Vol.2013-HPC-140 No.36
2013/8/2

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

池井 満[1,2]　佐藤 三久[2]

概要：我々はPGAS言語XMPの、メニーコアプロセッサ・クラスタ向けの実装の検討を行っている。対象のメニーコアプロセッサ・クラスタは、Intel Xeonサーバにメニーコアプロセッサ Xeon Phiのカードを搭載した計算ノード複数を、スイッチ接続したものである。このようなクラスタでプログラムを実行するには、共有メモリを持つ、Phi単体内の60+コアでプログラムを実行する際の並列性と、物理的に独立したメモリ空間を持つ、複数ノードのPhi間でプログラムを実行する際の並列性の、2種類の並列性を同時に生かす必要がある。本発表では、XMPでのプログラミングモデルの性質を利用して、このような2階層の並列性を有効に実現するために、そのプログラム実行モデルの検討を行った。

1. はじめに

インテル社から、60個以上のコアを持つ、メニーコアのプロセッサXeon Phiが発売された [1]。このXeon Phiは512ｋＢの2次キャッシュをもつ演算コア60個で最大16GBのメモリ空間を共用している。また、コアへのデータ供給はクロックあたり128バイト（64バイト×双方向)）の高バンド幅の共用リングバスで行われる。このようなアーキテクチャでは、メモリ転送バンド幅は大きいものの、メモリアクセスのレイテンシが大きい。このため、ローカルキャッシュへのデータの配給のタイミングとその有効利用と、フォールス・シェアリング等に起因する不要なキャッシュ間の整合性を保つためのデータ転送の抑制、が重要であると考えた。そこで我々は、XMPを用いてプログラムを記述し、そのコード生成部により、MPIを用いたプログラムを生成することより、Phiのようなメニーコアの SMP上で、MPIプロセスを用いて、コアごとに演算を行うためのメモリ空間を分離して並列化する試みを、姫野ベンチマークを用いて行った [2]。

この結果、単一Phiノードを用いた、Phiのノード内だけでの並列実行では、OpenMP版より優れた性能を実現でき、コア数の2倍強である、128プロセスまで性能がスケールすることを確認した。しかし、この方法の問題点は、多数のPhiカードを用いた場合のスケーラビリティにあった。例えば、16個のXeonクラスタに1枚ずつ挿入された

```
1 #pragma xmp loop (x) on t(x)
2 #pragma omp parallel for
3     for(x = 1; x < XSIZE−1; x++)
4       for(y = 1; y < YSIZE−1; y++)
5   u[x][y] = (uu[x−1][y] + uu[x+1][y] + uu[x][y−1] +
          uu[x][y+1])/4.0;
```

図 1　XMP と OpenMP のハイブリッド例

16枚のPhiカードを用いて、2048プロセスで実行すると、カード内、カード間のMPI通信量が増え、性能が頭打ちになることにあった。そこで前報では、図 **1** に示すようにXMPプログラムの並列化ループの内側にOpenMPディレクティブを挿入してPhiクラスタのスケーラビリティを得た。図1の1行目の #pragma xmp loop (x) on t(x) がXMPの並列化の指示であり、3行目のxをインデックスとしたループをMPIを用いたプロセス並列化を行う指示である。2行目が、同じ3行目のループをOpenMPのスレッドで並列化する指示である。*1 この結果、ノード内はOpenMPを用いて並列化し、16個のPhiカード間での通信にのみXMPプログラムから生成されたMPIを用いたる実行方法で実行でき、16個までのスケーラビリティを得られることを確認した。残念ながら、このようにシステムの並列性の階層に合わせて複数種類の並列化を行うと、プログラミングの生産性が低下してしまう。

そこで本報告では、このようなXMPに加えてOpenMPを用いるハイブリッドな手法を用いいなくても、用いた場合

1　インテル株式会社
Intel K.K.
2　筑波大学
University of Tsukuba

*1　XMP言語上でのこのようなハイブリッドの利用は、まだ規定されていない。

ⓒ 2013 Information Processing Society of Japan

と同様なスケーラビリティを得ることができる Phi 用のランタイムが実現できないか、その検討を行った。その方法として、最初に述べたように、プロセスを用いて Phi のコア間でローカル変数を保持するアドレス空間を独立して持たせ、カード内ではプロセス間の共有メモリを用いて直接コピーするようにした。データの交換がカード間をまたぐ場合に限り MPI 通信を行うような XMP のランタイムを作成してその評価を行った。

次章ではまず、XMP の概要について説明する。続けて以下は、3 章で Phi の特徴、4 章で改良した Xeon Phi 向け XMP のランタイム、5 章で性能測定、6 章で結果の考察の順に述べ、最後に、7 章で結論と今後の課題について述べる。

2. XcalableMP の概要

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 をサポートしており、PC クラスタコンソーシアムの XcalableMP 規格部会で議論して、策定された仕様である [3]。本報告では C 版を使用したので、以下 C の記述を用いる。また、XMP を用いたプログラムの並列化には、同一プログラムを複数のデータに対して実行させる SPMD モデルを用いる。これらは OpenMP を用いた並列化と XMP との共通の特徴である。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。本稿ではグローバルビューを用いたので、以下グローバルビューについてのみ説明する。

