ハードウェアを意識した数式処理ソフトの開発について

木村 欣司 (京都大学大学院情報学研究科)

講演内容

高速に計算を行う数式処理ソフトを作るにはどうしたらよいかを話す

さまざまなテクニックを網羅的に紹介する

キャッシュの話

キャッシュとは

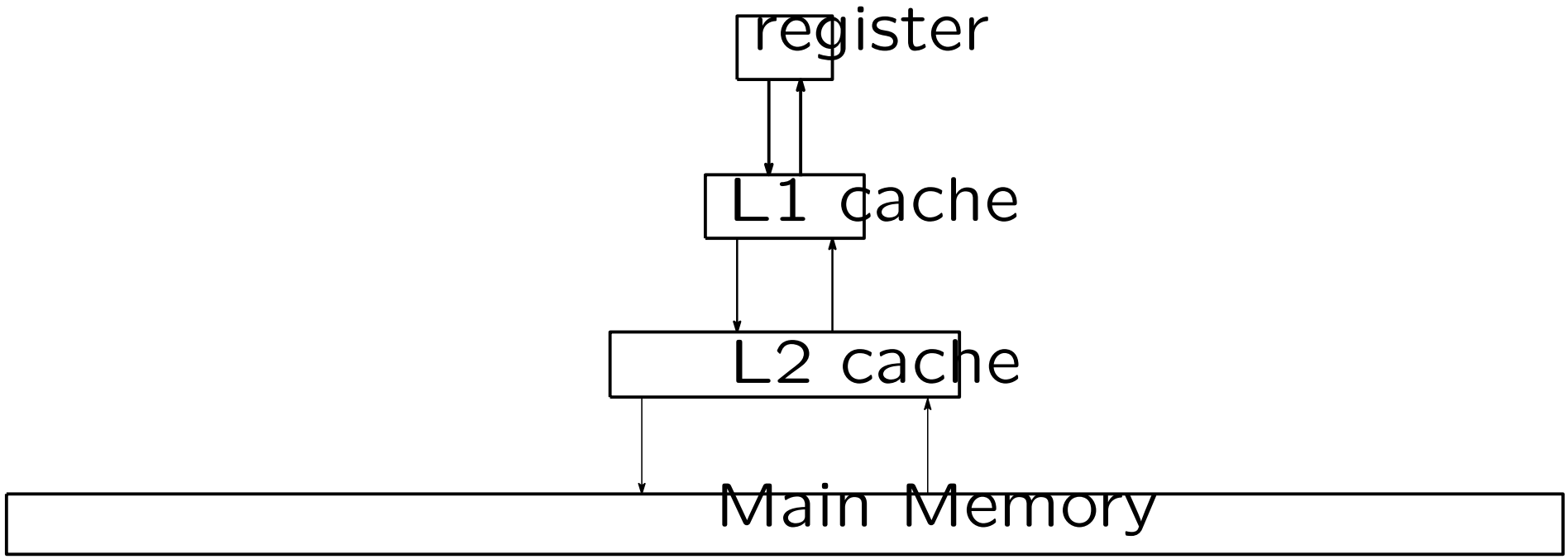
1.CPUの内部にあり，メインメモリよりもはるかに容量が小さい

2.最近読み込んだデータとその周辺のデータ，あるいは，最近計算した結果，あるいは最近使った命令列とその周辺の命令列を，一時的に蓄えておくところ，使われなくなったデータは，メインメモリに返される

3.メインメモリと registerの間よりも，キャッシュとregisterの間のほうが，単位時間当りのデータを移動できる量が多い

4.L1，L2，(L3)，(L4)の階層構造になっている

キャッシュ階層



例外として，メインメモリから，L2キャッシュを飛び越えて，L1キャッシュにデータが直接入るCPUがある

L2キャッシュから，L1キャッシュを飛び越えて，レジスタにデータが直接入るCPUがある

キャッシュ階層からみた優れたアルゴリズム

ベクトルの内積

・データの再利用性なし

行列ベクトル乗算

・ベクトル側にデータの利用性あり，行列側にはなし

行列行列乗算

・データの利用性大いにあり

⇒ **可能な限り行列行列乗算を中心としたアルゴリズムを設計するべし**

dgem2vuの話

行列ベクトル乗算は絶対に使ってはならないのか?

連立一次方程式の解法・行列の最小多項式の計算

Wiedemann algorithm

u_0, w_0は, ベクトル

s_iは, スカラー

$s_i = u_0^\top A^i w_0$を計算する

$t_0 = w_0, t_1 = At_0, t_2 = At_1, \cdots$

$s_0 = u_0^\top t_0, s_1 = u_0^\top t_1, s_2 = u_0^\top t_2, \cdots$

と計算してはいけない

$s_i = u_0^\top A^i w_0$の正しい実装法

$$
\begin{aligned}
s_0 &= (u_0^\top)(w_0) = (u_0)^\top (w_0) \\
s_1 &= (u_0^\top)(Aw_0) = (u_0)^\top (Aw_0) \\
s_2 &= (u_0^\top A)(Aw_0) = (A^\top u_0)^\top (Aw_0) \\
s_3 &= (u_0^\top A)(A^2 w_0) = (A^\top u_0)^\top (A^2 w_0) \\
s_4 &= (u_0^\top A^2)(A^2 w_0) = ((A^\top)^2 u_0)^\top (A^2 w_0) \\
&\vdots
\end{aligned}
$$

2系列を考える

$t_0 = w_0, t_1 = At_0, t_2 = At_1, \cdots$

$v_0 = u_0, v_1 = A^\top v_0, v_2 = A^\top v_1, \cdots$

この2系列は，互いに無関係

$$
\begin{aligned}
s_0 &= v_0^\top t_0 \\
s_1 &= v_0^\top t_1 \\
s_2 &= v_1^\top t_1 \\
s_3 &= v_1^\top t_2 \\
s_4 &= v_2^\top t_2 \\
&\vdots
\end{aligned}
$$

