Vol.2020-HPC-173 No.10
2020/3/17

ランタイムシステムを用いたマルチフロンタルコレスキー分解の開発

中野 智輝[1,a)]　横川 三津夫[1]　山本 有作[2]　深谷 猛[3]

概要：近年の計算機のアーキテクチャは多様化・複雑化している．単一の CPU で構成されているものだけでなく，それぞれのコアで計算性能が異なる SPU，1 つあたりのコアの計算性能は低いが大量にコアを持つ GPU など計算機の多様化が進んでいる．これら様々な計算機に対して，高い演算性能を得るには，それぞれのマシンで最適化・チューニングが必要となる．しかし，新しい種類の計算機が出るたびにこれらを行うのは非常に手間である．これを解決するために，近年ランタイムシステムの利用が注目されている．一般に，数値シミュレーションにおいては，偏微分方程式を離散化して大規模な連立一次方程式を解く問題に帰着させることが多く，またシミュレーションの大部分をその時間が占める．そのため，連立一次方程式を高速に解くことは非常に重要である．本稿では，疎で正定値対称な行列を係数行列とする連立一次方程式に対し，疎行列直接解法でよく使われるマルチフロンタル法に注目し，アルゴリズムを少し修正することによって，ランタイムシステムへの適用を行った．数値実験では，StarPU というランタイムシステムを用いて，メニーコアプロセッサ上での並列化を行い，高い並列化効率を得ることができた．

キーワード：コレスキー分解，スパースソルバ，Xeon Phi，マルチフロンタル法，スーパーノード，ランタイムシステム，StarPU

1. はじめに

数値シミュレーションにおいては，偏微分方程式を離散化して大規模な連立一次方程式を解く問題に帰着させることが多い．そして，多くの場合，連立一次方程式を解く時間は全体のシミュレーション時間の大部分を占める．よって，連立一次方程式を高速に解くことは非常に重要である．連立一次方程式の解法はガウスの消去法，LU 分解法，コレスキー分解法などの直接解法とヤコビ法，ガウスーザイデル法，共役勾配法，GMRES 法などの反復解法の 2 種類に分類される．それぞれ，安定性，計算量，並列計算の観点でメリット・デメリットが存在する．本稿では直接解法を扱う．

密行列の直接解法のライブラリは LAPACK[1] が有名である．疎行列の直接解法のライブラリは非常にたくさんあるが，中でもスーパーノード法を用いる PARDISO[2] や SuperLU[3]，マルチフロンタル法を用いる MUMPS[4] などが有名である．その他のライブラリについては，Davis らの文献 [5] が詳しい．

[1] 神戸大学
[2] 電気通信大学
[3] 北海道大学
[a)] tomoki_nakano@stu.kobe-u.ac.jp

一方，近年の計算機のアーキテクチャは多様化・複雑化している．単一の CPU で構成されているものだけでなく，それぞれのコアで計算性能が異なる Cell/B.E. などの SPU(Synergistic Processing Unit)，1 つあたりのコアの計算性能は小さいが大量にコアを持つ GPU など計算機の多様化が進んでいる．また，メモリアーキテクチャに関しても，それぞれのコアから直接アクセスできる共有メモリ，通信を用いてアクセスできる分散メモリ，異なるアクセス速度をもつ NUMA など様々になっている．これら様々な計算機に対して，高い演算性能を得るには，それぞれのマシンで最適化・チューニングが必要となる．しかし，新しい種類の計算機が出るたびにこれらを行うのは非常に手間である．これを解決するために，近年ランタイムシステムの利用が注目されている．本稿で扱うランタイムシステムとは，アプリケーションとアーキテクチャの間に位置するソフトウェアシステムであり，タスクの依存関係の管理とタスクのスケジューリングを処理するものとする．ランタイムシステムを利用した実行モデルでは，アプリケーションはハードウェアの詳細に関係なく，高レベルの API を使用して表現されるため，異なるアーキテクチャ間で高い演算性能を得ることが可能となる．

ランタイムシステムは，非常に多くの種類の言語仕様

やライブラリとして実装されている [6]．また，これらのランタイムシステムの開発に伴い，ランタイムシステムを用いた密行列線形ライブラリ（Dense Linear Algebra, DLA）が開発されてきた．PLASMA[7]，DPLASMA[8]，Chameleon[9] などがその例である．さらに，近年では，ランタイムシステムを利用した疎行列直接解法のライブラリとして，SpLLT[10], [11] や SyLVER[12]，PaStiX[13], [14] などが開発されている．近年の疎行列直接解法はマルチフロンタル法，スーパーノード法のいずれかが用いられている．これら 2 つの手法はそれぞれメリット・デメリットが存在する．現在のランタイムシステムによる疎行列直接解法のライブラリはすべてスーパーノード法を用いている．そこで，本稿では，ランタイムシステムを用いたマルチフロンタル法の実装および性能評価を行う．

2. スパースソルバ

2.1 概要

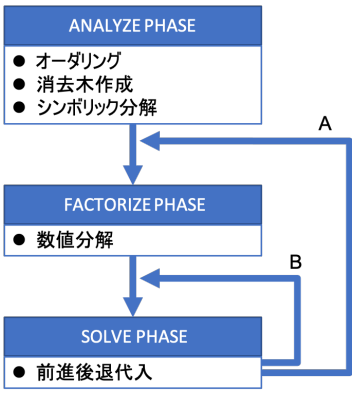
本節では，正定値対称行列に対するスパースソルバの基本的なアルゴリズムを述べる．スパースソルバは図 **1** のように 3 つのフェーズに分けることができる．係数行列の疎構造が決まると，ANALYZE フェーズを実行でき，係数行列の数値が決まると，FACTORIZE フェーズが実行できる．さらに，右辺ベクトルが決まると SOLVE フェーズが実行できる．つまり，係数行列の疎構造が不変で値のみが変わる場合，ANALYZE フェーズは 1 度のみ実行すればよい．同様に，係数行列が不変で右辺ベクトルのみが変わる場合，ANALYZE フェーズと FACTORIZE フェーズは 1 度のみ実行すればよい．一般に 3 つの手順の中で最も計算コストがかかるのは，FACTORIZE フェーズであるため，係数行列が不変の問題に対しては，上記に述べたことを利用することでかなり計算時間を節約できる．

一般に疎行列のコレスキー分解では，疎行列の疎性を考慮する．つまり，フィルインも考慮した非零要素のみを記憶し演算する必要がある．そのため，ANALYZE フェーズでは，分解後の非零要素の位置を先立って計算する必要がある．これをシンボリック分解という．また，FACTORIZE フェーズでは，実際の数値をコレスキー分解する．これを数値分解という．シンボリック分解や数値分解を効率的に行うために，消去木というグラフを用いる．これはシンボリック分解より前に作成する必要がある．また，コレスキー分解前に行列の行と列を置換する適切なオーダリングにより，フィルインを減らすようにし，並列計算が可能となる．以下の各節では，それぞれ処理についてより詳しく述べる．

2.2 オーダリング

A を $n \times n$ の正定値対称行列とし，解くべき方程式を $A\boldsymbol{x} = \boldsymbol{b}$ とする．オーダリングとは，適当な置換行列 P を用いて，A の行と列の対称置換 $\tilde{A} = PAP^{\mathrm{T}}$ を行い，与えられた方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を解く代わりに $\tilde{A}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$ を解く方法である．ここで，$\tilde{\boldsymbol{x}} = P\boldsymbol{x}$，$\tilde{\boldsymbol{b}} = P\boldsymbol{b}$ である．普通の置換ではなく，対称置換を行うのは，A の対称性を保つためである．正定値対称の場合，どのように P を選んでも $\tilde{A}$ は安定にコレスキー分解できる．そのため，一般的に P は $\tilde{A}$ がフィルインが少なくなるように選ぶ．また，並列化を行う場合は，並列化効率がよくなるように選ぶ．オーダリングの手法としては，次の 3 つがよく使われる：

- minimum degree オーダリング [15], [16]：消去を行う各ステップで，非零要素数が最も少ない列を消去するオーダリング，
- RCM[17]：バンド幅を縮小するオーダリング，
- nested dissection オーダリング [18]：行列 A を隣接行列としたときのグラフ $G(A)$ を分割していくようなオーダリング．



