

メニーコア用 Agent プログラミング環境の提案

堀 敦史^{1,a)} 島田 明男^{1,b)} 並木 美太郎^{2,c)} 佐藤 未来子^{2,d)} 深沢 豪^{2,e)} 辻田 祐一^{1,f)}
石川 裕^{1,3,g)}

概要: 本稿では、メニーコアの OpenMP 性能評価をベースに新しい機能並列を実現し、エクサスケールに向けたメニーコア、マルチコア混在型並列計算機のための新しい並列プログラミング環境を提案する。提案する環境は *PVAS Agent* と呼ばれ、我々が既に提案しているノード内並列のための新しいタスクモデル Partitioned Virtual Address Space 上に実現される。実装された PVAS Agent は、機能並列のための基礎的なオーバーヘッドが計測され、今後の応用プログラム開発のための指針を与えることができた。また、混在型アーキテクチャへの拡張についても検討をおこなった。

ATSUHI HORI^{1,a)} AKIO SHIMADA^{1,b)} MITARO NAMIKI^{2,c)} MIKIKO SATO^{2,d)} GO FUKAZAWA^{2,e)}
YUISHI TSUJITA^{1,f)} YUTAKA ISHIKAWA^{1,3,g)}

Abstract: This paper proposes a novel parallel programming environment for the exa-scale heterogeneous computing environment where computing nodes are consisting of multi-core and many-core CPUs. The proposed environment is named '*PVAS Agent* running on top of a new task model named "Partitioned Virtual Address Space" which has been proposed by authors. The basic performance of the implemented PVAS Agent is evaluated using micro-benchmark programs. The results reveal the hints how the applications of PVAS Agent should be implemented. It is also argued that the augmentation for the heterogeneous architectures.

1. 研究の背景

我々は、JST/CREST の「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送達」研究領域において、「メニーコア混在型並列計算機用基盤ソフトウェア」の研究開発を進めている [18]。ここでは、今後メニーコア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、そこで使われるシステムソフトウェアの研究開発を行っている。我々が想定しているメニーコア混在型並列計算機では、図 1 で示されるような計算ノードが、高性能なネットワークにより多数結合されている構成である。こ

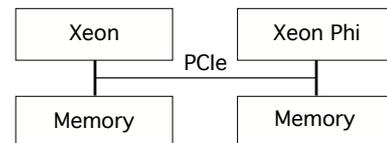


図 1 メニーコア混在型計算ノード

の図に示される構成では、ホスト CPU には通常のマルチコア CPU が用いられ、PCIe バスに別途メニーコア CPU が接続されており、どちらの CPU も個別のメモリを持っている。また、OS のサポートにより互いのメモリを PCIe を経由して参照することも可能になっているものとする。

メニーコア CPU を用いて高性能な並列計算環境を構築する際の鍵は、従来とは異なる設計思想に基づいたメニーコア CPU の活用にある。計算ノード間の通信ハードウェアに関しては従来の延長として考えられるが、メニーコアのコア間通信や、通信ハードウェアをメニーコアとしてどのように扱うという点が重要である。このような背景を踏まえ、本稿の目的は大きく以下の 2 点がある。

- 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

¹ 独立行政法人理化学研究所計算科学研究機構
RIKEN AICS
² 東京農工大学
Tokyo University of Agriculture and Technology
³ 東京大学情報基盤センター
Information Technology Center, University of Tokyo
a) aho@riken.jp
b) a-shimada@riken.jp
c) namiki@cc.tuat.ac.jp
d) mikiko@namikilab.tuat.ac.jp
e) fzawa@namikilab.tuat.ac.jp
f) yuichi.tsujita@riken.jp
g) ishikawa@is.s.u-tokyo.ac.jp

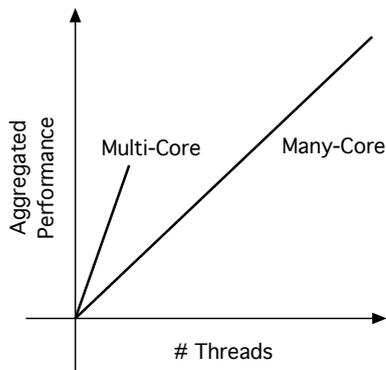


図2 メニーコアとマルチコアのマルチスレッド性能

本節では、メニーコア CPU として最初に商品化された Xeon Phi におけるマルチスレッド性能についての基礎性能を測定し、次いで我々が既に提案している Partitioned Virtual Address Space (PVAS) について簡単に紹介する。次節以降において、上記の目的を満たすための設計された PVAS Agent について提案し、関連研究に次いで、提案した PVAS Agent の予備評価をおこなう。最後に、メニーコアとマルチコアの連携機構を自然に実現するための PAVS Agent のアイデアについて述べる。

