Vol.2020-HPC-175 No.18
2020/7/31

SX-Aurora TSUBASA における有限要素解析のための共役勾配法の性能評価

菱沼 利彰[1]　井原 遊[1]　高村 守幸[2]　平野 哲[2]　萩原 孝[3]　岩田 直樹[3]　奥田 洋司[4]

概要：有限要素法を用いた構造解析は工学分野で広く利用されている．一般的に有限要素法を用いたシミュレーションにおいて最も時間がかかるのは疎行列を係数行列としてもつ連立一次方程式の求解である．近年，NEC よりベクトル型のアクセラレータボード (Vector Engine) を搭載した SX-Aurora TSUBASA が登場した．Vector Engine には 1 ボードあたり約 1.2 TB/s の帯域をもつ高速メモリと 8 つの高性能演算コアが搭載されており，各コアには倍精度 32 要素を同時演算できる FMA 演算器が 3 器搭載されている．有限要素解析は PC クラスタをはじめとするスカラプロセッサ・システム向けの高速化研究が進んでいるが，ベクトル型アクセラレータボード上における性能や有効性の評価は行われていない．本研究では有限要素解析ソフトウェアである FrontISTR に含まれる線形ソルバから疎行列の格納形式として BCRS 形式と JAD 形式，線形ソルバとして CG 法を選び，格納形式や実装ごとの OpenMP によるマルチスレッド化，およびコンパイラの自動ベクトル化による高速化効果や性能について評価した．

1. はじめに

有限要素法を用いた構造解析は工学分野で広く利用されている．一般的に有限要素法を用いたシミュレーションにおいて最も時間がかかるのは疎行列を係数行列としてもつ連立一次方程式の求解である．このような問題には Krylov 部分空間法を用いるのが一般的である．

連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ に対する Krylov 部分空間法は，初期近似解 $\boldsymbol{x}_0$ に対応する初期残差ベクトル $\boldsymbol{r}_0 = \boldsymbol{b}$ - $A\boldsymbol{x}_0$ を用いて A のべき乗と $\boldsymbol{r}_0$ の積の像が張る空間から近似解を探索する反復法アルゴリズムの一つのカテゴリで，行列の性質に応じて様々なアルゴリズムがある [1]．行列をベクトルに作用させ，近似解ベクトルの更新を繰り返すことで近似解を得る．

Krylov 部分空間法の高速化のために，GPU などのアクセラレータを用いた高速化の研究が盛んに行われている [2, 3]．一般的に CPU 向けに実装されたプログラムを GPU で動作させる場合，デバイスとの転送や GPU 向けの最適化など施す必要があり，プログラム全体に対して変更が必要になる．

様々な問題に対する離散化手法，連立一次方程式の求解アルゴリズム，結果の可視化などを一貫して提供するシミュレーションソフトウェアはコード量が膨大であり，このようなプログラム全体に対する変更や最適化を行うには膨大な工数が必要になる．

近年, NEC よりベクトル型のアクセラレータボード (Vector Engine, VE) を搭載した SX-Aurora TSUBASA [4, 5] が登場した．VE は 1 ボードあたり倍精度演算時に約 2.1 TFLOPS のピーク性能をもち，約 1.2 TB/s の帯域をもつ高速なメモリが搭載されている．

VE 向けにコンパイルされた一般的な CPU 向けのプログラムは，VH 上で実行することで VEOS とよばれるシステムによってデータやプロセスが自動的に VE 上に展開され，プログラムを書き換えることなく実行できる．ベクトル化はコンパイラによる自動ベクトル化，マルチスレッド化には OpenMP などを利用できる．

有限要素解析は PC クラスタをはじめとするスカラプロセッサ・システム向けの高速化研究が進んでいる．ベクトル計算機上で Krylov 部分空間法を実行した研究事例や大規模な実用ソフトウェアにおける動作検証例はある [6] が，ベクトル型のアクセラレータボード上においてマルチスレッド化や自動ベクトル化によってどの程度の性能が発揮できるかや，どの程度最適化のためにプログラムの書き換えが必要になるかは明らかになっていない．

我々は有限要素法ソフトウェアである FrontISTR [7] を選び，VE 上で有限要素法を用いた構造解析を高速に行うことを目標とした．本論文では研究の最初のステップ

[1] 株式会社科学計算総合研究所
[2] 一社）インダストリスパコン推進センター
[3] 日本電気 (株) AI プラットフォーム事業部
[4] 東京大学大学院新領域創成科学研究科

として，FrontISTR の線形ソルバに含まれる共役勾配法 (Conjugate Gradient method, CG 法) [8] を対象に高速化に必要な最適化方法や疎行列の格納形式を検討し，それらの性能について評価した．

2. SX-Aurora TSUBASA

2.1 SX-Aurora TSUBASA のアーキテクチャ

SX-Aurora TSUBASA は一般的な x86 プロセッサ (Vector Host, VH) をプログラムの制御に用い，PCIe で接続された VE にプログラムやデータをオフロードして計算を行う．

SX-Aurora TSUBASA には VH として用いる CPU のモデルや VE の動作周波数，搭載台数によっていくつかのモデルがあり，今回は Type10-B とよばれるモデルの VE が 4 枚搭載された A300-4 とよばれるモデルを使用した．なお，本論文では VE は 1 枚しか使用しない．

図 1 に今回用いた SX-Aurora TSUBASA の構成を示す．VE は 8 つの演算コアを搭載しており，各コアには型に依らず最大 256 要素を格納するベクトルレジスタが 64 本と，倍精度 32 要素に対して同時に FMA (Fused Multiply and Add) 演算を実行できる FPU が 3 器搭載されている．

ベクトルレジスタに格納，演算できるデータ数は可変で，データ数が 256 要素に満たない場合でもデータを 256 の倍数に揃えずにベクトル計算を実行できる．

ベクトルレジスタ内にデータが 256 要素格納されているとき，FPU はベクトルレジスタのデータに対して 256 / 32 = 8 サイクルかけて計算を行う．そのため VE において倍精度演算を行う場合のピーク性能は次のように計算できる．

1.4 [GHz] × 8 [core]× 32 (要素) × 6 (FMA × 3) = 2.15 [TFLOPS]

メモリ帯域は 1,228 GB/s で，Byte / Flop (B/F) は 1,228 / 2,150 = 0.57 である．コア共有の LLC (Last Level Cache) のサイズは 16 MB で，各コアと LLC 間の帯域は 358.4 GB/s である．

2.2 SX-Aurora TSUBASA のプログラミングモデル

本節では VE を用いたプログラミングの方法について述べる．SX-Aurora TSUBASA では VH 向けに VEOS とよばれる VE 上で動作するプロセスを制御するソフトウェアが提供されている．VEOS は VE からは OS のように見えており，VE に Linux システムコールなどを提供する．I/O やシステムコールなどの処理は自動的に VEOS を通じて VH と協調して行われる．

