4倍精度計算を利用した連立一次方程式の高速数値解計算の試み

A Trial Approach to Accelerated Numerical Solutions of Linear Equations Combined with Quadruple Precision Floating-point Arithmetic

幸谷智紀*
Tomonori KOUYA*

Abstract: As current numerical computations applied to various scientific simulations are larger, the number of these computations which must require more precise floating-point arithmetic over IEEE754 double precision are increasing. We have already proposed two kinds of accelerated multiple precison numerical storategies based on direct method to obtain more precise approximation of the linear system of equation to be solved: one is the application of mixed precision iterative refinement method for well-conditioned linear equations, the other is LU decomposition with Strassen's matrix multiplication algorithms to ill-conditioned problems. In this paper, we implemented these two storategies with Bailey's qd library and our BNCpack based on MPFR library, and then show that our codes are effective in the different conditioned linear equations.

1. 初めに

数値計算の大規模化に歩調を合わせるかのように，IEEE754倍精度 (2進仮数部 53bits，10進約 16 桁) 以上の浮動小数点演算を要求する科学技術シミュレーションが増えている．それに伴い，倍精度を組み合わせた 4 倍精度 (Quadruple precision, 倍々精度 (double-double, DD) とも呼ぶ) や 8 倍精度 (Octuple precision) 演算，更にそれを超えた多倍長精度 (Multiple precision) 演算の研究が盛んに行われ，その成果を利用した計算が随所で普通に利用されている．しかし，これらの計算はソフトウェアを用いて行う必要があり，ハードウェアベースの単精度・倍精度浮動小数点数単体の計算に比べると著しく遅くなる．そのため，新しいコンピュータアーキテクチャの機能を生かした高速化や，アルゴリズムの工夫による効率化が常に求められている．

昨年度，我々は任意精度の浮動小数点演算が可能な MPFR / GMP ベースの数値計算ライブラリ BNCpack の機能を用いて，Strassen の行列積アルゴリズムが倍精度演算よりも，多倍長演算に対しては極めて有効であることを示した [10]．更に，その成果を生かして LU 分解の高速化が可能であることも示しており [5]，特に大規模・高精度な計算が必要となる悪条件な連立一次方程式問題に対しては有効に使用できる．またこれとは別に，良条件の問題に対しては混合精度反復改良法が有効であることも既に示している [11]．問題の性質に応じてこれらの解法を組み合わせることで，ユーザの要求精度に応じて最短の計算時間で数値解を求めることが期待できる．

しかし，そのためには現在の多倍長精度演算において，精度に応じた最高性能のライブラリを適宜選択して利用することが必要である．現状，8 倍精度までは Bailey の qd ライブラリ [2]，および，その派生ライブラリが利用されることが多い．これらは倍精度浮動小数点数を組み合わせたものであり，比較的低精度 (10 進約 64 桁以下) で済む数値計算には有効であり，GPU 上での実装 [12] を含め，現在まで様々な実装が提供され続けている．それに対して，任意長の仮数部を利用できる MPFR/GMP は過去 20 年に渡る CPU アーキテクチャにアセンブラレベルで最適化を行い続けており，実装が難しいとされる整数演算ベースのライブラリであるにも関わらず，最高性能を維持し続けている．しかし直接 qd との比較を行った結果は公になっておらず，どの程度のアドバンテージが qd あるいは MPFR/GMP にあるのかは判然としない．

本稿では，昨年度までの連立一次方程式の解の計算に対する成果を qd ライブラリを用いて行った結果と比較することで，どの程度の性能向上が見込めるかを確認する．

2. 直接法による連立一次方程式の求解

本稿で扱う実係数連立一次方程式を (1) に示す．

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

ここで，正則な係数行列 $A \in \mathbb{R}^{n\times n}$ および定数ベクトル $\mathbf{b} \in \mathbb{R}^n$ は既知とし，各要素は要求精度に応じて正しく丸められた浮動小数点数として表現されているものとする．

