Vol.2014-HPC-144 No.4
2014/5/26

GPU/MICクラスタにおける疎行列ベクトル積の性能評価

前田 広志[1,a)]　高橋 大介[2,b)]

概要：疎行列ベクトル積は，科学技術計算をはじめとする多くのアプリケーションにおいて重要な計算カーネルである．近年ではアクセラレータとして NVIDIA 社の GPU や，Intel 社の提唱する MIC アーキテクチャに基づいた Xeon Phi コプロセッサ等を搭載した計算機システムが増加しており，これらのシステムを活用できるアルゴリズムが重要となっている．本研究では GPU を搭載したクラスタと Xeon Phi を搭載したクラスタにおいてそれぞれのシステムで効果的な並列 SpMV のアルゴリズムについて検討，実装および評価を行った．その結果，行列の形状や MPI プロセス数により適するアーキテクチャは異なるが，一部の行列において GPU や MIC 向けの実装では CPU 向けの実装以上の性能を達成することができた．

1. はじめに

疎行列ベクトル積（Sparse Matrix Vector Multiplication，以下 SpMV）は，科学技術計算をはじめとする多くのアプリケーションにおいて重要な計算カーネルである．SpMV は工学や物理学におけるシミュレーション時間に大きな影響を与えており，SpMV を高速化することで実アプリケーションの実行時間の削減が期待できる．これまでに並列 SpMV のアルゴリズム [1][2][3] が提案されてきたが，これらの効果は対象の行列の非零構造や計算機のアーキテクチャに大きく依存し，最適な高速化アルゴリズムは行列や計算機の性質によって異なってしまう．

近年ではアクセラレータとして NVIDIA 社の GPU（Graphics Processing Unit）や，Intel 社の提唱する MIC（Many Integrated Core）[4] アーキテクチャに基づいた Xeon Phi コプロセッサ等を搭載した計算機システムが増加している．スーパーコンピュータの性能をランク付けする TOP500[5] において，2013 年 11 月の発表では 1 位のシステムに Xeon Phi コプロセッサが，2 位のシステムには NVIDIA 社の GPU である K20X が搭載されており，アクセラレータの重要性が高まっていることが分かる．実際にアクセラレータを SpMV に応用する研究 [6][7][8] は盛んに行われており，GPU 等のアーキテクチャに適したアルゴリズムや格納形式等も提案されている．

また TOP500 に代わるベンチマークとして，SpMV が計算のボトルネックとなりやすい共役勾配法（Conjugate Gradient Method，以下 CG 法）を利用する HPCG（High Performance Conjugate Gradient）[9] が提案されるなど，これから SpMV はさらに注目されることになると考えられる．そのため，SpMV においてこれらのシステムを活用できるアルゴリズムが重要となっている．本研究では GPU 搭載ノードからなるクラスタと Xeon Phi コプロセッサ搭載ノードからなるクラスタにおいてそれぞれのシステムで効果的な SpMV のアルゴリズムについて検討，実装および評価を行う．

1　筑波大学大学院システム情報工学研究科
2　筑波大学システム情報系
a)　maeda@hpcs.cs.tsukuba.ac.jp
b)　daisuke@cs.tsukuba.ac.jp

2. 疎行列ベクトル積

疎行列とは多くの要素が零である行列を指す．SpMV は M 行 N 列の疎行列 A，N 行の密ベクトル x，M 行のベクトル y を用いて式 (1) で表される．

$$y = Ax \tag{1}$$

SpMV は特に CG 法などの線型方程式を解くアルゴリズムに必要な計算であり，処理時間においても大きな比率を占める．これらのアルゴリズムは反復計算を含み，行列は変わることなく繰り返し利用される．そのため，疎行列ベクトル積では反復間で行列が変わらないことを前提とした分散方法，前処理等に様々な手法が提案されている．

また，疎行列には様々な格納手法が提案されている．今回用いるのはスタンダードな手法である CRS（Compressed Row Storage）[2][10] 形式であり，この形式では式 (2) の行列を式 (3)，(4)，(5) のように 3 つの配列で格納する．val 配列では行列の値を保持し，idx 配列ではそれぞれの非零要素の列インデックスを保持する．ptr 配列ではそれぞれの行の先頭要素に対応する val 配列と idx 配列のインデックスを持つ．この手法は，必要となるメモリは少ないが，

並列化を行う際に行分割をすると負荷が不均一になってしまうなどの特徴がある．

$$A = \begin{Bmatrix} 1 & 0 & 0 & 4 \\ 0 & 3 & 0 & 0 \\ 0 & 5 & 8 & 0 \\ 6 & 0 & 9 & 2 \end{Bmatrix} \quad (2)$$

$$val = [\ 1 \quad 4 \quad 3 \quad 5 \quad 8 \quad 6 \quad 9 \quad 2\ ] \quad (3)$$

$$idx = [\ 0 \quad 3 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \quad 3\ ] \quad (4)$$

$$ptr = [\ 0 \quad 2 \quad 3 \quad 5 \quad 8\ ] \quad (5)$$

2.1 並列 SpMV

SpMV を並列に計算するために，疎行列 A とベクトル x とベクトル y を複数 MPI プロセスで保持するための分散手法や格納形式などを検討する必要がある．格納形式は前述の通り CRS 形式を用いるが分散手法には PETSc[1] 等で用いられる手法を選択した．