2.1 データ分割の方法

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 **2** 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。

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 node を定義している。次に template 指示文で、仮想的な配列 t(0:99) を宣言している。これは、データマッピングのための指示文で、グローバルな配列やその分割方法を指示するためのテンプレートとして t(0:99) を宣言している。3 行目の distribute 指示文で仮想配列 t をどのように P で宣言された 4 つのノードに分配するかを指定している。t(block) で、template t の 100 個の要素を 4 つの 25 要素のかたまり（block）に分割する指定をしている。

```
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  #pragma xmp shadow g[1][0]
9
10 double l[40][50];
```

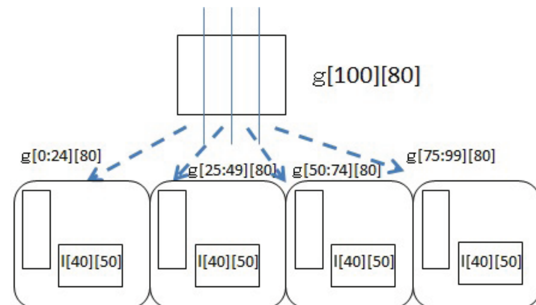

図 **2** XMP による配列の分割例

次にプログラム本文中 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。この結果、g はグローバルな配列となり、80 × 25 の大きさの 4 つに分割されて、4 つのノードに分配される。8 行目の shadow 指示文については次節で説明する。

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 2 に示したように、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。

2.2 shadow による袖領域の作成

データ分割の境界部分をノード間で重複して持つデータマッピング（shadow）とその重複部分（袖領域）の通信、について説明する。図 2 ではグローバルビューにおいて、配列 g[100][80] が 1 次元目で 4 つに分割することを示した。このとき、8 行目の shadow 指示文はグローバルな配列を分配するときの、境界領域の重複保持部分（袖領域と称される）の場所と大きさを指示している。この指示文では、g[1][0] を指定して、グローバル配列 g の 1 次元目で 1 個幅

ⓒ 2013 Information Processing Society of Japan

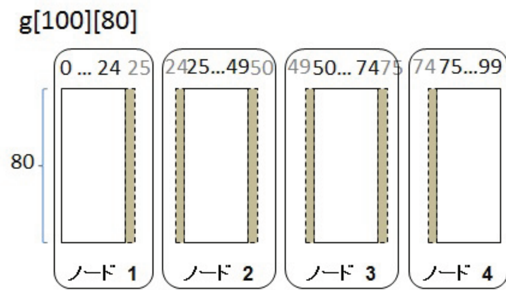

図 3　shadow による配列 g[100][80] の分割袖領域

の袖領域を持って分割することを指示している。図 **3** にこの場合のグローバル配列の分割方法を拡大して示した。

左端のノード 1 は配列の g[0:24][80] と g の 25 列の演算を行うが、これに加えて、ノード 2 の最初の列 g[25][80] の要素を持つ。この部分が参照は行うが、演算を行わない袖領域（図中の灰色部分）となる。この結果、ノード 1 の持つグローバル配列の要素数は計 26 列となる。中間の 2 つのノード、ノード 2 とノード 3 はそれぞれ両側のノードの最端の要素を袖領域として持ち、25 列の担当の要素に加えて 2 つの袖領域を持つ。その結果、それぞれの持つ領域は g[24:50][80] と g[49:75][80] となり、計 27 列の要素を持つ。ノード 4 は g[74:99][80] とノード 1 と同様に 26 列持つこととなる。

それぞれのノードは担当の配列要素の演算は行うが、袖領域の演算は行わない。したがって、袖領域の値は、演算を行う隣のノードから受け取る必要がある。XMP では、このデータ交換を行うタイミングを指定する為に、reflect 文を持っている。

本報告では、XMP として、筑波大学と AICS が共同で開発している Omni XcalableMP Compiler 1.1 (build1322) [4] を用いた。XMP を用いて記述されたプログラムは、SPMD モデルの MPI を用いた並列プログラムに変換される。すなわち、複数個実行される並列プログラムは、それぞれプロセスとして自分のアドレス空間に分配された部分配列とその袖部分を持つ。そして、refelect 文を実行したときに、MPI の通信を用いて、プロセス間でデータ交換する。

3. インテル Xeon Phi の特徴

3.1　アーキテクチャ

インテル Xeon Phi は、高速グラフィックカード等と同様な PCI Express のプロセッサボードである。図 4 に Phi の内部構成図を示す。Phi は 3 種類の要素がリングバスで接続されて構成されている。最初の要素は CPU コアであり、コアとキャッシュから構成される。今回用いた Phi では、61 個の CPU コアがリングバスに接続されている。2 番目の要素はメモリで、これは、2 チャネルの GDDR5 のメモリコントローラが 8 個リングバスに接続されている。3 つ目の要素は PCI-Express のコントローラで、メモリアクセスを除くすべての入出力は、この PCI-Express のバスを介して外部の機器と行われる。

