計算機アーキテクチャ 142−22
ハイパフォーマンス
コンピューティング 85−22
（2001. 3. 9）

GNU MP を用いた Fortran コンパイラ *omf77* の多倍精度浮動小数拡張

平野基孝[†] 佐藤三久[††] 建部修見[†††]

Omni OpenMP コンパイラシステムの Fortran 77 コンパイラである *omf77* に、GNU MP (GNU Multiple Precision library) を用いて多倍精度浮動小数拡張を施した。本論文ではその実装方法を解説すると共に、シンプソン法による数値積分の誤差の評価、C 言語で記述された GNU MP C 言語インターフェースを用いたプログラムとの速度比較等を行った結果を考察した。その結果、*omf77* による多倍精度浮動小数演算は、GNU MP C 言語インターフェースを用いた C プログラムより高速に動作し得り、また容易なプログラム作成を可能にするとの結論を得た。

Multiple precision floating point number extension for Fortran compiler *omf77* with GNU MP

MOTONORI HIRANO,[†] MITSUHISA SATO[††] and OSAMU TATEBE [†††]

We introduced a multiple precision floating point number extension for *omf77*, a Fortran 77 compiler included in Omni OpenMP compiler system. In this paper, we present an implementation of the extension, an error evaluation in numerical integration by Simpson method with the extension, and a performance evaluation between a program written in C language with GNU MP C language interface. We found that with *omf77*, the Fortran 77 program using multiple precision floating number type could run faster than the C program with GNU MP C language interface, and it is also effective for ease of programming.

1. はじめに

近年のハードウェアの処理能力の増大により、数値計算が扱う分野も拡大している。これに伴い、処理速度だけでなく、より高精度での数値演算に対する要求も高まっている。

Fortran 77 での実数型はこれまで `double precision` 型が最高精度であったが、商用 Fortran 77 コンパイラの多くが、前述の要求にこたえるために、`real*16`, `real*32` 型などの高精度実数型をサポートしはじめている。また、その他の手法として、ソフトウェアによる多倍精度演算も考えられ、これまでに FMLIB[1], bmp[2] 等の Fortran 用多倍精度演算ライブラリが多く開発されている。

多倍精度演算ライブラリを用いてプログラムを記述する場合、単純な加減乗除を行う際にも副プログラム呼び出しを介すことになり、記述が繁雑になる。

† 株式会社 SRA
Software Research Associates, Inc.
†† 技術研究組合 新情報処理開発機構
Real World Computing Partnership (RWCP)
††† 電子技術総合研究所
Electorotechnical Laboratory (ETL)

高精度実数型を用いる場合、それらはコンパイラ処理系基本型であるので、通常の演算子を用いて加減乗除を記述できることから、多倍精度演算ライブラリを用いる場合より記述は簡単になる。が、さらに高精度の演算が必要になった場合には対応出来ない。

そこで我々は、両者の欠点を補うべく、Omni OpenMP コンパイラシステムの Fortran 77 コンパイラである *omf77* に、多倍精度演算エンジンとして GNU MP[3] を用い、処理系基本型として多倍精度浮動小数型を実装した。処理系基本型として多倍精度浮動小数型をサポートすることで、多倍精度演算ライブラリを用いた場合のように副プログラム呼び出しを多用すること無く演算を記述でき、なおかつユーザが望む任意の精度での演算が可能になる。

1.1 本論文の構成

本論文ではまず **2** 節で Omni OpenMP コンパイラシステム、GNU MP の概要について触れ、次に **3** 節で *omf77* での多倍精度浮動小数型の実装について解説する。次に **4** 節で *omf77* の使用方法に触れた後、**5** 節で実際に多倍精度浮動小数型を用いた Fortran 77 プログラムの例を挙げて、その実行結果について考察する。最後に **6** 節で、結論及び今後の課題について述べる。