行列 A とベクトル x，y の分散の概要を**図 1** に示す．P0〜P7 は 8 個の MPI プロセスの番号となるランクを表す．この分散手法ではそれぞれの MPI プロセスが行列を行分割したものと，行列の対角要素に対応するベクトルの一部を保持し，それぞれの保持した行に対応する計算を行う．これにより非零要素が対角線上に多いタイプの行列には効果的な分散方法となる．しかし，逆に非零要素が対角要素以外に多い場合には非効率となる．

SpMV を計算する際に通信が必要な領域，不要な領域の概要を**図 2** に示す．計算時には他の MPI プロセスからベクトル x の各要素を受け取ることで行列を分散したままで SpMV の計算を可能にする．この通信では必要な要素のみを受け取ることで通信量を最小にする手法 [3] を用いた．また通信の前にベクトル x の保持している要素とその要素の行に対応する列に存在する行列 A の要素との計算を行うことができるため，通信と計算をオーバーラップする手法 [1] を用いた．

3. アクセラレータ

アクセラレータとはこれまで CPU の行ってきた一部の処理に特化し，高速に計算を行うハードウェアのことを指す．近年では NVIDIA 社の GPU や Intel 社の MIC アーキテクチャなどが話題となっている．これらのアクセラレータは PCI Express バスによりホストと接続されるという点で共通しているがハードウェアの構成や扱い方においては大きく異なる．本研究では NVIDIA 社の GPU を用いるための開発環境である CUDA（Compute Unified Device Architecture）[11] や，MIC アーキテクチャに基づいたプロセッサである Xeon Phi を用いた．本章では Xeon Phi の実行モデルについて述べ，CUDA や MIC アーキテクチャ

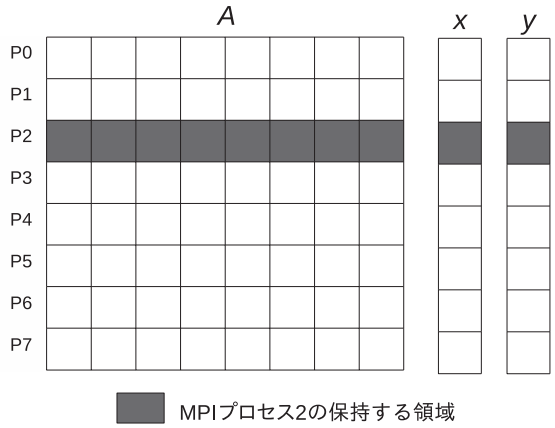

図 1 並列 SpMV における各 MPI プロセスのデータ保持

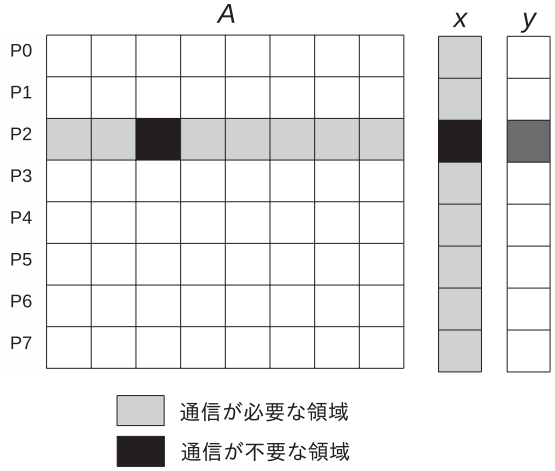

図 2 並列 SpMV における各 MPI プロセスの計算

の詳細については割愛する．また，本論文ではホストのプロセッサと区別するために GPU や Xeon Phi などを総称してコプロセッサと呼ぶ．

3.1 Xeon Phi の実行モデル

Xeon Phi を扱う方法にはネイティブモデルとオフロードモデルがある．ネイティブモデルとはプログラムの実行を全て Xeon Phi 上で行うモデルである．このモデルは既存のソフトウェアのコードの変更を最小限に抑えるモデルであり，並列性が高いプログラムには適しているが，並列性の低いプログラムや I/O の多いプログラムには適さない．

これに対し，オフロードモデルは基本的にホスト CPU 上でプログラムを実行し，一部のコード領域をプログラマーが指定することにより Xeon Phi 上で実行するモデルである．このモデルではオフロード時にコプロセッサへのデータ転送を必要とするため，オフロード領域は転送のオーバーヘッドと並列化による効率化を考慮し，慎重に決める必要がある．また，実行するターゲットとなるデバイスが存在しない場合はホスト CPU 上で実行されるため，クラスタ環境などにおいて柔軟な構成が可能となっている．

本研究ではマルチノード環境を対象としており，MPI（Message Passing Interface）を用いるため様々な実行方法がある．MPI を用いる場合の実行方法の概要を**図 3** に示す．まず，1 つ目の Host-only モデルはホスト CPU のみで MPI を用いた構成であり，これが従来のマルチノードの実行スタイルである．次に，2 つ目の Offload モデルは Host-only モデルのプログラムの一部をオフロードにより Xeon Phi 上で実行する．3 つ目の Symmetric モデルはホスト CPU と同じプログラムを Xeon Phi 向けにビルドしネイティブモデルで実行することで，ホスト CPU と同じ振る舞いをするモデルである．このモデルは全ての計算資源を活用するため計算資源を無駄にしない．4 つ目の MIC-only モデルは Xeon Phi 向けにビルドされたバイナリを用いて Xeon Phi だけでネイティブモデルで実行し，ホスト CPU を一切使わないモデルである．同一ノードに存在する複数の Xeon Phi ボード間だけでなく，異なるノードに付属する Xeon Phi ボード間でも MPI 通信を行うことが可能である．

