2.4 LexADV-AutoMT および AutoLinAS ライブラリの性能評価

諏訪東京理科大学　　河合浩志

2.4.1 はじめに

本研究では、著者を含む研究グループ ADVENTURE において現在あらたに開発中の LexADV フレームワークについて、そのコンポーネントである AutoMT および AutoLinAS ライブラリの各種 HPC プラットフォームにおける性能ベンチマークを行った。

エクサスケールコンピュータ等の次世代の並列計算機アーキテクチャにおいて、大規模な数値計算データ処理を必要とする実アプリケーションが高い演算効率を得るためには、プ$ロセッサやメモリなどハードウェアが持つ階層構造を考慮したプログラミングモデルを採用することが必要である。それに対し、ADVENTURE プロジェクトにおいてこれまで開発されてきた HDDM（Hierarchical Domain Decomposition Method、階層型領域分割法）の技術を応用した、大規模数値計算データ処理ライブラリの研究開発が現在、JST CREST からの支援をうけた HDDMPPS プロジェクト（プロジェクトリーダー：東洋大、塩谷隆二教授）において行われている。具体的には、「連続体力学向け DSL」、「DDM ソルバーライブラリ」、「DDM 入出力ライブラリ」、そして「連続体力学系シミュレータ」の 4 つの研究項目における研究開発が進められている。ここでは、大規模な数値計算データに対し各種 HPC アーキテクチャに応じてマルチレベルの多階層の領域分割を行うこと、またこれに基づく多階層計算格子データの生成, 操作および I/O をつかさどるライブラリを用意する。そしてこれらをもとに、大規模分散並列計算で実績のある構造・熱・流体・電磁場など複数の FEM（Finite Element Method, 有限要素法）解析ソフトウェアを備えているオープンソース CAE ソフトウェア ADVENTURE システムを作り直すことを最終的に目指している。

本性能ベンチマークにおいてとりあげる AutoMT（「おてもと」）および AutoLinAS（「おとりなし」）についてであるが、これらはそれぞれ、連続体力学分野向け DSL（Domain Specific Language：特定問題分野向けに特化した計算機言語）、および、HDDM 並列ソルバーライブラリにおける、HPC アーキテクチャごとの差を吸収するための低レベルインターフェイスとなっている。これらの部分における性能特性は、LexADV フレームワークや将来の ADVENTURE システムのパフォーマンスを決定する重要ポイントになると思われる。

2.4.2 有限要素解析コードのＨＰＣ環境向け実装

有限要素法に基づく構造解析ソルバーは対象となる人工物・構造物の機能や応答、安全性、健全性などを評価するためのツールであり、CAE 技術の最重要コンポーネントとして今やモノづくりにとって不可欠なものとなっている。産業界において現在、例えば自動車

の騒音・振動解析や衝突解析、金型のバーチャルマニュファクチャリング、航空機や船舶の構造健全性評価、建物やプラントの耐震解析など、多種多様なモノづくり・設計シーンでの CAE 技術の普及が進むと同時に、シミュレーション自体のさらなる高精度化とモデルの大規模化が求められている。

大規模構造解析のニーズにはさまざまなものがあるが、一例として、原子力プラントの耐震解析を挙げる。本事例では、鉄筋コンクリート壁、配管、圧力容器、燃料集合体などすべてがソリッド要素で表現されている。解析規模はプラント全体と付近の地盤を合わせて数百億自由度と見積もられ、原子炉圧力容器単体でも数十億自由度を要する。解析は動弾塑性解析であり、数千から数万ステップの計算が必要となる。

有限要素法を用いた構造解析の計算手法や解析の分類としては、線形弾性解析、弾塑性解析、振動解析、熱応力解析、衝撃解析、塑性加工解析など多岐にわたる。ここではこれらを、HPC 技術との関連という見地から、連立一次方程式を解かないもの（陽解法系）と、連立一次方程式を解くもの（陰解法系）の２つに分類する。前者の場合、各有限要素の単位で得られる情報だけを用いた行列ベクトル積が主要な演算パターンとなり、計算時間の大半は要素剛性行列評価や応力積分などの要素単位での計算処理に割かれる。一方、後者の場合、まず各要素の要素剛性行列を用いて、全体剛性行列と呼ばれる比較的大サイズの疎行列を組み上げる。非定常問題における陰解法ソルバーの場合、これに関する連立一次方程式を時間ステップあるいは非線形ステップごとに解くことになる。なおこれは、静解析、固有値解析含め、解析ソルバー内に連立一次方程式の解法が組み込まれているもの全般についてもあてはまる。数千万から数億自由度クラスの大規模解析の場合には、もっぱら反復型解法の系統が用いられる。その際、大規模な連立一次方程式を解くための超並列アーキテクチャ向けアルゴリズムに関して、並列反復法ソルバーおよび領域分割法の二つが考えられる。

並列反復法ソルバーでは、対称正定値であれば CG 法、非対称であれば Bi-CGSTAB あるいは GMRES 法などにより、全体剛性行列に関する連立一次方程式を解く。構造解析では一部の非線形問題を除き、対称正定値行列を扱うことが多い。一方、構造物の大型化あるいは軽量化により、梁や板状の薄い形状を扱うことも多く、この場合係数行列の条件数は悪化する。したがって、強力な前処理が必要とされることが多い。

並列反復法ソルバーでは、行列ベクトル積が繰り返される。有限要素法の場合、これは三種類の方法により実装することができる。まず、係数行列である全体剛性行列を要素剛性行列から組み上げ、その非ゼロ成分をメモリ上に保存する、非ゼロソルバー。次に、係数行列の全体化は行わず、要素剛性行列の形でメモリ上に保存する、EBE ソルバー。最後に、要素剛性行列すらメモリ上に保存せず、行列ベクトル積が必要とされる都度に評価する EBE-MSF(Matrix Storage-Free)ソルバー。以下、係数行列を対称とし、CG ソルバーにおいてこの３種類のソルバー実装の性能見積もりを行う。なお、行列ベクトル積部分の並列化は比較的容易であり、通信パターンは隣接間が主なものとなる。

非ゼロソルバーの場合、全体剛性行列とベクトルとの積であることから、メモリアクセスは主に行列成分のみとなる。行列の 1 成分をメモリから読み出すごとに、対称性を考慮して 4 個の浮動小数点演算が必要となる。ただし、行列は疎でしかも非構造格子からのものであるため、メモリアクセスは不規則でベクトル化の効率は低く、しばしば節点単位でのレジスタブロッキングが不可欠となる。またギャザー・スキャッタ機構は性能向上に役立つ。

次に EBE ソルバーの場合、メモリ上に記憶された個々の要素剛性行列とベクトルとの積結果を組み上げていくことになる。要素剛性行列は密で対称であるため、非ゼロソルバーと同様、行列の 1 成分ごとに 4 個の浮動小数点演算が必要となる。EBE ソルバーは非ゼロソルバーより記憶量、演算量とも多いが、メモリアクセスパターンは比較的規則的で性能が出しやすい。

最後に、EBE-MSF ソルバーの場合、これは陽解法ソルバーの場合と同様、EBE 演算がメインとなる。B/F 値の低い計算機でも有効である。もし B/F 値が極端に低い場合、本ソルバーは主にメッシュデータのみをメモリから読み出すため省メモリであり、しかも他のソルバーよりメモリアクセス量が少なくなるために結果的に最速ソルバーとなる可能性がある。

続いて、領域分割法について述べる。先に述べた線形代数レベルでの並列化を行った並列反復法ソルバーと異なり、領域分割法（Domain Decomposition Method: DDM）は偏微分方程式レベルでの並列化手法である。まず全体領域を複数の部分領域に分割する。各部分領域に適当な境界条件を与えて独立に解き、領域間境界条件が落ち着いてくるまで反復的にこれを繰り返す。したがってこの方法は反復解法の一種である。構造問題においては、並列反復法ソルバーの場合と同様、強力な前処理が不可欠となる。DDM の場合、FETI や BDD などのマルチグリッド型前処理が多用されている。

DDM の主要な演算パターンは部分領域ごとの有限要素解析（ローカルソルバー）である。DDM の並列化はローカルソルバーを単位として、その並列化は容易であり、通信パターンは隣接間がメインとなる。

部分領域ローカルソルバー実装にはいくつかの方法が考えられる。まず一つ目は、部分領域ごとの剛性行列について、これをまず直接法ソルバーを用いて行列分解し、スカイライン形式などでメモリ上に保存しておく。DDM 反復ごとに前進後退代入を行う。2 つ目は、部分領域ごとに反復法ソルバーを用いて部分領域問題を解く。このとき、メモリ上にはほとんどデータを記憶しない。3 つ目は、部分領域ごとに対応する密行列をあらかじめ作成しておき、これらとベクトルとの積を繰り返す。これは並列反復法ソルバーにおける EBE ソルバーのように、ローカル Schur 補元行列と呼ばれる、部分領域ごとの剛性行列を静的縮約により陽に導出してメモリ上に保存しておくものである。

