Vol.2021-HPC-178 No.2
2021/3/15

RBT前処理付き不定値密対称行列線形ソルバ

鈴木 智博[1,a)]　Zhongyu Shen[1]

概要：線形方程式の直接解法ではピボット選択が不可欠であるが，これにより発生する大量のデータ移動はソルバーの性能低下の要因となる．これに対して，解法の前進消去過程で対角要素が 0 とならない行列を生成する前処理がある．今回，名古屋大学不老スーパーコンピュータ上で，タイルアルゴリズムでこの前処理付きの不定値密対称行列の線形ソルバーを実装し，ソルバーの性能とこの前処理の効果を調査した結果を報告する．

A tiled solver for dense symmetric indefinite systems with RBT preconditioning

SUZUKI Tomohiro[1,a)]　Zhongyu Shen[1]

Abstract: Pivoting is essential for a direct solver of linear equations. However, the large amount of data movement caused by this process can degrade the solver's performance. There is a preconditioning method for this problem to generate a matrix whose diagonal elements do not become zero in the forward elimination process. In this report, we implement a tiled solver for dense indefinite symmetric systems with the randomized preconditioning on the Flow supercomputer of Nagoya University and report this solver's performance and the effects of this preconditioning.

1. はじめに

対称行列 A の二次形式 $x^\top Ax$ が正のとき，A を正定値対称行列，正負両方の値を取る得るとき**不定値対称行列**と呼ぶ．不定値対称行列の線形方程式は非圧縮性流体のシミュレーションやマクスウェル方程式の境界要素法による解法，様々な物性，物理分野の最適化問題などに現れる．通常，疎行列の線形方程式系のソルバーには反復解法が用いられ，密行列に対しては直接解法が用いられる．

線形方程式系の一般的な直接解法では，係数行列 A は三角行列の積に分解され，右辺ベクトル b の前進・後退代入により解が求められる．正定値対称行列 A は下三角行列 L により Cholesky 分解 $A = LL^\top$ が可能である．A が不定値対称行列の場合，単位下三角行列 L と対角行列 D により $A = LDL^\top$ の分解が可能である．これを修正 Cholesky 分解，または **LDLT 分解**と呼ぶ．正定値対称行列では，上記の分解過程で対角要素が正であることが保証されるため，対角要素の平方根を計算する Cholesky アルゴリズムで行列分解可能である．不定値対称行列では，対角要素が正であることが保証されないので平方根の計算を回避するアルゴリズムとなっている．

線形方程式系ソルバーの行列分解における要素の更新過程では，対角要素，またはその平方根による除算が発生する．係数行列が正則な場合でも対角要素が 0 となることがあるので，これを避けるためにピボット選択が行われる．対角要素を含む行または列と他の行または列の入れ替えを行うことでピボット選択が実行されるが，これには大量のデータ移動が伴う．近年，高速なメモリが開発されているが，依然として演算を伴わないデータ移動は避けるべき操作である．Parker は行列分解の過程で対角要素が 0 とならないような乱数行列による前処理を提案した [1].

行列分解は線形方程式系のソルバー以外に幾つかのアルゴリズムの前処理，後処理に用いられる．固有値分解，特異値分解も含めた行列分解は，行列サイズを n として $\mathcal{O}(n^3)$ の計算量を必要とする重い処理である．効率的な行列分解のための様々な工夫が古くから行われている．行列

1　山梨大学大学院総合研究部工学域
a)　stomo@yamanashi.ac.jp

ⓒ 2021 Information Processing Society of Japan

分解のブロックアルゴリズムは，階層化されたメモリ構造を持つアーキテクチャにおいて効率的に実装された行列行列積を使用することを目的として開発された．一方，近年主流であるマルチコア・メニイコアアーキテクチャの高並列な計算資源を効率的に利用するために，大量の細粒度のタスクを非同期に実行するタスク並列型プログラミングモデルを適用することが試みられている．**タイルアルゴリズム**は，このような目的のために開発された行列分解のアルゴリズムである．

