Design of Parallel Algorithms for Super Long Integer Operation Based on Multi-core CPUs

Shifeng Zhang [1]
[1] College of Information Engineering, Yangzhou University, Yang Zhou, China
yjbqzsf@163.com

Shenghui Su [2, 1]
[2] Laboratory of Trusted Computing, Beijing University of Technology, Beijing 100124, PRC
reesse@126.com

***Abstract*— In cryptographic applications, super long integer operations are often used. However, cryptographic algorithms generally run on a computer with a single-core CPU, and the related computing process is a type of serial execution. In this paper, we investigate how to parallelize the operations of super long integers in multi-core computer environment. The significance of this study lies in that along with the promotion of multi-core computing devices, and the enhancement of multi-core computing ability, we need to make the basic arithmetic of super long integers run in paralleling, which means blocking super long integers, running all data blocks on multi-core threads respectively, converting original serial execution into multi-core parallel computation, and storing multi-thread results after formatting them. According to experiments we have observed: if scheduling thread time is longer than computation, parallel algorithms execute faster; on the contrary, serial algorithms are better. On the whole, parallel algorithms can utilize the computing ability of multi-core hardware more efficiently.**

Keywords - super long integers, multi-core, parallel computation, algorithms, multi-thread

I. INTRODUCTION

In public key cryptosystems [1] and digital signature schemes [2], for example, RSA [3], ECC [4], and REESSE1+ [5, 6], super long integers arithmetic is a basic requirement.

There is not a uniform definition of super long integers. In general, super long integers are those integers whose lengths are larger than 64 bits, or scopes exceed what is allowed by a programming language compiler.

The basic operations discussed in this paper include addition, subtraction, multiplication, division, and conversion of number systems. We focus on addition and multiplication, for subtraction and division can be implemented by addition, multiplication and shift [7]. At the end of this paper, we will introduce conversion of number systems briefly.

The realization of parallel computing uses the ability of multi-core computers, decomposes the existing super long integers arithmetic process, and the decomposition uses segmented or fixed interval number of bits, uses multi-thread mechanism [8], allocates the decomposed data to corresponding thread, reduces the running time of serial operations, so that make full use of multi-core processing ability.

Our algorithms are applicable to any length of super long integers arithmetic, and have no minimum length limit. (Because of the programming language, we need to limit the maximum output length, the maximum bit we tested now is 1500.)

The following shows the meaning of symbols in these algorithms.

A denotes a super long integer of m bits, namely $a_1 \ldots a_m$; B denotes a super long integer of n bits, namely $b_1 \ldots b_n$; $\underline{A}_i$ denotes *i-th* block data of super long integer A; N denotes the number of processor(s) cores; T_x denotes *x-th* execution thread; $k = \lceil n / N \rceil$ means rounding up the result of n/N, and store in the k; $l = \lceil m / N \rceil$ means rounding up the result of m / N, and store in the l; *len* denotes the length of the super long integer, namely the number of bits; *br* denotes current borrow value; *cr* denotes current carry value. Throughout the paper, assume that $A >= B$ and $m >= n$.

II. REPRESENTATION OF SUPER LONG INTEGERS

A. Samples

In cryptography algorithms [9,10,11], super long integers cannot be represented directly by data types allowed by program language compiler, for example, 2^{1024}. And computing the product of two integers $[12345678987654321]_{10}$, $[456987123654789321]_{10}$ is difficult. Moreover, calculating the module of the product which is the power of two positive numbers, like $1024^{100} * 256^{2048} \% 2^{1024}$, is difficult. So we propose a new method to achieve the goal directly under the existing computer data type.

B. Data Storage and Representation

In the paper, we denote structure of super long integers by C Program Language.

For the convenience of maintenance and expansion, we predefine variables as follow.

```
typedef unsigned int un_int;
typedef unsigned long un_long;
```

Identifier *HBigInt* represents variable defined by super long integer structure:

978-1-4673-8660-9/15 $31.00 © 2015 IEEE
DOI 10.1109/CIS.2015.88

```
typedef struct {
    long length;
    int sign;
    un_int *pBigInt;
} HBigInt;
```

length indicates the size of the assigned storage space, *pBigInt* points to array which keep the data of a super long integer, *sign* is used to distinguish between positive or negative: 1 denotes non-negative; -1 denotes negative; 0 is initialized, not been assigned yet.

Especially, each "bit" of super long integers is stored in index-bit of an integer array.

