Vol.2020-HPC-177 No.20
2020/12/22

ブロックに基づくfill-in選択手法を利用したILU-GMRESソルバ

鈴木 謙吾[1,a)]　深谷 猛[2,b)]　岩下 武史[2,c)]

概要：ILU 分解前処理は非対称な線形方程式に対する前処理手法として広く知られている．ILU 分解における fill-in の考慮は前処理の性能に大きな影響を及ぼすため，その数，位置を適切に選択する必要がある．本稿では，ブロックに基づき fill-in を選択する ILU 分解前処理として，ILUB 分解前処理を提案する．本研究では，提案手法を用いた ILU-GMRES ソルバを開発し，その性能評価を行った．数値実験により，従来手法による ILU(0)-GMRES ソルバと比較し，反復回数，実行時間に関してより良い性能を示すことが確認された．

キーワード：線形反復法，不完全分解前処理，SIMD 演算，GMRES 法

1. はじめに

有限要素解析のような様々な数値シミュレーションの中核となる線形方程式の求解において前処理付きの Krylov 部分空間法が広く用いられている．対象とする問題の大規模化に伴い求解の高速化，安定化が求められ，これまで様々な前処理手法が研究されてきた．特に，非対称な線形方程式に用いられる前処理としては ILU 分解前処理が広く知られている [1]．ILU 分解前処理は ILU 分解の fill-in 制御の方策により性能が変化するため，fill-in 戦略に関し様々な研究がなされている．例えば，最も広く利用されている ILU(0) 分解前処理 [2] では，分解中の新たな fill-in は一切考慮されない．その他に，予め定めた fill-in の大きさに関する基準に基づいて，fill-in の選択を行う ILUT や ILUC 分解前処理が知られている [3], [4].

また，近年では計算コア単位での急速な性能向上は期待できないため，プロセッサの高性能化としてマルチコア化や SIMD 命令への対応というアプローチが用いられている．したがって，HPC アプリケーションの性能向上に関しソフトウェア側のマルチスレッド化や SIMD 演算への対応が重要となっている．

本論文では，上記の背景を踏まえ，SIMD 演算の効果的な利用を前提とした新たな ILU 分解前処理手法として，ILUB 分解前処理と呼ぶ手法を提案する．本手法は，著者らが [5] において対称行列を対象とした不完全コレスキー分解前処理に導入した手法と同様のコンセプトに基づいており，係数行列をブロック分割し，ブロック内に一つでも非ゼロ要素がある場合には，当該ブロック内の fill-in は全て許容する．ILUB 分解前処理では，ILU(0) 分解前処理と比較した場合，fill-in 考慮の影響により反復回数の減少が期待できる一方で，前処理過程である前進・後退代入計算における計算量の増加を伴う．しかしながら，本手法ではブロックに基づいた fill-in 選択を行うことから，前進・後退代入計算が小密行列計算により構成されることとなり，SIMD 演算に適合しやすい利点がある．即ち，SIMD 演算の活用により，前処理過程に要する計算時間の増大を低く抑えることができれば，反復回数削減の効果が現れ，ソルバ全体としての性能向上が期待できる．

本論文の構成は次の通りである．まず，2 章において ILU 分解前処理の基本と従来手法について説明する．次に，3 章で非対称線形方程式の解法として広く用いられる GMRES(Generalized Minimum Residual) 法 [6] について記述する．4 章で SIMD 演算についての説明と SIMD 演算を効率的に行うために本研究で用いる行列格納形式についての説明を行う．そして，5 章で ILUB 分解前処理について述べ，6 章で数値実験による結果を記述する．The SuiteSparse Matrix Collection[7] から選んだ行列を対象と

[1] 北海道大学工学部
School of Engineering, Hokkaido University, Sapporo, Hokkaido, Japan
[2] 北海道大学情報基盤センター
Information Initiative Center, Hokkaido University, Sapporo, Hokkaido, Japan
[a)] kiken50627@eis.hokudai.ac.jp
[b)] fukaya@iic.hokudai.ac.jp
[c)] iwashita@iic.hokudai.ac.jp

し，マルチコアプロセッサ (Intel Xeon Skylake) とメニーコアプロセッサ (Intel Xeon Phi KNL) で実験を行った．最後に，7 章でまとめを行う．

2. ILU 分解前処理

2.1 前処理

本稿では $n \times n$ 行列 $\boldsymbol{A}$ を係数とする連立一次方程式

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{1}$$

を対象とし，反復法を用いた線形ソルバで求解することを考える．

一般に反復法の収束性は係数行列の固有値分布や条件数の影響を受けることが知られ，式 (1) に直接反復法を適用しても収束までに多くの反復回数を要することがある．そこで前処理と呼ばれる手法を用いて収束性の改善を行うことが一般的である．前処理では，$\boldsymbol{A}$ を近似した前処理行列 $\boldsymbol{M}$ を適切な方法で定め，式 (1) を

$$\boldsymbol{M}^{-1}\boldsymbol{Ax} = \boldsymbol{M}^{-1}\boldsymbol{b} \tag{2}$$

