

新しいタスクモデルによる MPI ノード内通信の高性能化

島田 明男^{1,a)} 堀 敦史^{1,b)} 石川 裕^{1,2,c)}

概要：エクサスケールの HPC システム構築に向けて、メニーコアアーキテクチャの活用が進んでいる。メニーコアアーキテクチャを採用するシステム上で MPI プログラムを実行する場合、1 ノードあたりの MPI プロセス数の増加とともに、ノード内通信の発生回数が従来よりも増加するため、高速な MPI ノード内通信が求められる。また、メニーコアアーキテクチャのメモリ容量は従来の計算機よりも限られており、MPI ノード内通信によるメモリ消費量は小さいことが望ましい。しかし、MPI プロセスは個別のアドレス空間で実行されるため、アドレス空間越しに通信を行う必要があり、それが通信遅延とメモリ消費量の増加をまねく。そこで本研究では、新たなタスクモデルである Partitioned Virtual Address Space (PVAS) を用いて、MPI プロセスを同一アドレス空間で実行することを提案する。同一アドレス空間で実行される MPI プロセス同士は、アドレス空間を超えるためのコストを発生させることなく通信を行うことが出来る。PVAS を用いた MPI ノード内通信を MPI ランタイムに実装し、NAS Parallel Benchmarks に含まれるベンチマーク群を用いて評価したところ、メモリ消費量の増加をまねくことなく、最大で約 15% の実行性能改善を実現することができた。

1. はじめに

Message Passing Interface (MPI) は並列計算処理のための通信規格であり、ライブラリレベルでの並列化をサポートしている。MPI を用いたプログラムを実行するプロセスを MPI プロセスと呼ぶ。MPI は並列計算を行う各 MPI プロセスに、固有の識別子である MPI rank を付与する。各 MPI プロセスは、MPI rank を用いて通信先の MPI プロセスを指定し、メッセージの送受信を行うことが出来る。現在、多くの並列アプリケーションが MPI を用いて実装されている。

一方、低性能の CPU コアを多数集約することで、高スループットと高い電力効率を両立するメニーコアアーキテクチャの活用が、エクサスケールの HPC システム構築に向けて進んでいる。MPI プログラムを HPC システム上で実行するとき、システムの並列処理能力を余すことなく利用するため、MPI プロセス数とシステム全体の CPU コア数を一致させることが一般的になっている。よって、MPI プログラムをメニーコアアーキテクチャを採用するシステム上で実行する場合、1 ノードあたりの MPI プロセス数が、従来のマルチコアプロセッサを採用するシステム上で

実行する場合よりも増加する。それに伴い MPI ノード内通信 (同一ノード内の MPI プロセス間で行われる通信) の発生回数も従来よりも増加し、MPI ノード内通信の実行速度が、MPI プログラムの実行性能に大きな影響を与えることが予想される。

また、メニーコア環境における MPI ノード内通信には、高速な通信はもちろんのこと、通信によるメモリ消費量の低減も求められる。メニーコアアーキテクチャのメインメモリの容量は、従来のホストコンピュータと比べて少なく、コアあたりのメモリ容量は著しく制限される。例えば、PCI カードデバイスとして実装される Intel Xeon Phi coprocessor は、最大で 61 の物理コアを持ち、244 の論理コアをサポートするが、メインメモリの最大サイズは 16GB である。244 個の論理コアを使って並列計算を行う場合、コアあたりの最大メモリ量は 67MB となる。

MPI のランタイムは、並列計算を行うプロセスを生成し、ユーザが指定した MPI プログラムを実行させ、MPI プロセスとする。同一ノード内に存在する各 MPI プロセスは個別のアドレス空間で動作する。現在、様々な方式のプロセス間通信が提案されているが、それらを用いて MPI ノード内通信を行う場合、通信時にアドレス空間を超えるためのコストが必ず発生する。このコストが、通信遅延の増加やメモリ消費量の増加をまねく。

