PCクラスタコンソーシアム
HPCオープンソースソフトウェア普及部会ワークショップ
「高性能数値計算ライブラリLexADV」
2021年9月10日 （オンライン）

行列テンソル演算 ライブラリ LexADV_AutoMTの概要と実装

河合浩志（東洋大）

簡単な自己紹介など

河合　浩志

（現在）　東洋大学　総合情報学部　教授　（53歳）

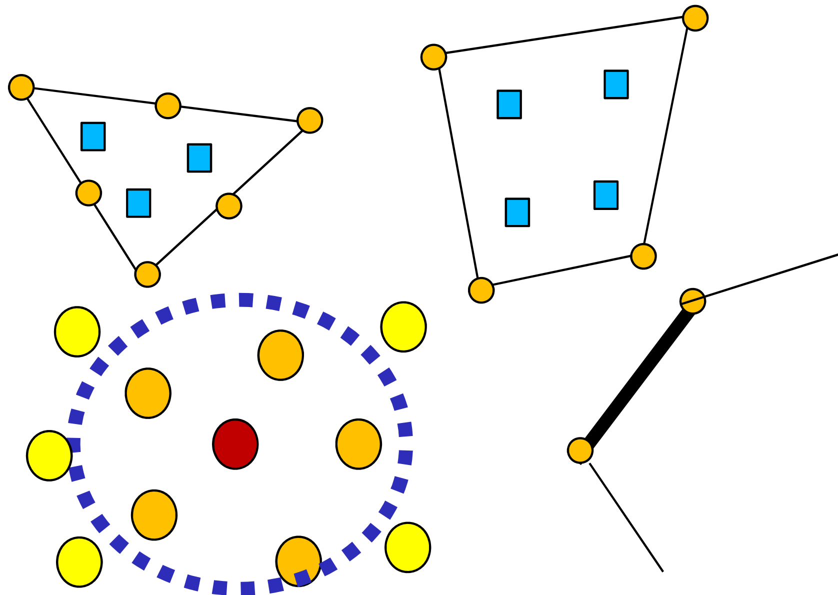
時期	プログラミング	計算機	仕事（＋趣味）
1985	FORTRAN77	（大学の）大型計算機	大学の演習
1986	BASIC、マシン語	（自宅の）富士通FM-77	RPGなど
1987	TurboPascal	（大学の）教育用センター	RPGなど
1988	C、Smalltalk-V	PC-9801	バイト
1989	C++	PC-9801	バイト
1990	C、C++	（研究室の）Sun-4	プリポスト処理
1993	C、ベクトル化	Cray C90	ソルバー
1994	PVM	WSクラスタ	ソルバー
1998	Java	Windows PC	プリポスト処理

簡単な自己紹介など（続）

時期	プログラミング	計算機	仕事（＋趣味）
			ADVENTUREプロジェクト関連
2000	C、C++	Linux PC	CAD、ツール類
2001	C、MPI	（研究室）PCクラスタ	ソルバー
2003	C、MPI、ベクトル化	地球シミュレータ	ソルバー
2004	C、SIMD	Linux PC	ソルバー
2008	C、MPI、OpenMP	（東大）T2K	ソルバー
2010	CUDA	NVIDIA GPU	ソルバー
2011	C、MPI、OpenMP、SIMD	京コンピュータ	ソルバー
2019	C、MPI、OpenMP、SIMD	富岳	ソルバー
2020	C、MPI、OpenMP、ベクトル化	NEC SX-Aurora TSUBASA	ソルバー
2021	C、OpenACC	NVIDIA GPU	ソルバー

