多倍長精度プログラムの自動生成機構 Xev-GMP における混合精度プログラムの生成と評価

斯波柾[†1] 菱沼利彰[†2] 田中輝雄[†1] 藤井昭宏[†1] 平澤将一[†3]

概要：大規模数値計算において高精度計算の需要が高まっている．高精度計算の実現手段の一つに GNU Multiple Precision Arithmetic Library (GMP)の利用が挙げられる．GMP を用いたプログラムは，一般的な浮動小数点数の基本データ型を用いた C 言語のプログラムから様々な変更が必要なため実装コストが問題になる．
我々はこれまで，倍精度浮動小数点数を用いた C のプログラム（C コード）から，GMP の高精度計算を行うプログラム（GMP コード）を自動生成する機構である Xev-GMP を開発してきた．しかし，複数の精度の自由な組み合わせを指定できないため，必要なメモリデータサイズや計算量の増加が問題となっていた．
本研究では，ユーザがディレクティブを用いて変数ごとの精度情報を与えるだけで，複数の精度を組み合わせた混合精度プログラムの自動生成機構を開発した．これにより，ユーザは精度情報を入力するだけで変数ごとに必要な精度を自由に変えることができ，問題規模・アルゴリズムに応じて精度依存性を容易に検証することが可能となる．
我々は CG 法のプログラムを対象に混合精度プログラムの生成を行い，その有用性を評価した．

1. はじめに

大規模数値シミュレーションの核である Krylov 部分空間法は，丸め誤差に影響を受ける．そのため，高精度演算を用いれば収束を改善できる事例が報告されている [1][2]．高精度演算の実現手段の一つに，GMP (GNU Multiple Precision Arithmetic Library) [3]の利用が挙げられる．GMP は任意の精度で演算ができるライブラリである．

GMP を用いた多倍長精度プログラム（GMP コード）は，すべての処理を手続きの形で書き，算術演算などは一時変数を用いて単項式にする必要があり，実装コストが高い．

その特有の記述のため C 言語の倍精度浮動小数点数を用いたプログラム（C コード）から正規表現等を用いた単純置換では不可能である．そこで我々は，コードを構文木として扱い，変換規則を作成できる Xevolver Framework [4][5]を用いて，C コードから，GMP コードを自動生成する機構 Xev-GMP を実装している [6][7][8]．

このとき，高精度演算ではひとつひとつのデータサイズが大きくなり全体のメモリデータサイズが増加し，それに呼応して計算量も増大する．

この問題に対して，Xev-GMP では変数ごとの細かい精度指定を可能とし，複数の精度を組み合わせた混合精度演算を行う GMP コードの生成を行う拡張が必要となる．

本研究では，自動生成機構 Xev-GMP の混合精度 GMP コードの生成を可能とする拡張を提案し，実装を行う．

この拡張した Xev-GMP を用いて，共役勾配法の C コードを対象に GMP コードを自動生成し，これを評価した．

以下 2 章で関連研究，3 章で Xev-GMP の設計思想，3 章で Xev-GMP の設計と実装，4 章で実験と評価，5 章でまとめについて説明する．

†1 工学院大学
†2 筑波大学
†3 東北大学

2. 関連研究

GMP を利用したライブラリとして，幸谷らの BNCPACK [9]，中田らの MPACK [10]がある．たとえば MPACK は数値計算ライブラリである BLAS や LAPACK に含まれる処理を，GMP を用いて計算できる．このように数値計算ライブラリを GMP に対応させる試みが各所で行われている．

GMP を用いたプログラミングを容易にする方法として，C++の演算子オーバーロードを用いた方法が挙げられる [11]．ここでは，ユーザが mpf_t 型の変数や関数宣言を記述することで実現されている．

3. Xev-GMP の設計思想

3.1 GMP

GMP は任意の精度で算術演算を行うライブラリである．GMP を用いて高精度演算を行うには，GMP 独自の高精度型と関数を用いる必要があり，実装コストが高い．

GMP では，任意多倍長浮動小数点型として“mpf_t”という構造体を定義している．mpf_t 型は初期化時に仮数部の bit 数を定義し，異なる仮数部との演算も同一の形式で可能であるが，倍精度や単精度との演算は不可能である．

表 1 に mpf_t 型のメンバ変数を示す. 仮数部は mp_limb_t のサイズに分割され配列として格納される．_mp_prec は仮数部の配列の最大長を表し，_mp_size は仮数部の配列中に意味のある値が入っている末尾を指している．mp_exp_t と，mp_limb_t は整数型であり，GMP をビルドする環境によってサイズが変わり，我々の実験環境では 64bit である．

mpf_t 型のデータサイズは以下の式で求まる. 式中の prec は指数部の bit 数である．

$$\text{mpf_t 型のデータサイズ} = \mathrm{sizeof(int)} * 2 + \mathrm{sizeof(mp_exp_t)} + \mathrm{sizeof(mp_limb_t} *) + \mathrm{sizeof(mp_limb_t)} * (\mathrm{prec}/\mathrm{sizeof(mp_limb_t)} + 2)$$

上式において int は 32bit，mp_exp_t，mp_limb_t*，mp_limb_t は 64bit とすると，精度が 128bit の場合 48Byte,

ⓒ2016 Information Processing Society of Japan

C コード	GMP コード
1. double a, b, c; 2. a = 0.0; 3. d = 1.0; 4. c = 2.0; 5. a = b * c; 6. printf("%lf¥n", a);	1. mpf_t a, b, c; 2. mpf_init2(a, 128); 3. mpf_init2(b, 128); 4. mpf_init2(c, 128); 5. mpf_set(a, 0.0); 6. mpf_set(b, 1.0); 7. mpf_set(c, 2.0); 8. mpf_mul(a, b, c); 9. gmp_printf("%Ff¥n", a); 10. mpf_clear(a); 11. mpf_clear(b); 12. mpf_clear(c);

