数理解析研究所講究録
第1907巻 2014年 154-165

Intel Hyper-Threading Technologyと数式処理に関する2つの話題

木村欣司
KINJI KIMURA
京都大学大学院情報学研究科
GRADUATE SCHOOL OF INFORMATICS, KYOTO UNIVERSITY

1 はじめに

Intel Hyper-Threading Technologyは, アプリケーションの実行時の性能を向上させるためのIntel社のCPUに搭載されている機能である. 実際には, すべてのアプリケーションに有効というわけではなく, 逆に, 実行時の性能を悪化させるものも存在することが報告されている.[1] 「数式処理ソフト」というアプリケーションについて, Intel Hyper-Threading Technologyの効果を検証してみたところ, ユーザーが必要とする数式処理ソフトの関数毎に, その有効性が決まるという事実が分かったので, それを報告する.

2 Intel Hyper-Threading Technologyとは

Intel社のweb pageより, Hyper-Threading Technology(HTT)の紹介を引用する.[2] 『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。』

並列計算において, CPUの資源を有効に活用できない場合, 著者の経験においては, メモリからCPUにデータが到着しないことによりリソースが有効に活用されない場合を除く, それ以外の場合において, 有効に活用されないリソースが存在するならば, Hyper-Threading Technologyは, それを活用する非常に有効な手段である.

具体的に, ベクトルや行列に対する並列演算について, 考察してみる. ベクトルの内積については, データの再利用性がないため, メモリからCPUにデータが到着しないことによりリソースが有効に活用されない場合に当てはまる. よって, Hyper-Threading Technologyの効果は期待できない. 行列とベクトルの乗算についても概ねデータの再利用性がないためベクトルの内積と同様である. 行列と行列の乗算については, データの再利用性が高

い為, メモリからCPUにデータが到着しないことによりリソースが有効に活用されないということは起きない. しかし, 行列と行列の乗算には, Intel Math Kernel Library[3] という優れた実装が存在し, その実装においては, 完全にCPUのリソースを使い切れている. よって, 行列と行列の乗算においても, Hyper-Threading Technologyの効果は期待できない.

逆に, 明らかに, Hyper-Threading Technologyの効果が期待される計算も存在する. 複数の桁長の長い多倍長整数の計算を並行して行う場合には, メモリからCPUにデータが到着しないために計算が進まないということは起きず, CPU内での計算がボトルネックになる. さらに, CPU内では, 命令順序の依存などによって, すべての回路資源を有効に活用することができず, 有効に活用されないリソースが存在するという状況が起きる. よって, Hyper-Threading Technologyは有効に機能する. 複数の桁長の長い多倍長整数の並行計算は, Hensel構成という数式処理ソフトの高速化テクニックにおいて重要であり, Hyper-Threading Technologyが数式処理ソフトにおいても, 有益な役割を果たすと予想される. また, 高速な整数の剰余の計算においても, 上記の多倍長整数の並行計算と似た計算が現れるため, Hyper-Threading Technologyの効果が期待される.

しかし, 整数行列の最小多項式の候補の計算においては, Hensel構成による高速化と中国剰余定理による高速化の2つの高速化をそれぞれ選択できる. Hensel構成による高速化のみを知っていた場合には, 整数行列の最小多項式の計算において, Hyper-Threading Technologyは有効に機能するという結論になるが, 中国剰余定理による高速化では, Intel Math Kernel Libraryを活用した実装が考えられ, 結果的に, Hyper-Threading Technologyは活用されない. すなわち, 「Hyper-Threading Technologyが有効に機能するアルゴリズムは, その目的のための最良のアルゴリズムである」という結論にはならない. このように, Hyper-Threading Technologyは, 広い視野を持って議論するべき話題である.

3 Intel Math Kernel Libraryについて

Intel社の日本の販売代理店であるxlsoftのweb pageより, Intel Math Kernel Libraryの紹介を引用する.[3] 『工学、科学、金融系アプリケーションのパフォーマンスを最大限に引き出すために、高度に最適化され、広範囲にスレッド化された演算ルーチンのライブラリーです。』

さらに, そのweb pageには, どのような機能が含まれているか記載されている. 本稿で注目するのは, BLASが含まれていることである. BLASについて, [4] には以下の記述がある.

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development

of high quality linear algebra software, LAPACK for example.

netlib[5] から提供される BLAS は, reference implementation と呼ばれ, CPU の利用効率は高くない. ユーザーは, 各 CPU のベンダーから提供される最適化された BLAS を用いる. その最適化された BLAS の一つが, Intel Math Kernel Library である.