有限桁の浮動小数点演算を用いた場合，丸め誤差の影響により，数値解 $\overline{\mathbf{x}}$ に含まれる誤差は，係数行列 A の条件数 $\kappa(A) = \|A\|\|A^{-1}\|$ に比例して拡大する可能性があることは良く知られている．条件数が大きい問題を悪条件問題，小さい問題を良条件問題と呼ぶ．浮動小数点演算の精度に応じてこれらの基準は変化する．

使用する浮動小数点演算の桁数に対して条件数が小さい良条件問題に対しては混合精度反復改良法が有効であり，悪条件問題に対しては LU 分解を経由した直接法を使用するのが数値計算の常識である．LU 分解に対しては Strassen の行列積アルゴリズムを利用することで，多倍長計算の場合は高速化が可能である．以下，これらの解法について簡単に解説する．

2.1 直接法ベースの混合精度反復改良法のアルゴリズム

本解法を提案した Buttari らは，混合精度反復改良法は後述する収束条件を満足すれば，通常の連立一次方程式の解法を全て L 桁で計算した時に得られる近似解の精度と同程度の精度が得られるとしている．この時，計算の効率を上げるために，計算量の多い部分の計算を $S(< L)$ 桁で実行する必要がある．ここで使用する解法は，低精度でも数値的に安定しているアルゴリズムが望ましいとしており，具体的には GMRES 法や直接法を挙げている．直接法を用いる場合，部分ピボッ

2015 年 3 月 3 日 受理
*総合情報学部コンピュータシステム学科

ト選択を用いたLU分解を経由するものとすると，予めAをLU分解しておけば，解の近似値$\mathbf{x}_k$に対する残差$\mathbf{r}_k := \mathbf{b} - A\mathbf{x}_k$を定数ベクトル，未知ベクトルを$\mathbf{z}_k$とする連立一次方程式は

$$(PLU)\mathbf{z}_k = \mathbf{r}_k$$

となるので，次の解の近似値は$\mathbf{x}_{k+1} := \mathbf{x}_k + \mathbf{z}_k$となる。この反復の前に$A = PLU$として分解しておき($P$は部分ピボット選択による行の入れ替えを表現する行列)，反復過程では前進・後退代入のみ行って$\mathbf{z}_k$の計算を行う．

以上をアルゴリズムの形でまとめると以下のようになる．実際に適用する際には，S, L桁計算における指数部の違いを吸収するために残差ベクトルに対して正規化を行う必要がある。ここで，$A^{[S]}, \mathbf{b}^{[L]}$はそれぞれ$S$桁，$L$桁の浮動小数点数で表現した行列・ベクトルを意味する．

1. $A^{[L]} := A, A^{[S]} := A^{[L]}, \mathbf{b}^{[L]} := \mathbf{b}, \mathbf{b}^{[S]} := \mathbf{b}^{[L]}$
2. $A^{[S]} := P^{[S]}L^{[S]}U^{[S]}$
3. Solve $(P^{[S]}L^{[S]}U^{[S]})\mathbf{x}_0^{[S]} = \mathbf{b}^{[S]}$ for $\mathbf{x}_0^{[S]}$
4. $\mathbf{x}_0^{[L]} := \mathbf{x}_0^{[S]}$
5. For $k = 0, 1, 2, ...$
   (a) $\mathbf{r}_k^{[L]} := \mathbf{b}^{[L]} - A\mathbf{x}_k^{[L]}$
   (b) $\mathbf{r}_k'^{[L]} := \mathbf{r}_k^{[L]} / \|\mathbf{r}_k^{[L]}\|$
   (c) $\mathbf{r}_k^{[S]} := \mathbf{r}_k'^{[L]}$
   (d) Solve $(P^{[S]}L^{[S]}U^{[S]})\mathbf{z}_k^{[S]} = \mathbf{r}_k^{[S]}$ for $\mathbf{z}_k^{[S]}$
   (e) $\mathbf{z}_k^{[L]} := \mathbf{z}_k^{[S]}$
   (f) $\mathbf{x}_{k+1}^{[L]} := \mathbf{x}_k^{[L]} + \|\mathbf{r}_k^{[L]}\|\mathbf{z}_k^{[L]}$
   (g) Exit if $\|\mathbf{r}_k^{[L]}\|_2 \le \sqrt{n}\,\varepsilon_R\,\|A\|_F\|\mathbf{x}_k^{[L]}\|_2 + \varepsilon_A$