最初の直接法ソルバーアプローチでは、行列分解されたデータに対する前進後退代入演算が主である。この場合、行列の 1 成分を読み出すごとに 2 個の浮動小数点演算が必要と

なる。

2 つ目の反復法ソルバーアプローチでは、メモリからメッシュ情報を読んでキャッシュ上で反復法ソルバーを起動する。大量の演算が L2 あるいはラストレベルキャッシュ上で行われることになる。並列反復法ソルバーで説明した非ゼロ、EBE、あるいは EBE-MSF ソルバーどれも利用可能である。なお、前処理の導入により反復回数を削減したい場合、前処理行列をあらかじめ作成してメモリ上にストアしておくことも考えられる。

最後のローカル Schur 補元行列保存の場合、EBE ソルバーと同様、対称な密行列とベクトルの積であり、1 成分ごとに 4 浮動小数点演算を要する。なお、部分領域ごとに静的縮約によりローカル Schur 補元行列を作成する際に、行列行列積を中心とした大量の浮動小数点演算が行われる。

2.4.3 連続体力学分野向け DSL のための行列、テンソル操作ライブラリ：AutoMT

上記の例でみられたように、有限要素法にもとづくシミュレーションでは要素を単位として手法が組み上げられているため、その計算パターンにおいても頻繁に要素単位の計算が求められる。ここでとりあげられる AutoMT（「おてもと」）ライブラリは、シミュレーションコードにおいて大量に実行される有限要素単位での小規模演算、特に小規模行列およびベクトル、あるいは二次元／三次元テンソルに関する演算パターンを高速化する。本ベンチマークでは、AutoMT ライブラリにおける小規模行列およびテンソル演算の性能ベンチマークテストを行った。

2.4.4 低レベル線形代数ソルバーライブラリ：AutoLinAS

一方、LexADV フレームワークにおける HDDM ライブラリは大規模な全体システムレベルでの連立一次方程式を並列環境において階層型領域分割法を用いて解くためのものである。前述したように、ここには部分領域ごとのローカルソルバーが含まれ、この部分の性能特性は領域分割法コードの性能を全般的に支配するものとなる。そこで、部分領域ローカルソルバーにおいて必要となる、シングルコアあるいはシングル計算ノード・ノード内 OpenMP 並列に対応する中規模連立一次方程式の解法を高速化するための低レベルインターフェイスが新たに求められる。AutoLinAS（「おとりなし」）はこのための線形代数ソルバーライブラリとなっている。これは、直接法としてスカイラインソルバー、反復法として CG ソルバーについて、それぞれプラットフォームごとに性能最適化された実装を有する。本ベンチマークでは、そのうち主に CG ソルバーにおける計算カーネルとなる、非ゼロ成分より構成されるスパース行列とベクトルとの積（SpMV）の性能ベンチマークテストを行った。

連続体力学向けドメイン特化言語AutoMT
（FX10でのベンチマーク）

河合浩志（諏訪東京理科大）

背景

- ***より複雑な物理現象、より高度な数値スキーム***
  ➔ 要素開発作業がたいへんに
  - 連成解析、大歪弾塑性、異方性シェル、破壊、乱流
  - コンシステント接線剛性、歪仮定要素、メッシュフリー、均質化法

- ***HPC向けプロセッサの高速化***
  - 数値計算向け命令セット拡張
    - SIMD命令（マルチメディア拡張命令）
      - SSE, AVX (インテル/ AMD), HPC-ACE (富士通), NEON (ARM)
  - メニーコアやGPGPUなど

大量のテンソル式
（または行列・ベクトル演算）の嵐　！！！

連続体力学向けDSL

- ドメイン特化言語　DSL（Domain Specific Language)
  - 問題分野向け専用言語
  - 特定の問題分野（数学、量子力学、図形処理、経済学など）向けに特化したプログラミング言語
    - Matlab, Mathematica
    - 各アプリケーションが有するマクロ・スクリプト
  - 問題分野の語彙を直接用いる
    - システムの高レベル設計に有効
    - ユーザーとの対話や要求仕様設定に有効
    - **トランスレータによりFortran、C、Javaなどコードを自動生成**
- 計算力学分野、特に連続体力学(Continuum Mechanics)向けに特化した計算機言語？
  - 行列・ベクトル演算
  - テンソル演算

連続体力学DSL：　数式からコード自動生成

テンソル形式で記述

$$\mathbf{C}^{ep} = 2GQ\frac{\boldsymbol{\sigma}^{tr'}}{\sigma_e^{tr}} \otimes \frac{\boldsymbol{\sigma}^{tr'}}{\sigma_e^{tr}} + 2GR\mathbf{I} + (K - \frac{2}{3}GR)\mathbf{I} \otimes \mathbf{I}$$

Fortranコード
（ライブラリサブルーチンをコール）

```
real*8 G, K, Q, R, sigma_tr_e
real*8 st_sigma_tr_prime(6), st_I(6)
real*8 mnst4_I(6,6), mnst4_II(6,6), mnst4_C_ep(6,6)

call automt_otimes_st_st_mnst4
   (st_sigma_tr_prime, st_sigma_tr_prime,
mnst4_tmp1)
call automt_prod_s_mnst4_mnst4
   (2.0d0 * G * Q / sigma_tr_e, mnst4_tmp1,
mnst4_tmp1)

call automt_prod_s_mnst4_mnst4
    (2.0d0 * G * R, mnst4_I, mnst4_tmp2)
call automt_add_mnst4_mnst4_mnst4
     (mnst4_tmp1, mnst4_tmp2, mnst4_tmp1)

call automt_prod_s_mnst4_mnst4
   (K – 2.0d0 / 3.0d0 * G * R, mnst4_II, mnst4_tmp2)
call automt_add_mnst4_mnst4_mnst4
   (mnst4_tmp1, mnst4_tmp2, mnst4_C_ep)
```

行列・ベクトル形式で記述

$$[D^{ep}] = [D^e] - \frac{3G}{\sigma_e^{\,2}(1 + \frac{h}{3G})}\{\sigma'\}\{\sigma'\}^T$$

C言語コード
（ライブラリ関数をコール）

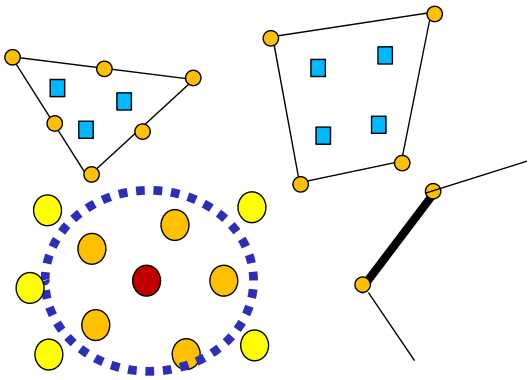
```
double G, h, sigma_e;
double v6_sigma_prime[6];
double m6x6_D_e[6][6], m6x6_D_ep[6][6];

automt_prod_v6_v6tp_m6x6
     (v6_sigma_prime, v6_sigma_prime, m6x6_tmp1);
s_tmp1 = (3.0 * G) / (sigma_e*sigma_e* (1.0 + h / (3.0 * G)));
automt_prod_s_m6x6_m6x6
     (s_tmp1, m6x6_tmp1, m6x6_tmp1);

automt_sub_m6x6_m6x6_m6x6
     (m6x6_D_e, m6x6_tmp1, m6x6_D_ep);
```

