SSE2 を用いた反復解法ライブラリ Lis 4 倍精度版の高速化

小武守　恒[†1,†3] 藤井　昭宏[†2]
長谷川　秀彦[†5] 西田　晃[†4,†1]

CG 法などのクリロフ部分空間法の収束性は丸め誤差に大きく影響される．収束の改善を図るには高精度演算が有効であるが演算量が大きくなる．われわれは開発中の反復解法ライブラリ Lis に 4 倍精度演算を実装し Intel SSE2 命令を用いて高速化を行った．Lis の倍精度と同じインタフェースで 4 倍精度演算が利用できるよう行列 A，右辺 b は倍精度のままで反復解法中のベクトルのみを 4 倍精度とした．その結果 FORTRAN の 4 倍精度より約 4.3 倍速く，倍精度の計算時間の約 4.5 倍程度で抑えることができた．

Fast Quad Precision Iterative Solver Library Lis using SSE2

Hisashi Kotakemori , Akihiro Fujii , Hidehiko Hasegawa
and Akira Nishida

The convergence of Krylov subspace methods, include CG method etc., is much influenced by the rounding errors. The high precision operation is effective for the improvement of convergence, however the arithmetic operations are costly. We implemented the quadruple precision operations for itaretive solver library Lis, and we accelated by using the Intel SSE2 instruction. We assumed the vector in the iterative method to be quadruple precision. Because the quadruple precision operations can be used by the same interface as current version of Lis. The caluculation time of our implemented quadruple operation is almost 4.5 times form that of the current Lis double precision operations.

1. はじめに

大規模疎行列に対する線型方程式 $Ax = b$ を解くため，われわれはさまざまな反復解法，前処理，格納形式を組み合わせて使用することができるライブラリ Lis[1) (a Library of Iterative Solvers for linear systems) を開発中である．

反復解法として CG, BiCG 法などのクリロフ部分空間法が提供されている．CG 法は，丸め誤差の影響が無ければ高々n 回（n は係数行列の次元数）の反復で収束する．しかしながら，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[2)．

そこで，収束性改善のため 4 倍精度演算を Lis に実装し，SSE2 を用いて 4 倍精度演算の高速化を行った．以下 2 節で反復解法ライブラリ Lis，3 節で 4 倍精度演算の実装，4 節で数値実験，5 節でまとめを述べる．

2. 反復解法ライブラリ Lis

Lis は大規模疎行列係数の線型方程式 $Ax = b$ に対する反復法ライブラリで，C 言語と Fortran 90 で記述されている．逐次版と並列版がある．共有メモリ並列版では OpenMP を使用している．分散メモリ並列版では MPI 単独，あるいは OpenMP ＋ MPI のハイブリッドを利用できる．

Lis には以下のような特徴がある．

- 反復解法，前処理の選択は，コマンドラインからも選択可能である
- さまざまな疎行列格納形式が利用できる
- 逐次と並列ともに共通のインターフェイスで処理できる逐次環境から並列環境への移行はプログ

†1 科学技術振興機構 戦略的創造研究推進事業
JST CREST
†2 工学院大学
Kogakuin University
†3 東京大学 情報理工学系研究科
Grad. Sch. of Info. Sci. & Tech., the Univ. of Tokyo
†4 中央大学 21 世紀 COE プログラム
21st Century COE Program, the Chuo Univ.
†5 筑波大学 図書館情報メディア研究科
Grad. Sch. of Lib., Info. & Media Stud., U. of Tsukuba

ラムの変更なし（あるいはごくわずかの変更）でよい

2.1 Components of Lis

解法は，一般実行列用で定常反復解法（SOR 等），非定常反復解法（GPBiCG 等）など 10 種類程度を用意している（表 1）．前処理としては，スケーリング，不完全 LU 分解などに加えて，定常反復解法に有効な $I+S$ 型[3]，smoothed aggregation に基づく代数的マルチグリッド SA-AMG[4]，SOR などの反復法を用いる Hybrid 法[5]，A-直交化に基づき逆行列 A^{-1} そのものを近似する近似逆行列 SAINV[6]，従来の ILU よりも安定な分解ができる Crout 版 ILU 前処理[7] などを用意している（表 2）．データの格納形式は CRS など 10 種類程度を用意している（表 3）．これらの解法，前処理に加え格納形式が容易に組み合わせて使用できる．

