Vol.2014-HPC-145 No.21
2014/7/29

様々な計算機環境におけるOpenMP/OpenACCを用いたICCG法の性能評価

大島 聡史[1,a)]　松本 正晴[1]　片桐 孝洋[1]　塙 敏博[1]　中島 研吾[1]

概要：今日のHPCにおいては高い演算性能と低い消費電力を達成するために様々な並列計算ハードウェアが用いられている．CPUは微細化と消費電力の制限等によりマルチコア化が進み，現在では1ソケットあたり10前後のコアを搭載したCPUが多く用いられている．GPUは非常に多くの計算コアと高いピーク性能を備え，CPUと比べると適したアプリケーションが限られるものの，様々なシーンにおいて活用が進んでいる．メニーコアプロセッサはCPUとGPUそれぞれのメリットを享受できるものとして注目が高まっている．一方で並列計算のためのプログラミング環境に着目すると，従来から使われているOpenMPやMPIの普及がますます進んでいる．さらに近年では従来のOpenMPでは対応できなかったGPUプログラミングについてもOpenMPのような指示文を用いた簡便な並列化プログラミングを可能にするべく，OpenMP 4.0やOpenACCが策定され，対応するコンパイラもいくつか公開・販売され始めている．そこで本稿では，様々な計算機環境を対象としてOpenMPやOpenACCを用いて同一の問題を実行し，性能やその傾向を調査し報告する．対象問題としては有限体積法アプリケーションにおけるICCG法を用いる．対象とする並列計算ハードウェアは以下の通りである：Intel Xeon (IvyBridge-EP), AMD Opteron (Piledriver), 富士通 SPARC64 IXfx, NVIDIA Tesla (Kepler), Intel Xeon Phi (Knights Corner).

1. はじめに

高い演算性能や電力当たり演算性能を得るために様々な並列計算ハードウェアが活用されている．CPUは微細化と消費電力の制限によりマルチコア化による性能向上が進んでおり，今日では1ソケットあたり10前後のコアを搭載したCPUがHPC用途には多く用いられている．GPUは，本来は画像処理用のハードウェアであったものの，高い並列計算性能やメモリバンド幅を持つ点が注目され,HPC分野での活用が進んでいる．今日ではワークステーションからスーパーコンピュータまで様々な計算環境においてGPUが活用されている．一方で現在のGPUはCPUと比べて汎用性に乏しい．GPUの備える計算コアはマルチコアCPUの備える計算コアと比べて1コアあたりの性能が低く，またGPUはCPUと組みあわせてアクセラレータとして利用する必要がある．さらにGPUを活用するにはGPU向けのプログラム開発環境を用いてGPU専用のプログラムを作成する必要もある．そこで,CPUの汎用性とGPUのピーク性能を同時に兼ね備えたハードウェアとして，メニーコアアーキテクチャが提案され,Intel社がXeon Phiとして製品化している (以下,MICと呼ぶ).

一方で，上記の並列計算ハードウェアにて用いられている主な並列化手法に着目してみると，マルチコアCPUにおいては従来から用いられているOpenMPとMPIによる並列化が引き続き行われている．MICでは既存のCPUと同様にOpenMPとMPIが利用可能である．さらにMICではMIC専用のオフロード指示文を用いたプログラミングも可能であるが，オフロード指示文は主にMICの制御とCPU-MIC間のデータ転送について記述するものであり，MIC上で実行される並列計算の記述にはOpenMPとMPIが利用されている．GPU向けのプログラミング環境については，CUDAやOpenCLが用いられているが，CPUとGPUで共通の開発方法や最適化が使えない，ソースコードの共通化がしにくい，などの課題もある．これに対して,GPUを含む様々な並列化環境を容易に利用できる並列化プログラミング環境として，指示文により並列化を行うOpenACCやOpenMP 4.0が策定された．OpenMP 4.0についてはまだ実装例が少ないが,OpenACCについては複数のコンパイラが対応を進めている．例えばPGI社のコンパイラはx86アーキテクチャのCPU,NVIDIA社のGPU,AMD社のGPUを対象としてOpenACCプログラムを作成可能である．

以上のように，今日のHPCにおいては特徴の異なる様々

[1] 東京大学 情報基盤センター
Information Technology Center, The University of Tokyo
[a)] ohshima@cc.u-tokyo.ac.jp

な並列計算ハードウェアが利用されているものの, ノード内並列化向けのプログラミング環境としては OpenMP と OpenACC が使えれば広範囲の計算環境を利用可能である. そのうえ,OpenACC と OpenMP は言語仕様レベルで同一のものではないものの, 並列化対象プログラム (ソースコード) の構造があまり複雑でないものであれば類似した指示文の挿入により並列化できるため,OpenMP と OpenACC の間での移行も容易である. しかしながら, 十分に高い性能を得るためにはそれぞれの言語仕様と計算機環境にあわせた最適化も必要となると考えられる.

そこで本稿では,OpenMP と OpenACC を用いて様々な並列計算ハードウェアにて同一の計算問題を実行し, 得られる性能や性能の傾向について調査する. 対象問題としては有限体積法 (FVM) アプリケーションにおける不完全修正コレスキー分解付き共役勾配法 (ICCG 法) を用いる.

本稿の構成は以下の通りである. 2 章では本稿で用いる計算機環境について説明する. 3 章では対象とするアプリケーションの内容と OpenMP および OpenACC による実装の内容について述べる. 4 章では性能評価結果を示し, 5 章はまとめの章とする.

2. 計算機環境

本稿では 6 種類の並列計算機環境を用いる. 本章では各ハードウェアの概要について述べる. 各ハードウェアの性能諸元は**表 1** の通りである.