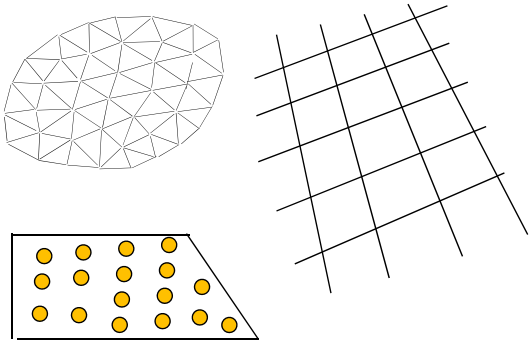
連続体力学シミュレーション分野向けDSL
解析プログラム・コードの特性

要素・セル・粒子系： ***物理モデル***



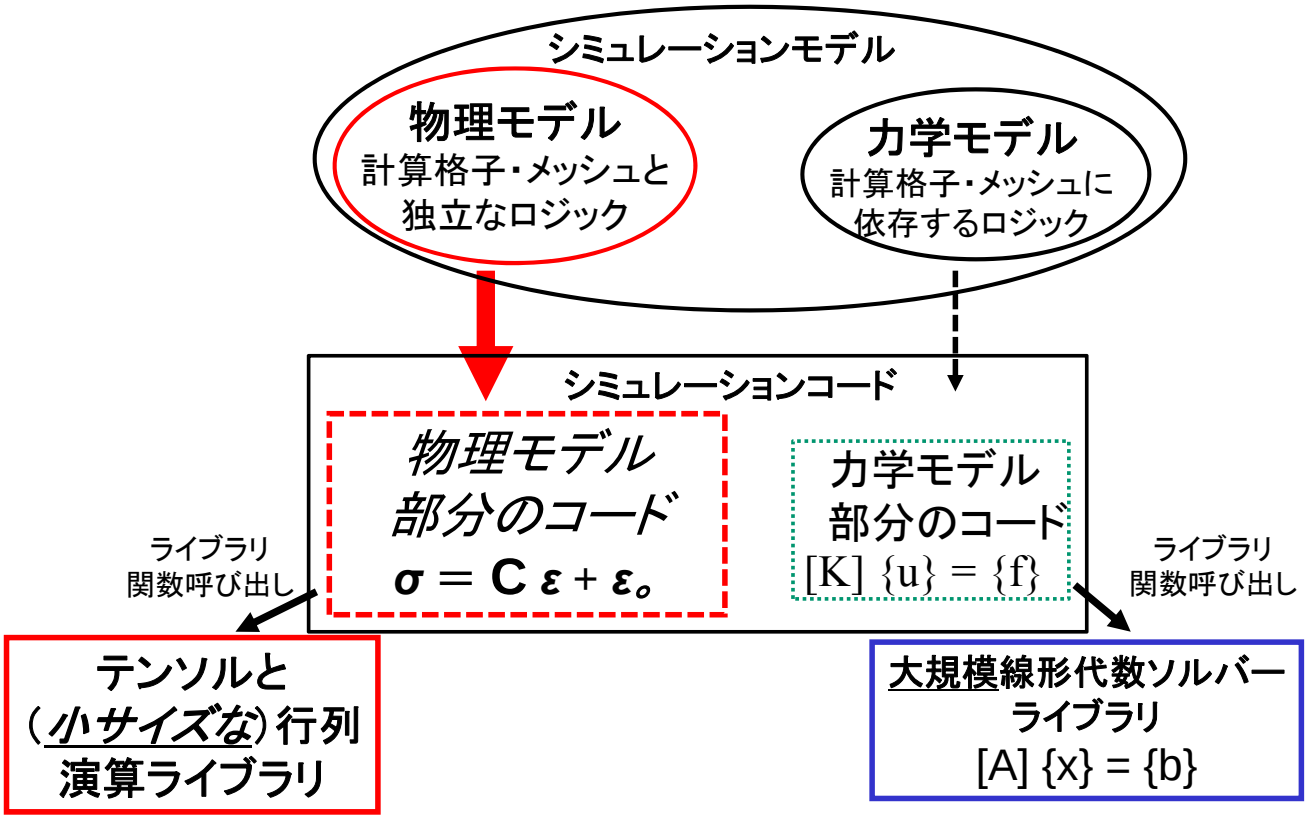
- 物理現象の詳細を記述
- ***テンソル***または***小規模行列***
- 点上（積分点、セル中心）で評価
  - ➔ 点ごとに独立・並列で
  - ➔ キャッシュ上で
- 大量の数式
  - ➔ <u>コードの大部分を占める</u>

全体システム系： ***力学モデル***

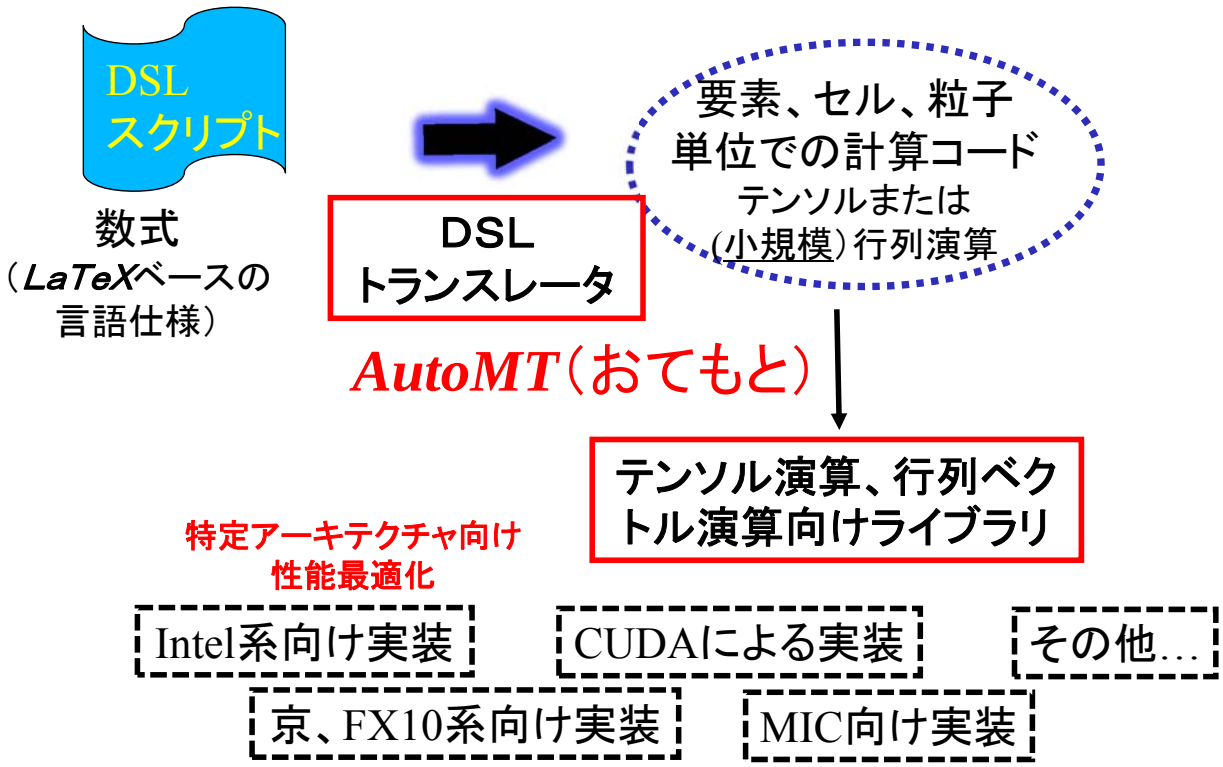


- 近似手法（FEM、FVM、SPH…
- 行列・ベクトル式または差分式
- 隣接関係を記述
  - ➔ 大規模な行列
  - ➔ ステンシル演算
- 数式の数自体は少ない
  - ➔ 線形代数ソルバー

典型的なシミュレーションコードの構造



連続体力学向けDSLシステム構成



（例） 超弾性neo-Hookean材

$$\hat{\mathbf{C}} = 2\mu III_C^{-1/3}\left[\frac{1}{3}I_C\mathbf{I} - \frac{1}{3}I \otimes C^{-1} - \frac{1}{3}C^{-1} \otimes I + \frac{1}{9}I_C C^{-1} \otimes C^{-1}\right]$$

LaTeXコード

```
{¥bf ¥hat{C}} =2 ¥mu III_C^{-1/3}
[1/3 I_C {¥bf I} - 1/3 I ¥otimes C^{-1} - 1/3 C^{-1} ¥otimes I + 1/9 I_C C^{-1} ¥otimes C^{-1}]
```

Cコード

```
tmp0 = 2.000000 * mu;
tmp1 = 1.000000 / 3.000000;
tmp2 = -tmp1;
tmp3 = pow (III_C, tmp2);
tmp4 = tmp0 * tmp3;
tmp5 = 1.000000 / 3.000000;
tmp6 = tmp5 * I_C;
AutoMT_prod_s_t4_t4 (tmp6, T4_I, tmp7);
tmp8 = 1.000000 / 3.000000;
AutoMT_prod_s_t_t (tmp8, T_I, tmp9);
AutoMT_inverse_t_t (T_C, tmp10);
AutoMT_otimes_t_t_t4 (tmp9, tmp10, tmp11);
AutoMT_sub_t4_t4_t4 (tmp7, tmp11, tmp12);
tmp13 = 1.000000 / 3.000000;
AutoMT_inverse_t_t (T_C, tmp14);
AutoMT_prod_s_t_t (tmp13, tmp14, tmp15);
AutoMT_otimes_t_t_t4 (tmp15, T_I, tmp16);
AutoMT_sub_t4_t4_t4 (tmp12, tmp16, tmp17);
tmp18 = 1.000000 / 9.000000;
tmp19 = tmp18 * I_C;
AutoMT_inverse_t_t (T_C, tmp20);
AutoMT_prod_s_t_t (tmp19, tmp20, tmp21);
AutoMT_inverse_t_t (T_C, tmp22);
AutoMT_otimes_t_t_t4 (tmp21, tmp22, tmp23);
AutoMT_add_t4_t4_t4 (tmp17, tmp23, tmp24);
AutoMT_prod_s_t4_t4 (tmp4, tmp24, T4_C_hat);
```

