Efficient Multithreading Computation of Modular Exponentiation with Pre-computation of Residues for Fixed-base

Ihor Prots'ko[a], Aleksandr Gryshchuk[b], Volodymyr Riznyk[a]

[a]*Lviv Polytechnic National University, str. S.Bandery, 12, Lviv, 79013, Ukraine*
[b]*LtdC "SoftServe", str. Sadova, 2d, Lviv, 79021, Ukraine.*

Abstract
Modular exponentiation is an important operation in many applications that requires a large number of operations. Efficient computations of the modular exponentiation are extremely necessary for efficient computations for provide high crypto capability of information data and in many other applications. Modular exponentiation is implemented using of the development of the right-to-left binary exponentiation method for a fixed base with pre-computation of redused set of residuals. To efficient compute the modular exponentiation over big numbers, the property of a periodicity for the sequence of residuals of a fixed base with exponents equal to an integer power of two is used. The multithreading software implementation of modular exponentiation is described. The MPIR library with an integer data type with the number of binary digits from 256 to 2048 bits is used to develop an algorithm for computing the modular exponentiation.
Comparison of the runtimes of four variants of functions for computing the modular exponentiation is performed. In the algorithm with pre-computation of residues for fixed-base provide faster computation of modular exponentiation compared to the functions of modular exponentiation of the MPIR, OpenSSL and Crypto++ libraries.

Keywords 1
Modular exponentiation, parallel computation, multithreading, big numbers.

1. Introduction

Computing the modular exponentiation for big numbers is widely used to find the discrete logarithm, in number-theoretic transforms, and in cryptographic algorithms. New methods, algorithms and means of their implementation are being researched for efficient computation of the modular exponentiation. There are three directions of modular exponentiation methods: general modular exponentiation, and computation of a modular exponentiation with a fixed exponent or with a fixed base [1]. Special functions have been developed to perform modular exponentiation in mathematical and cryptographic software libraries (Crypto++, OpenSSL, Pari/GP).

Modular exponentiation and discrete logarithm for big numbers require significant computing resources for their implementation. The discrete logarithm is considered a unidirectional function (1), because it is quite difficult to calculate it in a conventionally acceptable time, for example, for breaking a cryptographic code.

The Sixth International Workshop on Computer Modeling and Intelligent Systems (CMIS-2023), May 3, 2023, Zaporizhzhia, Ukraine
EMAIL: ihor.o.protsko@lpnu.ua (I. Prots'ko); ocr@ukr.net (O.Gryshchuk); volodymyr.v.riznyk@lpnu.ua (V. Riznyk).
ORCID: 0000-0002-3514-9265 (I. Prots'ko); 0000-0001-8744-4242 (O.Gryshchuk); 0000-0002-3880-4595 (V. Riznyk).

The discrete logarithm problem [2] is formulated as follows: for known integers *A*, *N*, *y*, find an integer *x* such that

$$x = log_A y,\ (0 \le x \le N - 1)\,, \tag{1}$$

where $(A, N)=1$; $A, N, y, x \in Z$).

The number $x > 0$ is called the discrete logarithm of the number y with base *A* and module *N* according to formula (1).

The solution of the discrete logarithm problem can be the solution of the equation

$$A^x \bmod N = y\,. \tag{2}$$

That is, having determined the number *x,* which is the solution of equation (2), we will find the discrete logarithm. Thus, the problem of the discrete logarithm is reduced to the calculation of the modular exponentiation in the form (2).

The paper considers the basic iterative algorithm for computing the modular exponentiation (2) using pre-computation to form a shortened sequence of residues of the fixed base *A*. The software implementation of multi-threaded computation of the modular exponentiation for big numbers is described. In the discussion section of the results, a comparison of the average executing time of computing the modular exponentiation with the primitives of cryptographic libraries (OpenSSL, Crypto++) is made. The conclusions summarize the possibility of applying the computation of the modular exponentiation (2) using the pre-computation of the remainders of the fixed base *A*.

2. Related works

Modular exponentiation, A^x mod *N*, is a basic arithmetic operation in most public-key cryptosystems. Many effective methods of modular exponentiation have been developed and are often used to reduce the execution time of the modular exponentiation operation, which are reviewed in works [3-6]. In paper [3] is described well-known exponentiation algorithms with their complexity analysis and general exponentiation algorithm of zero-one sequences.

There are three types of exponentiation algorithms A^x mod *N* [1], which include:

1) basic techniques for exponentiation;

2) fixed-exponent *x* exponentiation algorithms;

3) fixed-base *A* exponentiation algorithms.

Among the main modular exponentiation algorithms, the following effective solutions are distinguished:

- right-to-left *k*-ary exponentiation,
- right-to-left *k*-ary exponentiation,
- left-to-right *k*-ary exponentiation,
- exponent with a sliding window (sliding window exponentiation),
- Montgomery ladder,
- simultaneous multiple exponentiation and their modifications.

In many works, the method of binary exponentiation with a right-to-left shift is used. Cohen's book [7] presents a more complete discussion of the binary right-to-left and left-to-right methods together with their generalizations to the *k*-binary method.

A fixed element of a group (generally z/q_z) is repeatedly raised to many different exponent in several cryptographic systems. A popular application of fixed-base exponentiation is in elliptic curve cryptography, for instance for Diffie-Hellman key agreement and elliptic curve digital signature algorithm verification. Therefore, many research works have been focused on a fixed base of modular exponentiation [1, 8].