本報告では，Parker の前処理を実装した LDLT 分解をタイルアルゴリズムで実装し，最新のマルチコアアーキテクチャ上で性能評価を行う．

2. 関連研究

数値線形計算ライブラリのデファクトスタンダードとして広く利用されている LAPACK の参照実装は netlib で公開されている．その行列分解ルーチンではブロックアルゴリズムが採用されている．前述のとおり，行列分解のブロックアルゴリズムは，階層化されたメモリ構造を持つアーキテクチャ向きの手法として開発された．メモリ参照の局所性を高めて，キャッシュメモリを効果的に利用することが可能である．ブロックアルゴリズムにより fork-join 型の並列プログラミングモデルに基づく並列化が可能となり，並列化された (L3)BLAS ルーチンの恩恵により高速な実装が提供されている．

線形方程式の直接解法のための行列分解に，LU 分解，Chokesky 分解，LDLT 分解がある．一般の行列の行列分解にピボット選択は必須であり，LAPACK の LU 分解ルーチン `GETRF` では，対角要素を含む列の中から絶対値最大の要素を探す部分ピボット選択が採用されている．

正定値対称行列では Cholesky アルゴリズムの過程で対角要素が 0 にならないことが保証されており，LAPACK ルーチン `POTRF` はピボット選択を行わない実装となっている．ただし，ピボット選択を行う `PSTRF` も用意されており，より高精度な解が期待できる．

不定値対称行列は LDLT 分解可能であるが，ピボット選択を行った場合，行列 D は対角行列でなくブロック対角行列となる．ピボット選択付きの LDLT 分解としては Bunch-Kaufman の対角ピボッティング [2] がよく知られており，これが LAPACK ルーチン `SYTRF` で採用されている．

近年，マルチコア CPU のコア数の増加に伴い，すべての計算資源を無駄なく利用することが求められている．キャッシュを有効に利用することを目的とした従来の実装では高並列な計算資源を有効に活用できず，マルチコア CPU の性能を十分に発揮できていない．マルチコア CPU のすべての CPU コアを常に稼働状態にするために，大量に生成した細粒度のタスクを非同期に実行するタスク並列型プログラミングモデルが注目されている．

タイルアルゴリズムは高並列な計算環境向きの行列分解アルゴリズムとして開発された [3]．タイルアルゴリズムは，タイルサイズを適切に設定することで並列計算資源に見合う数のタスクを生成することが可能であり，これらを非同期に実行することですべての計算資源を常に稼働状態にすることを目的としている．

共有メモリ環境向けの PLASMA プロジェクトは LAPACK ルーチンをタイルアルゴリズムの実装に置き換えることを目的としている．初期の OpenMP はデータ並列プログラミングモデル向きのディレクティブ言語であったが，現在はタスク並列プログラミングモデル向きの機能を強化している．最新の PLASMA ライブラリのいくつかのルーチンは，OpenMP の task 構文を用いたタスク並列プログラミングモデルで実装されている．タスク並列プログラミングモデルは高並列な計算環境向きの新しい並列化手法であるとの認識が高性能計算の研究者の中で広がりつつある．

Babouline は [4] で，タイル LDLT 分解に Parker の乱数前処理を適用する手法を提案した．その後，[5] で複数のテスト行列で乱数前処理の有効性を示し，[6] では分散メモリ環境向けの実装を評価している．

この報告は [4] の提案手法に対して，OpenMP の task 構文によるタスク並列化を行った実装について名古屋大学不老 type I スーパーコンピュータ上で性能評価を行ったものである．

3. ピボット選択の必要性

以降，扱う行列は $n \times n$ 行列 A とし，その i 行 j 列要素を $a_{i,j}\,(0 \le i, j < n)$ と表す．

線形方程式の直接解法のための行列分解では，要素の更新過程で対角要素，またはその平方根による除算が発生する．このとき，対角要素が 0 ならば除算が行えず，解法は破綻する．これを避けるために，行入れ替え等で対角要素を非零にする操作を**ピボット選択**と呼ぶ．

LU 分解の前進消去過程の第 k ステップにおける更新操作を以下に示す．

$$a_{i,j} = a_{i,j} - (a_{i,k}/a_{k,k}) \cdot a_{k,j}\ (0 \le k < i, j < n) \quad (1)$$