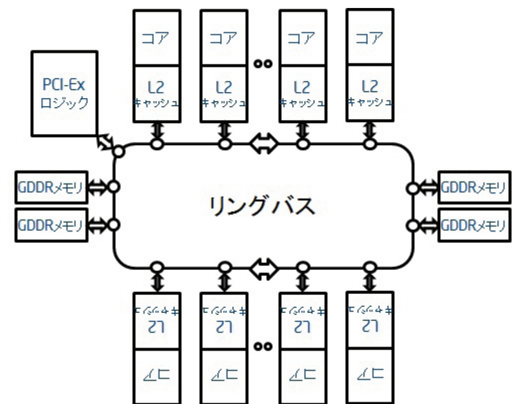

図 4　Phi の内部構成図

3.2　Phi のプログラミングモデル

Phi と GPGPU の大きな相違点として、Phi では汎用プロセッサ用のプログラミングツールをそのまま使えることが挙げられる。したがって、大きな利点として、既存プログラムを変更することなく、コンパイルしただけで、そのまま Phi 上で実行できることが挙げられる。しかし、この実行ができる状態から、実際に Phi 上で十分な性能向上を得られるようにするには無視できない量の作業が必要となる。

3.3　プログラムの実行環境

Phi 用のアプリケーションの起動（実行）方法としては、大きく 2 つの方法が使用できる。アプリケーションの一部だけをコプロセッサに任せる Off-load モデルと、アプリケーション全体をコプロセッサ側で実行する Native モデルである。Off-load モデルでは、プログラムの実行は、主プロセッサ側から開始される。そして、主プロセッサが、コプロセッサ側で実行したい部分に到達したときに、コプロセッサに必要なデータを転送し、実行の制御を渡す。その後の処理はコプロセッサ側で実行われ、終了した結果を、主プロセッサに戻し、制御も主プロセッサに返す。GPGPU 等のアクセラレータで一般的に用いられるモデルである。このモデルでは、高速処理の必要な部分のみ明示的に指定して効率的にコプロセッサを利用することが、期待できる。しかし、アプリケーション中の off-load 部分を指定する為に、必ず、プログラムの変更が必要となる。

一方 Phi の場合は、この他に、Native モードでプログラムを実行することができる。Native モードでは、アプリケーションの全体がコプロセッサで実行される。したがって、プログラム全体を、コプロセッサ用にコンパイルして、これをそのままコプロセッサに送り、コプロセッサのメモリ上で実行する。このモデルを用いる場合、OpenMP 等を

ⓒ 2013 Information Processing Society of Japan

用いて、既に並列化されたプログラムを用いる場合は、コプロセッサ上で実行するためのプログラムの変更は必要ではない。本稿で用いるのは Native モデルである。

4. Phi 向け XMP 実行ランタイムの改良

本報告で用いた Omni XcalableMP Compiler 1.1 (build1322) の x86 プロセッサ用のコンパイラは、XMP で記述したプログラムから、MPI を挿入した、プロセス単位で実行される SPMD の並列プログラムを生成することができる。前報では、この生成されたプログラムに対して Phi 用のインテル MPI を適用することにより実行バイナリを生成して Phi のクラスタ上で実行した。このため、2000 並列程度で Phi カード内と Phi カード間の MPI メッセージ通信数量の多さから実行性能が劣化した。そこで本報告では、Phi カード内では、MPI を用いず、プロセス間の共有メモリ上にグローバル配列をとり、直接データ転送を行う方法を検討した。Phi カード間の MPI 通信は、まとめて行えば通信回数はコア数ではなく、Phi カード数に比例した数に減らせるものと考えた。

4.1 Phi カード内のグローバル配列の配置

Phi カード内でのグローバル配列の実装は、Linux の共有メモリを用いて行うことにした。図 **5** に Phi カード内の変数配置の概念図を示す。まず、Phi カード内の並列化の方法としてはプロセスを用いることにした。図 5 に示すように、並列実行する P1,P2,P3... のプロセスはそれぞれ独立した自分のアドレス空間に XMP プログラム中のローカル変数を持つ。グローバル配列は、共有メモリ上に図 5 に示すように、灰色で示した各プロセスの袖部分を追加して割り当てることにした。図 5 は 1 つの shadow（袖）を持つ XMP のプログラムの 1 次元分割を利用した場合で、各プロセスが配列の n 列ずつを持つ例を記述している。プロセス P1 は 0 から n-1 までの、担当する列に加えて袖部分の第 n 列をプロセス間の共有メモリ上に持つ。プロセス P2 は n から 2n-1 までの担当列に加えて、担当部分の前の袖である、n-1 の列と、後ろの袖の 2n の列を持っている。これらの列を共有メモリ上で、連続して割り当てることにした。図 5 の例の場合、プロセス P1 と P2 の境界では P1 の担当列に連続して P1 の袖、第 n 列を追加し、この第 n 列の次に P2 の前の袖、n-1 列を配置する。P2 の担当の列は、ここから連続して割り当てられる。このように、グローバル配列は、並列実行するための分割境界で重複する袖領域を追加して、割り当てる。この結果、図 5 の場合は、(分割数-1) の 2 倍の列の分だけグローバル配列が増加する。