Considerable attention is paid to the hardware implementation aimed at efficient computation of the modular exponentiation. One of the ways to speed up the computation of modular exponentiation is the parallelization of calculations using modern technologies in universal computer systems [9, 10] or the creation of specialized computing tools [11].

The modern software libraries are used to implement the computation of modular exponentiation. The software implementation of the modular exponentiation computation is included in the software libraries Crypto++ and OpenSSL, PARI/GP, MPIR designed for working with big numbers [12-15].

For example, the Pari/GP software library [12] contains a large set of programs for efficient computations of mathematical functions. The Pari/GP library also includes computation of the modular exponentiation function for big numbers and other special numbers. A highly optimized modification of the well-known GMP or GNU Multiple Precision Arithmetic Library the MPIR library [13] contains the function of the realization the computation of modular exponentiation. The library of cryptographic algorithms and schemes Crypto++ is implemented in C++ and fully supports 32 and 64-bit architectures of many operating systems and platforms [14]. The library contains a set of available primitives for theoretical and numerical operations, such as generation and verification of prime numbers, arithmetic over a finite field, operations on polynomials.

The importance of speeding up the software calculation of modular exponentiation leads to new algorithmic solutions and their implementations [16].

3. The technique of the computation of modular exponentiation with pre-computation of residues for fixed-base

The parallelization of computation of modular exponentiation, based on the use of the exponent x as a binary number ($x_{(k-1)}\, x_{(k-2)} \dots\, x_2\, x_1\, x_0$), uses the application of the fundamental property of modularity was described in paper [9]. Accordingly, the computation of the modular exponentiation (1) takes the form

$$
\begin{aligned}
y &= A^{x_{(k-1)} x_{(k-2)} \cdots x_2 x_1 x_0} \bmod N = \\
&= (A^{x_{(k-1)} 2^{k-1}} \bmod N * A^{x_{(k-2)} 2^{k-2}} \bmod N * \dots \\
&* A^{x_2 2^2} \bmod N * A^{x_1 2^1} \bmod N * A^{x_0 2^0} \bmod N) \bmod N ;
\end{aligned}
\tag{3}
$$

In accordance with formula (3), a scheme for computing of the modular exponentiation [17] is shown on Figure 1.

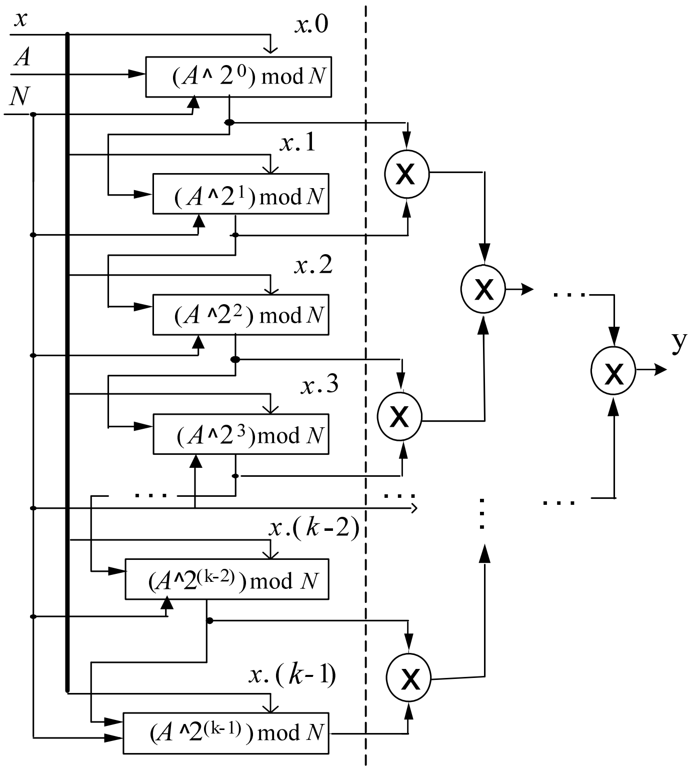


Figure 1: The scheme for computation of modular exponentiation A^x mod N.

The scheme for computing the modular exponentiation consists of the denotations:

- A is the input of the base number; N is the input of the module;
- (A^2^i) mod N are blocks of computation of the integer exponent of exponent 2^i of the number A by the module, $i = 0,1,2,\ldots,(k-1)$;
- x is the input of an exponent with binary digits $x.(k-1), x.(k-2),\ldots,x.2$, x.1, $x.0$;
- (X) mod N is the block of multiplier under modulo N;
- y is the output of the modular exponentiation.

Consistently determined residuals of the number a raised to the power of 2^i under modulo N in blocks (A^2^i) mod N, for i=0, 1,…, k-1 are multiplied. In the case of presence of the exponent of the number of binary the value of zero in i-th binary digit ($x.i$) block (A^2^i) mod N is not execute the operation, otherwise is computed the multiplication of exponent 2^i of the number A under modulo N, which corresponds to the denotation in Figure 1. The determined product, in the final stage, are formed in blocks (X) mod N the value y of the result of the computation of the modular exponentiation.