We store the super long integer number following Big-Endian, which means store the most significant bit in the smallest address. This way facilitates programming dynamic expansion and reduces the length of that.

III. DESIGN OF PARALLEL ALGORITHMS

A. Description of Addition Parallel Algorithm

The main idea of the addition parallel algorithm is to divide super long integers into pieces, then handles the addition of corresponding block to a new open thread, as the thread is allocated to one processor of multi-core processor by operating system, These multi date blocks compute simultaneously, saving overall computing time ultimately.

Input: $A = a_1 \ldots a_m$, $B = b_1 \ldots b_n$, Where $m, n>1$

Output: C

S1: Get N from host.

S2: If $N < n$, then exit.

S3: B will be divided into N blocks, each block k bits, that means $\underline{B}_i = b_i b_{i+1} \ldots b_{i*k-1}$, and $\underline{A}_i = a_i a_{i+1} \ldots a_{i*k-1}$, the remaining bits non-blocking.

S4: Multithreads T run data block, that T_x: $\underline{C}_i = \underline{B}_i + \underline{A}_i$.

S5: Make that $c_{m-n\ldots}c_m = a_{m-n\ldots}a_m$, then pick up carry from the low to high so as to merge the results until all threads finish computation.

S6: If the most significant bit carry, namely c_{m-1} is not less than the decimal value, then increase the length value of the super long integer.

S7: Format the result by $c_m = c_{m-1} / RADIX$, $c_{m-1} = c_{m-1} \% RADIX$, final result is C and $len = len + 1$; Otherwise the value of *len* is unchanged.
RADIX denotes custom radix.

B. Description of Subtraction Parallel Algorithm

It should be noted that, due to the result of subtraction may be negative, we need to convert the negative bit. In order to reduce this kind of process, we compare two unsigned super long integers before operating, and determine the final results of the positive or negative, then get larger as the minuend.

Input: $A = a_1 \ldots a_m$, $B = b_1 \ldots b_n$, Where $m, n>1$

Output: C

S1: Get N from host.

S2: If $N < n$, then exit.

S3: B will be divided into N blocks, each k bits, namely $\underline{B}_i = b_i b_{i+1} \ldots b_{i*k-1}$; and $\underline{A}_i = a_i a_{i+1} \ldots a_{i*k-1}$, the remaining bits non-blocking.

S4: Multithreads T run data block, that T_x: $\underline{C}_i = \underline{A}_i - \underline{B}_i$.

S5: Make that $c_{m-n\ldots}c_m = a_{m-n\ldots}a_m$, then handle borrow from the low to high in order to merge the results.

S6: If the most significant bit borrow, when the result of $c_{m-1} = c_{m-1} - br$ is zero, reduce the bits of the result.

S7: The ultimate difference is C, $len = len - 1$; Otherwise, the value of *len* is unchange.

C. Description of Multiplication Parallel Algorithm

The sign of the result of multiplication is not affected by the numerical size of two operands but its initial sign, so we can know the sign of the result before the operation.

Input: $A = a_1 \ldots a_m$, $B = b_1 \ldots b_n$, Where $m, n>1$

Output: C

S1: Get N from host.

S2: If $N < n$, then exit.

S3: Divide B into N blocks, each block of k bits, and divide A into N blocks, each block of l bits, namely $\underline{B}_j = b_i b_{i+1} \ldots b_{i*k-1}$, $\underline{A}_j = a_j a_{j+1} \ldots a_{j*l-1}$.

S4: Multithreads T run data block, that T_x: $\underline{C}_{i+j} = \underline{A}_j * \underline{B}_i$.

S5: Until all threads finish computing, pick up carry from the low to high in order to merge the results.

S6: When the most significant bit of a carry, namely c_{m*n-1} is not less than the decimal value, then increase the length value of the super long integer.

S7: Format the result by $c_{m+n-1} = c_{m+n-1} \% RADIX$, $c_{m+n} = c_{m+n-1} / RADIX$, final result is C and $len = m + n$; Otherwise $len = m + n - 1$.

D. Description of Division Parallel Algorithm

Due to the special nature of the division of super long integers (discard the fractional part), we use multiplication and shift to finish the division. To ensure the feasibility of parallel algorithms, we ignore the loss of precision generated by the displacement.

Since turning divisor into multiplier is not the focus of this paper, it is the key that makes the division in parallel, so we will give an example of the transformation processes in the concluding remarks.

