タイル単位でパック化を行う行列計算

工藤 周平[1,a)]　今村 俊幸[2]

概要：タイル型の行列計算は依存関係ベースのスケジューリング手法との組み合わせにより高度な並列化を容易に実現できるが, 個々の計算が小さくなるため行列計算の持つ本来の速度を引き出すことが難しい. そこで行列計算の主要部品である行列積の性能に不可欠なパック化を陽に扱うことが考えられる. 本稿では正定値対称行列の逆行列計算を題材に, タイル型行列計算とパック化の組み合わせを検討し, パック化のスケジューリング上の問題点やスレッド並列環境におけるパック化の性能向上効果について示す.

Tiled matrix computations with explicit packing

1. はじめに

行列のタイル型レイアウトとはブロック分割した行列の1ブロック内でデータが連続になるデータレイアウトである. タイル型レイアウトを使った行列計算 (タイル型行列計算) はブロック単位の計算として表せる行列計算においてはデータ局所性が高い利点がある. また, ブロック単位の個々の計算を1つのタスクと見立て, タスク型計算として表すと見通しよくアルゴリズムを記述できる上に, 並列化も容易になることがある. とくにタスク間の依存関係をベースに動的なスケジューリングを行う手法は, 手動では難しい複雑な並列化を可能にすることで大幅な高速化を実現できる場合がある. しかしながらタイル型行列計算は個々のタスク性能が理想的な値よりも低下する弱点がある. なぜなら, 行列積のような重要な行列計算の部品は行列サイズが大きくなるほどデータ再利用性が高まり限界性能に近い速度に達するが, タイル型・タスク型行列計算のブロックサイズは元の行列サイズよりも小さな値となるためである.

そこで本研究では行列積の内部処理に着目し, タスク間でのデータ再利用を阻害する要因である「パック化」を陽的に扱うことを検討する. パック化とは行列のデータ順序を並び換える処理であり行列積の高速化に重要な手順であるが, 行列サイズが小さいとオーバーヘッドが大きい. そこでパック化したデータをタスク間で共有することでオーバーヘッドを削減できると考えられる.

本稿ではタスク型・タイル型行列計算として正定値対称行列 (SPD 行列) の逆行列計算を扱う. この計算は行列積が計算の中心となるためピーク性能は高く, 一見複雑な手順を巧妙に並び換えれば高い並列性が得られるため, タイル型・タスク型行列計算のベンチマークとして優れている. また SPD 行列の逆行列計算自体も応用として, データ同化手法である Kalman Filter の計算であったり, 行列関数の近似計算などに出現するため有用である.

タイル型・タスク並列の行列計算実装は PLASMA [1] が代表的である. 他に分散並列向けのものとして SLATE [2] がある. ただしタイルのパック化については行われていない. パック化を行列積の外で行うことで行列計算を高速化した例として, Intel の KNC 向け LINPACK 実装のもの [3] があるが, タイル型・タスク並列との組み合わせまでは行われていない. SPD 行列の逆行列計算について, 通常の実装手法 (3 sweeps 法) と 計算手順を並び換えて並列性を向上させた single sweep 法との比較を行っているものとして FLAME [4] の例がある.

本稿は次の構成となっている. まずこの節で概要を示した. 次節では SPD 行列の逆行列計算についてアルゴリズムとタスク型の計算手法の手順を示す. 第3節ではパック化を組み込んだときのアルゴリズムとその問題点を示し, single sweep 法での実装ではその問題を回避できることを示す. 第4節では性能測定結果を示し, パック化による効果を検証する. 最後に第5節で本稿をまとめる.

1　電気通信大学
University of Electro-communications
2　理化学研究所
RIKEN
a)　shuhei-kudo@uec.ac.jp

2. SPD 行列の逆行列計算

2.1 逆行列の計算手順

SPD 行列の逆行列計算は行列の持つ性質を用いると少ない演算量で計算できる．この手順ではまず SPD 行列の Cholesky 分解を行うことで行列を三角行列の積に分解し，次に三角行列の逆行列を計算し，最後に三角行列の逆行列同士し積を計算する．つまり，入力行列を $A \in \mathbb{R}^{n \times n}$ としたとき:

$$A = LL^\top, \quad (1)$$
$$L_2 = L^{-1}, \quad (2)$$
$$A^{-1} = \left(LL^\top\right)^{-1} = L_2^\top L_2. \quad (3)$$

