Vol.2013-HPC-138 No.28
2013/2/22

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

池井 満[†1†2]　　中尾 昌広[†2]　佐藤 三久[†2]

メニーコアプロセッサのインテル Xeon Phi コプロセッサを搭載した Xeon サーバ・クラスタ上で、並列プログラミング言語 XcalableMP のベンチマークプログラムを実行し、その性能評価を行った。ベンチマークとしては HIMENO ベンチの L サイズを用い、Phi コプロセッサ単体での性能を測定した。さらに XL サイズを用いて 16 ノードの Xeon サーバ上に各 1 枚実装された 16 枚の Phi コプロセッサのクラスタで測定を行った。この結果、XcalableMP のベンチマークでは、単体、クラスタの両者で MPI 版と同等の性能が得られることが確認できた。さらに、XcalableMP のプログラムに OpenMP を適用したハイブリッドにより、16 計算ノード 2048 スレッドまでの性能向上することを確認できた。

Benchmarking program performance evaluation of Parallel programming language XcalableMP on Many core processor

MITSURU IKEI[†1†2]　MASAHIRO NAKAO[†2]
MITSUHISA SATO[†2]

We have evaluated Parallel programming language XcalableMP by running a standard performance benchmark on Xeon Phi many-core co-processor cluster. We used Himeno benchmark L size for single Phi node measurement and XL size for 16 nodes of Xeon computing server with a Phi card each. As the result, we could confirm that XcalableMP version of the program can get the performance of MPI version of the same benchmark both on single node of Phi and 16 nodes of Phi cluster. We also experiment XcalableMP + OpenMP hybrid version and confirmed its performance scales up to 2048 threads on 16 compute nodes.

1. はじめに

インテル社から、従来のプロセッサと比較して、より多数コアを持つプロセッサである、MIC（Many Integrated Core）アーキテクチャのプロセッサ Intel ® Xeon Phi™ が発表された。これは、PCI Express ソケット用の拡張カード上に GDDR5 メモリ等とともに搭載されるプロセッサで、60 個程度のコアをダイ上に集積化している。このような多数のコアを一つのダイ上で集積化した汎用プロセッサとしては、グラフィック処理用途から発展した GPGPU（General Purpose Graphics Processing Unit）の他、いくつか提案はされている。しかし、GPGPU を除くと広く利用された例は少なく、その最適化の手法については広く議論はされていない。また、GPGPU も含めて、一般的に、最適化自体が特定のハードウェアに依存することが多く、プロセッサの製品間、また同じ製品シリーズ間でも世代間で異なる独自プログラミングが、最適化のためばかりか、実行できるようにするだけのためにも、必要な場合が多い。このために、このようなアーキテクチャ上でのプログラム開発のコストが、このようなプロセッサを利用する上でも障害となっている。

一方、筆者らは、プログラム開発者の負担を少なくするために、様々なアーキテクチャの並列計算機上で、共通に使え、各々のアーキテクチャの持つ性能を最大限に活かせること目指したプログラミング言語拡張、XcalableMP（以下 XMP と称する）を提案している [1]。本研究は、上述した開発コストの障害を克服するために、この XMP の Phi への適用を行うもので、本稿では、まずその最初の試みとして、XMP で記述した HIMNO ベンチマーク [2]の性能評価を Phi 単体と Phi を用いたクラスタ上で行った。

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 の言語仕様上ではコメントとなる記述を用いて並列化の指示を行うものであり、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。本稿では Fortran 版を使用したので、以下 Fortran 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。

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

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。

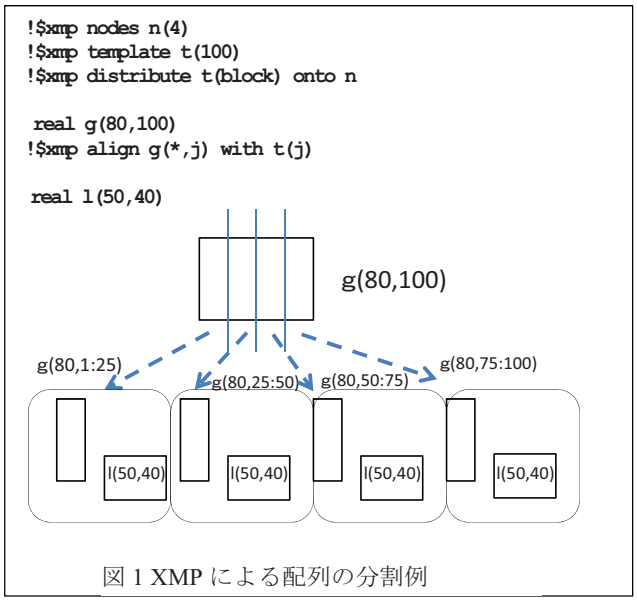

図 1 XMP による配列の分割例

以下これらのうち、データマッピングについて、図１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 Fortran のコメント行を意味する！記号で始まり、続く $ XMP で、XMP の指示文であることを表記する。図中の最初の 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(100)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(100)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。

