Vol.2010-ARC-192 No.18
Vol.2010-HPC-128 No.18
2010/12/17

疎行列固有値解法における4倍精度演算とその性能評価

西　田　　晃[†1]

計算科学において，計算精度の向上は本質的に重要な目標である．演算性能はメモリ性能と比較して速い速度で向上しており，このギャップが問題となることがある．本研究では並列反復解法ライブラリ Lis を対象に，double-double 精度による 4 倍精度演算を実装し，この問題を計算精度を向上させることによって緩和できるとの知見を得たが，ここではこれを固有値解法に適用し，4 倍精度演算の有効性について検討する．

Quadruple Precision Operation in Eigensolvers for Sparse Matrices and its Performance Evaluation

Akira Nishida [†1]

The improvement of the accuracy is a critically important target of computational science, FLOPS performance is increasing with faster speed than memory performance. This gap sometimes raises serious problem, but our previous study have shown that the problem can be reduced by improving floating point precision. In this paper, we discuss the validity of the quadruple precision arithmetics for eigensolvers, based on the double-double precision floating point operations implemented on the parallel iterative solver library Lis.

1. 背　　景

本研究では, 平成 14-19 年度科学技術振興機構 CREST 事業の一環として, 反復解法ライブラリ Lis [⋆1] を開発, 配布し, 様々な並列計算機上で大規模な線型方程式を解くための環境を提供してきた．また平成 20 年度からは九州大学情報基盤研究開発センターにおいて，大規模並列環境への対応を中心に研究を行うとともに，同年 11 月より疎行列固有値解法に対応した新版を公開している．

Lis は, Krylov 部分空間法を中心とする多様な反復解法を実装したライブラリである．同様な目的のライブラリとして，線型方程式については Argonne 米国立研究所の並列反復解法ライブラリ PETSc [⋆2] や Lawrence Berkeley 米国立研究所による並列反復解法ライブラリ Hypre [⋆3] などを挙げることができる．固有値解法については，Valencia 工科大学による SLEPc [⋆4] (PETSc を用いて開発されている) や, Colorado 大学による BLOPEX [⋆5] (Hypre を用いている) などに疎行列を対象とした固有値解法が実装されている [⋆6]．

Lis ではこれらの機能を単一のライブラリにおいて実現するとともに，多様な前処理アルゴリズムを実装している．また，近年一般的となったマルチコア環境に適したハイブリッド並列処理に対応している点も特徴として挙げることができる．表 1-3 に現時点で対応している固有値解法, 線型方程式解法, 行列格納形式の一覧を示す.

表 1　Lis で利用可能な固有値解法

Power Iteration
Inverse Iteration
Approximate Inverse Iteration
Subspace Iteration
Lanczos Iteration
Conjugate Gradient
Conjugate Residual

線型方程式の求解に用いられる共役勾配法等の反復解法は，理論的にはたかだか n 回の反復で収束することが知られている．しかしながら，実際には丸め誤差の影響により，有限精度計算では一般に収束までにはより多くの反復回数を要し，また収束が停滞する場合もある．このような収束の特性を改善する上で，多倍長演算は有効な手法であると考えられるが，ハードウェアで実装されている倍精度演算等に比べ，計算コストの大きさが問題となる．

この問題を解決するため，Lis では，近年のプロセッサに搭載されている SIMD 命令を

†1 九州大学情報基盤研究開発センター
Research Institute for Information Technology, Kyushu University

⋆1 http://www.ssisc.org/lis/

⋆2 http://www-unix.mcs.anl.gov/petsc/

⋆3 http://computation.llnl.gov/casc/hypre/

⋆4 http://www.grycap.upv.es/slepc/

⋆5 http://www-math.cudenver.edu/~aknyazev/software/BLOPEX/

⋆6 http://www.netlib.org/utk/people/JackDongarra/la-sw.html

Vol.2010-ARC-192 No.18
Vol.2010-HPC-128 No.18
2010/12/17

表 2 Lis で利用可能な線型方程式解法

CG	CR
BiCG	BiCR
CGS	CRS
BiCGSTAB	BiCRSTAB
GPBiCG	GPBiCR
BiCGSafe	BiCRSafe
BiCGSTAB(l)	BiCRSTAB(l)
Jacobi	Gauss-Seidel
SOR	Orthomin(m)
TFQMR	MINRES
GMRES(m)	FGMRES(m)
IDR(s)	

表 3 Lis で利用可能な行列格納形式

Compressed Row Storage	(CRS)
Compressed Column Storage	(CCS)
Modified Compressed Sparse Row	(MSR)
Diagonal	(DIA)
Ellpack-Itpack generalized diagonal	(ELL)
Jagged Diagonal	(JDS)
Block Sparse Row	(BSR)
Block Sparse Column	(BSC)
Variable Block Row	(VBR)
Dense	(DNS)
Coordinate	(COO)

