Vol.2012-HPC-135 No.3
2012/8/1

メニーコアOS向け新プロセスモデルの提案

島田 明男[1,a)]　バリ ゲローフィ[1,b)]　堀 敦史[1,c)]　石川 裕[1,2,d)]

概要：エクサスケールのスーパーコンピュータ実現に向けてメニーコアアーキテクチャが注目されている．メニーコア環境では，ノード内の計算処理の並列化が重要となる．本研究では，マルチプロセス型並列アプリケーションにおいて，低コストなプロセス間通信を実現するためのプロセスモデルとして，Partitioned Virtual Address Space (PVAS) を提案する．PVAS を用いることで，プロセス間通信で発生するコストを低減し，従来よりも効率的なノード内並列化を実現することができる．PVAS のプロセス間通信を利用する MPI 通信を実装し，評価したところ，通信のレイテンシとスループットを大幅に改善可能であることが分かり，本提案の有効性を確認することができた．

A Proposal of A New Process Model for Many-Core Architectures

Akio Shimada[1,a)]　Balazs Gerofi[1,b)]　Atsuhi Hori[1,c)]　Yutaka Ishikawa[1,2,d)]

Abstract: Many-core architectures are gathering attention toward Exa-scale computing. In many-core environments, intra-node parallelization is an important issue. In this paper, Partitioned Virtual Address Space (PVAS) that is a new process model for many-core architectures is proposed. PVAS provides the scheme for efficient inter-process communication to multi-process applications so that they can achieve high performance intra-node parallelization. The MPI communication library, which leverages the inter-process communication of PVAS, outperforms the original MPI in the evaluation of the intra-node communication.

1. はじめに

エクサスケールのスーパーコンピュータ実現に向けて，Intel Many-Integrated-Core (MIC) [2] に代表されるメニーコアアーキテクチャが注目されている．メニーコア環境では，1ノードあたりのコア数が従来よりも飛躍的に増加するため，ノード内での計算処理の並列化がより重要となる．本研究では，メニーコア環境において，より効率的なノード内の並列化を実現する方法について議論する．

ノード内で計算処理の並列化を実現する方法として，以下の並列アプリケーションを構築することが挙げられる．

- マルチプロセス型並列アプリケーション
- マルチスレッド型並列アプリケーション

マルチプロセス型並列アプリケーションとは，複数のプロセスによって並列して計算処理を行うアプリケーションを指す．プロセスとは，OS からハードウェア資源の割当を受け，プログラムを実行するインスタンスのことである．プロセスは，1つ以上の実行コンテキストから成り，独自のアドレス空間に，プログラム自身とプログラムの実行に必要なデータを格納する領域を持つ．図1は，マルチプロセス型並列アプリケーションの模式図である．計算に用いるデータを複数のプロセスのデータ格納領域に分散して配置し，各プロセスの実行コンテキストに並列して計算処理を実行させる．各プロセスが独立したアドレス空間で動作するため，あるプロセスの実行コンテキストが他のプロセスの持つ計算データを必要とする場合は，OS が提供するプロセス間通信の機能を利用して，プロセス間でデータの送受信を行う．MPI 通信ライブラリ [3][5] や PGAS 言語 [6]

1　独立行政法人理化学研究所計算科学研究機構
RIKEN AICS
2　東京大学情報基盤センター
Information Technology Center, University of Tokyo
a)　a-shimada@riken.jp
b)　bgerofi@riken.jp
c)　ahori@riken.jp
d)　ishikawa@is.s.u-tokyo.ac.jp

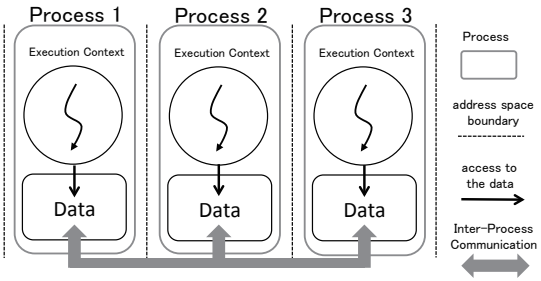

図 1 マルチプロセス型並列アプリケーション

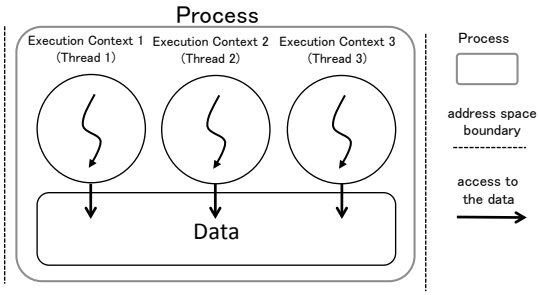

図 2 マルチスレッド型並列アプリケーション

[12] は，同一ノード内のプロセス間でデータの送受信を行うために，プロセス間通信を用いている．

同一ノード内のプロセス間でデータの送受信を行う場合，共有メモリによるプロセス間通信が広く用いられている．通信を実行するプロセス間に双方のプロセスからアクセス可能な共有メモリを設け，送信プロセスが共有メモリに送信データをコピーし，受信プロセスが共有メモリから受信用バッファにデータをコピーすることで，データの送受信を行う．