4.2 袖領域のデータの交換

XMP プログラム中で reflect 文が実行された場合、各プロセスが持つ袖領域の値は、その袖領域を担当している別

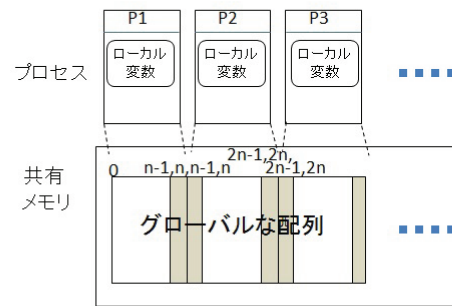

図 **5** XMP プロセスの変数配置

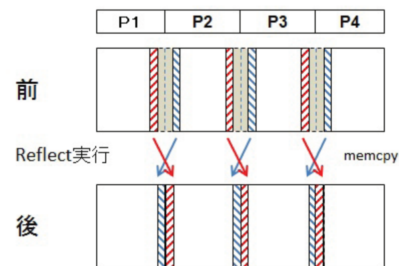

図 **6** Phi カード内の reflect の実行

プロセスの値に更新しなければならない。本報告のように、複数の Phi カードによるクラスタ上で XMP プログラムを実行するとき、袖領域のデータの更新には 2 つの場合がある。プロセスがプログラムを実行する際に、(1) 必要とする袖領域を持っているプロセスが同じ Phi カード内に存在する場合と、(2) 別の Phi カード内のプロセスが持っている場合である。(1) の場合の reflect の実行前後の配列データの内容を図 **6** に示す。

1 次元分割で、1 個の袖領域を持つ P1 から P4 の 4 つのプロセスの袖領域のデータの交換を図示している。例えばプロセス P2 に注目する。P2 は reflect 前の袖領域（自分の担当の配列の両側にある灰色部分）を更新する為に、P1 の右端と P3 の左端のデータが必要となる。本報告の方式では、グローバル配列を共有メモリに置いたので、P2 からグローバル配列のすべての領域を参照できる。従って、これらは memcpy で実装した。

別の Phi カードのプロセスが袖領域の更新データを持つ (2) の場合は (1) とは事情が異なる。本報告で扱う、1 次元ブロック分割の場合は、あるプロセスの袖領域は、必ずグローバル配列の隣接する部分を持つ、隣接するプロセスが持っている。このため、別の Phi カードとデータの交換をする必要があるのは、カード内の最初か最後のグローバル配列の部分を持つプロセスとなる。図 **7** にこのような場合の reflect の実行前後の配列データの内容を示す。

図中の Phi カード 1 の最後のプロセス Pn が持つ、最右端の袖領域（灰色）部分は、別の Phi カード 2 のプロセス P1 が持っている。したがって、プロセス Pn は別の Phi カードの P1 から更新データを受け取る必要がある。本報告の方式では、これらの Phi カード間をまたがるデータの

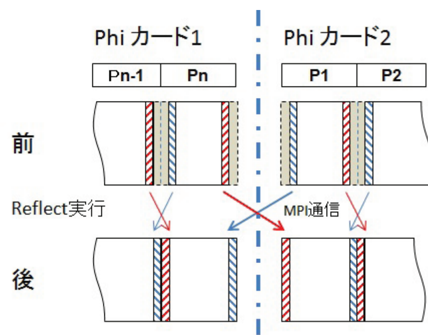

図 7　Phi カード間の reflect の実行

表 1　サーバの仕様

項目	Xeon	Xeon Phi
CPU 周波数	2.6 GHz	1.1GHz
ベクトル長	4	8
コア当たり倍精度演算ピーク	20.8GFLOPS	17.6GFLOPS
コア当たりスカラ倍精度演算性能	5.2GFLOPS	2.2GFLOPS
コア数	8	61
システム倍精度演算ピーク	332GFLOPS	1073GFLOPS
コア当たり最大キャッシュ容量	20MB	512KB
コア当たりのスレッド数	2	4
メモリ周波数	1600MHz	2750MHz
実装メモリ	32GB	8GB
システムメモリ転送ピーク性能*1	102.4GB/s	350GB/s

交換は MPI 通信を用いて行うこととした。この結果、1 回の reflect 文で実行される MPI 通信は、Phi カード枚数-1 程度になり、すべての袖領域の更新を MPI で行う方法に比べて、大きく MPI 通信量を減らすことができる。