For the organization of modular exponentiation execution, in accordance with Figure 1, it is possible to consider the possibility of creating two threads. The thread I computes the numbers a raised to the power of 2^i under modulo N. The thread II computes the product of the results (A^2^i) mod N, starting with the readiness of the first two values (A^2^i)mod N. Theoretical it is possible to reduce the computation time to 50% of the modular exponentiation in comparison with single thread computation. However, serial or parallel computations (A^2^i) mod N for big data create a significant delay in the thread I execution [9].

Consider the basic iterative algorithm for computing the modular exponentiation (3) using pre-computation to form a shortened sequence of residues of the fixed base A. Compute, respectively (3), the value modulo N for a simple fixed-base A with exponents $x = 2^i = 1, 2, 4, 8, 16\ldots$, ($i$ =0,1,2,…, r-1).

Let A and N be relatively prime positive integers (A, N) = 1 and denote the least positive integer $x = \exp_N A$. Accordance, if A and N relatively prime (A, N) = 1, positive integer $\boldsymbol{x}$ is solution of the congruence, if and only if

$$x = q \cdot \exp_N A, \tag{4}$$

Accordance the Euler's theorem, if A and N relatively prime (A, N) = 1, that $A^{\varphi(N)} \equiv 1$ (mod N). Consequently, we can do conclusion

$$\varphi(N) = q \cdot \exp_N A, \tag{5}$$

In case q=1, then $\varphi(N) = \exp_N R$, where R is the positive integer is called a primitive root modulo N. However the positive integer of modulo N, possesses a primitive root R if only if N=2, 4, P^k or $2P^k$, where k is positive integer. The primitive root for modulo $N = P_1^{k1} P_2^{k2} \ldots P_m^{km}$ does not have, except if $\varphi(P_1^{k1})$, $\varphi(P_2^{k2})$,…, $\varphi(P_m^{m1})$ are relatively prime.

Thus, calculating $(R)^i$ mod N (i = 0,1, 2 ,,… N-1), we form a sequence of residuals ($r_0, r_1, r_2, \ldots, r_{i,\ldots}, r_{N-1}$), which periodically repeated for $x > (N-1)$ exponents. For all values of $A \in Zp$, the sequence A^i mod P is cyclic for a non-primitive element.

For example, for a primitive element R = 7, the sequence of residual values $r_i = (7^i)$ mod 11,

$(r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9) = (1, 7, 5, 2, 3, 10, 4, 6, 9, 8)$.

The maximum period of repetitions is equal to $\exp_{11}7 = 10$, because 7^{10} mod 11=1, i =0, 1, 2, …, 9. The unique integer x with $1 \le x \le \varphi(N)$ and R^x mod $N \equiv A$ is called index (or discrete logarithm) $\text{ind}_R A$ of A to base R modulo N. Then the sequence of the indices is equal

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9) = ($\text{ind}_7 1$, $\text{ind}_7 7$, $\text{ind}_7 5$, $\text{ind}_7 2$, $\text{ind}_7 3$, $\text{ind}_7 10$, $\text{ind}_7 4$, $\text{ind}_7 6$, $\text{ind}_7 9$, $\text{ind}_7 8$).

Accordingly of the property of indeces

$\text{ind}_7\, 1 = 0$,

$\text{ind}_7\, 6 = \text{ind}_7\,(2*3) = \text{ind}_7\,(2) + \text{ind}_7\, 3 \bmod 10 = 3+4=7$;

$\text{ind}_7\, 9 = \text{ind}_7\,(3^2) = 2 * \text{ind}_7\, 3 \bmod 10 = 2*4=8$.

In the case of calculating 7^x mod 11 with index x = 32, the index will be equal to (32 mod ($\text{ind}_{11}7$)) = 2, and accordingly $\text{ind}_7 5$. In the case of determining ($7^{2\,^\wedge\,6}$) mod 11, we find the number of the residue in the sequence with the index $\text{ind}_7 3$, which is equal to (2^6) mod 10 = 4. After all, the value of the modular exponentiation for 2 elements in the sequence of residual values $r_4 = 3 = (7^{2\,^\wedge\,6})$ mod 11. For computations according to formula (3), we determine the residuals for exponents 2^i, (i =2, 3, 4,

…). As a result of computations r_i = ($7^{2\ ^\wedge\ i}$) mod 11, (i =2, 3, 4, …) we obtain the values of the residuals given in Table 1. For another primitive element R =13, the sequence of residual values r_i = (13^i) mod 17 for exponents 2^i, (i =2, 3, 4, …) are given in Table 1.

Table 1

Periodic repetition of residual values 7 ^ 2^i mod 11

7 ^ 2^i	7^0	$7^{2\wedge 0}$	$7^{2\wedge 1}$	$7^{2\wedge 2}$	$7^{2\wedge 3}$	$7^{2\wedge 4}$	$7^{2\wedge 5}$	$7^{2\wedge 6}$	$7^{2\wedge 7}$	$7^{2\wedge 8}$	$7^{2\wedge 9}$	$7^{2\wedge 10}$	…
T' =4	1	7	**5**	**3**	**9**	**4**	5	3	9	4	**5**	**3**	…

Periodic repetition of residual values 13 ^ 2^i mod 17

13 ^ 2^i	13^0	13^1	13^2	13^4	13^8	13^{16}	13^{32}	13^{64}	13^{128}	13^{256}	13^{512}	…
T' =1	1	13	16	**1**	1	1	1	1	1	1	1	…

That is, in the process of computing ($7^\wedge 2^i$) mod 11, starting with the exponent 2^1 = 2, we obtain periodic repetition of the values of the residuals $r_0, r_1, \boldsymbol{r_2}, \boldsymbol{r_4}, \boldsymbol{r_8}, \boldsymbol{r_{16}}, r_2, r_4, r_8, r_{16},\ldots$ with period T'= 4 and offset u = 0, because i =2^0 = 1.

