Vol.2018-HPC-165 No.33
2018/8/1

三重対角化におけるメニーコア環境に適した同期手法

工藤 周平[1,a)]　今村 俊幸[1]

概要：対称行列の三重対角化は固有値計算の前処理として利用される行列計算である．しかし，計算の大部分が逐次的な対称行列ベクトル積となっているため，B/F 値が高いだけでなく，同期のオーバーヘッドが大きいため高性能化が難しい．
発表者はストリーム数を減らすデータ配置と，データ再利用性を高めるためのループ融合手法を用いることで，京コンピュータの単体ノード向けにチューニングした三重対角化ライブラリを開発し公開しているが，これは，京コンピュータに特化しているため，現在の多数のコアを持ち高いスレッド並列性が要求される計算機においては十分な性能を出せないものとなっていた．
そこで，現在のマルチコア・メニーコア CPU における同期の問題へ対応するため，対称行列ベクトル積に出現するコア間総和処理に適した同期手法を開発した．この結果， Intel のメニーコア CPU Xeon Phi(Knights Landing) 上で，行列サイズ 5,000 程度の大きさにおいても Intel MKL の 2 倍以上となる高い性能を実現している．
本発表ではこの同期手法について解説し，データ配置やブロックサイズなどを変化させたときの詳細な性能測定結果を示す．また，内部的に特別なデータ配置を用いる行列計算ライブラリの応用について議論する．

キーワード：三重対角化, メニーコア CPU, 同期手法, ツリー型総和処理

Shuhei Kudo[1,a)]　Toshiyuki Imamura[1]

1. はじめに

三重対角化は対称行列の固有値計算に用いられる行列分解であり，実対称行列 $A \in \mathbb{R}^{n\times n}$ を直交行列 $X \in \mathbb{R}^{n\times n}$ と三重対角行列 $T \in \mathbb{R}^{n\times n}$ との積 $A = XTX^\top$ へ分解する．通常はこの後，三重対角行列の固有値分解 $T = YDY^\top$ を QR 法や分割統治法などの手法を用いて計算することで，元の行列 A の固有値分解 $A = (XY)D(XY)^\top$ を得る．このとき三重対角化が演算量の大きな部分を占めるため，固有値分解の高速化には三重対角化の高性能化が不可欠である．

三重対角化の計算手法としては，入力行列 A にハウスホルダー変換 H_i による両側からの直交変換を繰り返し適用する，ハウスホルダー法が精度と演算量の面で優れているため，現在用いられている．この両側変換は式変形をすることで，対称行列 2 階更新 (SYR2) と対称行列ベクトル積 (SYMV) へ書き直すことができる．前者については Dongarra-Sorrensen の手法（DS 法） [5] によって，より高速な行列積の一計算パターンである DSYR2K に置き換えられるが，後者についてはそのまま残るため，SYMV が三重対角化の性能ネックになる．筆者は以前に，別のハウスホルダー法高性能化手法である Wilkinson の手法 [2] が DS 法と組み合わせ可能であることを発見し，両手法を組み合わせた手法（DW 法）を示した [6]．DW 法は一部の SYMV を DSYR2K と同時計算可能にし，SYMV 単体計算の回数を減らすことができるが，依然として，SYMV が性能ネックとなることは変わらなかった．

SYMV の性能面での問題は，第一にデータ再利用性が低いことが挙げられるが，それとは別に総和処理の問題がある．SYMV を並列化した場合，計算の構造上，コアごとに求めた部分和の並べられた配列の総和計算を行う必要がある．この配列の長さは行列サイズ n と一致し，コア数 p と同じ本数となる．そこで，コア数が多くなるほど総和処理の負荷が増大し，SYMV の演算量 $O(n^2)$ に対して総和処理の演算量 $O(np)$ が無視できないほど大きくなってしまう．

我々はこれまで DW 法の適用による演算自体の効率向上を行ってきたが，近年，高性能計算において重要な位置を

[1] 理化学研究所 計算科学研究センター
R-CCS, Kobe-shi, Hyogo, 650–0047, Japan
[a)] shuhei.kudo@riken.jp

占めるようになったメニーコア CPU に対応すべく，三重対角化における総和処理の速度改善に取り組んだ．本研究では Intel 社のメニーコア CPU である Xeon Phi Knights Landing (KNL) を対象に，ツリー型の総和処理を実装することで，高性能化を試みた．ツリー型の総和処理は分散メモリ並列環境における総和処理のアルゴリズムの 1 つであり，コア間通信を減らすことに役立つと考えられるが，実際に高速化されるのかについては未知数である．本稿では，単純なバリア同期を用いた総和処理とツリー型の総和処理の性能比較を KNL 上で行う．また，ツリー型の総和処理を三重対角化に組み込んだときの性能を示す．また，他のマルチコア・メニーコア CPU においてツリー型総和処理の性能測定結果を示し，ツリー型総和処理の適用範囲について議論する．