2.1 CPU1: Intel Xeon E5 (IvyBridge-EP)

CPU1 として Intel 社の Xeon E5-2680 v2 [1] を用いた. 本 CPU は 2.8GHz の計算コアを 10 コア搭載しており, SMT(Simultaneous Multithreading) 機能 (Hyper-Threading) にも対応しているため 20 スレッド同時実行が可能である. ただし今回用いた環境においては Hyper-Threading は無効化されているため 10 コア同時実行まで可能となっている. 本 CPU は動作周波数が高速かつキャッシュ構成が他のハードウェアと比べてリッチな (階層が深く, 容量も多い)CPU である.

CPU1 向けのプログラム作成には OpenMP を用いた. コンパイル・リンクの際には Intel コンパイラ (icc 14.0.2) を使用し, *-O3 -openmp -ipo -xAVX* オプションを指定した.

2.2 CPU2: AMD Opteron (Piredriver)

CPU2 として AMD 社の Opteron 6386SE [2] を用いた. CPU2 は, 動作周波数こそ CPU1 と同様に 2.8GHz であるが, コアの構成には大きな違いがある. 本 CPU は計算コア 8 コアごとにクラスタ化されており,2 クラスタ=16 計算コアで 1 ソケットを構成している.

CPU2 向けのプログラム作成には OpenMP を用いた. コンパイル・リンクの際には PGI コンパイラ (pgcc 14.6) を使用し, *-fastsse -mp=numa* オプションを指定した.

2.3 CPU3: FUJITSU SPARC64 IX

CPU3 として富士通社の SPARC64 IXfx を用いた. 本 CPU は東京大学情報基盤センターに設置されている Oakleaf-FX[3] に搭載されている CPU である. 本 CPU は 1.848GHz の計算コアを 16 コア搭載しており,SMT 機能は搭載していない. 本 CPU は他の CPU と比べると動作周波数が低くキャッシュ階層も低い CPU であるが, コア数が多いため CPU 1 ソケットあたりの演算性能は高く, メモリバンド幅も高い.

CPU3 向けのプログラム作成には OpenMP を用いた. コンパイル・リンクの際には Oakleaf-FX 向けに導入されている標準コンパイラ (Fujitsu C/C++ Compiler Driver Version 1.2.1) を使用し, *-Kfast,openmp* オプションを指定した.

2.4 MIC: Intel Xeon Phi (Knights Corner)

MIC として Intel 社の Xeon Phi 5110P [4] を用いた. 本ハードウェアは動作周波数 1.053GHz の計算コアを 60 コア搭載しており, SMT 機能によりコアあたり 4 スレッド, 合計 240 スレッドを同時実行することができる. その代わり CPU 群と比べると動作周波数が低速であり, キャッシュは階層・容量ともに弱い. しかしコア数が多いために理論演算性能が高く, また高速な GDDR5 メモリを搭載しているためメモリバンド幅も高い.

MIC 向けのプログラム作成にはネイティブモデルとオフロードモデルの 2 種類の方法を用いた.

ネイティブモデルは既存のマルチコア CPU と同様に OpenMP や MPI を利用できるモデルであり, プログラムは MIC 上で起動されて全て MIC 上で実行される. ネイティブモデル向けのプログラム作成には OpenMP を用い, コンパイラは CPU1 同様の Intel コンパイラ (icc 14.0.2) を *-O3 -openmp -ipo -mmic* オプションとともに利用した.

オフロードモデルはオフロード対象として指定されたプログラムの一部分のみを MIC 上で実行するモデルであり, プログラムは CPU 上で起動されてオフロード対象のみが MIC 上で実行される. オフロード実行時には MIC 上のコアの 1 つが制御のために利用されてしまうため最大 59 コア 236 スレッド同時実行となる点に注意が必要である. オフロードモデルにおいては, オフロード実行に伴う動作の制御は専用の指示文を用いて記述し (3.4 節にて後述), オフロード対象部分の並列実行には OpenMP を用いる. プログラムのコンパイル・リンクにはネイティブモデル同様に Intel コンパイラを (icc 14.0.2) を利用した. コンパイルオプションは*-offload-attribute-target=mic -opt-threads-per-core=* -O3 -openmp -ipo -xAVX* (*にはコアあたりのスレッド数を指定する) とした.

表 1 実験環境

	CPU1	CPU2	CPU3
名称	Xeon E5-2680 v2 (IvyBridge-EP)	Opteron 6386 SE (Piledriver)	SPARC64 IXfx
動作周波数	2.8 GHz	2.8 GHz	1.848 GHz
コア数（有効スレッド数）	10 (20)	16	16
メモリ種別	DDR3	DDR3	DDR3
キャッシュ構成	L1 32KB+32KB/core L2 256KB/core L3 25MB/socket	L1 64KB+64KB/core L2 8x2MB/socket L3 16MB/socket	L1 32KB/core L2 12MB/socket
理論演算性能	224 GFLOPS	179.2 GFLOPS	236.5 GFLOPS
理論メモリ性能	59.7 GB/s	51.2 GB/s	85 GB/s
TDP	130W	140W	110W

	MIC	GPU
名称	Xeon Phi 5110P (Knights Corner)	Tesla K40 (Kepler)
動作周波数	1.053 GHz	745 MHz
コア数（有効スレッド数）	60 (240)	2880
メモリ種別	GDDR5	GDDR5
キャッシュ構成	L1 32KB/core L2 512KB/core	OnChip 64KB SMX R/O 48KB/SMX L2 1536KB/card
理論演算性能	1.01 TFLOPS	1.43 TFLOPS
理論メモリ性能	320 GB/s	288 GB/s
TDP	300W	235W

2.5 GPU: NVIDIA Tesla K40 (Kepler)

GPU として NVIDIA 社の Tesla K40 [5] を用いた．本 GPU は計算コア 196 コアからなるユニット (SMX) を 15 基搭載した GPU であり, 動作周波数は 745MHz である．合計計算コア数は 2880 であるが, 本 GPU はメモリアクセスのレイテンシをスレッド切替えにより隠蔽することで高い性能を得るアーキテクチャであるため, 十分な性能を得るためには対象プログラムに 2880 を越える高い並列度が必要である．