表 1 反復解法

非定常解法	CG 法
	BiCG 法
	CGS 法
	BiCGSTAB 法
	BiCGSTAB(l) 法
	GPBiCG 法
	TFQMR 法
	Orthomin(m) 法
	GMRES(m) 法
定常解法	Jacobi 法
	Gauss-Seidel 法
	SOR 法

表 2 前処理

Jacobi
SSOR
ILU(k)
I+S
SA-AMG
Hybrid
SAINV
Crout ILU

表 3 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)

3. 4 倍精度演算の実装

3.1 4 倍精度演算

Bailey[9] は倍精度浮動小数点数（倍精度浮動小数と省略）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 $a = a.hi + a.lo$, $|a.hi| > |a.lo|$ として 4 倍精度を実現している．IEEE 準拠の 4 倍精度[8] の表現形式では仮数部が 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 4 倍精度を実装した．以下にこのアルゴリズムの概要を述べる．

全ての演算は IEEE 倍精度で round-to-even 丸めと仮定する．$a+b$ の浮動小数加算結果を $\mathrm{fl}(a+b)$ と表す．$\mathrm{err}(a+b)$ を $a+b = \mathrm{fl}(a+b) + \mathrm{err}(a+b)$ とする．浮動小数乗算 $a \times b$ についても同様とする．

(1) FAST_TWO_SUM は $s = \mathrm{fl}(a+b)$, $e = \mathrm{err}(a+b)$ を計算する．ただし $|a| \geq |b|$ とする．

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

(2) TWO_SUM は $s = \mathrm{fl}(a+b)$, $e = \mathrm{err}(a+b)$ を計算する．(1) と異なり $|a| \geq |b|$ を仮定していない．

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

(3) 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．

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

(4) TWO_PROD は $p = \mathrm{fl}(a \times b)$, $e = \mathrm{err}(a \times b)$ を計算する．

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

(1) から (4) を用いることで以下の 4 倍精度加算と乗算が可能となる．

(5) ADD は 4 倍精度加算 $a = b+c$ を計算する．ただし，a=(a.hi,a.lo),b=(b.hi,b.lo),c=(c.hi,c.lo) である．

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

(6) MUL は 4 倍精度乗算 $a = b \times c$ を計算する．

```
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) }
```

3.2 Lis への適用

前述の 4 倍精度演算を Lis の反復解法に適用する．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．これらを 4 倍精度演算に置き換える．ただし，倍精度演算のユーザープログラムに影響を与えることなく 4 倍精度演算が利用できるよう

- 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は 4 倍精度

とした．これより，行列ベクトル積が倍精度 × 4 倍精度となるため，4 倍精度 × 4 倍精度よりわずかながら演算量を削減できる．また，倍精度と 4 倍精度のどちらで実行するか指定するオプション`-precision {double|quad}`を用意した．

`matvec`, `dot`, `axpy` に対して，4 倍精度の積和演算関数 `FMA` と混合精度（4 倍精度と倍精度）の積和演算関数 `FMAD` を作成した．`FMA` は `dot` と `axpy` に `FMAD` は `matvec` 中の行列とベクトル積に利用される．積和演算関数 `FMA` は以下のようになる．

(7) `FMA` は 4 倍精度積和演算 $a = a + b \times c$ を計算する．

```
FMA(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,p1,p2)
  TWO_SUM(a.hi,p1,sh,eh)
  TWO_SUM(a.lo,p2,sl,el)
  eh = eh + sl
  FAST_TWO_SUM(sh,eh,sh,eh)
  eh = eh + el
  FAST_TWO_SUM(sh,eh,a.hi,a.lo) }
```

`FMAD` は（7）の `p2 = p2 + (b.lo * c.hi)` を削除したものである．ただし，b は倍精度で $b = b.hi$ である．

第 4 節の数値実験の行列 $A1$（次数 $1,000,000$）に対して表 4 の Xeon 上で単純に 4 倍精度演算を実行した場合，倍精度演算の実行時間と比べて FORTRAN では 23.61 倍，今回の Lis の実装で `FMA` を使用した場合では 8.26 倍となる．さらに，`FMAD` を使用した場合でも 8.17 倍となる．そこで，高速化のために SSE2 SIMD 命令の利用を考える．