このS-L桁混合精度反復改良法の収束条件[1]より，S-L桁混合精度反復改良法は，，S桁計算のマシンイプシロンをε_Sとすると

$$\kappa(A)\varepsilon_S << 1 \tag{2}$$

でなければならないことが分かる．つまり，条件数$\kappa(A)$が大きければ，それに応じてSを大きく取れば良いことになるが，計算速度の向上は見込めなくなる．条件数が小さければ相応にSを小さくすることもできるが，そもそもL桁も必要な計算なのかという疑問が湧いてくる．従って，S-L桁混合精度反復改良法が有効なのは

- L桁の精度が必要であり，かつ$\varepsilon_S^{-1} > \kappa(A)$である時
- S, Lが固定されており，S桁計算が十分にL桁計算より高速である環境にある時

に限られる．

2.2 Strassenの行列積アルゴリズムを用いたLU分解の高速化

偶数次数nの正方行列C, Dに対して，行列積$E := CD$を計算するに際し，Strassenの行列積アルゴリズム(Strassen)は次のように行列を4等分割してブロック化して計算を行う．

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},\ D = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} \tag{3}$$

この時，$C_{ij}, D_{ij} \in \mathbb{R}^{n/2\times n/2}$である．このようにブロック化した$C$,$D$を用いて，まず次の$P_i\,(i = 1, 2, ..., 7)$の計算を行う．

$$\begin{aligned}
P_1 &:= (C_{11} + C_{22})(D_{11} + D_{22}) \\
P_2 &:= (C_{21} + C_{22})D_{11} \\
P_3 &:= C_{11}(D_{12} - D_{22}) \\
P_4 &:= C_{22}(D_{21} - D_{11}) \\
P_5 &:= (C_{11} + C_{12})D_{22} \\
P_6 &:= (C_{21} - C_{11})(D_{11} + D_{12}) \\
P_7 &:= (C_{12} - C_{22})(D_{21} + D_{22})
\end{aligned} \tag{4}$$

これらを用いて，Eの計算を次のように行う．

$$E := \begin{bmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 + P_3 - P_2 + P_6 \end{bmatrix} \tag{5}$$

更にこれを改良したWinogradのアルゴリズム(Winograd)は，偶数次数の行列C, Dを(3)のようにブロック化した後，次のように3段階で計算を行う．

$$\begin{aligned}
&S_1 := C_{21} + C_{22},\ S_2 := S_1 - C_{11},\ S_3 := C_{11} - C_{21}, \\
&S_4 := C_{12} - S_2,\ S_5 := D_{12} - D_{11},\ S_6 := D_{22} - S_5, \\
&S_7 := D_{22} - D_{12},\ S_8 := S_6 - D_{21}
\end{aligned} \tag{6}$$

$$\begin{aligned}
&M_1 := S_2S_6,\ M_2 := C_{11}D_{11},\ M_3 := C_{12}D_{21},\ M_4 := S_3S_7, \\
&M_5 := S_1S_5,\ M_6 := S_4D_{22},\ M_7 := C_{22}S_8
\end{aligned} \tag{7}$$

$$T_1 := M_1 + M_2,\ T_2 := T_1 + M_4 \tag{8}$$

(6) → (7) → (8)の順に計算した後，Eの計算を次のように行う．

$$E := \begin{bmatrix} M_2 + M_3 & T_1 + M_5 + M_6 \\ T_2 - M_7 & T_2 + M_5 \end{bmatrix} \tag{9}$$

これらの行列積アルゴリズムを用いることで演算量を減らすことができ，特に大規模な行列の積を行う際には有効になることが期待できる．反面，繰り返し再帰を行う必要があることから，行列用のメモリ確保，解放，および移動が頻繁に行われるため，単精度・倍精度演算ではあまり効果が期待できないが，多倍長計算では有用であることが示されている[10]．

