Vol.2015-HPC-152 No.6
2015/12/16

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

榊原巧磨[†1]　佐々木信一[†1]　菱沼利彰[†2]
藤井昭宏[†1]　田中輝雄[†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．高精度計算の実装手段の一つに任意多倍長ライブラリ GMP がある．GMP では算術演算の式を，演算子ではなく手続き呼び出しに変換する必要があり，プログラムの記述が容易ではない．既存の数値計算ライブラリを，GMP を用いて任意多倍長のライブラリにする試みが各所で行われており，正規表現等の単純置換が不可能なため手動で行われている．我々は，コードを構文木として扱い，構文木を解析，変換することで単純置換できない問題の解決を試みた．

本研究では，構文木としてコードを扱い，変換ルーチンを記述できる Xevolver フレームワークを用いて，C 言語の倍精度コードを，GMP を用いた任意多倍長コードへ自動変換する機構を提案する． 実際に変換対象として用意した CG 法のプログラムと，姫野ベンチマークの変換を行い，その有用性を示した．

1. はじめに

計算機環境の大規模化に伴い，大規模・悪条件な数値シミュレーションのニーズが高まっている．LAPACK チームによる調査ではチームの 16%のユーザが高精度計算を行っている[1]．特に今後のエクサスケールマシンでは，より大きな問題を扱うので，計算誤差が大きな問題になることが予想される．現在でもその問題の特性により精度が足りなくなることがある．

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，丸め誤差により収束が発散・停滞・増大する．収束の改善方法の１つに高精度演算がある．高精度演算を用いれば反復解法の収束を改善できる[2]．高精度演算の実装手法の一つに，GMP (GNU Multi-Precision) [3]という任意の精度で演算ができるライブラリを使用する方法がある．GMP では算術演算の式を, 演算子ではなく手続きの形式で行う必要があるほか, GMP 特有の型の変数を使用する必要があり，その変数の使用前，使用後にはそれぞれ初期化，解放関数に渡す必要がある．そのためプログラムの記述が容易ではない上，正規表現等を用いての単純置換が不可能である．既存の数値計算ライブラリを，GMP を用いて任意多倍長のライブラリにする試みが各所で行われている[4][5][6]．しかしながら，単純置換が不可能という問題があるため手動で変換されている．

我々は, この問題に対し, 構文木を用い, 構文木を解析, 変換することで単純置換できない問題を解決することを試みた．本研究では，コードを構文木として扱い，変換ルーチンを作成できる Xevolver[7][8]フレームワークを用いて，C 言語の倍精度プログラム(C コード)を，GMP を用いた任意多倍長プログラム(GMP コード)へ自動変換する機構を提案する．実装した変換機構を用い，変換対象として本研究で実装した変換機能を網羅したプログラムとして用意した共役勾配法 (CG 法)[9]のプログラムと，公開されている姫野ベンチマーク[10]の変換を行い，その有用性を示す．

以下 2 章で Xevolver をベースにした GMP コードへの変換，3 章で設計と実装，4 章で実験，5 章でまとめと今後の課題について説明する．

1.1 関連研究

GMP を利用したライブラリとして，幸谷らの BNCPACK[4]，中田らの MPACK[5]がある．MPACK では数値計算ライブラリである BLAS[11]・LAPACK[12]を，GMP を用いて任意の精度で計算できるようにしている．このように既存の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．これらのライブラリを用いて多倍長プログラムを作成する場合，使用するライブラリにあわせたコードを書く必要がある．

GMP の多倍長浮動小数点型(mpf_t 型)が倍精度型のように扱えるコンパイラとして omf77[6]コンパイラがある．し

```
1.  double x, y, z;
2.  x = 1.0;
3.  y = 2.0;
4.  z = 3.0;
5.  z += x * y;
6.  printf("%lf¥n", z);
```

```
1.  mpf_set_default_prec(128);
2.  mpf_t tmp1;
3.  mpf_init(tmp1);
4.  mpf_t x, y, z;
5.  mpf_init(x);
6.  mpf_set(x, 1.0);
7.  mpf_init(y);
8.  mpf_set(y, 2.0);
9.  mpf_init(z);
10. mpf_set(z, 3.0);
11. mpf_mul(tmp1, x, y);
12. mpf_add(z, z, tmp1);
13. gmp_printf("%Ff¥n", z);
14. mpf_clear(x);
15. mpf_clear(y);
16. mpf_clear(z);
17. mpf_clear(tmp1);
```

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

†1 工学院大学
Kogakuin University
†2 筑波大学
University of Tsukuba
†3 東北大学
Tohoku University

かし，ユーザはプログラム中の倍精度型を mpf_t 型に直さなくてはならない．そのため，C コードと GMP コードの両方を同時に管理する必要がある，

2. Xevolver による GMP コードへの変換

2.1 GMP

GMP(GNU Multi-Precision)は任意の精度で演算ができるライブラリである．GMP は，特有の書き方で表す必要がある．GMP では任意多倍長浮動小数点の型として，mpf_t 型というデータ型が用意されている．