または，

$$\boldsymbol{AM}^{-1}\boldsymbol{y} = \boldsymbol{b}, \quad \boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b} \tag{3}$$

のように変形する．変形後の方程式の係数行列 $\boldsymbol{M}^{-1}\boldsymbol{A}$, $\boldsymbol{M}^{-1}\boldsymbol{A}$ の固有値分布や条件数が $\boldsymbol{A}$ の固有値分布，条件数から改善している場合，収束性の向上が期待できる．

2.2 ILU 分解

係数行列が非対称である場合，前処理行列 $\boldsymbol{M}$ を定める代表的手法として ILU 分解がある．ILU 分解は $\boldsymbol{A}$ を下三角行列 $\boldsymbol{L}$ と，上三角行列 $\boldsymbol{U}$ に不完全 LU 分解する手法であり，不完全分解における fill-in 考慮の方策により ILU(0) 分解や ILUT 分解などが知られている．ILU(0) 分解では fill-in は一切考慮されず $\boldsymbol{L}$，$\boldsymbol{U}$ の非零パターンが $\boldsymbol{A}$ の下，上三角部分の非零パターンとそれぞれ等しくなる．また，ILUT 分解では予め定められた閾値に基づき，閾値よりも大きい絶対値を持つ fill-in が許容される．

ILU(0) 分解のように，事前に定めた分解後の行列の非零パターンに基づいて fill-in を制御する手法は，非零パターンを定めた集合 P を用いた Algorithm 1 により実行できる．$\boldsymbol{A}$ の要素の添字集合は，

$$\mathbb{I} = \{1, 2, 3, \ldots, n\}, \ \ i \in \mathbb{I}, \ \ j \in \mathbb{I} \tag{4}$$

を用いて，$\{(i,j) \,|\, (i,j) \in \mathbb{I} \times \mathbb{I}\}$ と表されるため，P は

$$P \subset \{(i,j) \,|\, (i,j) \in \mathbb{I} \times \mathbb{I}\} \tag{5}$$

を満たす．また，分解中の破綻（break down）を防ぐために，

$$P \supset \{(i,j) \,|\, i = j\} \tag{6}$$

とするのが一般的である．なお，式 (6) が満たされる場合においても，分解が破綻する可能性があることを付記する．

ILU(0) 分解の場合，P は，

$$P = \{(i,j) \,|\, a_{i,j} \neq 0\} \tag{7}$$

のように与えられる．

Algorithm 1 ILU factorization method based on nonzero pattern

```
1: Define any nonzero pattern set P
2: for i = 2, ..., n do
3:     for k = 1, ..., i − 1 and if (i, k) ∈ P do
4:         a_{i,k} = a_{i,k} / a_{k,k}
5:         for j = k + 1, ..., n and if (i, j) ∈ P do
6:             a_{i,j} = a_{i,j} − a_{i,k} a_{k,j}
7:         end for
8:     end for
9: end for
```

3. GMRES 法

本稿では，式 (1) を Krylov 部分空間法の一つである一般化最小残差（GMRES: Generalized Minimum Residual）法により解く．GMRES 法における s 反復目の近似解 $\boldsymbol{x}_s \in \boldsymbol{x}_0 + K_s(\boldsymbol{A}, \boldsymbol{r}_0)$ は，残差ベクトル $\boldsymbol{r}_s = \boldsymbol{b} - \boldsymbol{Ax}_s$ のノルムを最小化するように与えられる．ここで，$\boldsymbol{x}_0$ は任意に与えられる初期近似解であり，$K_s(\boldsymbol{A}, \boldsymbol{r}_0)$ は，

$$K_s(\boldsymbol{A}, \boldsymbol{r}_0) \coloneqq \mathrm{Span}(\boldsymbol{r}_0, \boldsymbol{Ar}_0, \boldsymbol{A}^2\boldsymbol{r}_0, \ldots, \boldsymbol{A}^{s-1}\boldsymbol{r}_0) \tag{8}$$

によって定義される．ただし，$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{Ax}_0$ である．GMRES 法は反復回数が増えるにつれ必要な記憶容量と計算量も増加していくため，多くの反復回数を必要とする問題を解くような場面において実用的ではない．そこで実際には，m 反復ごとに計算を打ち切り，$\boldsymbol{x}_0 = \boldsymbol{x}_m$ として再び計算を行うリスタート付き GMRES 法，即ち GMRES(m) 法が用いられる．なお，GMRES (m) 法ではリスタートごとに残差ノルムの局所的な最小化のみが行われるため GMRES 法と比べ収束性が悪くなる場合があるが，残差ノルムの単調減少性は維持される．

