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

Ihor Prots'ko¹, Oleksandr Gryshchuk²

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

D 0000-0002-3514-9265 (I. Prots'ko); 0000-0001-8744-4242 (O. Gryshchuk)

^{© 2024} Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

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

$$TR^{-1} \mod m$$
,

(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 $R \cdot R^{-1} \mod m = 1$.

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

$$Rv + mv = 1. \tag{2}$$

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

$$R = r^k = 2^k, \tag{3}$$

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 μ =-*m*⁻¹mod *R* for odd values *m* and *R*=2^{*k*} is presented in the Fig. 1.

Algorithm. Compute Montgomery constant μ =- m -1mod R
Input : Odd integer <i>m</i> and $R=2^k$
$\underline{\text{Output}}: \mu = -m^{-1} \mod R$
$y \leftarrow 1$
for $i = 2$ to k do
if $(m y \mod 2^i) \neq 1$ 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*⁻¹mod *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 precomputation is presented in Fig. 2.

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

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 $U = Tm' \mod R$, $m^* m^{-1} \mod R = 1$, then

$$TR^{-1} \mod m \equiv (T + Um) / R.$$
⁽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 $m' = -m^{-1} \mod R$, but rather $m' = -m^{-1} \mod r$.

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

$$z = x \cdot y \mod m. \tag{5}$$

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

 $x' = x R \mod m$ and $y' = y R \mod m$.

The Montgomery reduction of multiplication x'y' is:

 $x' \cdot y' R^{-1} \mod m \equiv (x R \mod m * y R \mod m) / R^{-1} \mod m = x \cdot y R \mod m.$ (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 precomputation. 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 $X Y (R^{-1}) \mod m$ Input : X, Y, m and $R=2^k$, Output : $X Y 2^{-k} \mod m$ S_0 : LSB of $S, x_i \in (x_{n-1} \dots x_1 x_0)_2$

<i>S</i> =0
for $i = 0$ to n do
$S = S + x_i Y;$
$S = S + S_0 m;$
S = S/2;
if $S \ge m$ then $S = S - m$;
return <i>S</i> ;

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 generalpurpose 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)
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 $m'=m^{-1}$ mod 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] :

$$m \cdot x \equiv 1 \mod 2^k \to m \cdot x \cdot (2 - m \cdot x) \equiv 1 \mod 2^{2k}.$$
(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

$$x \cdot R \mod m = x R^2 / R = x \cdot R^2, \tag{8}$$

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 $R^2 \mod m$ 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·b) mod 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

$$y = A^x \mod m. \tag{9}$$

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, ..., 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-1), x.(k-2),...,x.2, x.1, x.0;
- (A²i)_m are blocks of computation of the integer exponent of exponent 2ⁱ of the number A by the module m, i = 0,1,2,..., (k-1);
- *r*.0, *r*.1,..., *r*.*k*-1 are residuals $A^2^i \mod m$, (*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^2) 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 precomputation

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 (μ s microseconds) for 500 and 250 trials of

computing the modular exponentiation for pseudo-random data *base, exp, mod* for 1024 bits and 2048 bits.