この計算は対称または下三角構造を持つため上三角部分を計算する必要はなく，非零構造を使って計算すると演算量は合計で約 n^3 となる．

次にブロック化した場合のアルゴリズムを示す．[*1] 簡単のため行列サイズ n はブロック幅 b で割り切れるものとし，一辺のブロック数を L とする．そして A の (i, j) 位置のブロックを $A_{i,j}$ と表す．このとき上の 3 つの手順はそれぞれ次のように書ける．Cholesky 分解は $k = 1$ から L まで次の手順を行う:

$$A_{k,k} \leftarrow \mathrm{potrf}(A_{k,k}), \quad (4)$$
$$A_{i,k} \leftarrow A_{i,k}A_{k,k}^{-1}, \qquad k < i \leq L \quad (5)$$
$$A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}^\top \qquad k < j \leq i \leq L \quad (6)$$

次に三角行列の逆行列計算は，同じように $k = 1$ から L まで次の手順を行う:

$$A_{k,j} \leftarrow A_{k,k}^{-1}A_{k,j}, \qquad 1 \leq j < k \quad (7)$$
$$A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}, \qquad 1 \leq i < k < j \leq L \quad (8)$$
$$A_{i,k} \leftarrow A_{i,k}A_{k,k}^{-1}, \qquad k < i \leq L \quad (9)$$
$$A_{k,k} \leftarrow A_{k,k}^{-1}. \quad (10)$$

最後に三角行列同士の積は，同じように $k = 1$ から L まで次の手順を行う:

$$A_{i,j} \leftarrow A_{i,j} + A_{k,i}^\top A_{k,j}, \qquad 1 \leq j \leq i < k \quad (11)$$
$$A_{k,j} \leftarrow A_{k,j}A_{k,k}^\top, \qquad 1 \leq j < k \quad (12)$$
$$A_{k,k} \leftarrow A_{k,k}^\top A_{k,k}. \quad (13)$$

計算の実行順序は，上の行から下の行に進み，同じ行内の計算はどの順序 (並列) でもよいものとする．ただし式 (4) は LAPACK の関数 `Xpotrf` によって Cholesky 分解を計算し，下三角行列の結果を得ており，同様に式 (10) や式 (13) はそれぞれ LAPACK の関数 `Xtrtri` と `Xlauum` を使って計算する．また各式において入力行列が三角構造を持っていたり，結果が対称構造を持っていたりする場合はそれらの構造を使って演算量を削減する．

[*1] $b = 1$ とした場合に非ブロック版のアルゴリズムとなる．

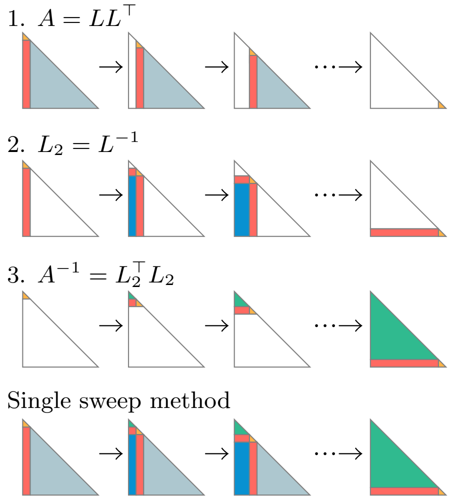

図 1 SPD 行列逆行列計算のデータ更新の様子
Fig. 1 An illustration of the computations of the inversion of a SPD matrix.

以上の通りブロック単位で記述すると簡潔に表記できるアルゴリズムであるためタスク型計算と相性が良い．つまり上の各行の式において添字の組を 1 つに固定した計算を 1 つのタスクとみなす．このときどの順序でタスクを実行していくかが問題である．上で記述したままに k を増加させながら一行ずつ計算していくと，k の値や行によってタスク数が大きく変化し，並列化が難しい．そこで手動でタスク実行順序を並び換えて並列性を高める手法 (single sweep 法) とタスク間の依存関係を用いて自動的にタスク実行順序を並び換える手法が考えられる．

2.2 single sweep 法

single sweep 法は逆行列計算の 3 つの手順を並び換え 1 回の走査で計算する手法であり，元の手順 (3 sweeps 法) と比べて同時に実行できるタスク数が多くなる．図 1 は 3 sweeps 法と single sweep 法のデータ更新の様子を図示したものである．図中の三角行列のうち色で塗り潰した面を各ステップにおいて書き換える．図で分かる通り，3 sweep 法の 3 つの手順は前の手順が完了した領域に対して書き換えを行うため，図中の縦に並ぶ計算 (同じ k に対する計算) のうち赤・オレンジ色で示した領域の計算のみ順に行えば，残りの計算は同時に実行できる．そのように計算順序を並び換えたものが single sweep 法である．

