The implementation of Montgomery modular reduction to speed up of modular exponentiation

Ihor Prots'ko [1], Oleksandr Gryshchuk[2]

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

Abstract
Modular exponentiation over large integers involves multiple modular multiplications, which is very computationally expensive. Many processing systems use the Montgomery modular multiplication method, which reduces the latency of software and hardware implementations. The main directions of software development and outlines of the parts of Montgomery modular multiplication for the implementation are presented. The class Montgomery Arithmetic over large integers is implemented using four methods for Montgomery modular multiplication. We present the computation of modular exponentiation using the right-to-left binary exponentiation method for a fixed basis with a developed pre-computation of a reduced set of remainders using modular Montgomery multiplication.
A comparison of the runtimes of three variants of functions for computing the modular exponentiation over large integers is performed. The algorithm with pre-computation of residues for fixed base provides a faster computation of modular exponentiation using Montgomery modular multiplication compared to the functions of modular exponentiation of the MPIR, OpenSSL libraries for large number more then 1K bits.

Keywords
Montgomery modular multiplication, modular exponentiation, multithreading, large numbers

1. Introduction

Modular reduction is the computation of x mod m. A basic operation in processing systems is computations in Zm integers modulo m, where m is a large positive integer, which may or may not be a prime. Modular reductions are normally used to create finite groups, rings, or fields. The most common usage for performance-driven modular reductions is in modular exponentiation algorithms. An efficient implementation of the modular reduction x mod m of large numbers is the key to high performance.

The classical algorithm of modular reduction has no restriction on the size of x, m and can easily be adapted to a division algorithm with quotient and remainder. The formalization consists of estimating the quotient digit as accurately as possible. This is justified by the fact that using multiplication and division are the most time-consuming operations in the inner loops of algorithms, especially when calculating Modular reduction over multi-bit numerical data.

Among the modular reduction algorithms: classical, Barrett, and Montgomery's, the Montgomery reduction is relatively simple and very efficient [1]. The baseline Montgomery reduction algorithm will produce the residue for any size input. Montgomery reduction is a common algorithm used for modulus reduction. The unique property of this algorithm is that it does not compute the modulus directly, but instead, the modulus multiplied by a constant.

The further development using Montgomery reduction for computing modular multiplication is much faster and does not require any division by m. This method is referred as Montgomery

CMIS-2024: Seventh International Workshop on Computer Modeling and Intelligent Systems, May 3, 2024, Zaporizhzhia, Ukraine
ihor.o.protsko@lpnu.ua (I. Prots'ko); ocr@ukr.net (O. Gryshchuk)
0000-0002-3514-9265 (I. Prots'ko); 0000-0001-8744-4242 (O. Gryshchuk)

modular multiplication and combines Montgomery reduction and multiple-precision multiplication.

The scientific problem of speeding up modular reduction for processing systems is relevant for the present stage of the development of information and computer technologies. The software implementations of modular multiplication over large integers on general-purpose processors are an important target for optimization. The further increase in the speed of the computational implementation of the modular reduction operation and then the full multiplication part can be achieved only by using the multithreading of multi-core processor architectures.

The paper is structured as follows: after the Introduction in Section 1 is described Montgomery Reduction as a common algorithm used for modulus reduction, and outlines the parts and basic stages of the Montgomery modular multiplication algorithm. Section 2 describes the developed software implementation of efficient Montgomery multiplication over large integers using the Multiple Precision Integers and Rationals library. For performance analysis in Section 3, the experiments and discussion of the software implementation of Montgomery modular multiplication for the computation of developed modular exponentiation are presented. As a result, the developed software implementation provides a faster computation of modular exponentiation using Montgomery modular multiplication compared to the general-purpose functions of modular exponentiation of the MPIR and OpenSSL libraries over large integers.

2. Literature review

There are different contemporary variations of Barrett and Montgomery algorithms, which have advantages and mines. Barrett reduction is a reduction algorithm proposed in 1986 by P.D. Barrett [2], designed to optimize the integers modulo *m* operation assuming *m* is constant and, divisions are replaced by multiplications. P. Barrett offered the idea of estimating the quotient *x* div *m* with operations that are less expensive in time than a classical multi-precision division by *m*. The only pre-computation $[2^{2n}/m]$ required for successful modular reduction use of Barrett's algorithm, where $2n$ is a number of bits. The computation of modular exponentiation based on Barrett's algorithm is better than the other known ones for small numerical values.

Montgomery reduction uses on the changing of the original reduction modulo by some other convenient modulo. By representing the residue classes modulo Montgomery's algorithm [3] replaces a division by *m* with a multiplication followed by a division by a power of radix *r*. In computer applications, *b* is usually defined as the power of 2, when $m = 2k$, k – the processor's word-size, this operation is very easy and inexpensive. The idea developed by P. Montgomery's method suggests that the operations of addition and subtraction are practically unchanged, but multiplication changes slightly in a simple procedure without using reductions modulo *m*. Montgomery's algorithm (only for modulo *m* for which $\gcd(m, r) = 1$) is faster than both the classical and Barrett's one and as fast as multiplication almost.

