PGAS言語によるデータ並列計算の
メニーコアプロセッサ向け最適化の研究

2016年　3月

池井　満

PGAS言語によるデータ並列計算の
メニーコアプロセッサ向け最適化の研究

池井　満

システム情報工学研究科

筑波大学

2016年3月

概 要

近年、高性能計算を行うためのプロセッサとして、多数のコアをチップ上に搭載したメニーコアプロセッサが注目されている。本研究ではプログラミングの対象をデータ並列計算として、メニーコアプロセッサ単体上と、これをノードとして用いたクラスタシステム上で、この計算プログラムを効率よく実行するための PGAS 言語の実行時システムの検討を行った。PGAS 言語は、グローバルなアドレス空間を持ち局所性を意識して最適化できる並列プログラミングモデルである。データ並列計算としてステンシル計算を対象とし、メニーコアプロセッサとしてはインテルの Xeon Phi を取り上げる。

クラスタシステム全体にわたる高い並列性を記述する PGAS 言語として、XcalableMP(XMP) を用いる。この XMP は、グローバルビューを用いてコア間だけでなく計算ノード間にまたがるグローバルな配列を記述することができる。姫野ベンチマークと 2 次元の Laplace 方程式を解くベンチマークを XMP で記述して得られたプログラムで Xeon Phi クラスタシステムの評価を行った。

その結果から、XMP のプログラム並列実行単位のプロセスをコアに割り当て、ステンシル計算の袖領域を交換する reflect 処理を同じノード内においては共有メモリを用いて行うこととした。コアごとにプロセスを割り当ててグローバル配列を分散配置することにより、配列データへのアクセスの局所性が向上し、キャッシュを有効に使える。また、共有メモリを用いることにより、従来の実装の MPI を用いる実装よりも、効率的にデータ交換ができる。

これらの実装を行った結果、これまでの MPI を用いた実装と比較して、1 ノードの性能では Laplace の 1 次元分割で 4.62%、姫野ベンチマークで 62.9%の性能向上が得られ、また 16 ノードの実行では、Laplace の 2 次元分割で 18%、姫野ベンチマークでは 2.46 倍の性能向上が得られた。XMP を用いることにより、ノードの内外を同じグローバルビューで記述することができ、かつ本研究の実行時ルーチンによりコアでのデータアクセスでの局所性と効率的な通信により、メニーコアプロセッサのクラスタシステム上で、データ並列計算を効率的に実行できる PGAS 言語の実行時システムを実現できた。

目 次

表目次

図目次

第1章　はじめに

2015年11月現在、世界最高性能の計算機のMPLinpack浮動小数点演算性能は33.86 PFlopsで、その消費電力は17,808 KWである[28]。したがって、この計算機システム全体の電力(W)あたりの演算性能は1.901 GFlops/Wということになる。この20 MW程の電力予算を守りながら、演算性能をさらに30倍に上げて1 EFlopsにするプロジェクトが、世界中で進められているが、これには、性能電力比を50 GFlops/W以上にしなければならず、計算機の様々なパーツの性能電力比の改善が必要となる。

ところで、現在の最新汎用プロセッサIntel Xeon E-5 2699v3の理論ピーク性能は547.2 GFlopsで、その消費電力は145 Wであるから、プロセッサ単体の性能電力比は3.77 GFlops/Wである。このプロセッサ単体での性能電力比には、メモリやその他の回路の消費電力を含まないため、システムとして実際に必要な値よりも高くなるが、それでも50 GFlops/Wに大きく及ばない。このように最高の計算機性能を求めるには、汎用のプロセッサでは性能電力比が低すぎるため、性能電力比を改善する新たなプロセッサの開発が進められている。

そのような、新たに提案されたプロセッサの一つとして、メニーコアプロセッサがあげられる。これは、プロセッサの単一スレッド性能、特に浮動小数点演算性能以外の性能を犠牲にして、コアを小さくすることにより、プロセッサ内のコア数を多くして性能電力比を高めるものである。例えば、本研究で取り上げたIntel Xeon Phi 7120Aは61個のコアを持っており、E-5 2699v3の18コアに対して3倍以上のコアを持つ。科学技術計算を汎用プロセッサ上で実行して、最高性能を得るためには、並列プログラミングが必須となっているが、メニーコアプロセッサで並列計算を行うための大きな課題としては、プロセッサ内部にも多数のコア持つため、その高い並列性を活かす並列プログラムを行う必要性がある。

並列プログラムの方法としてPGAS言語を用いる方法がある。この方法は、現在、大規模クラスタシステムで広く行われている、いわゆるハイブリッドの並列プログラムとは異なり、計算対象に対して統一的に計算やデータ分割の指示を行うだけで、並列プログラミングを行うことができるため、プログラムの生産性が高い。さらに大規模クラスタシステムのように、疎結合の計算機システムにも適用でき、しかも演算の局所性を活かした計算を実行できるため、メニーコアプロセッサを用いた計算機システムにも有効に適用できる

ものと考えられる。そこで、メニーコアプロセッサを用いた、性能電力比の優れた計算機システム上で、PGAS 言語を用いたプログラムを効率よく実行できるように、その最適化の研究を行うことにした。

1.1 研究の目的

本研究では、メニーコアプロセッサとそのクラスタシステム向けの、データ並列計算のプログラミングと実行環境の提案を行う。まず、並列計算のプログラミングには、PGAS 言語を用いることにより、明示的に通信を記述することなく、生産性高くこれを行う。そしてプログラム実行時には、その PGAS 言語の実行時システムが、メニーコアプロセッサ内部やメニーコアプロセッサ間の特性を活かして必要なデータの交換を行う。この結果、データ並列計算を生産性高く開発し、かつ、開発したプログラムを、メニーコアプロセッサの持つ特性を活かして、効率良く実行できる並列計算開発環境が実現できると考えた。

そのような環境の一つとして、ステンシル計算を XMP で記述し、これをメニーコアプロセッサの Xeon Phi をノードに用いたクラスタシステム上で効率よく実行できる方法を開発することを目標とする。このために、(1) メニーコアプロセッサの特性評価、(2)XMP のメニーコア向け最適化、(3) 共有メモリによる実装、(4) 新しい袖領域通信上での効果の 4 つの検討を行う。

まず、(1) メニーコアプロセッサの特性評価を行う。この特性評価では、メニーコアプロセッサ上での並列計算に、共有メモリを利用した OpenMP と、格子空間配列をコア間に分散配置した MPI の両者を比較して、メニーコアプロセッサに適した並列実行方法を検討する。

次に、その結果に基づき (2)XMP のメニーコア向け最適化を検討する。XMP を利用してステンシル計算を行う場合、グローバル配列を袖領域を付けて node に分配する。ステンシル演算中に、この袖領域を更新する reflect は、XMP が提供する実行時システム関数の一つである。XMP の基本的な実装をできるだけ変更することなく、この XMP の実行時システム関数を Xeon Phi 向けに最適化することで、ステンシル計算の最適化を行う。

reflect の最適化に、Xeon Phi 内の node 間での (3) 共有メモリによる実装を試みる。node のプロセスをコアに割り当てると、Xeon Phi 内ではコア間で物理的にリングバスを介してメモリを共有しているのにもかかわらず、node 間のデータの交換に MPI のような通信を行う必要が生じる。そこで、reflect で使用している MPI 通信を、データ交換する必要のある node が同じ Xeon Phi(ノード) 内にある場合は共有メモリ上のメモリ転送に置き換える最適化を検討する。

XMP には、2014 年に MPI の派生データ型を用いた新しい袖領域通信を行う reflect 処理が実装された。そこで、開発した共有メモリを用いた実装を、新しい reflect 処理に対して適用し、その効果を確認する。この評価には、(4)PVAS を用いた共有メモリによる実装 [37] を用いる。

本研究の貢献は、次の 3 点にある。

- 提案システムは、従来行われているハイブリッドを用いた並列方法と異なり、ノード内の計算でコアの持つ 2 次キャッシュ間のキャッシュラインのシェアリングに起因する性能の低下を防ぐことができる。多重の並列性を持つアーキテクチャに対応してステンシル計算をプログラムするには、ノード内では OpenMP を用い、マルチノード間で MPI を用いるハイブリッドの手法がある。ところが、Xeon Phi に、この方法を適用すると、多数のコア間でキャッシュラインのシェアリングの問題が発生する。

- ステンシル計算の並列実行で必要となるプロセス間での袖領域の通信時間は、ノード内のプロセス間通信の場合、共有メモリを用いることにより減少できる。また、ノード間で通信する必要がある場合にのみ MPI で行うように XMP 実行時システム関数を改良することにより、XMP のコンパイラの大部分を変更することなくこのような性能を実現できる。

- マルチノード・メニーコアプロセッサのような、ノード内とノード間に高い並列性を持つアーキテクチャに対して、PGAS 言語 XMP によるグローバルな単一の並列化を記述するだけで、ステンシル計算の性能を得られる実行時システムを提案する。この実行時システムにより、プログラマがノード内とノード間で別々の並列化を行う必要が無い。

1.2 関連研究

ステンシル計算を、実行する並列計算機のアーキテクチャに自動最適化する試みは、20 年以上前から行われている。本研究のように、汎用の言語を用いて記述したステンシル計算を分散メモリ並列計算に最適化を試みた研究としては、文献 [11] や文献 [26] が上げられる。これらの研究では、本研究と同様に袖領域を用いたデータ移動の最適化を試みているが、ステンシル計算の記述に、データを移動する特定の関数を使用させている点と、データの分割をマルチノード間だけで行い、ノード内においてはこれを行わない点が異なる。

また、ステンシル計算から並列性を抽出して、これを近年のマルチプロセッサに最適化する試みも、いくつか行われており、これらは、大きく 2 つのグループに分類される。1 つ目のグループは、ドメイン固有言語を用いるもので、代表的なものとしては、文献 [29] 等があげられる。この方法は、我々の方法と異なり、ユーザは計算の実行順序をループで記述するのではなく、ステンシル計算の内容を記述する。したがって、台形分割タイリング等の予め準備された、並列性の高いアルゴリズムが使えるものの、ユーザはアルゴリズムを記述できないため、準備された計算しか解けない問題がある。

もう 1 つのグループが、我々の方法と同様に汎用言語のループを変換して最適化するもので、文献 [19] 等があげられる。この文献では、本研究と同様の袖領域を利用するものも含めて、計算と通信のコスト解析を行って、この結果を利用して最適する方法提案がされている。しかし、この最適化の対象は均一的な計算ノードを持つマルチプロセッサであり、マルチノード・メニーコアプロセッサへの適用はまだ考慮されていない。

同様な方法で、Cell や GPU 等のメニーコアプロセッサと同等な高並列ノードを対象としたものも提案されている。例えば文献 [3] 等があげられる。しかしこれらは、並列ノードのコア間で明示的に共用する高速メモリの利用の使用を前提にしている。Xeon Phi のように、コアにローカルなキャッシュを使う場合は、本研究で述べた、キャッシュラインのコア間の共用の問題の考慮が重要となる。

また、PGAS 言語の Xeon Phi へ適用に関しては、Unified Parallel C (UPC) 用いた文献 [21] がある。これは、インテルのシンメトリック・コミュニケーション・インターフェース (SCIF) を用いて、単体ノード内で、Xeon Phi と、ホストの Xeon プロセッサ間、または、ホストに複数枚実装された Xeon Phi 間でグローバルアドレス領域を構成するものである。本研究のように、ホストの異なるマルチノード間の検討は行っていない。

本研究で扱った姫野ベンチマークの Xeon Phi への最適化に関しては、文献 [36] で、特定の配列において、Xeon Phi のメモリアクセスのレイテンシを減らすために、配列の方向やアライメントを変更するなど様々なプログラム変更して、単 1 ノードでより高い性能を実現している。本研究では、XMP で記述したユーザプログラムのアルゴリズムを変更しての最適化は行わない。

この他、ステンシルコードの Xeon Phi へのアプリケーションと評価については文献 [34] と文献 [10] で行っているが、これらは、両方とも、オフロード・モデルを使用したもので、得られた性能を、GPU と比較している。

1.3 本論文の構成

次章より、次に示す順で研究内容について述べる。まず 2 章では、本研究を行った背景について述べる。3 章では、本研究の最適化の対象であるメニーコアプロセッサ、Xeon Phi の特徴について述べる。4 章では、PGAS を含めた並列計算のプログラム・モデルついて述べる。5 章では、本研究で並列化に使用した XcalableMP (XMP) について、ステンシルコードの並列化を記述する方法も含めて述べる。6 章では、ステンシルコードをメニーコア上で実行するときの特性評価結果について述べる。7 章で、本研究でメニーコア用に最適化した XMP の実行時システムの実装ついて述べる。8 章では、2 つの XMP で記述した 2 つステンシルコード、Laplace と 姫野ベンチマークの性能の評価結果について述べる。9 章では、新しい袖通信での本研究の効果を確認する。最後の 10 章が、本研究のまとめと今後の課題である。

第2章　背景

2.1　プロセッサ技術の動向

本節では、メニーコアプロセッサが、本研究で対象とする高性能な計算機に使用されるようになった背景について、過去のマイクロプロセッサの性能向上の経緯の観点から述べる。計算機用の演算装置として使用されるプロセッサ集積回路の高性能化は、その集積回路を構成する半導体トランジスタの微細化技術とともに進んできている。図 2.1 に、このようなプロセッサの代表的なメーカであるインテル社のプロセッサ 1 個あたりに使用しているトランジスタ数、プロセッサの動作周波数と消費電力とクロックあたりの実行命令数 (ILP=Instruction Level Parallelizm) の推移を 1970 年代から 2005 年過ぎまで示した。インテ

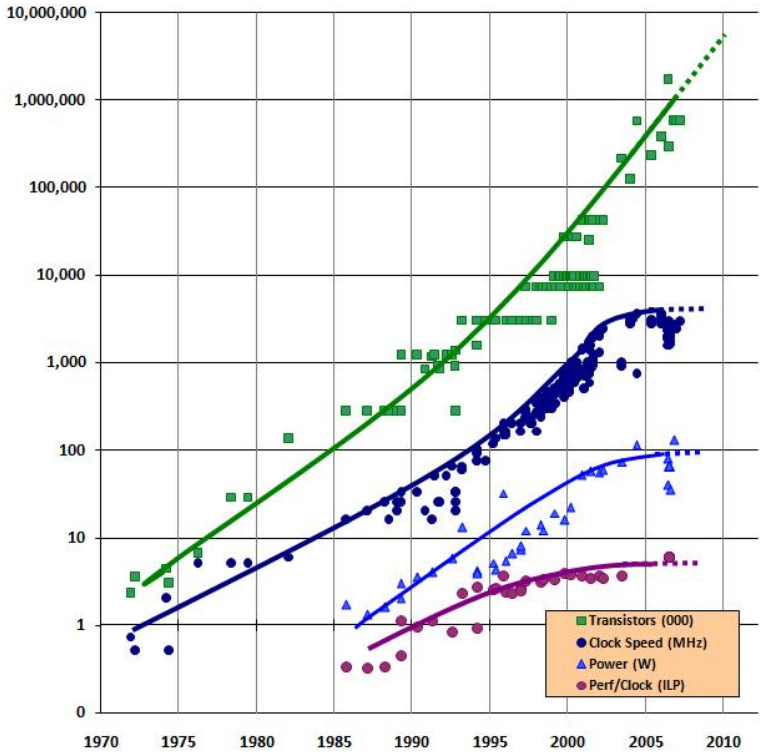

図 2.1: インテル社の プロセッサの 動向 (出展: インテル、Wikipedia、K. Olukotun 氏)

ル社のプロセッサのトランジスタ数は数千個から始まり、ムーアの法則と呼ばれる予測に沿い 2 年に 2 倍の割合で増え続け、図中では 2007 年に 10 億個を越えている。 近年では、2015 年に同社が発売した Xeon E7-8800 v3 は 22nm の半導体プロセスルールを用いており、

使用トランジスタ数 56 億個に達している。同社は既に 14nm のプロセスを用いた製品を発売しており、10nm の製品もアナウンスしていることから、この使用トランジスタ数の増加はまだあと数年は続くものと考えられる。

一方、他の 3 つの値は 2003 頃には一定の値に達し、それ以降、この対数グラフ上ではほとんど変化していない。周波数は 3 GHz 付近、消費電力は 100 W、ILP は 4 程度でほとんど変化していない。つまり、図 2.1 によると、ここ 10 年あまりの間プロセッサの周波数や性能は大きく変化していないことがわかる。それでは相変わらず増えつづけているトランジスタは何に使われているのであろうか。表 2.1 に過去 10 年に発売されたインテル社の汎用サーバ用のプロセッサの製品ファミリとその代表的な製品の仕様を示す。

表 2.1: 過去 10 年のプロセッサ

Xeon	5100	5500	5600	E5-2600	E5-2600v2	E5-2600v3
発売年	2006 年	2009 年	2010 年	2012 年	2013 年	2014 年
周波数 (GHz)	3.0	3.33	3.46	3.1	2.7	2.3
コア数 (個)	2	4	6	8	12	18
SIMD 幅 (bit)	128	128	128	256	256	256
プロセス (nm)	65	45	32	32	22	22

それぞれ、2006 年から 2014 年までに発売されたものである。図 2.1 で見たように、プロセッサの動作周波数は 3 GHz 近辺で、やはり増えていない。しかしこれらのプロセッサでは、プロセッサ内のコア数とプロセッサで使用できる SIMD 命令のビット幅が増加している。 コア数は 2006 年に 2 個であったものが、2014 年には 18 個まで増えている。プロセッサに使用している半導体プロセスの値が 65 nm から、45 nm、 32nm、 22nm と小さくなり、世代が変わるごとに使用できるトランジスタ数が倍になるのに合わせて、コア数も 2 倍かそれ以上に増えていることが確認できる。またこの間に、SIMD 命令レジスタのビット幅も、128 bit から、256 bit と倍に増えている。これは、1 つの SIMD 命令で処理できるデータ量が、倍精度の浮動小数点数であれば、2 個から 4 個と 2 倍に増えることを意味する。

表 2.1 から、2003 年以降の近年のプロセッサでは、増加したトランジスタは、コア自体の性能改善ではなく、主にコア数の増加や SIMD 命令レジスタのビット幅の拡張に用いられていることがわかる。ここから、近年の汎用プロセッサの浮動小数点数演算性能向上は、主にコア数の増加と SIMD 命令のビット幅の拡張から得られていることがわかる。しかし、前述したように、現在の最新汎用プロセッサの電力あたりの浮動小数点数演算性能は、次世代のスーパコンピュータに求められる性能に大きく満たない。そこでこの問題を解決す

るために、ILP を 4 から 2 へ落とすことによりコアの面積を小さくしてさらにコア数を増やし、SIMD 命令のビット幅を倍の 512 bit に増やして、相乗効果で浮動小数点演算性能の向上を狙う、メニーコアプロセッサが登場した。

2.2 メニーコアプロセッサ

汎用プロセッサの浮動小数点演算性能を向上させている大きな要因はそのコア数の増加であることを述べてきた。したがって、さらに大きくコアの数を増やせば演算性能をさらに上げられるはずである。このような観点から、コア数を増やしたメニーコアプロセッサが提案され、使われるようになってきた。現在広く使われているメニーコアプロセッサは、大きく 2 つのグループに分けられる。一つのグループは本研究で取り上げた、Xeon Phi のように、汎用のプロセッサコアをベースに用い、これを小型化してチップ内で多数接続したもので、IBM 社の Blue Gene[22] や Sony の cell[17] 等がこのグループにあげられる。もう一つのグループは、もともとグラフィックス・プロセッサとして、コンピュータ出力画像の作画処理に使われていた GPU の演算ユニットを、科学技術演算に使用しやすいように改良したものである。

まず、後者の代表として NVIDIA 社の Kepler アーキテクチャGK110 について述べる。Kepler は演算回路として、図 2.2 に示す Streaming Multiprocessor (SMX) アーキテクチャを用いている。そして、例えばこのアーキテクチャの GPU Tesla K40 は、1 個のプロセッサ中に、他の内蔵メモリコントローラ等の周辺回路とともに、この SMX を最大 15 個持つことができる [24]。SMX は 192 個の単精度 CUDA (Compute Unified Device Architecture) コアと 64 個の倍精度演算ユニットと 32 個のロード/ストア・ユニット等を持っている。Kepler では、1 つのブロック内のすべてのスレッドは SMX 上で同じ命令を実行する。このアーキテクチャでは、32 個のスレッドのグループを Wrap と呼ぶが、SMX 上には図に示すように 4 つの Wrap スケジューラをもっている。そしてこのスケジューラが最大 65.5 K 個の 32 bit レジスタを用いて演算を行わせる。演算には、SMX あたり最大 64 KB の L1 キャッシュ/共有メモリと、他の SMX と共有する 1.5MB の L2 キャッシュを利用できる。

Kepler では、192 個の単精度 CUDA コアを持った SMX を 15 個持てるのだから、同時に 2880 個の core が使用でき、並列性が非常に高い。しかし一方、図 2.2 から明らかなように、Kepler の機械語レベルのプログラミングは従来のものとは大きく異なる。SMX 用のプログラムの開発は、最初から複数の、しかもたくさんスレッドが同時実行されることを前提に行う必要がある。従来は、1 個のスレッドでのプログラミングを行い、そこから依存性等

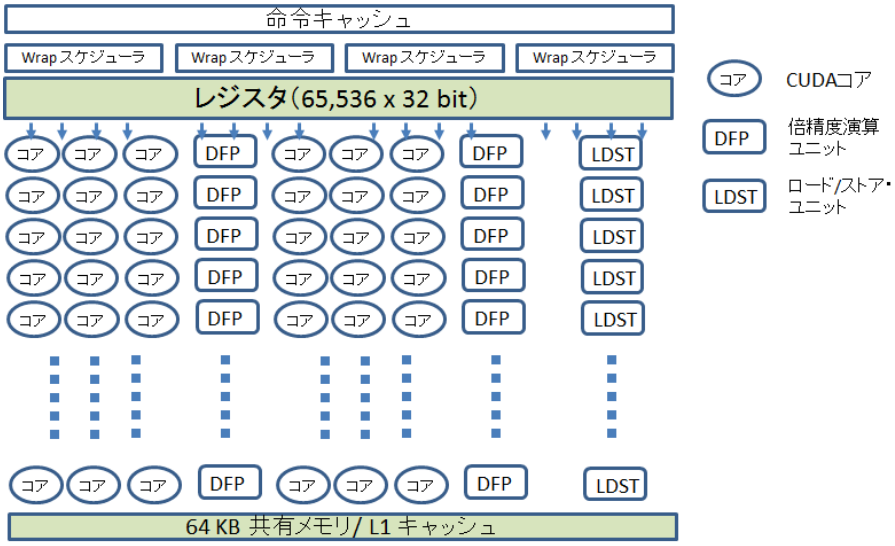

図 2.2: SMX アーキテクチャ

を考慮した上で並列化を行った。しかしこのアーキテクチャ上では、最初からスレッドのグループを用いてプログラミングを行う必要がある。

次に、本研究で用いたインテル社の Xeon Phi E-7100 を紹介する。図 2.3 にこのプロセッサのアーキテクチャの概要を示す。Xeon Phi は、それぞれ 512 KB の L2 キャッシュ(L2) とタグディレクトリ (TD) を持った 61 個の演算コアを、高速の双方向リングバスで接続した構成である。そしてこのリングバスは、最大 16 GB の GDDR5 のメモリに接続されており、すべてのコアからこの共用メモリを使用することができる。汎用の Xeon とは異なり、この L2 キャッシュはそれぞれ接続されているコア用のローカルなキャッシュで、共有のラストレベル・キャッシュとしては動作しない。TD は、それぞれの L2 キャッシュ間のコヒーレンシを保つために使用される。その各コアは 4 つのハードウェア・スレッドを持つため、この上で動作する Linux からは 244 個のプロセッサとして認識される。

図 3.1 の 61 個のコアは、図 2.2 の CUDA コアとは異なり、汎用のプロセッサのコアに近いため、かなり大きい。図 2.4 に、この Xeon Phi のコアのブロック図を示す。まず、Xeon Phi のコアはコア内にローカルな命令キャッシュと命令デコーダを持っている。このため、それぞれのコアは別々の命令列を通常のプロセッサと同様に処理することができる。実際に、図上の Xeon Phi のスカラユニットは、Pentium プロセッサと同等の能力をもち、同じ命令を実行することができる。また、FMA 等の 8 個の倍精度演算を行えるベクトルユニットもコア内に内蔵している。これらの演算を行うレジスタとして、32 bit に換算して、スカラで 30 個、ベクトル演算用に 240 個を持ち、32 KB のコアに専用のデータキャッシュを持っている。

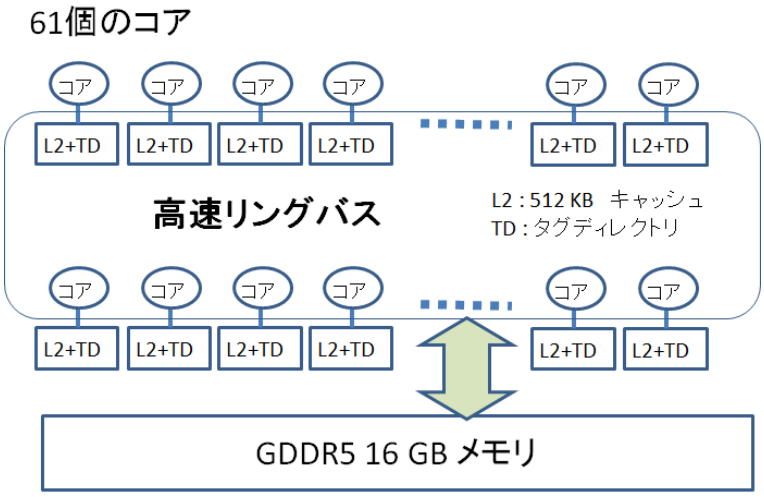

図 2.3: Xeon Phi アーキテクチャ

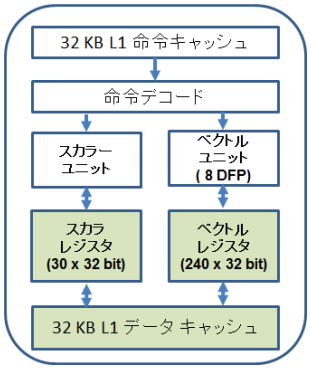

図 2.4: Xeon Phi のコア

図 3.1 と図 2.4 に示したように、Xeon Phi のアーキテクチャは汎用のマルチプロセッサのアーキテクチャに近く、プログラムの実行方法にも大きな違いはない。このため、コアが Kepler に対して大きい。

表 2.2 に、本節で述べた 2 つのメニーコアアーキテクチャ、Kepler アーキテクチャの K40 と Xeon Phi E-7100 の特性を比較して示す。まず、半導体技術は 28 nm と　22 nm で、Xeon Phi の方が集積度が高い。動作周波数も Xeon Phi が 1.66 倍高い周波数を用いている。ところが、コア数は K40 の方が 47.2 倍と圧倒的に多い。前述したように、両者のコアは大きさが異なるため、K40 の性能がこれだけ高いわけではない。プログラマが直接利用できる並列性に大きな違いがあるとみるべきであろう。実際、単精度と倍精度の演算ユニット数は、1.97 倍と 2.95 倍であり、動作周波数の差のため、倍精度演算のピーク性能は K40 が 1.18 倍優れているだけである。一方メモリの方は、両者とも DDR のメモリ転送性能の制限 (2.3 節で述べる。) を超えるため、グラフィックス用のメモリである GDDR5 を用い、その容量で 1.33 倍、転送幅で 1.22 倍 Xeon Phi の方が優れている。消費電力の差もメモリ容量の影

響が大きい。従って、性能上は、両者に大きな違いはない。

表 2.2: GPU K40 と Xeon Phi E-7100 の比較

項目	K40	Xeon Phi E-7100
半導体技術は	TSMC 28 nm	Intel 22 nm
周波数	0.745 GHz	1.238 GHz
コア数	2880 (CUDA)	61 (x86)
単精度演算ユニット数	2880	976
倍精度演算ユニット数	960	488
倍精度演算ピーク性能	1.43 TFLOPS	1.21 TFLOPS
メモリ容量	12 GB	16 GB
ピークメモリ転送幅	288 GB/s	352 GB/s
消費電力	235 W	300 W
実行方式	オフロード	ネイティブまたはオフロード
プログラム方法	CUDA	汎用のプログラム言語

両者の大きな違いは、表 2.2 の下に示した実行方式と、プログラミングの方法にある。K40 はプログラムの全体を単独で実行するようには作られていない。必ず汎用の主プロセッサがプログラムの実行を開始し、このプロセッサからの指示で、並列演算が必要な部分だけを実行する。このような実行方法を、主プロセッサの演算の一部を別のプロセッサに任せるということから、オフロードと呼ぶが、K-40 はこのオフロード実行しかできない。オフロードの問題は、多量のデータを主プロセッサの持つメモリ上から、演算を行うプロセッサ用のメモリに転送しなければならないことにある。プログラマは、オフロードに必要なデータを選び、さらに、それらをできるだけ演算時間に影響を与えないように転送しなくてはならない。

また、プログラム方法もコアの違いの影響が大きい。K40 は、CUDA コアと呼ばれる小型の多数のコアを利用すべく、最初からスレッドのブロックを前提に並列プログラミングを行う必要があり、このために CUDA と呼ばれる専用のプログラミング環境を準備している。これに対して、Xeon Phi のプログラミングは、汎用のプログラミング言語で行うことができ、その並列性を高くすることで多くなった演算ユニットを有効に利用しようするものである。

このように、同じような性能を持つメニーコアプロセッサのうち、本研究では次に述べる 2 つの理由で、Xeon Phi を研究の対象とした。

(1) プログラミングの生産性

既存のプログラムのほとんどは、汎用のプログラミング言語で既に書かれている。汎用の言語を用い、アーキテクチャに大きく依存しない標準化された並列計算用の拡張を用いてプログラミングできるプロセッサを対象にすれば、プログラミングは容易であり、また開発したプログラムは、将来のアーキテクチャにも有効利用が可能となり生産性が高い。ハードウェアの性能を最大限活かすことのできる CUDA のような専用のプログラミング環境を利用することは魅力的である。しかし、本章でも述べてきたとおり、ハードウェアは大きく変化し続ける。特定のアーキテクチャに依存するプログラミングを行うと、別のアーキテクチャに移行する際に大きなプログラムの書き換えが発生する。

(2) オフロードの煩雑性

オフロードを前提するプロセッサを用いると、必ず主プロセッサとコプロセッサ間で情報の交換が必要となる。科学技術計算のように大量のデータに対する計算を行う場合、このデータの移動に要する時間が演算時間に大きく影響することが考えられる。また、頻繁にプロセッサ間でのデータ交換を行うと、互いに依存するデータを持つ処理間への通信のレイテンシの影響が大きくなり、これも好ましくない。したがって、プログラミングの際にこれらの点を考慮してデータの交換を行う必要があり煩雑となる。

2.3 プロセッサ・アーキテクチャの進化

2.1 節で、過去 10 年間のプロセッサの主な性能向上は、コア数の増加と SIMD 命令の演算を行うベクトルレジスタ幅の増加から得られていることを示した。しかし、細かく各プロセッサのアーキテクチャを比較すると、電力あたりの性能を下げないという条件のもと、数%の性能向上のためにコア自体に機能を追加し、その他プロセッサ内のコア以外の機能を向上させている。これは、コア数や SIMD 命令といった倍々の性能向上には、当然のことながら、メモリ参照能力の同様な向上も必要となるからである。

このような、プロセッサ・アーキテクチャ上の進化としては、内蔵メモリコントローラの広帯域化、チャネル数の増加やキャッシュメモリの増加が上げられる。またこの他、特に

科学技術計算では、コアのレジスタだけで実行できるような、比較的少ない命令数のループの実行が実行時間を占めることも多く、このようなループを高速に実行する機能も重要である。これらのうち、本研究に関連のある、内蔵メモリコントローラ、ラストレベル・キャッシュとループについて述べる。

図 2.5 に、近年のプロセッサに共通の、主なコア以外の部分（アンコア部分と称される）を、コア自体も含めて示す。これらは、全コアで共有するラストレベル・キャッシュ（LLC）と、内蔵メモリコントローラの持つメモリチャネルである。図に示すように近年のプロセッサでは、2 MB から 2.5 MB の LLC 用のキャッシュメモリをコアに隣接して配置する構成をとっている。そして、これらのコアとキャッシュメモリのペアをリングバスで接続して、すべてのコアから、すべてのコアに隣接したキャッシュメモリを、コヒーレンシを保ちながら使用できるように論理接続している。したがって、プロセッサの持つ共有 LLC の容量は、このコアごとに割り当てられた 2 MB または 2.5 MB の値に、内蔵するコア数を乗じた値となる。

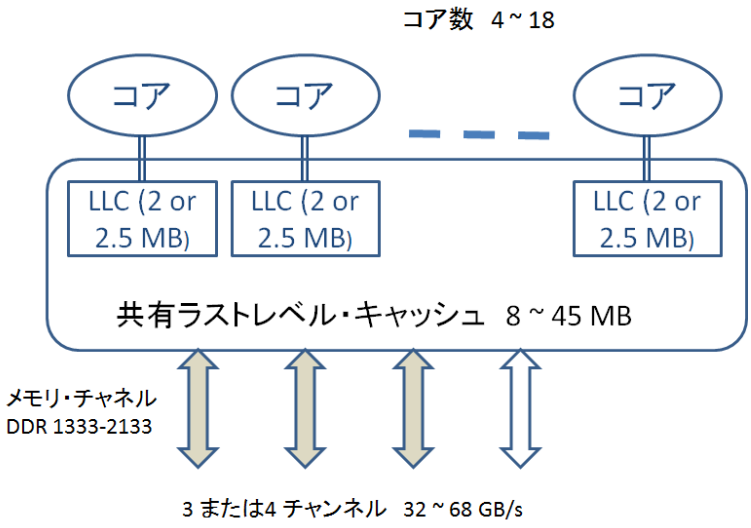

図 2.5: 共通のアンコア部分

またさらに、近年のプロセッサはメモリコントローラを内蔵しており、図 2.5 に示したように、3 本または 4 本のメモリチャネルを介して、直接メモリに接続できる。このメモリチャネルは、JEDEC 半導体技術協会の Double Data Rate (DDR) メモリの規格に基づいたものを使用しており、近年ではその転送レートは DDR3 の 1333 MT/s から DDR4 の 2133 MT/s のものを使用している。このためチャネル当たりのデータ転送帯域は、10.66 GB/s から 17.06 GB/s となり、全体では、1 個のプロセッサあたり約 32 GB/s から 68 GB/s のデータ転送能力を持っている。

表 2.3 に表 2.1 に示したプロセッサのうち、2009 年以降に発売されたプロセッサの倍精

度浮動小数点演算の理論ピーク性能、共有 LLC の容量と内蔵メモリチャネルのデータ転送帯域幅をコア数とともに示す。ここで示した理論ピーク性能は、クロックあたりにコアが最大行える演算数に定格周波数とコア数を乗じたものであり、コア数が増えるとともに性能が向上している。SIMD 命令の拡張の影響もあり、これは 53.28 GFLOPS から、662.4 GFLOPS と、12 倍以上の性能向上を実現している。また、共有 LLC の容量も、コア数に比例して、前述したコアに隣接して持つ LLC 用のキャッシュメモリの数が増えるので、順調に増え、これは 8 MB から　45 MB と、約 5.6 倍に増えている。

表 2.3: プロセッサ性能とアンコア部分の進化

Xeon	W5590	X5690	E5-2687	E5-2697v2	E5-2699v3
コア数	4	6	8	12	18
浮動小数点演算性能 (GFLOPS)	53.28	83.04	198.4	259.2	662.4
共有 LLC 容量 (MB)	8	12	20	30	45
データ転送能力 (GB/s)	32.0	32.0	51.2	59.7	68.3
拡張命令	SSE4.2	SSE4.2	AVX	AVX	AVX2
その他			PCI Ex3.0	PCI Ex3.0	PCI Ex3.0

