Vol.2015-HPC-150 No.14
2015/8/4

# Xeon Phiにおける並列2分法による固有値計算の性能評価

石上 裕之[1,2,a)]　木村 欣司[1,b)]　中村 佳正[1,c)]

**概要**：2 分法による実対称 3 重対角行列向け固有値計算について, メニーコアプロセッサ Intel Xeon Phi (Xeon Phi) 上での実装とその性能評価について述べる. 本研究では, LAPACK の 2 分法ルーチン DSTEBZ に対して OpenMP によるスレッド並列化を施し, その並列 2 分法コードから Xeon Phi 向けの実装コードを開発した. マルチコア CPU と 1 台の Xeon Phi を搭載した計算機上での数値実験では, 開発した Xeon Phi 向けの実装コードが良好な性能を示すことを確認した. 更に, マルチコア CPU や GPU といった他の並列計算ハードウェア上での 2 分法による固有値計算の性能とも比較を行い, Xeon Phi 上での 2 分法の性能の優位性を確認した.

## 1. はじめに

$n$ 次実対称密行列 $A$ に対する標準固有値問題

$$A\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i, \quad i = 1, \ldots, n$$

を考える. ここで, $\{\lambda_i\}_{i=1}^n$ は $A$ の実固有値 ($\lambda_1 \leq \cdots \leq \lambda_n$), $\{\boldsymbol{v}_i\}_{i=1}^n$ は $\lambda_i$ に対応する実固有ベクトルである. 以下では, 対応する固有値と固有ベクトルの組を固有対と呼ぶ.

実対称密行列の固有対計算は多くの科学技術計算で現れる基本的な線形計算で, 全ての固有対が要求される場合もあれば, 一部の固有対のみが要求されることもある. 近年では, 解くべき固有対問題も大規模化しており, 並列計算による高速化は欠かせないものとなっている. その一手法として近年, 元来画像処理のために用いられていた GPU や Intel 社の開発した MIC アーキテクチャといったアクセラレータへの注目が大きい. 特に, MIC アーキテクチャは CPU 向けに C 言語や Fortran 言語で実装されたコードを大きく改変することなく使用できるという利点がある. MIC アーキテクチャで構成されるコプロセッサ Xeon Phi には, 直接 Xeon Phi 上でプログラムを実行するネイティブモデルと, ホスト計算機の CPU と MIC 両者を使ってプログラムを実行するオフロードモデルといった, 2 つの実行モデルがある. 本研究では, マルチコア CPU と 1 台の Xeon Phi コプロセッサで構成されるシステム上において, 上記の 2 モデルに対する実装を考える.

一般に実対称行列の固有対計算は, 直交変換による 3 重対角化あるいは帯行列化を経て, 変換後の行列の固有対を計算する. ここで, 元の行列の固有値は変換後の行列の固有値そのものであり, 元の行列の固有ベクトルは 3 重対角化あるいは帯行列化の逆変換を施すことによって得られる. 3 重対角化や帯行列化, 逆変換の手法については高い並列化効率を達成する様々なアルゴリズムおよび実装法が提案されている.

これに対して本論文では, 変換後の行列に対する固有対計算法のうち, 実対称 3 重対角行列の全固有値あるいは一部の固有値のみ計算することのできる 2 分法による固有値計算に着目する. 尚, 実対称 3 重対角行列の固有ベクトル計算には, 逆反復法 [11], [16] や MRRR (Multiple Relatively Robust Representations) 法 [5], [7] があり, 分散メモリ型並列計算向け実装は数値計算ライブラリ ScaLAPACK (Scalable Linear Algebra PACKage) [2] にルーチンが提供されている. 特に, MRRR 法は全固有対計算にも適用可能であり, その場合には 2 分法を用いて全固有値を計算する必要がある. このような要求から, 本研究では, 2 分法による全固有値計算について評価を行う.

本論文の構成は以下の通りである. 2 章では, 本研究において対象とする Xeon Phi コプロセッサの特徴や実行モデルについて導入する. 3 章では, 性能評価において使用する実対称 3 重対角行列の固有値を計算するための 2 分法とそのスレッド並列実装について導入する. 4 章では, 2 分法の Xeon Phi コプロセッサ向け実装として, ネイティブモデル向け実装, オフロードモデル向け実装について導入する. 5 章では, まず, 4 章において示した 2 つの Xeon Phi 向けの並

[1] 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[2] 日本学術振興会特別研究員
Research Fellow of Japan Society for the Promotion
[a)] hishigami@amp.i.kyoto-u.ac.jp
[b)] kkimur@amp.i.kyoto-u.ac.jp
[c)] ynaka@i.kyoto-u.ac.jp

列 2 分法コードを数値実験により性能評価する. 更に, 共有メモリ型マルチコア計算機や GPU それぞれに適した 2 分法コードによる数値実験を行い, Xeon Phi を利用した並列 2 分法コードとの比較評価を行った結果についても示す. 6 章はまとめである.

## 2. Xeon Phi コプロセッサ

本章では, Xeon Phi コプロセッサの特徴や実行モデルについて導入する.

### 2.1 Xeon Phi コプロセッサの特徴

Xeon Phi コプロセッサは Intel 社により開発された MIC (Many Integrated Core) アーキテクチャに基づくアクセラレータで, GPU 同様, PCI Express に装着する形で使用できる. 計算コアの数は 60 個前後で，各計算コアが最大 4 スレッドの SMT (Simultaneous Multithreading) に対応する. GPU と同様に，Xeon Phi コプロセッサのそれぞれの計算コアは CPU に比べると低速であるが, 多数のスレッドを生かした並列計算により大幅な計算の高速化が期待できる.

また, CPU 向けのソースコードを Xeon Phi 向けに使用することや, OpenMP や MPI といった CPU 向けの並列化技法を適用することが可能である.