図 1　C コードと GMP コードの比較
（左：C コード，右：GMP コード）

256bit の場合 64Byte となる．

mpf_t 型の変数は指定する精度によって仮数部を動的に確保するため，使用前に初期化関数に，使用後に解放関数に渡す必要がある．また，構造体のため代入・演算に演算子は使えず，関数を用いる必要がある．

表 2 に以降の章で使用する GMP の関数を示す．表中の dst，src，src1，src2 は mpf_t 型の変数であり，prec は int 型の数値または変数である．

図 1 に，同一の処理を C コードと GMP コードで記述した例を示す．C コードの 1 から 4 行目が GMP コードの 1 から 7 行目に，C コードの 5 行目が GMP コードの 8 行目に，C コードの 6 行目が GMP コードの 9 行目に対応し，GMP コードの 10 行目以降では変数の解放を行う．

図 1 のとおり，GMP コードは C コードから多くの変更が必要である．演算は C++の演算子オーバーロードを用いれば対応が可能だが，変数の型宣言・関数宣言は書き換えを必要とする．

我々はこの問題を解決するため，Xevolver Framework をベースに GMP コードの自動生成機構を構築した．

3.2 Xevolver Framework

Xevolver Framework [4][5]は東北大学の滝沢らが開発した ROSE [12]ベースのフレームワークである．ROSE は C/C++, Fortran のプログラムを解析，分解，再構築できる．

Xevolver Framework は ROSE により生成された構文木を XML (Extensible Markup Language) [13]で表現された構文木に変換する．XML は“タグ”と呼ばれる特定の文字列を用いて，文書やデータの意味や構造を記述するための，マークアップ言語の一つである．

表 1　mpf_t のメンバ変数

型	変数名	備考
int	_mp_prec	仮数部の配列の最大長
int	_mp_size	仮数部の値がある末尾を指す
mp_exp_t	_mp_exp	指数部
mp_limb_t*	_mp_d	仮数部のポインタ

表 2　GMP の関数（初期化，代入，解放，演算）

#	関数名	動作
1	mpf_init2(dst, prec)	mpf_t 型の変数を，精度を指定して初期化
2	mpf_set(dst, src)	mpf_t 型の変数の代入
3	mpf_set_d(dst, value)	mpf_t 型の変数に double 型の値を代入
4	mpf_add(dst, src1, src2)	mpf_t 型の変数の和を計算
5	mpf_sub(dst, src1, src2)	mpf_t 型の変数の差を計算
6	mpf_mul(dst, src1, src2)	mpf_t 型の変数の積を計算
7	mpf_div(dst, src1, src2)	mpf_t 型の変数の商を計算
8	mpf_sqrt(dst, src)	mpf_t 型の変数の平方根を計算
9	mpf_clear(dst)	mpf_t 型の変数を解放

Xevolver Framework は C のプログラムと, XML で表された C のプログラムの構文木の相互変換が可能である．

我々は，このフレームワークを用いて C コードの構文木から GMP コードの構文木を生成することで，GMP コードを得るための自動生成機構を開発する．

3.3 設計思想

C コードから GMP コードの自動生成機構を実装するにあたり，ユーザは C コードのみを編集することにより，それに対応したあらゆる精度の GMP コードが作成できることを目的とした．以下に設計思想を示す．

(1) C コード，GMP コードの両方が得られる
(2) ユーザが GMP コードの変更を必要としない
(3) 最小限のディレクティブの記述で, 複数の精度が簡単に組み合わせられる

我々は，目的（3）の実現に向けディレクティブベースの GMP コードの生成を行ってきた [6][7][8].

これまでは，すべての変数の精度を指定するディレクティブに加えて，コード中に一箇所のみ指定できるディレクティブを用いて実現していた．

本論文では，各スコープにディレクティブを用いて精度を指定することで複数の精度の組み合わせを可能とした．

これにより，ユーザが気軽にアルゴリズムの精度依存性の検証を行うことが可能となる．

Xev-GMP のディレクティブを以下に示す．

(A) Pragma xev-gmp default(prec)
(B) Pragma xev-gmp set(prec) target(var1, var2…)
(C) Pragma xev-gmp set(prec) global-target(var1, var2,…)

ここで，各ディレクティブの prec は自然数，var1，var2 は変数名を指定する．

（A）は引数に精度を指定して宣言することで，mpf_t

ⓒ2016 Information Processing Society of Japan

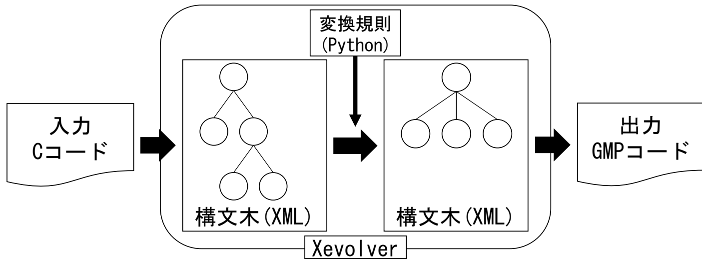

図 2　Xev-GMP の生成フロー

表 3　実装した機能

gmp.h のインクルード	構造体
デフォルト精度の指定	四則演算
変数ごとの精度の指定	比較演算子
変数宣言・初期化・解放	関数呼出
静的配列の生成	標準入出力
動的配列の生成	数学関数
グローバル変数	