GPU 向けのプログラム作成には OpenACC を用いた．コンパイル・リンクの際には PGI コンパイラ (pgcc 14.6) を使用し, `-fast -acc -Minfo=accel -ta=nvidia:cc3+` オプションを指定した．

3. 実装

本章では性能評価に用いたアプリケーションの概要と OpenMP および OpenACC を用いた実装の内容について述べる．

3.1 対象アプリケーション

対象とするアプリケーションは有限体積法 (Finite Volume Method, FVM) プログラムであり, 差分格子によってメッシュ分割された三次元領域においてポアソン方程式を解くものである．同プログラムを対象とした性能評価については, 共著者の一人である中島による Fujitsu FX10 (Oakleaf-FX, CPU3 と同様) および Cray XE6 において実施した例などが報告されている [6]．本稿で用いている問題設定は [6] とほぼ同様であるため, 対象アプリケーションの詳細や FX10 上での詳しい性能評価については [6] も参照されたい．

本アプリケーションが対象とする形状は規則正しい差分格子であるが, プログラムの実装としては非構造格子型のデータとして扱うことにより一般性を持たせている．本問題の求解においては, 最終的にはメッシュ数 N に対して N 個の方程式による連立一次方程式 $Ax=b$ を解く問題へと帰着する．三次元形状を直方体メッシュで解く都合上, 各メッシュに対応する非対角成分の数は最大 6 個となるため, 係数行列 A は疎 (sparse) な行列となる．

係数行列 A は対称かつ正定であるため, 前処理付き共役勾配法を適用する．また前処理には対象行列向けに広く使用されている不完全コレスキー分解を使用する．係数行列は対称であるものの, 本プログラムでは対象行列専用の格納形式や計算方法は用いず, 係数行列を上下三角成分を別々に格納するという方式を採用している．不完全コレスキー分解においては係数行列 A を不完全 LU 分解して前処理行列 M を生成し, この M を用いて共役勾配法を実施する．係数行列が疎行列である場合には LU 分解における fill-in についても考慮する必要があるが, 実用的には fill-in を考慮しない IC(0)(括弧内は fill-in のレベル) でも多くの問題に対応できることから, 本稿では IC(0) を使用する．すなわち, 係数行列 A と前処理行列 M では非ゼロ成分の位置が同じとなる．

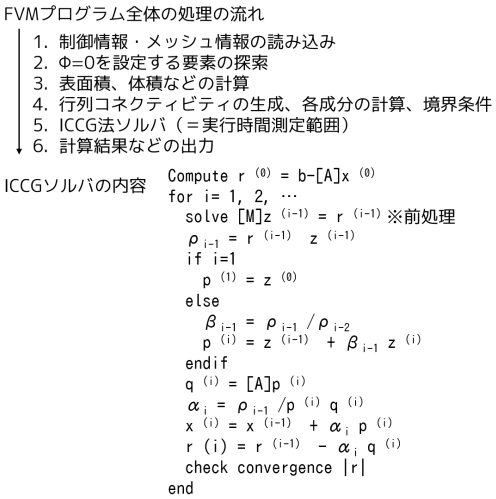

図 1 FVM プログラムと ICCG 法の処理の流れ

不完全コレスキー分解を前処理に用いる共役勾配法を ICCG 法と呼ぶ．対象とする FVM プログラムと ICCG 法の処理の流れを**図 1** に示す．ICCG 法は行列やベクトルに対する加算, 乗算, 集約などの演算を繰り返し行う計算方法であり, 特に長い実行時間を必要とするのは疎行列ベクトル積と前処理である．また不完全コレスキー分解時, 前進代入, 後退代入においてメモリへの書き込みと参照が同時に生じるため, これらを並列化するためには依存性のない計算対象を抽出する必要があり, 色づけによるリオーダリングを用いた並列性の抽出 (カラーリング) が必要である．

今回の対象プログラムにおいては,

マルチカラー法 (Multicoloring, MC) いわゆる Red-Black 法の多色版, 高い並列度やスレッド間の負荷分散を達成できるが, 悪条件問題では収束が悪化する．色数を増やし色ごとの要素数を減らすと OpenMP のオーバーヘッドが露見することがある．

Cuthill-Mckee 法 (CM) 隣接点数を元にレベル付けを行い, レベル順に再番号付けをする手法．MC 法と比べて, 同一色・同一レベルにおける各要素の独立性に加えて計算順序を考慮したレベル間の依存性が考慮される．

Reverse Cuthill-Mckee 法 (RCM) CM 法に加えてレベルの少ない順に並び替える処理が加わった手法. 悪条件に対しても収束性が良い．ただし, レベル毎の要素数が不均一になりやすいため並列ソルバ実行時の性能が低下することがある．

Cyclic Multicoloring RCM 法 (CM-RCM) RCM に対してサイクリックな再番号付けを行う手法. 並列ソルバ実行時の性能改善が期待される．

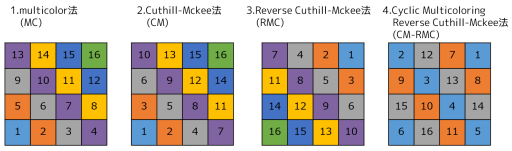

図 2 実装されているカラーリング手法

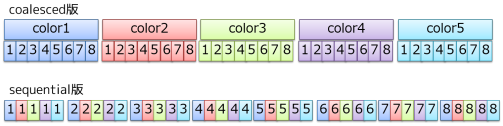

図 3 シーケンシャル版とコアレス版 (5 色 8 スレッドを想定したイメージ)

の 4 種類の色づけ手法が実装されている (**図 2**). カラーリングの結果,ICCG 法を実行する際にメモリアクセス順序などが変化してキャッシュヒット率に違いが生じるため, 収束性や性能 (実行時間) にも差が生じる.