A：係数行列の非零要素の値が変わる場合
B：右辺ベクトルが変わる場合

図 1: スパースソルバの流れ

2.3 消去木とシンボリック分解

密行列の場合，コレスキー分解の処理は第 1 列から第 n 列まで順番に行う必要がある．これに対し，疎行列の場合は，処理の順番に自由度が生まれる．その依存関係を表したグラフが消去木である．消去木は，n 個の節点を持つ木である．それぞれの節点 j は，節点 $i = \min\{k > j | L_{kj} \neq 0\}$ を親として持つ．消去木の例を図 **2** に示す．1 列目〜3 列目はそれぞれ子の節点がないので，いつでも計算可能であることや，6 列目の計算には 4 列目と 5 列目の計算が終わっていればよいということがわかる．消去木の計算は道圧縮の Liu のアルゴリズム [19] を用いて $\mathcal{O}(|A|)$[*1] で計算可能である．

シンボリック分解ではフィルインによる要素も含めた非零パターンのみに着目してコレスキー分解で現れる非零要素の位置を求める．$\mathcal{L}_j$ を行列 L における j 列の対角成分

[*1] $|A|$ は，行列 A の下三角部分の非零要素数である．

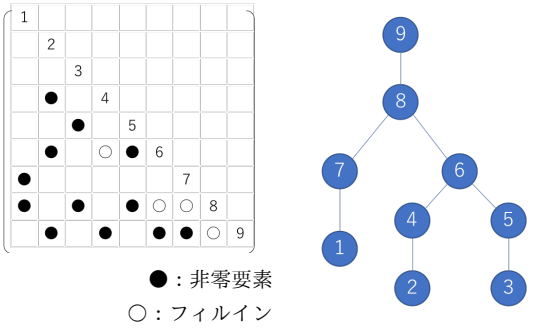


図 2: コレスキー分解後の行列 L と対応する消去木

を除く非零要素の行インデックス，$\mathcal{A}_j$ を行列 A における j 列の対角成分を除く非零要素の行インデックスとする．$\mathcal{C}_j$ を消去木におけるノード j の子の集合とする．このとき，

$$\mathcal{L}_j = \bigcup_{c \in \mathcal{C}_j} \mathcal{L}_c \cup \mathcal{A}_j$$

が成り立つ．これより，消去木を用いることによって，シンボリック分解を $\mathcal{O}(|L|)$[*2]で計算可能である．

また，シンボリック分解を行う前にそれぞれの列の非零要素数を計算する必要がある．この処理を列カウントという．列カウントでは，文献 [20] の方法を用いることによって，$\mathcal{O}(|A|)$ で計算可能である．また，この列カウントにより，数値分解の計算量を見積もることが可能である．数値分解の計算量は，

$$\sum_{j=1}^{n} |\mathcal{L}_j|^2$$

となる [21]．

2.4 数値分解

数値分解では，シンボリック分解で求めたデータ構造を利用して $\tilde{A} = LL^{\mathrm{T}}$ の非零部分についてのみ計算する．数値分解の主な計算方法として，本稿では，right-looking アルゴリズムの変形であるマルチフロンタル法を扱う．

2.5 前進後退代入

まず，前進代入で下三角行列を係数とする方程式 $L\boldsymbol{y} = \tilde{b}$ を解く．その後，後退代入で上三角行列を係数とする方程式 $L^{\mathrm{T}}\tilde{\boldsymbol{x}} = \boldsymbol{y}$ を解く．そして最後にベクトルの置換 $\boldsymbol{x} = P\tilde{\boldsymbol{x}}$ を行う．前進代入，後退代入の計算量はともに $\mathcal{O}(|L|)$ となる．これは数値分解と比べてコストが小さい．

3. マルチフロンタル法

マルチフロンタル法は，right-looking アルゴリズムの変形である．消去木のトポロジカルオーダリングの順に以下の処理を行う [22]．

(1) 行列 A の第 i 列を用いてフロンタル行列 F_i を作成する．子があれば，その子のアップデート行列を足し合わせる．

(2) 消去を 1 ステップのみ行う．

(3) フロンタル行列の第 1 列を行列 L の第 i 列として格納し，2 列目以降をアップデート行列 U_i として保存する．

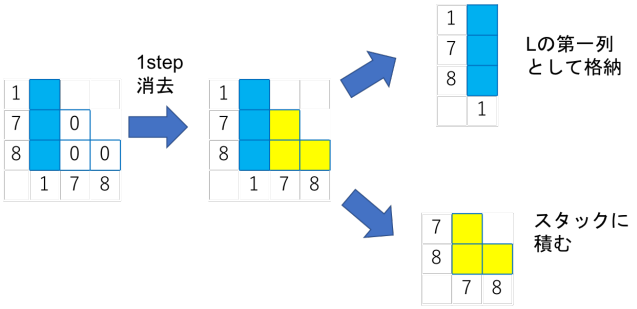


図 3: フロンタル行列と消去演算

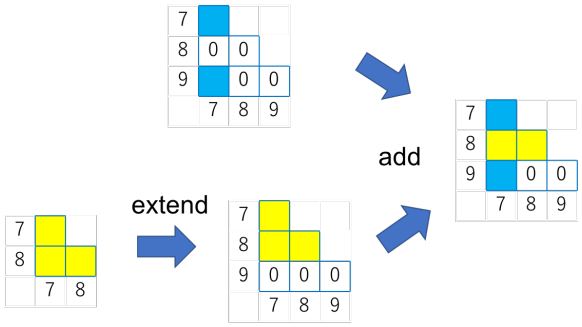
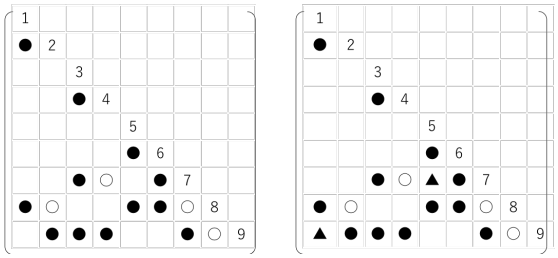


図 4: extend-add 演算

フロンタル行列とは，非零要素のみを圧縮した密行列のことである．図 2 の行列にマルチフロンタル法を適用すると，最初に消去が可能なのは，第 1, 2, 3 列である．まず，第 1 列は 1, 7, 8 行に非零要素を持つ．これらを行・列のインデックスとする空の行列を作成し，その第 1 列に行列 A の第 1 列を格納する．これが第 1 列のフロンタル行列 F_1 となる（**図 3**）．次に，F_1 で消去演算を 1 ステップのみ行い，第 1 列を行列 L に保存し，2 列目以降をアップデート行列 U_1 とする．第 1 列の処理が終わったあと，次に処理を行うことが可能な列は第 2, 3, 7 列となる．第 7 列に対しては，非零要素は 7, 8, 9 行に存在するため，同様に，それらをインデックスとする空の行列を作成し，第 1 列に A の第 7 列を格納する．アップデート行列 U_1 をこの行列に足しこむことによってフロンタル行列 F_7 を作成する．ただし，この処理はインデックスが対応するように行う必要がある．これを extend-add 演算と呼ぶ（**図 4**）．F_7 を作成したあと，F_1 の場合と同様に消去演算を 1 ステップのみ行い，第 1 列を L_7 として保存し，2 列目以降がアップデート行列 U_7 となる．このようにして，各列に対する処理を消去木のトポロジカルオーダリングの順で行うことにより，コレスキー分解が完了する．このように，マルチフロンタル法での主な処理は extend-add 演算と 1 ステップのみ行

[*2] $|L|$ は，行列 L の非零要素数である．



(a) 帰りがけ順　　(b) $max\_zero = 1$ としたときの行列

●：非零要素　○：フィルイン　▲：非零とみなした要素

図 5: スーパーノードの緩和

う消去演算で成り立っている．Extend-add 演算は，間接参照が必要なベクトル和を必要とするが，消去演算は密行列で行うため，Level-2 BLAS を用いることができるという利点を持つ．その反面，フロンタル行列やアップデート行列を保持するためのメモリ容量が必要である欠点を持つ．