NEC コンパイラを用いて VE 向けにコンパイルしたプログラムを VH から実行するだけで，VEOS によってプロセスが VE 上に展開されて処理が行われる．そのためユーザがプログラムを変更したり，VE と VH 間の転送を意識する必要はない．

また，SX-Aurora TSUBASA では VH と VE を連携させて計算するハイブリッド計算用の API も提供されている．

コア間の並列には OpenMP や pthread を用いてスレッド並列化を行う．コア内のベクトル化にはコンパイラによる自動ベクトル化，およびコンパイラへの指示句を利用できる．この指示句は一般的なコンパイラでは無視されるため，VH と VE のプログラムは共通化が可能である．

複数の VE を使用したい場合は MPI などを利用することで実現できるが，本論文では VE 1 枚によるシングルノードでの高速化を対象とする．

3. 実装と評価方法

本節では CG 法の核となる BLAS Lv. 1 相当のベクトル演算，および疎行列とベクトルの積 (SpMV) の実装や評価方法について述べる．

FrontISTR はメッシュデータと設定ファイルを入力することで有限要素法による離散化から線形方程式の求解までの一連のシミュレーションを行うソフトウェアである．多くの場合，最も時間のかかる処理は線形方程式の求解であるため，本論文では最初のステップとして FrontISTR に含まれる CG 法のプログラムを VE 上で高速化する．

なお，本論文では内積などの演算をベクトル演算，このときのベクトルの長さ (配列長) をベクトルサイズとよび，SX-Aurora TSUBASA によって 1 命令で複数のデータを同時処理することをベクトル計算，同時処理した数をベクトル長とよんで区別する．

有限要素法による離散化や疎行列の生成は条件分岐などを多く含み，並列化が難しい処理が含まれるため VE 上では性能がでない可能性があるが，本研究では対象外とした．FrontISTR では有限要素法による離散化を行うプログラムと線形方程式の求解を行うプログラムが分離された構造となっているため，VH と VE を連携させて計算するハイブリッド計算用の API を利用することで，離散化を行うプログラムを VH，求解は VE で実行するなどの対応が容易に可能であると考えている．

3.1 ベクトルに対する演算

BLAS Lv. 1 相当のベクトルに対する演算について述べる．現状，FrontISTR におけるベクトルに対する演算は並列化されていない．VE は LLC とコア間の帯域がコアあたり 358.4 GB/s で，HBM2 の帯域速度である 1,228 GB/s の 30 %程度しかないため，HBM2 の帯域を引きだせない．

また，コンパイラの指示句などによるプログラムの変更をせずに，コンパイラの最適化のみで自動ベクトル化がどの程度されるのかは不明である．そこで 4 章で FrontISTR の改良を行う前の事前実験として，OpenMP によるマルチスレッド化を行った内積のプログラムで性能を評価する．

SX-Aurora TSUBASA 向けの最適化は行わない．結果

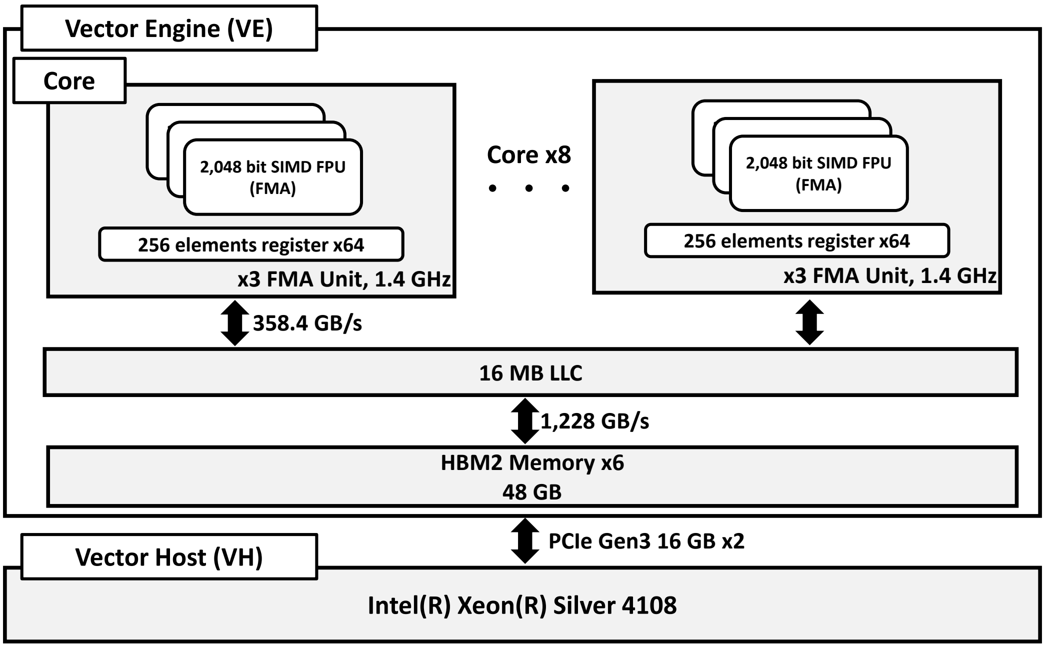

図 1　SX-Aurora TSUBASA (VE: Type-10B) の構成

をスカラ値に足し込む処理は OpenMP の reduction 句を用い，ベクトル化はコンパイラの自動化に任せる．プロファイラから得られるベクトル化率やスレッド数を変化させた際の性能を基に VH と比較した VE の性能を評価する．

3.2　疎行列とベクトルの積

FrontISTR ではいくつかの格納形式が実装されているが，有限要素法によって離散化された行列はサイズが節点あたりの自由度数によって決まるブロック構造になる性質を利用し，あらかじめブロック化された格納形式を用いたり，SpMV のループをブロック化している．疎行列は対角ブロック行列，対角を含まない拡張上三角行列，対角を含まない拡張下三角行列にわけて保持する．

我々はブロック構造に対する最適化はそのまま利用し，ブロック化されたプログラムに対してスレッド並列化やベクトル化を行うことにした．疎行列の格納形式として，FrontISTR のデフォルトとして設定されている Block Compressed Row Storage (BCRS) [9] 形式，およびベクトル計算機向けの Jagged Diagonal (JAD) [9] 形式の 2 つを比較することにした．本節では，それぞれの格納形式の概要と SpMV の実装について述べる．

BCRS 形式はサイズ $r \times c$ の小密行列（ブロック）としてブロック化する．図 2 に 8 × 8 の疎行列を $r = 2$, $c = 2$ の BCRS (BCRS2x2) 形式で保持した構成を示す．_U, _L はそれぞれ上三角，下三角行列に対応する配列を意味する．すべての成分が 0 となるブロックは作成しない．ブロック内では各成分に連続アクセスが可能なため，メモリアクセスの改善効果が期待できる．