2. 三重対角化の手順と並列化・高性能化手法

2.1 三重対角化

三重対角化は元の行列 A に対してハウスホルダー変換 H_i による直交変換を $n-2$ 回適用することで三重対角行列 T に変換する．最初のステップでは $A_n = A$ とおき，第 i 主小行列を除けば三重対角の形となっている A_i に対して

$$A_{i-1} = H_i^\top A_i H_i \tag{1}$$

を計算していき，最終的に $T = A_2$ を得る．ただし実際の計算では，データ量を削減するため新しい行列 A_{i-1} は元の行列のメモリ位置へ上書きする形で保存する．

ハウスホルダー変換 H は一般にゼロベクトルでないベクトル u[*1] によって $H = H^\top = I - \frac{2uu^\top}{u^\top u} = I + \tau uu^\top$ と表される変換であり，同じ長さを持つ 2 つのベクトルが与えられたとき，片方をもう一方に移す変換を与える．そこで行列を特定の非零構造を持つように変形する場合などに変形するときに用いられる．三重対角化では $H_i = I - \frac{2u_iu_i^\top}{u_i^\top u_i} = I + \tau_i u_i u_i^\top$ を，A_i の $i+1$ 行目から n 行目までと，$i+1$ 列目から n 列目までを普遍に保ちつつ，A_i の第 i 主小行列の i 行目と i 列目を所要の形に変形するようを定める．そこで H_i を計算は A_i の第 i 列目に依存している．

三重対角化の計算のうち主要な部分は式 (1) の行列積であるが，H_i の構造を用いると演算量を低減できるため，通常は次の式変形を行う．まず次の式を計算しておき：

$$y_i = A_i u_i, \tag{2}$$

$$v_i = \tau_i y_i + \frac{\tau_i^2}{2}\left(y_i^\top u_i\right) u_i, \tag{3}$$

この結果を用いて式変形を行う：

$$\begin{aligned} A_{i-1} &= H_i A_i H_i \\ &= (I + \tau_i u_i u_i^\top) A_i (I + \tau_i u_i u_i^\top) \\ &= A_i + u_i y_i^\top + y_i u_u^\top + \tau_i u_i y_i^\top u_i u_i^\top \\ &= A_i + u_i v_i^\top + v_i u_i^\top. \end{aligned} \tag{4}$$

ただしここでは A の対称性を用いて $A_i u_i = \left(u_i^\top A_i\right)^\top$ を使った．この結果，式 (2) に相当する対称行列ベクトル積 (SYMV) と式 (4) に相当する対称階数 2 更新 (SYR2) が主要な計算となる．ただし三重対角化では右下帯部分が不変であるため，SYMV と SYR2 ともに第 $i-1$ 小行列にのみ計算を行えば良い．

このように三重対角化は順次小さくなる行列に対し SYMV と SYR2 を交互に繰り返す計算となる．

2.2 Dongarra-Wilkinson 法

Wilkinson [2] は，A_i に対する SYR2 のうち u_{i-1} の計算に必要な第 $i-1$ 列目を先行計算する手法を開発した．u_{i-1} が事前にわかっていれば，A_i に対する SYR2 と A_{i-1} に対する SYMV の計算を融合させることができる．具体的には，SYR2 が要素ごとに独立に計算できるため，A_{i-1} の要素が 1 つ計算できたときに，そのデータをメモリに書き戻す前に SYMV の計算に用いるようループを書き換える．これによって要素ごとに 3 回のデータ移動が 2 回へ削減できる．計算手順の詳細については，Wilkinson の著書による解説は短いが，寒川 [3] や村上 [4] に詳しい解説や性能測定結果がある．

Dongarra と Sorensen [5] は SYR2 による行列更新を遅延し，複数回まとめて行う手法を開発した．具体的には，A_i の全体を更新はせずに一部分だけを更新することで，複数回 (ここでは K 回) 分の更新のためのベクトルを溜めておき，最後に

$$A_{i-K} = A_i + \sum_{k=i}^{i-K+1} u_k v_k^\top + v_k u_k^\top \tag{5}$$

を計算する．この形の計算パターンは Level-3 BLAS のサブルーチンの 1 つ SYR2K として定義されており，データ再利用性が高く，SYR2 と比べて高速に計算できる．ただしこの式変形のためには $O(nK)$ の演算量が余分に必要となるため，ブロック幅 K は計算性能向上効果と演算量増大のトレードオフ関係の中で良い値を選ぶ必要がある．また，この手法では K 回の SYR2 をより計算効率の高い SYR2K に置き換えるが，K 回の SYMV はそのまま残ることになる．

Dongarra-Wilkinson 法 (DW 法) はこれらの 2 つの手法を組み合わせたものである．すなわち，SYR2K によって A_i から A_{i-K} を計算するときに，A_{i-K} の第 $i-K$ 列目のみを先行計算しておくことで，依存性を解決し，SYR2K に続く 1 回の SYMV を SYR2K に融合させる．これによっ

[*1] u がゼロベクトルのときは $H = I$ と定義することによってより一般化した形となる．

て，演算性能の低い K 回の SYMV のうち 1 回だけ SYR2K の中に隠蔽される．当然，$K-1$ 回の SYMV はそのまま残るため効果は限定的だが，ルーフラインモデルを用いた性能予測においては，Dongarra の手法と比べて，同じ K を用いた場合は DW 法がより高い性能となり，また，ピーク性能へ達するために必要な K の大きさは DW 法がより小さくなることが言える．詳細については文献 [6] を参照されたい．

