高速な 4 倍精度演算を用いたクリロフ部分空間法の安定化

Stabilization of Krylov subspace methods using fast quadruple-precision operation

小武守恒[1), 藤井昭宏[2), 長谷川秀彦[3), 西田晃[4)
Hisashi KOTAKEMORI, Akihiro FUJII, Hidehiko HASEGAWA and Akira NISHIDA

[1)JST, 東京大学（〒 113-0033　文京区本郷 7-3-1, kota@is.s.u-tokyo.ac.jp）
[2)工学院大学（〒 163-8677　新宿区西新宿 1-24-2, fujii@cpd.kogakuin.ac.jp）
[3)筑波大学（〒 305-8550　つくば市春日 1-2, hasegawa@slis.tsukuba.ac.jp）
[4)中央大学（〒 163-8677　文京区春日 1-13-27, nishida@kc.chuo-u.ac.jp）

The convergence of Krylov subspace methods is much influenced by the rounding errors. The high precision operation is effective for the improvement of convergence, however the arithmetic operations are costly. Bailey developed the “double-double” precision algorithm, which uses two double precision floating point numbers. We implemented the quadruple precision operations for itaretive solver library Lis, and accelated by using the Intel SSE2 instruction. In this paper, we evaluate the stabilization of the Krylov subspace method using the fast quadruple precision operation. Results show that the quadruple precision is robust for a problem for which the choice of an appropriate iterative method and preconditioner are necessary in the double precision.

Key Words : *Krylov subspace method, fast quadruple operations, preconditioning*

1. はじめに

反復解法として CG, BiCG 法などのクリロフ部分空間法がよく使われている．倍精度演算では丸め誤差の影響のため収束するまでに多くの反復回数が必要となったり，停滞したりする．収束の改善には高精度演算，例えば 4 倍精度演算が有効であるが計算時間が多くかかってしまう [1]．Bailey[2] は倍精度浮動小数点数を 2 個用いた”double-double” 精度のアルゴリズムを開発している．われわれは，このアルゴリズムを採用して反復解法ライブラリ Lis[3] に 4 倍精度演算を実装し，さらに SSE2 を用いて高速化を行い計算時間の問題解決を図っている [4].

本稿では，Lis に実装した 4 倍精度演算の MPI 並列での性能と，いくつかの反復解法と前処理を用いて倍精度演算と 4 倍精度演算の比較を行い 4 倍精度演算の収束の安定化について検証する．以下 2 節で高速な 4 倍精度演算，3 節で前処理の実装，4 節で数値実験，5 節でまとめを述べる．

2. 高速な 4 倍精度演算

(1) 4 倍精度演算

4 倍精度演算を利用するには FORTRAN の`REAL*16` を用いるのが最も簡単ではあるが，倍精度と比較して 10 から 20 倍もの計算時間と 2 倍のメモリが必要となる．一方，Bailey は倍精度浮動小数点数を 2 個用いた”double-double” 精度のアルゴリズムを開発している．このアルゴリズムでは double-double 精度浮動小数 a を $a = a.hi + a.lo$, $|a.hi| > |a.lo|$（$a.hi$ と $a.lo$ は倍精度浮動小数）として倍精度の四則演算の組み合わせで 4 倍精度演算を実現している．このアルゴリズムは Dekker[5] と Knuth[6] を基にしている．double-double 精度は FORTRAN の`REAL*16` よりも高速である [7]. しかし，FORTRAN の`REAL*16` の表現形式 [8] では仮数部が 112 ビットであるのに対して，倍精度浮動小数を 2 個利用しているため仮数部が 104 ビットと 8 ビット少なくなっている．われわれは，このアルゴリズムを採用して反復解法ライブラリ Lis に 4 倍精度演算を実装した．

仮数部を 104 ビットとした高速な高精度演算によって反復解法の収束を改善するのが目的であるため，ユーザが直接 4 倍精度の変数を扱うことはない．そのため，IEEE 準拠の 4 倍精度と表現が多少違っていても影響は少ない．通常，係数行列 A と右辺ベクトル b は倍精度として与えられるため，

- 係数行列 A，右辺ベクトル b は倍精度
- 解ベクトル x の入出力は倍精度，反復解法中では 4 倍精度
- 反復解法中のベクトルとスカラーは 4 倍精度

とすることで，すべてが 4 倍精度である場合よりもメモリを削減することができる．また，ユーザインタフェイスも倍精度版と共通にすることが可能となる．

以下にこのアルゴリズムの概要を述べる．

(2)　4 倍精度加算

全ての演算は IEEE 倍精度で round-to-even 丸めと仮定する．a と b を倍精度浮動小数とする．$a+b$ の倍精度浮動小数加算結果を $\mathrm{fl}(a+b)$ と表す．$\mathrm{err}(a+b)$ は $a+b = \mathrm{fl}(a+b)+\mathrm{err}(a+b)$ を満たすものとする．これより，$a+b$ の丸め誤差のない加算は 2 つの倍精度浮動小数 $\mathrm{fl}(a+b)$ と $\mathrm{err}(a+b)$ で表すことができる．Dekker と Knuth は図 1 に示す方法で $a+b$ の丸め誤差のない加算が行えることを示している．ここで，$s = \mathrm{fl}(a+b)$，$e = \mathrm{err}(a+b)$，$|s| > |e|$ である．