次にプログラム本文中に、グローバルな配列として、ノード間に分配したい real 配列の g(80,100)宣言されている。次の 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 2 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。最下位行の real の配列 l(50,40)には指示文では何の指定もしていない。したがって、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 real の配列 l を独立して持つことになる。

なお、本稿では、XMP として、筑波大と AICS で開発した Omni XcalbleMP Compiler version 1.1（build 1222）を使用した。このコンパイラは、source-to-source コンパイラであり、出力プログラムとして MPI を利用したコードを生成する。実行コードの生成には、出力コードをインテル MPI を利用してコンパイルする。実行コードはインテル MPI の runtime で、MPI のプログラムとして実行される。

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 2 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

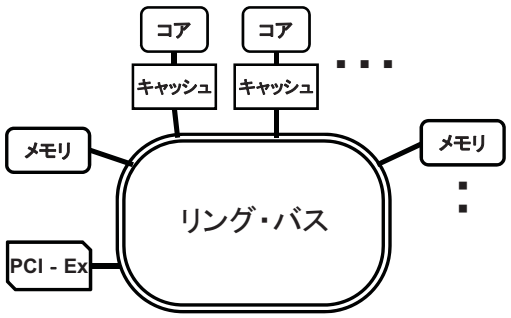

図 2 Phi の内部構成図

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。

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上げられる。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 上で意味のある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. 性能評価

4.1 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。

表 1 サーバの仕様

項目	値
CPU 周波数	2.6 GHz
メモリ周波数	1600 MHz
実装メモリ容量	32 GB
コア当たりピーク性能	GFLOPS
同上スカラ・ピーク性能	5.2 GFLOPS

4.2 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 HIMENO ベンチマークを選んだ。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には L（512x256x256）を、16 ノードのクラスタでの 16 枚の Phi での測定には XL（1024x512x512）を用いた。XMP 版の HIMENO ベンチマークプログラムは Fortran90＋OMP 版をベースに、問題サイズ L では 1 次元分割の XMP の指示文を入れたものを、使用した。図 3 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。

```
    module alloc1
    PARAMETER (mimax = 512, mjmax = 256, mkmax = 256)
    real p(mimax,mjmax,mkmax), a(mimax,mjmax,mkmax,4),
b(mimax,mjmax,mkmax,3)
    real c(mimax,mjmax,mkmax,3), bnd(mimax,mjmax,mkmax)
    real wrk1(mimax,mjmax,mkmax), wrk2(mimax,mjmax,mkmax)

!$xmp nodes n(*)
!$xmp template t(mimax,mjmax,mkmax)
!$xmp distribute t(*,*,block) onto n
!$xmp align (*,*,k) with t(*,*,k) :: p, bnd, wrk1, wrk2
!$xmp align (*,*,k,*) with t(*,*,k) :: a, b, c
!$xmp shadow p(0,0,1)
!$xmp shadow a(0,0,1,0)
!$xmp shadow b(0,0,1,0)
!$xmp shadow c(0,0,1,0)
!$xmp shadow bnd(0,0,1)
!$xmp shadow wrk1(0,0,1)
!$xmp shadow wrk2(0,0,1)
. . . （ 省略）
!$xmp reflect (p)
!$xmp loop (K) on t(*,*,K) reduction (+:GOSA)
     DO K = 2, kmax-1
       DO J = 2, jmax-1
         DO I = 2, imax-1
           S0 = a(I,J,K,1)*p(I+1,J,K)+a(I,J,K,2)*p(I,J+1,K)
```

図 3　1 次元分割のプログラム

4.3 単一 Phi カード上での評価

1 次元分割した L クラスのプログラムを単一のノードの Xeon サーバ上で評価を行った。結果を図 4 に示す。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。