『Wilkinsonの技巧』を使うと，$Ax, A^\top y$の同時計算には，データの再利用性の高い実装法が存在する

```
for (j = 0; j < N; j++) ATY_TMP[j]=0;
for (j = 0; j < N; j++){
  TMP1 = 0;
  for (k = 0; k < N; k++){
    TMP1 = (ULL) A[j][k] * X[k] + TMP1;
    ATY_TMP[k]=(ULL) A[j][k] * Y[j] + ATY_TMP[k];
  }
  AX[j] = TMP1 % P;
}
for (j = 0; j < N; j++) ATY[j]=ATY_TMP[j] % P;
```

Wiedemann algorithmは，有限体$\mathbb{Z}/p\mathbb{Z}$上で利用するアルゴリズムであるため，% Pがある，**1回のデータのロードに対して，そのデータを2回使う**

$Ax, A^\top y$の同時計算に限って，データの再利用性が高い実装法が存在することは，どのくらい知られているのか?

“Intel Math Kernel Libiraryには，BLASとLAPACKの関数が含まれている” と言われるが，正確にはBLASに存在しない関数?gem2vuも入っている

The ?gem2vu routines perform two matrix-vector operations defined as

y1 := alpha*A*x1 + beta*y1,

and

y2 := alpha*A'*x2 + beta*y2

two-stage algorithmの話

行列行列乗算による高速化

実対称行列の固有値問題

Aを$n \times n$の与えられた実対称行列とする,

$$AV = VD, \quad V^\top V = VV^\top = I_n,$$

$$D = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}, \; V = \begin{pmatrix} v_1 & \cdots & v_n \end{pmatrix}$$

となる, 固有値$\lambda_i (1 \leq i \leq n)$, 固有ベクトル$V(n \times n)$を計算する

実対称行列Aの固有値分解の解法手順

1. 固有値保存変形により，$A \rightarrow$ 実対称3重対角行列T

(1)Householder変換(Dongarra法，行列とベクトルの乗算を含む)

(2)Bischof/Wuのアルゴリズム＋村田法

2. 以下の問題を解く

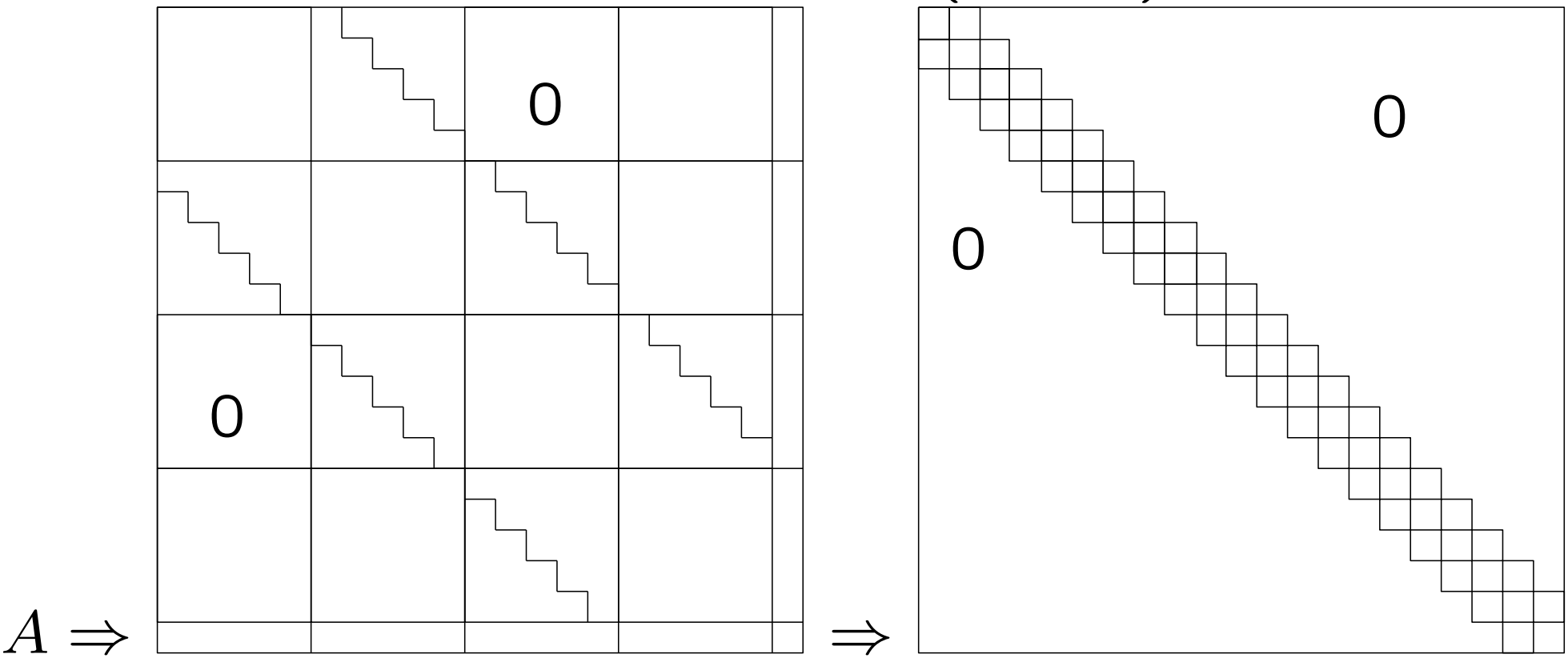
$$TV' = V'D, \quad V'^{\top}V' = V'V'^{\top} = I_n$$

3. 逆変換によりV'からVを得る

Bischof/Wuのアルゴリズム＋村田法

$n \times n$の実対称行列Aを，**ほぼ行列行列乗算のみ**を用いて帯行列に変換し(Bischof/Wuアルゴリズム)，

帯行列を３重対角行列Tに変換する(村田法)



Bischof/Wuアルゴリズムの計算量は，$O(n^3)$.