１行の数式（２行のLaTeXコード）から２６行のCコードに変換

テンソル演算および
行列ベクトル演算のためのライブラリ
AutoMT「おてもと」in ADVENTURE Auto

いまより楽にコードを書きたい

行列・テンソル演算を多用する
プログラミング作業の効率化・省労力化

ま、できれば、コードが速いに越したことはない

現在および将来のプロセッサ上で
高速に動作するようなライブラリ実装・コード生成

テンソル演算からコードへの
マッピング

ベクトルテンソル積、ベクトル同士のドット積

a ***X*・*b*** ⟶ automt_prod, automt_cdot

```
double a[3];
double X[3][3];
double b[3];
double s;
double tmp[3];
```

a
X
b
s = ***a X*・*b***

```
automt_prod_t_v_v (a, X, tmp);
```

入力 : a, X
出力 : tmp

```
automt cdot_v_v_s (tmp, b, s);
```

入力 : tmp, b
出力 : s

マッピング例

演算からFortranサブルーチンコール（C関数）へのマッピング

$\boldsymbol{X} + \boldsymbol{Y}$	automt_add_t_t_t
$\boldsymbol{X}\,\boldsymbol{a}$	automt_prod_t_v_v
$\boldsymbol{X}\,\boldsymbol{Y}$	automt_prod_t_t_t
$\boldsymbol{X} : \boldsymbol{Y}$	automt_colon_t_t_s
$\boldsymbol{X} \cdot\cdot \boldsymbol{Y}$	automt_dotdot_t_t_s
$\det \boldsymbol{X}$	automt_det_t_s
$\boldsymbol{X}^{-1}$	automt_inv_t_t

サポートされている演算 (1)

ベクトル演算

<単項演算>

$|\boldsymbol{a}|$

<二項演算>

$\boldsymbol{a} + \boldsymbol{b}$　　$\boldsymbol{a} - \boldsymbol{b}$　　$s\,\boldsymbol{a}$

$\boldsymbol{a} \cdot \boldsymbol{b}$　　$\boldsymbol{a} \times \boldsymbol{b}$　　$\boldsymbol{a} \otimes \boldsymbol{b}$

<3項演算>

$[\boldsymbol{a}\ \boldsymbol{b}\ \boldsymbol{c}]$

$\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$: ベクトル
s, t : スカラー

サポートされている演算 (2)

2階テンソル演算

<単項演算>

det $\boldsymbol{X}$　　tr $\boldsymbol{X}$

$\boldsymbol{X}^{T}$　$\boldsymbol{X}^{-1}$　$\boldsymbol{X}^{-T}$

<複雑な演算>

対称部・非対称部への分解

各種不変量

固有値、固有ベクトル

座標変換

$\boldsymbol{X}^{n}$ (n:実数)

極分解

<二項演算>

$\boldsymbol{X}+\boldsymbol{Y}$　$\boldsymbol{X}-\boldsymbol{Y}$　$s\boldsymbol{X}$

$\boldsymbol{X}\boldsymbol{Y}$　$\boldsymbol{X}:\boldsymbol{Y}$　$\boldsymbol{X}\cdot\cdot\boldsymbol{Y}$

$\boldsymbol{X}\boldsymbol{a}$　$\boldsymbol{a}\boldsymbol{X}$

$\boldsymbol{X}$,$\boldsymbol{Y}$: 2階テンソル
$\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$: ベクトル
s, t : スカラー

サポートされている演算 (3)

4階テンソル

<単項演算>

A^{-1}

<他の演算>

座標変換

行列形式への変換

<定数>

O　　I　　$\boldsymbol{I}\otimes\boldsymbol{I}$

<2項演算>

A + B　A - B　s A

A : $\boldsymbol{X}$　$\boldsymbol{X}$: A

A, B, C: 4階テンソル
$\boldsymbol{X}$,$\boldsymbol{Y}$: 2階テンソル
$\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$: ベクトル
s, t : スカラー

対称性

X a → automt_prod_st_v_v

```
real*8 X(6)
real*8 a(3)
real*8 b(3)
```

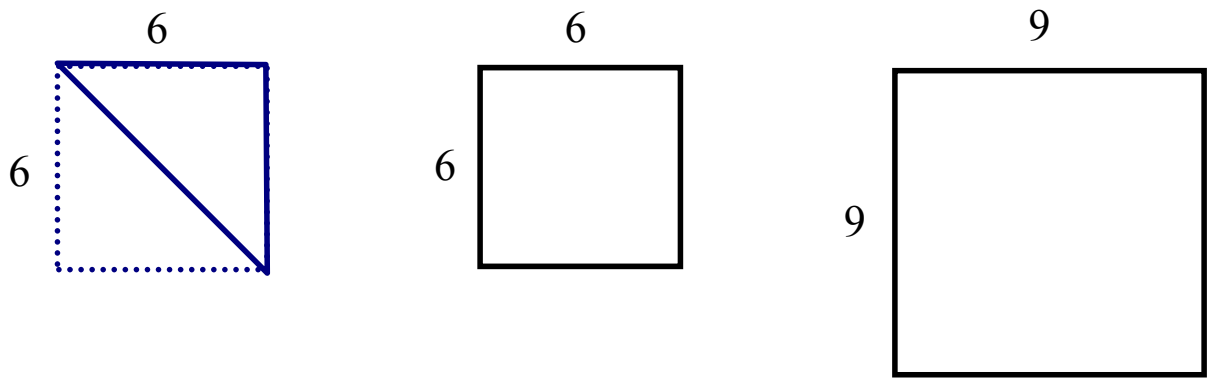
X (2階の対称テンソル)
a
b = ***X a***

```
X(1)=X11
X(2)=X22
X(3)=X33
X(4)=X12
X(5)=X23
X(6)=X31

call automt_prod_st_v_v (X, a, b)
```

入力 : X, a
出力: b

4階テンソルの対称性



Minor symmetry 6x6 Major symmetry upper triangle only (21成分)	Minor symmetry 6x6 No-major symmetry (36成分)	No-minor symmetry 9x9 No-major symmetry (81成分)
`real*8 C(21)`	`real*8 C(6,6)`	`real*8 C(3,3,3,3)`

転置

転置演算は、実際には浮動小数点演算を行わない

→ 他の演算と組み合わせる

$\boldsymbol{X}^T\boldsymbol{a}$ automt_prod_ttp_v_v

$\boldsymbol{X}\boldsymbol{Y}^T$ automt_prod_t_ttp_v

（例） 弾塑性、J2則、等方硬化

$$\mathbf{C}^{ep} = 2GQ\frac{\boldsymbol{\sigma}^{tr'}}{\sigma_e^{tr}} \otimes \frac{\boldsymbol{\sigma}^{tr'}}{\sigma_e^{tr}} + 2GR\mathbf{I} + (K - \frac{2}{3}GR)\mathbf{I} \otimes \mathbf{I}$$

```
real*8 G, K, Q, R, sigma_tr_e
real*8 st_sigma_tr_prime(6), st_I(6)
real*8 mnst4_I(6,6), mnst4_II(6,6), mnst4_C_ep(6,6)

call automt_otimes_st_st_mnst4
   (st_sigma_tr_prime, st_sigma_tr_prime, mnst4_tmp1)
call automt_prod_s_mnst4_mnst4
   (2.0d0 * G * Q / sigma_tr_e, mnst4_tmp1, mnst4_tmp1)

call automt_prod_s_mnst4_mnst4 (2.0d0 * G * R, mnst4_I, mnst4_tmp2)
call automt_add_mnst4_mnst4_mnst4 (mnst4_tmp1, mnst4_tmp2, mnst4_tmp1)

call automt_prod_s_mnst4_mnst4
   (K – 2.0d0 / 3.0d0 * G * R, mnst4_II, mnst4_tmp2)
call automt_add_mnst4_mnst4_mnst4 (mnst4_tmp1, mnst4_tmp2, mnst4_C_ep)
```

問題提起

- スパコン： ペタからエクサへ
  - 分散メモリ並列で数千～数万ノード ➔ 今後も増える？
    - 電力制約 （発電所が必要？？？）
  - （では、）計算ノード(~=PC,WS)あたりの性能が向上？
- PC、WS： ギガからテラへ
  - 現在の状況
    - Intel Core i7 3.4 GHz、6コア 3.4 x 8 x 6 = 163 GFLOPS
    - Intel Xeon Phi (Knights Corner) 、60コア 1.0+ TFLOPS
    - GeForce GTX Titan (Kepler) 1.0+ TFLOPS

      ➔ 10X ～ 100X の性能向上？？？
  - クロック周波数は伸び悩み （1～3GHz）
  - メモリバンド幅ボトルネック

問題提起 (2)

- 目の前のマシンでの性能乖離
  - 誰かさんのコードは10 T FLOPSで動作
    - 自分のコードでは1 G FLOPSしか。。。
      - *1万倍の格差！！！*
  - 何が違う？
    - アクセラレータ利用
    - 専用言語(CUDA)、ディレクティブ（OpenMP、OpenACC)、ライブラリ
    - 並列化 ＋ α
      - キャッシュ、SIMD命令拡張、ソフトウェアパイプライン、。。。
- 何が問題なのか？
  - 並列化する （マルチコア、メニーコア）
  - キャッシュに載せる （メモリバンド幅制約からの解放）
  - アーキテクチャをうまく使う （理論ピーク性能を目指す）