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

¹ 理化学研究所計算科学研究機構

² 東京大学

a) a-shimada@riken.jp

b) aho@riken.jp

c) ishikawa@is.s.u-tokyo.ac.jp

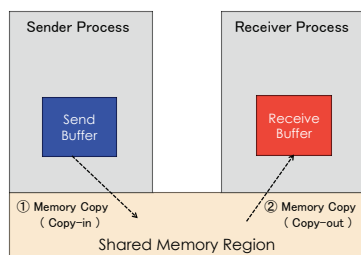


図 1 共有メモリ方式

を超えるためのコストを排除することを提案する．アドレス空間を超え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 MPI ノード内通信を実現する．本研究では，この提案を実現するため，Partitioned Virtual Address Space (PVAS)[12][13] と呼ばれる新たなタスクモデルを用いた．PVAS タスクモデルは，ノード内の MPI プロセス群を同一アドレス空間で実行することを可能にする．同一アドレス空間で実行される MPI プロセス同士は，アドレス空間を超えるためのコストを発生させることなく，MPI ノード内通信を行うことが出来る．PVAS タスクモデルを用いた MPI ノード内通信を，MPI ランタイムの一つである OpenMPI[2] に実装し，NAS Parallel Benchmarks を用いて評価したところ，メモリ消費量の増加をまねくことなく，最大で約 15% の実行性能改善を実現することが出来た．

本論文の構成は以下の通りである．次章にて既存の MPI ノード内通信について述べる．第 3 章にて PVAS タスクモデルの概要について述べ，第 4 章にて PVAS を用いた MPI ノード内通信の OpenMPI への実装について述べる．そして，第 5 章にて NAS Parallel Benchmarks を用いた評価について述べ，第 6 章にて関連研究について述べる．最後に，本研究の結論を述べる．

2. 既存方式

本章では，既存の MPI ノード内通信の方式について説明する．また，それらを利用した場合，通信時にアドレス空間を超えるためのコストがどのように発生するかを述べる．

2.1 共有メモリ方式

共有メモリ方式では，共有メモリを用いたプロセス間通信によって，MPI ノード内通信を実現する．本方式では，通信用の中間バッファとして，送信プロセスと受信プロセスの双方からアクセス可能な共有メモリ領域を確保する．送信プロセスは送信バッファから共有メモリ領域に送信メッセージをコピーし，受信プロセスは共有メモリ領域からメッセージを受信バッファにコピーする．通信ごとに 2 度のメモリコピーが発生するため，通信の遅延が大きくなる．図 1 は，本方式の概要を示している．

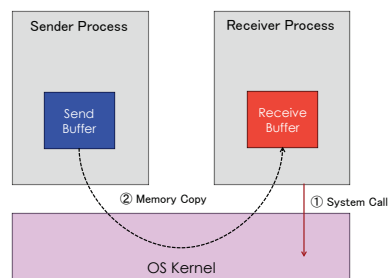


図 2 カーネル支援方式

2.2 カーネル支援方式

カーネル支援方式では，カーネルの支援によるプロセス間通信を利用して，MPI ノード内通信を実現する．本方式では，受信プロセスがメッセージの送受信を OS カーネルに委譲することで，メッセージの送受信を行う．OS カーネルは，送信プロセスと受信プロセス双方のメモリにアクセスすることが出来るため，1 度のメモリコピーでメッセージの送受信を行うことが出来る．ただし，メッセージの送受信を OS カーネルに委譲するために，通信ごとにシステムコールを呼び出す必要があり，それがメッセージ送受信のオーバーヘッドとなる．図 2 は，本方式の概要を示している．

カーネルの支援によるプロセス間通信をサポートするためのカーネルモジュールとして，KNEM[9] や LiMIC[11] が提案されている．また，Linux には，バージョン 3.2 以降，Cross Memory Attach と呼ばれるカーネル支援方式のプロセス間通信が組み込まれている．

2.3 メモリマッピング方式

メモリマッピング方式では，送信プロセスのメモリを受信プロセスのアドレス空間にマッピングすることによってプロセス間通信を行い，MPI ノード内通信を実現する．本方式では，まず，送信プロセスの送信バッファのメモリを，受信プロセスが自身のアドレス空間にマッピングする．受信プロセスは，送信バッファのメモリをマッピングした領域から，メッセージを自身の受信バッファにコピーする．カーネル支援方式同様，1 度のメモリコピーでメッセージの送受信が可能となる．送信バッファのメモリをマッピングするために，システムコールを呼び出す必要があるが，1 度マッピングした送信バッファをアンマッピングせずに維持すれば，同じ送信バッファを用いた繰り返しの通信では，システムコール呼び出しのオーバーヘッドは，最初の通信でしか発生しないメリットがある．図 3 は，本方式の概要を示している．