共有メモリによるデータ通信では，通信時のオーバヘッドが大きいという問題がある．送信プロセスのバッファから受信プロセスのバッファにデータをコピーする際，共有メモリを経由するため，2 度のメモリコピーを実行するための処理時間が必要となる．プロセス間通信を行っている間は，計算処理が停止されるため，通信時のオーバヘッドが計算処理の並列化のボトルネックとなり得る．特に，メニーコア環境では，従来よりも多数のプロセスが並列して動作するため，通信の発生頻度が高くなり，従来よりも通信のオーバヘッドの影響が大きくなると考えられる．

また，共有メモリによるプロセス間通信では，共有メモリ用の領域を確保するために，メモリリソースを消費してしまうという問題がある．特に，メニーコア環境では，多数のプロセスが並列して通信を行うため，共有メモリの確保のために多くのメモリリソースが消費されてしまう．プロセス間通信のために多くのメモリリソースを消費することは，計算処理を行うために割り当てられるメモリリソースが減少することを意味する．一方，メニーコア環境では，コアの数に対するメモリ容量は小さくなる傾向にある．例えば，Intel Knights Ferry では，32 コアに対してメモリの容量は 2GB である [10]．このような環境において，プロセス間通信のために多くのメモリリソースを消費することは，効率的な計算処理の並列化を妨げることになる．

このように，メニーコア環境で動作するマルチプロセス型並列アプリケーションでは，プロセス間通信で発生するオーバヘッドやメモリリソースの消費といったコストが，効率的な並列化の妨げになり得る．より効率的なノード内並列化を実現するためには，従来のプロセス間通信よりも，低オーバヘッドかつメモリリソースの消費量が少ないプロセス間通信を新たに提案する必要がある．

マルチスレッド型並列アプリケーションとは，1 プロセス内にスレッドと呼ばれる複数の実行コンテキストを作成し，各スレッドに並列して計算処理を実行させるアプリケーションを指す．図 2 はマルチスレッド型並列アプリケーションの模式図である．

マルチスレッド型並列アプリケーションでは，全スレッドが同一アドレス空間で動作し，データ格納領域を共有している．共有するデータ格納領域に計算用データを格納し，各スレッドが，その計算用データに各々アクセスして計算処理を実行する．効率的な並列化の妨げになるプロセス間通信を行うことなく，並列して計算処理を実行することができるため，この点では，マルチプロセス型並列アプリケーションよりも，効率的な並列化を実現することが期待できる．

しかし，マルチスレッド型並列アプリケーションでは，各スレッドがデータ格納領域を共有しているため，マルチプロセス型の並列アプリケーションでは起きなかった，格納データへのスレッド間のアクセス競合が，並列アプリケーションのパフォーマンス低下を引き起こしてしまう問題がある．メニーコア環境では，従来よりも多数のスレッドが並列して動作するため，スレッド間のアクセス競合によるパフォーマンス低下が大きくなる．

このように，メニーコア環境で動作するマルチスレッド型並列アプリケーションでは，スレッド間のデータアクセスの競合が効率的な並列化の妨げになり得る．より効率的なノード内並列化を実現するためには，メニーコア環境においても，パフォーマンスの低下が極力起きないような排他制御機構を，プログラマが注意深く実装する必要がある．また，アプリケーションがスレッドセーフなライブラリ関数を頻繁に呼び出す場合，ライブラリ関数内で行われる排他制御がパフォーマンスの低下を引き起こしてしまう可能性があるため，ライブラリ関数の排他制御機構を，メニーコア環境に適したものに，改良する必要がある．

以上で述べたように，メニーコア環境においてより効率的なノード内並列化を実現するためには，並列アプリケーションに応じて，それぞれ以下に示すような課題がある．

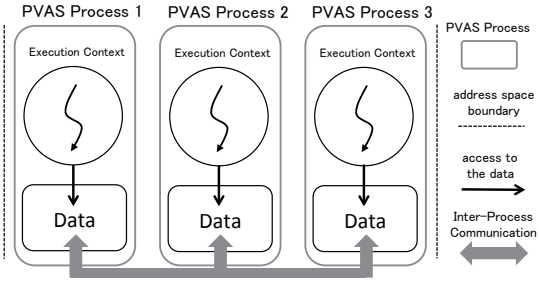

図 3 PVAS を適用したマルチプロセス型並列アプリケーション

- マルチプロセス型並列アプリケーション
  - より低コストなプロセス間通信を実現する．
- マルチスレッド型並列アプリケーション
  - 排他制御機構をプログラマが注意深く実装する．
  - スレッドセーフなライブラリ関数の排他制御機構を，メニーコア環境に適したものに改良する．

