The International Journal of Engineering and Science (IJES)
|| Volume || 6 || Issue || 4 || Pages || PP 01-06 || 2017 ||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

One More Comments on Programming with Big Number Library in Scientific Computing

Dongbo FU
Department of Computer Science, Guangdong Neusoft Institue FOSHAN CITY, GUANGDONG PROVINCE, PRC, 528200

ABSTRACT
This article makes a comment on programming with big number libraries like GMP and MPFR libraries. The comment proposes not using recursive procedure in programs since recursive procedure that performs big number computations might use big size of stack while neither the operation system nor a compile system can afford to provide the needed size. Based on the comments, the article recommends designing proper algorithm that avoids a recursive procedure. With an example that uses the GMP big number library in factorization of big odd number, the article demonstrates how to convert a recursive procedure into a non-recursive procedure and its numerical experiments sustain that the sample program reveals expected results.
Keywords *–Big number computation, GMP library, C/C++ programming, recursive procedure.*

Date of Submission:08 April 2017 Date of Accepted: 22 April 2017

I. INTRODUCTION

Article [1] has presented a practical guide to programming with GNU GMP big number library, including the data type, the conditional expressions, the incremental treatment, the loops and the pointer arguments in coding skills. It is really a necessary reference for a programmer who programs with big number libraries. However, the article omitted two more important things in programming with big number libraries, the prohibition of using recursive functions and the corresponding algorithm design that should avoid using the recursive functions. This article makes a supplement on the two issues.

II. RECURSIVE FUNCTION AND STACK MANAGEMENT OF OS

School textbooks, as [2] and [3], tell us that, recursive function is a subroutine that can call itself as part of its execution. A typical recursive function, say the function *RecursiveProc*, is illustrated by the following C++ pseudo-codes.

```
//============ C++ pseudo-codes ==============
type RecursiveProc (ParaLists)
{    type res;
     //Computing something
     //Computing new 'Paralists'

               res=RecursiveProc (ParaLists);  //call the function itself.

               //Computing some other things

       return res;
}
//========================================
```

Schoolbooks also tell us, any time a recursive function is called, a stack is used to save the '*ParaLists*' and the more arguments in the '*ParaLists*' the bigger size of the stack is required. In another word, a recursive function is always companied with a stack's pushing-and-popping operations.
Now comes a question how big a size of a stack can own. Actually, the answer to this question depends on at least two factors: the compiling system that is used to compile the source codes and the Operation System (OS). Referring to the manual of Visual Studio 2010, for example, it can see that, on Windows System, either 32 bits or

64 bits, the maximal stack size is reserved to be 1Mb by default. Although a veteran programmer can adjust the size to a little bigger, he/she will see that it cannot be enlarged to his/her expectations due to limitation of OS's managing capability.

III. ABANDON USING RECURSIVE FUNCTIONS

By the relationship between a recursive program and the stack it uses, one can draw a conclusion that, the size of 1Mb's stack might be enough for a conventional recursive computation that might manage several Kb's stack, but for a large number computing, the 1Mb's stack is never enough.

Let's take an example in finding an integer's divisor. According to article [4], an odd integer *N*=*pq* such that $3 \le p < q$ will have its divisor *p* to be found in the interval $[N^{N}_{(m+1,\xi(qp))}, N^{N}_{(m+1,2^m-1)}]$, where $N^{N}_{(m+1,2^m-1)} = 2^m N - 1$, $N^{N}_{(m+1,\xi(qp))} = N^{N}_{(m+1,2^m-1)} - 2\left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor$.