4 剰余算の高速化

具体的な数式処理ソフトの関数について, Hyper-Threading Technology の有効性を議論する前に, 数式処理ソフトの高速化において重要な役割を果たす Hensel 構成による高速化と中国剰余定理による高速化の 2 つの高速化の根幹を支える技術である剰余算の高速化について議論する.

4.1 C 言語の GNU 拡張を使う方法

$p, a, b \in \mathbb{Z}$, p を $p < 2^{63}$ を満たす整数とする. $0 \leq a, b < p$ に対して, $a \oplus b = (a+b) \bmod p$ と定義する. $0 \leq a_0, \cdots, a_{99999999} < p$ に対して, $\oplus$ についての総和を考える.

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

4.1.1 間違った実装の方法

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

4.1.2 正しい実装の方法

以下のような考えに基づいて, 実装を行う.

$$r \quad = \quad a_0 \oplus a_1 \oplus \cdots \oplus a_{99999999} = (a_0 + a_1 + \cdots + a_{99999999}) \bmod p$$

```
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;
```

『C 言語の整数変数の型（整数を保持する領域の種類）は, char(8bits),short(16bits),int(32bits),long(64bits/32bits) の 4 種類である』というのが, 全ての C 言語の教科書の記述である. しかし, それは事実に反する.

```
typedef unsigned int uint128_t __attribute__((mode(TI)));
```

と書くと, 最近の GNU GCC compiler では, 128bits の整数演算が使えるようになる.

4.2 浮動小数点数を使う方法

倍精度浮動小数点数は, 0から 1.0×2^{53} までの整数を正確に表現可能である. 行列サイズを N とするとき, $0 < p \leq \sqrt{\frac{2^{53}}{N}} + 1$ を満たす素数 p を法とする有限体 $\mathbb{Z}/p\mathbb{Z}$ を考える. BLASが内積, 行列ベクトル乗算, 行列・行列乗算を計算した後, p で剰余算を行うことで有限体 $\mathbb{Z}/p\mathbb{Z}$ を実現できる.

なお, 倍精度浮動小数点数には, 符号bitが存在するため, -1.0×2^{53} から 1.0×2^{53} までの整数を正確に表現可能であるというほうがより正確である. すなわち, 扱う数 x を $-\frac{p-1}{2} \leq x \leq \frac{p-1}{2}$ と規格化することで, $0 < p \leq 2\sqrt{\frac{2^{53}}{N}} + 1$ とすることができる. しかし, 後述の剰余算の高速化においては, 有限体 $\mathbb{Z}/p\mathbb{Z}$ に対して, 剰余を計算した後の数を, 0から $2p-1$ の範囲に規格化する. 加えて, $-x$ を計算するとき, $2p-x$ を行うのみで済むように, 0から $2p$ の範囲で考える. 符号を有効に活用するために, $-p$ を行うことで, 扱う数 x を $-p \leq x \leq p$ とする. 以上より, $0 < p \leq \sqrt{\frac{2^{53}}{N}}$ の範囲に p を制約することが合理的である.

4.3 符号なし整数のためのSIMD演算ユニットを使う方法

後述の剰余算の高速化においては, 有限体 $\mathbb{Z}/p\mathbb{Z}$ に対して, 剰余を計算した後の数を, 0から $2p-1$ の範囲に規格化する. 加えて, $-x$ を計算するとき, $2p-x$ を行うのみで済むように, 0から $2p$ の範囲に規格化していることにする. すると, 符号なし整数のためのSIMD演算ユニットを利用する場合には, $0 < p \leq \frac{1}{2}\sqrt{\frac{2^{64}-1}{N}}$ 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, 符号なし整数のためのSIMD演算ユニットを有効したベクトル a_i とベクトル b_i の内積を計算する為のC言語のプログラムである.

```
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=後述の剰余算の高速化をつかってtのpによる剰余を計算
```

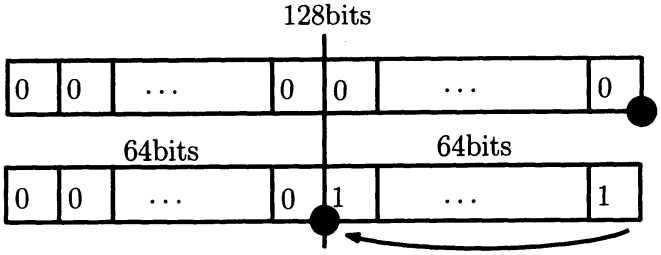
4.4 整数の剰余計算の高速化

$c_i = a_i \bmod p$ を考える. ただし, p を, $2^{64}-1$ 以下の整数とする. $0 \leq c_i < p$ 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.

$$M = \frac{2^{64}-1}{p} \quad \text{(ただし, この演算は切り捨て演算で行う)}$$

を用意する. M を, 擬似逆元と呼ぶことにする.