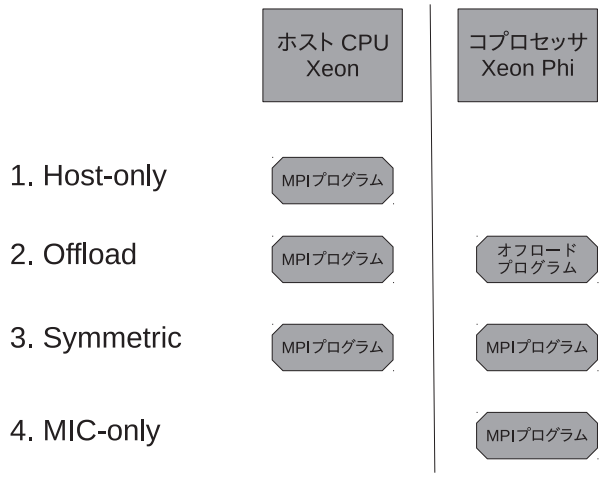

図 3 Xeon Phi の実行モデル

4. SpMV の実装

先行研究では CPU や GPU におけるマルチノード向け SpMV の実装のみで，MIC におけるマルチノード向け実装は我々の知る限りでは研究されていなかった．しかし多数のコアを備える MIC を複数利用することで CPU や GPU 搭載クラスタ以上の性能を達成できることが期待できる．本研究では通信部分に先行研究で用いられている手法 [1]，そして計算部分には既存のライブラリを用いて，CPU，GPU，MIC におけるマルチノード向けに実装を行った．処理の擬似コードを**図 4** に示す．

4.1 並列化

並列化には第 2.1 節で示した手法を用いた．CG 法などで用いられる SpMV では同じ行列を複数回反復して用いることを前提とし，事前に対象の行列を全ての MPI プロセスで出来るだけ行数が均等になるように連続した行を割り当てる（図 1）．割り当てが決定すると必要な通信が確定するので事前に送受信する MPI プロセスのランク，送受信するベクトル x のインデックスを保持しておく．SpMV ではまず，それぞれが通信により取得する必要のあるベクトル x の各要素の送受信を非同期で開始する．次に通信の完了を待たずに計算に通信の不要な部分，すなわち行列の対角要素に対応する計算を，通信とオーバーラップして行い，終了すると非同期通信の完了を待つ．最後に通信が完了すると対角外要素の計算を行う．

GPU や MIC オフロードモデルを利用する場合，図 4 の 27 行目と 31 行目の SpMV の計算前にベクトル x の要素をコプロセッサへ転送する．そして 31 行目の計算終了時にはベクトル y をホストに転送する処理を行う．

また，通信においてはベクトル x の各要素を送る際に 1 つずつ送信するのではなく，連続した領域に値を格納し，まとめて送信するためにベクトル x の各要素を配列に格納し直すパッキングを行う．この処理は通信するデータ量が多い場合にはオーバーヘッドとなることがあり，MIC ではその影響がホスト CPU と比べると大きいためパッキングにはスレッド並列化を行うことで性能の向上を図った．

4.2 計算カーネル

行列の格納形式には第 2.1 節で説明した CRS 形式を用い，SpMV を実際に計算する部分には既存のライブラリを使用した．CPU と MIC 向けの実装では Intel 社の提供する数値演算ライブラリである Intel MKL（Math Kernel Library）[12] を用いた．MKL には `mkl_dcsrmv` 関数が存在し，式 (6) を倍精度型で計算することができる．

$$y = \alpha \times Ax + \beta \times y \tag{6}$$

ローカル部分の計算では α を 1，β を 0 に設定し．ローカル外の計算では α を 1，β を 1 に設定することでベクトル y の計算を行った．

GPU 向けの実装には NVIDIA 社の提供する cuSPARSE（CUDA Sparse Matrix library）[13] を用いた．cuSPARSE には `cusparseDcsrmv` 関数が存在し，MKL と同様に式 (6) を倍精度型で計算することができるため， MKL と同様に α，β を設定することで y の計算を行った．

5. 性能評価

5.1 実験概要

本章ではそれぞれの実装において GPU/MIC クラスタ上で測定した結果を示す．GPU クラスタには筑波大学計算科学研究センターの HA-PACS，MIC クラスタには同じく筑波大学計算科学研究センターの COMA（PACS-IX）を用いた．HA-PACS，COMA の諸元を**表 1**，**表 2** に示

```
1: inputs
2:         行列 A,
3:         ベクトル x,
4:         保持しているベクトル x の要素数 nLocal,
5:         送るベクトル x のインデックスのリスト
           elementsToSend,
6:         送るベクトル x の要素数の合計 totalToBeSent,
7:         送るベクトル x の要素数のリスト sendLength,
8:         貰うベクトル x の要素数のリスト recvLength,
9:         MPI プロセス数 nProc,
10: outputs
11:        ベクトル y
12:
13: packed ← {}
14: for i = 0 to totalToBeSent do
15:        packed[i] ← x[elementsToSend[i]] { パッキング
           処理（スレッド並列化）}
16: end for
17: nSend ← 0
18: for i = 0 to nProc do
19:        AsynchronousSend(packed[nSend], i) {MPI プ
           ロセス i へ非同期送信 }
20:        nSend ← nSend + sendLength[i]
21: end for
22: nRecv ← nLocal
23: for i = 0 to nProc do
24:        AsynchronousReceive(x[nRecv], i) {MPI プロセ
           ス i から非同期受信 }
25:        nRecv ← nRecv + recvLength[i]
26: end for
27: y ← Ax { ローカル部分の計算 }
28: for i = 0 to nProc do
29:        WaitCommunication(i) {MPI プロセス i から送
           られるデータの受信完了まで待つ }
30: end for
31: y ← y + Ax { ローカル外の計算 }
32: return y
```