図 1 の (I) と (II) を用いることで 4 倍精度加算 $a = b + c$ を計算することができる．ただし，a=(a.hi,a.lo),b=(b.hi,b.lo),c=(c.hi,c.lo) である．b と c の上位 $b.hi$ と $c.hi$ を丸め誤差のない加算をすると

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

となる．$sh = \mathrm{fl}(b.hi + c.hi)$，$eh = \mathrm{err}(b.hi + c.hi)$ とする．次に，b と c の下位と eh の加算

$$eh = \mathrm{fl}(eh + b.lo + c.lo)$$

を行うと $sh + eh$ は 4 倍精度加算 $b + c$ の近似となる．下位の足し合わせをすると $sh > eh$ とならなくなる場合があるので再度，sh と eh で丸め誤差のない加算をする必要がある．図 2 に 4 倍精度加算 $a = b+c$ の方法を示す．

(3)　4 倍精度乗算

$\mathrm{fl}(a \times b)$ と $\mathrm{err}(a \times b)$ を加算の場合と同様に $a \times b = \mathrm{fl}(a\times b)+\mathrm{err}(a\times b)$ とする．Dekker は図 3 の`TWO_PROD`を用いることで $a\times b$ の丸め誤差のない乗算が行えることを示している．ここで，$p = \mathrm{fl}(a\times b)$, $e = \mathrm{err}(a\times b)$, $|p| > |e|$ である．`SPLIT` は倍精度浮動小数 a を $a = h+l$ に分割する．ただし，h は a の仮数部の最初の 26 ビット分を持ち l は残りの 26 ビット分を持つ．

図 1 と図 3 を用いることで 4 倍精度乗算 $a = b\times c$ を計算することができる．図 4 に 4 倍精度乗算 $a = b \times c$ の方法を示す．

(4)　SSE2 命令を用いた高速化

SSE2 は Intel の Pentium4 に搭載された x87 命令に代わる高速化命令であり，128bit のデータに対して SIMD 処理を行える（64bit 倍精度浮動小数なら同時に 2 つの

(I) $|a| \geq |b|$ が仮定されている場合：

```
FAST_TWO_SUM(a,b,s,e) {
  s = a + b
  e = b - (s - a)
}
```

(II) $|a| \geq |b|$ が仮定されていない場合：

```
TWO_SUM(a,b,s,e) {
  s = a + b
  v = s - a
  e = (a - (s - v)) + (b - v)
}
```

図–1　丸め誤差のない加算.

```
ADD(a,b,c) {
  TWO_SUM(b.hi,c.hi,sh,eh)
  eh = eh + b.lo + c.lo
  FAST_TWO_SUM(sh,eh,a.hi,a.lo)
}
```

図–2　4 倍精度加算.

```
SPLIT(a,h,l) {
  t = 134217729.0 * a
  h = t - (t - a)
  l = a - h
}

TWO_PROD(a,b,p,e) {
  p = a * b
  SPLIT(a,ah,al)
  SPLIT(b,bh,bl)
  e = ((ah*bh-p)+ah*bl+al*bh)+al*bl
}
```

図–3　丸め誤差のない乗算.

```
MUL(a,b,c) {
  TWO_PROD(b.hi,c.hi,p1,p2)
  p2 = p2 + (b.hi * c.lo)
  p2 = p2 + (b.lo * c.hi)
  FAST_TWO_SUM(p1,p2,a.hi,a.lo)
}
```

図–4　4 倍精度乗算.