一般的にブロックに含まれる零成分による演算量，メモリデータ量の増加が問題となるが，FrontISTR では有限要素法の節点あたりの自由度数に合わせてブロックサイズを決めているため，ほとんど零成分は含まれず，演算量やメモリデータ量もほとんど増加しない．BCRS 形式の SpMV は OpenMP によるスレッド並列化，および MPI によるプロセス並列化が行われているが，本論文では OpenMP によるスレッド並列化のみに着目する．

ブロック上三角行列，ブロック下三角行列はそれぞれ一般的な BCRS 形式と同じように値を保持する配列と，値に対するインデックス配列 2 本からなる 3 本の配列で構成される．生成されるブロックの数が the number of blocks (blk)，行列の行数を N，ブロックのサイズは正方形 ($r = c$) としたとき，それぞれ次のような配列である．なお，対角ブロックはインデックス配列を作る必要はない．

(1) ブロック内の成分の値を格納する長さ $blk \times r \times c$ の倍精度型 1 次元配列 VAL
(2) 配列 VAL に格納されたブロックの開始列番号を格納する長さ blk の整数型 1 次元配列 INDEX
(3) 各ブロック行の開始位置が配列 INDEX のどの成分から開始しているかを格納する長さ $((N - r)/r) + 1$ の整数型 1 次元配列 PTR

BCRS2x2 形式の疎行列に対する SpMV のプログラムを図 3 に示す．FrontISTR において実際には，ブロック行内ではアクセスする $\boldsymbol{y}$ の位置は変わらないため，r 本の $\boldsymbol{y}$ のワークベクトルを作成し，ブロック行の最後で対応する $\boldsymbol{y}$ にストアすることでインデックス計算を削減している．

プログラムから BCRS 形式の SpMV は次のような特徴をもつことがわかる．

- INDEX を用いたベクトル $\boldsymbol{x}$ への間接参照が発生．

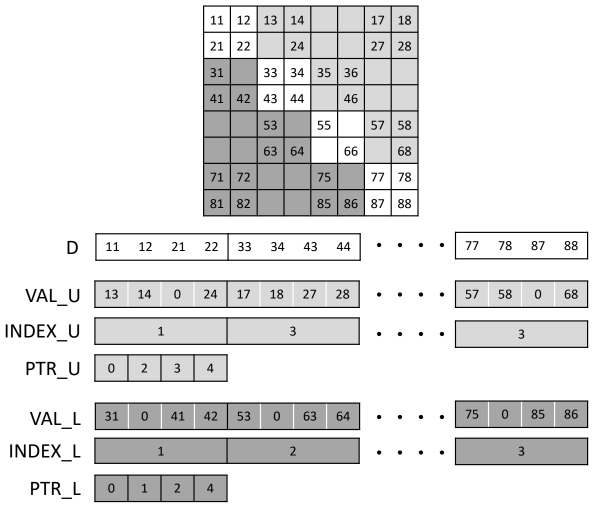

図 2 BCRS2x2 形式の構成

```
for(bi=0;bi<block_row;bi++){

    //diag. block matrix
    y[r*bi+0] += D[bi+0] * x[bi+0];
    y[r*bi+1] += D[bi+1] * x[bi+0];
    y[r*bi+0] += D[bi+2] * x[bi+1];
    y[r*bi+1] += D[bi+3] * x[bi+1];

    //Upper triangular block matrix
    for(bc=PTR_U[bi];bc<PTR_U[bi+1];bc++){
        bj   = INDEX_U[bc] * bnc;
        k = r * c * j;
        y[r*bi+0] += VAL_U[k+0] * x[bj+0];
        y[r*bi+1] += VAL_U[k+1] * x[bj+0];
        y[r*bi+0] += VAL_U[k+2] * x[bj+1];
        y[r*bi+1] += VAL_U[k+3] * x[bj+1];
    }

    //Lower block triangular matrix
    ...
}
```

図 3 BCRS2x2 形式の SpMV のプログラム

- ブロック行ごとの依存性はないため bi ループにおけるマルチスレッド化が可能.
- ブロック行内はベクトル命令で連続した処理が可能.

ブロック内やブロック行内 (bc のループ) ではベクトル化が期待できるが，ブロック数は空間数によって決まることから高いベクトル化効率は期待できない.

そこでベクトル計算機向けに FrontISTR に実装されている JAD 形式に着目した．JAD 形式は非零成分の個数が多い順に行の並べ替えを行い，行列を列方向に格納し直す格納形式である．図 4 に図 2 と同じ行列を JAD 形式で保持した場合の構成を示す.

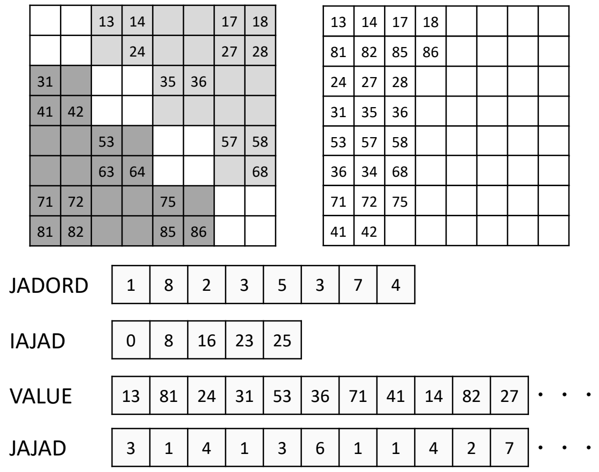

図 4 対角ブロック行列以外を JAD 形式で保持した場合の構成

FrontISTR では JAD 形式を対角ブロック行列とそれ以外にわけて実装されている．メモリレイアウト上ではブロック化されておらず，SpMV のループ内でブロックの成分をすべて計算するようにアンロールを行うことでブロック構造を利用する．なお，対角ブロックは BCRS 形式と同様にインデックス配列を作らない.

JAD 形式は次に示す 4 本の配列から構成される．行列の行数を N, 非零成分の数を $ndnnz$, 各行での非零成分数の最大値を nz としたとき，それぞれ次のような配列である.

(1) 並び替える前の行番号を格納する長さ N の整数型 1 次元配列 JADOLD
(2) 並び替え後の疎行列の各列の成分の開始位置を格納する長さ $nz+1$ の整数型 1 次元配列 IAJAD
(3) 非零成分の値を格納する長さ $ndnnz$ の倍精度 1 次元配列 VALUE
(4) 非零成分の列番号を格納する長さ $ndnnz$ の整数型 1 次元配列 JAJAD

有限要素法によって 2x2 のブロック行列が得られる場合の JAD 形式の疎行列に対する SpMV のプログラムを図 5 に示す．プログラムから JAD 形式の SpMV は次のような特徴をもつことがわかる.