3.3 SSE2 による実装

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．

われわれは，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関数 `ADD_SSE2`, `MUL_SSE2`, `FMA_SSE2` 等を作成した．しかしながら，この 3 種の関数では計算の依存関係のため全体の約 50%程度しか SSE2 の packed-double 命令で処理できない．

実際に `FMA` を使用するのは `dot`, `axpy`, `matvec` である．そこではループ内に 1 回ずつ `FMA` が使われている．そこで，2 段のループアンローリングを行うと `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．2 個同時に積和演算をする関数 `FMA2_SSE2`, `FMAD2_SSE2` を用意して，それらを使う関数 `axpy_fma2`, `dot_fma2`, `matvec_fma2` を作成した．以下にその関数を示す．ただし，x と y を次数 n のベクトル，行列 A は CRS（Compressed Row Storage）形式[10] であるとする．

(8) `axpy_fma2` は 4 倍精度の $y = y + ax$ を計算する．

```
axpy_fma2(a,x,y) {
  for(i=0;i<n-1;i+=2)
    FMA2_SSE2(a,&x[i],&y[i]);
  if(i!=n) FMA_SSE2(a,x[i],y[i]); }
```

(9) `dot_fma2` は 4 倍精度の $dot = x^T y$ を計算する．

```
dot_fma2(x,y,dot) {
  for(i=0;i<n-1;i+=2)
    FMA2_SSE2(tmp,&x[i],&y[i]);
  ADD_SSE2(dot,tmp[0],tmp[1]);
  if(i!=n) FMA_SSE2(dot,x[i],y[i]); }
```

(10) `matvec_fma2` は混合精度の $y = Ax$ を計算する．

```
matvec_fma2(A,x,y) {
  for(i=0;i<n;i++) {
    t[0] = t[1] = 0;
    for(j=A.ptr[i];j<A.ptr[i+1]-1;j+=2) {
      xx[0] = x[A.index[j]];
      xx[1] = x[A.index[j+1]];
      FMAD2_SSE2(t,xx,&A.value[j]); }
    ADD_SSE2(t,t[0],t[1]);
    for(;j<A.ptr[i+1];j++)
      FMAD_SSE2(t,x[A.index[j]],A.value[j]);
    y[i] = t[0]; } }
```

4. 数 値 実 験

疎行列反復解法において今回実装した 4 倍精演算の性能を評価するために

- Lis の倍精度
- FORTRAN で作成した 4 倍精度（行列，ベクト

表 4 計算環境
Table 4 Evaluation platforms.

CPU	Xeon	Opteron
Clock	2.8GHz	2.0GHz
L1D Cache	8KB	64KB
L2 Cache	512KB	1MB
Memory	1GB	1GB
OS	Linux 2.4.20smp	Linux 2.6.4smp

ルすべて 4 倍精度）
- Lis の 4 倍精度 (FMA を利用)
- Lis の 4 倍精度 (FMA_SSE2 を利用)
- Lis の 4 倍精度 (FMA2_SSE2 を利用)

の性能を前処理なしの BiCG 法を用いて比較した．係数行列 A はポアソン方程式を 5 点中心差分で離散化した行列 $A1$ と Toeplitz 行列

$$A2 = \begin{pmatrix} 2 & 1 & & & & \\ 0 & 2 & 1 & & & \\ \gamma & 0 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \gamma & 0 & 2 & 1 \\ & & & \gamma & 0 & 2 \end{pmatrix} \quad (1)$$

を用いた．右辺ベクトル b はすべてを 1，初期ベクトル x はすべてを 0 とした．収束判定基準は相対残差ノルム 10^{-12} とした．数値実験は表 4 に示す環境で行った．

コンパイラは Intel C++ Compiler Version 9.0 と Intel Fortran Compiler Version 9.0 を用いた．最適化オプションは-O3 を用いた．ただし，FMA を利用する場合は，FMA が記述されている C ファイルに浮動小数の最適化を抑制するオプション-mp を追加している．

係数行列 $A1$ の次数を 100 から $1,000,000$ まで変化させ BiCG 法を 50 回反復させたときの Xeon と Opteron 上での実行時間を表 5 と表 7 に倍精度・FORTRAN・FMA の実行時間に対する相対性能を表 6 と表 8 に示す．実行時間の単位は秒である．