### 2.2 Xeon Phi コプロセッサの実行モデル

Xeon Phi には, ネイティブモデルとオフロードモデルといった 2 種類のプログラム実行モデルがある.

ネイティブモデルは, Xeon Phi のみでプログラムを実行するモデルで, `-mmic` オプションを追加してコンパイルすることで, CPU 向けに実装したコードをそのまま転用することができる. このモデルでは, 粒度の高い並列計算が大半を占めるようなアプリケーションでは高速な計算が期待できる.

一方, オフロードモデルでは, 指定したコード領域のみを Xeon Phi 上で実行し, その他の領域をホスト CPU で実行する. このモデルでは, オフロード実行のためにホストメモリとデバイスメモリの間でデータ転送を必要とする. このデータ転送は PCI Express を通して行われるので大きなオーバーヘッドにもなり得るが, 非並列計算や並列化粒度が小さい計算はホスト CPU, 並列化粒度の大きな計算には Xeon Phi を用いるといった, 両者の特性を生かしたプログラミングを実現することも可能である.

また, 以上で説明した 2 つの実行モデル両方において, 著名な数値計算ライブラリである LAPACK (Linear Algebra PACKage) [1] や BLAS (Basic Linear Algebra Subprograms) [15] を提供する Intel Math Kernel Library (MKL) [10] のルーチンを使用することが可能である. 以下, 既存の LAPACK ルーチンを用いる場合には Intel MKL を利用することを前提として議論する.

## 3. 固有値計算のための 2 分法の実装

本章では, 実対称 3 重対角行列の固有値計算のための 2 分法について述べる. まず, 数値計算ライブラリ LAPACK の 3 重対角行列向け 2 分法の倍精度浮動小数点演算ルーチン DSTEBZ [11] について導入を行い, 5 章の性能評価で用いるスレッド並列実装, ネイティブモデル向け実装, オフロードモデル向け実装それぞれについて述べる.

以下では, 計算対象とする $n$ 次実対称 3 重対角行列を $T$, $T$ の対角成分を $\{d_i\}_{i=1}^{n}$, 副対角成分を $\{e_i\}_{i=1}^{n-1}$ とする. また, $T$ の固有値は全て実数であり, $\{\lambda_i\}_{i=1}^{n}$ $(\lambda_1 < \cdots < \lambda_n)$ とする. 尚, 実装上では, $\boldsymbol{d} = [\, d_1, \cdots, d_n \,]$, $\boldsymbol{e} = [\, e_1, \cdots, e_{n-1} \,]$, $\boldsymbol{\lambda} = [\, \lambda_1, \cdots, \lambda_n \,]$ といった配列として保持される.

また, $T$ のある副対角成分が非常に小さい場合には, その成分を 0 と見なすことで $T$ を 2 つの小行列に分割することも可能である. しかし, 議論の簡単化のため, 以下ではそのような状況はないものとする.

### 3.1 2 分法ルーチン DSTEBZ

2 分法は, 2 分探索によって実対称行列の固有値を計算する手法である. 実対称 3 重対角行列向けの 2 分法は [11] において示されており, 数値計算ライブラリ LAPACK にも DSTEBZ としてコード実装がなされている. 全固有値を求める場合の DSTEBZ の計算は, 主に以下のように示される.

(1) 全固有値の存在範囲 $[\mu_l, \mu_r]$ を計算する
(2) $\mu_l$ および $\mu_r$ よりも小さな $T$ の固有値の個数を計算する
(3) 2 分探索によって, 求めたい固有値全てを計算する

DSTEBZ のルーチン内では, (2) や (3) の計算は下位ルーチン DLAEBZ において実装されている. 2 分法の主ルーチン (3) を擬似コードの形で簡単に表したものが, Algorithm 1 である. ここで, 5 行目から始まる **do** ループでは, 探索点 $\mu_k$ よりも小さな $T$ の固有値の個数を計算している. 9 行目では, その反復において計算したそれぞれの探索点よりも小さな固有値の個数を基に, 2 分探索における探索点の追加や探索の終了を行うタスク管理の処理が行われている. 尚, Algorithm 1 の最初の探索点 $\mu_1$ には, (1) で計算した $\mu_l$, $\mu_r$ を基に, $\mu_1 = (\mu_l + \mu_r)/2$ が用いられる. また, $p_{\min}$ は倍精度浮動小数点演算において逆数を取ったときにオーバーフローが起きない最小値 (safe minimum) である [*1].

尚, また, DSTEBZ ルーチンを使用することで, 区間 $[v_l, v_r]$ $(v_l, v_r \in \mathbb{R})$ に含まれる全固有値や, 最小固有値から数えて $i_l$ 番目から $i_r$ 番目の固有値 $(1 \le i_l \le i_r \le n)$ など, ユーザの指定に応じた固有値のみを計算することもできる. このような場合, (1) において計算する最初の固有値区間 $[\mu_l, \mu_r]$ の与え方が異なることになるが, (3) の計算が他の計算に比べ

[*1] 副対角成分が 0 と見なせるほど小さい場合には, その値を用いて修正した値が使用される. 詳細については, LAPACK の DSTEBZ ルーチンを見よ.