村田法の計算量は，$O(Ln^2)$，Lは，帯幅.

Bischof/Wuアルゴリズムは，行列行列乗算を基本としているため，スカラ型CPUで高速に実行可能である.

Lが小さければ，村田法の計算量は無視できるため，近年では，固有値計算においても，行列行列乗算が有効に活用されるようになった

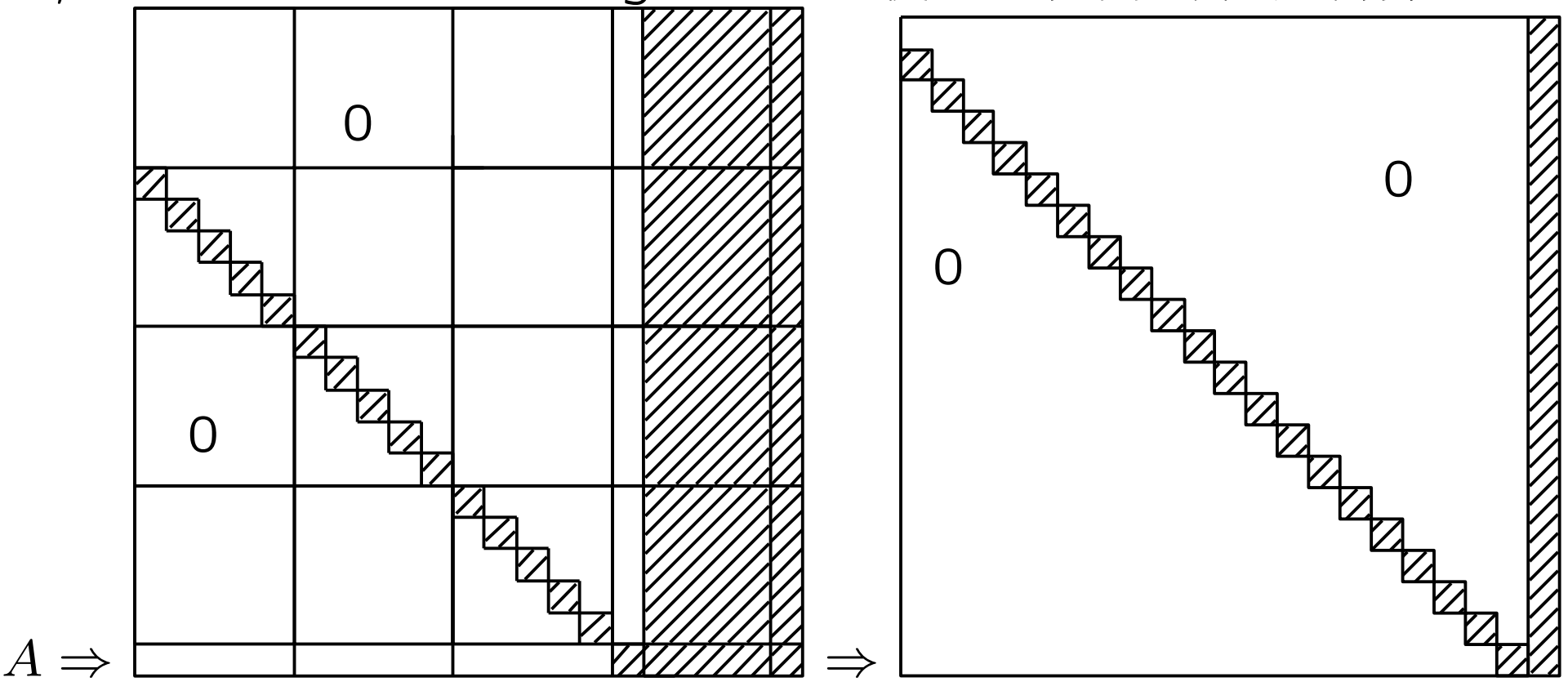
このようなアルゴリズムを，**two-stage algorithm**という

数式処理における two-stage algorithm

有限体$\mathbb{Z}/p\mathbb{Z}$上において，$N \times N$の非対称**密行列**Aの最小多項式を計算する，

固有多項式の次数＝最小多項式の次数を仮定，

$N \times N$の非対称行列Aを，行列乗算を利用して帯コンパニオン行列Cに変換し，CからWiedemann algorithmを使って最小多項式を計算する



タイミングデータ

$N \times N$の整数行列の最小多項式の計算時間の比較

各要素は, [1,10]の整数

N	Wiedemann	two-stage	Hensel
250	0.116sec	0.579sec	0.263sec
500	1.546sec	1.983sec	2.757sec
750	7.026sec	5.891sec	12.088sec
1000	22.816sec	14.785sec	36.994sec
1250	61.798sec	33.538sec	86.468sec

Intel(R) Core(TM) i7-4850HQ CPU @ 2.30GHz,

L4 キャッシュ:128MB, Wiedemann algorithmが高速に動作するCPU,

Mem 16GB, Fodora 20

dgem2vuの話は無駄では?

two-stage algorithmが，単純なWiedemann algorithmに勝てる可能性が高いのは，

密行列を対象として，固有多項式の次数＝最小多項式の次数が成立する場合のみ

どちらか1つでも条件が揃わなければ，two-stage algorithmが，単純なWiedemann algorithmに負ける可能性もある

Hensel構成

有限体上で，一次従属ならば，整数の世界でも，一次従属であろうと思い込み(候補の計算であるから)，

v_0を乱数ベクトルとして，$\mathbb{Q}$上において，連立一次方程式

$$\left(A^{k-1}v_0 \middle| A^{n-2}v_0 \middle| \cdots \middle| Av_0 \middle| v_0 \right) \begin{pmatrix} c_{k-1} \\ \vdots \\ c_0 \end{pmatrix} = -A^k v_0$$