Now consider we use a divide-conquer approach to find the divisor *p* and suppose we designed a recursive function to perform the search. Without loss of generality, we name the function by *DCsearch* and its C++ pseudo-codes (in GMP library) are as follows.

```
//========= DC Search C++ pseudo-codes =========
int DCsearch (mpz_t &p,mpz_t N, mpz_t left, mpz_t right)
{
    int res;
     mpz_sub( mid, right, left);        //where mid has been initialized earlier
     mpz_fdiv_q_ui( mid, mid,2);
     if(FindGCD(p, mid, N)) // test if mid contains p;
                return 1;

                mpz_sud_ui(lft, mid,2);
                mpz_add_ui(rht, mid,2);

                res=DCsearch(p,N,left,lft);  //recursive computing
                if(res==1) return 1;

                res=DCsearch(p,N,rht,right); //recursive computing
                if(res==1) return 1;

  return 0;
}
            //=======================================
```

The above computing procedure is a typical binary search one. If *N* is small, the procedure can work well. But when *N* is a big number, the procedure will cause a 'stack overflow' error, as RBarry Young declaimed in [5]. Why such problem occurs? Taking a not very large number N=*1123877887715932507* as an example, one can see $\left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor = 530065536$, which results in costing a vast overflow stack even through a binary recursive search.

Consequently, abandon using recursive procedure is a regulation in programming with large number library.

IV. CONVERT RECURSIVE PROCEDURE INTO NON-RECURSIVE ONE

Now that a recursive procedure cannot work with big number programming, it is necessary to convert a recursive procedure to a non-recursive one. Such conversional work has early been investigated. For example, the articles [6] and [7] both explored how to turn a recursive algorithm into a non-recursive one. There are various approaches to turn recursive algorithms to their non-recursive ones. This article does not intend to show their details. As an example, here is introduced a frequently used approach in finding an integer that has the greatest common divisor (GCD) with an integer *N* in an interval $Itv = [Il, Ir]$. Suppose Itv contains N_{sn} integers. We first subdivide Itv into $2m+1$ subintervals by

$$N_{sn} = Ir - Il + 1;$$

and

$$N_{sn} = (2m) \times \left\lfloor \frac{N_{sn}}{2m} \right\rfloor + N_{sn} \bmod (2m)$$

Let $M = \left\lfloor \frac{N_{sn}}{2m} \right\rfloor, M_{sn} = N_{sn} \bmod (2m)$; then there are 2*m* equal-length subintervals in each of which contains *M* integers and there is one subinterval that contains M_{sn} integers. Conventionally, a divide-conquer algorithm search can be applied on each of the $2m+1$ subintervals; however, as stated above, it is better to design a non-recursive algorithm for big number operations. An appreciative sample is shown below

```
============Non-recursive Search============
For $i = 0$ to $i = m-1$
  For $j = 0$ to $j = M-1$
    Begin
     $el = Il + 2(j*m+i)$ ; if(FindGCD(N, el)) Return(i, j,el);
     $er = Ir - 2(j*m+i)$ ; if(FindGCD(N, er)) Return(i, j,er);
    End
For $i = 0$ to $i = M_{sn}-1$
    Begin
     $el = Il + 2(n_{sn}*m+i)$ ; if(FindGCD(N,el,)) Return(i, 0,el);
    End
=========================================
```

V. INSTANCE OF BIG NUMBER COMPUTATION

This section presents an example of factoring a big odd number. The example is to realize the approach that was introduced in [4].

5.1 Lemma and Algorithm

Lemma 1 Suppose *N* is an odd composite number; then *N* can be factorized in at most $\left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor$ searches. And an algorithm is proposed below.

```
======== Squeeze Searching Algorithm=========
Input: Odd composite number N.
Step 1. Calculate searching level: $K = \lfloor \log_2 N \rfloor - 1$;
Step 2. Calculate the largest searching steps: $l_{\max} = \left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor$;
Step 3. Calculate variables:
 $ul = 2^K N - 1$; $ll = ul - 2l_{\max}$;
 $ml = ll + l_{\max}/2$; $left = ml - 2$; $right = ml + 2$
Step 4. If  FindGCD(N,ll) or FindGCD(N,ul)
             or FindGCD(N, ml) return GCD;
        Else
        Begin loop
           $ul = ul - 2$; $ll = ll + 2$; $left = left - 2$; $right = right + 2$
          If  FindGCD(N, ll) or FindGCD(N, ul)
             or FindGCD(N, left) or FindGCD(N, right)
           return GCD;
End loop
==========================================
```