3.1 ILU 分解前処理付き GMRES 法

Krylov 部分空間反復法において，前処理を用いる場合，式 (2) または (3) の連立一次方程式を陽的に構成することはせず，反復の内部でそれと等価な処理を行う．ILU 分解前処理を GMRES(m) 法に適用した場合，一反復ごとに，与えられたベクトル $\boldsymbol{z}$ に対して，$\boldsymbol{w} = \boldsymbol{M}^{-1}\boldsymbol{z}$ を計算する前処理過程が追加される．ILU 分解前処理では $\boldsymbol{M} = \boldsymbol{LU}$ であるため前処理過程では次の前進代入計算

$$\boldsymbol{t} = \boldsymbol{L}^{-1}\boldsymbol{z} \tag{9}$$

と，後退代入計算

$$\boldsymbol{w} = \boldsymbol{U}^{-1}\boldsymbol{t} \tag{10}$$

が行われる．

Algorithm 2 に式 (2) の形で前処理を行う，左前処理方式の ILU-GMRES(m) 法の実行手順を示す．

Algorithm 2 ILU-GMRES(m) method

1: Compute $(\boldsymbol{LU})\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, $\beta = \|\boldsymbol{r}_0\|$ and $\boldsymbol{v}_1 = \boldsymbol{r}_0/\beta$
2: **for** $j = 1, \cdots, m$ **do**
3: Solve $(\boldsymbol{LU})\boldsymbol{w}_{j+1} = \boldsymbol{A}\boldsymbol{v}_j$
4: **for** $i = 1, \cdots, j$ **do**
5: $h_{i,j} = (\boldsymbol{w}_{j+1}, \boldsymbol{v}_i)$
6: $\boldsymbol{w}_{j+1} = \boldsymbol{w}_{j+1} - h_{i,j}\boldsymbol{v}_i$
7: **end for**
8: $h_{j+1,j} = \|\boldsymbol{w}_{j+1}\|$
9: $\boldsymbol{v}_{j+1} = \boldsymbol{w}_{j+1}/h_{j+1,j}$
10: Define $\boldsymbol{V}_j = \{\boldsymbol{v}_1, \cdots, \boldsymbol{v}_j\}, \bar{\boldsymbol{H}}_j = \{h_{k,l}\}_{1 \le k \le l+1; 1 \le l \le j}$
11: Compute $\boldsymbol{y}_j = \mathrm{argmin}_y \|\beta \boldsymbol{e}_1 - \bar{\boldsymbol{H}}_j \boldsymbol{y}\|$, $\boldsymbol{x}_j = \boldsymbol{x}_0 + \boldsymbol{V}_j \boldsymbol{y}_j$
12: if $\|\boldsymbol{r}_j\|$ is small enough, quit
13: **end for**
14: if converges then stop, else set $\boldsymbol{x}_0 = \boldsymbol{x}_m$ and go to 1

4. 行列格納形式と SIMD 演算

4.1 BCRS 形式

一般に疎行列は要素のほとんどが零であるという性質を利用しデータを圧縮した形でメモリ上に格納される．行列ベクトル積の実装のしやすさなどから広く使用される格納形式として CRS(Compressed Row Strage) 形式がある．CRS 形式では非零要素の値，列番号をそれぞれ行優先で格納する二つの配列と，それらの配列での各行の開始位置を格納する配列が用いられる．図 1(a) のサンプル行列に対する CRS 形式の格納方法を図 1(b) に示す．

CRS 方式を拡張した疎行列格納形式として, BCRS(Block Compressed Row Storage) 形式 [8] がある．BCRS 形式では，行列は $n_l \times n_w$ のブロックに分割され，ブロックを要素とみなした CRS 形式で格納される．つまり，少なくとも一つの非零要素を含むブロック（非零ブロック）の位置情報とブロック内の各要素の値が保持される．なお，本研究では，非零ブロック内の各要素の値は列優先で格納することとする．図 1(c) にサンプル行列に対する BCRS 形式の格納方法を示す．なお，BCRS($n_l \times n_w$) 形式とはブロックサイズとして $n_l \times n_w$ を用いた BCRS 形式を表すこととする．BCRS 形式は，各節点に複数の自由度が置かれた有限要素構造解析のように，係数行列がブロック構造を内包する問題に適しており，こうした問題では，CRS 形式と比べてメモリアクセス量を減らすことができる．また，各ブロックは密行列として扱うことができるため SIMD(Simgle Instruction Multiple Data) 演算に親和性が高く，その観点から利用される場合がある．

$$\begin{pmatrix} a_{11} & a_{12} & 0 & a_{14} \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

(a) サンプル行列

$$\begin{aligned} &\mathrm{val} = \{a_{11}, a_{12}, a_{14}, a_{22}, a_{33}, a_{43}, a_{44}\} \\ &\mathrm{col\_ind} = \{1, 2, 4, 2, 3, 3, 4\} \\ &\mathrm{row\_ptr} = \{1, 4, 5, 6, 8\} \end{aligned}$$

(b) CRS 形式

$$\begin{aligned} &\mathrm{bval} = \{a_{11}, 0, a_{12}, a_{22}, 0, 0, a_{14}, 0, a_{33}, a_{43}, 0, a_{44}\} \\ &\mathrm{bcol\_ind} = \{1, 2, 2\} \\ &\mathrm{brow\_ptr} = \{1, 3, 4\} \end{aligned}$$