図 1 は，同じ処理を C コードと GMP コードで書いた例である．C コードの 1 から 4 行目で変数の初期化を行っており，GMP コードの 1 から 10 行目に対応している．C コードの 5 行目で演算を行っており，GMP コードの 11，12 行目に対応している．C コード 6 行目で演算結果を表示しており，GMP コードの 13 行目と対応している．GMP コードの 14 行以降では mpf_t 型変数の解放関数を呼び出している．

mpf_t 型の変数は，使用する前に初期化関数に渡す必要がある．そのため，C コードで宣言と同時に初期値の代入が行われている場合，変数の宣言，初期化，初期値の代入を順に行うように書き換えなければならない．また，GMP の多倍長型の変数は使用後に解放関数に渡す必要がある．そのため， 1 対 1 対応の単純な置換ができないため，GMP でのプログラムは実装コストが高いという問題がある．

2.2 Xevolver

Xevolver は東北大学の滝沢らが開発した，ROSE[13]ベースのコード変換のためのフレームワークである．ROSE は C 言語のプログラムを解析，分解，再構築する事ができるフレームワークである．

Xevolver は ROSE により生成された構文木を XML (Extensible Markup Language)[14]で表された構文木に変換する. XML はタグという特定の文字列で地の文に意味を持たせるマークアップ言語のひとつである．XSLT (Extensible Stylesheet Language Transform)[15]は XML で記述された文書を変換する簡易言語である．

図 2 に Xevolver を用いて C コードを GMP コードに変換する際の処理フローを示す．

Xevolver は以下の 3 段階の処理により入力されたプログラムを変換する．

(1) 入力された C コードを，ROSE を用いて XML で表された構文木に変換
(2) XML であらわされた構文木を XSLT により記述された変換規則により変換
(3) 変換された XML を C コードに変換

Xevolver はコードを変換する際に XML を経由させ，XSLT で変換することでユーザが容易に独自のディレクティブを開発できるようにした．ユーザは XSLT を用いて構文木を変換し，元のプログラムを変換する．

2.3 設計思想

我々は Xevolver 上で，精度を指定するだけで GMP の構文を知らなくても容易に C コードを GMP コードに変換できるディレクティブを開発した．

C コードから GMP コードへの自動変更機構を作成するに当たり, 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プログラムが作成できることを目的とする．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) ユーザがコードの変更をする必要がない
(3) ディレクティブの記述は最小限

我々が開発した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)

(A)は C コードから GMP コードに変換するために必要なディレクティブである．ユーザは最低限このディレクティブを指定するだけでよい．このディレクティブは，引数に精度を指定して宣言することで，mpf_t 型の変数のデフォルトの精度を指定することができる．これは，mpf_t 型の変数のデフォルトの精度を指定する関数 mpf_set_default_prec(prec)を用いる．この関数は main 関数の先頭で 1 度呼び出すだけでよい．

(B)は一部の変数の精度を変更したい場合に使用する．引数に精度を指定することで，囲われた部分の変数の精度を設定することができる．ただし現在は一つのソースコードに 1 箇所のみしか記述できない．

ユーザはファイルの先頭にデフォルトの精度を指定するディレクティブを最低 1 つ追記することで, GMP の構文を知らなくても任意多倍長コードに変換することができる．このような自動変換機構を用いれば, 精度を指定するだけで, C コードに対応したあらゆる精度の GMP コードが得られるため，ユーザは C コードと GMP コードの両方を管理しなくてよい．

3. 設計と実装

ここでは，Xevolver を用いた GMP コードへの自動変換機構の具体的な設計と実装を説明する．今回我々が実装した機能を表 1 に示す．特徴的な変換機能として，3.1 で変数の宣言・初期化・解放，3.2 で入出力関数，3.3 で算術演

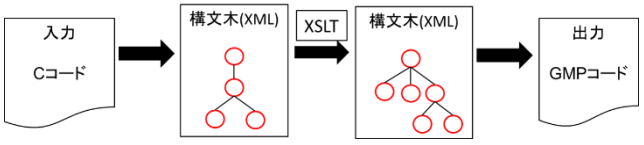

図 2 Xevolver による変換フロー

表 1 作成した変換機能

#	変換項目	備考
1	gmp.h のインクルード	
2	default 精度の指定	
3	特定範囲の精度の指定	1 箇所のみ
4	変数宣言・初期化・解放	
5	動的配列の生成	
6	グローバル変数	
7	構造体	入れ子不可
8	四則演算	
9	比較演算子	三項演算子不可
10	関数呼出	
11	標準入出力	
12	ファイル入出力	
13	数学関数	GMP が定義する関数のみ

