Vol.2013-HPC-141 No.7
2013/9/30

メニーコア混在型並列計算機向け 大域仮想アドレス空間モデルMultiple PVASの提案

深沢 豪[1,a)]　佐藤 未来子[1,b)]　吉永 一美[2,c)]　辻田 祐一[2,d)]　島田 明男[2,e)]　堀 敦史[2,f)]
並木 美太郎[1,g)]

概要：本論文では我々がこれまでに提案しているメニーコア向け新プロセスモデル PVAS（Partitioned Virtual Address Space）と，機能並列によるメニーコア CPU の有効活用を追求する PVAS Agent モデルをメニーコア混在型並列計算機へ適用するための “Multiple PVAS” を提案する．Multiple PVAS では Host のマルチコア CPU とメニーコア CPU を包括する大域仮想アドレス空間を提供することで，並列度の高いタスクをメニーコアで，シーケンシャルな処理をマルチコアで処理するための高効率な連携基盤を実現する．Xeon Phi を搭載したメニーコア混在型並列計算機において Multiple PVAS を実装し，CPU 間での通信性能を測定したところ，Xeon Phi から Host へ 2〜17us 程度で，Host から Xeon Phi へ 2〜7us 程度で Agent への処理依頼が可能であるという結果を得た．今後は本指標を元に各種 Agent を設計・開発し，実アプリケーションの実行性能向上を目指す．

1. はじめに

スーパーコンピュータの性能は日々進化し続けており，近年の傾向としてコア単体の性能向上よりもシステムに搭載するコア数を増加させてより高い性能を実現している．最新のスーパーコンピュータの性能ランキング Top500[1] で世界最高性能を記録した天河 2 号（Tianhe-2）は，Intel のマルチコアプロセッサ Xeon 2 基と，Intel のメニーコアプロセッサ Xeon Phi 3 基を搭載する 1 万 6000 のノード，総コア数 312 万個という莫大な並列環境により 33.86 PetaFLOPS を達成している．今後もこのようなメニーコアプロセッサ混在型のシステムが HPC の分野で広く使われていくと想定し，筆者らはシステムソフトウェアの研究開発を行っている [2][3][4][5][6].

本研究で対象とするメニーコア混在型並列計算機を**図 1** に示す．ホスト CPU には通常のマルチコア CPU を備え，ノード内の PCIe バスに別途メニーコア CPU を複数接続し，各 CPU に備えるメモリを PCIe 経由で互いにアクセスできるシステムである．本研究では，これを計算ノードとして高性能なネットワークで多数結合されている計算機システムを想定している．メニーコア CPU としては，Intel 社の MIC（Many Integrated Core）を想定しており，市場で入手可能な Xeon Phi ではキャッシュコヒーレンシを保ちながら 240 個のスレッドを同時実行できる．Xeon Phi では Linux が稼働しており，従来の Linux システムで開発された並列プログラムをメニーコア CPU へ移行するための開発環境も提供されている [7].

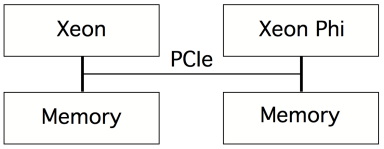

図 1　メニーコア混在型並列計算機の概念図 [3]

しかし，現状の Xeon Phi は従来の高性能なマルチコア CPU に比べてコア単体性能が低く，タスクの並列度を上げて適切なデータ分割と同期処理を施した並列化により性能を向上させ，並列化できないシーケンシャルな処理を性能の高いホスト CPU で実行させるというハイブリッドプログラミングが必要となる [8]．また，Xeon Phi にはハードディスク装置や Ethernet などの I/O 資源を直接接続できないため，I/O 資源を利用したメニーコアアプリケーションを作成したい場合にも，ホスト CPU と連携実行するよ

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[2] 独立行政法人理化学研究所計算科学研究機構
RIKEN AICS
[a)] fzawa@namikilab.tuat.ac.jp
[b)] mikiko@namikilab.tuat.ac.jp
[c)] kazumi.yoshinaga@riken.jp
[d)] yuichi.tsujita@riken.jp
[e)] a-shimada@riken.jp
[f)] ahori@riken.jp
[g)] namiki@cc.tuat.ac.jp

うプログラミングする必要がある．

このような背景から，従来とは異なる設計思想に基づき，ホスト CPU とメニーコア CPU を併用する新しいスタイルで実アプリーションを構築し，メニーコア CPU を有効活用するためのプログラム実行基盤が重要であると考えている．本研究では，

- メニーコアの特性を活かした効率的な並列実行環境の実現
- メニーコアとマルチコアの効率的な連携機構の提案

の二点を研究目的としたシステムソフトウェアの研究開発を進めている．これまで，CPU 上で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 Partitioned Virtual Address Space（PVAS）[4][5] や，メニーコア CPU 上の一部のコアを Helper Thread として稼動させて機能並列によるメニーコア CPU の有効活用を追求する PVAS Agent モデル [3] を提案している．本研究では，PVAS および PVAS Agent のアプローチをメニーコア混在型計算機へ適用するための Multiple PVAS を提案し，異なる CPU 上のプロセス同士が一つのまとまった仮想アドレス空間を自由に直接アクセスすることのできる仕組みを既存 OS 上に設計し，試作した．Intel Xeon と Intel Xeon Phi を搭載するシステムにおいて Multiple PVAS の基礎評価を行い，OS におけるメモリ管理オーバヘッド，およびリモート CPU との通信性能について考察し，Multiple PVAS を用いたシステム構築の指標を示す．

2. 本研究の課題と目標

本章では，本研究の基盤となる PVAS および PVAS Agent について述べ，PVAS をメニーコア混在型計算機へ適用するために提案する Multiple PVAS について述べ，Multiple PVAS 実現にむけた課題と本研究の目標を示す．

2.1 PVAS および PVAS Agent

PVAS はプロセスとスレッドの中間に位置する新しいタスクモデルである [4][5]．PVAS においてコアを割り当てる実行実体であるプロセスを “PVAS Task” と呼び，**図 2** に示すように一つの仮想アドレス空間に PVAS Task 個別の空間を配置し，ページテーブルを共有する．そのため，PVAS Task 間では，既存のプロセスで必要であった共有メモリを確保することなく，仮想アドレスの情報を交換することで直接参照による情報の受け渡しが行える．