を生成する，それを，Hensel構成(p進整数)によって解く

v_0がgenericならば，真の最小多項式を得る

3つのアルゴリズムの計算時間の差の要因

(1) 計算量が違う

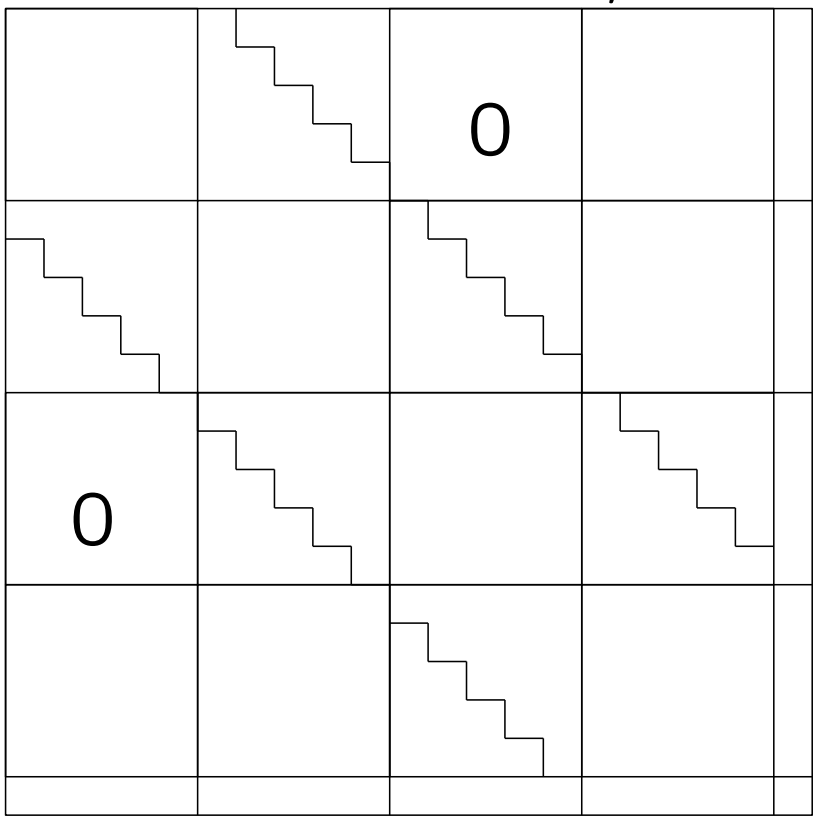
(2) どの演算器を使っているか

1. 整数 AVX2
2. 浮動小数点数 AVX2
3. 古典的64bits整数レジスタ

余談:two-stage algorithmの別の利点

最小固有値のみが欲しい人

$n \times n$の実対称行列Aを，帯行列に変換する

$A \Rightarrow$ 

$\Rightarrow$ 帯行列に対する安定な2分法

帯行列に対する安定な2分法は，日本人には既知，実装は複雑

SIMD演算の話

SIMD(single instruction multiple data)演算の利用について

$$IP \leftarrow \vec{x} \cdot \vec{y}$$
$$\vec{y} \leftarrow \alpha \vec{x} + \vec{y}$$

このような演算は, SIMD演算(SSE,AVXなど)を用いてベクトル化できる

SIMD演算というと, 行列演算やグラフィック処理のことばかりが例題に上る

$$s = \min_i a_i$$

という演算も，SIMD演算を用いてベクトル化できる

$s_0 = a_0, s_1 = a_1, s_2 = a_2, s_3 = a_3$と初期化

$s_0 = \min(s_0, a_4), s_1 = \min(s_1, a_5), s_2 = \min(s_2, a_6), s_3 = \min(s_3, a_7)$

$s_0 = \min(s_0, a_8), s_1 = \min(s_1, a_9), s_2 = \min(s_2, a_{10}), s_3 = \min(s_3, a_{11})$

...

$s = \min(s_0, s_1, s_2, s_3)$

なぜ，$s = \min_i a_i$をSIMD演算の例題として扱わないのか?

C言語の現状

```
tmp=A[0];
for(i=step;i<N;i=i+step){
    tmp=A[i] < tmp ? A[i]:tmp;
}
```

このような複数の文を理解して，ユーザーのやりたいことが，$s = \min_i a_i$であると判断できるCコンパイラは，未だ存在しない

FORTRANならば

Fortran

変数＝minval(“配列名” (開始位置：終了位置：ストライド))

とすると, SIMD演算を用いて, 高速に “配列名” の最小値を計算してくれるコードを自動的に生成

このＣ言語の欠点はどのくらい認知されているのか?

少なくとも，Intel社は理解している

配列表記（Array Notation）

変数＝__sec_reduce_min(配列名[開始位置： 要素数： ストライド]);

マニュアルより

“配列表記（Array Notation）は、インテル C++ コンパイラーのバージョン 12 でサポートされた新しい言語拡張です。**これらの機能を利用した場合、他のコンパイラーではコンパイルできなくなります。**”

GNUのコンパイラのみで頑張る方法

minvalを含むFortranのソースコードを以下のようにコンパイルする

gfortran -O3 -mtune=native -march=native -c program_min.f

program_min.oというオブジェクトコードを作成する

gcc -O3 -mtune=native -march=native -o test test.c program_min.o

として, Fortranコンパイラに作ってもらったコードをC言語から呼び出せばなんの問題もない

２次元配列(行列)を確保する方法の話

C89の場合

double A[5][4];

は，OK

一方，

int N=atoi(argv[1]);

double A[N][N];

とは書けない

仕方がないから

```
double **A;
A=(double **)malloc(sizeof(double *)*N);
for(i=0;i<N;i++){
A[i]=(double *)malloc(sizeof(double)*N); }
```

とすると，メモリの連続アクセス性が確保できない

本当は，2次元の配列が使いたいのに，

```
double *A;
A=(double *)malloc(sizeof(double)*N*N);
```

として，A[i*N+j] とプログラムの至る所に書くと，コンパイラのアドレス計算の技量によっては，悪いコードが生成される

C99の場合

```
int N=atoi(argv[1]);
double A[N][N];
```

と書けるが，この方法で配列が確保されるのは，動的領域スタックサイズを自由に変更できる環境にない場合には，巨大な行列を扱えない

静的領域に格納したいから，

static double A[N][N];

と書くと，この文は，コンパイラエラーになる

正しいやり方

C99限定

int N=atoi(argv[1]);

double (* restrict A)[N];

A=(double (* restrict)[N])malloc(sizeof(double)*N*N);

Aは，要素数Nを1単位としたものが，複数個連なっているものの先頭アドレスを指し示す

これ以降，A[i][j]として，アクセス可能

restrictってなに?