上式において，$a_{k,k}$ の絶対値が小さい場合，それは前のステップの更新で発生した桁落ち計算の結果かもしれない．そのとき $|a_{i,k}/a_{k,k}| > 1$ となる可能性もまた高いので，$a_{k,k}$ の誤差は拡大されて $a_{i,j}$ に伝搬する．$a_{k,j}$ の誤差もまた拡大されて $a_{i,j}$ に伝搬する [7]．

部分ピボット選択では，第 k 列の中から，

$$|a_{p,k}| = \max_{k \le i < n} |a_{i,k}| \quad (2)$$

となる p を探し，第 k 行の要素 $a_{k,j}\,(k \le j < n)$ と第

p 行の要素 $a_{p,j}\,(k \le j < n)$ の交換を行う．これにより，$|a_{i,k}/a_{k,k}| \le 1$ が保証され，先に述べた誤差の伝搬を少なくすることができる．

ピボット選択の目的をまとめると以下となる．

(1) ゼロによる除算の回避

(2) 計算誤差の低減

4. RBT 前処理

ピボット選択の必要性は上に述べた通りだが，これによって発生する大量のデータ移動は行列分解アルゴリズムの速度を低下させる要因となる．

Parker は乱数行列による前処理により，行列分解の第 k ステップで対角要素 $a_{k,k}$ がゼロとならない手法を提案した [1]．この前処理は Random Butterfly Transformation (**RBT**) と呼ばれる．[5] では RBT 前処理の修正版である Partial Random Butterfly Transformation (**PRBT**) が提案された．以下では PRBT の概略を示す．

$n \times n$ **Butterfly 行列**は以下で定義される乱数行列である [4].

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix} \tag{3}$$

ここで，$n \ge 2$ であり，R，S は $(n/2) \times (n/2)$ の乱数対角行列である．

大きさ $n \times n$，深さ d の**再帰 Butterfly 行列**は以下で定義される．

$$U = U_d \times \cdots \times U_1 \tag{4}$$

ここで，$U_k\,(1 \le k \le d)$ は以下で表されるブロック対角行列である．

$$U_k = \begin{pmatrix} B_1 & & \\ & \ddots & \\ & & B_{2^{k-1}} \end{pmatrix} \tag{5}$$

ここで，B_i はサイズ $n/2^{k-1}$ の Butterfly 行列であり，U_1 は (3) で定義された B である．

正方行列 A に対する深さ d の PRBT は以下で定義される [5].

$$A_r = U^\top A V \tag{6}$$

ここで，U，V は深さ d の再帰 Butterfly 行列である．

定義 (4) から，再帰 Butterfly 行列のサイズは 2^d の倍数に限定される．任意のサイズの行列に対する PRBT は，もとの行列 A に対角要素に 1 を追加して行列サイズを膨らませることで対応する．

PRBT 前処理付きの線形方程式 $Ax = b$ の解法の手順は以下となる [5].

step 1　2 つの再帰 Butterfly 行列 U, V から, $A_r = U^\top A V$ とする

step 2　A_r にピボット選択なしの行列分解を適用する

step 3　$A_r y = U^\top b$ を y について解く

step 4　$x = Vy$ により解 x を得る

A が対称行列の場合は $V = U$ とし，$A_r = U^\top A U$ とする．これを Symmetric Random Butterfly Transformation (**SRBT**) と呼ぶ.

(3) の R，S が乱数対角行列であることから，Butterfly 行列 B_i の要素はサイズ n のベクトルとして保存可能である．したがって，(4) の U の要素は $d \times n$ の行列として保存可能である．

PRBT の計算量は，再帰 Butterfly 行列の構造から，$O(n^3)$ でなく $O(n^2)$ となるため [5]，前処理のオーバーヘッドは十分小さいと予想される．

線形方程式を解く上で，前処理によって A_r の条件数がもとの行列 A の条件数よりも増加してしまうことは避けなければならない．反復解法においては条件数の増加による反復回数が増加することを避けるために，条件数が小さくなるような前処理が適用される．r を $[-1/2, 1/2]$ の乱数として，$\exp(r/10)$ を再帰 Butterfly 行列 U を生成するブロック対角行列の対角要素とすることで，U の条件数は 1.1^d 程度に見積もることができる [5]. $\mathrm{cond}_2(A_r) \le \mathrm{cond}_2(U)\mathrm{cond}_2(A)\mathrm{cond}_2(V)$ より，PRBT 前処理による条件数の増加は 1.1^{2d} 程度であることが分かる．結論として，乱数行列の要素を適切に選ぶことにより，PRBT 前処理による数値解の精度低下を抑えることができる．