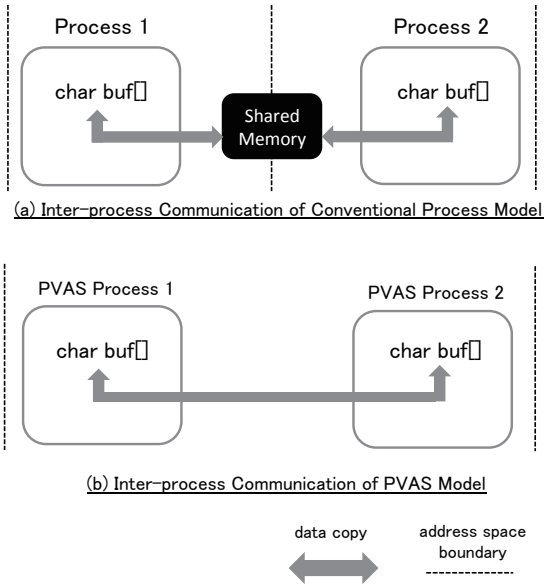

図 4 プロセス間通信

本研究では，マルチプロセス型並列アプリケーションにおいて，より低コストなプロセス間通信を実現するため，Partitioned Virtual Address Space (PVAS) と呼ぶ新たなプロセスモデルを提案する．PVAS によって，従来のマルチプロセス型並列アプリケーションで問題になっていた，プロセス間通信のコストを低減すると同時に，マルチスレッド型並列アプリケーションでの問題を回避し，より効率的なノード内並列化を実現する．

PVAS を適用したマルチプロセス型並列アプリケーションでは，PVAS プロセスと呼ぶ特殊なプロセス群を同一アドレス空間で実行させる．PVAS プロセスは，通常のプロセスと同様，1 つ以上の実行コンテキストから成り，PVAS プロセスごとに個別のデータ格納領域を持つ．通常のプロセスと異なるのは，並列アプリケーションとして動作する各 PVAS プロセスが，同一アドレス空間で動作することである．図 3 は，PVAS プロセスモデルで動作する並列アプリケーションの模式図である．各 PVAS プロセスのデータ格納領域に計算用データを分散して配置し，各 PVAS プロセスの実行コンテキストに並列して計算処理を実行させる．ある PVAS プロセスの実行コンテキストが他の PVAS プロセスの持つ計算データを必要とする場合は，PVAS のプロセス間通信を利用して，PVAS プロセス間でデータの送受信を行う．

2. Partitioned Virtual Address Space

2.1 PVAS でのプロセス間通信

PVAS では，従来のマルチプロセス型並列アプリケーションよりも，低コストなプロセス間通信を実現することが可能になる．

従来のマルチプロセス型並列アプリケーションでは，並列アプリケーションとして動作する各プロセスが異なるアドレス空間で動作するため，プロセス間でデータの送受信を行う際は，図 4(a) のように，共有メモリを経由する必要があった．よって，データの送受信に 2 度のメモリコピーを実行する必要があり，プロセス間通信のオーバヘッドが大きくなっていた．また，共有メモリを確保するために，メモリリソースが消費されてしまう問題があった．

それに対し，PVAS では，並列アプリケーションとして動作する各プロセスが，同一アドレス空間で動作するため，図 4(b) のように，プロセス間で, 互いのデータ格納領域に直接メモリコピーを行うことができる．よって，PVAS のプロセス間通信では，1 度のメモリコピーで通信処理を完了することができ，通信によるオーバヘッドを低減することができる．また，共有メモリを確保するために必要なメモリリソースを消費することがないため，計算処理のために割り当て可能なメモリリソースがマルチプロセスモデルの並列アプリケーションよりも増加する．このように，PVAS では，プロセス間通信をより低コストで実現することができ，メニーコア環境において，従来のプロセスモデルよりも，効率的なノード内並列化を実現することができる．

PVAS では，並列アプリケーションとして動作する各 PVAS プロセスは，同一アドレス空間で動作するが，PVAS プロセスごとにデータ格納領域は独立している．各 PVAS プロセスの実行コンテキストは，各々のデータ格納領域に分散配置された計算データにアクセスして計算処理を実行するため，マルチスレッド型並列アプリケーションのように，データアクセスの競合が，パフォーマンスの低下を引き起こすことはない．