2.3 並列化手法

プログラムの並列化においてはワークバランスを均等にすることや同期の回数を減らすことが重要である．三重対角化のアルゴリズムでは，相対的に演算量の大きい SYMV や SYR2K を並列化する．SYR2K は要素ごとに独立な計算であるため，容易に並列化できる．このときデータの分割方法が重要だが，三重対角化では行列サイズを小さくしながら反復を繰り返すことと，行列が対称性を持つことから，我々は 1 次元ブロックサイクリック分割を採用している．以下は，OpenMP を用いて SYR2 を並列化したときの C 言語のソースコードである．

```
void syr2(int n, double* a, int lda,
   double* u, double*v){
   #pragma omp for nowait schedule(static, N)
   for(int i=0; i<n; ++i){
      for(int j=0; j<=i; ++j){
         a[lda*i + j] += u[j] * v[i] + v[j] * u[i];
}}}
```

ここで N はブロックサイクリック分割のブロックサイズである．また行列 a はリーディングディメンジョンを lda とする BLAS 風の二次元配列としている．このプログラムにおいて u や v を行列として，最内側処理の内積を延長することで SYR2K のプログラムも作ることができる．

SYMV や，DW 法に出現する SYR2K と SYMV の融合計算 (SYR2KMV) についても，同様の方針で並列化を行うが，行列ベクトル積の結果 y_i が存在するため，工夫が必要である．我々の実装では，y_i について，コアごとに別々の領域を確保しておき部分和を計算し，処理の完了後にその総和を計算する，という二段階の手順を採用している．以下は，この手法を用いて SYMV を並列化したときのソースコードである．このコードは総和を取る前の部分和を計算するところまでを示している．

```
void symv(int n, double* a, int lda,
   double* x, double* myy){
   for(int i=0; i<n; ++i) myy[i] = 0.;
   #pragma omp for nowait schedule(static, N)
   for(int i=0; i<n; ++i){
      for(int j=0; j<=i; ++j){
         myy[j] += a[lda*i + j] * x[i];
         myy[i] += a[lda*i + j] * x[j];
}}}
```

このコードにおいて myy は部分和を格納するためのコアごとに独立に確保された領域である．内側ループにおいて，a[lda*i + j] を読み込んだ際，このデータは 2 つの myy の位置に対応する計算に用いられる．これによってメモリバンド幅を削減できるが，代わりに myy へのアクセスが複雑化する．これが y_i に対してデータ並列にすることができない理由である．

総和を取るためには次のようなコードが必要になる．

```
void reduce(int nc, int n, double**ylist){
   #pragma omp barrier
   for(int j=1; j<nc; ++j){
      #pragma omp for nowait schedule(static)
      for(int i=0; i<n; ++i){
         ylist[0][0] += ylist[j][i];
   }}
   #pragma omp barrier
}
```

このコードは各コアが計算した部分和の配列へのポインタが並べられた ylist を受け取り，コア数 nc 個分の総和を計算する．結果は ylist のはじめの要素が指す配列に格納される．このコードでは，symv の計算結果を待つため，そして総和計算後にその結果を後の計算で用いるための 2 回の同期が必要である．

2.4 他の SYMV 並列化手法

SYMV の並列化手法については他にいくつか考えられる．第一に，共有の y_i を用い，適切なタイミングで排他処理を行う手法，第二に，行列の対称性を用いない手法である．

第一の排他処理を行う手法では，複数のコアが同じ y_i の位置へ書き込みを行うことを防ぐため，アトミック演算，またはミューテックスを用いた排他制御を行うものである．この手法では，より頻繁に共有変数へアクセスすることになるためコア間データ移動量は増大するが，SYMV の演算と同時にデータ移動を行えるという点で，通信隠蔽の効果が得られる可能性がある．浮動小数点数に対するアトミック演算がサポートされた CPU は珍しいが，NVIDIA の GPU にはサポートされたものがあり，そのような場合には有力な選択肢となる．KNL などのようにサポートされていない場合は，compare-and-swap 命令を用いてソフトウェア的に行う必要があり，オーバーヘッドが大きい．ミューテックスもまたオーバーヘッドの大きい操作である．さらに，これらの排他制御を行うアルゴリズムでは足し算の順序が一定にならず，実行ごとに結果が変化してし

まうという問題もある．

第二の手法としては，行列の対称性を用いない方法が考えられる．対称性を用いなければ y_i へのアクセスが簡単化し，データ並列にすることは容易である．単に，コアごとに異なる行列 A の行を担当するようにデータ分割すれば良い．ただし，対称行列を通常の密行列として格納した場合データ量が 2 倍に増え，CPU のデータキャッシュを無駄にすることになり，また，三角部分のみを格納したままでデータアクセスを工夫する場合は，結果的に三角行列の転置を行うことになり，データ移動量が増大する．