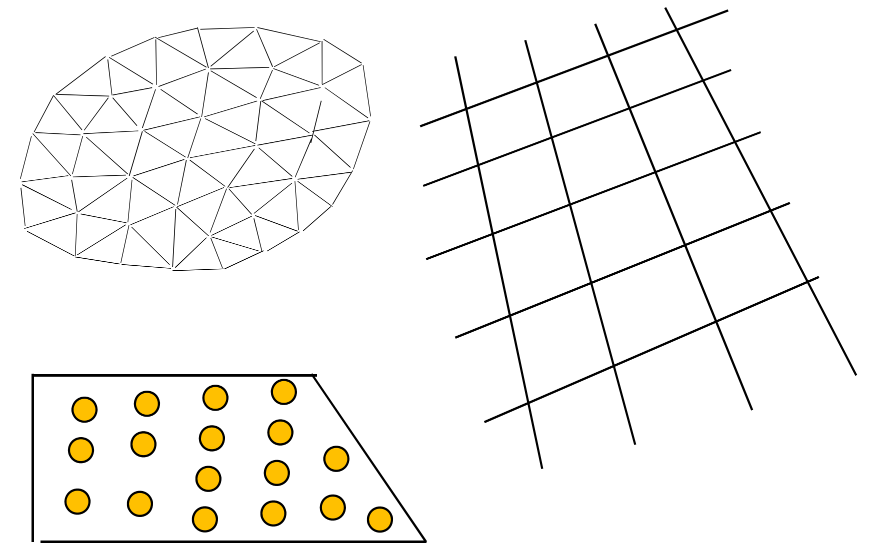
連続体力学シミュレーション
解析プログラム・コードの特性

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
  - ➔　コードの大部分を占める

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

数値計算のための抽象データ型

- 基礎的な型
  - ベクトル、行列
  - 複素数
  - 多倍長精度数 （4倍精度、double-doubleなど）

- 連続体力学向け （要素、セルごとの計算で頻出）
  - 3次元ベクトル
  - テンソル（3次元）
  - 一定サイズ・小規模な行列 (6x6、9x9など）

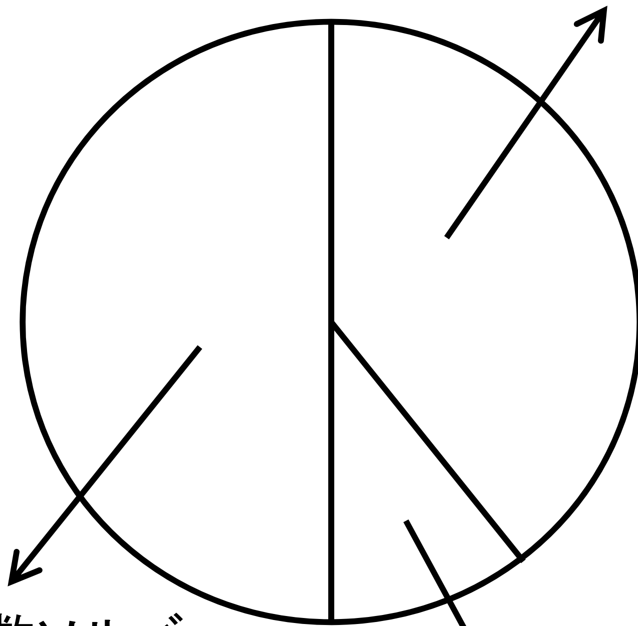
- その他
  - 4元数 （クォータニオン）
  - 超双対数 (HDN : Hyper Dual Number)