PVAS Task には仮想アドレス空間上に配置した順に識別番号を割り振ることで，仮想アドレス空間にある PVAS Task の各領域（“PVAS Partition” と呼ぶ）を識別可能としている．また，PVAS Partition の先頭には “Export Segment” と呼ばれる領域を設け，PVAS Task の識別番号からそのアドレスを知ることにより，Export Segement を用いた PVAS Task 間の情報の受け渡しや同期を実現可能にしている．

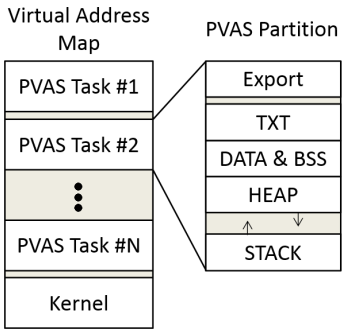

図 2 PVAS アドレスマップ

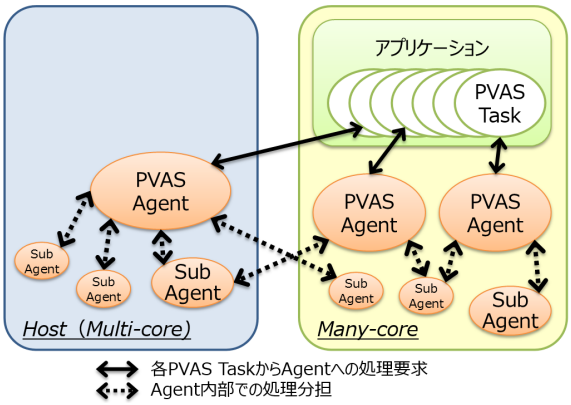

図 3 メニーコア混在型計算機へ適用した PVAS Agent の概念図

また，PVAS を適用するメニーコア CPU においては，メニーコア CPU 上の利用しきれないハードウェアスレッドの一部を利用した機能並列を実現するための Agent モデルを提案している [3]．PVAS Task と同じ仕様で PVAS Agent を同じ仮想アドレス空間上に配置して，アプリケーションを構成する PVAS Task が望みの機能を備える PVAS Agent に対して処理要求を発行し，その結果を待つというモデルである．他の PVAS Partition 内のデータを自由にアクセスできるという PVAS の特長から，PVAS Task と Agent 間の通信には PVAS Agent の Export Segment を利用し，そこへ要求を書込み結果を得ることで Agent への処理を通知できる．大容量データや複雑なデータ構造を PVAS Task/Agent 間で受け渡す場合は，それらを指すアドレス情報のみを Export Segment を経由し，実データをメモリ参照により直接読み書きすることで，不要なデータコピーを回避した効率のよい通信を可能としている．

2.2 Multiple PVAS の提案

2.1 節で述べた研究はメニーコア混在型計算機への適用を目的としており，特に Agent モデルについては，**図 3** に示すように，Agent や Agent 内部で機能並列を実現するための Sub Agent をマルチコア CPU 側へ配置することも想定している．例えば，現状の Xeon Phi 上ではむしろ性能が低下してしまうシーケンシャルな処理や，I/O 資源を活

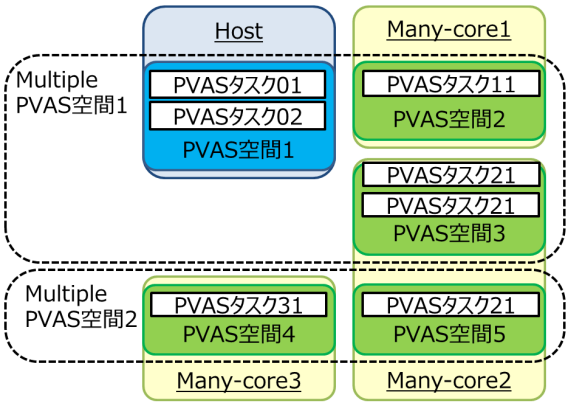

図 4 Multiple PVAS 空間の概念図

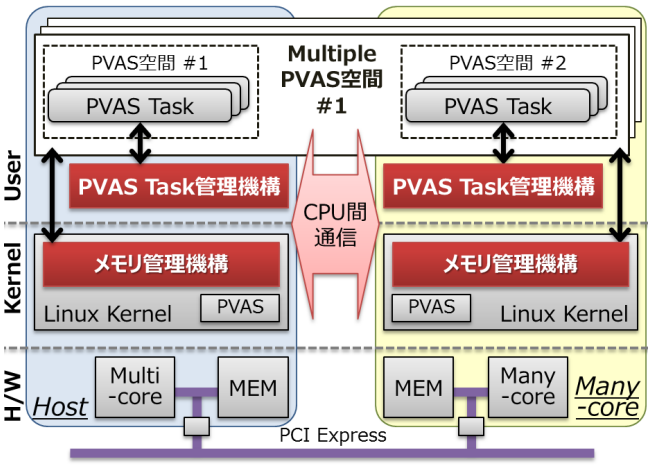

図 5 Multiple PVAS を含む全体構成

用する処理などをホスト側の Agent として稼働させるなど，柔軟に PVAS Agent を配置することで性能向上に向けた処理構成の最適化を図れると考えている．

本研究では，メニーコア混在型並列計算機における PVAS Agent を実現するための基盤として，異なる CPU における複数の PVAS 空間を包括する形へ拡張した “Multiple PVAS” を提案する．本 Multiple PVAS では**図 4** に示すように，マルチコア CPU とメニーコア CPU が混在するシステムにおいて，既存の PVAS の枠組みで CPU ごとに独立していた PVAS 空間を包括する “Multiple PVAS 空間” を提供し，本 Multiple PVAS 空間に属する PVAS Task がすべて同じ仮想アドレス空間上にマップされる．すなわち，Multiple PVAS がメニーコア混在型並列計算機において提供されれば，従来の PVAS と同様に，Multiple PVAS 空間内の仮想アドレスによる情報の受け渡しが PVAS Task 間で行えるようになるため，複数の CPU で稼働する PVAS Task/Agent システムを自由に構成できる．