Enter number of threads (240) > 12 Terminimum bits 1094 trials 5-80 = Terminimum bits 5-8	🖾 Командний радок × + ~			×	
base = 1272146663975440962529226689357659654098576596540985765565119628880866329675209515892279289818914996688318014966688315173136666 337974268231855652740655657733409274405765218422814084052824794765585918 exp = 142835289786405997178873255697946573556448214084628524065421188224099687322984673543614176852871855347415698252622 57384156389754529824749225178411567278659689136988584344756479483918444179573315771335693592746851644252279738368411597397 57762611944739235586893446318759847488394736819718851471546337925349998164 mod = 3682232565393616758382899474888592740683231485614475648313561935675865892058431345272676782139889436981496978 563960180663359668927340673253250413181966833980664078756085777309156580172345608629471161924548982647158267357756118464079 65325964734156963129827478488592747240833512148561447564083136919365738658920584313452726767821398894369814697 forcompute_montgomery average time = 302 microseconds. precompute_montgomery average time = 135 microseconds. precompute_parallel_montgomery average time = 135 microseconds. precompute_parallel_montgomery average time = 136 microseconds. precompute_source time = 1380 micr	Enter number of threads (264) > 12 ====================================				
0'9'2'0'3'153'53'10'35'2'1415'51'1415'152'556'7'349'2'14475'551'04280'30'2'159'45'7'355'554'7'455'554'554	base = 127210606307544096252922605035765963498576355611962888086526965619752021589291226329611184363940538	1744887	61896	32	
<pre>exp = 14283520978648999717087325556794657355644821408421408224069547322084673523844176582871085509382746851664252279738366841157573575611846415757397 5778263159784578247942515741157672786098015088683344477564794833184417957332157713350938927468516642522797378368411597397 5778263194473925515881387582736973553360118318383921641563379253499951544 mod = 368223265393616765838289047484087924724083351214856164756408135619936573865892058431345272076782139089436981490798 5530618060335506893273007523355306118318330216661780501404921220381528643 mprc_mpute_modulo average time = 30 microseconds. precompute_parallel_montgomery average time = 153 microseconds. precompute_parallel_montgomery average time = 71 microseconds. precompute_parallel_montgomery average time = 7200000000000000000000000000000000000</pre>	8/9/2005/15598554054/211/111451162615961600688606459520000666226558076554592042254552158927728968180149600 33797426823103562740550577334927448756210422810699954369405284794763585910	0021217	1200	00	
<pre>mod = 36822326533361676583228004748408792472408335121485616475640313691936573865892058431345272076782139089436981490798 58300180633596893273067552325201318319065833980664670756068577730915650017234560082947161924548982647158267357750118464079 633252964727345614653189326733536011831830216661780501404921220381528643 mpz_powm average time = 301 microseconds. precompute_montgomery average time = 180 microseconds. precompute_parallel_montgomery average time = 180 microseconds. ====================================</pre>	$exp = 142835209786489997170873255507946573556448214084628524605421188224099687322984673543614176852071053547415698252622 \\ 5738456830754529824719225178413672786090891369885834477564079483918441795733215771335093892746851604252279730368411597397 \\ 577626130400000000000000000000000000000000000$				
50300180633596893273007523252013138190683398066467075600577730915050017234560082947101924548982647158267357750118464079 633252964273456146531893267353360118318332216661708501404921220381528643 mpz_powm average time = 301 microseconds. precompute_montgomery average time = 138 microseconds. precompute_parallel_montgomery average time = 153 microseconds. precompute_parallel_montgomery average time = 153 microseconds. ====================================	mod = 3682232653936167658382890474840879247240833512148561647564031369199365738658920584313452720767821396	8943698	314907	98	
Display=Days average time = 301 microseconds. precompute_modulo average time = 180 microseconds. precompute_parallel_modulo average time = 110 microseconds. precompute_parallel_modulo average time = 71 microseconds. precompute_parallel_montgomery average time = 71 microseconds. precompute_parallel_modulo average time = 71 microseconds. precompute_parallel_modulo average time = 71 microseconds. precompute_parallel_moduloseconds. precompute_straite base = 204098020931211213516565822840229751951320835651346614633177942400833390309831230100711643032845595314134625325028311364 83021531573074763251660460909209703082695735960838862474347168661641109513208597216683293179549253284095 2668425599866449116364092884727533197752224087971594266914272198657863071926847963659147521186234116278941177627452 28342173791692628840738 exp = 266971186674662942893291319268428623712608587189616225423903941285779658834620654647264762656215005844221010903248 de50594758094265664258653437187125765172491285762485391319544665818485397979742680589526761532351678447533647171257623717195202336 910674202766840819523964084411554460264006668359867092442913886632444189834421374281408991589590160597574944920659811536578740429276628085978745494217456891852278551323516784718232827662808597 711874424054824117451346461265848804255929554183515697922429188623722668569894871268972465688543718824117889272929788256 20697118667436848989	5030018060335968932738075232520131381906883980664670756005577730915050017234560082947101924548982647158267	5775011	84640	79	
