The International Journal Of Engineering And Science (IJES)
|| Volume || 6 || Issue || 2 || Pages || PP 06-11 || 2017 ||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

Practical Guides on Programming with Big Number Library in Scientific Researches

Jianhui LI[1], Xingbo WANG[2]
[1]Department of Computer Science, Guangdong Neusoft Institute, FOSHAN CITY, GUANGDONG PROVINCE, PRC
[2]Department of Mechatronic Engineering, FOSHAN University, FOSHAN CITY, GUANGDONG PROVINCE, PRC

ABSTRACT
This article demonstrates some practical skills in programming with GMP and MPFR big number libraries according to the traits of big numbers and C/C++ programming conventions. It points out that, the conditional expressions, the incremental treatment, the loops together with their stops, and the pointer arguments are critical issues in C/C++ programming with the big number libraries. By exhibiting certain guidance and demonstration, the article summarizes a framework to the program that treats big number computations and presents a sample as well as numerical experiments to factorize big odd composite number.

Keywords: *Big number computation, GMP library, C/C++ programming, Integer factorization.*

Date of Submission: 25 January 2017 Date of Accepted: 09 February 2017

I. INTRODUCTION

Whenever a scientific project related big number computations, such as encryption and decryption, the GNU GMP big number library is a mandatory tool for a programmer to choose, as stated in articles [1] and [2]. In point-of-view of programming, big number computations form a special programming style distinguishing from the conventional programming styles, as seen in articles [3] to [5]. Therefore, conventional programming habits need a lot of modification to fit the big numbers' traits, which exceed the conventional computers' processing capabilities. However, literatures of guidance to such a change are seldom seen in the public publications. This article presents some key programming issues of using big number library so as to tell the beginners to program in an predictable efficiency and quality.

II. TRAITS OF BIG NUMBERS

Big numbers means their representations exceed that of a conventional computer's. For example, the biggest integer that can be expressed by a conventional computer of 32 bits is $2^{32}-1=4294967295$ while in some scientific computation, an integer can be over 100 bits of decimal digits. For example, the number RSA100 is 1522605027922533360535618378132637429718068114961380688657908494580122963258952897654000350692006139, which has 101 bits of decimal digits. Except for the big integers, scientific computations also often meet very big real numbers, as introduced in article [6].
Hence big numbers have the following traits:
1. They are necessary in scientific computations;
2. They are too big to be represented in a conventional way;
3. There are special libraries to treat computations of big numbers;
4. Programming with big numbers requires special skills.

III. GMP AND GMFR BRIEFS

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. GMP has a rich set of functions, and the functions have a regular interface.The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc. The first GMP release was made in 1991. Now it is version 6 and GMP is distributed under the dual licenses, GNU LGPL v3 and GNU GPL v2, which make the library free to use, share, and improve. GMP is said to be fastest both for small operands and for huge operands and is available for C/C++ programming, which can be referred to GMP library website at *gmplib.org*.
GNU MPFR is a GNU portable C library for arbitrary-precision binary floating-point computation based on GMP Library. The computation is both efficient and has a well-defined semantics: the functions are completely specified on all the possible operands and the results do not depend on the platform. MPFR provides each number a precision. The floating-point results are correctly rounded to the precision of the target variable, in any of the four IEEE-754 rounding modes. MPFR implements all mathematical functions from C99 and other usual mathematical functions,

including the logarithm and exponential in natural base, base 2 and base 10, the $\log(1+x)$, the six trigonometric and hyperbolic functions and their inverses, the gamma, zeta and error functions, the arithmetic geometric mean, the power (*xy*) function. All those functions are correctly rounded over their complete range. Detail functions and programming reference can be seen at *www.mpfr.org*

IV. PROGRAMMING SKILL WITH GMP OR MPFR LIBRARY

Since both MPFR and GMP are C-style programmable, this section only takes GMP as an example to demonstrate some key skills in the programming on six aspects: new data type, conditional expressions, increment and decrement, loops and their stops, pointer arguments and their returning values, and using string to exchange values