ただし，メモリマッピングの維持のために，メモリ消費量が大きくなるというデメリットも存在する．メモリマッピングを維持し続ける場合，メモリのマッピング情報を記録するためのページテーブルが肥大化し，OS カーネル内で多くのメモリを消費することになる．例えば， N プロセ

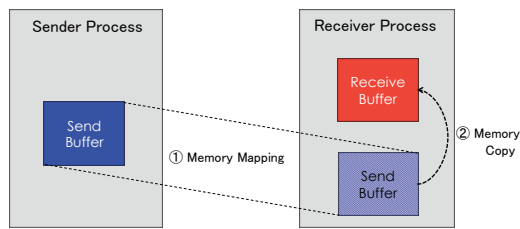


図3 メモリマッピング方式

スが全対全の通信を行う場合、 N^2 のメモリマッピングが作成される。結果として、 N^2 のオーダに従って、メモリ消費量が増加する。メニーコアアーキテクチャ上で MPI プログラムを実行する場合、ノード内のプロセス数は従来よりも飛躍的に増加するため、これは深刻な問題となる。x86 系のアーキテクチャでは、4 KB のページテーブル 1 つにつき、2 MB のメモリをマッピングすることが出来る。よって、244 プロセスが 2 MB の送信バッファを用いて通信する場合は、約 233 MB ものメモリがページテーブルのためだけに消費されてしまう。

メモリマッピング方式のプロセス間通信をサポートするためのカーネルモジュールとして、XPMEM[4] が提案されている。XPMEM は、プロセスが別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と、マッピングしたメモリをアンマッピングする機能を提供する。

3. PVAS タスクモデル

第2章で述べた通り、既存の MPI ノード内通信の方式では、アドレス空間を超えるための余分なコストが通信時に発生し、通信遅延の増加やメモリ消費量の増加をまねいている。逆に言えば、もし通信を行う MPI プロセス同士が同一アドレス空間で動作するならば、余分なコストを発生させることなく、MPI ノード内通信を行うことができると言える。本章では、MPI プロセスを同一アドレス空間で動作することを可能にする PVAS タスクモデルについて説明し、PVAS タスクモデルを用いた MPI ノード内通信について述べる。

3.1 概要

図4は、既存のタスクモデルと PVAS タスクモデルのアドレス空間レイアウトを示している。既存のタスクモデルでは、プロセスが個別のアドレス空間で動作する。TEXT/BSS/HEAP/STACK といったメモリセグメントは、各プロセスの個別のアドレス空間にマッピングされるため、MPI ノード内通信を行う際に、アドレス空間を超えるためのコストが発生する。

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つの仮想アドレス空間を PVAS パーティションと呼ぶサブセグメントに分割し、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。各プロセスは、自身の PVAS パーティションをプライベートなメモリ領域として利用することが出来る。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。PVAS タスクは各自、個別のメモリセグメント (TEXT/BSS/HEAP/STACK) を自身の PVAS パーティション内に保持する。通常のプロセス同様、自身の BSS/HEAP/STACK セグメントをプライベートなデータ領域として利用することができる。TEXT セグメントのメモリは同一バイナリを実行する PVAS タスク同士で共有される。同一アドレス空間で動作する PVAS タスクは、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。各 PVAS タスクの独立性を確保するため、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 パーティションを対象にして実行することは出来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 の仮想アドレス空間を扱うことが出来る。この場合、PVAS パーティションのサイズが 64 GB のときは、4096 の PVAS タスクが、同一アドレス空間で動作することが出来る。

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無しに、既存の MPI プログラムを PVAS タスクとして実行することが出来る。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、位置独立コードとしてビルドしておく必要がある。

PVAS タスクモデルを、Intel Manycore Platform Software Stack (MPSS)[1] に含まれる Xeon Phi 用の Linux カーネルに実装した。実装の詳細については、文献 [12][13] に記述されている。

3.2 システムコール

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()` の返す識別子によって指定することが出来る。`pvas_destroy()` は、不要になったアドレス