コード最適化

HPC業界における重要課題

(実アプリにおける)ピーク性能比向上

(ノード内)単体性能 ＋ ノード間並列化

マルチコア／メニーコア化により、
相対的にメモリが遅くなっているので… (Memory Wall)
メモリの壁

キャッシュに載せる！

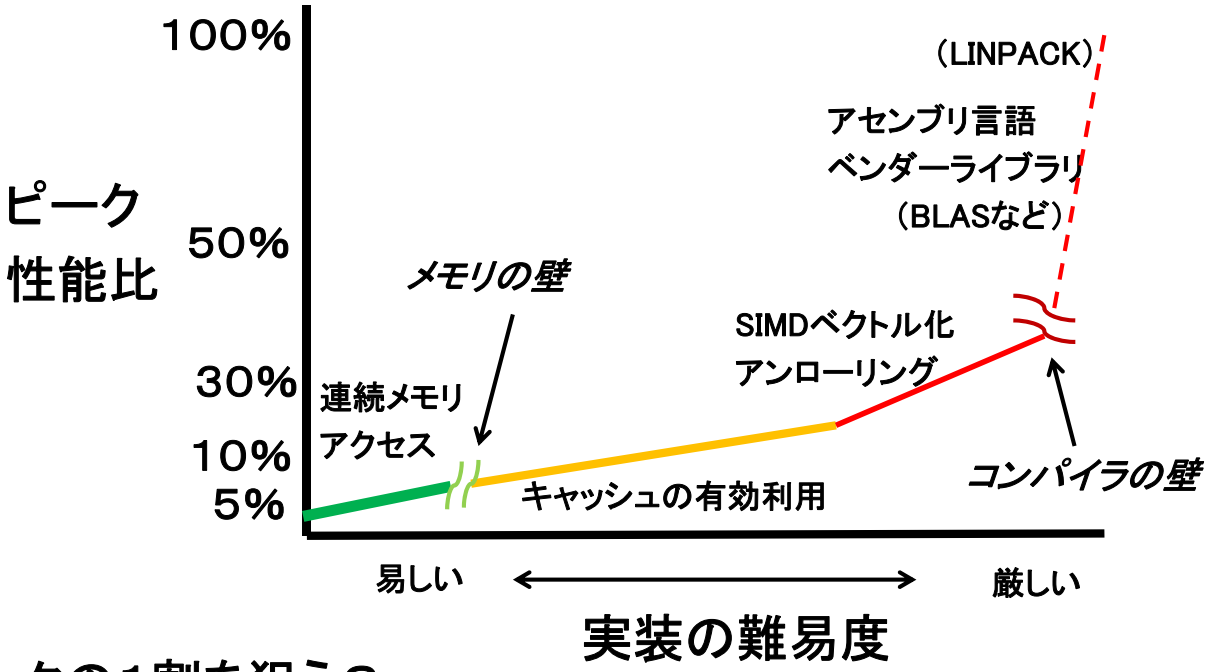
↓

キャッシュに載っていても…

キャッシュ上で性能が出ない！！！

Compiler Wall ???
コンパイラの壁

単体性能の向上： 最新スカラー機事情



ピークの1割を狙う？

- IF キャッシュに載る ➡ 並列性能の確保＋命令実行効率の向上
- ELSE (メモリバンド幅ネック) ➡ 並列性能の確保 (だけしか。。。)

オンキャッシュなのに性能がでない？

キャッシュに載っていても…

キャッシュ上で性能が出ない！！！

~~Compiler Wall ???~~

正確には…

ILP (Instruction-Level Parallelism) Wall

命令レベルの並列性に問題あり

1クロックに何個の演算を並列実行できるか？

	スーパースカラー	積和命令（FMA）	SIMD命令	計
京(富士通SPARC64 VIIIfx)	2	2	2	8
IBM Power7	2	2	2	8
Sandy Bridge	2	1	4	8
Haswell	2	2	4	16
Knights Corner	2	1	8	16

1クロックに複数命令同時実行するには？

- スーパースカラ、FMA： 積と和の同時実行
  - 積の数と、和（差）の数のバランスが必要
  - 除算、ルート、sin、logなどは対象外
- SIMD： 複数データに対する同一種類演算の一括処理
  - 演算パターンの規則性が重要
    - 配列とループ ： *ショートベクトル化*
    - ループボディ内の大量の演算からペアリング（UXSIMD）
- メモリアクセス
  - （ほとんど）レジスタに載っていることが重要
  - 配列へのシーケンシャルアクセスが望ましい

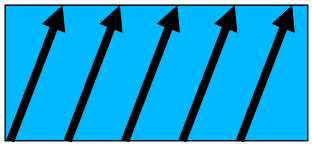
*NINJA*ギャップ(by Intel)の発生

さて、実際のコードは？

- １重ループ （ループ長が長い）
  - ボディが小さい
    数行程度

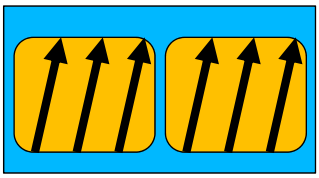
× キャッシュ 悪
命令数 少

  - ボディが大きい
    数十～数百行



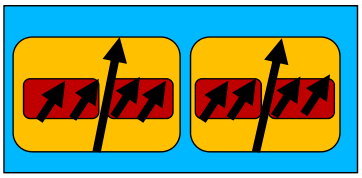
◎ キャッシュ 良
レジスタ 良
命令数 多

- 多重ループ （ボディが小さい）
  - 内側ループ長が長い
    数十～数百



○ キャッシュ 良
レジスタ 良
命令数 増やせる

  - 内側ループ長が短い
    １０以下



△ キャッシュ 良
レジスタ 悪
命令数 増やせない

コード最適化の例 （行列ベクトル積）
ー 従来型のスカラー向け ー

通常のコード

```
double  a[3][3];
double  b[3];
double  c[3];

for (i = 0; i < 3; i++) {
  c[i] = 0.0;
  for (j = 0; j < 3; j++) {
      c[i] += a[i][j] * b[j];
  }
}
```

最適化されたコード

```
double  a_0_0, a_0_1, a_0_2;
double  a_1_0, a_1_1, a_1_2;
double  a_2_0, a_2_1, a_2_2;
double  b_0, b_1, b_2;
double  c_0, c_1, c_2;

c_0 = a_0_0 * b_0 + a_0_1 * b_1 + a_0_2 * b_2;
c_1 = a_1_0 * b_0 + a_1_1 * b_1 + a_1_2 * b_2;
c_2 = a_2_0 * b_0 + a_2_1 * b_1 + a_2_2 * b_2;
```

- 短いループ ➔ ループ展開 （ループのフル・アンロール）
- 短い配列 ➔ スカラー変数に
- 関数コール ➔ マクロに （関数インライン展開）

コード最適化の例（続）
ー SIMD強化されたスカラー向け ー

ショートベクトル化されたコード

```
double  a_0_0[4], a_0_1[4], a_0_2[4];
double  a_1_0[4], a_1_1[4], a_1_2[4];
double  a_2_0[4], a_2_1[4], a_2_2[4];
double  b_0[4], b_1[4], b_2[4];
double  c_0[4], c_1[4], c_2[4];

For (k = 0; k < 4; k++) {
  c_0[k] = a_0_0 [k] * b_0 [k] + a_0_1 [k] * b_1 [k] + a_0_2 [k] * b_2 [k];
  c_1[k] = a_1_0 [k] * b_0 [k] + a_1_1 [k] * b_1 [k] + a_1_2 [k] * b_2 [k];
  c_2[k] = a_2_0 [k] * b_0 [k] + a_2_1 [k] * b_1 [k] + a_2_2 [k] * b_2 [k];
}
```

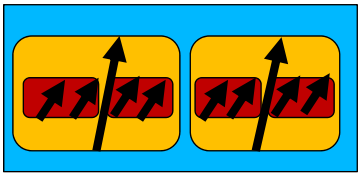
- ショートベクトル化のための人工的なループを追加
  ４，８，１６（…)個ごとに並列実行

コードチューニングのガイドライン

- 関数呼び出しをマクロに
  - 明示的にインライン展開
- 配列を使わず、（大量の）スカラー変数で代用
  - 明示的にループアンローリング
- 複数の要素について同時処理
  - 要素数方向にベクトル化・SIMD化
  - ループを明示的に記述
    - ループにコンパイラディレクティブを適宜挿入
  - 最適ループ長はHWアーキ依存
    - x86系では4、8、16あたり
    - 京・FX10ではベクトル長を数十程度に（例：48、72、96）
  - ベクトル化・SIMD化できない部分を分離して別ループに
    - 配列への間接インデックス参照
    - 分岐

例： 構造解析のホットスポット

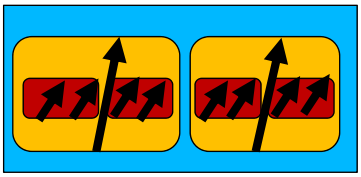
- 要素単位の演算
  - 要素剛性行列の評価
    - 数値積分　（各積分点で$[B]^t[D][B]$を評価）
    - 面積・体積積分公式の利用 （三角形と四面体の場合）
  - 非線形構造解析
    - 材料構成則の扱い （[D]マトリクス、C 4階テンソル）
    - 応力積分
  - 陽解法系コード
  - Element-by-element行列無記憶型反復法ソルバー