single sweep 法の手順を式で書くと次のとおり．$k = 1$ から L について

$$A_{k,k} \leftarrow \mathrm{potrf}(A_{k,k}), \quad (14)$$
$$A_{i,k} \leftarrow A_{i,k}A_{k,k}^{-1}, \qquad k < i \leq L \quad (15)$$

$$A_{k,j} \leftarrow A_{k,k}^{-1} A_{k,j}, \qquad 1 \le j < k \tag{16}$$
$$A_{i,j} \leftarrow A_{i,j} - A_{i,k} A_{j,k}^{\top}, \qquad k < i \le L \tag{17}$$
$$A_{i,j} \leftarrow A_{i,j} - A_{i,k} A_{k,j}, \qquad 1 \le i < k < j \le L \tag{18}$$
$$A_{i,j} \leftarrow A_{i,j} + A_{k,i}^{\top} A_{k,j}, \qquad 1 \le j \le i < k \tag{19}$$
$$A_{i,k} \leftarrow A_{i,k} A_{k,k}^{-1}, \qquad k < i \le L \tag{20}$$
$$A_{k,k} \leftarrow A_{k,k}^{-1}, \tag{21}$$
$$A_{k,j} \leftarrow A_{k,j} A_{k,k}^{\top}, \qquad 1 \le j < k \tag{22}$$
$$A_{k,k} \leftarrow A_{k,k}^{\top} A_{k,k}. \tag{23}$$

この内, 式 (15) と式 (16) は並列実行可能であり, また式 (17) から (19) までの行列積についても並列実行可能である．つまり, 単に式の実行順序を並び換えただけでより高い並列性を得ることができる．

single sweep 法の欠点は, 元の手順が持つ論理的な境界を無くしてしまっている点, そのために部品ごとの再利用性が失われており, 加えてプログラムが複雑になってしまっている点がある．

2.3 依存関係ベースのタスク並列化

より汎用性のある手法として依存関係ベースのタスク並列化手法がある．依存関係ベースのタスク並列化手法はタスク間の依存関係を表すグラフ (タスクグラフ) を解析し, 依存関係を破壊しないように制御しつつタスクを (静的または動的に) 並列スケジューリングする手法であり, 前項のような手動の並び換えでは困難となる高度なスケジューリングを可能とする．依存関係ベースの動的タスク並列化の仕組みはすでに OpenMP の仕様 [5] (15.9 節) に取り込まれ各種コンパイラにも実装されているため, 容易に実現することが可能となっている．

逆行列計算に対して依存関係ベースのタスク並列化を行うことは簡単にできる．上式からどのタスクがどのブロックを読み・書きするかは明らかであるため, その情報を OpenMP のランタイムに伝えれば良い．依存関係ベースのタスク並列化では single sweep 法のような並び換えを手動ですることは不要であり, 3 sweeps 法の自然な記述を用いてタスクグラフを構築すれば自動的に実行順序の並び換えが行われる．

2.4 ブロックサイズについてのトレードオフ

以上の計算手順ではタスク数が多くなるほどコアへ均等にタスクを割り振ることができるため, 並列化効率は向上する．一方で同じ行列サイズでタスク数を増やすにはブロックサイズを小さくする必要があるが, ブロックサイズが小さくなると個々のタスクの実行効率が減少する．そこで両者のバランスの中で良いブロックサイズを決定する必要がある．

このトレードオフ関係を打開するには, 前述のような手法によってより少ないタスク数であってもより均等にタスクをスケジューリングすることと同時に, より小さいブロックサイズでも個々のタスクの実行効率を維持することが必要となる．そこで後者に取り組むために行列積計算の内部処理であるパック化のオーバーヘッドに着目し, パック化の結果をタスク間で共有することでオーバーヘッドを減らす可能性を検証することが本研究の目的である．

3. パック化を陽的に扱う実装の問題点と解決策

パック化とは行列のデータを並び換えながら別の領域へコピーする行列積の前処理である．パック化自体は追加コストであるが, パック化したデータを十分な回数だけ再利用できれば追加コスト以上の性能向上を得られる可能性がある．そこでパック化をタスクごとに冗長に行うのではなく, パック化したデータを複数のタスクで共有することでパック化のオーバーヘッドを減らしたい．しかしながら, タスク間でのデータ共有は本来ないはずのタスク間の依存関係を引き起こし, スケジューリングの問題が発生する．またパック化したデータに対する行列積は通常の行列積ルーチンとは異なるため, 実装面の問題にも対処する必要がある．