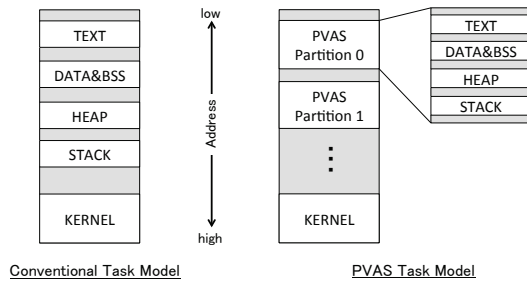


図 4 アドレス空間レイアウト

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()` の返す識別子によって指定する。

3.3 メモリ保護モデル

Opal[10] や Mungi[8] のような Single Address Space OS が、同一ノード内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、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 PVAS タスク群を動作させることが出来る。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破壊を起こさないようなメモリ保護モデルを採用している。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出来る。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出来ない。

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。

3.4 PVAS を用いた MPI ノード内通信

PVAS タスクモデルを用いた MPI ノード内通信を PVAS 方式とする。図 5 は、PVAS 方式の概要を示している。

本方式では、同一アドレス空間に複数の PVAS タスクを生成し、MPI プログラムを実行させる。PVAS タスクとして同一アドレス空間で動作する MPI プロセス同士は、`load/store` 命令によって、互いのメモリにアクセスすることが出来る。よって、PVAS 方式では、1 度のメモリコピーでメッセージの送受信を行うことが可能である。カーネル

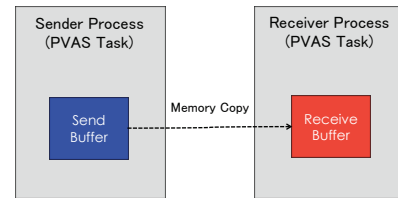


図 5 PVAS 方式

支援方式のように、通信のたびにシステムコールを呼び出す必要もないため、高速な MPI ノード内通信を実現できる。

また、同一アドレス空間で動作する PVAS タスク同士は同一ページテーブルを用いて、メモリマッピングの管理を行うため、メモリマッピング方式のように、他のプロセスのメモリを自身のアドレス空間に改めてマッピングする必要はない。よって、ページテーブルの肥大化は起こらず、MPI ノード内通信が余分なメモリ消費の増加をまねくことはない。

このように PVAS 方式では、アドレス空間を超えるためのコストを払うことなく、MPI プロセス同士がメッセージの送受信を行うことができる。表 1 にて、既存の MPI ノード内通信の方式と PVAS 方式の特徴を比較した。

4. MPI ランタイムの実装

前章で述べた PVAS 方式の MPI ノード内通信を、代表的な MPI ランタイムの一つである OpenMPI に実装した。MPI ランタイムは、MPI プロセスの生成を行うプロセスマネージャと通信ライブラリによって構成される。本章では、PVAS 方式の MPI ノード内通信を実装するために、OpenMPI のプロセスマネージャと通信ライブラリに行った改変について述べる。

4.1 プロセスマネージャ

通常の OpenMPI のプロセスマネージャは、`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。

PVAS 方式の 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、ユーザの指定した数だけ 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。生成した全 PVAS タスクが終了したら、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。

4.2 通信ライブラリ

OpenMPI の通信ライブラリはモジュール化されており、

表 1 既存方式と PVAS を用いた MPI ノード内通信の比較

	共有メモリ方式	カーネル支援方式	メモリマッピング方式	PVAS 方式
メモリコピーの回数	2	1	1	1
システムコールのオーバーヘッド	なし	あり	あり (初回の通信時のみ)	なし
ページテーブルの肥大化	なし	なし	あり	なし

通信アルゴリズムやデータ送受信の方式を柔軟に追加・変更することが出来る．Byte Transfer Layer (BTL) は、メッセージデータの送受信を担当するレイヤで、様々な通信方式をサポートするコンポーネントによって構成される．ノード内通信については、以下のものが主要なコンポーネントとして用意されている．

