# Parallel Implementation of the RSA Cryptografic Library Using High Precision Arithmetics and MPI

*Author: Maryna Babayeva*
*maryna.babayeva@gmail.com*
*Student-ID: F100185*

**Course: Scientific Computing**
Department of Mathematics
Utrecht University

Lecturer: Rob Bisseling
February 3, 2011

**Abstract**

RSA has been introduced by Rivest, Shamir, and Adleman in 1978 [10]. Since then many efforts have been done to make the algorithm more efficient and less memory intense. Within the scope of the Scientific Computing lecture a library for parallel RSA cryptographic transformation has been implemented using the Message Passing Interface (MPI). This report is based on two preceding reports on parallel prime sieving [1] and parallel computation of $\pi$ [2]. Here, I will describe the implemented routines for parallel RSA transformation and further analyze the speed-up on the parallel high performance architecture Huygens. My analysis revealed that it is not feasible to run RSA in parallel using the approach as described in detail in this report. The main routine within the RSA algorithm is the modular exponentiation. The latter has been implemented using the binary right-to-left method. Here I will show that a different approach needs to be chosen in order to gain from the parallelism of the application and alternative solutions will be discussed.

## 1 Introduction

With todays fast development of virtual applications on the Internet, such as online banking or secure e-mail communication, the need for fast cryptographic transformations has grown significantly. Typically, data in transit needs to be cryptographically protected to provide either confidentiality or authenticity. As those applications are run in real time, there are strict requirements on processing delays introduced by cryptography.

This work will deal with parallel implementation of the RSA algorithm and can be regarded as an extension to the previously mentioned report on parallel high precision calculation of $\pi$, where basic mathematical operations as addition, subtraction, multiplication, and division

have been already implemented [2]. The following parallel routines have been added to the existing high precision mathematical library:

- Modulo
- Greatest Common Divisor
- Modular Multiplicative Inverse
- Modular Exponentiation
- Squaring
- Floor
- Smaller
- Equal

Here I will make a proposal for parallel implementation of RSA cryptographic transformations. As modular exponentiation is the main operation within the encryption, decryption as well as the generation of prime numbers in terms of runtime, the focus will be laid on the evaluation of this routine. Modular exponentiation will be implemented using parallel modulo as well as parallel multiplication. As the modulo operation uses parallel division, which is a very time consuming procedure, the overall behavior of the modular exponentiation is heavily dominated by it. After describing the implemented routines in detail the Results and Discussion sections will deal with the evaluation of the speed-up of paraModExp and then a proposal on how to make it more efficient will be made.

## 1.1 The RSA Algorithm

RSA depends on key pairs. There is a public and a private key. Three main operations are required to implement the RSA algorithm, namely, encryption, decryption, and key generation. Basically, RSA private key consists of three integers ($p$, $q$ and $e$) and the public key consist of two integers ($m$ and $e$) with a key length $n$. The definition of $p$, $q$, $e$ and $m$ values are [4]:

- $p$ and $q$ are two prime numbers of length $n/2$, where $p > q$.
- $m$, the modulus, is a positive integer of length $n$ bits, where $m = pq$.
- $e$, the exponent, is a positive integer $e$. $\phi(m)$ and $e$ are co-prime, $\gcd(e, \phi(m)) = 1$ with $\phi(m) = (p-1)(q-1)$ as the totient of $m$.

The basic cryptographic operation is applied as follows:

**Encryption:** $c = w^e \pmod{m}$ where $w$ is the data in integer format and $w < m$.

**Decryption:** $w = c^d \pmod{m}$ where $d$ is the private exponent which satisfies the congruence relation $de \equiv 1 \pmod{\phi(m)}$ or $d = e^{-1} \pmod{\phi(m)}$.

**Key Generation:** generating two primes $p$ and $q$. Simply an odd, random number of length $n/2$ is generated, then the number is tested using a probabilistic test. If the test failed, the value is incremented by two, until the test is succeeded.