There are different implementation algorithms of Montgomery reduction, which are improving to simpler and higher regularity. The paper [4] proposes new residue number system Montgomery reduction algorithms, which achieve less number of unit multiplications. Traditional Montgomery approaches are combined with multiply-reduce methods at the bit-level in hardware implementations or based on the processor's word-size level for software implementations [5]. The parallel execution of modular operations "square and multiplications" based on Montgomery algorithm are described in the papers [6]. The implementation of the Montgomery algorithm has been improved over the years, both at the software and hardware levels [7].

3. Montgomery Reduction and Modular Multiplication

The Montgomery reduction of number *T* is defined as

$$T R^{-1} \bmod m, \tag{1}$$

where *m* is a positive integer, *T* and *R* are integers such that $R > m$, $\gcd(m, R) = 1$, and $0 \le T < mR$.

The formula (1) is called a Montgomery reduction of number T modulo m with respect to R. Using Montgomery reduction easy to carry out modular reduction in the residue number system. The residue number system is a method for representing an integer as an n-tiple of its residues with respect to a given base. Montgomery Reduction $i R^{-1}$mod m is a one-to-one mapping defined from Z/m_Z to Z/m_Z, for $0 \le i < m$.

To compute the Montgomery reduction, it is necessary to determine the value of R^{-1} that meets the condition $R \cdot R^{-1} \bmod m = 1$.

To find the inverse modulo, you can use the extended Euclidean algorithm. Indeed, if gcd(m, R) =1, then the following integers will be found u and v, that

$$Rv + mv = 1. \qquad (2)$$

If we pass to congruence modulo m in the last equality, then we obtain $Ru \equiv 1 \pmod m$, which gives $(R^{-1}) \equiv u \pmod m$.

For working with large numbers, where Montgomery multiplication is implemented, is common to write the Montgomery radix R as

$$R = r^k = 2^k, \qquad (3)$$

where k is the word-size of the computer architecture. Higher radices may be used but the radix-2 provides a simple algorithmic and hardware implementation.

The algorithm to compute Montgomery constant $\mu = -m^{-1} \bmod R$ for odd values m and $R=2^k$ is presented in the Fig. 1.

Algorithm. Compute Montgomery constant $\mu = -m^{-1} \bmod R$
Input : Odd integer m and $R=2^k$
Output : $\mu = -m^{-1} \bmod R$
$y \leftarrow 1$
for i =2 to k do
    if ($m\,y \bmod 2^i$) ≠ 1 then
        $y \leftarrow y + 2^{i-1}$
    end if
end for
return $\mu \leftarrow R - y$;

Figure 1: Algorithm of Computation of the Montgomery constant $-m^{-1}$mod R

There are different fast Modular Reduction Methods to implementing Montgomery modular reduction. The algorithm Montgomery Reduction for radix 2, which does not require some pre-computation is presented in Fig. 2.

Algorithm Montgomery Reduction $X R^{-1} \bmod m$
Input : X, m and $R=2^k$
Output : $X\, 2^{-k} \bmod m$
$x = X$
for i= 1 to k do
    if x is odd then
        $x = x + m$;
        $x = x/2$;
return x;

Figure 2: Algorithm of Computation of the Montgomery Reduction for radix 2

This algorithm is based on scanning the bit of a large number X from the right (the least significant bit) to the left (the most significant bit).

In the paper [8] is described the efficiently computes Montgomery reduction. Let $m' = -m^{-1}$ mod R, if $U = Tm' \bmod R$, $m^* m^{-1} \bmod R = 1$, then

$$T R^{-1} \bmod m \equiv (T + U\, m) / R. \qquad (4)$$

Taking the remainder modulo m was replaced by division by R, and also taking the remainder modulo R in the numerator of the formula (4). As a result, we can choose such R that truncation can be used instead of division. If we have long arithmetic with some radix r, then the degree of this radix r_i. That is, modulo residues and divisions will turn into shifts and throw out extra numbers. In the chapter 14.3.2 Montgomery reduction [8] are presented the algorithms and examples of Montgomery reduction based on formula (4). The algorithm does not require $m' = -m^{-1} \bmod R$, but rather $m' = -m^{-1} \bmod r$.

Most processing systems are implemented by repetition of a modular multiplication with a large modulus m, that is,

$$z = x \cdot y \bmod m. \qquad (5)$$

where m is usually a large prime or a product of two large primes $x = (x_{n-1} \dots x_1 x_0)_r$ and $y = (y_{n-1} \dots y_1 y_0)_r$, which are non-negative integers in a radix r representation such that $x < m$ and $y < m$.

Let us represent x' and y' of a number x and y in the Montgomery space as follows

$x' = x\,R \bmod m$ and $y' = y\,R \bmod m$.

The Montgomery reduction of multiplication $x'y'$ is:

$$x' \cdot y' R^{-1} \bmod m \equiv (x R \bmod m * y R \bmod m) / R^{-1} \bmod m = x \cdot y R \bmod m. \qquad (6)$$

This means that, after doing the multiplication of two numbers in the Montgomery space, we need to reduce the result by multiplying it by R^{-1} and taking of modulo m. There is an efficient way to use Montgomery reduction. This operation called the Montgomery modular multiplication. Montgomery modular multiplication itself is fast, but it requires some pre-computation. Montgomery multiplication algorithm involves three basic stages:

1. The conversion of operands from integer domain to Montgomery space;
2. The multiplication of operands in the Montgomery space;
3. The conversion of operands back from Montgomery space to integer domain.