3.1 パック化を陽的に扱う手順と問題点

パック化の問題点を説明するため, 上記の Cholesky 分解の手順にパック化を組み込むことを考える．その手順は次のように $k = 1$ から L について:

$$A_{k,k} \leftarrow \mathrm{potrf}(A_{k,k}), \tag{24}$$
$$\Big[A_{i,k} \leftarrow A_{i,k} A_{k,k}^{-1}, \quad X_i \leftarrow \mathrm{packA}(-A_{i,k}), \quad Y_i \leftarrow \mathrm{packB}(A_{i,k}^{\top})\Big], \qquad k < i \le L \tag{25}$$
$$A_{i,j} \leftarrow \mathrm{gemmPP}(X_i, Y_j, A_{i,j}), \qquad k < j < i \le L \tag{26}$$
$$A_{i,i} \leftarrow \mathrm{syrkPP}(X_i, Y_i, A_{i,i}). \qquad k < i \le L \tag{27}$$

ここでは説明のために次のルーチンを使った．`packA` は行列積の左側に使うデータのパック化を行うルーチンであり, `packB` は同様に右側のためのルーチンである．`gemmPP` は引数を 3 つとり, 1 つ目と 2 つ目のパック化された行列の行列積の結果に 3 つ目の行列を足した値を計算する．`gemmPP` のバリエーションとして, 下三角部分の要素のみを計算する `syrkPP` と, 一つ目の行列が三角行列である場合の行列積である `trmmPP` とがある．また複数の手順を 1 つのタスクとして実行するものを角括弧 ([,]) でまとめて書いた．

この手順ではパック化したデータを保存するためにワーク領域である X_i, Y_j を新たに用意している．パック化したデータは最大 $L-1$ 回再利用されるため, タイルごとにパック化するのに比べてパック化の回数を減らせる．しかしな

がら隠された問題点に気付く必要がある．それは**ワーク領域を介した本来不要な依存関係が発生**していることである．つまり k で使用したワーク領域 X_i, Y_i は $k+1$ のループで再利用するため, 次のループのパック化は前のループの完了を待つ必要がある．言い換えれば, パック化の結果をタスク間で共有することでタスクグラフが変化し, タスクスケジューリングの自由度が減少している．

3.2 パック化による疑似依存

実際にはパック化によって新たに発生した依存関係は疑似依存である．なぜならば, ワーク領域への読み込みを書き込みが待たなければならない関係, すなわち R → W 依存であり, リネーミングによって解決できるためである．

しかしながらその解決策が現実的であるかや実現が容易であるかは別問題である．例として, 解決策の 1 つは完全なリネーミングを行い, パック化のたびに新しいワーク領域を割り当てることであるが, この方法では大量のワーク領域を確保する必要がある．さらに高度な解決策として, タスクスケジューリング管理機構とワーク領域管理機構とを協調動作させ, ワーク領域の容量上限を満たしつつ良いスケジューリングを行う機構を作ることが考えられるが, システム構築に加えてスケジューリング手法の開発を行うことになり大きな労力が必要となる．

より単純かつ現実的な解決策は, 手動でスケジューリングを行った後にパック化を導入することである．つまり 3 sweeps 法にパック化を入れる代わりに single sweep 法に変換した後にパック化を入れれば, 疑似依存によるスケジューリング制約を大幅に緩和しつつパック化を導入することが可能となるのである．もちろんこの場合であっても疑似依存によるスケジューリング制約は完全には除去できないが, 現実的な手間で実現可能であることは大きな利点である．

3.3 single sweep 法へのパック化の導入

single sweep 法にパック化を組み込んだ手順を示すと次のようになる．上記と同じように $k=1$ から L について:

$$\begin{aligned}
\Big[A_{k,k} &\leftarrow \mathrm{potrf}(A_k, k), \\
Y_k &\leftarrow A_{k,k}, \\
A_{k,k} &\leftarrow A_{k,k}^{-1}, \\
X_k &\leftarrow \mathrm{packA}(A_{k,k}), \\
A_{k,k} &\leftarrow A_{k,k}^{\top} A_{k,k}\Big], && \quad (28)
\end{aligned}$$

$$\begin{aligned}
\Big[A_{k,j} &\leftarrow Y_k^{-1} A_{k,j}, \\
X_j &\leftarrow \mathrm{packA}(A_{k,j}^{\top}), \\
Y_j &\leftarrow \mathrm{packB}(A_{k,j})\Big], && 1 \le j < k \quad (29)
\end{aligned}$$

$$\begin{aligned}
\Big[A_{i,k} &\leftarrow A_{i,k} Y_k^{-\top}, \\
X_i &\leftarrow \mathrm{packA}(-A_{i,k}), \\
Y_i &\leftarrow \mathrm{packB}(A_{i,k}^{\top}), \\
A_{i,k} &\leftarrow A_{i,k} A_{k,k}^{-1}\Big], && k < i \le L \quad (30)
\end{aligned}$$