**Algorithm 1** 逐次計算における 2 分法主ルーチン

```
 1: k_b = 1, k_e = 1
 2: repeat
 3:     do k = k_b to k_e
 4:         c_k = 0
 5:         do j = 1 to n
 6:             r_j = d_j − e_{j−1}^2 / r_{j−1} − μ_k (e_0 = 0)
 7:             if r_{j+1} ≤ p_min then
 8:                 r_{j+1} = min(r_{j+1}, −p_min), c_k = c_k + 1
 9:     Manage tasks & Check convergence          ▷ Update k_b, k_e, μ
10: until k_b > k_e
11: return λ = μ
```

**Algorithm 2** スレッド並列化を施した 2 分法主ルーチン

```
 1: k_b = 1, k_e = 1
 2: repeat
 3:     !$omp parallel do private(r_t, c_t)
 4:     do k = k_b to k_e
 5:         c_t = 0
 6:         do j = 1 to n
 7:             r_j = d_j − e_{j−1}^2 / r_{j−1} − μ_k (e_0 = 0)
 8:             if r_t ≤ p_min then
 9:                 r_t = min(r_t, −p_min), c_t = c_t + 1
10:         c_k = c_t
11:     Manage tasks & Check convergence          ▷ Update k_b, k_e, μ
12: until k_b > k_e
13: return λ = μ
```

て多くの計算量を要するルーチンであることに変わりない.

### 3.2 2 分法のスレッド並列実装

Intel Math Kernel Library [10] は多くの LAPACK, BLAS のルーチンに対して, 共有メモリ型マルチコア CPU 向けにスレッド並列化されたものを提供しているが, 現在の Intel MKL が提供する DSTEBZ ルーチンは並列化が施されていない. そのため本節では, スレッド並列向けの 2 分法の実装について議論する.

#### 3.2.1 ScaLAPACK における並列 2 分法

分散メモリ型並列計算向け数値計算ライブラリ ScaLAPACK (Scalable LAPACK) [2] に提供されている 2 分法ルーチン PDSTEBZ と同じアイディアに基づく実装である.

PDSTEBZ ルーチンでは, 2 分法による固有値計算が固有値毎に独立であるという性質に基づき, 固有値番号についての並列化が施されている. これは [3] にも述べられている手法で, データ同期による通信が非常に少ない実装となっているため, 高い並列化効率が得られると期待できる. しかし, 各プロセッサの計算において重複した計算を有することになるため, LAPACK の DSTEBZ ルーチンに比べると全体の計算量は増加してしまう.

OpenMP によるスレッド並列化を用いることで, 共有メモリ環境向けに PDSTEBZ ルーチンのような実装が可能である. この場合, 求めたい固有値を番号順に計算スレッド数と同じ数のグループに分け, それぞれのグループに属する固有値を DSTEBZ でスレッド毎に計算すればよい. しかし, 共有メモリ環境においては, データ同期による計算遅延は必ずしも性能に大きく影響するとは限らない. 特に, 実対称 3 重対角行列の固有値計算のための 2 分法においては, 重複計算が生じることによる計算量の増加の方が計算遅延に影響すると考えられる. このため, 本手法が共有メモリ環境の特徴に適した並列化手法であるとは言い難い.

#### 3.2.2 本研究での並列 2 分法

本研究では, DSTEBZ において最も計算量を要する Algorithm 1 において, OpenMP によるスレッド並列化を施した実装を考える. そのコードは, Algorithm 1 による計算部分を Algorithm 2 のように書き換えたものとなる.

Algorithm 2 は, DLAEBZ の引数 NB が特殊な値のケースにおいて呼び出されるルーチンのコードを改変したものとなっており, 3 行目から始まる **do** ループについて並列化が施されている. また, 並列化の都合上, 変数 $c_t$ と $r_t$ を各スレッドのプライベート変数に設定した. 11 行目における 2 分探索のタスク処理は, 並列化する場合には排他的な処理が必要となる. しかし, **do** ループの計算に比べ, 非常に計算量の少ない処理であることから, 本研究では並列化は行わずにシリアル実行で処理するものとする. また, Algorithm 2 以外の計算には DSTEBZ と同じルーチンを使用し, シリアル実行で計算を行う.

この並列化手法では, 2 行目から始まる **Repeat-Until** の反復回数だけの同期が必要となるが, 排他的処理を要しないため, 計算遅延にはあまり影響しない. また, ScaLAPACK の PDSTEBZ ルーチンと異なり, LAPACK の DSTEBZ ルーチンと全く同じ計算量で計算を行うことができるため, 計算量の増加に起因する計算遅延は起こり得ない. その一方, 2 分探索の序盤では探索点の総数が計算コアに比べて少ないため, 一部の計算コアがアイドル状態になってしまう問題点がある. しかし, 2 分探索における探索点の総数 $(k_e - k_b + 1)$ は反復毎に最大 2 倍に増えるため, 探索点の数が計算コアに比べて少ないときの計算量は全体の計算量に比べて非常にわずかなものになる. したがって, 一部の計算コアがアイドル状態のままであっても, 全体の計算量が大きな問題に対しては, 計算時間の全体にあまり影響はないものと考えられる.

尚, 上記に示した 2 つの実装どちらについても, 計算コアの数に対して求めたい固有値の数が十分に多くない場合は, 計算コアがアイドル状態となる時間が増えてしまうという問題がある. このような状況では, 探索点を複数に増やすことで並列性を抽出する多分法 [13], [17] やその改良アルゴリズムである多固有値多分法 [12] が有効である.