That is, in the process of computing ($13^\wedge 2^i$) mod 17, starting with the exponent 2^2= 4, we obtain periodic repetition of the values of the residuals $r_0, r_1, r_2, \boldsymbol{r_4}, r_4, r_4, r_4, r_4, r_4, r_4,\ldots$ with period T'= 1 and offset u = 2, because i = 2^2 = 4.

The value $A^\wedge 2^i$ mod N is found by the condition

$$A^\wedge 2^i \bmod N \equiv A^\wedge 2^{(i-t\cdot T')+u)} \bmod N, i > u + T' \ , \qquad (6)$$

where t={i /T'} is integer part from division.

Therefore, for a fixed-base A of the modular exponentiation (3), which is equal to the product of the residuals of the exponent ($A^\wedge 2^i$) mod N, (i = 2, 3, 4, …), you can speed up the process of computing the modular exponentiation by pre-computing (Figure 2) the sequence of residuals, what repetitions with the period T' after the offset u.

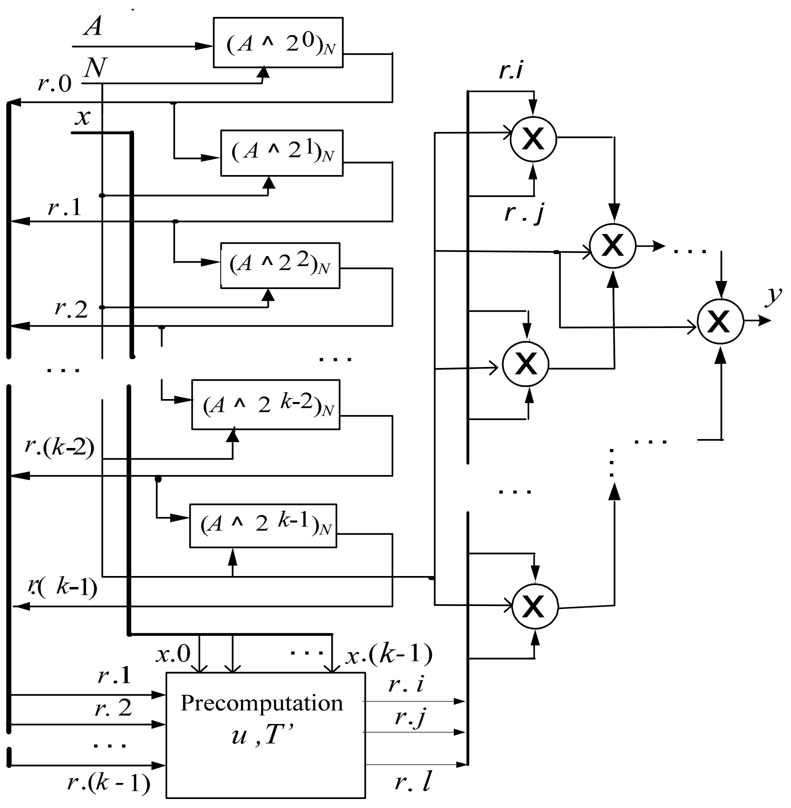


Figure 2: The scheme for computation of modular exponentiation A^x mod N with pre-computation.

Thus, on base to the parallel execution of the computation of the modular exponentiation with pre-computation, in accordance with Figure 2, consider the possibility of creating the threads of execution of the product of the residual values under modulo defined in the parallel previous threads residuals of the exponents (A^2^i) mod N, (i = 2, 3, 4, …) under modulo operations. The only difficulty of organizing the computation with such threads is the need to synchronize the performance of threads to ensure the correct computation of the final value y of the modular exponentiation.

4. Software implementation of the computation of modular exponentiation with pre-computation of residues for fixed-base

The algorithm is implemented in C++ language as modular exponentiation function *precompute_parallel*() with aim to compare the performance execution. To implement the algorithm, the library functions mpz_init_set (*mul, base*), mpz_sizeinbase (*exp,* 2), mpz_tstbit (*exp, i*), mpz_mul (*r, r, mul*) from the MPIR library are used, the parameters of which are multi-bit data up to 2048 bits. The software implementation consist the functions:

precompute_parallel (*const RemaindersData&data, const mpz_class&exp, ctpl::thread_pool&pool*);
parallel (*const mpz_class& base, const mpz_class& exp, const mpz_class& mod*);
precompute (*const RemaindersData& data, const mpz_class& exp*);
find_remainders (*const mpz_class& base, const mpz_class& mod, size_t max_exp_bits*);
find_period (*const std::vector<mpz_class>& remainders*);
get_remainder (*const RemaindersData& data, size_t power*);
update_remainders (*RemaindersData& data*);
thread_function(*bool* quit_thread*);
multiplication_thread (*int id, MultiplicationThreadData* data*) and others.