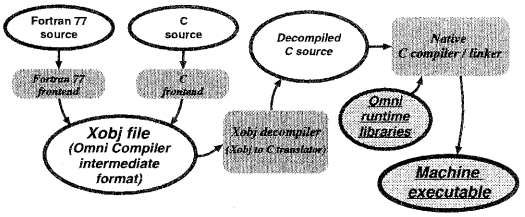

図 1: Omni OpenMP コンパイラシステムの動作概要

2. 背　　景

2.1 Omni OpenMP コンパイラシステム

omf77 は RWCP つくば研究センタにおいて開発された Omni OpenMP コンパイラシステムの Fortran 77 コンパイラである。Omni OpenMP コンパイラシステムの動作概要を図 1 に示す。

Omni OpenMP コンパイラシステムは、C, Fortran 77 のソースコードから native C コンパイラ用のソースコードを生成する C トランスレータとして動作する。入力となる C, Fortran 77 ソースコードに OpenMP 指示文が挿入されていた場合、指定された並列プログラミングインターフェース (pthread, solaris thread 等) を呼び出すよう実装された Omni ランタイムライブラリを用いる native C コンパイラ用 C ソースコードを生成することで、並列実行可能な実行形式を生成する OpenMP コンパイラとして動作する。

2.2 GNU MP 多倍精度演算ライブラリ

GNU MP は開発開始から 10 年近い歴史を持つ、広く普及した多倍精度演算ライブラリである。

GNU MP は C 言語で使用するためのライブラリであり、C 言語で多倍精度浮動小数を表現する型として、`mpf_t` 型を提供している。プログラマは `mpf_t` 型を使用し、それらを初期化、演算、出力するための `mpf_*()` の関数 (以下 GNU MP 関数群) を呼び出す事で多倍精度浮動小数演算を行うことになる。

以下に `mpf_t` 型の定義と、GNU MP 関数群を用いた典型的な C 言語プログラム例を示す。

```
typedef struct {
    int _mp_prec;
    int _mp_size;
    mp_exp_t _mp_exp;
    mp_limb_t *_mp_d;
} __mpf_struct;
typedef __mpf_struct mpf_t[1];
      :
int
main()
{
    mpf_t x, y, z;

    mpf_set_default_prec(128);

    mpf_init_set_d(x, 1.0);
    mpf_init_set_d(y, 2.0);
    mpf_init(z);

    mpf_mul(z, x, y);

    mpf_out_str(stdout, 10, 0, z);

    mpf_clear(x);
    mpf_clear(y);
    mpf_clear(z);

    return 0;
}
```

`mpf_t`(`__mpf_struct` 構造体) 型内の `mp_exp_t` と `mp_limb_t` は、64 bit アーキテクチャであれば 64 bit 整数型、32 bit アーキテクチャであれば 32 bit 整数型である。実際に仮数部として使用されるのは `_mp_d` メンバである。

`mpf_set_default_prec()` の引数は精度を bit 数で表したもので、`mpf_t` の精度がこれで決定される。

`mpf_init_set_d()`, `mpf_init()` の呼び出し時に、`_mp_d` の実体が動的に (`malloc(3)` により) heap 領域に確保される。確保される大きさ lSz(`mp_limb_t` の数) は、`mpf_set_default_prec(`p`)` の引数で指定された精度 p から、bL を `sizeof(mp_limb_t)`×8 として

$$lSz = int(\frac{max(53,p) + 2\times bL - 1}{bL}) + 1 \quad (1)$$

で算出される。

以降、GNU MP 関数群を用いて計算を行い、最終的に `mpf_t` 型が必要無くなった時点で `mpf_clear()` を呼び出し、確保した `_mp_d` を解放する。

3. *omf77* における多倍精度浮動小数拡張の設計と実装

3.1 コード生成における問題点

GNU MP を用いて多倍精度浮動小数型を実装する場合、**2.2 項** で示した C 言語でのプログラムスタイルを踏襲したコード生成を行うような実装にすると、以下のような問題点が考えられた。

(1) 多倍精度浮動小数オブジェクトとしての `mpf_t` 型 (特に `_mp_d`) の生存期間を *omf77* フロントエンドで手続き間に渡って詳細に解析しなければ、いつ `mpf_clear()` を呼んで良いか判らない。