表 2　変数の宣言，初期化，解放，算術演算の GMP の関数

#	関数名	動作
1	mpf_set_default_prec(prec)	mpf_t 型の変数のデフォルトの精度を指定
2	mpf_init(dst)	mpf_t 型の変数をデフォルトの精度で初期化
3	mpf_init2(dst, value)	mpf_t 型の変数を，精度を指定して初期化
3	mpf_set(dst, src)	mpf_t 型の変数の代入を行う
4	mpf_set_d(dst, src)	mpf_t 型の変数に double 型の値を代入
5	mpf_add(dst, src1, src2)	mpf_t 型の変数の和を計算
6	mpf_aub(dst, src1, src2)	mpf_t 型の変数の差を計算
7	mpf_mul(dst, src1, src2)	mpf_t 型の変数の積を計算
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

算について説明する．

表 2 に以降の章で使用する関数を示す．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．

3.1 変数宣言・初期化・解放

GMP コードでは多倍長浮動小数点型の変数は mpf_t という独自の型を用い，変数の使用前に初期化する関数 (mpf_init)，mpf_t 型の変数に倍精度の値を代入する関数 (mpf_set_d)，使用後には解放する関数 (mpf_clear)をそれぞれ呼ばなければならない．

C コードでは，変数の宣言時に初期化することができるが GMP コードではできない．したがって図 3 のように宣言の後に初期化関数を追記し，その後に初期値を代入する必要がある．

また，解放処理を変数の使用が終了する位置に挿入する必要もあるため，コード全体の探索が必要になり，単純な置換では変換できないという問題がある．

この問題を解決するため，以下の手順で変換を行った．

(1) 変数の宣言部分を GMP 独自形式の型に変換
(2) 変数の宣言部分の直後に変数の初期化関数を挿入
(3) 変数の宣言時に初期化が行われていた場合は初期化関数の直後に代入関数を挿入
(4) 変数の宣言されている関数の最後までたどり，return 文が存在する場合その直前に，ない場合は関数の末尾に変数の解放関数を挿入

図 4 は図 3 のコードの構文木である．図 4 にあるように C コードでは，変数宣言と初期化が同時に行われているが，変換後の GMP コードでは変数の宣言部分と，その直後に初期化関数，関数の最後に解放関数が追加される．

3.2 算術演算

ここでは，ひとつの式中に複数の演算子が存在する式の変換について説明する．GMP を用いた任意多倍長プログラ

```
1.  int main(){
2.      double a = 0.0;
3.      …
4.      return 0;
5.  }
```

```
1.  int main(){
2.      mpf_t a;
3.      mpf_init(a);
4.      mpf_set(a, 0.0);
5.      …
6.      mpf_clear(a);
7.      return 0;
8.  }
```

図 3 変数の宣言，初期化，解放
(左:C コード，右:GMP コード)

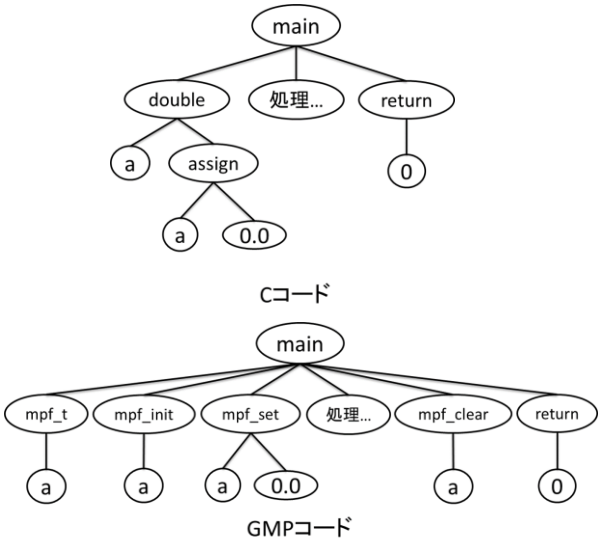

図 4　変数宣言の変換

ムでは，演算子を使用することができず，すべての演算を手続きにより行う必要がある．例えば，mpf_t 型へ mpf_t 型の代入を行う mpf_set，mpf_t 型の加算を行う mpf_add，mpf_t 型の乗算を行う mpf_mul がある．

ここで問題になるのは，GMP の関数は mpf_t 型の値を返せないという制約があることである．下記のように関数の戻り値を別の関数の引数に渡すような書き方ができず，一次変数を用いて関数の戻り値を受け取り，次の関数に渡すように変換する必要がある．

そのため以下のような C コードの場合，

```
a = b + c * d;
```

次のような変換は許されず，

```
mpf_set(mpf_add(b, mpf_mul(c, d)));
```

以下のように一時変数を用いて変換する必要がある．

```
mpf_mul(tmp, c, d);
mpf_add(a, b, tmp);
```

式の変換を行う際，我々は演算子に注目した．式をオペ演算ごとに分解し，それぞれの演算結果を，一時変数を介して次の演算に渡すことで，一般的な式への対応を試みた．例えば，

```
a = b + c * d;
```

という C コードを，演算ごとに分解し，

```
tmp3 = c * d;
tmp1 = b + tmp3;
a = tmp1;
```

のように一時変数を使用する．この分解された式を，

```
mpf_mul(tmp3, c, d);
mpf_add(tmp1,b,tmp3);
mpf_set(a,tmp1);
```