<pre>precompute_montgomery average time = 382 microseconds. precompute_parallel_modulo average time = 153 microseconds. precompute_parallel_modulo average time = 71 microseconds. precompute_parallel_modulo average time = 71 microseconds. ====================================</pre>	mpz_powm average time = 301 microseconds.				
<pre>precompute_montgomery average time = 180 microseconds. precompute_parallel_montgomery average time = 71 microseconds. ====================================</pre>	precompute_modulo average time = 302 microseconds.				
<pre>precompute_parallel_modulo average time = 153 microseconds. precompute_parallel_montgomery average time = 71 microseconds. ====================================</pre>	precompute_montgomery average time = 180 microseconds.				
<pre>precompute_parallel_montgomery average time = 71 microseconds. ====================================</pre>	precompute_parallel_modulo average time = 153 microseconds.				
<pre>base = 20498029031211213516565822840229761961320835651346614663317794240083339030983123010071164303284509646555769270863 pl2393542850240253530970605341865669423311669663064208489768715890569302112265638341318004988435595314134625325628311364 830215315733074763251664649909209703082695735960838862474347161067863717087046061641109513208507216683290127954925328495 2608425599866449110364928847273533197752224987571954206991427219865786307492684706630599143554615293794640765714058225304 80569236918545088194329833349554571850460220927615973334687338530705406541937500298179827364715521180234116278941177627452 28342173791692628840738 exp = 206971186674602942893291319268428623712608587189616525423903941285779658834620654647264762656215005844221010903248 805994788942650645268533718712576517249128576248539133195446658184539707742688659362765132351678047573496671171360229336 910074202766840819523964084411554460264006668359867024199348668244418903647742281408991589590100597574944920659811539558 910074202766840819523964084411554460264006668359867024199348668244418903647742281408991589590100597574944920659811530558 72187442495482411745134646126894804555929554183515697922429188623722060569894871268972465008089744410455423282766208959 387777042971769880327762031558579804823032389188893339269852036299148231352285535347016859173882001780159272297308256 14885660545884722373680 mod = 393793078364300889899892185920314902063611742217165348963161727358370939139777984108129782707778124892417387555876961711823427165348963161727755837993840812978277073812806 mod = 393793078364300889899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 844432980632123675587498856339647337965506831677871186228119371663269525134974575837993591397777911164435476376471186234721716348963161727735837093591397779841808129775233297111844366 6065795214405971895267 mpz_poum average time = 2088 microseconds. precompute_modulo average time = 1290 microseconds. precompute_motgomery average time = 11050 microseconds. precom</pre>	precompute_parallel_montgomery average time = 71 microseconds.				
base = 20498029031211213516565822840229761961320835651340614663317794240083339030983123010071164303284509406555769278663 91239354285024025535097060534186566942331166966304208489768715890569302112265638341318004988435595314144253252628311364 830215315733074763251664649909209703082095735960838862474347161067863717087046061641109513208507216683299127954925328495 260842559986644911036492884727353319775222498757195420609142721986578630749268470630599143554615293794640755714058225304 805692369185450819432983334955457185046022092761597333468733853070540541037500298179827364715521180234116278941177627452 28342173791692628840738 exp = 206971186674602942893291319268428623712608587189616225423903941285779658834620654647264762656215005844221010903248 8059947889420506452685343718712576517249128576248539133195446658184539707742065863936276513235167804757330671171360229336 910074202776684081952396408441155446022640066683598670241993486682444189036477422814089915895901065975749444920659811530558 7218744424954824117451346461205848045559295541835156979224291886237220605698948712689724650080897444104535423282766208959 3877777042971769880327762033155857980482303238918889333926985203629914823135222855353547016859173882001780159272297308256 14585660545884722373680 mod = 3937930783643008898989892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 8440329865321236755874598744592744633067337966506831678671186228119376669260851219675338956665255148777711164835476370474879 58360269823032675587459527446330673379650606831678671186228119376662608512196753389566952551248777711164835476370474879 58360269823032675587459927446330673379650606831678671186228119376652685481227558370935913977798460819278277075831296 604577521444059718925267 mpz_pomu average time = 2088 microseconds. precompute_modulo average time = 1050 microseconds. precompute_modulo average time = 1050 microseconds. precompute_motgomery average time = 1050 microseconds.	======================================				