2.2 設計

PVAS プロセスモデルを Linux カーネルに実装した．本

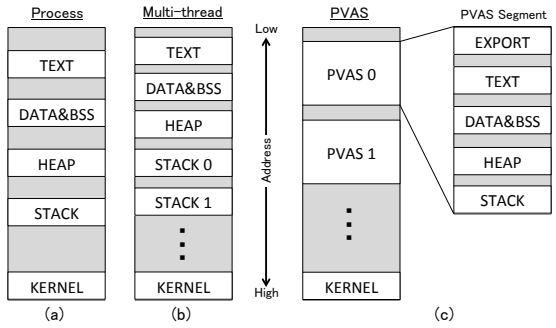

図 5 アドレス空間レイアウト

節では，PVAS のカーネル内設計について述べる．

図 5(a) は，Linux における，通常プロセスのアドレス空間レイアウトを示している．1 プロセスに対して，ひとつの text/data/bss/heap/stack セグメントが確保される．残りの空間は，mmap 用の領域として利用される．

図 5(b) は，Linux において，マルチスレッド化されたプロセスのアドレス空間レイアウトを示している．Linux では，`clone()` システムコールによって，1 プロセス内に複数の実行コンテキスト（スレッド）を作成することができる．各スレッドは，同一アドレス空間で動作し，text/data/bss/heap セグメントは，複数のスレッドに共有される．stack セグメントは，スレッドごとに確保される．残りの空間は，mmap 用の領域として利用される．

図 5(c) は，並列アプリケーションとして動作する PVAS プロセスのアドレス空間レイアウト示している．PVAS プロセスモデルでは，並列アプリケーションを構成する複数の PVAS プロセスが同一アドレス空間で動作する．複数の PVAS プロセスが動作するアドレス空間全体を PVAS アドレス空間と定義する．

PVAS では，1 つの PVAS アドレス空間を固定長で分割して，各 PVAS プロセスに割り当てる．各 PVAS プロセスに割り当てた領域を PVAS セグメントと定義する．PVAS プロセスは，独立した text/data/bss/heap/stack セグメントを持ち，各セグメントは，自身の PVAS セグメントに確保される．また，PVAS プロセスが mmap を実行した場合，mmap 用の領域は，自身の PVAS セグメントから確保される．

通常のプロセスは，独自のアドレス空間で動作するため，各々異なるページテーブルを使用するが，並列アプリケーションを構築する各 PVAS プロセスは，同一アドレス空間で動作するため，ひとつのページテーブルを共有する．

各 PVAS プロセスが，同一アドレス空間で動作するため，共有メモリやカーネルを介さず，他の PVAS プロセスのデータに直接アクセスすることができ，低コストなプロセス間通信を実現することができる．各 PVAS プロセスは同一アドレス空間で動作するが，データ格納領域（data/bss/heap セグメントおよび mmap 領域）は，PVAS プロセスごとに用意されるため，マルチスレッド型並列アプリケーションで発生するような実行コンテキスト間のデータアクセスの競合を回避することができる．

PVAS プロセスは，プロセス作成時に設定される固有の PVAS ID (≧ 0) を持つ．PVAS ID によって PVAS プロセスに割り当てられる領域（PVAS セグメント）が決定される．PVAS セグメントのスタートアドレスは，PVAS ID と割当領域のサイズを乗算した値となる．例えば，割当領域のサイズが 4GB で，PVAS ID が 2 の場合は，当該 PVAS プロセスには，仮想アドレス 0x200000000 から 0x2fffffff の領域が，PVAS セグメントとして割り当てられる．

また，PVAS では，PVAS セグメントの先頭にエクスポート領域を設ける．この領域は，プロセス間通信を行う際の制御情報を格納するための領域である．例えば，プロセス間通信を行う際，受信プロセスのエクスポート領域に受信バッファのアドレスを格納しておけば，送信プロセスに送信データのコピー先を通知することができる．エクスポート領域は，必ず PVAS セグメントの先頭に配置される．よって，各 PVAS プロセスは，互いのエクスポート領域のアドレスを，PVAS ID から計算で求めることができる．

通常のプロセスとして実行可能なプログラムは，そのまま PVAS プロセスとして実行することができる．ただし，実行するプログラムは，Position Independent Executable (PIE) フォーマットでコンパイル・リンクされている必要がある．また，実行プログラムにリンクされる共有ライブラリは，Position Independent Code (PIC) フォーマットでコンパイルされている必要がある．