このStrassenおよびWinogradの行列積アルゴリズムの応用例の一つとしてLU分解がある．行列積を用いたLU分解(ピボット選択なし)の次のようなアルゴリズムになる[3]．

今，係数行列AのLU分解を行う単位としてブロックサイズ$K(< n)$を規定する．このとき，Fig.1のようにAを分割し，LU分解計算のうち下線部分の計算に行列積を適用する．

1. 係数行列Aを$A_{11} \in \mathbb{R}^{K\times K}, A_{12} \in \mathbb{R}^{K\times(n-K)}, A_{21} \in \mathbb{R}^{(n-K)\times K}$, and $A_{22} \in \mathbb{R}^{(n-K)\times(n-K)}$の4つに分割.
2. A_{11}を$L_{11}U_{11}(= A_{11})$にLU分解し，その後A_{12}をU_{12}に，A_{21}をL_{21}に変換する．
3. $A_{22}^{(1)} := A_{22} - \underline{L_{21}U_{12}}$

上記の計算を行った後$A := A_{22}^{(1)}$として$n - K \ge 0$となるまで繰り返す．

これはピボット選択を行わない通常のLU分解と数学的には全く同じものであり，計算の順序を入れ替えて行列積を適用できるようにしただけのものである。従って，Strassen, Winogradのアルゴリズムを用いて計算時間や計算精度に変化があったとすれば，それはこれらのアルゴリズムの適用によるものと言える。

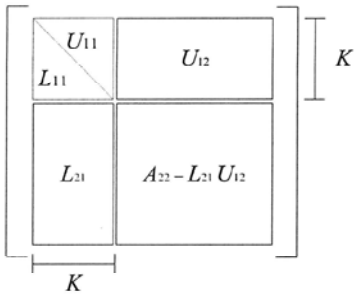

Fig. 1: 行列積を用いた LU 分解

3. 4 倍精度・8 倍精度・任意精度浮動小数点演算ライブラリの概要

ここでは今回使用した 4 倍・8 倍精度演算ライブラリ qd[2] と，多倍長精度演算ライブラリ MPFR[7]/GMP[8] の簡単な紹介を行う．

3.1 Bailey の qd ライブラリ

Bailey らによる 4 倍精度・8 倍精度演算ライブラリ qd は，倍精度浮動小数点数を繋ぎ合わせることで，4 倍精度 (106bits)，8 倍精度 (212bits) 演算を実現している．基本的な演算は C++ で記述されており，C 言語用の API も用意されている．IEEE 規格の 4 倍精度，8 倍精度に比較して仮数部や指数部の長さが若干短くなっているが，倍精度型の配列として比較的簡単に実装できるため，様々な研究において利用されている．

基本となるのは，二つの p bits 浮動小数点数 α,β を用いて行う演算 $\circ$ が丸め誤差 $e = \mathrm{err}(\alpha \circ \beta)$ を伴うことで，$s = \mathrm{fl}(\alpha \circ \beta)$ として表現されること，即ち

$$\mathrm{fl}(\alpha \circ \beta) = \alpha \circ \beta + \mathrm{err}(\alpha \circ \beta) \Leftrightarrow s = \alpha \circ \beta + e$$

となることと，次の二つの補題である．

補題 **1** $|\alpha| \geq |\beta|$ であれば，常に $|\mathrm{err}(\alpha+\beta)| \leq |\beta| \leq |\alpha|$ が成立する．

補題 **2** $s = |\mathrm{err}(\alpha+\beta)| = (\alpha+\beta) - \mathrm{fl}(\alpha+\beta)$ は必ず p bits で誤差なしで表現可能である．

このことを利用して，Quick-Two-Sum, Two-Sum アルゴリズムを実現できる．

Quick-Two-Sum(α,β)
1. $s := \mathrm{fl}(\alpha+\beta)$
2. $e := \mathrm{fl}(\beta - (\mathrm{fl}(\alpha+\beta)))$
3. return (s, e)