83821531577307476325166464990920970308269573596083886247143471610678637170827046061641109513208507216683290127954925328495 260842559986644911036492884727353319775222498757195420699142721986578630749268470630599143554615293794640765714058225304 805692369185450819432983330955457185046022092761597333468733853070540540541037500298179827364715521180234116278941177627452 28342173791692628840738 exp = 2069711866746029428932913192684286237126085871896162254239039412857796588346206546472647626562150058442210109093248 80599478894265064526853318712576517249128576248539133195446658184539709774268865936276513235167804757330671171360229336 910074202766840819523964084411554460264006668359867024199348668244418903647742281408991589590106597574944920659811530558 72187442495482411745134646126884804555929554183515697922429188623722060569894871268972465008089744410455423282766208959 387777042971769880327762033155857980482303238918889333926985203629914823135228553547016859173882001780159272297308256 14885660545884722373680 mod = 3937930783643008889899892185920314992063610143414215425345540627854542150881576612854892278971719513827185241896934 8449432980632123675587498856339640733796650683167867118622811937166326085121967533895669525512487774111648354763670474879 5836026982303267858749885633964013464734873875587806171182324716534896316172773583709359139777978416812978277075831296 56866254202071251222632965862484634354372671824932447221319815753309084769255407490226142154721378943229488820161164938 614827865775795822129061667030446623011885703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_poum average time = 1290 microseconds. precompute_modulo average time = 1290 microseconds. precompute_modulo average time = 1290 microseconds. precompute_modulo average time = 1290 microseconds. precompute_motgomery average time = 1650 microseconds.	base = 204980290312112135165658228402297619613208356513466146633177942400833390309831230100711643032845096 9123935428502402535309706053418656694233116696630642084897687158905693021122656383413180049884355953141346	4655576 2532562	592708 283113	63 64	
260842559986644911036492884727353319775222498757195426099142721986578630749268470630599143554615293794640765714058225304 8056923691854586194329833349554571850460220927615973334687338530705405410375060298179827364715521180234116278941177627452 28342173791692628840738 exp = 206971186674602942893291319268428623712608587189616225423903941285779658834620654647264762656215005844221010903248 80599478894205506452685343718712576517249128576248539133195446658184539707742058059362765132351678047577330671171360229336 910074202766840819523954048411155446022640066683598670241993486682444189036477422814089915895901005975749444920659811530558 72187444249548241174513464612058480455592955418351569792242918862372206056989487126897246500880897444104535423282766208959 3877777049297176988032776203315585798048230323891888933392698520362991482313522285535547016859173882001780159272297308256 14585666945884722373680 mod = 393793078364300889899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 84043298053212367558745892744633067337966506831678671186228119376602608512196753389566652551248777111648354763670474879 58360269823032678587498856339404136473488173875587601711862281193766026085121967533895666525512487777111648354763670474879 583602698230326785874988563394041364734817387555870617118323427166348963161727358870935913977798460819278277075831296 568662542020712512226329658682484634354372671824932447221319815753300884769254074990226142154721378943229428220611614938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_pomm average time = 2088 microseconds. precompute_modulo average time = 1050 microseconds. precompute_modulo average time = 1050 microseconds. precompute_motgomery average time = 1050 microseconds. precompute_notgomery average time = 1050 microseconds.	8302153157330747632516646499092097030826957359608388624743471610678637170870460616411095132085072166832901	2795492	253284	95	