型の変数のデフォルトの精度を指定できる．このディレクティブの記述位置は問わない．ユーザは最低限このディレクティブを記述するだけでよい．

（B）は関数内の変数に対して（A）で指定した精度以外の精度を指定する場合に使用する．引数に精度と変数名を指定して宣言することで，指定された変数のみ精度を変更できる．このディレクティブは関数定義の直前に記述し，記述した位置の直後の関数にのみ適用される．

（C）はグローバル変数に対して（A）で指定した精度以外の精度を指定する場合に使用する．このディレクティブの記述位置は問わない．引数は（B）と同様である．

ディレクティブは一般的なコンパイラでは無視されるため，C コード自体には影響を与えないディレクティブの記述だけで，GMP コードを得ることが可能である．これにより，1 つのコードの管理のみで C コード・GMP コードの 2 つのコード管理が可能となる．

4. Xev-GMP の設計と実装

本章では，Xev-GMP の具体的な設計と実装を説明する．

図 2 は Xev-GMP が Xevolver を用いて C コードから GMP コードを生成する際の処理フローである．

図 2 のとおり，Xev-GMP の GMP コードの生成は以下の 3 つの手順で行う．

(1) ディレクティブが記述された C コードを XML（構文木）形式のファイル（XML ファイル）に変換

(2) C コードの XML ファイルから GMP の XML ファイルを生成

(3) GMP の XML ファイルを GMP コードに変換

（1），（3）に Xevolver を，（2）に Python で実装した変換

```
1.  #pragma xev-gmp default(128)
2.  #pragma xev-gmp set(256) target(b)
3.  int main(){
4.      double a
5.      double b;
6.      ・・・
7.      return 0;
8.  }
```

```
1.   int main(){
2.       mpf_t a;
3.       mpf_init2(a, 128);
4.       mpf_t b;
5.       mpf_init2(b, 256);
6.       ・・・
7.       mpf_clear(a);
8.       mpf_clear(b);
9.       return 0;
10.  }
```

図 3　変数宣言・初期化・解放の生成コード
（左：C コード，右：GMP コード）

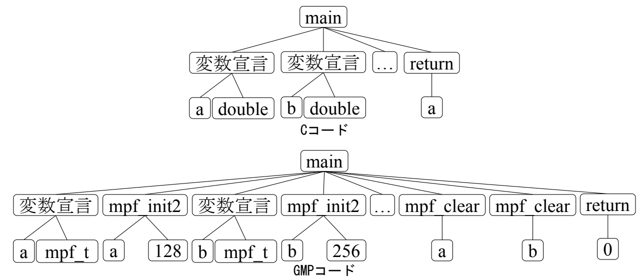

図 4　変数宣言・初期化・解放の構文木

規則を用いる．

優先度の観点から，GMP に用意されていない C の数学関数，三項演算子を非対応とした．また，今回のアプローチでは困難な，mpf_t 型では不可能な union，メモリダンプ系，ビット演算系，関数ポインタ，倍精度や単精度との混合演算を対象外とした．

定数はユーザによる入力なので高精度にはしない．

4.1 混合精度

GMP を用いたプログラムは，算術演算を単項式で行うため，C コードにはない一時変数を宣言する必要がある．また，混合精度演算を行う際には計算の中間結果を低い精度の一時変数に代入することによる影響をなくすために，式ごとに一時変数の精度を使い分ける必要がある．Xev-GMP では，式中の変数の精度を判別し自動的に適切な精度の一時変数を用いた式を生成する．

本節では，複数の精度の mpf_t 型の変数を組み合わせて演算を行う GMP コードの生成について説明する．

4.1.1 mpf_t 型の変数宣言・初期化・解放の生成

GMP コードで用いる mpf_t 型の変数は，使用前に初期化関数 (mpf_init2)に，使用後に解放関数 (mpf_clear)に渡す必要がある．図 3 の GMP コードにおいて，3，5 行目で変数の初期化を，7，8 行目で変数の解放を実行している．

変数ごとに精度を変えるため，初期化関数の生成時に引数として与える精度の値を変える必要がある．そのため，ディレクティブから精度情報を取得する際，デフォルトの精度の値と，スコープごとの変数名をキーとしたデータベ

ⓒ2016 Information Processing Society of Japan

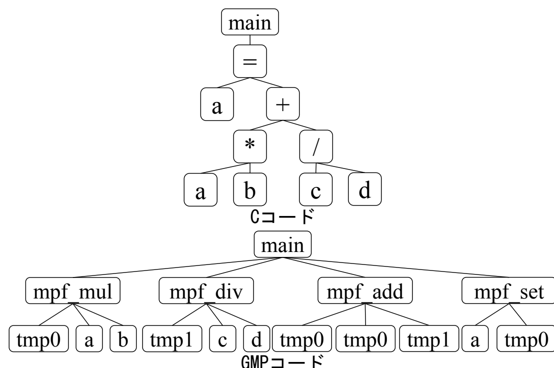

図 5　算術演算の生成

ースを作成することで精度の組み合わせに対応した.

図 4 は図 3 のコードの main 関数をルートとした構文木である. なお, ディレクティブに対応するノードは省略した.

図 4 のとおり，GMP コードの構文木では，変数宣言の型を mpf_t 型に変換し，初期化関数の呼出を変数宣言の直後に, 解放関数の呼出を return 文の直前にそれぞれ挿入した.