(c) BCRS(2×2) 形式

図 1: CRS 形式と BCRS 形式

4.2 SIMD 演算

SIMD は一つの命令を複数のデータに適用し並列に処理する技術である．SIMD 演算の拡張命令をサポートする CPU では主に専用レジスタ上にデータを読み込み演算を行うため，効率的に SIMD 演算を行うためには，データ間に依存関係がないことに加え，データがメインメモリ上に連続に保持されていることが要求される．

5. ILUB 分解前処理

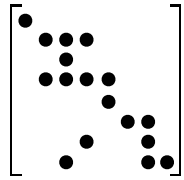
本論文では，SIMD 演算の効果的な利用を前提に，ILU 分解前処理における fill-in 選択手法を検討することにより，ILU-GMRES 法の性能改善を試みる．一般に ILU 分解において多くの fill-in を考慮することで ILU-GMRES 法の収束性は向上することが期待されるが，これは計算量の増加を伴うため，ソルバ全体の性能向上には，一反復当たりの計算時間を短縮する技術の導入が望ましい．そこで、本論文では，BCRS 形式が有する SIMD 演算との親和性に着目し，BCRS 形式と同様のブロック構造に基づいて fill-in 選択を行う新しい ILU 分解前処理手法を提案する．

5.1 ILUB 分解

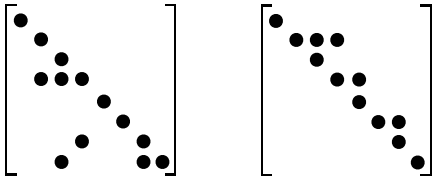
本研究では，ブロックを単位とし fill-in を考慮する ILU 分解前処理として ILUB 分解前処理と呼ぶ前処理手法を提案する．ILUB 分解では，係数行列をブロックに分割したものを考え，非零ブロック内の全要素に関しては fill-in が許容される形で ILU 分解が行われる．非零ブロックの添字集合 Q は式 (4) を用い，

$$Q = \{(\lceil i/n_l \rceil, \lceil j/n_w \rceil) \,|\, a_{i,j} \neq 0, i \in \mathbb{I}, j \in \mathbb{I}\} \tag{11}$$

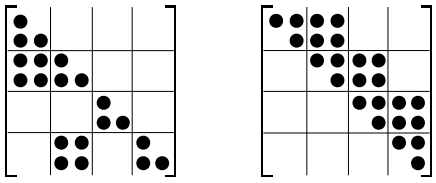
と表される．また，ブロックを要素と考えた場合の行，列番号を表す添字を I，J とすれば，ブロックの添字集合は，



(a) サンプル行列



(b) ILU(0) 分解後の比零パターン



(c) ILUB(2×2) 分解後の比零パターン

図 2: 前処理行列の非零パターン

$$\mathbb{I}_{n_l} = \{1, 2, 3, \cdots, \lceil n/n_l \rceil\} \tag{12}$$

$$\mathbb{I}_{n_w} = \{1, 2, 3, \cdots, \lceil n/n_w \rceil\} \tag{13}$$

を用いて，$\{(I,J) \,|\, (I,J) \in \mathbb{I}_{n_l} \times \mathbb{I}_{n_w}\}$ と表されるため，Q は

$$Q \subset \{(I,J) \,|\, (I,J) \in \mathbb{I}_{n_l} \times \mathbb{I}_{n_w}\} \tag{14}$$

を満たす．したがって，ILUB 分解では 2 章に示した ILU 分解後の非零パターンを定めた集合 P は，

$$P = \{(i,j) \,|\, n_l(I-1) < i \le n_l I, \\ n_w(J-1) < j \le n_w J, (I,J) \in Q\} \tag{15}$$

となる．

行列の非零要素を黒円で表せば，図 2(a) のサンプル行列を ILU(0)，ILUB で分解した結果はそれぞれ図 2(b)，図 2(c) に示す通りになる．なお，本稿ではブロックサイズとして $n_l \times n_w$ を用いる ILUB 分解前処理を ILUB($n_l \times n_w$) と表記する．

5.2 ILUB 分解前処理の SIMD 化

前節の通り，ILUB 分解では非零ブロック内の全要素に対し fill-in が許容されるため，ILUB 分解により生成される前処理行列は小さな密行列からなる上，下三角行列と考えることができる．これにより，前処理過程における代入計算はデータの逐次性が存在する対角要素を含むブロックに関する計算を除き，密行列の行列ベクトル積として実装が可能で，効率的な SIMD 化を行うことができる．本研究では，前処理行列に対してブロック内は列優先とした

```
//Solve Lt=z
for(i=0; i<N; i++){
  t[i] = z[i];
  for(j=row_ptr[i]-1; j<row_ptr[i+1]-1; j++){
    if(col_ind[j]-1 < i) {
      t[i] = t[i] - val[j] * t[col_ind[j]-1];
    }
  }
}
```