<pre>805692369185450819432983334955457185046022092761597333468733853070540541037500298179827364715521180234116278941177627452 28342173791692628840738 exp = 206971186674602942893291319268428623712608587189616225423903941285779658834620654647264762656215005844221010903248 805904788942050645268534371871257651724912857624853913319544665818453970774205805936276513235167804757330671171360229336 910074202766840819523964084411554460264006668359867024199348668244418903647742281408991589590100597574944920059811538558 7218744249548241174513464612058480455592955418351569792242918862372206056989487126897246500808974441045354238226605999 3877770429717698803277620331558579804823032891888933392698520362991482313522855353547016859173882001780159272297308256 14585660545884722373680 mod = 3937930783643008889899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 84043298053212367587445592744633067337966506831678671186228119376692688121967533895669525512487774111648354763670474879 58360269823022678587498856339640136473488173875558706171182342716634896316172735583709359139773984608192782277075831296 56866254126207125122263296586824846343543726718249324472213198157533008847692554074902261421547213789432294888206161164938 61482786577579582219290166703044662301180570363563387741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mp2_powm average time = 1290 microseconds. precompute_montgomery average time = 1290 microseconds. precompute_montgomery average time = 1290 microseconds.</pre>	260842559986644911036492884727353319775222498757195420699142721986578630749268470630599143554615293794640	6571405	582253	04	
28342173791692628846738 exp = 206971186674602942893291319268428623712608587189616225423903941285779658834620654647264762656215005844221010903248 805904788942650645268534371871257651724912887624853913319544665818453970774205885936276513235167804757330671171360229336 9100742027668408195239640844115544602640066668359867024199348668244418903647742281408991589590100597574944920059811530558 7218744249548241174513464612685484845592955418351569792242918862372206056989487126897246500808974441404535423282766208955 38777704297176988032776203315585798048230323891888933392698520362991482313522855353547016859173882001780159272297308256 1458566054588472237368 mod = 3937930783643008898998992185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 8404329805321236758745587455927446330673379665068316786711862281193766926085121967533895669525512487774111648354763670474879 583602698236326785874988566339640136473488173875558706171182342716634896316172735583709359139773798460812978277075831296 5686625420207125122263296586824846343543726718249324472213198157533000847692554074902261421547213789432294808206161164938 614827865775795822192901667030446623011865703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_powm average time = 1290 microseconds. precompute_montgomery average time = 1290 microseconds. precompute_montgomery average time = 1290 microseconds. precompute_montgomery average time = 1290 microseconds.	8056923691854508194329833349554571850460220927615973334687338530705405410375002981798273647155211802341162	7894117	76274	52	
<pre>exp = 206971186674662942893291319268428623712608587189616225423903941285779658834620654647264762656215005844221010905248 8059047889420506445268534371871257651724912857624853913319544665818453970774205805936276513235167804757330671171360229336 918074202766840819523964084141155446026490066683598670241993486682444189036477422814089915895901065975749414920059811530558 72187442495482411745134646120584804559295541835156979224291886237220605698948712689724650080897444104555423282766208959 387777042971769880327762033155857980482303238918889333926985203629914823135222855353547016859173882001780159272297308256 148585660545884722373680 mod = 393793078364300889899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 84404329805321236755874988563396401364734895706171822427165489631617273558370951397777981460812977821708531296 583602698230326785874988563396401364734895706171822492716349063161727358370935913977797984608129772977841608129778217067831296 568662542020712512226329658682484634354372671824932447221319815753300884769255407490226142154721378943229480820161164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_poum average time = 2088 microseconds. precompute_modulo average time = 1050 microseconds. precompute_modulo average time = 1050 microseconds. precompute_motgomery average time = 1050 microseconds.</pre>	28342173791692628840738				
<pre>8059047889420506445268534371871257651724912857624853913319544665818453970774205805936276513235167804757330671171360229336 910074202766840819523964084411554460264006668359867024199348668244418903647742281408991589590100597574944920059811530558 72187442495482411745134646120584804555929554183515650792242918866327220605698948712689744204650980897444104535423282766208959 387777042971769880327762033155857980482303238918889333926985203629914823135222855353547016859173882001780159272297308256 14585660545884722373680 mod = 393793078364300889899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 840432980532123675587455927446330673379665068316786711862281193766926085121967533895669525512487774111648354763670474879 583602698230326785874988566339640136473488173875558706171182342716634896316172735583709359139773798460819278277075831296 56866254202071251222632965868248463435437267182493244722131981575330088476925540749422614215472137894322948082061164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_powm average time = 1290 microseconds. precompute_modulo average time = 1290 microseconds. precompute_motgomery average time = 1290 microseconds. precompute_motgomery average time = 1291 microseconds. precompute_motgomery average time = 1290 microseconds.</pre>	exp = 2069711866746029428932913192684286237126085871896162254239039412857796588346206546472647626562150058	4422101	09032	48	