Two-Sum(α,β)
1. $s := \mathrm{fl}(\alpha+\beta)$
2. $v := \mathrm{fl}(s-\alpha)$
3. $e := \mathrm{fl}(\alpha - \mathrm{fl}(\mathrm{fl}(s-v) + \mathrm{fl}(\beta - v)))$
4. return (s, e)

他にも浮動小数点数の仮数部を 2 分割する Split，乗算を行う Two-Prod アルゴリズムがあり，これらを組み合わせることで 4 倍精度，8 倍精度を実現する．従って，4 倍精度では二つの倍精度浮動小数点数が，8 倍精度では 4 つの浮動小数点数が同符号，同指数部を持つ．そのため，倍精度への丸めは最初の浮動小数点数を返すだけでよい．

3.2 MPFR/GMP に基づいた BNCpack

MPFR(GNU MPFR) は GMP(GNU MP) の自然数演算 (mpn) カーネルを土台として実装された IEEE754 浮動小数点演算と同等の機能を備えた多倍長浮動小数点演算ライブラリである．GMP と切り離して使用することはできないため，本稿では MPFR/GMP と略記する．これらのソフトウェア構造を Fig.2 に示す．

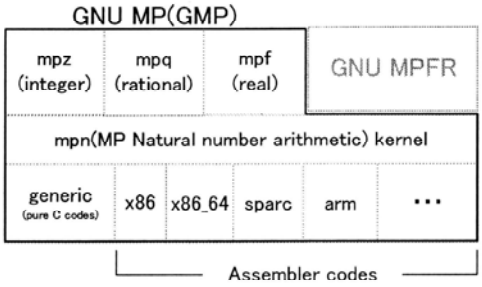

Fig. 2: MPFR/GMP のソフトウェア構造

このような構造のため，MPFR の演算性能は下支えとなる GMP の mpn カーネルの速度に依存して決まる．mpn カーネルは既に 20 年以上の CPU アーキテクチャチューニングを経ており，x86 系の CPU では最高速の性能を誇る．そのため，他にも mpn カーネルに依存した多倍長計算の実装が存在し，MPFR は早くからその一つとしてバージョンを重ねてきた最古参の多倍長浮動小数点演算ライブラリである．

我々は既にこの MPFR/GMP を土台とした数値計算ライブラリ BNCpack を実装し，保守改良を行っている [9]．今回初めて上述の qd ライブラリの機能を実装するにあたり，この BNCpack の線型計算 API のフレームワークをそのまま用いて利用することにする．なお MPFR/GMP, qd には互いにリンクする機能はないが，中田真秀による LAPACK/BLAS の API を備えた多倍長線型計算ライブラリ MPACK[6] にはその機能が実現されている．今回使用した MPFR/GMP, qd の相互変換の機能は文字列変換を介した簡易なもので，我々が独自に実装したものである．

4. ベンチマークテスト

今回利用した計算環境は下記の通りである．

CPU Intel Xeon E5-2620 v2 (2.10GHz)
RAM 32GB
OS CentOS 6.5 x86_64
C compiler Intel C/C++ Version 13.1.3
Multiple Precision Library MPFR 3.1.2[7]/GMP 6.0.0a[8] + BNCpack 0.8[4], qd 2.3.15[2]

4.1 qd と MPFR の比較

まず qd と MPFR/GMP の演算ごとの速度比較を行う．利用したプログラムは MPFR サイトで公開されているプログラムを元に，qd ライブラリの C インターフェース用に提供された関数群を用いて作り替えたものである．その結果を Table 1 に示す．dd は 4 倍精度，qd は 8 倍精度を意味し，それぞれと同じ仮数部桁 (106bits, 212bits) にして計算した MPFR/GMP の結果もその隣に示してある．

