数理解析研究所講究録 1395 巻 2004 年 218-223

行列計算と基本線形演算の実装法について

兵頭 礼子
NORIKO HYODO
(株) アルファオメガ*

村尾 裕一
HIROKAZU MURAO
電気通信大学†

齋藤 友克
TOMOKATSU SAITO
(株) アルファオメガ‡

1 はじめに

我々はこれまで，数式処理で用いられる行列関連の演算で用いられる標準的な機能を，効率のよいライブラリとして実装する方法の研究を行ってきている．その研究の一部として，代数的計算のためのBLAS(Basic Linear Alebra Subprorams) に相当する副プログラム集の開発を進めている．本稿では主として，その開発および高速化に必要な実装技術をまとめ検討する．特に，元来除算として計算される mod p 演算の高速化の方法として，表を用いることの有効性を示す．

2 $\mathbb{Z}_p$ の表現

我々の実装では 32 ビットのプロセッサを主なターゲットとし，また，多倍長の p は考えないこととする．数式の計算のために今日最も広く使われているプロセッサは Pentium 系のものであり，それらのプロセッサでは 32bit の整数演算は普通に実装されている一方，拡張精度や浮動小数の演算は補助的であるというものも少くない．また，メモリ所要量を節約することも想定して，p の大きさに応じて複数のビット幅のデータ型（精度）を扱えるようにしたい．

2.1 データ表現

文献 [DGP02] でも述べられているように，$\mathbb{Z}_p$ の要素を表現するデータ型として次の 3 種類の方法が考えられる．

(1) 整数型（char/short/int）—— 中間値に long long も利用
- 複数の精度での計算が可能

(2) 実数型（float/double）：仮数部を整数とみなし精度を考慮して計算
- 最低 32bit → 空間効率が悪い
- 数値計算用のハードウェア（ベクトル演算器等）で有効

(3) Jacobi 対数 [LN97]: 原始的な要素 ω を底とする離散対数，即ち，元 $E = \omega^e$ を e で表現
- 有限体 $GF(p^m)$ の表現用．素体 $\mathbb{Z}_p$ を表すには大袈裟過ぎる
- 元の乗除算は指数の加減算．
- 加減算には，$1 + \omega^x = \omega^{Z(x)}$ の $Z(x)$ の表（大きさ p^m）を利用 $\longrightarrow$ p^m が小さい時のみ（現実には $\approx 2^{16}$ 程度まで）実用的

メモリの利用効率と，今日広く使われているプロセッサの前記の特性の 2 点を考慮し，当面の generic 版では「整数型」を用いることとする．整数は符号無しとして，$\mathbb{Z}_p$ 要素は $0 \sim p-1$ の整数値で表すこととする．ここで，Jacobi 対数による表現では，予め計算しておいた値の表を用いることが実用上必要であり，[DGP02] での測定からもわかるように p が小ければ現実的かつきわめて高速な方法であることに注意．

*noriko@a2z.co.jp
†murao@cs.uec.ac.jp
‡saito@a2z.co.jp

2.2 演算

$\mathbb{Z}_p$ での算術演算を実現する最も直接的な方法は，整数として演算を行った後 mod p の計算（p による剰余の計算）を行う方法である．しかし，ハードウェアによる演算でも，除算は他の算術演算に比べてはるかに高価であるため，高速化し実用性の高めるためには除算の回数はできるだけ減らすことが必要である．加減算では，除算は整数の加減算よりはるかに高価だが，既に多くの実装で行われているように，この剰余の計算を除去することができる．

加算 ：$S = a + b \bmod p$ を求めるには $\cdots$ 整数で $w := a + b$ を計算する（但し，オーバーフローは無視）．

$$S = \begin{cases} w & \text{if } a \le w < p \\ w - p & \text{otherwise.} \end{cases}$$

減算 ：$D = a - b \bmod p$ は $\cdots$ 整数で $w := a - b$ を計算する（但し，アンダーフローは無視）．

$$D = \begin{cases} w + p & \text{if } a < w \\ w & \text{otherwise.} \end{cases}$$

乗算系 ：addmul 型の $(a * b + c \bmod p)$ と内積型の $(\sum_i a_i * b_i \bmod p)$ の形の計算が主で，途中の値を表現するには倍の精度 (long long まで) が必要となる．「mod p」の除算の高速化が鍵である．高速化の具体的な方法を次節で検討する．