これらの表から Xeon の場合は
- FMA_SSE2 は FMA との比較で平均 1.29 倍の高速化
- FMA2_SSE2 は FMA との比較で平均 2.02 倍の高速化
- FMA2_SSE2 は FORTRAN との比較で平均 6.24 倍の高速化
- FMA2_SSE2 は倍精度の計算時間の平均 5.88 倍程度で抑えられている

ことが分かる．Opteron の場合は
- FMA_SSE2 は FMA との比較で平均 1.46 倍の高速化
- FMA2_SSE2 は FMA との比較で平均 2.07 倍の高速化
- FMA2_SSE2 は FORTRAN との比較で平均 2.42 倍の高速化
- FMA2_SSE2 は倍精度の計算時間の平均 6.67 倍程度で抑えられている

ことが分かる．平均は算術平均である．

図 1 に Xeon 上で係数行列 $A1$ の次数を $1,000$ から $100,000$ まで変化させ BiCG 法を 50 回反復させたときの実行時間のグラフを示す．"（比例）"が付いているものは次数 $1,000$ のときの実行時間から係数行列の次数に比例させた場合のグラフである．表 5，表 7 と図 1 より，倍精度演算の実行時間はベクトルデータが L1D キャッシュから外れた 10,000 と L2 キャッシュから外れた 100,000 で大きな時間増加が見られるが 4 倍精度演算の実行時間は行列 A の次数にほぼ比例して増加していることが分かる．これは，4 倍精度演算はデータのロードとストアよりも演算の割合が大きいため，データのロードがうまく隠蔽されるためだと思われる．

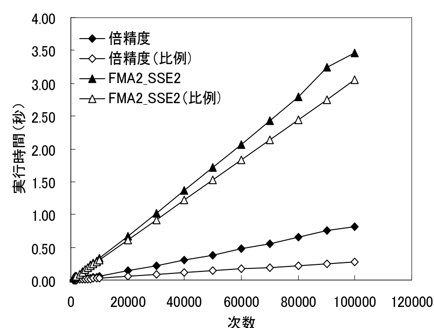

図 1 Xeon 上で $A1$ に対して BiCG 法を 50 回反復させたときの実行時間（秒）．
Fig. 1 Execution time (Sec.) of 50 BiCG iterations for matrix $A1$ on Xeon.

次に，Xeon 上で (1) の係数行列 $A2$ の γ を 1.0 から 1.4 まで変化させたときの実行時間, 反復回数, 残差ノルムを表 9 に示す．実行時間の単位は秒である．また，次数は $100,000$ とした．表中の "–" は 1000 回の反復で収束しなかったことを意味する．また，$\gamma = 1.3$ のときの残差履歴を図 2 に示す．この表と図から
- $\gamma = 1.2$ までは倍精度と 4 倍精度ともに収束している
- $\gamma = 1.3$ 以上では倍精度は停滞しているのに対して 4 倍精度は収束している
- FORTRAN と FMA2_SSE2 はほぼ同等の収束性である

表 5 Xeon 上で $A1$ に対して BiCG 法を 50 回反復させたときの実行時間（秒）.
Table 5 Execution time (Sec.) of 50 BiCG iterations for matrix $A1$ on Xeon.

次数	倍精度	FORTRAN	FMA	FMA_SSE2	FMA2_SSE2
100	0.00034	0.02141	0.00646	0.00492	0.00312
1,000	0.00279	0.20786	0.06356	0.04818	0.03032
10,000	0.05717	2.01729	0.66783	0.52825	0.33641
100,000	0.82734	20.09851	6.81664	5.29807	3.46160
1,000,000	8.44022	199.23818	68.93871	53.41777	34.91778
平均	1.86557	44.31665	15.29864	11.85944	7.74985

表 6 Xeon 上で $A1$ に対して BiCG 法を 50 回反復させたときの倍精度・FORTRAN・FMA の実行時間に対する相対性能.
Table 6 Relative performance to execution time of double precision, FORTRAN and FMA of 50 BiCG iterations for matrix $A1$ on Xeon.

