することができる．残りのエントリは，自プロセスを含む同一ノード内のプロセスのメモリをマッピングするために用いられる．**図 17** に示すように，同一ノード内の各プロセスの PGD の最初のエントリを，自身の PGD の残り 511 エントリのいずれかにコピーすることで，同一ノード内のプロセスのメモリを自身のアドレス空間にマッピングする．SMARTMAP によって相互にメモリをマッピングするプロセスどうしは，第 2 階層以降のページテーブルを共有するため，共有メモリや XPMEM のように，システム全体のページテーブルサイズが肥大化することはない．よって，PVAS タスクモデルの代わりに SMARTMAP を用いても，本研究の提案する高速かつメモリ消費量の少ない MPI ノード内通信を実現することができる．

SMARTMAP のメモリマッピングの仕組みでは，他のプ

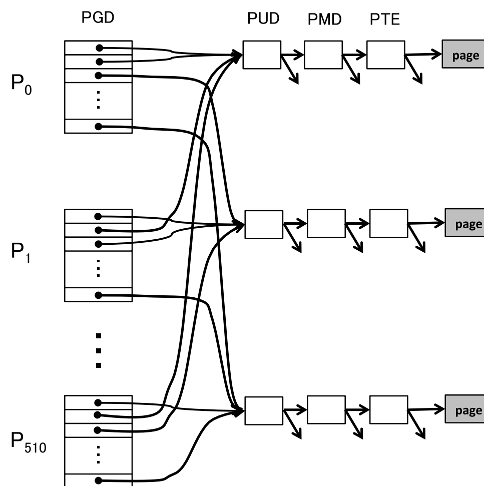

図 17　SMARTMAP によるメモリマッピング
Fig. 17　Memory mapping of SMARTMAP.

ロセスのメモリにアクセスする際は，当該メモリのアドレスに適切なオフセット値を加算する必要がある．メモリアクセスの際に，アクセスするメモリのアドレスが，自プロセスのものか他プロセスのものかを意識する必要がある．一方，PVAS タスクモデルでは，そのような違いを意識する必要はない．

SMARTMAP の実装は x86_64 アーキテクチャのページテーブルの構造に深く依存しているため，ポータビリティが低いという問題がある．また，511 プロセス間でしか，相互にメモリのマッピングを行うことができないという制限がある．

現在 SMARTMAP は米国 Sandia 国立研究所の開発した軽量 OS カーネルである kitten [25] と Catamount [26] に実装されている．これらの OS カーネルはプロセスの起動時に当該プロセスが利用する全メモリの割当てを行う方式をとっており，デマンドページングをサポートしていない．よって，SMARTMAP を Linux のような汎用 OS に移植する際は，SMARTMAP を利用しているプロセスどうしでページテーブル更新の同期をとる仕組みを導入し，デマンドページングに対応させる必要がある．PVAS タスクモデルでは同一アドレス空間で動作するプロセスが同一ページテーブルを使用するため，ページテーブル更新時の同期はスレッドの仕組みを流用することができる．SMARTMAP では，プロセス間でページテーブルを部分的に共有する特殊な構造をとっており，独自の実装が必要になる．

5.3　分散共有メモリ

Memory Channel [9] のような通信アーキテクチャや Cenju [18] のような分散共有メモリアーキテクチャを用いると，リモートノード上のプロセスが使用しているメモリをローカルノード上のプロセスのアドレス空間にマッピングすることができる．リモートノード上のプロセスが使

© 2015 Information Processing Society of Japan

用しているメモリをPVASタスクモデルのアドレス空間にマッピングすることで，リモートノード上のプロセスとローカルノード上のプロセスを同一アドレス空間で動作させることが理論上可能になる．ノードをまたいだグローバルなアドレス空間を構築することが可能になるため，ノード間とノード内の違いを意識せずに，MPIのような通信ライブラリを実装することができるようになる．

6. 議論

6.1 Hybrid MPI

本研究では，ノード間の並列化とノード内の並列化を，ともにMPIで行うことを前提とし，高速かつメモリ消費量が少ないMPIノード内通信を提案した．しかし，ノード間の並列化はMPIで行い，ノード内の並列化はOpenMP [23]やPthreadのようなスレッドを用いた並列化で行うアプリケーションも存在する．ノード内の並列化をスレッドによって行う場合は，MPIノード内通信は発生せず，本研究で提案したMPIノード内通信を用いる意義は失われる．