**Algorithm 3** 2 分法主ルーチンのオフロードモデル向け実装

```
1: k_b = 1, k_e = 1
2: !dir$ offload_transfer target(mic:0)
   in(d) in(,e), nocopy(μ), nocopy(c)
3: repeat
4:     !dir$ omp offload target(mic:0)
       nocopy(d), nocopy(e), in(μ[k_b : k_e]), out(c[k_b : k_e])
5:     !$omp parallel do private(r_t, c_t)
6:     do k = k_b to k_e
7:         c_t = 0
8:         do j = 1 to n
9:             r_t = d_j − e_{j−1}^2/r_t − μ_k (e_0 = 0)
10:            if r_t ≤ p_min then
11:                r_t = min(r_t, −p_min), c_t = c_t + 1
12:        c_k = c_t
13:    Manage tasks & Check convergence          ▷ Update k_b, k_e, μ
14: until k_b > k_e
15: !dir$ offload_transfer target(mic:0)
    nocopy(d), nocopy(e), nocopy(μ), nocopy(c)
16: return λ = μ
```

表 1: 転送の必要な配列について
**Table 1** Summary of transfer data

| 配列 | 転送サイズ | 回数 | 転送 |
|---|---|---|---|
| $\boldsymbol{d}$ | $n$ | 1 | to MIC |
| $\boldsymbol{e}$ | $n-1$ | 1 | to MIC |
| $\boldsymbol{\mu}$ | $k_e - k_b + 1$ | #iter. | to MIC |
| $\boldsymbol{c}$ | $k_e - k_b + 1$ | #iter. | to Host |

## 4. 2 分法の Xeon Phi 向け実装

本章では, Xeon Phi コプロセッサ向けの 2 分法の実装法について述べる.

3.2.2 項で述べた 2 分法のスレッド並列実装のように, 計算量の意味で主要となる計算箇所について同期の少ない並列化した実装は多くの計算コアを持つ Xeon Phi に対しても非常に有効な実装である. このことから, ネイティブモデル向け実装としては, 3.2.2 項で述べた 2 分法のスレッド並列実装をそのままネイティブモデル向けにコンパイルすることで利用することにする. 一方, 2.2 節で述べたように, オフロードモデル向け実装ではホスト計算機とコプロセッサの間での通信を最適化する必要がある.

### 4.1 オフロードモデル向け実装

3.2.2 項で述べた 2 分法のスレッド並列実装に改変を加える形でオフロードモデル向け実装を構築する.

スレッド並列化の対象となっていた Algorithm 2 において OpenMP で並列化した部分をオフロード実行の計算領域とするような実装を施したものが, Algorithm 3 である. オフロード実行においてはホストメモリとデバイスメモリにおけるデータ転送が実行時間のオーバーヘッドとなり得るため, 転送の回数および転送量を最適化する必要がある. このため, 挿入したオフロード宣言子それぞれに適したコピー修飾子を設定した. ここで, 4 行目のオフロード宣言子に設定したコピー修飾子 `in`($\boldsymbol{\mu}[k_b : k_e]$), `out`($\boldsymbol{c}[k_b : k_e]$) は, それぞれの配列の第 $k_b$ 要素から第 $k_e$ 要素のみをデータ転送の対象とすることを示す. また, 2 行目および 15 行目では `offload_transfer` を用いることでデバイスメモリへの配列の確保および解放を 1 度ずつに制限している. 尚, Algorithm 3 中ではコピー修飾子に対して付与した修飾子オプションの記載を省略している.

表 1 は, オフロード実行において使用される配列それぞれについてホスト-デバイス間のデータ転送に関してまとめたものである. 表中の “転送サイズ” は転送 1 回当たりの要素数を示しており, $\boldsymbol{\mu}$ や $\boldsymbol{c}$ は 1 回当たり高々求めたい固有値の個数分までしか増大しない. また, #iter. は 3 行目から始まる **Repeat-Until** ループが収束するまでの反復回数を示しており, どの程度固有値区間を狭めるかによって異なる値となる.

## 5. 性能評価

本章では, 本研究で行った性能評価とその結果について報告する. 性能評価では表 2 に示した 3 つの計算機上で, それぞれに適した 2 分法の実装コードによって実対称 3 重対角行列の全固有値計算を行い, 実行時間を計測した.

テスト行列には, 固有値分布の異なる 2 つの $n$ 次実対称 3 重対角行列 $T_1$ および $T_2$ を用いた. $T_1$ は, Glued-Wilkinson 行列 [4], [6] である. この行列の固有値は, 14 のクラスタに分かれる行列で, 各クラスタに属する固有値はそれぞれ非常に密集した分布になる. テスト行列 $T_2$ は, LAPACK ルーチン DLARNV によって生成された (0, 1) の乱数で行列要素を定めた実対称 3 重対角行列である. この行列の固有値は Glued-Wilkinson 行列に比べると, 特に密集することもなく, 満遍なく分布する. 以上のような 2 つの行列の固有値分布の違いから, $T_2$ に対する 2 分法による固有値計算の計算量は $T_1$ に比べると多くなる.

計算機 I は, マルチコア CPU と Xeon Phi コプロセッサを搭載した計算機環境で, 4 章で示した 2 分法のネイティブモデル向け実装コード **MIC_Native** およびオフロードモデル向け実装コード **MIC_Offload** によって固有値計算を行った. ここで, **MIC_Native** には 3.2.2 節で導入した共有メモリ型マルチコア CPU 環境向けの並列 2 分法をネイティブモデル向けにコンパイルしたコードを, **MIC_Offload** には 4.1 節で導入したものを用いた. 尚, **MIC_Offload** の実行では, ホスト計算機の CPU で行う計算部分もあるが, 4.1 節で述べたように全てシリアル実行で計算を行った. 計算機 II は, 計算コアが計 28 コアの共有メモリ型マルチコア CPU 環境で, 3.2.2 項で示した並列 2 分法コード (以下, **CPU**) を使用した. ここで, 計算機 II に搭載された CPU もは, 計算コアあたり最大 2 スレッドの SMT を割り当てることができる. このため, 予備実験として, 28 スレッド実行と 56 スレッド実