4.1.2 mpf_t 型の算術演算の生成

GMP の mpf_t 型の演算を行う関数は mpf_t 型の変数を返せないという制約がある．そのため，1 つの式中に複数の演算子が存在する場合，関数の戻り値を一時変数に受け取り，次の関数に渡すように式を再構築する必要がある.

そのため以下のような C コードの式の場合,

a = a * b + c / d;

以下のように計算順序と一時変数の変数名を考慮し，関数呼出の式を生成する必要がある.

```
mpf_mul(tmp0, a, b);
mpf_div(tmp1, c, d);
mpf_add(tmp0, tmp0, tmp1);
mpf_set(a, tmp0);
```

式の構文木を図 5 に示す．木の末端から順に処理することで，もとの式の計算順序を変えることなく，式を GMP の演算関数に分解することができる.

そこで，式の構文木に対して深さ優先探索の帰りがけ順で演算子を GMP の関数呼出に変換することで，式を再構築した．たとえば，図 5 の構文木の場合，*，/，+，=の順に GMP の演算関数に分解される.

一時変数の変数名は，関数および精度ごとに一時変数につける番号を管理するリストを作成し，そのリストを用い Xev-GMP は以下の手順で式の再構築を行う. 式の再構築の際に用いられる一時変数の精度は，もとの式中で使用さて変数名を決定した.

れている変数内で最も高い精度に合わせることで一時変数への代入が計算結果に悪影響を及ぼさないようにした.

(1) 式中で使われている変数名を取得し, ディレクティブの精度情報から式中の最大精度を求める

```
1.  #pragma xev-gmp default(128)
2.  #pragma xev-gmp set(256) target(B.d)
3.  struct s_a{
4.      double d;
5.  };
6.  typedef struct{
7.      double d[2];
8.      struct s_a A;
9.  }s_b;
10. int main(){
11.     s_b B;
12.     return 0;
13. }
```

```
1.  #include<gmp.h>
2.  struct s_a{
3.    mpf_t d;
4.  };
5.  typedef struct {
6.    mpf_t d[2];
7.    struct s_a A;
8.  }s_b;
9.  int main(){
10.   int i_gmp;
11.   s_b B;
12.   for (i_gmp = 0; i_gmp < 2; i_gmp++){
13.     mpf_init2(B . d[i_gmp],256);
14.   }
15.   mpf_init2(B . A . d,128);
16.   for (i_gmp = 0; i_gmp < 2; i_gmp++){
17.     mpf_clear(B . d[i_gmp]);
18.   }
19.   mpf_clear(B . A . d);
20.   return 0;
21. }
```

図 6　構造体の生成コード
（左：C コード，右：GMP コード）

```
1.  #pragma xev-gmp default(128)
2.  double d1;
3.  int main(){
4.      return 0;
5.  }
```

```
1.  #include<gmp.h>
2.  mpf_t d1;
3.  int xev_gmp_global_init();
4.  int xev_gmp_global_final();
5.  int main(){
6.    xev_gmp_global_init();
7.    xev_gmp_global_final();
8.    return 0;
9.  }
10. int xev_gmp_global_init(){
11.   int i_gmp;
12.   mpf_init2(d1,128);
13.   return 0;
14. }
15. int xev_gmp_global_final(){
16.   mpf_clear(d1);
17.   return 0;
18. }
```

図 7　グローバル変数の生成コード
（左：C コード，右：GMP コード）

(2) （1）で求めた精度の一時変数の変数名を管理するリストを初期化

(3) 図 5 のように, 深さ優先探索の帰りがけ順で演算子を GMP の関数呼出に分解，その際の一時変数名は上記の手順で生成

4.2　高度な C コードの変換

4.2.1 構造体

構造体のメンバに mpf_t 型の変数が存在する場合，構造体の変数宣言時に mpf_t 型のメンバの初期化・解放を行う必要があるため，図 6 のようなコードが必要になる.

そこで，構造体の定義からメンバの型，変数名を取得し，メンバ変数のリストを作成する．このリストを用いてメンバの初期化・解放の関数を生成することで，構造体がネストされている場合にも対応した.

4.2.2 グローバル変数

C コードのグローバル変数に double 型の変数が存在する場合，GMP コードでは，図 7 のように関数内の変数と同様

```
1.  double add(double a, double b){
2.    double tmp;
3.    tmp=a+b;
4.    return tmp;
5.  }
```

```
1.  int add(mpf_t tmp_rcv,mpf_t a,mpf_t b){
2.    mpf_t tmp_gmp_0_128;
3.    mpf_init2(tmp_gmp_0_128,128);
4.    mpf_t tmp;
5.    mpf_init2(tmp,128);
6.    mpf_add(tmp_gmp_0_128,a,b);
7.    mpf_set(tmp,tmp_gmp_0_128);
8.    mpf_set(tmp_rcv,tmp);
9.    mpf_clear(tmp);
10.   mpf_clear(tmp_gmp_0_128);
11.   return 0;
12. }
```

図 8　ユーザ定義関数の生成コード
（左：C コード，右：GMP コード）

に初期化・解放を行う必要がある．

そこで，グローバル変数に対して 3.1.1 と同様に初期化関数と解放関数を生成し，その初期化関数をまとめた関数(xev_gmp_global_init)と解放関数をまとめた関数 (xev_gmp_global_final)を生成する．

生成した xev_gmp_global_init の呼出を main 関数の最初に，xev_gmp_global_final の呼出を return 文が存在する場合は return 文の直前に，存在しない場合は main 関数の最後に挿入することでグローバル変数の初期化と解放を行った．