5.2 C++ Program to Realize the Algorithm

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

```
void Bsearch (mpz_t &Rt1,mpz_t &Rt2,mpz_t &steps,mpz_t left, mpz_t right)
{
  //Search from mid of [left, right], to find common divisor between Root and an activeNode;
   unsigned found=0;
   unsigned cmp=0;

   mpz_sub(ml,right,left);                //calculate the middle point
   mpz_fdiv_q_ui(ml,ml,2);
   mpz_add(ml,ml,left);
   if(mpz_odd_p(ml)==0)                   //ensure the mid-point is odd
      mpz_add_ui(ml,ml,1);

   mpz_gcd(Rt1,Root,ml);                  //Find gcd between Root and ml and save it to Rt1
   cmp=mpz_cmp_ui(Rt1,1);                 //Check if gcd=1

   if(cmp>0)                              //Find it! Use it to obtain Rt2, and then return
   { mpz_divexact(Rt2,Root,Rt1);
     return;  }

    //Not Found! Continue finding

    mpz_add_ui(rht,ml,2);                  //to right half-interval
    mpz_sub_ui(lft,ml,2);                  //to left half-interval

   mpz_sub(tmp1,ml,left);                  //length of the left half-interval
   mpz_fdiv_q_ui(tmp1,tmp1,2);
   mpz_add(tmp1,tmp1,left);                //mid of the left half-interval
   if(mpz_odd_p(tmp1)==0)
      mpz_add_ui(tmp1,tmp1,1);

    mpz_gcd(Rt1,Root,tmp1);                 //check the mid point
    cmp=mpz_cmp_ui(Rt1,1);
   if(cmp>0)                                //Find it! Use it to obtain Rt2, and then return
   { mpz_divexact(Rt2,Root,Rt1);
     return;  }

   //Not Found! Continue finding
   mpz_sub_ui(llft,tmp1,2);                 //to left half of the left half-interval
   mpz_add_ui(rlft,tmp1,2);                 //to right half of the left half-interval


   mpz_sub(tmp1,right,ml);                  //to right half of [left, right]
   mpz_fdiv_q_ui(tmp1,tmp1,2);
   mpz_sub(tmp1,right,tmp1);                //mid of the right half-interval
   if(mpz_odd_p(tmp1)==0)                   //ensure the mid-point is odd
      mpz_add_ui(tmp1,tmp1,1);

    mpz_gcd(Rt1,Root,tmp1);                 //check the mid of the right half
   cmp=mpz_cmp_ui(Rt1,1);
   if(cmp>0)                                //Find it! Use it to obtain Rt2, and then return
   { mpz_divexact(Rt2,Root,Rt1);
     return;  }

    //Not Found! Continue finding
   mpz_sub_ui(lrht,tmp1,2);                 //to left half of the right half-interval
   mpz_add_ui(rrht,tmp1,2);                 //to right half of the right half-interval

   while(1)
   {  mpz_add_ui(steps,steps,1);             //increment of the counter

   mpz_gcd(Rt1,Root,left);                   // check the left end
   cmp=mpz_cmp_ui(Rt1,1);
   if(cmp>0)                                 //Find it! Use it to obtain Rt2, and then return
   { mpz_divexact(Rt2,Root,Rt1);
     return;   }

   mpz_gcd(Rt1,Root,llft);                    //check left-end of the left half
   cmp=mpz_cmp_ui(Rt1,1);
   if(cmp>0)                                  //Find it! Use it to obtain Rt2, and then return
   { mpz_divexact(Rt2,Root,Rt1);
     return;  }

   mpz_gcd(Rt1,Root,rlft);                    //check the right-end of the left-half
   cmp=mpz_cmp_ui(Rt1,1);
   if(cmp>0)                                  //Find it! Use it to obtain Rt2, and then return
   { mpz_divexact(Rt2,Root,Rt1);
```

```
  return;  }

 mpz_gcd(Rt1,Root,lft);                    //check lft
mpz_cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)                                  //Find it! Use it to obtain Rt2, and then return
{ mpz_divexact(Rt2,Root,Rt1);
  return; }

mpz_gcd(Rt1,Root,rht);                     //check rht
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)                                  //Find it! Use it to obtain Rt2, and then return
{mpz_divexact(Rt2,Root,Rt1);
  return; }

mpz_gcd(Rt1,Root,lrht);                    //check lrht
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)                                  //Find it! Use it to obtain Rt2, and then return
{ mpz_divexact(Rt2,Root,Rt1);
  return;  }
 mpz_gcd(Rt1,Root,rrht);                   //check rrht
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)                                  //Find it! Use it to obtain Rt2, and then return
{mpz_divexact(Rt2,Root,Rt1);
  return;  }

mpz_gcd(Rt1,Root,right);                   //check right-end
if(cmp>0)                                  //Find it! Use it to obtain Rt2, and then return
{mpz_divexact(Rt2,Root,Rt1);
  return; }

mpz_sub_ui(right,right,2);                 //shrink the intervals to their middle  by 2
mpz_add_ui(left,left,2);
mpz_sub_ui(lft,lft,2);
mpz_add_ui(rht,rht,2);

mpz_sub_ui(llft,llft,2);
mpz_add_ui(rlft,rlft,2);
mpz_sub_ui(lrht,lrht,2);
mpz_add_ui(rrht,rrht,2);
}
}
```