Fortran信者がC言語に対して言うこと

「double precision A(N,N),B(N,N)と書けば，Fortranならば，配列Aと配列Bの領域は違うことが分かるため，それをヒントにコンパイラが最適化を行ってくれる，C言語には，ポインタがあるからそれができないでしょ」

restrictとは，そのポインタが指している領域は，そのポインタにしか指されていないことを確約する修辞語

中国剰余定理による数式処理ソフトの高速化の話

中国剰余定理

未知の整数をXとする

$$X \bmod 2 = 1$$

$$X \bmod 3 = 2$$

この情報から，**中国剰余定理**により

$$X = \cdots, -13, -7, -1, 5, 11, 17, \cdots$$

が，Xの候補となる，解は一意に定まらない

もし，Xは1桁の整数であるという付加情報が手に入ったとすると，$X = -7, -1, 5$，まだ，3つも候補が残っている

もうすこし情報が手に入ったとする

$$X \bmod \underline{2} = 1$$
$$X \bmod \underline{3} = 2$$
$$X \bmod \underline{5} = 3$$

よって，答えは，中国剰余定理と付加情報により$X = -7$と一意的に定まる，**下線の部分には，素数を使う**

(付加情報)行列式に付随した2つの上界公式

2つのノルムを定義する: qは, 多変数多項式

$$\|q\|_1 = \sum_{\alpha_1,\cdots,\alpha_s} |c(\alpha_1,\cdots,\alpha_s)|,$$
$$\|q\|_2 = \sqrt{\sum_{\alpha_1,\cdots,\alpha_s} c(\alpha_1,\cdots,\alpha_s)^2},$$

ここで,

$q = \Sigma_{\alpha_1,\cdots,\alpha_s} c(\alpha_1,\cdots,\alpha_s) x_1^{\alpha_1}\cdots x_s^{\alpha_s} \in \mathbb{Z}[x_1,\cdots,x_s]$

$A = (a_{i,j}) \in \mathbb{Z}^{N \times N}$ についてのHadamardの上界は,

$$|\det(A)| \leq \min \left\{ \prod_{i=1}^{N} \sqrt{\sum_{j=1}^{N} |a_{i,j}|^2}, \prod_{j=1}^{N} \sqrt{\sum_{i=1}^{N} |a_{i,j}|^2} \right\}$$

$A = (a_{i,j}) \in \mathbb{Z}[x_1, \cdots, x_s]^{N \times N}$ についてのGoldsteinとGrahamの上界は,

$$\|\det(A)\|_2 \leq \min \left\{ \prod_{i=1}^{N} \sqrt{\sum_{j=1}^{N} \|a_{i,j}\|_1^2}, \prod_{j=1}^{N} \sqrt{\sum_{i=1}^{N} \|a_{i,j}\|_1^2} \right\}$$

固有多項式の係数の上界もこの式から得られる

$$B = \begin{vmatrix} a & 2b \\ 3c & 4d + f \end{vmatrix} = \underline{4}ad + \underline{1}af - \underline{6}bc$$

G.&G.の上界$= \min(\sqrt{1+4}\sqrt{9+25}, \sqrt{1+9}\sqrt{4+25}) <$ 13.1

固有多項式の中国剰余定理による計算法

整数行列Aの固有多項式$f(x) = \det(xI - A)$を計算

$$\begin{aligned} f_1(x) &= f(x) \bmod p_1 \\ f_2(x) &= f(x) \bmod p_2 \\ &\vdots \end{aligned}$$

中国剰余定理により，$f(x)$を復元

有限体$\mathbb{Z}/p\mathbb{Z}$の実装の話

技法.1:C言語のGNU拡張を使う方法

$p, a, b \in \mathbb{Z}, p < 2^{63}, 0 \leq a, b < p, a \oplus b = (a + b) \bmod p$と定義する

$0 \leq a_0, \cdots, a_{99999999} < p$について, 総和を考える

$$r = a_0 \oplus a_1 \oplus \cdots \oplus a_{99999999}$$

間違った実装の方法

```
unsigned long long a[1000000],r;
r=0;
for(i=0;i<1000000;i++)
    r=(r+a[i]) % p;
```

正しい実装の方法

$$
\begin{aligned}
r &= a_0 \oplus a_1 \oplus \cdots \oplus a_{99999999} \\
&= (a_0 + a_1 + \cdots + a_{99999999}) \bmod p
\end{aligned}
$$

```
typedef unsigned int uint128_t __attribute__((mode(TI)));
unsigned long long a[1000000],r;
uint128_t t=0;
for(i=0;i<1000000;i++) t+=a[i];
r=t % p;
```

技法.2:浮動小数点数を使う方法

Basic Linear Algebra Subprograms (BLAS)

：倍精度浮動小数点数のベクトルと行列に関する基本線型代数操作ライブラリ

倍精度浮動小数点数：-1.0×2^{53}から1.0×2^{53}までの整数を正確に表現可能

代表元を，$-\frac{p-1}{2}$から$\frac{p-1}{2}$の中で選ぶことにすると，行列サイズをNとするとき，内積の絶対値最大に対する条件は，

$$N\left(\frac{p-1}{2}\right)^2 \leq 2^{53}$$

$$p \leq 2\sqrt{\frac{2^{53}}{N}} + 1$$

この不等式を満たすpを採用すれば，BLASを利用可能

技法.3:整数のためのSIMD演算ユニットを使う方法

$0 < p \leq \sqrt{\frac{2^{64}-1}{N}} + 1$を満たす素数$p$を法とする有限体$\mathbb{Z}/p\mathbb{Z}$を考える,