- ベクトル $\boldsymbol{x}$ に対する JAJAD を用いた間接参照が発生 (図 5 の (*) 部).
- ベクトル $\boldsymbol{y}$ 用のワークベクトル $w1$, $w2$ に対する IA-JAD を用いた間接参照が発生 (図 5 の (**) 部).
- 並び替えを行ったことで計算結果を JADORD を用いて並び替えるためのワークベクトルがブロック列数だけ必要.
- 計算の最後に JADORD を用いてワークベクトルの結果を $\boldsymbol{y}$ に足し込む処理が必要.
- 対角ブロックや列内の計算は依存性がなく並列化可能.

JAD の SpMV を行う場合にメモリからロード，ストアされるデータ量について考える．ここで，間接参照が行われる場合はキャッシュによってデータを使い回せず，必ずメモリからデータをロードすると仮定する．非対角ブロッ

```
//diag. block matrix
for(ii=0;ii<N;ii++){
    int k = r*c*ii;
    y[k+0] += D[k+0] * x[k+0];
    y[k+0] += D[k+1] * x[k+1];
    y[k+1] += D[k+2] * x[k+0];
    y[k+1] += D[k+3] * x[k+1];
}

// Lower and Upper triangular block matrix
for(j=0;j<NZ;j++){
    for(i=IAJAD(j);i<IAJAD(j+1)-1;i++){
        int ixx = i - IAJAD(j) + 1; // (*)
        int k = r * c * i;
        //(**)
        w1[ixx] += VALUE[k+0]*x[JAJAD[k]*r+0]
        w1[ixx] += VALUE[k+1]*x[JAJAD[k]*r+1]
        w2[ixx] += VALUE[k+2]*x[JAJAD[k]*r+0]
        w2[ixx] += VALUE[k+3]*x[JAJAD[k]*r+1]
    }
}

for(ii=0;ii<N;ii++){
    y[r*ii+0] += w1[JADORD[ii]];
    y[r*ii+1] += w2[JADORD[ii]];
}
```

図 5 ブロック数 2x2 における JAD 形式の SpMV

ク行列の非零成分数を $ndnnz$，有限要素法の空間数を r，インデックス配列は 4 byte の整数型とする．以下，データ量の評価におけるデータの単位は byte である．

対角ブロックの計算に必要なメモリデータ量は長さ $r \times N$ の対角ブロックと長さ N のベクトル $\boldsymbol{x}$, $\boldsymbol{y}$ へのアクセスが必要なため，8 (倍精度) $\times\ r \times N + 2N$.

非対角ブロックに対する計算は長さ $ndnnz$ の VALUE に対するロード，r 本のワークベクトル ($w1$, $w2$) と x に対するインデックス配列を用いた間接参照が必要である．長さ $nz+1$ の IAJAD は他の配列と比べて十分に小さいため無視できると考えると，行列のアクセスに $8 \times ndnnz$，ベクトルのアクセスに $8 \times ((r+1) \times ndnnz)$，長さ $ndnnz$ の JAJAD のアクセスに $4 \times ndnnz$ が必要である．

最後にベクトル $\boldsymbol{y}$ への r 本のワークベクトルの結果の足し込みを行うために必要なデータ量は $8 \times (r+1)$ で，合計で $((r+1+1)N) + (8ndnnz + 4ndnnz + 8((r+1) \times ndnnz) + ((r+1)N)$ のデータアクセスが必要である．

多くの場合，BCRS 形式の SpMV において最内側ループで計算される非零成分数と比べ，JAD 形式の SpMV において最内側ループで計算される列の成分数は多くなるため，JAD 形式の方が BCRS 形式よりも最内側でのベクトル長を長くとれることが期待できる．

一方で JAD 形式の SpMV は $\boldsymbol{y}$ へのストアも間接参照になるため，マルチスレッド化を行う場合は書き込みが競合し，スレッドセーフにならないという問題が生じる．

我々はスレッド並列のために次の 2 つの実装方法を考えた．それぞれ “inner”，および “outer” とよぶ．

inner 内側ループ (図 5 の i loop) に対して OpenMP によるスレッド並列化を行う．

outer スレッド本数分のワークベクトルを用意し，外側ループ (図 5 の j loop) に対して OpenMP によるスレッド並列化を行う．各スレッドは計算結果をそれぞれのワークベクトルに足し込み，ループの最後に同期をとってから総和をとる．

inner 方式はワークベクトルを必要としないが，内側ループのスレッドあたりの処理量が減少するため，ベクトル長が十分に確保できない可能性がある．

outer 方式はループ内ではベクトル長を長くとれるため効率よく並列化できるが，ワークベクトルの初期化や総和などが必要になり，演算量，メモリデータ量が増加する．

4 章でこれら JAD 形式の 2 つの実装方法，および BCRS 形式を比較する．

表 1 VH 諸元

CPU	Intel Xeon Silver 4108@1.80 GHz 8 core × 2
Peak (DP)	460.8 GFLOPS × 2
Memory	96 GB (127.8 GB/s)
OS	CentOS 7.6
Compiler	gcc 4.8.5

表 2 VE 諸元

VE	Type 10B (1.4 GHz, 8 core)
Peak (DP)	2.15 TFlops
Memory	48 GB (1,228 GB/s)
LLC	16 MB
C Compiler	ncc 2.5.1
Fortran Compiler	nfort 2.5.1
Profiler	ftrace 10.11

4. 性能評価

4.1 実験環境

実験に用いる VH の環境を表 1 に，VE の環境を表 2 に示す．

コンパイルオプションとして，比較対象として用いる VH には最適化を行う “-O3”，OpenMP を有効化する “-fopenmp” をつけ，VE にはこれらに加えて自動ベクトル化を行う “-mvector” をつけた．

性能評価には NEC 製のプロファイラである ftrace を用いた．プロファイラによって得られるベクトル長はベクトルレジスタに収まるデータ量である 256 を最大値として，ベクトル計算したときの平均の長さを意味する．

表 3 実験に用いる疎行列

	N	nnz	nnz/N
cube(10)	3,993	268,119	67.1
cube(50)	397,953	30,986,559	77.9
cube(100)	3,090,903	245,438,109	79.4
hinge	252,168	19,043,712	75.5
Gear16	1,859,214	154,479,996	83.1

比較対象として用いる VH は 8 コアの Intel 製 CPU を 2 器搭載しており，ハイパースレッディングを有効にしているため，スレッド数は最大 32 である．

4.2 対象とする問題

行列サイズを任意に変更できる有限要素法のメッシュデータとして，一辺 1.0 m の立方体に対する線形弾性解析のデータに対する離散化を行った．このときヤング率 206.0 GPa，ポアソン比 0.3，密度 7,874 kg/m^3 として，z 方向下向きに重力加速度 9.8 m/s^2 における自重相当の力を付与した．立方体の各辺の分割数はすべて同一に $nx = ny = nz$ とし，この値を変更することで様々な行列サイズに対する評価を行うことにした．分割数を $n \times n \times n$ としたときのこの問題を “cube(n)” とよぶ．