3.1 ローカルインデックス

Extend-add 演算では整数配列によるローカルインデックスを用いてスパースのベクトル和を行う [23]．ローカルインデックスは消去木の親のフロンタル行列における行番号を保存したものである．図 2 の行列の場合，第 2 列のローカルインデックスは 1, 2，第 3 列のローカルインデックスは 1, 3 となる．

3.2 スーパーノード

スーパーノードとは，「L において上部三角領域が全て非零で，各列が同じ非零構造を持った列の集合」である．スーパーノードに含まれる列の数を次数という．**図 5(a)** は図 2 の行列を帰りがけ順にリオーダリングした行列である．3, 4 列目が次数 2 のスーパーノード，7〜9 列目が次数 3 のスーパーノードとなっている．

最後の列以外が消去木において子を 1 つしか持たない場合，そのスーパーノードを基本スーパーノードと呼ぶ．基本スーパーノードは文献 [24] の手法を用いることで $\mathcal{O}(|A|)$ で計算が可能である．

3.3 スーパーノードの緩和

零要素を非零要素とみなすことによっていくつかの基本スーパーノードを合併し，より大きなスーパーノードを得ることができる．これをスーパーノードの緩和という．これにより，行列 × 行列の演算を行なうことができ，高速化が期待できる．スーパーノードの緩和の手順は，基本スーパーノードの消去木を帰りがけ順で走査すればよい [25]．まず節点 J とそれぞれの子を合併したときに，合併したスーパーノードに含まれる零要素の数を計算する．もしその数が緩和パラメータ max_zero より小さくなる子がある

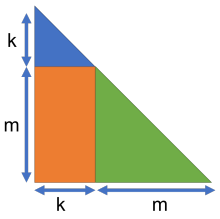


図 6: フロンタル行列

とき，合併したスーパーノードの零要素が最も小さくなる子と J を合併する．max_zero は各スーパーノードにいくつ非零要素とみなして良いかの数である．上記の手順を合併できる子節点がなくなるか，子節点がなくなるまで行う．**図 5(b)** に $max\_zero = 1$ で緩和した例を示す．この場合，1 と 2 列，3 と 4 列，5 と 6 列が次数 2 のスーパーノード，7〜9 列が次数 3 のスーパーノードとなる．$max\_zero = 0$ にすると通常のスーパーノードとなる．

3.4 スーパーノードを用いたマルチフロンタル法

マルチフロンタル法にスーパーノードを導入することで，より最適化が効く Level-3 BLAS を使用できる．スーパーノードを用いたマルチフロンタル法では，主に以下の処理を行う．

(1) コレスキー分解 (_POTRF)：$\mathcal{O}(k^3)$
(2) 下三角行列の行列方程式を解く (_TRSM)：$\mathcal{O}(k^2m)$
(3) 行列とその転置の積 (_SYRK)：$\mathcal{O}(km^2)$
(4) Extend-add 演算：$\mathcal{O}(m^2)$

それぞれの処理の演算量も示した．ここで，k はスーパーノードに含まれる列の数，m は最後の列の対角成分を除く非零要素数である（**図 6**）．一般に $m \gg k$ となるので，(1)，(2)，(3) の 3 つの密行列演算のうち，最も計算時間を占めるのは _SYRK である．

スーパーノードの緩和における緩和パラメータとマルチフロンタル法との関係について説明する．緩和パラメータを大きくすれば演算数は増えるが，以下のメリットがある．

- スタックへのコピー回数削減
- Extend-add 演算の演算回数削減
- 行列サイズが大きくなることによる密行列演算のピーク性能比上昇

したがって，行列や使用する計算機によって上記のメリットを得つつ，演算量があまり大きくならない最適なパラメータ値を探すことが重要となる．

3.5 並列化

マルチフロンタル法には，大きく 2 つの並列性がある [26]．1 つは木並列と呼ばれるもので，消去木において親子関係にない複数の部分木に対する処理を並列に行うことが可能である．そしてもう 1 つはノード内並列と呼ばれるもので，ある 1 つのノードの消去は密行列の計算であるため，

これをブロック化することによって並列に処理できる．根に近付けば近付くほど，ノードの数は少なくなるため，木並列の並列性は小さくなる．しかし，一般的に，ノードが根に近付くにつれて，1 つのノードでの計算量は多くなるため，ノード内並列の並列性は大きくなる．

3.6 スーパーノード法との比較

近年の疎行列直接解法では，スーパーノード法とマルチフロンタル法のいずれかが使われる場合が多い．これらの違いは 4 つある．1 つ目は，必要なワーキングストレージである．マルチフロンタル法では，フロンタル行列の分のワーキングストレージが必要である．2 つ目は，各ノード間の関係である．マルチフロンタル法では，親のノードと子のノードのみでやりとりを行う．これに対し，スーパーノード法では，多数の子孫とのやりとりが必要になる．分散メモリ環境において，このやりとりはノード間通信で実現されるため，マルチフロンタル法はスーパーノード法と比べて通信の面では優れていると言える．3 つ目は，1 プロセッサあたりで行う処理について，マルチフロンタル法はスーパーノード法と比べてより大きな行列で BLAS を呼ぶことができ，また間接アクセスの処理が少ない．4 つ目はマルチフロンタル法では，フロンタル行列からアップデート行列，アップデート行列からフロンタル行列へのコピーが必要な点である．このように，マルチフロンタル法とスーパーノード法ではそれぞれメリット・デメリットが存在する．

4. DAG-based マルチフロンタルコレスキー分解

本章では, 提案手法である Directed Acyclic Graph(DAG) によるタスク並列を用いたマルチフロンタル法について説明する．

4.1 Sequential Task Flow

本稿では，DAG を Sequential Task Flow(STF) を用いて定義する．STF では，入力データと出力データの依存関係により，タスク間の依存関係を構成する．そのため，逐次のコードから簡単に並列コードを作成できる．STF のコードでは，タスクをカーネル関数とアクセスモードを付加したデータにより定義する．アクセスモードでは，データの読み取りには R，データの書き込みには RW を使用する．また，RW について，1 つ前の処理とその処理が可換の場合には，COMMUTE 属性（C）を付与できる．STF は並列化しやすい一方で，欠点もある．タスクはランタイムシステムに順次送信されるため，タスクの実行にかかる時間が DAG の構築やタスクの送信に必要な時間よりも短い場合，その DAG の構築やタスクの送信がボトルネックとなってしまう可能性がある．

4.2 ノード間並列

従来のマルチフロンタル法では，木並列とノード内並列のみを考慮していた．このとき，消去木の親ノードの処理を行うためには，子ノードの処理が完了していないといけない．しかし，実際には，子ノードの処理が完了していなくても，処理を行える部分が親ノードに存在する．ノード間並列を用いたスーパーノード法は，文献 [27] で提案されている．マルチフロンタル法においても同様にノード間並列を用いることで子ノードの計算を待たずに親ノードの計算を行うことが可能となる．**図 7** に例を示す．子ノードのブロック行列の番号の extend-add 先が親ノードのブロック行列の番号と対応している．木並列・ノード内並列のみの場合，33 行列の extend-add 演算が行われるまでノード 3 の計算を待たなければならない．33 行列の extend-add 演算はノード 1 の処理の中で一番最後に行われる．つまりノード 1 の計算が終わるまでノード 3 の計算を待たなければならない．しかし，ノード間並列を用いることでそれを待たずに計算できる．

4.3 データ構造

4.3.1 行列データ

一般的なマルチフロンタル法では，フロンタル行列 F_J で消去を行ったあと，L_J と対応する部分を factor storage に保存し，残りをアップデート行列 U_J として保存する．つまり，F_J から L_J と U_J へのコピーが発生する．文献 [28] では，L_J をフロンタル行列ではなく，直接 factor storage 上で計算を行い，フロンタル行列ではアップデート行列のみを計算する方法を提案している．これにより，L_J の分の容量を節約でき，またフロンタル行列から factor storage へのコピーも不要となる．しかしフロンタル行列において，ブロック化を行った場合，factor storage 部分とアップデート行列部分両方を含むブロックが発生する．そこで，L_J と U_J 両方のデータが入っているブロックについては，分解前に，L_J と U_J に分ける方法をとる．L_J のみで構成されている部分をスーパーノード領域，U_J のみで構成されている部分をアップデート領域，そして L_J と U_J で構成されている部分を境界領域を呼ぶ（**図 8**）．