1.1 Xeon Phi の OpenMP 性能

図1にある Xeon Phi は昨年末にメニーコア製品として世に最初に出された Intel 社の製品である [12]。メニーコアアーキテクチャでは、非力だが消費電力に対する性能に優れた小さいコアが多数チップ内部のネットワークで結ばれている。したがって、コア単体の性能は、現在広く使われているマルチコアの性能には及ばないが、総体としてメニーコアの性能を上回ることが期待されている (図2)。特筆すべき点は Xeon Phi には 60 もの計算コアがあり、そのそれぞれで 4 way の SMT (Simultaneous Multi-Threading, Intel は "Hyper Threading" と呼んでいる) をサポートしており、Xeon Phi 全体として 240 ものスレッドを同時に実行できることである*1。また、60 のコアのそれぞれにキャッシュを持ち、コア間でキャッシュのコヒーレンシーが保たれている。したがって、Xeon Phi 上でプログラムする場合、OpenMP などの従来の並列プログラミング手法をそのまま用いることができる。

図3は NAS 並列ベンチマークプログラム (以下、"NPB") [1] の OpenMP 版 (Version 3.3) にある IS, LU, FT, EP, SP, MG, CG, BT の性能を、横軸にスレッド数、縦軸にプログラムが出力する性能値 ("MOPS total") をそれぞれプロットしたものである。表1に本稿で用いられた Xeon Phi のシステム構成と評価環境を示す。この評価実験の目的は、主にマルチコア用に使われて来た OpenMP が Xeon Phi でどのような挙動を示すかを明らかにし、実際

*1 実際のコアの数は、Xeon Phi の製品により若干異なる。

表1 Evaluation Environment

Xeon Phi	5110P Stepping: B1
GDDR	8GB
# Cores	60
uOS (Linux)	Version 2.6.38.8-g9b2c036
Intel Compiler	Version 13.1.0.146
Compile Options	-O3 -mmic -simd -openmp -parallel
NAS Parallel Benchmarks	NPB-OMP Version 3.3

に 240 ものスレッドを実行した時に、全てのスレッドを効率よく利用できるか否かという点にある。NPB には問題サイズを選択できるが、ここではメモリ (8GB) の許す限りなるべく大きい問題を実行した。図中、例えば "IS.C" とあるのは、IS ベンチマークプログラムのクラス C の問題サイズでの実行を示す。なお、意図的に KMP_AFFINITY は scatter に設定し、特に SMT の挙動を調べることに重きを置いた [11]。こうすることで、横軸の、1-60 の範囲では、それぞれのコアに高々ひとつの SMT が走り、61-120 までは 2 つの SMT が、121-180 では 3 つの SMT、そして 181-240 では 4 つの SMT が走る。文献 [12] では、Xeon Phi のコアの特性から少なくとも 2 つの SMT が必要である、としている。この評価の目的は、コア当たりの SMT の数が 2, 3 あるいは 4 のうちどれがベストであるか、という点である。

図3において、左下の CG.C のグラフを除き、多くの場合で 180 スレッドまでの範囲で性能のピークを見て取る事ができる。一方、CG.C の性能はスレッド数に比例して性能が伸びている。CG は疎行列の計算であり、メモリアクセスがボトルネックとなっていることが判明したため、別途プリフェッチ命令を挿入して計測した ("CG-pre.C")。この結果、プリフェッチを入れる前よりも大幅に性能が向上すると同時に、180 スレッドまでの性能のピークと 240 までの性能ピークの差が小さくなっている。図3には他にもいくつかの特徴がみられるが、この点に関しては別な報告の機会を設けることとし、本稿の範囲外とする。

1-180 スレッド数の範囲のピーク性能と (コア当たりの SMT 数が最大 3)、181-240 スレッド数の範囲 (コア当たりの SMT 数が最大 4) のピーク性能を比べると、1-180 の方が性能が高いケースは IS.C, LU.C, BT.C, SP.C, FT.B, MG.B の 6 プログラム (プログラム数は全部で 8)、逆のケースは EP.C, CG.C (CG-pre.C) の 2 プログラムだけである。さらに、1-180 の性能と 181-240 の性能の比を比べると、1-180 の方が性能が高い場合は最大性能の比率は 15%、逆の場合の最大性能比は 10% である。このことから、Xeon Phi の OpenMP プログラムにおいてはコア当たり 3 スレッドの実行が多くの場合で適切と考えられる。また同様の結果は Cramer らからも報告されている [2]。