表 2: 実験環境
**Table 2** Specifications of experimental environments

| | 計算機 I | 計算機 II (Cray XC30) | 計算機 III |
|---|---|---|---|
| Peak Performance for double precision | 1.208TFLOPS (only Xeon Phi) | 1.030TFLOPS | 1.300TFLOPS (only GPU) |
| Host CPU | Intel Xeon E5-1620 v2 (3.70GHz, 4 cores) | Intel Xeon E5-2695 v3 × 2 (2.30GHz, 14 cores × 2) | Intel Core i7 4771 (3.50GHz, 4 cores) |
| RAM | DDR3-1600 32GB | DDR4-2133 64GB | DDR3-1600 32GB |
| Accelerator | Intel Xeon Phi 7120A (1.238GHz, 61 cores) | | NVIDIA GeForce GTX TITAN |
| Device RAM | GDDR5 16GB | | GDDR5 6GB |
| Compiler | Intel Fortran 15.0.2 | Intel Fortran 15.0.3 | Intel Fortran 15.0.1 |
| Compile Options | -O3 -xHOST -ipo -no-prec-div -static† -openmp | | |
| Libraries | Intel MKL 11.2.2 | Intel MKL 11.2.3 | Intel MKL 11.2.1 CUDA 6.5 & CULA R18 |

†: GPU 向けには, -static オプションは不使用

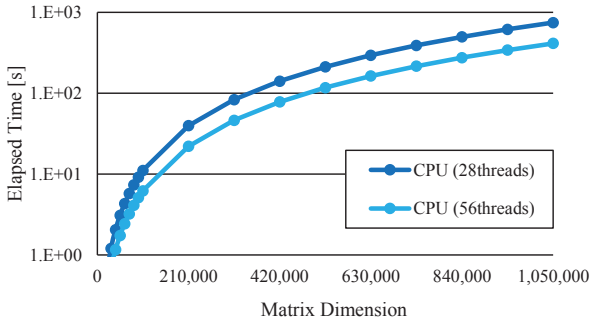
行における性能差を調べた. その結果をグラフにプロットしたものが図 1 で, $T_1$ および $T_2$ どちらの行列に対しても, 56 スレッド実行時の方が高速であった. 尚, 28 スレッド実行と 56 スレッド実行における性能差は, $n = 1,050,000$ の $T_1$ に対して 1.81 倍, $n = 1,000,000$ の $T_2$ に対して 1.81 倍であった. 計算機 III は NVIDIA 社の GPU, Geforce GTX TITAN を搭載した計算機である. 計算機 III 上では, CULA [8] に提供されている実対称 3 重対角行列向け 2 分法ルーチン culaDeviceDstebz(以下, **GPU**) により 2 分法の固有値計算を行った.

### 5.1 性能評価 I

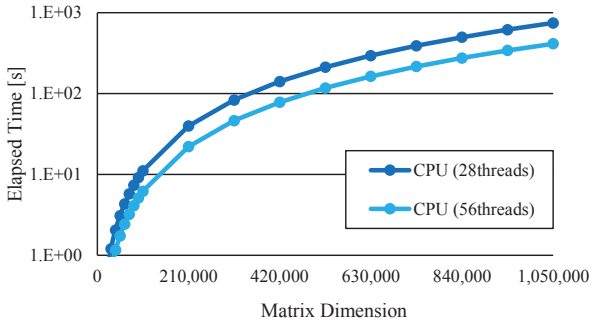
まず, マルチコア CPU と 1 台の Xeon Phi を搭載した計算機 I 上で, ネイティブモデル向け実装 **MIC_Native** およびオフロードモデル向け実装 **MIC_Offload** それぞれの 2 分法コードによって各テスト行列の全固有値計算を行った結果を示す. Xeon Phi 上の計算では各計算コアが発行する SMT の数を最大 4 つまで指定できるため, それぞれのコードの計算において SMT の使用数を変更した場合の計算時間を比較した.

図 2 は **MIC_Native** の実行時間を示しており, 61, 122, 244 スレッドそれぞれの場合で実行した結果を比較している. また, 図 3 は **MIC_Offload** の実行時間を示しており, 60, 120, 240 スレッドそれぞれの場合で実行した結果を比較している. ここで, **MIC_Native** と **MIC_Offload** の実行において異なるスレッド数を利用しているのは, オフロードモデル実行の際には総計算コア数に対して 1 コアを減らして実行することが推奨されているからである. 尚, 図 2a および図 3a がテスト行列 $T_1$, 図 2b および図 3b が $T_2$ の場合に対応している.

ある程度次数の大きなテスト行列に対しては, どちらの行列の場合でも, 1 コアあたりの SMT の実行スレッドが



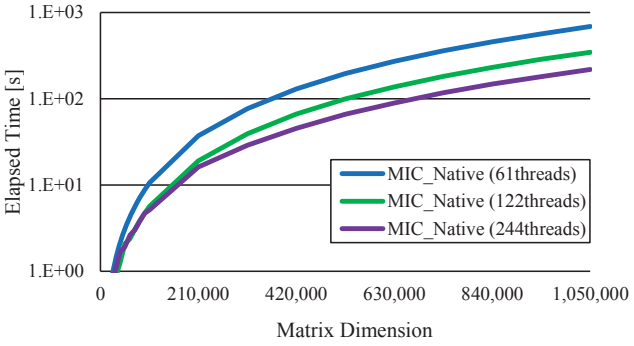

(a) Cases of $T_1$



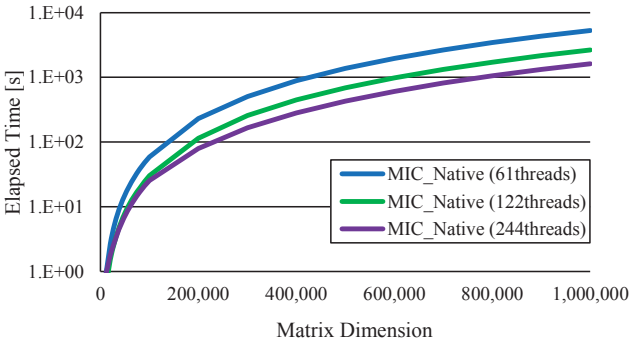

(b) Cases of $T_2$

図 1: 計算機 II における各テスト行列の全固有値計算に対する並列 2 分法コードの異なるスレッド数による実行時間
**Fig. 1** Dimensions and elapsed times for computing all the eigenvalues of each target matrix using parallel bisection code with the different number of threads on Computer II.