表 2.3 上で、4 コアから 18 コアと 4.5 倍に増えたコア数の増加に呼応するように向上している浮動小数点演算性能や LLC の容量に対して、プロセッサとメモリ間のデータ転送能力の増加は望ましくない。この間、DDR の転送レートが 1333 MT/s から、2133 MT/s に改善したのと、メモリチャネル数を 3 から 4 に増やすことで、32 GB/s から 68.3 GB/s と 2.13 倍にしか増えていない。この浮動小数点演算性能に対して、低いメモリ転送能力しか持てないことが、科学技術計算に、ここで説明したような汎用のプロセッサを用いることへの大きな障害の一つである。2.2 節で述べたように、本研究の対象である、メニーコアプロセッサでは、DDR の代わりにより高速な画像処理プロセッサ用の GDDR メモリを使用して、これに対応している。

また、表 2.3 の拡張命令の項目に示したように、これらのプロセッサでは、上位互換の 3 世代の拡張命令セット SSE4.2、AVX と AVX2 をそれぞれサポートしている。これらの命令セットが表 2.1 で説明した 128 bit と 256 bit の SIMD 命令を含んでいる。これらについては次節以降で説明する。また、その他の項目に示したように、Xeon E5-2687 以降のプロセッサでは、メモリコントローラに加えて PCI Express Gen3 コントローラを内蔵しており、最大 40 レーンのポートを介して全体で最大 40 GB/s の入出力を、直接プロセッサと行うことができる。半導体の微細化技術で得られたトランジスタは、このように周辺のシステム

回路を内蔵することにも使われており、プロセッサの入出力時のレイテンシの削減や消費電力の削減に貢献している。

ここまでは、主にプロセッサのアンコア部分について、近年のプロセッサでの先進性を示した。この他、実際にプログラムを実行するコア部分でも、いくつかの先進的な改良が行われている。これらのうち、本研究で対象としているステンシル計算のような、比較的小さなループを多重に繰り返すコードに適用できる改良点として、Xeon E5-2687 以降のプロセッサコアのフロントエンドの部分を紹介する。

図 2.6 に、主にプログラムのデコードを行う、Xeon コアのフロントエンドの部分を示す。図の左下、データと共用の L2 キャッシュから、実行プログラムは、命令実行時に 32 KB の命令専用のキャッシュに格納される。そして、実行される命令がフェッチされると、デコーダでデコードされ、マイクロ命令として命令キューを介してバックエンドへ渡され、処理が行われる。この際、マイクロ命令に簡単に変換できないものは、デコーダの下のマイクロ・シーケンサ（MS）ROM により、マイクロ命令の列として命令キューに提供される。

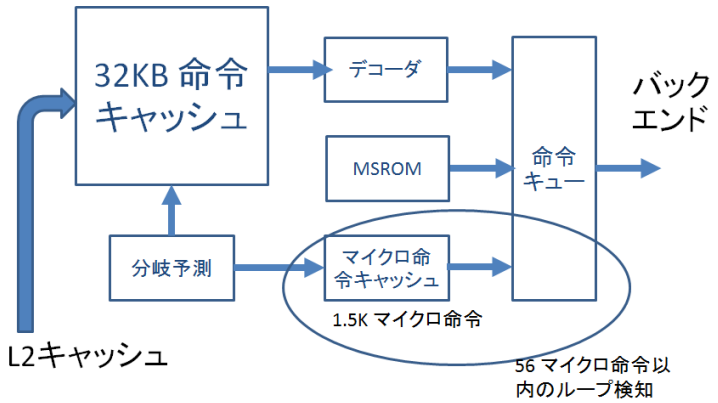

図 2.6: 最新コア・フロントエンド

以上が、図 2.6 の基本的な処理の流れだが、この他このコアは 1500 命令のデコード済みのマイクロ命令を格納できるマイクロ命令キャッシュを持っている。コアの分岐予測ユニットは、ループ等で分岐が起きる先を予測するが、これは通常の命令キャッシュだけではなく、このマイクロ命令キャッシュを参照することができる。このため、分岐予測先がこのマイクロ命令キャッシュ内にあり、その分岐予測がヒットした場合は、命令キャッシュを含むメモリ参照関連のロジックへの影響がほとんど無く、しかもデコード回路も使用せずに、マイクロ命令を実行し続けることができる。

さらに、命令キューには 56 マイクロ命令までのループを検知する、ループ検知ロジックを持っており [16]、コアあたり、同時に 56 か所のループを検知することができる。SIMD 命令を使ったループでは、限られた数の SIMD レジスタを利用した少ない命令数でのルー

プを構成することが多く、このような場合フロントエンドでは、この機能を使って、低消費電力で、しかもメモリ関連のロジックに負担をかけずに、ループを処理することができる。このため、LLC やメモリチャネルの全帯域幅を演算のためのデータ転送に占用させることができて効率が良い。

2.4 チップ上共有キャッシュの限界

コア数の増加にともない、これに対応するために、プロセッサのアーキテクチャが追加されている。2.3 節で述べたように、汎用のプロセッサではコア間で共有している LLC を持っている。コア数が増えれば、コア間での共有リソースに対する競合の可能性はコア数の２乗で増えるので、キャッシュのコヒーレンシを保つためのデータ通信量が急激に増加し、演算性能が悪化することは知られている [1][35]。このため、各コアに隣接して持たれる LLC 用のローカルメモリは、コヒーレンシを保つためのデータ通信量の１つのコアへの集中を避けるため、キャッシュのタグディレクトリをそれぞれ持っている。

図 2.7 に 10 コア以上を持つプロセッサのバスの構成図を示す。図に示すように 10 個以上のコアを持つプロセッサでは、コア間を接続しているリングバスは 2 つ存在し、これらはバス・インターフェースで接続され、論理的にはすべてのコアと LLC にタグデレクトリ（TD）を加えたモジュールは、一つのリングバスにつながっているように動作する。各リングバスには、ホーム・エージェント（HA）と呼ばれる、リング内のキャシュのコヒーレンシを確保するための回路を持っている。通常の設定では、リングバス１に接続している HA1 が両方のリングに接続されているすべての LLC の管理を行っている。

ところが、10 個以上の多数のスレッドからのリクエストを、一つの HA1 で処理するとレイテンシが大きくなることが問題となってきた。そこで、コヒーレンシを直接管理する領域を 2 つのクラスタに分ける、クラスタ・オン・ダイ（COD）の機能が追加された。COD を有効にすると、プロセッサ上のコアは 2 つのクラスタ、クラスタ 1 とクラスタ 2 に分けられる。そして、クラスタ 1 に属するコアは、今まで通り HA1 がそのコヒーレンシの管理を行うが、クラスタ 2 に属するコアの管理は HA2 が担う。このことにより、クラスタ内のコアだけを使う場合、キャッシュにヒットした場合やメモリアクセス時のレイテンシが小さくなる。

このように、プロセッサ内のコア数が増加すると、多数のコア間で LLC のコヒーレンシを低レイテンシで処理することが難しくなり、プロセッサ内でもコアを 2 つのクラスタに分けるなど、均一なマルチプロセッサとしてだけ見たのでは、性能を得ることが難しくなっ

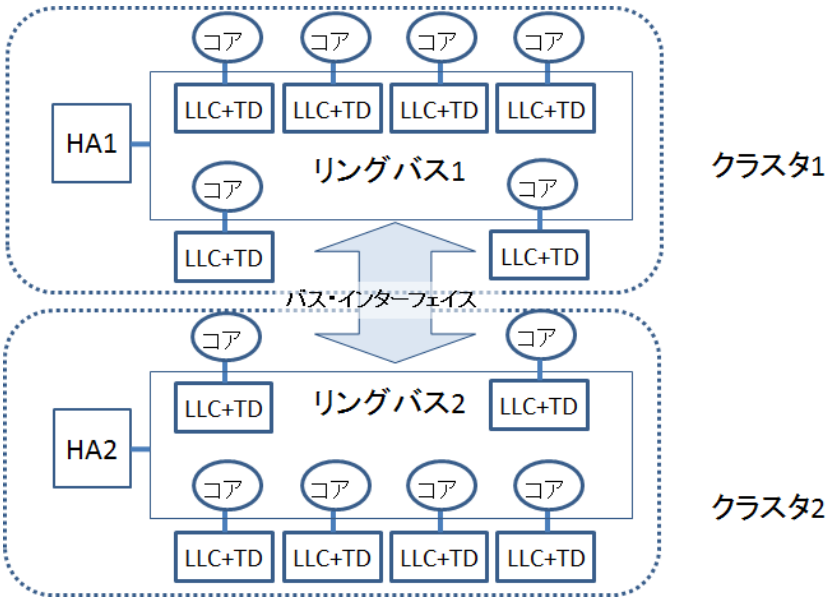

図 2.7: クラスタ・オン・ダイ

てきた。

2.5 SIMD 命令の進化

本研究では、科学技術計算を効率よく実行する方法として、SIMD 命令を利用しているが、プロセッサに内蔵する SIMD 命令は、もともと画像処理用にプロセッサに追加された。図 2.8 に、この SIMD 命令の拡張された推移を 1999 年から示す。図の最初の世代の SIMD 命令は、Streaming SIMD Extensions (SSE) という名称のビット幅が 128 bit のもので、画像処理を主なアプリケーションとしていたので、従来の命令セットに、単精度ベクトルの浮動小数点四則演算を中心に 70 種類ほどの SIMD 命令が、追加された。翌年には SSE2 となり、これに倍精度や整数のベクトル演算が追加され、基本的な演算命令がそろった。

SSE3 では、複素数やインターネットでセキュリティ上の要請から、暗号のデコードに有用な命令が追加された。さらに SSE4.1 と SSE4.2 ではベクトルレジスタ間での演算に加えて、ベクトルレジスタ内の要素間の演算、比較や要素中の 1 のビットを数えるなど、現在、SIMD 演算中で用いられる命令がすべて出そろった。そして、2011 年に新しい命令体系として、Advaced Vector eXtensions (AVX) と呼ぶ、256 bit の倍精度と単精度の浮動小数点演算を行う SIMD 命令を追加した。現在のプロセッサで利用できる AVX2 は、AVX に主に整数のベクトル演算機能等の 124 個の命令追加したものである。

そして表 2.3 に示したように、現在はそのレジスタの幅が 128 bit の SSE4.2 と、256 bit の AVX と AVX2 が主に用いられている。図 2.9 にこれらの命令で用いるレジスタの利用方

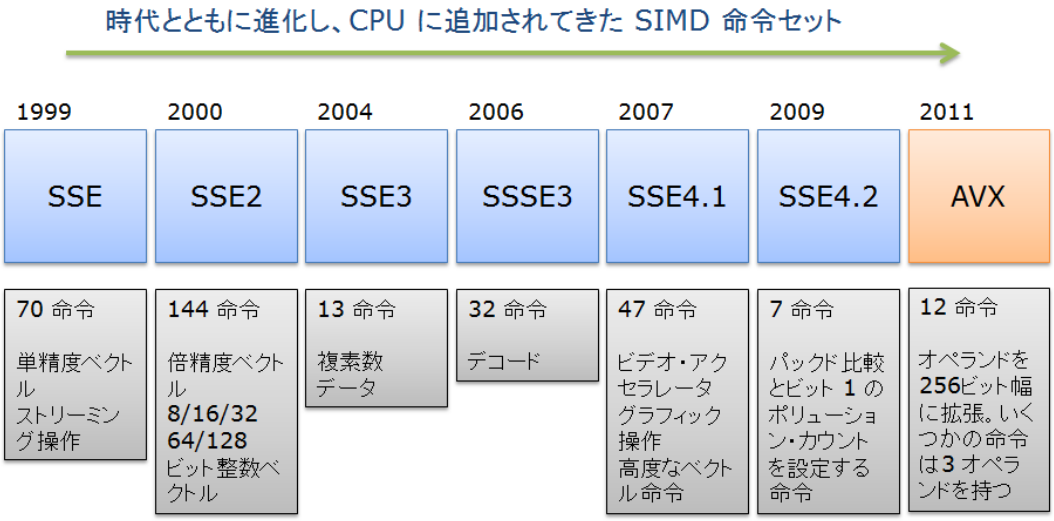

図 2.8: SIMD 命令拡張の推移

法を示す。

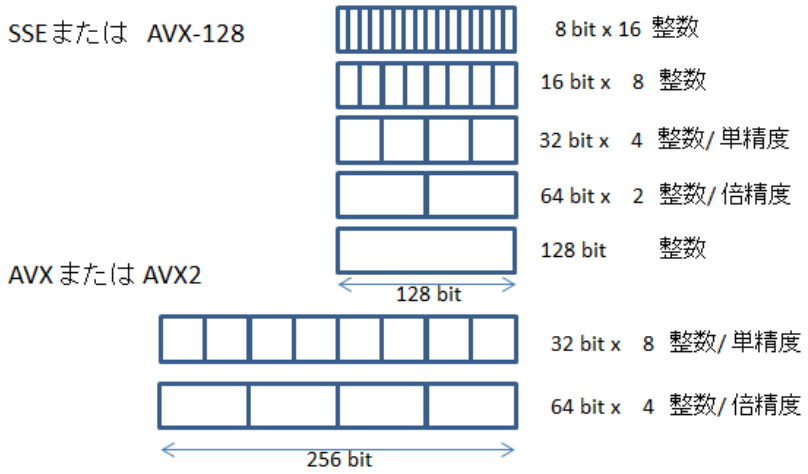

図 2.9: SIMD レジスタの利用方法

まず、プロセッサを 64 bit のアドレスモードで使用する場合、SSE では 128 bit のレジスタとして、XMM0 から XMM15 までの 16 個の SIMD レジスタを使用することができる。そして命令により、これらのレジスタに、図に示すように、整数では、16 個の 8 bit 整数、8 個の 16 bit 整数、4 個の 32 bit 整数、2 個の 64 bit 整数、または 128 bit 整数を格納して、同時に処理を行うことができる。さらに、IEEE-754 で規定される 32 bit 単精度浮動小数点数や 64 bit 倍精度浮動小数点数も、それぞれ 4 個または 2 個を同じレジスタに格納して、格納された複数の浮動小数点数に対して同時に浮動小数点演算を行うことができる。

AVX ではこの、128 bit のレジスタを、256 bit に拡張した。この結果 AVX2 の SIMD 命令を使用すると、図 2.9 の下に示すように、8 個の単精度浮動小数点数か、または 4 個の倍精度浮動小数点数かのどちらかをレジスタに格納して、これらの演算を同時に実行するこ

とができる。この 256 bit のレジスタは、YMM0 から YMM15 までの 16 個で、YMM レジスタの下位の 128 bit 部分は、XMM レジスタと共用している。AVX では、SSE の命令を用いて XMM レジスタを使用することもできるが、AVX-128 の命令で XMM のレジスタを利用することもできる。

SSE の命令は、オペランドが 2 つの命令が基本であり、命令実行時の入力レジスタの片方が出力にも用いられた。これに対して、AVX ではオペランドを 3 つ用いて、入力レジスタ値を破壊することなく演算できるように改善された。また、AVX では、レジスタ要素間の通常の演算の他に、Fused Multiply-Add (FMA) 演算も追加している。この演算では、3 つの YMM レジスタを用いて $a \times b + c$ の演算を 1 つの SIMD 命令で実行することができるので、さらに演算の効率を上げることができる。

このように、SIMD 命令は、時代の要請するアプリケーションに対応すべく、命令を追加しながら進化し、さらにビット幅を広げることによりプロセッサの性能向上に貢献してきている。

2.6 ステンシル並列計算

本研究では、データ並列計算としてステンシル計算を対象とすることにした。ステンシル計算は、熱拡散、流体力学、電磁気解析等の様々な計算科学アプリケーションに用いられる最も重要な計算手法の一つである。また、比較的容易に並列計算機向けの並列化した計算コードを書くことができ、様々な並列計算機向けに並列プログラミングを行うことができる。本研究で用いた 2 次元の空間のラプラス方程式のヤコビ法により解く例を用いてステンシル計算を説明する。

次のような x と y の関数 f の偏微分方程式はラプラス方程式と呼ばれ、これを解く方法として、ステンシル計算が上げられる。

$$\Delta = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \tag{2.1}$$

いま、この方程式の解 $f(x, y)$ を x 方向に微小変位 a だけ加減算して 3 次の項までテイラー展開すると、次の 2 式が得られる。

$$f(x + a, y) \simeq f(x, y) + \frac{\partial f}{\partial x}a + \frac{1}{2!}\frac{\partial^2 f}{\partial x^2}a^2 + \frac{1}{3!}\frac{\partial^3 f}{\partial x^3}a^3 \tag{2.2}$$

$$f(x - a, y) \simeq f(x, y) - \frac{\partial f}{\partial x}a + \frac{1}{2!}\frac{\partial^2 f}{\partial x^2}a^2 - \frac{1}{3!}\frac{\partial^3 f}{\partial x^3}a^3 \tag{2.3}$$

この 2 式を加算して変形すると、次式が得られる。

$$\frac{\partial^2 f}{\partial x^2} \simeq \frac{1}{a^2}\{f(x+a,y) - 2f(x,y) + f(x-a,y)\} \tag{2.4}$$

同様に、$f(x,y)$ を y 方向に微小変位 a だけ加減算することで次式が得られ、

$$\frac{\partial^2 f}{\partial y^2} \simeq \frac{1}{a^2}\{f(x,y+a) - 2f(x,y) + f(x,y-a)\} \tag{2.5}$$

これらの 2 つの式を元の 2 次元ラプラス 方程式 (2.1) に代入すれば、 次式が得られる。

$$f(x+a,y) + f(x-a,y) + f(x,y+a) + f(x,y-a) \simeq 4f(x,y) \tag{2.6}$$

この式 (2.6) は、微小値 a が十分小さければ、等式として扱える。そこで、関数 f を解く空間を十分に小さい幅 a の 2 次元の格子上の値として差分化する。図 2.10 に、差分化に用いた 10000×10000 の 2 次元の配列 u[10000][10000] を示した。

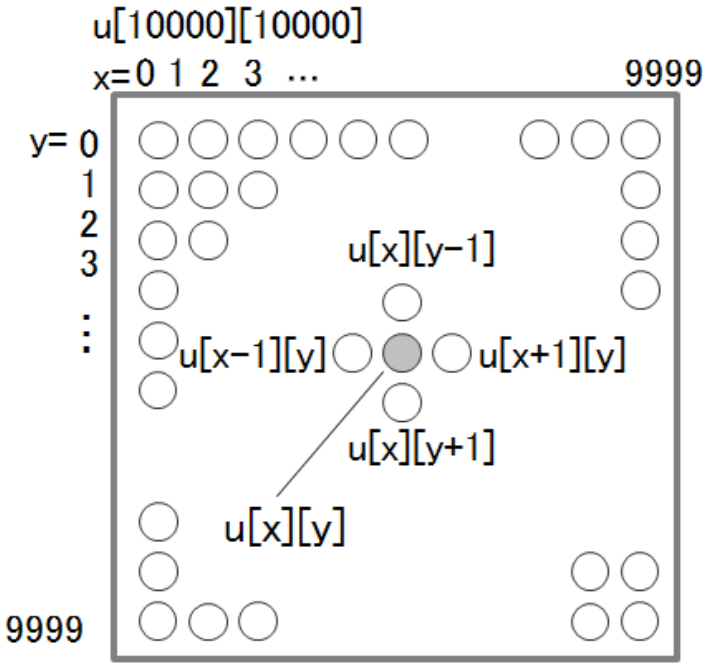

図 2.10: 2 次元 Laplace 方程式の差分化

図 2.10 の○で示される配列 u の要素は、それぞれ元の関数 f の空間の x 方向と y 方向に $a \times x$ または $a \times y$ の地点の関数値を示している。すると、式 (2.6) の f を差分化した u に替えてこれを変形した次式がステンシル式として使用できる。

$$u\_update(x,y) = \frac{1}{4}\{u(x+1,y) + u(x-1,y) + u(x,y+1) + u(x,y-1)\} \tag{2.7}$$

式 (2.7) の $u\_update(x,y)$ は、繰り返し演算で得られるの $u(x,y)$ 新しい値を示しており、図 2.10 に示すように、$u(x,y)$ の上下左右の隣の値の平均値をとる。ヤコビ法でこの繰り返し演算を行うプログラムの一部を図 2.11 に示す。

```
1  #define XSIZE (10000)
2  #define YSIZE XSIZE
3
4  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
5  ...
6    /* Jacobi iteration */
7    for(l=0, l<N, l++){
8
9      /* old <- new */
10     for(x = 1; x < XSIZE-1; x++)
11       for(y = 1; y < YSIZE-1; y++)
12         uu[x][y] = u[x][y];
13
14     /* update */
15     for(x = 1; x < XSIZE-1; x++)
16       for(y = 1; y < YSIZE-1; y++)
17         u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
18   }
19 ...
```

図 2.11: Laplace プログラム

このプログラムではまず、演算空間となる同じ大きさの 2 つの配列、u[10000][10000] と uu[10000][10000] を宣言している。ヤコビ法では、空間全体の値を一斉に更新するため、演算空間の u[][] の他にループ中で演算の途中経過を保持するための配列が必要であり、この目的に uu[][] を用いる。

図 2.11 の 7 行めループが繰り返し演算を行うためのもであり、その中に、演算データの準備とステンシル演算を行う 2 つループが含まれている。10 行目のループは、演算データの準備を行うもので、一定回数の繰り返し後の演算結果を示す配列 u の値を演算用の配列 uu にコピーしている。同図の 15 行目のループがステンシル演算を行うものである。

この様に、ステンシル計算では、2 次元や 3 次元の空間格子点に対する、近傍格子点の値を用いた計算を、全格子点に対して行う作業を繰り返し行う必要がある。また、例に示したように、繰り返し演算中に含まれる演算空間を走査する 2 つのループには、ループ依存が無いため、並列化が可能である。一般には、この空間格子の大きさが十分に大きいため、この格子点を分割して複数の計算機上で実行することにより、データ並列計算を行うことができる。

このようにして実行されるステンシル並列計算の性能を向上させるには次の 2 点に留意する必要がある。

(1) 性能を律速するメモリ転送幅

ステンシル演算中の多くの演算は、は配列の特定要素の近傍の複数点の要素自身かまたはその定数倍の値を加減算である。したがって、データ参照回数と演算回数があまり変わらない。この結果、ステンシル演算の性能はメモリ転送性能に大きく依存する。

(2) 格子点の近傍点参照のローカリティ

ステンシル演算は、特定の格子点の近傍の格子点の値を用いて行われる。これらの近傍格子点の値をキャッシュ等上に置くようにすることにより、性能を向上させることができる。

メニーコアプロセッサは性能向上のために、GDDR メモリを使用するなどして、メモリ転送幅を大きくしている。したがってステンシル計算にも対応できる。しかし、近傍格子点演算でのローカリティの利用は、コア数が多いため注意を要する。不用意に多くのコアがその隣接コアとのデータの交換を行うと、キャッシュ間のコヒーレンシを保つための通信が起こる可能性がある。

2.7 並列計算モデルとプログラミング言語

並列計算を行うための言語は、様々な目的に様々なものが提案されている。ステンシル計算のように科学技術目的には、従来 C/C++ と Fortran が広く用いられてきたため、これらの言語を拡張して並列計算を可能にしたものが多く提案され用いられている。本研究でも、これらの従来言語を拡張したものを用いることにした。

これらの並列計算用の言語は、使用する計算機が並列実行するプロセッサ間でメモリを共有できるかどうかで大きく 2 つに分けられる。共有できるメモリを持つ並列計算機用の言語拡張としては、OpenMP[2] が、最も広く用いられている。OpenMP はベース言語で書かれたプログラム中に、プラグマやコメントを用いて並列化のための指示を記入できる言語拡張で、主要なコンピュータ関連のハードウェア、ソフトウェアメーカの作った非営利団体 OpenMP ARB により、仕様が策定され保守されている。

一方、共有メモリを使えないプロセッサ間での並列計算には Message Passing Interface[23] (MPI) が広く用いられている。こちらは、プロセッサ間でデータを送信し受信するための API を提供するもので、OpenMP と同様なコンピュータ関連企業と大学関係者で構成される MPI Forum により仕様が策定されている。

MPI によるプログラミングは、プログラマに対して明示的なデータの分配を要請する。プログラマは、独立したメモリ空間をもつ各プロセッサの持つデータを管理して、必要に応じてプロセッサ間での通信を行う必要がある。このような、煩雑性を回避するために、新たに、Partitioned Global Address Space (PGAS)[18] と呼ばれるモデルを用いた言語として、Unified Parallel C（UPC）や本研究で用いる XcalableMP (XMP) が誕生してきた。これらの言語では、共有メモリを使えないプロセッサ間に、共通に使えるグローバルアドレスを提供するため、煩雑な通信を記述しなくてもプロセッサ間でデータの交換が行え、プログラムが容易になる、

本研究の対象のメニーコア並列計算機のプロセッサ・ノード内では、コア間で共有メモリを使用することができる。また、ノード間は Infiniband のような、一般的な高速のファブリックで接続することを想定している。したがって、ノード間では共有メモリを持たず、明示的なデータの交換を行う必要がある。そこで、これら OpenMP と MPI の並列計算用の言語拡張について、C 言語をベースにしたものについて概要を説明する。

図 2.12 に前述した Laplace のプログラム図 2.11 を OpenMP を用いて並列化したプログラムを示す。図の 10 行目と 16 行目に C の pragma を用いて OpenMP の指示文である、#pragma omp parallel for を挿入している。これは、その下に続く for 文を複数のスレッドで実行することを指示している。両方とも、次は x を 1 から XSIZE-2 まで増加させる for 文であり、本来ならば、x の値を変化させ、ループの内側を逐次実行するべきであるが、この指示により、複数のスレッドで同時に実行される。

```
1  #define XSIZE (10000)
2  #define YSIZE XSIZE
3
4  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
5  ...
6    /* Jacobi iteration */
7    for(l=0, l<N, l++){
8
9      /* old <- new */
10 #pragma omp paralell for
11       for(x = 1; x < XSIZE-1; x++)
12         for(y = 1; y < YSIZE-1; y++)
13           uu[x][y] = u[x][y];
14
15     /* update */
16 #pragma omp paralell for
17       for(x = 1; x < XSIZE-1; x++)
18         for(y = 1; y < YSIZE-1; y++)
19           u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
20   }
21 ...
```

図 2.12: Laplace の OpenMP 並列化プログラム

実際に何個のスレッドで実行するのか、また、並列に実行する際に x の値をどのように各スレッドに割り当てるのかは、プログラム中の指示を増やしてさらに指定するか、またはプログラム実行時の環境変数によって制御することができる。このように、この並列化の方法では、一つのプログラムを複数のスレッドが実行し、配列の異なる部分のデータを処理する。このような、データ並列計算のプログラミング方法は、Single Program Multiple Data[8] (SPMD) モデルと呼ばれる。

OpenMP を用いたプログラムは、一般に共有メモリ上の複数のスレッドで実行される。プログラム中の全スレッドが参照する配列、例えば Laplace の u[][] と uu[][] はこの共有メモリ上に置かれるために、すべてのスレッドから参照することができる。しかし、クラスタシステムのノード間のように、このような共有メモリを持たない場合は、データの分割後に分割されたデータを処理するスレッドが、別のスレッドの持つデータを必要とする場合は、明示的に 2 つのスレッド間でデータを転送する必要がある。

ステンシル計算で演算空間を別々のアドレス空間に分割する場合、ステンシル演算で必要な演算要素の近傍の配列要素を明示的に転送する必要が生じる。図 2.13 に、演算空間を 4 つのノードに分割した場合の例を示す。図のノード 2 上の配列の灰色で示した要素 u[x][y] のようにデータ分割の境界に接した要素を計算する場合、図 2.13 の右側に拡大して示したように、ノードの境界の外側の要素の値が必要となる。この場合、u[x-1][y] はノード 2 上にはないので、この値は、ノード 1 から転送する必要が生じる。すなわち、分割した配列の分割境界に接した要素の演算には、その境界の反対側のノードの配列要素のデータが必要となる。

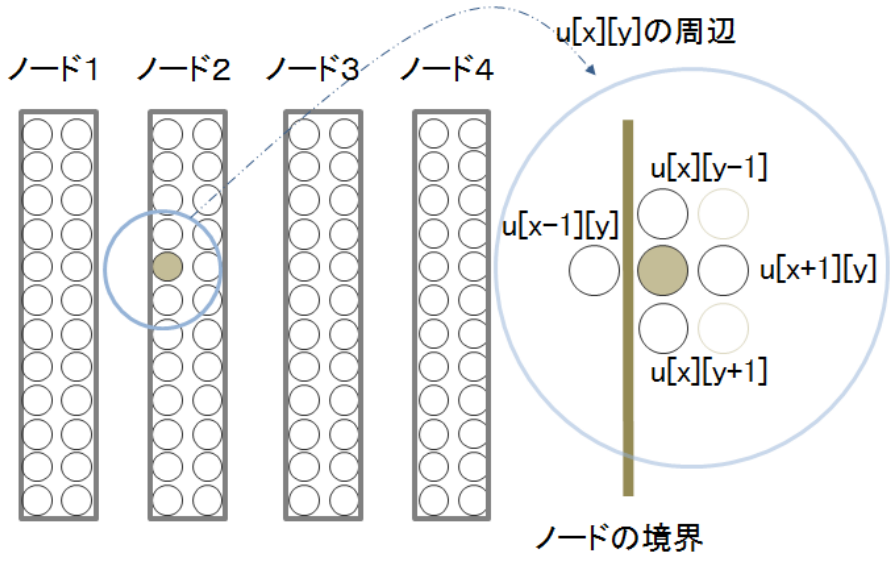

図 2.13: 並列境界のステンシル計算

そこで、ステンシル計算の演算空間配列の分割には、図 2.14 に示すように演算空間の配列を分割する際に、袖領域と呼ばれる、分割境界の隣の領域を付加して分割することがよく行われる。図 2.14 の各ノードでは、実線の矩形内の要素の演算を行うが、この際、隣のノードの端のデータを、予め袖領域と呼ばれる破線で囲まれた領域に複写して持っているため、演算中にデータの転送を行う必要は無くなる。ステンシル演算を開始する直前に、新しく計算された配列要素のうち、分割境界の要素を隣接ノード間で交換する、袖領域の更新のための通信が必要となる。

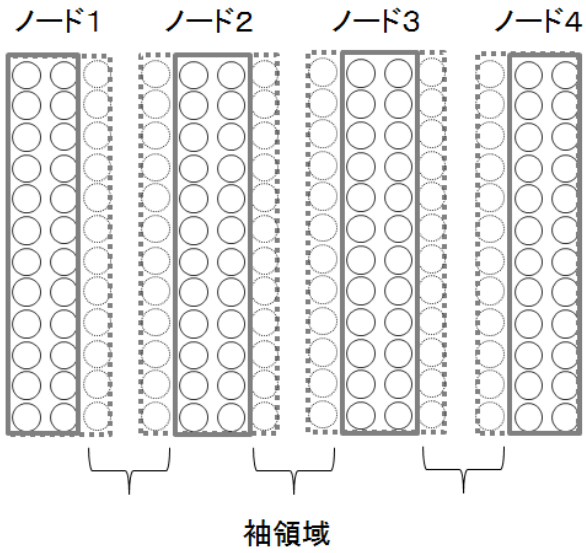

図 2.14: 袖領域付きの配列分割

袖領域を付けて配列を分割する、MPI による並列計算プログラムの主要部分を、図 2.15 に示す。一般に MPI プログラムは、独立したアドレス空間を持つ複数のプロセスで実行される。図の 3 行目にでは、並列数 nproc で演算空間の配列を x 方向に分割するために、XSIZE から分割後の x 方向の幅、xsize を定義している。4 行目と 5 行目は、袖領域の交換を行う隣接するプロセスを指定するためのマクロで、prev(x) は x が 0 でない場合は、プロセス x の手前のプロセスが x-1 と定義されている。x が 0 の場合は、手前のプロセスは存在しないため、MPI で存在しないプロセスを示す定数、MPI_PROC_NULL と定義される。next(x) も同様に x が最大のプロセスの id（MPI では rank と称される）、npoc-1 より小さい場合は x+1 で、それ以外は MPI_PROC_NULL と定義される。

図 2.15 の 7 行目は袖付で分割された演算空間を格納するための配列の確保で、xsize に袖領域分の 2 を加算して u[xsize+2][YSIZE] と uu[xsize+2][YSIZE] の 2 つの配列を確保している。9 行目と 10 行目では MPI で利用できる API、MPI_Comm_size() と MPI_Comm_rank() を用いて、このプログラムを実行する全プロセス数 nproc と、プログラムを実行しているプ

```
1  #define XSIZE (10000)
2  #define YSIZE XSIZE
3  #define xsize XSIZE/nproc
4  #define prev(x) ((x!=0)?x−1:MPI_PROC_NULL)
5  #define next(x) ((x<nproc−1)?x+1:MPI_PROC_NULL)
6  ...
7  double u[xsize+2][YSIZE], uu[xsize+2][YSIZE];
8  ...
9     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
10    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
11 ...
12    /* Jacobi iteration */
13    for(l=0, l<N, l++){
14       /* sode update */
15      MPI_Sendrecv(&u[1][1], YSIZE, MPI_DOUBLE, prev(myid), 202, &u[xsize+1][1], YSIZE,
              MPI_DOUBLE, next(myid), 202, MPI_COMM_WORLD, &stat);
16      MPI_Sendrecv(&u[xsize][1], YSIZE, MPI_DOUBLE, next(myid), 203, &u[0][1], YSIZE,
              MPI_DOUBLE, prev(myid), 203, MPI_COMM_WORLD, &stat);
17
18       /* old <− new */
19       for(x = 1; x < xsize+1; x++)
20          for(y = 1; y < YSIZE−1; y++)
21             uu[x][y] = u[x][y];
22
23       /* update */
24       for(x = 1; x < xsize+1; x++)
25          for(y = 1; y < YSIZE−1; y++)
26             u[x][y] = (uu[x−1][y] + uu[x+1][y] + uu[x][y−1] + uu[x][y+1])/4.0;
27    }
28 ...
```

図 2.15: Laplace の MPI 並列化プログラム

ロセスの rank 値 myid を得ている。このとき用いられている MPI_COMM_WORLD は、この MPI プログラムを実行する全ノードが参加する通信領域に関する処理である指定をするためのものである。

このプログラム中の 14 行目から始まる袖領域の更新の部分が、この Laplace のプログラムに新たに追加された部分である。15 行目では、MPI_Sendrecv() を用いて、自分の演算した配列の最初の列 u[1][1] から YSIZE の長さの倍精度データを、自分の前のノード、prev(myid) に送り、一方、next(myid) より受信したデータを、自分の持つ袖領域の列 u[xsize+1][1] より YSIZE に格納する。16 行目では、MPI_Sendrecv() を用いて、自分の演算した配列の最後の列 u[xsize][1] から YSIZE の長さの倍精度データを、自分の後ろのノード、next(myid) に送り、一方、prev(myid) より受信したデータを、自分の持つ袖領域の列 u[0][1] より YSIZE に格納する。このように、MPI を用いたステンシル計算のデータ並列プログラムは、袖領域をを付加したデータ分割と袖領域の更新を明示的なデータ通信 API を利用して行うことで構成することができる。