PRBT 前処理により，計算過程で A_r の対角要素がゼロにならず，その条件数は A に比べてさほど大きくならないとしても，ピボット選択の 2 つめの目的である計算誤差の低減は行われない．これについては，得られた数値解に対して**反復改良** [8] を実施することで精度を向上することが可能である．

5. タイル LDLT 分解

行列分解のブロックアルゴリズムは「パネル分解」と「後続行列更新」の二つの処理を繰り返し実行する．パネル分解は L2 BLAS 演算主体であるのに対し，後続行列更新は L3 BLAS 演算を多用するため，並列化された BLAS 実装を使用することで逐次アルゴリズムよりも高速となる．並列化された L3 BLAS 実装により高速化される行列分解は典型的な fork-join 並列計算モデルである．fork-join モデルに代表される Bulk Synchronous Parallel (BSP) モデルは並列化が容易であるが，すべての並列計算資源を常に稼働状態にすることは原理的に不可能である．

タイルアルゴリズムは対象とする行列を行方向, 列方向に

分割し，分割後の小行列（タイル）について分解，更新の操作を行う．小行列に分割すること自体は，例えば Cholesky 分解のブロックアルゴリズムと同じだが，行列データの格納方式として一般的な列優先方式ではなく，小行列内でデータが連続配置される Block Data Layout (**BDL**) を採用すること，分解，更新タスクを動的にスケジューリングすることが特徴として挙げられる．タイルアルゴリズムは，タイルサイズを適切に設定することで並列計算資源に見合う数のタスクを生成することが可能であり，これらを非同期に実行することですべての計算資源を常に稼働状態にすることを目的としている．タイルサイズ（ブロックサイズ）が小さいほどタスクを大量に生成できるため，並列計算資源の負荷不均衡が発生しにくいが，L3 BLAS 性能は低下する．このトレードオフがあるため，タイルアルゴリズムにおいてタイルサイズチューニングは重要な課題となっている [9].

行列分解のタイルアルゴリズムのタスクはタスク間のデータ依存を観察し，依存関係が解決されたタスクから順に実行される．逐次版のタイルアルゴリズムに，OpenMP 4.0 以降の task 構文と depend 節を追記することで，共有メモリ環境において容易にタスク並列プログラムを記述することが可能となっている．タスク並列版タイル LDLT 分解の擬似コードを Algorithm 1 に示す.

Algorithm 1 Tiled LDLT factorization

```
#pragma omp parallel {
  #pragma omp master {
  for (k=0; k<p; k++) {
     #pragma omp task \\
     depend(inout:A(k,k)) depend(out:D(k))
     SYTRF2( A(k,k), D(k) );

    for (i=k+1; i<p; i++) {
       #pragma omp task \\
       depend(in:A(k,k), D(k)) depend(inout:A(i,k))
       TRSM( A(k,k), D(k), A(i,k) );
    }
    for (i=k+1; i<p; i++) {
      #pragma omp task \\
         depend(in:A(i,k), D(k)) depend(inout:A(i,i))
       SYDRK( A(i,k), D(k), A(i,i) );

      for (j=k+1; j<i; j++)
       #pragma omp task \\
          depend(in:A(i,k),A(j,k), D(k)) depend(inout:
               A(i,j))
         GEMDM( A(i,k), A(j,k), D(k), A(i,j) );
    }
  }
} }
```

Algorithm 1 において，対称行列 A のサイズを $n \times n$，タイルサイズを $b \times b$ とする．タイル行数，列数を $p = \lceil n/b \rceil$ とする．タイル $A_{i,j}$ = `A(i,j)` と表記している．また，`D(k)` は対角行列 $D_{k,k}$ の対角要素のベクトルとする．各計算カーネルの機能を以下に列挙する．