活用することにより，4 倍精度演算を効率よく実行できるよう実装を行っている．反復解法の丸め誤差を減少させるため，線型方程式における係数行列と右辺ベクトル，初期ベクトルに倍精度を使用し，内部処理のみを 4 倍精度化することを考える．Lis では，この目的のために Bailey らによって提案された倍精度浮動小数点数を 2 個使用する “double-double” 精度アルゴリズム[1] を用い，Intel プロセッサに実装されている SIMD 命令の一種である SSE2 により，これを実装している．“double-double” 精度の実装については Hida ら[2] による QD ライブラリなどがあり，これを用いた反復解法ライブラリ GMM++[3] が存在する．本研究は，倍精度演算によって実装される 4 倍精度演算を，SIMD 命令によって高速化した点でこれらの研究と異なっている．以下では，具体的な実装手法について説明する．

2. 4 倍精度演算の実装

Bailey のアルゴリズムでは，double-double 精度浮動小数 a を $a = a.hi + a.lo$, $\frac{1}{2}\mathrm{ulp}(a.hi) \geq |a.lo|$ (上位の $a.hi$，下位の $a.lo$ とも倍精度) として，4 倍精度演算を倍精度の四則演算の組み合わせで実現する．なお $\mathrm{ulp}(x)$ は x の仮数部の誤差範囲を示す “unit in the last place” を意味する．$\frac{1}{2}$ulp は丸め誤差の上限である．

すべての演算は，IEEE 倍精度演算で round-to-even 丸めと仮定する．x と y を倍精度とし，$x+y$ の倍精度加算の結果を $\mathrm{fl}(x+y)$ と表す．$\mathrm{err}(x+y)$ は $x+y = \mathrm{fl}(x+y)+\mathrm{err}(x+y)$ を満たすものとする．この時，以下のようにして 4 倍精度加算 $a = b + c$ が計算できる．ただし $a = (a.hi, a.lo)$, $b = (b.hi, b.lo)$, $c = (c.hi, c.lo)$ とする．

まず b と c の上位 $b.hi$ と $c.hi$ に，丸め誤差のない加算

$$b.hi + c.hi = \mathrm{fl}(b.hi + c.hi) + \mathrm{err}(b.hi + c.hi) \quad (1)$$

を行い，

$$\mathrm{fl}(b.hi + c.hi) + \mathrm{fl}(\mathrm{err}(b.hi + c.hi) + b.lo + c.lo) \quad (2)$$

を $a+b$ の近似値とする．今回の実装では高速性を重視して下位の誤差 $\mathrm{fl}(\mathrm{err}(b.hi+c.hi)+b.lo+c.lo)$ は無視している．なお，下位の足し合わせによって繰り上がりの可能性があるため，この計算は丸め誤差のない加算によって行う．

4 倍精度乗算についても $x \times y = \mathrm{fl}(x \times y)+\mathrm{err}(x \times y)$ とする．ここで $\frac{1}{2}\mathrm{ulp}(\mathrm{fl}(x \times y)) \geq |\mathrm{err}(x \times y)|$ である．4 倍精度乗算は，まず b と c の上位 $b.hi$ と $c.hi$ に丸め誤差のない乗算を行い，

$$b.hi + c.hi = \mathrm{fl}(b.hi \times c.hi) + \mathrm{err}(b.hi \times c.hi) \quad (3)$$

とする．次に，b の上位と c の下位，b の下位と c の上位の乗算結果と $\mathrm{err}(b.hi \times c.hi)$ との間で丸め誤差のない加算を行い，$b \times c$ の近似値を得る．

4 倍精度ベクトルのデータ構造としては，上位と下位をそれぞれ別な配列に格納する方法，交互に格納する方法が考えられるが，前者は上位を格納する配列を用いて倍精度演算を行うことができるため，ここでは前者の方法を採用する．

反復解法の計算の主要部は，疎行列-ベクトル間演算，内積，及びベクトル間演算からなる．これらを 4 倍精度演算に置き換える．ただし倍精度演算と同一のインタフェースを保つため，

- 係数行列，右辺ベクトルは倍精度
- 解ベクトルの入出力は倍精度，内部演算は 4 倍精度
- 反復解法中のベクトルとスカラーは 4 倍精度

とする．このため，疎行列-ベクトル間演算では倍精度-4 倍精度間演算，内積，ベクトル間演算，スカラ間演算では 4 倍精度-4 倍精度間演算を実装する必要がある．

これらの演算では実行時間が問題となるため，SIMD 命令の利用による高速化を検討した．Intel プロセッサの場合を考えると，SIMD 命令として，SSE (Streaming SIMD Extensions)