The functions find_period (*const std::vector<mpz_class>& remainders*) computes the products modulo over the pre-computed values of the residuals (A ^2^i) mod N, which are read using the function get_remainder (*const RemaindersData & data, size_t power*). In the cycle of the function mpz_tstbit (*exp, i*) binary bits *x.i* of exponent *exp* are analyzed to determine to perform or not a multiplication operation modulo, accordance the Figure 3 in the pre-computation unit. The computation of the value of the modular exponentiation ends by writing the result in the variable *period_mod_exp_result.*

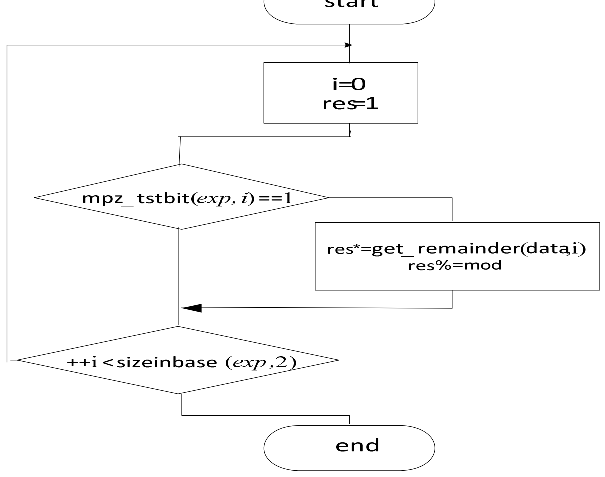


Figure 3: The chart of the algorithm for determining to perform or not a multiplication under modulo in the function *find_period* () to compute the value of the modular exponentiation.

The precompute_parallel() function finds the sequence of residuals for fixed numbers *Base* and *mod* for $Exp = 2^i$ (i = 0, 1, 2, …) and analysis of periodicity. In the program for computing the sequence of residuals is performed by the function find_remainders (*const mpz_class & base, const mpz_class & mod, size_t max_exp_bits*), which contains the bool function find_period (*const std::vector<mpz_class> & remainders*) to set the indication of finding the period. The precomputation have been made in a separate function find_remainders () to optimize multiple residual searches (A ^2^i) mod N. The function update_remainders (*RemaindersData & data*), shortens the length of the sequence of residuals to the end of the first periodicity. This function writes the offset *period_offset* beginning of the period and the length of the period *period_size* in the corresponding fields of the structure

```
RemaindersData
{ mpz_class base;
  mpz_class mod;
  std::vector <mpz_class> remainders;
  size_t period_offset;
  size_t period_size;
} also.
```

The peculiarity of the large values of *Base*, *mod* and *Exp* is also taken into account for which the residuals must be calculated, in case when the value of the period T' is many orders of magnitude greater than the number of bits of the *Exp* exponent.

To organize efficient multithreading computation of modular exponentiation according the precompute_parallel() function, the thread_function(*bool* quit_thread*) and parallel (*const mpz_class& base, const mpz_class& exp, const mpz_class& mod*) are used. Accordingly, these threads are created to perform the computation of the modular exponentiation. The result of the function is written to the variable *s_thread_result*, and the computation time is fixed and averaged to output.

To protect the queue from simultaneous access of the threads is used the *s_mutex* mutex with the basic *lock*() and *unlock*() methods. The *s_mutex* object is passed to our functions, which should use it for interacting. Before accessing the shared data queue *s_thread_queue*, the mutex is locked by the method *lock*(*s_mutex*) and is unlocked after the work with the shared data is completed. The use of a combination a conditional variable and a mutex lock indicates the complexity of the synchronization the performance of threads for computing the modular exponentiation.

5. Results and Discussions

To compare the computation efficiency of the developed modular exponentiation functions *precompute_parallel* () for a fixed base with pre-computation with three functions implemented from the Crypto++ 8.2, MPIR and OpenSSL libraries are used. The developed *precompute_parallel* () function uses multithreads computation of the modular exponentiation.

The Crypto++ library of cryptographic algorithms and schemes is implemented in the C++ language and supports Unix platforms (AIX, OpenBSD, Linux, MacOS, Solaris, etc.), Win32, Win64, Android, iOS, ARM [14]. The library contains a set of available primitives for number-theoretic operations, such as generation and verification of prime numbers, arithmetic over a finite field, operations over polynomials. Each of the primitives of the Crypto++ library includes a set of functions, among which is the function *exp_crypto* () for calculating the modular exponent.

The OpenSSL library (The Open Source toolkit for SSL/TLS) contains a set of tools for cryptography that implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols, as well as the corresponding cryptography standards [15]. OpenSSL supports Solaris, HP-UX, Linux, Android, BSD, UnixWare, Win 32 and 64, macOS, iOS, and other platforms. The library includes three functions to calculate the modular exponent using Montgomery multiplication:

BN_mod_exp_mont () calculates A to the power of p modulo m;

BN_mod_exp_mont_consttime () calculates A to the power of p modulo m. This is a variant of the pre-function, that uses fixed windows and dedicated memory for pre-computation to keep data dependency to a minimum to protect secret exponents;

BN_mod_exp_mont_consttime_x2 () calculates two independent raises of A_1 to the power of p_1 modulo m_1 and A_2 to the power of p_2 modulo m_2. For some fixed and equal modulus values of m_1 and m_2, the function uses optimizations that make it possible to speed up the calculation.