2.3 User-level API

PVAS の User-level API を表 1 に示す．API は C 言語のライブラリとして実装した．今回は，PVAS をユーザプログラムから利用するための基本 API のみの設計および実装をおこなった．今後，他の部分も検討し，設計実装していく予定である．基本 API の使用例を図 6 に示す．図 6 は，並列アプリケーション parallel を PVAS プロセスとして実行する際の例である．

まず，`pvas_create` で，PVAS プロセス群を動作させる PVAS アドレス空間を作成する (1)．次に，`pvas_fork` を実行し，PVAS プロセスを作成する (2)．そして，作成した PVAS プロセスが，`pvas_execve` で，プログラムを実行する (3)．各 PVAS プロセスは，`pvas_fork` 実行直後は，独自のアドレス空間で動作している．`pvas_execve` 実行後に，PVAS アドレス空間上で動作するようになる．PVAS プロセス群の親プロセスは，最後に，`pvas_destroy` で，PVAS アドレス空間を，削除する (4)．

3. MPI ライブラリの実装

PVAS のプロセス間通信を利用する MPI のランデブー

表 1 User-level API

#	関数名	書式	説明
1	pvas_create	int pvas_create(*void*);	PVAS アドレス空間を作成する．この関数は，PVAS プロセスの親プロセスが実行する．返り値として，作成した PVAS アドレス空間に対応するディスクリプタを返す．
2	pvas_fork	pid_t pvas_fork(int *pvd*, int *pvid*);	PVAS プロセスを作成する．この関数は，PVAS プロセスの親プロセスが実行する．引数 *pvd* は，PVAS プロセスを作成する PVAS アドレス空間のディスクリプタである．引数 *pvid* は，作成する PVAS プロセスに設定する PVAS ID である．返り値として，親プロセスには，作成した PVAS プロセスのプロセス ID を返す．作成された PVAS プロセスには，0 が返る．
3	pvas_execve	int pvas_execve(const char **path*, char *const **argv[]*, char *const *env[]*);	指定したプログラムを PVAS プロセスとして実行する．この関数は，PVAS プロセス自身が実行する．引数 *path* は，実行するプログラムのファイルシステム上のパスである．引数 *argv[]* は，実行プログラムの引数リストである．引数 *env[]* は，プログラムの実行に際し設定する環境変数のリストである．
4	pvas_destroy	int pvas_destroy(int *pvd*);	指定された PVAS アドレス空間を削除する．この関数は，PVAS プロセスの親プロセスが実行する．引数 *pvd* は，削除する PVAS アドレス空間に対応するディスクリプタである．PVAS アドレス空間の削除実行時に，当該 PVAS アドレス空間上で実行されている PVAS プロセスが存在する場合，当該 PVAS プロセスは強制的に削除される．実行に成功した場合 0 を，失敗した場合は-1 を返り値として返す．

```
--- <省略> ---
#define PROCESS_NUM 32
int main(int argc, char *argv[]) {
  int pvd,i;
  int pid[PROCESS_NUM];
  char *const arg[];

--- <省略> ---

  pvd = pvas_create(); ———————————————— (1)

  for(i = 0; i < PROCESS_NUM; i++) {
    pid[i] = pid_fork(pvd, i); ———————————— (2)
    if(pid[i] == 0) {
      pvas_execve("./parallel", arg, NULL); ———— (3)
    } else if(pid[i] > 0) {
      printf("fork pvas process %d\n", pid[i]);
    }
  }

--- <省略> ---

  pvas_destroy(pvd); ———————————————— (4)
  return 0;
}
```

図 6 サンプルコード

通信を，代表的な MPI ライブラリとして広く用いられている MPICH2 [3] に実装した．MPICH2 の通信レイヤの実装である Nemesis[9] に対して，ノード内の MPI 通信に PVAS のプロセス間通信を利用する改造を行った．本章では，Nemesis の共有メモリを用いた MPI 通信の実装と，それに対して実施した改造について述べる．

3.1 Nemesis の MPI 通信

Nemesis では，ノード内の MPI 通信において，共有メモリを介した通信をサポートしている．本節では，共有メモリを介した MPI のランデブー通信の実装について，ブロッキング通信を例に述べる．