$$A_{i,j} \leftarrow \mathrm{gemmPP}(X_i, Y_j, A_{i,j}), \quad 1 \le j < i \le L, \quad i, j \ne k \quad (31)$$

$$A_{i,i} \leftarrow \mathrm{syrkPP}(X_i, Y_i, A_{i,i}), \quad 1 \le i \le L, \quad i \ne k \quad (32)$$

$$A_{k,j} \leftarrow \mathrm{trmmPP}(X_k, Y_j). \quad 1 \le j < k \quad (33)$$

この手順は single sweep 法とパック化という 2 つの工夫を組み合わせているためこれまでのどの手順よりも長くなってしまっているが, 手順としては簡単なものとなっている．つまり式 (28) の (k,k) ブロックに対する処理と, 式 (29), 式 (30) の 1 行 1 列に対する処理, そして最後に残りのブロックに対する行列積の 3 つの手順である．

3.4 パック化と行列積カーネルの実装手法

パック化は行列積の内部ルーチンとして組み込まれており, 通常は両者を分離して用いることができない．一部の環境向けにはパック化を分離した行列計算ライブラリが手に入るが, 機能が不足しており使用に工夫が必要である．

3.4.0.1 既存のパック化 API を使う方法

Intel MKL にはパック化向けの行列積用の API が用意されている [6]. その関連ルーチンは次の 3 つである:

- `Xgemm_pack`: 行列積に使う入力データについて, 左右, 転置, スケーリングを指定して並び替えを行う．
- `Xgemm_pack_size`: パック化したデータを格納するのに必要なデータサイズを指定する．
- `Xgemm_compute`: パック化した (またはしていない) 行列を与えて行列積を計算する．

残念ながら他のタイプの行列計算, `syrkPP` や `trmmPP` に相当する機能は含まれていないため, それらの機能を実現するには工夫が必要となる．1 つの方法は結果の対称性や行列の非零構造を無視し, 単純な行列積 `Xgemm_compute` で代用する方法であるが, 演算量が無視できないほど増大する．もう一つの方法は, パック化する前のデータを残しておき, `syrkPP` や `trmmPP` を行う際には対応する非パック化向けのルーチンを用いるものがあり, 手順が複雑になるが演算量の増大を防ぐことが可能となる．

我々は Intel MKL のパック化向け行列積用 API を使った逆行列計算実装も作成したが, 次に示す行列積を自作するもの比べて性能差はほとんど無かった．

3.4.0.2 行列積を自作する方法

既存の実装を用いない場合は自前でパック化向け行列積を実装する他ない．幸い, 主要な CPU 向けには Open-

BLAS [7] や BLIS [8] といったオープンソースの行列積実装が存在するため, それらを参考にすれば大きな手間をかけずに行列積を実装することができる. 実際に既存ライブラリと同程度の性能を得られるかどうかは大きな課題であるが, 本稿において我々は AVX2 命令セットと FMA 命令セットを持った x86_64 環境向けのパック化を分離した行列積を実装し, 既存ライブラリに匹敵する性能に到達できることを示す.

4. 性能測定

ここでは複数の CPU 環境上で我々の作成した SPD 行列逆行列計算ルーチンの性能を比較し, パック化の効果を示す. 使用する CPU は表 1 に示した 3 つの構成 (CZN, KBL, ADL-S) であり, ベースの命令セットは同じであるがメーカーや世代の異なるものとなっている. また ADL+E は上記 ADL-S の Hybrid architecture を活かしたものとなっており, Golden Cove と Gracemont という 2 種類の性能の異なるコアを含む構成である. そこで ADL+E はコア数以上に高いスケジューリング性能が求められるものとなっている.

性能測定ではまず前準備として, 我々が作成したパック化向け行列積ルーチンの性能を Intel MKL や OpenBLAS のものと比較し, 十分に実用的な性能となっていることを示す.

次に行列サイズとタイルサイズを変化させながら, 5 種類の逆行列計算手法の計算時間を比較する. 使用した逆行列計算手法のうち lapack は標準的なライブラリ (lapack) を用いるものでありタイルレイアウトやタスク並列化を行っていないものである. G3N と G3P, G1P は依存関係ベースの動的なタスク並列化を行うものであり, 3 sweeps 法を用いるもの (G3N, G3P), single sweep 法を用いるもの (G1P), 通常の BLAS を用いるもの (G3N, パック化を組み合わせたもの (G3P, G1P) とがある. [*2] また比較対象として L1P という手法を用意した. L1P はパック化を行う single sweep 法であるが依存関係ベースの並列化ではなく手動での並列化を行うものであり, 行列積におけるキャッシュ再利用性を高めるループ実行順序の設定と先読み計算によるタスク並列性の向上を手動で行うため, 後に示すように行列サイズが大きいところでのピーク性能比が他と比べて高くなっている.