擬似逆元の意味を大まかに説明する. 原点を, 左へ 64bit 移動して考える.



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

より, $M = (2^{64}-1)/p \approx 1.0/p \times 2^{64}$. 注意として, p が一定ならば, M も一定である. $(a_i \times M)$ 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) ことが証明できる. 以上より, c_i は次のように計算できる,

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

次に, 『c_i は, $0 \leq c_i < 2p$ の範囲にあればよい』という緩い条件について議論する. 著者の経験では, 有限体 $\mathbb{Z}/p\mathbb{Z}$ 上のすべてのアルゴリズムにおいて, 内積を用いて記述することは困難であるが, $a \times b + c \bmod p$ の形式で記述することが可能なアルゴリズムは多い. また, $-x$ すなわち $p - x$ の演算を容易に可能にするためには, $0 \leq a, b, c \leq 2p$ と仮定するべきである. よって, $2p \times 2p + 2p = 2p(2p+1)$ が, $2^{64} - 1$ 以下という条件を満たせば, 問題は起きない. 以上より, 『c_i は, $0 \leq c_i < 2p$ の範囲にあればよい』という緩い条件は, 妥当な条件である.

4.4.1 64bits の数同士の乗算と計算結果の上位 64bits を取り出す演算の記述法

さまざまな CPU に対応する記述の方法として, GNU GCC compiler では, 以下のように記述する.

```
typedef unsigned int uint128_t __attribute__((mode(TI)));
unsigned long long a1,a2,r
r=((uint128_t)a1*a2) >> 64;
```

Intel 社の CPU に対して, GNU GCC compiler のインラインアセンブリ言語では, 以下のように記述する. xxx は, ダミーの変数である.

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

この"mulq" という演算は, 多倍長整数の演算でも重要な役割を果たす演算である. よって, "整数の剰余計算の高速化" が中心的な役割を果たすアルゴリズムでは, 実行時に Hyper-Threading Technology が有効に機能する可能性が高い.

5 Hyper-Threading Technologyが有効に機能するアルゴリズム

5.1 $\mathbb{Z}/p\mathbb{Z}$ 上の終結式の計算

x の多項式 $f(x), g(x)$ の終結式を計算することを考える, ただし, x の次数について $\mu = \deg_x f(x)$, $\nu = \deg_x g(x)$, $\mu \geq \nu$, $f(x), g(x)$ の係数が, $\mathbb{Z}/p\mathbb{Z}$ の元であるとする. Euclid の互除法

$$R_1(x) = f(x), R_2(x) = g(x), R_{i-1}(x) = Q_i(x)R_i(x) + R_{i+1}(x)$$

を行ったとき, $n_k = 0$ ならば, $\mathrm{lc}(R_k)^{n_{k-1}} \prod_{i=1}^{k-2} (-1)^{n_i n_{i+1}} \mathrm{lc}(R_{i+1})^{n_i - n_{i+2}}$ が終結式であり, $\mathrm{lc}(R_i)$ は, 多項式 R_i の最高次の係数である.

Euclid の互除法では, 剰余算が $O(n^2)$ 回必要になる. 剰余以外の演算も $O(n^2)$ 回必要なので, "整数の剰余計算の高速化" が必要である. よって, $\mathbb{Z}/p\mathbb{Z}$ 上の終結式の計算を大量に必要とするアルゴリズムは, Hyper-Threading Technology が有効に機能する可能性が高い.

5.2 多項式補間法による終結式の計算

多項式補間法は, 補間点でのデータのサンプリングと補間と中国剰余定理による係数の復元の3つのステージに分かれる. 補間点でのデータのサンプリングには, $\mathbb{Z}/p\mathbb{Z}$ 上の終結式の計算が大量に必要となる.

5.2.1 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.

5.2.2 タイミングデータ

$E6_k(a) = E6(a) \bmod a^{k+1}$ として, 性能評価を行う. 計算機環境は, CPU:Intel Core i7 980X(6 Core), Mem:24G, OS:Fedora 13 を用いる.

GNU GCC compiler 4.8.2 Option:-O3 -mtune=native -march=native -fopenmp では, 以下の結果が得られた.

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 では, 以下の結果が得られた.

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

以上より, 多項式補間法による終結式の計算では, Hyper-Threading Technology が有効に機能することが確かめられる. 実際には, 6 Core の CPU において, 6 倍以上の性能を達成している (super-linear).