5.3 Numerical Tests

Applying the previous codes to test factorizing some big 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 Big Numbers

N's Factorization	*Searching Steps*
N1= 11238778877159325O7=299155897×3756830131	*17061564*
N2=11293671024548668 81=25869889×43655660929	*1025702*
N3=29742315699406748437=372173423×79915205819	*1834479*
N4=35249679931198483=59138501×596052983	*5166741*
N5=208127655734009353=430470917×483488309	*12869593*
N6=331432537700013787=114098219×2904800273	*2605343*
N7=3070282504055021789=1436222173×2137748993	*61027776*
N8=3757550627260778911=16053127×234069700393	*3502182*
N9=24928816998094684879=347912923×71652460573	*30523926*
N10=10188337563435517819=70901851×143696355169	*667123*

III. CONCLUSION

GMP and MPFR big number libraries are conventional tools that are frequently used in scientific computations. Programming with these libraries requires a programmer to know their essences. Use non-recursive process is a rule for the programming. As stated in this article, proper algorithm design is a key to the problem. As an example, I list the source codes and hope it a valuable reference to the related developers and also hope to lear more.

ACKNOWLEDGEMENTS

The research work is supported by Foshan Bureau of Science and Technology under projects 2016AG100311, 2016AG100652, 2016AG100792 and 2016AG100382. The author sincerely presents thanks to them all.

REFERENCES

[1]. Jianhui LI, X Wang. *Practical Guides on Programming with Big Number Library in Scientific Researches*, The International Journal of Engineering and Science,2016,5(9):64-66
[2]. S S Skiena. *The Algorithm Design Manual*. Springer London, 2008.
[3]. Aho A V , Hopcroft J E. *The design and analysis of computer algorithms*. China Machinery Press, 2006.
[4]. Xingbo WANG. *Genetic Traits of Odd Numbers With Applications in Factorization of Integers* [J]. Global Journal of Pure and Applied Mathematics,2017,13(2): 493-517
[5]. RBarry Young.*Spiraling number in Array - Stack Overflow with large numbers*, http://stackoverflow.com/ questions/11417966/spiraling-number-in-array-stack-overflow-with-large-numbers
[6]. Zhu Z Y, Zhu C. *Non-recursive Implementation of Recursive Algorithm*, Mini-micro Systems, 2003, 24(3): 567-570
[7]. Gao Y, Guan F. *Explore a New Way to Convert a Recursion Algorithm into a Non-recursion Algorithm*, International Federation for Information Processing, 2007, 258:187-193.

Author's Biography

Dongbo Fu is a lecturer of Guangdong Neusoft Institute. He obtained his master degree in Beijing University of Posts and Telecommunications in 2008, and since then has been a staff in charge of affairs of computer network in the university. He is skill at computer programming, network development and software engineering.