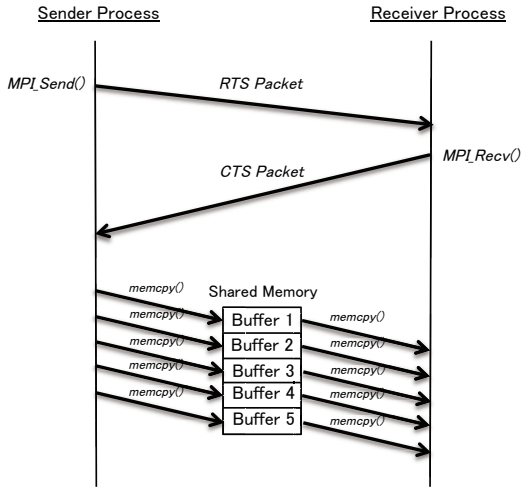

図 7 共有メモリによる MPI 通信

図 7 は，共有メモリを用いた MPI のブロッキング通信の動作を示している．まず，送信プロセスが，RTS パケットを送信することで，送信リクエストが発行されたことを，受信プロセスに通知する．受信プロセスは，受信リクエストが発行された際，対応する RTS パケットをすでに受信していれば，CTS パケットを送信することで，受信準備ができたことを送信プロセスに通知する．RTS パケットを未受信の場合は，RTS パケットを受信するまで待機する．CTS パケットを受信した送信プロセスは，送信バッファから共有メモリに，memcpy() によって，送信データをコピーする．共有メモリ上には，複数のバッファを用意しておき，バッファサイズ分のデータをコピーしたら，次のバッファ

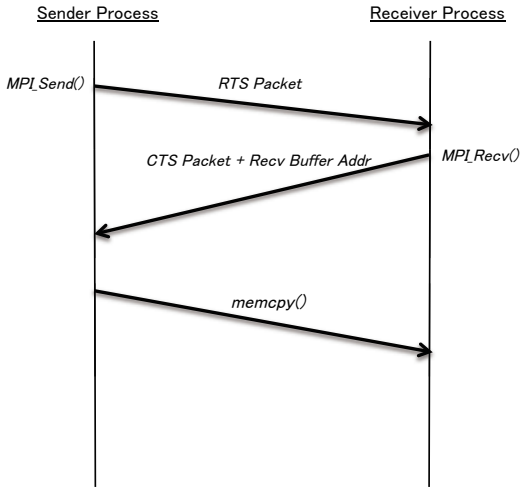

図 8 PVAS による MPI 通信

にデータを逐次コピーする．受信プロセスは，送信データのコピーが完了したバッファから，データを逐次受信バッファにコピーしていく．よって，データ通信の実行ごとに 2 回の memcpy() が発生する．

3.2 PVAS の MPI 通信

Nemesis の共有メモリを介した MPI のランデブー通信を，PVAS のプロセス間通信を用いたランデブー通信に改造した．

図 8 は，PVAS のプロセス間通信を用いた MPI のブロッキング通信の動作を示している．まず，送信プロセスが，RTS パケットを送信することで，送信リクエストが発行されたことを，受信プロセスに通知する．受信プロセスは，受信リクエストが発行された際，対応する RTS パケットをすでに受信していれば，CTS パケットを送信することで，受信準備ができたことを送信プロセスに通知する．この際，受信バッファの仮想アドレスを，CTS パケットに含めて送信する．RTS パケットを未受信の場合は，RTS パケットを受信するまで待機する．CTS パケットを受信した送信プロセスは，CTS パケットに含まれる受信バッファのアドレスをもとに，memcpy() によって，送信データを受信バッファにコピーする．PVAS では，並列アプリケーションとして動作するプロセスがアドレス空間を共有しているため，送信プロセスのバッファから受信プロセスのバッファに，データを直接コピーすることができる．よって，1 回の memcpy() で，通信を完了することができる．

4. 予備評価

3 章で述べた，PVAS のプロセス間通信を利用する MPI 通信を，共有メモリを用いた MPI 通信と比較するため，Intel MPI Benchmark[11] を用いて，両実装の Ping-pong 通信の性能を測定した．PVAS の MPI 通信については，常にランデブー通信を行った場合の性能を測定した．共有メモリを用いた MPI 通信については，常にランデブー通信を行った場合と常にイーガー通信を行った場合の性能を測定した．図 9，図 10 はそれぞれ，Ping-pong 通信のレイテンシとバンド幅の測定結果を示している．