以下のプログラムは, 整数のためのSIMD演算ユニットを有効に活用できる

```
typedef unsigned int UI;
typedef unsigned long long ULL;
UI a[N],b[N],r,p;
ULL t=0;
for(i=0;i<N;i++) t+=(ULL)a[i]*b[i];
r=t % p
```

技法.4:整数の剰余計算の高速化

$c_i = a_i \bmod p$を考える

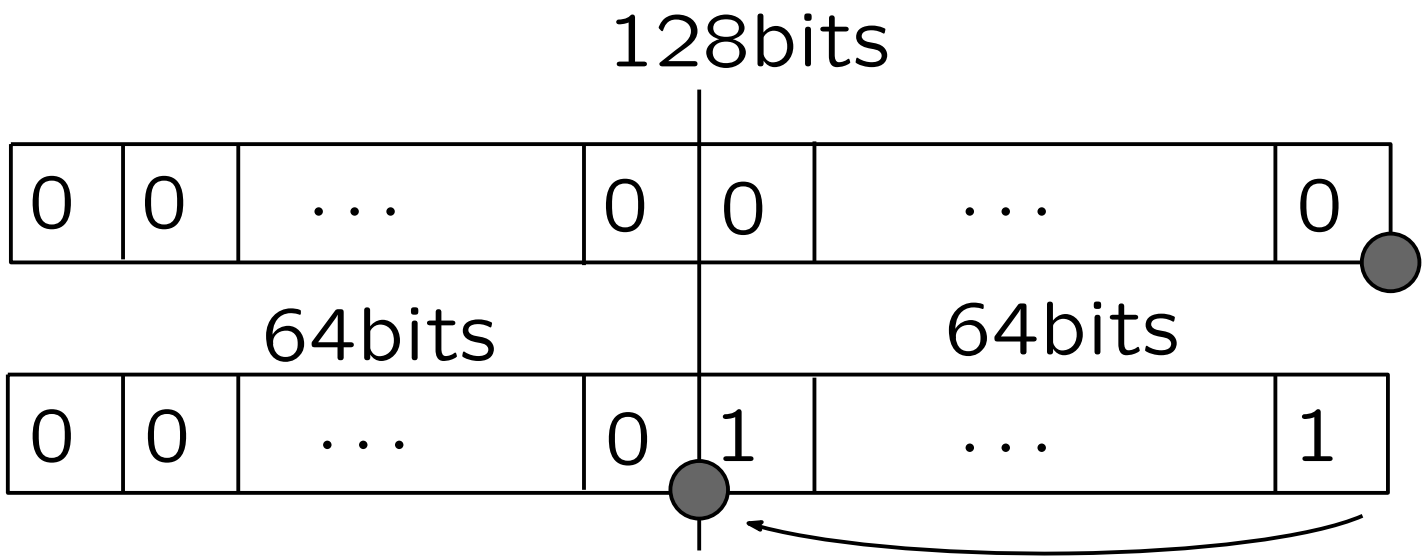
$0 \leq c_i < p$を要求されると，CPUの命令をそのまま呼び出す以外に方法はない

もし，c_iが，$0 \leq c_i < 2p$の範囲にあればよいという緩い条件の場合には，次のようにして演算を高速化することが可能である

$$M = \frac{2^{64} - 1}{p} \quad \text{(切り捨て演算)}$$

を用意する，Mを，擬似逆元という

擬似逆元の意味を考える，原点を移動する



$0.1111111111111111111111111111111111111111111111111111111111111111_{(2)} \approx 1.0$

より，$M = (2^{64} - 1)/p \approx 1.0/p \times 2^{64}$

pが一定ならば，Mも一定

$(a_i \times M)$の結果の上位64bits

は，a_i/pの除算をした時の切り捨て演算における真の商に等しいか，または，切り捨て演算における真の商より1小さい(ただし，負の数にはならない)

$c_i = a_i - p \times (a_i \times M$の結果の上位64bits)

64bitsの数同士の乗算と計算結果の上位64bitsの取り出し

インラインアセンブリ言語，xxxは，ダミーの変数

```
__asm__ ("mulq %3":"=a"(xxx),"=d"(r):"a"(a1),"g"(a2))
```

Tropical Determinantの話

行列式の定義

A:n次正方行列$A = (a_{ij})$に対して，その行列式det Aを

$$\det A = |A| = \sum_{\sigma \in S_n} \mathrm{sgn}(\sigma) a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)}$$

により定義する

Permanentの定義

A:n次正方行列$A = (a_{ij})$に対して，その行列式per Aを

$$\mathrm{per}\, A = \sum_{\sigma \in S_n} a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)}$$

により定義する

ultradiscretization

以下の規則に従って，演算を変更すること

$$a + b \rightarrow \max(a, b)$$
$$a \times b \rightarrow a + b$$
$$a/b \rightarrow a - b$$

ultradiscrete permanentの定義

A:n次正方行列$A = (a_{ij})$に対して，その行列式udper Aを

$$\text{udper } A = \max_{\sigma \in S_n} a_{1\sigma(1)} + a_{2\sigma(2)} + \cdots + a_{n\sigma(n)}$$

により定義する

ultradiscrete permanentは，Tropical Determinantとも呼ばれている

Tropical Determinantの計算アルゴリズム

Linear Assignment Problem

最短路問題に帰着

Tomizawa, N. : On some techniques useful for the solution of transportation problems. Networks 1, 173 194 (1971).