- `SYTRF2`: ピボット選択なし LDLT 分解（ $A_{k,k} = L_{k,k}\, D_{k,k}\, L_{k,k}^{\top}$ ）．ただし，$L_{k,k}$ は $A_{k,k}$ に上書きされる．
- `TRSM`: 三角行列解法（ $L_{ik}(L_{k,k}D_{k,k})^{\top} = A_{i,k}$ を $L_{i,k}$ について解く）．ただし，$L_{i,k}$ は $A_{i,k}$ に上書きされる．
- `SYDRK`: $A_{i,i}$ の更新（$A_{i,i} = A_{i,i} - L_{i,k}\, D_{k,k}\, L_{i,k}^{\top}$）．
- `GEMDM`: $A_{i,j}$ の更新（$A_{i,j} = A_{i,j} - L_{i,k}\, D_{k,k}\, L_{j,k}^{\top}$）．

Algorithm 1 において，task 構文で指定されたブロックがタスクとして実行される．データ依存がある変数を depend 節で指定し，`in`，`out`，`inout` の依存型を指定する．依存型はそれぞれ指定された変数が参照データ，出力データ，更新データであることを表す．

表 1 実験環境
Table 1 Experimental environment

CPU	A64FX (Armv8.2-A + SVE), 48 コア + 2 アシスタントコア 2.2GHz
メモリ	HBM2 32GiB
(理論演算性能	倍精度 3.38 TFLOPS)
C++コンパイラ	FCCpx
コンパイオプション	-Kfast -Kopenmp -Nclang -Nlibomp
ライブラリ	スレッド並列版 BLAS/LAPACK (-SSL2BLAMP)

6. 実験

名古屋大学情報基盤センターの不老 Type I 1 ノードを使い SRBT 前処理付きのタイル LDLT 分解の評価実験を行った．実験環境を表 1 に示す.

図 1 に不老 type I 1 ノードにおけるタイル LDLT 分解のプログラムトレースを示す．行列サイズ $n = 16384$，タイルサイズ $b = 512$，$p = n/b = 32$ より，各カーネルの呼び出し回数は `SYTRF2` が 32，`TRSM` と `SYDRK` が 496，`GEMDM` が 5456 となっている.

同図より，実行の初期と終期には並列実行可能なタスクが少なく，休止状態となっているスレッドが見られる．それ以外の区間ではすべてのスレッドが休止状態となることなく稼働しており，タイルアルゴリズムが目的通りに動作していることが分かる.

タイルサイズを小さくすることで，図 1 の初期と終期に見られるスレッド休止状態を少なくすることが可能だが，これは L3 BLAS 演算性能の低下をもたらすことは既に述べた．図 2 にタイル LDLT 分解のタイルサイズによる実行時間の変化を示す．この図において，実行時間は LDLT 分解と線形方程式の求解を合わせた時間としている．この実験の範囲の行列サイズではタイルサイズによって大きく性能が変化することはなく，$b = 384$ を選べばよいことが分かる.

図 3 に，SRBT 前処理付きタイル LDLT 分解，タイル LDLT 分解と LAPACK の `DSYTRF` ルーチンとの性能比較を示す．タイルサイズは図 2 に示された 4 種類のうち，最も高速だったものがプロットされている．この図において，実行時間は LDLT 分解と線形方程式の求解を合わせた