OpenMPで実装されたNPBとMPIで実装されたNPBの実行性能を比較した結果を図**18**に示す．OpenMPの実行性能を基準とし，相対的な実行性能の差をグラフ中に示した．評価環境は表2に示したものを用い，問題サイズはクラスA，B，Cを用いた．ただし，FTベンチマークについては，メモリ不足によりクラスCを実行することができなかった．MG，CG，FT，IS，LUベンチマークにおいては，プロセスおよびスレッド数を128とし，SP，BTベンチマークにおいては，プロセス数およびスレッド数を225とした．MPIについては，eager limitを4KBに設定した．実行結果が示すとおり，ノード内の並列化をMPIで行うべきかスレッドを用いて行うべきかは，実行する処理や問題サイズ等に依存する．MPIでノード内の並列化を行うべきケースでは，本研究で提案したMPIノード内通信が効果を示す．

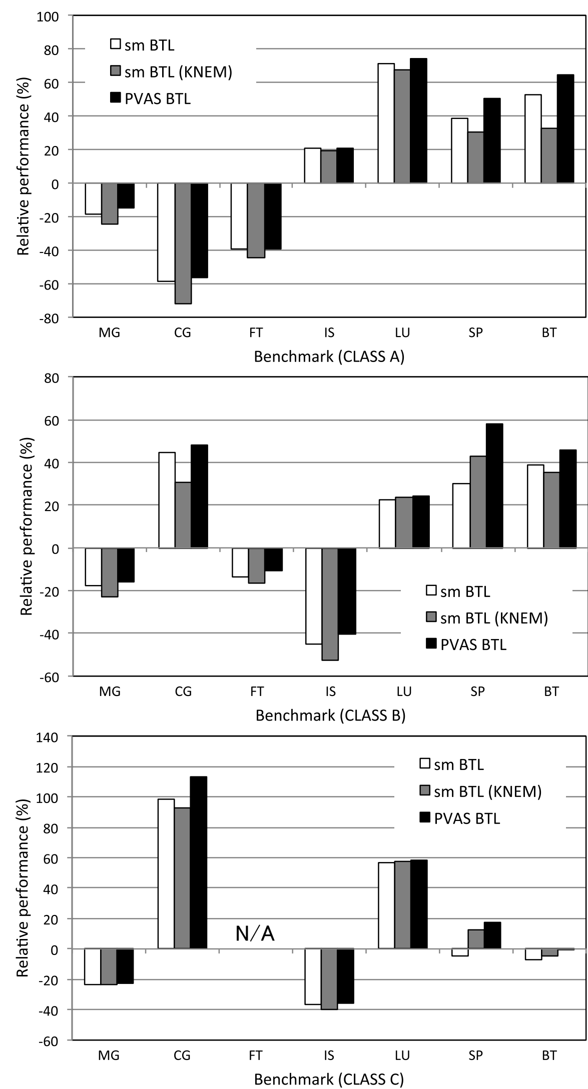

図 **18**　MPIとOpenMPの比較
Fig. 18　MPI vs. OpenMPI.

6.2 Huge Page

メモリマッピングによるページテーブルサイズの肥大化を回避する方法として，Huge pageを用いることが考えられる．Huge pageを用いると，1ページテーブルエントリで，より大きなサイズのメモリをマッピングすることができるため，ページテーブルサイズが小さくなる．PVASタスクモデルとHuge pageの双方を用いてMPIノード内通信を実装することで，ページテーブルサイズの増加によるメモリ消費量をより抑制することができる．

しかし，Huge pageを用いるとメモリの管理単位が大きくなるため，かえってメモリの使用効率が低下してしまう可能性がある．また，Huge pageを用いるとメモリページのマイグレーションが発生する．メモリページのマイグレーションは，システムの性能低下を招いてしまう懸念があるうえ，Remote-DMA（RDMA）との相性が悪いという問題がある．多くのMPIライブラリはinfinibandのRDMA機能を用いたMPIノード間通信をサポートしている．RDMAにより転送されるメモリは，メモリの仮想アドレスと物理アドレスのマッピングが変更されないように，`mlock`によってピンダウンしておく必要があるが，ピンダウンされるメモリに対しては，マイグレーションを実行することができなくなってしまう．

元来Huge pageは，巨大なメモリを効率良く管理するために用いられる機能であるため，コアあたりのメモリサイズが小さくなるメニーコア環境に適しているとはいえない．メニーコア環境でHuge pageを用いる際は，前述した問題が発生しないよう，注意深く実装を行う必要がある．