Jonker-Volgenant algorithm(LAPJV), 1987

Tropical Determinantの数式処理ソフトへの応用

$$A = \begin{vmatrix} 2x^3+1 & x^3+6x^2+5 \\ 7x^5+11x^4+3 & 9x^3 \end{vmatrix}$$

より,

$$B = \begin{pmatrix} 3 & 3 \\ 5 & 3 \end{pmatrix}$$

を得る, BのTropical Determinantを計算することで, Aの計算結果の最大次数の上界が手に入る

Newton補間の話

Newton補間の実装法

1変数n次多項式$f(x)$のNewton補間

$$f(x) = f[x_0] + (x - x_0)f[x_0, x_1] + ... \\ +(x - x_0) \cdots (x - x_n)f[x_0, x_1, x_2, ..., x_n]$$

における係数$f[x_0, x_1, x_2, ..., x_n]$は，1階の差分商を，

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0},$$

2階の差分商を，

$$f[x_0, x_1, x_2] = \frac{f[x_0, x_2] - f[x_0, x_1]}{x_2 - x_1},$$

と定義し，n階の差分商を

$$f[x_0, x_1, ..., x_n] = \frac{f[x_0, x_1, ..., x_{n-2}, x_n] - f[x_0, x_1, ..., x_{n-2}, x_{n-1}]}{x_n - x_{n-1}},$$

と定義する.

$x_n - x_{n-1}$などを逆元として保持するテクニックは使える，しかし，$\mathbb{Z}/p\mathbb{Z}$では，桁溢れをおこす危険を回避するため，1回の差分商毎に，剰余演算を行う必要がある

Newton補間の正しい実装法(1)

$(x_0, f(x_0)), (x_1, f(x_1)), \cdots, (x_N, f(x_N))$が与えられているとき，すべての点を通る多項式

$$f(x) = f_0 + (x - x_0)f_1 + (x - x_0)(x - x_1)f_2 + \cdots + (x - x_0)(x - x_1)\cdots(x - x_N)f_N$$

を求めよ

$$f(x_0) = f_0, f(x_1) = f_0 + (x_1 - x_0)f_1$$

より,

$$f_1 = \frac{f(x_1) - f_0}{x_1 - x_0}$$

$$f(x_2) = f_0 + (x_2 - x_0)f_1 + (x_2 - x_0)(x_2 - x_1)f_2$$

より,

$$f_2 = \frac{f(x_2) - (f_0 + (x_2 - x_0)f_1)}{(x_2 - x_0)(x_2 - x_1)}$$

多項式の評価$f_0 + (x_2 - x_0)f_1$が, このアルゴリズムでは重要な役割を果たす

Newton補間の正しい実装法(2)

$$f(x_j) = f_0 + (x_j - x_0)f_1 + (x_j - x_0)(x_j - x_1)f_2 + \cdots + (x_j - x_0)(x_j - x_1)\cdots(x_j - x_N)f_N$$

$(x_j - x_0), (x_j - x_0)(x_j - x_1), \cdots, (x_j - x_0)(x_j - x_1)\cdots(x_j - x_N)$をあらかじめ計算しておくとすると，多項式の評価とは，

$$f(x_j) = 1 \times f_0 + \left\{(x_j - x_0)\right\} \times f_1 + \left\{(x_j - x_0)(x_j - x_1)\right\} \times f_2 + \cdots + \left\{(x_j - x_0)(x_j - x_1)\cdots(x_j - x_N)\right\} \times f_N$$

ベクトルの内積と思うことができる，**内積は，整数のSIMD演算ユニットを使って実装できる**

Hyper-Threading Technologyの話

```
1.x1=(ULL)a[i]*(ULL)b[i]+(ULL)c[i]
2.__asm__ ("mulq %3":"=a"(xxx),"=d"(x2):"a"(x1),"g"(INV_CMX))
3.z[i]=(UI)x1-(UI)x2*(UI)CM
```

すなわち, $z[i] = \mathrm{mod}(a[i] * b[i] + c[i], p)$ をおこなう (ULL=unsigned long long,UI=unsigned int), i の動く範囲が小さい場合には, 1.2.3. は, ベクトル化できない

“CPUの資源を使い切ることができない場合の定石” (**メモリバンド幅依存のプログラムでCPUの資源を使い切ることができない場合を除く**)

Hyper-Threading Technologyを活用する

Intel Hyper-Threading Technology(HTT)とは

Intel社のweb pageより，Hyper-Threading Technologyの紹介を引用する．『オペレーティング・システムがより多くの処理をより高いパフォーマンスで実行できる仕組みインテル ハイパースレッディング・テクノロジー (インテル HT テクノロジー)は1つのコアで複数のスレッドを同時に実行することにより、プロセッサーのリソースをより効率的に使用します。また、パフォーマンス面でも、インテル HT テクノロジーはプロセッサーのスループットを高め、マルチスレッド・ソフトウェアの全体的なパフォーマンスを改善します。』