- **sm BTL:** sm BTL は、共有メモリ方式のノード内通信をサポートする．また、KNEM カーネルモジュールを利用することで、カーネル支援方式のノード内通信を行うことも可能である．KNEM カーネルモジュールが OS カーネルにロードされている場合、このコンポーネントは、OS カーネルの支援によるプロセス間通信を用いて、ノード内通信を行う．カーネル支援方式のノード内通信を行うときは、このコンポーネントを sm-knem BTL と呼ぶ．
- **vader BTL:** vader BTL は、メモリマッピング方式のプロセス間通信を利用して、ノード内通信を行う．このコンポーネントを用いるためには、XPMEM カーネルモジュールを事前にロードしておく必要がある．

sm BTL コンポーネントを改造することで、PVAS 方式の MPI ノード内通信をサポートする BTL コンポーネントを実装した．このコンポーネントを sm-pvas BTL と呼ぶ．このコンポーネントを用いるとき、送信プロセスは、自身の送信バッファのアドレスを、受信プロセスとの間に設けられた共有メモリ領域に書き込むことで通知する．受信プロセスは、共有メモリ領域に書き込まれたアドレスを確認し、自身の受信バッファにメッセージをコピーする．

5. 評価

PVAS 方式の 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．

5.1 評価環境

評価環境を表 2 に示す．Xeon Phi 用 Linux カーネルは、PVAS タスクモデルを実装したものをを用いた．また、OpenMPI は、前章で述べた sm-pvas BTL を実装したものをを用いた．PVAS 方式の MPI ノード内通信を既存方式と比較し、評価するため、sm BTL、sm-knem BTL、vader BTL、sm-pvas BTL のそれぞれを用いた場合について、ベンチマークによる評価を行った．

表 2 評価環境

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	Xeon Phi 用 Linux カーネル ・ Intel MPSS バージョン 3.1.2 ・ Linux 2.6.38.8 ベース
MPI ランタイム	OpenMPI ・ バージョン 1.8

OpenMPI は、通信プロトコルとして、eager 通信と rendezvous 通信をサポートしている．eager 通信の場合、受信側のプロセスが受信の準備が出来ていない場合、送信プロセスはメッセージを受信プロセスの内部バッファにコピーして送信完了とする．受信プロセスは受信準備ができ次第、内部バッファからメッセージを受信バッファにコピーする．受信用の内部バッファによりメモリ消費量が大きくなるが、送信プロセスと受信プロセスとの間で同期を行う必要がないため、通信遅延が小さくなるというメリットがある．一方、rendezvous 通信の場合は、受信プロセスが受信の準備が終えるのを待ってから、送信プロセスがメッセージを送信するため、通信遅延が大きくなる．通信にどちらのプロトコルを用いるかは、メッセージサイズによって切り替わる仕様になっている．本評価では、4 KB 以下のメッセージの送受信には、eager 通信を用い、メッセージサイズが 4 KB より大きい場合は、rendezvous 通信を用いる設定で測定を行った．

5.2 ベンチマーク

評価には、NAS Parallel Benchmarks (NPB)[6] のバージョン 3.3 を用いた．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．NPB に含まれるベンチマークを表 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であるクラス A、B、C をを用いた．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出来なかった．

表 3 NAS Parallel Benchmarks

	内容
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	Synmetric 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	5x5 block size ADI iteration による CFD アプリケーション

また, SP ベンチマークを実行したときのメモリ消費量の測定を行った. このとき, 問題サイズにはクラス A を用いた.

5.3 性能測定結果

性能測定の結果を図 6 に示す. MG, CG, FT, IS, LU ベンチマークについては, プロセス数が 2 の累乗数でなければならないため, 128 プロセスで性能測定を行った. SP, BT ベンチマークについては, プロセス数が平方数でなければならないため, 225 プロセスで性能測定を行った. グラフの縦軸は 1 秒間あたりに全プロセスによって実行されたオペレーション数 (Mop/s total), 横軸はクラスである.

NPB には, MPI ではなく, OpenMP[3] を用いて実装されたベンチマークも含まれている. OpenMP は, 共有メモリ型の並列計算機において, 並列計算を行うための規格である. OpenMP を用いると, 並列化したい計算処理の前に, 並列化を指示するディレクティブを挿入するだけで, 自動的に処理を並列化することができる. OpenMP はスレッドにより計算処理を並列化するため, MPI のように明示的にメッセージ交換を行う必要がないという特徴がある. MPI のみを用いて実装されたピュア型の MPI プログラムだけでなく, ノード内の並列化は OpenMP で行い, ノード間の並列化は MPI で行うハイブリッド型の MPI プログラムも存在する. 本評価では, 比較対象として OpenMP で実装されたベンチマークの測定も行い, 測定結果を *omp* としてグラフ中に示した.