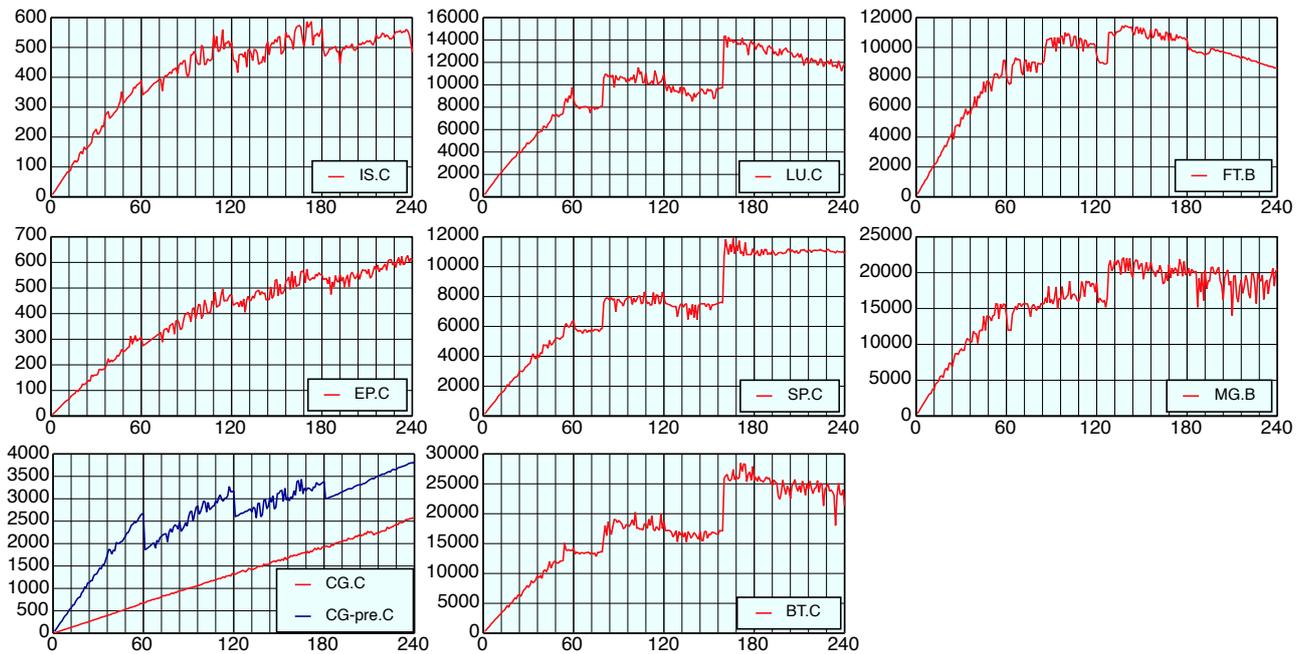


図 3 Xeon Phi における OpenMP スレッド数の違いによる NPB 性能の変化

1.2 Partitioned Virtual Address Space

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の提案である [13], [21]. マルチプロセスによるプロセス間通信のオーバーヘッドを回避すると同時に、マルチスレッドにおける共有資源の競合を回避することができる (図 4). 図 4 は、我々が既に提案している PVAS のアドレスマップを示している. 通常の Linux のプロセスでは、他のプロセスのメモリを直接アクセスすることはできない. 別途、カーネルが提供する機能を用いる必要があり、通常、それにはなんらかのオーバーヘッドが伴う. 一方、マルチスレッドでは、個々のスレッドは同じアドレス空間に存在するが、DATA や BSS といった大域的なデータはプログラマの意志とは関係なく共有されてしまい、競合を回避するためのプログラミング上の工夫や、排他制御などを導入するオーバーヘッドが存在する.

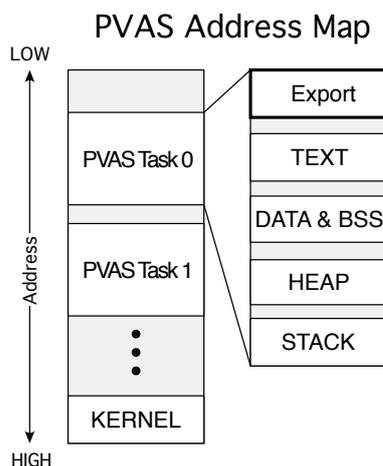


図 4 Partitioned Virtual Address Space

メニーコアにおいては、ひとつの計算ノード内での並列度は高く、実行の実体をプロセスで実現してもスレッドで実現しても、上記のようなオーバーヘッドが伴う. PVAS は、ひとつのアドレス空間に複数のプロセスを配置したもの、あるいは、スレッドで共有されている DATA, BSS, HEAP 領域を全てスレッド固有のプライベートとしたもの、と考えることができる. このため、上記の問題の多くを改善できると考えている. PVAS では実行の実体を "PVAS task" と呼んでいる.