4.1 New Data Types

GMP library supports three types of data, integer as well as rational and floating-point number. It uses keyword *mpz_t* , *mpq_t* and *mpf_t* to denote the three kinds of big numbers. For example, the following codes declare two variables of big integers and two variables of big floating-point numbers, respectively.

```
mpz_t  m,n;              /* declare two variables of big integers: m, n*/
mpf_t  s,t;              /* declare two variables of big floating-point numbers: s, t*/
```

The types *mpz_t*, *mpq_t* and *mpf_t* are actually pointer type of struct in C/C++. Thus these kinds of variables cannot be directly evaluated. For example, the following statement is wrong.

```
m=1234567890102020;  /*Wrong statement! */
```

Variables of big number type must be first initialized and then set a value, as following examples.

```
mpz_init2(m,200);        /*Initialize variable m by 200 bits of space to store it */
mpz_set_ui(m, 50);       /* Set unsigned integer 50 to m*/
```

There are several methods to set value to a variable of big number; readers can refer to the GMP library manual to find them, which are very easy to understand.

4.2 Conditional Expression

In conventional C/C++ programming, a conditional expression is usually very simple. For example, *A<B*, *A==B* or *A<=B* always occur in conventional C/C++ programming. However, these expressions cannot be directly used in a program related with big numbers unless they are treated normally in advance. Note that, either C or C++ requires type-compatible in compiling process; hence one needs first to treat in conditional expression the two sides to be the same type, especially, the types of big numbers. For example, when judging if two big integers are equal, it is recommended to use the following statements.

```
unsigned cmp;                /*Define an unsigned cmp*/
cmp=mpz_cmp(m, n);           /*Calling mpz_cmp function to compare m and n*/
if(cmp<0) gmp_printf("%Zd is smaller than %Zd \n", m, n);
 /*Note: should use gmp_printf  and %Zd format because m and n are big integers */
if(cmp=0) gmp_printf("%Zd is equal to %Zd \n", m, n);
```

Generally speaking, using *mpz_cmp* function to compare two big numbers is a better custom in programming related with big number operations.

4.3 Increment and Decrement

In conventional C/C++ programming, expressions such as *i++*, *--i*, *a+=b* and *a-=b*, are frequently seen in programs. But these seldom occur in program of big numbers. It is recommended to use the following statements to treat increment or decrement of a variable *m* of big numbers.

```
unsigned inc=2;              /* Define an unsigned integer inc*/
mpz_add_ui(m, m, inc);       /* Add inc to m*/
mpz_sub_ui(n, n, inc);       /*Subtraction inc from n*/
```

If the increment or decrement is another big number, the following statements can be a reference.

```
mpz_add(m, m, n);            /* Add n to m*/
```

```
        mpz_sub_ui(n, n, m);        /*Subtraction m from n*/
```

4.4 Loops and Their stops

Loops might occur anywhere in program if they are required. In conventional program, there are three kinds of loops, the for-loop, the while-loop and the do-while-loop. Programming with big numbers also applies these loops. The difference between the big number programs and the conventional programs are their stops. A loop in program with big numbers seldom has *explicit conditions* to stop the loop because many computations with big numbers are highly related algorithms of *searching*, *divide-and-conquer*, etc., which performed by *iteration* or *recursion*. Consequently, controlling the stop of a loop is a key issue in programs of big number. In practical computations, while-loop and do-while-loop are appreciate and a stop activator is always placed in the loop, as the following codes demonstrate.

```
/*===Search from S to find common divisor d between R and N====
======== until searching RcylNumber steps or find a d ========
=====where S, R, d, N and RcylNumber are all big numbers======*/

  unsigned found=0;                  /* found: stop activator . If found =1, stop loop*/
 unsigned cmp=1;
 mpz_set_ui(tmp,0);                  /* Big number tmp is used to be a recycle variable*/
 while(1)                            /* A never end loop */
 { mpz_add_ui(tmp,tmp,1);            /* Recycle variable increases by 1*/
   mpz_gcd(d , R, N);                /* Compute common divisor d between R and N*/
   cmp=mpz_cmp_ui(d,1);              /* Judge if d=1?*/
   if(cmp>0)                         /*d>1 means a common divisor is found*/
  { found=1;                         /*Activate stop of the loop*/
    break;                           /* break the loop*/
   }
  mpz_add_ui(N,N,2);                 /*Otherwise, prepare next computation by changing N*/
  cmp=mpz_cmp(tmp, RcylNumber);      /*Judge if reaching the maximal computational steps*/
  if(cmp==0) break;                  /*If it reaches, stop the loop*/
    }
```