# 2 Basic High Precision Mathematical Operations in Parallel

This chapter is mainly based on the book 'Introduction to Algorithms' by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein [4].

## 2.1 High Precision Number Representation and Basic Mathematical Operations

To represent an $n$-digit number, a base-B representation of a floating point number $x$ is the shortest sequence of the digits $x_i$ such that each digit satisfies $0 \leq x_i < B$. The floating point number can thus be stored according to equation 1 as a vector $(x_0, x_1, ..., x_{n-1})$

$$x = sB^e \sum_{i=0}^{n-1} x_i B^i \tag{1}$$

with an exponential shift $e$ and a sign $s$. To prevent possible overflows an unsigned 32-bit integer array will be used to store the significant $x$. For the sign $s \in (-1; 1)$ a signed 8-bit integer will be used. The exponent $e$ will be stored in a 32-bit signed integer. In this report all numbers will be with the base $B = 10$.

As the RSA library will use previously implemented mathematical operations on parallel $\pi$ calculation [2], the high-precision number will be distributed among the processors block-wise. The reasons for this decisions lie in the fact that data shifts are required frequently. In order to keep the communication between the processors low, block distribution has been chosen as the most efficient way for data parallel implementation.

For reasons of simplification the basic high-precision parallel operations (addition etc.) will be symbolized as follows. The pseudocode for parallel modulo operation is described in the appendix 6.1.

| | | | |
|---|---|---|---|
| $\ddot{+}$ | paraAdd | $\ddot{mod}$ | paraMod |
| $\ddot{-}$ | paraSub | $\ddot{<}$ | smaller |
| $\ddot{*}$ | paraMult | $\ddot{=}/\ddot{\neq}$ | =/!= |
| $\ddot{/}$ | paraDiv | $\ddot{\lfloor x \rfloor}$ | paraFloor |

## 2.2 Greatest Common Divisor (GCD)

An efficient way for computing the GCD is the Euclidean algorithm, which is named after the Greek mathematician Euclid.

The GCD of two numbers $x$ and $y$ is the largest number $d = \gcd(x, y)$, that divides both of them without leaving a remainder. The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change, if the smaller number is subtracted from the larger number. Since the larger of the two numbers is reduced, repeating this process gives successively smaller numbers until one of them is zero. When that occurs,

the GCD is the remaining non-zero number. The implementation of the described procedure can be found in the algorithm 2.1.

**Algorithm 2.1:** paraGCD$(x, y, d, s, p)$

*input* : $x$ $n$-digit number
$y$ $n$-digit number
$s$ for the processor ID
$p$ for the number of processors
*output* : $d = gcd(x, y)$

(0) *Store input in auxiliary variables, with* $x < y$
$min \leftarrow x$
$max \leftarrow y$
(1) *perform the Euclidean algorithm*
**while** $min \ddot{\neq} 0$
**do** $\begin{cases} temp \leftarrow min \\ min \leftarrow max \ (\ddot{mod}\ min) \\ max \leftarrow temp \end{cases}$
$d \leftarrow max$
**return** $(d)$

Euclidean algorithm computes the GCD of large numbers efficiently, because it never requires more steps than five times the number of digits $n$ in base 10 of the smaller integer, which was proved by Gabriel Lamé. The number of steps required by the Euclidean algorithm can be approximated by equation 2 [9].

$$Y(n) \approx (12/\pi^2) \ln 2 \ln n + 0.06 \tag{2}$$

Thus, the algorithm has a theoretical complexity of $T_{paraGCD}(n) = Y(n) \cdot T_{paraMod}(n)$.