（比較的）小規模で固定サイズのものが多い

シミュレーションコードの特性
--実行時間--

小規模問題

節点、要素、セル、材料など
（***ローカルレベル・物理***）

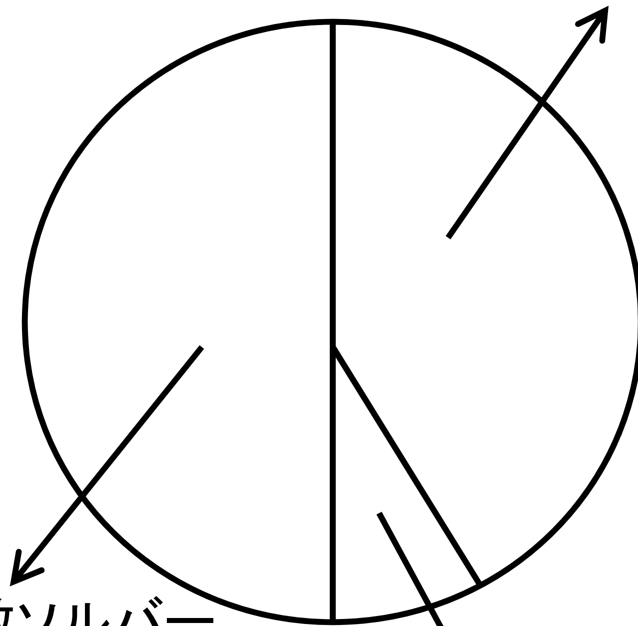


線形代数ソルバー、
非線形性、時間依存性
（***グローバルレベル・力学***）

ファイル入出力

大規模問題

節点、要素、セル、材料など
（***ローカルレベル・物理***）



線形代数ソルバー、
非線形性、時間依存性
（***グローバルレベル・力学***）

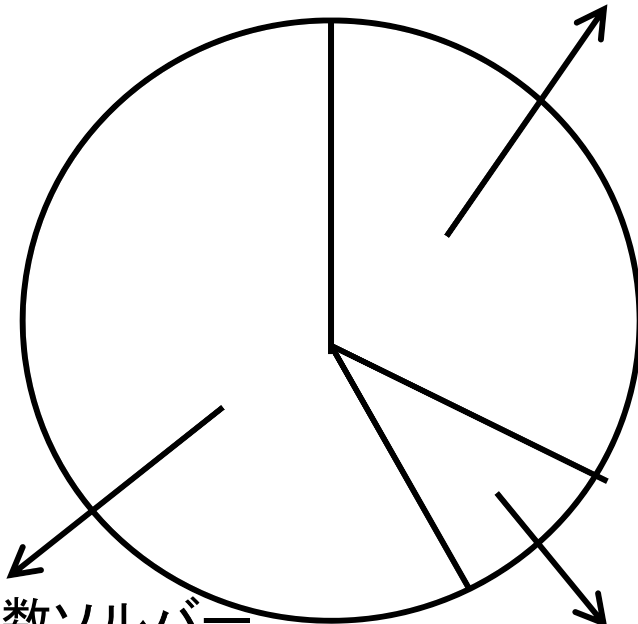
ファイル入出力

問題サイズにかかわらず、要素・セル単位計算が一定の割合を占める
（非線形、非定常問題の場合）

シミュレーションコードの特性