sm-pvas BTL を用いる場合, どのベンチマークにおいても, 他のコンポーネントを用いた場合と比べてほぼ同等か, それ以上の性能を示している. 最も良いケース (SP ベンチマークのクラス B) では, 約 15% の性能改善がみられた. これは, 第 3 章で述べた通り, PVAS を用いた MPI ノード内通信では, 必要なメモリコピーの回数が 1 度だけであることに加え, システムコールのオーバーヘッドも発生しないためである.

sm BTL と sm-knem BTL を比べた場合, 必要なメモリコピーの回数が少ないにも関わらず, sm-knem BTL を用いた場合の方が低い性能を示していることが多い. これ

は, 通信のたびにシステムコールのオーバーヘッドが発生することに加え, sm-knem BTL では OS カーネルがメモリコピーを行っていることに起因する. 本評価で用いた OpenMPI は, インテルコンパイラによってビルドした. よって, sm BTL, vader BTL, sm-pvas BTL を用いる場合は, MPI ノード内通信時に, Xeon Phi プロセッサに最適化されたメモリコピー用の関数が呼び出され, メモリコピーが実行される. それに対し, sm-knem BTL の場合は, Linux カーネルに実装されたメモリコピー用の関数が MPI ノード内通信時に呼び出される. これは, Xeon Phi 向けに最適化されていないため, MPI ノード内通信の性能が他のコンポーネントと比べ, 低下することになる.

メモリマッピング方式をサポートする vader BTL を用いると, 同じ通信バッファを使って繰り返し通信を行うアプリケーションの場合, sm-pvas BTL とほぼ同等の性能を示すことが予想される. そうでない場合でも, sm-knem BTL と同等以上の性能を示すはずである. にもかかわらず本性能測定では, 他のコンポーネントと比べて, 最も低い性能となっている. vader BTL と XPMEM カーネルモジュールの実装を調査し, 性能低下の原因を特定することが今後の課題である.

OpenMP と MPI 比較すると, LU, SP, BT ベンチマークにおいては, MPI が OpenMP とほぼ同等か, それ以上の性能を示した. また, CG ベンチマークにおいても, クラス A 以外では, OpenMP よりも優れた実行性能を示した.

5.4 メモリ消費量の測定

メモリ消費量の測定結果を図 7 に示す. グラフの縦軸は, SP ベンチマークを実行する直前のシステム全体のメモリ消費量とベンチマークを終了する直前のメモリ消費量の差を表している. 横軸はプロセス数である. システム全体のメモリ消費量は, free コマンドによって確認した.

vader BTL を使用した場合, 他のコンポーネントを使用したときと比べて, 最大で約 1.2GB ほどメモリの消費量が大きい. これは, 第 2 章で述べたページテーブルの肥大化が一因になっていると推測される. そこで, ベンチマーク実行前と終了直前の, システム全体のページテーブルによるメモリ消費量の差を測定した. ページテーブルによるメモリの消費量は, `/proc/meminfo` ファイルを参照することで確認した. 測定結果を図 8 に示す. vader BTL を使用した場合, 他のコンポーネントと比べて, 最大で約 180MB ほど, ページテーブルによるメモリ消費量が大きい. また, 累乗近似曲線を見ると, vader BTL を使用する場合は, N^2 のオーダーに従って, ページテーブルによるメモリの消費量が増加していることがわかる.

ページテーブルの肥大化によるメモリ消費量の差を除いても, 他のコンポーネントと比べて vader BTL は最大で約 1.02GB ほどメモリの消費量が大きい. vader BTL と