The Montgomery multiplication needs to convert x and y into Montgomery space and their product out of Montgomery space (Fig. 3). In this method the costly division operation usually needed to perform modular reduction is replaced by simple shift operations by conversing the operands into the reduced number system domain before the operation and re-conversing the result after the operation. Montgomery modular multiplication involves: first conversion of operands into the Montgomery space, multiplication and then after the result is re-conversed into the Montgomery space.

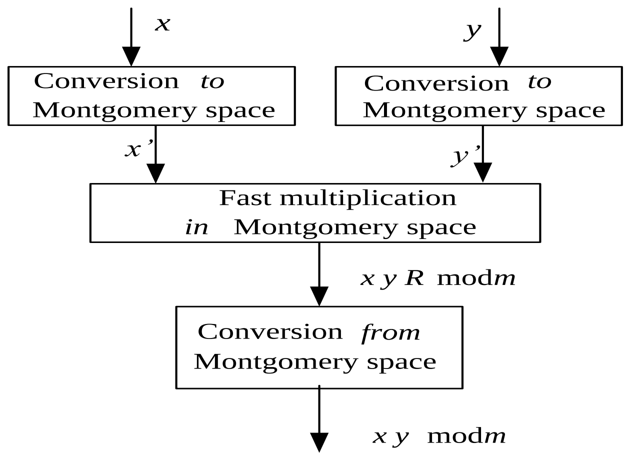

Figure 3: Computation of modular reduction using Montgomery modular multiplication

For practical (Fig. 4) interest the $R=r^n$ will suffice when there can be a power of 2 and $R=2^n$ [9]. The condition $R > m$ is clearly satisfied, but gcd (m, R) =1 needs to be relatively prime i.e. must not have any common non-trivial divisors which will hold only if gcd (m, r) =1.

Montgomery modular multiplication algorithm $X\,Y\,(R^{-1}) \bmod m$
Input : X, Y, m and $R=2^k$,
Output : $X\,Y\,2^{-k} \bmod m$
S_0 : LSB of S, $x_i \in (x_{n-1} \dots x_1 x_0)_2$

```
S=0
for i= 0 to n do
    S = S + x_i Y ;
    S = S + S_0 m ;
    S = S/2 ;
if  S ≥ m then S = S-m;
return S;
```

Figure 4: Algorithm of computation of the Montgomery modular multiplication

There are different implementations of Montgomery modular multiplication: the digit-serial architectures [10], special purpose circuits, what perform multiplication and reduction simultaneously [11], and parallel execution of modular multiplication [12]. In practice at the software and hardware levels, Montgomery multiplication is the most efficient method when is used a very regular structure, which speeds up the implementation [13, 14].

The software implementations of modular multiplication over large integers on general-purpose processors are an important target and has been improved over the years. In the next Section, we describe the software implementation of efficient Montgomery multiplication over large integers using the Multiple Precision Integers and Rationals library.

4. The software implementation of Montgomery reduction to modular multiplication

The software implementations of Montgomery modular multiplication on the general purpose processors are an important target for optimization. Important focus is on the software implementation of the full multiplication parts including the efficient reduction. Many works improve the performance of a Montgomery Multiplication [15, 16]. Almost all the implementations of modular multiplication in many processing systems are performed in assembly languages to take advantage of the specific architectural properties of the processor [17].

In this section, we describe software implementations of modular multiplication on the basis of the realization of Montgomery modular multiplication, which includes the efficient modular reduction and multiplication parts.

The modular multiplication is implemented in C++ language. The developed *class MontgomeryArithmetic* (Fig. 5) implements the Montgomery modular multiplication and reduction using the Multiple Precision Integers and Rationals library (MPIR) [18].

```
Class MontgomeryArithmetic
-------------------------------------------------------------
private:
   const mpz_class mod_;
   size_t mod_size_;
   mpz_class inv_;
   const size_t limbs_;
   const size_t bits_;
-------------------------------------------------------------
public:
   explicit MontgomeryArithmetic(const mpz_class& mod);
   mpz_class init(const mpz_class& x) const;
   void multiply(mpz_class& a, const mpz_class& b) const;
   void reduce(mpz_class& x) const;
```

Figure 5: The *MontgomeryArithmetic* class

According to the markings in Fig. 5, the member variables of the *Class MontgomeryArithmetic* are:
*size_t mod_size_* is a divisor size in MPIR limbs (64-bit integers);
*mpz_class inv_* is a pre-computed inverse factor for the Montgomery reduction;
*const size_t limbs_* is the same as *size_*, but a more convenient name;

*const size_t bits_* is a bit count for the modular arithmetic.
The parameters of the methods are:
mod is a divisor for modular arithmetic;
x is a number for the conversion;
a and *b* are the first and second numbers converted to the Montgomery space.

The constructor *MontgomeryArithmetic*(*const mpz_class& mod*) computes a modular inverse factor for the Montgomery reduction and initializes other member variables, where the argument mod is a divisor for modular arithmetic.

