数理解析研究所講究録 1456 巻 2005 年 144-150

Risa/Asirの行列演算の実装II

兵頭礼子 HYODO NORIKO* AlphaOmega Inc.
村尾裕一 MURAO HIROKAZU† 電気通信大学
齋藤友克 SAITO TOMOKATSU‡ AlphaOmega Inc.

概　要

実用域での行列の乗算演算を高速化することを目標とする．本研究では，疎行列を扱うのに適したデータ構造を検討し実装実験をした．演算時間の比較実験から，数式処理に適したデータ型の考察を行なった．

1 始めに

数式処理システムにおいて『線形計算』の適用範囲を拡大する事が本研究の目的である [3, 4, 8]．そのためには，計算時間の短縮と利用する計算機資源の低減が必要である．従来様々な実装実験をおこなった [5, 6, 7]．特に今回の報告は，メモリー使用量の低減化を重点的に見直した．

一般にメモリー使用量を低減化した場合，実行速度が低下することが普通である．しかし今回の実装実験では，データ構造を見直しメモリー使用量を低減化したにもかかわらず，実行速度の低下は見られていない．このことは，従来から我々が主張している数式処理における行列計算の計算時間の短縮は，個々の演算よりも領域の使用量に強く依存する事を裏付けている [8]．

2 従来の実装

Risa/Asir の従来の実装は，人間の感覚には大変分かりやすいものである．しかし，ゼロ成分であってもデータとして保持するため，行列成分の疎密具合にかかわらず，行列サイズ分の領域を必要としていた．また，行列の各成分の参照でしかゼロ行列等の行列の性質を判定する方法がなく，微小ではあるがサーチ時間，また 0 に対する演算時間も発生し，大規模疎行列を扱う際に無駄な処理時間を要している．

行列の実装は次のようになっている．

```
typedef struct oMAT {
        short id;        /*  行列であることを示す識別番号        */
        short pad;
        int row,col;     /*  行列のサイズ                        */
        pointer **body;  /*  ポインタを格納する配列へのポインタ  */
} *MAT;
```

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

実際の行列要素は，2次元配列で表されるポインタの配列から指し示されている．つまり，要素が0であったとしても0へのポインタが確保される．

このような実装は，大規模疎行列を扱う場合には記憶装置の効率的な利用からみて不向きな実装となっていた．

3 疎行列の取り扱いのための実装

3.1 実装の目的

今回は，大規模疎行列の取り扱いに適した行列型を検討し，実装することとした．新たな実装を行なうにあたって，設計の目標としたことは以下の二点である．

- 使用するメモリ領域を節約する
- 各成分へのアクセスは必要最低限にする

この二点を考え，行列の表現方法を多項式の表現方法と同様に，非ゼロ成分のリストとなるように構造体を構成した．しかし，単純なリスト構造とした場合行列要素の行番号と列番号を常に保持する必要がある．このメモリ領域の使用は，疎行列といえども節約は限られてしまう．そのため，リスト構造にチャンクと呼ばれるリストの集団を設定し，メモリのさらなる節約を図った．

3.2 新しい行列型の実装

以下にRisa/Asirへの実際の実装コードと，構造の模式図を示す．

```
#define IndMatCH 64          /*  チャンクサイズ                    */
typedef struct oIndMat {     /*  実際の行列構造体                  */
        short id;
        int row, col;        /*  行列のサイズ                      */
        int clen;            /*  チャンクがいくつ存在するか        */
        pointer *root;       /*  先頭チャンクデータへのポインタ    */
        pointer *toor;       /*  末尾チャンクデータへのポインタ    */
} *IndMat;
typedef struct oIndMatC {    /*  チャンクデータ格納のための構造体  */
        pointer *fore;       /*  直前のチャンクへのポインタ        */
        pointer *next;       /*  直後のチャンクへのポインタ        */
        IndEnt ient[IndMatCH];
} *IndMatC;
typedef struct oIndEnt {     /*  実要素データ格納のための構造体    */
        int cr;              /*  行列要素の指標                    */
        int row, col;        /*  行列要素の行番号と列番号          */
        pointer *body;       /*  実際の要素データ                  */
} IndEnt;
```