A highly optimized modification of the well-known GMP or GNU Multiple Precision Arithmetic Library the MPIR library [13] contains the function *mpz_powm* () to realize computation of ME. The MPIR library uses an optimized version of the ME algorithm, called the "Sliding-window method" [3], which reduce the average number of multiplication operations.

Numerical experiments were carried out on a computer system with a multi-core microprocessor with shared memory in a 64-bit Windows. Testing was performed on computer systems with processors) an Intel Core i5-10600 (6 cores, 12 threads, 3.30GHz) and an Intel Core i9-10980XE (18 cores, 36 threads, 3.0GHz. According to hyper-threading technology, each physical core consists of two virtual ones.

To compute the modular exponentiation with a given number of trials the values of exponent *Exp*, number *Base* and *mod* were given by pseudo-random numbers with number of binary digit of 1024, 2048 bits. To reduce the total computation time on increasing the number of binary digits of long numbers the number of trials of latch-up of the computation time is decrease correspondent. The result of the execution of the function is recorded in the variable *expected_result*, and the computation time is fixed and averaged with the output of the average value "*average time*" in microseconds.

The results are presented in Table 2, which contains the values of average execution time (µs microseconds) of computing the modular exponentiation for pseudo-random data *Base*1, *Exp*1, *mod*1 for 1024 bits and *Base*2, *Exp*2, *mod*2 for 2048 bits, that are

```
    ================ bits=1024 trials=1000 =================
Base 1 =
1559258775283944826146106259936745880191007710663592180785545871625708812395675768044611
2611588790379930841984509857918081567009603552187487090896179963826919606856010375230866
9818128860677719460381304397587862593607968358286567068579479763671817955144283945749615
768573725580291910494735428411976050787788916
Exp 1 =
1428352097864899971708732555079465735564482140846285246054211882240996873229846735436141
7685207105354741569825262257384568307545298247492251784136727860908913698858344775647948
3918441795733215771335093892746851604252279730368411597397577626119447392355080344631875
081748708394736819710836176156337925349995164
mod 1 =
3682232653936167658382890474840879247240833512148561647564031369199365738658920584313452
7207678213908943698149079850300180603359689327300752325201313819068839806646707560057773
0915050017234560082947101924548982647158267357750118464079633252964273456146531893267335
3360118318330216661780501404921220381528643

    ================ bits=2048 trials=500 =================
Base 2 =
2567738760497917474565011343924187031994869216998758698706421709528086295599290907230065
3168631621794286691440902481665333116951445888344416180966407345111071065313623560713743
2150724953185446158678719720295928259781123638183830596292580376934671270834776657789712
9937849667886402861740861770565666978446548767497029913351286923714957597816949211708272
7320204008519907241837229067993684410038784610185215488903193461143855868161082171512473
4828847448121160578454206154924267974589088650928312748724335173725158853105514943013486
1136434044363046876468018170052569298949044683219014189194447340722437694507812846021234
0
```

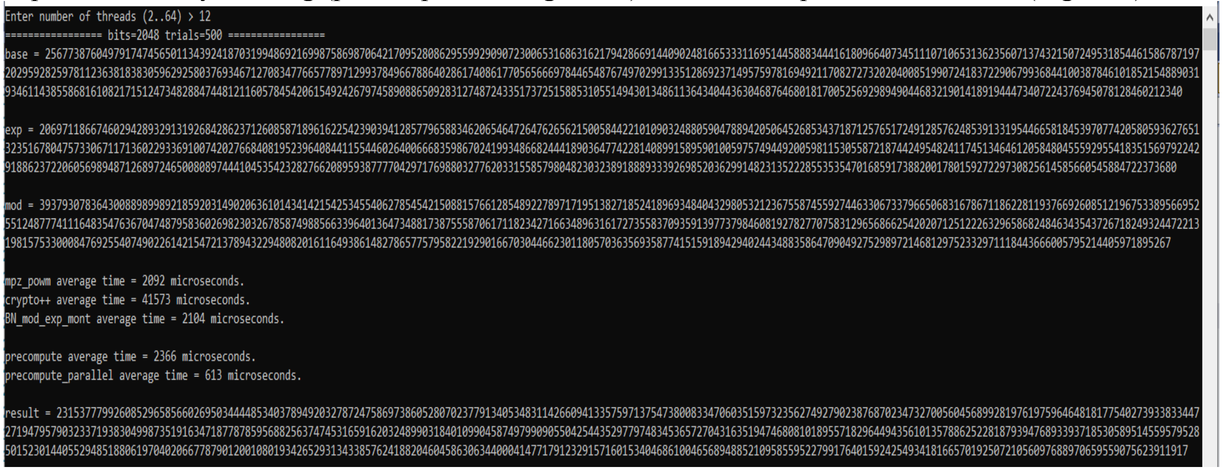
Exp 2 =
2069711866746029428932913192684286237126085871896162254239039412857796588346206546472647
6265621500584422101090324880590478894205064526853437187125765172491285762485391331954466
5818453970774205805936276513235167804757330671171360229336910074202766840819523964084411
5544602640066683598670241993486682444189036477422814089915895901005975749449200598115305
5872187442495482411745134646120584804555929554183515697922429188623722060569894871268972
4650080897444104535423282766208959387777042971769880327762033155857980482303238918889333
9269852036299148231352228553535470168591738820017801592722973082561458566054588472237368
0
mod 2 =
3937930783643008898998921859203149020636101434142154253455406278545421508815766128548922
7897171951382718524189693484043298053212367558745592744633067337966506831678671186228119
3766926085121967533895669525512487774111648354763670474879583602698230326785874988566339
6401364734881738755587061711823427166348963161727355837093591397737984608192782770758312
9656866254202071251222632965868248463435437267182493244722131981575330008476925540749022
6142154721378943229480820161164938614827865775795822192901667030446623011805703635693587
7415159189429402443488358647090492752989721468129752332971118443666005795214405971895267