2.3 本研究の課題と目標

メニーコア混在型並列計算機において Multiple PVAS を実現するには，いくつかの課題がある．各 CPU を包括する仮想アドレス空間を実現するためには，CPU ごとに OS が稼働しメモリ管理を行っていることを考慮しなければならない．すなわちページテーブルの構成・維持方式やアドレス変換方式を検討する必要がある．メニーコア混在型並列計算機は CPU ごとに物理メモリを備えているため，リモート CPU の物理メモリへの高効率なアクセス方法を検討する必要がある．また，Multiple PVAS を用いた並列アプリケーションを立ち上げるためには，同一 CPU 内だけでなく異なる CPU への PVAS Task 生成・削除を実現する必要がある．

本研究ではこれらの課題の解決策を示した上で Multiple PVAS を実装し，複数の CPU で PVAS Task/Agent システムを稼働させる場合に生じるオーバヘッドを評価により明らかにすることで，メニーコア混在型並列計算機向け PVAS Agent の設計に向けた指標を得ることを目標とする．HPC への適用を目的としているため，アプリケーションの実行性能を落とすことなく，またオーバヘッドを最低限に抑えることを方針として Multiple PVAS の設計を行なう．

3. システム構成

図 5 に Multiple PVAS を含む全体構成を示す．多数のコアを集積したメニーコア CPU と，単スレッド性能に優れるがコア数が少ないマルチコア CPU を搭載するヘテロジニアス型並列計算機を想定し，CPU が有する互いの物理メモリに対して MMIO や DMA によりリモートアクセス可能なアーキテクチャを対象とする．マルチコア CPU を搭載する Host 側とメニーコア CPU 側において個別の Linux Kernel が稼動し，両 CPU 上の PVAS 空間を包括する形で Multiple PVAS 空間を形成する．Linux Kernel には Multiple PVAS の大域仮想アドレス空間を構築・維持するため “メモリ管理機構” を備え，ユーザレベルにはローカル CPU の PVAS Task の生成・監視を行なう “PVAS Task 管理機構” を備える．これらの機構を各 CPU に設け，CPU 間通信により相互に連携することで Multiple PVAS を実現する．各 CPU のソフトウェアをシンメトリックな構成とすることで，本研究で想定するヘテロジニアス型並列計算機の構成だけでなく，複数のメニーコア CPU，あるいは，複数のマルチコア CPU で構成されるホモジニアス型計算機においても Multiple PVAS を適用できる設計としている．

4. メモリ管理機構

Multiple PVAS の大域仮想アドレス空間を提供するためには，単一 CPU 上で動作する既存 OS とは異なるメモリ管理手法が必要となる．本機構により複数の CPU を包括する仮想アドレス空間を構築・維持することで，Multiple PVAS 空間内での CPU をまたいだ PVAS Task 間通信を実現する．

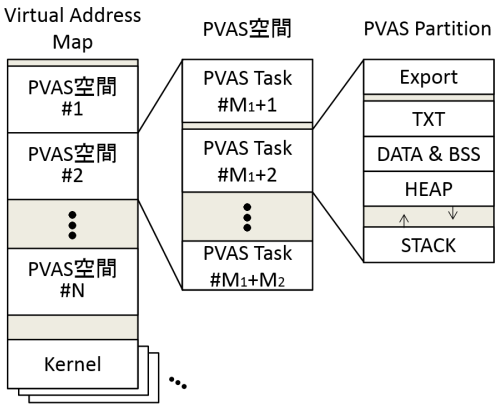

図 6 Multiple PVAS の仮想アドレス空間

図 6 に本機構が提供する仮想アドレス空間の構成を示す．Multiple PVAS では PVAS Task 生成にともなう仮想アドレス空間の構築や，Export Segment の割り当てなどの PVAS 空間管理を各 CPU へ分散させるため，大域仮想アドレス空間を CPU ごとの PVAS 空間へ分割した上で，さらに PVAS Task に対応する PVAS Partition へと分割する．このような構成をとることで，仮想アドレス空間に関する CPU 間での協調処理を極力減らし，PVAS 空間管理の処理時間増大を防ぐことで，スケーラビリティを阻害することがない設計としている．OS の Kernel 用領域を除いた仮想アドレス空間を各 CPU で同一内容とすることで，PVAS Task がどの CPU で動作しているかを考慮することなく，直接メモリアクセスによる PVAS Task 間通信を実現する．

5. PVAS Task 管理機構

Multiple PVAS 空間上でアプリケーションを動作させるためには，CPU をまたいだ PVAS Task の生成・削除を可能とする必要がある．各 CPU へ配置した本機構が，リモート CPU に代わり PVAS Task 生成などを代行することで，本課題を解決する．

本機構では PVAS Task の生成・削除を行なうとともに，生成した PVAS Task の生存状態を監視することで，以下の機能を Multiple PVAS 空間に属する全 CPU へ提供する．

- PVAS Task の生成・削除
- PVAS Task の Export Segment の仮想アドレス取得
- PVAS Task 終了の通知

PVAS Task 終了の通知機能は，ローカル CPU の PVAS Task 終了を待機する wait システムコールと同等の機能をリモート CPU へ提供するものである．以上の機能によりリモート CPU による PVAS Task 生成から終了待機・削除までの一連の操作を可能とすることで，複数 CPU に PVAS Task を配置する並列アプリケーションを単一 CPU から立ち上げ可能とする．

6. Multiple PVAS API

Multiple PVAS では，Multiple PVAS Library を介して Multiple PVAS 空間や PVAS Task の生成管理機能を提供する．**表 1** に主要 API を示す．API は基本的に PVAS の API[5] を踏襲しており，PVAS API の関数名の先頭に'm' を冠した関数名とし，必要に応じてパラメータや戻り値を追加・変更している．これにより既存の PVAS を対象としたプログラムからの移植作業の手間を軽減している．