6.3 PVAS タスクモデルについて

6.3.1 バイナリ形式

PVASタスクは，既存のMPIプログラムをソースコー

© 2015 Information Processing Society of Japan

ドへの改変なしに実行することができる．ただし，MPIプログラムがロードされるアドレスはPVASタスクごとに異なるため，プログラムは位置独立実行形式（Position Independent Executable（PIE））としてビルドする必要がある．PIEとしてビルドされたプログラムでは，シンボルへのアクセスがシンボルテーブルを経由した間接参照になるため，頻繁にグローバル変数にアクセスするようなプログラムでは，非PIEとしてビルドした場合と比べて，性能が低下する可能性がある．

HPCアプリケーションのような大規模な計算データを扱うプログラムでは，メモリアクセスの大半はこの大規模データへのアクセスとなる．メモリアクセスのうち，シンボル解決が必要となるメモリアクセスの比率は相対的に少なくなるため，PIEと非PIEの性能差は一般的なアプリケーションよりも小さくなる．よって，プログラムをPIEとしてビルドしても大きな問題にはならないと考えられる．

これを確認するため，NPBの各ベンチマークをPIEと非PIEでビルドした場合の性能比較を行った．NPBのMG，CG，FT，IS，LU，SP，BTベンチマークを，非PIEとしてビルドして実行した場合と，PIEとしてビルドして実行した場合の実行性能を**図19**に示す．実行結果のグラフ中には，非PIE時の性能を基準とした性能差を併記した．評価環境は表2に示したものを用いた．問題サイズは，クラスB用いた．MG，CG，FT，IS，LUベンチマークにおいては，プロセス数を128プロセスとしてベンチマークを実行した．SP，BTベンチマークにおいては，プロセス数を225プロセスとしてベンチマークを実行した．MPIノード内通信にはsm BTLを用いた．PIEとしてビルドした場合と非PIEとしてビルドした場合の性能差は–0.5%から+1.4%の範囲にとどまっている．また，PIEと非PIEの性能差に明確な相関関係はなく，両者の間で有意な差は見られない．以上の結果から，HPCアプリケーションについては，プログラムをPIEとしてビルドしても大きな問題にはならないといえる．

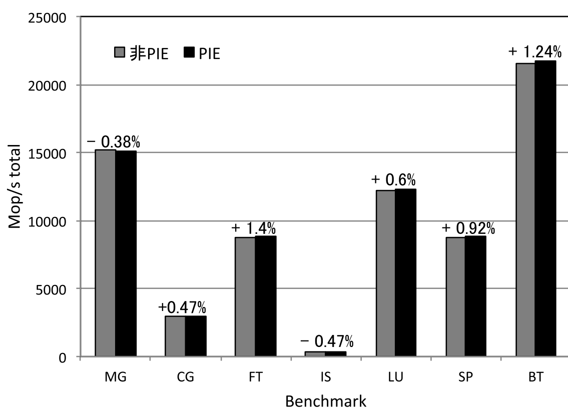

図 19　NPBの実行結果（PIEと非PIE）
Fig. 19　NPB results (PIE and Non-PIE).

6.3.2　PVASのメモリ保護モデル

Opal [4] やMungi [8] のようなSingle Address Space OSがノード上の全プロセスを同一アドレス空間で動作させるのに対して，PVASタスクモデルは同一ノード内に複数のアドレス空間を作成し，それぞれのアドレス空間で複数のプロセス（PVASタスク）を動作させることができる．また，通常のタスクモデルとの共存が可能である．

PVASタスクモデルは，ある並列ジョブが他の並列ジョブのメモリを破壊しないようなメモリ保護モデルになっている．並列ジョブJを構成するPVASタスク群をアドレス空間Aで実行し，並列ジョブKを構成するPVASタスク群をアドレス空間Bで実行することで，並列ジョブJ，Kが，互いの使用するメモリを破壊することを防ぐことができる．しかし，同一ジョブ内のPVASタスク群が，互いのメモリを破壊することを防ぐことはできない．

通常，並列ジョブ内のあるプロセスが異常な動作を起こした場合，HPCシステムのジョブスケジューラは，並列ジョブ内の全プロセスを終了させ，チェックポイントから再始動するか，別の並列ジョブの実行を開始する．あるプロセスが，なんらかの異常な動作を起こした場合，それが同一ジョブ内の他のプロセスのメモリを破壊するものであろうとなかろうと，ジョブスケジューラによって全プロセスが終了させられるのに変わりはない．よって，同一ジョブ内のプロセス間のメモリ保護を行わなくても，大きな問題にはならないと考えられる．