より実際的な例題として，2 種類のメッシュデータを用意した．それぞれ “hinge”, “Gear16” とよぶ．cube(10), hinge, Gear16 のメッシュデータ，および疎行列の非零パターンを付録に載せた．

これらのメッシュデータから有限要素法を用いて 5 種類の行列を作成した．これらは有限要素法により 3x3 のブロック構造の疎行列が生成される問題である．疎行列の行数を N, 非零成分数を nnz としたとき，それぞれの疎行列のサイズを表 3 に示す．

4.3 ベクトル演算の性能

はじめに，メモリ帯域および自動ベクトル化，マルチスレッド化による性能への影響を確認するため，BLAS Lv. 1 に含まれる倍精度内積演算 (ddot) の性能を評価した．スレッドの立ち上げから計算の終了までを計測した．実行時間として 100 回の測定時間の平均を用いた．

倍精度のベクトルに対する内積演算が要求する B/F は 8 × 2 / 2 = 8 で，VE のハードウェアの B/F は 0.57 であるため，データがキャッシュに収まらない場合の性能はメモリ性能に制約を受けることが予想される．

内積演算は結果をスカラ値に足し込む必要がある．コンパイラがコンパイル時に出力するレポートを確認し，ループがベクトル化されていること，足し込みがベクトルレジスタのリダクションとして生成されていることを確認した．

図 6 に VE のスレッド数を 8 に固定し，ベクトルサイズを 10^3 から 10^8 まで変化させたときの性能を示す．内積演算における計算性能はメモリ性能に制約を受けることが予想されるため，性能指標には 2 本のベクトルのサイズと時間から求めた帯域性能を用いた．2 本の倍精度ベクトルのデータはサイズ 10^6 まで LLC に収まる．

ベクトルサイズの増大に伴い性能が増加し，ベクトルサイズ 10^8 では約 1,051 GB/s となった．これは VH のメモリ帯域である 1,228 GB/s の約 85 %で，メモリ帯域を十分に引きだせている．このとき，プロファイラによって取得した平均ベクトル長は 256 で，ハードウェア最大値である 256 に対して 100 %のベクトル化率がでている．

一方でベクトルサイズが小さいときは性能が低い．例えばベクトルのデータがすべて LLC に収まるサイズ 10^5 では約 170 GB/s の性能しかでていない．プロファイラによって取得したベクトル化率は 99.6 %で十分な値が得られており，ベクトル化率だけでは性能は予測できない．

試行ごとにスレッドを立ち上げず，最初にスレッドを立ち上げて 100 試行を行ったところ性能が改善したことから，これはスレッドの立ち上げやコアにスレッドを割り当るためのオーバヘッドが大きいためと考えられる．なお，これは十分にベクトルサイズが大きければ問題にならない．

VH と比べた VE の性能はすべてのデータサイズで高く，ベクトルサイズが小さい 10^5 の場合で VH の約 2.4 倍，ベクトルサイズが大きい 10^8 の場合で約 9.0 倍である．

図 7 にベクトルサイズを 10^8 に固定し，VE のスレッド数を 1 から 8 まで変化させたときの性能を示す．

4 スレッドまでは性能はスレッド数の増加に従って良好にスケールするが，5 スレッド以上では約 1,050 GB/s にとどまった．これは VE は 1 コアあたりの LLC との帯域が 358.4 GB/s であることから，メモリバンド幅である 1,228 GB/s を引きだすためには 1,228 / 358.4 = 3.4 コア相当の帯域が必要になるためである．

1 スレッドにおけるメモリ転送の性能は約 270 GB/s で，コアあたりの LLC との帯域に対し約 75 %の効率がでており，性能はデータ転送速度に制約を受けていると考えられる．なお，計算量を $2N$ として計算した演算性能は 30 GFLOPS で，コアあたりのピーク性能は 268.75 GFLOPS である．このとき，VE の 1 スレッドにおける性能は VH の 32 スレッドと比べて約 2.3 倍で，VE は 1 スレッドでも VH より性能が高い．

ベクトル演算の結果から次のようなことがわかった．

- ベクトル演算においてメモリ帯域を引きだすためには 4 スレッド以上で並列化する必要がある．
- 小さいベクトルサイズではスレッドの立ち上げなどが問題となり性能がでにくい．
- 小さいベクトルサイズや 1 スレッドの場合でも VE は VH よりも性能が高い．
- 最適化のためのコンパイラへの指示句などを入れなくてもコンパイラによる自動ベクトル化と OpenMP で十分な性能が引きだせる．

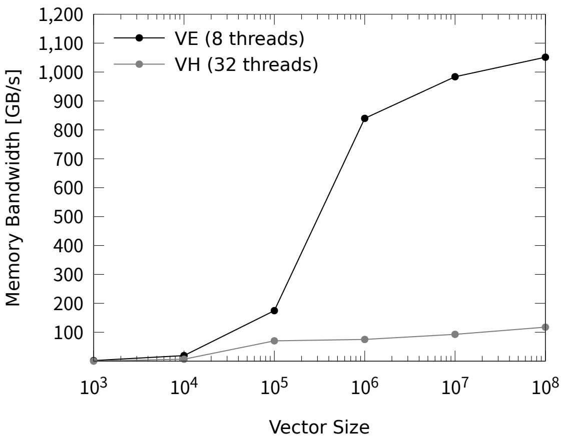

図 6 内積演算におけるメモリアクセスの影響

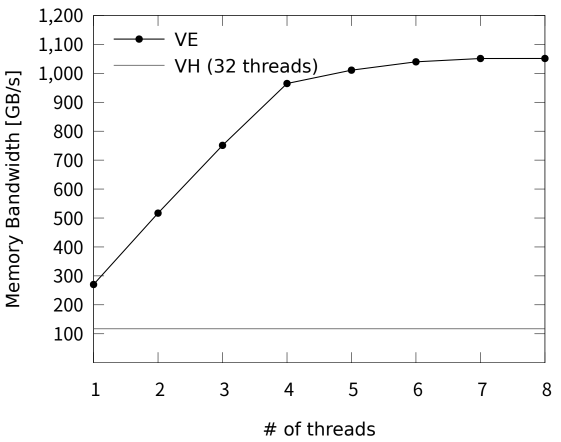

図 7 内積演算におけるマルチスレッド化の効果 (ベクトルサイズ 10^8, VH はすべて 32 スレッド)

4.4 SpMV の性能

4.4.1 各実装方式の性能

表 4 に，BCRS3x3 形式，および JAD 形式の inner 方式，outer 方式の SpMV を 100 試行したときの平均時間とプロファイラから取得した平均ベクトル長を示す．なお，VH における JAD 形式の SpMV はすべてのケースで JAD 形式の性能が BCRS 形式の性能よりも 20 から 40 %程度低かったため，VH では BCRS 形式のみを用いた．