4.2.3 関数呼出

GMP コードでは関数から mpf_t 型の変数を返すことができない．そのため，double 型の値を返す関数は，返り値を格納する引数を追加する必要がある．しかし，他ライブラリで定義されている関数は引数を追加できない．

そこで，以下の 3 パターンに分けて実装を行った．

(1)　GMP の関数

(2)　ユーザが定義した double 型の値を返す関数（ユーザ定義関数）

(3)　GMP 以外のライブラリの関数

（1）の場合，1 対 1 の変換が可能なため，C の関数と GMP の関数の変換テーブルを用いて変換を行う．

たとえば，以下のような C コードの場合，

a = sqrt(b);

以下のように変換される．

mpf_sqrt(tmp0, b);
mpf_set(a, tmp0);

（2）の場合は，図 8 に示すように変換を行う．mpf_t 型は return できないため呼出だけでなく関数定義の書き換えが必要となる．関数の引数に返り値を格納する mpf_t 型の変数を追加し，return で返される値を追加した引数に代入することで返り値を受け取る．呼出は（1）と同じテーブルに対応関係を追加することで変換できる．

（3）の場合，関数の引数を追加することができないため返り値を代入関数(mpf_set_d)で一時変数に格納する．

以下のような C コードの場合，

a = omp_get_wtite();

次のように変換する．

mpf_set_d(tmp0, omp_get_wtime());
mpf_set(a, tmp0)

5. 実験と評価

本章では，CG 法の C コードを対象に GMP コードの生成を行い，その評価について述べる．

5.1 実験環境

使用した CPU は Intel® Xeon® E5-2650 v3 @ 2.3GHz，OS は CentOS release 6.6(Final)，コンパイラは gcc-4.4.7，コンパイルオプションは-lgmp -O2 である．

5.2 CG 法

CG 法は科学技術計算で一般に用いられている反復解法である．CG 法の核となる CGSolver 関数について説明する．

この関数は，CG 法を用いて$Ax = b$の解xを求める関数である．引数として，CRS 形式の行列データ (CRS *matrix)，初期解 (double *x)，初期値 (double *b)，反復回数の上限 (int *max_iter)，収束条件 (double *eps)を入力し，出力として，$Ax = b$の解x (double *x)，反復回数 (int *max_iter)，最終残差ノルム (double *eps)を出力する．

例として，デフォルトの精度を 64bit，CGSolver 関数内のすべての変数を 128bit にした CG 法のカーネル関数と呼出部を図 11 に示す. 200 行の C コードに 2 行のディレクティブを追加するだけで, 277 行の GMP コードが生成された．

このコードは，変数の宣言・初期化・解放，メモリの動的確保，構造体，算術演算，関数呼出，入出力関数を含む．

GMP コードの 22，23 行目は Xev-GMP が独自に定義した変数である．22 行目の i_gmp は初期化・解放に用いるイテレータ，23 行目の tmp_gmp_0_64 は算術演算に用いる一時変数である. 24 行目で初期化され, 72 行目で解放される．

次に, C コードの 27 行目で配列 r の動的確保が行われる．GMP コードの 29 行目で mpf_t 型の動的確保, 30-32 行目で各要素の初期化，67-69 行目で解放される．

C コードの 39 行目の式は, 算術演算と関数呼出が行われている．GMP コードの 45-47 行目に一時変数を用いた関数呼出の式が生成される．

C コードの 44 行目でフォーマット指定子%le を用いた printf 関数で double 型の変数 err を出力する．GMP コードの 55 行目でフォーマット指定子%Fe を用いた gmp_printf 関数に変換される．

5.3 数値実験と評価

Xev-GMP を用いた混合精度プログラムの生成について評価するため，サイズ 1,000 のヒルベルト行列とフロリダ大学の疎行列コレクション [14]からサイズ 25,710，非ゼロ要素数 3,749,582 の smt という行列を用いて評価を行った．

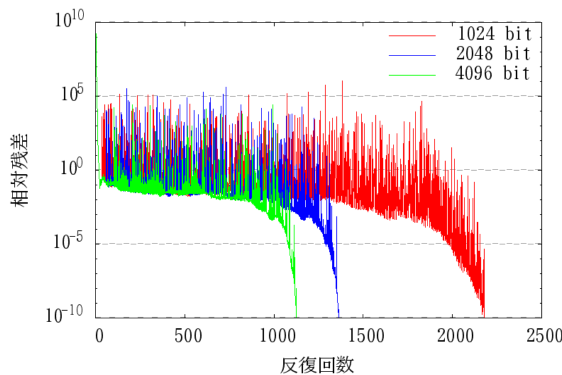

図 9　ヒルベルト行列の収束履歴

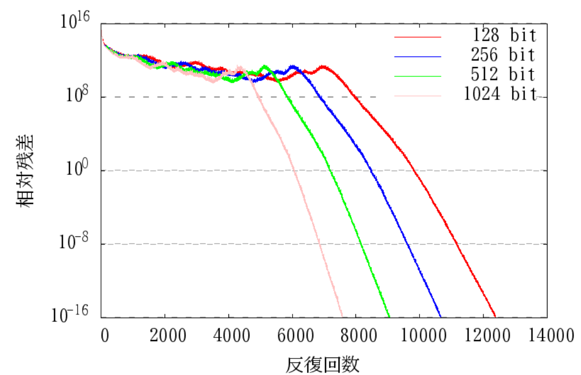

図 10　smt の収束履歴

表 4　メモリデータサイズ [MB]

	(A)	(B)	(C)	(D)
1024bit	160.8	40.8	40.8	40.8
2048bit	289.4	41.4	41.4	41.4
4096bit	546.7	42.7		