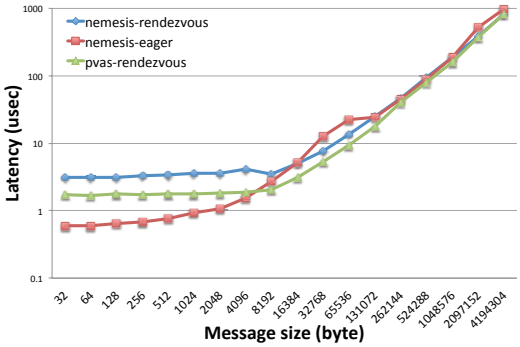

図 9 レイテンシ

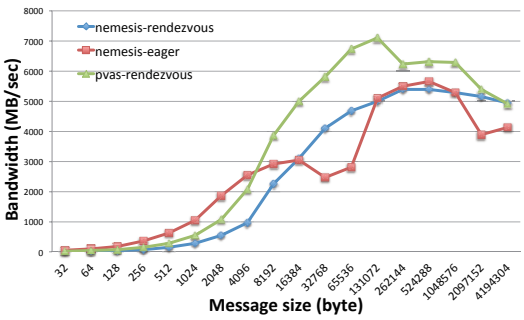

図 10 バンド幅

メッセージサイズが 8KB 以下の場合は，共有メモリを用いたイーガー通信が最も良い性能を示している．これは，イーガー通信の場合は，MPI ライブラリが持つ内部バッファに送信データをコピーした段階で送信完了となるためである．

メッセージサイズが 8KB 以上の場合は，イーガー通信の性能が，他の実装と比べて低下する．これは，MPI ライブラリの内部バッファのサイズを，送受信するメッセージの総サイズが超えてしまうためであると推測される．

共有メモリのランデブー通信と比べ，PVAS のランデブー通信の性能が優れているのは，通信の完了までに必要な memcpy() の回数が，共有メモリによる通信と比べて少ないためである．PVAS のランデブー通信は，メッセージサイズが 8KB 以上の場合，最も良い性能を示す．

PVAS のランデブー通信の性能は，MPI 通信の実装を改良することで，さらに改善可能であると考える．

現在の実装では，ランデブーを行うための制御メッセージ（RTS パケット，CTS パケット）の送受信が，通信のオーバヘッドになっている．メッセージ送信時に受信プロセスの受信リクエストキューを確認して，送信リクエストに対応する受信リクエストがあれば 即座に送信プロセスが memcpy() する，あるいはその逆に，メッセージ受信時

に送信プロセスの送信リクエストキューを確認して，受信リクエストに対応する送信リクエストがあれば 即座に受信プロセスが memcpy() する，という最適化を行えば，ランデブーのためのオーバヘッドを低減することができる．

また，メッセージサイズが大きい場合は，メッセージを分割して，送信プロセスと受信プロセスが同時にデータコピーを実行する等の最適化が考えられる．

5. 関連研究

5.1 カーネルによるプロセス間通信

高橋らは，カーネル内でプロセス間のデータコピーを行うことで，ノード内通信を高速化する手法を提案している．[13]．通常並列動作するプロセスは，それぞれ異なるアドレス空間で動作するため，プロセス間通信を行うためには共有メモリを経由して，データコピーを行う必要がある．しかし，OS カーネルは，全てのプロセスのアドレス空間にアクセスすることが可能であるため，カーネルにデータコピーの処理を委譲することで，送信プロセスのバッファから受信プロセスのバッファに直接データコピーを行うことができる．PVAS のプロセス間通信と同様に，1 度のデータコピーで通信処理を完了できるため，共有メモリによる通信よりも，低オーバヘッドなプロセス間通信を実現することができる．

しかし，この方式では，プロセス間通信を行うたびに，カーネルへのコンテキストスイッチを行う必要があり，それが通信のオーバヘッドとなる．PVAS のプロセス間通信では，カーネルへのコンテキストスイッチを必要とせず，ユーザプロセスがデータコピーを実行することができるため，カーネル内のデータコピーによる通信よりも，低オーバヘッドな通信を実現できる．

5.2 プロセス間マッピング方式

XPMEM[7] は，他のプロセスが使用している物理メモリのページを，自身のアドレス空間にマッピングする機能をプロセスに提供する Linux のカーネルモジュールである．他のプロセスが使用している計算データに共有メモリを経由せずにアクセスすることが可能になるため，PVAS と同様に，1 度のデータコピーでプロセス間通信を完了できる．しかし，この方式では，他のプロセスが使用している物理メモリのページを自身のアドレス空間にマッピングするために，ページテーブルのサイズが肥大化してしまう問題がある．例えば，32 プロセスが各々 100MB の計算データを用いて並列処理を実行していたとする．各プロセスが，他のプロセスの計算データを自身のアドレス空間にマップしようとすると，全体で，100GB(= 32×32×100MB) 分の仮想アドレス空間を消費する．MIC のような x86 アーキテクチャ上で動作させると仮定すると，1GB の仮想アドレス空間をマップするために必要なページテーブルのサイズは 2MB なので，200MB 分のメモリがページテーブルによって，消費されてしまう．メニーコアアーキテクチャでは，搭載されるメモリ量が限られているため，ページテーブルのサイズが肥大化することは好ましくない．