行列やバッファ領域のメモリは全て Transparent Huge Page の仕組みを用いて可能な限り Huge Page を用いるようにしている. また, 性能測定時間が長くなるため Intel の Turbo boost のようなオーバークロック機能を無効にすることでベース周波数で動作するように設定した上で, 平均消費電力や CPU 温度が設定値を越えないように適度なタイミングで計算停止時間を入れることで動作周波数が固定されるように工夫している. 性能測定は同じパラメータの組み合わせのものを 12 回行い, 内部バッファの初期化処理が行われる最初の 1 回の結果を除いた残りの 11 回の標本平均と標本標準偏差を示している.

4.1 行列積性能の測定結果

表 2 に 4 つの環境におけるタイル型レイアウトにおける行列積の性能を示した. ここではパック化を内部で行う行列積 (Intel MKL または OpenBLAS) とパック化したデータに対する行列積 (Intel MKL または自前の実装のもの) とを複数の環境で比較しており, 前者が内部のパック化の時間を含むのに対して後者はパック化の時間を一切含んでいない点が大きな違いである. またブロックサイズはそれぞれの実装・環境で良い性能がでるものを探索し使用しており, 全体の行列サイズはブロックサイズで割り切れる中で大きなものとしている.

表の結果から全ての環境においてパック化向けルーチンを用いたものが高い性能を示しており, ピーク性能比を 6% から 15% 向上させていることが分かる. この結果はパック化向け行列積をタイル型行列計算に用いることの動機を単的に説明している. また自作のパック化向け行列積ルーチンが Intel MKL のパック化向け行列積ルーチンを同程度の速度となっており, オープンな情報のみから CPU ベンダーの開発した行列積ルーチンと同程度の性能を出せることを示せた. そこで, 実際の行列計算に出現するような (単純な形の行列積以外の) 様々なパターンのタスクに対してパック化の分離を行える可能性を示している.

4.2 逆行列計算の性能測定結果

ここでは行列サイズやブロックサイズを変えたときの各種逆行列計算手法の性能を測定した結果を示す. 行列サイズは $n = 470$ から $8,200$ までの E-12 系列とし, ブロックサイズは環境に応じて次の 2 種類としている. Intel 環境 (KBL, ADL-S, ADL+E) でのブロックサイズは $b = 96$, 120, 180, 360 であり, 残りの環境 (CZN) でのブロックサイズは $b = 96$, 120, 180, 256, 360, 480 である. Intel 環境におけるブロックサイズは行列積の性能測定結果から判明したよいブロックサイズ $b = 180$ を中心に, キリのよい数字を選択したものである. CZN においてブロックサイズの種類を 2 つ増やした理由は, OpenBLAS のブロックサイズの内の 1 つに一致する $b = 256$ と, 自作の行列積ルーチンのブロックサイズ $b = 480$ に一致するブロックサイズを含めるためである.

まず図 2 に各パラメータの組み合わせごとに最も平均性能の良いブロックサイズの場合の性能を示す. ここから, 全ての環境に共通して LAPACK が最も遅いこと, 3 sweeps

---

[*2] single sweep 法とパック化無しの組み合わせはタスクグラフ上では 3 sweeps 法とパック化無しの組み合わせと同一であるため省いている.

表 1 CPU の仕様一覧
Table 1 CPU specifications

	ADL-S/ADL+E		KBL	CZN
platform	Alderlake-S		Kabylake	Cezanne
micro architecture	Golden Cove	Gracemont	Kabylake	Zen3
# of cores	8	4	4	8
L1D cache size (core, KB)	48	32	32	32
L2/LL cache size (cpu, MB)	25		7	20
Flop/cycle	16	8	16	16
clock freq. (GHz)	2.1	1.6	2.5	3.3
peak perf. (GFlop/s)	268.8	51.2	160	422.4
compiler	GCC `12.1.1 200220507`			
BLAS library	Intel MKL `2020.4.304`			OpenBLAS `0.3.20`