さらに本プログラムには性能向上を目指していくつかの最適化実装が適用されている.

カラーリング後に色ごとに並列計算を行う際に, マルチコア CPU では各色ごとにデータが連続しているよりもスレッドごとに扱うデータが連続している方がキャッシュやプリフェッチの機能が有効に働き高い性能が期待できる. 一方で GPU のように複数のスレッドが連続・近接したメモリアドレスを一度に参照することで高いメモリアクセス性能が得られるアーキテクチャの場合には各色ごとにデータが連続している方が高い性能が期待できる. そのため本プログラムでは行列コネクティビティ生成時に上述した 2 種類のメモリアドレス参照それぞれに対応した実装を切り替えて利用することができる. 本稿では前者 (スレッドごとに扱うデータが連続する) をシーケンシャル版 (グラフ中の表記では sequential 版), 後者 (色ごとに扱うデータが連続する) をコアレス版 (グラフ中の表記では coalesced 版) と呼ぶ (**図 3**).

NUMA アーキテクチャ環境においては計算コアとメモリの配置の都合上, 異なる CPU ソケットに接続されたメモリ (変数や配列) を参照するとメモリアクセス性能が低下する. 一方でプログラムにおいて確保されたメモリは, メモリが確保された瞬間ではなく, メモリへのアクセスが初めて行われた際にアクセスを行った CPU コアの属するソケットのローカルメモリに割り当てられる. そこで, 各メモリを確保した後に実際に計算を行う際と同様のアクセスパターンによるメモリアクセスを行えば, 実際に計算を行う際のメモリアクセス性能低下を防ぐことができる. これは First-Touch として広く知られている手法である. NUMA アーキテクチャにおけるコア割り当てを制御する numactl と組み合わせて適切に使用することも重要である.

疎行列ベクトル積 (SpMV) を初めとする疎行列計算の性能は, 行列を構成する非ゼロ要素の配置や疎行列を格納

図 4 疎行列格納形式

するデータ構造 (疎行列格納形式) に大きな影響を受ける. 本プログラムでは疎行列格納形式として Compressed Row Storage 形式 (CRS 形式) を用いている (**図 4**). CRS 形式は疎行列を行方向に圧縮し, 非ゼロ要素の列番号と値および行頭要素の出現位置を記録する形式である. 本形式は非ゼロ要素を一切保持しないためメモリ効率が良く, また SpMV のような計算を行う際に行ごとのメモリアクセスがしやすいことから, 多くのアプリケーションにおいて利用されている.

一方で, 対象とする疎行列の形状特性によっては CRS 形式よりも高い SpMV 性能を得ることができる行列格納形式として, Ellpack 形式 (ELL 形式) が知られている. ELL 形式は疎行列を行方向に圧縮した後に列方向へとメモリへ格納する形式である. ELL 形式は行ごとの非ゼロ要素数が異なる場合にゼロで埋める形式であるため CRS 形式と比べてメモリ効率の面で不利となりやすいが, SpMV 計算においてはベクトル長の大きなループに対する計算を行うことになるため, 特に行あたりの非ゼロ要素数が小さな大規模疎行列においては CRS 形式よりも高い SpMV 性能が期待される. またベクトルの加算や集約など ICCG 法にて必要な計算においても不利とはならない. 本プログラムにおける問題設定においては各行に配置された非ゼロ要素の数は対象問題の物理特性により最大 6 までと小さいため, ELL 形式を用いることで CRS 形式よりも高い性能が得られることが期待される. そのため本プログラムは CRS 形式の代わりに ELL 形式を用いる実装を備えている. 行列形状の選択は行列コネクティビティ生成時に行われる. 本稿では CRS 形式の行列を用いた ICCG 法と ELL 形式の行列を用いた ICCG 法をそれぞれ CRS 版および ELL 版と呼ぶ. なお,ELL 版については CM,RCM,CM-RCM 法のみが実装されている.

3.2 OpenMP による並列化実装 (CPU,MIC 向け)

OpenMP による実装は,ICCG 法における各種の行列やベクトルに対する計算を OpenMP 化したものである. For-

tran や C 言語では行列やベクトルの演算はループ構造で表現されるため, 各ループを omp do や omp for といった OpenMP のループ並列化指示文により並列化されている.

3.3 OpenACC による並列化実装 (GPU 向け)

OpenACC による GPU 向けの実装については,OpenMP プログラムにて並列化されていた部分を GPU に実行させる, という方針で行った. また,OpenACC 2.0 から導入された新たな指示文・機能である *enter data* と *exit data* を用いて CPU-GPU 間データ転送の明確化および不要データ転送が行われることを防いでいる.

図 5 の (1) と (2) に OpenMP プログラムと OpenACC プログラムの対応を示す. 図中に示すように, 本稿における OpenACC プログラムの実装は OpenMP による実装と似通っている. OpenMP における *parallel* 節や *for* 節といった並列化対象の指定は OpenACC における *kernels* 節や *loop* 節に置き換えられている. OpenMP では *private* 節によって制御していた変数や配列のふるまいについては, OpenACC ではループ外にて *enter data* 節で GPU へデータを転送し, ループ内の *present* 節で転送済のデータを使用することを明示している. ループの並列化については,GPU 上の多数の計算コアを十分に使い切れるだけの高い並列度を得るために, OpenMP では並列化を行っていなかった内側のループについても並列化されるように指示文の挿入を行っている. また *gang* 節や *vecotr* 節を記述することによって, GPU の備える階層的な並列性, すなわち NVIDIA GPU における CUDA Core 単位の並列性と SMX や WARP 単位の並列性や, および AMD GPU における CU 内の並列性と CU 単位での並列性を活用しやすくなる. ただし,PGI コンパイラにて OpenACC を使う場合, OpenACC 指示文にて並列化対象としたループ内部に依存性がないことが自明でない処理 (配列の読み書き) がある際には並列化が行われない. そこで本稿では, 図中に示すように *independent* 節を記述することで強制的に並列化を行わせている.