Multiple PVAS 空間を生成する mpvas_create/destroy は，メモリ管理機構と PVAS Task 管理機構のインタフェースである．各 CPU への PVAS 空間生成と，生成した PVAS 空間を包括する Multiple PVAS 空間の構築を行なう．各々の PVAS 空間に対応する “PVAS 識別子” と，Multiple PVAS 空間に対応する “Multiple PVAS 識別子” を発行し，PVAS Task の生成先などを識別可能とする．mpvas_spawn と mpvas_wait/waitpid/waitid，および mpvas_ealloc/get_export_info は PVAS Task 管理機構のインタフェースである．PVAS Task 管理機構は CPU ごとに存在するため，操作対象の PVAS Task が位置する CPU を PVAS 識別子により特定し，CPU 間通信により処理を依頼する．

以上の API を用いたサンプルプログラムを PVAS Task の実行管理プロセスと PVAS Task の別に**図 7**，**図 8** に示す．図 7 では Multiple PVAS 空間に含める CPU のリストを引数とし create を呼び出す (1)．ここで CPU ごとの PVAS 空間識別子が発行されるため，以降 spawn で本識別子を利用し，CPU 別に PVAS Task を生成している (2)．立ち上げた PVAS Task は waitpvid にて終了を待機するが，プログラム例では PVAS Task を特定せず，立ち上げたいずれかの PVAS Task が終了するまで待機するコードとした (3)．全ての PVAS Task の終了後，destroy で Multiple PVAS 空間を削除する (4)．このように，Multiple PVAS であれば複数の CPU 上で並列動作するアプリケーションを CPU 番号を指定するのみで立ち上げることができる．

図 8 では他の PVAS Task との通信に備え Export Segment の割り当てを行ない (1)，値を格納しておく (2)．次に他の PVAS Task の Export Segment 情報の取得を試み，通信に必要な仮想アドレスを得る (3)．以降，Export Segment や他の領域を用いて自由に他の PVAS Task との通信が可能となる (4)．Multiple PVAS にて複数 CPU を包括する大域仮想アドレス空間を構築することで，CPU をまたいだタスク間の通信をローカル CPU 内での通信と同様の手順で実現できる．

7. Multiple PVAS の実現方式

本章では Multiple PVAS を実現する上で課題となる，大

表 1　Multiple PVAS Library の主要 API

関数名	概要	パラメータ	戻り値
mpvas_create	Multiple PVAS 空間を生成	空間に含める CPU のリスト， Common セグメント情報のリスト	Multiple PVAS 空間識別子， PVAS 空間識別子のリスト
mpvas_destroy	Multiple PVAS 空間を削除	Multiple PVAS 空間の識別子	-
mpvas_spawn	PVAS Task を起動	Multiple PVAS 空間識別子， PVAS 空間識別子，PVAS Task ID， バイナリファイルパス，引数リスト	-
mpvas_wait， mpvas_waitpvid， mpvas_waitid	PVAS Task 終了を待つ	Multiple PVAS 空間識別子， 待機条件（PVAS 空間識別子， PVAS Task ID，その他オプションなど）	PVAS Task の状態
mpvas_ealloc	Export Segment を生成	領域のサイズ	-
mpvas_get_export_info	指定 PVAS Task の Export Segment 情報を取得	PVAS 空間識別子，PVAS Task ID	領域の先頭アドレス，サイズ

```
void main() {
  int mpvd, pvd[2], status, i;
  mpvas_cpuid_t cpu[2] = {HOSTID, MCID};
  char *argv[3] = {0}, rpvd[16];
  char *binpath[2] = {"./host.bin", "./mc.bin"};
  mpvas_create(NULL, 0, &mpvd, cpu, 2, pvd); —— (1)
  argv[0] = binpath[0];
  argv[1] = rpvd;
  sprintf(rpvd, "%d", pvd[1]);
  mpvas_spawn(mpvd, pvd[0], 0, binpath[0], argv);
  argv[0] = binpath[1];
  sprintf(rpvd, "%d", pvd[0]);
  mpvas_spawn(mpvd, pvd[1], 0, binpath[1], argv); – (2)
  for (i=0; i<2; i++)
    mpvas_waitpvid(mpvd, -1, -1, &status, 0); ——— (3)
  mpvas_destroy(mpvd); ———————————— (4)
}
```

図 7　PVAS Task の実行管理プロセスのプログラム例

```
void main(int argc, char *argv[]) {
  int *laddr, *raddr, rpvd, ret;
  size_t size;
  rpvd = atoi(argv[1]);
  mpvas_ealloc(sizeof(int)); ———————————— (1)
  mpvas_get_export_info(PVAS_MY_SPACE,
                        0, &laddr, &size);
  *laddr = 0x123; ———————————————— (2)
  do {
    ret = mpvas_get_export_info(rpvd, 0, &raddr, &size);
  } while (ret || size<=0); ———————————— (3)
  printf("remote = %X\n", *raddr); ———————— (4)
}
```

図 8　PVAS Task のプログラム例

域仮想アドレス空間を実現するためのメモリ管理方式や，リモート CPU の物理メモリへのアクセス方式，また CPU 間での通信方式の設計について述べる．

7.1　大域仮想アドレス空間の実現方式

複数の CPU を包括する大域仮想アドレス空間を実現するためには，CPU ごとに OS が稼働しメモリ管理を行っていることを考慮しなければならない．以下では仮想アドレス空間を実現するページテーブルの構成・維持方式について述べる．

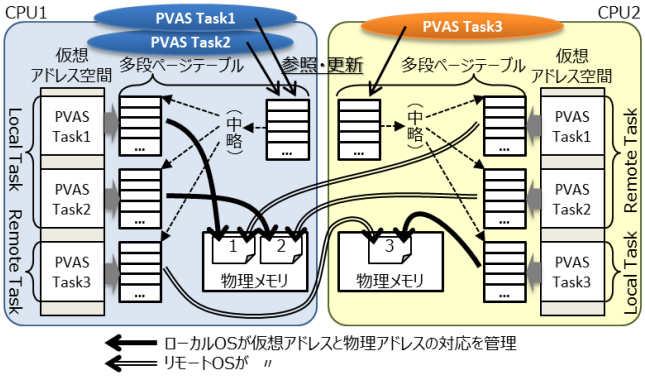

図 9　ページテーブルの構成

7.1.1　ページテーブルの構成

Multiple PVAS におけるページテーブルの構成を**図 9** に示す．Multiple PVAS では，アプリケーション実行性能を落とすことなく大域仮想アドレス空間を実現するため，大域仮想アドレス空間を構成するページテーブルを CPU ごとに別個に構築する設計とした．各 CPU の OS は自身の上に存在する PVAS 空間の仮想アドレスのみ物理メモリアサインなどのページテーブル管理を行ない，リモート CPU 上の PVAS 空間の仮想アドレスはリモート CPU のページテーブルの内容をコピーする．全 CPU が同一のページテーブルを共有する設計と比較し，ページテーブルが必ずローカルの物理メモリ上に存在するため，TLB ミスオーバヘッドが小さいという利点がある．