表 5　1 反復あたりの時間 [sec.]　(相対性能)

	(A)	(B)	(C)	(D)
1024bit	0.43 (1.0)	0.15 (1.0)	0.15 (1.0)	0.14 (1.0)
2048bit	1.18 (2.7)	0.22 (1.5)	0.22 (1.5)	0.22 (1.5)
4096bit	3.50 (8.1)	0.37 (2.5)		

右辺ベクトルは$b = (0,1,\dots,n-1)^T$，初期ベクトルはx_0は倍精度乱数，ヒルベルト行列の反復回数の上限は 3000 回，収束条件は10^{-10}，smt の反復回数の上限は 25,710 回，収束条件は 10^{-16} とし，10,000 秒で解けないものは打ち切った．

事前実験を行った結果，ヒルベルト行列は 256bit では解けず，4096bit 以上になると反復回数は 1100 回程度で落ち着く．smt は 64bit では解けず，128bit では解けた．

上記の結果をふまえ，Xev-GMP を用いて以下の 4 種類の GMP コードを生成した．

(A)　デフォルトの精度をヒルベルト行列では 1024 から 4096bit，smt では 128bit から 1024bit に指定
　　コードの先頭にディレクティブ（A）を記述

(B)　（A）に加え，行列 A のみ 64bit に固定
　　行列データの読み込みを行う関数 (matrix_input)の直前に，CRS 型の構造体の要素配列 val を 64bit に指定するディレクティブ（B）を記述

(C)　（B）に加え，内積計算を 4096bit で行う
　　内積関数 (dot)の直前に，返り値を格納する変数を 4096bit に指定するディレクティブ（B）を記述

(D)　（C）に加え，疎行列ベクトル積を 4096bit で行う
　　疎行列ベクトル積関数 (spmv)の直前に，演算結果を格納する変数を 4096bit に指定するディレクティブ(B)を記述

表 6　メモリデータサイズ [MB]

	(A)	(B)	(C)	(D)
128bit	186.2	156.2	156.2	156.2
256bit	248.2	158.2	158.2	158.2
512bit	372.3	162.3	162.3	162.3
1024bit	620.5	170.6	170.6	170.6

表 7　1 反復あたりの時間 [sec.]　(相対性能)

	(A)	(B)	(C)	(D)
128bit	Time out	0.36 (1.0)	0.36 (1.0)	0.38 (1.0)
256bit	Time out	0.39 (1.1)	0.39 (1.1)	0.41 (1.1)
512bit	Time out	0.48 (1.3)	0.47 (1.3)	0.47 (1.2)
1024bit	Time out	0.63 (1.7)	0.63 (1.7)	0.62 (1.6)

なお，GMP は倍精度変数との混合演算はできないため，コード（B）から（D）の要素配列 val は，GMP の仮数部 64bit で，厳密には倍精度と異なる．

たとえば，コード(C)は$a = x \cdot y$を計算するための式を，

$a \mathrel{+}= x[i] * y[i];$

ディレクティブ（B）を用いて，aを 4096bit に指定することで，以下の式が得られる．

mpf_mul(tmp_gmp_0_4096, x[i], y[i]);
mpf_add(a, a, tmp_gmp_0_4096);

これにより，演算結果を 4096bit とした式が生成されベクトルx，yを 4096bit でもつことなく，計算内部のみ高精度にした内積関数を得ることができる．

ヒルベルト行列，smt それぞれの収束履歴を図 9，図 10 に，プログラム中の全変数の合計メモリデータサイズを表 4，表 6 に，1 反復あたりの時間を表 5，表 7 に示す．

（A）と（B）の比較結果から，どちらの問題でもデータサイズが大きく，反復中に書き換えが行われない行列Aの精度を落とすことで，メモリデータ量を大幅に削減できた．

ヒルベルト行列の場合(A)から（B）にすることで 1 反復あたりの時間を 1024bit で約 65%，4096bit で約 90%削減できた．smt では，どの精度でも 10,000 秒以上かかっているのに対し(B)は 128bit の場合 4,541 秒で計算できた．

ⓒ2016 Information Processing Society of Japan

（B）の 1 反復あたりの時間は，ヒルベルト行列の場合1024bit から 4096bit に変化させると，（A）は約 8.1 倍に増加するのに対し（B）は約 2.5 倍となり，精度を向上に対する計算時間の増加比を大幅に削減できた．

また，ヒルベルト行列において 1024bit から 4096bit に精度を増やしても 5%しかメモリデータサイズは増加しない．

次に（C），（D）の結果から，今回用いた問題では，axpy などの関数の誤差も大きく影響しているためか，内積や疎行列ベクトル積を高精度にしても収束は改善できなかった．

ヒルベルト行列に対し，すべてのパターンでベクトルや変数の精度を上げて実験を行ったが，変数・ベクトルが 1 つでも低い精度だとその精度の影響が大きく，（C），（D）と比べて大きく異なる結果は得られなかった．

しかし，一部の変数の精度を上げても，メモリデータサイズや 1 反復あたりの時間はほとんど増加しないことから，GMP において変数のみの高精度化は，性能面では実用的だということがわかった．より多くの問題・アルゴリズムに対して検証し，混合精度反復解法の有効な適用先を調査していく必要がある．

6. まとめ

本論文では，コードを構文木として扱い，C コードから GMP コードを自動生成する機構 Xev-GMP によるディレクティブベースの混合精度 GMP コードの生成を提案した．