4.5 Pointer Arguments and Their Returning Values

As stated previously, the types *mpz_t* and *mpf_t* are actually pointer types of struct in C/C++; hence the arguments of *mpz_t* or *mpf_t* in a function can by default return values of computations in the body of the function. This behavior occurs both in gmp-defined function and in user-defined function. For example, the following user-defined function SetMersenne (See in Section 5.2) can return a big number by its argument *M*.

4.6 Using String to Exchange Values

Sometimes, it is necessary to change part of a big number into a conventional number. Then the string-type can play an important role in exchanging the data. Note that, gmp library provides function to change *unsigned* type into *mpz_t* type, but does not provide a function to change the *mpz_t* type into the *unsigned* type. Hence using *sscanf* function can easily solve the problem, as the following codes demonstrate.

```
unsigned getTail(mpz_t T)
{unsigned tail;          /* An unsigned variable tail*/
  char s[10];            /* An string s*/
  mpz_t r;               /* mpz_t type r*/
  mpz_init2(r,10);       /*Initialize r*/
  mpz_mod_ui(r,T,10);    /*r=T mod 10*/
  mpz_get_str(s,10,r);   /* Save the mpz_t type r to the string s*/
  sscanf(s,"%d",&tail);  /* Obtain tail from s*/
  mpz_clear(r);          /*Release r*/
  return tail;           /*Return tail*/
}
```

V. INSTANCE OF BIG NUMBER COMPUTATION

As stated above, programming with big number computations has its own traits based on two considerations. One is that the program should meet the needs of the traits of big number libraries, and the other is that the program should match to the algorithms of big number computations. This section presents a complete example that factorizes a big odd number, as introduced in article [7].

5.1 Lemma and Algorithm

Lemma 1 Suppose N is an odd composite number; then the following statements are true.

(1) If $N = 6n+1$ and there exists a $k < \frac{\sqrt{n}+1}{2}$ such that $(6k+1) \mid (n-k)$, then $(6k+1) \mid N$.

(2) If $N = 6n+1$ and there exists a $k < \frac{\sqrt{n}+1}{2}$ such that $(6k-1) \mid (n+k)$, then $(6k-1) \mid N$

According to the lemma, algorithm to factorize a big odd integer can be simply designed by dealing with the following issues.

```
=======Algorithm To Factorize an Odd Number of 6n+1=======
(1) Determine the form of an big odd number in terms of its form $6n \pm 1$;
(2) Extract n from N;
(3) Set a maximal computational limit $k_{max} \le \left\lfloor \frac{\sqrt{n}+1}{2} \right\rfloor$;
(4) Search and find a k such that $(6k+1) \mid (n-k)$ or $(6k-1) \mid (n-k)$.
==============End of Algorithm Descript================
```