表 2 タイルごとの行列積とパック向け行列積のピーク性能比
Table 2 The peak-ratio of GEMM and tiled-GEMM

	$n = m$	k	BLAS (MKL or OpenBLAS)	Intel MKL pack API	our `gemmPP`
ADL-S	$7,960$	180	0.916 ± 0.000	0.971 ± 0.000	0.977 ± 0.001
ADL+E	$7,960$	180	0.766 ± 0.002	0.815 ± 0.000	0.820 ± 0.000
KBL	$7,960$	180	0.795 ± 0.001	0.950 ± 0.001	0.938 ± 0.001
CZN	$7,680$	480	0.805 ± 0.001	n/a	0.889 ± 0.001

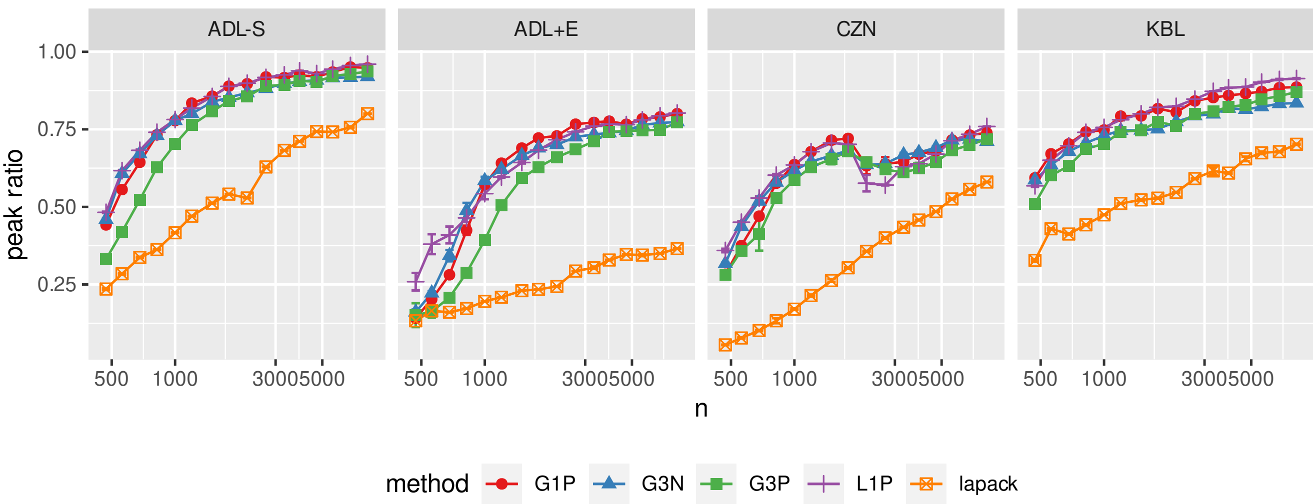

図 2 SPD 行列逆行列計算の実行性能比
Fig. 2 The peak-ratio of the SPD matrix inversion

表 3　いくつかの行列サイズにおけるピーク性能比
Table 3　Selected numbers from the above figure

machine	n	G1P	G3N	G3P	L1P	lapack
ADL-S	1000	0.778 ± 0.004	0.778 ± 0.002	0.703 ± 0.004	0.782 ± 0.002	0.417 ± 0.001
ADL-S	3300	0.917 ± 0.000	0.898 ± 0.001	0.893 ± 0.001	0.926 ± 0.000	0.682 ± 0.001
ADL-S	8200	0.948 ± 0.001	0.920 ± 0.000	0.936 ± 0.001	0.960 ± 0.000	0.801 ± 0.001
ADL+E	1000	0.573 ± 0.013	0.584 ± 0.013	0.393 ± 0.007	0.543 ± 0.018	0.196 ± 0.001
ADL+E	3300	0.772 ± 0.001	0.734 ± 0.003	0.711 ± 0.007	0.761 ± 0.001	0.304 ± 0.004
ADL+E	8200	0.801 ± 0.001	0.774 ± 0.001	0.772 ± 0.004	0.803 ± 0.000	0.366 ± 0.001
CZN	1000	0.636 ± 0.003	0.617 ± 0.002	0.588 ± 0.003	0.635 ± 0.002	0.170 ± 0.003
CZN	3300	0.646 ± 0.002	0.668 ± 0.006	0.611 ± 0.005	0.630 ± 0.001	0.435 ± 0.002
CZN	8200	0.738 ± 0.001	0.711 ± 0.001	0.719 ± 0.003	0.759 ± 0.001	0.580 ± 0.001
KBL	1000	0.748 ± 0.004	0.730 ± 0.002	0.703 ± 0.003	0.756 ± 0.001	0.474 ± 0.001
KBL	3300	0.852 ± 0.003	0.799 ± 0.002	0.807 ± 0.001	0.873 ± 0.000	0.616 ± 0.012
KBL	8200	0.886 ± 0.002	0.835 ± 0.001	0.870 ± 0.003	0.914 ± 0.002	0.702 ± 0.002