多くなるほど高い実行性能が得られることが分かる. 実際, **MIC_Native** は, 行列サイズ $n = 1,050,000$ の $T_1$ に対して, 61 スレッド実行時に比べ, 122 スレッド実行時に 1.99 倍, 244 スレッド実行時には 3.15 倍の高速化が認められる. $n = 1,000,000$ の $T_2$ に対して, 61 スレッド実行時に比べて, 122 スレッド実行時には 2.00 倍, 244 スレッド実行時には 3.26 倍の高速化が認められる. 一方, **MIC_Offload** は, 行



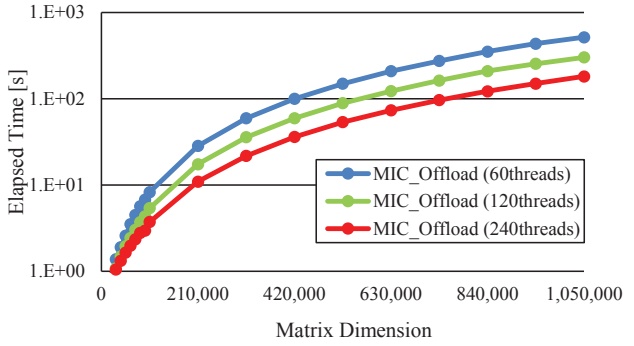

(a) Cases of $T_1$



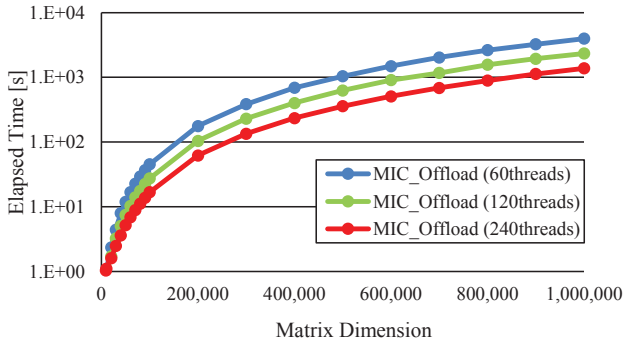

(b) Cases of $T_2$

図 2: 計算機 I における各テスト行列の全固有値計算に対するネイティブモデル向け 2 分法コードの実行時間

**Fig. 2** Dimensions and elapsed times for computing all the eigenvalues for each target matrices using bisection code of the Native programming model for Xeon Phi co-processors with the different number of threads on Computer I.




(a) Cases of $T_1$




(b) Cases of $T_2$

図 3: 計算機 I における各テスト行列の全固有値計算に対するオフロードモデル向け 2 分法コードの実行時間

**Fig. 3** Dimensions and elapsed times for computing all the eigenvalues for each target matrices using bisection code of the Offload programming model for Xeon Phi co-processors with the different number of threads on Computer I.

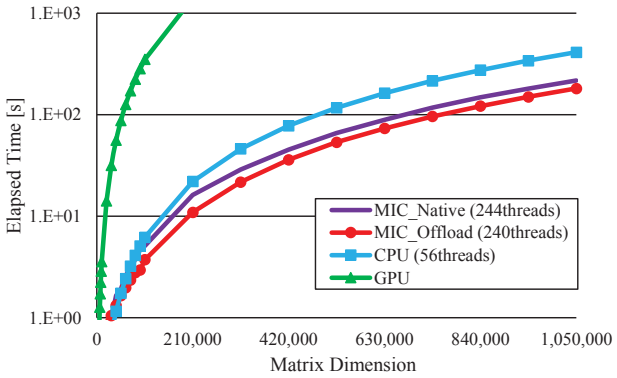
列サイズ $n = 1,050,000$ の $T_1$ に対して， 60 スレッド実行時に比べて, 120 スレッド実行時には 1.71 倍, 240 スレッド実行時には 2.84 倍の高速化が認められる. $n = 1,000,000$ の $T_2$ に対して， 60 スレッド実行時に比べて, 120 スレッド実行時には 1.69 倍, 240 スレッド実行時には 2.88 倍の高速化が認められる. 以上のように, 計算コアの数よりスレッド数を使用した場合でも性能向上が認められるのは, 命令パイプラインがうまく最適化されているからであると考えられる. この原因としては, 本研究で評価を行っている実対称 3 重対角行列向けの 2 分法において, 条件分岐が多く含まれることが挙げられる. 条件分岐は他の命令に比べて実行サイクル数が多いため, 1 コアに対して 1 スレッドを実行する場合には命令パイプラインのストールが生じやすくなる. しかし, SMT のように 1 コアで複数スレッドの計算を行う場合, 各計算コアの命令パイプラインが動的にスケジュールされる. ある計算コアのスレッド 1 つが条件分岐の命令を実行している際に, その間に他のスレッドが四則演算を行うといった命令パイプラインの最適なスケジュールがなされれば, そのようなストールは起きにくくなり, 性能向上につながると考えられる.

また, 行列サイズが小さい場合には, どちらの実装コードについても 1 コアあたりの SMT の実行スレッド数が少ない方が高速になる傾向がある. 行列サイズが小さい時は, 行列サイズが大きい時に比べて計算量が少ない. その結果, 先述したようなパイプラインの最適化による性能向上があまりみられず, データ転送を含めたスレッド間の同期がボトルネックとなる性能低下が生じたと考えられる.