7.　まとめ

近年，HPCシステムを構成するノード1台あたりのコア数は飛躍的に増加してきている．一方で，1コアあたりのメモリ量は減少する傾向にある．PVASタスクモデルを用いると，このようなメニーコア環境で効率的に並列処理を実行することができる．PVASタスクモデルは，並列処理を実行するノード内のプロセスを同一アドレス空間で動作させることを可能にする．同一アドレス空間で動作するプロセスどうしは，通信遅延の増加やメモリ消費量の増加といった，アドレス空間越しにデータを送受信するためのコストを払うことなく，ノード内通信を実行できる．

本研究では，PVASタスクモデルをMPIに適用し，高速かつメモリ消費量の少ない，よりメニーコア環境に適したMPIノード内通信を実現した．PVASタスクモデルを用いてノード内のMPIプロセスを同一アドレス空間で動作させ，アドレス空間越しにデータを送受信するためのコストをMPIノード内通信から排除した．

PVASタスクモデルを用いたMPIノード内通信をNPBによって評価したところ，Rendezvous通信を高速化し，最大で約18%実行性能を改善することができた．また，IMBにより，MPIノード内通信のメモリ消費量を測定したところ，共有メモリのマッピングによるページテーブルサイズ

© 2015 Information Processing Society of Japan

の増加を抑制し，最大で約 264 MB，MPI ノード内通信によるメモリ消費量を低減させることができた．MPI ライブラリ自体のメモリ消費量を削減する研究はすでに行われているが，MPI ノード内通信によるページテーブルサイズの増加に着目した研究は行われておらず，これは本研究の大きな成果といえる．

さらなる MPI ノード内通信の高速化，メモリ消費量の低減が今後の課題となる．また，PVAS タスクモデルを用いたノード内通信を他の並列化モデルに適用し，評価することも今後行っていく予定である．

謝辞　本研究の一部は，科学技術振興機構（JST）の戦略的創造研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」によるものである．

参考文献

[1] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R. and Träff, J.L.: MPI on a Million Processors, *Proc. 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp.20–30, Springer-Verlag, Berlin, Heidelberg (2009).

[2] Berkeley UPC — Unified Parallel C: The UPC Language, available from ⟨http://upc.lbl.gov/⟩.

[3] Brightwell, R., Pedretti, K. and Hudson, T.: SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor, *Proc. 2008 ACM/IEEE Conference on Supercomputing*, Piscataway, NJ, USA (2008) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1413370.1413396⟩.

[4] Chase, J., Levy, H., Baker-Harvey, M. and Lazowska, E.: Opal: A Single Address Space System for 64-Bit Architectures, *Proc. IEEE Workshop on Workstation Operating Systems* (1992).

[5] Bailey, D.H. et al.: The NAS Parallel Benchmarks, *International Journal of Supercomputer Applications*, Vol.5, No.3 (1991).

[6] Deitz, S.J., Chamberlain, B.L. and Hribar, M.B.: Chapel: Cascade High-Productivity Language An Overview of the Chapel Parallel Programming Model, *Cray User Group*, Lugano, Switzerland (2006).

[7] Dongarra, J., Choudhary, A., Kale, S., et al.: The International Exascale Software Project Roadmap, White paper, Argonne National Laboratory (2010).

[8] Gernot, H., Kevin, E., Stephen, R. and Jerry, V.: Mungi: A distributed single address-space operating system, Technical Report UNSW-CSE-TR-9314, School of Computer Science and Engineering, The University of New South Wales (1993).

[9] Gillett, R.B.: Memory Channel Network for PCI, *IEEE Micro*, Vol.16, No.1, pp.12–18 (1996).

[10] Goglin, B. and Moreaud, S.: KNEM: A Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework, *Journal of Parallel and Distributed Computing* (*JPDC*), Vol.73, No.2, pp.176–188 (online), DOI: 10.1016/j.jpdc.2012.09.016 (2013).

[11] Goodell, D., Gropp, W., Zhao, X. and Thakur, R.: Scalable Memory Use in MPI: A Case Study with MPICH2, *Proc. Recent Advances in the Message Passing Interface — 18th European MPI Users' Group Meeting, EuroMPI 2011*, Santorini, Greece, Sep. 18–21, pp.140–149 (2011).

[12] Huang, C., Lawlor, O. and Kalé, L.V.: Adaptive MPI, *Proc. 16th International Workshop on Languages and Compilers for Parallel Computing* (*LCPC 2003*), LNCS 2958, College Station, Texas, pp.306–322 (2003).

[13] Hybrid Memory Cube Consortium: Hybrid Memory Cube Specification 1.1 (2014).

[14] Intel Corporation: Intel Manycore Platform Software Stack, available from ⟨https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss⟩.