91007420276684081952396408441155444602640066683598670214993486682444189036477422814089915895901065975749444920659811538558 72187442495482411745134646120584804555929554183515697922429188623722060569894871268972465008089744410453542382766208959 387777042297176988032776203315585798048230328918889333926985203629914823135222855353547016859173822001780159272297308256 14585660545884722373680 mod = 3937930783643008889899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 840432980652123675587455927446330673379665068316786711862281193766926085121967533895669525512487774111648354763570474879 583602698230326785874988566339640136473488173875558706171182342711634896316172735583709359139773798460819278277075831296 568662541202071251222632965868248463435437267182493244722131981575330088476925540749022614215472137894322948882161164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 60085795214405971895267 mpz_powm average time = 1290 microseconds. precompute_modulo average time = 1290 microseconds. precompute_montgomery average time = 1290 microseconds.	8059047889420506452685343718712576517249128576248539133195446658184539707742058059362765132351678047573306	7117136	602293	36	
7218744249548241174513464612658486445592955418351569792242918862372266656989487126689724656086897444164555423282766268959 38777764297176988632776263315585798648236323891888933392698520362991482313522855353547616859173882061786159272297368256 14885660545884722373680 mod = 393793678364306889899899285285920314992063610143414215425345540627854542156881576612854892278971719513827185241896934 84043298653212367558745587455927446336673379665668316786711862281193766926685121967533895669525512487774111648354763670474879 5836026082369226785874988566339640136473488173875558766171182342716634896316172735583709359139773798460812978277075831296 568662542020712512226329658682484634354372671824932447221319815753300684769255407490226142154721378943622978277075831296 5686625420207125122263296586824846343543726718249324472213198157533006847692554074902261421547213789438229480820161164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_powm average time = 2088 microseconds. precompute_modulo average time = 1290 microseconds. precompute_modulo average time = 1650 microseconds. precompute_montgomery average time = 1650 microseconds.	9100742027668408195239640844115544602640066683598670241993486682444189036477422814089915895901005975749449	2005981	15305	58	
387777042971769880327762033155857980482303238918889333926985203629914823135222855353547016859173882001780159272297308256 mod = 39379307836430088989899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 8404329805321236755874589274463306733796650683167867118622811937669260851219675338956695255124877741116483547636704714879 583602698230326785874988566339640136473488173875558706171182342716634896316172735583709359139773798460819278277075831296 5686625420207125122263296586824846343543726718249324472213198157553300884769255407499024125472137894322948822061164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_poum average time = 2088 microseconds. precompute_modulo average time = 1050 microseconds. precompute_nontgomery average time = 1050 microseconds.	7218744249548241174513464612058480455592955418351569792242918862372206056989487126897246500808974441045354	2328276	62089	59	
14585660545884722373686 mod = 393793078364300889899892185920314902063610143414215425345540627854542150881576612854892278971719513827185241896934 840432986532123675587455927446330673379665068316786711862281193766926885121967533895669525512487774111648354763670474879 583602698230326785874988566339640136473488173875558706171182342716634896316172735583709359139773798460819278277075831296 568662542020712512226329658682484634354372671824932447221319815753300084769255407490226142154721378943229480820161164938 6148278657757958221929016670304466230118057036356935877415159189429402443488358647099049275298972146812975233297111844366 6005795214405971895267 mpz_powm average time = 2088 microseconds. precompute_modulo average time = 1290 microseconds. precompute_montgomery average time = 1050 microseconds.	3877770429917698803277620331558579804823032389188893339269852036299148231352228553535470168591738820017801	5927229	973082	56	