PVAS における通常の PVAS task の実行では、その PVAS task に割り当てられたアドレス領域 ("PVAS segment") の中だけで実行されるが、他の PVAS task のアドレスも必要に応じてアクセスすることが可能であり、これにより PVAS task 間の情報の受け渡しや同期が実現可能になっている. PVAS task はアドレス空間の低い方から連続に番号が割振られており、これにより同じアドレス空間にある PVAS task を識別することができる. また、PVAS segment の先頭には、"Export region" と呼ばれる領域があり、これは PVAS task の識別番号からそのアドレスを知ることができるため、他の PVAS task との情報の受け渡しは同期のために領域として使うことを目的としている.

PVAS では実行の実体を同じアドレス空間に配置することで不要なデータのコピーを伴わずに情報のやり取りが可能である一方、データの共有をできるだけ排除することでプログラミングの容易さを実現している. エクサ規模の並列計算機を構築する上で最も大きな障害は消費電力であり、データのコピーや移動における消費電力は計算に必要な電力よりも大きいことが指摘されている [3]. PVAS はデータのコピーを排除することで、計算効率や電力効率の

向上を実現できるものと考えている。

1.3 PVAS Agent の提案

第 1.1 節で示したように、OpenMP を用いたマルチスレッドによるデータ並列で Xeon Phi の全てのハードウェアスレッドを有効活用することは難しい。そこで、データ並列で利用しきれない 60 程度のハードウェアスレッドを機能並列あるいは HelperThread (例えば, [8], [9], [10]) として利用し、実装としては PVAS を用いる方式を考え、これを *PVAS Agent* と名付ける (図 5)。

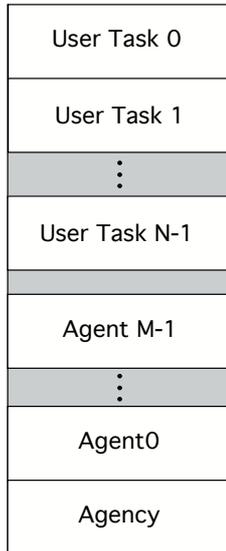


図 5 PVAS Agent

ここで機能並列のための PVAS task は *agent* と呼ばれる。ユーザプログラムの並列実行は、複数の PVAS task として実行される。別途、agent 用の PVAS task も同時に存在する。ユーザの PVAS task は PVAS 識別子の小さい方から、agent の PVAS task は PVAS アドレス空間の最も大きいアドレスのほうから順に割り当てられる。PVAS アドレス空間の最下部には、”(PVAS) Agency” と呼ばれる agent があり、Agency 以外の agent を管理する役割を担うが、ユーザプログラムからは陽には見えない存在である。Agent の生成や削除や Agency に依頼する。Agency は常に PVAS の最大識別子に位置するため、その export region のアドレスは既知である。

Agency を通じて動的に生成された agent は、agent に定められたある特定の機能を実現する。Agent は PVAS task であるため、ユーザプログラムとは別な実行の実体を持つ。ユーザプログラムは望みの機能を実現するための agent に対し、要求を発行し、その結果を待つ。実際には、agent の export region に要求を書込み、そこから結果を得る、という動作になる。Agent が具体的にどのように export region を使ってユーザプログラムとやり取りするかは、agent の実装次第であるが、一般的には有限なキュー構造を想定し

ている。キュー構造とすることで、

- Agent の処理が逐次化されるため、agent 内で排他制御の必要がなくなる
 - ユーザプログラム側からは、要求が非同期で処理されるため、他の処理との実行のオーバーラップが可能となる
 - ひとつの機能を複数の agent で実現した場合、互いに work stealing が可能となり、効率的な処理が実現可能である
- などといった利点が考えられる。

また、他の PVAS task のデータを自由にアクセス可能という PVAS の特長から、大きな配列や複雑なデータ構造もポインタとしてキューに入れることが可能であり、不要なデータコピーを回避することができる。

2. 既存の技術や研究との関連

PVAS Agent は並行自律的にある機能が実現されるという観点からは Actor 言語 (例えば [16]) やエージェント指向言語と類似性があるが、PVAS Agent はオブジェクト指向的な意味あいは薄く、また処理の粒度としてはオブジェクト指向のそれよりも大きなものを想定している。

処理粒度の観点からは、分散処理における client-server モデルに近い。しかしながら、多くの client-server モデルの実現では、プロセスが用いられ、クライアントとサーバの間で通信 (データのコピー) が発生してしまう。

Helper Thread では、メインとなる計算があって、その効率を高めるために別なスレッドを補助的に用いる。PVAS Agent では、agent を Helper Thread として用いることも可能であるし、補助ではなく、全く別な機能を実現する場合に用いることも可能である。

PVAS Agent の動作は動的ライブラリのそれに近い。動的ライブラリでは、コード片を動的に読込み、同じアドレス空間に配置し、ある機能を実現する。PVAS Agent の動作は常に自律的かつ並列/並行的である点が大きく異なっている。