3.4 オフロード実装 (MIC 向け)

MIC 向けのオフロード実装については,ICCG 法内の反復計算内のほとんどの計算を MIC にオフロードし, 収束判定のみをホスト CPU で行うという実装を採用した. もちろん ICCG 法内の反復計算そのものを全てオフロードするという実装も可能である. 最大性能を求めるのであれば, ホスト CPU と MIC のどちらに適した計算が含まれるかや CPU-MIC 間のデータ転送に必要な時間を勘案して最適なオフロード位置を定めるべきである. 今回は ICCG 法の全てをオフロードした場合にはネイティブモデルによる実行と実質的に変わらなくなるうえに, オフロード実行の場合には有効な物理コア数が 1 減少し性能が低下することが予想されたため, このような実装を選択した.

3.5 MIC 向けの性能改善手法

詳細は次章にて述べるが,MIC 上で実行した OpenMP プログラムの性能は他のハードウェアと比べて非常に低いものであった. そこで,MIC については容易に効果が得られる性能改善手法を適用した. 適用内容は, メモリのアラインメントの調整と SIMD 指示文の挿入である。すなわち,C 言語プログラムにおいて一般的な malloc 関数を用いたメモリ確保の代わりに, _mm_malloc 関数を用いたメモリの確保を行い, 全ての動的メモリが 64byte 境界に揃うようにした. 64byte 境界に揃えられたメモリは, MIC にて用いられる SIMD 命令 (AVX-512) の適用条件や MIC 上のキャッシュラインサイズに適合するため, 演算性能やメモリアクセス性能が向上し, 全体の実行時間が短縮されることが期待される. さらに *omp for* 指示子に *simd* 節を付加して SIMD 化の促進をはかった.

4. 性能評価

本章では性能評価の結果について示す. 問題設定については, 三次元の各方向の分割数を 100 とした (100x100x100=1,000,000 要素). 評価対象とした計算機環境は表 1 に示したとおりである. また測定範囲については図 1 に示すように ICCG 法全体である. OpenACC やオフロード実行については CPU-GPU 間や CPU-MIC 間のデータ転送時間が含まれている.

スレッド数については,CPU は物理コア数と等しく,MIC については物理コアあたり 4 スレッド, すなわちネイティブ実行では合計 240 スレッド, オフロードでは 236 スレッドを用いた. ICCG 法は演算性能ではなくメモリ性能がボトルネックとなるアプリケーションであるためスレッド数をやや減らした方が性能が向上する可能性がある. MIC においては実例が確認できているが, 性能差は 10%程度未満とそれほど大きくはなく, 性能の傾向が全く異なるものではないため, 今回はスレッド数の最適化については議論しない.

OpenACC における並列度の指定については, 特に具体的な値は設定せず,OpenACC の処理系に任せた. ただし ICCG プログラムの設計上, 色ごとの対応する要素を計算する並列化ループ部分の並列度はプログラム実行時 (実行パラメタを与える入力ファイル) にて決め打ちとなる. そこで,GPU 上の演算器群 (15 基の SMX) が十分に使い切れるだけの値として 120 を与えた.

性能測定結果を**図 6**, **図 7**, および**図 8** に示す. なお, 今回は全体の傾向と最速ケースの性能に着目したため, 色数の多い MC 法を中心に一部のデータがグラフに収まっていないことを付記しておく.

はじめに全体の性能を俯瞰する. 全体に共通した傾向として,MC 法の性能は他の色づけ方法を用いたものと比べて性能が低めである. また色数が多い場合に特に性能が低下

```
// ICCG法内の反復計算
for(……){

// ソルバー内 SpMV実行部分

#pragma omp parallel for private (ip, i, VAL, j)

for(ip=0; ip<PEsmpTOT; ip++) {

   for(i=SMPindexG[ip]; i<SMPindexG[ip+1]; i++) {
      VAL = D[i] * P[i];
      for(j=indexL[i]; j<indexL[i+1]; j++) {
         VAL += AL[j] * P[itemL[j]-1];
      }
      for(j=indexU[i]; j<indexU[i+1]; j++) {
         VAL += AU[j] * P[itemU[j]-1];
      }
      Q[i] = VAL;
   }
}

if(……){……} // 収束判定
} // 反復計算の終了
```

(1) OpenMPによる実装

```
// ICCG法ループ外
#pragma acc enter data ¥
 copyin (AL[0:NPL], itemL[0:NPL], ……) ¥
 create (P[0:N], Q[0:N])

// ICCG法内の反復計算
for(……){

// ソルバー内 SpMV実行部分
#pragma acc data present(P, Q, D, indexL, itemL, AL, ……)
#pragma acc kernels
#pragma acc loop independent gang
for(ip=0; ip<PEsmpTOT; ip++) {
#pragma acc loop independent vector
   for(i=SMPindexG[ip]; i<SMPindexG[ip+1]; i++) {
      VAL = D[i] * P[i];
      for(j=indexL[i]; j<indexL[i+1]; j++) {
         VAL += AL[j] * P[itemL[j]-1];
      }
      for(j=indexU[i]; j<indexU[i+1]; j++) {
         VAL += AU[j] * P[itemU[j]-1];
      }
      Q[i] = VAL;
   }
}

if(……){……} // 収束判定（ホストCPU上）
} // 反復計算の終了
```