(2) `mpf_clear()` をまとめて呼べるように GC を導入すると、実行速度の低下を招く。

(3) 仮数部が大きい場合には、`_mp_d` の動的確保、解放のための `malloc(3)`, `free(3)` の呼び出しコストも実行速度の面から無視できない。

3.2 `_omQReal_t` 型

前項で挙げた問題点は、全て `_mp_d` が動的に heap 領域に確保されている事に起因する。これを解決するには、`mpf_t` が持っているメンバ全ての他に、仮数部を表す領域をメンバに持った構造体を用いて多倍精度浮動小数を表現すればよい。しかし、`mpf_t` と別の多倍精度浮動小数表現型を用いる事で、GNU MP 関数群を使用する際に `mpf_t` への変換・再変換が必要となり、この型変換オーバーヘッドが大きいと、別表現にする利点が相殺されてしまう。

そこで *omf77* では、以下に示す `_omQReal_t` 型を用いて多倍精度浮動小数を表現することにした。

```
typedef struct {
    int _mp_prec;
    int _mp_size;
    mp_exp_t _mp_exp;
    mp_limb_t *_mp_d;
    mp_limb_t _omLimb[LIMB_SIZE];
} _omQReal_t;
```

`_omQReal_t` の先頭から `_mp_d` までのメンバ構成は `mpf_t(__mpf_struct)` と完全に同一である。違いは `_mp_d` の実体となる `_omLimb[LIMB_SIZE]` メンバを持っていることである。`LIMB_SIZE` の算定は、`mpf_t` 同様式 1 で、Fortran 77 ソースコードのコンパイル時に、*omf77* のコマンドラインオプション `-m`, `-md` で指定された精度を使用して行い、それ以外の時点での精度指定 (すなわち `LIMB_SIZE` の算定および変更) は行わないものとした。したがって、本来 GNU MP で可能であるはずの、プログラム実行時、もしくはプログラム内部での動的な精度変更は *omf77* では出来ない。

型変換オーバーヘッドの問題への対処は、`_mp_d` までのメンバ構成を `mpf_t` と同一にすることで、GNU MP 関数群の呼び出し前のどこかで、

```
xPtr->_mp_d = &(xPtr->_omLimb[0]);
```

のような初期化を行えば、`_omQReal_t` のポインタを `__mpf_struct` のポインタに cast することで GNU MP 関数群を直接使用できるので、型変換オーバーヘッドも最小であろう。

omf77 が native C コンパイラ用に出力する C 言語ソースコード上では、多倍精度浮動小数は全て `_omQReal_t` で表現されている。実際の演算は、`_omQReal_t` のポインタを引数に取る *omf77* ランタイム関数群の呼び出しで行われる。`_omQReal_t` のポインタを引数に取る *omf77* ランタイム関数群は、前述の、`xPtr->_mp_d = &(xPtr->_omLimb[0])` の初期化を最初に行った後、`_omQReal_t` のポインタを `__mpf_struct` のポインタに cast して、然るべき GNU MP 関数群を呼び出すよう実装されている。

この cast による呼び出しを行うためには、以下の条件が満たされていなければならない。

(1) `mpf_init()` 以外に `_mp_d` を確保する関数は無い。

(2) `mpf_clear()` 以外に `_mp_d` を解放する関数は無い。

(3) 実行途中で動的に精度を変えない限り、一度 `mpf_init()` で確保された領域が、再確保されることは無い。

(1), (2) は GNU MP のソースコードを調査して確認した。(3) に関しては、前述のコンパイル時にしか精度を指定できないという仕様により満たされている。

3.2.1 定数表現

omf77 が native C コンパイラ用に出力する C ソースコード上では、多倍精度浮動小数定数は、初期化子付きの `_omQReal_t` 型変数として表現されている。Fortran 77 ソースコード上では文字列として表現されているので、*omf77* フロントエンドが、数値文字列を GNU MP 関数群を用いて `_omQReal_t` 型に変換する際に、極僅かだが誤差が発生することがある。これが問題となる場合、*omf77* のコマンドラインオプション `-disableQCZFolding` を付加することで、DATA 文、PARAMETER 文以外において *omf77* フロントエンドでの多倍精度浮動小数の constant folding を全く行わず、多倍精度浮動小数演算は全て実行時に行うような C 言語ソースコードを生成するようになる。この場合、多倍精度浮動小数定数は文字列として C 言語ソースコードに出力され、実行時に逐一 `_omQReal_t` 型に変換されることになり、実行速度が著しく低下することもある。