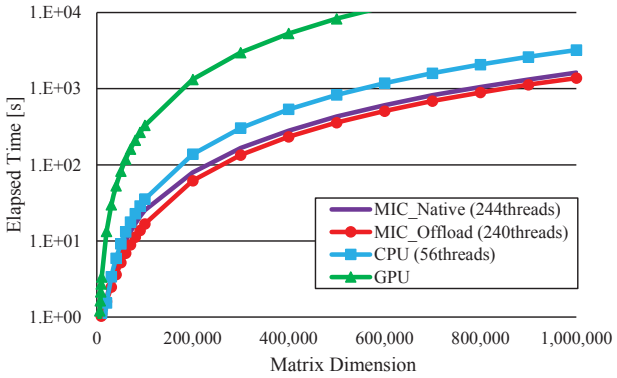
### 5.2 性能評価 II

図 4 は, 計算機 II および III 上において 2 分法コードによる各テスト行列の全固有値計算を行った際の実行時間を計測し, 性能評価 I で得られた結果との比較をしたものである. ここで, 図 4a がテスト行列 $T_1$, 図 4b が $T_2$ の場合に対応している. 尚, 計算機 I および計算機 II 上での計算結果については, 大次元のテスト行列で高速な性能を示していたスレッド数での実行結果, すなわち, **MIC_Native** は 244 スレッド実行時, **MIC_Offload** は 240 スレッド実行時, **CPU** は 56 スレッド実行時のものをプロットした. また, 図 5 は, 図 4 にプロットした結果から, 各テスト行列の次元が比較的小さな場合に対する結果のみを表したものである.

図 4 からは, どちらのテスト行列においても, 大次元の場合には **MIC_Offload** が最も高速で, **MIC_Native**, **CPU**, **GPU**



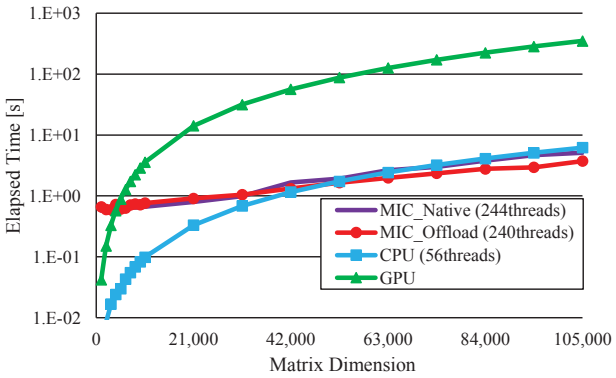

(a) Cases of $T_1$



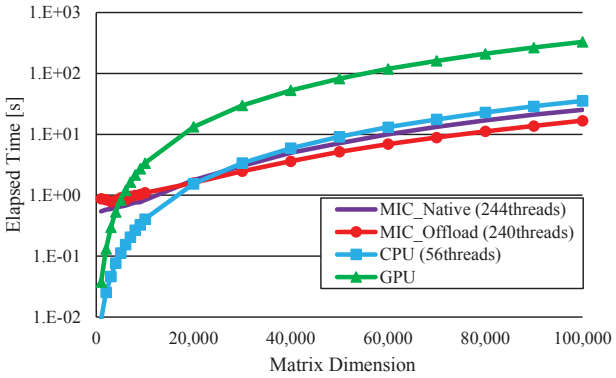

(b) Cases of $T_2$

図 4: 各テスト行列の全固有値計算に各計算機環境における 2 分法コードの実行時間

**Fig. 4** Dimensions and elapsed times for computing all the eigenvalues of each target matrix using different bisection code. **MIC_Native** and **MIC_Offload** are run on Computer I. **CPU** is run on Computer II. **GPU** is run on Computer III.




(a) Cases of $T_1$




(b) Cases of $T_2$

図 5: 次数の小さなテスト行列の全固有値計算におけるにおける 2 分法コードの実行時間

**Fig. 5** Dimensions and elapsed times for computing all the eigenvalues of each lower dimensional target matrix using different bisection code. **MIC_Native** and **MIC_Offload** are run on Computer I. **CPU** is run on Computer II. **GPU** is run on Computer III.

の順に性能が良いということが分かる. 一方で, 図 5 から, Xeon Phi 向けのコード **MIC_Native** および **MIC_Offload** は, $T_1$ では $n = 42,000$, $T_2$ では $n = 20,000$ までの行列に対して **CPU** よりも性能が悪く, 数千次の行列に対しては **GPU** よりも性能が悪い, ということが分かる. このように, 小さな行列に対する性能が悪いのは, 計算量が十分に多くないために SMT の性能をうまく引き出しきれていないかったと考えられる.

また, **MIC_Offload** は, **MIC_Native** に対して, $n = 1,050,000$ の $T_1$ の場合には 1.20 倍, $n = 1,000,000$ の $T_2$ には 1.18 倍高速であった. この結果の原因の一つには, 全体の計算量に比べると非常に小さいものの, 両者の実装コードにおいてシリアル実行を要せざるを得ない部分が残っていることが挙げられる. このシリアル実行の部分の計算において, **MIC_Offload** では周波数の点で有利であるホスト計算機の CPU を使っているため, **MIC_Native** よりも良い性能が得られたのではないかと考えられる.