5. 性能評価

5.1 実験環境

Phi は PCI Express カードであり、単独では動作しない。本研究では表 **1** に示す仕様の Xeon サーバを測定に用いた。

消費電力当たりの演算性能を上げるために、Phi の CPU 周波数は Xeon の 2.6GHz に対して 1.1GHz と半分以下に設定されている。一方、ベクトル演算のベクトル長は Phi が Xeon の 4 に対して 2 倍 8 倍精度数（512bit）となっている。この結果、Phi のコア当たりの倍精度演算ピーク性能は Xeon に対して 2 割弱劣る程度である。しかし、プログラムの主要部分をベクトル化できない場合の性能は、Xeon のスカラ性能の 4 割程度である 2.2 GFLOPS となってしまう。

Xeon はソケット内のコア間で共用のラストレベルキャッシュ 20MB を持っているため、1 個のコアだけを使用しても、20MB のキャッシュ全体を使用することができる。これに対して、Phi のラストレベルキャッシュは、1 個のコアを使用した場合はそのコアに直結された、512KB に限られてしまう。61 個のコアで、合計 30MB のキャッシュがあるが、すべてのコア使用して、データにアクセスしない限り、全てのキャッシュを利用することができない。

2 ソケットの Xeon のサーバであれば、16 コア、一方 Phi は 61 コアを持っている。したがって、それぞれのコア数だけ並列化できれば、最大のピーク性能を得ることができる。この値は Phi では、1073GFLOPS であり、また一方、メモリ転送性能も、Phi は GDDR5 を用いることにより、ピーク性能は 350GB/s と設計されている。

5.2 ベンチマークプログラム

ベンチマークプログラムとしては単純な 2 次元の Laplace ベンチマークを選んだ。並列化の際のデータ分割は X 方向を 1 次元分割して行った。ベンチマークの問題サイズは、1 ノード 1 枚の Phi カードでの性能測定と 16 ノードのクラスタでの 16 枚の Phi でストロング・スケーリングの測定には XSIZE × YSIZE（10000×10000）を用いた。16 ノードクラスタでの測定では、最大で 40000×40000 まで使用してウィーク・スケーリングの測定も行った。XMP 版の Laplace ベンチマークプログラムは使用した XMP 実装に標準で付属している 2 次元分割の XMP 版の Laplace プログラムを 1 次元分割に変更したものを、使用した。図 **8** に 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。

```
1  #define XSIZE (10000)
2  #define YSIZE XSIZE
3
4  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
5
6  #pragma xmp nodes p(*)
7  #pragma xmp template t(0:(10000)−1)
8  #pragma xmp distribute t(block) onto p
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 . . . ( 省略)
14     /* old <− new */
15 #pragma xmp loop (x) on t(x)
16     for(x = 1; x < XSIZE−1; x++)
17       for(y = 1; y < YSIZE−1; y++)
18         uu[x][y] = u[x][y];
19
20 #pragma xmp reflect (uu)
21
22     /* update */
23 #pragma xmp loop (x) on t(x)
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;
```

図 8　1 次元分割のプログラム

*1　Xeon は 2 ソケットのシステム全体、Xeon Phi は 1 ノードの最大性能を示す。

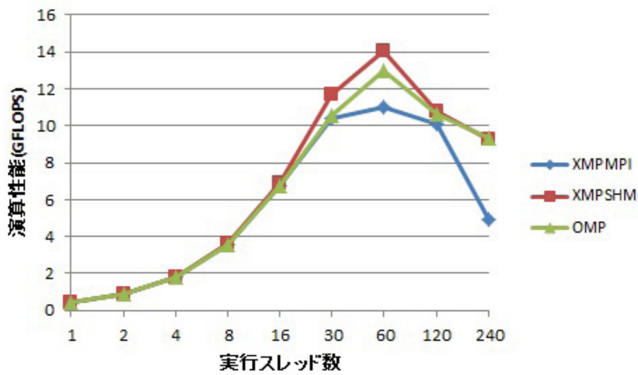

図 9　単一 Phi カードでの性能比較

5.3　単一 Phi カード上での評価

1 次元分割した Laplace のプログラムの性能を単一のノードの Phi カード上で評価した。結果を図 **9** に示す。性能の比較は図 8 の XMP プログラムを標準的なコンパイラで MPI のプログラム変換した実行モジュールでの結果 XMPMPI と本報告で実装した結果 XMPSHM と、XMP の代わりに OpenMP で並列化した結果 OMP の 3 つで行った。図 9 の横軸は実行したスレッド数で縦軸は演算性能を GFLOPS で示している。MPI [5] 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) を用いている [6]。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。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 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、次の 3 つのことが観測できた。

(1) いずれの実行方法でも、30 スレッドまでは、Laplace の性能がスレッド数にスケールする。
(2) 単一 Phi カードの Laplace の最高性能は 60 スレッドで実行した場合で、14.1GFLOPS である。
(3) 本研究で実装した、プロセスを用いて並列化したものが、最も高い性能を実現している。