6 Hyper-Threading Technology が有効に機能しないアルゴリズム

Hyper-Threading Technology が有効に機能しないアルゴリズムとして, 整数行列の最小多項式の候補の計算を紹介する.

6.1 最小多項式の候補とは?

与えられた行列 A に対して, $f(A)=0$ となる, 次数が最小の多項式 $f(x)=x^k+c_{k-1}x^{k-1}+\cdots+c_0$ を, 最小多項式という. あるベクトル v_0 について, $g(A)v_0=0$ となる次数が最小の多項式 $g(x)=x^k+d_{k-1}x^{k-1}+\cdots+d_0$ を, 最小消去多項式という. さらに, あるベクトル u_0, v_0 について, $u_0^\top h(A)v_0=0$ となる次数が最小の多項式 $h(x)=x^k+e_{k-1}x^{k-1}+\cdots+e_0$ を, 考える場合がある. $h(x)|g(x)|f(x)$ の関係がある. u_0, v_0 が generic(固有ベクトルの全成分を含む) ならば, $f(x)=g(x)=h(x)$. $g(x)$ や $h(x)$ を, 最小多項式の候補と呼ぶ.

6.2 Hensel 構成による $g(x)$ の生成法

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

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

を生成する. それを, Hensel 構成によって解く. v_0 が generic ならば, 真の最小多項式を得る. k は, あらかじめ, 有限体上において, ベクトルの一次独立性の問題を解いて決める.

6.3 中国剰余定理による $h(x)$ の生成法

あるベクトル u_0, v_0 について, $u_0^\top h(A)v_0 = 0$ となる次数が最小の多項式 $h(x) = x^k + e_{k-1}x^{k-1} + \cdots + e_0$ を求める. $s_i = u_0^\top A^i w_0$ とすると,

$$\begin{pmatrix} s_0 & s_1 & \cdots & s_{k-1} \\ s_1 & s_2 & \cdots & s_k \\ \vdots & \vdots & \vdots & \vdots \\ s_{k-1} & s_k & \cdots & s_{2k-2} \end{pmatrix} \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{k-1} \end{pmatrix} = - \begin{pmatrix} s_k \\ s_{k+1} \\ \vdots \\ s_{2k-1} \end{pmatrix} \tag{1}$$

$h(x)$ を求める問題は, Hankel 行列の連立一次方程式 (1) を解き

$$s_{2k} + e_{k-1}s_{2k-1} + \cdots + e_0 s_k = 0$$

を確認する問題と等価である. $k = k_0 + 1$ 時点で, 以下の行列が,

$$\begin{pmatrix} s_0 & s_1 & \cdots & s_k \\ s_1 & s_2 & \cdots & s_{k+1} \\ \vdots & \vdots & \vdots & \vdots \\ s_{k-1} & s_k & \cdots & s_{2k-1} \end{pmatrix},$$

始めて特異行列になるならば, k_0 の時点の解が, $h(x)$ になる. k_0 を求めて, k_0 における Hankel 行列の連立一次方程式 (1) を解く問題は, Padé 近似の問題と等しい. early terminated Berlekamp/Massey algorithm を使って解く. このアルゴリズムの計算量は $O(k^2)$ であるため, 主要な計算は, $s_i = u_0^\top A^i w_0$ のほうである. $s_i = u_0^\top A^i w_0$ の正しい計算法として,

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

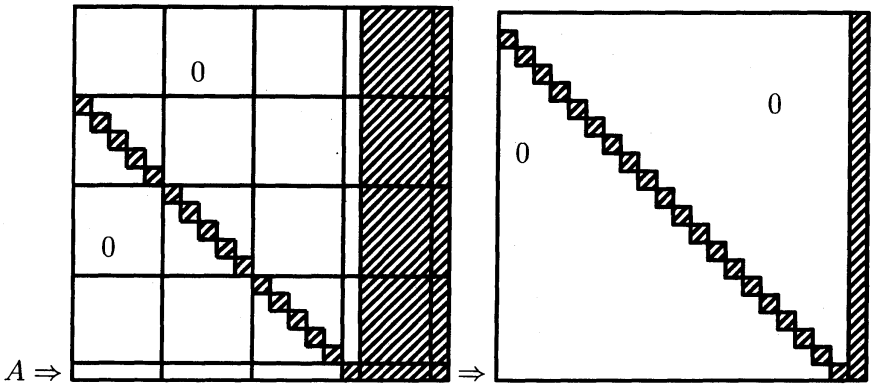
次の 2 系列を考える. $w_0, Aw_0, A^2w_0, \cdots$, $u_0, A^\top u_0, (A^\top)^2 u_0, \cdots$. この 2 系列は, 互いに無関係であるため, 並列に計算できる. さらに, Ax, $A^\top y$ は, 以下のように合理的に計算できる.

```
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; ‘‘整数の剰余計算の高速化’’を用いる
  }
  for (j = 0; j < N; j++) ATY[j]=ATY_TMP[j] % P; ‘‘整数の剰余計算の高速化’’
```
を用いる