図 4 SpMV の擬似コード

す．HA-PACS の Tesla M2090 の理論ピーク演算性能は約 665GFLOPS（倍精度）であるのに対し，COMA の Xeon Phi 7110P は約 1208GFLOPS（倍精度）であるため，MPI プロセス当たりの演算性能は MIC 向け実装の評価環境の方が性能が高いと考えられる．

入力となる行列は The University of Florida Sparse Matrix Collection[14] から正方行列で unsymmetric なものを選んだ．それぞれの行列の情報を**表 3** に示す．

MIC クラスタに関しては MPI 並列化した Offload モデルと MIC-Only モデル向け実装を用いた．それぞれの MPI プロセスは MIC のオフロードモデルとネイティブモデルにより実現されているため，以下ではそれぞれオフロードモデル，ネイティブモデル向け実装と呼ぶ．Symmetric モデルについてはホストとコプロセッサ間での MPI プロセス数やスレッド数を検討する必要があり，ロードバランスを取ることが困難なため今回は割愛する．COMA では利用形態により複数のパーティションが用意されており，今回は汎用 CPU 20 コアと MIC 2 基を占有できる混合パーティションを利用した．

実験ではプログラム中の図 4 に相当する部分のみを測定した．GPU，MIC オフロードモデル向け実装では行列の領域をコプロセッサ側で事前に確保，転送しておき，測定時にはベクトル x のみをアクセラレータに送る形とした．また，測定にはそれぞれの行列に対し SpMV を 10 回繰り返し，性能が最も高かった値を採用した．COMA では Xeon Phi のクロックソースに jiffies[15] を用いているため低い分解能でしか測定ができなかった．そのためネイティブモデル向け実装では 1 回の測定を測定時間が 1 秒に達するまで SpMV を繰り返し，要した時間を SpMV の実行回数で割るようにした．

表 1 HA-PACS の諸元

CPU	Intel Xeon E5-2670 2.6GHz × 2
メインメモリ	DDR3 1600MHz 128GB
GPU	NVIDIA Tesla M2090 × 4
GPU メモリ	GDDR5 6GB × 4
ノード間接続	Infiniband QDR × 2 レール
ノード数	268
コンパイラ	Intel C++ Compiler 14.0
MPI	Intel MPI 4.1.3
CUDA Toolkit	5.0.35
数値演算ライブラリ	Intel MKL 11.1.0

表 2 COMA の諸元

CPU	Intel Xeon E5-2670v2 2.5GHz × 2
メインメモリ	DDR3 1866MHz 64GB
MIC	Intel Xeon Phi 7110P（61 コア）× 2
MIC メモリ	GDDR5 8GB × 2
ノード間接続	Infiniband FDR × 2 レール
ノード数	393
コンパイラ	Intel C++ Compiler 14.0
MPI	Intel MPI 4.1.3
数値演算ライブラリ	Intel MKL 11.1.2

5.2 実験結果と考察

それぞれの実装の実験結果について考察し，最後にアーキテクチャ間の比較を行う．またグラフで横軸に行列を扱っている場合は非零要素数が昇順になるように並べている．なお本論文では通信時間と言った場合，MPI 通信に要する時間のみを指し，GPU と MIC オフロードモデルを用いた際のホストとコプロセッサ間のデータ転送時間は含まない．

表 3　測定対象とする行列

行列名	行数，列数	非零要素数
lhr34	35152	764014
g7jac200sc	59310	837936
twotone	120750	1224224
mac_econ_fwd500	206500	1273389
raefsky3	21200	1488768
ASIC_680ks	682712	2329176
thermomech_dK	204316	2846228
stomach	213360	3021648
webbase-1M	1000005	3105536
sme3Dc	42930	3148656
laminar_duct3D	67173	3833077
xenon2	157464	3866688
para-4	153226	5326228
Chebyshev4	68121	5377761
Hamrle3	1447360	5514242
PR02R	161070	8185136
torso1	116158	8516500
ohne2	181343	11063545
TSOPF_RS_b2383_c1	38120	16171169
Freescale1	3428755	18920347
rajat31	4690002	20316253
FullChip	2987012	26621990
RM07R	381689	37464962
circuit5M	5558326	59524291
cage15	5154859	99199551
ML_Geer	1504002	110879972
HV15R	2017169	283073458

5.2.1　CPU 向け実装

CPU 向け実装では HA-PACS 上で 1 ノード当たり 4MPI プロセス，1MPI プロセス当たり 4 スレッドを割り当てて測定を行った．MPI プロセス数を変化させた時の測定結果のうち特徴的な行列について**図 5** に示す．CPU 向け実装においては 64MPI プロセスで実行することで最大約 53.38 倍性能が向上した．一部，MPI プロセス数が 2 倍になると性能が 2 倍以上となる結果が見られたが，これは 1MPI プロセス当たりの非零要素数が減少しキャッシュヒット率が増加したことが影響していると考えられる．

sme3Dc では MPI プロセス数が 8 以下では MPI プロセス数の増加に従って性能も向上しているがそれより多くなると伸びが緩やかになった．さらに MPI プロセス数が 64 になると計算時間は短縮されるものの，通信時間が増加し，全体の性能が低下する結果となった．これは他の多くの行列においても同様で，ある一定の MPI プロセス以上になると性能の伸びが緩やかになり，一部の行列では低下する結果となった．