5.4　16 ノード Phi クラスタ上での評価

単一 Phi カード上での評価に使用した同一のプログラムの性能評価を 16 ノードの Phi クラスタで行った。単一カードでの結果から、カードあたり 30 スレッド使用する場合と、60 スレッド使用する場合の 2 つの結果をそれぞれ、図 **10** と図 **11** に示す。図の XMPMPI と XMPSHM は、それぞれ単一 Phi カードでの実行場合と同様に、標準的な XMP の実装で MPI で実行するものと、本実装の共有メモリを使ったものでの性能を示す。XMPHY は XMP プログラムの並列ループの内側に OpenMP ディレクティブをおき、カード内は OpenMP で並列化し、カード間は MPI を使用するハイブリッドモデルである。それぞれ、単一カード内 のプロセス数（OpenMP の場合はスレッド数）を 30 か 60 に固定した上で、使用するカード枚数を、2 のべき乗倍して、Phi カード 16 枚まで測定した。

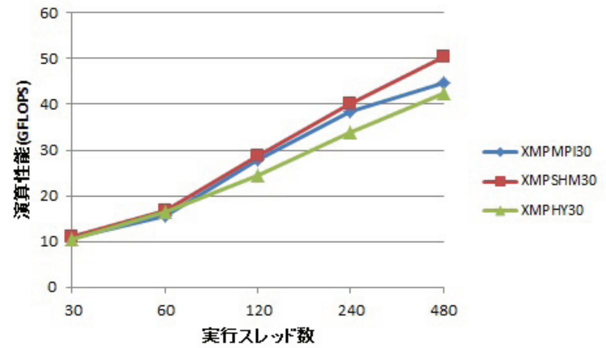

図 **10**　カード当たり 30 スレッドのクラスタ性能比較

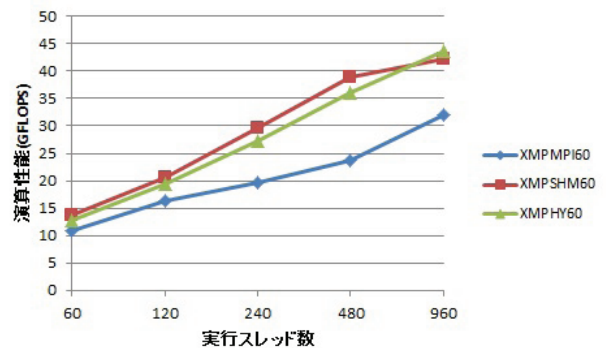

図 **11**　カード当たり 60 スレッドのクラスタ性能比較

この結果、次の 3 つのことが観測できた。

(1) いずれの実行方法でも、スケールしない。16 枚のカードを用いて 30 スレッドの場合で 5 倍、60 スレッドの場合で 4 倍弱程度
(2) 最高性能はカード当たり 30 スレッドで本報告の実装で実行した場合で、16 個のカードで 50.4GFLOPS である。
(3) カード当たり 60 スレッドで実行した場合のスケーラビリティは XMPMPI で特に悪い。

5.5　ウィークスケーラビリティの評価

単純に 1 次元分割した Laplace のプログラムでは Phi クラスタ上でスケーラビリティが得られないことがわかった。そこで、カードあたりの配列サイズを固定して、Phi カードの枚数に合わせて配列サイズを増加させることにより、ウィークスケーラビリティ性能を評価した。カード当たりの使用スレッド数を 30 と 60 にしたときの結果を、それぞれ図 **12** と図 **13** に示す。配列のサイズは、これまでは XSIZE × YSIZE（10000 × 10000）の固定であったが、

ⓒ 2013 Information Processing Society of Japan

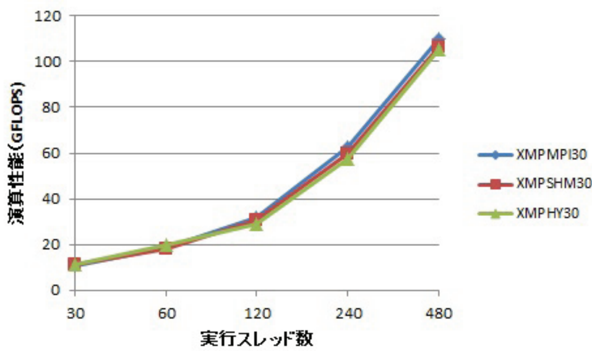

図 **12** カード当たり 30 スレッドのウィークスケーリング性能比較

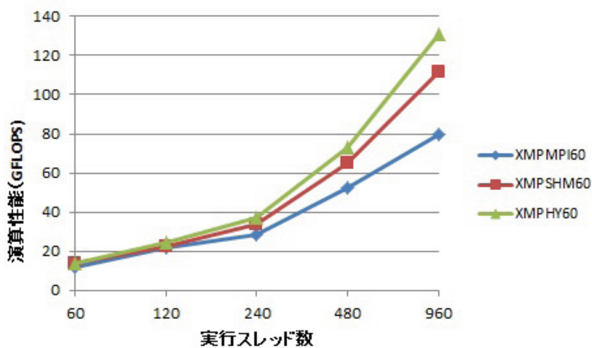

図 **13** カード当たり 60 スレッドのウィークスケーリング性能比較

この測定では、これをベースにして、カードの枚数を 2 倍する度に、まず、XSIZE を 2 倍、次は YSIZE を 2 倍という順に交互に 2 倍して行った。図のプロットは、配列サイズを固定して測定したのと同様に、XMPMPI が標準的な実装、XMPSHM が本実装を用いたもので、XMPHY が XMP で OpenMP を用いたハイブリッドで結果である。