多重ループ
外側ループ長い
内側ループ短い → ***行列または テンソル演算***

対策

$$ {}_3[B]^t \; {}_6[D] \; {}_6[B] $$



△ キャッシュ　良
レジスタ　悪
命令数　増やせない

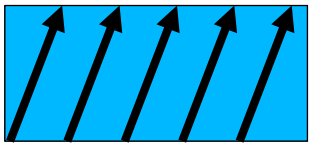
```
for i = 0〜2
  for j = 0〜5
    for k = 0〜2
      BtDB[i][j] = …
```

（*行列・テンソルライブラリの*）
マクロによる実装、インライン展開

配列変数からスカラー変数へ

短ループのフルアンロール

行列・テンソル演算の
主要パターンを網羅



◎ キャッシュ　良
レジスタ　良
命令数　多

```
BtDB_0_0=…
BtDB_0_1=…
BtDB_0_2=…
BtDB_1_0=…
BtDB_1_1=…
```

有限要素法向けベンチマークテスト

	Intel (Sandy Bridge) インテルコンパイラ		東大Oakleaf-FX	
	オリジナル	チューニング	オリジナル	チューニング
構造解析・要素剛性	22%	70%	2%	44%
非線形材料構成則	12%	31%	10%	40%
熱伝導解析・要素剛性	24%	50%	12%	38%

Intel系　Cコンパイラ、-O3 –xAVX
ショートベクトル化によるSIMD命令生成　　最適ベクトル長＝４

FX10(および京）　Cコンパイラ、-Kfast　（＋ディレクティブ）
VSIMDによるSIMD化　　最適ベクトル長＝２～８０

LexADV-AutoMTおよびAutoLinAS
（FX100でのベンチマーク）

河合浩志（諏訪東京理科大）

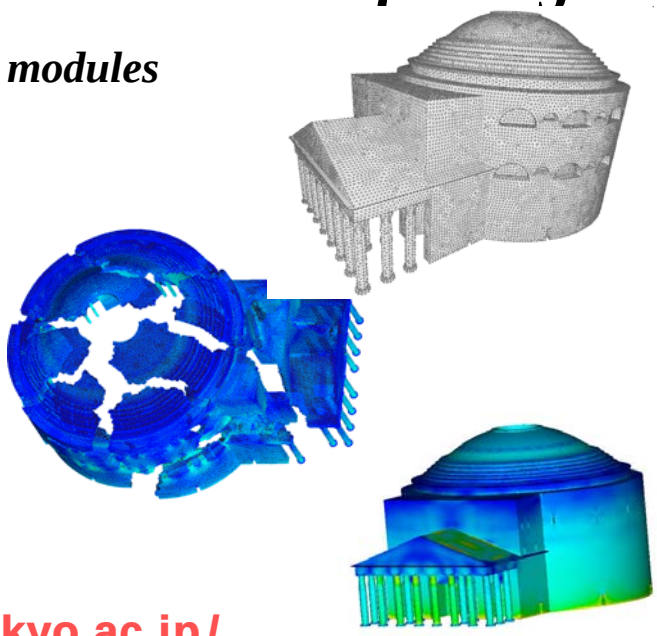
1

ADVENTURE : Open-source CAE package

Module-based Architecture : 20+ modules

CAD-connection, Mesh Generation,
Domain Decomposition, FE-Solvers,
Visualization, Optimization, etc.

AdvIO, AdvTriPatch, AdvTetMesh,
AdvBCtool, **AdvSolid**, AdvVisual,
AdvCAD, AdvThermal,
AdvMagnetic, AdvForge,
AdvImpact, AdvFluid, AdvIAgent,
AdvOpt, AdvShape, AdvMaterial,
AdvAuto,

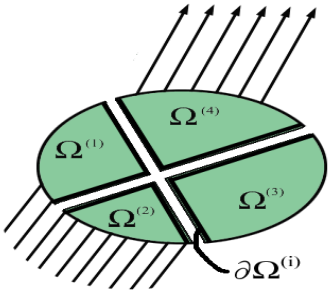


http://adventure.sys.t.u-tokyo.ac.jp/

It's open-source !

2

大規模問題の並列化



- （反復法で）行列ベクトル積などを
  並列化

  ex.) 並列CGソルバー

  前処理: ブロックIC, 近似逆行列、AMG, ...

  特徴:　全体行列を扱う　[A]{x}={b}

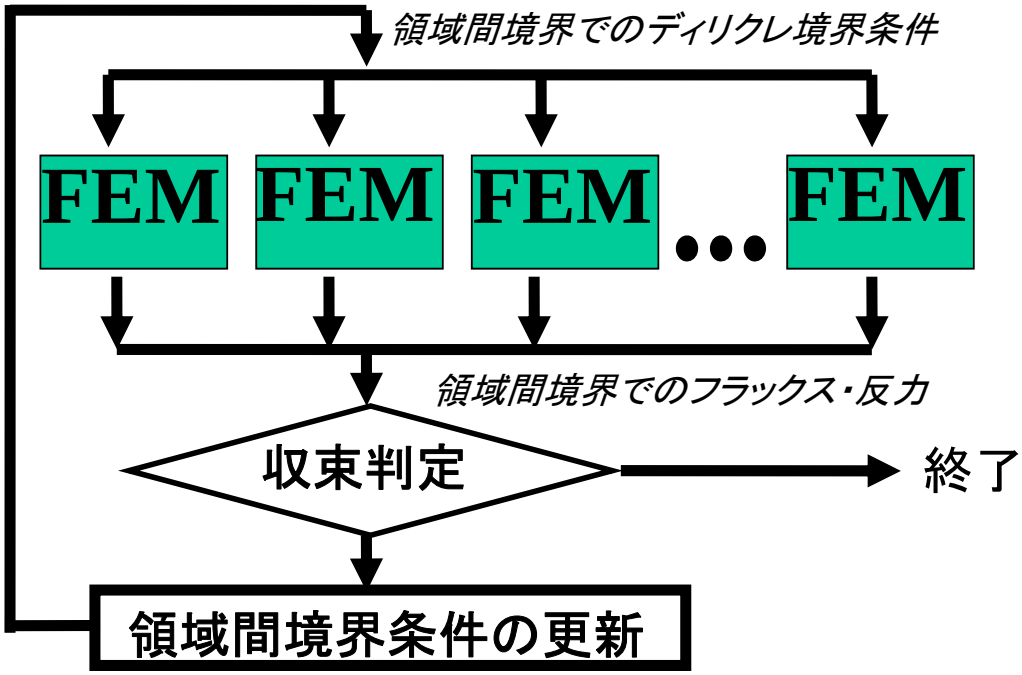
- ***領域分割法*** (反復型サブストラクチャ法)

  ex.) DDM

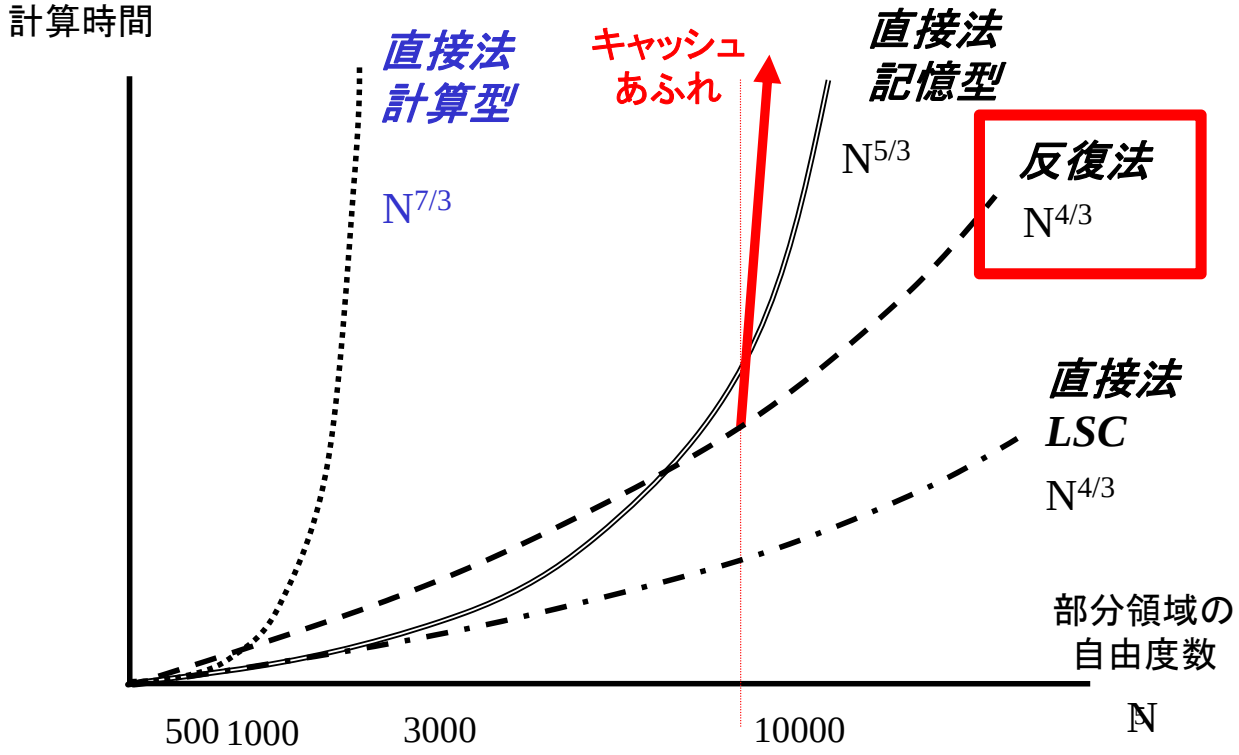
  前処理: BDD, FETI, ...