ベンチマーク問題

```
E6(a)=a^27+12*p2*a^25+60*p2^2*a^23-48*p1*a^22+(168*p2^3+96*q2)*a^21
-336*p2*p1*a^20+(294*p2^4+528*q2*p2+480*p0)*a^19+(-1008*p2^2*p1-1344*q1)*a^18
+(144*p1^2+336*p2^5+1152*q2*p2^2+2304*p0*p2)*a^17
+((-1680*p2^3-768*q2)*p1-5568*q1*p2)*a^16
+(608*p2*p1^2+252*p2^6+1200*q2*p2^3+4768*p0*p2^2+17280*q0-1248*q2^2)*a^15
+((-1680*p2^4-2688*q2*p2+2304*p0)*p1-8832*q1*p2^2)*a^14
+(976*p2^2*p1^2+3264*q1*p1+120*p2^7+480*q2*p2^4+5696*p0*p2^3+
(43776*q0-4800*q2^2)*p2+12288*q2*p0)*a^13
+(832*p1^3+(-1008*p2^5-3072*q2*p2^2+5888*p0*p2)*p1-6528*q1*p2^3
+10752*q2*q1)*a^12
+((704*p2^3+4224*q2)*p1^2+2688*q1*p2*p1+33*p2^8-144*q2*p2^5+4384*p0*p2^4
+(41472*q0-6720*q2^2)*p2^2+34560*q2*p0*p2-34560*p0^2)*a^11
+(2560*p2*p1^3+(-336*p2^6-768*q2*p2^3+3584*p0*p2^2+64512*q0+8448*q2^2)*p1
-2112*q1*p2^4+23040*q2*q1*p2-70656*p0*q1)*a^10
+((176*p2^4+8960*q2*p2-18944*p0)*p1^2-5504*q1*p2^2*p1+4*p2^9-192*q2*p2^6
+2176*p0*p2^5+(22528*q0-3840*q2^2)*p2^3+32768*q2*p0*p2^2-39936*p0^2*p2
+110592*q2*q0-40704*q1^2+5120*q2^3)*a^9
```

```
+(2688*p2^2*p1^3+4608*q1*p1^2+(-48*p2^7+768*q2*p2^4-1536*p0*p2^3
+(82944*q0+16128*q2^2)*p2-73728*q2*p0)*p1-192*q1*p2^5+13824*q2*q1*p2^2
-64512*p0*q1*p2)*a^8
+(-2560*p1^4+(-32*p2^5+5376*q2*p2^2-16384*p0*p2)*p1^2+(-6144*q1*p2^3
-15360*q2*q1)*p1-48*q2*p2^7+608*p0*p2^6+(9600*q0-480*q2^2)*p2^4
+10752*q2*p0*p2^3-20992*p0^2*p2^2+(156672*q2*q0-38400*q1^2+9984*q2^3)*p2
-165888*p0*q0-56832*q2^2*p0)*a^7
+((1024*p2^3-10240*q2)*p1^3+10240*q1*p2*p1^2+(384*q2*p2^5-1792*p0*p2^4
+(21504*q0+6912*q2^2)*p2^2-57344*q2*p0*p2+49152*p0^2)*p1+1536*q2*q1*p2^3
-19456*p0*q1*p2^2-110592*q1*q0-21504*q2^2*q1)*a^6
+(-1536*p2*p1^4+(-16*p2^6+768*q2*p2^3-4608*p0*p2^2+27648*q0-19200*q2^2)*p1^2
+(-1344*q1*p2^4+10752*q2*q1*p2-9216*p0*q1)*p1+64*p0*p2^7
+(2304*q0+192*q2^2)*p2^5-3072*p0^2*p2^3+(55296*q2*q0-12288*q1^2
+4608*q2^3)*p2^2+(-110592*p0*q0-46080*q2^2*p0)*p2+73728*q2*p0^2)*a^5
+((64*p2^4-4096*q2*p2+8192*p0)*p1^3-512*q1*p2^2*p1^2+(-256*p0*p2^5+
(3072*q0-768*q2^2)*p2^3-8192*q2*p0*p2^2+16384*p0^2*p2+73728*q2*q0
-39936*q1^2-18432*q2^3)*p1-1024*p0*q1*p2^3+(-36864*q1*q0-3072*q2^2*q1)*p2
+24576*q2*p0*q1)*a^4
+(256*p2^2*p1^4+15360*q1*p1^3+(128*q2*p2^4-1024*p0*p2^3+(-6144*q0
```

```
-2560*q2^2)*p2+8192*q2*p0)*p1^2+(-128*q1*p2^5+2048*q2*q1*p2^2
-14336*p0*q1*p2)*p1+256*q0*p2^6-256*q2*p0*p2^5+256*p0^2*p2^4+(9216*q2*q0
-2560*q1^2-256*q2^3)*p2^3+(-18432*p0*q0-7680*q2^2*p0)*p2^2
+24576*q2*p0^2*p2-110592*q0^2+55296*q2^2*q0-30720*q2*q1^2-16384*p0^3
-6912*q2^4)*a^3
+(-1024*p1^5+4096*p0*p2*p1^3+24576*q2*q1*p1^2-12288*q1^2*p2*p1)*a^2
+(-2048*q2*p1^4+2048*q1*p2*p1^3+((-3072*q0-256*q2^2)*p2^2+4096*q2*p0*p2
-4096*p0^2)*p1^2+(512*q2*q1*p2^3-1024*p0*q1*p2^2-36864*q1*q0
+9216*q2^2*q1)*p1-256*q1^2*p2^4-6144*q2*q1^2*p2+12288*p0*q1^2)*a
+(4096*q0-1024*q2^2)*p1^3+(2048*q2*q1*p2-4096*p0*q1)*p1^2
-1024*q1^2*p2^2*p1-4096*q1^3
```

$E6(a)$の判別式を計算する

タイミングデータ

$E6_k(a) = E6(a) \bmod a^{k+1}$として, 性能評価

計算機環境は, CPU:Intel Core i7 980X(6 Core), Mem:24G,

OS:Fedora 13

GNU GCC compiler 4.8.2 Option:-O3 -mtune=native

-march=native -fopenmp

k	Kimura Serial	Kimura Parallel without HTT	Kimura Parallel with HTT
7	5m46.000s	1m13.400s	52.923s

Intel C++ compiler 14.0.1 Option:-fast -openmp

k	Kimura Serial	Kimura Parallel without HTT	Kimura Parallel with HTT
7	6m11.804s	1m11.837s	52.634s

6 CoreのCPUにおいて，6倍以上の性能を達成している (super-linear)

$E6(a)$の判別式計算

CPU:Intel Core i7 980X(6Core)

Mem:24G

Compiler:GCC 4.8.0

Option:-O3 -mtune=native -march=native -fopenmp

計算結果の項数:27329463項(txt形式:2.5G)

Serial: 10913m45.857s

Parallel: 1773m28.272s

Speed Up: 6.15 **“superlinear”**