(2) OpenACCによる実装

```
// ICCG法ループ外
#pragma offload_transfer target(mic:0) ¥
in (AL[0:NPL], itemL[0:NPL], …… ¥
: alloc_if(1) free_if(0) align(64)) signal(AL)

// ICCG法内の反復計算
for(……){

#pragma offload target(mic:0) ¥
nocopy(AL, itemL, ……) ¥
{ // ここからオフロード対象（mic上で実行）

// ソルバー内 SpMV実行部分
#pragma omp parallel for private (ip, i, VAL, j)
for(ip=0; ip<PEsmpTOT; ip++) {
   for(i=SMPindexG[ip]; i<SMPindexG[ip+1]; i++) {
      VAL = D[i] * P[i];
      for(j=indexL[i]; j<indexL[i+1]; j++) {
         VAL += AL[j] * P[itemL[j]-1];
      }
      for(j=indexU[i]; j<indexU[i+1]; j++) {
         VAL += AU[j] * P[itemU[j]-1];
      }
      Q[i] = VAL;
   }
}
} // オフロード対象の終了
if(……){……} // 収束判定（ホストCPU上）
} // 反復計算の終了
```

(3) オフロード向けの実装

図 5 OpenMP プログラム,OpenACC プログラム, オフロードプログラムの比較

する傾向が確認できる．これらの傾向は [6] などにて確認済の傾向と一致している．

つづいて, それぞれの計算機環境ごとの性能を確認する．

CPU1 では coalesced 版,sequential 版,ELL 版のいずれもあまり大きな性能の差はないが, ELL 版におけるある程度色数の多い CM-RCM 法や CM 法,RCM 法の性能がやや良いことが確認できる．CPU2 では coalesced 版の性能がやや低く,sequential 版と ELL 版の性能はほぼ同等であった．CPU3 の性能は,[6] にて報告されている通り, coalesced 版と sequential 版の性能には目立った差はなく,ELL 版の性能が良い．このように,CPU ごとにやや異なる性能の傾向を示した．

CPU 同士の性能を比較すると, いずれの実装についても最も高速な CPU は CPU1 である．CPU2 の実行時間は CPU1 と比べると 2 倍程度かかっている．CPU1 と CPU2 の理論上の演算性能やメモリ性能にはそこまで大きな差はないため, アーキテクチャ上の特徴もしくはコンパイラの差が影響しているものと考えられる．一方で CPU3 については, 最も高速である ELL 版の性能において CPU1 に匹敵している．理論上の演算性能やメモリ性能では CPU3 が最も高性能であるため, それぞれのコンパイラ向けに最適なコンパイラオプションや環境変数を精査すれば CPU3 が最速となる可能性は十分に考えられる．

つづいて MIC の性能に着目すると,MC 法の性能が他の色づけより遅く色数が増えると性能低下することや ELL 版の性能が良いといった傾向は CPU 群と同様に確認できた．一方, 全体の実行時間は CPU 群の 10 倍程度遅かった．そこで,3.5 節に示したように,64byte 境界に適合させる最適化を適用し, さらに OpenMP により並列化しているループに対して simd 節を加えて SIMD 化を促進した．また, これらの最適化を施したプログラムと同様の計算カーネルを持つオフロード実行版のプログラムも作成し性能を比較した．結果を図 7 に示す．今回は ELL 版の実装が間に合わなかったため,coalesced 版と sequential 版の性能のみを示している．最適化の結果, 実行時間は CPU 群と遜色ないレベルにまで引き上げられた．MIC の性能に対してメモリアクセスの最適化や SIMD 化が非常に大きな影響を持つことが確認できた．ただし, ネイティブモデルとオフロードモデルで coalesced 版と sequential 版の性能の逆転が起きていたり, CM 法や RCM 法の性能がふるわない, 色数の多い CM-RCM 法の性能低下が大きいなど性能の傾向については気になる点も残っており, 今後さらなる性能の調査が必要である．

最後に GPU の性能に着目する．GPU の性能は, 大まかに見て CPU2 と同程度であり, 特別良い性能は得られていない．またアーキテクチャに適したメモリアクセスパターンという観点から, GPU は sequential 版よりも coalesced 版のほうが良い性能が得られることを予想していたが, 結果としてあまり高い性能は得られなかった．ELL 版の性能についても, ほとんど差は生じなかった．

5. おわりに

本稿では三次元有限体積法によるポアソン方程式ソルバーから得られる連立一次方程式を OpenMP および OpenACC によって並列化された ICCG 法によって解き, 性能評価を行った．対象計算機環境としては, Intel Xeon (IvyBridge-EP), AMD Opteron (Piledriver), 富士通 SPARC64 IXfx, NVIDIA Tesla (Kepler), Intel Xeon Phi (Knights Corner) と多様な環境を用いた．

測定結果を比較した結果, 全体としては CPU1 (IvyBridge-EP) と CPU3 (SPARC64 IXfx) の実行時間が短く, CPU2 と GPU は, 非常に遅いわけではないにせよ,CPU1