Table 1: qd, MPFR/GMP の演算ごとのベンチマーク結果 (単位:millisecond)

	dd	MPFR(106)	qd	MPFR(212)
x+y	0.000010	0.000026	0.000159	0.000031
x*y	0.000022	0.000047	0.000290	0.000065
x/y	0.000049	0.000076	0.001190	0.000198
sqrt(x)	0.000053	0.000127	0.003624	0.000201
exp(x)	0.001199	0.003397	0.013123	0.004547
log(x)	0.001278	0.003452	0.043458	0.006378
sin(x)	0.001172	0.002811	0.013092	0.004040
cos(x)	0.001180	0.001991	0.013100	0.002930

4 倍精度演算 (dd) は MPFR/GMP より高速であり，8 倍精度演算 (qd) は MPFR/GMP より低速であることが分かる．なお qd ライブラリは近年の CPU には搭載されている SIMD 命令を一切使っていない素の C++プログラムであるため，GMP の mpn カーネル並のアセンブラチューニングを行うことで高速になる可能性が高い．また，線型計算では重要な加算・乗算に関してはクラス化された C++の*+演算子をそのまま用いた方が効率が良い．今回の結果はあくまでオリジナルの C++ソースをコンパイルしただけの qd ライブラリの C インターフェースを用いたものとの比較であることを強調しておきたい．

この結果より，以下のベンチマーク結果は qd ライブラリの 4 倍精度 (dd) と MPFR(106 bits) の計算結果のみ示す．

4.2 混合精度反復改良法

ここでは係数行列として，[0,1] 区間における一様乱数行列 $R \in \mathbb{R}^{n\times n}$ と，対角行列 $D = \mathrm{diag}(n-i+1)$ を用いて生成した $A := R^{-1}DR$ を使用する．真の解は $\mathbf{x} = [1\ 2\ \ldots\ n]^T$ とし，定数ベクトル $\mathbf{b} := A\mathbf{x}$ は使用精度まで正しく得られているものを使用した．また，混合精度反復改良法における停止則は $\varepsilon_R = 2^{-L}, \varepsilon_A = 0$ とし，倍精度-MPFR(double-MPFR)，4 倍精度-MPFR(dd-MPFR) および MPFR/GMP による LU 分解を用いた直接法と計算時間，最大相対誤差を求めた．その結果を Table 2 に示す．

得られた数値解の精度はどれも 10 進 1〜2 桁程度の範囲で一致している．反復改良法は S 桁に比して L 桁が大きくなると反復数が増えるため，$L \geq 2048$ では 4 倍精度-MPFR による反復改良法の方が，倍精度-MPFR よりも計算時間が少なくなっている．この結果より，ユーザの要求精度が大きく，良条件問題であれば 4 倍精度-MPFR による反復改良法は有用と言える．

4.3 Strassen の行列積アルゴリズムと LU 分解

ここでは 4 倍精度 (dd) と，同じ桁数の MPFR の Strassen および Winograd の行列積アルゴリズムを用いたベンチマーク結果を示す．使用した行列 $E := CD$ は次の通りである．

$$C = \left[\sqrt{5}\,(i+j-1)\right]_{i,j=1}^{n},\ D = \left[\sqrt{3}\,(n-i)\right]_{i,j=1}^{n}$$

正方行列積の計算時間結果を Table 3 に示す．Simple は 3 重ループによる単純行列積，Block は 32×32 ごとにブロック化して行列積を計算したものであり，Strassen, Winograd はそれぞれの行列積アルゴリズムを使用した結果である．計算精度の差は殆どなく，dd の方が MPFR(106 bits) より全体的に 10 進 1 桁程度良い程度である．

Table 4: dd 演算による LU 分解の計算時間 (単位:second)

	n:次元数				
K	256	512	1024	2048	4096
32	0.27	2.2	17.4	138.4	1124.0
64	0.25	2.0	15.8	124.9	1003.8
96	0.26	1.9	14.6	114.1	905.7
128	0.27	2.0	14.9	114.1	928.4
160	0.28	2.0	14.1	106.3	836.6
192	0.29	2.0	14.3	106.3	829.2
224	0.3	2.1	14.4	105.9	831.3
256	0.3	2.2	14.7	107.2	861.2
288	0.29	2.2	14.6	103.5	870.3
320	0.29	2.3	14.8	103.4	914.6
Normal LU	0.29	2.4	18.6	148.9	1249.8