また，PVAS Task 間通信のオーバヘッドを削減するため，同一 CPU 上では全ての PVAS Task が一つのページテーブルを共有する設計とした．ページテーブルを共有することで，一度物理メモリがアサインされた仮想アドレスは，以降同一 CPU 上の PVAS Task であればページフォルトを起こすことなくアクセス可能となる．PVAS Agent

モデルでは少数の Agent に対して多数の PVAS Task が処理要求を行なうことが想定されるため，ページフォルトを最小限に抑えられる本設計により Agent に対する PVAS Task 数が増加した場合のスケーラビリティ向上を望める．

7.1.2 ページテーブルの一貫性維持方式

Multiple PVAS は PVAS Agent モデルをメニーコア混在型並列計算機上で実現することを想定したシステムであり，リモート CPU の PVAS 空間に対するメモリアクセスはごく一部の領域に限定される．このため，Multiple PVAS ではアプリケーション実行性能とメモリ使用量削減のため，ページテーブルのエントリ（多段ページテーブルの末端，4KB ページと対応）単位で一貫性を保つ．一貫性を保つ対象は CPU をまたいだメモリアクセスが行われたエントリに限定し，同一 CPU 内からのみアクセスされるエントリの一貫性は保たない設計とする．これによりアクセスが行われないと考えられる多くの領域について，一貫性を維持するための同期オーバヘッドを削減できるほか，ページテーブル追加にともなう物理メモリ資源の浪費を避けることができる．

エントリ単位での一貫性維持は，リモート CPU 上の PVAS 空間に対応する仮想アドレスについてデマンドページングを実施するとともに，ページマイグレーション，munmap を全ての CPU が同期して実施することで実現する．すなわち，ページフォルト時にリモート CPU のページテーブルエントリをローカル CPU のページテーブルへ追加し，リモート CPU でエントリの内容が更新される際にはローカル CPU のエントリも同様に更新する．

7.1.3 リモート CPU のページテーブルの参照方式

リモート CPU の PVAS 空間へのアクセスにともなうページフォルト処理では，リモート CPU のページテーブルを参照後，ローカル CPU のページテーブルへエントリを追加する．ページテーブル参照方式としてリモート CPU の物理メモリへ直接アクセスする “直接参照方式” と，CPU 間通信でエントリ内容を照会する “間接参照方式” の 2 方式が考えられる．直接参照方式はオーバヘッド要因となる CPU 間通信を行わないため間接参照方式よりも高速であると考えられるが，参照を行なうにあたりリモート CPU のページテーブルをローカル CPU のページテーブルへマップする必要があり，システムの性能によっては間接参照方式が高速となる場合も考えられる．また，物理アドレスがアサインされていないエントリの場合は，間接参照方式を用いてリモート CPU の OS にて物理メモリをアサインをする必要がある．

そこで，8.2 節で両方式の性能を評価した上で，直接参照方式が間接参照方式に比べ高速であれば，はじめに直接参照方式を用い，アサインされていない場合に限り間接参照方式を用いる 2 段階の方式をとる．間接参照方式の方が高速であれば，直接参照方式を用いず間接参照方式のみで参照を行なう．これにより，低遅延なページテーブル参照と，リモート CPU でのデマンドページングへの対応を両立することができる．

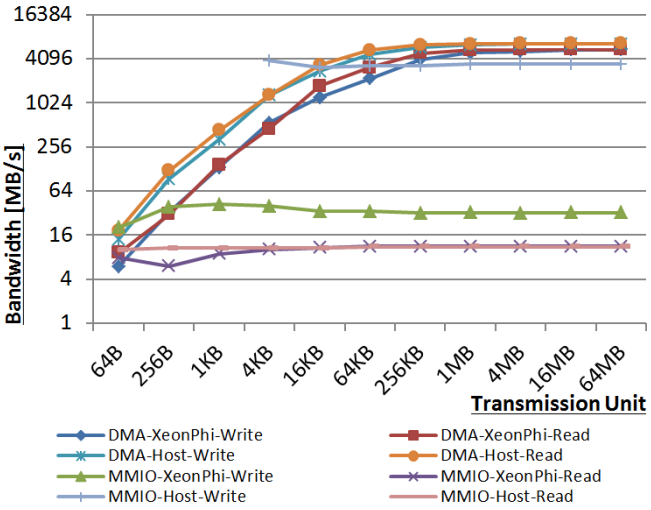

図 10 Xeon Phi - Host 間でのデータ転送帯域

7.2 リモート CPU の物理メモリへのアクセス方式

本研究では CPU ごとに物理メモリを備える計算機を対象としている．このため，リモート CPU 上の PVAS 空間に対するメモリアクセスを実現するためには，CPU をまたいだリモートメモリアクセスが必要となる．ヘテロジニアス型のメニーコア CPU である Intel 社の Xeon Phi など，メニーコアと Host を PCI Express バスで接続するアーキテクチャで利用できる 2 種類の CPU 間データ転送方式 [9] を以下に示す．

(a) MMIO（PIO）による直接アクセス

本手法はローカルメモリへのアクセスと同様，CPU 命令で直接リモートメモリへアクセスするものである．利点は CPU 命令においてローカルメモリとリモートメモリを等価に扱える点と，データコヒーレンシを CPU キャッシュを含めて保つことができる点である．欠点は，CPU 命令で一度に扱えるデータサイズが小さい（64bit 版 x86 プロセッサでは 8 バイト，拡張命令で 16 バイト）ことによる，大量データ転送時の帯域幅低下である．