PVAS Agent に最も近い概念としては、FlexSC[14], [15] (似たような考え方の OS として GenerOS[17] もある) が挙げられよう。FlexSC は本来は OS の構成手法のひとつであるが、PVAS Agent は FlexSC のユーザレベルでの実現と考えることもできる。

3. PVAS Agent の予備評価

本説では、実装された PVAS Agent の基本性能を知るために、要求があつたら直ちに値 (int, 4 バイト) を返すという null agent を生成し、その処理速度を Xeon (Ivy Bridge Xeon E3-1280 V2, 3.60GHz, 4 cores, 1 socket, 2-way SMT) と Xeon Phi (表 1) の上で計測した。

図 6 に計測に用いたユーザプログラムを示す。このプ

```
for( i=0; i<LOOP; i++ ) {
    pvasagent_eval( agent_null, req ); /* 同期 */
    pvasagent_wait_request( req );
}
```

図 6 Null Agent の同期呼出しプログラム

```
for( i=0; i<LOOP; i++ ) {
    pvasagent_eval( agent_null, NULL ); /* 非同期 */
}
```

図 7 Null Agent の非同期呼出しプログラム

プログラムは null agent に対する同期呼出しの速度を、図 7 に示したプログラムでは null agent に対する非同期呼出しの速度を計測する。繰り返し回数は以下全てのケースで 1,000,000 回とした。図 8 および図 9 は Xeon における呼出し速度（縦軸：1 マイクロ秒における処理の回数）を agent とユーザプログラムの論理コアへのバインドを X-Y 軸としてそれぞれプロットしたものである。ただし、agent とユーザプログラムを同じ論理コアに配置した場合は計測から除外し、グラフ上はゼロとなっている。

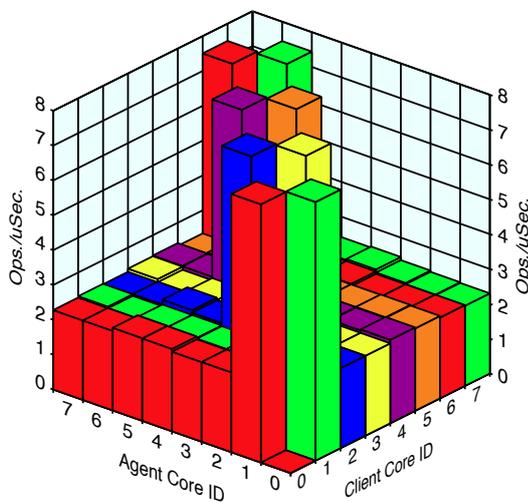


図 8 Xeon 上での PVAS Agent の同期呼出し速度

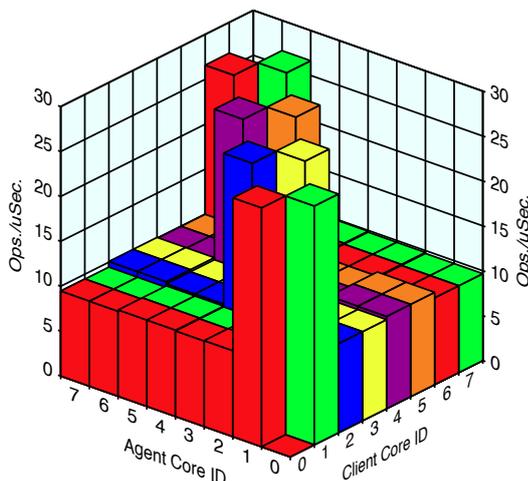


図 9 Xeon 上での PVAS Agent の非同期呼出し速度

図 10 および図 11 は Xeon Phi における呼出し速度を、先の図と同様にプロットしたものである。グラフ作成上の都合から、これらのグラフでは論理コア 0-60 の範囲でのみ表示した。