これに対し，ML_Geer では MPI プロセス数が増えるに従って性能が大きく向上している．これは ML_Geer がほとんどの要素が対角要素上に存在する行列で，MPI プロセス数を増やしても通信量がほとんど増加しないからであった．

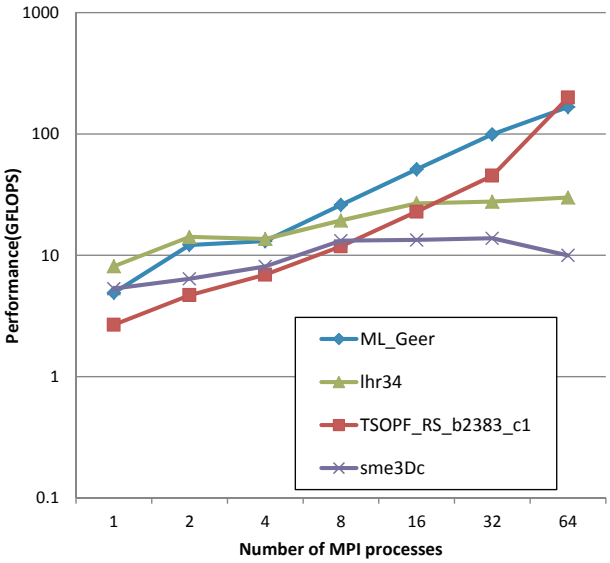

図 5　CPU 向け実装の性能（HA-PACS）

5.2.2　GPU 向け実装

GPU 向け実装では HA-PACS 上で 1 ノード当たり 4MPI プロセス，1MPI プロセス当たり 4 スレッドと 1GPU を割り当てて測定を行った．割り当てたスレッドは通信の前のパッキング処理のみに利用した．MPI プロセス数を変化させた時の測定結果のうち特徴的な行列について**図 6** に示す．

GPU 向け実装では非零要素数が少ない時には性能が低く，多くなるに従って性能が高くなる傾向にあった．これは GPU では計算カーネルを起動する前にベクトル x をコプロセッサに転送し，計算カーネルの実行が終わると計算結果である y をホスト側に転送する必要があり，このデータ転送が非零要素数が少なく演算時間の割合の少ない場合に大きく影響するためである．そのため，測定した行列中で非零要素数が最少の lhr34 では比較的非零要素数の多い ML_Geer や TSOPF_RS_b2383_c1 に比べて低い性能となっている．

ML_Geer では MPI プロセス数の増加に従って性能が向上している．ML_Geer は非零要素数が多い割に行列の行数，列数が比較的少ない．そのため，GPU の転送時間が少なく本来の演算性能を発揮できたと考えられる．

5.2.3　MIC オフロードモデル

オフロードモデル向け実装では COMA 上で 1 ノード当たり 2MPI プロセス，1MPI プロセス当たりホスト CPU10 スレッドと 1MIC を割り当てて測定を行った．また MIC のスレッドのアフィニティは compact，スレッド数は 240 とした．MPI プロセス数を変化させた時の測定結果のうち特徴的な行列について**図 7** に示す．