Testing for the average execution time of computation of modular exponentiation (Table 2) was performed by the functions: *crypto*++() from the Crypto++ library, *mpz_powm* () from the MPIR library, *BN_mod_exp_mont* () from the OpenSSL library. The comparison is performed with developed functions *precompute_parallel* ().The pre-computation time *precompute* () to determine of the sequence of redused set of residues is taken into account.

Table 2.
The average execution time (µs) of the functions of computing the modular exponentiation

Release/x86	Intel Core i5-10600 number of threads (2..64) >12		Intel Core i9-10980XE number of threads (2..64) > 36	
Data	Base1, Exp1, mod1	Base2, Exp2, mod2	Base1, Exp1, mod1	Base2, Exp2, mod2
bits / trials	1024 / 1000	2048 / 500	1024 / 1000	2048 / 500
crypto++()	442	3309	500	3713
BN_mod_exp_mont ()	312	2228	340	2497
mpz_powm ()	279	1983	311	2321
Precompute ()	236	1242	260	1399
Parallel ()	61	244	106	254
precompute_parallel ()	297	1486	366	1653

The developed function *precompute_parallel*() performs the computation of modular exponentiation by forming (precompute average time) a reduced sequence of residuals (Figure 4).



```
Enter number of threads (2..64) > 12
================= bits=2048 trials=500 =================
base = 2567738760497917474565011343924187031994869216998758698706421709528086295599290907230065316863162179428669144090248166533311695144588834441618096640734511107106531362356071374321507249531854461586787197
2029592825978112363818383059629258037693467127083477665778971299378496678864028617408617705656669784465487674970299133512869237149575978169492117082727320204008519907241837229067993684410038784610185215488903 1
93461143855868161082171512473482884744812116057845420615492426797458908865092831274872433517372515885310551494301348611364340443630468764680181700525692989490446832190141891944473407224376945078128460212340

exp = 2069711866746029428932913192684286237126085871896162254239039412857796588346206546472647626562150058442210109032488059047889420506452685343718712576517249128576248539133195446658184539707742058059362765 1
3235167804757330671171360229336910074202766840819523964084411554460264006668359867024199348668244418903647742281408991589590100597574944920059811530558721874424954824117451346461205848045559295541835156979224 2
91886237220605698948712689724650080897444104535423282766208959387777042971769880327762033155857980482303238918889333926985203629914823135222855353547016859173882001780159272297308256145856605458847223736 80

mod = 3937930783643008898998921859203149020636101434142154253455406278545421508815766128548922789717195138271852418969348404329805321236755874559274463306733796650683167867118622811937669260851219675338956695 2
5512487774111648354763670474879583602698230326785874988566339640136473488173875558706171182342716634896316172735583709359139773798460819278277075831296568662542020712512226329658682484634354372671824932447221 3
19815753300084769255407490226142154721378943229480820161164938614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366600579521440597189526 7

mpz_powm average time = 2092 microseconds.
crypto++ average time = 41573 microseconds.
BN_mod_exp_mont average time = 2104 microseconds.

precompute average time = 2366 microseconds.
precompute_parallel average time = 613 microseconds.

result = 2315377799260852965856602695034444853403789492032787247586973860528070237791340534831142660941335759713754738008334706035159732356274927902387687023473270056045689928197619759646481817754027393383344 7
2719479579032337193830499873519163471877878595688256374745316591620324899031840109904587497990905504254435297797483453657270431635194746808101895571829644943561013578862522818793947689339371853058951455957952 8
50152301440552948518806197040206677879012001080193426529313433857624188204604586306344000414771791232915716015340468610046568948852109585595227991764015924254934181665701925072105609768897065955907562391191 7
```

Figure 4: The result of testing the functions of calculating the modular exponent on a computer system with an Intel Core i5-11400H processor with a number of threads of 12

The *precompute_parallel*() function of the modular exponentiation reduces the computation time relative to other functions of software libraries Crypto++, OpenSSL and MPIR designed for working with big numbers. The execution time of modular exponentiation on computer systems with Intel Core i5-10600 and Intel Core i9-10980XE processors does not change significantly according to Table 2. The optimal number of threads is 12...16 for fast computation of modular exponentiation for universal computer systems.

In the process of solving many applied problems of efficient computation of modular exponentiation, one of the most important parameters is the calculation time. An important component of this parameter is, along with the use of algorithmic and software tools, the performance of computing tools. Therefore, based on of the developed software the further implementation of the computation of modular exponentiation using multithreaded technologies will provide an opportunity the efficient computation of modular exponentiation with a fixed base [19].