3.2.2 多倍精度浮動小数の書式無し入出力

omf77 の入出力は Netlib で公開されている `libf2c` を元にした *omf77* ランタイムライブラリで実現されている。`_omQReal_t` の書式無し入出力は、`_omQReal_t` 構造体がほぼそのままダンプされる形で実装されているので、*omf77* 以外の Fortran 77 コンパイラとの互換は全く無い。また、**3.2 項** で述べたように、`LIMB_SIZE` がコンパイル単位で可変であるから、`sizeof(_omQReal_t)` の大きさもコンパイル単位で可変となる。これが変るとダンプされる大きさも変るので、*omf77* 同士においても互換が無い場合がほとんどとなる。

3.2.3 DATA 文による多倍精度浮動小数型変数・配列の文字列での初期化

多倍精度浮動小数型変数、配列が DATA 文により文字列で初期化される場合、その他の数値型と異なり、`sizeof(_omQReal_t)` が可変であるため、以下のような処理がなされている。

(1) 多倍精度浮動小数型配列全体が文字列で初期化されている場合、配列全体を

`sizeof(_omQReal_t)` * 配列サイズの char の配列として、配列の先頭アドレスから全初期化文字列を繋げて copy する。終端は NUL terminate される。

(2) 多倍精度浮動小数型配列の一部が文字列で初期化されている場合、当該要素を `sizeof(_omQReal_t)` の大きさの char の配列として、初期化文字列を当該要素の先頭アドレスから copy する。余った部分は 0x20 で初期化される。

(3) 多倍精度浮動小数型要素変数だった場合、(2) に準ずる。

このように、プログラマが意図した文字列イメージと異なる結果を生じる場合もあるので注意が必要である。

4. 使用方法

4.1 Omni の設定

omf77 の多倍精度浮動小数型サポートは、Omni コンパイラシステムのインストール時に、`configure` スクリプトに `--enable-mreal` フラグを付加する事で可能になる。この時、GNU MP が `/usr/local` にインストールされていない環境では、`--with-gmpLibDir=`*libDIR*,`--with-gmpIncDir=`*incDIR* のフラグで、*libDIR* に `libgmp.a` (もしくは `libgmp.so*`) のあるディレクトリ、*incDIR* に `gmp.h` のあるディレクトリを指定する必要がある。

なお、使用する GNU MP は、最新版の GNU MP 3.1.1 であることが必須である☆。

4.2 Fortran 77 ソースコード

4.2.1 変数および配列

Fortran 77 ソースコード内で多倍精度浮動小数型を使用する場合、`multiple precision` 型を用いる。また、`real*16` 型も多倍精度浮動小数型となり、`multiple precision` 型と全く同一の精度を持つ。

```
multiple precision x(10)
real*16 y
```

上記 x, y は両者とも、同一の精度を持つ多倍精度浮動小数型となる。`real*16` を多倍精度浮動小数型にしたのは、`real*16` をサポートしたその他の Fortran 77 コンパイラ用に書かれたソースコードを無変更のまま、より高精度で動作させるのに都合が良いからである。

これら型宣言の後、プログラマは通常の数値型と全く同様に、加減乗除の演算を多倍精度型の変数、配列に関して行えば良い。その他の数値型と混合して演算を行う場合、普通の Fortran 77 プログラム同様 (例えば単精度と倍精度を混合して演算するような場合と同様)、*omf77* は適切にそれら相互の型変換を行う。これら暗黙の型変換に伴うオーバーフローの発生等は、プログラマの責任となる。

4.2.2 定　　数

多倍精度の定数を表現するため、プログラマは倍精度実定数指定 `1.0D10` と同様に、多倍精度実定数指定として、`1.0Q10` のように "Q" を用いる事ができる。"Q" (Quad) を用いたのは、**4.2.1 項** で述べた理由同様、`real*16` をサポートした Fortran 77 コンパイラとの互換のためである。