SMARTMAP[8] は，XPMEM と同様，他のプロセスが使用している物理メモリのページを自身のアドレス空間にマッピングする機能をプロセスに提供する．サンディア国立研究所開発の軽量カーネルである Kitten[1] および Catamount[4] に実装されており，x86 アーキテクチャを前提に動作する．1 段目のページテーブル（PML4）の 512 エントリの内，最初のエントリを，当該プロセスが使用するアドレス空間とする．残りの 511 エントリを他のプロセスのアドレス空間をマップするために使用している．他のプロセスの PML4 の最初のエントリを，残りの 511 エントリにコピーすることで，自身のアドレス空間に他のプロセスのアドレス空間をマップする．よって，511 プロセスのアドレス空間しか，マッピングできないという制限がある．2 段目以降のページテーブルは，他のプロセスと共有するため，XPMEM 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，プロセス間でページテーブルを同期する機能を持たないため，プロセス起動時に，物理メモリのページを事前にプロセスに割り当てる必要がある．使用することがない物理メモリを割り当ててしまう可能性があるため，メモリの使用効率が悪くなる問題がある．

6. 考察

PVAS では，並列して動作するプロセスがアドレス空間を共有しているため，通常のプロセスでは機能するプロセス間のデータ保護が機能しない．よって，あるプロセスがバグにより，他のプロセスの使用しているデータを破壊してしまうことがあり得る．

並列アプリケーションでは，あるプロセスがバグで予期せぬ動作を起こした場合，正しい計算結果を得られないことが多い．そういったケースでは，そのバグが他のプロセスのデータを破壊するものであってもなくても，正しい結果を得られないことにかわりはないので，PVAS によってプロセス間のデータ保護が機能しなくても，大きな問題にはならないと考える．しかし，あるプロセスがバグで予期せぬ動作を起こしても，正しい計算結果を得られるようなフェイルセーフな設計にプログラムがなっている場合は，PVAS によってプロセス間のデータ保護が機能しないことは，アプリケーションの耐障害性を低下させることになる．

プログラマは，並列アプリケーションの性能と耐障害性のトレードオフを考慮して，PVAS を使用する必要がある．

PVAS では，あるプロセスが予期せぬ動作を起こしても，同じ並列アプリケーションとして動作するプロセスにしか，影響を与えることはないため，最低限のデータ保護はできていると考える．

7. 結論

本研究では，マルチプロセス型並列アプリケーションにおいて，低コストなプロセス間通信を実現するためプロセスモデルとして，PVAS を提案した．PVAS では，並列して動作するプロセスが同一アドレス空間で動作するため，共有メモリを介さずに，プロセス間で直接データの送受信を行うことができ，より低コストなプロセス間通信が可能となる．よって，PVAS を適用したマルチプロセス型並列アプリケーションでは，従来のプロセスモデルで動作するマルチプロセス型並列アプリケーションよりも，メニーコア環境において，より効率的なノード内の並列化を実現することができる．

PVAS のプロセス間通信を MPI 通信に適用し，共有メモリを用いた MPI 通信と比較したところ，8KB 以上のメッセージサイズにおいて，Ping-pong 通信のレイテンシとスループットが大幅に改善することを確認することができた．

PVAS を利用する MPI ライブラリの最適化や，マルチプロセス型並列アプリケーションの実装言語である PGAS に PVAS を適用して評価すること等が，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] Kitten Lightwaight Kernel. https://software.sandia.gov/trac/kitten.
[2] Many Integrated Core (MIC) Architecture - Advanced. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.
[3] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.
[4] Open Catamount. http://www.cs.sandia.gov/~rbbrigh/OpenCatamount/.
[5] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/.
[6] The UPC Language. http://upc.lbl.gov/.
[7] Ron Brightwell and Kevin Pedretti. An Intra-Node Implementation of OpenSHMEM Using Virtual Address Space Mapping. In *PGAS 2011: Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, 2011.
[8] Ron Brightwell, Kevin Pedretti, and Trammell 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*, SC '08, pp. 25:1–25:12, Piscataway, NJ, USA, 2008. IEEE Press.
[9] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pp. 521–530, Washington, DC, USA, 2006. IEEE Computer Society.
[10] Mike Giles. Runners and Riders in GPU Steeplechase. Technical report, Oxford University Mathematical Institute Oxford e-Research Centre, 2010. NAG Technical Forum.
[11] Intel Corporation. Intel MPI Benchmarks 3.2.3. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.
[12] Lee Jinpil and Sato Mitsuhisa. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *The 39th international Conference on Parallel Processing Workshops*, ICPPW10, 2010.
[13] Toshiyuki Takahashi, Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Hiroshi Harada, Yutaka Ishikawa, and Peter H. Beckman. Implementation and evaluation of mpi on an smp cluster. In *IPPS/SPDP Workshops*, pp. 1178–1192, 1999.