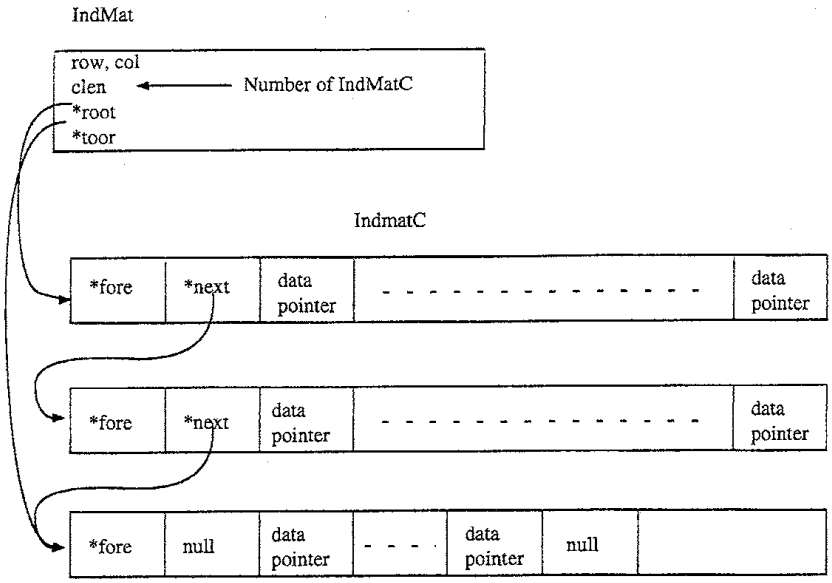

図 1: Index 型行列の概要

また、各要素の行と列から次のような指標を導入した．行列を $n \times m$ とした場合，a_{ij} への指標 (index) を $(i-1) \times m + j$ とし，IndEnt 構造体の cr に格納する．この指標 (index) を各成分データの参照の際に使用する．この指標は，要素の行番号と列番号を与えれば計算できるため本来不要な指標である．しかし，ある要素を特定するために判定の回数を 1 回節約する効果がある．一方，この指標より要素の行番号と列番号は求められる．その意味では，この指標と行番号列番号はどちらか一方が存在すればよい．しかし，計算効率を上げるためあえて導入した．

また，IndMat 構造体の clen には，IndMatC 構造体の数を格納し，clen の値を参照することで，行列がゼロ行列であるかを全要素データをチェックする事なく判定することが可能となる．

さらに各チャンクを効率的に探査するため，個々のチャンクからなるリストは，双方向リストとした．この双方向性を実現するポインタが IndMatC の fore と next である．この双方向リストの root は IndMat の root と toor である．

チャンクが双方向リストの構造を持つため，行列要素全体は，自動的に双方向リストとなる．個々の要素に双方向リストのポインタを持たせる必要がなく，メモリー容量の減少がはかれ双方向リストになり実行効率も向上する．

以降，この新しい行列型を index 型と呼ぶ．

3.3 従来の行列と index 型行列の疎密具合による演算時間の比較

次の $n \times n$ 行列 A,B（各々の成分は a_{ij}, b_{ij} とする）を考える．各行列の要素は

$$\begin{cases} a_{ij} &= (i+1)(j+1)(x^5+x^4+x^3+x^2+x), \\ b_{ij} &= (i+1)(j+1)(y^5+y^4+y^3+y^2+y) \end{cases} \qquad 1 \le i,j \le n$$

とする．この行列 A,B を使用し，行列の疎密具合を変化させ，従来のデータ型行列と index 型行列で $A \times B$ を実行し，$A \times B$ を実行し，演算時間を計測した．実験は，AMD Athlon^{TM} XP 3200+, Memory:512MB 上，OS は FreeBSD 5.2 上で行なった．表 1 に，行列中のゼロ成分が 1/2 以上の場合の演算時間を示す．

表 1: 従来の行列型と index 型行列での演算速度 (単位:sec)