法とパック化の組み合わせである G3P は single sweep 法とパック化の組み合わせである G1P と比べてほぼ全ての行列サイズにおいて有意に性能が低いことが分かる. G1P はパック化を行わない G3N よりも行列サイズが大きいところで高速となっているが, 環境によって傾向が異なる. KBL では $n = 640$ 以上のところから, また ADL-S と ADL+E では $n = 1,000$ より大きいところから性能差が生まれているが, CZN では $n = 1,000$ から $n = 5,000$ の間で性能の曲線が交差している. 一方 L1P は行列サイズが最大付近や小さいところで高い性能となっており, とくに最大の行列サイズ $n = 8,200$ においては全環境で他の手法よりも高速となっているが, $n = 1,000$ から 3,000 程度の中程度の行列サイズにおいては場合によっては他手法に大きく劣る性能となっている.

図 2 から 3 つの行列サイズにおける性能数値を抜き出したものが表 3 である. 行列サイズが大きいところでは G1P と G3N との差は約 2% から約 5% となっており, パック化の効果を確認することができる.

5. まとめ

本稿では SPD 行列の逆行列計算を例題として, タイル型行列計算における個々のタスク性能低下の問題に対処するため行列積の内部処理であるパック化を直接扱い, パック化した結果を共有することによるオーバーヘッド削減効果について検証した. このとき, パック化した結果を共有することで本来無かった新たなタスク間の依存関係が生じタスクスケジューリングに制約が加わってしまうことが判明した. ただしこの新たな依存関係は疑似依存であり高度なリソース管理・タスクスケジューリング技術を用いれば解決可能だと考えられる. 我々はこの問題の SPD 行列の逆行列計算向けの簡易的な対処法として, 手動での事前のタスクスケジューリングが効果的であることを示した.

4 つの異なる環境を用いた実験結果では, SPD 行列の逆行列計算へ単にパック化を組み合わせると性能を大きく劣化させてしまうことが確認できたが, 我々の簡易的な対処法を適用することで改善でき, 数%程度であるが非パック化実装を上回る性能を実現できることが示された.

この結果はタイル型・タスク型行列計算における陽的なパック化の組み合わせの可能性を示すものであり, 今後は他の行列計算アルゴリズムへの展開を検討したい. また, パック化向けのリソース管理を同時に行う高度なタスクスケジューリング手法の開発は今後の大きな課題である.

謝辞 本研究は JSPS 科研費 19H04127 の助成を受けたものです。

参考文献

[1] Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Wu, P., Yamazaki, I., Yarkhan, A., Abalenkovs, M., Bagherpour, N., Hammarling, S., Šístek, J., Stevens, D., Zounon, M. and Relton, S. D.: PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP, *ACM Trans. Math. Softw.*, Vol. 45, No. 2 (online), DOI: 10.1145/3264491 (2019).

[2] Gates, M., Kurzak, J., Charara, A., YarKhan, A. and Dongarra, J.: SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3295500.3356223 (2019).

[3] Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Shet, A. G., Chrysos, G. and Dubey, P.: Design and Implementation of the Linpack Benchmark for Single and Multinode Systems Based on Intel® Xeon Phi Coprocessor, *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 126–137 (online), DOI: 10.1109/IPDPS.2013.113 (2013).

[4] Bientinesi, P., Gunter, B. and Geijn, R. A. v. d.: Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix, *ACM Trans. Math. Softw.*, Vol. 35, No. 1 (online), DOI: 10.1145/1377603.1377606 (2008).

[5] OpenMP Architecture Review Board: OpenMP Application Programming Interface, (online), available from ⟨https://www.openmp.org/wp-content/uploads/OpenMP-API-specification-5.2.pdf⟩ (accessed 2022-06-23).

[6] Intel Corporation: Developer Reference for Intel® oneAPI Math Kernel Library - C, (online), available from ⟨https://www.intel.com/content/www/us/en/develop/documentatio developer-reference-c/top.html⟩ (accessed 2022-06-23).

[7] Xianyi, Zhang: OpenBLAS An optimized BLAS library, (online), available from ⟨http://www.openblas.net⟩ (accessed 2022-06-23).

[8] Van Zee, F. G. and van de Geijn, R. A.: BLIS: A Framework for Rapidly Instantiating BLAS Functionality, *ACM Transactions on Mathematical Software*, Vol. 41, No. 3, pp. 14:1–14:33 (online), available from ⟨https://doi.acm.org/10.1145/2764454⟩ (2015).