## 2.3 Modular Multiplicative Inverse

The modular multiplicative inverse of an integer $a$ modulo $b$ is an integer $x$ such that $a^{-1} \equiv x \pmod{b}$. The modular multiplicative inverse of $a$ modulo $b$ can be found with the extended Euclidean algorithm. The algorithm finds solutions to Bézout's identity $ax + by = \gcd(a, b)$, where $a$, $b$ are given and $x$, $y$ are the integers to determine. Since the modular multiplicative inverse is the solution to $ax \equiv 1 \pmod{b}$, then $b$ is a divisor of $ax-1$. This leads to $ax-1 = yb$ and then to $ax - by = 1$ with $a$ and $b$ given, $x$ the inverse, and $y$ an integer multiple that is not of interest. Additionally, It is necessary that $a$ is coprime to the modulus $b$, or the inverse

won't exist. Algorithm 2.2 describes the multiplicative inverse operation.

**Algorithm 2.2:** paraExtEuclid$(a, b, x, y, s, p)$

*input* : $a$, $b$ $n$-digit numbers
$x$, $y$ $n$-digit numbers
$s$ for the processor ID
$p$ for the number of processors
*output* : $1 = ax - by$

(0) *Store input in auxiliary variables*
$a_T \leftarrow a$
$b_T \leftarrow b$
$x \leftarrow 0,\ x_{last} \leftarrow 1$
$y \leftarrow 1,\ y_{last} \leftarrow 0$

(1) *perform the extended Euclidean algorithm and initialize x, y, $x_{last}$, $y_{last}$*
**while** $b_T \ddot{\neq} 0$
**do**
$temp \leftarrow b_T$
$q \leftarrow \ddot{\lfloor} a_T \ddot{/} b_T \ddot{\rfloor}$
$b_T \leftarrow a_T \ddot{mod} b_T$
$a_T \leftarrow temp$
$temp \leftarrow x$
$x \leftarrow x_{last} \ddot{-} q \ddot{*} x$
$x_{last} \leftarrow temp$
$temp \leftarrow y$
$y \leftarrow y_{last} \ddot{-} q \ddot{*} y$
$y_{last} \leftarrow temp$
$x \leftarrow x_{last}$
$y \leftarrow y_{last}$
**return** $(x, y)$

This algorithm has the same number of steps as the standard Euclidean algorithm. Thus the total cost is approximated by $T_{paraExtEucl}(n) = Y(n)(T_{paraMod}(n) + 2T_{paraSub}(n) + 2T_{paraMult}(n) + T_{paraDiv}(n))$.

## 2.4 Modular Exponentiation

Modular exponentiation is a type of exponentiation performed over a modulus. Doing a modular exponentiation means calculating the remainder when dividing by a positive integer $m$, called a modulus. A positive integer $b$, which will be referred to as the base, raised to the $e^{th}$ power. In other words, problems take the form where given base $b$, exponent $e$, and modulus $m$, $c$ will be calculated such that $c \equiv b^e \pmod{m}$.

To speed things up, the binary exponentiation method (also known as the square-and-multiply method) will be implemented [9]. The basic idea of the binary exponentiation method is to compute the $b^e$ using the binary expression of exponent $e$. Thus, the exponentiation operation is broken into a series of squaring and multiplication operations, in order to keep the numbers involved small.

To achieve the described behavior, $e$ will have the form

$$e = \sum_{i=0}^{n-1} a_i 2^i \tag{3}$$

Here the length of $e$ is $n$ bits. $a_i$ can take the value 0 or 1 for any $i$ such that $0 \leq i < n-1$ and by definition $a_{n-1} = 1$.

The exponentiation can then be written as:

$$b^e = b^{\left(\sum_{i=0}^{n-1} a_i 2^i\right)} = \prod_{i=0}^{n-1} \left(b^{2^i}\right)^{a_i} \tag{4}$$

The solution $c$ becomes

$$c \equiv \prod_{i=0}^{n-1} \left(b^{2^i}\right)^{a_i} \pmod{m} \tag{5}$$

There are two commonly used algorithms how the binary method can convert the exponentiation into a series of multiplications, *i.e.*, the least signicant bit (LSB) binary algorithm and the most signicant bit (MSB) binary exponentiation algorithm. The LSB binary exponentiation algorithm computes the exponentiation starting from the least signicant bit position of the exponent $e$ and proceeding to the left, which was my choice for the implementation. By that means it is not needed to transform the exponent from decimal representation into binary, which simplifies the algorithm. The exponent will be divided by 2 in each iteration and then it will tested if the remainder is equal to 1, meaning the number is odd.