		従来の実装			index 型の実装		
ゼロ率	Size	cpu	gc	計	cpu	gc	計
0.5	8×8	0.001073		0.001073	0.001045		0.001044
(1/2)	16×16	0.01257	0.006776	0.01937	0.01219	0.006805	0.01906
	32×32	0.1176	0.07408	0.1921	0.1134	0.06389	0.1776
	64×64	0.9671	0.6314	1.607	0.9192	0.4999	1.427
	128×128	8.052	5.2275	13.34	7.342	2.078	9.462
0.75	8×8	0.0003922		0.0003898	0.0003724		0.0003725
(3/4)	16×16	0.002055		0.002054	0.001981		0.002017
	32×32	0.02145	0.01136	0.03285	0.02352	0.01200	0.03578
	64×64	0.2342	0.1139	0.3459	0.2099	0.08117	0.2917
	128×128	1.957	1.101	3.069	1.766	0.6476	2.423
0.875	8×8	0.0001008		0.0001002	0.0000686		0.00006757
	16×16	0.0005610		0.0005600	0.0004030		0.0004029
(7/8)	32×32	0.005104		0.005113	0.003765		0.003801
	64×64	0.03684	0.01401	0.05092	0.03682	0.01346	0.05039
	128×128	0.4353	0.1282	0.6094	0.3563	0.1754	0.5377
0.9375	8×8	0.00004640		0.00004496	0.0000250		0.00002418
	16×16	0.000272		0.0002708	0.000103		0.000102
(15/16)	32×32	0.002370		0.002377	0.0009950		0.0009961
	64×64	0.01777		0.01783	0.007307		0.007318
	128×128	0.1546	0.02321	0.1781	0.0689	0.02234	0.09137
0.96875	8×8	0.00004520		0.00004215	0.0000168		0.00001426
	16×16	0.000215		0.0002131	0.000039		0.00003481
(31/32)	32×32	0.001579		0.001575	0.0002730		0.0002699
	64×64	0.01226		0.01227	0.001862		0.001861
	128×128	0.102		0.1021	0.01575		0.01579

行列成分中に非ゼロ成分が多い場合は，従来の行列型と index 型行列の演算速度には差はあまり発生しない．しかし，ゼロ成分が増加するにつれ index 型行列の演算速度が高速になり，128×128 行列では，非ゼロ成分が 1/32 の場合で，演算時間は index 型の方がおよそ 1/10 の速度で演算を実行することが出来る．

3.4 index 型行列への Strassen-Winograd アルゴリズムの実装

3.3 の実験結果より，index 型が疎行列を扱う際には有効であることが分かった．そこで，従来の行列型で実装し，演算の高速化に有効であった Strassen-Winograd アルゴリズム [1] を index 型行列でも実装した．

3.4.1 内積アルゴリズムと Strassen-Winograd アルゴリズムの演算時間の比較

3.3 で使用した行列 A, B を使用して，行列の疎密具合を変化させ，index 型行列で内積アルゴリズムと Strassen-Winograd アルゴリズムでの演算速度の比較を行なう．表 2 に行列中のゼロ成分が 1/2 以上の場合

の演算時間を示す.

表 2: index 型行列での演算速度 (単位:sec)

		内積アルゴリズム			Strassen-Winograd		
ゼロ率	Size	cpu	gc	計	cpu	gc	計
0.5	8×8	0.001045		0.0010442	0.003812	0.002853	0.006760
(1/2)	16×16	0.01219	0.006805	0.01906	0.03550	0.02113	0.05678
	32×32	0.1134	0.06389	0.1776	0.3226	0.2092	0.5328
	64×64	0.9192	0.4999	1.427	2.457	1.954	4.430
	128×128	7.342	2.078	9.462	18.31	17.56	36.12
0.75	8×8	0.0003724		0.0003725	0.001784	0.002440	0.004227
(3/4)	16×16	0.001981		0.002017	0.01663	0.01128	0.02795
	32×32	0.02352	0.01200	0.03578	0.206	0.1209	0.3276
	64×64	0.2099	0.08117	0.2917	1.888	1.363	3.269
	128×128	1.766	0.6476	2.423	15.47	12.38	28.02
0.875	8×8	0.00006860		0.00006757	0.0002712		0.0002704
	16×16	0.0004030		0.0004030	0.003140	0.003339	0.01077
(7/8)	32×32	0.003765		0.003801	0.09762	0.05160	0.1495
	64×64	0.03684	0.01401	0.05092	1.269	0.8533	2.131
	128×128	0.3563	0.1754	0.5377	12.01	9.545	21.71
0.9375	8×8	0.00002500		0.00002418	0.0001206		0.0001192
	16×16	0.0001030		0.0001020	0.003140		0.003143
(15/16)	32×32	0.000995		0.0009961	0.04279	0.0145	0.05739
	64×64	0.007307		0.007318	0.6120	0.3456	0.9593
	128×128	0.0689	0.02334	0.09137	7.884	6.28	14.28
0.96875	8×8	0.00001680		0.00001426	0.00003240		0.00002961
	16×16	0.0000390		0.00003481	0.000966		0.0009651
(31/32)	32×32	0.000273		0.0002699	0.01713	0.006751	0.02394
	64×64	0.001862		0.001861	0.2400	0.0795	0.3203
	128×128	0.01575		0.01579	4.148	3.26	7.49

行列中にゼロ成分が増加すると，index 型行列の場合では内積アルゴリズムのほうが Strassen-Winograd アルゴリズムを適用するよりも高速に乗算が実行され，Strassen-Winograd アルゴリズムによる高速化ははかられない.