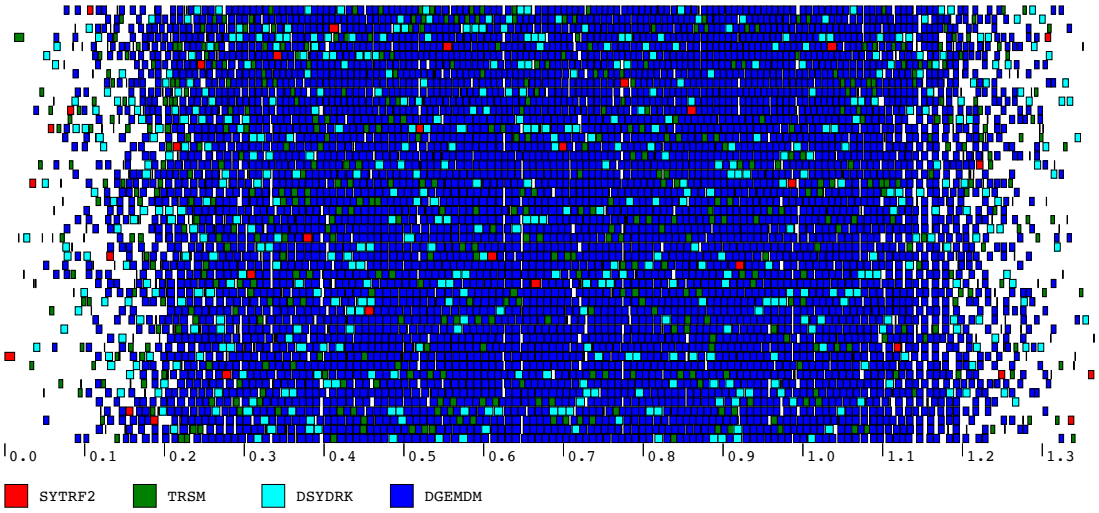

図 1 48 スレッド実行時のタイル LDLT 分解のプログラムトレース
Fig. 1 Program trace of tiled LDLT factorization (48 threads execution)

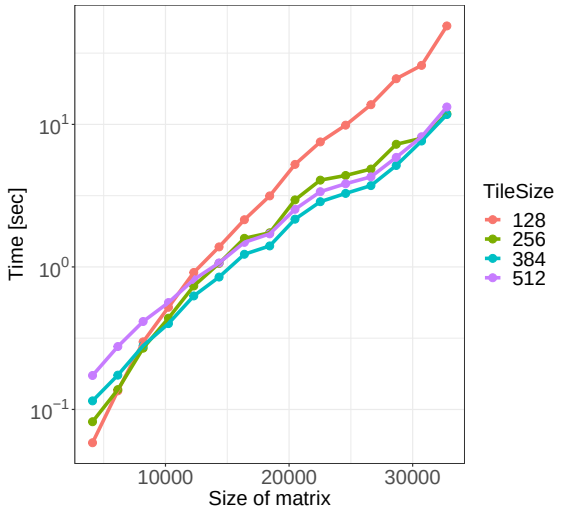

図 2 タイルサイズ毎の性能の違い
Fig. 2 Performance change of tiled LDLT with different tile sizes

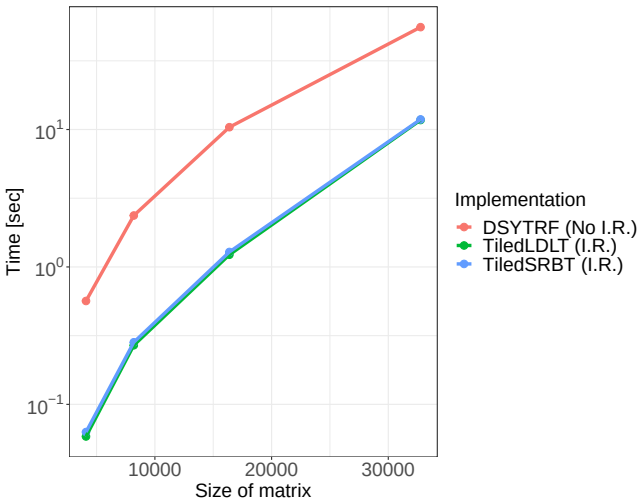

図 3 SRBT 前処理あり/なしのタイル LDLT と DSYTRF の性能
Fig. 3 Performance of tiled LDLT with/without SRBT and DSYTRF

時間としている．この後に示す通り，ピボット選択なしでは数値解の精度が悪いことから，ピボット選択なしの実装では反復改良を 1 回実行し，その時間を含めた実行時間がプロットされている．さらに，タイルアルゴリズムの実装では行列データの列優先方式から BDL への変換，逆変換の時間が含まれている．この結果より，SRBT 前処理の実行時間への影響は少なく，ピボット選択を行う LAPACK ルーチンよりも高速であることが分かる．

ここで，RBT 前処理とタイル行列分解の実装について触れる．RBT 前処理は Butterfly 行列の構造から少ない計算量で実施できることは既に述べた．しかし，BDL 変換後の行列に対してタイル毎に RBT 前処理を適用すると，離れた位置にある複数のタイルに対する処理を行う必要があり，タスク間の同期を取らないと依存関係の抽出が複雑になる．実際に PRBT 前処理を行った後に BDL 変換する場合と，BDL 変換後の行列に PRBT 前処理を行う場合を比較したが，前者の方が高速であった．図 3 の SRBT 前処理も同様にしている．