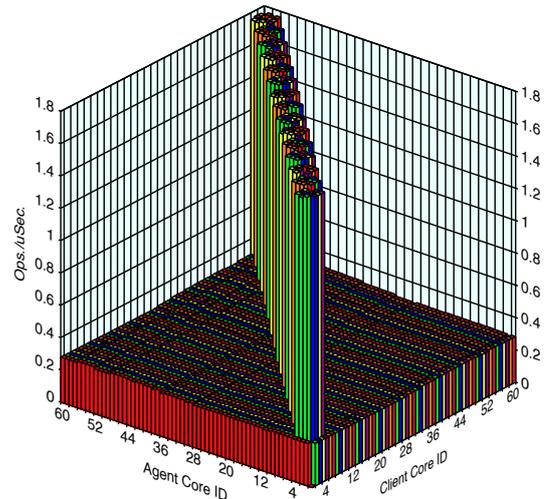


図 10 Xeon Phi 上での PVAS Agent の同期呼出し速度

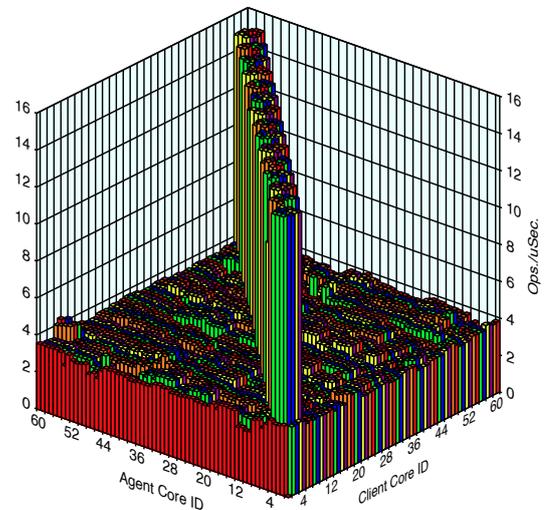


図 11 Xeon Phi 上での PVAS Agent の非同期呼出し速度

Xeon でも Xeon Phi でも、agent とユーザプログラムを同じ物理コアに配置した場合（図 8、図 9、図 10 および図 11 における対角線のすぐ隣の部分）の性能が高い。これは同じ物理コアの同じキャッシュを共有しているためと考えられる。逆に、この対角領域を除いた部分では大きな差は見られない。

表 2 は図 8、図 9、図 10 および図 11 の中からある特定のコアバインドにおける呼出し 1 回あたりの時間（先のグラフの縦軸の値である処理速度の逆数）を示したものである。Xeon と Xeon Phi の処理速度を比べた場合、同期呼出しの同一物理コアの場合で約 4 倍、異なる物理コアの場合で約 9 倍、Xeon phi の方が遅いことが分かる。Xeon Phi のコアの単体性能は、クロック周波数が Xeon に比べ遅い、

命令発行に2クロックサイクルを要する, out-of-order 実行されないこと [12], などの影響により Xeon コアの単体性能に較べ大きく劣り, このような結果になったと推察される. 一方, 非同期の場合では, 同一物理コアで2倍弱, 異なる物理コアで3倍弱となっている. 非同期の場合に速度差が小さくなる現象についてはより詳細な調査が必要である.

表 2 呼出し 1 回に要する時間

	同期		非同期	
	同じ物理コア	異なる物理コア	同じ物理コア	異なる物理コア
Xeon	0.14	0.42	0.04	0.11
Xeon Phi	0.57	3.8	0.07	0.29

単位: マイクロ秒

表 2 の結果から, agent とユーザプログラムのコア割当の指針を得ることができる. Agent に依頼する処理において遅延時間が重要となる場合は同じ物理コアに割り当てる方が良い. 一方, キャッシュ汚染を回避したい場合には, 別な物理コアを割り当てた方が良い.

4. Multiple PVAS とその上での PVAS Agent

第 1 で述べたように, 本研究の最終的なターゲットは, マルチコア CPU とメニーコア CPU が混在するアーキテクチャである. 本説では, これまでに我々が提案してきた PVAS および PVAS Agent をこのようなアーキテクチャ上で有効に機能する方式について考える.

Multiple PVAS は, 複数の PVAS アドレス空間をひとつにするというアイデアである. この時, ひとつにまとめる前に PVAS アドレス空間は, 例えば, メニーコア CPU で, もうひとつはホスト CPU であっても構わない. 制約としては, なんらかのバスや通信機構を経由して相手のメモリにアクセス可能であれば, 相手のメモリをうまくマッピングすることで実現が可能になる. これは, 例えば, ひとつの計算ノードに複数の Xeon Phi が存在するような場合でも適用可能である.

図 12 は Multiple PVAS 上に PVAS Agent を適用した例を示す. この図では, 目的の並列計算をおこなうユーザタスクが Xeon Phi 上で走り, その計算を補助する PVAS Agent 群が, Xeon Phi 上とホスト CPU である Xeon 上の両方で動作している. PVAS Agent を経由することで, ユーザプログラムは, agent が Xeon Phi なのか Xeon なのかを区別することなく, 使うことが可能になる.

5. PVAS Agent の応用

図 13 は, これまで説明してきた PVAS Agent の応用例を示したものである. 図中 “MPI Agent” とあるのは MPI

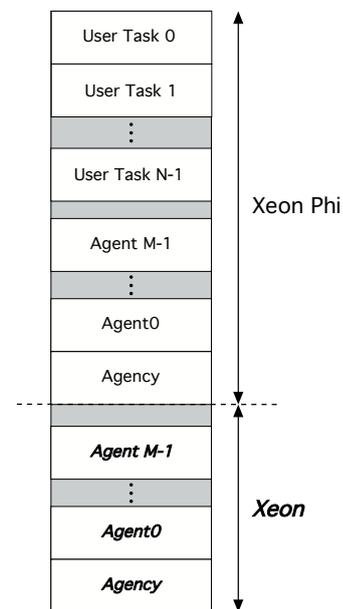


図 12 Multiple PVAS 上での Agent

の機能を提供する agent, “IB Agent” とあるのは Infiniband[5] による通信機能を提供する agent, “IO Agent” はファイル I/O 機能を提供する agent である.