  特徴:　インターフェイス問題（Schur補元行列）
  を扱う　[S]{u}={g}

領域分割法（DDM）のフロー



部分領域ローカルFEソルバー
-- 部分領域自由度と計算コストの関係 --



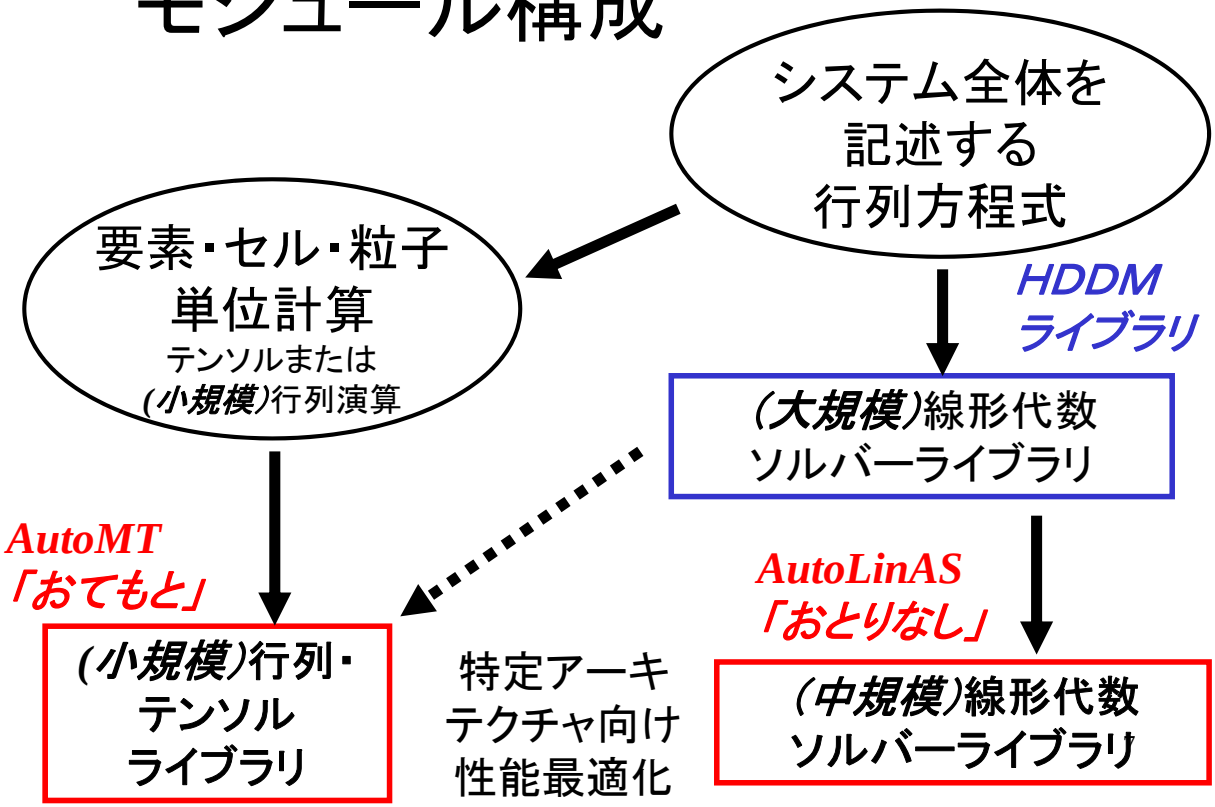
反復法ベースの
ローカルFEソルバーの可能性？

- ポイント
  - 計算量が$N^{4/3}$に比例
  - キャッシュに入れられる（？）
  - SSOR前処理(+Eisenstat技法)は効率的

- 議論
  - 部分領域が大きくなるとキャッシュに入らない
    - 並列化が必要 ➔ 前処理の並列化はどうする？

  - キャッシュ上での実行効率（ピーク性能比）が低い！！！
    - 間接インデックス参照、scatter/gatherの多用
    - 無理にキャッシュに入れず、メモリから読んでも同じ？？？

DDMに基づくコードのモジュール構成



線形代数ソルバーライブラリ AutoLinAS

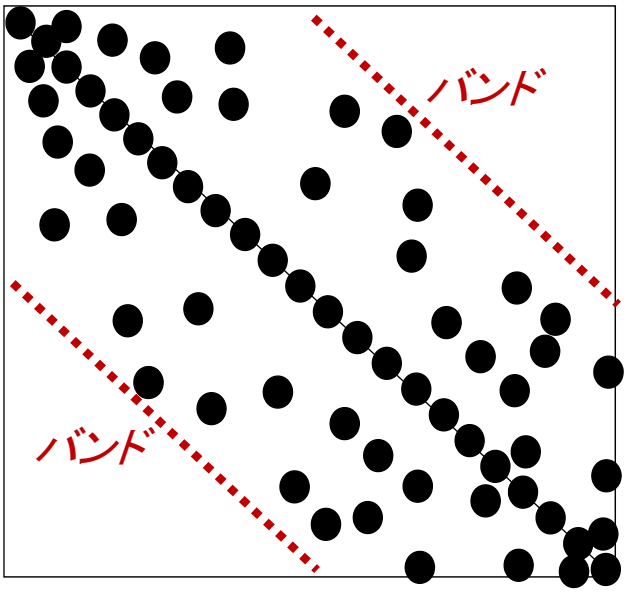
- 直接法ソルバー
  - 対称行列向け： [L][L]^T、[L][D][L]^T
  - フル、スカイライン
  - プラットフォームごとの最適化およびスレッド並列化

- 反復法ソルバー
  - 対称行列向け： CG法
  - 前処理： 対角スケーリング、SSOR、ICT
  - プラットフォームごとの最適化およびスレッド並列化

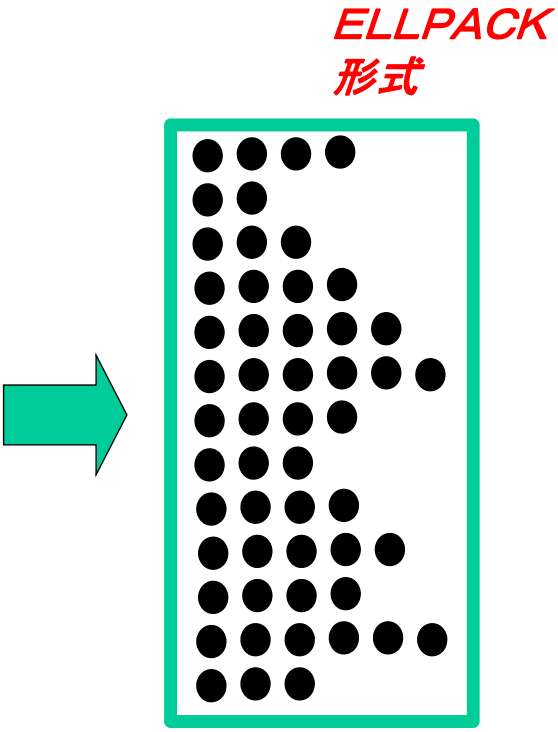
- ツール
  - 疎行列用インデックス管理、リナンバリング
  - 4倍精度(double-double)

反復法ソルバーの記憶形式

非ゼロ成分のみ

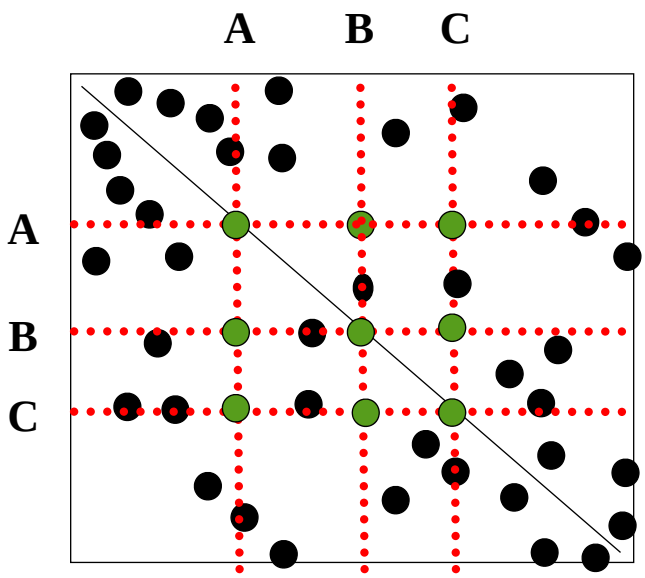


FEMでの疎行列

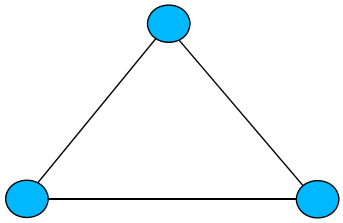


非ゼロ成分のみ記憶する
(*CRS形式*)