下線はその次元数における最小計算時間を意味する．

dd 演算に対しても，MPFR 演算に対しても，キャッシュヒット率の影響により，単純行列積の計算時間は n^3 に比例しておらず，ところどころ極端に計算時間を要している．対してブロック化した行列積アルゴリズムはどちらも次元数にほぼ比例した計算時間の増加傾向を示しており，昨年度，より高精度の MPFR 演算を使用して得られた結果を踏襲している．Strassen, Winograd のアルゴリズムの効果はどちらも次元数が上がるにつれて高まっていることが示されている．dd 演算ではどちらも同程度の計算時間となっているが，MPFR では Winograd の方が若干計算時間が短くなる傾向がある．

この Strassen, Winograd のアルゴリズムを用いた LU 分解の計算時間を示したのが Table 4 である．係数行列 A は Frank 行列，解として $\mathbf{x} = [0\ 1\ \ldots\ n-1]^T$ を用いた．ブロック単位 K は両アルゴリズムを適用する最小行列サイズ 32 の倍数にしてある．なおこの結果による数値解の精度は 10 進 1 桁程度の違いしかない．

我々が既に MPFR/GMP を用いた LU 分解で得られた結果と比較すると次の相違がある．

- より高精度の場合は $K = 32\times5 = 160$ が最小計算時間であったが，dd 演算の場合は次元数が大きくなるにつれて最短の K も大きくなる傾向がある
- $n = 1024$ 次元の時の最大 24.2%の計算時間削減が実行できたが，より高精度の場合は 30%を超える削減が達成できた．

大規模な問題に対しては dd 演算による Strassen，Winograd アルゴリズムは有効に働くことが示されたと言える．

5. 結論と今後の課題

以上のベンチマーク結果より，qd ライブラリにおいて，特に 4 倍精度 (dd) 演算は MPFR/GMP 演算より高速であり，混合精度反復改良法においても，Strassen, Winograd の行列積アルゴリズムを用いた LU 分解においても，大規模問題に対しては有効であることが示された．

しかしながら，文中でも述べた通り，今回の結果は低速な qd の C インターフェースを用いて得られたものであり，実際にはもっと高性能な計算が可能であると予想される．また，全てシリアルに計算したものであるため，現状のメインストリー

Table 2: 混合精度反復改良法の計算時間 (単位: second)(数値解の最大相対誤差の常用対数, 反復回数)($n = 512$)

	L(*bits*)			
	256 (lg(relerr), #Iter.)	512 (lg(relerr), #Iter.)	1024 (lg(relerr), #Iter.)	2048 (lg(relerr), #Iter.)
double-MPFR	0.76(-73.32, 3)	1.29(-152.6, 9)	3.95(-306.7, 19)	15.18(-615.67, 39)
dd-MPFR	4.19(-75.59, 1)	4.43(-151.9, 3)	5.79(-306.7, 8)	11.87(-614.40, 18)
MPFR LU	6.94(-74.55)	10.23 (-151.4)	21.05(-305.5)	48.78(-613.44)

Table 3: dd および MPFR/GMP 演算による正方行列積の計算時間 (単位：second)