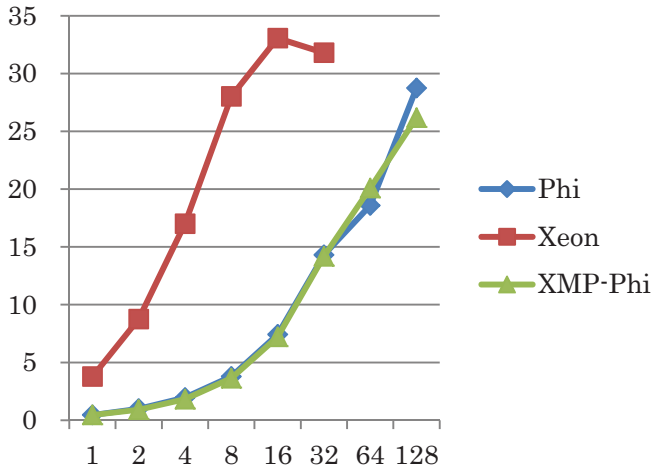

図 4 単一 Phi カードでの性能比較

評価は Fortran90 の MPI 版を Xeon 用、Phi 用にそれぞれインテルコンパイラ mpiifort で -O3 –AVX（Xeon の場合）と -mmic （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で実行した結果と比較して示している。それぞれ、MPI ノード数（スレッド数）を 1 から 2 のべき乗ごとに測定した。Xeon では最大コア数の 4 倍である 32 まで測定し、Phi ではコア数の 2 倍を少しオーバするが、128 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) 単一 Phi カードの Himemo ベンチマークの性能は 2 ソケットの Xeon サーバとほぼ同等である。
(2) Phi の 1 コア当たり、2 スレッド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きた。

4.4 プログラムの 2 次元分割

単一カードでの測定結果をふまえて、InfiniBand で接続された 16 ノードのサーバ上で、同様の測定を行った。各 Xeon ノードには、それぞれ 1 枚の Phi のカードが搭載されている。

今回の条件では、Phi で 1 ノード当たり 128 スレッド使用した場合に最高性能が得られたので、これは固定して性能測定を行うことにした。この結果、16 ノードでは最高 2048 スレッドまでの測定を行うことにした。Himeno ベンチマークのサイズとしては、XL（1024x512x512）を用いる。

XMP の 1 次元分割では、必要な並列度が得られないため、2 次元分割を用いることにした。図 5 に 2 次元分割の主要部分を示す。

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

図 5　2 次元分割の主要部分

このプログラムでは、node を n(4,*)と宣言して template t の 2 次元目を 4 つブロックに分割している。これ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、XL でも、2048=4x512 スレッドまでの並列実行を測定した。

4.5 Phi カードを用いたクラスタ上での評価

2 次元分割を用いた XMP のプログラムを、16 ノードの Xeon クラスタ上でスレッド数を変えて実行した。結果を図 6 に示す。単一カードの場合と同様に Fortran90 の MPI 版を Xeon 用、Phi 用にそれぞれインテルコンパイラ mpiifort で -O3 -AVX 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＝16x16 個まで測定した。Phi では、2 枚（256=128x2）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 点の測定となっている。

この結果、次のことが観測できた。

(1) 4 台の Phi カード（512 スレッド）と 2 ソケットの Xeon サーバ 4 台（64 スレッド）の性能は近い値であるが、スレッド数が増える程、Xeon の方が性能が良くなる。
(2) Phi 用に MPI で並列化したプログラムでは、1024 スレッド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 スレッドまで性能向上得ることがきた。

4.6 XMP を用いたハイブリッド

MPI プログラムでのサーバ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 版のプログラムから得られた中間出力の MPI プログラムに手入力で OpenMP の指示文を入れることで、プログラムをハイブリッド化することを試みた。現在、OpenMP との混在利用についてはいま、XMP 規格部会で議論中であり、Omni XcalableMP コンパイラでも一部はサポートしているが、今回はマニュアルで OpenMP の入力を行った。図 7 にこの中間プログラムの一部を示す。

詳細は説明しないが、使用したリファレンス用 XMP コンパイラの中間出力は、一般的な MPI プログラムとは異なり、コンパイル前の XMP プログラムの構成をそのままの残した Fortran プログラムとなっている。このため、ハイブリッド化は比較的容易におこなえる。図は基本的に 3 つの部分に分かれている。