ブロック行列のデータの保持の仕方については，

- 再帰ブロック化 [29]，
- 固定サイズのブロック化 [30]，

がよく用いられる．データ構造は，固定サイズのブロック化を適用し，それぞれのブロックについて行方向優先とした（**図 9(a)**）．これは，境界領域かつスーパーノード領域である対角ブロックの台形部分について，行列の分解を行う際，それぞれのブロック行列を連続的にするためである．

密行列のコレスキー分解を行うサブルーチン `_POTRF` とその行列方程式を解くサブルーチン `_TRSM` が Level-3 BLAS で構成されている．これに対し，packed 形式のコレスキー

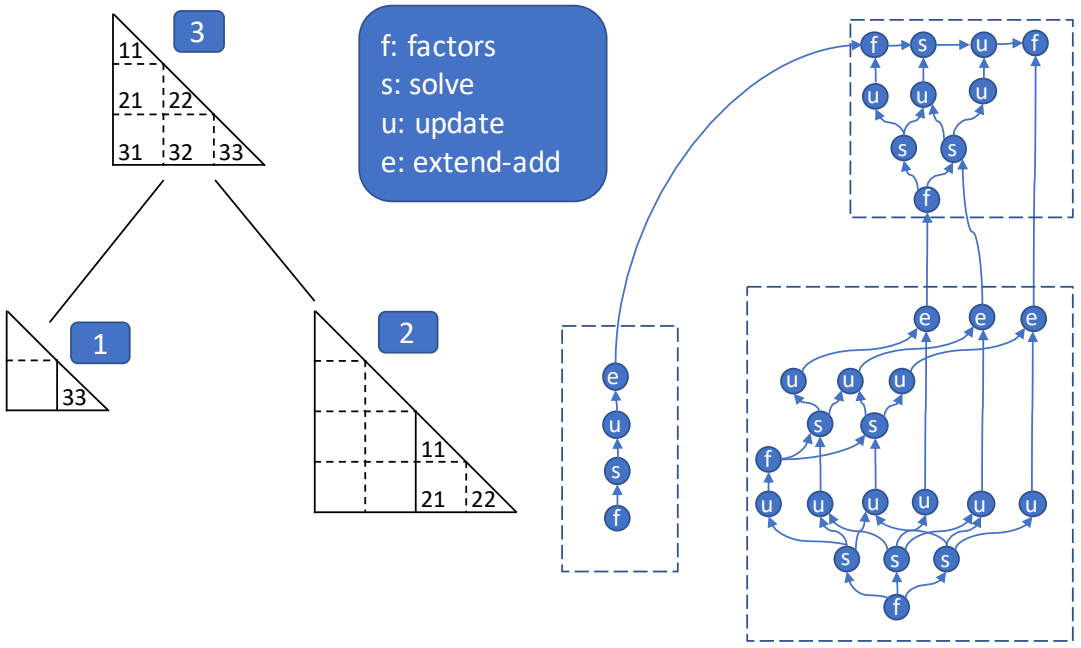


図 7: マルチフロンタル法におけるノード間並列

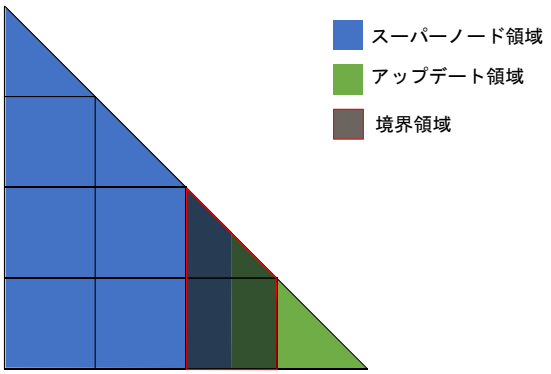
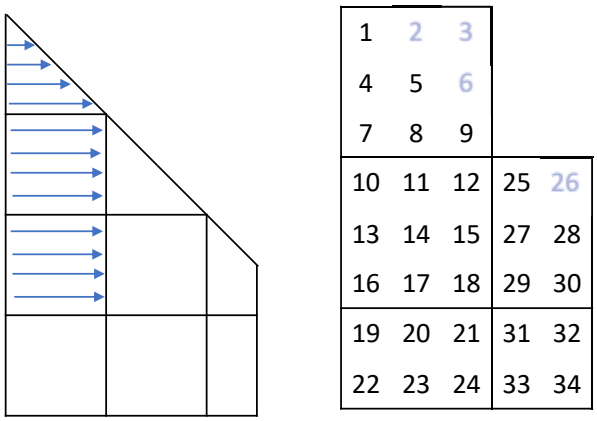
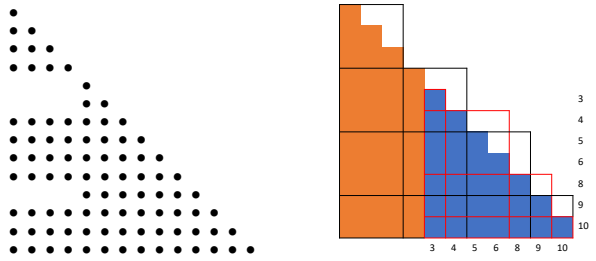


図 8: フロンタル行列のデータ構造



(a) 格納順　(b) 配列のインデックス

図 9: 行ハイブリッドブロック構造



(a) 行列　(b) フロンタル行列

図 10: ブロックローカルインデックスの例

分解を行うサブルーチン_PPTRF とその行列方程式を解くサブルーチン_PPTRS は Level-2 BLAS で構成されている．したがって，対角ブロックは，容量を考えると下三角部分のみを保持する packed 形式の方が良いが，性能の観点から密行列で保持する．例を図 **9(b)** に示す．

4.3.2 ローカルインデックス

フロンタル行列のブロック化に合わせたローカルインデックスのデータ構造について説明する．各ノードにおいて，子ノードの 1 つのブロックに対して親ノードの 1 つのブロックに対応するようにローカルインデックスをブロックに分け，そのブロック内でのインデックスを再計算する．また，それぞれのブロックに対して，対応する子ノードのアップデート行列におけるブロック番号（child_block）と親ノードのフロンタル行列におけるブロック番号（parent_block）を保持する．このデータ構造をブロックローカルインデックスと呼ぶ．図 **10** の行列の場合の child_block, parent_block, local_index を図 **11** に示す．

4.4 アルゴリズム

消去木の各ノードについて，次の処理を行う：

(1) 境界領域でないスーパーノード領域の分解

(2) 境界領域が存在すれば，スーパーノード領域とアップデート領域へ再配置後，分解を行う

$local\_index =$ | 3 | 1 | 2 | 3 | 2 | 3 | 1 |

$child\_block =$ | 2 | 2 | 3 | 3 | 4 | 4 |

$patent\_block =$ | 1 | 2 | 2 | 3 | 3 | 4 |

図 11: ブロックローカルインデックスのデータ構造

(3) アップデート領域の子から親への extend-add 演算

以下では，スーパーノード領域の行列を S，境界領域の行列を B，アップデート領域の行列を U で表すこととする．また，F は，境界領域であればその行列を，そうでなければスーパーノード領域もしくはアップデート領域の行列を表すこととする．実装したプログラムには，次のタスクがある：

(1) 境界領域でないスーパーノード領域の処理

- `factorize`(S_d:R): 対角ブロック S_d のコレスキー分解を行う．この処理は LAPACK のサブルーチンである `_POTRF` を使用する．
- `solve`(S_d:R, S_r:RW): 非対角ブロック S_r を対角ブロック S_d を用いて，

$$S_r \leftarrow S_r S_d^{-\mathrm{T}} \tag{1}$$

と計算する．この処理は Level-3 BLAS のサブルーチンである `_TRSM` を使用する．

- `update`(S_r:R, S_c:R, F_u:RW|C): 非対角ブロック S_r, S_c を用いて F_u を更新する．

$$S_u \leftarrow F_u - S_r S_c^{\mathrm{T}} \tag{2}$$

この処理は L_u が対角ブロックの場合は Level-3 BLAS のサブルーチンである `_SYRK` を使用する．非対角ブロックであれば Level-3 BLAS のサブルーチンである `_GEMM` を使用する．