[15] Intel Corporation: Intel MPI Benchmarks 4.0, available from ⟨http://software.intel.com/en-us/articles/intel-mpi-benchmarks/⟩.

[16] Jin, H.-W., Sur, S., Chai, L. and Panda, D.K.: LiMIC: Support for High-Performance MPI Intra-node Communication on Linux Cluster, *ICPP*, pp.184–191 (2005).

[17] Jinpil, L. and Mitsuhisa, S.: Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems, *The 39th International Conference on Parallel Processing Workshops*, *ICPPW10* (2010).

[18] Kanoh, Y., Nakamura, M., Hirose, T., Hosomi, T. and Nakata, T.: User-level Network Interface for a Parallel Computer Cenju-4 (Special Issue on Parallel Processing), *IPSJ Journal*, Vol.41, No.5, pp.1379–1389 (2000).

[19] Kemal, E., Vijay, S. and Vivek, S.: X10: Programming for hierarchical parallelism and non-uniform data access, *International Workshop on Language Runtimes, OOPSLA* (2004).

[20] MPI: A Message-Passing Interface Standard Version 3.0, available from ⟨http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf⟩.

[21] MPICH: High-Performance Portable MPI, available from ⟨http://www.mpich.org/⟩.

[22] Open MPI: Open Source High Performance Computing, available from ⟨http://www.open-mpi.org/⟩.

[23] OpenMP: The OpenMP API specification for parallel programming, available from ⟨http://openmp.org/⟩.

[24] Pérache, M., Carribault, P. and Jourdren, H.: MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption, *PVM/MPI*, Ropo, M., Westerholm, J. and Dongarra, J. (Eds.), Lecture Notes in Computer Science, Vol.5759, pp.94–103, Springer (2009).

[25] Sandia National Laboratory: Kitten Lightweight Kernel, available from ⟨https://software.sandia.gov/trac/kitten⟩.

[26] Sandia National Laboratory: Open Catamount, available from ⟨http://www.cs.sandia.gov/˜rbbrigh/OpenCatamount/⟩.

[27] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: PGAS Intra-node Communication towards Many-Core Architecture, *6th Conference on Partitioned Global Address Space Programing Model*, Santa Barbara, California, USA (2012).

[28] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: Proposing A New Task Model towards Many-Core Architecture, *Proc. ACM International Workshop on Manycore Embedded Systems 2013*, Tel-Aviv, Israel, ACM (2013).

[29] Tang, H. and Yang, T.: Optimizing Threaded MPI Execution on SMP Clusters, *Proc. 15th International Conference on Supercomputing*, *ICS '01*, pp.381–392, New

© 2015 Information Processing Society of Japan

York, NY, USA, ACM (online), DOI: 10.1145/377792.377895 (2001).
[30] Woodacre, M., Robb, D., Roe, D. and Feind, K.: The SGI® Altix™ 3000 Global Shared-Memory Architecture.
[31] 島田明男，堀　敦史，石川　裕：新しいタスクモデルによる MPI ノード内通信の高性能化，並列/分散/協調処理に関するサマーワークショップ（SWoPP），新潟 (2014).

島田 明男　（正会員）

2006 年慶應義塾大学理工学部情報工学科卒業．2008 年同大学大学院理工学研究科修士課程修了．同年株式会社日立製作所入社．ファイルストレージの研究開発に従事．2012 年より理化学研究所計算科学研究機構に出向．並列システムソフトウェアの研究開発を行っている．

堀 敦史　（正会員）

1979 年早稲田大学電気工学科卒業．1981 年同大学大学院理工学研究科修士課程修了．同年株式会社三菱総合研究所に入社．1992 年技術研究組合新情報処理開発機構に出向．1999 年東京大学より博士（工学）の学位を取得．2001 年株式会社スイミーソフトウェア設立．2004 年英国 Allinea Software 社に移籍．2008 年東京大学情報基盤センター特任教授．2010 年より理化学研究所計算科学研究機構上級研究員．並列システムソフトウェアの研究に興味を持つ．

石川 裕　（正会員）

1987 年慶應義塾大学大学院工学研究科電気工学専攻博士課程修了．工学博士．同年電子技術総合研究所（現，産業技術総合研究所）入所．1993 年技術研究組合新情報処理開発機構出向．2002 年より東京大学大学院情報理工学系研究科コンピュータ科学専攻教授．2010 年より同大学情報基盤センターセンター長兼務．2014 年より理化学研究所計算科学研究機構プロジェクトリーダ，チームリーダ．

© 2015 Information Processing Society of Japan