次に，数値解の精度に関する結果を示す．図 4 は各実装の数値解 $\hat{x}$ に対して，$||b - A\hat{x}||_2$ を行列サイズを変えてプロットしたものである．行列 A は要素が $[-10, 10]$ の一様乱数，右辺ベクトル b のすべての要素を 1 とした．この図において，`DSYTRF` は LAPACK ルーチン，`LDLT` は SRBT 前処理なしのタイル LDLT 分解，`SRBT` は SRBT 前処理ありのタイル LDLT 分解，のそれぞれの数値解の残差を意味し，(`I.R.`) と (`No I.R.`) は反復改良あり/なしを意味する．

図 4 より，以下が分かる．

ⓒ 2021 Information Processing Society of Japan

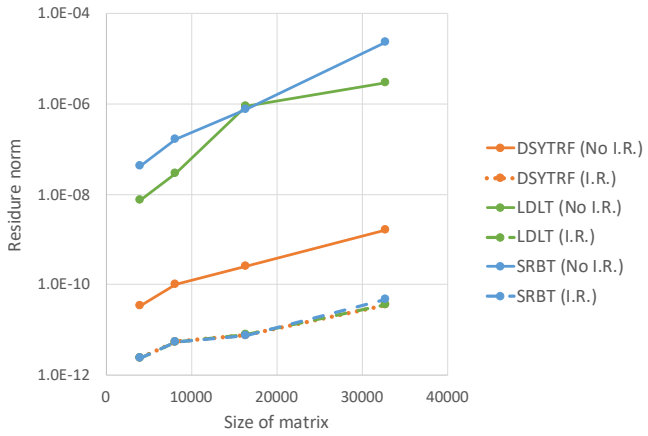

図 4 残差ノルム
Fig. 4 Residure norms

表 2 テスト行列セット
Table 2 Test matrix sets

Matrix class	$a_{i,j}$, b_i
[0,1]	$(0,1)$ の一様乱数
[-1,1]	$(-1,1)$ の一様乱数
{0,1}	$(0,1)$ の正規乱数
|i-j|	$a_{i,j} = \|i-j\|, b_i = 1$
max(i,j)	$a_{i,j} = \max(i,j), b_i = i$

- 反復改良なしを比べると，LAPACK ルーチンが高精度であり，SRBT 前処理あり，なしでは精度に大きな差は見られない．
- 反復改良を 1 回実施すれば，すべての実装の数値解の精度は同程度となる．

最後に，[1] の実験にある 11 種類のテスト行列から 5 つの組について我々の実装の精度を確認した．テスト対象の組を表 2 に示す．

テスト行列に対する各実装の残差を表 3 に示す．ここで，テスト行列のサイズは $n = 16,384$，タイルアルゴリズムでのタイルサイズを $b = 256$ とした．

この実験では，4, 5 番目のテスト行列に対して，SRBT 前処理なしの LDLT 分解が求解に失敗しているが，前処理により数値解が得られていることが分かる．

7. おわりに

線形方程式系の直接解法で必須とされるピボット選択は大量のデータ移動を伴うことから，ソルバーの性能低下の要因となる．係数行列に対して，行列分解の過程で対角要素が 0 とならないような RBT 前処理を適用するピボット選択なしの行列分解をタイルアルゴリズムで実装した．これについて不老 type I スーパーコンピュータ上で計算速度，数値解の精度を LAPACK ルーチンと比較した．その結果，一度の反復改良を加えることでピボット選択付きの LAPACK ルーチンと同等の精度が得られ，計算時間については数倍の高速化が得られることが分かった．また，ピボット選択なしでは分解できない行列が RBT 前処理により分解可能となる例を示した．

今後の課題として以下の 2 つを挙げる．

- 低ランク近似手法の適用
- 分散メモリ型並列環境向け実装

低ランク近似により行列データがよく圧縮されるような行列は，低ランク性を持つ，と言われる．低ランク性を持つ行列は，低ランク近似によりデータが圧縮されるだけでなく，それによる演算データ量の低下から高速化が期待できる．ブロック化された係数行列に対して低ランク近似を適用した線形ソルバの研究 [10] がある．この中で，小行列を圧縮するタイミングにより複数のアルゴリズムが提案されている．また，低ランク行列に近似するための手法についても複数の提案がある．低ランク性を持つ係数行列に対して，RBT 前処理付きタイル LDLT 分解にこれらを導入することでより高性能なソルバを開発できる可能性がある．