このように、ステンシル計算を並列実行するプログラムを OpenMP や MPI で記述するこ

とができ、それぞれ次のような特長を持っている。

(1) OpenMP による並列化には共有メモリが前提

ステンシル並列計算では演算空間の配列のすべての要素をループを用いてスキャンして演算を行うため、OpenMP では、外側のループを、pragma を用いて並列ループとして指示することで、比較的容易に並列化することができる。しかし、この場合、配列をすべてのスレッドが直接参照できるように、スレッド間で共有メモリが必要である。

(2) MPI による並列化では、データの分割と更新の考察と、それらを実現するためのプログラムの追加が必要

共有メモリの存在を仮定しない MPI を用いた場合、データの分割方法は、袖付きの分割等、プログラマが設計し、それに基づいてプログラムを変更しなければならない。また、ノード間でデータの交換が必要な場合は明示的にこれを行う必要があり、プログラムが煩雑になる。

2.8 PGAS 言語

本研究で対象としているメニーコアプロセッサのクラスタシステムでは、クラスタを構成するメニーコアプロセッサ・ノード間にプロセッサ間の共有メモリは無い。したがってクラスタ全体の並列化に、そのまま OpenMP を用いることはできない。一方、MPI による並列化では煩雑なプログラミングが必要で、プログラミングの生産性や保守性に問題がある。近年、このような問題点を解決する手段として、Partitioned Global Address Space (PGAS) モデルが提案されている。この言語モデルの特長は、独立したアドレス空間を持つノードをクラスタ接続した並列計算機に、ノード全体を参照できる広域的な共用アドレス空間を提供し、かつノード内等の局所的なアドレスを区別できるようにする手段を提供するものである。この PGAS を用いた言語の代表的なものとしては、Unified Parallel C[31] (UPC)、Co-Array Fortran (COF)[5]、XcalableMP[33] (XMP)、 X10、 Chapel[4] 等があげられる。

本研究では、クラスタシステム全体にわたる高い並列性を記述するには、PGAS を用いた並列プログラミング言語の XcalableMP[33] を用いることにした。この XMP は、グローバルビューを用いて計算ノード間にまたがるグローバルな配列を記述することができる。さらに、template を用いて、そのグローバルな配列をどのように計算ノード間に分配するかを指示することができ、その結果、プログラマはプログラム中の複数の配列のノード間

の分配方法を分割次元やローカル（全計算ノードにコピー）等を含めて指示することができる。

さらにXMPでは、DO文等で構成する繰り返し空間もtemplateで配列等のデータ空間と対応づけることができ、また、ステンシル計算等の処理に有効な袖領域の設定もできる。この結果、演算時のノード間のデータの流れを指定することができるので、メニーコアの持つノードやコアレベルの高い並列性の活用については、このXMPの実行時システムを改良することで対応できると考えられる。

第3章　Xeon Phi プロセッサ

本章では、本研究で最適化の対象として用いたインテル社のメニーコアプロセッサ Xeon Phi E-7100 の特徴について述べる。

3.1　プロセッサの構成

図 3.1 に、本研究で用いたインテル社の Xeon Phi E-7100 のアーキテクチャを示す。Xeon Phi は、基本的に、高速の双方向リングバスに接続された 3 種類の要素から構成されている。まず、1 番目の構成要素は演算用のコアである。Xeon Phi は、それぞれ 512 KB の L2 キャッシュ(L2) とタグディレクトリ (TD) を持った 61 個の演算コアが、このリングバスに接続されて構成されている。その各コアは 4 つのハードウェア・スレッドを持つため、この上で動作する Linux からは 244 個のプロセッサとして、認識される。

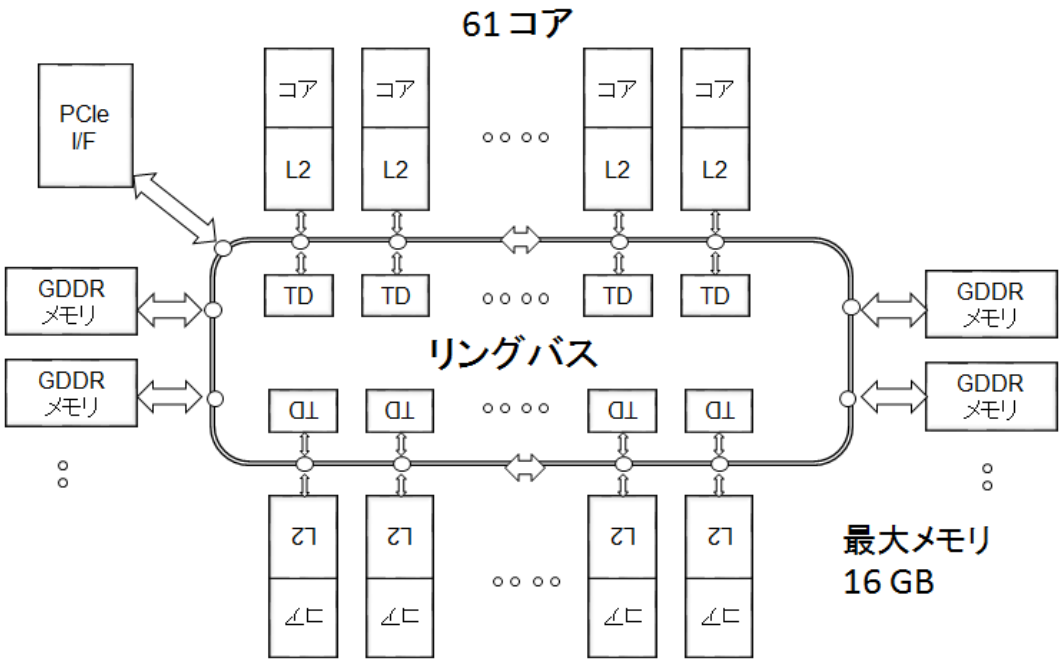

図 3.1: Xeon Phi アーキテクチャ

次に、Xeon Phi は 16 個の GDDR メモリチャネルを持っており、最大 16 GB のメモリを実装することができる。これらのメモリチャネルも、8 個のメモリ・インターフェース回

路を介してこのリングバスに接続されている。コプロセッサ上のすべてのコアは 1 つのアドレス空間を参照でき、ノード上に実装される最大 16 GB の GDDR メモリを、コヒーレントなローカル L2 キャッシュを介して利用することができる。

このコヒーレンシィを 61 個のコア間で保つ処理を高速化するために、各コアは分割したメモリ空間ごとに担当するタグディレクトリを持っている。参照するメモリ空間によって決まるコアのタグディレクトリに問い合わせることで、目的のアドレスのキャッシュラインをどのコアが持っているか、または誰も持っていないか等を知ることができる。そしてハードウェアにより、L2 キャッシュ間のコヒーレンシィが保たれる。最後の構成要素は、PCI Express のインターフェース回路 (PCIe I/F) である。すべての入出力の通信はこの回路を介して外部に要求される。

簡単にまとめると、Xeon Phi は 61 個という汎用マルチプロセッサの数倍のコアを持ち、専用のメモリを持つマルチプロセッサといえる。

3.2 コアの特徴

マルチプロセッサとはいえ、そのコアは多数実装するために小型化され、また倍精度浮動小数点演算性能を向上させるために、SIMD 命令を 512 bit に拡張している。図 3.2 に Xeon Phi のコアの詳細を示す。

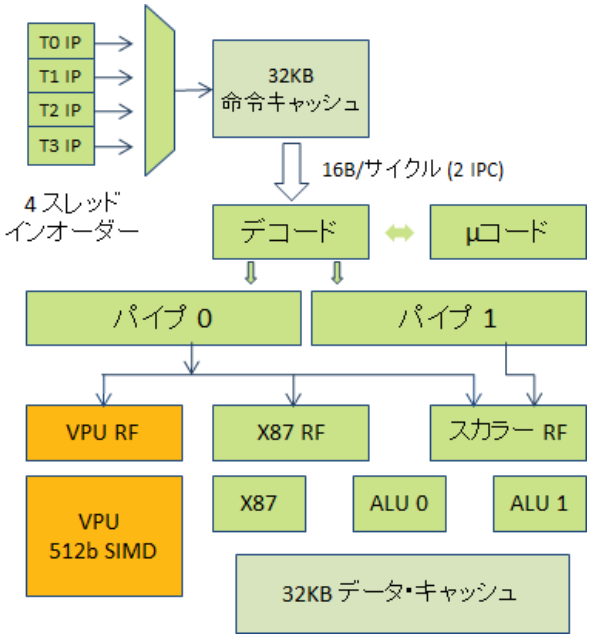

図 3.2: Xeon Phi コアの詳細

まず 、このコアは、2 命令発行の Pentium プロセッサのコアをアドレスを 64 bit に拡張して用いている。このため、32 KB の L1 命令キャッシュから送られた命令は、デコーダで

μコードにデコードされ、2本のパイプ、パイプ0とパイプ1に送られ、最大2命令が同時に実行される。この命令の実行はイン・オーダで、例えば一つの命令がキャッシュミス等で演算データ待ちでストールした場合は、後続の命令はすべて待たされる。このため、このようなレイテンシを隠ぺいするために、レジスタ等のリソースを4つ持ち、4つのスレッドをラウンドロビンで実行する。この結果OSから見ると、1つのコアあたり4つのプロセッサが動作しているように見える。

2つのパイプのうち、パイプ1は従来のスカラのx86命令専用で、パイプ0の方はこれに加えてこのコア用に新たに追加されたVector Processor Unit (VPU)の512 bitのSIMD命令を実行することができる。このVPUは、スレッドあたり32個の512 bitレジスタと8個のマスクレジスタを持っている。そして、FMAを含む算術演算を、16個の32 bit単精度浮動小数点数か整数または、8個の倍精度浮動小数点数に対して実行することができる。さらにこのようなSIMD演算に必要な、Gather, Scatter等100個以上の命令が追加されている。但し、残念なことに、この命令は汎用のXeonの持つ256 bitのSIMD命令である、AVXとの互換性が無い。

3.3 プリフェッチの方法

Xeon Phiプロセッサは、コア間で共用するラストレベル・キャッシュをもっていない。各コアが持つL2キャッシュは、キャッシュ間のコヒーレンシはハードウェアで担保されるものの、そのコア専用にしか使えない。また、このL2キャッシュをミスした場合、リングバスを介してGDDRメモリからデータを読み込む必要が生じるが、このときのレイテンシは、100クロック以上で大きい。したがって、このプロセッサで効率よく演算を行うには、演算に必要なデータを、L2キャッシュさらには、L1キャッシュにプリフェッチすることが、必要条件である。

そこでXeon PhiのL2キャッシュには、コード、読み込みまたは書き込み用のいずれかに、特定のキャッシュラインを自動的にプリフェッチすることのできるハードウェア(HW)・プリフェッチャを持っており、最大4KBページのデータを16個まで、プリフェッチすることができる。このHWプリフェッチャは、プリフェッチの必要性を検知したら、最大、同時に4つまでのプリフェッチを要求できる。

しかし残念ながら、ステンシル計算のように演算時に多くのデータを必要とするような場合には、このHWプリフェッチャだけでは十分でない。なぜならば、まずHWプリフェッチャは、プリフェッチの必要性を検知するには、複数回の等間隔のメモリ参照等が必要で

あり、この間はキャッシュミスが発生する。また、Xeon Phi はL1 キャッシュへの HW プリフェッチャは持っておらず、このため、ループに合わせて、効率的に、L2 から L1 とプリフェッチすることができない。

Xeon Phi はもう一つのプリフェッチ方法としてプリフェッチ命令を持っており、プログラム中でソフトウェア (SW)・プリフェッチの指示を、L1 キャシュや L2 キャッシュに対して行うことができる。図 3.3 に、Laplace の OpenMP 版プログラムで SW プリフェッチを行う例を示す。

```
1  ...
2  #pragma omp for
3          for(x = 1; x <= XSIZE; x++)
4            for(y = 1; y <= YSIZE; y++){
5              _mm_prefetch((const char *)&uu[x][y+32],_MM_HINT_ET1);
6              _mm_prefetch((const char *)&uu[x][y+8],_MM_HINT_ET0);
7              _mm_prefetch((const char *)&u[x][y+32],_MM_HINT_T1);
8              _mm_prefetch((const char *)&u[x][y+8],_MM_HINT_T0);
9              uu[x][y] = u[x][y];
10           }
11         /* update */
12 #pragma omp for
13         for(x = 1; x <= XSIZE; x++)
14           for(y = 1; y <= YSIZE; y++){
15             _mm_prefetch((const char *)&uu[x-1][y+32],_MM_HINT_T1);
16             _mm_prefetch((const char *)&uu[x-1][y+8],_MM_HINT_T0);
17             _mm_prefetch((const char *)&uu[x+1][y+32],_MM_HINT_T1);
18             _mm_prefetch((const char *)&uu[x+1][y+8],_MM_HINT_T0);
19             _mm_prefetch((const char *)&uu[x][y-1+32],_MM_HINT_T1);
20             _mm_prefetch((const char *)&uu[x][y-1+8],_MM_HINT_T0);
21             _mm_prefetch((const char *)&u[x][y+32],_MM_HINT_ET1);
22             _mm_prefetch((const char *)&u[x][y+8],_MM_HINT_ET0);
23             u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
24           }
25 ...
```

図 3.3: Laplace プログラムのソフトウェア・プリフェッチ

まず、図 3.3 の最初の 3 行目と 4 行目の 2 重ループでは、9 行目がループ本体の実行文である。そして、5 行目から 8 行目までは、SW プリフェッチを行うインテルコンパイラの内部関数_mm_prefetch() を呼び出している。この関数の最初の引数は、プリフェッチするアドレスを指しており、2 つ目の引数はプリフェッチの種類を指定している。この種類を表 3.1 に示す。

図 3.3 の 9 行目で、u[x][y] を参照することから、8 行目で、それよりも 64 byte 先の u[x][y+8]、つまり次のキャッシュラインを L1 キャッシュにプリフェッチしている。また、7 行目では、4 キャッシュライン先の u[x][y+32] を今度は L2 キャッシュにプリフェッチしている。また、同じ 9 行目で uu[x][y] に書き込むことから、今度は、6 行目と 5 行目で、こちら

表 3.1: プリフェッチ命令の種類

引数	プリフェッチの種類
_MM_HINT_T0	L1 キャッシュにプリフェッチ
_MM_HINT_T1	L2 キャッシュにプリフェッチ
_MM_HINT_ET0	L1 キャッシュに所有権付きプリフェッチ
_MM_HINT_ET1	L2 キャッシュに所有権付きプリフェッチ

は所有権付で、それぞれ L1 と L2 のキャッシュへ同様なプリフェッチを行っている。

このプログラム、図 3.3 の 13 行目と 14 行目の 2 重ループでも同様なプリフェッチが行われているすなわち、23 行目のループの本体で参照する uu[x-1][y], uu[x+1][y], uu[x][y-1] に対して前のループと同様なプリフェッチを 15 行目から 20 行目で行い、書き込む u[x][y] に対しても同様な所有権付きのプリフェッチを、21 行目と 22 行目で行っている。ここでは、uu[x][y+1] も参照するが、これは、uu[x][y-1] と同じキャッシュライン上にあるので、別途プリフェッチする必要はない。

ここでは、内部関数で SW プリフェッチを行う例を示した。実際は、プログラマが明示的に、このような指示をしなくても、プログラム上で参照パターンが明確であれば、コンパイラが自動的に SW プリフェッチの命令を挿入してくれる場合が多い。なお、Xeon Phi は、ハードウェアと SW 合わせて L1 と L2 で最大コアあたり同時に 38 個のプリフェッチ命令の処理を行うことができる。

3.4 Xeon Phi システム構成

現在市場に提供されている Xeon Phi は、独立してプログラムを実行できるが、単独で動作することはできない。高速グラフィックカード等と同様な PCI Express のインターフェースを持ったプロセッサボードとして提供されている。図 3.4 に Xeon Phi を使用する場合の一般的なシステム構成を示した。図の左側は、汎用のホストプロセッサで、上下 2 個のプロセッサを搭載したサーバシステムである。2 個のプロセッサは Quick Path Interconnect (QPI) で接続されており、それぞれのプロセッサには DDR メモリが搭載されている。また、プロセッサからはこの他入出力用の PCI Express のバスが提供されており、これを介して Disk や Infiniband (IBA) ネットワークに接続することができる。

右側の Xeon Phi のプロセッサボードも、この PCI Express を介してサーバに接続されて

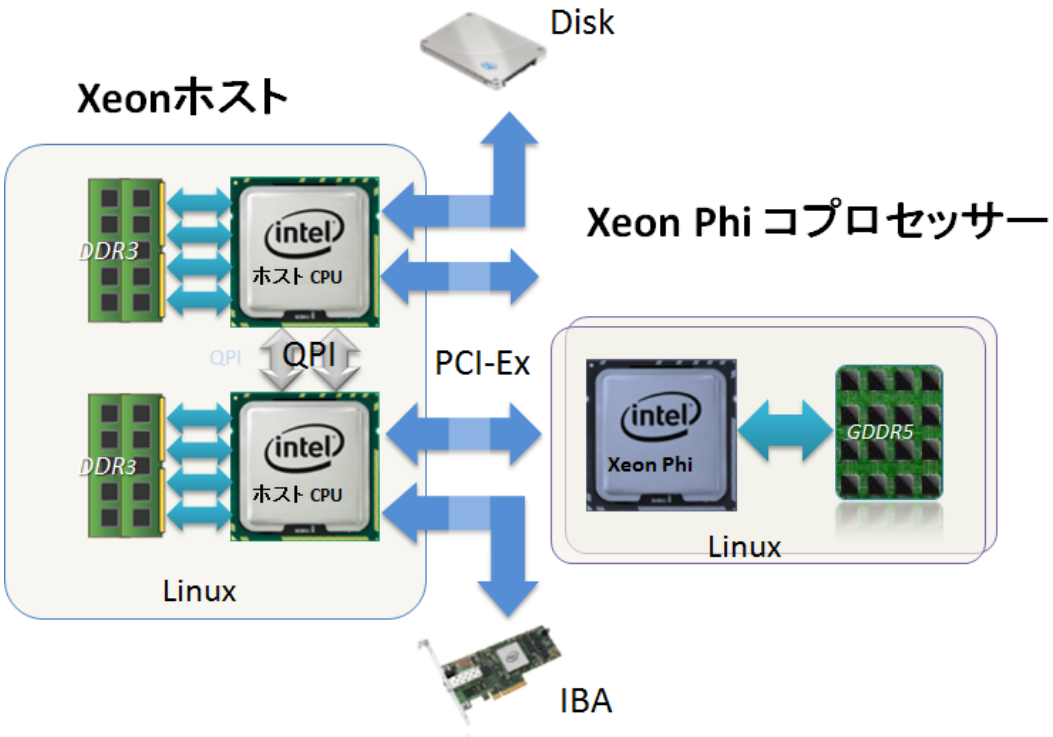

図 3.4: Xeon Phi サーバ・ノード

いる。Xeon Phi は専用のメモリとして、最大 16 GB の GDDR5 のメモリを搭載しており、サーバとは独立して Linux 等の OS を動作させることができる。Xeon Phi を動作させるには、Xeon から Manycore Platform Software Stack (MPSS) を用いて Xeon Phi を boot する必要がある。この MPSS 中には、Xeon Phi が動作するために必要な、基本的な IO ドライバや Xeon Phi 上で動作する Linux のイメージファイルが含まれる。これらのファイルは、この MPSS に含まれるツールで、PCI Express 経由で Xeon Phi のメモリ上に展開される。そして、Xeon Phi を reset することにより、Xeon Phi は Linux を boot して Xeon とは独立した計算機として動作を開始できる。

ただし、Xeon Phi は専用の入出力装置を持っていない。このシステムに接続されたハードディスクや IBA の利用には、Xeon プロセッサや Xeon 内部の DMA エンジンの助けを借りることになる。別のサーバに搭載された Xeon Phi との通信は、ホスト上の IBA のインターフェースを介して行う必要がある。

3.5 汎用の Xeon プロセッサとの比較

表 3.2 に、本研究で使用したメニーコアプロセッサ Xeon Phi E-7100 の仕様を、同世代の汎用プロセッサ Xeon E5-2600v2 と比較して示す。まず、演算コアを小さくするために、クロックあたりの命令実行数を 2 に減らしている。これは、Pentium と呼ばれる 1999 年まで

250 nm の半導体プロセスで作られていたプロセッサと同等である。つまり、現在の 22 nm に対して、100 分の 1 以下の数のトランジスタ数で作っていたプロセッサのコアに相当する。さらに、コアあたりの消費電力を下げるために、Xeon Phi の周波数は Xeon の 2.7GHz に対して 1.238GHz と半分以下に設定されている。一方、SIMD 演算の SIMD 幅は、Xeon の 256 bit に対して、Xeon Phi では 2 倍の 512 bit と 8 倍精度演算を同時に行えるように SIMD 命令の実行回路を拡張している。したがって、Xeon Phi のコア当たりの倍精度演算ピーク性能は、18 GFLOPS と、Xeon の 21. 6 GFLOPS に対して 2 割弱程度劣る程度である。

表 3.2: メニーコアプロセッサの仕様

項目	Xeon	Xeon Phi
クロックあたりの実行命令数	4	2
周波数	2.7 GHz	1.238 GHz
SIMD 幅	256 bit	512 bit
コア当たり倍精度演算ピーク	21.6 GFLOPS	18.0 GFLOPS
コア数	12	61
プロセッサ倍精度演算ピーク	259 GFLOPS	1100 GFLOPS
ピークメモリ転送幅	60 GB/S	350 GB/S
コア当たり最大キャッシュ容量	30MB	512KB
コア当たりのスレッド数	2	4

そして Xeon Phi では、この小型化して低消費電力化したコアを採用することで、プロセッサのコアの数を 61 に増やすことができている。この結果、プロセッサの倍精度演算性能は、Xeon の 259 GFLOPS の 4 倍以上の 1100 GFLOPS を実現している。これに合わせて、プロセッサあたりのピークメモリ転送幅も、60 GB/S から 350 GB/S に大きく改善している。このように、コアを小型して、コア数を増やすことで浮動小数点演算性能を大きく改善できる。しかし、コア数を増やすことにより問題も発生する。その一つが、1 つのコアが使用できる最大キャッシュ容量である。

3.6 Xeon Phi のプログラミングの問題

Xeon はすべてのコアで共有する 30 MB のラストレベル・キャッシュを持っている。これに対して、コア数の多い Xeon Phi はコア間で共有するキャッシュを持っていない。基本

的に各コアは、ローカルに持つ 512 KB の L2 キャッシュ内で演算を行わなければ、レイテンシが大きいメモリアクセスを行う必要が生じてしまう。小型化したコアがイン・オーダ実行であるため、このようなメモリアクセスや、その他原因で生じるレイテンシを隠蔽するために、Xeon Phi ではコアあたり 4 つのスレッドを持ち、ラウンドロビンでこれをスケジュールしている。

Xeon Phi は、高性能グラフィックス・カードによく似た PCI Express のインターフェースを持った拡張カードとして提供される。標準的な Xeon サーバに挿入接続して使用されるが、独立した計算機として、Linux を立ち上げて動作し、専用の計算資源、メニーコアとメモリで動作する。

Xeon Phi を用いてクラスタを構築する場合、Infiniband 等のアダプタカードを介して Xeon Phi のメモリ間で DMA 等を用いてデータの通信を行うが、これらもリングバスを介してこのインターフェース回路経由で行われる。リングバスは、プロセッサのクロックに同期して、すべてのポートでクロックあたり 512 ビットのデータを同時に入出力ができ、転送帯域幅は大きい。しかし、これを介した参照のレイテンシは、同じリングに物理的につながるポート数に比例して大きくなる。

このように、Xeon Phi はデータ並列計算に適した高い並列性を持つ一方で、命令の処理がイン・オーダである等、単一スレッドでの演算性能に影響する命令レベルの並列性が低いうえ、動作周波数も 1.128 GHz と汎用のマルチプロセッサより低い。また、コア間で共用できるラストレベル・キャッシュを持っておらず、このため、コアが 2 次キャッシュをミスした時のペナルティが大きい。

このようなアーキテクチャ上でのプログラムの並列化と実行方法には次に述べる 2 つの課題がある。

(1) クラスタシステム全体にわたる非常に高い並列性を記述し、実行する必要

(2) ノードやさらにはコアレベルでの局所性を利用した、効率的な並列実行の実現

本研究では、これらの課題を解決するデータ並列プログラムの最適化の検討を行った。

第4章　メニーコア並列システムのプログラミングモデルとPGASモデル

並列計算の実行には、従来通り逐次にプログラムを実行するのに加えて、複数の計算を同時に実行するための機構とそれらの実行計算間でデータの交換や同期を行う手段が必要になる。従来の並列計算機用の言語であるOpenMPやMPIは、それぞれ、共有メモリを持つSMPと、分散メモリのサーバ・クラスタといった異なる並列計算機アーキテクチャと密接に関連した言語であり、そのようなアーキテクチャ上で並列計算を行う手段を提供している。一方、XMPをはじめとするPGAS言語は、分散メモリのアーキテクチャを対象にグローバル共有メモリを提供するために、配列等の変数の新たな宣言方法や、並列計算の記述方法を持っている。本章ではこれらについて説明する。

4.1　OpenMPとMPI

現代のマルチプロセッサ用のLinux等のOSはプログラムを並列実行するために、プロセスとスレッドの2つの実行手段を提供する。プロセスは独立してプログラムを実行できる最小単位で、メモリ上に、実行するプログラム、データ、スタック領域を持ち、最低1個のスレッドが割り当てられる。スレッドはプロセッサのレジスタやプログラム・カウンタ等のプロセッサの実行情報を持つ、プロセッサの実行単位で、プロセッサの持つハードウェア・スレッド（コア）はこのスレッドとしてOSから管理され、プログラムを実行する。OpenMPはこのスレッドを利用して並列計算を行い、またMPIはプロセスを用いて並列計算を行うので、これらの違いについて述べる。

プログラムを実行するシステムコールはexecve()であるが、これは引数で与えられたプログラムの実行ファイルから、プログラム本体等を上述したメモリ領域に展開した後、共に割り当てられたスレッドのプログラム・カウンタを、メモリ上のプログラムの先頭アドレスに設定することで、プログラムの実行を開始させる。OpenMPを用いたプログラムは、実行開始後、#pragma OMP parallelで始まるparallelリージョンに達した時点で、最初のスレッドも含めて、環境変数OMP_NUM_THREADS等で指定した数になるように、スレッド

を生成し、複数のコアで並列に計算が実行される。図 2.12 に示したような#pragma OMP parallel for を用いてループを並列化した場合、ループが分割されて別々のコアで実行される。このとき、すべてのスレッドは、同じデータ領域中のそれぞれのコアが演算に必要な領域を参照することになる。

このようにスレッドを生成するときに、OpenMP の実行時システムは OS のシステムコールを利用して、物理的にどのような順番でコアにそのスレッドを割り当てるかを決める。マルチプロセッサの場合はコア間でキャッシュ等のリソースを共有している場合がある。また、Xeon Phi の場合は、コアあたり 4 つのスレッドを実行することができるので、スレッドを生成する場合に、新しく生成されたスレッドを近くの（あるいは同じ）、リソースを共有したコアに割り当てるのか、それとも遠くのコアに割り当てるのかによって、プログラムの実行性能に大きな差異が生ずることがある。さらに、Xeon Phi のように、ラストレベルの共有キャッシュを持たず、プリフェッチを前提として演算を行うアーキテクチャ上では新たな問題が起きるので、次にこれを説明する。

図 4.1 に OpenMP の並列計算実行時に 2 つのコアが参照するデータ領域を示す。図の左側はコア 0 の行うデータ参照領域、右側はコア 1 のデータ参照領域を示している。それぞれ、演算実行時に参照する配列データの要素を、○を矩形状に並べて示している。コア 0 の場合、演算するのは破線よりも左側の領域であるが、前述したように、例えば図中の灰色の要素を演算するには、その隣接した上下左右のデータ（図中の円内）を参照する必要があるため、矩形で囲まれた領域を参照する必要がある。ところで、3.3 節で説明したように、メニーコアのプロセッサにおいては、ローカルの L2 キャッシュに演算に必要なデータをフェッチする必要がある。したがって、図中の矩形領域は演算前に、L2 キャッシュにプリフェッチされる。並列に演算実行するコア 1 でも同様なため、図の斜線で塗られた領域は、コア 0 とコア 1 の両方の L2 キャッシュにプリフェッチされることになる。

当然ながら、これらの重複領域のデータを片方のコアが更新した場合は、別のコアの L2 キャッシュ上の同データは無効になるため、データの再転送を行わなければならない。この結果、プリフェッチしたのにもかかわらず、演算時に再度データの転送が発生する。このような現象は、すべてのコア間の配列データ境界で起きる。また、分割数が多くなれば、境界数も増えるため、高い並列数での実行を目指すメニーコアでは、特に対策が必要である。この他にも、同一のアドレス空間上の配列を分割して複数のコアで参照すると、L2 キャシュのコヒーレンシを保つために多くのメモリ参照のデータ転送が発生することが懸念される。

一方、MPI で実行する場合は、mpirun -n < 並列数 > のように、プログラム開始時に並列数を指定し、複数のプロセスを生成して、並列にプログラムを実行する。図 4.2 に、この

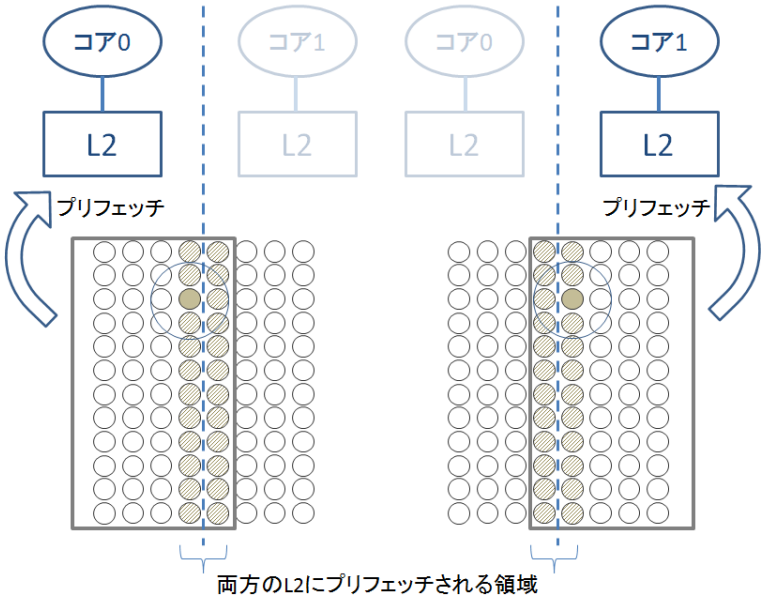

図 4.1: OpenMP での演算領域の参照

場合の各コアのデータ参照領域を示す。OpenMP の場合と異なり、それぞれのコアの演算する矩形で囲まれた領域は、破線で囲まれた袖領域とともに、それぞれのプロセスの独立したアドレス空間に置かれる。このため、演算中に L2 キャッシュにプリフェッチされれば、そのままローカルキャッシュ内で演算を行うことができる。

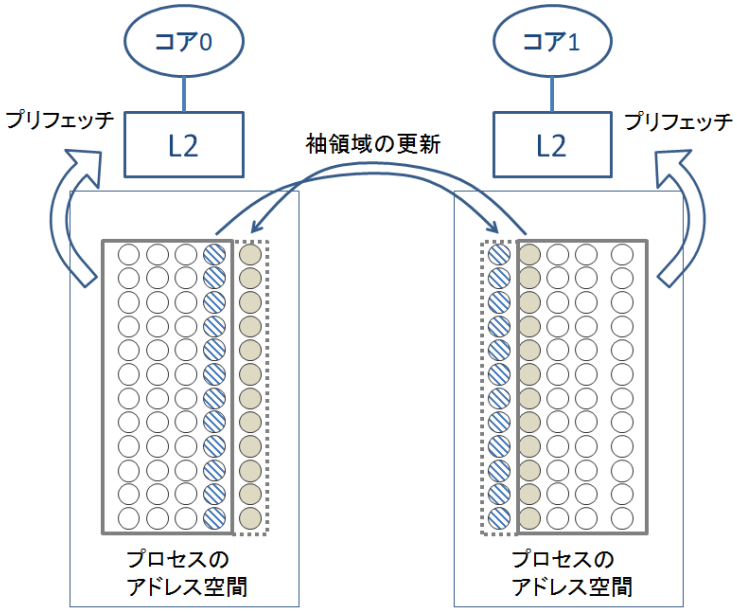

図 4.2: MPI での演算領域の参照

前述したように、袖領域の更新は、MPI の通信 API を使って別途、明示的に行う必要があり、斜線で示すコア 0 の担当する領域境界に隣接する領域を、コア 1 の袖領域に転送し、また逆に灰色で示すコア 1 の担当する領域境界に隣接する領域を、コア 0 の袖領域に転送

する。

以上のべたように、OpenMP や MPI を用いた従来の並列プログラムの実行方式をメニーコアに適用すると、次のような性能上の問題点を持つ。

(1) OpenMP 版の L2 キャッシュのコア間でのシェアリング

アドレス空間を多数のコアで共有する OpenMP 版のプログラムでは、ステンシル演算中に各コアの演算する空間格子点値のうち、各コアが担当する空間の境界領域の値が、その境界の隣接する 2 つのコア間に重複してキャッシュにプリフェッチされる。この領域に片方のコアが書き込むと、もう一つのコアでは、値を再格納する必要が生じるため、プリフェッチが有効に働かない。

(2) MPI 版で追加される MPI 通信時間の影響

MPI 版の場合は、独立したアドレス空間に分割した空間格子点のデータを、袖領域を加えて格納するため、ステンシル演算に追加して、明示的に MPI 通信で袖領域のの更新を行う必要があり、この時間が演算時間に加算される。

4.2 ハイブリッドモデル

一般的なマルチプロセッサのサーバを用いたクラスタシステムで、性能を得るために最もよく用いられるのは、OpenMP と MPI のハイブリッドと呼ばれる並列計算の実行方式である。これは、共有メモリを使用できる個別のサーバ（ノード）内の並列計算は OpenMP で行い、ノード間の並列化は MPI で行う方法である。この方法では、ステンシル計算を実行する際に、ノード内とノード間の 2 種類のデータ並列の計算実行を行うが、これらが、メニーコアプロセッサのクラスタに有効かどうか考察する。

ハイブリッド方式では、基本的に、MPI を用いたノード間の並列化が主な並列化の作業となる。すなわち、空間データを SPMD モデルで袖領域を付加しながら分割するようにプログラムを書き換え、そのようにして出来上がったプログラムに OpenMP のループに対する並列化を適用する。図 4.3 に前述した MPI による並列化プログラムをハイブリッド化したものの、ヤコビ部分を示す。

この方法では、図 4.3 の 9 行目と 15 行目に示すように、MPI により並列化したプログラムの、その並列化の対象となったループ部分に対して OpenMP による並列化を行う。一方、ノード間の並列化に対しては、MPI のライブラリが準備する、RDMA 等を用いたノード間での通信に最適化された API を用いて、効率よく袖領域の交換等の通信を行うことができ

```
1  ...
2    /* Jacobi iteration */
3    for(l=0, l<N, l++){
4      /* sode update */
5      MPI_Sendrecv(&u[1][1], YSIZE, MPI_DOUBLE, prev(myid), 202, &u[xsize+1][1], YSIZE,
           MPI_DOUBLE, next(myid), 202, MPI_COMM_WORLD, &stat);
6      MPI_Sendrecv(&u[xsize][1], YSIZE, MPI_DOUBLE, next(myid), 203, &u[0][1], YSIZE,
           MPI_DOUBLE, prev(myid), 203, MPI_COMM_WORLD, &stat);
7
8      /* old <- new */
9  #pragma omp parallel for
10       for(x = 1; x < xsize+1; x++)
11         for(y = 1; y < YSIZE-1; y++)
12           uu[x][y] = u[x][y];
13
14     /* update */
15 #pragma omp parallel for
16       for(x = 1; x < xsize+1; x++)
17         for(y = 1; y < YSIZE-1; y++)
18           u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
19   }
20 ...
```