IV. EXAMPLES

Because of the super long integer data block division, the segmented data blocks will be handled by independent running threads, and the results of all the threads are centralized treated finally.

A. Choose Radix

Decimal storage and calculation cannot meet the ability of the 32-bit PC, for 32-bit machine, the computing ability in a clock cycle is 2^{16} size level (taking into account the value of the multiplication operation number contain two numbers,

$(2^{16} - 1) * (2^{16} - 1)$ will not cause the overflow of the data results), thus choose 2^{16} as radix.

B. Resolve Particle

Due to the schedule of the operating system, it will cost time to generate threads or wait for all child threads ending, and such scheduling time shows a sharp growth in critical value with the increase of the number of child threads, the critical value of the sub-thread 1.5 times the number of CPU cores. Therefore, the more detailed division of the data does not mean better, but the more adaptation the better and the principle is that the current number of threads are equal to the number of CPU cores. The number of cores can be fetched when the software is running, thence decomposition of data is allocated dynamically.

C. Parallel Program

According to the choice of radix and particle size, decompose the integration of super long integer arithmetic, there are two alternative ways. One is dividing segment corresponds to a thread for processing, the size of the segment depends on the actual number of threads (i.e. the number of CPU cores [12]); the other is the interval bit corresponds to a thread to process; the size of the interval bit is the number of threads (i.e. the number of CPU cores). The former puts a number of consecutive bits into a block, the later disperses each bits into mesh. Both of them are basically the same in the load balance, for using the average approach [13]. The former has one more step than calculation, calculating size of "chunks", while the latter has advantages than former in accessing memory continuity.

D. Examples of Addition Parallel Computing

Calculate [123456789 + 345879] (dual-core for example):

123456789 is decomposed into 123 456 789, 123 is "extra" high, is not involved in operations, directly assigned to the provisional results; and 345879 are decomposed into 345 and 879

Thread 1 computing:

```
    4  5  6
+   3  4  5
    7  9  11
```

Thread 2 computing:

```
    7  8  9
+   8  7  9
   15  15 18
```

Temporary results, namely

1 2 3 7 9 11 15 15 18

Then traverse from low to high, and handle the bit which has carry data (carrying process is abbreviated), namely

1 2 3 8 0 2 6 6 8

Theoretically save nearly half computing time, because the thread 1 and 2 are performed simultaneously. By extension, when using a 4-core, 8-core, computation time will be the original serial 1/4 and 1/8.

Subtraction is similar to addition, the only difference is that intermediate results generated by addition may be greater than the radix (for example greater than 10 when radix is decimal), and the subtraction may produce a negative. Calculate [123456789 - 345879], intermediate result is:

1 2 3 1 1 1 -1 1 0

Then traverse from low to high, and handle the bit which has borrow data (borrowing process is abbreviated), namely

1 2 3 1 1 0 9 1 0

E. Examples of Multiplication Parallel Computing

Calculate [189 * 34] (dual-core example):

189 is decomposed into 1, 8 and 9, while 34 is decomposed into 3 and 4

Thread 1 computing:

```
    1  8  9
*      4
    7  5  6
```

Thread 2 computing:

```
    1  8  9
*      3
5   6  7
```

Temporary results, namely

```
       7  5  6
+  5   6  7
   5   13 12 6
```

carry bit 6 4 2 6

Finally, traverse from the low to the high, and handle the bit which has carry data. But there is a difference compared with addition operations, the generation of intermediate results have dependency relationship, since the overlap of portion of the data bits (columns 2 and 3 in thread 1 corresponding to columns 1 and 2 in thread 2 respectively), we need to do some exclusive treatment during the cumulative, you can also use an intermediate variable to circumvent this exclusive phenomenon. So this operation is also saving half time of computing the bits of theoretical. By extension, when using a 4-core, 8-core, computation time will be 1/4 and 1/8 of the original serial.

In summary, the parallel algorithms are feasible and practical.

V. Efficiency Analysis

A. Experimental Data

The "bits" column in the table below shows the number of bits involved in computing the super long integers.

For example, [12345678909876543210] represents the "bits" of 20-bit integer.

The main experimental source code can reference the appendix.