この測定の結果、次の 2 つのことが観測できた。

(1) Phi カード当たり 30 スレッドの実行では、いずれの方式でも実行結果に大きく差異は無く、16 枚の Phi カードで単一 Phi カード性能の 9.5 倍程度、最大 110GFLOPS の性能が得られる。

(2) Phi カード当たり 60 スレッドの実行では、実装方式で結果が異なり、ハイブリッドで最大、単一 Phi カード性能の 9.4 倍の 130GFLOPS、続いて本報告の方式 112GFLOPS(8.2 倍)、標準的な MPI で 80GFLOPS(6.8 倍) が得られた。

6. 結果の考察

6.1 単一 Phi カード上での実行

Laplace ベンチマークは、図 8 に示したように、u と uu の 2 つのグローバル配列を用いて演算を行っている。演算は上下左右 4 か所の値を足仕込む 3 演算と 4 で割る除算の、配列の要素あたり 4 演算を行っている。一方、データの転送は、演算の為に参照する 4 回と演算結果を u の配列に代入する 1 回の合わせて 5 回と、参照用に使う古い値を更新する為の uu=u の 2 回計 7 回であるである。したがって、60 プロセス実行時の単一 Phi カードの最大性能 14.1GFLOPS を得るには、$14.1/4 \times 7 \times 8 = 197.4$GB/s のデータ転送性能が要求される。表 1 にしたように、Phi 単体のピークメモリ転送性能は 350GB/s であり、一見十分に見える。しかし、インテル社の実測測定による stream ベンチマーク triad の性能は、ピークの約半分の 180GB/s 程度であり [7]、Laplace ベンチマークでは本実装方式で、単一 Phi カードの、ほぼ上限の性能を実現できていることが分かる。

XMP では、袖領域を持っているため、この部分のデータの転送にシステムのメモリ転送容量を使用する。この量は 1 次元分割の場合、(分割数-1) の 2 倍の列であり、60 プロセスでの実行の場合 $(60-1) \times 2/10000 = 1.18\%$である。標準的な MPI を使用する方法では、MPI のランタイムが MPI 通信にこれを使用する。本実装方式では、これを Phi に最適化された memcpy で実行しており、最適なメモリ転送が期待できる。

一方、OpenMP では、袖領域を持たないため、袖領域の転送に、メモリ転送容量を使用しない。しかしながら、並列実行するスレッドが、演算実行時に、グローバル配列の境界付近で、互いにキャッシュラインを取り合う等の、キャッシュの整合性を保つためのメモリ転送が発生する。図 9 に示したように、60 プロセス (スレッド) で実行し、メモリ転送性能の上限に達したときにのみ、各実装方式に性能が出ており、本実装方式が最も優れていることが示せた。

6.2 Phi カードを用いたクラスタ上での実行

単一 Phi カードで測定した 10000×10000 の配列サイズで、クラスタで実行した場合、カード数を増やしても、カード数にスケールする性能向上は得られなかった。これは、カード当たり 30 スレッドを用いた図 10 の場合は、どの方式でも大きな差異は無かった。カード当たり 60 スレッドの図 11 の場合も単一 Phi カードでの性能の低い XMPMPI60 の性能が、他の 2 つに比べて低いものの、全体の傾向は変わらない。本測定での 1 次元分割では、XMPMPI、XMP-SHM、XMPHY いずれの方式でもカード間で通信するデータ量は変わらない。それぞれのカードは自分の持つ、2 列分袖領域を受信し、2 つの隣のカードにそれぞれ 1 列の袖領域を送る。1 回の演算で、計 4 列 $10000 \times 4 \times 8 = 320$KB のデータをカード間で送受信する必要がある。我々の使用したインテル MPI と同一条件ではないが、オハイオ大学の研究 [8] によると、MVAPICH2 を使用した場合、今回我々が使用したものと同等の FDR の Infiniband で Phi 間でデータ転送では、転送速度は 1GB/s 程度であり、単純な 300KB のデータ転送に、300μS 程度を要する。(レイテンシも同程度）一方、60 スレッドでの実行の場合、一回の配列全体の

ⓒ 2013 Information Processing Society of Japan

演算に要する時間は、$10000/60 \times 10000 \times 4/14.1 = 473\mu S$ である。従って、このような条件では、データ転送に要する時間が律速になってスケーラビリティが得られないことは理解できる。