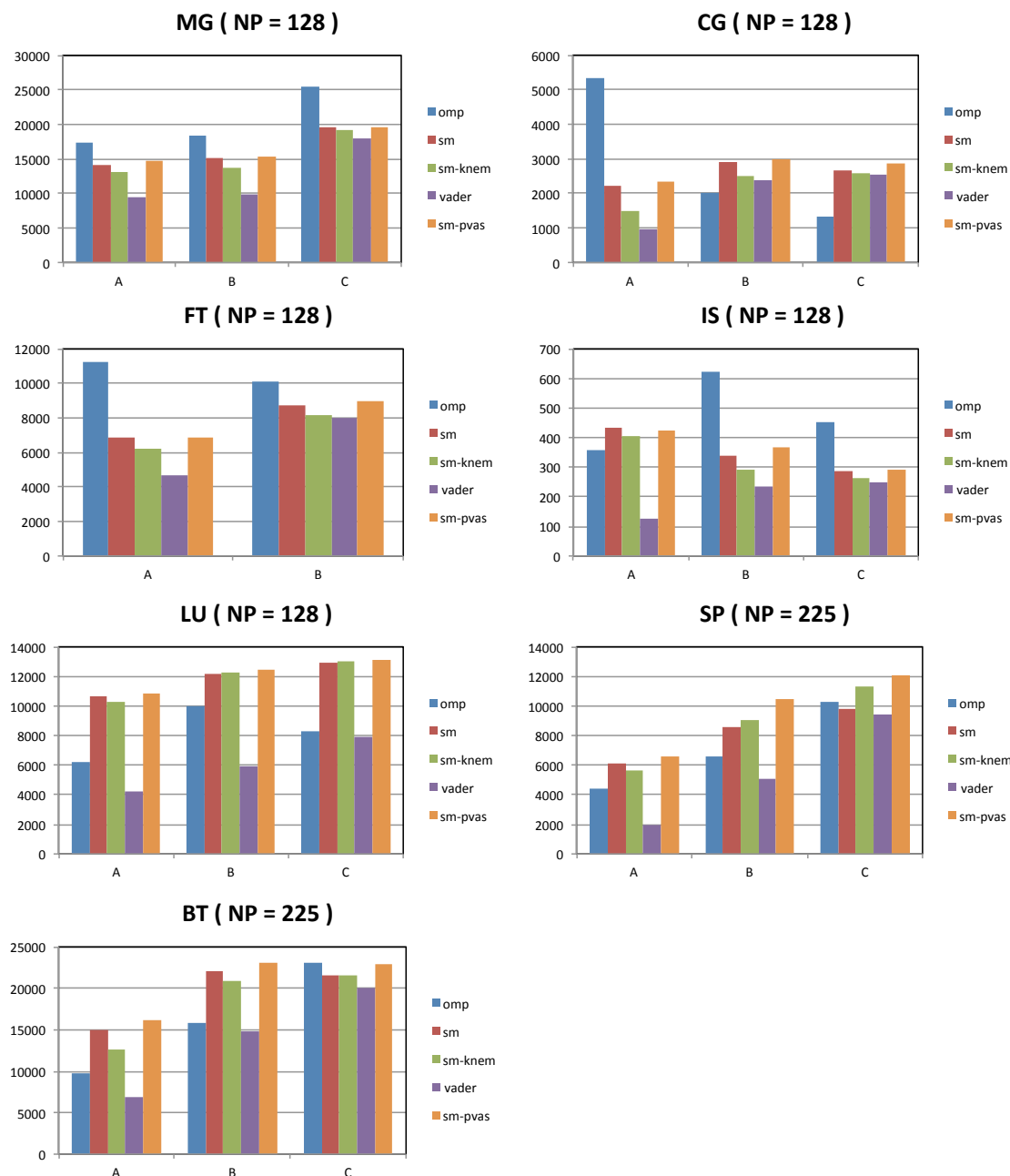


図 6 性能測定結果 (横軸:クラス, 縦軸: Mop/s total)

XPMM カーネルモジュールの実装を調査し, このメモリ消費量増加の原因を調査することが今後の課題である。

6. 関連研究

6.1 Hybrid MPI

Hybrid MPI[7] は, 同一ノード内で動作する全 MPI プロセスがアクセス可能な共有メモリ領域を生成し, メモリプールとする。また, mmap() と sbrk() をフックし, MPI プロセスが動的に確保するメモリ領域を, このメモリプールから割り当てる。通信のバッファが, このメモリプールから割り当てられている場合, MPI ノード内通信を 1 度のメモリコピーで実行することが出来る。また, カー

ネル支援方式のように, 通信時にシステムコールを呼び出す必要もないため, 高速な MPI ノード内通信を実現できる。しかし, この方式では, メモリマッピング方式と同様に, ページテーブルが肥大化してしまう問題がある。

6.2 SMARTMAP

SMARTMAP[5] 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し, 1 度のメモリコピーでのプロセス間通信を実現可能にする。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, 4 段階の階層構造になっている。SMARTMAP を