4.3 コンパイラ

3.2 項 で述べたように、多倍精度浮動小数型の精度指定は、Fortran 77 ソースコードのコンパイル時に、*omf77* のコマンドラインオプション `-m`*#bit* もしくは `-md`*#dig* の *#bit*, *#dig* で行う。それ以外の方法はない。

`-m`*#bit* オプションは、プログラム中の多倍精度浮動小数型変数、配列、定数を、*#bit* ビットの仮数部を持つ多倍精度浮動小数とする。`-md`*#dig* オプションは、それらを、10 進数で *#dig* の有効桁をサポートする仮数部を持つ多倍精度浮動小数とする。

これら両オプションの指定がない場合 128 bit が使用される。

また、これら `-m`, `-md` オプションで指定した精度は、同一実行形式にリンクされるオブジェクト群について同一でなければならない。理由は `_omQReal_t` 型の大きさが異なるオブジェクト同士での `_omQReal_t` 型の交換が出来ないためである。

4.4 intrinsic 関数

4.4.1 precof intrinsic 関数

多倍精度浮動小数を整形出力する時には、実際にそれらが 10 進数で何桁になるかという情報が必要であろう。が、*omf77* では、実際に何桁になるかはコンパイルオプションで決まるので、プログラムソース内でそれを取得するための仕掛けが必要となる。

このような場合のため、*omf77* は `precof` intrinsic 関数を用意している。`precof` は、引数の型の指数部の大きさを bit 単位で返す。以下に使用例を示す。

```
multiple precision x
integer dig
dig = anint(float(precof(x))
@             * log10(2.0)) + 1
```

4.4.2 mpeps intrinsic 関数

数値計算において、machine epsilon の値は大変良く参照される。倍精度型においては、ほとんどの環境で machine epsilon は 2^{-53} となるが、多倍精度浮動小数型においては、(指数部の大きさを自分自身で計算するなどして) そのプログラム自身が算出する以外にプログラム中で使用可能とならない。指数部が大きい場合、machine epsilon の計算にかなりの実行時間

☆ Linux で使用する場合、多くの Linux ディストリビューションでは version 2 ベースの GNU MP がインストールされているのだが、それを使う事はできないので注意が必要である。

が費される事となる。

omf77 ではこのように多倍精度浮動小数型の machine epsilon の計算に実行時間を費さなくても済むよう、コンパイル時に多倍精度浮動小数型の machine epsilon を算定し、それをプログラムから参照できるよう、mpeps intrinsic 関数を用意している。mpeps は、引数の型での machine epsilon を引数の型と同じ型で返す。以下に使用例を示す。

```
multiple precision eps
eps = mpeps(eps)
```

5. 多倍精度浮動小数型を用いたプログラム

5.1 シンプソン法による数値積分の誤差評価

異なった精度で数値計算の誤差の振舞いにどのような違いがあるかを評価する例として、$\frac{4}{1+x^2}$ の区間 [0,1] の積分値を、シンプソン公式区間 2 分法で求め、$\int_0^1 \frac{4}{1+x^2}dx$ の真値 π との差 (誤差) の絶対値を出力する Fortran 77 プログラムを作成した。このプログラムを *omf77* により倍精度、4 倍精度、8 倍精度でそれぞれコンパイルし、分割数 N を 2 ～ 2^{28} まで実行した結果を 図 **2** に示す。測定は Compaq ES40 (Alpha ev67/Linux) にて行った。同じ Fortran 77 ソースコードで倍精度型と多倍精度型をコンパイルオプションで切替えるようにするためには、

```
#ifdef MREAL
#define PREC 16
#else
#define PREC 8
#endif
      real* PREC x, y
```

のような記述を行った上で、ファイルサフィックスを .F として、*omf77* の C プリプロセッサ機能を用いた。4 倍精度と 8 倍精度の切替えは、*omf77* の -m オプションと、-DMREAL を付加することで行った。

また、*omf77* での 4 倍精度、8 倍精度は、仮数部のみでそれぞれ 128 bit, 256 bit を占めるので、その他の Fortran 77 コンパイラでの 4 倍精度、8 倍精度より仮数部が大きい事 (かつ machine epsilon が小さい事) に注意されたい。