(2) 境界領域の処理

- `rearrange`(B:R, S:RW, U:RW): 境界領域の行列 B をスーパーノード領域の行列 S とアップデート領域の行列 U に再配置する（**図 12**）．下三角行列部分はすべて密行列として持つ．
- `border_factorize`(S_d:RW, U_d:RW): S_d の上部三角領域 L_d をまずコレスキー分解し，S_d の長方形領域 L_r を

$$L_r \leftarrow L_r L_d^{-\mathrm{T}} \tag{3}$$

として求める．その後，U_d を

$$U_d \leftarrow U_d - L_r L_r^{\mathrm{T}} \tag{4}$$

として更新する．

- `border_solve`(S_d:R, S_r:RW, U_r:RW): S_d の上部三角領域 L_d を用いて，

$$S_r \leftarrow S_r L_d^{-\mathrm{T}} \tag{5}$$

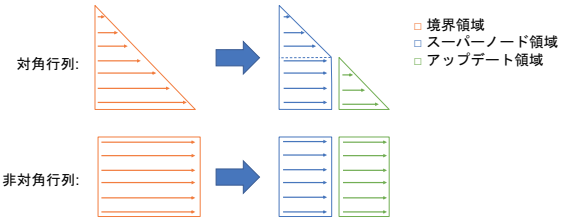


図 12: 境界領域のブロック行列からスーパーノード領域のブロック行列と更新領域のブロック行列への再配置

を解く．解いた S_r と S_d の長方形領域 L_r を用いて

$$U_r \leftarrow U_r - S_r L_r^{\mathrm{T}} \tag{6}$$

を計算し，U_r を更新する．

- `border_update`(S_r:R, S_c:R, F_u:RW|C): 非対角ブロック S_r, S_c を用いて F_u を更新する．

$$U_r \leftarrow U_r - S_r S_c^{\mathrm{T}} \tag{7}$$

(3) アップデート領域の処理

- `extend_add`(U_c:R, F_p:RW, $ilocal$, $jlocal$): 子ノードのブロック行列 U_c から親ノードのブロック行列 F_p への extend_add 演算を行う．$ilocal$ は列方向のローカルインデックス，$jlocal$ は行方向のローカルインデックスである．U_c, F_p が対角成分の場合，下三角部分のみの計算を行う．

ここで，タスクとともに必要な引数とデータアクセスモードについても記述した．

5. 数値実験

本章では，4 章で示した手法を実装し，数値実験を行った．実験に使用した零要素を非零要素とみなす数 max_zero, ブロックサイズ nb とスレッド数 $num\_threads$ は次の通りである．

- $nb = 64 \times i$ for $i = 4, 16$
- $max\_zero = 2^i$ for $i = 8, 20$
- $num\_threads = 1, 2, 4, 8, 16, 32, 64$

5.1 実験対象

実験に用いた 10 種類の行列は**表 1** にまとめた．いずれも The SuiteSparse Matrix Collection[31] から入手した．すべての行列は正定値対称行列で，既約行列である．

6. 実験環境

プログラムは Fortran2008 で実装した．Intel Xeon Phi (Knights Landing, KNL) を CPU とする京都大学のスーパーコンピュータ Camphor 2 上で実験を行った．KNL の仕様を**表 2** に示す．使用したメモリモードはキャッシュモード，クラスタモードは Quadrant である．

BLAS，LAPACK は Intel MKL に含まれているもの

表 1: 使用したテスト行列（$nz(A)$ は行列 A の非零要素数，$nz(L)$ は分解後の行列 L の非零要素数，”Flops”はコレスキー分解の演算量）

Name	n (10^3)	$nz(A)$ (10^6)	$nz(L)$ (10^6)	Flops (10^9)	Application/Description
CEMW/tmp_sym	726.7	2.9	29.5	9.5	Electromagnetics
DNVS/m_t1	97.6	4.9	32.0	18.3	Tubular joint
Boeing/pwtk	217.9	5.9	47.1	21.6	Pressurised wind tunnel
Rotherberg/cfd2	123.4	1.6	36.3	28.7	CFD pressure matrix
GHS_psdef/crankseg_2	63.8	7.1	41.9	43.1	Linear static analysis
Schenk_AFE/af_shell3	504.9	9.0	90.9	49.7	Sheet metal forming matrix
AMD/G3_circuit	1585.5	4.6	90.6	44.4	Circuit simulation
GHS_psdef/bmwcra_1	148.8	5.4	65.0	52.1	Automotive crankshaft model
Oberwolfach/boneS10	914.9	28.2	268.5	267.5	Bone micro-finite element model
GHS_psdef/apache2	715.2	2.8	129.2	165.4	3D structural problem

表 2: KNL の仕様

CPU	Intel Xeon Phi 7250
クロック数	1.40GHz
コア数	68
L1 命令キャッシュ	32KB
L1 データキャッシュ	32KB
分散 L2 キャッシュ	1MB×34
理論演算性能	3.05 TFlops
DDR4	64GB×6 (90GB/s)
MCDRAM	2GB×8 (400GB/s)

を使用した．オーダリングライブラリとして，Metis[32] の Ver 5.1.0 を使用し，ランタイムシステムとして，StarPU[33] の Ver 1.3.2 を使用した．StarPU のスケジュールは lws(locality work stealing) を選択した．これは，タスクをコアごとにキューに追加していき，もしコアが idle 状態になれば，近くのコアからタスクを取るスケジューリングである．コンパイラは，Intel Fortran を使用し，コンパイラオプションは-qopenmp -fast -mkl=sequence を指定した．

6.1 各フェーズの実行時間

まず，ANALYZE, FACTORIZE, SOLVE のそれぞれのフェーズが全体の計算時間のどれくらいの割合を占めるかについて調べた．その結果を**図 13** に示す．作成したプログラムは，各タスクを並列実行可能であるが，使用したスレッド数を 1 に指定して計測した．この図では，FACTORIZE フェーズの計算時間が最速となるような max_zero, nb について示している．ANALYZE フェーズのうち，オーダリングの時間は別にして求めた．この結果から，FACTORIZE フェーズ及び ANALYZE フェーズが全体の大部分を占めることが分かる．また，一般的に，ANALYZE フェーズは係数行列の疎構造のみに依存しているため，疎構造が変わらない場合，FACTORIZE フェーズのみを実行すれば良い．このことから，FACTORIZE フェーズの高速化が重要

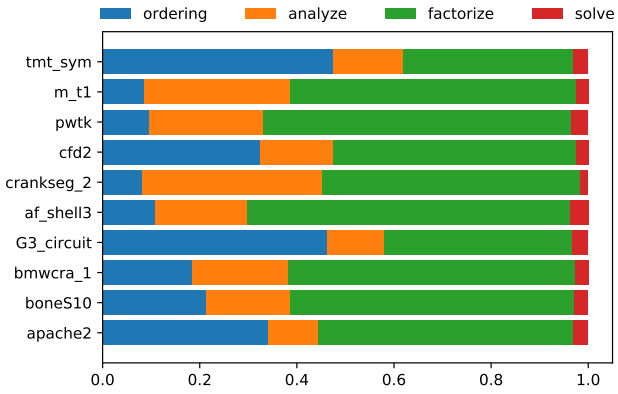


図 13: それぞれのフェーズの実行時間に占める割合

である．

6.2 パラメータ *max_zero* とメモリ容量

各行列に対し，FACTORIZE フェーズにおける max_zero とメモリ容量について求めた．max_zero を変化させ，行列 L の保存に必要な容量（スーパーノード領域）と，実行時に必要なワーキングストレージ（アップデート領域・ボーダー領域）を求めた．このときの nb はコレスキー分解の実行時間が最速となる値である．その結果を**図 14** に示す．max_zero を大きくすれば，行列 L の保存に必要な容量は増えるが，ワーキングストレージの容量は小さくなる．これら 2 つがどれくらいの割合で増減するかによって合計の容量が異なる．行列 tmt_sym，G3_circuit では max_zero を大きくするにつれて全体の容量も大きくなる結果となった．これに対し，行列 m_t1，crankseg_2, bmwcra_1, bonseS10 は max_zero を大きくするにつれて全体の容量も小さくなる傾向が見られる．行列 pwtk, cfd2, af_shell3, apache2 では max_zero がある程度まで大きくなると全体の容量は小さくなるが，ある値からまた全体の容量が大きくなった．