オフロードモデル向け実装では GPU と同様に非零要素数が少ないと性能が低くなる傾向にあった．このモデルで

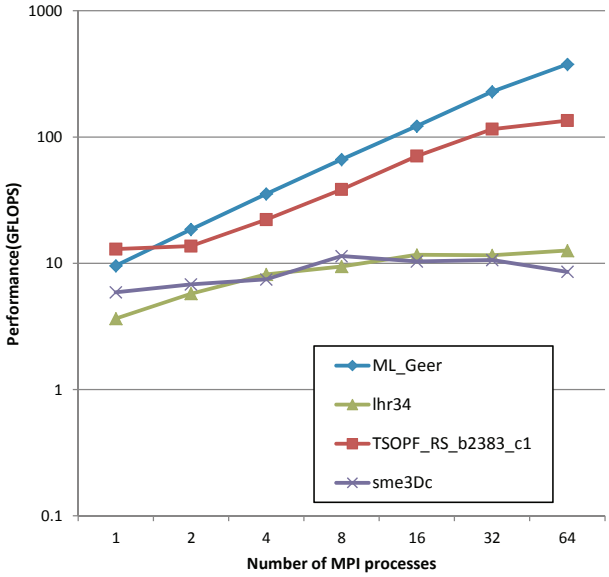

図 6 GPU 向け実装の性能（HA-PACS）

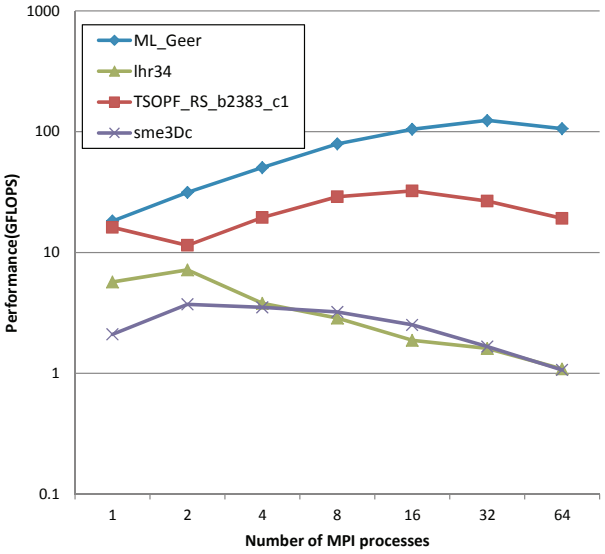

図 8 MIC ネイティブ向け実装の性能（COMA）

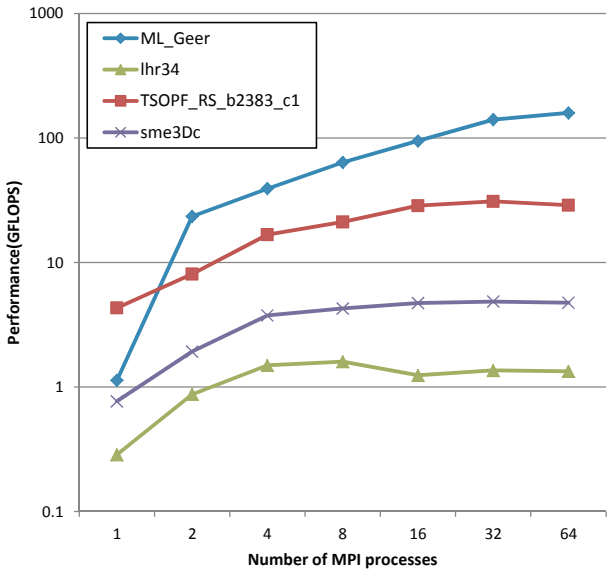

図 7 MIC オフロード向け実装の性能（COMA）

は GPU と同様に計算前にベクトル x をコプロセッサに転送し，計算が終わると計算結果である y をホスト側に転送する必要がある．この転送が非零要素数が少ない場合に性能に大きな影響を与えていると考えられる．

ML_Geer のように非零要素数が多く，MPI プロセス数が増えてもそれぞれの MPI プロセスの保持する非零要素数が極端に偏らないような行列では MPI プロセス数が増えるに従って性能も向上する結果となった．これは GPU と同様でそれぞれの MPI プロセスの保持する非零要素数が多く，コプロセッサへのデータ転送の影響が少ないからであると考えられる．

5.2.4 ネイティブモデル

ネイティブモデル向け実装では COMA 上で 1MIC 当たり 1MPI プロセスを割り当てて測定を行った．また MIC のスレッドのアフィニティは compact，スレッド数は 240 とした．MPI プロセス数を変化させた時の測定結果のうち特徴的な行列について**図 8** に示す．

ML_Geer のように非ゼロ要素数が多い行列では MPI プロセス数が少ない場合に，他の実装に比べて比較的性能が高くなる傾向にあった．これは十分な非零要素があったことで MIC の演算能力を引き出すことができたからであると考えられる．

lhr34 では MPI プロセス数が増えるに従って性能が低下している．これは MIC が MPI で通信を行う際に PCI Express バスを経由するためレイテンシが比較的大きく，通信と演算をオーバーラップしても通信時間を隠すことができなかったためであると考えられる．OSU Micro-Benchmarks 4.3[16] の osu_latency により測定したレイテンシの比較結果を**図 9** に示す．OSU Micro-Benchmark 4.3 により測定した結果，MIC 間（MIC-MIC）ではホスト間（HOST-HOST）に比べ通信のレイテンシが大きかった．この通信のレイテンシは他の行列にも同様に影響している．特に sme3Dc では非零要素が散らばっており MPI プロセス数の増加に伴いほとんどの MPI プロセス間で小さなデータを送受信する通信が発生するため，この影響を大きく受け MPI プロセス数の増加に従って性能が低下する結果となった．

また ML_Geer でも MPI プロセス数が増えるに従って演算時間は短縮されるものの，多くの MPI プロセスと通信を行うことでレイテンシが積み重なり，性能の伸びが緩やかになり，一部の行列では低下する結果となった．

5.3 異なるアーキテクチャ間の比較

それぞれのアーキテクチャにおける性能を比較した結果を**図 10**，**図 11** に示す．図 10 は 1MPI プロセス，図 11

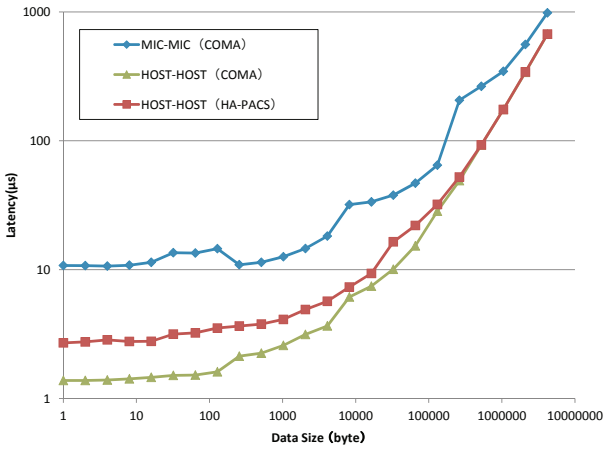

図 9 レイテンシの比較

は 64MPI プロセスにおける性能を表している．HV15R は非零要素数が今回使用した行列中で最も多く，ネイティブモデル向け実装 1MPI プロセスではメモリ容量が不足し，測定できなかったためグラフでは割愛した．