倍精度演算を行える）．SSE2 の利用には SSE2 の組込み関数を利用する．

クリロフ部分空間法系統の反復解法は行列ベクトル積（`matvec`），ベクトルの内積（`dot`），ベクトルおよびその実数倍の加減（`axpy`）で実現されている．`matvec`, `dot`, `axpy` の主な処理は積和演算（`FMA`）であるので 4 倍精度の`FMA` 関数を用意した．4 倍精度演算関数`ADD`, `MUL`, `FMA` 等に対して SSE2 の組み込み関数を用いた場合，計算の依存関係のため全体の約 50%程度しか SSE2 の packed-double 命令で処理できず，高速化の効果が

不十分である．

実際に`FMA`を使用するのは`dot`，`axpy`，`matvec`であり，そこではループ内に 1 回ずつ`FMA`が使われていることに注目した．2 段のループアンローリングを行うとループ内で`FMA`が 2 回となり，すべて SSE2 の packed-double 命令で同時に処理できる．

3. 前処理の実装

前処理は線型方程式 $Ax = b$ を反復解法でとくとき，行列 M を係数行列 A を近似するような行列として元の方程式と同値な問題

$$M^{-1}Ax = M^{-1}b$$

に変形することである．クリロフ部分空間法においては前処理を用いることで，収束性が著しく向上することが多い．

Lis に実装した 4 倍精度演算の反復解法での前処理の利用について考える．前処理行列 M は A を近似するような行列であることと，係数行列 A と右辺ベクトル b は倍精度で与えられることを考慮すると，前処理行列の生成は倍精度演算でも十分であると考えられる．

次に，反復解法中で前処理を解く処理では，反復解法中のベクトルは 4 倍精度で与えられていて，前処理行列は倍精度であるので 4 倍精度と倍精度の混合演算を行うのが望ましいと思われるが，今回は計算速度を重視してすべて倍精度演算で処理することにする．したがって，4 倍精度のベクトルは上位のみが利用され，処理の終わりに下位はゼロクリアされる．

これらのことより，Lis 4 倍精度での前処理部分は最後のゼロクリアをのぞいて倍精度と同じ処理となる．

4. 数値実験

Lis に実装した 4 倍精度演算の特性を調べるために，倍精度と FORTRAN `REAL*16` との比較を行う．数値実験は PC Cluster（CPU：Xeon 2.8GHz，メモリ：1GB，OS：Linux 2.4.20 32bit，Network：Gigabit Ethernet，MPI ライブラリ：LAM 7.0）を使用し，コンパイラは Intel C/C++ Compiler 7.0 と Intel Fortran Compiler 9.0 を用い，最適化オプションは “-O3 -xW”を使用した．

(1) Lis 4 倍精度反復解法の性能

まず，実行時間の性能を確認するためにポアソン方程式を有限差分で離散化した行列（次数：1,000,000）に対して BiCG 法を 50 回反復させた実行時間を，倍精度と FORTRAN `REAL*16` と比較する．行列の格納形式は CRS[9] を用いた．実験結果は表 1 のようになる．表中のカッコ内の値は倍精度の実行時間を 1 とした場合の比である．FORTRAN `REAL*16` は MPI 並列用に実装していなため 2PE 以上の計測は行っていない．この表より，1PE では Lis4 倍精度は FORTRAN `REAL*16` より 7.1 倍速く，倍精度の 3.2 倍程度の実行時間となっている．また，4 倍精度演算はデータアクセスよりも演算の割合が大きいため PE 数が増加するごとに倍精度との性能差が縮まっている．

表–1 Execution time (in seconds with the ratio to the execution time of double precision in parentheses) of 50 BiCG iterations.

PE	倍精度	FORTRAN `REAL*16`	Lis 4 倍精度
1	7.56	173.03 (22.9)	24.21 (3.2)
2	3.90		12.22 (3.1)
4	2.02		6.23 (3.1)
8	1.11		3.18 (2.9)

次に，Lis4 倍精度と FORTRAN `REAL*16` の収束特性を確認するために MatrixMarket の行列 rdb2048l（次数:2,048，条件数：1.8 × 10^3，分野：Chemical engineering）と olm1000（次数：1,000，条件数：3.0×10^6，分野：Hydrodynamics）を用いた．右辺ベクトル b は $b = (1, \ldots, 1)^T$，初期ベクトル x_0 は $x_0 = (0, \ldots, 0)^T$，収束判定基準は $||r_{k+1}||_2/||r_0||_2 \leq 10^{-12}$ とした．表 2 に収束するまでの反復回数を示す．これより，数値的な収束特性は最大 1 割増程度であることが分かる．これらのことより，高速で実用的な 4 倍精度演算が実現可能となっている．

表–2 Number of iterations for rdb2048l and olm1000.