Vol.2010-ARC-192 No.18
Vol.2010-HPC-128 No.18
2010/12/17

が搭載されている．ここでは，128 ビットデータに対する処理が可能な SSE2 を用いて，倍精度浮動小数に対して同時に 2 つの演算を行うことを考える．計算の依存関係による性能の低下を考慮し，2 段のループアンローリングを併用した．また各演算のループ内で使用するスカラ変数については，必要に応じて 128 ビット XMM レジスタに格納するとともに，16 バイトのアラインメント境界を考慮したデータ処理を行っている．

3. 固有値解法への適用

疎行列固有値解法の適用例として，2 次元 Helmholtz 問題を考える．矩形領域 $[0,l]\times[0,m]$ の膜の振動を表す 2 次元 Laplace 作用素を 5 点中心差分により離散化した場合，対応する行列の固有値は

$$\pi^2\left(\frac{\sigma^2}{l^2}+\frac{\tau^2}{m^2}\right),\ \sigma,\tau\in N \qquad (4)$$

で与えられる．$l=m=20$，すなわち行列サイズ $20^2\times 20^2$，として，Lis に実装した部分空間反復法により絶対値最小のものから順に 6 個の固有対を求めた結果を図 1 に示す．なお内部の解法には逆反復法，前処理なし共役勾配法を使用している．Intel Xeon 5570 サーバ (2.93GHz クアッドコアプロセッサ ×2) の 1 コア上で倍精度，4 倍精度での逐次計算を行った場合の計算時間の内訳のうち，1%を越えるものを図 2-3，同様の計算を行列サイズを $200^2\times 200^2$ として行った場合の内訳を図 4-5 に示す．行列サイズを $200^2\times 200^2$ とした場合，倍精度，4 倍精度での計算時間はそれぞれ 63.32 秒，44.64 秒であった．求める固有値の数を 1 とした場合には，倍精度，4 倍精度での計算時間はそれぞれ 1.75 秒，1.74 秒となった．また，内訳から CRS 形式での疎行列ベクトル積を計算するルーチン lis_matvec_crs() の計算時間の減少分が大きいことが分かるが，このことから，4 倍精度演算は直交化における反復回数の減少に関連があることが見て取れる．

各固有値の計算に要する反復回数を，表 5 に示す．モード 3 については，残差の閾値 10^{-12} に達していないため，反復回数の上限 1000 回 (残差 1.04×10^{-12}) で打ち切っている．よって，この固有値の計算における残差の停滞が，全体の計算時間に影響を与えていることが分かる．

一方，計算する固有値の数を 20 個に増やした場合の結果を表**??**に示す．モードの小さい固有値については倍精度での計算と比較して少ない反復回数で収束しているが，モードが大きくなると 4 倍精度であっても収束が停滞するものが出てきている．なお，この場合の倍精度での計算時間は 298 秒であったのに対し，4 倍精度での計算時間は 396 秒であった．さらに，MPI 版を使用して使用コア数を 8 に増やし，30 個までの固有値計算を行った結果を**??**に示す．倍精度での計算時間は 123 秒，4 倍精度での計算時間は 489 秒であった．モードが大きくなると収束が停滞する傾向はより顕著になっているが，この場合には 4 倍精度演算でも十分な効果が得られていないことが分かる．

以上の結果から，計算する固有値の量が増えるにつれ，要求される精度が高くなり，4 倍精度演算でも十分な精度が得られなくなっていると考えられる．4 倍精度演算が有効なのは，

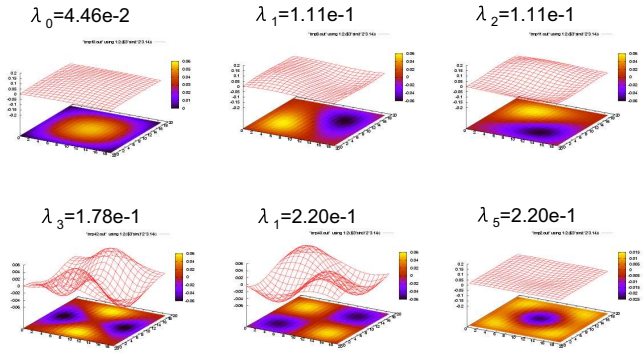

図 1 2 次元 Helmholtz 問題（膜の振動）におけるモード

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
66.67	0.04	0.04	10374	0.00	0.00	lis_matvec_crs
16.67	0.05	0.01	12154	0.00	0.00	lis_vector_copy
16.67	0.06	0.01	1768	0.01	0.01	lis_vector_set_all

図 2 部分空間反復法における各処理の割合（Intel Xeon 5570 サーバ 上，問題サイズ $20^2\times 20^2$，倍精度．Lis のプロファイリングオプションによる．）

