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

Dongbo FU

Department of Computer Science, Guangdong Neusoft Institute 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.

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.

```cpp
//============= 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 accompanied 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 \( 3 \leq p < q \) will have its divisor \( p \) to be found in the interval \( \left[ N^{\ast}_{\left( m+1\right), \left( m \right)} , N^{\ast}_{\left( m+1\right), \left( m \right)} \right] \), where \( N^{\ast}_{\left( m+1\right), \left( m \right)} = 2^{m}N - 1 \).

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.

```cpp
//========= 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

\[
\left[ \frac{\sqrt{N} + 1}{2} \right] = 530065536
\]

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 \( \left[ l_l, l_r \right] \). Suppose \( \left[ l_l, l_r \right] \) contains \( N_{m} \) integers. We first subdivide \( \left[ l_l, l_r \right] \) into \( 2m + 1 \) subintervals by

\[
N_{m} = l_r - l_l + 1
\]

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

Let \( M = \left\lfloor \frac{N}{2m} \right\rfloor \). Then there are \( 2m \) equal-length subintervals in each of which contains \( M \) integers and there is one subinterval that contains \( M \) 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

\begin{equation}
N_{\omega} = (2m) \times \left\lfloor \frac{N_{\omega}}{2m} \right\rfloor + N_{\omega} \mod (2m)
\end{equation}

\[ \text{Let } M = \left\lfloor \frac{N_{\omega}}{2m} \right\rfloor, M_{\omega} = N_{\omega} \mod (2m); \text{ then there are } 2m \text{ equal-length subintervals in each of which contains } M \text{ integers and there is one subinterval that contains } M \text{ integers.} \]

\[ \text{Conventionally, a divide-conquer algorithm search can be applied on each of the } 2m + 1 \text{ subintervals; however, as stated above, it is better to design a non-recursive algorithm for big number operations. An appreciative sample is shown below} \]

---

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 \( \left\lfloor \sqrt{N + 1} \right\rfloor / 2 \) searches. And an algorithm is proposed below.

\[ \text{Squeeze Searching Algorithm} \]

\[ \text{Input: Odd composite number } N. \]

\[ \text{Step 1. Calculate searching level: } K = \left\lfloor \log_2 N \right\rfloor - 1; \]

\[ \text{Step 2. Calculate the largest searching steps: } l_{\max} = \left\lfloor \frac{\sqrt{N + 1}}{2} \right\rfloor; \]

\[ \text{Step 3. Calculate variables: } \\
ul = 2^K N - 1; \quad ll = ul - 2l_{\max}; \]

\[ m1 = ll + l_{\max} / 2; \quad \text{left} = m1 - 2; \quad \text{right} = m1 + 2; \]

\[ \text{Step 4. If } \text{FindGCD}(N, ll) \text{ or FindGCD}(N, ul) \]

\[ \text{or FindGCD}(N, m1) \text{ return GCD;} \]

\[ \text{Else Begin loop } \]

\[ ul = ul - 2; \quad ll = ll + 2; \quad \text{left} = \text{left} - 2; \quad \text{right} = \text{right} + 2; \]

\[ \text{If } \text{FindGCD}(N, ll) \text{ or FindGCD}(N, ul) \]

\[ \text{or FindGCD}(N, left) \text{ or FindGCD}(N, right) \]

\[ \text{return GCD;} \]

\[ \text{End loop} \]

---

5.2 C++ Program to Realize the Algorithm

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

```cpp
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;
}
```

DOI: 10.9790/1813-0604020106 www.theijes.com Page 3
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);
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_add_ui(right,ml,2);
//to right half-interval
mpz_sub_ui(left,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);
if(mpz_odd_p(tmp1)==0)
    //ensure the mid-point is odd
    mpz_add_ui(tmp1,tmp1,1);
mpz_gcd(Rt1,Root,tmp1);
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_add_ui(right,tmp1,2);
//to right half of the left half-interval
mpz_add_ui(left,tmp1,2);
//to left half of the left half-interval
mpz_sub_ui(right,tmp1,2);
//length of the left half-interval
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);
 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(left,tmp1,2);
//to left half of the left half-interval
mpz_add_ui(right,tmp1,2);
//to right half of the left half-interval
while(1)
    { mpz_add_ui(steps,steps,1);  //increment of the counter

mpz_gcd(Rt1,Root,left);
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,root);
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)
    //check the end
    { mpz_divexact(Rt2,Root,Rt1);
        return; }

mpz_gcd(Rt1,Root,right);
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)
    //check the end of the right half
    { mpz_divexact(Rt2,Root,Rt1);
        return; }

mpz_gcd(Rt1,Root,left);
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)
    //check left-end of the left half
    { mpz_divexact(Rt2,Root,Rt1);
        return; }

mpz_gcd(Rt1,Root,right);
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)
    //check right-end of the left half
    { mpz_divexact(Rt2,Root,Rt1);
        return; }

mpz_gcd(Rt1,Root,root);
cmp=mpz_cmp_ui(Rt1,1);
if(cmp>0)
    //check the right-end of the left-half
    { mpz_divexact(Rt2,Root,Rt1);
        return; }
return; }
mpz_gcd(Rt1,Root,lft); //check lft

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

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

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

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-

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_add_ui(lft,lft,2);
mpz_add_ui(right,right,2);

mpz_sub_ui(lft,lft,2);
mpz_add_ui(rht,rht,2);
mpz_add_ui(llft,llft,2);
mpz_add_ui(rlft,rlft,2);
mpz_add_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. Table 1 shows the experimental results, which are made on a PC with an Intel Xeon E5450 CPU and 4GB memory.

<table>
<thead>
<tr>
<th>N</th>
<th>Factorization</th>
<th>Searching Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>1123877887715932507</td>
<td>3756830131</td>
</tr>
<tr>
<td>N2</td>
<td>11293671024866881</td>
<td>25869889</td>
</tr>
<tr>
<td>N3</td>
<td>2974231569940674837</td>
<td>32173423 79915205819</td>
</tr>
<tr>
<td>N4</td>
<td>35249679311984883</td>
<td>593138501 596052983</td>
</tr>
<tr>
<td>N5</td>
<td>20812765573409353</td>
<td>430470919 483488309</td>
</tr>
<tr>
<td>N6</td>
<td>33143253770015787</td>
<td>114098219 2904800273</td>
</tr>
<tr>
<td>N7</td>
<td>30728250405021789</td>
<td>1436222173 2137748909</td>
</tr>
<tr>
<td>N8</td>
<td>37575867226077891</td>
<td>160531273 234069700393</td>
</tr>
<tr>
<td>N9</td>
<td>249288169989046648879</td>
<td>3479132923 716524600573</td>
</tr>
<tr>
<td>N10</td>
<td>1018833756435517819</td>
<td>70901851 143696355169</td>
</tr>
</tbody>
</table>

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 learn 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


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.