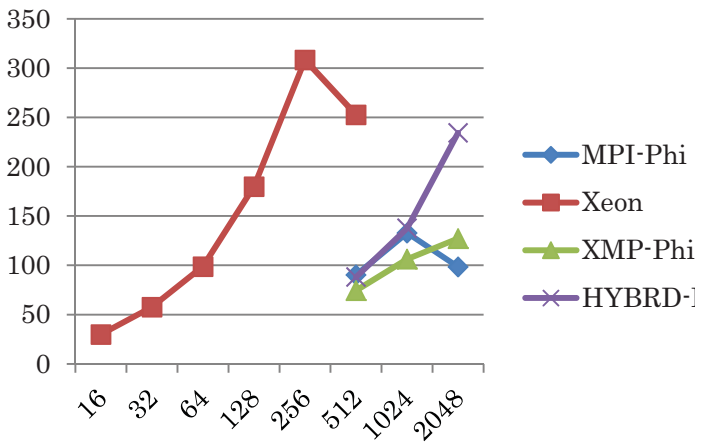

図 6 Phi クラスターでの性能評価

```
!$OMP PARALLEL SHARED (kmax,jmax,imax,nn, &
!$OMP t5,XMP_loop_lb9,XMP_loop_ub10,XMP_loop_step11,gosa) &
!$OMP PRIVATE (k6,k8,j,i,s0,ss,gosa1,loop)

DO loop = 1 , nn , 1
# 133 "himeno1.f90"
!$OMP BARRIER
!$OMP MASTER
 gosa = 0.0_8
 CALL xmpf_reflect_ ( XMP_DESC_p )
 CALL xmpf_ref_templ_alloc_ ( t5 , XMP_DESC_t , 3 )
 CALL xmpf_ref_set_loop_info_ ( t5 , 0 , 2 , 0 )
 CALL xmpf_ref_init_ ( t5 )
 XMP_loop_lb9 = 2
 XMP_loop_ub10 = kmax - 1
 XMP_loop_step11 = 1
 CALL xmpf_loop_sched_ ( XMP_loop_lb9 , XMP_loop_ub10 ,
XMP_loop_step11 , 0 ,
t5 )
!$OMP END MASTER
!$OMP BARRIER

 gosa1 = 0.0_8
!$OMP DO COLLAPSE(2)
 DO k6 = XMP_loop_lb9 , XMP_loop_ub10 , XMP_loop_step11
  DO j = 2 , jmax - 1 , 1
   DO i = 2 , imax - 1 , 1
```

図 7　XMP ハイブリッドプログラム

図中の!$OMP で始まる行が手入力された部分である。最初の部分が、OpenMP の並列化の方法と部分を指定する Parallel 文関連である。2 番目が、メモリの確保や MPI の実行をしている XMP の内部ルーチンで、これらは XMP のプロセス 1 個で 1 回実行すれば良いので、MASTER を用いて、1 個の OpenMP スレッドのみで実行している。3 番目の部分が、Do 文の並列実行を指定する部分である。

ハイブリッドプログラムは、XMP による並列化は Phi の枚数と同じにして実行した。カードごとのスレッド数は 128 である。結果は図 6 に示した。この XMP と OpenMP のハイブリッド版では 512, 1024 スレッドの時には、MPI 版と同等の性能が得られた。また、2048（=128x16）スレッドの場合でも性能が向上しており、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、期待通りの性能向上を得ることができた。

5. 結論と今後の課題

我々の研究の目的は、Phi のような MIC アーキテクチャのシステムにおいて、そのアーキテクチャを活かせる実行コードをコンパイラに生成させることを目指している。Phi のようなプロセッサはコア数や単一コアの性能、コア間の接続トポロジー等、従来のプロセッサとは多く異なっている。このため、一般のプログラム開発者が、どのよう並列プログラムをすれば期待できる性能が得られるのかは、まだ、よく理解されていない。

一方、並列プログラミング言語の 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、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、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。

我々は、XMP を用いて MIC アーキテクチャのプログラムを行うことにより、それぞれのアーキテクチャに適した実行コード生成できるのではないかと考え、この研究を始めた。本稿はその最初の一歩として、HIMENO ベンチマークの XMP を用いた並列化を試みた。今後、様々なベンチマークプログラムを用いて同様な評価を行い、XMP を用いた MIC アーキテクチャ向けの並列化プログラム手法の可能性を追求し、また、生成コードに用いられる XMP ランタイムの MIC 向け最適化行って行こうと考えている。

参考文献

1)　XcalableMP / What's XcalableMP
http://www.xcalablemp.org
2)　姫野ベンチマーク /
http://accc.riken.jp/2145.htm