分散メモリ環境でのタイル LDLT 分解については既に [6] で評価されている．RBT 前処理を適用しピボット選択を行わない行列分解は，そのためのノード間通信が削減されることから分散メモリ環境で高い性能を発揮することが期待できる．一方，OpenMP の task 構文を使ったタスク並列版のタイルアルゴリズムの場合，データ依存の追跡手法が大きな問題となることが予想される．行列分解の通信回避アルゴリズム [11] に基づいて，データ依存追跡をノード内に閉じたものに出来れば分散メモリ型並列環境で有効なタイルアルゴリズムの実装が可能であると思われる．

謝辞 本研究は JSPS 科研費 19H04127 の助成を受けたものである．

参考文献

[1] Parker, D. S.: Random butterfly transformations with applications in computational linear algebra, Technical report, UCLA (1995).
[2] Bunch, J. R. and Parlett, B. N.: Direct methods for solving symmetric indefinite systems of linear equations, *SIAM Journal on Numerical Analysis*, Vol. 8, No. 4, pp. 639–655 (1971).
[3] Buttari, A., Langou, J., Kurzak, J. and Dongarra, J.: Parallel tiled QR factorization for multicore architectures, *Concurrency and Computation: Practice and Experience*, Vol. 20, No. 13, pp. 1573 – 1590 (2008).
[4] Baboulin, M., Becker, D. and Dongarra, J.: A parallel tiled solver for dense symmetric indefinite systems on multicore architectures, *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IEEE, pp. 14–24 (2012).
[5] Baboulin, M., Dongarra, J., Herrmann, J. and Tomov, S.: Accelerating linear system solutions using randomization techniques, *ACM Transactions on Mathematical Software (TOMS)*, Vol. 39, No. 2, pp. 1–13 (2013).
[6] Baboulin, M., Becker, D., Bosilca, G., Danalis, A. and Dongarra, J.: An efficient distributed randomized algo-

表 3 テスト行列に対する残差ノルム

Table 3 Residure norms of each implementation for test matrices

	No I.R.			I.R.		
Matrix class	DSYTRF	TiledLDLT	TiledSRBT	DSYTRF	TiledLDLT	TiledSRBT
`[0,1]`	2.51E-10	9.00E-07	5.75E-07	7.67E-12	7.76E-12	7.61E-12
`[-1,1]`	7.80E-10	3.42E-06	6.36E-07	1.64E-11	1.66E-11	1.63E-11
`{0,1}`	9.54E-10	1.44E-06	5.84E-06	2.06E-11	2.02E-11	2.05E-11
`|i-j|`	1.28E-12	—	2.52E-13	8.06E-13	—	1.97E-13
`max(i,j)`	5.12E-11	—	9.58E-10	3.48E-11	—	1.05E-10

rithm for solving large dense symmetric indefinite linear systems, *Parallel Computing*, Vol. 40, No. 7, pp. 213–223 (2014).

[7] 杉浦洋：数値計算の基礎と応用 [新訂版]，サイエンス社 (2009).

[8] Higham, N. J.: *Accuracy and stability of numerical algorithms*, SIAM (2002).

[9] Suzuki, T.: Faster method for tuning the tile size for tile matrix decomposition, *Proceedings of IEEE 10th International Symposium on Embedded Multicore/Manycore SoCs (MCSoC-16)*, IEEE, IEEE, pp. 329 – 336 (online), DOI: 10.1109/MCSoC.2016.24 (2016).

[10] Amestoy, P., Ashcraft, C., Boiteau, O., Buttari, A., L'Excellent, J.-Y. and Weisbecker, C.: Improving multifrontal methods by means of block low-rank representations, *SIAM Journal on Scientific Computing*, Vol. 37, No. 3, pp. A1451–A1474 (2015).

[11] Demmel, J., Grigori, L., Hoemmen, M. F. and Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations, *SIAM Journal on Scientific Computing*, Vol. 34, No. 1, pp. A206 – A239 (2012).

ⓒ 2021 Information Processing Society of Japan