（注）符号付き整数で表現した場合，はるかに複雑な場合分けが必要となる．

3 高速化の方法

加算では，p が十分にデータ型に対し十分に小さければ，オーバーフローも起きず上の場合分けの条件判定も簡単になる．この p に対する条件判定は，ベクトル/行列の全要素に対し共通なので要素に対する繰り返しの外で行うべきである．

乗算系の演算では，必ず mod p の簡単化 (reduction) が必要となり，その除算を如何に省いたり高速化するかが鍵である．ここでは，次の方法を提案・検討する．

- ［古典的定石］ 除算（商の計算）の乗算への変換
- 内積型の計算における mod p reduction の遅延化 (delayed reduction)
- z のあらゆるビットパターンに対し予め $z \bmod p$ を計算し，表に蓄えておく (table-driven reduction)

さらに，高速乗算アルゴリズムの効用や，将来的には整数演算自体へのベクトル演算器などのハードウェアの活用を検討する．

3.1 除算の乗算への変換

除算の商は，精度に注意すれば，予め計算しておいた除数の逆数と非除数の乗算から求めることも可能であり，高速化の手法として有用である [Mur91]．具体的には，逆数 $r = 1/p$ を十分な精度をもった実数として計算しておき，$A \bmod p$ の計算では商 $\lfloor A/p \rfloor$（の近似値）を $\lfloor rA \rfloor$ より得る．

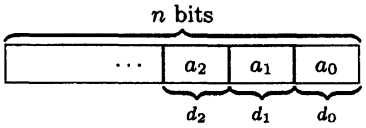
```
typedef struct ModuloInfo {
    unsigned int p;  /* modulo */
    double r;        /* 1/p    */
    unsigned long long thre; /* threshold */
} ModInfo;

#define FastApproxMod(A,p,r) \
  (A-((unsigned int)(r*a))*((unsigned long long)p))
```

3.2 $e \bmod p$ の表の利用

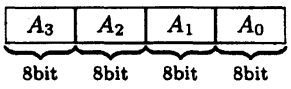
n ビットの整数 A の $A \bmod p$ を計算する（n=8/16/32/64 など）．n が十分に小ければ，n ビットのあらゆるビットパターンの整数値に対し $\bmod p$ を計算し表に蓄えておけば，$A \bmod p$ の値は A をインデックスとして表をひけば得られる．一般の n に対してこの方法を実現するには，n ビットを複数のブロックに分割し，その各部分に対応する表をそれぞれ用意する．

- $n = \sum_{i=0} d_i$ と分割する．$e_k = \sum_{i=0}^{k-1} d_i$ とおく：MSB



- $v[j] = j \times 2^{e_k} \bmod p,\ j = 0, ..., 2^k - 1$ を予め計算しておく
- 【表の利用】 A のビット位置 $(e_{k+1} - 1):e_k$ のビット列を取り出し整数とみなした値を a_k とすると $A \bmod p$ の値は $\sum_{k=0} v[a_k] \bmod p$ で得られる（この $\sum$ では加算の回数が既知なので（プログラム化可能）除算も不要）

（例：表の利用法） A が 32bit 整数で $A = A_0 + 2^8(A_1 + 2^8(A_2 + 2^8 A_3))$ の時（図）



表： $v_k[j] = j \times 2^{8k} \bmod p$
$v_0[\,], ..., v_3[\,]$

$$A \bmod p = v_0[A_0] + v_1[A_1] + v_2[A_2] + v_3[A_3]$$

我々の実装では $n \leq 64$ を想定する．では，どのように分割すればよいだろうか．大きな表を用いるほど高速化すると期待できるが，では，どれくらいまで大きくとるのが現実的なのだろうか．次節では，3次の 3 種類の分割法について実験的に比較検討する．

- (T8) 32 ビット = 8+8+8+8 ビット：(4× 256 個の値)
- (T12) 32 ビット = 12+12+8 ビット：(2× 4096 個の値 + 256 個の値)
- (T16) 32 ビット = 16+16 ビット：ひとつの表が巨大（2× 64K 個の値）

64 ビットに対しては，この 32 ビットの各場合の繰り返しとみなすので，表は倍だけ必要となる．

3.3 内積型の計算における $\bmod p$ reduction の遅延化

内積型の計算 $\sum_i a_i * b_i \bmod p$ では，$(a_i b_i \bmod p)$ 毎に $\bmod p$ の計算を行わずに，ある閾値になるまで和 $(\sum a_k * b_k)$ として足しこんでいきその和の $\bmod p$ を求めることにより除算の回数を減らすことができる．閾値は，商を求めるのに必要な精度やオーバーフローを起こす可能性のある和の値などから決める．[1)]

```
{ unsigned long long s = 0;  ModuloInfo *m;
  ...
  for (i=0; i<n; i++)
    if ((s += a[i]*b[i]) >= m->thre)
      s = FastApproxMod(s, m->p, m->r);
}
```