For computing the inverse factor $m' = m^{-1} \bmod R$ efficiently, we can use the mathematical dependence, which is inspired by Newton's method. The algorithm for calculating the inverse factor is described and proved in [19] :

$$m \cdot x \equiv 1 \bmod 2^k \rightarrow m \cdot x \cdot (2 - m \cdot x) \equiv 1 \bmod 2^{2k}. \quad (7)$$

This means we can start with $x = 1$, as the inverse of m modulo 2^1, apply the trick of power times and in each iteration we double the number. This algorithm uses only shifts, subtractions and multiplication of large numbers in each iteration and has the same computational complexity as the algorithm, which is shown in Fig. 1.

The method *init mpz_class init*(*const mpz_class& x*) *const* converts a number to the Montgomery space. It is required to convert all numbers before applying the Montgomery multiplication. The algorithm for the conversion is described in [19], where the relation is used

$$x \cdot R \bmod m = x R^2 / R = x \cdot R^2, \quad (8)$$

where x is a number for the conversion. Converting the number into the space is just a multiplication inside the space of the number with R^2. Therefore, we can pre-compute $R^2 \bmod m$ and just perform a multiplication instead of shifting the number. This algorithm uses the shifts and the subtractions and multiplications of large numbers in each *bits_* iteration.

The method returns the converted value, which can be used for the Montgomery multiplication. The method *void multiply*(*mpz_class& a, const mpz_class& b*) *const* multiplies two numbers, where *a*, *b* are the numbers converted to the Montgomery space. The method returns the result via first argument in place and then performs the Montgomery reduction. It modifies the first argument in place to improve efficiency and avoid copying. For multiplication, it uses regular multiplication provided by the MPIR library, which is optimized using AVX2 SIMD instructions.

The method *void reduce*(*mpz_class& x*) *const*, where argument x is a number for the reduction in place, computes the Montgomery reduction in place. Any number from the Montgomery space can be converted back using this method. This is one of the most performance-critical methods. The MPIR library [18] offers a few low-level implementations, which can be further optimized for specific use cases. This method calls the *mpn_redc1*() function provided by MPIR to compute the Montgomery reduction in place.

The methods and initialized member variables in the developed *class MontgomeryArithmetic* provide an implementation of Montgomery modular multiplication corresponding to Fig. 3. The operations of multiplication and division by $R=2^k$ are very fast in the methods of class, as they are just bit shifts. Thus, Montgomery's algorithm is faster than the usual $(a \cdot b) \bmod m$, which contains division by m. However, the computation R^{-1}, m^{-1} and conversion of numbers to the remains and vice versa are time-intensive operations, as a result, of which it is inefficient to use the product for a single computation. Montgomery reduction is the fastest in computing a reasonably long series of modular reductions, for instance in computing exponential function. This algorithm is a time critical step in the computation of the modular exponentiation operation.

5. Experiments and discussions of the software implementation of Montgomery modular multiplication for the computation of modular exponentiation

Modular exponentiation over large integers involves multiple modular multiplications, which is very computationally expensive. Modular exponentiation of large numbers is extremely necessary for providing high crypto capability of information data, for finding the discrete logarithm, in number-theoretic transforms and many other applications.

Considerable attention is paid to the development of effective methods of modular exponentiation aimed at effective computation and reduction of the execution time of the modular exponentiation operations [20, 21]. One of the ways to speed up computations of modular exponentiation is parallelization of computations using modern software technologies for universal computer systems or creation of specialized computing tools. The software implementation of the Montgomery multiplication and modular exponentiation computation is included in the software libraries Crypto++, OpenSSL, PARI/GP, MPIR designed for working with large numbers.

The production-grade software library and full-featured toolkit popular on Linux and other systems is OpenSSL library. OpenSSL library contains a set of tools that implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) [22]. The functions *BN_mod_mul_montgomery*, *BN_MONT_CTX_new* of OpenSSL library implement Montgomery multiplication. The library includes three functions to calculate the modular exponentiation using Montgomery multiplication: *BN_mod_exp_mont*(), which calculates A to the power of x modulo m, and *BN_mod_exp_mont_consttime*(), *BN_mod_exp_mont_consttime_x2*().

Let's compare the use of Montgomery modular multiplication with the usual modular multiplication operation on the example of an efficient computation of modular exponentiation of large numbers. Consider the basic iterative algorithm using pre-computation to form a shortened sequence of residues of the fixed base A for computing the modular exponentiation

$$y = A^x \bmod m. \qquad (9)$$

The central idea to calculate A^x mod m is to use the binary representation of the exponent x. For a fixed-base A of the modular exponentiation (9), which is equal to the product of the residuals $r.0, r.1, \ldots, r.k$-1 of the exponent (A^{2i}) mod m, (i = 0, 1, 2, …, k-1). Modular exponentiation is implemented using the development of the right-to-left binary exponentiation method for a fixed base with pre-computation of a reduced set of residuals. That can speed up the process of computing the modular exponentiation by pre-computing (Fig. 6) the sequence of residuals, and repetitions with the period T' after the offset u in the unit Precomputation u, T' [23].

The scheme (Fig. 6) for computing the modular exponentiation consists of the denotations:

- A is the input of the base number; m is the input of the module;
- x is the input of an exponent with binary digits $x.(k\text{-}1), x.(k\text{-}2), \ldots, x.2, x.1, x.0$;
- $(A\text{^}2^i)_m$ are blocks of computation of the integer exponent of exponent 2^i of the number A by the module m, i = 0,1,2,…, (k-1);
- $r.0, r.1, \ldots, r.k$-1 are residuals $A\text{^}2^i$ mod m, (i = 0, 1, 2, …, k-1),
- (X) mod m is the block of modular multiplication;
- y is the output of the modular exponentiation.