Xev-GMP は，3 種類のディレクティブを C コードに追記することで GMP コードを生成する．一般的なコンパイラではディレクティブが無視されるため，ユーザは C コードの管理のみで C コードと GMP コードの両方を利用できる．

算術演算の式から自動的に適切な精度の一時変数を用いた式を生成するため，複数の精度を用いた混合精度演算を行う GMP コードを生成できる．

Xev-GMP を用いて，CG 法のプログラムを対象に GMP コードの生成を行い，ヒルベルト行列と smt に対して適用することでその有用性を示した．

実験の結果から，データサイズが大きく，反復中に書き換えが行われない行列Aの精度を落とすことで，メモリデータ量を大幅に削減できた．（A），（B）のヒルベルト行列の結果から，精度を 1024bit から 4096bit に変化させたとき，行列Aの精度を落とした場合は約 2.5 倍しかかからず，約 2.5 倍の時間の増加で精度が 4 倍にできることから，混合精度プログラムの生成は有用であると考えられる．

また，内積や疎行列ベクトル積内部の変数の精度を上げても，今回の問題では収束改善効果が得られなかったが，メモリデータサイズや 1 反復あたりの時間は増加しない．変数のみの高精度化は実用的だと思われる．

内積や疎行列ベクトル積を高精度にしてもヒルベルト行列では効果はなく，axpy などの関数で大きく誤差が蓄積されていることがわかった．このように，Xev-GMP を用いれば数行の追記のみで精度依存性が簡単に確認できる．

また，一部の変数を高精度にしても計算時間が変わらない理由は，GMP の内部実装によるものと考えられる．

今後の課題として, GMP における精度と計算時間の関係を調査することで，ユーザがどのように精度を組み合わせれば良いかの指標を提示していく必要がある．

また, Xev-GMP を用いることでどのようなアルゴリズムや問題に混合精度手法が有効であるかを簡単なスクリプトで機械的に検証可能になったことから，より多くのケースで混合精度化の効果の検証を進めていきたい．

実装面での課題としては，OpenMP や MPI を用いた並列化コードの変換対応を行っていく必要がある．

ユーザへの提供の方法として，ユーザがコードをアップロードすることでオンラインコード生成を行う Web インタフェース Xev-GMP-Web [15]を公開している．

謝辞 本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」の支援により行った．また，本研究の一部は，JSPS 科学研究費 25330144 の助成を受けた．

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] T. Saito, et. al., Analysis of the GCR method with mixed precision arithmetic using QuPAT, Journal of Computational Science, Volume 3, Issue3, pp. 87-91 (2012).
[3] The GNU MP Arithmetic Library, https://gmplib.org/.
[4] Xevolver, http://xev.arch.is.tohoku.ac.jp/ja/software/#xevxml.
[5] H. Takizawa, S. Hirasawa, et.al., Xevolver: An XML-based Code Translation Framework for Supporting HPC Application Migration, HiPC2014, pp.1-11 (2014).
[6] 榊原巧磨, 佐々木信一, 菱沼利彰, 藤井昭宏, 田中輝雄, 平澤将一, GMP ライブラリを用いた任意多倍長プログラムへの自動変換機構の提案, 情報処理学会研究報告, vol.2015-HPC-152, No. 6, pp. 1-8 (2015).
[7] T. Hishinuma, T. Sakakibara, A. Fujii, T. Tanaka, and S. Hirasawa, Xev-GMP: Automatic Code Generation for GMP Multiple-Precision Code from C Code, 19th IEEE International Conference on Computational Science and Engineering (CSE 2016), pp.1-4 (2016).
[8] 丸地賢, 佐々木信一, 菱沼利彰, 藤井昭宏, 田中輝雄, 平澤将一, Xevolver を用いた GMP コードへの自動変換機能の実装, 第 77 回情報処理学会全国大会, pp.1-2 (2015).
[9] Basic Numerical Calculation PACKage, http://na-inet.jp/na/bnc/bnc.html.
[10] MPACK, http://mplapack.sourceforge.net/.
[11] The GNU MPFR Library, http://www.mpfr.org/.
[12] ROSE compiler infrastructure, http://rosecompiler.org/.
[13] Extensible Markup Language, https://www.w3.org/XML/.
[14] The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/TKK/smt.html
[15] Xev-GMP-Web, http://hpcl.info.kogakuin.ac.jp/lab/software/xev-gmp