全体的に CPU では行列の構造が大きく影響を与えており，GPU や MIC を用いた場合ではさらに非零要素数も性能に大きな影響を与えている．

5.3.1 1MPI プロセスにおける比較

1MPI プロセスにおいては非零要素数の少ない行列では CPU 向け実装の性能が高く，非零要素数が増えるとネイティブモデル向け実装の性能が高くなる傾向にあった．1MPI プロセスでは通信が発生しないため非零要素数の多い行列ではネイティブモデル向け実装が本来の演算性能を引き出すことができ，他の実装を上回る性能を達成できたと考えられる．MIC では全ての行列においてオフロードモデル向け実装よりもネイティブモデル向け実装の性能が高かったがこれはオフロードモデル向け実装ではコプロセッサへのデータ転送の時間が影響しているためである．

また，GPU 向け実装も非零要素数が少ない場合には CPU 向け実装よりも性能が低いが，非零要素数が増えるに従って CPU 向け実装の性能を上回る結果となった．

5.3.2 64MPI プロセスにおける比較

64MPI プロセスにおいては 1MPI プロセスの場合とは大きく異なる結果となった．まず CPU 向け実装においては行列の形状により性能の伸びが大きく異なった．GPU 向け実装では多くの行列において CPU 向け実装よりも性能が低いが，非零要素数の多い RM07R，cage15，ML_Geer，HV15R においては他の実装の性能を上回る結果となった．また，MIC ではほとんどの行列においてネイティブモデル向け実装よりもオフロードモデル向け実装の性能が高かった．ネイティブモデル向け実装では全ての行列において CPU 向け実装よりも性能が低く，通信時間の増加が大きく影響する結果となった．

6. まとめ

SpMV を並列計算することにより多くの行列において性能の向上を達成できた．CPU 向け実装においては 64MPI プロセスで実行することで 1MPI プロセスの場合と比べて最大約 53.38 倍性能が向上した． しかし，並列化により処理にバラつきが発生し性能が低下しまうこともあり，行列の形状に大きく影響を受ける結果となった．

また，アクセラレータを用いることで行列と MPI プロセス数によっては CPU 以上の性能を達成できた．GPU 向け実装では非零要素数が多くなるほどデータ転送の影響が小さくなり，本来の演算性能を発揮することができた．MIC ではネイティブモデル向け実装は MPI プロセス数が少ない場合には本来の演算性能を発揮することができたが，MPI プロセス数の増加に伴い通信の影響を大きく受け性能が低下してしまった．またオフロードモデル向け実装は全体としては性能が低かったが，行列と MPI プロセス数次第では CPU 向け実装とネイティブモデル向け実装を上回る結果となった．

今後の課題として，現状では非零要素のバラつきを考慮していないため，異なる行の非零要素数のバラつきに対する MPI プロセス間の負荷分散を考慮する必要がある．また，MIC アーキテクチャにおいては資源を有効活用できる Symmetric モデルについても検討する必要がある．今回はホスト CPU とアクセラレータ間の負荷バランスが困難であったため割愛したが，適切に処理を割り当てることで MIC-Only モデル以上の性能が出ることも期待できる．

参考文献

[1] Michael, L., Gerard, G., Michele, W., Lawrence, M. and James, S.: Achieving Efficient Strong Scaling with PETSc Using Hybrid MPI/OpenMP Optimisation, *Proc. 28th International Supercomputing Conference (ISC 2013)*, Lecture Notes in Computer Science, Vol. 7905, Springer, pp. 97–108 (2013).

[2] Alexandersen, J., Lazarov, B. and Dammann, B.: *Parallel Sparse Matrix - Vector Product*, Technical University of Denmark (2012). IMM-Technical Report-2012.

[3] 工藤誠，黒田久泰，片桐孝洋，金田康正：並列疎行列ベクトル積における最適なアルゴリズム選択の効果，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， Vol. 2002, No. 22, pp. 151–156 (2002).

[4] Intel Corporation: インテル Xeon Phi コプロセッサー，Intel Corporation（オンライン），入手先 ⟨http://www.intel.co.jp/content/www/jp/ja/processors/xeon/xeon-phi-coprocessor.html⟩ （参照 2014-04-16）.

[5] Meuer, H., Strohmaier, E., Dongarra, J. and Simon, H.: TOP500, TOP500 (online), available from ⟨http://www.top500.org/⟩ (accessed 2014-04-16).

[6] 大島聡史，金子勇，片桐孝洋：Xeon Phi における SpMV の性能評価，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， Vol. 2013, No. 33, pp. 1–8 (2013).

[7] Teng, T. W., Jun, T. W., Rajarshi, R., Wen, W. Y., Weiguang, C., Shyh-hao, K., Mong, G. R. S., John,