TABLE I. The execution time of addition

bits/(ms)	Serial	Dual-Core	Quad-Core

70	0.003352	0.360660	0.375574
150	0.004531	0.378718	0.383751
300	0.008095	0.382632	0.391063
600	0.015419	0.395561	0.397107
1500	0.038750	0.413783	0.406129

TABLE II. THE EXECUTION TIME OF MULTIPLICATION

bits/(ms)	Serial	Dual-Core	Quad-Core
70	0.123479	0.470730	0.495437
150	1.285638	0.992584	0.913955
300	4.692521	3.376838	2.075612
600	13.785123	5.938657	4.156538
1500	30.618759	11.798465	9.369179

- Dual-Core PC configuration: Windows XP + Intel T2250 1.73GH + 1GB
- Quad-Core PC configuration: Windows XP + Intel i3-3240 3.40GH + 2GB

B. Result Analysis

From the time of parallel and serial computation above, we can see that, with the increasing of bits of super long integer, the time of serial execution grow linearly. However, The time of parallel execution is less than that of serial execution, while The time of parallel execution is less than the increment of linear growth. In addition for example, compared with the serial execution approximately 100-fold in 70 bits parallel execution reduced to 10-fold in 1500 bits; then take multiplication for example, compared with the serial execution approximately 4 times in 70 bits parallel execution reduced to 1/3 times in 1500 bits.

Analysis time-consuming of algorithms:

The execution time of "one-bit" calculation is A, the cycle execution time is C, the rest definition of variables and formatting functions are essentially changeless, so the total time can be defined as D, then the "n-bit" super long integer serial execution time can be expressed as:

$$T_{serial} = (A + C) * n + D.$$

Due to "n-bit" will be decomposed, such formal expression is incomplete in the parallel environment, assume that the number of cores is N, then the actual parallel operation time is: $T_{parallel} = (A + C) * n / N + D$. If the expression is established, then why the actual time of addition implementation is even longer than the parallel computing? The reason is that ignoring a new thread generated before the parallel computing, and the time that thread is waiting to be released.

Considering the precise millisecond level (Microsecond, Nanosecond), this neglect is fatal. Because the cost of time that an operating system schedules a thread is three tenths of milliseconds or so, of course, not increase in linear growth with the increase of thread-number. In practice it shows that when the number of threads does not exceed the threshold, the scheduling time of the operating system is basically unchanged, maintaining the level of three tenths of milliseconds or so, when the scheduled time exceeds the threshold, it will increase to milliseconds level. Therefore, we need to modify the expression of parallel algorithms for the computation time: $T_{serial} = (A + C) * n + D + K$. Where K is the time that operating system scheduling thread required.

Due to different hardware environment (affected by CPU clock speed, register read and write rates, Cache size, etc.), the required time of "one-bit" operation is different, we use a uniform symbol A, and similarly, represent the cycle time by C.

Compare the required time of T_{serial} and $T_{parallel}$.

Let $S = T_{parallel} - T_{serial}$

$= [(A + C) * n](1 / N - 1) + K$

$= K - [(A + C) * n](1 - 1 / N)$

As seen from the comparison result, with the increasing of N value (i.e. CPU cores), the result approximates $K - [(A + C) * n]$ (N is large enough, for example, $N = 8$), that is the difference value between the operating system scheduler threads require and actual computing time cost. In other words: If the time of scheduling threads is longer than the computing time, the implementation of parallel algorithms will be faster; on the contrary, the executing time of the serial algorithms is faster.

From the above data conversion, the complexity of serial computation is:

$$O((A + C) * n + D) = O((A + C) * n) + O(D) \approx O(n).$$

Since $A+C$ is a constant, D is also a constant, so the time complexity of serial computing is about $O(n)$.

From the above data conversion, the complexity of muti-core parallel computation is $O((A + C) * n / N + D + K) = O((A + C) * n / N) + O(D + K) \approx O(n / N)$.

Since $A+C$ is a constant, $D+K$ is also a constant, so the time complexity of parallel computing is about $O(n / N)$.

Note: The time given in conclusions is the execution time of the costing by serial computing.

VI. CONCLUSIONS

With the enhanced hardware integration of PC and mobile clients, parallel algorithms can make best use of hardware resources constantly, and get excellent quality and reasonable price. As the time of addition showed, parallel algorithms cannot be used blindly, in super long integers (currently only authenticate to 1500 integer) bits operation, the serial algorithms still has a great advantage, even with the increase of bits gap continued narrow. And this advantage will be enhanced with the promotion of hardware capabilities, because the promotion of hardware capabilities is shortened.