図 3: CRS 形式による前進代入の実装例

```
//Solve Lt=z
for(i=0; i<Nl; i++){
  for(j=0; j<2; j++) t[i*2+j] = z[i*2+j];
  for(j=brow_ptr[i]-1; j<brow_ptr[i+1]-2; j++){
    t_idx = (bcol_ind[j]-1) * 2;
    for(k=0; k<2; k++)
      t[i*2+k] -= bval[j*4+k] * t[t_idx];
    for(k=0; k<2; k++)
      t[i*2+k] -= bval[j*4+2+k] * t[t_idx+1];
  }
  t[i*2+1] -= bval[j*4+1] * t[i*2];
}
```

(a) BCRS(2×2) 形式による実装

```
//Solve Lt=z
__m128d XL0, XL1, XZ0, XZ1, XT;
for(i=0; i<Nl; i++){
  XT = _mm_load_pd(&z[i*2]);
  for(j=brow_ptr[i]-1; j<brow_ptr[i+1]-2; j++){
    t_idx = (bcol_ind[j]-1) * 2;
    XL0 = _mm_load_pd(&bval[j*4]);
    XL1 = _mm_load_pd(&bval[j*4+2]);
    XZ0 = _mm_load1_pd(&t[t_idx]);
    XZ1 = _mm_load1_pd(&t[t_idx+1]);
    XT = _mm_sub_pd(XT, _mm_mul_pd(XL0, XZ0));
    XT = _mm_sub_pd(XT, _mm_mul_pd(XL1, XZ1));
  }
  _mm_store_pd(&y[i*2], XT);
  t[i*2+1] -= bval[j*4+1] * t[i*2]
}
```

(b) BCRS(2×2) 形式による実装（Intel 組み込み関数あり）

図 4: BCRS 形式による前進代入の実装例

BCRS 形式を用い，前処理過程である前進・後退代入計算を SIMD 化する．なお，ILUB 分解前処理を用いる場合においても，係数行列に対しては CRS 形式を用いる．前進代入計算の実装方式について，図 3 に従来型の CRS 形式を用いたものを示し，図 4 に (a)BCRS 形式を用いたもの，(b)BCRS 形式を用い，Intel 社が提供する組み込み関数を利用して SIMD 化を行ったものの 2 種を示す．後退代入計算についても同様の実装を行っている．

5.3 ILUB 分解前処理の効果

一般に，ILUB 分解により得られる前処理行列は ILU(0) 分解によるものと比較し多くの非零要素をもつ．この性質により，ILUB 分解前処理は ILU(0) 分解前処理と比べ反復回数の削減が期待できるが，前処理過程における計算量の増加という悪影響を伴う．計算量の増加による計算時間の増加を SIMD 化により抑制することができれば，反復回数の削減効果がより顕著となり，ソルバ全体としての性能向上が期待できる．

ILU(0)-GMRES ソルバの計算時間 T_0，ILUB-GMRES ソルバの計算時間 T_{B} は，それぞれ，反復回数 N_0，N_{B}，一反復あたりの計算時間の平均値 $\bar{t}_{0\mathrm{G}}$，$\bar{t}_{\mathrm{BG}}$ を用いて，

$$T_0 = N_0 \times \bar{t}_{0\mathrm{G}} \tag{16}$$

$$T_{\mathrm{B}} = N_{\mathrm{B}} \times \bar{t}_{\mathrm{BG}} \tag{17}$$

のように与えられる．ここで，ILU(0) 前処理，ILUB 前処理で行われる 1 回の前進・後退代入計算に要する計算時間をそれぞれ $t_{0\mathrm{fb}}$，t_{Bfb} と表す．このとき，$\bar{t}_{0\mathrm{G}}$ は，

$$\bar{t}_{0\mathrm{G}} = t_{0\mathrm{fb}} + \bar{t}_{\mathrm{G}} \tag{18}$$

のように書くことができ，$\bar{t}_{\mathrm{G}}$ は代入計算以外の部分に要した計算時間の一反復当たりの平均値である．ここで，代入計算（前処理過程）以外の GMRES 法に関する実装が両ソルバで同一であることから，

$$\bar{t}_{\mathrm{BG}} \fallingdotseq t_{\mathrm{Bfb}} + \bar{t}_{\mathrm{G}} \tag{19}$$

のように近似する．次に，前進・後退代入計算の演算量は前処理行列の非零要素数に比例することを考慮し，

$$t_{\mathrm{Bfb}} \fallingdotseq \gamma_{\mathrm{B0}} \times \frac{1}{\eta} \times t_{0\mathrm{fb}} \tag{20}$$

と近似する．ここで，γ_{B0} は ILUB 前処理と ILU(0) 前処理の前処理行列の非零要素数の比であり，η は BCRS 形式導入の影響を含む SIMD 命令による効果を表す無次元数（計算速度の比）である．さらに，定数 α を用いて $t_{0\mathrm{fb}} = \alpha \cdot t_{0\mathrm{G}}$ とすれば，ILUB-GMRES ソルバの ILU(0)-GMRES ソルバに対する求解性能比 S は，

$$\begin{aligned} S = \frac{T_0}{T_{\mathrm{B}}} &\fallingdotseq \frac{N_0}{N_{\mathrm{B}}} \frac{t_{0\mathrm{fb}} + \bar{t}_{\mathrm{G}}}{\gamma_{\mathrm{B0}} \cdot \frac{1}{\eta} \cdot t_{0\mathrm{fb}} + \bar{t}_{\mathrm{G}}} \\ &= \frac{N_0}{N_{\mathrm{B}}} \frac{\alpha + 1}{\gamma_{\mathrm{B0}} \cdot \frac{1}{\eta} \cdot \alpha + 1} \end{aligned} \tag{21}$$