**Algorithm 2.3:** paraModExp$(b, e, m, c, s, p)$

*input* : $b$, $e$ $m$ $n$-digit numbers
$s$ for the processor ID
$p$ for the number of processors
*output* : $c = b^e \ddot{mod}\ m$

(0) *Store input in auxiliary variables*
$e_T \leftarrow e$
$b_T \leftarrow b$

(1) *Perform square-and-multiplies*
**while** $e_T \ddot{\neq} 0$
**do**
- **if** $e_T$ is odd
  - **then** $rest \leftarrow 1$
  - **else** $rest \leftarrow 0$
- $e_T \leftarrow e_T \ddot{/} 2$
- **if** $rest = 1$
  - **then** $c \leftarrow (c\ q \ddot{*} b_T) \ddot{mod}\ m$
- $b_T \leftarrow b_T^2 \ddot{mod}\ m$

**return** $(c)$

It is to mention that the square operation is a modified parallel multiplication, where one Fourier transform (FFT) and one data redistribution is skipped. That can be done, as both factors are the same and it is redundant to calculate the same values twice (see [2] for further reference on parallel multiplication using the FFT). The cost of a squaring operation can be approximated by $T_{paraSq}(n) = T_{paraAdd} + 2T_{FFT}(n) + 2n/pg + 2l$ with $g$, and $l$ being architecture specific parameters of a supercomputer (see [3] for more information). The overall cost of the algorithm in an average case will be given by

$$T_{paraModExp}(n) = \frac{\log_2 e}{2}(T_{paraMult}(n) + T_{paraMod}(n)) + \log_2 e(T_{paraSq}(n) + T_{paraMod}(n))$$

# 3 RSA Operations

## 3.1 Random Prime Generation

According to Joye *el al.* [7] the classic prime number generator picks first a random candidate $q$, which is already a co-prime to a small prime $p_s$. Second, if $T(q) = false$ then $q$ is updated as $q = q + 2$.

**Algorithm 3.1:** PRIMEGEN$(n, p_s, q, s, p)$

*input* : $p_s$ small prime
$n$ number of digits for the prime
$s$ for the processor ID
$p$ for the number of processors
*output* : random prime $q$ with $n$ digits

(0) *Pick a random n-digit odd number q,* $\gcd(q, p_s) = 1$
**while** $\gcd(q, p_s) \neq 1$
**do** $q \leftarrow$ RAND$(n)$
**comment:** n is number of digits
(1) *Increment q until it becomes prime*
**while** TEST$(q) =$ **false**
**do** $q \leftarrow q + 2$
**return** $(q)$

To test whether a number is a prime I have chosen the Miller–Rabin primality test as described by D. Knuth in [8]. The Miller–Rabin test relies on a set of equalities that hold true for prime values, then checks whether or not they hold for a number that we want to test for primality. If $q$ is composite then the MillerRabin primality test declares $q$ probably prime with a probability at most $4^{-k}$.

The Miller–Rabin primality test is based on the contrapositive claim. That, if we can find an $a$ such that

$$a^d \neq 1 \pmod{q} \tag{6}$$

and

$$a^{2^r d} \neq -1 \pmod{q} \forall r, 0 \leq r \leq s - 1 \tag{7}$$

then $a$ is a witness for the compositeness of $q$. Otherwise $a$ is called a strong liar, and $q$ is a strong probable prime to base $a$. The term 'strong liar' refers to the case where $q$ is composite, but nevertheless the equations hold as they would for a prime.