図 4.3: Laplace のハイブリッド並列化プログラム

る。したがって、ノード内の OpenMP による並列化で性能向上が得られる場合は、有望な方法といえる。

しかし、ハイブリッドで並列化されたプログラムのメニーコア・クラスタ上での実行には、4.1 節で述べたように、ノード内の OpenMP による並列演算で、L2 キャッシュのコア間でのシェアリングの問題が懸念されメニーコアのノード上での OpenMP のプログラムの実行が適切なのかという疑問が残る。この他、ハイブリッド方式は、プログラマがノード間とノード内の 2 段階の並列化を記述しなければならず、そのプログラム開発の生産性の問題も、従来から指摘されている。

4.3 PGAS モデル

クラスタシステムのように、ノード間で共有メモリを持たない並列計算において、ノード間の通信を行う方法として、4.1 節で MPI を紹介した。また、4.2 節でメニーコアコアクラスタ向けに、この MPI と OpenMP をハイブリッドに用いた並列実行の方法と、その問題点について述べた。これらの問題点の解決するために、Partitioned Global Address Space (PGAS) モデルが提案されている。

図 4.4 に、4 つのノード上で実行する場合の、PGAS モデルの概念図を示す。PGAS モデルとは、図に示すように複数のノードごとに独立したアドレス空間を持つクラスタシステ

ム上で、システム全体に及ぶ、グローバルなアドレス空間を提供するモデルである。さらに、このような空間を、ノードごとのアドレス空間に境界に合わせて分割して管理し、プログラマが使用するアドレス空間が、プログラムを実行するノードにローカルな領域にあるかどうかを、意識してプログラムする手段を提供する。例えば、図 4.4 の矩形で囲まれたノード 0 にローカルな空間上に、ノード 0 だけが必要な変数を置くことにより、クラスタシステムの持つ局所性を活かしたプログラミングが可能となる。

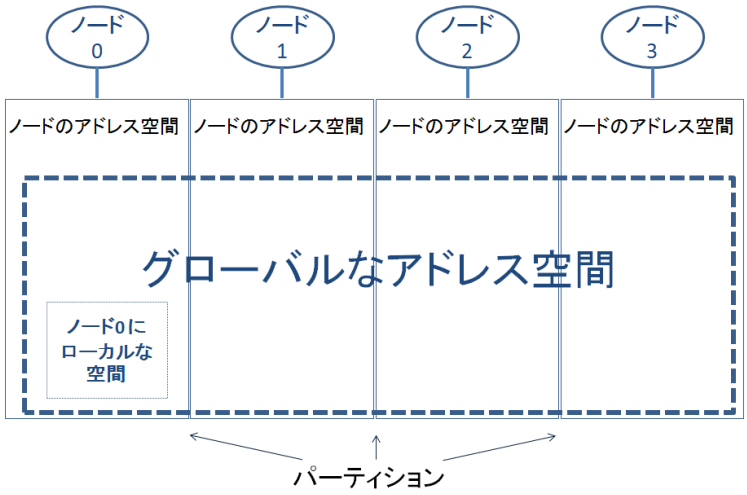

図 4.4: PGAS モデルの概念図

PGAS モデルは、従来の言語に無い、新たな PGAS の空間を提供するモデルであり、従来言語を拡張する形でも実現することができる。また、PGAS の空間を意識してプログラムする方法は様々考えられる。したがって、PGAS モデル利用できる様々な PGAS 言語が提案されている。表 4.1 に、代表的な PGAS 言語を本研究で使用する XMP と比較して示す。表で比較した項目は、ベース言語、グローバルビューの有無とプログラムの並列実行時の実行単位である。

表 4.1: 代表的な PGAS 言語

名称	ベース言語	グローバルビュー	実行単位
XMP	C, Fortran	○（袖領域）	node
Unified Parallel C (UPC)	C		(process 等)
Co-Array Fortran (COF)	Fortran		(process 等)
X-10 (IBM)	(Java and Scala)	○ (region 等)	place+activity
Chapel(Cray)	(C and Modula)	○（domain）	locale

まず、XMP のベース言語は C と Fortran である。そして、UPC は C で、COF は Fortran とこれらの 3 つは既存の科学技術計算で広く用いられている言語をベースにして PGAS モデルを使えるように拡張している。これに対して、IBM 社の X-10 と Cray 社の Chapel は既存の言語を拡張するのではなく、いくつかの言語シンタックスを用いて新たな言語を作っている。C や Fortran で書かれた既存のアプリケーションの移植は、XMP、UPC か COF の 3 つの言語がより適している。

一方、プログラムを並列化する際に、SPMD プログラムの個々のプログラムの挙動を記述するプログラミングの視点をローカルビューと呼ぶとすると、これに対して全体で行うプログラム挙動を記述しさえすれば良い方法は、グローバルビューと呼べる。XMP は後述するように、このグローバルビューをサポートして袖領域の処理を自動化している。また、新しい言語である X-10 と Chapel は、グローバルビューを実現するために、それぞれ region、domain といった新しいオブジェクトを準備している。

グローバルビューを持つ言語は、これを利用して並列に処理行う実行単位として、XMP では node、X-10 や Chapel では place や locale を持っている。これに対して、ローカルビューしか持たない言語では、実行単位としてはインプリシットに、ノードで実行するプロセスが用いられるだけである。

本研究では、既存の科学技術計算が移植し易い点と、グローバルビューをサポートしている点から、XMP を PGAS 言語として用いる。

4.4 PGAS 言語の実行時システム

PGAS 言語はクラスタシステムのような分散メモリ環境上にグローバルなアドレス空間を提供する。そしてこの空間上のデータ交換を、ノード内のローカルメモリ上にある場合でも、他のノードが持っている場合でも言語上で行うことができる。しかし実際は、ノード間をまたいでデータを交換する場合は、何らかの方法でノード間の通信を行ってデータを交換する必要がある。このような機能を実現するために、PGAS 言語ではコンパイルしたプログラムを実行するときに、専用の実行時システムを用いている。

そこで、代表的な PGAS 言語である、UPC の実行時システムについて、その通信部分を担当する GASNet[7] を含めて説明する。GASNet はもともと NERSC の PGAS 言語プロジェクトである UPC の実行時システムとして開発された。前述した PGAS 言語のうち、UPC、COF、 Chapel 等は、その実行時システムとして GASNet を使用している。また、本研究で用いた XMP も一部の機能を実現するために、この使用も考慮されている。

図 4.5 に UPC プログラム実行時の、実行に必要なモジュールの階層構造を示す。GASNet はコア API と拡張 API の 2 つの API から構成されている。コア API は特にプログラムを実行するハードウェアに密接な API で、より一般的な拡張 API はこのコア API を用いて実現されている。さらに、UPC の実行時システムは、この拡張 API を用いて実装されている。UPC のコンパイラでコンパイルされたプログラムはこの実行時システム上で動作をする。

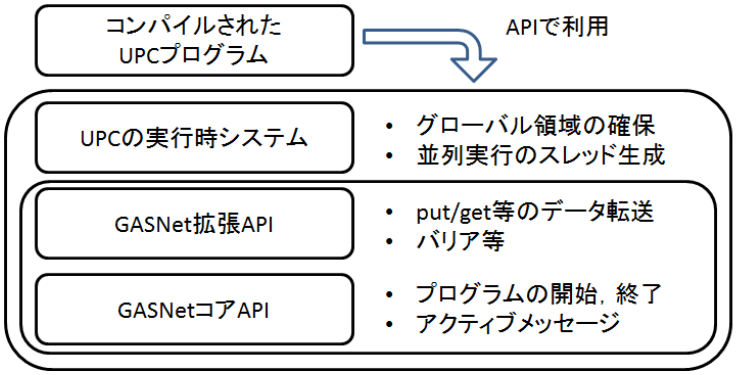

図 4.5: UPC プログラムの実行方法

GASNet のコア API は、SPMD プログラムの開始や終了のジョブコントロールを行う API とアクティブメッセージ [6] と呼ばれる API から構成されている。アクティブメッセージは、軽量の RPC コールを用いて、順序に依存しない信頼性の高いメッセージ通信を実現する。そして、このコア API を用いて拡張 API では、片方向通信の put/get といったメモリ - 転送やバリア等を提供する。UPC の実行時システムは、これらを用いて UPC の機能、特にグローバルアドレス中の他ノードとのデータ交換等を実現している。コンパイルされた UPC の実行プログラムは、UPC の実行時システムの API を利用することで、グローバルなアドレス空間で並列にプログラムを実行することができる。

第5章　XcalableMP言語

XcalableMP (XMP) は OpenMP 等と同様な、指示文を用いてプログラムの並列化を行う言語拡張で、C と Fortran をサポートしている。この言語の仕様は、PC クラスタコンソーシアムの XcalableMP 規格部会で策定されている [32]。本研究ではこの XMP を PGAS 言語として使用したので、本章で、この言語を選んだ理由と概要を述べる。

5.1　XMPの特長

本節では PGAS 言語 XMP の特徴について述べる。XMP は、指示文を挿入してベースとなる言語にプログラムに並列実行を指示する機能を追加できる言語拡張で、ベース言語として、Fortran と C をサポートしている。XMP は並列プログラムの実行モデルとして SPMD を採用しており、並列に処理を行う複数のワーカーが同一の実行モジュールを実行し、複数のデータの処理を同時に行う。分散メモリ計算機環境でのプログラム実行を想定しているのにもかかわらず、 XMP はこれらの点では OpenMP に類似している。ただし OpenMP とは異なり、複数の計算機間にまたがるグローバル配列のデータの分配方法の指示も行うことができる。この指示では分割配列境界の袖領域の有無や大きさを指定できる。本研究ではこの指示を利用して並列化したステンシルコードからの実行モジュールの生成を行う。

下にこれらの特長とこの言語を採用した理由をまとめる。

(1) 対象言語は C と Fortran

現在広く使用されている科学技術計算用のプログラムは C や Fortran で書かれているものが多い。実際に、本研究で対象とした、ステンシル計算の姫野ベンチマークも、これらの言語で書かれている。これらのソフトウェア資産を活かすために、C や Fortran で書かれたプログラムを使えるものが好ましい。

(2) Pragma やコメントによる指示文

並列プログラミング言語の中には、並列化の為にまったく別のプログラムを開発しなければならないものもある。OpenMP のように、pragma やコメントを追加すること

で並列化等に基本的な指示ができれば、オリジナルの逐次版のプログラムを残したまま並列化の作業が行え、開発が容易となる。

(3) グローバルビューによるプログラミング

XMP は並列化のために、グローバルビューとして、HPF 等で採用されたデータ分割の指示方法に改良を加えたものを用いている。グローバルビューは、その下に隠れている実装を、あまり意識する必要がないため、アーキテクチャに依存しないプログラミングができて好ましい。

(4) 実行時システムのノード間の通信に MPI を使用

XMP は実行時システムで行うノード間の通信に MPI を使用している。MPI の API は標準化されて仕様が定義されており、しかも次々に新しく登場するノード間通信手段に最適化されている。実行時システムをメニーコアに最適化する際に、これらを利用できる。

5.2 実行モデル

XMP は、その実行モジュールが、クラスタシステム内の複数のノードのような独立したアドレス空間をもった計算機上で実行されることを想定して作られている。その実行モデルとしては SPMD モデルを採用している。XMP では、並列実行を行う基本的な処理単位を node と呼ぶ。それぞれの node は、全 node で構成されるグローバルなデータに対して、指示文等で処理を行うことができる。

この XMP の実行モデルを図 5.1 に示す。図の node0、node1、node2 のような複数の node が SPMD プログラムを実行する。このとき、それぞれの node が複数のデータ領域を適切に処理するようにプログラムする必要があるが、XMP ではこれを、データ分配、計算分配と通信や同期の 3 種類に分類される指示文により記述してプログラム中に挿入する。

このうち、データ分配は、プログラム中の変数やデータの宣言を行う部分に追加し、これにより、各 node は、計算を担当するデータ領域を知ることができる。残りの 2 つは、いずれも、ループ等のアルゴリズムを記述するプログラム本体部分に追加する。このうち計算分配は、例えばプログラム中のループの前に挿入し、この指示により各 node は自分の実行するループ部分を得ることができる。そして、通信や同期の指示で、プログラム中で全 node でデータを交換し、また同期をとる。

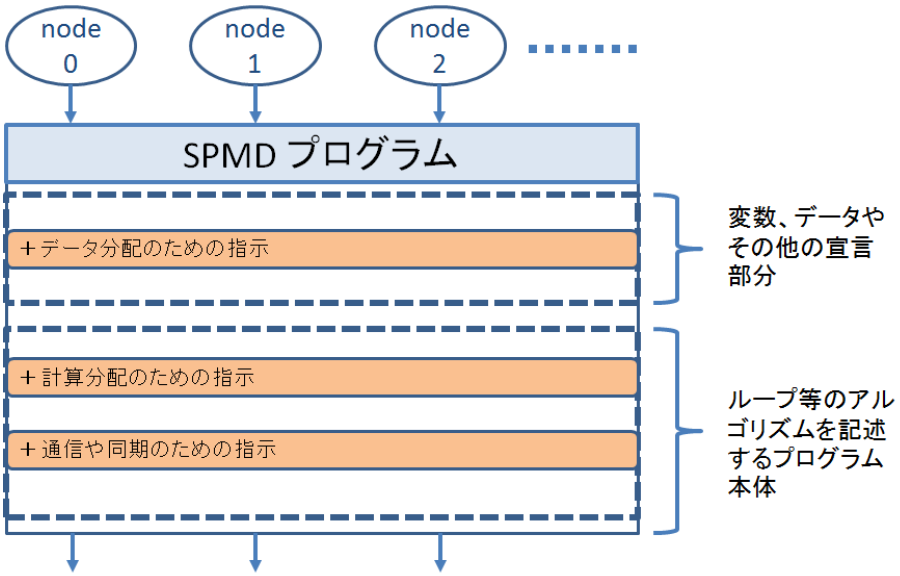

図 5.1: XMP の実行モデル

5.3 グローバルビュー

XMP ではプログラミングのモデルとして、グローバルビューとローカルビューの 2 つのモデルを持っている。XMP のグローバルビューを用いたプログラミングを図 5.2 に示す Laplace の XMP を用いた並列化プログラムで説明する。ステンシルコードの演算空間の配列のように、それぞれの実行プロセスにその配列を分散配置したい場合は、これをグローバル配列として指示する。この指示には、XMP の指示文 align を用いる。プログラム中の配列が align 指示文で、既に分配されている template 変数に関係付けられている場合には、その配列はグローバル配列となる。図 5.2 に、Laplace の 1 次元分割での XMP 版プログラムのうち、データ分配と並列化の部分を示す。

XMP の指示文は C では pragma を用いて#pragma xmp のように記述する。図 5.2 のプログラム 6 行目の nodes 指示文は、n(*) で、n(1) から定数個 (Nc とする) 並んでいることを宣言している。このように指定しておくと、XMP 実行時システムは、実行時に実際に生成されたプロセスの数を node 数として動的に決定し、その数でデータを分割することができる。7 行目の template 指示文は、プログラム中の分散配列を包括的に代表するための仮想行列である template 変数 t(0:10000-1) を定義している。そして 8 行目の distribute 指示文で、この仮想行列 t(0:10000-1) を、文中の t(block) のように、block を置いて示した次元で分割して、Nc 個の node n 上にブロック分割すること指示している。グローバル配列を定義するには、この t(0:10000-1) を利用する。

このプログラム中では、2 つの 2 次元の配列 u[10000][10000] と uu[10000][10000] が定義

```
1  #define XSIZE (10000)
2  #define YSIZE XSIZE
3
4  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
5
6  #pragma xmp nodes n(*)
7  #pragma xmp template t(0:10000-1)
8  #pragma xmp distribute t(block) onto n
9  #pragma xmp align u[j][*] with t(j)
10 #pragma xmp align uu[j][*] with t(j)
11 #pragma xmp shadow uu[1][0]
12
13 ...
14
15     /* old <- new */
16 #pragma xmp loop (x) on t(x)
17     for(x = 1; x < XSIZE-1; x++)
18       for(y = 1; y < YSIZE-1; y++)
19         uu[x][y] = u[x][y];
20
21 #pragma xmp reflect (uu)
22
23     /* update */
24 #pragma xmp loop (x) on t(x)
25     for(x = 1; x < XSIZE-1; x++)
26       for(y = 1; y < YSIZE-1; y++)
27         u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
```

図 5.2: Laplace の XMP 並列化プログラム

されている。これらのうち u は、図中の 9 行目で、u[j][*] が t[j] に align されており、これは、添え字変数の j で参照されている配列 u の最初の次元を t(j) と同じように分配する指示となる。uu も同様に 10 行目で uu[j][*] が t[j] に align されており、u と同様に最初の次元で分配される。この結果、2 つの配列 u と uu はグローバル配列となり、図 5.4 に示す配列 uu のように Nc 個のブロックに分割されて、Nc 個の node に分配される。

5.4 グローバル配列によるデータ分割

XMP は、XMP の指示文で修飾されないすべての変数を、生成される SPMD プログラムごとに別々に持つ。本稿ではこれを、実行プロセスが別々の値を持てるという意味で、ローカル変数と呼ぶ。グローバルビューを用いた場合、指示文で指定した配列は、並列処理を行う対象全体（グローバル）のデータを記述しているものとして扱う。その他の指定されない配列や変数はすべて、ローカルなものになる。

また、グローバルビューを用いるときの XMP の指示文は大きく 3 つのグループに分類できる。これらは、データマッピング、ワークマッピング（並列化）と通信および同期に関するものである。以下これらのうち、データ分割について、図 5.3 のグローバルビュー

を用いた擬似的な XMP プログラムを用いて簡単に説明する。

```
1 #pragma xmp nodes n(4)
2 #pragma xmp template t(0:99)
3 #pragma xmp distribute t(block) onto n
4
5 double g[100][80];
6
7 #pragma xmp align g[j][∗] with t(j)
8
9 double l[40][50];
```

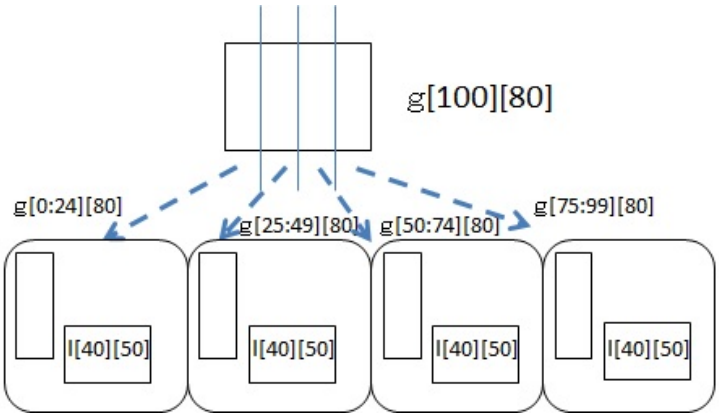

図 5.3: XMP による配列の分割例

XMP は、 XMP の指示文を持たないすべての変数をローカル変数として扱う。すなわち、実行する複数のワーカーはこれらの変数のコピーを別々に保有する。しかし、プログラム中の配列が align 指示文で既に分配されている template に関係付けられている場合には、その配列はグローバル配列となる。図 5.3 に、このグローバル配列の指示を行う簡単な例を示す。

まず、XMP の指示文は C の pragma を用いて記述し、#pragma xmp で、XMP の指示文であることを表記する。図中の 1 行目の XMP 指示文は nodes 指示文である。この指示文は、並列実行を行う node、つまり実行対象（実行環境によって、プロセスや計算ノードとなるもの）の配列を定義している。ここでは、n という名称の一次元に並べられた 4 つの nodes n(4) を定義している。

2 行目の template は、プログラム中の配列を包括的に代表することができる仮想配列を記述する。プログラム中のすべての配列の、包括的なインデックス空間として用いられる。そして、この仮想配列は、そのいくつかの次元に沿って行う、node 間での配列分配の指示に利用される。2 行目では、そのような template t を 0 から始まって 100 個の要素を持つ、t(0:99) として定義 している。そして、その次の行の distribute 指示文は、どのように仮想配列 t(0:99) の要素が 4 個の node n(4) に分配するかを記述する。この t(block) は、t(0:99)

を 4 つの同じ大きさのブロックに分割して、それぞれ 4 個のノードに分割することを指示している。

グローバル配列を定義するには、この template t(0:99) を利用する。このプログラム中には、2 つの 2 次元の配列 g[100][80] と l[40][50] が使用されている。このうちプログラム中 5 行目で、グローバルな配列として、ノード間に分配したい double 配列の g[100][80] 宣言している。

そして 7 行目の align 指示文で g の分配の指示を template t を用いて行っている。g[j][*] with t(j) と、j を用いて配列 g の 1 次元目を template t に合わせてのブロック分割することを指定している。これは、添え字変数の j で参照されている配列 g[100][80] の最初の次元を t(j) と同じように分配する指示となる。この結果、配列 g[100][80] はグローバルな配列となり、図 5.3 の下に示すように 4 個の 80×25 の大きさのブロックに分割されて、4 つの node に分配される。

一方、10 行目の double の配列 l[40][50] には指示文では何の指定もしていない。したがって、図 5.3 に示すように、この配列 l はローカルな配列となる。この結果、並列実行を行うすべての node は 50×40 の double の配列 l を独立して持つことになる。

5.5 Shadow 宣言と Reflect 操作

図 5.2 のプログラムには、別の XMP 指示文 shadow uu[1][0] が 11 行目に使用されている。これは、コンパイラに対して、配列を最初の次元で分割する際に、本来の配列 uu の分割部分に加えて、分割配列の最初の列の 1 つ前の列と、分割配列の最後の列の 1 つ次の列の、2 つの列を追加して分配することを指示している。図 5.4 に、この結果配列 uu がどのように分割されるかを示す。図では、10000/Nc の値を Nb とし、縦方向にアドレスが連続しているものとしている。

図 5.4 の灰色で示した列が、この指示文により、追加で挿入された列となる。例えば、node2 はグローバル配列 uu の分割矩形の uu[Nb:2Nb-1][:] が分配された担当計算部分で、これに加えて、uu[Nb-1][:] と uu[2Nb][:] の 2 つの列がこの node に保持される。これらの 2 つの列はデータの参照だけのために用いられ、袖領域とも呼ばれる。これらの領域の計算は他の node で行われるため、データが計算されるたびに、このデータを更新する必要がある。そこで一般には、分配されたグローバル配列の全データの計算が行われるたびに、全 node 間でこの袖領域のデータを交換して、データ更新を行う。

XMP では、この更新を行うタイミングを指示するために、図 5.2 の 21 行目に示すよう

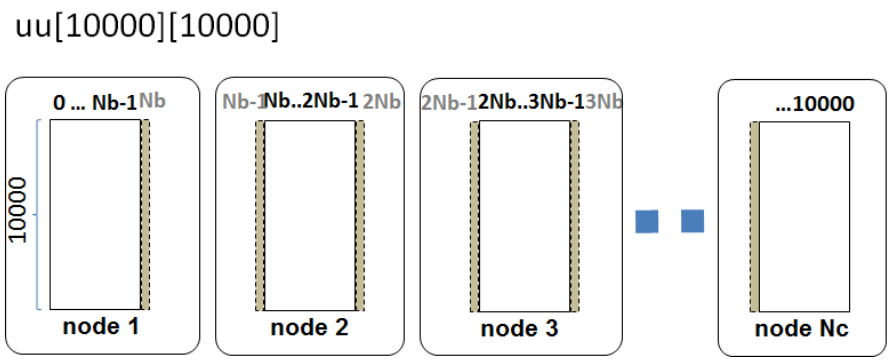

図 5.4: 配列の袖領域

に reflect 指示文をプログラム中に挿入することができる。なお、図 5.4 に示すように、最初の node（node 1）と最後の node（node Nc）はそれぞれ 1 つの袖領域しか持たない。これは、配列の最初の列の前や、最後の列の後にデータは無いからである。このような境界条件はコンパイラが自動的に処理する。

5.6 XMP の実行時システム

本研究で用いた PGAS 言語の XMP の実行時システム [20] の構成を図 5.5 に示す。まず、XMP のプログラムは XMP の実行時システムの API を利用して、クラスタ間にまたがるグローバルなメモリ空間上で並列にプログラムを実行する。XMP は UPC が用いているローカルビューの他に、グローバルビューを用いてより簡便に並列処理を記述することができる。このため、ローカルビュー用の API の他にグローバルビュー用の API をサポートしている。これら XMP の実行時システムは、ノード間にまたがるデータの交換のための通信に MPI を用いている。また、XMP プログラムを実行するクラスタの通信インターフェースが片サイド通信や RDMA 等の高度な機能を持っている場合は、これを直接利用することができる他、ローカルビューの一部では UPC の GASNet も利用できる。

XMP の実行時システムの UPC の実行時システムに対する大きな違いは次の 2 点である。

(1) グローバルビューを実現するモジュール

前述したように、XMP は、UPC で用いているローカルビューによる並列化のプログラム手法の他に、グローバルアドレス空間に生成したグローバルなデータに対して並列化の指示をする、グローバルビューを持っている。実行時システムにもこのグローバルビュー用の API を処理するモジュールを持っている。

(2) ノード間通信の通信に MPI を利用

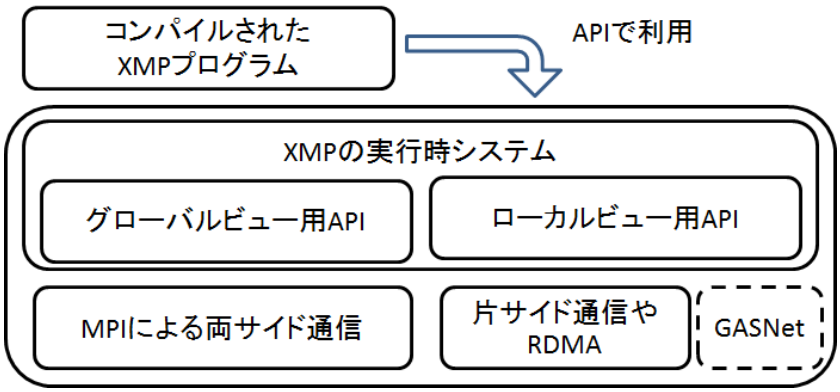

図 5.5: XMP プログラムの実行方法

UPC の実行時システムは、通信用のモジュールとして、GASNet を使用している。GASNet はアクティブメッセージを用いた比較的新しい通信 API で、条件によっては性能上のメリットもある [7]。一方、XMP が使用している MPI はクラスタシステム用の標準の通信 API として、広く使われており、汎用性が高く、新たな通信 API を移植する手間を削減できる。

5.7 XMP プログラムの実行方法

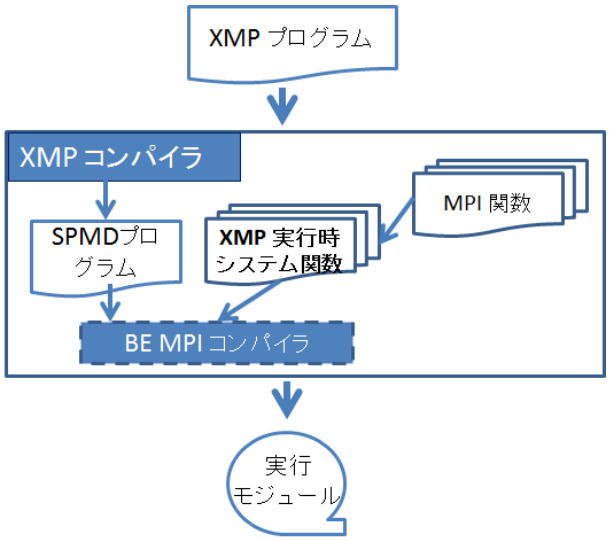

図 5.6: XMP 実行時システム

図 5.6 に XMP で書かれたプログラムのコンパイル方法を示す。まず、XMP プログラムは、XMP コンパイラによって SPMD プログラムに変換される。この時、XMP で準備された並列化やデータ分割の機能により、袖領域を利用した通信の最適化を行うことができる。さらに、多重ループを用いて多次元配列の処理を行う際には、その演算ループのタイリン

グを行うこともできる。これらはすべて、SPMD プログラムから、XMP 実行時システムの関数を呼び出すことで、実現されている。

XMP の実行時システム関数は、XMP の様々な機能を実現する関数群で、プログラムを実行するアーキテクチャ向けに最適化されて準備されている。今回はこのうち、x86 汎用クラスタ向けのソースコードを利用してその改良を行った。この実行時システム関数のうち、プロセス間での通信や同期を必要とするものは、MPI 関数を用いてこれを実装している。したがって、生成された SPMD プログラムと XMP 実行時システムの関数を Xeon Phi 用のバックエンド (BE) MPI コンパイラでコンパイルすれば、実行モジュールを生成することができる。

本研究では、Omni XMP compiler 1.1 (build1322)[30] をベースのコンパイラとして使用した。このコンパイラは、XMP のソースプログラムを XMP の実行 API を呼びだす C プログラムに変換することで、実行モジュールを生成できる。また、この XMP の実行時システムは C と MPI を用いて書かれている。したがって、Xeon Phi のバイナリは、XMP プログラムから生成された C コードと XMP 実行時システムライブラリを Intel C コンパイラと Intel MPI でコンパイルすることで得ることができる。このコンパイラには、それぞれ、Intel complier version 13.1.3.192 (build 2013607) と Intel MPI library 4.1.1.036 を使用した。

第6章　メニーコアプロセッサの特性評価

XMP の実行時システムをメニーコアプロセッサに最適化する方法を検討するために、その対象となるメニーコアプロセッサの特性を調査した。これには、2 つのステンシルコード、Laplace と姫野ベンチマークを OpenMP と MPI を用いて並列化して、それぞれ Xeon Phi 上で実行し、その性能とスケーラビリティを比較検討した [38][40]。また、性能の絶対値の確認のため、3.5 節で仕様を比較した同世代の汎用プロセッサ Xeon E5-2600v2 とも性能を比較した。本章ではこれらについて述べる。

6.1　実験環境

まず、実験を行った計算機と実験に用いたソフトウェアを説明する。3 章で述べたように、Xeon Phi は PCI Express のカードであり、単独では動作できない。この特性評価は、表 6.1 に示すように、の Xeon E5-2600v2 とベースのサーバに Xeon Phi E-7100 を 1 枚挿入して行った。

表 6.1: 実験環境

項目		内容
ホスト	CPU	Xeon 8 コア 2.6 GHz Ivy Bridge×2
	OS	Red Hat EL 6.4
	メモリ	64 GB
メニーコア	CPU	Xeon Phi 61 コア 1.238 GHz
	OS	MPSS 2.1.6720
	メモリ	16GB
PGAS 言語		Omni XMP compiler 1.1 (build1322)
コンパイラ		Intel complier version 13.1.3.192 (build 2013607)
		Intel MPI library 4.1.1.036

まず、ホスト側の計算機には、2.6 GHz の Ivy Bridge の Xeon プロセッサを 2 個実装したサーバを用いた。サーバの OS は Red Hat 社の Enterprise Linux 6.4 を用い、メモリは 64 GB 実装したものを用いた。このサーバには、PCI Express v3.0x16 のインターフェースを 2 個持っているが、この 1 つに Xeon Phi E-7100 を実装しもう一つには Mellanox Infiniband HCA を実装している。ただし、E-7100 は　PCI Ex v2.0x16 のデバイスとして動作する。

Xeon Phi のプロセッサは 1.238 GHz のものを使用した。Xeon Phi の OS 等を含む Manycore Platform Software Stack (MPSS) は 2.1.6720 を使用し、Xeon Phi 上では 16GB のメモリを実装している。前述したように、XMP のコンパイラは Omni XMP compiler 1.1 (build1322) をベースにしており、このコンパイルには、Intel complier version 13.1.3.192 (build 2013607) と Intel MPI library 4.1.1.036 を使用したが、OpenMP や MPI による並列化プログラムのコンパイルも同じ Intel 社のコンパイラと MPI を使用した。

6.2　メニーコア単一ノード上での評価

XMP の Xeon Phi 向けの実行時システムに必要な特性を調べるため、Xeon Phi の性能測定を 2 ソケットの Xeon プロセッサと比較して行った。測定には、対象としているステンシル計算の一つである姫野ベンチマークの L（512×256×256）サイズを用いた。この姫野ベンチマークは、3 次元格子空間のポアソン方程式を解くための 19 点ステンシルコードで、XMP 版のベンチマークプログラムは、オリジナルの Fortran90+OpenMP 版をベースに、1 次元分割の XMP の指示文を入れたものを使用した。図 6.1 に、この 1 次元分割のプログラムの内、データの分割と並列化の一部を示す。

まず、図 6.1 の 8 行目で 1 次元で任意（プログラム実行時に数を指定）の長さの node n(*) を宣言しており、10 行目で、9 行目で定義した 3 次元の template t(mimax,mjmax,mkmax) の最も外側の次元をブロック分割している。そして、11 行目と 12 行目で、p,bnd,wrk1,wrk2 といった 3 次元の配列と a,b,c の 4 次元の配列をこの template に align することで、グローバルな配列として分割配置する指示を行っている。続く 13 行目から 19 行目でそれぞれの配列を分割する際に、袖領域を分割した境界に 1 つずつ持たせている。21 行目の reflect で袖領域を更新し、22 行目の loop で、23 行目から始まる K のループを並列実行する指示を行っている。なお、この際に reduction で指定した変数 GOSA は全 node で加算のリダクション演算を行う。

測定は、Xeon Phi 上では XMP 版と MPI 版の両者を実行し、一方 Xeon 上では MPI 版のみを実行し、これら 3 つの結果を比較した。XMP のコンパイラは、XMP 版のベンチマーク

```
1
2           module alloc1
3           PARAMETER (mimax = 512, mjmax = 256, mkmax = 256)
4           real p(mimax,mjmax,mkmax), a(mimax,mjmax,mkmax,4), b(mimax,mjmax,mkmax,3)
5           real c(mimax,mjmax,mkmax,3), bnd(mimax,mjmax,mkmax)
6           real wrk1(mimax,mjmax,mkmax), wrk2(mimax,mjmax,mkmax)
7
8   !$xmp nodes n(*)
9   !$xmp template t(mimax,mjmax,mkmax)
10  !$xmp distribute t(*,*,block) onto n
11  !$xmp align (*,*,k) with t(*,*,k) :: p, bnd, wrk1, wrk2
12  !$xmp align (*,*,k,*) with t(*,*,k) :: a, b, c
13  !$xmp shadow p(0,0,1)
14  !$xmp shadow a(0,0,1,0)
15  !$xmp shadow b(0,0,1,0)
16  !$xmp shadow c(0,0,1,0)
17  !$xmp shadow bnd(0,0,1)
18  !$xmp shadow wrk1(0,0,1)
19  !$xmp shadow wrk2(0,0,1)
20  ...
21  !$xmp reflect (p)
22  !$xmp loop (K) on t(*,*,K) reduction (+:GOSA)
23              DO K = 2, kmax-1
24                  DO J = 2, jmax-1
25                      DO I = 2, imax-1
26                          S0 = a(I,J,K,1)*p(I+1,J,K)+a(I,J,K,2)*p(I,J+1,K) &
27                                  +a(I,J,K,3)*p(I,J,K+1) &
28                                  +b(I,J,K,1)*(p(I+1,J+1,K)-p(I+1,J-1,K) &
29                                  -p(I-1,J+1,K)+p(I-1,J-1,K)) &
30  ...
```