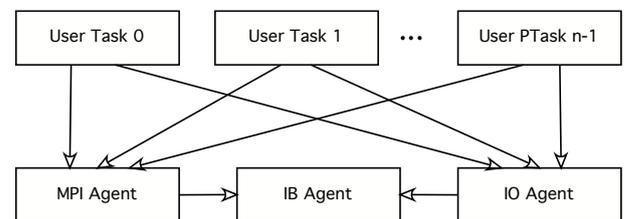


図 13 PVAS Agent の応用

各 agent はユーザタスクから呼び出されるだけでなく, 例えば IB Agent のように他の agent から呼び出すことができる. また, 例えば, IB Agent や IO Agent をホスト CPU 側におくことで, ホスト CPU や OS の長所を活かすことが可能になる. 特に, Xeon Phi は単体コア性能が低いため, ある程度粒度が大きくて並列化が難しい処理はホストに依頼した方が有利と考えられる. 一方, MPI のような従来ライブラリとして提供されてきたものを agent とする理由としては, MPI-3 で提供される非同期集団通信で特に有効と考えられるからである. 非同期集団通信においては, プロトコルの処理をユーザプログラムの実行と並列におこなう必要があるため, agent で MPI を実現するとこの問題を解消することができる.

IB ganet の実現の例では, 複数のユーザタスクからの通信要求をひとつ agent で処理するため, 受信の扱いが困難になると予想される. この問題に関しては, Infiniband の RDMA 機能のみをサポートすることで解決可能である. 京コンピュータの Tofu ネットワークでは RDMA しサポートされておらず, RDMA だけで MPI の機能を

実現することが実証されている [19]. また, IB agent がユーザとは別なタスクとして動作するため, 完了キュー (Completion Queue) の刈り取りは簡単に実現できる. IB Agent とする場合の最も大きな利点は, IB Agent のみが end-point を持つことで, ノード内の全てのタスクへの通信サービスを提供できる点であると考えられる. この結果, 大規模なクラスター型並列計算環境においても, end-point の総数を減らすことができる. 特にメニーコアにおいてはノード内のタスク数が従来よりも桁違いに多くなる可能性があり, IB Agent とすることでメモリ資源の節約に大きく貢献すると期待される. そして, この特性はユーザプログラムが flat MPI でも, OpenMPI と MPI の hybrid 並列でも失われない.

Multiple PVAS において, メニーコアとマルチコアのそれぞれのメモリはアクセス可能ではあるが, 実際のアクセスは PCIe バスを経由するため直下のメモリアクセスより遅くなる. 場合によっては, 両者のメモリ間でハードウェア DMA を用いてコピーをおこなう DMA Agent の必要性も検討に値するであろう.

6. 議論

Intel 社は Xeon Phi のノード内 (Xeon Phi のプロセス間, および, Xeon Phi とホスト CPU 間) 通信のための通信ライブラリ SCIF [7] を提供している. 基本的に SCIF は通信時に相手の CPU に対しハードウェア割り込みをかけるため, そのオーバーヘッドは無視できない. 第 1.3 節で述べたように, PVAS Agent における agent との情報のやり取りは基本的にキュー構造であり, 待ちを表現するフラグをビジーウェイトすることで同期を実現している. これによりハードウェア割り込を使わずに済み, 低いオーバーヘッドで同期や待ちが実現できる. 本稿では PVAS Agent の API についての詳細は省略したが, 複数の要求を取り扱うためのオブジェクトがあり, このオブジェクトのフラグをビジーウェイトすることで, 複数の要求に対する待ちも実現されている.

しかしながら, ビジーウェイトは消費電力という観点からは望ましくない. Intel 社のプロセッサには monitor および mwait 拡張命令がサポートされている [6]. これは, monitor 命令で指定されたアドレスのメモリに書込みがあった場合, mwait 命令による待ちが終了する, という仕様である. 我々は試験的にこれらの命令を用いて Shadow と呼ばれるマルチスレッドライブラリを既に開発しており [4], [20], この結果から省電力なビジーウェイトが monitor/mwait 命令により実現できることを確認済みである. 現時点では, monitor/mwait 命令は特権命令であること, Xeon Phi ではサポートされていないこと, という状況であるが, 上記のビジーウェイトによる待ちの実装は, 近い将来 monitor/mwait 命令がメニーコアやマル

チコア上で非特権命令の通常の命令として実行可能になった場合を想定してのことである.