(b) DMA による異なるメモリ間でのデータコピー

本手法はメモリ間でのデータ転送を行なうもので，リモートメモリから CPU のレジスタへ直接データを転送できる手法 (a) とは性質が異なる．利点は大量データ転送時の広い帯域幅である．欠点は，ハードウェア初期化オーバヘッドにともなう少量データ転送時の帯域幅低下と，ローカル・リモートに同一データのコピーが存在するためにデータコヒーレンシを維持できない点である．

Multiple PVAS の目的である PVAS Agent モデルを高効率に実現するためには，処理依頼に必要な数 10 バイト

程度のデータ転送を低遅延に実施する必要がある．またリモート CPU の物理メモリへのアクセスをローカル CPU の場合と同様の手順で行うためには，データのコヒーレンシを維持できるリモートアクセスを実現する必要がある．両方式のデータ転送性能を比較するため，Intel Xeon Phi と Host 間でデータ転送を実施した際の帯域幅を 8.1 節で述べる実装環境で測定した．DMA エンジンは Xeon Phi に搭載されており，Host からも MMIO で直接操作可能なものを用いた．結果を**図 10** に示す．グラフの項目名はデータ転送の方式とイニシエータ，および転送方向を示しており，例えば「MMIO-Host-Read」であれば，Host が主体となり Xeon Phi のメモリから Host のメモリへ MMIO で Read アクセスを実施したことを示す．

図 10 では大きな転送単位では圧倒的に DMA が広帯域だが，想定する転送サイズに近い 64 バイトでは，MMIO が DMA と互角か若干広帯域である．またデータコヒーレンシの観点から見ても，ハードウェアレベルでコヒーレンシを維持できる MMIO 方式が適切である．よって Multiple PVAS では MMIO を用いてリモートメモリアクセスを実施する．なお，PVAS Task 間で大量のデータ転送を広帯域に実施するための基盤として，PVAS Agent モデルに基づいた DMA Agent を別途検討している．少量データの転送は Multiple PVAS，大量データの転送は DMA Agent と異なる基盤を構築することで，ハードウェアを余すことなく活用した高性能なアプリケーション実行基盤を目指す．

7.3 CPU 間通信方式

Multiple PVAS では，メモリ管理におけるリモート CPU への物理メモリアサイン依頼と，リモート CPU の PVAS Task 管理機構呼び出しにおいて CPU 間通信を実施する．これらの通信を低遅延に実施することで PVAS Task 間通信や PVAS Task 生成管理オーバヘッドを削減する．

送信側・受信側の同期にともなう CPU 時間浪費を避けるため，メッセージキューを用いたメッセージパッシング型の非同期通信を実施する．また，メッセージキューを受信側 CPU の物理メモリ上に設け，送信側 CPU がリモートメモリアクセスによりメッセージを格納する設計とすることで，特権レベルの遷移を必要としない通信を実現する．通信ハードウェアを用いる通常の通信であれば，ユーザ空間にて通信を行なう場合にハードウェア呼び出しのためカーネルモードへの遷移が必要となるが，本方式ではユーザ空間のみで通信が完結するため通信遅延の削減が見込まれる．

送信処理はキューへメッセージを格納することで完了するが，受信処理ではキューに新たなメッセージが格納されたことを検知する必要がある．方式としてはキューのポーリングとハードウェア割り込みの 2 種類が考えられる．ポーリングを選択した場合，受信側 CPU コアの一部をポーリング用に割り当てる必要が生じるため，マルチコアやコア数がない少ないメニーコアではシステムの演算性能低下を招く恐れがある．また割り込みを選択した場合，ユーザ空間での通信に特権レベル切り替えが発生するほか，受信側 CPU でコンテキスト切り替えが発生するなどオーバヘッド増大が見込まれる．Multiple PVAS ではアプリケーション実行性能とオーバヘッド削減の両方を重視しているため，割り込みがポーリングと比較して同等か若干遅いようであれば割り込みを選択し，ポーリングが割り込みよりも圧倒的に高速である場合にはポーリングを選択すべきである．本論文ではポーリング方式のみの実装であるが，今後両方式の性能評価を実施した上で適切な方式を決定する．

8. 基礎評価

本章では Multiple PVAS の実装について述べ，メモリ管理や PVAS Task 間通信性能の評価結果について述べる．

8.1 実装

Intel のメニーコアプロセッサ Xeon Phi（5110P，60 コア 240 スレッド，RAM8GB）と，メニーコアプロセッサ Xeon（X5680 x2，12 コア 24 スレッド，RAM24GB）で構成したヘテロジニアス並列計算機上で Multiple PVAS を実装した．両プロセッサの OS には PVAS に対応した Linux カーネル（メニーコア: 2.6.38，マルチコア: 2.6.32）を用いた．メニーコア・Host 間でのリモートメモリアクセスには，Intel の SCIF（メニーコア・Host 間通信ライブラリ）[10] の機能を一部で利用しているが，Mulitple PVAS と SCIF で Xeon Phi のハードウェア操作に競合が生じるため，SCIF へ必要な改修を行なった．

7.1.3 節で述べたリモート CPU のページテーブル直接参照に備え，Xeon Phi の物理メモリ全量を予め Host のカーネル仮想アドレス空間へマップした．Host のメモリは Xeon Phi に比べて量が多く，全量をマップした場合にページテーブルが消費する物理メモリ量を無視できないため，Xeon Phi から Host のページテーブルを参照する場合にはオンデマンドでマップを実施する．

8.2 ページフォルト発生時のリモートページテーブル参照方式の性能比較

本評価では，リモート CPU 上の PVAS 空間へのアクセスにともなうページフォルトのオーバヘッドを 7.1.3 節で述べたリモート CPU のページテーブルの参照方式別に測定した．結果を参照方式，アクセスを行なう CPU 別に**表 2** へ示す．Xeon Phi が Host の PVAS 空間へアクセスする場合は間接参照方式が直接参照方式より約 4.8 倍高速である．一方，Host が Xeon Phi の PVAS 空間へアクセスする場合は直接参照方式が間接参照方式より約 3.5 倍高速となっ

表 2　ページフォルトのオーバヘッド
直接参照方式によりリモートページテーブルを参照する場合

測定項目	所要時間	
	Xeon Phi が Host へアクセス	Host が Xeon Phi へアクセス
A1: リモートページテーブルのマップ	49.31 us	-
A2: リモートページテーブル Walk	4.106 us	4.017 us
A3: ローカルページテーブルの設定	2.256 us	0.5433 us
合計	55.67 us	4.561 us