ⓒ2016 Information Processing Society of Japan

```
1.  #pragma xev-gmp default(64)
2.  /* include・defineの省略 */
3.  typedef struct{
4.      double *val;
5.      int *ind;
6.      int *ptr;
7.      int N;
8.      int nnz;
9.  }CRS;
10. void spmv(CRS *matrix, double *x, double *y);
11. double dot(double *x, double *y, int n);
12. void axpy(double a, double *x, double *y, int n);
13. void xpay(double a, double *x, double *y, int n);
14. void matrix_input(CRS *matrix, char *filename);
15. int CGSolver(CRS *matrix, double *b, double *x, int *max_iter, double *eps);
16. int main(int argc, char **argv){
17.     /* CGSolver関数の入力データ読み込みなどの省略*/
18.     flag = CGSolver(matrix, b, x, &max_iter, &eps);
19.     /* CG法の結果出力・変数の解放などの省略*/
20. }
21. #pragma xev-gmp set(128) target(r,p,y,rrk,rrk1,alpha,beta)
22. int CGSolver(CRS *matrix, double *b, double *x, int *max_iter, double *eps){
23.     int k;
24.     int flag = FALSE;
25.     double *r, *p, *y;
26.     double rrk, rrk1, alpha, beta, err, tmp;
27.     r = (double *)malloc(sizeof(double) * matrix->N);
28.     p = (double *)malloc(sizeof(double) * matrix->N);
29.     y = (double *)malloc(sizeof(double) * matrix->N);
30.     spmv(matrix, x, r);
31.     tmp = -1;
32.     xpay(tmp, b, r, matrix->N);
33.     rrk = dot(r, r, matrix->N);
34.     for(k = 0; k < matrix->N; k++){
35.         p[k] = r[k];
36.     }
37.     for(k = 0; k < *max_iter*5; ++k){
38.         spmv(matrix, p, y);
39.         alpha = rrk / dot(p, y, matrix->N);
40.         axpy(alpha, p, x, matrix->N);
41.         axpy(-alpha, y, r, matrix->N);
42.         rrk1 = dot(r, r, matrix->N);
43.         err = sqrt(rrk1);
44.         printf("LOOP¥t%d¥tError¥t%le¥n", k, err);
45.         if(err < *eps){
46.             flag = TRUE;
47.             break;
48.         }
49.         beta = rrk1 / rrk;
50.         xpay(beta, r, p, matrix->N);
51.         rrk = rrk1;
52.     }
53.     *max_iter = k++;
54.     *eps = err;
55.     free(r);    r = NULL;
56.     free(p);    p = NULL;
57.     free(y);    y = NULL;
58.     return flag;
59. }
```

```
1.  #include<gmp.h>
2.  /* include・defineの省略 */
3.  typedef struct {
4.      mpf_t *val;
5.      int *ind;
6.      int *ptr;
7.      int N;
8.      int nnz;
9.  }CRS;
10. void spmv(CRS *matrix,mpf_t *x,mpf_t *y);
11. int dot(mpf_t tmp_rcv,mpf_t *x,mpf_t *y,int n);
12. void axpy(mpf_t a,mpf_t *x,mpf_t *y,int n);
13. void xpay(mpf_t a,mpf_t *x,mpf_t *y,int n);
14. void matrix_input(CRS *matrix,char *filename);
15. int CGSolver(CRS *matrix,mpf_t *b,mpf_t *x,int *max_iter,mpf_t *eps);
16. int main(int argc,char **argv){
17.     /* CGSolver関数の入力データ読み込みなどの省略*/
18.     flag = CGSolver(matrix,b,x,&max_iter,&eps);
19.     /* CG法の結果出力・変数の解放などの省略*/
20. }
21. int CGSolver(CRS *matrix,mpf_t *b,mpf_t *x,int *max_iter,mpf_t *eps){
22.   int i_gmp;
23.   mpf_t tmp_gmp_0_64;
24.   mpf_init2(tmp_gmp_0_64,64);
25.   int k;
26.   int flag = 0;
27.   mpf_t *r, *p, *y;
28.   /* mpf_t型の変数宣言と初期化の省略 */
29.   r = ((mpf_t *)(malloc(sizeof(mpf_t ) * (matrix -> N))));
30.   for (i_gmp = 0; i_gmp < (matrix -> N); i_gmp++) {
31.     mpf_init2(r[i_gmp],64);
32.   }
33.   /* メモリの動的確保の省略 */
34.   spmv(matrix,x,r);
35.   mpf_set_d(tmp_gmp_0_64,(- 1));
36.   mpf_set(tmp,tmp_gmp_0_64);
37.   xpay(tmp,b,r,matrix -> N);
38.   dot(tmp_gmp_0_64,r,r,matrix -> N);
39.   mpf_set(rrk,tmp_gmp_0_64);
40.   for (k = 0; k < matrix -> N; k++){
41.     mpf_set(p[k],r[k]);
42.   }
43.   for (k = 0; k <  *max_iter * 5; ++k){
44.     spmv(matrix,p,y);
45.     dot(tmp_gmp_0_128,p,y,matrix -> N);
46.     mpf_div(tmp_gmp_0_128,rrk,tmp_gmp_0_128);
47.     mpf_set(alpha,tmp_gmp_0_128);
48.     axpy(alpha,p,x,matrix -> N);
49.     mpf_neg(tmp_gmp_0_128,alpha);
50.     axpy(tmp_gmp_0_128,y,r,matrix -> N);
51.     dot(tmp_gmp_0_128,r,r,matrix -> N);
52.     mpf_set(rrk1,tmp_gmp_0_128);
53.     mpf_sqrt(tmp_gmp_0_128,rrk1);
54.     mpf_set(err,tmp_gmp_0_128);
55.     gmp_printf("LOOP¥t%d¥tError¥t%Fe¥n",k,err);
56.     if (mpf_cmp(err, *eps) < 0) {
57.       flag = 1;
58.       break;
59.     }
60.     mpf_div(tmp_gmp_0_128,rrk1,rrk);
61.     mpf_set(beta,tmp_gmp_0_128);
62.     xpay(beta,r,p,matrix -> N);
63.     mpf_set(rrk,rrk1);
64.   }
65.    *max_iter = k++;
66.   mpf_set( *eps,err);
67.   for (i_gmp = 0; i_gmp < (matrix -> N); i_gmp++) {
68.     mpf_clear(r[i_gmp]);
69.   }
70.   free(r);  r = ((void *)0);
71.   /* 動的確保した領域の解放の省略 */
72.   mpf_clear(tmp_gmp_0_64);
73.   /* 変数の解放の省略 */
74.   return flag;
75. }
```

図 11　CG 法のカーネル関数の定義と呼出
（左：C コード，右：GMP コード）

ⓒ2016 Information Processing Society of Japan