<pre>mod = 393793078364300889899892185920514992063610143414215425345540627854542150881576612854892278971719513827185241896934 8404329806532123675587455874988566339640136473488173875587061711862281193766926085121967533895669525512487774111648354763670474879 583602698239026785874988566339640136473488173875558706171182342716634896316172735583709359139773798460819278277075831296 5686625412020712512226329658682484634354543726718249324472213198157533008847692554074990226142154721378943229488820161164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6085795214405971895267 mpz_powm average time = 2088 microseconds. precompute_montgomery average time = 1290 microseconds. precompute_montgomery average time = 1050 microseconds.</pre>	14585669545884722373680				
840432980532123675587498565339605067337966506831678671186228119376692608512196753389566952551487774111648354763670474879 5836062698230326785874988566339640134673488570617118234271663489631617275558370959513977379846081297782707678831296 568662542020712512226329658682484634354372671824932447221319815753300084769255407490226142154721378943229480820161164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_powm average time = 1200 microseconds. precompute_montgomery average time = 1050 microseconds. precompute_natore time = 1050 microseconds.	mod = 3937930783643008898998921859203149020636101434142154253455406278545421508815766128548922789717195138	2718524	18969	34	
583602698236326785874988566339640136473488173875558706171182342716634896316172735583709359139773798460819278277078831296 56866254202071251226329658682484634354372671824932447221319815753300084769255407490226142154721378943229480820161164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6005795214405971895267 mpz_powm average time = 2088 microseconds. precompute_modulo average time = 1050 microseconds. precompute_montgomery average time = 1050 microseconds.	840432980532123675587455927446330673379665068316786711862281193766926085121967533895669525512487774111648	5476367	04748	79	
568662542626712512226329658682484634354372671824932447221319815753300084769255467496226142154721378943229480820161164938 614827865775795822192901667030446623011805703635693587741515918942940244348835864709049275298972146812975233297111844366 6065795214405971895267 mpz_powm average time = 2088 microseconds. precompute_montgomery average time = 1050 microseconds. precompute_parallel_modulo average time = 411 microseconds.	583602698230326785874988566339640136473488173875558706171182342716634896316172735583709359139773798460819	7827707	/58312	96	
614827865775795822192991667838446623011885783635693587741515918942948244348835864709049275298972146812975233297111844366 6085795214405971895267 mpz_powm average time = 2088 microseconds. precompute_montgomery average time = 1050 microseconds. precompute_parallel_modulo average time = 411 microseconds.	568662542020712512226329658682484634354372671824932447221319815753300084769255407490226142154721378943229	8082016	511649	38	
6005795214405971895267 mpz_powm average time = 2008 microseconds. precompute_modulo average time = 1290 microseconds. precompute_montgomery average time = 1050 microseconds. precompute_parallel_modulo average time = 411 microseconds.	618278657757958221929916670304466230118057036356935877415159189429402443488358647090492752989721468129752	3329711	18443	66	
mpz_powm average time = 2008 microseconds. precompute_modulo average time = 1290 microseconds. precompute_montgomery average time = 1050 microseconds. precompute_parallel_modulo average time = 411 microseconds.	6005795214405971895267				
precompute_modulo average lime = 1290 microseconds. precompute_montgomery average time = 1050 microseconds. precompute_parallel_modulo average time = 411 microseconds.	mp2_powm_average lime = 2008 microseconds.				
precompute_montgomery average time = 1050 microseconds. precompute_parallel_modulo average time = 411 microseconds.	precompute_modulo average time = 1290 microseconds.				
precompute_parallel_modulo average time = 411 microseconds.	precompute_montgomery average time = 1950 microseconds.				
ana annu ta annu ta annu annu an tara a ann aireanna da	precompute_parallel_modulo average time = 411 microseconds.				

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* =1701*µs*;

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 the result of testing average time (Fig. 7) are equal to:

 $modexp()=(1290+71)\mu s=251 \ \mu s;$

 $modexp()=(1050+173)\mu s=1223\mu s;$

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 $454\mu s/251\mu s=1,8$ and $1701\mu s/1223\mu s=1,4$ 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 (μ s) of the functions of computing the modular exponentiation

Release/x86	Intel Core	Intel Core
	i9-10980XE	і9-13900К
Data bits / trials	1024 / 500	1024 / 500
precompute_parallel modulo modexp()	554	225
precompute_parallel_ <i>montgomery_modexp(</i>)	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.