7. まとめと今後について

以上, メニーコアである Xeon Phi 上での OpenMP 性能評価をベースに, 機能並列の必要性があること示し, 機能並列を実現するために我々が提案する PVAS 上で PVAS Agent と呼ばれる計算ノード内並列プログラミング環境を提案した. 提案された PVAS Agent はマルチコアである Xeon とメニーコアの Xeon Phi 上でそれぞれその基本性能を評価した. 評価の結果, Xeon および Xeon Phi の両方で, agent を計算タスクと同じ物理コア上に配置すべきかどうかを決める指針として, 処理の粒度 (遅延時間) とキャッシュ汚染に配慮すべきとの結論を得た. また, 本研究の最終目標であるメニーコア混在型並列計算機にたいしては, PVAS を拡張した Multiple PVAS と, その上で動作する PVAS Agent システムを提案した. 今後は, Multiple PVAS の実装を進め, その上に本稿で提案した IB Agent, MPI Agent, IO Agent などの開発を進める予定である.

謝辞 本研究は, 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] Bailey, D. H.: The NAS Parallel Benchmarks, *Encyclopedia of Parallel Computing* (Padua, D., ed.), Springer (2011).
- [2] Cramer, T., Schmidl, D., Klemm, M. and Mey, D.: OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison, *MARC @ RWTH'2012 : Many-core Applications Research Community Symposium 2012* (2012).
- [3] Dongarra, J., Choudhary, A., Kale, S. et al.: The International Exascale Software Project Roadmap, White paper, Argonne National Laboratory (2010).
- [4] Hori, A. and Ishikawa, Y.: Poster: MINT: a fast and green synchronization technique, *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion, SC '11 Companion*, New York, NY, USA, ACM, pp. 23–24 (2011).
- [5] Infiniband Trade Association: InfiniBand, <http://www.infinibandta.org/>.
- [6] Intel Corp.: *Intel 64 and IA-32 Architectures Software Developer's Manual* (2011).
- [7] Intel Corp.: *Intel Many Integrated Core Symmetric Communications Interface (SCIF) User Guide*, Revision 1.0 edition (2012).
- [8] Jung, C., Lim, D., Lee, J. and Solihin, Y.: Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems, *In Proceedings of 20th IEEE International Parallel & Distributed Processing Symposium* (2006).

- [9] Kamruzzaman, M., Swanson, S. and Tullsen, D. M.: Inter-core prefetching for multicore processors using migrating helper threads, *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, New York, NY, USA, ACM, pp. 393–404 (2011).
- [10] Kim, D., Liao, S. S.-w., Wang, P. H., Cuvillo, J. d., Tian, X., Zou, X., Wang, H., Yeung, D., Girkar, M. and Shen, J. P.: Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, Washington, DC, USA, IEEE Computer Society, pp. 27– (2004).
- [11] Reed, R.: Best Known Methods for Using OpenMP on Intel Many Integrated Core (Intel MIC) Architecture (2013).
- [12] Reinders, J.: An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors (2012).
- [13] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: Proposing A New Task Model towards Many-Core Architecture, *Proceedings of the ACM international workshop on manycore embedded systems 2013*, MES'13, Tel-Aviv, Israel, ACM (2013).
- [14] Soares, L.: Operating System Techniques for Reducing Processor State Pollution, PhD Thesis, University of Toronto (2012).
- [15] Soares, L. and Stumm, M.: FlexSC: flexible system call scheduling with exception-less system calls, *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Berkeley, CA, USA, USENIX Association, pp. 1–8 (2010).
- [16] Yonezawa, A.(ed.): *ABCL: an object-oriented concurrent system*, MIT Press, Cambridge, MA, USA (1990).
- [17] Yuan, Q., Zhao, J., Chen, M. and Sun, N.: GenerOS: An asymmetric operating system kernel for multi-core systems., *IPDPS*, IEEE, pp. 1–10 (2010).
- [18] 科学技術振興協会: CREST, <http://www.jst.go.jp/kisoken/crest/index.html>.
- [19] 志田直之, 住元真司, 宇野篤也: MPI Library and Low-Level Communication on the K computer, *Fujitsu*, Vol. 63, No. 3, pp. 299–304 (2012).
- [20] 堀 敦史, 山本啓二, 大野善之, 今田俊寛, 亀山豊久, 石川 裕: ハードウェア同期機構を用いた超軽量スレッドライブラリ, 情報処理学会研究報告. [ハイパフォーマンスコンピューティング], Vol. 2011, No. 6, pp. 1–8 (2011).
- [21] 島田明男, Gerofi, B., 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, *IPSI SIG Notes*, No. 3 (2012).