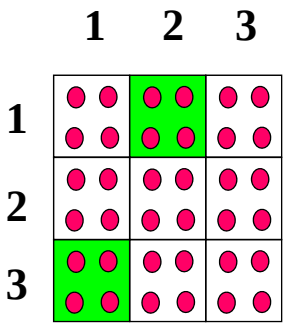
節点ブロック化



全体剛性行列へのアセンブル



(二次元)三角形一次要素



要素剛性行列

構造問題:三次元ソリッド要素では*3x3*節点ブロック

SIMD拡張命令

- 複数命令を同時実行する仕組み（IPCの向上）
  - **同一の**演算命令を**複数の**データに適用（***SIMD***）
  - レジスタサイズを拡大（256ビットであればdouble4個分）
- コンパイラによる支援 （***SIMDベクトル化***）
  - “いわゆる”ベクトル化可能なループを対象
  - （現在は）配列へのシーケンシャルアクセスのみ
    - ***gather/scatter命令***のサポートは？？？

例)　c[i] = a[i] × b[i]

レジスタ0	a[0]	a[1]	a[2]	a[3]
	×	×	×	×
レジスタ1	b[0]	b[1]	b[2]	b[3]
	‖	‖	‖	‖
レジスタ2	c[0]	c[1]	c[2]	c[3]

疎行列ベクトル積(SpMV)ベンチマーク

- チューニング対象
  - x86アーキテクチャ
    - Intel Core i7 / Xeon (***Haswell***)
    - Intel Xeon Phi (***Knights Corners***)
  - 富士通 （京、FX10、***FX100***）
- 実装手法
  - 節点ブロック化(3x3)、対称性の利用
  - ELLPACK形式 （行あたりの成分数を固定）
  - 節点ブロック単位SIMDベクトル化
  - 列方向SIMDベクトル化

ベンチマーク詳細 (1)

- 基本方針
  - C言語を使用
  - コンパイラの最適化機能に頼る
    - Intelコンパイラおよび富士通コンパイラ
      - SIMDベクトル化
    - アセンブリ言語やintrinsic関数は使わない
  - 動的メモリ確保(malloc)
    - 静的確保の場合に対し、性能が低下しないか？
  - １部分領域を１スレッドで解く
    - 部分領域数に関してOpenMP並列化
    - 部分領域データがコアごとのキャッシュに載るか？

13

ベンチマーク詳細 (2)

- 富士通FX100 (SPARC64 XI 2.2GHz, 32 core)
  - Ｂ／Ｆ値0.426（readはその半分）、4-way SIMD、キャッシュ0.75MB / core
  - オプション : -Kfast –Kocl
  - ディレクティブ : novrec、swp、simd
- Intel Xeon E3-1270, 3.5 GHz, 4 core (***Haswell***)
  - Ｂ／Ｆ値0.114、4-way SIMD、キャッシュ 2MB / core
  - オプション: -O3 –xCORE-AVX2
  - ディレクティブ : ivdepおよびsimd
- Intel Xeon Phi 5110P, 1.05GHz, 60 core (***Knights Corner***)
  - Ｂ／Ｆ値0.317、8-way SIMD、キャッシュ0.5MB / core
  - オプション : -O3 -mmic
  - ディレクティブ : ivdepおよびsimd

14

疎行列ベクトル積ベンチマーク

	オンキャッシュ	キャッシュに載らない
FX100	14 – 11 % (7 %)	8.4 – 8.1 % (6.7 – 6.3 %)
Haswell	16 – 14 %	8.2 %
Knights Corner	5.7 – 3 %	4.8 – 3.2 %

注意1: （多くの場合）最大値はELLPACK、最小値はCRS
注意2: ()内は動的メモリ確保(malloc)を行った場合

SIMDがほとんど効いていない？？？

15

考察（1）

- 3x3節点ブロックは確かに効く
- 対称性を考慮した方がよいことが多い
  - キャッシュに載らない場合は特に
  - 場合により、一般行列でもそこそこ高速なことがある
  - とりあえず、メモリ使用量は半分で済む
- ELLPACK vs. CRS
  - ELLPACKは命令実行効率は良いが、演算量が多い
    - 要素タイプやメッシュごとに調整が必要
    - CRS含め複数の記憶タイプを組み合わせる必要
      - 実装が複雑に

16

考察（2）

- Xeon Phi： キャッシュの効果は？
  - 二次キャッシュ容量： コア当たり512KB
  - 60コアに対し、120または240スレッドが最適
    - コアごとに2～4スレッドを実行
  - スレッドごとに1部分領域を扱う
    - 部分領域ごとに使えるキャッシュはきわめて少なくなる

- FX100： mallocすると遅くなる？？？
  - プラグマnovrec、swp、simd alignedを使用
  - オンキャッシュ状態での性能劣化が著しい

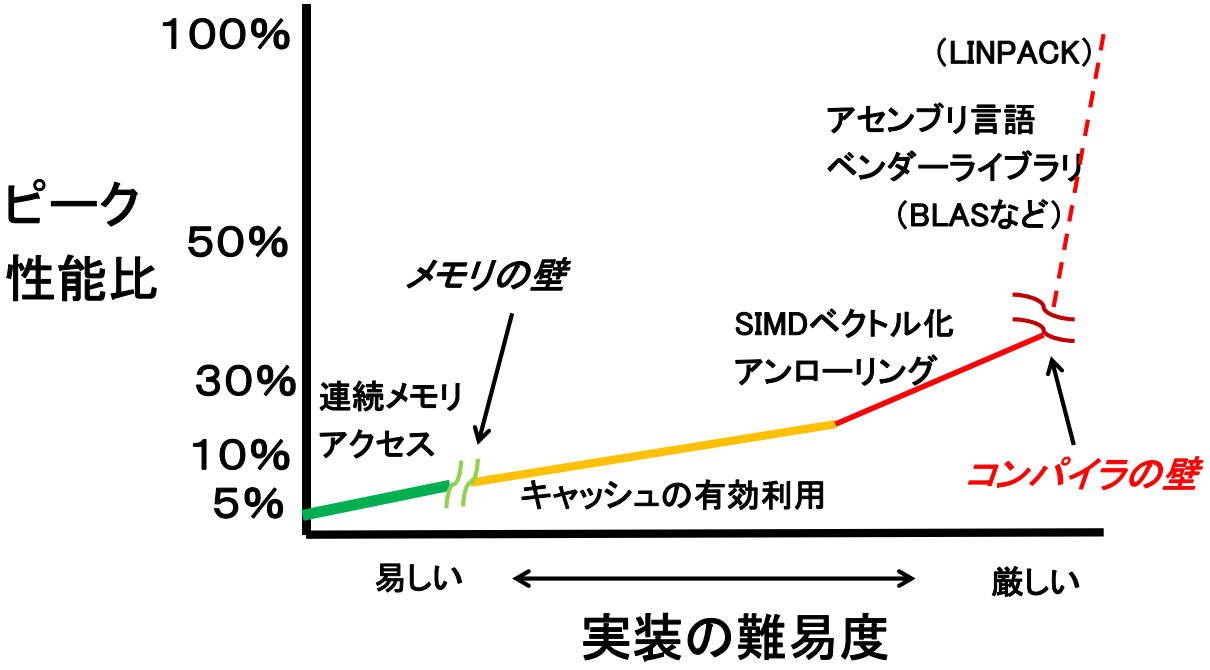
17

考察（3）

- SIMDは本当に効いているのか？
  - 本当にSIMD化されているのか？
    - gather / scatter命令が使われているのか？
    - 生成されたアセンブリ命令の確認が必要
  - たとえgather/scatterを使っていたとしても
    - それで本当に速くなるのか？

18

単体性能の向上： 最新スカラー機事情



コンパイラの壁 ???

19

オンキャッシュなのに性能がでない？

キャッシュに載っていても…

キャッシュ上で性能が出ない！！！

~~Compiler Wall ???~~

正確には…

ILP (Instruction-Level Parallelism) Wall

命令レベルの並列性に問題あり

1クロックに何個の演算を並列実行できるか？

	スーパースカラー	積和命令 (FMA)	SIMD命令	計
京(富士通SPARC64 VIIIfx)	2	2	2	8
富士通FX100	2	2	4	16
Haswell	2	2	4	16
Knights Corner	2	1	8	16

20