6. Conclusion

The work compares and analyses the developed software implementation *precompute_parallel*() function of the computation of modular exponentiation and the software implementation of the functions of Crypto++, OpenSSL and MPIR libraries. The computational scheme of modular exponentiation, the software implementation of the algorithm for computing of modular exponentiation, the run time results of the computation on multi-core microprocessors of universal computer systems have been described. As a result, has been developed the computation function *precompute_parallel*() speedups the execution of the computations using modular exponentiation of the right-to-left binary exponentiation method for a big number of fixed base with pre-computation of redused set of residuals.

The scientific novelty of obtained results lies in the implementation of parallelism using multithreading in the algorithm of computing the modular exponentiation based on the use of the representation of exponent in the form of a binary number and pre-computation for a fixed base of a reduced set of residuals.

The practical significance of the work lies in the fact that the obtained results can be successfully apply in the modern asymmetric cryptography, for efficient computation of number-theoretic transforms and other computational problems. The developed function *precompute_parallel*() speedups the execution of the computations of modular exponentiation of big numbers in comparison with existing libraries. The especially effective will be use function *precompute_parallel*() for application with fixed-base exponentiation is in elliptic curve cryptography, for instance for Diffie-Hellman key agreement and elliptic curve digital signature algorithm verification.

Prospects for further research are the development and implementation Montgomery Modular Multipliers in developed function *precompute_parallel*() of multithreading computations of modular exponentiation for big numbers.

7. References

[1] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, Handbook of applied cryptography. 5th printing, CRC Press, Boca Raton, 2001. doi:10.1201/9780429466335.

[2] C. Studholme, The Discrete Log Problem, 2002. URL: http://www.cs.toronto.edu/~cvs/dlog/research_paper.pdf

[3] A. Jakubski, R. Perlinski, "Review of general exponentiation algorithms", Scientific Research of the Institute of Mathematics and Computer Science, 10.2 (2011): 87-98.

[4] A. Rezai, P. Keshavarzi, "Algorithm design and theoretical analysis of a novel CMM modular exponentiation algorithm for large integers". RAIRO – Theoretical Informatics and Applications, 49.3, (2015): 255-268. doi:10.1051/ita/2015007.

[5] I. Marouf, M. M. Asad, Q. A. Al-Haija, "Comparative Study of Efficient Modular Exponentiation Algorithms". COMPUSOFT, International journal of advanced computer technology, 6.8 (2017): 2381-2392.
[6] S. Vollala, K. Geetha, N. Ramasubramanian, "Efficient modular exponential algorithms compatible with hardware implementation of public-key cryptography". Security and Communication Networks, 9.16 (2016): 3105-3115.
[7] H. Cohen, *A* course in computational algebraic number theory. Berlin, Heidelberg: Springer, 1993. doi:10.1007/978-3-662-02945-9.
[8] J.-M. Robert, C. Negre, T. Plantard, "Efficient Fixed Base Exponentiation and Scalar Multiplication based on a Multiplicative Splitting Exponent Recoding", Journal of Cryptographic Engineering, Springer, 9.2 (2019): 115-136. doi:10.1007/s13389-018-0196-7.
[9] P. Lara, F. Borges, R. Portugal, N. Nedjah, "Parallel modular exponentiation using load balancing without precomputation". Journal of Computer and System Sciences, 78.2 (2012): 575-582. doi:10.1016/j.jcss.2011.07.002.
[10] N. Emmart, F. Zheng, C. Weems, C. Faster Modular Exponentiation using Double Precision Floating Point Arithmetic on the GPU, in: Proceedings of the 25th IEEE Symposium on Computer Arithmetic (ARITH), Amherst, MA, USA, 2018, pp. 126-133. doi:10.1109/ARITH.2018.8464792.
[11] S. Li, J. Tian; H. Zhu; Z. Tian; H. Qiao; X. Li; J. Liu, Research in Fast Modular Exponentiation Algorithm Based on FPGA, in: Proceedings of the 11th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), Qiqihar, China, 2019, pp. 79-82.
[12] PARI/GP home, 2021. URL: http://pari.math.u-bordeaux.fr/.
[13] MPIR: Multiple Precision Integers and Rationals. 2021.URL: http://mpir.org/.
[14] Crypto++ Library 8.6, 2021. URL: https://www.cryptopp.com
[15] OpenSSL. Cryptography and SSL/TLS Toolkit, 2021. URL: http://www.openssl.org/.
[16] C. Negre, T. Plantard, "Efficient Regular Modular Exponentiation Using Multiplicative Half-Size Splitting", Journal of Cryptographic Engineering, Springer, 7.3 (2017): 245-253. doi:10.1007/s13389-016-0134-5.
[17] I. Prots'ko, N. Kryvinska, O. Gryshchuk, "The Runtime Analysis of Computation of Modular Exponentiation", Radio Electronics, Computer Science, Control, 3 (2021): 42-47. doi:10.15588/1607-3274-2021-3-4.
[18] I. Prots'ko, O. Gryshchuk, "The Modular Exponentiation with precomputation of redused set of resedues for fixed-base". Radio Electronics, Computer Science, Control, 1 (2022): 58-65. doi:10.15588/1607-3274-2022-1-7 .
[19] Exponentiation by squaring, 2022. URL: https://en.wikipedia.org/wiki/Exponentiation_by _squaring.