この例は $a[*], b[*]$ が 32bit の場合で，値を蓄積する変数 s は 64bit である．8bit や 16bit の場合は 32bit の変数を用いれば十分であろう．

4 計算機実験

次の 3 種類の計算について計算機実験を行い，上記の高速化の方法の比較検討を行った．

(ex1) 値の列（ベクトル）に対する $\bmod p$ の計算

(ex2) $\alpha\vec{x} + \vec{y} \bmod p$ の計算

(ex3) 行列乗算：一般的な内積計算による算法と Strassen-Winograd のアルゴリズム

1) その後筆者うちの村尾は，「オーバーフローを検知すればオーバフローの値（2 のべき乗）の $\bmod p$ で置き換えればよい」ことを Dumas から指摘された．

実験に用いた計算機環境は，次の 2 種類のハードウェアで稼働する Red Hat Linux 9 (kernel 2.4.20)，コンパイラ gcc 3.2.2 である．[2)]

(Pen4) Pentium 4@2.80B GHz (L2 cache 512KB), PC2700 1GB

(PenM) Pentium M@1.40GHz (L2 cache 1MB), PC2100 768MB

下の表には，(ex1) と (ex2) で 1 秒あたり処理可能なベクトル長の 10^6 分の 1 の値をまとめた．

- 各列は，mod p の各計算法に対応する．(%) は C 言語の整数剰余計算，(T8),(T12),(T16) は表を用いる方法，(D) は § 3.1 の方法（除算 → 乗算）を示す．
- 各行は，非除数と除数 (p) のビット数の色々な組合せに対応し，(非除数のビット数/除数のビット数) を示す．

	(Pen4)					(PenM)				
#bits	(%)	(T8)	(T12)	(T16)	(D)	(%)	(T8)	(T12)	(T16)	(D)
(ex1)										
(16/8)	89.3	122.0	←	89.3		15.9	33.8	←	75.8	
(32/8)	89.3	80.7	98.0	84.8	22.7	14.5	21.7	27.3	25.9	8.2
(16/16)	100?	294.1		277.8		28.2	75.8		61.7	
(32/16)	87.7	84.7	74.6	70.4	22.1	14.5	22.4	24.4	23.0	8.0
(32/32)	83.3	153.8	142.9	142.9	18.2	14.5	55.6	64.5	57.1	6.8
(64/32)	14.8	33.9	33.9	19.8	18.3	4.8	10.1	10.1	13.7	6.8
(ex2)										
(16/8)	66.7	69.0		250.0	22.5	15.4	25.3		74.1	7.8
(32/16)	64.5	60.6	58.8	64.5	22.7	15.5	21.7	21.3	50.0	7.5
(64/32)	43.5	62.5	45.5	27.8	36.4	12.5	24.1	20.0	20.4	13.2

次に，行列の積の計算に要した CPU 時間（単位は秒）を下表にまとめる．

p のビット数	(/8)				(/16)				(/32)		
matrix size	128	256	512	1024	128	256	512	1024	128	256	512
(Pen4)											
(%)	.05	.35	2.94	76.41	.06	.54	7.29	89.86	.10	.93	23.77
(T8)	.01	.05	.46	59.29	.01	.07	4.44	68.65	.02	.20	9.56
(D)	.01	.12	1.99	63.31	.00	.08	4.47	68.80	.03	.23	11.41
Strassen-Winograd algorithm											
(T8)	.07	.51	3.59	27.44	.07	.48	3.11	22.07	.09	.65	4.39
(D)	.08	.51	3.60	25.27	.08	.53	3.80	26.48	.09	.61	4.25
(PenM)											
(%)	.15	1.18	9.41	76.49	.16	1.29	10.39	84.30	.33	2.66	29.89
(T8)	.02	.20	1.48	27.69	.03	.22	1.81	49.07	.05	.42	6.24
(D)	.04	.32	2.47	36.75	.03	.26	2.12	48.09	.08	.60	8.59
Strassen-Winograd algorithm											
(T8)	.24	1.68	11.87	83.38	.25	1.72	12.12	85.08	.29	2.02	14.16
(D)	.25	1.78	12.40	87.20	.26	1.86	13.02	91.21	.27	1.96	13.45