BCRS 形式の SpMV は 8 スレッドにおいてすべてのケースで VH より 2.5 倍から 3.1 倍高速だが，平均ベクトル長が 10 から 13 と短い．プログラムではブロックをまたいだベクトル化は考慮しておらず，一方で対角ブロック行列の処理はベクトル化できるため，ブロックサイズである 9 よりもやや長い値になったと考えられる．

マルチスレッド化によるスピードアップはどの行列でも 7 倍以上と高い並列化効果を得られているが，JAD 形式と比べて計算時間がかかる．これはベクトル化率が低いために VE の計算性能が引きだせておらず，計算性能がメモリ性能の制約を受けないためマルチスレッド化の効果が高く見えているためである．このことから，BCRS 形式は VE に不向きであることが確認できた．

JAD 形式における inner 方式は 8 スレッドの実行時間が他の実装方式と比べて最も高速である．平均ベクトル長は問題サイズの小さい cube(10) の 8 スレッド以外では 250 以上で，列方向にスレッド並列化を行っても十分なベクトル長がとれることがわかった．

cube(10) では平均ベクトル長が 137.5 と低い．これは cube(10) の行数が 3,993 しかないため，8 スレッドで分割すると最大で 1 列あたり 500 成分程度しか連続で計算できないためである．

outer 方式を用いれば cube(10) のような小さいサイズの行列においても 254.8 と長いベクトル長をとることができるが，計算時間は inner 方式と比べてどのケースにおいても長い．これはスレッド本数分のワークベクトルを初期化，総和する必要があるためである．

試験的に答えは合わなくなるが outer 方式においてワークベクトルを用いず，同期せずに並列計算を行うと計算性能が 20 から 30 %向上した．大きい行列では inner 方式でも十分にベクトル長を確保できていることや，スレッド本数分のワークベクトルに対する処理の影響が大きいことから，inner 方式の実装がよいことがわかった．

4.4.2 JAD 形式の inner 方式と outer 方式によるマルチスレッド化の効果

図 8 に SpMV のスレッド数を 1 から 8 まで変化させ，1 スレッドを基準に規格化した並列化の効果を示す．すべての問題サイズにおいてマルチスレッド化の効果は 4 スレッドまでは得られるが，5 スレッド以上ではメモリ性能に制約を受けて並列化の効果が得られていないことがわかる．

cube(10) における SpMV の実行時間は 5 スレッド以上では増大した．inner 方式では 4 スレッドにおける平均ベクトル長が 165.5 であるのに対し，8 スレッドでは 137.5 で，スレッド数を増やすことで平均ベクトル長が短くなり，各スレッドが同時に非連続な領域にアクセスすることでメモリへの非連続なデータ要求が増加したために性能が低下したと考えられる．

outer 方式では，5 スレッド以上ではコアが増えたことによるメモリ帯域の増加の恩恵が受けられず，スレッドが増えたことでワークベクトルに対する処理だけが増加したために計算時間が増大した．4 スレッドにおける inner 方式の計算時間は outer 方式より約 15 %多く，ベクトル長が十分にとれないサイズの小さい行列では outer 方式の 4 スレッドが最も高速である．

cube(50), cube(100) に対する inner 方式の SpMV では，8 スレッドでも平均ベクトル長が 250 以上で，スレッド数を 5 スレッド以上にしても計算時間は増大せず，すべてのスレッド数で outer 方式よりも高速である．問題サイズが十分に大きく，ベクトル長が十分にとれる場合は inner 方式がよいことがわかった．

これらの実験により，比較的大きいサイズの行列であれ

表 4 SpMV の実行時間 [ミリ秒] (平均ベクトル長)

		cube(10)	cube(50)	cube(100)	hinge	Gear16
VH, CRS, 32threads		3.4	26.9	202.0	31.5	165.0
VE, BCRS3x3	1 thread	8.3 (10.7)	77.2 (12.5)	632.4 (12.7)	83.2 (12.1)	441.2 (11.2)
	8 threads	1.1 (10.7)	10.8 (12.5)	87.1 (12.7)	9.3 (12.1)	57.3 (11.2)
VE, JAD	1 thread	0.6 (252.6)	4.2 (255.8)	38.2 (256.0)	3.7 (255.1)	19.9 (256.0)
	8 threads (inner)	0.5 (137.5)	1.2 (254.3)	11.2 (255.7)	1.1 (253.1)	7.5 (256.0)
	8 threads (outer)	0.5 (254.8)	1.7 (255.7)	15.4 (256.0)	1.5 (255.2)	8.3 (256.0)

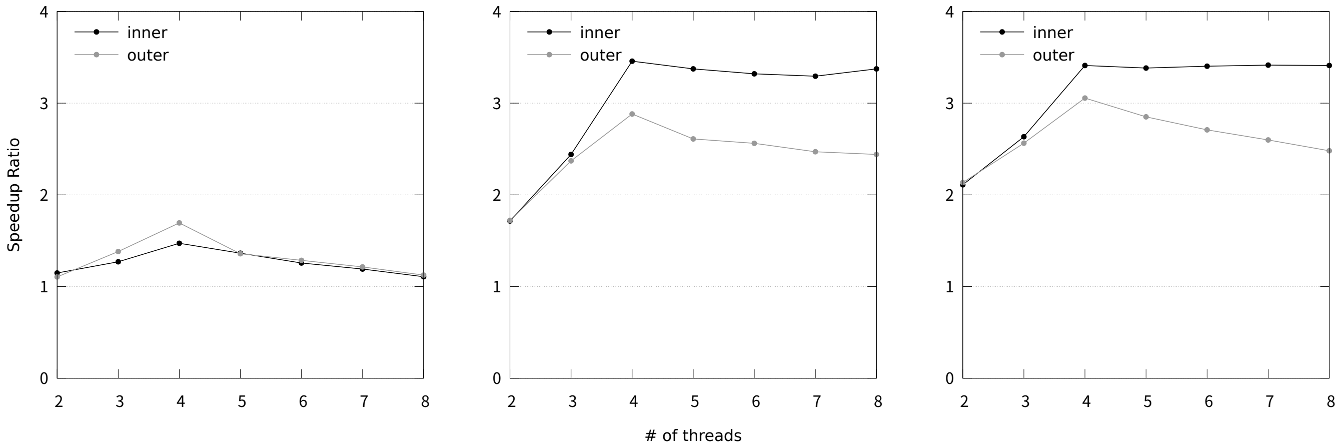

図 8 SpMV のマルチスレッド化の効果 (1 スレッドとの比, VE, JAD 形式)

表 5 CG 法 1000 反復の実行時間 [秒]

		cube(10)	cube(50)	cube(100)	hinge	Gear16
VH, CRS, 32 threads		3.9	38.2	245.1	31.2	203.4
VE, BCRS	1 thread	10.1	109.3	868.2	93.5	720.6
	8 threads	1.6	15.1	116.3	12.6	96.5
VE, JAD	1 thread	4.5	7.7	53.3	6.2	49.2
	8 threads (inner)	0.8	1.7	14.1	1.5	11.7