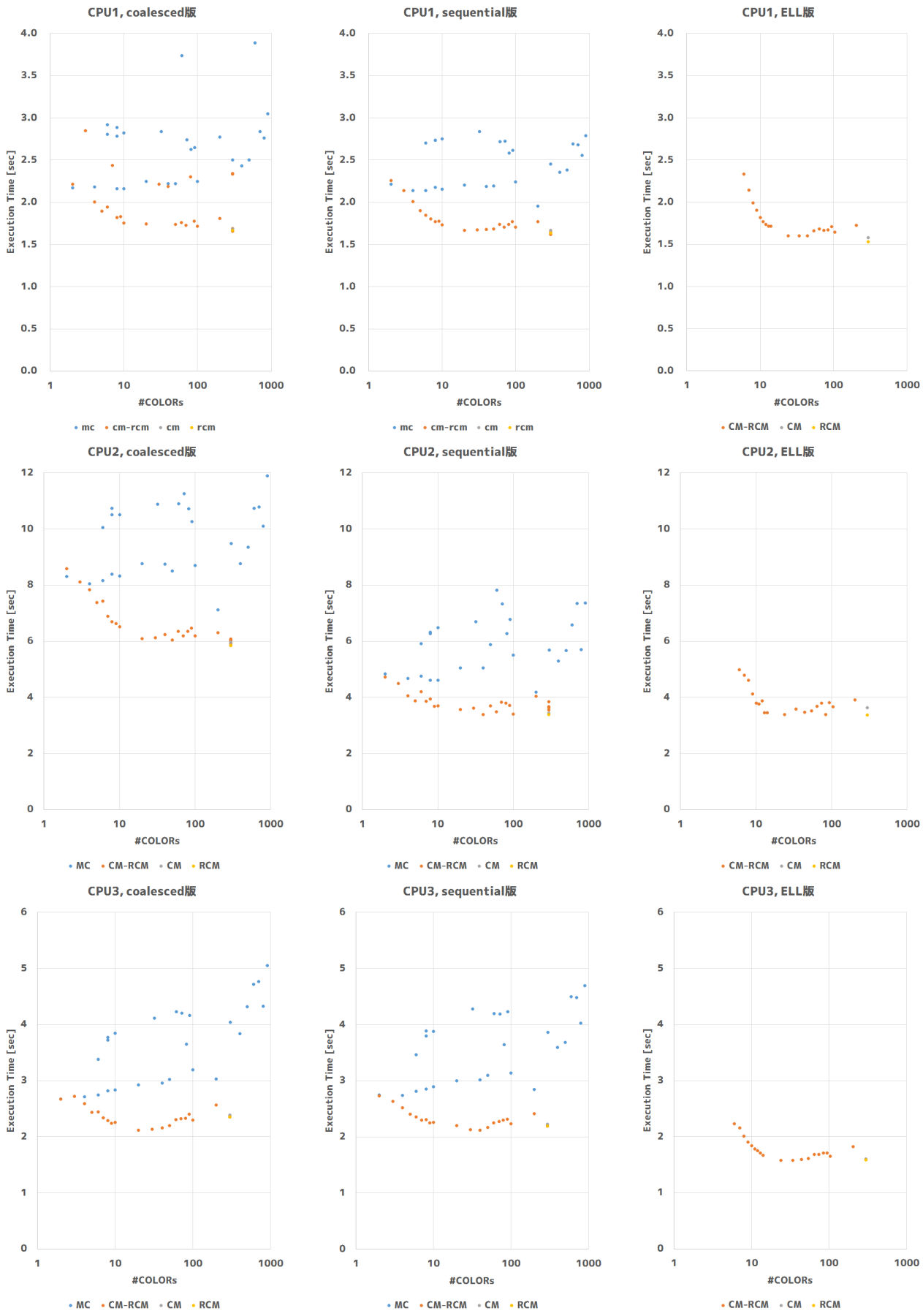

図 6　性能評価結果 (CPU)

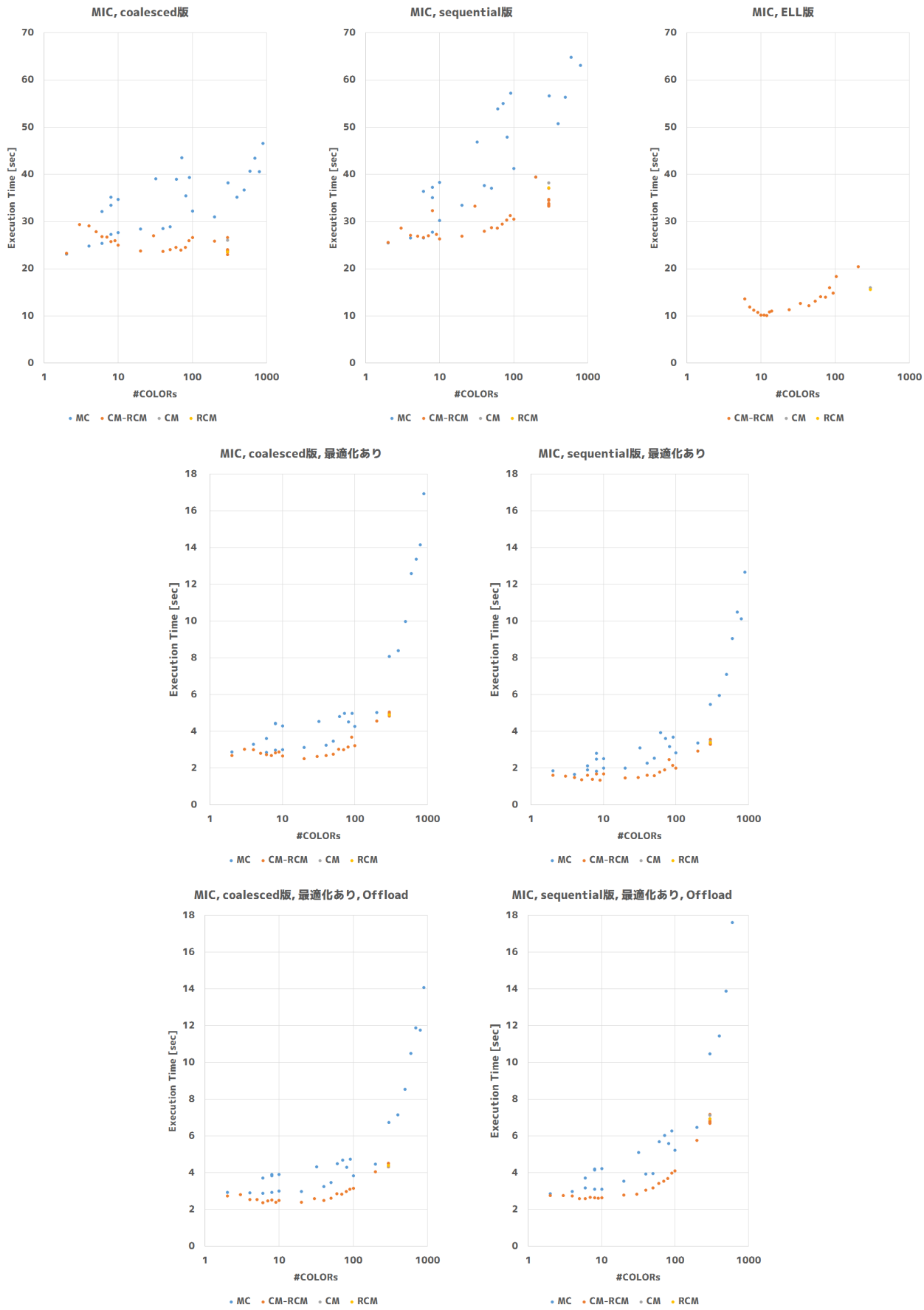

図 7　性能評価結果 2 (MIC)