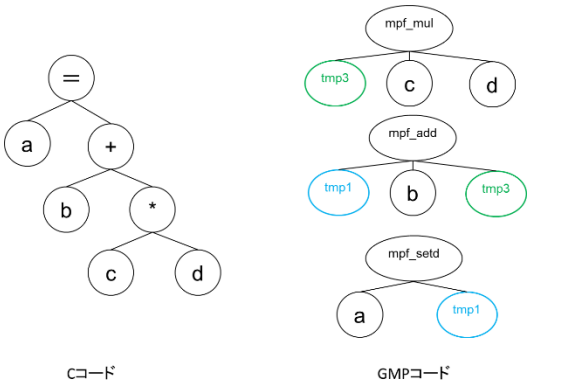

図 5　算術演算の式の変換

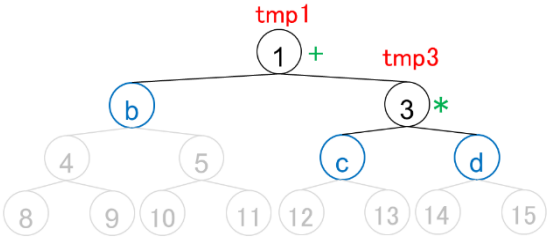

図 6　一時変数の ID に使用する番号の決定

という関数呼出の形に書き換えることで変換する．

式が構文木に直される際，図 5 のように先に演算されるオペレーションが木の末端になる．木の末端から変換すれば演算順序が変わってしまうことなく変換できる．

次に，使用する一時変数の名前が問題になる．式は構文木に変換された際，二分木の形になる．そこで，図 6 に示すように，式が理想的な二分木になったと仮定した時のノードの番号を変数の ID にすることで変数の名前の衝突を回避した．

最後に，変換時に使用した一時変数の宣言，初期化，解放処理を行う必要がある．これは，変換を 2 段階にして対応した．まず 1 段階目では必要な一時変数を使用した式の変換を行う．2 段階目で，関数内で使用されている一時変数に対し，関数の先頭で宣言と初期化，末尾で解放の重複がないように追加した．

以上のことから，算術演算は以下の手順で変換することができた．

(1) 末端まで構文木をたどる．
(2) 図 5 のように演算を関数呼び出しに再構築しながら階層を登っていき，その際に必要な一時変数を生成
(3) 生成した一時変数の宣言，初期化，解放処理を追加

今回の変換では，一時変数を作成する際に，二分木でのノードの位置を使用した．そのため，最小の個数と比べ，余分な変数が取られてしまい，式中での変数の再利用はできなかった．

しかし，変数の宣言，初期化を関数の先頭で，解放を関数の末尾で行うことで，式をまたいだ変数の再利用をすることができた．そのため，すべての式に対する一時変数を用意することは避けられた．

この変換規則での関数内で使用される一時変数の最大数は，その関数内で最も多く演算が含まれている式の演算子の個数になる．

3.3 関数呼出

GMP を用いた任意多倍長プログラムでは関数から GMP 独自の型の変数を返すことができないため，返却値を格納する引数を追加する必要がある．

関数の変換のパターンを，3 種類に分けて実装した．

(1) GMP で用意されている関数
(2) ユーザが定義した double 型の値を返す関数
(3) GMP 以外のライブラリで用意されている関数

以降 3.4.1 で，GMP で用意されている関数，3.4.2 でユーザが定義した double 型の値を返す関数，3.4.3 で GMP 以外のライブラリで用意されている関数を説明する．

3.3.1 GMP で用意されている関数

基本的な数学関数 (sin や cos など)は GMP が用意している．GMP で用意されている関数は，第一引数が結果を格納

するための変数として実装されている．

プログラム中で，GMP で用意されている関数が使用されている場合，関数名を変更し，戻り値を格納する一時変数を引数に渡すように変換した．これらは，我々が事前に用意した関数のテーブル (GMP 関数テーブル)を用いて，一致するかを判定し，一致している関数を置き換える．