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
55.56	0.05	0.05	9398	0.01	0.01	lis_matvec_crs
11.11	0.06	0.01	17252	0.00	0.00	lis_vector_dot
11.11	0.07	0.01	4650	0.00	0.00	lis_vector_duplicateex
11.11	0.08	0.01	2849	0.00	0.00	lis_vector_axpyex_mmm
11.11	0.09	0.01	772	0.01	0.01	lis_cg_check_params

図 3 部分空間反復法における各処理の割合（Intel Xeon 5570 サーバ 上，問題サイズ $20^2\times 20^2$，4 倍精度．）

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
64.62	40.92	40.92	124700	0.00	0.00	lis_matvec_crs
12.00	48.52	7.60	251966	0.00	0.00	lis_vector_axpy
9.16	54.32	5.80	253778	0.00	0.00	lis_vector_dot
5.50	57.80	3.48	124700	0.00	0.00	lis_vector_xpay
4.34	60.55	2.75				__intel_new_memcpy
2.91	62.39	1.84	130137	0.00	0.00	lis_vector_nrm2
1.06	63.06	0.67				__intel_new_memset

図 4 部分空間反復法における各処理の割合（Intel Xeon 5570 サーバ 上，問題サイズ $200^2 \times 200^2$，倍精度.）

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
62.19	27.76	27.76	84732	0.00	0.00	lis_matvec_crs
11.20	32.76	5.00	167724	0.00	0.00	lis_vector_axpy
8.92	36.74	3.98	167724	0.00	0.00	lis_vector_dot
4.86	38.91	2.17	84732	0.00	0.00	lis_vector_xpay
3.67	40.55	1.64				__intel_new_memcpy
3.16	41.96	1.41	87343	0.00	0.00	lis_vector_nrm2
2.51	43.08	1.12	4203	0.00	0.00	lis_vector_dotex_mmm
1.90	43.93	0.85	3333	0.00	0.00	lis_vector_axpyex_mmm

図 5 部分空間反復法における各処理の割合（Intel Xeon 5570 サーバ 上，問題サイズ $200^2 \times 200^2$，4 倍精度.）

少数の固有値を求める場合に限られる可能性が高いが，適用範囲については検討が必要である．

表 4 問題サイズ $200^2 \times 200^2$ の場合に各固有値の計算に要する反復回数.

Mode	Iteration (double precision)	Iteration (quad precision)
0	18	18
1	78	60
2	122	117
3	1000	120
4	102	96
5	492	459

表 5 問題サイズ $200^2 \times 200^2$ の場合に各固有値の計算に要する反復回数．計算する固有値の数を 20 個とした場合.

Mode	Iteration (double precision)	Iteration (quad precision)
0	18	18
1	78	60
2	122	117
3	1000	120
4	102	96
5	492	459
6	252	232
7	114	118
8	667	604
9	1000	234
10	249	253
11	439	401
12	272	312
13	299	345
14	995	1000
15	300	1000
16	263	253
17	1000	1000
18	1000	1000
19	1000	1000

4. ま と め

本稿では，並列反復解法ライブラリ Lis を対象に，double-double 精度による 4 倍精度演算を固有値解法に適用し，4 倍精度演算の有効性について検討した．直交化を行いながら複数の固有値を求める必要のある問題の場合，数値実験で示したように，計算を進める過程で特定の固有値に関して残差の収束が停滞することがあり，そのような場合の対策として，本手法は有効であると考えられる．ただ，計算する固有値が多い場合には，内部での 4 倍精度演算には限界がある可能性があり，適用範囲についてはより詳細に検討する必要がある．

参 考 文 献

1) Bailey, D. H.: QD: C++/Fortran-90 double-double and quad-double package, http://crd.lbl.gov/dhbailey/mpdist/ (2010).

表 6 問題サイズ $200^2 \times 200^2$ の場合に各固有値の計算に要する反復回数．計算する固有値の数を 30 個とし，MPI 版で 8 並列での計算を行った場合．

Mode	Iteration (double precision)	Iteration (quad precision)
0	18	18
1	82	64
2	112	122
3	1000	99
4	103	97
5	449	451
6	242	245
7	120	118
8	598	660
9	241	225
10	243	243
11	409	471
12	316	331
13	330	322
14	789	917
15	285	293
16	257	247
17	1000	1000
18	1000	1000
19	1000	1000
20	1000	1000
21	440	471
22	451	441
23	1000	1000
24	1000	1000
25	449	445
26	1000	1000
27	1000	1000
28	1000	1000
29	917	1000

2) Hida, Y., Li, X.S. and Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic, *Proceedings of 15th Symposium on Computer Arithmetic*, pp. 155–162 (2001).
3) Renard, Y. and Pommier, J.: GMM++ User Guide, http://home.gna.org/getfem/gmm_int (2010).