	dd				MPFR(106 bits)			
n	Simple	Block	Strassen	Winograd	Simple	Block	Strassen	Winograd
127	0.10	0.11	0.09	0.08	0.22	0.25	0.19	0.19
128	0.11	0.11	0.08	0.08	0.25	0.25	0.18	0.18
129	0.10	0.20	0.09	0.09	0.22	0.28	0.20	0.20
191	0.33	0.36	0.27	0.26	0.76	0.84	0.59	0.59
192	0.35	0.35	0.26	0.26	0.87	0.85	0.57	0.58
193	0.34	0.56	0.27	0.26	0.78	0.91	0.61	0.62
255	0.79	0.84	0.61	0.59	1.85	1.99	1.27	1.29
256	0.95	0.84	0.61	0.59	2.15	2.00	1.23	1.23
257	0.82	1.19	0.61	0.61	1.89	2.14	1.30	1.29
383	2.78	2.80	1.86	1.85	6.71	6.75	4.11	4.06
384	3.06	2.83	1.86	1.81	7.82	6.78	4.05	3.98
385	2.82	3.59	1.87	1.87	6.93	7.05	4.13	4.12
511	6.53	6.66	4.27	4.33	17.44	15.99	8.35	8.34
512	7.64	6.68	4.23	4.26	19.05	16.06	8.34	8.21
513	6.81	8.01	4.26	4.36	17.30	16.45	8.59	8.42
767	22.35	22.39	13.06	13.20	64.98	54.19	28.33	28.35
768	25.26	22.71	13.21	13.12	64.50	54.32	27.99	28.25
769	22.50	25.29	13.53	13.14	67.51	55.30	28.36	28.33
1023	53.43	53.15	30.46	29.83	167.13	128.16	56.13	57.01
1024	68.31	53.11	30.25	29.48	312.80	128.63	55.98	55.39
1025	54.69	58.38	30.08	29.65	136.70	130.05	57.11	56.49
1535	179.77	179.33	91.95	90.72	561.01	435.31	195.30	195.22
1536	236.99	180.79	91.82	92.99	1017.75	436.73	191.74	194.03
1537	180.84	191.22	92.15	91.67	494.37	446.05	195.33	194.86

ムアーキテクチャであるマルチコア，メニーコア環境に適した並列化も行っておらず，さらなる性能向上が可能であると思われる．

従って，今後の課題は更なる高速化を目指すため，次の 2 点を中心に行っていきたい．

1. qd の C インターフェースの高速化
2. マルチコア CPU，メニーコア GPU を用いた並列化

謝辞

今回利用した計算環境は私学助成金の援助により 509 教室に設置してあるものである．導入にあたりご尽力いただいた杉本・前経理課長はじめとする関係各位に感謝致します．

参考文献

1) A.Buttari, J.Dogarra, Julie Langou, Julien Langou, P.Luszczek, and J.Karzak. Mixed precision iterative refinement techniques for the solution of dense linear system. *The International Journal of High Performance Computing Applications*, Vol. 21, No. 4, pp. 457–466, 2007.
2) D.H. Bailey. QD. `http://crd.lbl.gov/~dhbailey/mpdist/`.
3) G.H.Golub and C.F.van Loan. *Matrix Computations (4th ed.)*. Johns Hopkins University Press, 2013.
4) Tomonori Kouya. BNCpack. `http://na-inet.jp/na/bnc/`.
5) Tomonori Kouya. Accelerated multiple precision matrix multiplication using Strassen's algorithm and Winograd's variant. *JSIAM Letters*, Vol. 6, pp. 81–84, 2014.
6) MPACK. Multiple precision arithmetic BLAS and LAPACK. `http://mplapack.sourceforge.net/`.
7) MPFR Project. The MPFR library. `http://www.mpfr.org/`.
8) T.Granlaud and GMP development team. The GNU Multiple Precision arithmetic library. `http://gmplib.org/`.
9) 幸谷智紀. MPFR/GMP を用いた多倍長数値計算ライブラリ BNCpack について. 応用数理, Vol. 21, No. 3, pp. 197–206, sep 2011.
10) 幸谷智紀. Strassen のアルゴリズムによる倍精度・多倍長正方行列積の高速化. 静岡理工科大学紀要, Vol. 22, pp. 35–40, 2014.
11) 幸谷智紀. 倍精度と多倍長精度浮動小数点数を用いた反復改良法による連立一次方程式の高精度高速解法について. 日本応用数理学会論文誌, Vol. 19, No. 3, pp. 313–328, 2009-09-25.
12) 椋木大地, 高橋大介. GPU による 4 倍精度 BLAS の実装と評価. 情報処理学会研究報告. 計算機アーキテクチャ研究会報告, Vol. 2009, No. 13, pp. 1–6, nov 2009.