関数が算術演算の式の一部で呼び出されている場合，式の演算子の一つが関数呼出に置き換わったと考え，そのため，算術演算の式の変換と同じ要領で変換できる．

例えば，

```
a = b + sqrt(c);
```

というＣコードの場合，

```
mpf_sqrt(tmp3, c);
mpf_add(tmp1, b, tmp3)
mpf_set(a, tmp1);
```

のように変換する．構文木としては，図 7 のように変換できる．現在，関数の引数に式が含まれている関数呼出(sqrt(a+b)など)の変換には非対応である．

3.3.2 ユーザが定義した double 型を返す関数

ユーザが定義した double 型を返す関数(ユーザ定義関数)は，以下の 2 つの条件で判定できる．

(1) その関数が GMP 関数テーブルに存在しない．
(2) ユーザのプログラムに定義が存在する．

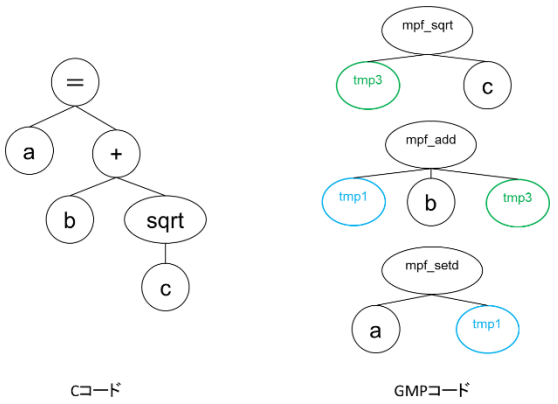

図 7　関数呼出を含んだ式の変換

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

```
1.  double add(mpf_t tmp_rcv,
    mpf_t a,   mpf_t b){
2.      mpf_t tmp1;
3.      mpf_init(tmp1);
4.      mpf_t c;
5.      mpf_init(c);
6.      mpf_add(tmp1, a, b);
7.      mpf_set(c, tmp1);
8.      mpf_set(tmp_rcv, c);
9.      mpf_clear(c);
10.     return 0.0;
11. }
```

図 8 ユーザ定義関数の変換 (左:C コード，右:GMP コード)

mpf_t 型の変数は関数から返すことができないため，

```
double function();
```

という関数を，単純に，

```
mpf_t function();
```

とすることはできない．

そこで，ユーザ定義関数は以下の 2 つの変換により実装した．

(1) 引数に戻り値を格納する mpf_t 型の変数の追加 (式中の一時変数と区別するため tmp_rcv とした)
(2) 関数内の return で返されている値を引数に追加した一時変数に代入する

図 8 のように，return 文の前では多倍長型の変数の解放関数が呼び出されるため，解放関数が呼ばれる前に戻り値を追加した引数に格納しなければならない．そこで，図 9 のような引数の追加と終了処理直前での代入が追加される．

3.3.3 GMP 以外のライブラリで用意されている関数

GMP 以外のライブラリで用意されている関数の場合，引数を書き換えることはできない．よって GMP 以外のライブラリで定義された関数が式中で使われた場合，その関数の戻り値を一時変数に格納してから演算に使用する必要がある．その関数が GMP 以外のライブラリで用意されているかは，以下の 2 つの条件で判定した．

(1) その関数が GMP 関数テーブルに一致する関数がない．
(2) ユーザのプログラムに定義が存在しない．

例えば，以下のような C コードの場合，

```
a = b + omp_get_wtime();
```

次のように変換する．

```
mpf_set_d(tmp2, omp_get_wtime());
mpf_add(tmp1, b, tmp2);
mpd_set(a, tmp1);
```

この際，一時変数は多項式と同じアプローチが使える．

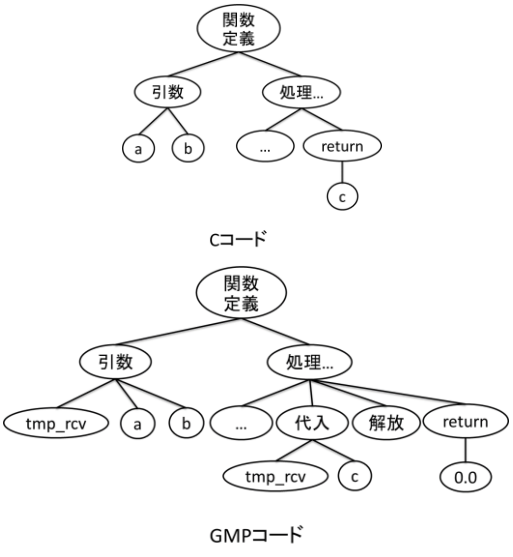

図 9　ユーザ定義関数の変換

4. 実験

今回，姫野ベンチマークと CG 法のプログラムの変換を行った．以下 4.1 で実験環境，4.2 で CG 法，4.3 で姫野ベンチマークの変換について述べる．

4.1 実験環境

使用した CPU は Intel® Xeon ® CPU E5-2630 @ 2.3-GHz 12Core，メモリは 64GB，OS は CentOS release 6.2，コンパイラは gcc-4.4.7，ROSE は rose-edg3-0.95a-20584，boost は boost-1.47，XSLT，XPath は xalan-C-1.11，XML パーサは xerces-C-3.1.1，Xevolver は XevXML commitID:[6354fb6] (20150618)である．

4.2 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．この変換ではノルム演算の部分に重要な変換が多く含まれているため，そこを中心に説明する．nrm2.c のプログラムを図 10 に示す．このプログラムには変数の宣言，初期化，解法処理，メモリの動的確保，算術演算，関数呼び出し，ユーザ定義関数の変換が含まれている．