5.2 C Program to Realize the Algorithm

The previous algorithm can be realized by using GMP big number library as follows.

```
void SetMersenne(mpz_t M, unsigned int p)
/* Produce Mersennne Number M*/
{   mpz_set_ui(M,1);
    mpz_mul_2exp (M, M, p);//M=M*2^p
    mpz_sub_ui(M,M,1);
    printf("\n FactorizeMersenneNumber 2^(%u) - 1\n",p);}

void get_n_from_N(mpz_t n, unsigned *tail, mpz_t N)
{/* Note: argument n is a pointer */
    unsigned whatN;
    char s[10];                    /*Use string s*/
    mpz_mod_ui(n,N,6);             /* n=N mod 6, resulting 1,3,5*/
    mpz_get_str(s,10,n);           /* Change n to s*/
    sscanf(s,"%d", &whatN);        /* Obtain whatN from s*/
    switch(whatN)
   { case 3: break;                /* N is 3's multiple*/
     case 1: mpz_sub_ui(n,N,1);  /* When N=6n+1*/
            *tail=1; break;
     case 5: mpz_add_ui(n,N,1);  /* When N=6n-1*/
            *tail=5;  break; }
     mpz_divexact_ui(n,n,6);       /*Extracting n from N = 6n+1 or N= 6n -1*/
     return ;}

void GetMaxLoopNumber(mpz_t lmax, mpz_t n)
{  mpz_sqrt(lmax,n);                  /*lmax=sqrt(N)*/
    mpz_add_ui(lmax,lmax,1) ;      /*lmax=lmax+1*/
    mpz_fdiv_q_ui(lmax,lmax,2) ;  /*lmax=lmax/2*/
}

void SetD_1(mpz_t d, mpz_t k)
{mpz_set(d,k);                /*Thus d=k*/
 mpz_mul_ui(d,d,6);           /* Thus d=6*k*/
 mpz_add_ui(d,d,1);           /*Thus d=6*k+1*/
}
void SetD_2(mpz_t d, mpz_t k)
```

```
{mpz_set(d,k);                /*Thus d=k*/
mpz_mul_ui(d,d,6);            /* Thus d=6*k*/
mpz_sub_ui(d,d,1);            /*Thus d=6*k+1*/
}

int DivTest1(mpz_t res, mpz_t d, mpz_t k, mpz_t n)
{mpz_sub(res,n,k);                /*res=n-k*/
 if(!mpz_divisible_p(res,d))  /* d not div res*/
   return 0;
mpz_set(res,d);                 /*Set d to res */
return 1;}

int DivTest2(mpz_t res, mpz_t d, mpz_t k, mpz_t n)
{   mpz_add(res,n,k);              /*res=n+k*/
if(!mpz_divisible_p(res,d))     /* d not div res)*/
   return 0;
mpz_set(res,d);                   /*Set d to res */
return 1;}

void Factorize(mpz_t res, mpz_t k, mpz_t s, mpz_t d, mpz_t n,  mpz_t lmax)
{   int cmp=1;
   mpz_set_ui(k,0);                      /* Set initial value of tmp by 0 */
   cmp=mpz_cmp(k, lmax);                /* Compare k with the maximal loop-number lmax*/
while(cmp<0)                            /* cmp<0 means k<lmax */
{mpz_add_ui(k,k,1);                     /* k increases by 1*/
     SetD_1(d, k);                      /* Set d=6*k+1 */
     if(DivTest1(res,d,k,n)) break;      /*When d|(n-k) then d|N, return res and stop loop*/
                                        /*Otherwise test the other case*/
     SetD_2(d,  k);                     /* Set d=6*k-1 */
     if(DivTest2 (res, d,k, n))
     break;       /*When d|(n-k) then d|N, return res and stop loop*/
     cmp=mpz_cmp(k, lmax);              /* Compare k with the maximal loop-number lmax*/
  }
  return ;}

void InitBigData(mpz_t &d1, mpz_t &d2, mpz_t &d3, mpz_t &d4, mpz_t &d5,
                    mpz_t &d6, mpz_t &d7, unsigned nBits)
{mpz_init2(d1,nBits); mpz_init2(d2,nBits);mpz_init2(d3,nBits);mpz_init2(d4,nBits);
mpz_init2(d5,nBits);mpz_init2(d6,nBits); mpz_init2(d7,nBits);}

void ClearMpzData(mpz_t d1, mpz_t d2, mpz_t d3, mpz_t d4,
                     mpz_t d5, mpz_t d6 , mpz_t d7)
{  mpz_clear(d1);   mpz_clear(d2);  mpz_clear(d3);   mpz_clear(d4);
   mpz_clear(d5);   mpz_clear(d6);  mpz_clear(d7);   }

int main()
{  unsigned tail, Bits=200,p=113;
   mpz_t N, factor,s,d,n,l,lmax;
   InitBigData(factor, s, d, n, l, lmax, N,Bits);
SetMersenne(N, p);
   get_n_from_N(n, &tail, N);               /*Get tail of N and extract n from N*/
   GetMaxLoopNumber(lmax, n);
if(tail==1)
{Factorize(factor, l, s, d, n, lmax);
 gmp_printf("l=%Zd Factor=%Zd\n", l,factor);
 ClearMpzData(factor, s, d, n, l, lmax, N);
 return;}
gmp_printf("Not fit the form 6n+1\n");
```

```
ClearMpzData(factor, s, d, n, l, lmax, N);
Return 0;}
```