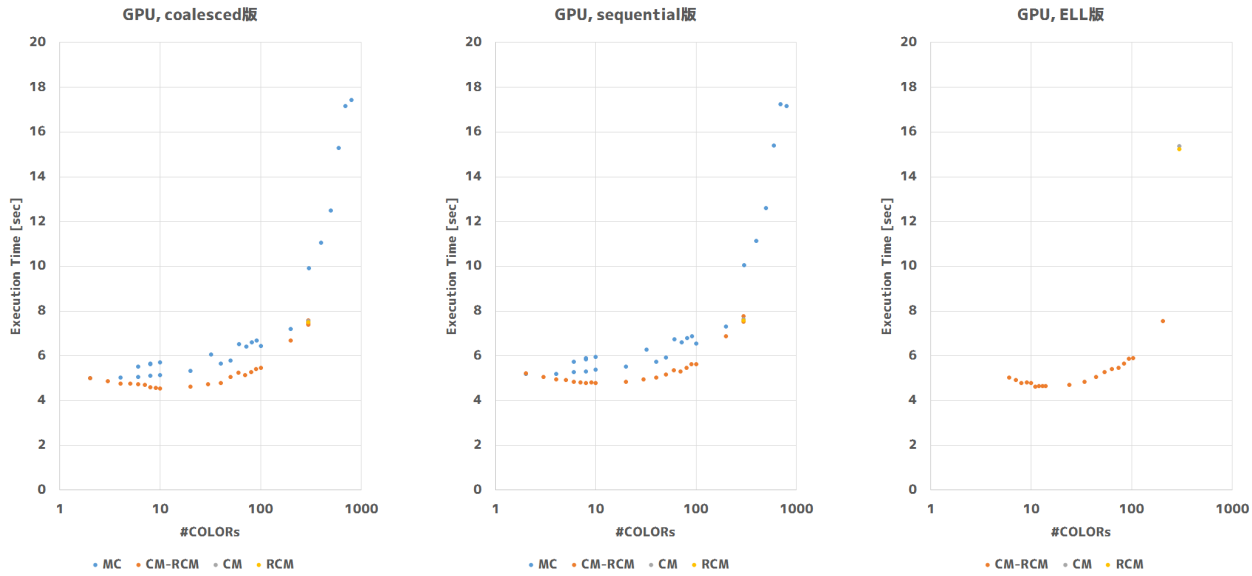

図 8　性能評価結果 3 (GPU)

と CPU3 には遅れる結果となった．MIC については,CPU と完全に同一のコードを用いた場合は非常に遅かった一方で, メモリのアラインメントを調整するなど少しの最適化を施した場合には他のハードウェアと遜色ないレベルの性能となった．

本稿では,MIC におけるメモリ境界の調整をのぞいて各計算機環境に特化した性能最適化は行わなかった．それぞれの環境においてさらに高い性能を得るには, 例えばコンパイラオプションや実行時環境変数の精査といったソースコードの改変を伴わない最適化, アンローリングや SIMD 化など場合によってはソースコードの改変も必要となる最適化, さらには配列の確保の仕方や配列処理の細かな調整などソースコードの改変がほぼ必須となる最適化など様々な方法が考えられる．また GPU と OpenACC については, OpenACC を用いることにより OpenMP 程度の容易さで CPU 並の性能が得られるのは良いことではあるが, そもそも CPU を上回る性能が得られないのでは, 素直に CPU を使った方が良い．OpenACC の最適化についてはまだ未知の部分も多いため, 今後さらに最適化手法の調査が必要であろう．また GPU アーキテクチャが異なる場合の評価として, AMD 社の GPU を用いた性能評価も実施中である．

今後は特定の計算機環境や問題設定に絞った性能最適化や, 計算機環境ごとの性能の傾向の違いについての分析などを実施する予定である．

謝辞　日頃より最適化プログラミングについて議論をさせていただいている東京大学情報基盤センタースーパーコンピューティング研究部門の皆様に感謝します．本研究は JSPS 科研費 24300004(実行時自動チューニング機能付き疎行列反復解法ライブラリのエクサスケール化), JST CREST「自動チューニング機構を有するアプリケーション開発・実行環境:ppOpen-HPC」の助成を受けたものです．

参考文献

[1] Intel Xeon Processor E5-2680 v2 http://ark.intel.com/ja/products/75277/
[2] AMD Opteron Processor (Opteron 6386 SE) http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=814
[3] FX10 スーパーコンピュータシステム (Oakleaf-FX) website, http://www.cc.u-tokyo.ac.jp/system/fx10/
[4] Intel Xeon Phi Coprocessor 5110P http://ark.intel.com/ja/products/71992
[5] NVIDIA Tesla GPUs (Tesla K40) http://www.nvidia.com/object/tesla-servers.html
[6] 中島研吾: 前処理付きマルチスレッド並列疎行列ソルバー, 情報処理学会 研究報告 (2013-HPC-139), No.6, pp.1-6 (2013).
[7] 中島研吾: T2K オープンスパコン (東大) チューニング連載講座 (その 5), OpenMP による並列化のテクニック: Hybrid 並列化に向けて, スーパーコンピューティングニュース (東京大学情報基盤センター)11-1 (2009).
[8] Saad, Y.: Iterative Methods for Sparse Linear Systems Second Edition, SIAM (2003).