Thus, applying the parallel execution of the computation of modular exponentiation with the pre-computation, threads are created during the software execution of the modular multiplication of residual values $r.i$, where $i \le T'$, in the block of modular multiplication. These residual values $r.i$ are determined in the process of computing of residual exponents $(A\text{^}2^i)$ mod m, (i = 0, 1, 2, ..., k-1). The only difficulty in organizing computations with such threads is the need to synchronize the streams and the unit of Precomputation u, T' to ensure the correct computation of the final value y of modular exponentiation.

To implement the algorithm for computing the integer power of a number A^x by modulus m, the MPIR library is used, which is written in C and assembler and provides the ability to compile its functions in Visual Studio C++. Accordingly, in the MPIR library, the data type *mpz_t* represents large numbers of arbitrary length, which are chosen for the power of the number base and mod with the number of bits from 256 to 2048 bits for testing. However, using the function *mpn_redc1*() implement Montgomery multiplication is not efficient enough in the process of modular exponentiation.

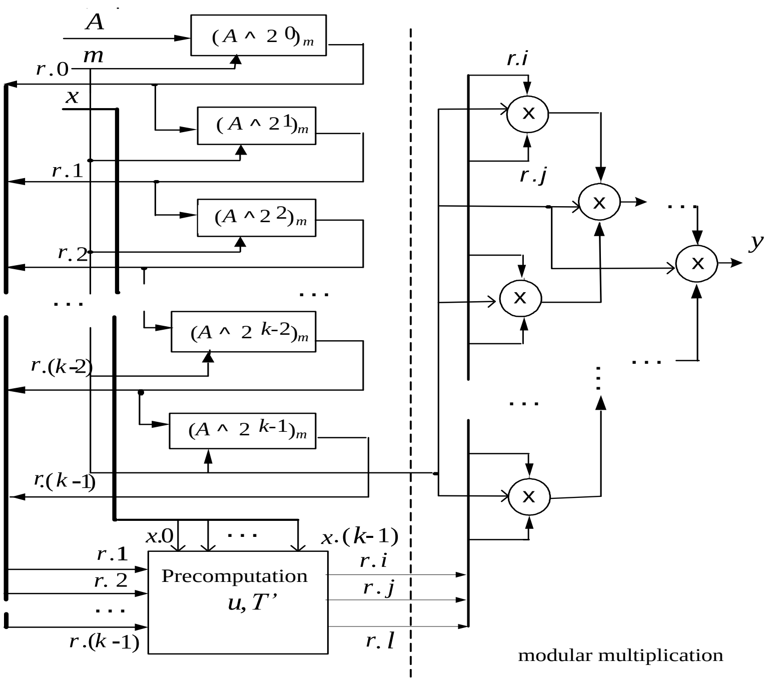

Figure 6: The scheme for computation of modular exponentiation y = A^x mod m with pre-computation

The algorithm consists of *precompute*() and *precompute_parallel*() functions. The *precompute*() function determines the sequence of a reduced set of residues. The *precompute*() function calculates the sequence of remainders for fixed numbers *base* and *mod* for *exp* = 2^i (i = 0, 1, 2, …) and analyzes the periodicity with the appearance of each defined remainder *r.i*, which are calculated by the *find_remainders*() function. The pre-computation has been made in a separate *find_remainders*() function to optimize multiple remainder searches (A^2^i) mod m. The function *update_remainers*() reduces the length of the sequence of remainders as a result of fixing the periodicity T', taking into account the offset u.

The *precompute_parallel*() function aim to compare the performance execution with the use of Montgomery modular multiplication and usual modular multiplication operation. To implement the algorithm, the *mpz_init_set* (*mul*, *base*), *mpz_sizeinbase* (*exp*, 2), *mpz_tstbit* (*exp*, *i*), *mpz_mul* (*r*, *r*, *mul*) functions from the MPIR library are used, the parameters of which can be multi-bit data limited to bit size 2048 bits. To organize efficient multithreading computation of modular exponentiation according to the *precompute_parallel*() function, the *thread_function*() and *parallel*() are implemented. The developed *precompute_parallel*() function uses multiple threads for the computation of the modular exponentiation. The method *run*() runs parallel exponentiation using multiple threads. It has the following steps:

1) creates a collection of the active exponent bits;
2) splits the exponent bits among the defined number of threads;
3) waits for every thread execution.
4) calculates the final result by multiplying partial results calculated by the threads.

The final result of the function is written to the variable *s_thread_result*, and the computation time is fixed and averaged to output.

We compare the time of calculating the modular exponent using the usual modular multiplication with the Montgomery modular multiplication based on the developed functions *precompute_modulo*(), *precompute_parallel_modulo*() *and* *precompute_montgomery*(), *precompute_parallel_montgomery*(), respectively.

Testing of the calculation of modular exponentiation were carried out on a computer system with a multi-core microprocessor Intel Core i9-10980XE (18 cores, 36 threads, 3.0GHz) with shared memory in a 64-bit Windows. According to hyper-threading technology, each physical core of 18 consists of two virtual 36 ones. The numerical results are presented in Figure 7, which contains the values of average execution time (μs microseconds) for 500 and 250 trials of computing the modular exponentiation for pseudo-random data *base, exp, mod* for 1024 bits and 2048 bits.