計算機実験のまとめ

- 処理速度は p の大きさに強く依存する．
- 16bit の表は極端に速い場合もあるが，あまりメリットはない（低速となる場合もあるし，大きすぎる）
- 12bit の表の利用も 8bit の表の利用に比べ殆どメリットはない（余計な演算が必要なため）
- 8bit の表の利用は殆ど全ての場合に % よりも高速
- Pen4 ならば %でも殆どの場合悪くない．それでも，short mod char(2 倍)，short mod short(2~3 倍)，long long mod long(2~2.5 倍) などの高速化

以上より，表 (T8) を用いた mod p の計算が実用上最も優れる．

[2)] 本稿では，口頭発表時のものではなく，最新のデータを載せる．

5 ModBLAS または MBLAS の機能

素体 $\mathbb{Z}_p$ 上の基本線形演算副プログラムとして必要な機能をまとめ，その機能概を説明する．これらは，計算機代数のアルゴリズムを実装するための副プログラムのカーネルを構成する．基本演算及び応用範囲としては

- 行列基本変形．線形方程式の解法（掃き出し法，Wiedemann 算法）および逆行列．行列の標準形のためのモジュラー算法
- 多項式因数分解のための Berlekamp アルゴリズム
- 最小多項式を求める Berlekamp/Massey アルゴリズム
- モジュラー F4 アルゴリズム（代数方程式系の解法）
- $\mathbb{Z}_p$ 係数の一変数多項式の算術演算

などを想定している．

サブルーチン群の構成：BLAS にならって，適切な命名規則を導入し機能を明確にする．

- サブルーチン名の先頭に型 (char,short,int) 固有の名前（以下，T と表す）．型（ビット数）を識別する T は，`MB8`（char, 8bit），`MB16`（short, 16bit），`MB32`（long, 32bit）とする．
- T の直後の 1 文字「_」の有無により，計算結果を新たな領域に格納する（_つき）か，与えられた領域を書き換える（なし）かを区別する．

また，ベクトルや行列の引数については次のとおり扱うこととする

- ベクトルや行列の引数には，1 次元配列と stride (配列要素を幾つおきに取り出すかを示すステップと個数) を指定する．
- ベクトル/行列の一部を定型的に取り出して新たなベクトル/行列とみなしての処理が自然に実現されるであろう．
- 将来は，分散と統合を自動で行うことを計画中である（BLAS に対する BLACS）．

機能一覧

以下の記述では，M や M_i は行列を，$\vec{x},\vec{y},\vec{z}$ は列ベクトルを，α は $\mathbb{Z}_p$ の要素を表すものとする．

【Level 1】 ··· ベクトル操作，一重ループ

	書き換え型		生成型	
(a) データの移動	T`swapv()`	$\vec{z} \leftrightarrow \vec{x}$,	T`_copyv()`	$\vec{z} \leftarrow \vec{x}$
(b) 符号反転	T`negv()`	$\vec{z} \leftarrow -\vec{z}$,	T`_negv()`	$\vec{z} \leftarrow -\vec{x}$
(c) 加算	T`addv()`	$\vec{z} \leftarrow \vec{z}+\vec{x}$,	T`_addv()`	$\vec{z} \leftarrow \vec{x}+\vec{y}$
(d) 減算	T`subv()`	$\vec{z} \leftarrow \vec{z}-\vec{x}$,	T`_subv()`	$\vec{z} \leftarrow \vec{x}-\vec{y}$
(e) スカラー倍	T`scalv()`	$\vec{z} \leftarrow \alpha\vec{z}$,	T`_scalv()`	$\vec{z} \leftarrow \alpha\vec{x}$
& 及び加算	T`addscalv()`	$\vec{z} \leftarrow \alpha\vec{x}+\vec{z}$,	T`_addscalv()`	$\vec{z} \leftarrow \alpha\vec{x}+\vec{y}$
(f) 内積	T`dotprod()`	$\vec{x}^T\vec{y}$		