図 6.1: Fortran 版姫野ベンチマークの XMP1 次元分割のプログラム

プログラムを、MPI を用いたプログラムに変換する。XMP 版のプログラムは、この変換された MPI プログラムを、Xeon Phi 用にコンパイルして実行した。MPI[15] と OpenMP のコンパイラはいずれもインテルコンパイラ Version 13.1.1.163 (Build 20130313)[14] を用いている。MPI 版の測定には Fortran90 の MPI 版姫野ベンチマークを、インテル MPI を用いて Xeon Phi 用と Xeon 用にそれぞれ、mpiifort で -O3 -xAVX（Xeon の場合）と-mmic （Xeon Phi の場合）オプションでコンパイルし、それぞれのシステム上で実行した。結果を図 6.2 に示す。

図 6.2 の Phi-MPI と Xeon-MPI は、それぞれ MPI 版プログラムを Xeon Phi 上と Xeon 上で実行した場合の演算性能を、GFLOPS で示している。残りの Phi-XMP が XMP 版のプログラムを実行した場合の性能である。図は、それぞれのシステムで MPI プロセス数を 1 から 2 のべき乗ごとに実行した結果を比較して示している。Xeon では最大コア数の 4 倍である 32 まで測定し、Xeon Phi ではコア数の 4 倍を少しオーバするが、256 まで測定を行った。この結果、次の 3 つのことが観測できた。

(1) 並列数は 8 倍程度必要なものの、単一 Xeon Phi カードの姫野ベンチマークの性能は

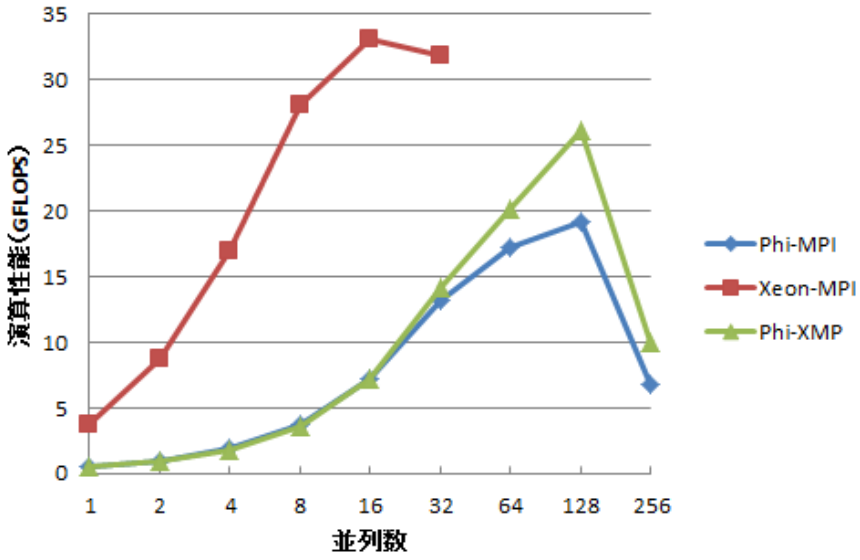

図 6.2: 単一 Xeon Phi カードでの性能比較

2 ソケットの Xeon サーバと同等かやや劣る程度である。

(2) Xeon Phi の 1 コア当たり、2 プロセスを実行した場合に最大性能が得られる。

(3) XMP で並列化したプログラムにおいても MPI 版で並列化したものと同等の性能を得ることができる。

なお、XMP 版と MPI 版の Xeon Phi 上での性能が近いのは、XMP 版も MPI のプログラムに変換されるからである。

6.3 マルチノード・システム上での評価

単一ノードでの測定結果をふまえて、InfiniBand で接続された 16 ノードのクラスタ上で、同様の測定を行った。各 Xeon ノードには、それぞれ 1 枚の Xeon Phi のカードが搭載されている。Xeon Phi で実行する場合は、Xeon は用いずに Xeon Phi のみで演算を行う。測定には姫野ベンチマークの XL サイズを用い、1 次元分割では必要な並列度が得られないため、2 次元分割を用いる。図 6.3 に、この姫野ベンチマークプログラムの 2 次元分割の主要部分を示す。

このプログラムでは、まず 2 行目で node を n(4,*) と宣言して、4 行目で template t の 2 次元目を 4 つブロックに分割して、さらに 3 次元目を任意の数に分割している。5 行目と 6 行目で、この t に演算用を行うグローバル配列（p,bnd,wk1 等）を align することにより、2 次

```
1 ...
2 !$xmp nodes n(4,*)
3 !$xmp template t(mimax,mjmax,mkmax)
4 !$xmp distribute t(*,block,block) onto n
5 !$xmp align (*,j,k) with t(*,j,k) :: p, bnd, wrk1, wrk2
6 !$xmp align (*,j,k,*) with t(*,j,k) :: a, b, c
7 !$xmp shadow p(0,1,1)
8 !$xmp shadow a(0,1,1,0)
9 ...
```

図 6.3: Fortran 版姫野ベンチマークの XMP2 次元分割の主要部分

元のデータ分割を行った。7 行目から始まる。shadow では、分割する 2 つの次元に袖領域を持たせている。

Xeon Phi で実行する際は 1 ノード当たり 128 プロセスに固定し、16 ノードで最高 2048 プロセスまでの測定を行った。Xeon で実行する際は 1 ノード当たり 16 プロセスとし、16 ノードで 256 プロセスまで測定した。結果を図 6.4 に示す。図の Phi-MPI は、MPI 版プログラムを評価用システムのクラスタ上で Xeon Phi だけを用いて実行した場合の演算性能を、GFLOPS で示している。Xeon-MPI は同じクラスタを用いて Xeon のみで実行したもので、Phi-XMP が、XMP 版のプログラムを Xeon Phi のみで実行した場合の性能である。

単一カードの場合と同様に Fortran90 の MPI 版を Xeon 用、Xeon Phi 用にそれぞれインテルコンパイラ mpiifort で-O3 -xAVX または-O3 -mmic オプションでコンパイルしたものと比較している。Xeon サーバは十分なメモリ容量を持っているので、MPI プロセスを 16 個から、256 = 16 × 16 個まで測定した。Xeon Phi では、2 枚（256 = 128 × 2）では、XL モデルを実行するのに必要なメモリに満たないので、512、1024、2048 の 3 点の測定を行う。

なお、姫野ベンチマークの Fortran90 の MPI 版は配列を 3 次元分割したものである。XMP 版と同じ条件では最高性能が得られなかったので、表 6.2 に示すような 3 次元分割を用いて測定を行った。

表 6.2: 3 次元 MPI 版の分割方法

MPI プロセス数	分割方法
512	8 × 8 × 8
1024	8 × 8 × 16
2048	8 × 16 × 16

この結果、次のことが観測できた。

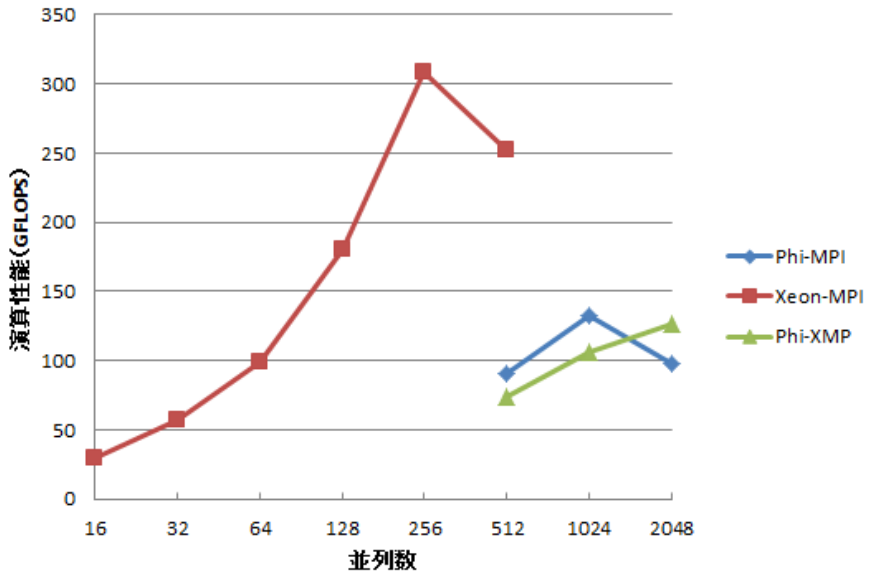

図 6.4: クラスタ上での性能比較

(1) 4 個の Xeon Phi ノード（512 プロセス）での実行と 2 ソケット Xeon の 4 ノード（64 プロセス）の性能は近い値であるが、ノード数が増える程、Xeon の方の性能が良くなる。

(2) Xeon Phi 用に MPI で並列化したプログラムでは、1024 プロセスまでは性能向上するが、2048 では性能が落ちてしまう。

(3) XMP で並列化したプログラムでは、MPI 版と比較してやや性能が劣るもの、2048 プロセスまで性能向上得ることがきた。

6.4 XMP を用いたハイブリッドの評価

MPI プログラムでのノード間をまたいだ通信に起因すると思われる性能低下を軽減するために、XMP 版のプログラムをハイブリッド化することを試みた。図 6.5 にこのプログラムの主要部分を示す。このプログラムでは、8 行目の xmp loop を用いてプログラムの 3 重ループの最も外側の K のループを 1 次元分割して、並列化している。そして、その内側の 9 行目で OpenMP による並列化の指示を行っている。COLLAPSE(2) の指示により、ノード内では、K とその内側の J のループが COLLAPSE され、並列化される。

このハイブリッドプログラムは、XMP による並列化を Xeon Phi のノード数と同じ並列数にして、ノードごとの OpenMP のスレッド数を 128 にして実行した。結果を図 6.6 に示す。

```
1  …
2  !$xmp nodes n(∗)
3  !$xmp template t(mimax,mjmax,mkmax)
4  !$xmp distribute t(∗,∗,block) onto n
5  !$xmp align (∗,∗,k) with t(∗,∗,k) :: p, bnd, wrk1, wrk2
6  !$xmp align (∗,∗,k,∗) with t(∗,∗,k) :: a, b, c
7  ...
8  !$xmp loop (K) on t(∗,∗,K) reduction (+:gosa)
9  !$OMP PARALLEL DO COLLAPSE(2) REDUCTION(+:gosa)
10             DO K = 2, kmax−1
11                 DO J = 2, jmax−1
12                     DO I = 2, imax−1
13                         S0 = a(I,J,K,1)∗p(I+1,J,K)+a(I,J,K,2)∗p(I,J+1,K) &
14                                 +a(I,J,K,3)∗p(I,J,K+1) &
15                                 +b(I,J,K,1)∗(p(I+1,J+1,K)−p(I+1,J−1,K) &
16                                 −p(I−1,J+1,K)+p(I−1,J−1,K)) &
17 ...
```

図 6.5: Fortran 版姫野ベンチマークの XMP ハイブリッド版

図 6.6 の Phi-MPI, Xeon-MPI と Phi-XMP は、それぞれ図 6.4 に示したものと同じデータで、MPI のプログラムを Xeon Phi, Xeon のクラスタで実行したときの性能と、XMP のプログラムを Xeon Phi 上で実行したときの性能である。図 6.6 で追加された Phi-HYBRD がハイブリッド版の性能を示している。並列数は、MPI 版でのプロセス数を示しており、ハイブリッドの場合は 1 ノードのスレッド数 128 とノード数を乗じた値を用いている。

この XMP と OpenMP のハイブリッド版を 4 ノードと 8 ノード上での実行した場合は、MPI 版と同等の性能が得られた。また、MPI 版とは異なり、16 ノードの場合でも性能が向上しており、このことから、ハイブリッド版では、Xeon Phi から InfiniBand のネットワークを介して送受信されるパケット数が減った結果、16 ノードでも性能向上が得られた考えられる。

6.5 OpenMP と MPI の単一ノード上での再評価

姫野ベンチマークでの測定の結果、ノード内を OpenMP で記述し、マルチノード間は XMP から生成した MPI 通信を行うハイブリッド方式で最も良い性能が得られることがわかった。この場合、1 ノード当たり計算プロセスは 1 個となり、測定の条件では 128 個の OpenMP スレッドを用いた。一方、XMP の実装では 128 個の MPI プロセスを用いて同じ計算を行っており、両者の性能は 8 ノードまではあまり相違はない。そこで、1 ノード内での OpenMP と MPI での実行性能を比較するために、別のステンシル計算、Laplace を用いて単一ノード上で両者の性能を比較した。図 5.2 に示したように、Laplace は 2 次元の Laplace 方程式を 5 点差分して、ヤコビ法で解く単純なステンシル計算である。2 次元の空間配列

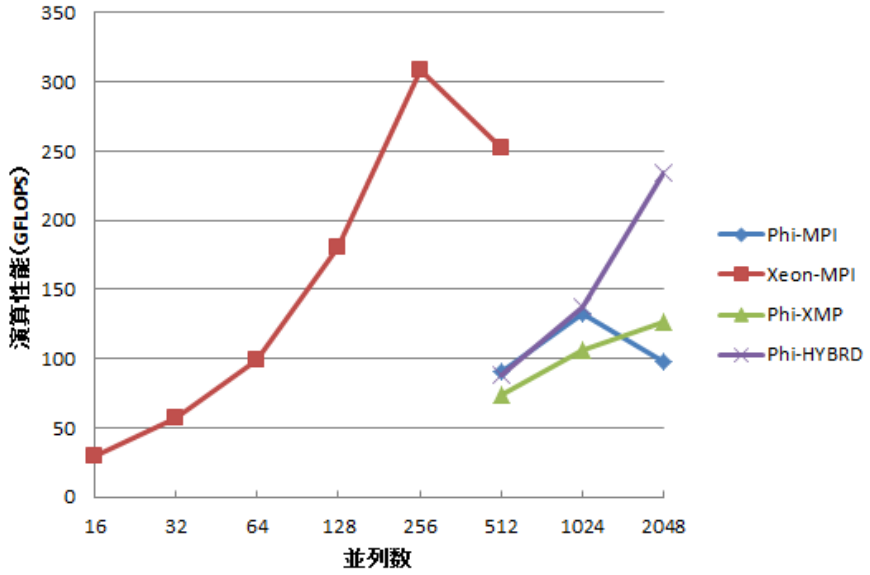

図 6.6: クラスタ上での XMP 版ハイブリッド性能

として XSIZE×YSIZE (10000 × 10000) を用い、これを 1 次元分割して並列化した。

結果を図 6.7 に示す。性能の比較は図 5.2 の XMP プログラムをコンパイラで MPI のプログラム変換した実行モジュールでの結果 Lap_XMP_MPI と、図 2.12 の OpenMP で並列化した結果 Lap_OMP とで行った。図 6.2 の横軸は実行した並列数で縦軸は演算性能を GFLOPS で示している。MPI[15] と OpenMP のコンパイラはいずれもインテルコンパイラ Version 13.1.1.163 (Build 20130313)[14] を用いている。Xeon Phi 用のバックエンドのコンパイルは、いずれの実行ファイルも-O3 -fno-alias -mmic のオプションを利用して生成している。それぞれ、MPI のプロセス数（OpenMP の場合はスレッド数）をまず、1 から 2 のべき乗ごとに 16 まで測定した。Xeon Phi ではコア数が 61 のため、16 からは 30 の倍数で 30, 60, 120, 240 と Xeon Phi の持つ最大のスレッド数 244 付近まで測定を行った。

この結果、240 を除く、ほぼすべての並列数において、XMP の生成する MPI 版の方が、OpenMP 版より優れた性能を示していることがわかった。すなわち、ノード内の並列化にも XMP を用いて並列化する方法が、OpenMP で単純にスレッド並列にするよりも、優れていることがわかった。また、このベンチマークの場合、最高性能は、並列数が 60 の時に得られた。

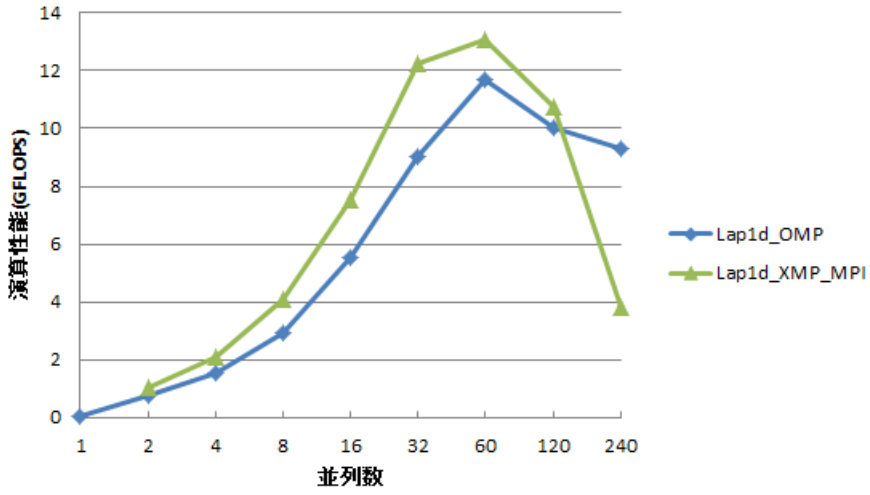

図 6.7: 単一ノードでの XMP と OpenMP の性能比較

6.6 特性評価のまとめ

2 つのステンシル計算、Laplace と姫野ベンチマークでを用いて単体及び、16 ノードまでの Xeon Phi のクラスタを用いてその特性評価を行った結果、次の 4 点が分かった。

(1) Xeon E5-2600V2 の 2 ソケットのサーバと同程度かやや劣る程度の性能を得ることができる。この際、8 倍程度の並列度（プロセス数またはスレッド数）が必要となる。

(2) 指示文により、XMP で並列化したプログラムは、コア間やノード間を MPI で通信するが、プログラマがデータ分割して作成したより複雑な MPI 版と同等の性能を得ることができる。

(3) 16 ノードのクラスタシステム上での実行のように、XMP の node 数が 2000 を超えるような場合は、node 単位での MPI プロセスの実行では通信がボトルネックになり性能が得られない。したがって、ハイブリッドのような方法が有効である。

(4) Laplace をノード内で実行する場合、OpenMP よりも MPI の性能が優れている。したがって、単純にノード内を OpenMP でノード間を MPI で行うハイブリッドプログラミングが良いとは限らない。

これらの結果から、ステンシル並列計算の Xeon Phi クラスタ上での実行に、XMP を用いてより容易にプログラムを並列化しても、MPI でユーザがデータ分割や通信を明示的に行ったプログラムと同等の性能が得られる可能性があることが分かった。また、XMP の node をそのまま MPI プロセスにするやり方ではクラスタシステムでのスケーラビリティに

問題があるが、逆に Xeon Phi のノード単位で OpenMP による並列化を行っても、同等の条件での MPI よりも性能が得られないことがあり、単純なハイブリッドでも最大性能が得られるとは限らないことが分かった。

第7章　XMPのメニーコア向け実行時システムの最適化

XMP のレファレンス実装である Omni XMP compiler[30] は、ソースプログラムを XMP の実行時システム関数を呼びだす C プログラムに変換し、既実装されているプログラム開発システムのコンパイラでコンパイルすることで、実行モジュールを生成する。この XMP 実行時システムは C と MPI を用いて書かれており、我々はこの実行時システム関数のうち、次に述べるグローバル配列の配置と袖領域の更新を行う部分を、メニーコア向けに改良した [12][39]。これには特にプロセッサ内での処理に関して、6 章での OpenMP での実行に見られたような、演算時のコア間の干渉を少なくし、また、コア間でのデータ交換が効率的に行えるように設計した。

7.1　最適化の方法

Xeon Phi で性能を得るには、ノード内でもコア数かまたはその 2 倍程度の並列数が必要である。また、ノード内の並列化に OpenMP を使用しないとすると、ノード内外に係わらず、プログラムの並列化には XMP の node を用いるのが最も自然な実装となる。前章のメニーコアプロセッサの特性評価で用いた XMP のプログラムは、この方法を用いて、node に割り当てられたプロセスが行う計算処理と、これらのプロセス間で行う MPI 通信として実行された。この際 XMP のコンパイラは、XMP を用いて記述されたプログラムを必要な並列数で計算を実行するために、SPMD の MPI ソースプログラムに変換した。これを Xeon Phi 用の MPI のコンパイラでコンパイルして XMP の実行時システム関数のライブラリとリンクすることで実行モジュールを生成した。

このような、既存の XMP の処理系をそのまま利用して、XMP を Xeon Phi に最適化するために、次のような方法をとる。

(1) 並列化の最小単位である XMP の node は、プロセスで実装し、複数の Xeon Phi ノードを用いたクラスタシステムの場合は、それぞれの 60 か、あるいはその数倍の N 個の node もち、全体では、(N × ノード数) 個の均一な node の集合として取り扱う。

(2) Xeon Phi ノード間のデータ交換には、XMP の実行時システム関数を用いる。ただし、従来の実行時システム関数は、すべての node 間の通信を MPI 通信で行っていたが、これを通信する node の位置、例えば同じ Xeon Phi ノード内なのか違うのかによって、最適な通信方法を選ぶように改良する。

この方針により、従来通りの比較的簡単な PGAS 言語 XMP の統合的なセマンティックスと、その処理系の大半をそのまま活かしながら、メニーコアのクラスタといった、大規模な科学技術計算のためのアーキテクチャに対応できると考えた。

7.2 XMP 実行時システム関数と reflect 操作

XMP の実行時システム関数は、XMP の様々な機能を実現する関数群で、プログラムを実行する様々なアーキテクチャ向けに最適化されて準備されている。XMP のコンパイラで変換されたプログラムは、XMP で準備された並列化、データ分割、袖領域の通信といった様々な処理を行うことができる。これらはすべて、変換後のプログラムから、XMP 実行時システムの関数を呼び出すことで実現されている。本研究では、このうち x86 汎用クラスタ向け実行時システム関数のソースコードを利用して、その改良を行った。この実行時システム関数のうち、プロセス間での通信や同期を必要とするものは、MPI 関数を用いてこれを実装している。

XMP 実行時システムは、プログラム中の reflect 指示文の挿入された位置で、node 間の袖領域の更新を行う。この袖領域の更新は、グローバル配列に対して、その配列の分割した次元ごとに、下に示す 3 つの処理を行うことにより実現している。分割した配列の次元の添え字が小さくなる方向を上位、逆に大きくなる方向を下位と呼ぶことにする。それぞれのプロセスは、自分が担当して演算する最初の列を、その次元の自分の上位のプロセスに送る必要があり、また自分が担当して演算する最後の列を、その次元の自分の下位のプロセスに送る必要がある。そこで、各プロセスはその次元の上位方向にデータを送るためと、下位方向にデータを送るための 2 つの袖通信用のバッファとして、上位バッファと下位バッファを持っている。

そして、図 7.1 に示す 3 つの実行時システム関数を呼び出して、この袖領域の交換を実行する。

(1) Pack: 各プロセスがグローバル配列の自らの担当領域の上位方向／下位方向の端のデータを上位方向用／下位方向用のバッファにコピーする。

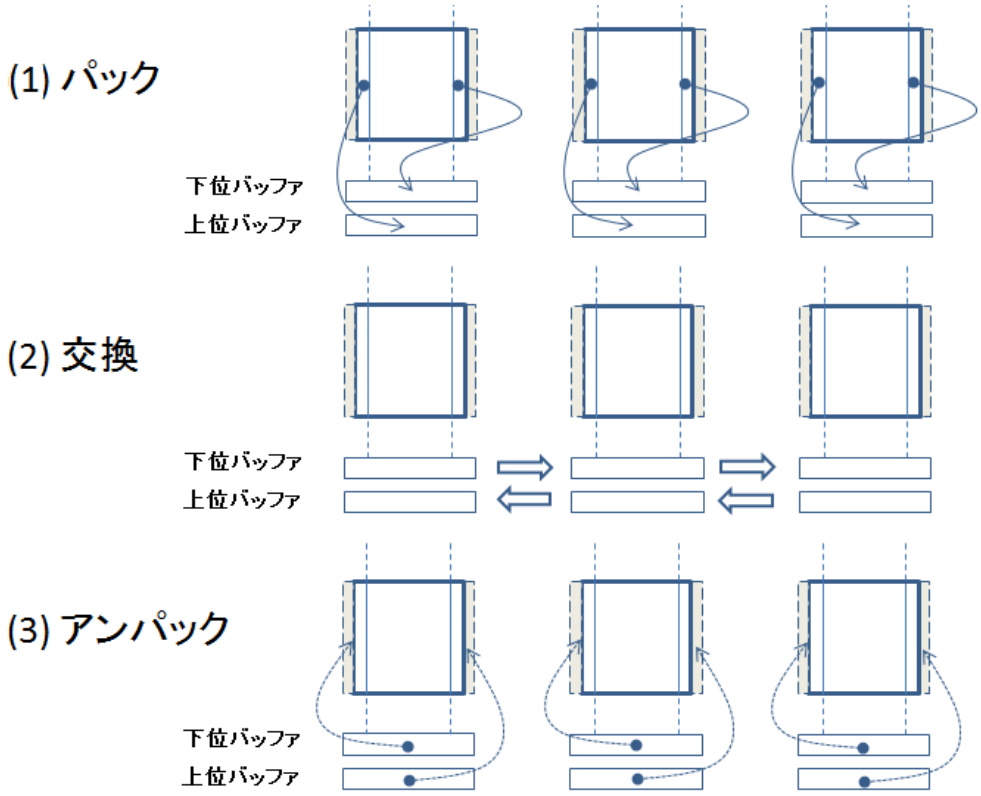

図 7.1: 配列袖領域の更新

(2) Exchange: 全プロセスが一斉に、自プロセスの上位方向用／下位方向用バッファの内容を、上位方向／下位方向に隣接するプロセスの上位方向用／下位方向用バッファに（MPI のデータ送受信 API を利用して）転送する。

(3) Unpack: 各プロセスが上位方向用／下位方向用のバッファのデータをグローバル配列の自らの担当領域の下位方向／上位方向の袖領域にコピーする。

したがって、これらの 3 つの XMP 実行時システム関数が、本研究で改良を行う主な対象となる。

XMP の実行時システム関数中で行われる Xeon Phi ノード内の MPI 通信を、node プロセス間の共有メモリ転送に置き換えるために、node に分割したグローバル配列の配置方法を変更した。従来の実行時システムでは、分割されたグローバル配列は、その他のローカル変数や配列とともに、そのプロセスのアドレス空間に配置している。我々は、このグローバル配列の配置に共有メモリを使用する。分割され、袖領域を付加した配列は、上位方向用／下位方向用バッファと、node ごとにまとめられ、ノード上の node 数の分、共有メモリ上へ確保する。この結果、すべての node から他の node の配列データや通信バッファを参照できる。

7.3 共有メモリ上でのデータの配置

本研究で用いた実装とレファレンス実装ではグローバル配列の配置方法が大きく異なるので、まずこれについて述べる。XMP の実行時システムは、プログラム中で宣言した node に対してプロセスを割り当てるが、従来の実行時システムでは、分割されたグローバル配列は、その他のローカル変数や配列とともに、そのプロセスのアドレス空間に配置している。我々は、このグローバル配列の配置に Linux の mmap() で生成した共有メモリを使用する。図 7.2 に 2 次元のグローバル配列に 1 列ずつの袖領域を付けて 1 次元分割を行う例を示す。

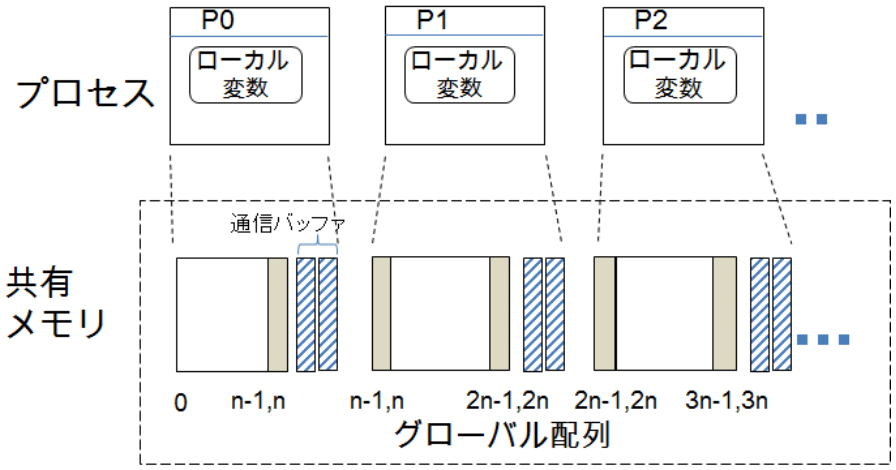

図 7.2: XMP 配列の配置

図 7.2 中では、全プロセスの内 3 つのプロセス、P0、 P1 と P2 のアドレス空間を示している。ローカルの配列を含むローカル変数は、それぞれのプロセスのローカルなアドレス空間に配置する。一方、グローバル配列は、分割してから（図中で灰色の）袖領域を加えた上で、共有メモリ空間上に配置する。図の例では、プロセス P0 に分配するグローバル配列は共有メモリ上に、0 列目から n-1 列目までの n 列と、これに加えて袖領域の n 列目を配置する。

また、図 7.2 の共有メモリの、それぞれのプロセスが分割配列を置く領域の間には、袖領域のアップデートを行うための通信バッファ（図中の斜線の矩形）を置いた。これらの通信バッファの大きさはそれぞれの配列に必要な最大の袖領域サイズとした。この通信バッファは後述するように、下位バッファと上位バッファがあり、通信方向が異なるため、別々に 2 列分を確保した。

7.4　共有メモリによる reflect の改良

7.2 節で従来の XMP の実行時システムでの reflect の処理を示したが、Xeon Phi ノード内の共有メモリを用いた実装では、直接他のプロセスのメモリを参照できるので、3 つの処理のうち MPI 通信による (2)Exchange を行う必要はない。これを省くため、(3)Unpack の処理を行う転送元の上位、下位のバッファのアドレスを固定から、共有メモリ上の他のプロセスのバッファを参照できるように、可変に変更した。そして交換では、アンパックで用いる下位バッファのアドレスを自分の下位のプロセスの下位バッファへ、上位バッファは逆に上位プロセスのそれに代入する。その結果、MPI 通信を行う (2) Exchange を省き、(1) Pack と (3) Unpack のみで袖領域が交換できる。しかし、Xeon Phi ノード内で、担当する配列の次元の下限や上限に位置する node は、上位方向／下位方向に隣接する node が、メモリを共有している同一ノード内に無い。そこで、そのような場合にのみ袖領域の交換に、MPI を使用するように変更した。

XMP の実行時システムで行う、袖領域更新の典型的な例として、袖領域を持つ 1 つの次元の更新例について示す。XMP の実行時システムでは、この袖領域データの更新は。パック（pack)、交換（exchange)、アンパック（unpack）の 3 つの関数を呼び出すことで行っている。共有メモリを用いた実装では、このうち交換とアンパックを変更した。まず、アンパックの処理を行う転送元の上位、下位のバッファのアドレスを固定から、共有メモリ上の他のプロセスのバッファを参照できるように、可変に変更した。そして交換では、隣接するプロセスが同じノード内にある場合は、アンパックで用いる下位バッファのアドレスを自分の下位のプロセスの下位バッファへ、上位バッファは逆に上位プロセスのそれに代入する。その結果、図 7.3 に示すようにこの実装では、MPI 通信を行う (2) 交換を省き、(1) パック（実線）と (3) アンパック（点線）のみで袖領域が交換できる。

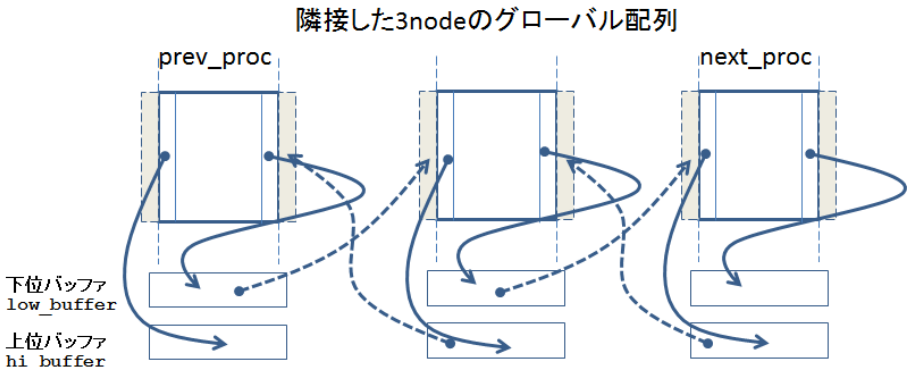

図 7.3: 共有メモリでの更新

図では、隣接した 3 つの node のグローバル配列を示しており、真ん中の配列の袖領域の

アップデートを行う例を示している。(1) パックの処理には変更は無く、図では実線で示すように、すべての node がプロセスの最上位と最下位の列を通信バッファにコピーしている。そして、(3) アンパックは、点線で示すように隣接するプロセスの通信バッファを直接参照することにより行っている。この結果、(2) 交換に伴うデータの移動を無くすことができる。

実際には、袖領域は多次元に存在するので、この場合について処理を考える。XMP を用いて、ある次元 m の配列 uu[]..[] を、この m 個の次元のうち、n 個の次元で分割した場合の reflect の処理は、前述した 3 つの XMP 実行時システム関数、パック（pack）、交換（exchange）、アンパック（unpack）を用いて、図 7.4 のような疑似コードで示すことができる。

```
1 for( d=1; d<=m; d=d の次の分割されている次元){
2   if( 配列uu の d 次元方向に袖領域が
3                                定義されている場合 ){
4       pack(uu,d); exchange(uu,d);unpack(uu,d);
5   }
6 }
```

図 7.4: XMP による多次元袖交換のアルゴリズム

疑似コード中の 3 つの関数がそれぞれ 2 つずつ持つ引数は、いずれも最初の引数 uu が処理の対象となるグローバル配列を示しており、次の引数はその配列の袖領域がある次元を示している。つまり、n 個の次元方向に分割した対象配列 uu[]..[] の分割する次元ごとに、袖領域が定義されているかどうかを確認し、定義されている場合はその次元に対して 3 つの処理を行う。

次に、3 つの実行時システム関数の本提案での実装について、それぞれ従来システムと比較して説明する。大きな変更を行ったのは、exchange のみである。まず、pack の処理関しては何の変更も無い。指定された配列の、処理の対象次元 d の自分の担当する最初の領域を、通信用のバッファhi_buffer（上位バッファ）にコピーし、同じく最後の領域を low_buffer（下位バッファ） にコピーする。

袖領域の交換は、従来の実装では、指定された配列の処理の対象となる次元について、自分の隣接上位の領域を持つ MPI プロセス (prev_proc) と、自分の隣接下位の領域を持つ MPI プロセス (next_proc) を用いて図 7.5 のような 4 つの処理を行っていた。

本稿の提案する交換の実装では、隣接する node を持つプロセスが、メモリを共有している同一ノード内にあるかどうかで、袖領域の交換に共有メモリを使うか、MPI を使用するかを切り替える。これを図 7.6 の疑似コードに示す. 図 7.6 では、従来実装で行っていた 4 つの処理を、その番号を残して示している。新たに、袖領域を交換するプロセスが自分と

```
1 (1) hi_buffer の内容を、prev_proc にMPI で送信
2 (2) next_proc から MPI で受信した内容を、
3                                   hi_buffer に格納
4 (3) low_buffer の内容を next_proc に MPI で送信
5 (4) prev_proc から MPI で受信した内容を、
6                                   low_buffer に格納
```