行列	倍精度	FORTRAN `REAL*16`	Lis 4 倍精度
rdb2048l	195	159	159
olm1000	2471	504	576

(2) 収束の安定化

ホール効果やイオンスリップといった効果を含めた一般化されたオームの法則から生じる電場ポテンシャル ϕ の $\nabla \cdot \sigma(-\nabla\phi + V \times B) = 0$ の関係から生じる対角性の悪いポアソン方程式 [10] を離散化して生じる行列（次数:23,994，非零要素数:214,060）を用いた．初期ベクトル x_0 は $x_0 = (0, \ldots, 0)^T$，収束判定基準は $||r_{k+1}||_2/||r_0||_2 \leq 10^{-12}$ とした．前処理は Jacobi, ILU(0)，SSOR，Crout 版 ILU（ILUC）を用いた．反復解法は BiCG 法と GPBiCG 法を用いた．表 3 に実行結果を示す．“sec.” は実行時間（秒）を，“iter.” は反復回数を，“TRR” は真の相対残差を，“–” は 10,000 回で収束しなかったことを，“break” はブレイクダウンが発生したことを意味する．

表–3 Results of preconditioned BiCG and GPBiCG methods.

前処理	倍精度			Lis 4 倍精度		
	sec.	iter.	TRR	sec.	iter.	TRR
	BiCG 法					
Jacobi	–			26.67	1833	7.68E-15
ILU(0)	–			44.30	2073	9.35E-15
SSOR	–			18.04	842	9.76E-15
ILUC	–			14.91	475	9.50E-15
	GPBiCG 法					
Jacobi	break			34.50	1403	6.89E-15
ILU(0)	2.99	407	1.91E-14	7.47	225	8.80E-15
SSOR	break			20.92	623	8.46E-15
ILUC	6.04	189	1.55E-14	16.45	308	1.04E-14

この表から以下のことが分かる．

- Lis 4 倍精度では，すべての組み合わせで収束している．一方，倍精度では，BiCG 法ではすべての前処理で収束せず，GPBiCG 法でも Jacobi と SSOR 前処理でブレイクダウンが発生している．このことより，倍精度では適切な解法と前処理を選択しなければ収束しな場合が多いことが分かる．逆に，Lis 4 倍精度を用いれば任意の解法と前処理の選択でも安全に収束することが分かる．
- 前処理部分は倍精度と Lis 4 倍精度どちらも同じ処理なので，倍精度と Lis 4 倍精度の反復回数の差は演算精度の影響だと思われる．
- Lis 4 倍精度は倍精度と比較して真の相対残差がより小さくなっている．

また，実行時間については，最速であった ILU(0)-GPBiCG 法で Lis 4 倍精度と倍精度の実行時間の差は 2.50 倍であるが，8PE で ILU(0)-GPBiCG 法を実行すると倍精度では 2.77 秒，Lis 4 倍精度では 3.10 秒となり，Lis 4 倍精度は倍精度の 1.12 倍と差がより縮まっている．このことより，倍精度と Lis 4 倍精度の実行時間の差は PE 数を増やして並列実行することでその差が縮まり，倍精度と同程度の実行時間となると考えられる．

5. まとめ

本稿では，反復解法ライブラリ Lis に実装した高速な 4 倍精度演算を用いた前処理付反復解法の安定化について検証した．

数値実験で，反復解法の反復回数を固定すると Lis 4 倍精度は FORTRAN `REAL*16` よりも 7.1 倍速く，数値的な収束特性は最大 1 割増程度であることを示した．また，Lis 4 倍精度は PE 数が増加するごとに倍精度との性能差が縮まり，8PE で倍精度の 2.9 倍程度であることを示した．さらに，Lis 4 倍精度を用いることで倍精度では適切な解法と前処理を選択しなければ収束しない問題でも，任意の解法と前処理を選択しても精度よく安定して収束していることを示した．

参考文献

1) H. Hasegawa. Utilizing the Quadruple-Precision Floating-Point Arithmetic Operation for the Krylov Subspace Methods. the 8th SIAM Conference on Applied Linear Algebra, 2003.
2) D. H. Bailey. A fortran-90 double-double library. http://www.nersc.gov/˜dhbailey/mpdist/mpdist.html.
3) http://ssi.is.s.u-tokyo.ac.jp/lis/
4) 小武守恒，藤井昭宏，長谷川秀彦，西田晃., SSE2 を用いた反復解法ライブラリ Lis 4 倍精度版の高速化., 情報処理学会研究報告, 2006-HPC-108, pp.7–12, 2006.
5) T. Dekker. A floating-point technique for extending the available precision. Numerische Mathematik, vol.18 pp.224–242, 1971.
6) D. E. Knuth., The Art of Computer Programming: Seminumerical Algorithms, vol.2., Addison-Wesley, 1969.
7) D. H. Bailey., High-precision arithmetic in scientific computation. In *SC06 Technical Paper*, 2006.
8) Intel Fortran Compiler User's Guide Vol I.
9) R. Barrett, et al.. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, 1994.
10) Fujino, T., et al., Influences of Electrical Conductivity of Wall on Magnethydrodynamic Control of Aerodynamic Heating. Journal of Spacecraft and Rockets, Vol. 43, No. 1, pp.63–70, 2006.