また, **MIC_Offload** は, **CPU** に対して, $n = 1,050,000$ の $T_1$ の場合には 2.28 倍, $n = 1,000,000$ の $T_2$ には 2.34 倍高速であった. ここで, 倍精度浮動小数点数演算において, 計算機 I に搭載された Xeon Phi のピーク性能の理論値 (1,208GFLOPS) は計算機 II のピーク性能の理論値 (1,030.4GFLOPS) の 1.17 倍である. しかし, これらのピーク性能は SIMD 演算を最大限利用することを前提とした議論に基づいているが, 本研究で使用している実対称 3 重対角行列向けの 2 分法では条件分岐が多いために SIMD 演算をうまく活用できているとは言えず, ピーク性能に基づく議論をあてはめることは難しい. 一方, 先述したように, **MIC_Offload**, **CPU** どちらについても SMT によるパイプラインの最適化が性能向上大きく係わっているため, それぞれで使用できる SMT の数に違いがあることが, 以上のような性能差を示す一つの要因であると考えられる. 実際, 計算機 I の Xeon Phi と計算機 II の CPU について, それぞれのクロック周波数, 計算コア数, 使用 SMT 数の積の比を取ってみると,

$$\frac{1.238 \times 60 \times 4}{2.30 \times 28 \times 2} = 2.306\cdots$$

となり, 実験における性能比に近いものとなっている.

計算機 III に搭載した GPU はピーク性能の点で他の計算

機環境よりも優れているが, テスト行列が数千次の場合を除いて, **GPU** は他のコードよりも低速であるという結果であった. 使用した culaDeviceDstebz の実装は公開されていないため詳しい解析をすることはできないが, 実対称 3 重対角行列に対する 2 分法自体が GPU においてあまり性能を発揮できない条件分岐の多いアルゴリズムであることに起因すると考えられる.

## 6. まとめと今後の課題

本論文では, Intel 社が開発した MIC アーキテクチャ Xeon Phi 上において, 2 分法に基づく実対称 3 重対角行列の固有値計算の性能を評価した. また, GPU やマルチコア CPU 上での数値実験との比較により, 特に大次元のテスト行列に対して, これらのアーキテクチャよりも Xeon Phi を利用した計算の方がより良い性能を示すことを確認した.

他の課題として, 複数の Xeon Phi コプロセッサ向け, あるいは, Xeon Phi コプロセッサ搭載ノードからなるクラスタ向けに 2 分法を実装し, 性能評価することが挙げられる. 更に, 逆反復法や MRRR 法の Xeon Phi コプロセッサ上での実装および性能評価も重要な課題である.

また, 本研究では実対称 3 重対角行列の固有値計算を対象としたが, 2 分法に基づいた実対称帯行列の固有値計算も可能で, [9], [14] において効率の良い実装法が示されている. このような拡張と Xeon Phi 上での性能評価もまた, 今後の課題である.

**謝辞** 本研究は科学研究費補助金特別研究員奨励費 (課題番号：25・2820), 基盤研究 (B) (課題番号: 24360038) の補助を受けている. また，本研究の結果の一部は，京都大学学術情報メディアセンターのスーパーコンピュータ Cray XC30 を利用して得られたものである.

### 参考文献

[1] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J. W., Dongarra, J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A. and Sorensen, D.: *LAPACK Users' Guide (Third ed.)*, SIAM, Philadelphia, PA, USA (1999).

[2] Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R.: *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, USA (1997).

[3] Demmel, J. W., Dhillon, I. S. and Ren, H.: On the correctness of parallel bisection in floating point, Technical Report UCB/CSD-94-805, EECS Department, University of California, Berkeley (1994).

[4] Demmel, J. W., Marques, O. A., Parlett, B. N. and Vömel, C.: Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers, *SIAM J. Sci. Comput.*, Vol. 30, No. 3, pp. 1508–1526 (2008).

[5] Dhillon, I. S.: A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem, PhD Thesis, EECS Department, University of California, Berkeley (1997).

[6] Dhillon, I. S., Parlett, B. N. and Vömel, C.: Glued matrices and the MRRR algorithm, *SIAM J. Sci. Comput.*, Vol. 27, No. 2, pp. 496–510 (2005).

[7] Dhillon, I. S., Parlett, B. N. and Vömel, C.: The design and implementation of the MRRR algorithm, *ACM Trans. Math. Softw.*, Vol. 32, No. 4, pp. 533–560 (2006).

[8] EM Photonics: CULA, (online), available from ⟨http://www.culatools.com/⟩ (accssesed 2015-01-16).

[9] 長谷川秀彦：ベクトル計算機と汎用計算機のための対称帯行列固有値解法，情報処理学会論文誌，Vol. 30, No. 3, pp. 261–268 (1989).

[10] Intel: Intel Math Kernel Library, (online), available from ⟨http://software.intel.com/en-us/intel-mkl⟩ (accssesed 2015-01-16).

[11] Kahan, W.: Accurate eigenvalues of a symmetric tridiagonal matrix, *Technical Report, Computer Science Dept. Stanford University*, No. CS41 (1966).

[12] Katagiri, T., Vömel, C. and Demmel, J. W.: Automatic performance tuning for the multi-section with multiple eigenvalues method for symmetric tridiagonal eigenproblems, *Applied Parallel Computing. State of the Art in Scientific Computing*, Lecture Notes in Computer Science, Vol. 4699, Springer Berlin Heidelberg, pp. 938–948 (2007).

[13] Lo, S., Philippe, B. and Sameh, A.: A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem, *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 2, pp. s155–s165 (1987).

[14] 村田健郎：標準形対称行列固有値解法の見直し II 帯行列に対する直接のスツルム・逆反復法，図書館情報大学研究報告， Vol. 5, No. 1, pp. 25–45 (1986).

[15] Netlib: BLAS, (online), available from ⟨http://www.netlib.org/blas/⟩ (accssesed 2015-01-16).

[16] Parlett, B. N.: *The Symmetric Eigenvalue Problem*, SIAM, Philadelphia, PA, USA (1998).

[17] Simon, H.: Bisection is not optimal on vector processors, *SIAM J. Sci. Stat. Comput.*, Vol. 10, No. 1, pp. 205–209 (1989).