図 7.5: 従来の交換処理

同一の Xeon Phi カード内にある場合の処理 (2-i) 及び (4-i) と、このプロセスが別のカード内の場合でも、次にアンパックで行うバッファの参照に、ポインタを用いるためのポインタの代入処理 (2-p) 及び (4-p) を追加した。また、カード間をまたいで MPI 通信で袖領域を交換するために、2 つの通信用バッファex_hi_buffer と ex_low_buffer を追加している。まず図 7.6 の 1 行目で、hi_buffer に収められた袖領域を送る prev_proc が同一ノード内に存在するかどうかを判断する。これが別ノード内に存在する場合は、このプロセスに MPI 通信でこの袖領域を送信する。さらに、この prev_proc から MPI 通信で受け取る low_buffer の内容を hi_buffer に格納し、low_buffer を指すポインタ用の変数 low_buffer_p にこの hi_buffer のアドレス値を代入する. また、prev_proc が同一ノード内にある場合は、MPI 通信を行う必要がないので、7 行目以降のように、low_buffer_p に prev_proc の low_buffer のアドレス値を代入することにより、アンパックで prev_proc の low_buffer を直接参照できるようにする。

次に 11 行目で、low_buffer に収められた袖領域を送る next_proc が同一ノード内にあるかどうかを調べ、これが別ノード内に存在する場合は、このプロセスに MPI 通信でこの袖領域を送信する。さらに、MPI 通信でこのプロセスから袖領域を受け取り、low_buffer のアドレス値を後々アンパックで用いる、hi_buffer を指すポインタ用の変数 hi_buffer_p に代入する. また、next_proc が同一ノード内にある場合は、MPI 通信を行う必要がないので、17 行目以降に示すように、hi_buffer_p に next_proc の hi_buffer のアドレス値を代入することにより、アンパックで next_proc の hi_buffer を直接参照できるようにする。

そして、この交換を実行した後、プロセスがこの手順で得られた low_buffer_p と hi_buffer_p から、それぞれ自分の下側、上側の袖領域に対してアンパックを実行すれば、1 次元の袖領域の更新の処理が完了する。実際には、XMP のグローバル配列のそれぞれの次元について、その次元が袖領域を持っているかを判定しながら、持っている次元に対してこの処理を繰り返すことで、1 つ配列の袖領域の更新が完了する。

```
1  if(prev_proc が別ノード内にある場合){
2     (1)hi_buffer の内容を prev_proc にMPI 送信
3     prev_proc からの MPI 受信内容を high_buffer に格納
4     (4−p)low_buffer_p = hi_buffer へのポインタ値
5  }
6  else{
7     (4−i)low_buffer_p = prev_proc の low_buffer へのポインタ値
8  }
9
10 if(next_proc が別ノード内にある場合){
11    (3)low_buffer の内容を next_proc に MPI 送信
12    next_proc からの MPI 受信内容を low_buffer に格納
13    (2−p)hi_buffer_p = low_buffer へのポインタ値
14 }
15 else{
16     (2−i)hi_buffer_p = next_proc の hi_buffer へのポインタ値
17 }
```

図 7.6: 交換処理の新実装

第8章　提案方式による実装の評価

メニーコアプロセッサの特性評価の結果から、XMP の実行時システムのデータの分散方法と node 間のデータ交換方法、特に reflec 指示について改良を行った。本章では、これらを実装した提案方式の性能評価を行う。この性能評価には、2 次元のステンシルコードの Laplace と 3 次元のステンシルコード姫野ベンチマークを用いた。そして、1 次元または 2 次元の node を用いて並列実行を行い、従来の XMP の実行時システムとその性能を比較した。性能評価を行う環境としては、Xeon Phi 単一ノードと 16 台までのマルチノード・システムを用いた。

8.1　実験環境

本章で用いた実験環境は、メニーコアプロセッサの特性評価の章 6.1 節の表 6.1 に示したものと、まったく同様である。すなわち、Ivy Bridge ベースの 2.6 GH ｚの Xeon を 2 個使用しているサーバの PCI Express に、Xeon Phi を 1 個搭載したものである。この Xeon Phi は、7210 (1.238 GHz) で Manycore Platform Software Stack (MPSS) 2.1.6720 を使用して動作させた。

XMP としては、Omni XMP compiler 1.1 (build1322)[30] をベースのコンパイラとして使用した。バックエンドのコンパイラと MPI には、それぞれ、Intel complier version13.1.3.192 (build 2013607) と Intel MPI library 4.1.1.036 を使用した。

これらの環境やコンパイラは、2016 年時点の最新のものに比べると、やや古い。Xeon Phi は新しいアーキテクチャであるがゆえに、ここで使用したコンパイラの完成度は最新のものに劣るところがある。実際に、本研究で行った性能改善のための工夫の一部、例えば、ソフトウェア・プリフェッチ命令をループ内に挿入する等は、新しいコンパイラでは不要になるケースもあった。これはコンパイラが自動で挿入してくれるように改善されたからである。しかし、もともと対象としているステンシル計算が比較的単純な構成であるため、コンパイラや OS の差異自体の影響はさほど大きくなかった。

8.2 メニーコア単一ノード上での評価

我々の実装を評価するために、2 つのステンシルコード、Laplace と姫野ベンチマークをコンパイルして、実機上で性能を評価した。本節では、Xeon Phi コプロセッサ単体での性能評価について述べる。

8.2.1 Laplace ベンチマーク 1 次元分割

最初の性能評価には、簡単な 2 次元の Laplace ベンチマークを使用した。このベンチマークは 2 次元の Laplace 方程式を 5 点差分して、ヤコビ法で解く単純なステンシル計算である。2 次元の空間配列の片方の次元を分割する 1 次元分割でこのベンチマークの性能測定を行った。

8.2.1.1 データの 1 次元分散配置

この Laplace の 1 次元分割での XMP 版プログラムについては、既に 5.3 節で図 5.2 に、そのデータ分配と、並列化の部分を示し、使用するグローバル配列の説明を行った。図 8.1 に再掲する。演算を行う配列の大きさとしては、XSIZE×YSIZE (10000×10000) を用いた。データ分割には、配列の最初（もっとも左側）の次元の x 方向を用いることにして、倍精度浮動小数点数の 2 次元配列、u[ ][ ] と uu[ ][ ] のこの方向側を分割する。

この Laplace プログラムは、図 8.1 に示すように、17 行目と 25 行目の 2 つの for ループのブロックを持っている。プログラム中の両方のブロックとも、その前の行で #pragma xmp loop(x) on t(x) のように XMP の指示文で、修飾されている。この指示文は、次のループのインデックス変数 x で周回するループを並列に実行することを指示している。

8.2.1.2 1 次元分割性能評価

提案する実装のノード内の性能の優位性を確認するために、この 1 次元分割の Laplace プログラムの、我々の実装を用いたときの性能 Lap_XMP_SHM を、従来の何も変更を加えていない XMP 版の性能と、標準的な OpenMP を用いて並列化したものと比較した。この結果を図 8.2 に示す。図 8.2 中では、従来の XMP 版を Lap_XMP_MPI、OpenMP 版は Lap_OMP として示した。

Lap_XMP_SHM は、60 プロセスで実行したときに、13.6 GFLOPS の倍精度演算性能を実現することに成功している。この値を演算に必要なメモリ転送帯域幅に換算すると、194.6

```
1  #define XSIZE (10000)
2  #define YSIZE XSIZE
3
4  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
5
6  #pragma xmp nodes n(*)
7  #pragma xmp template t(0:10000-1)
8  #pragma xmp distribute t(block) onto n
9  #pragma xmp align u[j][*] with t(j)
10 #pragma xmp align uu[j][*] with t(j)
11 #pragma xmp shadow uu[1][0]
12
13 ...
14
15     /* old <- new */
16 #pragma xmp loop (x) on t(x)
17     for(x = 1; x < XSIZE-1; x++)
18       for(y = 1; y < YSIZE-1; y++)
19         uu[x][y] = u[x][y];
20
21 #pragma xmp reflect (uu)
22
23     /* update */
24 #pragma xmp loop (x) on t(x)
25     for(x = 1; x < XSIZE-1; x++)
26       for(y = 1; y < YSIZE-1; y++)
27         u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
```

図 8.1: Laplace1 次元分割 XMP プログラム

GB/秒となり、これは、Stream Triad のベンチマーク性能で得られた Xeon Phi の最大メモリ転送帯域幅 [25] とほぼ同等である。それぞれの実装のスケーラビリティを確認するために、プロセス数（Lap_OMP の場合はスレッド数）を、1, 2, 4, 8, 16, 32, 60, 120, 240 と変化させ、その性能を測定した。

ノード単体での実行で、Lap_XMP_SHM は OpenMP で並列化した Lap_OMP に対して、1.17 倍の性能向上を実現している。また、MPI で実装されている Lap_XMP_MPI も Lap_OMP よりも優れた性能を示しており、これらについては次節で検討を行う。

8.2.1.3 考察

我々の実装を含めた XMP での Laplace の実行では、OpenMP 版と異なり、袖領域の転送を行う必要があり、この転送に余分なメモリ参照を行う必要がある。しかしながら、それにもかかわらず、Lap_XMP_SHM と Lap_XMP_MPI は図 8.2 の並列数が 240 を除く領域でより良い性能を得ている。この原因を調べるために、プログラム実行時のメモリ参照回数を測定した。測定したのは、Xeon Phi の性能カウンタのうち、L2 キャッシュへのデータ転送回数（L2D_MISS）と、1 次キャッシュからコアへのメモリ転送回数（DataRD/WR）で、

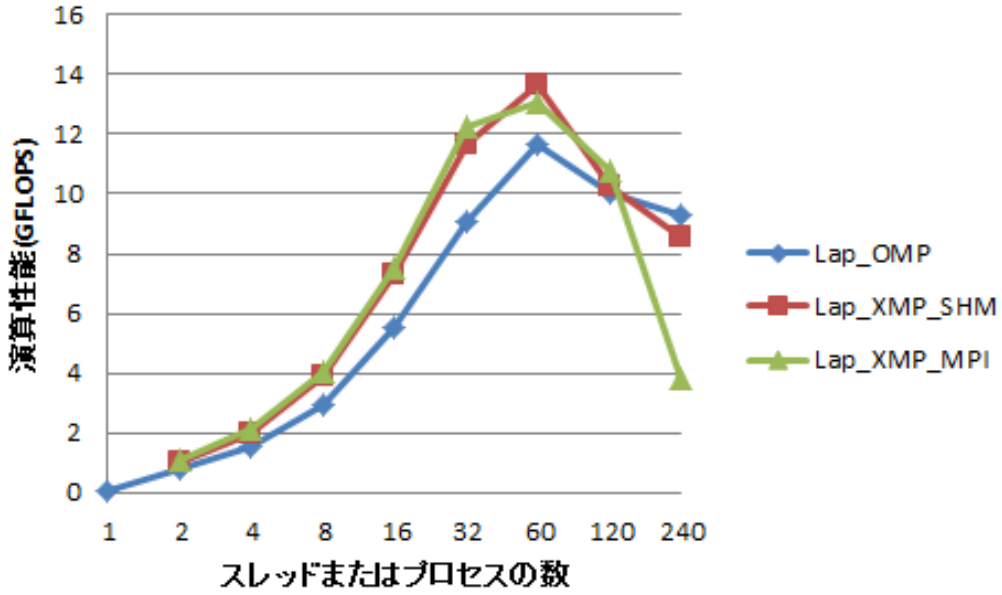

図 8.2: Laplace 1 次元分割の測定結果

その結果を図 8.3 に示す。

L2D_MISS は、プリフェッチも含み、コアのローカルな L2 キャッシュへキャッシュライン分のデータ（64 バイト）が転送された回数を示す。Xeon Phi はコア間の共有キャッシュを持たないため、コアにローカルな 2 次キャッシュをミスすると、数 100μ 秒のストールが発生し、この L2D_MISS がメモリ参照に要する時間を支配する。一方、DataRD/WR は、プログラム実行時のプリフェッチを除くすべてのデータの読み出しと、書き込みの回数（これもキャッシュライン単位）である。プログラム内計算ループ中の 1 回のループ実行演算でのこれらの値を測定した。なお、これらの性能カウンタの詳細は文献 [13] を参照されたい。

この測定は、32、60 と 120 の並列数で行った。袖領域のコピーと更新を行うため、Lap_XMP_SHM と Lap_XMP_MPI では、DataRD/WR の数が Lap_OMP よりも多くなる。これの DataRD/WR の数の差が、袖領域の処理のために増えたデータ参照回数といえる。データ参照の数が増えたのにもかかわらず、いずれの実装でも、L2D_MISS の数は、約 50 M 回のキャッシュライン参照と、ほぼ同等の値を示している。

これらの測定結果から、Lap_XMP_SHM では、メモリ操作数（DataRD/WR）が OpenMP 版 Lap_OMP よりも多いにも関わらず、演算時に、より 2 次キャッシュを有効に利用して、2 次キャッシュへのデータの転送回数（L2D_MISS）を減らし、同等以上の性能を得ていることが確認できた。

さらに、Lap_XMP_SHM がより優れた性能を得られる原因を確認するために、1 次キャッシュへのヒット率を調べた。表 8.1 にこの結果を示す。袖領域を持つ、Lap_XMP_SHM

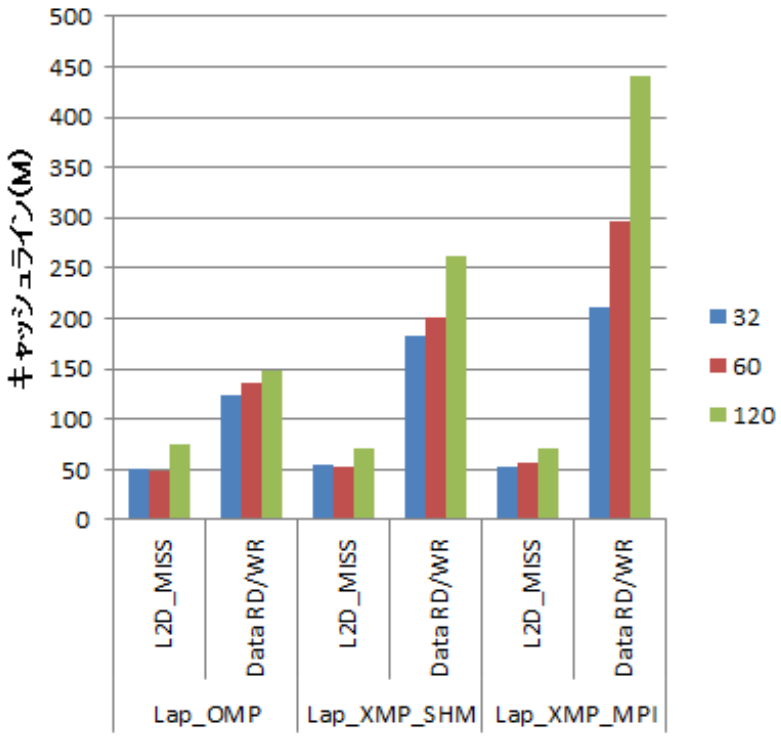

図 8.3: Laplace 1 次元分割のメモリ転送事象

と Lap_XMP_MPI がすべての領域において、ほぼ 1.00 のヒット率を得ているのに対して、Lap_OMP ではヒット率が低く、我々の実装の優位性が説明できる.

表 8.1: 1 次キャッシュのヒット率

スレッド/プロセス数	32	60	120
Lap_OMP	0.92	0.91	0.98
Lap_XMP_SHM	1.00	1.00	1.00
Lap_XMP_MPI	0.99	1.00	0.99

8.2.1.4 Xeon Phi 上の配列境界参照の影響

8.2.1.3 節で、本研究の実装 Lap_XMP_SHM はメモリ参照回数が多いのにもかかわらず 2 次キャッシュへのデータ転送回数が少ないことを示した。この理由として、コアの 2 次キャッシュ間のキャッシュラインの取り合いが考えられる. この影響を確認するために、Laplace の 1 次元分割の実行において、グローバル配列を分割せずに、そのままプロセス間共有メモリを上に置くプログラムを人手で作成して、その性能を比較確認した。図 8.4 にその Lap1d_g_shm の結果を示す。

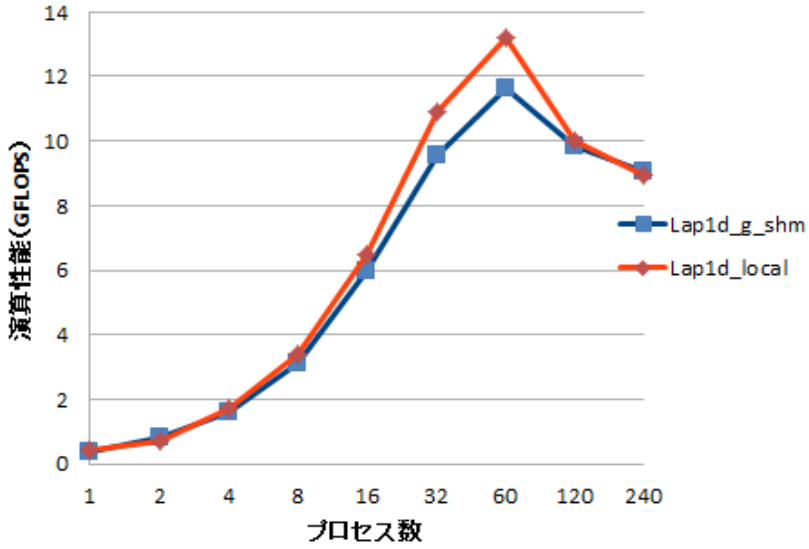

図 8.4: 配列境界参照の影響

図 8.4 中の Lap1d_g_shm は Lap_OMP のプログラムを MPI を用いて書き換えて、プロセス間共有メモリ上のグローバル配列を直接参照している。このため、配列境界領域では複数コアから参照が発生する。一方 Lap1d_local の方は、本研究で採用した実装と同様に、複数コア間で同じ配列領域を参照しないようにしたものである。ここでは袖領域の更新も行っていない。測定した結果、図 8.4 の Lap1d_g_shm は、図 8.2 の Lap_OMP と同様に低い性能であることがわかった。また、Lap1d_local も、Lap_XMP_SHM と同等の高い性能を示している。

以上から、グローバル配列をコアごとに分割して配置することが、Xeon Phi の性能向上に貢献していることが確認できた。

8.2.2 Laplace ベンチマーク 2 次元分割

Laplace の 1 次元分割の XMP プログラムと、OpenMP プログラムの実行比較では、他の２つの実装と比較して、Lap1d_SHM で最も良い性能が得られることが確認できた。そこで次に、同じ Laplace を用いて、グローバル配列を XMP で 2 次元分割した場合の測定を行う。

図 8.5 に、この 2 次元分割のプログラムを示す。XMP の指示文は、グローバル配列の u[ ][ ] と uu[ ][ ] を 6×10 の 2 次元に分割するように挿入する。まず 1 行目で、2 次元の矩形の nodes の p(6,10) を定義している。次に、template t(0:10000-1,0:10000-1) が定義され、3 行目でこれを p(6,10) に対して t(block,block) と、両方の次元で均一ブロックに分割することを指示する。この結果 template t(0:10000-1,0:10000-1) は、[10000/6×10000/10] の 60 個の

矩形に分割される。グローバル配列の u[ ][ ] と uu[ ][ ] は、5、6 行目で、この t(j,i) に align されることにより、それぞれ同様な形に分割される。また 7 行目の shadow uu[1][1] に示すように、uu[ ][ ] の分割の両方の次元に袖領域を持たせる。この、2 次元分割の形としては、1 次元分割と同様な並列数でスケーラビリティを確認するために、分割の nodes p の形として、(1,2), (2,2), (2,4), (4,4), (4,8), (6,10), (10,12) ,(15,16) を使用する。

```
1  #pragma xmp nodes p(6,10)
2  #pragma xmp template t(0:10000−1,0:10000−1)
3  #pragma xmp distribute t(block,block) onto p
4  #pragma xmp align u[j][i] with t(j, i)
5  #pragma xmp align uu[j][i] with t(j, i)
6  #pragma xmp shadow uu[1][1]
7
8       /* old <− new */
9  #pragma xmp loop (x,y) on t(x,y)
10      for(x = 1; x < XSIZE−1; x++)
11        for(y = 1; y < YSIZE−1; y++)
12          uu[x][y] = u[x][y];
13
14 #pragma xmp reflect (uu)
15
16      /* update */
17 #pragma xmp loop (x,y) on t(x,y)
18      for(x = 1; x < XSIZE−1; x++)
19        for(y = 1; y < YSIZE−1; y++)
20          u[x][y] = (uu[x−1][y] + uu[x+1][y] + uu[x][y−1] + uu[x][y+1])/4.0;
```

図 8.5: 2 次元分割 Laplace

8.2.2.1 Laplace の 2 次元分割の性能評価

この 2 次元分割の Laplace での性能測定結果を図 8.6 に示す。図中の本実装での測定結果は Lap2d_SHM として表示している。測定結果は、1 次元分割と同様に、オリジナルの XMP での実装と、標準的な OpenMP での実装と比較して示した。図中これらはそれぞれ、Lap2d_MPI と Lap1d_OMP として表示している。ただし、OpenMP ではループ単位の 2 次元の分割はサポートされていないので、OpenMP のみ 1 次元分割の結果を使用している。

2 次元分割では、袖領域の大きさは小さくなり、また大きなグローバル配列を両次元方向に小さく分散データ配置することによる、局所性の向上が望めるため、より大きな性能の向上が得られると考えた。しかしながら実際は、Lap2d_SHM の性能は、同じ並列数で 1 次元分割実行したときの、Lap1d_SHM の性能はもとより、Lap1d_OMP よりも悪い結果となった。

この性能低下の原因を調べるために、1 次元分割の場合と同様なメモリ参照の測定を行った。図 8.7 に、この 2 次元分割で Laplace を実行したときのメモリ参照に伴うキャッシュラ

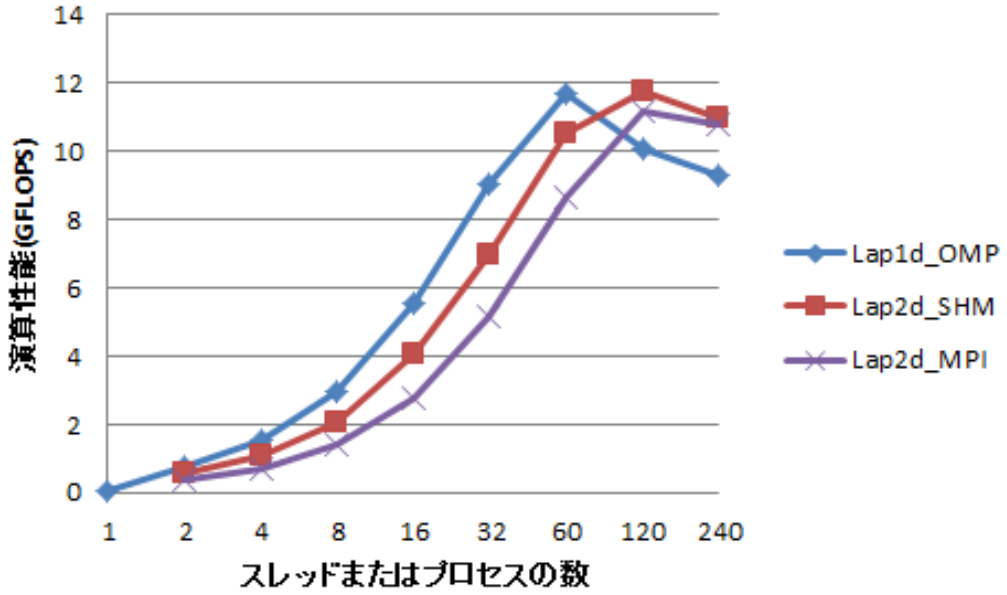

図 8.6: Laplace 2 次元分割の測定結果

イン転送の回数を示す。この測定の Lap2d_SHM と Lap2d_MPI の値から、この 2 次元の分散データ配置によって、キャッシュライン転送の回数が、予想とは逆に増えていることがわかる。このことから、袖領域の大きさを小さくすることで、必ずしもキャッシュライン転送の回数は少なくならないことがわかる。

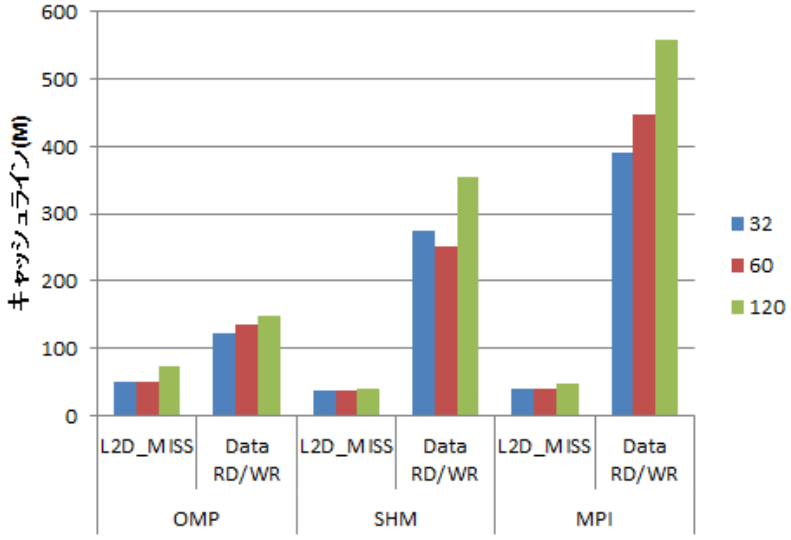

図 8.7: Laplace 2 次元分割のメモリ転送事象

8.2.2.2 Laplace の 2 次元分割の実行時間解析

Laplace 2 次元分割プログラムを実行時に、Xeon Phi が実行時間を要している関数を調べるために、各測定における、関数別の実行時間の比較を行った。図 8.8 にその結果を示す。比較は、Lap1d_OMP、 Lap2d_SHM と Lap2d_MPI において、それぞれ 120 並列で行った。それぞれ、図 8.8 中で色付けされている部分は、右側にリストされている各関数を実行する時間を、全コア分総和したものを示している。図の libiomp5.so は、OpenMP の実行時システムの実行時間を示す。others は、図中の他の時間に分類できない時間の総和を示す。xmpc_main は、XMP の実行時システムのメインプログラムの実行時間、UnPack_2D と Pack_2D は、それぞれ、XMP の reflect 処理中の Pack と UnPack を行うのに要する時間を示す。libmpi_dbg.so.4 は、MPI の実行時システムの実行時間で、lap_mpi が Laplace のプログラムの演算時間である。図 8.8 では lap_mpi が最も大きいので、詳細を拡大するため実行時間の目盛は、300 秒を原点にしている。

まず、Laplace のプログラムの本体である、lap_main の実行時間を比較してみると、Lap1d_OMP ではこの部分に 450 秒を要しているのに対して、Lap2d_SHM と Lap2d_MPI の 2 つはともに 350 秒以下しか要していない。この、約 100 秒ほどは、配列を 2 次元に分散配置した効果だと考えられる。しかし、この代わり、Lap2d_SHM と Lap2d_MPI では、それぞれ合計約 70 秒と約 90 秒の時間が、Pack2D、UnPack2D と libmpi_dbg.so.4 に費やされている。これは、袖領域の更新のための通信を、2 次元分割された配列の 2 つ方向で実行している時間と考えられる。Lap2d_MPI と比較すると Lap2d_SHM の方がこの通信により少ない時間を要しており、この結果、より優れた性能を得ることができる。

なお、メモリ転送事象の測定と、実行時間解析には、本研究の条件でのノード単体でのプログラム実行性能の相違は無いことを確認したうえで、デバッグ用の MPI の実行モジュール libmpi_dbg.so.4 を用いた。差異はないのだが、その他の MPI 版のプログラム実行性能の測定はすべて MPI の標準の実行モジュール libmpi.so.4 を用いた結果を示している。

8.2.2.3 不連続な配列境界参照の影響

Laplace の 2 次元分割での性能低下の原因は、袖領域の通信の影響であることが分かった。これが、部分配列の 2 つの次元の境界のうち、隣り合う要素同志が、メモリ上で連続しない次元側の袖領域の影響ではないかと考え、これを調べるために、さらに 2 種類のプログラムで測定を行い、Lap1d_OMP とこれらを比較した。図 8.9 にこの結果を示す。図 8.9 の Lap2d_SHM_NUP は、演算結果を無視して、袖領域の更新を行わない場合の性能を示して

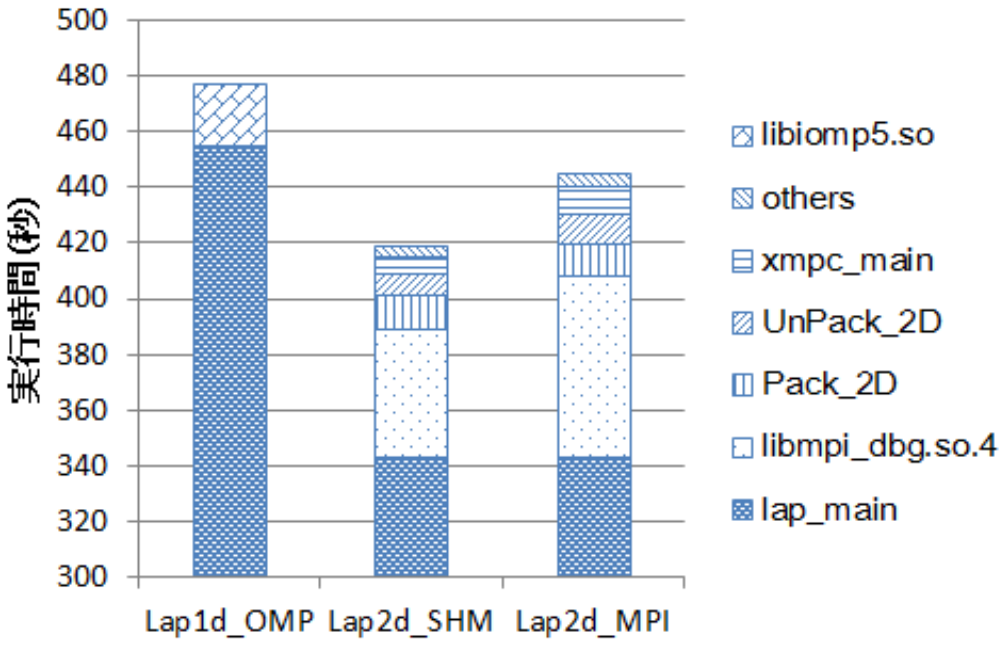

図 8.8: Lap2d 120 スレッドまたはプロセス関数別実行時間

おり、もう一つの Lap2d_SHM_1UP は、袖領域の最初の次元だけの更新を行い、2 次元目の更新を行わない場合の性能である。いずれの場合も、 より良い結果が得られている。このことから、袖領域の要素がメモリ上で連続していない、2 次元目の更新が、この場合の性能の低下を引き起こしていることが分かった。

8.2.2.4　2 次元分散データ配置効果の確認

図 8.8 の lap_main の実行時間で確認した、配列の 2 次元分散配置の効果を、並列数を変えて見るために、図 8.10 に示すように、L1D と L2D のキャッシュヒット率を測定した。図 8.10 では、Lap1d_OMP、 Lap2d_SHM と Lap2d_SHM_NUP のキャッシュヒット率の、それぞれ、並列数が、32、60、120 で実行したときの測定値を示す。分散データ配置を行った Lap2d_SHM と lap2d_SHM_NUP の 2 つは、L1D のキャッシュヒット率がいずれの場合も 0.95 近くに達している。これに対して、Lap1d_OMP だけが、並列数が 32 と 60 で 0.91 程度と低くなる。L2D のキャッシュヒット率の違いはもっと大きい。前者がすべての場合で 0.85 程度なのに対して、後者は 0.65 程度か、それ以下である。このことから、これらの並列数では、いずれも 2 次元分散データ配置の効果が得られていることが確認できた。

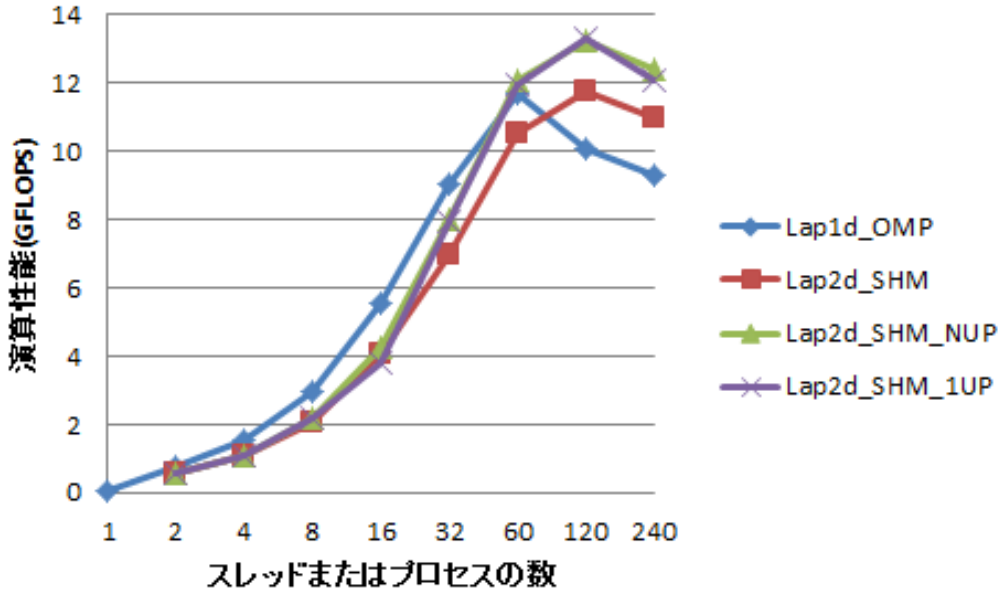

図 8.9: Laplace 2 次元分割の袖更新を止めた測定結果

8.2.2.5 Laplace 2 次元分割問題サイズ解析

ここまで、この Laplace のベンチマーク測定では、1 つの問題サイズ、配列サイズ 10000×10000（表 8.2 中では medium とした。）だけを対象に行った。配列サイズの大きさの性能への影響を考察するために、表 8.2 に示す、これより小さい配列サイズの small（4000×4000）と、大きい配列サイズ large（20000 × 20000）の性能測定を行った。

表 8.2: Laplace 問題サイズ

	small	medium	large
配列サイズ	(4000,4000)	(10000,10000)	(20000,20000)
使用メモリ (GB)	0.256	1.6	6.4

図 8.11 に、表 8.2 に示した 3 つ配列サイズでの性能測定結果を示す。この測定は、我々の実装の Lap2d_SHM を、OpenMP 版と比較して Xeon Phi 上で実行することで行い、この際、3 つの異なる並列数、60、120、240 を用いた。図 8.11 に示すように、SHM の large が、この測定での最高性能の 12.44 FLOPS を並列数 120 で実現しており、SHM の medium も、これにはやや劣るものの、同様の性能を出している。この SHM の large と medium の両方とも、並列数に対しても似たような傾向を示す。一方、OMP の中では、small が最高性能の 12.3 GFLOPS を 120 スレッドで実行時に達成している。OMP の medium と large では、それぞれ、60 スレッドでの実行性能が最も高く、120、240 とスレッド数を増加させると性能が低下してしまう。

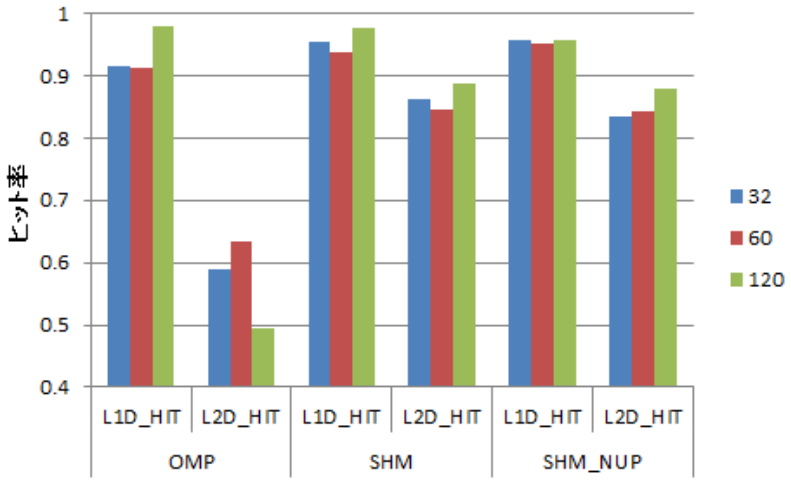

図 8.10: L1D の L2D キャッシュヒット率

この解析から、我々の実装は、medium から large の 1.6 6̃.4 GB 程度の配列サイズの演算では OpenMP よりも優れている。small のような 256 MB の配列サイズの演算のときのみ、OpenMP 版がより良い性能を実現できることがわかった。