総和処理を後で行う方式のアルゴリズムには GPU 向けライブラリの MAGMA で用いられており [8]，対称性を用いない手法との比較も行っている．他に CPU 向けとしては OpenBLAS [9] がある．今村ら [10] は排他処理を行う手法について，GPU 向け高性能実装を議論している．本研究では KNL を対象とするため，総和処理を後で行う方式を選択し，その高性能化手法を検討した．

2.5 高速化手法

実際に用いたコードは上記のように単純なものではなく，いくつかのプログラム高速化テクニックを適用したものとなっている．我々の実装では，完全なアセンブラチューニングではなく，C 言語と SIMD Intrinsic 命令を用いたものになっており，ある程度の可用性を残す形となっている．

ここではチューニング済みのコードから端数処理など細かな部分を除いた概要を示す．

```
void syr2_opt(int n, double* a, int lda,
   double* u, double*v){
   #pragma omp for nowait schedule(static, 1)
   for(int i=0; i<n; i+=N){
      double* ca = a + lda * i;
      for(int j=0; j+V<=i; j+=V){
         ca[j:j+V-1] += u[j:j+V-1] * v[i] +
            v[j:j+V-1] * u[i];
         (ca+lda)[j:j+V-1] += u[j:j+V-1] * v[i+1] +
            v[j:j+V-1] * u[i+1];
         // ...
         (ca+lda*(N-1))[j:j+V-1] +=
            u[j:j+V-1] * v[i+N-1] +
               v[j:j+V-1] * u[i+N-1];
      }
      // 上三角形状に対する端数処理
}}
```

この擬似コードでは，SIMD Intrinsic 命令に相当する部分を Fortran 風の配列演算 (`j:j+V-1` など) で表しているが，実際のプログラムでは`_mm512_fmadd_pd()` などの Intrinsic 命令で実装する．ここで行われている最適化は，列ブロックサイズに関する幅 `N` のループを最内側に移動し，ループアンロールすること，行番号に関する `j` のループを SIMD 幅 `V` ごとに SIMD 並列化すること，積と和の代わりに積和命令を用いること，の 3 点である．

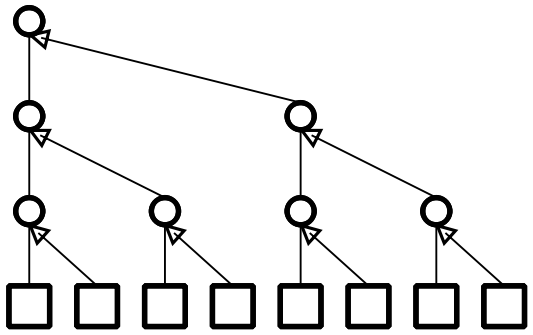

図 1 ツリー型総和処理の模式図

筆者は上記プログラムにおける行列 `a` へのアクセスがストライドアクセスになることに着目し，事前に行列データを並び替えることで最内側ループをストリームアクセスにするデータパッキング手法を示した [6]．またそこで使われた実装はオープンソースライブラリとして公開している [7]．これは京コンピュータの CPU である Sparc 64VIIIfx を目的として開発したものであり，残念ながらこのデータパッキング手法は今回の目的である KNL では有効ではなかったため，以下の実験では用いていない．

一方，KNL は Sparc 64VIIIfx と比べて 8 倍以上のコア数を持つため，SYMV における総和処理が新たな問題となっていた．そこで我々はこの総和処理の高速化に取り組んだ．

3. 総和処理のアルゴリズム

3.1 バリア同期を用いたアルゴリズム

前節で示したプログラム `reduce` はバリア同期を用いたアルゴリズムの実装である．このアルゴリズムでは，データ並列にするため，それぞれのコアが配列の異なる添字の位置を担当するように分割している．そこで `symv` と `reduce` の間でデータ分割が変化し，通信が発生する．

この通信パターンは，`reduce` の外側ループのたびに `MPI_Scatter` のような，あるコアの持つ配列を分割して全体に送信するものとなっている．そこで 1 つのコアにアクセスが集中し，データ移動性能が低下することが考えられる．また，並列化軸が配列の長さ方向であるため，内側ループの回転数が小さくなる問題がある．今回ターゲットとしている KNL では，コア間通信はメモリーアクセスと同程度のレイテンシーを持つため，回転数の短さは実行効率の低下に繋がるものと考えられる．

3.2 ツリー型総和処理

ツリー型総和処理は基本的な集団通信アルゴリズムの 1

つとして，例えばMPIのような通信ライブラリに実装されているものである．アルゴリズムの解説としては例えばGramaらによる教科書 [1], Algorithm 4.3がある．ツリー型総和処理ではまず**図 1**のような，各コアを葉ノードとしコアのペアを積み上げる平衡二分木を作り，葉の階層からroot方向に1ステップずつ，ペアの中で部分和を計算することで，最終的に総和を得る．

ツリー型総和処理は，ペアごとに処理を進めていくため，通信パターンはコア間の1対1の形となる．また，ツリー型総和処理のペア間の計算では，ループの回転数が配列の長さと一致し，バリア同期を用いたアルゴリズムと比べて長くなる．これは，コア間通信のレイテンシー隠蔽に役立つと考えられる．また，ツリー型総和処理では処理の中にツリー型同期が埋め込まれており，バリア同期を別に行う必要がないことも利点である．