ばベクトル長を長くとれるため inner 方式が有効であることがわかった．また，3.1 節で求めたメモリからロード，ストアするデータ量を用いて計算時のデータ転送性能を計算すると約 778 GB/s で，間接参照を必要とする SpMV においてもメモリ帯域に対して高い転送性能がでており，inner 方式がよいことがわかった．

4.5 共役勾配法の性能

最後に，これらを共役勾配法に組み込み，1000 反復を行った結果を表 5 に示す．BCRS 形式，JAD 形式のどちらにおいても 8 スレッド並列を行えば VH と比べて高速である．BCRS 形式は VH と比べて約 2.1 倍から 2.5 倍高速である．JAD 形式は cube(10) では約 4.9 倍，cube(10) 以外では約 17.3 倍から 22.4 倍高速である．

今回の実装では SX-Aurora TSUBASA のための特殊な最適化は行っておらず，コンパイラへの NEC 独自の指示句を入れるなどのプログラムの変更も行っていない．ベクトル長を十分に長くとることのできる JAD 形式のような格納形式に対して OpenMP によるマルチスレッド化を行えば高い実行効率をだすことができ，CG 法の反復計算を効率よく高速に実行できることがわかった．

5. まとめ

本研究では SX-Aurora TSUBASA の VE 上で高速な有限要素法解析を行うため，有限要素解析ソフトウェアの FrontISTR に含まれる CG 法の高速化を行い，性能を評価した．

SX-Aurora TSUBASA は VEOS によって VE 向けにコンパイルされたプログラムを VH 上から実行することで，自動的にデータやプログラムが VE にオフロードされ，プログラムを変更することなく実行できる．

FrontISTR の動作自体はコンパイラおよびコンパイルオプションの切り替えのみで実現できたが，高速化のために FrontISTR に含まれるベクトル演算と SpMV のプログラムの並列化を考えた．

内積演算の実験の結果から次のようなことがわかった．
- コンパイラによる自動ベクトル化は内積のような足し込みが必要な演算においても効率的に動作する．
- コアと LLC 間の帯域が LLC とメモリの帯域より狭いため，帯域性能を引きだすためには 4 スレッド以上の

並列化が必要.

この結果から，すべてのベクトル演算および SpMV に対し OpenMP によるスレッド並列化を行うことにした.

次に FrontISTR に含まれる BCRS 形式と JAD 形式の SpMV の比較を行った．FrontISTR は JAD 形式の SpMV はスレッド並列化されていなかったため，スレッド並列化の実装手法として内側ループで並列化を行う inner 方式と外側ループで並列化を行う outer 方式を検討した．inner 形式は列方向にスレッド並列を行うため，スレッド数を増やすと連続して計算できる数が少なくなる．outer 方式では書き込みの競合を避けるためにスレッド本数分のワークベクトルを確保し，最後に足し込む必要がある.

実験から，BCRS 形式の SpMV は VH よりすべてのケースで 2.5 倍から 3.1 倍高速だが，JAD 形式と比べすべての問題で遅かった．BCRS 形式の SpMV はブロック内でしかベクトル化できないため，ベクトル長が 12 から 13 程度しかとれず，VE には不向きであることがわかった.

cube(50) や hinge などのある程度サイズが大きい行列に対する JAD 形式の SpMV の性能から次のようなことがわかった.

- inner 方式でも列方向に十分なベクトル長が確保でき，ワークベクトルを必要とする outer 方式と比べ，ほとんどの問題で inner 方式が高速.
- cube(50) における inner 方式のメモリ転送性能は約 778 GB/s で，高いメモリ性能が引きだせている.
- inner 方式はスレッド数を増加させると 5 スレッド以上ではメモリ性能に制約を受けて性能が向上しない.
- outer 方式は 5 スレッド以上ではメモリ性能に制約を受けて並列化による高速化の恩恵を得られず，ワークベクトルが増えるため計算時間が増加する.

一方，cube(10) のような行数が 3,993 しかなくサイズが小さい問題では，inner 方式の SpMV はスレッド数の増加に従ってベクトル長が長くとれなくなり，8 スレッドより 4 スレッドが高速だった．さらに 4 スレッドの outer 方式の SpMV は 4 スレッドの inner 方式の SpMV と比べ約 15 %高速で，サイズが小さい問題では inner 方式のスレッド数を減らすか，outer 方式を用いれば性能を向上できる.

これらを共役勾配法に組み込んだ結果，次のような結果が得られた.

- BCRS 形式，JAD 形式のどちらにおいても 8 スレッドでは VH と比べて高速.
- BCRS 形式は VH と比べて約 2.1 倍から 2.5 倍高速.
- JAD 形式の inner 方式は小さい問題では VH の約 4.9 倍，それ以外の問題では約 17.3 倍から 22.4 倍高速.

VE は BCRS 形式のようなベクトル長を長くとれない格納形式でも VH より高速で，ベクトル長が長くとれる JAD 形式を用いることで更に高速化が可能であることがわかった.

これらのことから実用的な問題サイズであれば OpenMP によるマルチスレッド化やベクトル長を長くとれる JAD 形式を用いることで，VE 向けの特殊な最適化を必要とせず，CG 法で十分に高い性能を発揮できることがわかった.

今後の課題として，今回は FrontISTR の実装に従って JAD 形式をループ内でブロック化したが，メモリレイアウトとしてブロック化を行うことが挙げられる．JAD 形式の SpMV がメモリ性能に制約を受けることから，ブロック化することでインデックス配列のデータサイズを削減すれば高速化が可能と考えられる.

また，inner 形式と outer 形式のハイブリッドな実装も考えられる．JAD 形式では疎行列の構造によっては長さの短い列が発生するが，長さの短い列を判定して outer 形式を用いたり，スレッド数を少なくすることで更に性能が向上できると考えられる.

謝辞　本論文を執筆するにあたり，筑波大学助教 森田直樹氏，および科学計算総合研究所 研究員の五十嵐亮氏には有益なコメントを頂きました．また，本研究の一部は東京大学と NEC による共同研究「並列有限要素法のベクトルプロセッサ向け高速化に関する研究」として実施されました．ここに記して謝意を表します.