そこで，同じアルゴリズムで，データ型の違いによる演算速度の比較を行ない，index 型行列が Straseen-Winograd アルゴリズムに適していないのか，また Strassen-Winograd が疎行列を扱うのに適していないのかの検証を行う．実験に使用した行列 A, B は 3.3 と同様で，64×64 行列において行列中のゼロ成分の割合を 0 から 2047/2048 まで変化させ，$A \times B$ の演算を行ない，演算速度を計測する．表 3 に計測結果を示す.

行列が密である場合，アルゴリズムが同じ場合では，従来の行列型と index 型行列の演算速度に大きな差は見られない．しかし，行列中のゼロ率があがるにつれ，どちらのアルゴリズムでも index 型行列の方が高速に演算を行なうことが可能となる．これらの結果から，数式処理の行列表現で index 型を使用することに不都合は無いと考える.

表 3: 64 × 64 行列における行列型による演算速度の比較 (単位:sec)

	内積アルゴリズム		Strassen-Winograd	
ゼロ率	従来の実装	index 型	従来の実装	index 型
0	7.264	7.324	2.244	2.113
1/2048	7.367	7.256	2.288	2.199
1/1024	7.268	7.256	2.266	2.280
1/512	7.289	7.061	2.298	2.337
1/256	7.235	7.038	2.490	2.480
1/128	7.237	6.982	2.654	2.665
1/64	7.113	6.974	2.981	3.099
1/32	6.845	6.652	3.450	3.484
1/16	6.424	6.242	3.929	3.960
1/8	5.589	5.392	4.453	4.453
1/4	4.082	3.859	4.813	4.766
1/2	1.607	1.427	4.360	4.430
3/4	0.3459	0.2917	3.274	3.269
7/8	0.05781	0.05092	2.024	2.131
15/16	0.01783	0.007318	0.8046	0.9593
31/32	0.01227	0.001861	0.2494	0.3203
63/64	0.01102	0.0005422	0.1004	0.1289
127/128	0.01064	0.0001462	0.03898	0.05984
255/256	0.01057	0.00009203	0.03317	0.03170
511/512	0.01045	0.00002599	0.03125	0.02442
1023/1024	0.01040	0.00001597	0.02871	0.01264
2047/2048	0.01036	0.00001597	0.02700	0.0006859

4 まとめ

本研究では，ターゲットを疎行列にしぼり，行列の乗算の高速化と行列型の実装方法について検討を Risa/Asir 上で行なった．今回提案し，実装した index 型行列は，疎行列を扱うには適しており，演算速度は格段に高速化される．また，行列が密の場合でも，従来より実装されていた行列型と遜色ない演算速度で計算を行なうことが出来る．

今回は行列に特化した検討，実験であるが，数式処理全体のデータ型の検討，検証が今後の課題である．

参 考 文 献

[1] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[2] S. Winograd. On multiplication of 2 × 2 matrices. *Linear Algebra and its Applications*, 4:381–388, 1971.

[3] 兵頭 礼子, 村尾 裕一, 齋藤 友克. Risa/asir の matrix 演算の検討. 数式処理, 9:10–11, 2002. 第 11 回大会報告.

[4] 兵頭 礼子, 村尾 裕一, 齋藤 友克. Risa/asir の matrix 演算の新しい実装について. 講究録 *1295* 「Computer Algebra – Algorithms, Implementations and Applications, 2001」, pages 213–219. 京都大学数理解析研究所, 2002.

[5] 兵頭 礼子, 村尾 裕一, 齋藤 友克. 数式処理のための行列演算の効率的な実装法について. 数式処理, 10:18–19, 2003. 第 12 回大会報告.

[6] 兵頭 礼子, 村尾 裕一, 齋藤 友克. 多項式表現と行列演算の改良. 講究録 *1335* 「Computer Algebra – Algorithms, Implementations and Applications, 2002」, pages 28–32. 京都大学数理解析研究所, 2003.

[7] 兵頭 礼子, 村尾 裕一, 齋藤 友克. 行列計算と基本線形演算の実装法について. 講究録 *1395* 「Computer Algebra – Algorithms, Implementations and Applications, 2003」, pages 218–223. 京都大学数理解析研究所, 2004.

[8] Hyodo, N., Murao, H., and Saito, T.. Matrix operation made fast — practical view of fast matrix operation for computer algebra system. 『数式処理』, Vol. 11, No. 3,4, pp. 3–20, 2005.