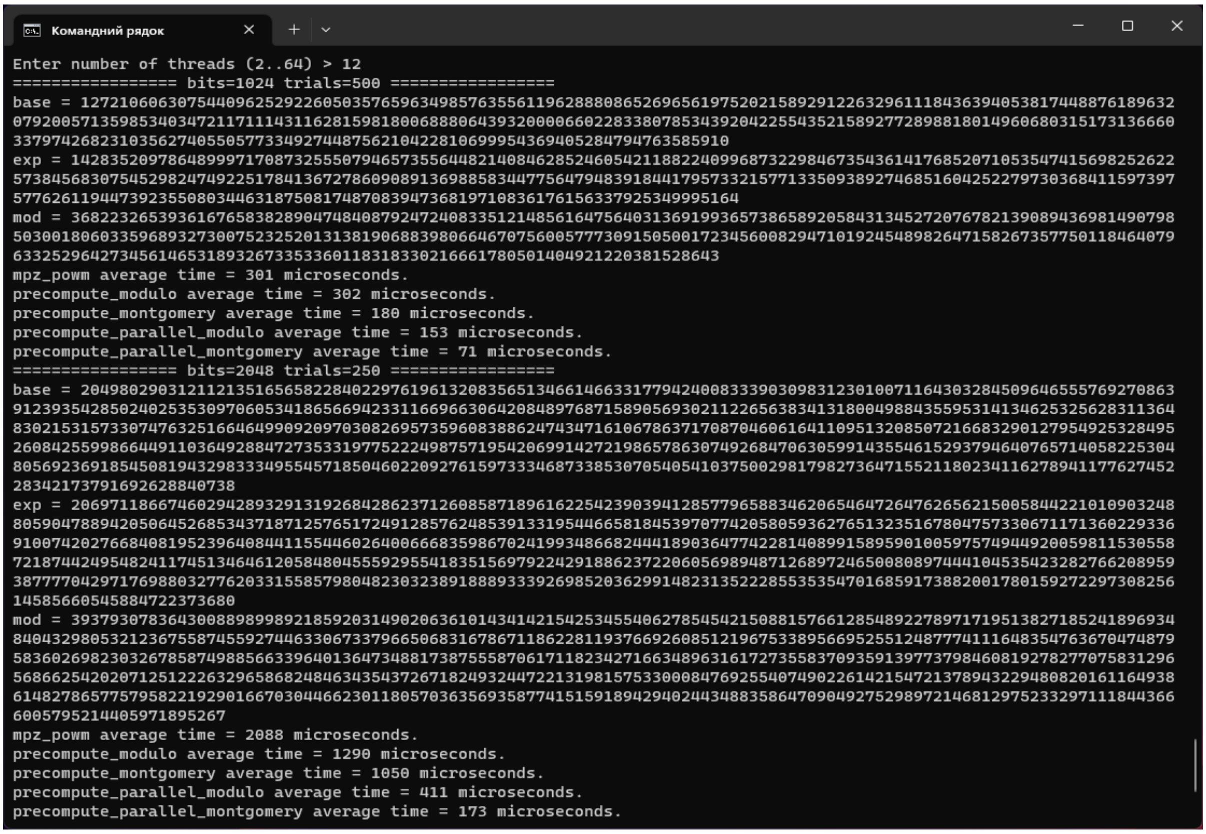

Figure 7: The results of testing the functions of computing the modular exponentiation on a computer system with an Intel Core i9-10980XE processor with a chosen number of threads of 12

The pre-computation time to determine the sequence of a reduced set of residues is taken into account, therefore the total average time for computing the modular exponentiation *modexp*() is equal to:

1) for the usual modular multiplication operation:

modexp() = *precompute_modulo*() time + *precompute_parallel_modulo*() time.

In accordance with *t*he result of testing (Fig. 7) average time are equal to:

modexp()=(301+153)*μs*=454*μs;*

modexp()=(1290+411)*μs =1701μs;*

for pseudo-random data the base, exp, mod of 1024 bits and 2048 bits respectively.

2) for the Montgomery modular multiplication:

modexp()= *precompute_montgomery*() time + *precompute_parallel_montgomery*() time.

In accordance with *t*he result of testing average time (Fig. 7) are equal to:

modexp()= (1290+71)*μs* =251 *μs;*

modexp()= (1050+173)*μs*=1223*μs*;

for pseudo-random data the base, exp, mod of 1024 bits and 2048 bits respectively.

Therefore, the implementation of the Montgomery modular multiplication is based on the developed *class MontgomeryArithmetic* for computing the modular exponentiation speed up 454μs/251μs=1,8 and1701μs/1223μs=1,4 times for pseudo-random data the base, exp, mod of 1024 bits and 2048 bits.