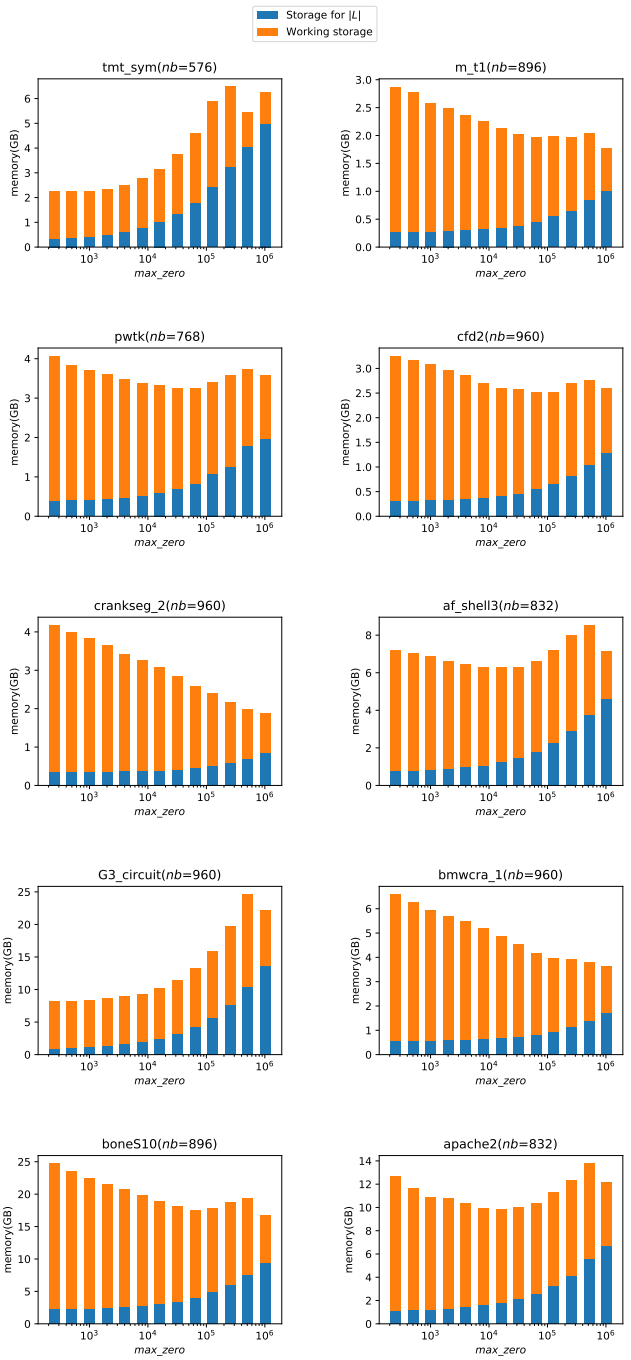


図 14: max_zero を変化させたときの各行列における必要なメモリ容量

6.3 パラメータ max_zero, nb と実行時間

FACTORIZE フェーズでは，次の 3 つの処理を行う：
(1) フロンタル行列を 0 で初期化，
(2) 係数行列 A をフロンタル行列にセット，
(3) コレスキー分解．
パラメータ max_zero, nb を変化させてコレスキー分解の実行時間を計測し，各パラメータの実行時間について，最速となった実行時間を 1 として，その比率を求めた．

まず，1 スレッドのみで実行した結果を図 **15** に示す．図では，濃い色から薄い色に向けて比率が大きくなっている．パラメータ nb については，すべての行列で nb を大きくすればするほど実行時間は短くなっていき，ある程度大きい nb では実行時間があまり変わらなくなった．これは BLAS がある程度大きい行列で性能が出ることに起因する．パラメータ max_zero については，すべての行列で max_zero を大きくするにつれて実行時間は小さくなり，ある点からまた大きくなるという傾向が見られる．max_zero を大きくすると，演算量は増えるが，全体に占める密行列の計算が多くなる．したがって，ある程度密行列の計算の割合を多くし，かつ計算量が増えすぎない値が最適な max_zero となる．

次に，64 スレッドを用いて実行した結果を図 **16** に示す．パラメータ nb に関しては，大きくしすぎると実行時間が遅くなるという結果が得られた．これは，nb が大きすぎると，合計のタスクの数が少なくなってしまい．idle となるコアが多く発生してしまうことが原因だと考えられる．また，実行時間が短くなるような max_zero は 1 スレッドで実行を行ったときよりも値が大きくなっている．これは，max_zero を大きくしても 64 スレッドで実行しているので，1 スレッドあたりの実際の計算時間は増えにくいからだと考えられる．

6.4 スケーラビリティ

コレスキー分解の実行時間が 64 スレッドを用いた場合に最速となる max_zero，nb を固定し，スレッド数を変化させて実行時間を計測した．図 **17** に，64 スレッドと 1 スレッドと比べたときの速度向上率を示す．すべての行列で 32 スレッドまでは実行時間が短くなった．しかし，m_t1, pwtk, cfd2, bmwcra_1 においては，64 スレッドを使用しても 32 スレッドよりは速くならなかった．速度向上率は行列によってさまざまである．最も悪かったのは，m_t1 の 6.2 倍であり，最も良かったのは boneS10 の 26.8 倍である．

7. 結論

本稿では，正定値対称疎行列を係数行列とする連立一次方程式の問題に対して，ランタイムシステムを用いたマルチフロンタル法によるアプローチによって並列化を行った．従来手法のマルチフロンタル法では，ノード内並列・木並列のみを考慮しており，ノード間の並列性については考慮されていなかった．本稿では，ノード間並列も考慮し，実装を行った．数値実験では，StarPU というランタイムシステムを用いて，共有メモリ上での並列化を行った．数値実験の結果から，高い演算性能を得るには，アルゴリズム上に現れるパラメータ nb, max_zero を適切に設定する必要があることがわかった．また，行列によって並列化効率も差が現れた．最も高速化率が良かった行列では，1 スレッド実行時に比べて 64 スレッド実行時に 26.8 倍の高速化が実現できた．今後の課題としては，アクセラレータ