ツリー型同期処理の欠点としては，演算の並列性の悪化が挙げられる．ツリー型総和処理ではツリーの根に行くほど演算に関与できるコアの数が減少するため，常にすべてのコアを利用するバリア同期を用いたアルゴリズムと比べて，並列化効率が悪化する．そこで，コア間通信が問題にならない場合は，バリア同期を用いたアルゴリズムの方が高性能となると考えられる．

ツリー型の総和処理のソースコードは次のようになる．

```
void reducetup(int id, int nc,
   volatile int* count, int* myc,
   int n, double** ylist){
   for(int skip=1; skip<nc; skip<<=1){
      ++*myc;
      if((id%(skip<<1)) || (id+skip >=nc)){
         set(count + id, *myc);
      } else {
         wait(count + id + skip, *myc);
         for(int i=0; i<n; ++i)
            ylist[id][i] += ylist[id+skip][i];
         set(count+id, *myc);
}}}
```

この関数は，自分のコア番号idとコア数ncを必要とする．nやylistはreduceと同じものである．countとmycはツリーを構築するために必要なデータであり，前者は各コアがツリー型総和処理のどの段階に進んでいるかを保存する配列であり，mycはcountのうち，自分のコアに対応するもののコピーを保存している．ただし実装では，単純な配列ではなく，countの配列のそれぞれの要素がキャッシュラインの先頭に位置するよう，適切な大きさのパディングを入れている．このアルゴリズムは，各階層ごとに，自分がroot側であればペアとなるコアを待ち (wait)，その後，ペアの中での部分和を計算して，自分の完了を通知する (set)．root側でなければ，単に自分の完了を通知するという手順を階層分だけ繰り返す．

このアルゴリズムではcount配列内のデータをカウントアップし続けるため，int変数がオーバーフローすると，デッドロックに陥るというよく知られた問題がある．しかし三重対角化においてこの関数が呼ばれる回数はたかだか$3n$回であるため，極めて大きな行列でなければ問題ない．

この関数から呼び出されるset関数やwait関数は，ペアとなるコア間の同期を取る機構となっており，コードは次のようになっている．

```
void set(int* ptr, int v){
   __atomic_store(ptr, &v, __ATOMIC_RELEASE);
}
void wait(int* ptr, int v){
   while(1){
      int t;
      __atomic_load(ptr, &t, __ATOMIC_ACQUIRE);
      if(t >= v) break;
      // nop; nop; nop; ...
}}
```

これらの実装では，ロード・ストア命令における順序を一定にするため，gccの__atomicビルトイン関数を用いている．wait関数中のコメント行には，spin-waitの回転速度を抑えるために適当な回数のnopを，Intel環境ではより適切なpause命令を入れている．

reducetupは以上の同期機構のために，rootのコア (ここではコア番号が0のコア) がこの関数をぬけ出すときには，他のすべてのコアはすくなくともこの関数を呼び出していることが保証できる．その逆方向の同期のためにはツリーを下る操作が必要となる：

```
void tdw(int id, volatile int* count, int* myc){
   int twidth = 4;
   ++*myc;
   if(id) wait(count + (id-1)/twidth, *myc);
   set(count + id, *myc);
}
```

tdwでは総和演算を行う必要がなく，自由度が大きいため，reducetupとは異なる木構造を作っている．

3.3 三重対角化へのツリー型総和処理の適用

三重対角化の手順では，総和演算を行った後，その結果を用いた計算 (u_{i-1} や v_i の計算など) は1つのコアで行われる．そこで他のコアはこの1コア上の計算の結果を待たなければならない．つまり次のような擬似コードになる：

```
reducetup(id, ...);
tdw(id, ...);
if(id==0){
```

```
   // some computations
}
#pragma omp barrier
```

最後の omp barrier は，他のすべてのコアが root のコアを待つだけであるから，tdw に置き換えることができる．そうすると，二行目にある tdw は無駄な処理となるため，次のように簡単化できる：

```
reducetup(id, ...);
if(id==0){
   // some computations
}
tdw(id, ...);
```

これによって 1 ステップあたり 3 回必要であったバリア同期の回数が 2 回に減少した．ただし，omp barrier の中で用いられている同期アルゴリズムと reducetup や tdw において用いられているアルゴリズムとでは全く異なるものであることに注意が必要である．

Ramos ら [11] はツリー構造を最適化することにより KNL 上で，バリアや総和 [*2] などの演算の高速化ができることを示している．我々の実装ではごく単純な構造を使っているため最適化されたバリア同期と比べると性能が低いだろうと予測される．また，同期だけではなく総和処理全体の時間や，三重対角化全体の時間がどう変化するのかがここでは重要であるため，実際のどの程度の性能になるのか，実験的に調査する必要がある．

4. 実験結果

4.1 実験の概要

我々は次の 2 つの実験を行った．1 つは総和処理自体の実行時間の測定であり，複数のアーキテクチャ上で実験を行った．後に示す通り，この実験の結果，ツリー型総和処理が効果的であったのは KNL のみであった．そこで，2 つ目の実験では KNL 上で 2 種類の総和アルゴリズムを用いて三重対角化を実装したときの，実行性能を測定した．

実験で用いたシステムは次の 3 つである．

- KNL: Intel Xeon Phi 7210, 1.3GHz, 64 cores, 32MB L2 cache, 16GB MCDRAM,
- SKYL: Intel Xeon Gold 6148, 2.4GHz, 20 cores × 2 sockets, 37.5 × 2MB L2 + L3 cache,
- FX100: Fujitsu SPARC64 XIfx, 1.975GHz, 32 cores, 24MB L2 cache.