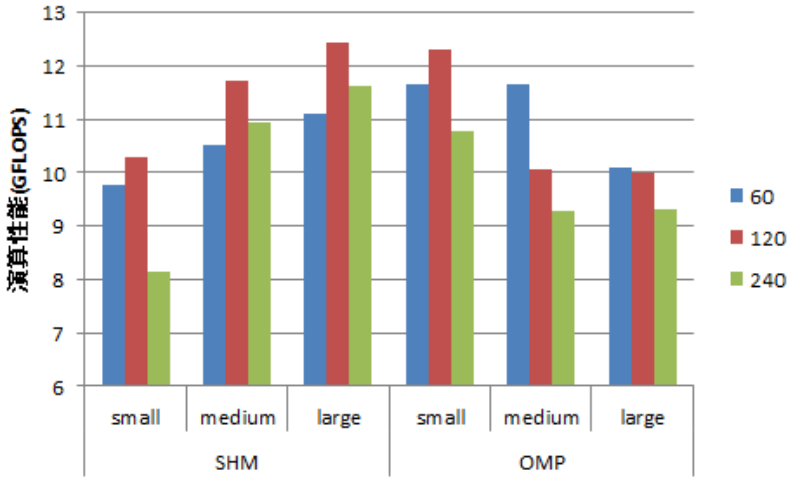

図 8.11: 異なる問題サイズでの性能比較

8.2.3 姫野ベンチマークでの性能評価

実際的なステンシルコードを使用した場合の性能を評価するために、姫野ベンチマーク [41] を用いて本実装の評価を行った。この姫野ベンチマークは、3 次元格子空間のポアソン方程式を解くための、19 点ステンシルコードである。

8.2.3.1 データ配列の分散配置

本評価では、3 次元格子空間のうち 2 つの次元を分割する 2 次元分割を採用した。本ベンチマークには使用する問題サイズとして、5 つの問題サイズがあり、このうち、L(256×256×512) を用いてその評価を行った。図 8.12 に、この姫野ベンチマークを並列化するために挿入した XMP の指示文をプログラム中から抜き出して示す。

```
1  ...
2  static float p[257][257][513];
3  static float a[4][257][257][513], b[3][257][257][513],
4                       c[3][257][257][513];
5  static float bnd[257][257][513];
6  static float wrk1[257][257][513], wrk2[257][257][513];
7
8  #pragma xmp nodes n(6,10)
9  #pragma xmp template t(0:512,0:256,0:256)
10 #pragma xmp distribute t(*,block, block) onto n
11 #pragma xmp align [i][j][*] with t(*, j, i) :: p,bnd,wrk1,wrk2
12 #pragma xmp align [*][i][j][*] with t(*, j, i) :: a,b,c
13 #pragma xmp shadow p[1][1][0]
14 ...
15 #pragma xmp reflect (p)
16 #pragma xmp loop(i,j) on t(*,j,i) reduction (+:gosa)
17       for(i=1 ; i<imax-1 ; i++)
18         for(j=1 ; j<jmax-1 ; j++)
19           for(k=1 ; k<kmax-1 ; k++){
20             s0 = a[0][i][j][k] * p[i+1][j ][k ]
21                + a[1][i][j][k] * p[i ][j+1][k ]
22                + a[2][i][j][k] * p[i ][j ][k+1]
23                + b[0][i][j][k] * ...
24 ...
```

図 8.12: XMP による姫野ベンチマークの記述

図 8.12 の 2 行目から 6 行目では、本プログラムで使用される 7 つのグローバル配列が定義されている。この XMP プログラムでは 6 × 10 の 2 次元の分割を行うため、8 行目で nodes n(6,10) を宣言している。次の行で、使用する template t の形を計算する 3 次元配列 p[257][257][513] と同じ形に宣言した後、これを nodes n 上に t(*,block,block) と 2 つの次元でブロック分割する。この結果、配列の最も左側の次元では 6 個の区間に分割され、次の

次元は 10 個の区間にされる。最後の次元は*で指定され、513 個の要素はそのまま保持される。この結果、配列は 60 個のブロックとなる。

図 8.12 の 11 行目で、配列 p と同じ形を持ったすべての配列が template t と同じ方法で分配される。配列の a、b、c は、p と同じ形をした配列を含む形だが、次元数が一つ多いので 12 行目で分配を指示している。配列 p に関しては、13 行目で分割した 2 つの次元方向にそれぞれ 1 個ずつの袖領域を追加する指示を行っている。15 行目の reflect は、袖領域のアップデートのタイミングを示している。プログラムの演算メインループの直前、図 8.12 の 17 行目から 19 行目では、XMP の並列化のための指示文 xmp loop(i,j) on t(*,j,i) に、変数 gosa の reduction の指示を付加したものを使用している。

8.2.3.2 性能評価

この姫野ベンチマークの測定結果を図 8.13 に示す。スケーラビリティを確認するために、測定する並列数として 8 個の並列数を用い、表 8.3 に示すように、それぞれ 2 次元分割の形を指定する XMP の nodes n を決めた。これらは、同じ並列数を持つ複数の 2 次元分割の中から、最も高い性能が得られるものを選んだ。

表 8.3: ノード内の XMP node の配置

並列数	2	4	8	16
node	n(1,2)	n(2,2)	n(2,4)	n(4,4)
並列数	32	60	120	240
node	n(4,8)	n(6,10)	n(10,12)	n(10,24)

このベンチマークの測定でも、本実装 Hime2d_XMP_SHM を従来の XMP での実装 Hime2d_XMP_MPI と OpenMP での実装 Hime1d_OMP と比較した。また、OpenMP 版で並列化する最外ループの繰り返し数が 256 と並列数に対して少ないので、Hime1d_OMP の他に、これに collapse (2) を用い、演算用のメインループの外側の 2 つのループを並列化した、Hime1d_OMP_COL の測定も行った。本測定では、Hime2d_XMP_SHM が並列数 120 で、50 GFLOPS の最高性能を得ている。Hime1d_OMP と Hime1d_OMP_COL の 2 つが、ほぼ同じ性能で 45GFLOPS に達している一方、Hime2d_XMP_MPI は並列数が 60 以上で性能が低下してしまっている。

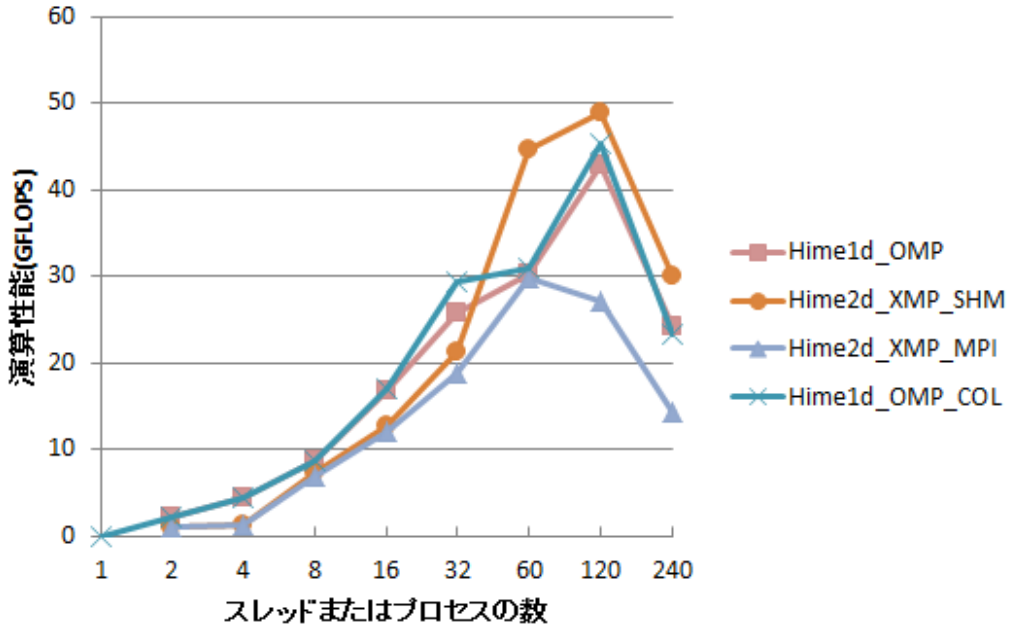

図 8.13: 姫野ベンチマークの性能

8.2.3.3 考察

図 8.13 の 60 より少ない並列数では、コア間にまたがる袖領域を明示的に通信しない Hime1d_OMP や Hime1d_OMP_COL のような OpenMP 版のプログラムが、Hime2d_XMP_SHM や Hime2d_XMP_MPI のような XMP 版より優れた性能を示している。

XMP 版で必要となる node 当たりの通信バッファの大きさは、隣接する node の数と、分割したグローバル配列の隣接 node との切断面の大きさとの積の 2 倍である。XMP 版のように 2 次元分割の場合、端を除き隣接 node 数は 4 つとなるが、並列数が少ない場合は分割数自体が少ないためにこれより少なく、並列数 2, 4, 8 で隣接 node 数は 1, 2, 3 となる。逆に、切断面の大きさは、各次元の分割数に反比例して減るため、この結果、本測定条件で並列数が 32 より少ない場合の通信バッファの大きさは約 1 MB 程度とほぼ一定となる。

隣接 node 数が 4 になる並列数が 16 以上の、並列数 32, 60, 120, 240 では、それぞれ 804 KB, 566 KB, 394 KB, 304 KB と分割数が増えるほど減少する。XMP 版では、この通信バッファを介して通信を行うので、通信量は通信バッファの大きさの 2 倍となる。

ところで、OpenMP 版は明示的な通信は行わないが、演算のために XMP 版と同様に、隣接スレッドの算出した演算結果が必要となり、これと同様な方法で分割した XMP 版で必要となる通信量と、同量の共用データへの参照が発生すると考えられる。OpenMP 版は 1 次元分割のため、node 当たりの通信量は、両端の node を除き 2108 KB で、並列数によらず一定である。

したがって、並列数が 32 より小さい場合は、XMP 版で必要な通信量は、OpenMP 版とほぼ同等の 2 MB で、32 の場合でも 2 割程度少ない程度と両者の通信量は同程度である。このような場合、全 node が一斉に通信バッファに値を出力した後、別のバッファから同量の値を入力する XMP 版では、全 node が、全 node の通信の完了を待ってから次の演算に移る必要があるため、このための時間が演算時間に別途加算され、より時間を要する。

後述するように、XMP 版では明示的な通信を行うことで、演算部分の実行時間を、通信時間の増加分より大きく短縮しようとしている。残念ながら、並列数が 60 より少ない場合には、両者の演算時間に大きな相違はなく、このため、XMP 版は明示的な通信を行わない OpenMP 版よりも性能が劣ってしまう。

図 8.13 で OpenMP 版は並列数 32 から 60 で性能向上が滞っており、これは共用データ参照のためのスレッド間の同期の影響と考えられる。ところで OpenMP 版は並列数が 120 で再び性能が向上しているが、この原因を確定できなかった。並列数 60 と 120 の大きな違いは、2 つのスレッドがコアを共用することで、この場合、同期がコア内で行われることの他、Xeon Phi は 2 スレッド同時実行時で最大性能が得られるアーキテクチャであり、このことが、この原因として考えられる。

姫野ベンチマークでの測定でも、我々の実装の並列数が 120 で最も高い性能を実現しており、これはグローバル配列に袖領域を追加して分割配置し、通信を最適化した効果と考えられる。これを確認するため、このベンチマークを並列数 120 で実行したときの各関数の合計実行時間を測定した。図 8.14 に、その結果を示す。この測定では、Hime1d_OMP_COL、Hime2d_XMP_SHM と Hime2d_XMP_MPI の 3 つを比較した。Hime2d_XMP_SHM と Hime2d_XMP_MPI は、ステンシル計算の繰り返し演算を行っている、jacobi の実行時間が Hime1d_OMP_COL よりも 75 秒程短い。

この理由を確認するため、並列数 120 で 1 回の演算を行う際の、2 次キャッシュの読み込みミスの回数を、Hime2d_XMP_SHM と Hime1d_OMP_COL で比較した。図 8.15 の L2D_RD_MISS にこの結果を示す。

Hime2d_XMP_SHM の 17.0 M 回に対して、Hime1d_OMP_COL では、倍近い 33.4 M 回のキャッシュミスを起こしていることがわかる。図 8.15 には、L2D_RD_MISS を生じさせる可能性のある事象、2 次キャッシュに対するプリフェッチの回数 L2_DATA_PF と、コアのデータ入出力回数 DataRD/WR も示した。Hime2d_XMP_SHM では、L2_DATA_PF と DataRD/WR の合計回数が 327 M 回で、Hime1d_OMP_COL の 208 M 回対して 1.57 倍に増えているにもかかわらず、演算時の待ちをもたらす L2D_RD_MISS が半分近くに減っており、これは本実装のグローバル配列の分割実装の効果だと考えられる。

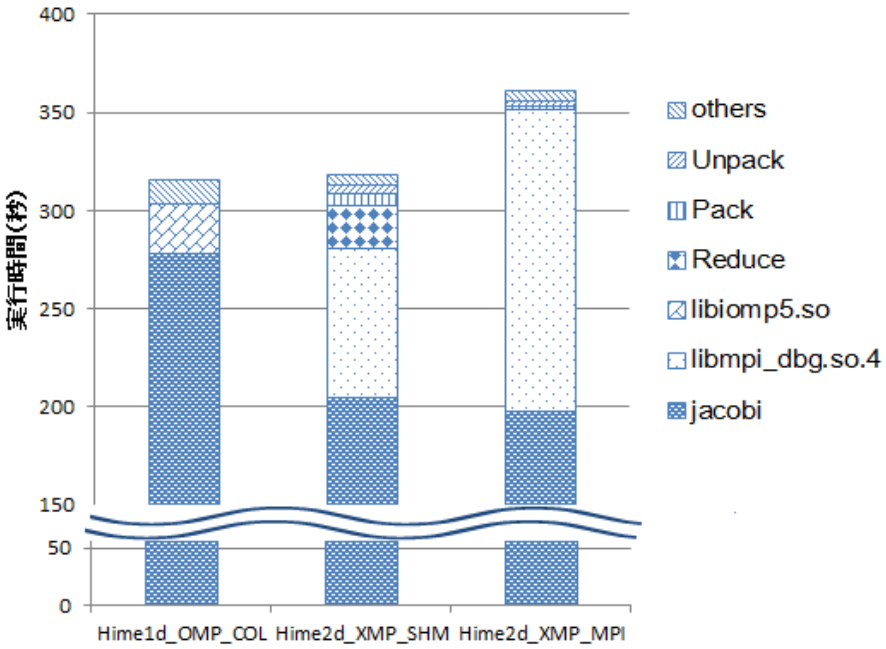

図 8.14: 姫野 120 スレッドまたはプロセスの内訳

しかし一方、図 8.14 の Hime2d_XMP_SHM と Hime2d_XMP_MPI では、短くなったのと同等か、もしくはそれ以上の時間が MPI ライブラリやその他の通信関連の関数に費やされている。このうち、Hime2d_XMP_SHM は、袖領域を交換する MPI 通信を削除しているため、libmpi_dbg.so.4[1] に要する時間が短く、最も優れた性能を実現している。Hime2d_XMP_SHM では、図 8.14 内の別の XMP 実行時関数 Reduce を実装するために MPI_Reduction を使用しており、残っている libmpi_dbg.so.4 はこの分である。なお、図 8.14 では、測定上必要な Xeon Phi 上の Linux カーネルの実行時間を除いている。OpenMP と MPI ではその時間が異なるため、Hime1d_OMP_COL の実行時間が、実際の実行時間よりも小さく示されている。

以上から、ノード単体において、グローバル配列に袖領域を加えて分割分配することで、ステンシル演算のカーネル実行時間を Hime1d_OMP_COL と比較して小さくでき、また袖領域の交換の MPI 通信を共有メモリ上の memcpy に置き換えることで Hime2d_XMP_MPI よりもコア間の通信時間を少なくできることが確認できた。

8.2.4 単一ノード性能評価のまとめ

本実装を評価するために、2 つのステンシルコード、Laplace と姫野ベンチマークをコンパイルして、実機上で性能を評価した。

[1]デバッグ用の MPI の実行モジュール libmpi_dbg.so.4 を用いることにより Xeon Phi ノード単体でのプログラム実行性能に相違が無いことは、別途確認済みである。

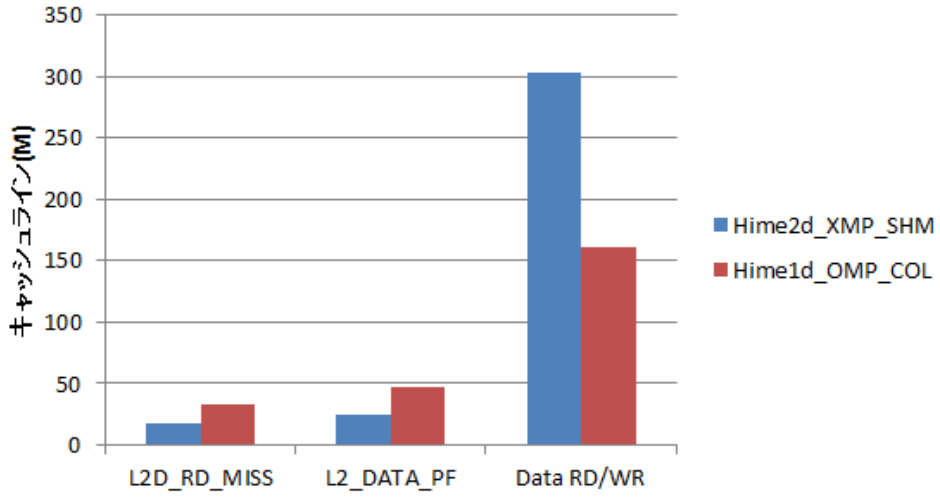

図 8.15: 姫野 2 次元分割のメモリ転送事象

Laplace の 1 次元分割では、OMP と XMP を比較して、XMP を用いて配列の分散と reflect によるコア間の袖領域交換を行う方が、OMP のように配列を分散せずに、演算のデータ参照時に各コアの L2 キャッシュ間で隣接データの交換を行うより高い性能が得られることが分かった。さらに、袖領域の交換に、MPI を用いる従来方式よりも、共有メモリを用いてこれを行う方が高い性能が得られる。

2 次元配列を用いた Laplace の 2 次元分割のように、片方の次元の分割境界の列（または行）の隣接要素が、メモリ上で連続しておらず、1 要素ごとに 64 バイト以上の間隔で飛び飛びに続く場合は、同じ並列数での XMP の性能が OMP より劣ることが分かった。これは、袖領域の通信に、１要素あたり 64 バイトのキャッシュライン分のデータ転送が必要となり、列の倍精度浮動小数点数がメモリ上で連続している場合の 8 倍の、要素の数と同じ数キャッシュラインを交換する必要が生じるためと考えられる。

3 次元配列を用いた姫野ベンチマークの 2 次元分割でも、本実装の XMP 版が、1 次元分割や collapse による並列化を行った OMP 版よりも、優れた性能を得ることを確認できた。交換する袖領域の大きさが 1　 MB 程度と、Laplace と比較して大きいため、reflect のための通信回数多い XMP の従来版は大きく性能を落としており、このような場合の本実装の有用性を確認できた。

8.3　マルチノード・システム上での評価

XMP で記述したプログラムは、必要に応じて並列数等を増やせば、そのままマルチノード上でも実行できる。単体での評価と同じベンチマークの並列数を増やしたものを用いて、

Xeon Phi を 1 枚実装した Xeon サーバ 16 台を QDR の Infiniband 経由でクラスタ接続したマシン上で実行時システム を評価した。本評価では、単体の評価と同様に、ネイティブ・モデルで Xeon Phi のみを用いて、Laplace と姫野ベンチマークを実行した。

8.3.1 ノード間の配列分配

グローバル配列は、XMP の node に align することで、node の形に分割され分配される。例えば、node が 2 次元で n(10,24) の場合、配列の 2 つの次元が、10 と 24 に分割されて、各 node には分割された配列が分配される。Xeon Phi の各ノードへの配列の分配は、各ノードに、この node を分割して分配することで行う。このノードへの 2 次元の分割を div(x,y) と表記する。例えば、n(10,24) の場合、div(2,4) では、図 8.16 に示すように、node n の各次元が 2 と 4 に分割され、この場合、それぞれ分割された部分 node sbn(5,6) が 8 個のノードに割り当てられる。

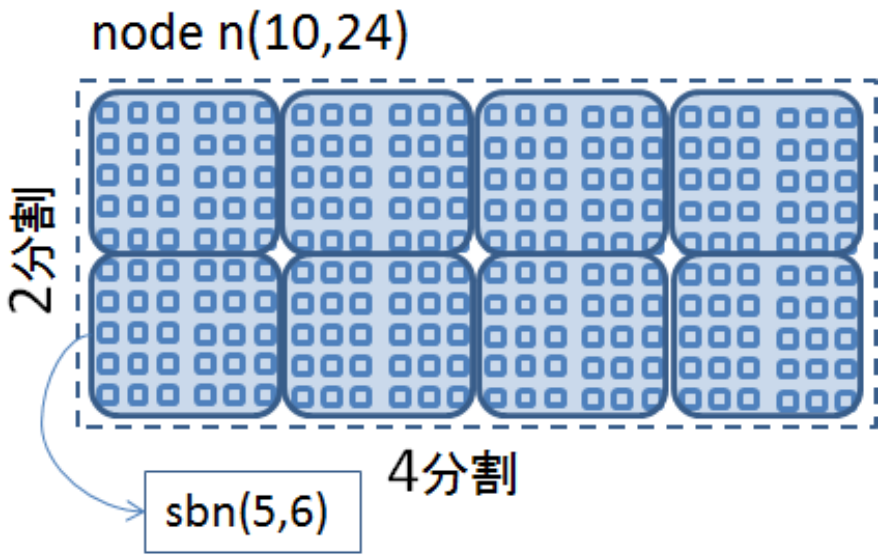

図 8.16: node の 2 次元分割 div(2,4)

8.3.2 Laplace 2 次元分割の性能評価

マルチノード上での Laplace の 2 次元分割の性能測定を、ノード数を 1 から 16 に倍々に変化させて行った。各ノード数での並列数と分割方法は表 8.4 に示す。ノード当たりの並列数は、単一ノードで最高性能が得られた 60 を用い、グローバル配列のノードへの分割方法は、数種類分割方法の評価を行い最も高い性能が得られたものを採用した。Laplace のグ

ローバル配列のサイズは、1 次元分割で行ったノード単体での測定と同じ 10000 × 10000 を用いた。図 8.17 にこの性能測定結果を示す。

表 8.4: Laplace node 分配

ノード数	1	2	4	8	16
並列数	60	120	240	480	960
node	n(6,10)	n(10,12)	n(20,12)	n(20,24)	n(24,40)
分割方法	div(1,1)	div(1,2)	div(2,2)	div(2,4)	div(2,8)

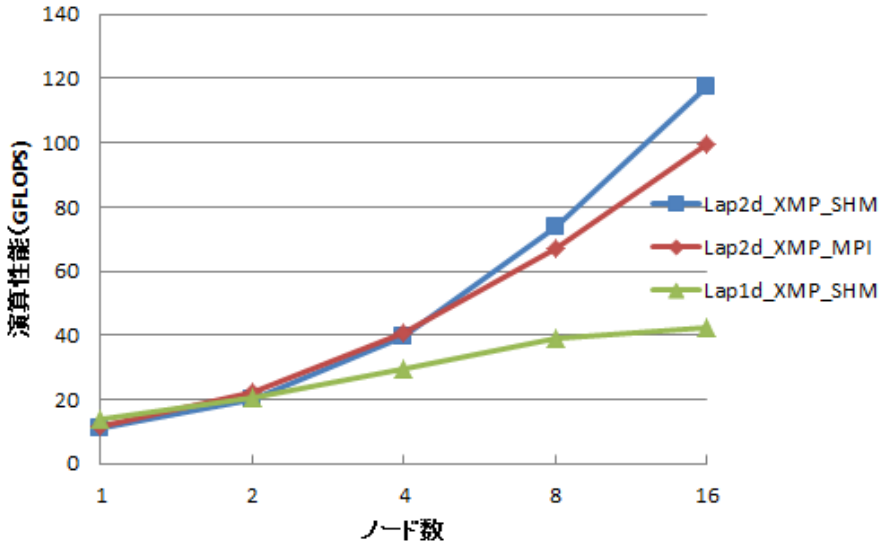

図 8.17: Laplace マルチノード実行結果

図 8.17 の Lap2d_XMP_SHM と Lap2d_XMP_MPI は、両者とも、XMP を用いて 2 次元分割した Laplace のプログラムを、表 8.4 の各ノード数での実行した場合の性能を示している。このうち、Lap2d_XMP_SHM は、提案する共有メモリを用いた reflect 処理を行う実行時システムを使用しており、Lap2d_XMP_MPI は、改良前の MPI のみを用いた実行システムを用いた場合の結果である。図 8.17 に示すように、本実装の Lap2d_XMP_SHM は、16 ノードで、117.7 GFLOPS と、1 ノードの性能 11.26 GFLOPS の 10.5 倍の性能を実現できた。これは、従来の XMP の実装 Lap2d_XMP_MPI が 99.47 GFLOPS に対して 1.18 倍の性能改善である。また、Lap1d_XMP_SHM は、我々の実装で、Laplace の 1 次元分割をそのままマルチノード上で実行した結果である。

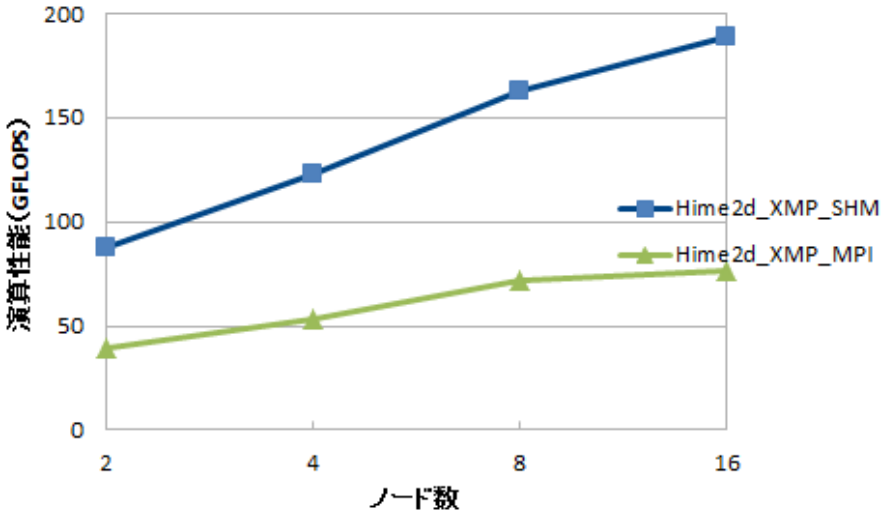

図 8.18: 姫野ベンチマークのマルチノード実行結果

8.3.3 姫野ベンチマーク 2 次元分割の性能評価

本実装の性能を確認するため、姫野ベンチマークについても同様な性能測定を行った。こちらは、ノード単体の測定のときに用いた問題サイズ $L(256 \times 256 \times 512)$ より大きい、$XL(512 \times 512 \times 1024)$ の問題サイズを用いて、2 から 16 にノード数を増加させて測定した。このときのグローバル配列の分割方法は、表 8.5 に、測定結果は図 8.18 に示す。

表 8.5: 姫野 node 分配

ノード数	2	4	8	16
並列数	120	240	480	960
node	n(12,10)	n(10,24)	n(10,48)	n(6,160)
分割方法	div(1,2)	div(2,2)	div(2,4)	div(2,8)

まず、図 8.18 の従来の XMP の実装 Hime2d_XMP_MPI と、本実装 Hime2d_XMP_SHM を比較すると、いずれのノード数でも 2 倍以上の性能向上が得られていることが分かる。Hime2d_XMP_SHM の 16 ノードでの性能は 189 GFLOPS で、Hime2d_XMP_MPI の 76.7 GFLOPS に対して 2.46 倍の性能改善を実現している。一方スケーラビリティをみると、本実装では、2 ノードで 87.3 GFLOPS の性能を実現しており、問題サイズが異なるので単純な比較はできないが、単一ノードで得られた最高性能の 50 GFLOPS に対して 1.75 倍である。16 ノードでの性能は、189 GFLOPS で、8 倍のノード数で演算性能が 2.16 倍である。

本実装上での実行性能向上の理由を確認するため、16 ノードで姫野ベンチ測定時の、1 ノード上での Hime2d_XMP_SHM と Hime2d_XMP_MPI の実行関数の内訳を調べた。図 8.19

に、この結果を示す。まず、姫野ベンチの 1 回の実行時間は、Hime2d_XMP_SHM が 3.46 秒に対して、Hime2d_XMP_MPI が 5.72 秒と、測定を行ったノードでは 1.65 倍優れている。内訳をみてみると、Hime2d_XMP_MPI では、演算を行う jacobi 以外に、MPI の関数を実行する libmpi.so.4.1 と vmlinux で合計 4.6 秒とかなりの時間を占めているのに対して、Hime2d_XMP_SHM ではこれらは 0.5 秒以下に抑えられている。一方、Hime2d_XMP_SHM ではこの他に、XMP の実行時関数が利用している node_barrier が表れているが、これは 1.8 秒しか要していない。この node_barrier は、ノード内の共有メモリを用いて実装した、演算プロセス間のバリア同期を行うもので、袖領域の交換の同期に用いている。このことから、マルチノード実行時に、我々の実装がノード内の MPI 通信をより効率の良い実行時関数に置き換えて、性能を向上させていることがわかる。8.3.2 の Laplace 2 次元分割についても同様の原因で性能が向上していると考えられる。

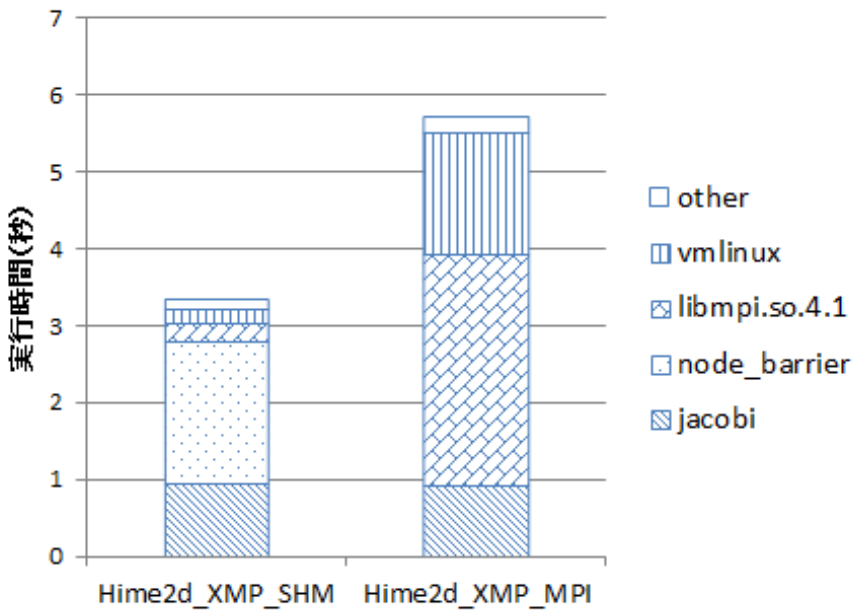

図 8.19: 姫野 16 ノード実行時プロセスの内訳

8.3.4 マルチノード・システム性能評価のまとめ

XMP で記述したプログラムは、そのままマルチノード上でも実行できる。単体での評価と同じベンチマークの並列数を増やしたものを用いて、Xeon Phi を 1 枚実装した Xeon サーバ 16 台を QDR の Infiniband 経由でクラスタ接続したマシン上で、実行時システムを評価した。まず、マルチノード上での Laplace の 2 次元分割の性能測定を、ノード数を 1 から 16 に倍々に変化させて行った。グローバル配列のサイズは、ノード単体での測定と

同じ 10000 × 10000 を用いた。Lap2d_XMP_SHM は、16 ノードで、117.7 GFLOPS と、1 ノードの性能 11.26 GFLOPS の 10.5 倍の性能を実現できた。これは、従来の XMP の実装 Lap2d_XMP_MPI が 99.47GFLOPS に対して 18%の性能改善となる。

姫野ベンチマークについても同様な性能測定を行った。こちらは、ノード単体の測定のときに用いた問題サイズ L(256×256×512) より大きい、XL(512×512×1024) の問題サイズを用いて、2 から 16 にノード数を増加させて測定した。従来の XMP の実装 Hime2d_XMP MPI と、本実装 Hime2d_XMP_SHM を比較すると、いずれのノード数でも、2 倍以上の性能向上が得られていることが分かる。Hime2d_XMP_SHM の 16 ノードでの性能は、189 GFLOPS で、Hime2d_XMP_MPI の 76.7 GFLOPS に対して 2.46 倍の性能改善を実現している。

本実装上での実行性能向上の理由を確認するため、16 ノードで姫野ベンチ測定時の、1 ノード上での Hime2d_XMP_SHM と Hime2d_XMP_MPI の実行関数の内訳を調べた。姫野ベンチの 1 回の実行時間は、Hime2d_XMP_SHM が Hime2d_XMP_MPI に対して 1.65 倍優れている。Hime2d_XMP_MPI では、演算を行う jacobi 以外に、MPI の関数と vmlinux で 80%の時間を占めているのに対して、Hime2d_XMP_SHM ではこれらは 14%以下に抑えられている。このことから、マルチノード実行時に、我々の実装がノード内の MPI 通信をより効率の良い実行時関数に置き換えて、性能を向上させていることがわかる。

第9章　PVASによる実装との比較

XMP のリファレンス実装である omni compiler は、2014 年 11 月にバージョン 20141104 を公開した。この実装では、新たな reflect の処理として、上位方向用／下位方向用の通信バッファを使用せず、MPI の派生データ型を用いて直接分割された配列を参照し、隣接する node の袖領域に内容を転送する実装が追加された。これを用いれば、通信量が、通信バッファを介す場合に比べて半分に減ることが期待できる。この実装での本研究の提案の効果をしらべるため、PVAS を用いた共有メモリの実装でこの効果を確認した。

9.1　バッファを用いない袖領域交換

XMP のコンパイラは、様々な条件での袖領域の処理を実現している。袖領域の隣接した要素は、メモリ上で連続しているとは限らない。そのため図 7.1 に示したように、MPI 通信する際には、必ず通信用バッファに pack してからデータを送受信していた。

しかし、本研究で用いたような 1 次元分割の場合は、袖領域がメモリ上で連続するように XMP で分割を指定することができる。また 2 次元分割では、一つの連続する袖領域を持つ次元の他に、隣りあわせには連続しないものも持つ。しかし、これもメモリ上で一定間隔を保って続いている。このようなデータを送受信する場合に MPI では、飛び飛びに要素を持つ派生データ型使用することができる。

この派生データ型を用いれば、配列の袖領域を直接参照できるため、通信用のバッファを持ちずに直接ノード間でデータの交換を行うことができる。omni compiler のバージョン 20141104 は、このような場合のサポートを追加した。