間接参照方式によりリモートページテーブルを参照する場合

測定項目	所要時間	
	Xeon Phi が Host へアクセス	Host が Xeon Phi へアクセス
B1: CPU 間通信によるエントリ照会	7.052 us	11.60 us
B2: リモート CPU でのページテーブル Walk	2.221 us	3.779 us
B3: ローカルページテーブルの設定	2.256 us	0.5433 us
合計	11.53 us	15.92 us

た．オーバヘッドとなる CPU 間通信を要さない直接参照方式が，Xeon Phi が Host の PVAS 空間へアクセスする場合に間接参照方式よりも低速となった要因は，リモートページテーブルのマップ処理（A1）である．Xeon Phi のページテーブルは Host 同様 4 段で構成され，全 4 ページ（16KB）のマップが必要となるほか，マルチコアに比べ低いクロック周波数や out-of-order 実行されない [7] ことから，マップ処理に多大な時間を要したと考えられる．Xeon Phi のコア単体性能の低さは，ローカルページテーブルの設定（A3/B3）で Host の約 4.2 倍の時間を要している点からも読み取れる．なお，間接参照方式において Xeon Phi が Host の PVAS 空間へアクセスする場合の所要時間が，Host が Xeon Phi の PVAS 空間へアクセスよりも約 1.4 倍高速である要因は，CPU 間通信の受信処理やページテーブル Walk がコア単体性能の優れる Host で実施されるためである．

以上の結果から，Host では直接参照方式と間接参照方式を組み合わせたページテーブル参照方式が性能上優れていることが検証された．Xeon Phi においては，直接参照方式のオーバヘッドが大きいため，間接参照方式のみを用いるのが性能上適切であると判明した．

8.3　異なる CPU 間での PVAS Task 間通信性能

PVAS Agent モデルをメニーコア混在型並列計算機上で実現することを想定し，Host と Xeon Phi それぞれに配置した PVAS Task 間での通信性能を評価した．PVAS

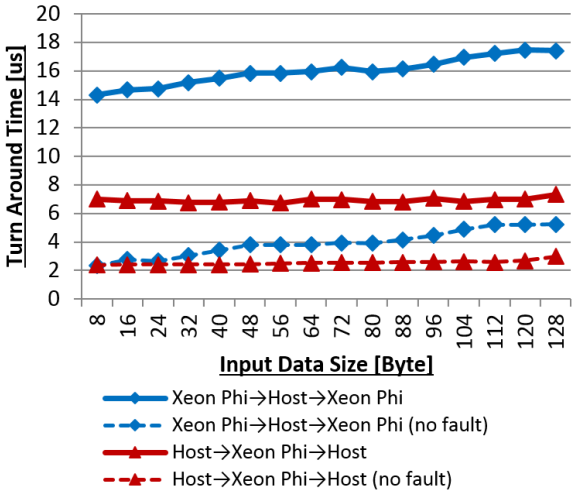

図 11　異なる CPU 間での PVAS Task 間通信の Turn Around 時間

Agent モデルでは Infiniband や MPI などの機能を提供する Agent に対して，PVAS Task が処理依頼と処理結果待ちを行なう．本評価では，Agent を想定した PVAS Task をリモート CPU 上に配置し，ローカル CPU 上の PVAS Task から可変長（8 バイトから 128 バイト）の入力データを渡し，固定長（8 バイト）の処理結果を受け取るまでの Turn Around 時間を測定した．データの受け渡しにはリモート CPU の PVAS Task のメモリを用いており，ローカル CPU の PVAS Task は入力データの書き込みと処理結果データの読み込み（ポーリング），リモート CPU の PVAS Task は入力データの読み込み（ポーリング）と処理結果データの書き込みを行なう．なお，リモート CPU のページテーブルの参照は，8.2 節の結果を受け Host では直接参照方式と間接参照方式を組み合わせた 2 段階の方式，Xeon Phi では間接参照方式により実施する．

図 11 に測定結果を示す．項目名が例えば「Host → Xeon Phi → Host」であれば，Host の PVAS Task が Xeon Phi の PVAS Task へ処理依頼を行なうことを示す．また，項目名の「no fault」は一度アクセスが行われた仮想アドレスを用いて PVAS Task 間通信を行なった測定結果であり，ページフォルトが発生しないためメモリ間のデータ転送のみが所要時間に含まれる．図 11 では，入力データサイズが 8 バイトの際の所要時間が，no fault 同士では差が見られないのに対し，ページフォルトをともなう場合の Host → Xeon Phi → Host が Xeon Phi → Host → Xeon Phi より約 2.1 倍高速な結果となった．これは Host → Xeon Phi → Host でのページフォルト処理が，直接参照方式により低遅延に完了するためである．また，Xeon Phi → Host → Xeon Phi における入力データサイズと所要時間がほぼ比例関係である一方，Host → Xeon Phi → Host では所要時間の増加が見られない．これは Host から Xeon Phi のメモリに対して入力データを書き込む際，CPU にて入力デー

タを集約化した上で PCIe バスへデータを送信する Write Combining 機能 [11] が作用するためである．

ページフォルト発生の有無で 2 種類の所要時間測定結果を示したが，PVAS Agent への処理要求は Export Segment という決まった領域で行なうため，最初の処理要求以降はページフォルトを要さない．図 11 の no fault の測定結果と Host のメモリアクセス帯域を踏まえて机上計算すると，Xeon Phi から Host に対して 1 秒間あたり 1 コアで約 43 万回，60 コアで約 2600 万回の処理要求（8 バイト）が可能である．多数の PVAS Task から一つの Agent へ処理要求を発行する場合の所要時間を評価する必要があるが，机上計算の結果から 1 秒あたり約 2600 万回もの MPI やその他 I/O などの要求を出す並列アプリケーションに対して，Multiple PVAS は Host の高いコア単体性能を活かした機能並列の基盤を提供できると考えている．