このうち SKYL と FX100 は理化学研究所のスーパーコンピュータシステム HOKUSAI のうち，それぞれ BWMPC と GWMPC を用いている．KNL と SKYL は演算内容に応じてクロック周波数が変化し，本稿のような浮動小数点数中心の演算ではそれぞれ 1.1GHz [12], 1.6GHz [13] にダウンクロックされる．そこでそれぞれのシステムにおけるピーク性能は，KNL, SKYL, FX100 の順に，2252.8GFlop/s, 2048GFlop/s, 1011.2GFlop/s となる．KNL はメモリシステムのモード切替が可能であるが，今回の実験ではすべて Quadrant/Flat モードに固定している．また，KNL は MCDRAM の他に DDR4 DRAM を搭載しており，どちらを利用するか選択できるが，今回は numactl -m 1 を指定することで，MCDRAM のみを用いるように設定している．FX100 は 3 システムの中でも特別に，ハードウェアバリア機構を持った CPU となっている．このため，単純なバリア同期は他の CPU と比べて速い．

それぞれのシステムで用いたコンパイラとコンパイルオプション，環境変数は次のとおりである．

- KNL: icc 18.0.1, -O3 -fno-alilas -g -qopt-prefetch -xMIC-AVX512 KMP_HW_SUBSET=64c,1t
- SKYL: icc 17.4.196, -O3 -fno-alilas -g -qopt-prefetch -xCORE-AVX512 OMP_NUM_THREADS=40 OMP_PLACES=sockets OMP_PROC_BIND=true
- FX100: fccpx 2.0.0, -Xg -Kfast,openmp,ocl -KHPC_ACE -KHPC_ACE2 OMP_NUM_THREADS=32 FLIB_CPUBIND=chip_pack

また FX100 では lpgparm コマンドによってページサイズを 512KB に設定している．

4.2 総和処理の実行時間

第 1 の実験では，3 システムにおけるバリア同期を用いたアルゴリズムとツリー型総和処理との性能比較を行う．実験では配列のサイズを 2 から 2 倍ずつ大きくしたときの実行時間を測定しており，各サイズで 100 回実行したときの実行時間を 100 で割った値を平均値としている．ただし各実行では必ず配列の初期化も行っており，実行時間はその初期化時間をも含んだものとなっている．これはキャッシュ上のデータ位置が性能に影響を及ぼすため，三重対角化の中でのデータアクセス順序をある程度反映したテストにするためである．擬似コードは次のようなものになる：

```
#pragma omp parallel
{
   int id = omp_get_thread_num();
   ylist[id] = malloc(sizeof(double)*n);
   double time_begin = current_time();
   for(int r=0; r<100; ++r){
      for(int i=0; i<n; ++i) ylist[id][i] = r + i;
      // reduce algorithm
   }
   double time_end = current_time();
```