A highly optimized modification of the well-known GMP or GNU Multiple Precision Arithmetic Library the MPIR library [18] contains the function *mpz_powm* () to realize the computation of modular exponentiation. The MPIR library uses an optimized version a floating-window algorithm of the modular exponentiation with Montgomery multiplication/reduction, which reduces the average number of multiplication operations. The function of the MPIR library *mpz_powm*(*expected_result, base, exp, mod*) better performs modular exponentiation than function *BN_mod_exp_mont*() of the OpenSSL and Crypto++ libraries in accordance with the results received in [24], therefore we chose the function *mpz_powm*() for comparison (Fig. 5). At testing results, the total average time of *mpz_powm* () function for computing the modular exponentiation *modexp*() is 301μs and 2088μs and is greater than the average time for computing the modular exponentiation the Montgomery modular multiplication 251μs and 1223μs for pseudo-random data the base, exp, mod of 1024 bits and 2048 bits respectively.

The closest scientific work for comparing research results is work [25], where an approach that uses vector SIMD instructions for parallel computation of multiple Montgomery multiplications is applied. This work [25] describes the fact of the comparison of a parallel version of Montgomery multiplication using vector SIMD instructions to the implementation of the function of modular exponentiation in the OpenSSL library. The parallel version of Montgomery multiplication using vector SIMD instructions performance increases by more than a factor of 1.5 compared to the implementation in the OpenSSL library in the classical arithmetic logic unit on the Atom platform for 2048-bit moduli. Our implementation of the modular Montgomery multiplication to compute the modular exponentiation has factors 1.8 and 1.4 for the pseudorandom data the base, exp, mod of 1024 bits and 2048 bits compared to the sequential implementation in MPIR library. According to the obtained results of modular exponentiation [24], the MPIR library is faster for large numbers than OpenSSL.

The values of an average execution time of modular exponentiation depend on the computing capabilities in universal computer systems. Testing results was received on two computer systems with different computing capabilities with processors an Intel Core i9-10980XE (18 cores, 36 threads, 3.0GHz) and Intel Core i9-13900K (24 cores, 32 threads, 3.0GHz). The results are presented in Table 1, which contains the values of average execution time (μs microseconds) for 500 trials of the functions *modexp*() and *montgomery_modexp*() using developed Montgomery modular multiplication for computing the modular exponentiation with pseudo-random data of 1024 bits.

Table 1
The average execution time (μs) of the functions of computing the modular exponentiation

Release/x86	Intel Core i9-10980XE	Intel Core i9-13900K
Data bits / trials	1024 / 500	1024 / 500
precompute_ parallel modulo *modexp*()	554	225
precompute_parallel_*montgomery_modexp*()	255	153

The optimal number of threads is 12...16 for fast computation of modular exponentiation for universal computer systems [24].

Therefore, based on the developed Montgomery modular multiplication software the further implementation of the computation of modular exponentiation using multithreaded technologies will provide an opportunity for the efficient computation of modular exponentiation with a fixed base.

Conclusions

In the work is compared and analysed the developed software implementation of the class *MontgomeryArithmetic* in modular exponentiation function. The main directions of software development and outline of the parts of Montgomery modular multiplication for the implementation are presented. Modular exponentiation with a fixed base is implemented using the development of the right-to-left binary exponentiation method with pre-computation of a reduced set of residuals with the use of Montgomery modular multiplication or the usual modular multiplication. The average run time of the computation on multi-core microprocessors of universal computer systems have been defined. As a result, an algorithm with pre-computation of residues for fixed base provides faster computation in average 1,5 times of modular exponentiation using Montgomery modular multiplication compared to the functions of modular exponentiation using the usual modular multiplication.

The scientific novelty of obtained results lies in the implementation of parallelism using multithreading in the function of computing the modular exponentiation based on Montgomery modular multiplication, which is the best among the known modular exponentiation functions of Crypto++, OpenSSL and MPIR libraries for large numbers more than 1K bits.

The practical significance of the work lies in the fact that the obtained results can be successfully applied in modern asymmetric cryptography, for efficient computation of number-theoretic transforms and other computational problems.

Prospects for further research are the parallel implementation of Montgomery Modular Multipliers in the developed function of the modular exponentiation for large numbers using the computation on the video cards.

Acknowledgements

The authors are grateful to Roman Rykmas of the team leader of Uniservice LtdC for participation in testing and discussing the results obtained by the functions of computing the Montgomery modular multiplication.

The work was carried out within the framework of the state-budget scientific-research work of DB "Neuroruh" of the Lviv Polytechnic National University.

References

[1] M. J. Ferrao, K. Kiran V. G, N., M. Megha, Implementation of Modular Reduction and Modular Multiplication Algorithms. IOSR Journal of VLSI and Signal Processing (IOSR-JVSP), 8(6), Ver. I (2018). doi: 10.9790/4200-0806013438.

[2] Barrett reduction, 2023. URL: https://handwiki.org/wiki/Barrett_reduction.

[3] Montgomery Reduction Scheme Functions, 2022. URL: https://www.intel.com/content/www/us/en/docs/ipp-crypto/developer-reference/2022 -2 /montgomery-reduction-scheme-functions.html.

[4] S. Kawamura, Y. Komano, H. Shimizu, T. Yonemura, RNS Montgomery reduction algorithms using quadratic residuosity. Journal of Cryptographic Engineering, 9 (2019). doi: 10.1007/s13389-018-0195-8.

[5] Bing Li, Jinley Wang, Guocheng Ding, Haisheng Fu, Bingjie Lei, Haitao Yang, Jiangang Bi, Shaochong Lei. “A high-performance and low-cost Montgomery modular multiplication based on redundant binary representation.” IEEE Trans. Circuits Syst. II Express Briefs 68.7(2021): 2660-2664. doi:10.1109/tcsii.2021.3053630.