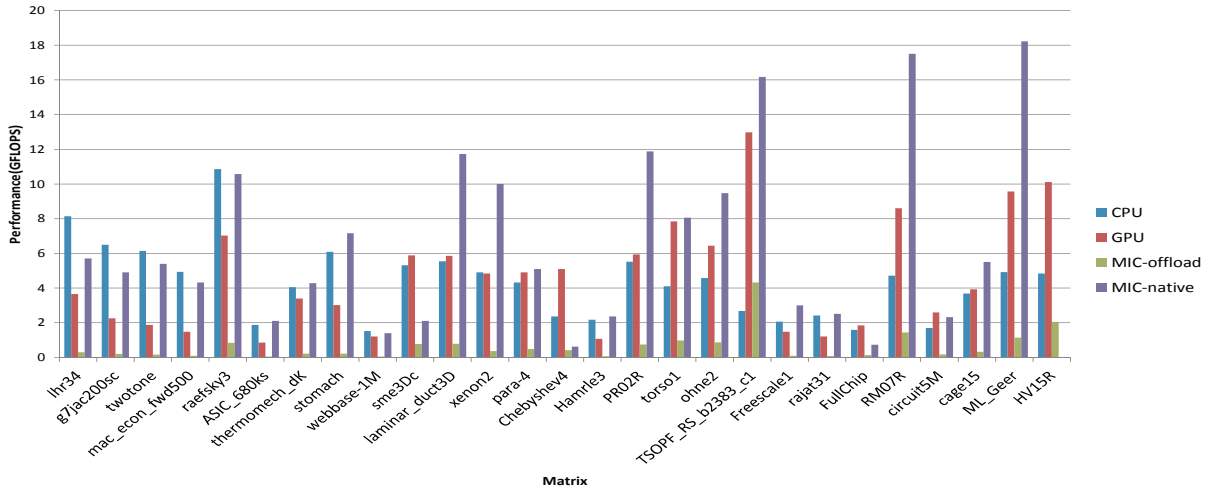

図 10 アーキテクチャ間の比較（1MPI プロセス）

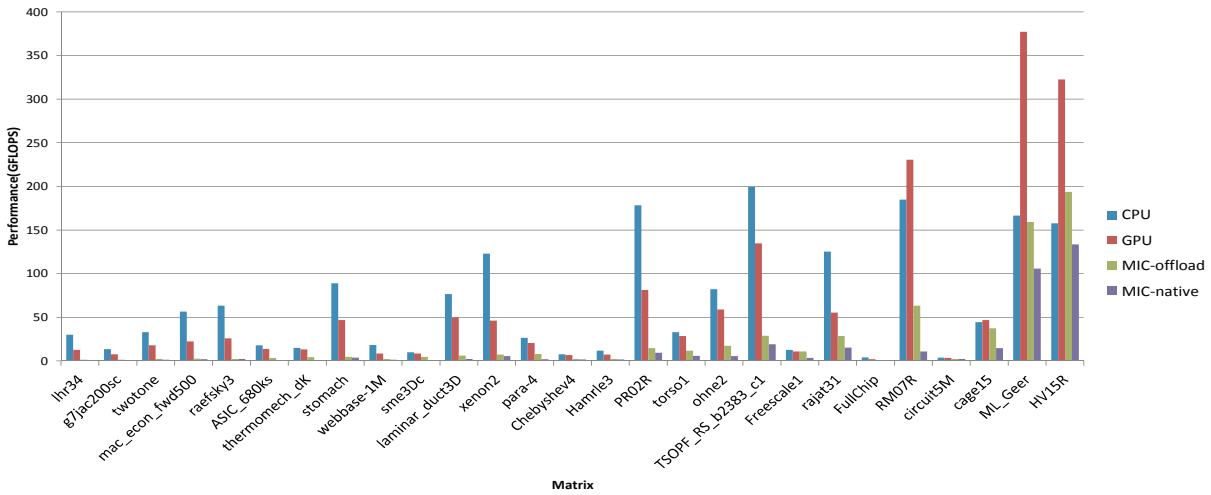

図 11 アーキテクチャ間の比較（64MPI プロセス）

T. S. and Weng-Fai, W.: Accelerating Sparse Matrix-vector Multiplication on GPUs Using Bit-representation-optimized Schemes, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis(SC'13)*, pp. 26:1–26:12 (2013).

[8] Alexander, M., Anton, L. and Arutyun, A.: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, Lecture Notes in Computer Science, Vol. 5952, Springer, pp. 111–125 (2010).

[9] Hroux, M., Dongarra, J. and Luszczek, P.: HPCG - Home, High Performance Conjugate Gradients (online), available from ⟨https://software.sandia.gov/hpcg/⟩ (accessed 2014-04-16).

[10] Ali, P. and Heath, M. T.: Improving Performance of Sparse Matrix-vector Multiplication, *Proc. 1999 ACM/IEEE Conference on Supercomputing* (1999).

[11] NVIDIA: 並列プログラミングおよびコンピューティングプラットフォーム— CUDA — NVIDIA — NVIDIA，NVIDIA（オンライン），入手先 ⟨http://www.nvidia.co.jp/object/cuda-jp.html⟩（参照 2014-04-19）.

[12] Intel Corporation: Intel Math Kernel Library, Intel Corporation (online), available from ⟨https://software.intel.com/en-us/intel-mkl⟩ (accessed 2014-04-21).

[13] NVIDIA: cuSPARSE — NVIDIA Developer Zone, NVIDIA (online), available from ⟨https://developer.nvidia.com/cuSPARSE⟩ (accessed 2014-04-19).

[14] Davis, T.: University of Florida Sparse Matrix Collection : sparse matrices from a wide range of applications, University of Florida (online), available from ⟨http://www.cise.ufl.edu/research/sparse/matrices/⟩ (accessed 2014-04-16).

[15] Intel Corporation: Intel Xeon Phi Coprocessor Software Ecosystem, Intel Corporation (online), available from ⟨https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Software_Ecosystem.pdf⟩ (accessed 2014-04-30).

[16] Ohio State University: MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, Ohio State University (online), available from ⟨http://mvapich.cse.ohio-state.edu/benchmarks/⟩ (accessed 2014-04-29).