5.3 Numerical Tests

Applying the previous codes to test factorizing some of the Mersenne numbers reveals expected results. Table1 1 shows the experimental results, which are made on a PC with an Intel Xeon E5450 CPU and 4GB memory.

Table 1 Experiments on Mersenne and Fermat Numbers

N	*N's factor*	Searching number *k*
M67=2^{67}-1	193707721	32284620
M71=2^{71}-1	228479	38080
M83=2^{83}-1	167	28
M97=2^{97}-1	11447	1908
M103=2^{103}-1	2550183799	435030633
M109=2^{109}-1	745988807	124331468
M113=2^{113}-1	3391	565

VI. CONCLUSION

GMP and MPFR big number libraries are frequently used in scientific computations. Programming with these libraries requires certain skills, especially the expressions of conditions, increment, decrement, loops and their stops. Only plentiful experiences can help a programmer to program in an efficient and highly skillful way. The contents in this article summarize the authors' consideration on the program from practical point of view. The authors hope it a valuable reference to the related developers.

ACKNOWLEDGEMENTS

The research work is supported by Department of Guangdong Science and Technology under projects 2015A030401105 and 2015A010104011, Foshan Bureau of Science and Technology under projects 2016AG100311, 2016AG100652, 2016AG100792 and 2016AG100382 Special Innovative Projects 2014KTSCX156, 2014SFKC30 and 2014QTLXXM42 from Guangdong Education Department. The authors sincerely present thanks to them all.

REFERENCES

[1]. L. Rutten and M.V. Eekelen. *Efficient and formally proven reduction of large integers by small moduli*, Acm Transactions on Mathematical Software, 2008, 37(2):163-172.
[2]. Isa, Mohd Anuar Mat, et al. *A Series of Secret Keys in a Key Distribution Protocol*. Transactions on Engineering Technologies. Springer Netherlands, 2015:615-628.
[3]. W.B. Hart. *Fast Library for Number Theory: An Introduction* (Mathematical Software – ICMS 2010. Springer Berlin Heidelberg, 2010)
[4]. L. M. Surhone, M. T.Tennoe and S. F. Henssonow. *Fast Library for Number Theory* (Betascript Publishing, 2010)
[5]. T. Granlund and G. D. Team. *GNU MP 6.0 Multiple Precision Arithmetic Library* (Samurai Media Limited, 2015).
[6]. L Fousse,G Hanrot, V Lefevre, et al. *MPFR: A multiple-precision binary floating-point library with correct rounding*, ACM Trans. Math. Softw. 2007:00000818.
[7]. Xingbo WANG. *Seed and Sieve of Odd Composite Numbers with Applications In Factorization of Integers*, OSR Journal of Mathematics (IOSR-JM), 2016, 12(5, Ver. VIII): 01-07

Author's Biography

Jianhui Li is an assistant professor of Guangdong Neusoft Institute. He obtained his Ph D degree in Hunan Agriculture University in 2013, and since then has been a staff in charge of affairs of scientific researches in the university. He is skill at computer programming, wireless-sensor network development and computer image process.