6.4 ブロッククリロフ部分空間法

最小多項式の次数が行列サイズ N と等しい $N \times N$ の非対称行列 A を, 行列積を利用して帯コンパニオン行列 C に変換し, C から early terminated Berlekamp/Massey algorithm を用いて最小多項式の候補を計算する. 概念的に図示すると, 以下のようになる.



6.4.1 数式処理におけるブロックコンパニオン行列 C への変換法

(1)Dumas のアルゴリズムのブロック化, または, (2) ブロック Berlekamp-Massey アルゴリズムの 2 つの選択肢があるが, ここでは (1) を採用する.

6.4.2 帯コンパニオン行列 C から固有多項式を得る方法

(1)Berlekamp-Massey アルゴリズム, または, (2)N 回の行列式計算 (N サンプリングポイント) と補間の 2 つの選択肢があるが, ここでは (1) を採用する.

6.5 行列積を利用した固有多項式の計算法

$N \times NB(NB < N)$ の縦長の乱数行列 V を用意する. C を帯コンパニオン行列, C' を帯コンパニオン行列の右の部分とする. N と $NB \times l$ が等しくない場合には, 微調整が必要

であるが, $N = NB \times l$ の場合には,

$$\begin{pmatrix} V & AV & \cdots & A^{l-1}V \end{pmatrix} C = \begin{pmatrix} AV & A^2V & \cdots & A^lV \end{pmatrix}$$
$$= A \begin{pmatrix} V & AV & \cdots & A^{l-1}V \end{pmatrix}$$

この形の連立一次方程式を解くと, A と同じ固有値を持つ帯コンパニオン行列 C が手に入る. 実際には, 以下の連立一次方程式の問題を解けばよい,

$$\begin{pmatrix} V & AV & \cdots & A^{l-1}V \end{pmatrix} C' = A^lV.$$

6.6 実験

計算機環境として, CPU:Core i7-4770K 3.50GHz, Mem:32GB, OS:Fedota 19, Compiler:gcc 4.8.2, Option:-O3 -mtune=native -march=native -fopenmp, library:Intel Math Kernel Library 11.1 を用いる.

密行列, 各要素が 1 以上 10 以下の整数を乱数で生成する. 単位は sec.

n	$g(x)$ without HTT	$h(x)$ without HTT	ブロッククリロフ without HTT
250	0.212	0.084	0.396
500	2.522	1.066	1.776
750	11.766	5.804	4.972
1000	33.165	26.106	12.142
1250	74.581	97.090	26.515

密行列, 各要素が 1 以上 10 以下の整数を乱数で生成する. 単位は sec.

n	$g(x)$ with HTT	$h(x)$ with HTT	ブロッククリロフ with HTT
250	0.213	0.085	0.530
500	2.132	1.147	2.484
750	9.519	8.834	5.283
1000	27.698	39.838	12.779
1250	63.736	106.267	28.341

6.6.1 考察

Hensel 構成による $g(x)$ の生成法の結果のみをみると, 整数行列の最小多項式の候補の計算において, Hyper-Threading Technology は有益であると考えられる. しかし, 少なくとも, 最小多項式の次数が行列サイズと等しい場合には, ブロッククリロフ部分空間法のほうが高速である. ブロッククリロフ部分空間法は, Intel Math Kernel Library を有効に活用する計算法であるため, Hyper-Threading Technology は有効に機能しない. また, $h(x)$ を計算するアルゴリズムは, メモリから CPU にデータが到着しないことによりリソースが有効に活用されない場合であり, このアルゴリズムにおいても Hyper-Threading Technology は有効に機能しない.

7 まとめ

Intel Hyper-Threading Technology は, 有効に機能するアルゴリズムとそうでないアルゴリズムが存在する. しかし, 有効に機能するアルゴリズムの中には, Intel Hyper-Threading Technology によって, super-linear を達成できる場合があり, 大変有益な機能である.

[1] http://japan.cnet.com/news/ent/20091397/

[2] http://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[3] https://www.xlsoft.com/jp/products/intel/perflib/mkl/

[4] http://www.netlib.org/blas/faq.html

[5] http://www.netlib.org/blas/faq.html