図 9.1 に、この派生データ型を用いた新しい袖領域の交換方法を示す。図は隣接した 3 つのノードで、1 つの次元の袖領域を交換する方法を示している。中央のノードは、自分の持つグローバル配列の最下位の列を prev_proc の最上位に隣接した袖領域に送り、また、グローバル配列の最上位の列を next_proc の最下位に隣接した袖領域に送る。これらは、派生データ型を用いた MPI 通信で行うことができる。同様に、自分の袖領域は prev_proc と next_proc から受信することができる。

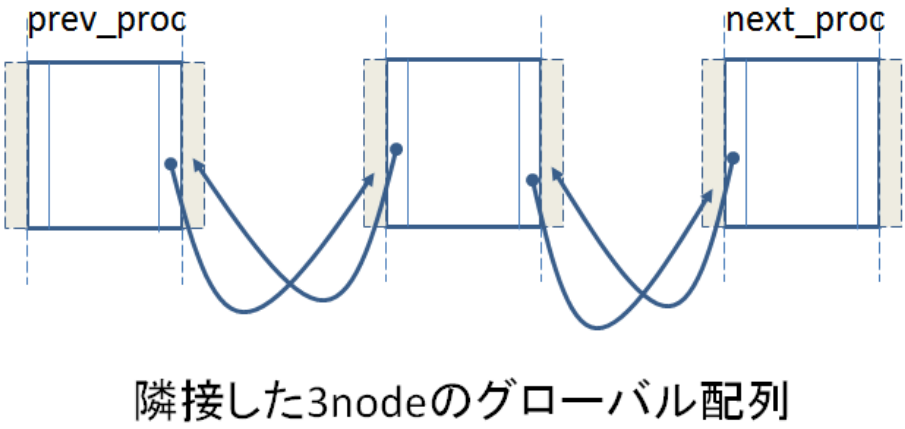

図 9.1: バッファを用いない reflect の実装

9.2 PVAS を用いた XMP の実装

PVAS は理化学研究所計算科学研究機構で開発された、メニーコアを対象としたプロセスモデルである [27]。このモデルでは、PVAS タスクとよばれる複数のプロセスを、同一の PVAS アドレス空間内に生成する。すべての PVAS タスクは同一の仮想アドレス空間に存在するため、特別な共有メモリを準備しなくても、それぞれのプロセスのデータにアクセスすることができる。このため、XMP の生成する MPI プログラムのプロセスに PVAS タスクを用いれば、容易に共有メモリと同様な実装が可能となる。

図 9.2 に PVAS のタスクを、標準的なプロセスと比較して示す。図の左に示すように、Linux 等の OS 上でプロセスを生成する場合、プロセスを構成する TEXT, DATA&BSS, STACK の領域が、生成されたプロセス毎に、図のアドレス空間 (1)、アドレス空間 (2) のように、別々のアドレス空間にとられる。このため、一つのプロセスから、別のプロセスのデータ領域を直接参照することはできない。これに対して、PVAS では、プロセスと同等の領域を持った PVAS タスクを持っているが、この PVAS タスクは単一アドレス空間内に、それぞれのタスクを構成する領域が確保される。このため、一つのタスクから、別のタスクのデータ領域を直接参照することができる。

PVAS は、XMP のように、並列実行にプロセスを生成する処理システムとの相性が良い。プログラム実行時に、PVAS に最適化された OpenMPI[9] を用いて、プロセスの代わりに PVAS タスクを生成するよう変更することができる。そして、そのタスクが利用する実行時システム内の MPI 通信関数を用いて、図 9.1 に示した袖領域の交換を行うことができる。さらに、本研究で提案する共有メモリを用いた実装も、その MPI で行う図 9.1 の送受信の処理を、PVAS 上でのメモリ転送に置き換えるだけで実験し、その効果を確認することが

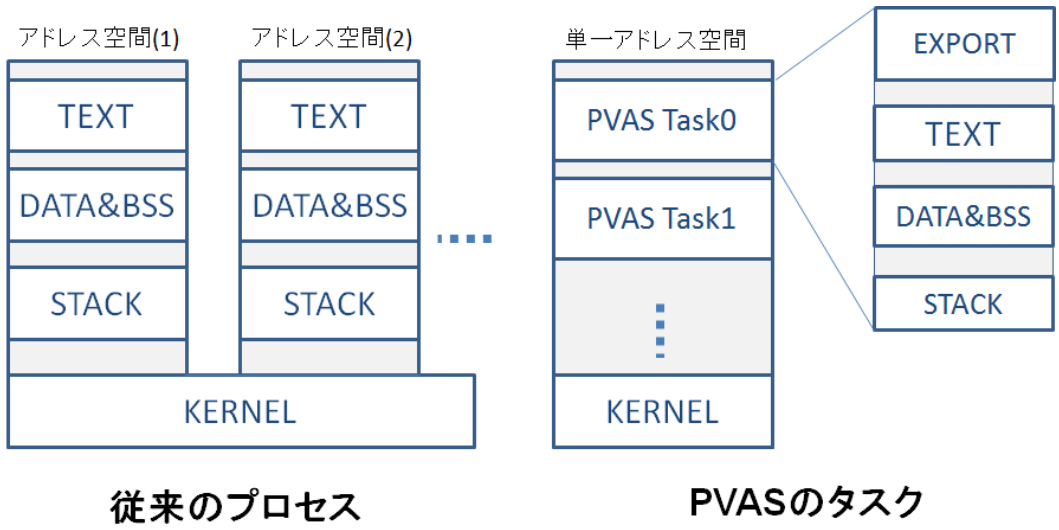

図 9.2: PVAS の実行モデル

できる。そこで、PVAS を用いて、通信バッファを用いない実装での本提案の効果を確認することにした。

ただし、同じ共有メモリを用いた実装でも、PVAS の実装と 7 章で述べた実装では、いくつか異なる点があり注意を要する。例えば PVAS では、用いる仮想アドレス空間の数が少ない。共有メモリを用いた実装では、mmap() を用いて、複数のプロセスの仮想アドレス空間を、共有する一つの物理メモリ領域にマップすることで、共有メモリを実現している。したがって、それぞれのプロセスは仮想アドレス空間を構成するための、ページテーブルを持つ必要があるが、PVAS ではこれは必要ない。

9.3 実験環境

PVAS 上での実験を行った計算機と、この実験に用いたソフトウェアを説明する。まず、単一ノードで用いた計算機は、PVAS を使用するために、MPSS を 2.6.38.8-pvas+mpss3.3 にしたことを除き、表 6.1 とまったく同一のものである。XMP、コンパイラ、MPI 等のソフトウェアは、次に述べるマルチノードの測定で用いたものと同じである。

PVAS 上でのマルチノードの測定実験を行ったシステムの構成を、表 9.1 に示す。8 章までの測定に用いた計算機環境、表 6.1 と細かな点が異なる。ホスト側の計算機は、Ivy Bridge の Xeon であるが、こちらは 2 コア多い、10 コアのプロセッサを 2 個実装したサーバを用いた。OS は同じ Red Hat 社の RHEL 6.5 でメモリは半分の 32 GB を実装している。Xeon Phi のプロセッサは、1.053 GHz で 60 コアのものを使用した。Xeon Phi の OS 等を含む Manycore Platform Software Stack (MPSS) は、3.5 を使用し、Xeon Phi 上では 8 GB のメモリを実装し

ている。

表 9.1: PVAS 実験環境

項目		内容
ホスト	CPU	Xeon 10 コア 2.8 GHz Ivy Bridge×2
	OS	Red Hat EL 6.5
	メモリ	32 GB
メニーコア	CPU	Xeon Phi 60 コア 1.053 GHz×2
	OS	MPSS 3.5 + 3.6.38.8-pvas
	メモリ	8 GB
PGAS 言語		Omni XMP compiler 20141104
コンパイラ		Intel complier version 16.0.109
		OpenMPI 1.8.2

しかも、本実験で環境と 8 章で用いた環境とは、次のような相違があり、両者を直接比較することはできない。まず、XMP のコンパイラのバージョンが異なる。PVAS で使用した 20141104 は、2014 の 11 月にそれまでの build 番号に基づくもの、例えば、表 6.1 での build1322 から、20141104 のような表記に変わったもので、他の性能測定に用いたものより 1 年以上新しい。このため、9.2 節で述べたように、実行時システムの実装にも、いくつか機能が追加されて、改善されている。

また、本測定では、インテル社の MPI を使うことができない。XMP からコンパイルされ、出力された MPI のプログラムを、PVAS 上で実行するには、プロセスではなく、PVAS タスクでプログラムを実行する必要がある。このためには、MPI の実行コマンドの mpiexec を変更する必要がある。そこで、Open Source の MPI、OpenMPI を使用した。MPSS 中の Xeon Phi 用 Linux のカーネルは、PVAS を使用するために、3.6.38.8-pvas に置き換えている。XMP コンパイルには、Intel complier version 16.0.109 と OpenMPI 1.8.2 を使用した。

9.4 メニーコア単一ノード上での評価

PAVS による実装の性能を評価するために、姫野ベンチの L($256 \times 256 \times 512$) を XMP で 2 次元分割し、オリジナルの実装 Hime2d_XMP_MPI と MPI 版の Hime2d_OMP_COL と比

較した。並列数としては、1,2,4,9,16,32,60,120,240 を用いた。図 9.3 に測定結果を示す。この結果、本実装 Hime2d_XMP_PVAS は並列数 120 で、57.1 GFLOPS で、同じ並列数での Hime2d_OMP_COL の 56.9 GFLOPS や Hime2d_XMP_MPI の 54.8 GFLOPS に対して最も高い性能を得られた。

この結果、本実装 XMP_PVAS は並列数 120 で、57.1 GFLOPS で、同じ並列数での OMP_COL の 56.9 GFLOPS や XMP_MPI の 54.8 GFLOPS に対して最も高い性能を得られた。

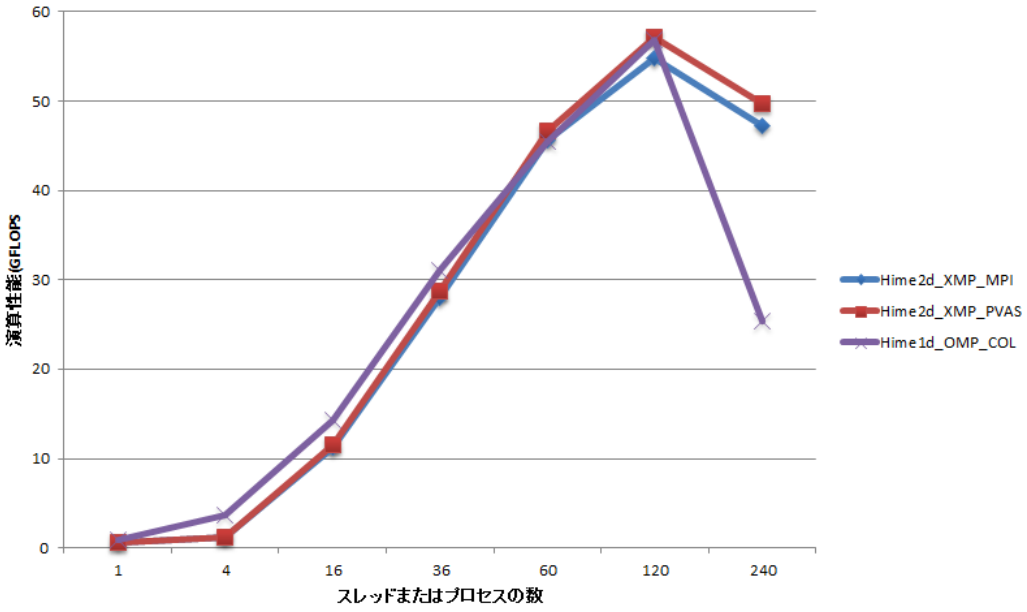

図 9.3: 姫野ベンチ 1 ノードでの実行結果

この図 9.3 の測定結果を、8 章の通信バッファを用いた袖領域の交換での性能測定結果、図 8.13 と比較する。まず XMP の従来実装 Hime2d_XMP_MPI の値が、並列数 60 で比較して 1.5 倍以上と大きく性能を改善した。通信バッファを用いる袖交換では、Pack、Exchange（MPI 通信）と Unpack の 3 回のデータ転送を行っていたが、新しい袖領域の更新では、配列から袖領域への 1 回の MPI 通信で処理が完了する。この性能改善は、データ転送回数が減少した効果だと考えられる。

一方 Hime2d_XMP_PVAS の性能を、通信バッファを使用する図 8.13 の Hime2d_XMP_SHM と比較すると、最も高い性能の得られた並列数 120 で、こちらは 1、2 割程度の性能向上である。こちらは袖交換の Pack と Unpack の 2 回のデータ転送から、PVAS 上の 1 回のデータ転送に、転送回数が減った効果と考えられる。これらの結果から、提案する実装は、1 回の袖領域の転送に要する時間分、PVAS を用いた実装よりも劣るものと思われる。

XMP_PVAS の性能向上の理由を調べるために、並列数 120 と 240 でのベンチマークの実行時間の内訳を、XMP_MPI と比較した。図 9.4 に、その結果を示す。図は、それぞれの計

算や通信を単独で 1000 回ループさせたときの時間を測定している、

XMP_PVAS と XMP_MPI の並列数 120 のときの reflect 通信の時間を比較すると、XMP_PVAS は 60%の実行時間を減少することができている。これは、MPI 通信をメモリ転送に置き換えたためである。

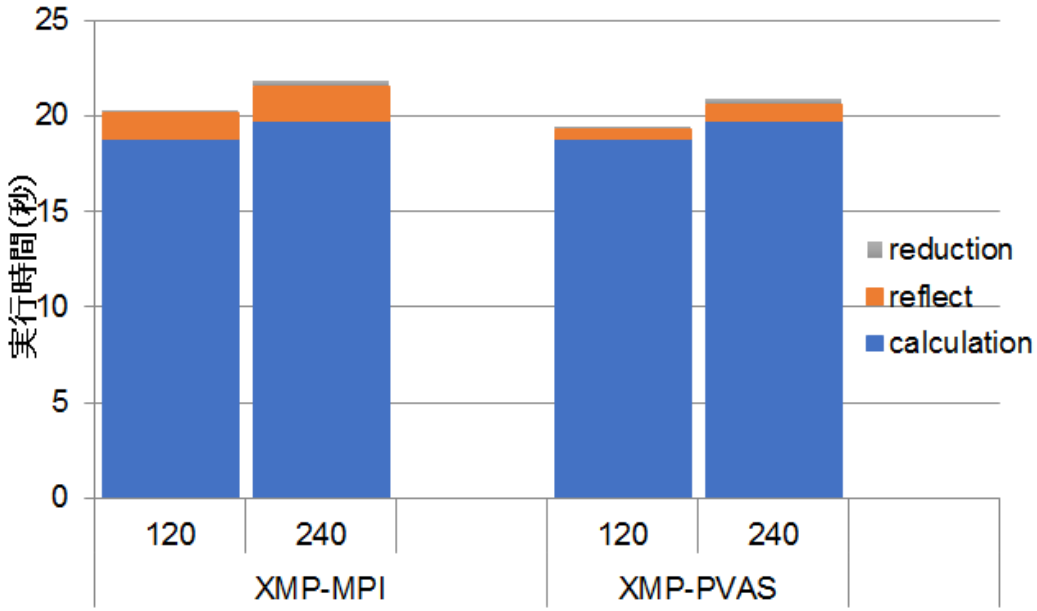

図 9.4: 姫野ベンチマーク実行時間の内訳

8.3 節で行った、提案方式でのマルチノードの評価の条件とはまったく異なるが、姫野ベンチの 2 ノード上での評価を参考までに行ってみた。この評価には単一ノード測定と同じ L の問題サイズを用い、1 ノードと 2 ノードで測定を行った。図 9.5 に、ノード当たりのプロセス数が 30 と 60 で、2 ノードで 60 と 120 のプロセス数での測定結果を、同じ計算機上で、単一ノード上のプロセス数 60 と 120 で行ったものと比較して示す。

XMP_PVAS が、いずれの場合でも、XMP_MPI よりも優れた性能を示しており、メモリ転送で直接行っている reflect 処理の効果がみられる。2 ノードで実行した場合、使用できるコア数が 120 と倍であり、また、使用できるメモリ転送幅も 2 倍となるため、プロセス数が 120 のときは、1 ノードに対して XMP_PVAS で 1.66 倍の性能向上が得られている。

この性能向上の理由を確認するため、1 ノードで 120 プロセスで実行した場合と、2 ノードで 120 プロセスで実行した場合の XMP_PVAS の実行時間の内訳を、比較した。結果を表 9.2 に示す。

やはり演算部分は、倍のコアを使用できる 2 ノードでの実行の方が、11.2 秒と 1.69 倍に高速に処理している。一方 reflect や reduction の処理時間は、ノード間の MPI による通信

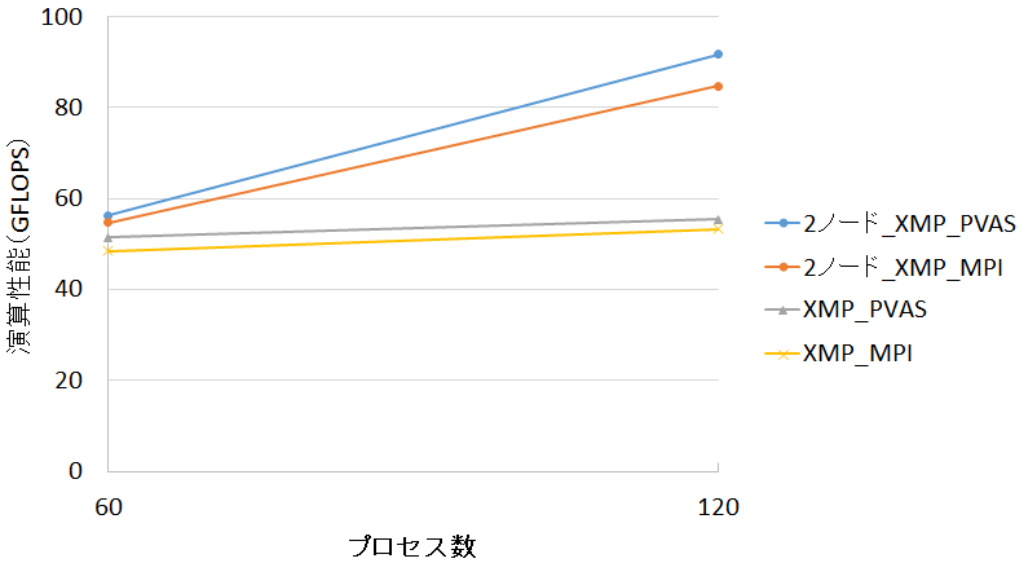

図 9.5: 姫野ベンチ 1 ノードと 2 ノードの比較

表 9.2: 120 プロセス実行時の XMP_PVAS 処理内訳

処理内容	2 ノード実行 (秒)	1 ノード実行 (秒)
演算部分	11.2	18.9
Reflect	0.491	0.580
Reduction	0.113	0.111

を含む 2 ノードでの測定でも、1 ノードの場合と大きな差異はなく、演算時間の違いがそのまま両者の性能差となる。

9.5 PVAS を用いた実装評価のまとめ

単一ノードでは、通信バッファを用いない袖領域での実装でも、共有メモリを用いた実装の方が、MPI を用いた従来実装よりも性能が優れていることが確認できた。これは、共有メモリでの実装の方が reflect の処理に要する時間が少なく、MPI 通信の送受信のプロトコル処理等のオーバーヘッドの無い単純なメモリ転送の実装を用いるメリットである。また、マルチノードでの実験では、2 ノードまでの実行で PVAS を用いた場合、ノード間をまたいで OpenMPI 経由でデータ交換を行っても、ノード内でのデータ交換の処理との差異は大きくないことがわかった。

この方式のマルチノードでの実装は、MPI プロセスの実装に PVAS のタスクを用いているだけで、ノード間通信に MPI を用いていることには差異はない。したがって、提案する

実装方式で、16 ノードまでの性能評価を行った 8.3 節で行った分割と同様な測定が PVAS 版で行える場合、両実装それぞれでのノード内の、ノード間の通信を行う MPI プロセスの数は変わらない。そうすると、両者の多数ノードでの性能は、MPI を介して行うデータ交換の性能に大きく依存する。

つまり、通信バッファを用いない新しい XMP の実装と、その単一ノードでの性能向上は、マルチノードの性能向上に寄与する。しかし、Intel　MPI と OpenMPI の実装上の性能の差異が十分小さいとすると、そのマルチノード性能のスケーラビリティは、図 8.13 に示した提案する実装と同等と考えられる。

PVAS 版では、そのタスクを用いることにより、容易に共有メモリを用いた実行時システムを開発できる。しかし、PVAS を用いるには Xeon Phi で boot する OS を PVAS 用のものに置き換える必要があった。大型のクラスタシステムは計算機センタ等で多くのユーザでの共用することも多く、実験のためだけに OS を標準でないものにするのには、様々な手間が必要となる。検討している計算方式と直接関連しないが、この点が改善されれば、より PVAS を使いやすくなる。

第10章　結論

10.1　まとめ

ステンシル計算のようなデータ並列処理をサポートするXMPの機能を、新たに開発されつつあるメニーコア・クラスタのような、プロセッサ内コア間と計算ノード間の両者で大きな並列度を持つような計算機システム上で、効率的に実現する実行時システムの設計を行った。

この方法では、コア間で共有メモリの使用できるメニーコアプロセッサ内で、グローバル配列をコアのL2キャッシュを効率よく使用できるように分散配置する。袖領域のデータの交換には、メニーコアプロセッサ内では共有メモリを用いてmemcpyでデータ転送を行い、別のプロセッサと通信する必要があるときのみMPIを用いてデータ交換を行う。

この結果、共有メモリを用いた実装は、単一ノード性能は従来のXMPの実行時システムに対して、Laplaceの1次元分割で4.62%、姫野ベンチマークで62.9%の性能向上を、またOpenMPで並列化したプログラムに対しても、それぞれ16.8%と13.8%の性能向上を実現することができた。また、マルチノード性能では、従来のXMPの実行時システムに対して16ノードの実行でLaplaceの2次元分割で18%、姫野ベンチマークでは2.46倍の性能向上を実現できた。

また、通信バッファを用いない新しい袖領域通信を行うXMPの場合でも、PVASを利用して本提案の実装と同等のmemcpyを従来のMPI通信に置き換える手法が有効なことが、メニーコアプロセッサ単体や2ノードのマルチノードの実行で確認できた。

以上のことから、本提案のステンシル並列計算の、メニーコアプロセッサ向けの性能改善の手法が有効であることが、実証できた。

10.2 今後の課題

本実装にはマルチノード性能のスケーラビリティに課題がある。原因は、ノード内の XMP node の実装方法と MPI 通信の利用方法にあると考えている。この node の実装にはプロセスが用いられ、袖領域交換のプロセス間での MPI 通信を止めたにもかかわらず、全プロセスが MPI の ID を持っている。このため、MPI のランク数が XMP の全 node 数と同じになり、必要以上に大きい。これは、通信するプロセスのみを MPI 通信用に使用し、他は単純に子プロセスとするか、別の並列化の手段を用いることで改善できる。また、マルチノード間の MPI 通信については、同じノード間でも、通信する必要のあるノードどうしで別々に１対１の通信をしている。これは、MPI 版の使用している隣接間の持続通信に置き換えるか、通信をまとめる等で改善できると考えている。これらの実装は今後の課題である。

メニーコアプロセッサは、プロセッサ内に高い並列性があるために、並列演算を行うときに、プロセッサ内であっても、均一的な並列処理を行うことだけで十分な性能を得ることはできない。メニーコアプロセッサ内の多数のコアは、プロセッサのパッケージ内にあり、プロセッサに直接メモリが接続される現在のプロセッサのアーキテクチャを前提に考えると、これらのコアが物理的に同じメモリを共有することは、自然に思える。しかし、本研究で取り上げた Xeon Phi のように 60 を越えるコア数では、ラストレベル・キャッシュをすべてのコアで共有することは、性能上の観点からは難しくなり、それぞれのコアはローカルなキャッシュを持つ必要がある。したがって、メニーコアプロセッサでは、プロセッサ内での局所性を考慮して並列プログラミングを行う必要が生じる。

一方、近年の高性能計算機システムの主流は、マルチプロセッサのクラスタシステムである。これらに有効なプログラミング手法として、共有メモリを利用できるノード内では、OpenMP のような共有メモリを前提とした並列プログラミング手法を用い、分散メモリであるクラスタ内のノード間では MPI のような分散メモリを考慮してつくられた並列プログラミング手法を用いる、ハイブリッドによるプログラミングが広く行われるようになった。このハイブリッドは、対象とするプログラム上の並列計算が均一なものであっても、共有メモリと分散メモリのそれぞれのプログラミングにあわせるようにプログラムを分割して再構成する必要があり、プログラミングが煩わしい。

ここでとりあげた、メニーコア向けに行う並列計算での、プロセッサ内での局所性考慮の必要性は、メニーコアプロセッサでのクラスタシステム上でのプログラミングを行う場合に、このハイブリッドに、さらに新たなハードウェアに依存する並列プログラミングの導入を示唆するものであり、それを加えてプログラマに要請するのはプログラミングの生

産性の観点から好ましくない。

ところで、マルチプロセッサのクラスタシステムに良く用いられる、ハイブリッドによるプログラミングの煩雑性を軽減しながらも、データの局所性をプログラマが利用できるようにして、システムの性能を十分に活かせる言語として、XMP のような PGAS 言語の利用が注目されている。この言語では、システムで実行する並列計算をグローバルな視点で記述でき、かつ、演算やデータの分割や局所的なデータへの指示をプログラマが行うことができる。したがって、ステンシル並列計算のようにシステム全体で均一的な並列処理を行う場合、プログラマは実行するハードウェアに合わせて、その処理を再構成することから解放される。

メニーコアプロセッサのクラスタで新たに加わるプロセッサ内の局所性の利用についても、PGAS 言語の局所データへの指示で対応できると考えられ、本研究で取り上げた言語の実行時システムの他、必要に応じて言語仕様の拡張を行えば、十分対応可能で、PGAS のグローバルなプログラミングモデルを用いた平易なプログラムで、メニーコアプロセッサのクラスタ上で高い性能が得られるようになるものと、期待できる。

本研究の対象はメニーコアプロセッサであった。科学技術計算用の高性能計算へのニーズは限りが無く、性能電力比の大きな向上はこれからも求められ続けると思われる。このような、ハードウェアへの進化の要求の一端を、現在のような半導体技術の進化、すなわち、トランジスタ数の増加により実現させていくものと考えると、求められる性能向上を実現するには、行う科学技術計算を、利用できるトランジスタに効率よくマップするための最適化問題を解く必要がある。

ところが、既に開発されて広く用いられている科学技術計算用のプログラムの多くは、現在主流の汎用プロセッサとメモリの組み合わせを前提として記述されている。これらの資産を活かしつつ、継続的にプログラムの性能向上を行うには、やはり現在の高性能計算機が採用しているような、コアの多数化、SIMD 幅の拡張やノード数の増加といった処理の高並列化と、データの局所性を活かすための階層的メモリの有効利用等を、さらに発展させた計算機を用いるのが最も近道に思われる。XMP のような、PGAS 言語による並列性や局所性を利用するプログラミングは、このような今後開発される高性能計算機システムにも適用させやすく、今後のさらなる応用が期待される。

謝辞

本研究を行う機会を与えてくださり、終始あたたかいご指導と激励を賜りました、筑波大学大学院システム情報工学研究科 佐藤三久教授に心から感謝します。本研究への貴重なご意見を頂いた筑波大学大学院システム情報工学研究科朴 泰祐教授、櫻井鉄也教授、前田敦司准教授　と東京大学 情報基盤センタ塙 敏博准教授に感謝します。

本研究のため、在職中にも係わらず、筑波大学大学院への入学を許可して頂いた、インテル株式会社の関係者の方々に感謝します。特に入学時にお世話になった加藤仁さんに感謝します。

また、本研究にご協力いただいた、筑波大学大学院システム情報工学研究科 HPCS 研究室の皆様、特に PVAS データの測定等にご協力いただいた、大川千聡さんに感謝します。Xeon Phi 用の XMP の処理系については、理化学研究所計算科学研究機構の中尾昌広さんにご協力を頂きました。感謝します。本研究について有益なご意見を頂いた、インテル株式会社の同僚、堀越将司さん、小林広一さん、Michael Mccool さんに感謝します。

参考文献

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 353–362, New York, NY, USA, 1998. ACM.

[2] OpenMP ARB. The OpenMP® API specification for parallel programming. `http://openmp.org/wp/`.

[3] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[4] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[5] Cristian Coarfa. *Portable High Performance and Scalability of Partitioned Global Address Space Languages*. PhD thesis, Rice Univ., 2007.

[6] D. E. Culler, K. Keeton, L. T. Liu, A.Mainwaring, R.Martin, S. Rodrigues, and K. Wright. Generic Active Message Interface Specification v1.1. U.C. Berkeley Computer Science Technical Report, 2 1995.

[7] Dan Bonachea and Jaein Jeong. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. CS258 Parallel Computer Architecture Project, 2002.

[8] Frederica Darema. SPMD Computational Model. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1933–1943. Springer US, 2011.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] G.Crimia, F.Mantovanib, M.Pivantid, S.F.Schifanoe, and R.Tripiccionec. Early experience on porting and running a Lattice Boltzmann code on the Xeon-Phi co-processor. *ICCS*, pages 551 – 560, 2013.

[11] William George, S. Lennart Johnsson, and Alan Ruttenberg. A stencil compiler for the connection machine models cm-2/200. In *TR-22-93*, pages 1–8, 11 1993.

[12] Mitsuru Ikei and Mitsuhisa Sato. A PGAS Execution Model for Efficient Stencil Computation on Many-Core Processors. *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing(CCGrid)*, pages 305–314, May 2014.

[13] Intel. Intel Xeon Phi Coprocessor(codename: Knights Corner) Performance Monitoring Units. https://software.intel.com/sites/default/files/forum/278102/intelr-xeon-phitm-pmu-rev1.01.pdf.

[14] Intel. インテル C++ Composer XE 2013 Linux 版. http://www.xlsoft.com/jp/products/intel/compilers/ccl/index.html.

[15] Intel. インテル mpi ライブラリー 4.1. http://www.xlsoft.com/jp/products/intel/cluster/mpi/index.html.

[16] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 9 2015. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[17] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005.

[18] Katherine Yelick et. al. Productivity and performance using partitioned global address space languages. PASCO '07, Proceedings of the 2007 international workshop on Parallel symbolic computation, 2007.

[19] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, June 2007.

[20] Jinpil Lee and M. Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 413–420, Sept 2010.

[21] Miao Luo, Mingzhe Li, Akshay Venkatesh, Xiaoyi Lu, and Dhabaleswar K DK Panda. Upc on mic: Early experiences with native and symmetric modes. *7th International Conference on PGAS Programming Models*, page 198, 10 2013. http://www.pgas2013.org.uk/sites/default/files/finalpapers/Day2/R5/2_paper30.pdf.

[22] Jose E. Moreira. *Computational Science — ICCS 2001: International Conference San Francisco, CA, USA, May 28–30, 2001 Proceedings, Part I*, chapter Blue Gene: A Massively Parallel System, pages 10–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[23] MPI-Forum. Message Passing Interface Forum Home Page. http://www.mpi-forum.org/.

[24] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[25] Karthik Raman. Optimizing memory bandwidth on stream triad, 2 2013. http://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad.

[26] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *Proceedings of ACM/IEEE Conference on Supercomputing*, SC '97, pages 1–20, New York, NY, USA, 1997. ACM.

[27] Akio Shimada, Balazs Gerofi, Atsushi Hori, and Yutaka Ishikawa. Proposing a new task model towards many-core architecture. In *Proceedings of the First International Workshop*

on Many-core Embedded Systems, MES '13, pages 45–48, New York, NY, USA, 2013. ACM.

[28] Erich Strohmaier, Jack Dongarra, Horst Simon, Martin Meuer, and Hans Meuer. The TOP500 list. http://www.top500.org/.

[29] Y. Tang, R.A. Chowdhury, B.C. Kuszmaul, C.-K. Luk, and C.E. Leiserson. The pochoir stencil compiler. In *In Proceedings of ACM symposium on Parallelism in algorithms and architectures*, pages 117–128, 2011.

[30] University of Tsukuba HPCS Lab and RIKEN AICS Programming Environment Research Team. Omni XcalableMP Compiler, 11 2012. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/download.html.

[31] UPC Consortium. UPC Language Specifications, Version 1.3, 11 2013.

[32] XcalableMP Specification Working Group. XcalableMP Specification Version 1.1, 11 2012. http://www.xcalablemp.org/spec/xmp-spec-1.1.pdf.

[33] XcalableMP Specification Working Group. XcalableMP Specification Version 1.2.1, 11 2014.

[34] Yang You, Haohuan Fu, Xiaomeng Huang, Guojie Song, Lin Gan, Wenjian Yu, and Guangwen Yang. Accelerating the 3d elastic wave forward modeling on gpu and mic. *IPDPSW*, pages 1088–1096, May 2013.

[35] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. A Tagless Coherence Directory. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 423–434, New York, NY, USA, 2009. ACM.

[36] 小林広和 and 中田真秀. スレッド間空間的ブロッキングを利用した Xeon Phi 上の姫野ベンチマークの最適化. **情報処理学会研究報告**, 2013-ARC-207(4):1–8, dec 2013.

[37] 大川 千聡, 堀 敦史, 島田 明男, 池井 満, and 佐藤 三久. メニーコアプロセッサ向けプロセスモデル pvas の pgas 言語実行時ライブラリの設計. **情報処理学会研究報告**, 2015-HPC-148(22):1–6, 2015.

[38] 池井 満 and 佐藤 三久. PGAS 言語 XcalableMP のメニーコアプロセッサ・クラスタ向け実行モデルの検討. **情報処理学会研究報告**, 2013-HPC-140(30):1–8, 2013.

[39] 池井 満 and 佐藤 三久. 効率的なデータ並列ステンシル計算のためのマルチノード・メニーコアプロセッサ向け PGAS 言語実行時システムの設計. **電子情報通信学会論文誌** *(D)*, J99-D(2):138–151, February 2016.

[40] 池井 満, 中尾 昌広, and 佐藤 三久. メニーコアプロセッサ・クラスタにおける並列プログラミング言語 xcalablemp のベンチマークプログラム性能評価. **情報処理学会研究報告**, 2013-HPC-138(28):1–5, 2013.

[41] 理化学研究所. 姫野ベンチマーク. `http://accc.riken.jp/supercom/himenobmt/`.

論文リスト

査読付き参考論文

[1] Mitsuru Ikei and Mitsuhisa Sato. A PGAS Execution Model for Efficient Stencil Computation on Many-Core Processors. *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing(CCGrid)*, pages 305–314, May 2014.

[2] 池井 満 and 佐藤 三久. 効率的なデータ並列ステンシル計算のためのマルチノード・メニーコアプロセッサ向け PGAS 言語実行時システムの設計. **電子情報通信学会論文誌** *(D)*, J99-D(2):138–151, February 2016.

その他の論文

[3] 池井 満, 中尾 昌広, and 佐藤 三久. メニーコアプロセッサ・クラスタにおける並列プログラミング言語 XcalableMP のベンチマークプログラム性能評価. **情報処理学会研究報告**, 2013-HPC-138(28):1–5, 2013.

[4] 池井 満 and 佐藤 三久. PGAS 言語 XcalableMP のメニーコアプロセッサ・クラスタ向け実行モデルの検討. **情報処理学会研究報告**, 2013-HPC-140(30):1–8, 2013.