--規模・コード行数--

＜小規模／自家製コード＞

節点、要素、セル、材料など
（*ローカルレベル・物理*）



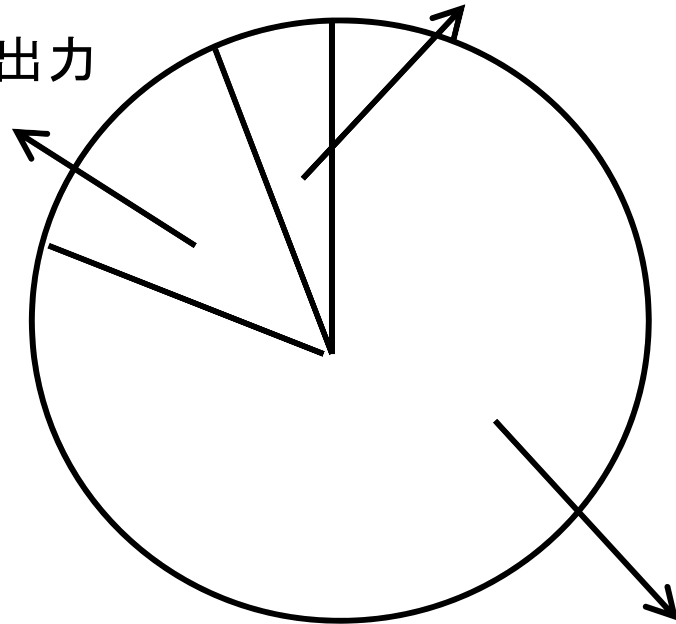
線形代数ソルバー、
非線形性、時間依存性
（*グローバルレベル・力学*）

ファイル入出力

＜大規模／汎用コード＞

線形代数ソルバー、
非線形性、時間依存性
（*グローバルレベル・力学*）

ファイル入出力



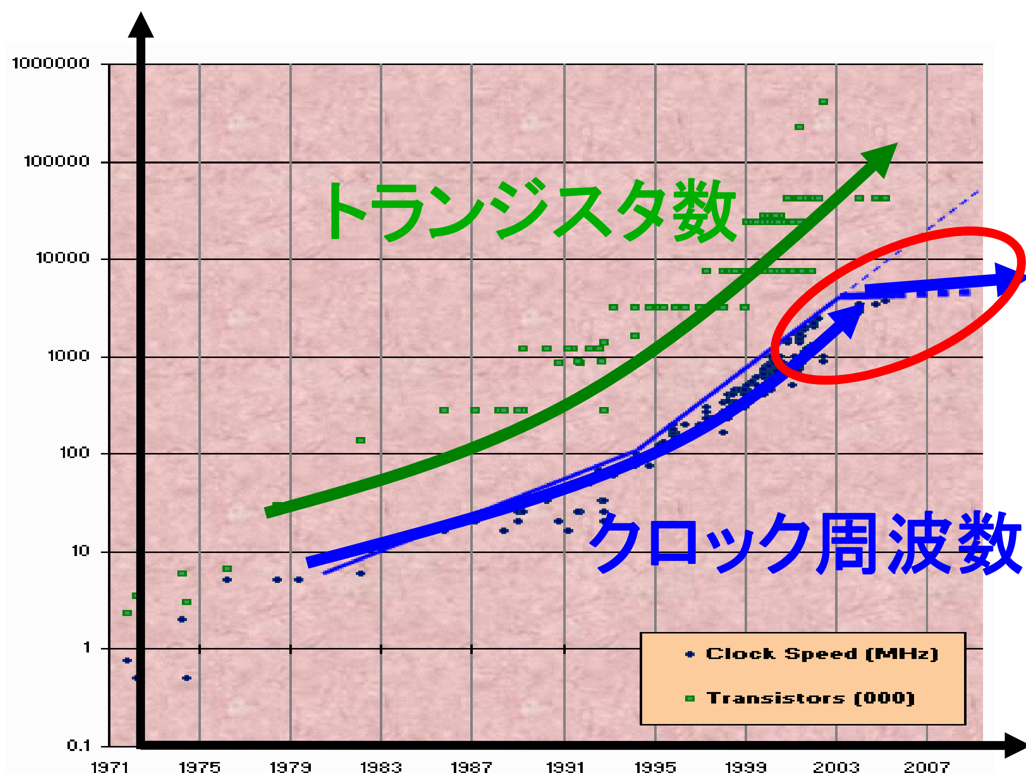
節点、要素、セル、材料など
（*ローカルレベル・物理*）

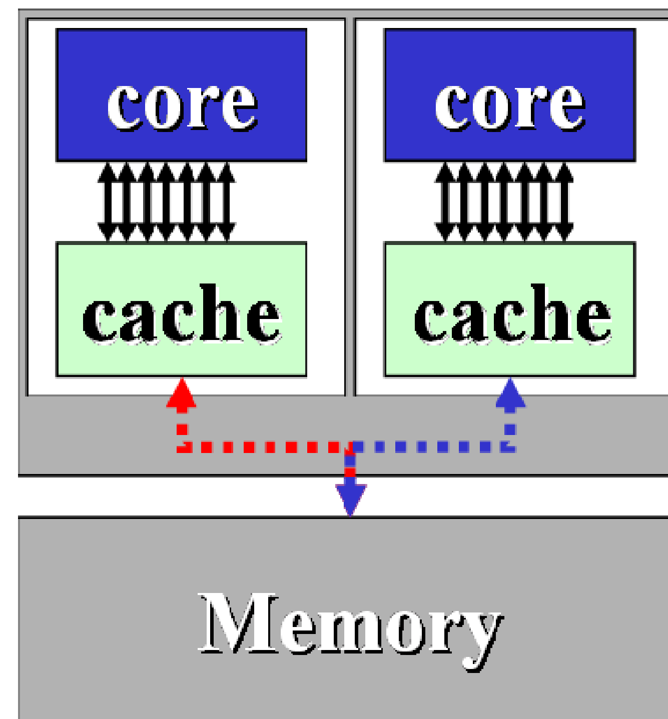
コードが大きくなればなるほど、要素・セル単位での処理が増える
（より強い非線形性、より多くの材料や要素タイプ）