[*2] 彼らは 1 要素の総和のみを議論しており，本稿のように配列の総和は取り扱っていない．

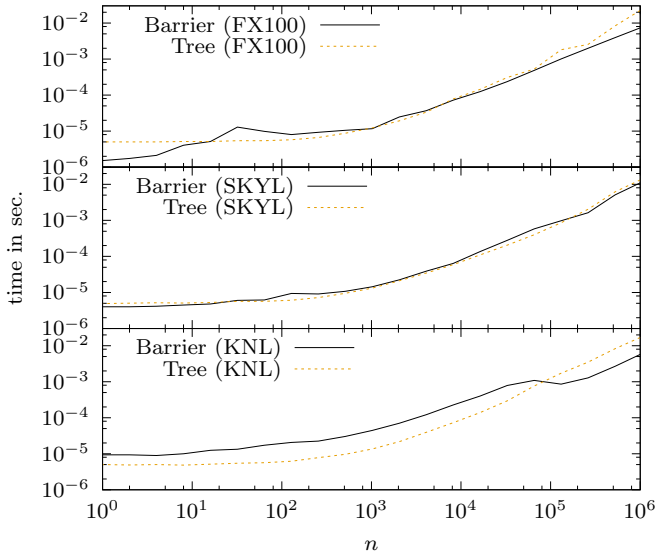

図 2　3 システムにおける総和処理の実行時間（秒）．横軸は配列の要素数．

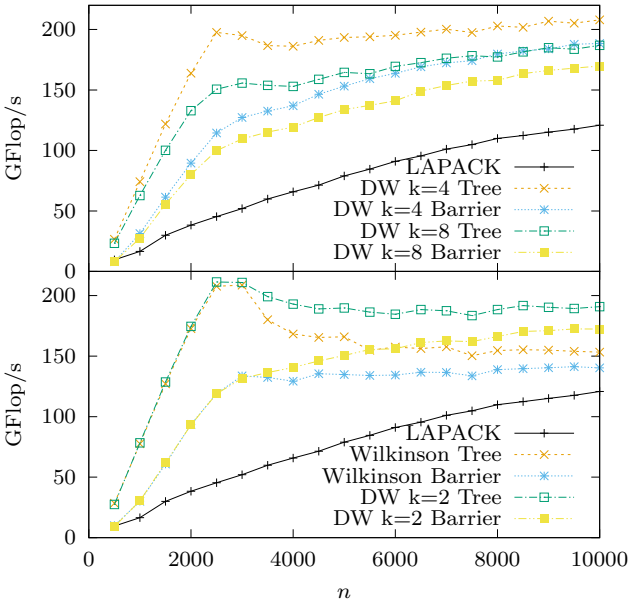

図 3　三重対角化の実行性能（GFlop/s）．横軸は行列サイズ．

```
    print(time_end - time_begin);
}
```

擬似コード中のコメント行を `reduce` または `reducetup` と `tdw` のペアのどちらかに置き換える．また `current_time` は高精度タイマーで計測した時刻であり，`clock_get_time` を利用している．また `reduce` の実際の実装においては，高性能化のため，変数 `j` に関する外側ループをブロック幅 4 でブロック化し，最内側処理を 4 つ分だけアンロールしている．

図 2 はそれぞれ以上の実験を行ったときの，KNL, SKYL, FX100 上での総和処理の実行時間である．図中の Barrier はバリア同期を用いたアルゴリズム，Tree はツリー型総和処理である．3 システムそれぞれ図の様相が異なるが，三重対角化で重要となる図の左右中央付近（要素数 100 から 10,000 の範囲）において，FX100，SKYL では両手法が同程度の速度になっているのに対して，KNL では Tree が Barrier よりも高速になっている．実行時間の比は要素数 64 から 8192 の間ではすべて 3 倍以上となっており，性能が逆転した要素数 131072 以降は 3 分の 1 程度となっている．

以上の通り，KNL では三重対角化で重要となるような要素数の時に，Tree が Barrier よりも高速となった．そこで KNL 上で実際にこの総和アルゴリズムを三重対角化に組み込んだときの速度に興味がある．

4.3　三重対角化の性能

図 3 は 2 つの総和アルゴリズムを用いたときの，KNL 上での三重対角化の実行性能を示している．ここで三重対角化の実行性能は，行列サイズ n に対して，演算量を $\frac{4n^3}{3}$ とおいたときの Flop/s 値としている．実験では三重対角化手法として，Wilkinson の手法，DW 法の $K = 2, 4, 8$ のものの 4 種類を用いており，グラフ下部では Wilkinson の手法と DW 法の $K = 2$，グラフ上部では DW 法の $K = 4, 8$ のものを示している．またそれぞれのグラフで LAPACK の三重対角化サブルーチンである DSYTRD の性能も示している．LAPACK は Intel MKL 2018.0.1 に付属のものを用いている．行列サイズは $n = 500$ から 500 刻みで $n = 10,000$ までを用いている．

図から，バリア同期を用いたアルゴリズムとツリー型総和処理とでははっきりと性能差が読み取れる．どの程度後者が速いかは，行列サイズと三重対角化手法などによって異なるが，$n = 2,500$ のときは約 50%から 77%後者が高性能となっており，$n = 5,000$ のとき，約 23%から 26%，$n = 10,000$ のとき，約 9%から 11%，後者が高性能となっている．この実験の範囲では前者が後者よりも高速となるものはなかった．

ツリー型総和の中でも，$n = 2,500$ のときのピーク性能は DW 法の $K = 2$ のものが最も高く，約 211GFlop/s となっており，これは同サイズの LAPACK の約 4.7 倍の性能である．$n = 5,000$ のときは $K = 4$ の DW 法が最も速く，約 194GFlop/s であり，LAPACK の約 2.5 倍の性能である．また $n = 10,000$ のときも $K = 4$ の DW 法が最も速く，約 208GFlop/s であり，LAPACK の約 1.7 倍の性能である．

このように，KNL 上において，ツリー型総和処理は三重対角化の中でも高速化に効果があり，とくに行列サイズが小さいときに顕著となった．

表 1　データパッキングによる三重対角化の実行性能 (GFlop/s)

		$n = 2,500$,	5,000,	10,000
Wilkinson	Direct	207.8	166.0	153.3
	Packed	161.0	145.4	144.7
DW $K = 2$	Direct	211.3	189.8	191.0
	Packed	166.2	161.7	177.7
DW $K = 4$	Direct	197.6	193.5	208.0
	Packed	157.2	173.1	193.9
DW $K = 8$	Direct	150.6	164.5	187.0
	Packed	152.1	165.3	192.0