にように見積もられる．

6. 数値実験

6.1 実験条件

テスト行列として，The SuiteSparse Matrix Collection から比較的問題サイズの大きな 7 つの係数行列を選択した．

表 1: テスト行列データ

データセット	問題の内容	問題サイズ	非零要素数	対称性
atmosmodd	計算流体力学	1,270,432	8,814,880	非対称
atmosmodj	計算流体力学	1,270,432	8,814,880	非対称
G3_circuit	回路シミュレーション	1,585,478	7,660,826	対称
parabolic_fem	計算流体力学	525,825	3,674,625	対称
StocF-1465	計算流体力学	1,465,137	21,005,389	対称
t2em	電磁気学	921,632	4,590,832	非対称
Transport	構造解析	1,602,111	23,487,281	非対称

各行列の次元数，非零要素数等の概要を表 1 に示す．本研究では，対称行列を扱う場合においても対称性は考慮せず，非対称行列と同様に扱っている．右辺ベクトルは全要素が 1 であるベクトルを採用し，収束判定条件は相対残差ノルムが 10^{-8} 以下とした．また，GMRES(m) 法の初期近似解には零ベクトルを用い，リスタート周期は 50 とした．

北海道大学情報基盤センターの Fujitsu CX2550 M4 と，Fujitsu CX1640 M1 の二種類のノードにより数値実験を行った．Fujitsu CX2550 M4 は 20 コアの Intel Xeon Gold 6148 プロセッサ (Skylake) 二つと 384GB のメモリを備えており，Fujitsu CX1640 M1 は 68 コアからなる Intel Xeon Phi プロセッサ 7250 (KNL) 一つとプロセッサ内蔵の広帯域メモリ（MCDRAM，フラットモード）16GB の他に 96GB のメモリを備えている．いずれのノードのプロセッサも拡張命令セットとして Intel AVX-512 をサポートしており 512 ビット幅の SIMD 演算が実行可能である．プログラムは C 言語により実装し Intel コンパイラ v19.0.5.281 によりコンパイルを行った．コンパイルオプションとして Fujitsu CX2550 M4 では-xCORE-AVX512 -O3 -fp-model consistent，Fujitsu CX1640 M1 では-xMIC-AVX512 -O3 -fp-model consistent を用いた．本実験では計算ノードの 1 スレッドのみを使用した．

6.2 数値実験結果

本実験では ILUB 分解のブロックサイズとして，(2×1)，(2×2)，(4×1)，(4×4)，(8×1)，(8×2) を用いた．これは，プロセッサの SIMD 幅が最大で倍精度浮動小数点数 8 つ分であることや，ブロックサイズを大きくしすぎると計算量が多くなり性能向上が期待できないことなどを考慮したためである．なお，横長のブロックサイズを選択することも可能であるが，横長のブロックを用いた場合，行方向に SIMD 演算が実行されリダクション処理を必要とすることから効率的な SIMD 化が実現できないため，本研究では横長のブロックサイズは選択していない．

6.2.1 マルチコアプロセッサにおける実験結果

本項では汎用マルチコアプロセッサである Intel Xeon Skylake プロセッサによる数値実験結果を示す．

表 2 に G3_circuit と t2em について，従来手法である ILU(0)-GMRES(m) 法と提案手法である ILUB-GMRES(m) 法の比較を示す．どちらの行列に対しても，

ILUB-GMRES(m) 法の全てのブロックサイズにおいて ILU(0)-GMRES(m) 法と比較し反復回数，実行時間ともに減少していることが分かる．G3_circuit では ILUB(4×4) で最良の結果となり，反復回数が大きく減少したために実行時間は 86%削減された．また，t2em では ILUB(8×1) で最良となり実行時間は 75%削減された．

表 3 に全テスト行列に対する ILU(0)-GMRES(m) 法と ILUB-GMRES(m) 法で最良のブロックサイズを選択した場合の比較を示す．これより，適切なブロックサイズを選択することで，ILUB-GMRES(m) 法は全てのテスト行列に対し ILU(0)-GMRES(m) 法と比べ良い性能を示すことが確認できる．

表 4 に t2em の場合において，ILUB 分解前処理がソルバの性能向上に影響を与える効果の内訳について示す．Case1 は従来手法の ILU(0)-GMRES(m) 法を用いた場合である．Case2，Case3 はそれぞれ，組み込み関数なし，組み込み関数ありの ILUB(8×8)-GMRES(m) 法を用いた場合であり，実装例は図 4 の (a)，(b) が対応している．Case1 と Case2 の比較から，fill-in を考慮したことの効果により反復回数が大幅に削減され一反復あたりの計算時間は増加したものの全体としての計算時間が減少していることが分かる．さらに，Case2 と Case3 の比較から，前進・後退代入計算における明示的な SIMD 組み込み関数の利用が一反復あたりの計算時間の低減に寄与することが分かる．