速いコードを書くには？

- 事実１:　PCはすでにかなり早い　（ただし、ピーク性能においてだが）
  - NVIDIA Tesla : 6 T FLOPS
  - Intel Xeon Phi : 3 T FLOPS
  - Intel Core i9 (Skylake-SP) : 2 T FLOPS (18 core)
- 事実２:　でも、あなたのコードはせいぜい1～2G FLOPS程度かも
  （チューニングされていなければ）
  - クロック周波数自体はせいぜい2～3GHzなので
- 速いコードを書くには、（ピークに近い性能を出すには）
  - ***関数・サブルーチンコールをしない***
  - ***（短い）ループを使用しない***
  - ***（小サイズの）配列を使用しない***

いまどきのスカラーCPU





クロック周波数は変わらず
（2～4GHzで停滞)
➔ クロックあたり演算数の増加

コア数の増加、
メモリバンド幅制約
➔ キャッシュの重要性

(ピークで)TFLOPS級CPU/GPUの出現
➔ 実際には(何もしなければ)～GFLOPS程度

クロックあたり、何個の演算を同時に実行できるのか？

ILP (Instruction-Level Parallelism) Wall
命令レベルの並列性に問題あり

プロセッサ アーキテクチャ世代	命令セット	スーパー スカラ	積和命令 (FMA)	SIMD 命令	total
K Computer(富士通)	HPC-ACE	2	2	2	8
Sandy Bridge	AVX	2	1	4	8
PRIMEHPC FX100 (富士通)	HPC-ACE Ver.2	2	2	4	16
Haswell	AVX2	2	2	4	16
Knights Landing	AVX512	2	2	8	32
Skylake-SP	AVX512	2	2	8	32
富岳・FX1000	ARM-SVE	2	2	8	32

SIMDベクトル化

```
double  A[10000],  B[3][10000]
double  D[10000];

for (i = 0;  i  < 10000;  i++) {

    double  a,  b_x,  b_y,  b_z,  c,  d;

    a  =  A[i];
    b_x  =  B[0][i];     b_y  =  B[1][i];     b_z  =  B[2][i];

    c  =  a  *  (b_x  +  b_y  +  b_z);
    d  =  c + b_x  *  b_y  *  b_z;
    d  +=  a  *  (b_x  -  b_y  -  b_z);

    D[i]  =  d;
}
```

グローバル配列（SOA形式）

長めのループ

ローカル・スカラー変数

配列からスカラー変数へロード

計算部分

スカラー変数を配列へストア

ハイパフォーマンス・デザインパターン

- アプリケーション分野ごとの抽象データ型を扱う
  - 抽象データ型： データ型と対応する演算手続き群
    - ベクトル、行列、テンソル、複素数、4元数、4倍精度…
- Cプリプロセッサマクロによる実装
  - 結合演算子の利用　　aa##bb ➔ aabb
  - データ型をスカラー変数群で表現

    declare(a) ➔ double a_x, a_y, a_z;
  - 関連する演算手続きをCプリプロセッサマクロで表現

    add(a, b, c) ➔ c_x=a_x+b_x; c_y=a_y+b_y; …

例: 3-Dベクトル

変数宣言

```
#define declare_v3(a)¥
  double a##_x, a##_y, a##_z;
```

配列からのロード

```
#define load_v3(a, array)¥
  a##_x=array[0]; ¥
  a##_y=array[1]; ¥
  a##_z=array[2];
```

代入

```
#define assign_v3_v3(a, b)¥
  b##_x=a##_x; ¥
  b##_y=a##_y; ¥
  b##_z=a##_z;
```