As for the shift operation, it will need to convert super long integer between decimal and binary, so we give brief introduction of radix conversion. In this paper, 2^{16} is chosen as the "one-bit" radix, in the radix conversion directly from

"one-bit" store number corresponding to the sixteen bits binary number, for example:

$[8589869055]_{10} = [1\ 65534\ 65535]_{2}^{16}$

$= [1\ 1111,1111,1111,1110\ 1111,\ 1111,\ 1111,\ 1111]_2$

On the other hand the process of binary to decimal is as follows:

$[\ 1\ 1111,1111,1111,1110\ 1111,1111,1111,1111\ ]_2$

$= [1\ 65534\ 65535]_{2}^{16} = [8589869055]_{10}$

It can be seen that the storage number "every-bit" is unrelated to each other when doing a conversion, in line with data decomposition characteristics of independence, so parallel algorithms described in this article is also supported by the conversion operations.

Finally, introduce the process of division by multiplication and shift simply. Assume *a* is the dividend, *b* is the divisor, and *b* > 1. We know that *a* / *b*= (1 / *b*) * *a*, since *b* > 1, so 1 / *b* is less than 1, and can only handle integer CPU general purpose registers, so we must find a way to 1 / *b* increased to *2* ^ *e* times, is left *e* bits position, and the results obtained *e* bits in the right place to get business, namely: *a* / *b* = *((*2 ^ *e* / *b) * a)* / 2 ^ *e* , where *e* is an integer greater than 0, specifically determined by the *b*.

For example: *a* = Dividend, *b* = 0xAAAAAAAB; (i.e., *b* = (2 ^ 33 + 1) / *3*) *r* = (*a* * *b*) >> 33, *r* is the quotient of *a*/3; similarly, *b* = 0x24924925; *r* = (*a* * *b*) >> 32, *r* is the quotient of *a*/7.

ACKNOWLEDGMENT

The authors also would like to thank the Professors Shuwang Lü and Xinchun Yin for their important suggestions and corrections.

REFERENCES

[1] W. Diffie and M. E. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Thery, v22(6), 1976, pp. 644-654.

[2] T. ElGamal. A Public-key Cryptosystem and a Signature Scheme Based on Discrete Logarithms[C]. Advances in Cryptology. Springer Berlin Heidelberg, 1985, pp. 10-18.

[3] R. Rivest, A. Shamir, L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems[J]. Communications of the ACM, v2(21), 1978, pp. 120-126.

[4] I. F. Blake, G. Seroussi and N. Smart. Elliptic Curves in Cryptography[M], Cambridge University Press, Cambridge,UK, 1999.

[5] S. Su, S. Lu and J. Cai. Lightweight digital signature method an ultra-based logarithm problem of State Intellectual Property, 201110297654. 1, 2011. 10

[6] S. Su and S. Lu. A Public Key Cryptosystem Based on Three New Provable Problems. Theoretical Computer Science, v426-427, Apr. 2012, pp. 91-117.

[7] Su, S. Lü, and X. Fan. Asymptotic Granularity Reduction and Its Application. Theoretical Computer Science, vol. 412(39), 2011, pp. 5374-5386

[8] WU Jianyu, PENG Manman. Private LLC Optimization of Chip Multi-processors Oriented toMulti-threaded Application[J]. Computer Engineering, 2015, 41(1): 316-321.

[9] YongJe Choi, HoWon KIm, MooSeop Kim, YoungSoo Park, Kyoil Chung, "Design of Elliptic Curve Cryptographic Coprocessor over Binary Fields for the IC Card", ITC-CSCC, July, 2001, pp.299-302.

[10] A Kumar, N Rajpal, Application of Genetic Algorithm in the Field of Steganography, in Journal of Information Technology, Vol. 2, No. 1, Jul-Dec. 2004, pg 12-15.

[11] Sania Jawaid and Adeeba Jamal, "Generating the Best Fit Key in Cryptography using Genetic Algorithm", International Journal of Computer Applications (0975 – 8887)Volume 98 – No. 20, July 2014.

[12] Carlsson, N. and M. Arlitt, 2011. Towards more effective utilization of computer systems. SIGSOFT Softw. Eng. Notes, 36(5): 235-246.

[13] Yuping Wang, Chuangyin Dang, An Evolutionary Algorithm for Global Optimization Based on Level-Set Evolution and Latin Squares, IEEE Trans. Evolutionary Computation, 11(5), 579-595, 2007.