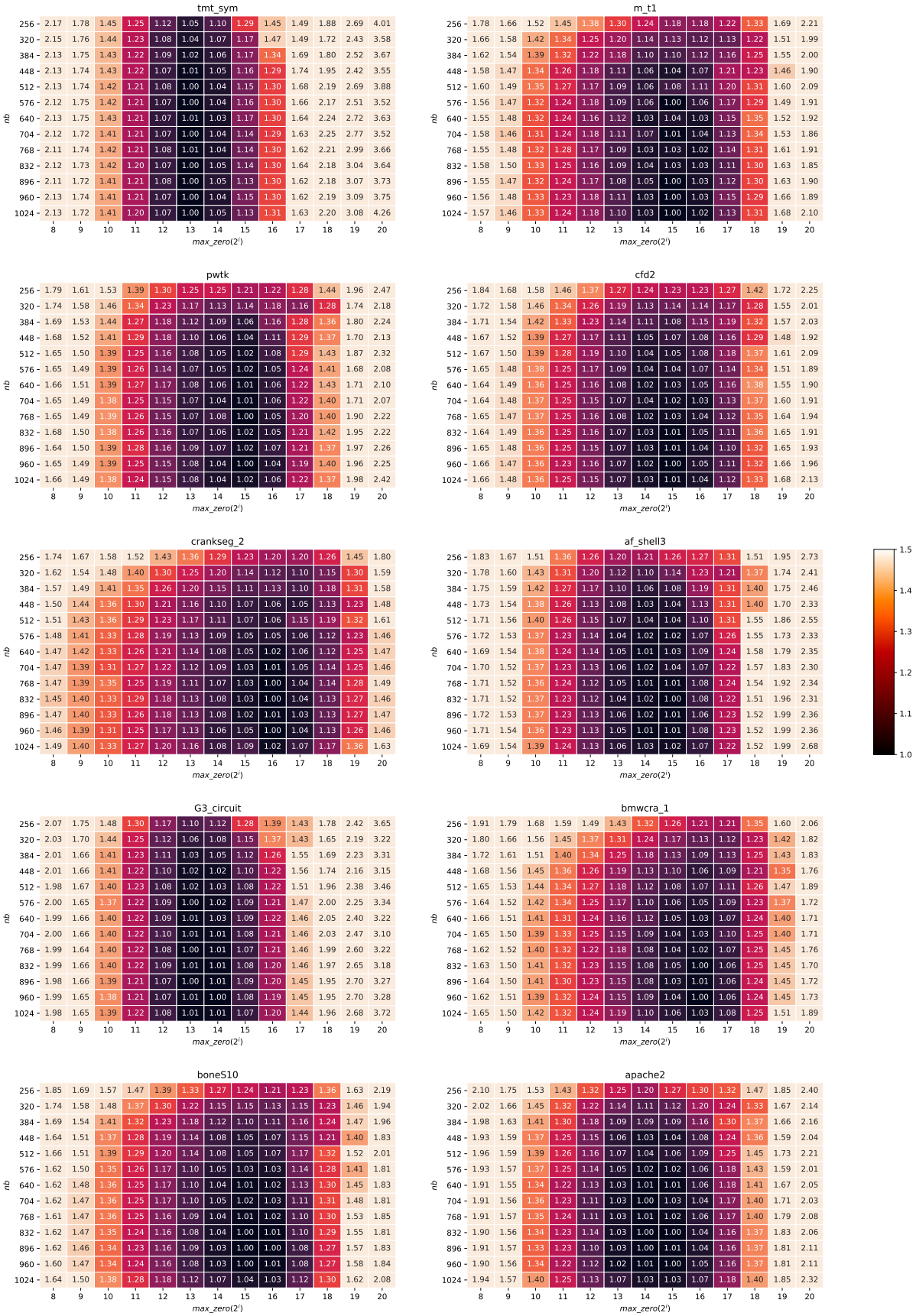


図 15: パラメータ max_zero および nb の変更による各行列での 1 スレッドでの最速実行時間の割合

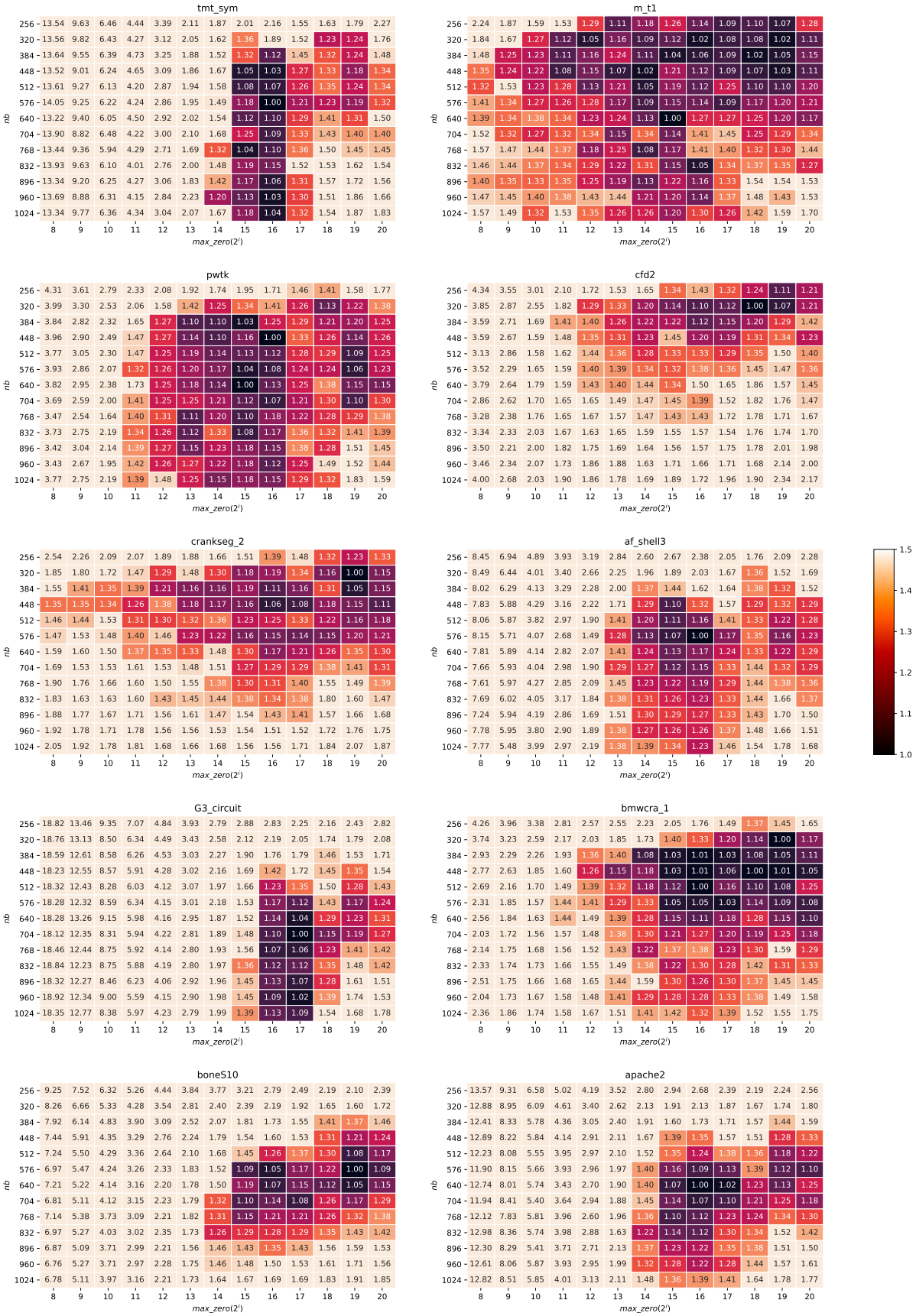


図 16: パラメータ max_zero および nb の変更による各行列での 64 スレッドでの最速実行時間の割合

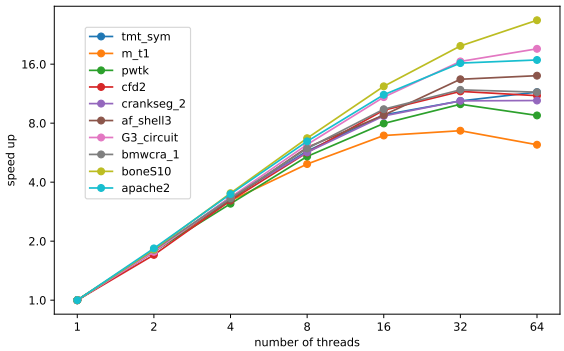


図 17: スレッド数を変化させたときの高速化率

（演算加速器）を搭載した計算機での実装および性能評価や，OpenMP，PaRSEC[34] など他のランタイムシステムの利用，枝刈り [35] によるコレスキー分解の性能改善などがあげられる．

謝辞 本研究成果の一部は，JSPS 科研費 JP17H02828 の助成を受けたものです．

参考文献

[1] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D.: *LAPACK Users' guide*, Vol. 9, SIAM (1999).

[2] Schenk, O. and Gärtner, K.: Solving unsymmetric sparse systems of linear equations with PARDISO, *Future Generation Computer Systems*, Vol. 20, No. 3, pp. 475–487 (2004).

[3] Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S. and Liu, J. W.: A supernodal approach to sparse partial pivoting, *SIAM Journal on Matrix Analysis and Applications*, Vol. 20, No. 3, pp. 720–755 (1999).

[4] Amestoy, P. R., Duff, I. S. and L'excellent, J.-Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers, *Computer methods in applied mechanics and engineering*, Vol. 184, No. 2-4, pp. 501–520 (2000).