加算

```
#define add_v3_v3_v3(a, b, c)¥
  c##_x = a##_x + b##_x; ¥
  c##_y = a##_y + b##_y; ¥
  c##_z = a##_z + b##_z;
```

<オリジナルCソースコード>

```
double aa[3], cc[3];
declare_v3(a);
declare_v3(b);
declare_v3(c);

load_v3(a, aa);
assign_v3_v3(a, b);
add_v3_v3_v3(a, b, c);
store_v3(c, cc);
```

<Cプリプロセッサ出力>

```
double aa[3], cc[3];
double a_x, a_y, a_z;
double b_x, b_y, b_z;
double c_x, c_y, c_z;

a_x=aa[0]; a_y=aa[1]; a_z=aa[2];
b_x=a_x; b_y=b_y; c_z=c_z;
c_x = a_x + b_x;
c_y = a_y + b_y;
c_z = a_z + b_z;
cc[0]=c_x; cc[1]=c_y; cc[2]=c_z;
```

4倍精度のコード例

```
for (jDof = 0; jDof < nColumns; jDof++) {
   int offset = jDof * nRows;

   AutoMT_inline_declare_dd(b);

   AutoMT_inline_load_dd_d_d(b, vector_hi[jDof], vector_lo[jDof]);

#pragma ivdep
#pragma simd
   for (iDof = 0; iDof < nRows; iDof++) {
    AutoMT_inline_declare_dd(M_ij);
    AutoMT_inline_declare_dd(c);
    AutoMT_inline_declare_dd(d);

    AutoMT_inline_load_dd_d_d
     (M_ij, cmps_hi[offset + iDof], cmps_lo[offset + iDof]);

    AutoMT_inline_load_dd_d_d
     (d, result_hi_OUT[iDof], result_lo_OUT[iDof]);

    AutoMT_inline_mul_dd_dd_dd(M_ij, b, c);

    AutoMT_inline_add_dd_dd_dd(d, c, d);

    AutoMT_inline_store_dd_d_d
     (d, result_hi_OUT[iDof], result_lo_OUT[iDof]);
   }
 }
```

スカラー変数宣言
double b_hi, b_lo;

bに配列変数の値をロード

ベクトル化のためのコンパイラディレクティブ
(Intelコンパイラ向け)

スカラー変数宣言
double M_ij_hi, M_ij_lo;
double c_hi, c_lo;
double d_hi, d_lo;

M_ijに配列変数の値をロード

dに配列変数の値をロード

mul : 掛け算　$c = M_ij * b$

add: 足し算　$d = d + c$

dを配列変数にストア

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
LexADV_AutoMT「おてもと」

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

例1：移動硬化則、相当応力
（非対称）テンソル形式

$$\sigma_e = \{\frac{2}{3}(\boldsymbol{\sigma}' - \mathbf{x}') : (\boldsymbol{\sigma}' - \mathbf{x}')\}^{\frac{1}{2}}$$

```
real*8 t_sigma_prime(3,3), t_x_prime(3,3)
real*8 sigma_e

real*8 s_tmp
real*8 t_tmp1(3,3)

call automt_sub_t_t_t (t_sigma_prime, t_x_prime, t_tmp1)
call automt_colon_t_t_s (t_tmp1, t_tmp1, s_tmp1)
sigma_e = sqrt (2.0d0 / 3.0d0 * s_tmp1)
```