関数内で，OpenMP を用い，並列で計算し，最後に足しあわせた結果の平方根を求めている．本実装では総和を求める際に，スレッドの数だけ足しこみ用の配列を用意して，各スレッドで部分和を求め，最後に逐次に足しこみを行っている．

```
1.  double nrm2(double *x, int n)
2.  {
3.      int i,is,ie,nproc,mypid;
4.      double *r;
5.      double ret;
6.      nproc = omp_get_max_threads();
7.      r = (double*)malloc(sizeof(double) * 256);
8.   #pragma omp parallel private (i,is,ie,mypid)
9.  {
10.     mypid = omp_get_thread_num();
11.     ie = n / nproc;
12.     is = mypid * ie;
13.     ie = ie +is;
14.     if(mypid==nproc-1){ ie = n; }
15.     r[mypid] = 0.0;
16.     for(i=is; i<ie; ++i)
17.     {
18.         r[mypid] += x[i] * x[i];
19.     }
20. }
21.     for(i=1; i<nproc; ++i)
22.     {
23.         r[0] += r[i];
24.     }
25.     ret = sqrt(r[0]);
26.     free(r);
27.     return ret;
28. }
```

```
1.  double nrm2(mpf_t tmp_rcv,mpf_t *x,int n)
2.  {
3.    mpf_t tmp_gmp1;
4.    mpf_init(tmp_gmp1);
5.    int i_gmp;
6.    int i;
7.    int is;
8.    int ie;
9.    int nproc;
10.   int mypid;
11.   mpf_t *r;
12.   mpf_t ret;
13.   mpf_init(ret);
14.   nproc = omp_get_max_threads();
15.   r = ((mpf_t *)
      (calloc(((size_t )(sizeof(mpf_t ))),((size_t )256))));
16.   for (i_gmp = 0; i_gmp < 256; i_gmp++)
17.   {
18.     mpf_init(r[i_gmp]);
19.   }
20. #pragma omp parallel private ( i, is, ie, mypid )
21. {
22.     mypid = omp_get_thread_num();
23.     ie = (n / nproc);
24.     is = (mypid * ie);
25.     ie = (ie + is);
26.     if (mypid == (nproc - 1)) {      ie = n;    }
27.     mpf_set_d(r[mypid],0.0);
28.     for (i = is; i < ie; ++i)
29.     {
30.       mpf_mul(tmp_gmp1,x[i],x[i]);
31.       mpf_add(r[mypid],r[mypid],tmp_gmp1);
32.     }
33. }
34.   for (i = 1; i < nproc; ++i)
35.   {
36.     mpf_add(r[0],r[0],r[i]);
37.   }
38.   mpf_sqrt(tmp_gmp1,r[0]);
39.   mpf_set(ret,tmp_gmp1);
40.   mpf_clear(r);
41.   mpf_set(tmp_rcv,ret);
42.   mpf_clear(ret);
43.   mpf_clear(tmp_gmp1);
44.   return 0.0;
45. }
```

図 10　nrm 関数の変換前後 (左:C コード，右:GMP コード)

図 10 の C コードの 1 行目の関数の宣言で，double 型を返す関数として宣言されている．GMP コードの 1 行目で第一引数に結果を格納する tmp_rcv が追加されていることがわかる．図 10 にはないが，nrm 関数の呼び出し部分も第一引数に結果を格納する一時変数を渡すように書き換わっている．

次に，C コードの 18 行目の式は，GMP コードの 30，31 行目で一時変数を使用した関数呼出に置き換わっている．ここで使用している一時変数は GMP コード 3，4 行目で宣言，初期化されており，43 行目で解放されている．

C コード 25 行目に GMP で用意されている関数を使用した式があり，GMP コードの 38，39 行目に，一時変数を使用し，GMP で定義されている関数に置き換わっている．使用している一時変数は先ほどの算術演算部分のものを再利用している．

C コードの 27 行目で，計算結果が返されているが，GMP コードでは 41 行目で引数に追加された結果を格納する変数に代入されている．

OpenMP では足し合わせや掛け合わせといったリダクション演算のために，プライベート化と演算を同時に行う reduction 節が用意されている．

しかし，このプログラム中で書かれている OpenMP のディレクティブでは，reduction 節を使用していない．これは OpenMP の reduction 節が mpf_t 型に対応していないため，変換は可能であるが，実行時エラーになってしまうからである．