	倍精度				FORTRAN			FMA	
次数	FORTRAN	FMA	FMA_SSE2	FMA2_SSE2	FMA	FMA_SSE2	FMA2_SSE2	FMA_SSE2	FMA2_SSE2
100	0.02	0.05	0.07	0.11	3.32	4.36	6.86	1.31	2.07
1,000	0.01	0.04	0.06	0.09	3.27	4.31	6.86	1.32	2.10
10,000	0.03	0.09	0.11	0.17	3.02	3.82	6.00	1.26	1.99
100,000	0.04	0.12	0.16	0.24	2.95	3.79	5.81	1.29	1.97
1,000,000	0.04	0.12	0.16	0.24	2.89	3.73	5.71	1.29	1.97
平均	0.03	0.09	0.11	0.17	3.09	4.00	6.24	1.29	2.02

表 7 Opteron 上で $A1$ に対して BiCG 法を 50 回反復させたときの実行時間（秒）.
Table 7 Execution time (Sec.) of 50 BiCG iterations for matrix $A1$ on Opteron.

次数	倍精度	FORTRAN	FMA	FMA_SSE2	FMA2_SSE2
100	0.00037	0.00826	0.00669	0.00457	0.00316
1,000	0.00357	0.08037	0.06687	0.04561	0.03171
10,000	0.04402	0.78882	0.68794	0.47166	0.33072
100,000	0.67599	7.95484	7.03384	4.84892	3.47609
1,000,000	7.19525	79.85377	70.19153	48.42959	34.90340
平均	1.58384	17.73721	15.59738	10.76007	7.74901

表 8 Opteron 上で $A1$ に対して BiCG 法を 50 回反復させたときの倍精度・FORTRAN・FMA の実行時間に対する相対性能.
Table 8 Relative performance to execution time of double precision, FORTRAN and FMA of 50 BiCG iterations for matrix $A1$ on Opteron.

	倍精度				FORTRAN			FMA	
次数	FORTRAN	FMA	FMA_SSE2	FMA2_SSE2	FMA	FMA_SSE2	FMA2_SSE2	FMA_SSE2	FMA2_SSE2
100	0.05	0.06	0.08	0.12	1.23	1.81	2.61	1.47	2.12
1,000	0.04	0.05	0.08	0.11	1.20	1.76	2.53	1.47	2.11
10,000	0.06	0.06	0.09	0.13	1.15	1.67	2.39	1.46	2.08
100,000	0.08	0.10	0.14	0.19	1.13	1.64	2.29	1.45	2.02
1,000,000	0.09	0.10	0.15	0.21	1.14	1.65	2.29	1.45	2.01
平均	0.06	0.07	0.11	0.15	1.17	1.71	2.42	1.46	2.07

ことが分かる.

直接解法における反復改良法のように，倍精度で収束しないときは，倍精度で解いた解（あるいは途中の値）を初期値として 4 倍精度で解く（演算精度を変えたリスタート），あるいは何回かの反復に 1 度だけ高精度演算をすることが可能となる．$\gamma = 1.3$，倍精度の収束判定基準を 10^{-6} とした場合は

- 倍精度の反復回数 35 回，実行時間 0.44886 秒
- 4 倍精度の反復回数 69 回，実行時間 3.98022 秒

となり最初から 4 倍精度で解いた場合 7.40245 秒と比べて 1.67 倍も速くなっている.

5. まとめ

本稿では，反復解法ライブラリ Lis に対する 4 倍精度演算の実装と SSE2 を用いて高速化について述べた．数値実験で FMA2_SSE2 は FMA と比較して平均 2.05 倍，FORTRAN と比較して平均 4.33 倍の高速化を示した．われわれがターゲットとしているのは大規

表 9 Xeon 上での行列 $A2$ のサイズを $100,000$ とした場合の BiCG 法の実行結果.
Table 9 Result of BiCG method when size of matrix $A2$ is $100,000$ on Xeon.