例2: 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)
```

サポートされている演算

２階テンソル演算

<単項演算>

det ***X*** tr ***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>

X + ***Y*** ***X*** - ***Y*** s ***X***

X Y ***X*** : ***Y*** ***X*** ･･ ***Y***

X a ***a X***

X,***Y*** : ２階テンソル
a, ***b***, ***c*** : ベクトル
s, t : スカラー

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

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	21%	60%	8%	35%
非線形材料構成則	16%	27%	6%	14%
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（AVX）

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＋ソフトウェアパイプライニング

まとめ

- テンソル演算と(小規模）行列・ベクトル演算のための C/C++/Fortran向けライブラリを開発
- ハイパフォーマンス・デザインパターンの提案
  – スカラー機上でピーク性能比3～7割を達成

今後の課題

- GPU対応
- 連続体力学向けDSLの開発 (LaTeX形式からのコードジェネレータ）

コードが「速い」とは？

プログラムの
実行速度
（効率）

＝

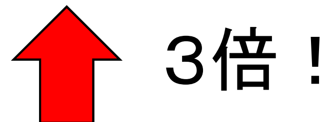
アルゴリズムの
効率

×

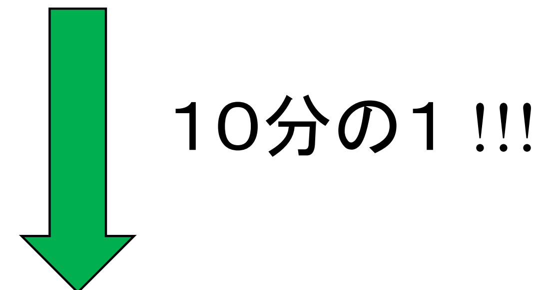
コードの
効率

必要な
浮動小数点演算
(Floating Point Operation)
総数を減らす

単位時間あたり
浮動小数点演算
実行数を増やす
(FLOPS)



３倍！



１０分の１ !!!

速いコードを書くには？

- 事実１：　PCはすでにかなり早い　（ただし、ピーク性能においてだが）
  - NVIDIA Tesla : 6 T FLOPS
  - Intel Xeon Phi : 3 T FLOPS
  - Intel Core i9 (Skylake-SP) : 2 T FLOPS (18 core)
- 事実２：　でも、あなたのコードはせいぜい1～２G FLOPS程度かも
  （チューニングされていなければ）
  - クロック周波数自体はせいぜい２～３GHzなので
- 速いコードを書くには、（ピークに近い性能を出すには）
  - *関数・サブルーチンコールをしない*
  - *（短い）ループを使用しない*
  - *（小サイズの）配列を使用しない*

古き良き時代
(20年以上むかし...)

大型計算機
（スパコン）

ミニコンピュータ

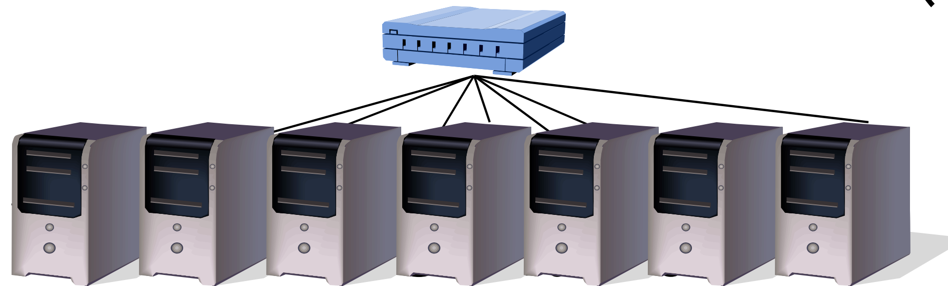
ワーク
ステーション

PC

同じコードで
コードが速くなり、
より大きな問題を
（ベクトル化。。。）

わりと最近まで
(2000-2010)

スパコン
（箱、大杉！）



PCクラスタ
（箱の束！）

PC・ワークステーション＝「箱」？

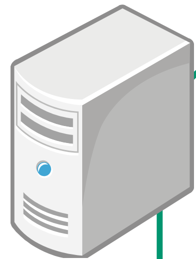
同じコードで
コードが速くなり、
より大きな問題を
（ただし、MPI並列）

今後、エクサ時代に向けて(2010-2020 ?)

ワークステーション

PCクラスタ

スパコン



CPU

SIMD命令の強化

…

二次キャッシュ

三次キャッシュ

オンチップ
メモリ

メイン
メモリ

SSD

HDD

アクセラレータ (GPU, MIC)

…

SIMD cores

シェアードメモリ
グローバルメモリ

同じコードで.....???

CUDA、OpenCL、OpenACC
SIMDベクトル化
OpenMP
しかも、MPIに加えて