シンプソン公式を用いて数値積分する場合、打ち切り誤差の主要項は h を刻み幅 ($h = \frac{1}{N}$) とすると $O(h^4)$ であるが、$\int_0^1 \frac{4}{1+x^2}dx$ の場合には、この項の係数が 0 となり、打ち切り誤差の主要項は $O(h^6)$ となる。図 **2** 中の一点鎖線は、この打ち切り誤差の主要項 $\frac{10h^6}{16128}$ を示している。

'+' でプロットされているのは倍精度での誤差である。N が 2^8 から丸め誤差等が累積し、誤差が倍精度の machine epsilon である 2^{-53} オーダから N の増加とともに徐々に大きくなっているのが観察できる。

'×' でプロットされているのは *omf77* 4 倍精度で

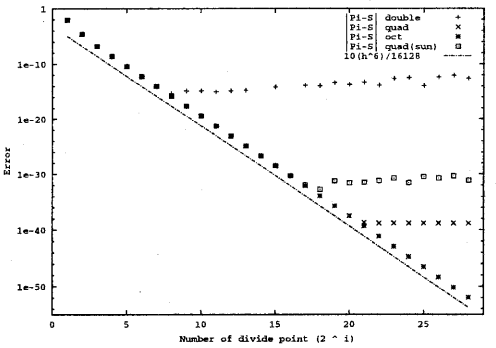

図 **2**: $\int_0^1 \frac{4}{1+x^2}dx$ のシンプソン積分による誤差

の誤差で、N が 2^{21} から、誤差が *omf77* 4 倍精度の machine epsilon である 2^{-128} オーダでほとんど変化しない。GNU MP のドキュメント[4] には、無限精度演算を行った上で、mpf_t の精度で切り捨てる ("truncate") との記述があり、これが影響しているものと推測される。

参考のため、同程度の精度を持つ SUNWspro f77 の 4 倍精度を用いて同じプログラムを Sun Ultra Enterprise 450 (UltraSPARC-II 300MHz/Solaris 2.6) で動作させた結果を、'□' でプロットしている。こちらは倍精度と同様に、SUNWspro f77 の 4 倍精度の machine epsilon のオーダから N の増加とともに徐々に誤差が大きくなっているのが観察できる。

'※' でプロットされているのは *omf77* 8 倍精度の誤差である。2^{28} 程度の N では誤差の増加もしくは横這い状態は観測できていない。したがって、さらに N を増加させれば誤差が少なくなることが推測できる。

この実験ではシンプソン法を用いた数値積分を例にとったが、その他の数値計算アルゴリズムに関しても同様の実験が行える。重要なのは、同じソースコードすなわち同じアルゴリズムのプログラムで、精度のみを変化させて誤差の振舞いを評価できる点である。

5.2 GNU MP C 言語インターフェースを直接用いたプログラムとの実行速度比較

omf77 が GNU MP を用いて多倍精度浮動小数演算を行う場合と、直接 C 言語により GNU MP を用いる場合との速度比較のため、自然対数の底 e を $1 + \sum_{i=1}^{\infty} \frac{1}{i!}$ により $\frac{1}{i!}$ < machine epsilon を収束条件として算出するプログラムを、GNU MP C 言語インターフェースを用いた C 言語プログラム、*omf77* 多倍精度浮動小数型を用いた Fortran 77 プログラムの両者で作成した。このプログラムを、**5.1** 項 同様 Compaq ES40 において精度 128 bit ～ 512 bit まで変化させて実行速度を比較したグラフを 図 **3** に示す。