OpenMP のディレクティブと我々の開発した Xevolver のディレクティブとの共存は可能であることがわかった．し

```
1.      for(n=0;n<nn;++n){
2.          gosa = 0.0;
3.          for(i=1 ; i<imax-1 ; ++i)
4.              for(j=1 ; j<jmax-1 ; ++j)
5.                  for(k=1 ; k<kmax-1 ; ++k){
6.                      s0 = a[i][j][k][0] * p[i+1][j  ][k  ]
7.                         + a[i][j][k][1] * p[i  ][j+1][k  ]
8.                         + a[i][j][k][2] * p[i  ][j  ][k+1]
9.                         + b[i][j][k][0] * ( p[i+1][j+1][k  ] - p[i+1][j-1][k  ]
10.                                - p[i-1][j+1][k  ] + p[i-1][j-1][k  ] )
11.                        + b[i][j][k][1] * ( p[i  ][j+1][k+1] - p[i  ][j-1][k+1]
12.                                - p[i  ][j+1][k-1] + p[i  ][j-1][k-1] )
13.                        + b[i][j][k][2] * ( p[i+1][j  ][k+1] - p[i-1][j  ][k+1]
14.                                - p[i+1][j  ][k-1] + p[i-1][j  ][k-1] )
15.                        + c[i][j][k][0] * p[i-1][j  ][k  ]
16.                        + c[i][j][k][1] * p[i  ][j-1][k  ]
17.                        + c[i][j][k][2] * p[i  ][j  ][k-1]
18.                        + wrk1[i][j][k];
19.                     ss = ( s0 * a[i][j][k][3] - p[i][j][k] ) * bnd[i][j][k];
20.                     gosa = gosa + ss*ss;
21.                     wrk2[i][j][k] = p[i][j][k] + omega * ss;
22.                 }
23.         for(i=1 ; i<imax-1 ; ++i)
24.             for(j=1 ; j<jmax-1 ; ++j)
25.                 for(k=1 ; k<kmax-1 ; ++k)
26.                     p[i][j][k] = wrk2[i][j][k];
27.     }
```

```
1.      for (n = 0; n < nn; ++n)
2.      {
3.        mpf_set_d(gosa,0.0);
4.        for (i = 1; i < (imax - 1); ++i)
5.          for (j = 1; j < (jmax - 1); ++j)
6.            for (k = 1; k < (kmax - 1); ++k)
7.            {
8.              mpf_mul(tmp_gmp512,a[i][j][k][0],p[i + 1][j][k]);
9.              mpf_mul(tmp_gmp513,a[i][j][k][1],p[i][j + 1][k]);
10.             mpf_add(tmp_gmp256,tmp_gmp512,tmp_gmp513);
11.             mpf_mul(tmp_gmp257,a[i][j][k][2],p[i][j][k + 1]);
12.             mpf_add(tmp_gmp128,tmp_gmp256,tmp_gmp257);
13.             mpf_sub(tmp_gmp1036,p[i + 1][j + 1][k],p[i + 1][j - 1][k]);
14.             mpf_sub(tmp_gmp518,tmp_gmp1036,p[i - 1][j + 1][k]);
15.             mpf_add(tmp_gmp259,tmp_gmp518,p[i - 1][j - 1][k]);
16.             mpf_mul(tmp_gmp129,b[i][j][k][0],tmp_gmp259);
17.             mpf_add(tmp_gmp64,tmp_gmp128,tmp_gmp129);
18.             mpf_sub(tmp_gmp524,p[i][j + 1][k + 1],p[i][j - 1][k + 1]);
19.             mpf_sub(tmp_gmp262,tmp_gmp524,p[i][j + 1][k - 1]);
20.             mpf_add(tmp_gmp131,tmp_gmp262,p[i][j - 1][k - 1]);
21.             mpf_mul(tmp_gmp65,b[i][j][k][1],tmp_gmp131);
22.             mpf_add(tmp_gmp32,tmp_gmp64,tmp_gmp65);
23.             mpf_sub(tmp_gmp268,p[i + 1][j][k + 1],p[i - 1][j][k + 1]);
24.             mpf_sub(tmp_gmp134,tmp_gmp268,p[i + 1][j][k - 1]);
25.             mpf_add(tmp_gmp67,tmp_gmp134,p[i - 1][j][k - 1]);
26.             mpf_mul(tmp_gmp33,b[i][j][k][2],tmp_gmp67);
27.             mpf_add(tmp_gmp16,tmp_gmp32,tmp_gmp33);
28.             mpf_mul(tmp_gmp17,c[i][j][k][0],p[i - 1][j][k]);
29.             mpf_add(tmp_gmp8,tmp_gmp16,tmp_gmp17);
30.             mpf_mul(tmp_gmp9,c[i][j][k][1],p[i][j - 1][k]);
31.             mpf_add(tmp_gmp4,tmp_gmp8,tmp_gmp9);
32.             mpf_mul(tmp_gmp5,c[i][j][k][2],p[i][j][k - 1]);
33.             mpf_add(tmp_gmp2,tmp_gmp4,tmp_gmp5);
34.             mpf_add(tmp_gmp1,tmp_gmp2,wrk1[i][j][k]);
35.             mpf_set(s0,tmp_gmp1);
36.             mpf_mul(tmp_gmp4,s0,a[i][j][k][3]);
37.             mpf_sub(tmp_gmp2,tmp_gmp4,p[i][j][k]);
38.             mpf_mul(tmp_gmp1,tmp_gmp2,bnd[i][j][k]);
39.             mpf_set(ss,tmp_gmp1);
40.             mpf_mul(tmp_gmp3,ss,ss);
41.             mpf_add(tmp_gmp1,gosa,tmp_gmp3);
42.             mpf_set(gosa,tmp_gmp1);
43.             mpf_mul(tmp_gmp3,omega,ss);
44.             mpf_add(tmp_gmp1,p[i][j][k],tmp_gmp3);
45.             mpf_set(wrk2[i][j][k],tmp_gmp1);
46.           }
47.       for (i = 1; i < (imax - 1); ++i)
48.         for (j = 1; j < (jmax - 1); ++j)
49.           for (k = 1; k < (kmax - 1); ++k)
50.             mpf_set(p[i][j][k],wrk2[i][j][k]);
51.     }
```