使用 Phi カード数に比例して演算データ量を増やすウィークスケーラビリティの測定でも、基本的に Phi カード間の通信時間の影響は避けられない。図 12 に示すように、Phi カード間の通信量の変わらない 3 つの XMP の方式で同様なスケーラビリティが観測できる。一方カード内のメモリ転送容量をカード内部の演算で使い切ったカード当たり 60 スレッドで実行した図 13 の場合は、XMP のハイブリッド方式 XMPHY の性能に対して本報告の実行方式 XMPSHM では、4 枚と 8 枚のカードで実行したときに 10%、16 枚のカードで実行したときに 17%の性能低下が起きている。

この原因は、今回の測定の演算データ量の増やし方にあったと考えられる。分割次元が XSIZE の 1 次元側であったが、グローバル配列のデータの形状を正方に近づけようと考え、グローバル配列を、(10000,10000)、(20000,10000)、(20000,20000)、(40000,20000)、(40000,40000) の順に増加させた。このため、2 次元側の要素数に比例する袖領域の通信量が、4 枚と 8 枚のカードで実行したときに 2 倍、16 枚のカードで実行したときは 4 倍になる。Phi カード間の通信量は前述した通り、XMPMPI、XMPSHM、XMPHY いずれの方式でも変わらない。しかし、Phi カード内でも XMP を使用している XMPMPI、XMPSHM の方式は、カード内の袖部分のデータ通信量の増加によりカード内のメモリ転送容量の消費が増加する。この影響で本実装方式の単一カード性能が劣化したと考えられる。

7. 結論と今後の課題

XMP で記述した並列プログラムを、効率良く実行できる、MIC アーキテクチャ・クラスターシステム向けの、実行モデルの検討を行った。Phi のような MIC アーキテクチャの計算ノードを複数個接続したクラスターシステムにおいては、計算ノード内の Phi の提供する 240+ものスレッドによる並列性と、インターコネクトで結合される複数の計算ノードによる並列性の、2 つのレベルの並列性を同時に生かさかければ、システムの性能を十分に引き出すことはできない。我々は、XMP で記述した並列プログラムを、この様なシステムで効率よく実行する目的で、この 2 つのレベルの並列性に対応した XMP プログラムの実行モデルを検討した。このモデルでは、並列プログラムはすべてプロセスとして、独立したメモリ空間の中で演算を行う。そして、すべての並列プログラムで参照する可能性のある、グローバル配列のみ、計算ノード内にその参照要素が存在する場合と、他の計算ノードに参照要素がある場合で、異なる扱いする。まず計算ノード内には、ノード内のプロセスすべてが直接参照可能な、共有のメモリ空間を確保し、ここにグローバル配列を割り当てる。このため、計算プロセスは、計算ノード内のグローバル配列の要素を直接参照することができる。開発したランタイムは、参照要素が他の計算ノードにある場合にのみ該当する計算ノードと MPI 通信を行って値を参照する。

XMP で並列化した Laplace の 1 次元分割ベンチマークを用いて、まず、本報告のランタイムの性能を、単一計算ノード内で評価した。この結果、本ランタイムでの実行が、従来の、ノード内でも MPI 通信を用いる実行モデルや、OpenMP を用いて並列したものよりも、性能が優れていることが確認できた。次に、16 計算ノードのクラスタを用いて、同じ Laplace の、配列サイズを固定した場合と、並列数に比例して配列サイズを増加させた場合の、性能の比較を行い、これらの 3 つの実行モデルをクラスタ上で比較検討した。この結果、本提案の方式が、ノード内を OpenMP で並列化し、ノード間の並列化に MPI を用いるハイブリッド方式と同等の性能が得られることが確認できた。XMP のような単一の並列化の記述言語で、ハイブリッドと同等の性能が得られる効果は大きいと考える。

本報告では Laplace のベンチマークの 1 次元分割を用いて、MIC アーキテクチャ・クラスターシステム向けの XMP の新しい実行ランタイムの検討を開始した。今後は、並列化のための分割の多次元化や、他のベンチマークで実験等で、本方式の有効性を検証していきたいと考えている。

参考文献

[1] Intel: インテル Xeon Phi コプロセッサー. http://www.intel.co.jp/content/www/jp/ja/processors/xeon/xeon-phi-coprocessor.html.
[2] 池井 満，中尾昌広，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，Vol. 2013-HPC-138, No. 28, pp. 1–5 (2013).
[3] XcalableMP Specification Working Group: XcalableMP Specification Version 1.1 (2012). http://www.xcalablemp.org/spec/xmp-spec-1.1.pdf.
[4] University of Tsukuba HPCS Lab and RIKEN AICS Programming Environment Research Team: Omni XcalableMP Compiler (2012). http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/download.html.
[5] 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.
[6] Intel: インテル C++ Composer XE 2013 Linux 版. http://www.xlsoft.com/jp/products/intel/compilers/ccl/index.html.
[7] Raman, K.: Optimizing Memory Bandwidth on Stream Triad (2013). http://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad.
[8] Panda, D. K. D.: MVAPICH2 for Intel MIC (2013). http://mvapich.cse.ohio-state.edu/publications/ofa_apr13_mvapich2_mic.pdf.

ⓒ 2013 Information Processing Society of Japan