[5] Davis, T. A., Rajamanickam, S. and Sid-Lakhdar, W. M.: A survey of direct methods for sparse linear systems, *Acta Numerica*, Vol. 25, pp. 383–566 (online), DOI: 10.1017/S0962492916000076 (2016).

[6] Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H. et al.: A taxonomy of task-based parallel programming technologies for high-performance computing, *The Journal of Supercomputing*, Vol. 74, No. 4, pp. 1422–1434 (2018).

[7] Buttari, A., Langou, J., Kurzak, J. and Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures, *Parallel Computing*, Vol. 35, No. 1, pp. 38–53 (2009).

[8] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H. et al.: Distibuted dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA (2010).

[9] Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S. and Tomov, S.: A hybridization methodology for high-performance linear algebra software for GPUs, *GPU Computing Gems Jade Edition*, Elsevier, pp. 473–484 (2012).

[10] Duff, I. and Lopez, F.: Experiments with sparse Cholesky using a parametrized task graph implementation, *International Conference on Parallel Processing and Applied Mathematics*, Springer, pp. 197–206 (2017).

[11] Duff, I., Hogg, J. and Lopez, F.: Experiments with sparse Cholesky using a sequential task-flow implementation, *Numerical Algebra, Control & Optimization*, Vol. 8, No. 2, pp. 237–260 (2018).

[12] Duff, I., Hogg, J. and Lopez, F.: A new sparse symmetric indefinite solver using A Posteriori Threshold Pivoting, *NLAFET Working Note* (2018).

[13] Lacoste, X., Ramet, P., Faverge, M., Ichitaro, Y. and Dongarra, J.: Sparse direct solvers with accelerators over DAG runtimes (2012).

[14] Lacoste, X., Faverge, M., Bosilca, G., Ramet, P. and Thibault, S.: Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes, *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IEEE, pp. 29–38 (2014).

[15] George, A. and Liu, J. W.: The evolution of the minimum degree ordering algorithm, *SIAM review*, Vol. 31, No. 1, pp. 1–19 (1989).

[16] Heggernes, P., Eisestat, S., Kumfert, G. and Pothen, A.: The computational complexity of the minimum degree algorithm, Technical report, INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING HAMPTON VA (2001).

[17] George, J. A.: Computer implementation of the finite element method, Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE (1971).

[18] George, A. and Liu, J. W.: An automatic nested dissection algorithm for irregular finite element problems, *SIAM Journal on Numerical Analysis*, Vol. 15, No. 5, pp. 1053–1069 (1978).

[19] Liu, J. W.: A compact row storage scheme for Cholesky factors using elimination trees, *ACM Transactions on Mathematical Software (TOMS)*, Vol. 12, No. 2, pp. 127–148 (1986).

[20] Gilbert, J. R., Ng, E. G. and Peyton, B. W.: An efficient algorithm to compute row and column counts for sparse Cholesky factorization, *SIAM Journal on Matrix Analysis and Applications*, Vol. 15, No. 4, pp. 1075–1091 (1994).

[21] Rose, D. J., Tarjan, R. E. and Lueker, G. S.: Algorithmic aspects of vertex elimination on graphs, *SIAM Journal on computing*, Vol. 5, No. 2, pp. 266–283 (1976).

[22] Liu, J. W.: The multifrontal method for sparse matrix solution: Theory and practice, *SIAM review*, Vol. 34, No. 1, pp. 82–109 (1992).

[23] Schreiber, R.: A new implementation of sparse gaussian elimination, *ACM Trans. Math. Softw.*, Vol. 8, No. 3, pp. 256–276 (online), DOI: 10.1145/356004.356006 (1982).

[24] Liu, J. W., Ng, E. G. and Peyton, B. W.: On finding supernodes for sparse matrix computations, *SIAM Journal on Matrix Analysis and Applications*, Vol. 14, No. 1, pp. 242–252 (1993).

[25] Ashcraft, C. and Grimes, R.: The influence of relaxed supernode partitions on the multifrontal method, *ACM Trans. Math. Softw.*, Vol. 15, No. 4, pp. 291–309 (online), DOI: 10.1145/76909.76910 (1989).

[26] Duff, I. S., Erisman, A. M. and Reid, J. K.: *Direct methods for sparse matrices*, Oxford University Press (2017).

[27] Hogg, J. D., Reid, J. K. and Scott, J. A.: Design of

a multicore sparse Cholesky factorization using DAGs, *SIAM Journal on Scientific Computing*, Vol. 32, No. 6, pp. 3627–3649 (2010).

[28] Ng, E. G. and Peyton, B. W.: Block sparse Cholesky algorithms on advanced uniprocessor computers, *SIAM Journal on Scientific Computing*, Vol. 14, No. 5, pp. 1034–1056 (1993).

[29] Gustavson, F., Henriksson, A., Jonsson, I., Kågström, B. and Ling, P.: Recursive blocked data formats and blas' s for dense linear algebra algorithms, *International Workshop on Applied Parallel Computing*, Springer, pp. 195–206 (1998).

[30] Andersen, B. S., Gunnels, J. A., Gustavson, F. G., Reid, J. K. and Waśniewski, J.: A fully portable high performance minimal storage hybrid format Cholesky algorithm, *ACM Transactions on Mathematical Software (TOMS)*, Vol. 31, No. 2, pp. 201–227 (2005).

[31] Davis, T. A. and Hu, Y.: The University of Florida sparse matrix collection, *ACM Transactions on Mathematical Software (TOMS)*, Vol. 38, No. 1, p. 1 (2011).

[32] Karypis, G. and Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM Journal on scientific Computing*, Vol. 20, No. 1, pp. 359–392 (1998).

[33] Augonnet, C., Thibault, S., Namyst, R. and Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, *Concurrency and Computation: Practice and Experience*, Vol. 23, No. 2, pp. 187–198 (2011).

[34] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P. and Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing, *Parallel Computing*, Vol. 38, No. 1-2, pp. 37–51 (2012).

[35] Geist, G. and Ng, E.: Task scheduling for parallel sparse Cholesky factorization, *International Journal of Parallel Programming*, Vol. 18, No. 4, pp. 291–314 (1989).

付　　録

A.1 アルゴリズム

本稿で実装したプログラムのアルゴリズムを **Algorithm 1** に示す.

Algorithm 1: STF によるマルチフロンタルコレスキー分解の並列化

```
1  Let num_node be the number of nodes in the elimination tree.
2  for n ← 1 to num_node do
3      Let num_block be the number of blocks.
4      Let num_supernode be the number of block columns which belong supernode area but not border area.
5      for j ← 1 to num_supernode do
6          submit (factorize, supernode[j][j][n]:RW)
7          for i ← j + 1 to num_block do
8              submit (solve, supernode[j][j][n]:RW, supernode[i][j][n]:R)
9          end
10         for i ← j + 1 to num_block do
11             for k ← i to num_block do
12                 submit (update, supernode[k][j][n]:R, supernode[i][j][n]:R, frontal[k][i][n]:RW|C)
13             end
14         end
15     end
16     if exist border area then
17         Let b be the column index of block matrices in border area.
18         for i ← b to num_block do
19             submit (rearrange, border[i][b][n]:R, supernode[b][b][n]:RW, update[b][b][n]:RW)
20         end
21         submit (border_factorize, supernode[b][b][n], update[b][b][n])
22         for i ← b + 1 to num_block do
23             submit (border_solve, supernode[b][b][n]:R, supernode[i][b][n]:RW, update[i][b][n]:RW)
24         end
25         for i ← b + 1 to num_block do
26             for k ← i to num_block do
27                 submit (border_update, supernode[k][b][n]:R, supernode[i][b][n]:R, update[k][i][n]:RW|C)
28             end
29         end
30     end
31     Let num_local_block be the number of local index.
32     for j ← 1 to num_local_block do
33         for i ← j to num_local_block do
34             ci = child_block[i][n]
35             cj = child_block[j][n]
36             pi = parent_block[i][n]
37             pj = parent_block[j][n]
38             submit (extend_add, update[ci][cj][n]:R, frontal[pi][pj][p]:RW, local_index[i][n], local_index[j][n])
39         end
40     end
41 end
```