図 11　姫野ベンチマークのーネル部分の変換前後 (左:C コード，右:GMP コード)

かし，GMP は演算を手続きで行わなければならないため，reduction 節のような処理には対応できていない．

4.3 姫野ベンチマーク

姫野ベンチマークは Poisson 方程式の解法をヤコビ法で解く場合の処理速度を計るベンチマークプログラムである．ダウンロードしたプログラムでは，浮動小数点は単精度で宣言されていたが，倍精度の変換のみの対応なので，単精度の部分を倍精度に置換した．

図 11 のコードはヤコビ法のカーネル部分を抜粋した. 図 11 の C コードの 6 から 18 行目が一つの式であるが，GMP コードでは 8 から 35 行目になっている. このような複雑な式でも矛盾なく変換できることが確認できた．

同じように C コードの 19 行目，20，21 行目の式はそれぞれ GMP コード 36 から 39 行目, 40 から 45 行目に置き換えられ，使用している一時変数は再利用されている．

変換コードを実行したところ，正しく動作しており，姫野ベンチマークでも正しく変換できることを確認した．

ただし，複数の式で一時変数を再利用する変換を行ったため，関数中で宣言される一時変数の個数は，その関数中に存在する最も演算子を多く使用した式の演算子の個数分にとどめた．

5. おわりに

本研究では, GMP を用いた任意多倍長プログラムの自動変換機構として変数宣言，入出力関数，算術演算，関数呼び出しの変換を作成し，CG 法，姫野ベンチマークのプログラムの変換を行い，その有用性を示した．現在，変換のすべてを XSLT で記述しているため，今後 GMP や変換対象のプログラムにおいて拡張が行われた際にも，ユーザが容易に変換規則を拡張も行うことができる．

C コードから GMP コードへの変換はほぼ一定のルールで変換できるため，変換時にユーザからは精度を指定してもらうだけでよいことがわかった．

今回の変換では，式の変換の部分で，式をまたいでの一時変数の再利用はできたが，式中での再利用はできなかったため，余分な一時変数が宣言された．また，式は演算子ごとでの変換のため，変数の代入の処理が余分に行われてしまっている．そのため，式中での一時変数の再利用と，余分な処理の削減は今後の課題である．

式の変換が再帰的に行うことで可能であることがわかったため，現在対応していない構造体の入れ子なども，再帰的に処理を行うことで変換できると考えている．

今回の変換では 1 箇所のみであるが，部分精度指定も変換できたほか，reduction 節を使用しない OpenMP による並列化への対応ができた．今後，複数個所の精度指定，OpenMP の reduction についても実装を行う．

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] Demmel,J.，http://www.cs.berkeley.edu/~demmel/Future_Sca-LAPACK_v7.ppt, (2005).
[2] Hasegawa, H., "Utilizing the Quadruple-Precision Floating-Point Arithmetic Operation for the Krylov Subspace Methods", The 8th SIAM Conference on Applied Linear Algebra (2003).
[3] The GNU MP Bignum Library，https://gmplib.org/.
[4] Basic Numerical Calculation PACKage, http://na-inet.jp/na/bnc/bnc.html.
[5] MPACK, http://mplapack.sourceforge.net/.
[6] 平野基孝 他, "GNU MP を用いた Fortran コンパイラ omf77 の多倍精度浮動小数拡張", 情報処理学会研究報告, Vol.2001-HPC-085, No.22, pp127 (2001).
[7] Xevolver, http://xev.arch.is.tohoku.ac.jp/ja/software /#xevxml.
[8] Hiroyuki Takizawa, Shoichi Hirasawa, Yasuharu Hayashi, Ryusuke Egawa, Hiroaki Kobayashi, “Xevolver: An XML-based Code Translation Framework for Supporting HPC Application Migration”, IEEE International Conference on High Performance Computing, (2014).
[9] HESTENES, M. R., and STIEFEL, E., "Methods of conjugate gradients for solving linear systems", Res. N.B.S., Vol.49, pp409 (1952).
[10] 姫野ベンチマーク, http://accc.riken.jp/supercom/himenobmt/.
[11] Lawson,C. L., "Basic Linear Algebra Subprograms for Fortran Usage", ACM Transactions on Mathematical Software, pp308 (1979).
[12] Anderson, E., "LAPACK Users' Guide Third Edition", SIAM, Philadelphia, (1999).
[13] ROSE compiler infrastructure，http://rosecompiler.org/.
[14] Extensible Markup Language, http://www.w3.org/XML/.
[15] XSL Transformations, http://www.w3.org/TR/xslt/.