5. 議論

本稿では，三重対角化の共有メモリ並列化において，総和処理が問題になることを示し，ツリー型総和処理を用いたその改善手法と，三重対角化への適用手法について述べた．また，多数のコアを持つ複数の CPU 上でツリー型総和処理の性能評価をした結果，単純なバリア同期を用いたアルゴリズムと比べると，2 つの CPU 上では同程度の性能となったが，KNL 上では大きな改善効果があった．そこで三重対角化プログラムにツリー型総和処理を組み込み，KNL 上で性能評価を行った結果，とくに行列サイズが小さいところで大きな性能改善効果が見られた．

本研究のツリー型総和処理は最も基本的な形のものである．そこで，他の総和アルゴリズムや，SYMV の実装手法との組み合わせなどによる性能改善が可能であるか，検討の余地がある．また，総和処理の実行時間が，三重対角化で使われるような配列長さの範囲であれば，要素数に対して線形に近い実行時間となっている点は興味深い．これは，帯幅の増大を代償に複数の SYMV を同時計算可能にする Bischof のアルゴリズム [14] を適用した場合でも，総和処理の部分の性能改善は小さいことを示している．そこで今後の性能改善で実行時間がより線形に近づくのか，それとも非線形的となるのかについても興味がある．

本研究は筆者の以前の研究 [6] の成果を最新のプロセッサに適用することを出発点としていたが，当時有効だったデータパッキング手法は今回取り入れられていない．**表 1** に，ツリー型総和処理を用いた場合の，データパッキングの有無による三重対角化の実行性能を示す．実験条件は 4.3 節のものと同じである．表からわかる通り，$K = 8$ を除いて，データパッキングが性能を劣化させている．実行時間の詳細を確認すると，実際にはデータパッキングによって SYMV や SYR2KMV の計算自体は高速化されているが，SYR2KMV に用いられる u_i や v_i のデータパッキングの実行時間の増加が上回っている状況である．そこで行列のデータパッキングの有無と u_i, v_i のデータパッキングの有無を独立に設定できれば，より高い性能が得られる可能性がある．

しかしながら，このような多くのパラメータの組み合わせに対応するプログラム生成が現状の大きな障害となっている．また今後，キャッシュ制御命令の有無やプリフェッチ距離の変更など，性能改善に繋がると推測されるパラメータを取り込み，さらに，ここで用いた技術は三重対角化だけではなく，似たような構造を持つヘッセンベルグ化や二重対角化などにも応用したいと考えている．これまではアドホックに開発されたスクリプトによってしのいできたが，より汎用性と記述性に優れたプログラム生成言語の開発を検討している．

謝辞　電気通信大学 山本有作教授には研究の多くの面でサポートしていただきました．感謝いたします．

参考文献

[1] Grama, A., Gupta, A., Karypis, G., Kumar, V.: *Introduction to Parallel Computing Second Edition,* Addison Wesley, Massachusetts (2003).
[2] Wilkinson, J. H.: *The algebraic eigenvalue problem,* Chap. 5, Sec. 32, Oxford Univ. Press (1965).
[3] 寒川 光: RISC 超高速化プログラミング技法, 共立出版株式会社 (1995).
[4] 村上 弘: 両側ハウスホルダ変換に対する Wilkinson の著書 AEP の中の「技巧」について, *IPSJ HPC*, Vol. 208, No. 125, pp. 67-72 (2008).
[5] Dongarra, J. J. and Sorensen, D. C.: Block reduction of matrices to condensed forms for eigenvalue computations, *Journal of Comp. and Appl. Math.,* Vol. 27, Issues 1-2, pp. 215-217 (1989).
[6] 工藤 周平, 山本 有作, 横川 三津夫: 三重対角化に対する Dongarra-Wilkinson 法の性能解析と実装手法について, *IPSJ HPCS*, Vol. 2015, pp. 19-28 (2015).
[7] 工藤 周平：KEVD github リポジトリページ，入手先 ⟨https://github.com/shuheikudo/Kevd⟩（参照 2018-06-29）.
[8] Nath, R., Tomov, S., Dong, T.T. and Dongarra, J.J.: Optimizing symmetric dense matrix-vector multiplicatoin on GPUs, *Proc. SC '11,* Article No. 6 (2011).
[9] Xianyi, Z., Qian, W. and Saar, W.: OpenBLAS An optimized BLAS library, 入手先 ⟨www.openblas.net⟩,（参照 2018-7-2）.
[10] A high performance SYMV kernel on a Fermi-core GPU, *VECPAR2012, LNCS,* Vol. 7851, pp. 59-71 (2012).
[11] Ramos, S., Hoefler, T.: Capabiity models for manycore memory systems: a case-study with Xeon PHI KNL, *Proc. IPDPS' 17* (2017).
[12] Intel: Intel Xeon Phi processor: your path to deeper insight, 入手先 ⟨www.intel.co.jp/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-processor-product-brief.pdf⟩（参照 2018-6-29）.
[13] Intel: Intel Xeon processor scalable family specification update, 入手先 ⟨www.intel.co.jp/content/www/jp/ja/processors/xeon/scalable/xeon-scalable-spec-update.html⟩（参照 2018-6-29）.
[14] Bischof, C., Lang, B. and Sun, X.: Parallel tridiagonalization through two-step band reduction, *PRISM Working Note,* TR 17 (1996).