[6] I. Boiarshyn, O. Markovskyi, B. Ostrovska, Organization of parallel execution of modular multiplication to speed up the computational implementation of public-key cryptography, Information, Computing and Intelligent systems 3 (2022). doi:10.20535/2708-4930.3.2022.265418.

[7] J. Bos, S. Friedberger, Faster modular arithmetic for isogeny-based crypto on embedded devices, Journal of Cryptographic Engineering, 10.2 (2020). doi: 10.1007/s13389-019-00214-6.

[8] A. Menezes, J. Oorschot, A. Vanstone, Handbook of Applied Cryptography, Taylor & Francis excl. spl reprint, India, 2018.

[9] R. K. Venkata, S. C. Simranjeet, V. Desalphine, D. Selvakumar, A Low Latency Montgomery Modular Exponentiation, Procedia Computer Science, 171 (2020). doi:10.1016/j.procs.2020.04.087.

[10] S. Fatemi, M. Zare, A. Khavari, M. Maymandi-Nejad, Efficient implementation of digit-serial Montgomery modular multiplier architecture, IET Circuits Devices Syst., 13.7 (2019). doi:10.1049/iet-cds.2018.5182.

[11] S. Srinitha, S. Niveda, S. Rangeetha, V. Kiruthika, A High Speed Montgomery Multiplier used in Security Applications. In: Proceedings of the 3rd International Conference on Signal

Processing and Communication (ICPSC), Coimbatore, India, 2021, pp. 299–303, URL: https://www.proceedings.com/content/059/059276 webtoc.pdf.

[12] B. Buhrow, B. Gilbert, C. Haider, Parallel modular multiplication using 512-bit advanced vector instructions: RSA fault-injection countermeasure via interleaved parallel multiplication, Journal of Cryptographic Engineering 2 (2021). doi: 10.1007/s13389-021-00256-10.

[13] Jinan Ding, Shuguo Li. "A low-latency and low-cost Montgomery modular multiplier based on NLP multiplication." IEEE Trans. Circuits Syst. II Exp. Briefs, 67.7 (2020): 1319-1323. doi:10.1109/TCSII.2019.2932328.

[14] Z. Zhang, P. Zhang, A Scalable Montgomery Modular Multiplication Architecture with Low Area-Time Product Based on Redundant Binary Representation, Electronics 11.3712 (2022). doi: 10.3390/electronics11223712.

[15] M. Issad, B. Boudraa, M. Anane, A. M. Bellemou, Efficient PSoC Implementation of Modular Multiplication and Exponentiation Based on Serial-Parallel Combination, Journal of Circuits, Systems and Computers 28.13 (2019). doi: 10.1142/s0218126619502293.

[16] D. Mukhopadhyay, Improvement over Montgomery Modular Multiplication. Information Systems Security. In: Proceedings of 17th International Conference (ICISS 2021), Patna, India, 2021, pp. 212–217. doi: 10.1007/978-3-030-92571-0_14N.

[17] Drucker, S. Gueron, Fast modular squaring with AVX512IFMA, Report 2018/335, Cryptology ePrint Archive, Preprint. MINOR revision (2018). URL: http://eprint.iacr.org/2018/335.

[18] MPIR: Multiple Precision Integers and Rationals. 2021. URL: http://mpir.org/.

[19] Montgomery Multiplication, 2022. URL: https://cp-algorithms.com/algebra/montgomery_ multiplication.html#implementation.

[20] J. Robert, C. Negre, T. Plantard, Efficient Fixed Base Exponentiation and Scalar Multiplication based on a Multiplicative Splitting Exponent Recoding, Journal of Cryptographic Engineering, 9.2 (2019). doi: 10.1007/s13389-018-0196-7.

[21] N. Emmart, F. Zhengt, C. Weems, Faster modular exponentiation using double precision floating point arithmetic on the GPU. In: Proceedings IEEE 25th Symposium on Computer Arithmetic (ARITH), Amherst, MA, USA, 2018, pp. 130–137. doi: 10.1109/ARITH.2018.8464792.

[22] OpenSSL . Cryptography and SSL/TLS Toolkit, 2024. URL: http://www.openssl.org/.

[23] I. Prots'ko O. Gryshchuk, The Modular Exponentiation with precomputation of reduced set of resedues for fixed-base, Radio Electronics, Computer Science, Control 1 (2022). doi: 10.15588/1607-3274-2022-1-7.

[24] I. Prots'ko, O. Gryshchuk, V. Riznyk, Efficient Multithreading Computation of Modular Exponentiation with Pre-computation of Residues for Fixed-base. In: Proceedings of Sixth International Workshop on Computer Modeling and Intelligent Systems (CMIS 2023), Zaporizhzhia, Ukraine, 2023, pp.224-234. doi:10.32782/cmis/3392-19.

[25] J. Bos, P. Montgomery, Montgomery arithmetic from a software perspective, in: J. W. Bos, A. K. Lenstra, (Ed.), Topics in Computational Number Theory Inspired by Peter L. Montgomery, 1st. ed., Cambridge University Press, 2017, pp.10-39. doi: 10.1017/9781316271575.