次に式 (21) による性能評価に関して，t2em を例に考察する．式 (21) において，N_0，N_B の値は，実際に求解を実行して明らかとなる値である．一方，γ_{B0} については，対象とする問題とブロックサイズを確定した時点で求められる．また，α についても問題が確定した時点で，演算量やメモリからの転送量に基づき，その概算値は導出可能である．次に，η について考える．η は利用する SIMD 命令の SIMD 幅が大きいほど高い値となると期待されるが，その値は計算の対象や計算プラットフォームに依存し，一般的な見積もり手法が確立されていない．例えば，プロセッサによっては SIMD 命令を利用した場合，実態としてクロックが低下するような事例が報告されており，利用する SIMD 命令に対してアプリケーション上でどの程度の速度向上が得られるかに関して統一的な見積もり法の確立を困難にしている．そこで，本研究では，t2em を用いた数値実験において，1000 回の前進後退代入計算を行い，式 (20) から η を実験的に算出した．表 5 に本実験における 1 回あたりの前進・後退代入計算の計算時間，η，γ_{B0} を示す．表中において，CRS は前処理行列を CRS 形式で格納し，ILU(0) 前処理で用いる実装方式を利用した場合の結果である．ブロックサイズが 2×1 や 2×2（SIMD 幅：2）の場合には，η は 2 に近く，ほぼ理想的といえる効果が得られている．その結果，η と γ_{B0} はほぼ同程度となり，SIMD 命令を用いた BCRS 形式の導入により計算量増加の影響を効果的に抑制できていることが分かる．また，ブロックサイズが 8×1，8×2（SIMD 幅：8）の場合，組み込み関数を利用した明示的な SIMD 命令の利用の効果が大きいことが読み取れる．本数値実験結果から，利用した計算環境における η の概算値は，SIMD 幅が 2 の場合で 1.7 程度，SIMD 幅が 4，8 の場合で 2.5 程度と見積もられる．

表 2: Intel Xeon Skylake プロセッサでの ILUB-GMRES(m) 法と ILU(0)-GMRES(m) 法の比較

(a) G3_circuit

	反復回数	計算時間 [s]	時間/反復 [s]
ILU(0)	1656	214.08	1.29E-1
ILUB(2×1)	1472	188.55	1.28E-1
ILUB(2×2)	615	81.53	1.33E-1
ILUB(4×1)	1018	136.55	1.34E-1
ILUB(4×4)	201	29.91	1.49E-1
ILUB(8×1)	557	84.67	1.52E-1
ILUB(8×2)	312	50.35	1.61E-1

(b) t2em

	反復回数	計算時間 [s]	時間/反復 [s]
ILU(0)	3806	272.85	7.17E-2
ILUB(2×1)	1502	107.96	7.19E-2
ILUB(2×2)	1487	107.73	7.25E-2
ILUB(4×1)	1210	89.87	7.43E-2
ILUB(4×4)	1273	98.54	7.74E-2
ILUB(8×1)	815	69.02	8.47E-2
ILUB(8×2)	882	76.20	8.64E-2

表 3: Intel Xeon Skylake プロセッサでの全テスト行列に対する ILUB-GMRES(m) 法と ILU(0)-GMRES(m) 法の比較

	ILUB-GMRES(m) 法			ILU(0)-GMRES(m) 法	
データセット	ブロックサイズ	反復回数	計算時間 [s]	反復回数	計算時間 [s]
atmosmodd	8×2	112	15.34	232	23.49
atmosmodj	4×1	137	15.02	204	21.07
G3_circuit	4×4	201	29.90	1656	214.08
parabolic_fem	8×2	1654	100.38	4230	181.02
StocF-1465	2×1	1270	193.48	2228	347.92
t2em	8×1	815	69.02	3806	272.85
Transport	4×4	819	157.97	4150	660.86

表 4: Intel Xeon Skylake プロセッサでの t2em に対する ILUB 分解前処理の効果の内訳

Case	前処理	反復回数	計算時間 [s]	時間/反復 [s]
Case1	ILU(0)	3806	272.85	7.17E-2
Case2	ILUB(8×1) 組込関数なし	819	74.89	9.19E-2
Case3	ILUB(8×1) 組込関数あり	819	69.02	8.47E-2

6.2.2 メニーコアプロセッサにおける実験結果

メニーコアプロセッサである Intel Xeon Phi プロセッサによる数値実験結果について本項に示す．

表 6 に全てのテスト行列に対して，最良のブロックサイ

表 5: Intel Xeon Skylake プロセッサでの t2em に対する前進・後退代入計算の計算時間と SIMD 化の効果

前処理	γ_{B0}	計算時間 [ms]			η
		CRS	BCRS	BCRS 組込関数あり	
ILU(0)	1.00	12.98	-	-	-
ILUB(2×1)	1.60	15.28	13.25	12.00	1.73
ILUB(2×2)	2.00	16.79	15.13	13.26	1.96
ILUB(4×1)	2.80	19.80	18.42	15.58	2.33
ILUB(4×4)	4.00	26.46	24.22	18.41	2.82
ILUB(8×1)	5.20	32.80	33.70	26.35	2.56
ILUB(8×2)	5.60	34.84	34.86	27.08	2.69