本評価結果より Xeon Phi から Host への処理依頼が 2〜5us，ページフォルト発生時に 14〜17us 程度で，Host から Xeon Phi への処理依頼が 2〜3us，ページフォルト発生時に 7us 程度で実現可能であるという指針を得られた．Xeon Phi に直接接続できない I/O 機器へのアクセスを Host へ依頼することを考えた場合，IB での MPI Ping 遅延が 1us[12]，MLC 型 SSD へのアクセス遅延が 80us 程度 [13] であることから，IB では一定の転送サイズが必要となるものの，Multiple PVAS を適用することで顕著な性能低下なしで Host への I/O 依頼を実現できる．また，コア単体性能の高い Host へ並列化できない処理を依頼することを考えた場合，表 2 の A3/B3 のように Host が Xeon Phi より 4 倍程度高速な処理では，Xeon Phi での処理時間が 3〜7us を超えるようであれば所要時間短縮を実現できるといえる．Xeon Phi 上での MPI 集団通信（Alltoall，128 プロセス，16 ノード）に 500us 以上を要したという報告 [14] があることから，Multiple PVAS は MPI の集団通信や集団 I/O の性能向上へ適用可能であると考える．

9. 関連研究

複数の計算機の物理メモリを包括する単一の論理メモリを構築する分散共有メモリ（DSM）は，メニーコア・Host が混在する計算機上で大域仮想アドレス空間を構築する本研究に似ている．Mermaid[15] はアーキテクチャの異なる 2 つの計算機で構成したヘテロジニアス型システムにおいて DSM を構築している．また，Gelado らは Host とアクセラレータで構成されたヘテロジニアス型計算機で DSM を提案している [16]．両研究ともヘテロジニアス型のシステムを対象としている点で本研究と同様であるが，Mermaid はエンディアンや浮動小数点表現の差異に着目した研究であるほか，Gelado らは Host からアクセラレータへの単方向メモリアクセスを対象としている点で，それぞれ本研究とは異なる．

Intel 社はメニーコア・Host 間で共有メモリ領域を構築する MYO [17][7] を提供しているが，共有メモリの生成やアクセス制御をプログラマが明示的に実施する必要があり，PVAS Task 間通信を CPU 配置を考慮せずに可能とすることを目指した本研究とは方向性が異なる．また，MYO では共有メモリへのアクセス時にデータをローカルメモリにコピーする点が，MMIO による直接アクセスでデータアクセス遅延の低減を図った本研究と異なる．

10. おわりに

本研究では，メニーコア混在型並列計算機における PVAS Agent モデルの実現を目的に，メニーコアと Host を包括する Multiple PVAS 空間を提案した．CPU 内，CPU 外を含めたメモリアクセスを実現する大域仮想アドレス空間をアプリケーション実行性能を重視したページテーブル構成や，オーバヘッド削減を図ったページテーブル一貫性維持方式により実現し，Xeon Phi から Host へ 2〜17us 程度で，Host から Xeon Phi へ 2〜7us 程度でリモート CPU 上の PVAS Task への処理依頼が可能であるという結果を得た．本指標を元に，今後はヘテロジニアス型計算機の特性を活かした各種 Agent を開発し，アプリケーション実行性能向上を目指す．

謝辞 本研究は，科学技術振興機構（JST）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」研究課題「メニーコア混在型並列計算機用基盤ソフトウェア」によるものである．

参考文献

[1] TOP500 Supercomputer Sites: June 2013, http://www.top500.org/lists/2013/06/ (2013).
[2] 科学技術振興機構：CREST, http://www.jst.go.jp/kisoken/crest/.
[3] 堀敦史, 島田明男, 並木美太郎, 佐藤未来子, 深沢豪, 辻田祐一, 石川裕：メニーコア用 Agent プログラミング環境の提案，情報処理学会「ハイパフォーマンスコンピューティング研究会」第 140 回研究報告，Vol. 2013-HPC-140, No. 32, pp. 1–8 (2013).
[4] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: Proposing a new task model towards many-core architecture, *Proceedings of the First International Workshop on Many-core Embedded Systems*, MES '13, New York, NY, USA, ACM, pp. 45–48 (online), DOI: 10.1145/2489068.2489075 (2013).
[5] 島田明男，バリゲローフィ， 堀敦史， 石川裕：メニーコア OS 向け新プロセスモデルの提案，情報処理学会「ハイパフォーマンスコンピューティング研究会」第 135 回研究報告， Vol. 2012-HPC-135, No. 3, pp. 1–8 (2012).
[6] Sato, M., Fukazawa, G., Yoshinaga, K., Tsujita, Y., Hori, A. and Namiki, M.: A Hybrid Operating System for a Computing Node with Multi-Core and Many-Core Processors, *International Journal of Advanced Computer Science (IJACSci)*, Vol. 3, No. 7 (2013).

[7] Reinders, J.: *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors*, Intel (2012).
[8] Green, R. W.: Native and Offload Programming Models, `http://software.intel.com/en-us/articles/native-and-offload-programming-models` (2012).
[9] Intel: *Intel Xeon Phi Coprocessor System Software Developeres Guide* (2013).
[10] Intel: *Symmetric Communications Interface (SCIF) For Intel Xeon Phi Product Family Users Guide* (2013).
[11] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual* (2013).
[12] Mellanox Technologies: *ConnectX-3 VPI Product Brief* (2013).
[13] Intel: *Intel Solid-State Drive 525 Series Product Specification* (2013).
[14] Panda, D. K.: MVAPICH2 for Intel MIC, OFA Developers Workshop (2013).
[15] Zhou, S., Stumm, M., Li, K. and Wortman, D.: Heterogeneous Distributed Shared Memory, *IEEE Trans. Parallel Distrib. Syst.*, Vol. 3, No. 5, pp. 540–554 (online), DOI: 10.1109/71.159038 (1992).
[16] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N. and Hwu, W.-m. W.: An asymmetric distributed shared memory model for heterogeneous parallel systems, *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, New York, NY, USA, ACM, pp. 347–358 (online), DOI: 10.1145/1736020.1736059 (2010).
[17] Yan, S., Zhou, X., Gao, Y., Chen, H., Wu, G., Luo, S. and Saha, B.: Optimizing a shared virtual memory system for a heterogeneous CPU-accelerator platform, *SIGOPS Oper. Syst. Rev.*, Vol. 45, No. 1, pp. 92–100 (online), DOI: 10.1145/1945023.1945035 (2011).