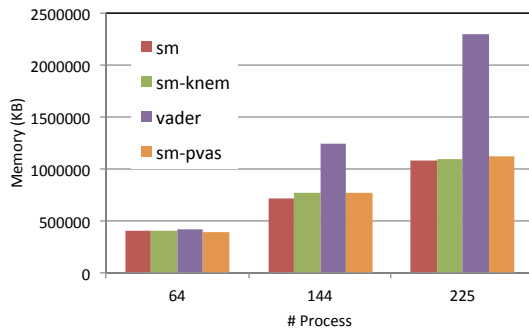


図 7 システム全体のメモリ消費の増加量

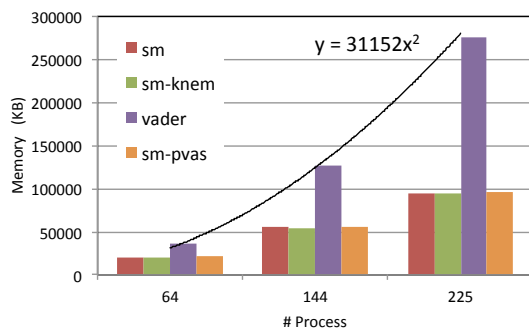


図 8 ページテーブルによるメモリ消費の増加量

利用するプロセスは、第 1 階層のページテーブル (PML4) の 512 エントリの内、最初のエントリのみを使用することが出来る。残りのエントリは、他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。他のプロセスの PML4 の最初のエントリを、自身の PML4 の残り 511 エントリのどれかにコピーすることで、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。第 2 階層以降のページテーブルは、SMARTMAP を利用するプロセス間で共有されるため、メモリマッピング方式のように、ページテーブルが肥大化することはない。しかし、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。

7. 結論

本研究では、MPI ノード内通信において、アドレス空間を超えるためのコストが通信遅延の増加やメモリ消費量の増加をまねくことを示した。そして、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を超えるためのコストを排除することを提案した。本研究で提案した MPI ノード内通信を PVAS タスクモデルを利用して実現し、NAS Parallel Benchmarks を用いて既存の MPI ノード内通信と比較したところ、メモリ消費量の増加をまねくことなく、最大で約 15%、実行性能

が改善した。

本研究では、sm BTL を改良することで PVAS タスクモデルを用いた MPI ノード内通信を実装したが、MPI プロセスを同一アドレス空間で動作させるという PVAS タスクモデルの特徴をより活かした MPI ノード内通信の実装を行うことが今後の課題である。

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「ポストベ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] Intel Manycore Platform Software Stack (MPSS). <https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss>.
- [2] Open MPI: Open Source High Performance Computing. <http://www.open-mpi.org/>.
- [3] The OpenMP API specification for parallel programming. <http://openmp.org/wp/>.
- [4] . XPMEM: Cross-Process Memory Mapping. <http://code.google.com/p/xpmem/>.
- [5] R. Brightwell, K. Pedretti, and T. Hudson. SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, 2008.
- [6] e. a. Davide H.Bailey. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3), 1991.
- [7] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine. Hybrid mpi: Efficient message passing for multi-core systems. In *Proceedings of the 2013 ACM/IEEE conference on Supercomputing*, NY, USA, 2013.
- [8] H. Gernot, E. Kevin, R. Stephen, and V. Jerry. Mungi: A distributed single address-space operating system. In *Technical Report UNSW-CSE-TR-9314, School of Computer Science and Engineering*, November 1993.
- [9] B. Goglin and S. Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing (JPDC)*, 73(2):176–188, Feb. 2013.
- [10] C. J., L. H., L. E., and B.-H. M. Opal: A Single Address Space System for 64-Bit Architectures. In *In Proc. IEEE Workshop on Workstation Operating Systems*, April 1992.
- [11] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *ICPP*, pages 184–191, 2005.
- [12] A. Shimada, B. Geroft, A. Hori, and Y. Ishikawa. PGAS Intra-node Communication towards Many-Core Architecture. In *6th Conference on Partitioned Global Address Space Programming Model*, Santa Barbara, California, USA, 2012.
- [13] A. Shimada, B. Geroft, A. Hori, and Y. Ishikawa. Proposing A New Task Model towards Many-Core Architecture. In *Proceedings of the ACM international workshop on manycore embedded systems 2013*, Tel-Aviv, Israel, June 2013. ACM.