(g) 標準形計算用：2 つのベクトル操作の同時実行

- データの入替え：$\mathrm{row}(M,k) \leftrightarrow \mathrm{row}(M,j)$ と $\mathrm{col}(M,k) \leftrightarrow \mathrm{col}(M,j)$
- スカラー倍：$\mathrm{row}(M,j) \leftarrow \alpha\,\mathrm{row}(M,j)$ と $\mathrm{col}(M,j) \leftarrow \alpha^{-1}\,\mathrm{col}(M,j)$
- $\mathrm{row}(M,j) \leftarrow \mathrm{row}(M,j) + \alpha\,\mathrm{row}(M,k)$ と $\mathrm{col}(M,k) \leftarrow \mathrm{col}(M,k) - \alpha\,\mathrm{col}(M,j)$

ここで，$\mathrm{row}(M,j)$ および $\mathrm{col}(M,j)$ はそれぞれ行列 M の j 番目の行および列を表すものとする．

【Level 2& 3】 ··· 行列演算，多重ループ

- level 1 の (a)~(e) の機能の行列版（サブルーチン名は "—v" から "—m" に変更）
- ベクトル外積 T`_mulvv()` $M \leftarrow \vec{x}\vec{y}^T$
  & 及び加算 T`_addmulvv_m()` $M_1 \leftarrow \vec{x}\vec{y}^T + M_2$
- 行列とベクトルの積 T`_mulmv()` $\vec{z} \leftarrow M\vec{x}$
  & 及び加算 T`_addmulmv_v()` $\vec{z} \leftarrow M\vec{x}+\vec{y}$
  T`_addmulmv_mulsv()` $\vec{x} \leftarrow M\vec{y}+\alpha\vec{z}$
- 行列乗算 $M_1 \leftarrow M_2M_3$ T`_mulmm_normal()` and T`_mulmm_winograd()`

(非決定的) Wiedemann アルゴリズム用

- T`_addmulmv_v()` $\vec{z} \leftarrow M\vec{x}+\vec{y}$
- T`_addmulvv_m()` $M_1 \leftarrow \vec{x}\vec{y}^T + M_2$
- T`_addmulmv_mulsv()` $\vec{x} \leftarrow M\vec{y}+\alpha\vec{z}$

6 その他

実現法：ファイルの管理 ：データ型毎に同様の計算を行うサブルーチンを作成する必要がある．また，要素毎の検査や動作の変更はできるだけ省く（ループの外へ）等，最適化を意識したプログラミングを行うと記述は複雑になる．そのような状況では，ソースを一元的に管理することが望まれる．C 言語の typedef と CPP のマクロ機能により型をパラメータ化したファイル（テンプレート）を作成し，それを #include することにより C++の template 相当の機能を実現している．

（漸近的）高速アルゴリズムの組み合わせ ：ここまでの内容とやや異るが，多項式を要素とする行列どうしの積の計算において，多項式の乗算及び行列の乗算の高速アルゴリズムを適宜組み合わせることにより，より高速なアルゴリズムを構成できないかを検討した．即ち，行列を $\left(A\right) = \left(A^{(0)}\right) + x^n \left(A^{(1)}\right)$ 及び $\left(B\right) = \left(B^{(0)}\right) + x^n \left(B^{(1)}\right)$ とみなし，これらの積を Karatsuba 流に 3 つの行列積 $\left(A^{(0)}\right)\left(B^{(0)}\right)$, $\left(A^{(1)}\right)\left(B^{(1)}\right)$, $\left(A^{(0)} - A^{(1)}\right)\left(B^{(0)} - B^{(1)}\right)$ から構築することも可能である．これを効率的に実現するには適切なデータ構造による行列の表現が必要になる．これらを詳細に検討すると，要素毎に多項式乗算の高速アルゴリズムを使う場合と同等であり，この変形により Winograd/Strassen に適した形になる場合にだけ高速化が見込まれることが判明した．

7 まとめと今後の課題

本稿では主として，mod p 演算の高速化の方法を検討し，実証実験より表を用いた方法が有効であることを示した．今後，この方法を十分に生かして，本稿でもまとめた MBLAS の機能を実現していくことを計画している．実装にあたっては特に，高速化のための高級言語レベルでのチューニング（p の大きさを意識したプログラミングなど）を行い，ループを意識しかつわかり易い記述を行うためのマクロ化をすすめている．

今後は，さらに，SSE2 などのベクトル演算命令の利用可能性や，応用アルゴリズムでの実用実験もすすめて行くことを予定している．

参 考 文 献

[DGP02] J. G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *Proc. ISSAC '02*, pages 63–74. ACM Press, 2002.

[LN97] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge U.P., 1997.

[Mur91] H. Murao. Vectorization of symbolic determinant calculation. *SUPERCOMPUTER*, VIII(3):36–48, May 1991.