表 6: Intel Xeon Phi プロセッサでの全テスト行列に対する ILUB-GMRES(m) 法と ILU(0)-GMRES(m) 法の比較

データセット	ILUB-GMRES(m) 法			ILU(0)-GMRES(m) 法	
	ブロックサイズ	反復回数	計算時間 [s]	反復回数	計算時間 [s]
atmosmodd	8×2	112	47.73	232	86.54
atmosmodj	8×2	118	49.72	204	77.85
G3_circuit	4×4	201	100.84	1656	776.84
parabolic_fem	8×2	1654	320.67	4230	735.77
StocF-1465	2×1	1270	778.60	2228	1461.19
t2em	8×1	815	224.67	3806	999.80
Transport	4×4	819	522.77	4150	2724.19

表 7: Intel Xeon Skylake プロセッサと Intel Xeon Phi プロセッサにおける ILUB-GMRES(m) 法の求解性能比

データセット	求解性能比			
	Intel Xeon Skylake		Intel Xeon Phi	
	BCRS 組込関数あり	BCRS	BCRS 組込関数あり	BCRS
atmosmodd	1.53	1.33	1.81	1.58
atmosmodj	1.40	1.30	1.57	1.36
G3_circuit	7.16	6.44	7.70	6.92
parabolic_fem	1.80	1.54	2.29	1.95
StocF-1465	1.80	1.70	1.88	1.78
t2em	3.95	3.64	4.45	4.10
Transport	4.18	3.51	5.21	4.56

ズを選択した ILUB-GMRES(m) 法と ILU(0)-GMRES(m) 法を用いた場合の実験結果の比較を示す．ILUB 分解前処理を用いることで反復回数が減少し全テスト行列において性能が改善されている．特に性能が向上した行列は G3_circuit と Transport であり，実行時間はそれぞれ 87%，81%削減された．

6.2.3 マルチコアプロセッサとメニーコアプロセッサでの結果比較

表 7 に Intel Xeon Skylake プロセッサと Intel Xeon Phi プロセッサのそれぞれにおいて，最良のブロックサイズを用いて実行した ILUB-GMRES(m) 法の ILU(0)-GMRES(0) 法に対する求解性能比を示す．全てのテストデータに対し，Intel Xeon Phi プロセッサでは Intel Xeon Skylake プロセッサを上回る性能向上が見られる．これは，ILUB 分解前処理が Intel Xeon Phi プロセッサにおいてより有効であることを示している．

7. おわりに

本研究では，SIMD 演算の活用を前提としたブロック化に基づく fill-in 戦略を用いた ILU 分解前処理（ILUB 分解前処理）を提案し，GMRES(m) 法に適用し，その性能を検証した．行列データベースから得られる 7 つのテスト行列に対し，二つの Intel Xeon Skylake プロセッサからなるノードと，一つの Intel Xeon Phi プロセッサからなるノードを用い数値実験を行った．その結果，全てのテスト行列に対し，提案手法である ILUB 分解前処理は従来手法である ILU(0) 分解前処理と比較し性能向上を実現した．また，ILUB 分解前処理はメニーコアプロセッサである Intel Xeon Phi プロセッサにおいてより大きな効果が得られた．

今後の課題としては，並列化ソルバによる性能評価や，ILUB 分解前処理における適切なブロックサイズの自動調整などが挙げられる．

参考文献

[1] Saad, Y.: *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2nd edition (2003).
[2] Vorst, V. and Henk, A.: *Iterative Krylov Methods for Large Linear Systems*, Cambridge University Press, UK (2003).
[3] Saad, Y.: ILUT: A dual threshold incomplete LU factorization, *Numer. Linear Algebra Appl.*, Vol. 1, pp. 387–402 (1994).
[4] Li, N., Saad, Y. and Chow, E.: Crout Versions of ILU for General Sparse Matrices, *SIAM J. Sci. Comput.*, Vol. 25, pp. 716–728 (2003).
[5] Iwashita, T., Takemura, N., Ida, A. and Nakashima, H.: A New Fill-in Strategy for IC Factorization Preconditioning Considering SIMD Instructions, *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 3, pp. 37–44 (2015).
[6] 杉原正顯，室田一雄：線形計算の数理，岩波書店 (2009).
[7] Davis, T. and Hu, Y.: The university of Florida sparse matrix collection, *ACM Trans. Math. Softw.*, Vol. 38, pp. 1:1–1:25 (online), available from ⟨https://doi.org/10.1145/2049662.2049663⟩ (2011).
[8] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and van der Vorst, H.: *Templates for the solution of linear systems: building blocks for iterative methods*, SIAM, Philadelphia, PA (1994).