図 **3** の縦軸は、

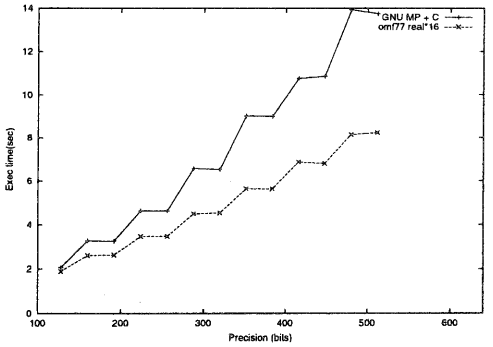

図 3: GNU MP C 言語インターフェースを直接用いたプログラムとの実行速度比較

- machine epsilon の算定☆
- machine epsilon の精度での e の計算

の部分を 1 万回行った場合の総時間である。

図 3 より、測定した全ての精度域で、*omf77* 多倍精度浮動小数型を用いた Fortran プログラムの方が実行速度が速いことが観察できる。これは、*omf77* では `mpf_init_set_d()` や `mpf_set()` 等の `mpf_t` 初期化関数が全く呼ばれていないことによるものだと推測できる。

5.3 大浦氏の Gauss-Legendre 改良型 AGM アルゴリズムによる π の計算

AGM (算術幾何平均) による π の計算アルゴリズムとして、Gauss-Legendre 公式を改良した大浦氏のアルゴリズムがある。このアルゴリズムの実際の実装として、C 言語による FFT 積算を用いた多倍長 π 計算プログラムが同氏により公開されている[5]。GNU MP も FFT 積算をサポートしており☆☆、64 bit アーキテクチャでは 491393 bit 以上の精度より、自動的に FFT 積算ルーチンが使用される。ここでは、プログラマが一切多倍長演算が行われることを意識せず、このアルゴリズムをそのまま Fortran 77 プログラムとして実装し、同氏の実装による C プログラム (最速であった `pi_cas5`) との速度比較を、2^{20} 桁の π の計算で行ってみた。この桁数であれば GNU MP の FFT 積算ルーチンが使用される。測定は **5.2 項** 同様 Compaq ES40 で行った。**表 1** にその結果を示す。

表 1: `pi_cas5` との実行時間比較

`pi_cas5`	18 秒
omf77	78 秒

両者の速度差は 4.3 倍という結果を得た。人間の手である程度チューニングされた C プログラムに対して、全く何も考えずにアルゴリズムをそのまま Fortran 77 に置き換えただけのプログラムが、これだけの性能差しか生んでないことに注目したい。

また、π を 2^{20} 桁計算するために書かれたコード量を見ると、`pi_cas5` が 4948 行, *omf77* 版が 101 行と、*omf77* 版の方が少ない。このことから、プログラミングの容易さは、コンパイラでの多倍精度浮動小数演算サポートのある *omf77* を用いた方が上であると言える。

☆ *omf77* 版では `mpeps` は用いていない。

☆☆ GNU MP 自身のインストール時に、`configure` スクリプトに `--enable-fft` フラグを付加することで可能となる。

6. ま と め

我々は Omni OpenMP コンパイラシステムの Fortran 77 コンパイラである *omf77* に多倍精度浮動小数型を実装し、以下の結論を得た。

- コンパイラが基本型として多倍精度浮動小数型をサポートしていると、プログラム作成が容易になる
- GNU MP を多倍長演算エンジンとして使用した場合、プログラムによっては GNU MP を直接使用して C 言語で記述するより高速になる

今後の予定として、

- 現時点で未実装の超越関数 intrinsic 関数 `sin, asin, sinh, cos, acos, cosh, tan, atan, atan2, tanh, log, log10` と、多倍精度浮動小数型[多倍精度浮動小数型]の巾乗の実装
- GNU MP `mpf_t` 型でなく、IEEE754 準拠の丸めをサポートした `mpfr_t` 型での実装

を考えている。

謝　　辞

omf77 の多倍精度浮動小数拡張にあたり、貴重なご意見を頂いた TEA meeting グループの皆様に感謝の意を表します。

参 考 文 献

1) `http://www.lmu.edu/acad/personal/faculty/dmsmith2/FMLIB.html`
2) `http://www.netlib.org/bmp`
3) `http://www.swox.com/gmp/`
4) GNU MP Info, "Floating-point Functions" (GNU MP がインストールされている環境であれば、`info gmp` のコマンドを発行する事で参照可能)
5) `http://momonga.t.u-tokyo.ac.jp/~ooura/pi_fft.html`