	倍精度			FORTRAN			FMA2_SSE2		
γ	実行時間	反復回数	残差	実行時間	反復回数	残差	実行時間	反復回数	残差
1.0	0.78266	58	1.84E-10	18.60900	58	1.84E-10	3.78737	58	1.84E-10
1.1	0.94361	70	2.23E-10	22.70676	70	2.23E-10	4.58301	70	2.23E-10
1.2	1.17442	86	3.03E-10	27.97903	86	3.03E-10	5.61647	86	3.03E-10
1.3	–	–	–	36.25341	113	2.47E-10	7.40245	113	2.47E-10
1.4	–	–	–	70.18287	155	2.85E-10	10.11311	155	2.85E-10

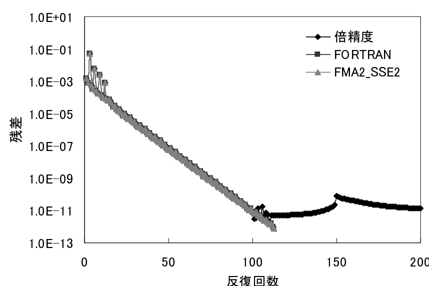

図 2 残差履歴（$\gamma = 1.3$）.
Fig. 2 Residual history ($\gamma = 1.3$).

模行列であるのでキャッシュに収まらない次数で比較すると，FMA2_SSE2 の計算時間は倍精度の約 4.5 倍程度で抑えられている．今回実験したデータでは，行列とベクトルすべてが 4 倍精度である FORTRAN の 4 倍精度と行列 A と右辺 b は倍精度でその他が 4 倍精度というわれわれの実装は，反復解法の数値的な収束特性はほぼ同等であるが，計算時間は約 4.3 倍と十分実用的である．

Lis に対して 4 倍精度演算，混合精度演算にすることで反復解法の収束の改善を図る新たな手法が可能となった．また，精度を上げれば ILU 前処理などの逐次的な前処理を利用しなくとも少ない反復回数で収束する可能性がある．さらに，メモリアクセスの高速化は期待できないが CPU コアを増やし並列処理することで演算性能を高速化している現状を考えると，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く並列化には適している．したがって，並列化によっては倍精度との性能差は縮まりより現実的になると思われる．

より精度が必要なら，Hida[11] らが開発した倍精度浮動小数を 4 個用いた"quad-double" 精度のアルゴリズムを用いることで 8 倍精度が実現可能である．また, 最近注目を集めている Cell のような単精度演算が高速な CPU で倍精度反復解法を実現するには, 本稿で提案した 4 倍精度・倍精度のアルゴリズムを倍精度・単精度用に修正すればよい.

今後の課題として，4 倍精度をうまく活用して収束性を向上，安定して解けるロバストなライブラリの開発があげられる．

参 考 文 献

1) http://ssi.is.s.u-tokyo.ac.jp/lis/
2) H. Hasegawa. Utilizing the Quadruple-Precision Floating-Point Arithmetic Operation for the Krylov Subspace Methods. the 8th SIAM Conference on Applied Linear Algebra, 2003.
3) T. Kohno, H. Kotakemori and H. Niki. Improving the Modified Gauss-Seidel Method for Z-matrices. Linear Algebra and its Applications, Vol. 267, pp. 113–123, 1997.
4) 藤井昭宏, 西田晃, 小柳義夫. 領域分割による並列 AMG アルゴリズム. 情報処理学会論文誌, Vol. 44, No. SIG6(ACS1), pp. 1–8, 2003.
5) 阿部邦美, 張紹良, 長谷川秀彦, 姫野龍太郎. SOR 法を用いた可変的前処理付き一般化共役残差法. 日本応用数理学会論文誌, Vol. 11, No. 4, pp. 157–170, 2001.
6) R. Bridson and W. P. Tang. Refining an approximate inverse. J. Comput. Appl. Math., Vol. 123, pp. 293–306, 2000.
7) N. Li, Y. Saad and E. Chow. Crout version of ILU for general sparse matrices. SIAM J. Sci. Comput., Vol. 25, pp. 716–728, 2003.
8) Intel Fortran Compiler User's Guide Vol I.
9) D. H. Bailey. A fortran-90 double-double library. http://www.nersc.gov/~dhbailey/mpdist/mpdist.html.
10) R. Barrett, et al.. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, 1994.
11) Y. Hida, X. S. Li and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. Proceedings of the 15th Symposium on Computer Arithmetic, pp.155–162, 2001.
12) T. Dekker. A floating-point technique for extending the available precision. Numerische Mathematik, vol.18 pp.224–242, 1971.
13) 西田晃. SSI: 大規模シミュレーション向け基盤ソフトウェアの概要. 情報処理学会研究報告, 2004-HPC-098, pp. 25–30, 2004.