参考文献

[1] 櫻井鉄也，松尾宇泰，片桐孝洋, “数値線形代数の数理と HPC,” pp. 34–35, 共立出版, 2018.
[2] I. Kiss, S. Gyimothy, Z. Badics, and J. Pavo, “Parallel realization of the element-by-element fem technique by cuda,” *IEEE Transactions on magnetics*, vol. 48, no. 2, pp. 507–510, 2012.
[3] 大島聡史, 林雅江, 片桐孝洋, 中島研吾, “三次元有限要素法アプリケーションの CUDA 向け実装と性能評価,” **情報処理学会研究報告**, vol. 2011, no. 20, pp. 1–6, 2011.
[4] Y. Yamada and S. Momose, “Vector engine processor of nec's brand-new supercomputer sx-aurora tsubasa,” in *Proceedings of A Symposium on High Performance Chips, Hot Chips*, vol. 30, pp. 19–21, 2018.
[5] NEC, “NEC SX-Aurora TSUBASA Documentation.” `https://www.hpc.nec/documents/`, (参照 2020-06-22).
[6] H. Okuda, K. Nakajima, M. Iizuka, L. Chen, and H. Nakamura, “Parallel finite element analysis platform for the earth simulator: Geofem,” in *International Conference on Computational Science*, pp. 773–780, Springer, 2003.
[7] 一般社団法人 FrontISTR Commons, “FrontISTR, オープンソース大規模並列 FEM 非線形構造解析プログラム.” `https://www.frontistr.com/`, (参照 2020-04-21).
[8] Y. Saad, “Iterative Methods for Sparse Linear Systems,” pp. 151–243, SIAM, 2003.
[9] R. Barrett, M. W. Berry, T. F. Chan, J. W. Demmel, J. Donato, J. J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, “Templates for the solution of linear systems: building blocks for iterative methods,” pp. 55–65, SIAM, 1994.

付　録

実験で用いたメッシュデータ，および離散化によって得られる疎行列の非零パターンを図 A·1, A·2, A·3 に示す.

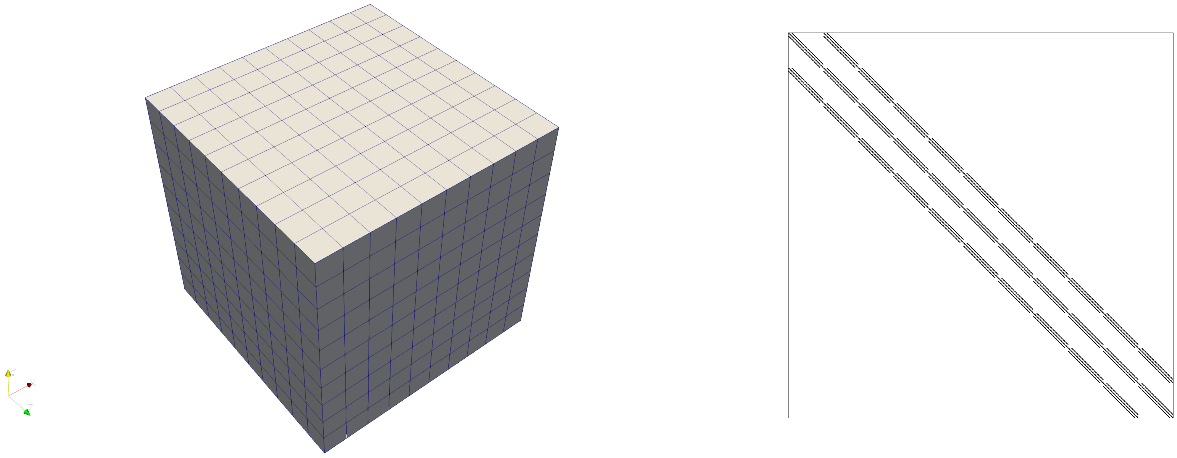

図 A·1 cube(10) のメッシュデータ (左) と離散化によって得られる疎行列の非零パターン (右)

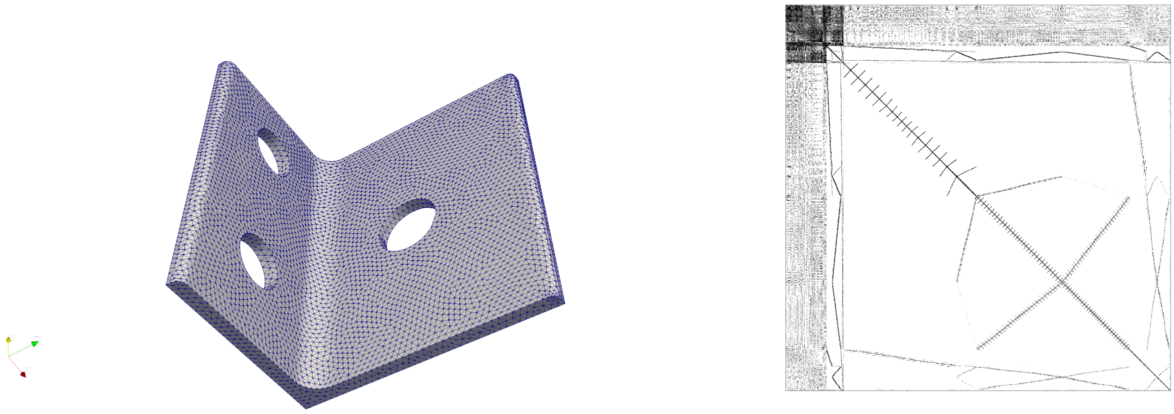

図 A·2 hinge のメッシュデータ (左) と離散化によって得られる疎行列の非零パターン (右)

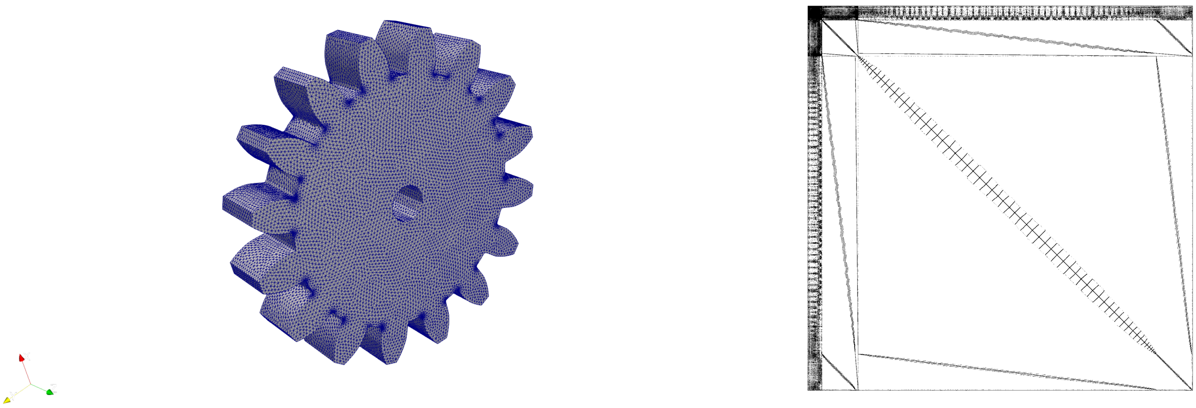

図 A·3 Gear16 のメッシュデータ (左) と離散化によって得られる疎行列の非零パターン (右)