For every odd composite $q$, there are many witnesses $a$. However, no simple way of generating such an $a$ is known. The solution is to make the test probabilistic by choosing a non-zero $a$, such that $2 < a < q$ randomly, and check whether or not it is a witness for the compositeness of $q$. If $q$ is composite, most of the choices for $a$ will be witnesses, and the test will detect $q$ as composite with high probability. There is, nevertheless, a small chance that an $a$ will be

hit, which is a strong liar for $q$. The probability of such error may be reduced by repeating the test for several independently chosen $a$.

**Algorithm 3.2:** MillerRabin$(q, a, s, p)$

*input* : $q$ odd number to test for primarility
$a$ a witness for compositeness/primarility
$s$ for the processor ID
$p$ for the number of processors
*output* : composite or prime

(0) *Generate $s$ and $d$*
$d \leftarrow q - 1$
**while** $d$ is odd
**do** $\begin{cases} d \leftarrow d \stackrel{..}{/} 2 \\ s \leftarrow s + 1 \end{cases}$

(1) *Get test parameter $x$*
$x \leftarrow a^d \stackrel{..}{mod}\ q$

(1) *Now the test itself*
**if** $x \stackrel{..}{=} 1$ **or** $x \stackrel{..}{=} q \stackrel{..}{-} 1$
**then return** (*prime*)
**else** $\begin{cases} \textbf{for } r \leftarrow 0 \textbf{ to } s - 1 \\ \quad \textbf{do} \begin{cases} x \leftarrow x^2 \stackrel{..}{mod}\ q \\ \textbf{if } x = 1 \\ \quad \textbf{then return } (\textit{composite}) \quad \textbf{return } (\textit{composite}) \\ \textbf{if } x = q \stackrel{..}{-} 1 \\ \quad \textbf{then return } (\textit{prime}) \end{cases} \end{cases}$

In addition to the Miller–Rabin primarility test, which is described by the algorithm 3.3, $q$ will be also tested against a small number of primes $< 10^3$ to make it more efficient. This list of small primes is being generated by a parallel version of the Sieve of Eratosthenes as discussed in [1]. By adding this step, the paraModExp call during the Miller-Rabin test may be omitted if $q$ can be detected as composite already before.

**Algorithm 3.3:** ISPRIME$(q, primes, k, s, p)$

*input* : $q$ odd number to test for primarility
$primes$ a list of small primes $< 10^3$
$k$ probability requirement
$s$ for the processor ID
$p$ for the number of processors
*output* : **true** or **false**

(0) *Test q against small primes first*
**while** $primes_i \neq 0$
**do**
$rem \leftarrow q \ddot{m}od\ prime_i$
**if** $rem \ddot{=} 0$
**then return** ( **false** )
**else** $i \leftarrow i + 1$

**while** $k > 0$
**do**
(1) *Pick a random number a* $(2 < a < q)$
**while** $a \ddot{<} 2$ **or** $a \ddot{>} q$
**do** $a \leftarrow$ RAND$(n)$ **comment:** n is number of digits

(2) *Run compositeness test*
**if** MILLERRABIN$(q, a, p, s) = composite$
**then return** ( **false** )
**return** ( **true** )

## 3.2 Key Generation for RSA

RSA involves a public key pair and a private key pair. The public keys can be known to everyone and is used for encrypting messages. Messages encrypted with the public keys can only be decrypted using the private key pair. The keys for the RSA algorithm are generated as shown by the pseudocode of the algorithm 3.4. It is to mention that not only the keys $d$ and $m$ are calculated by this routine, but also three additional values $d_P$, $d_Q$, and $q_{Inv}$ for an efficient decryption with the Chinese Remainder Theorem (CRT). This will be described in the following section on decryption in more detailed way.

**Algorithm 3.4:** RSA_genKey$(n, e, p, q, d, m, d_P, d_Q, q_{Inv})$

*input* : $n$ number of decimal digits for the keys
$e$ public key $e$ as a preset value (will be set to 65537)
*output* : $p$, $q$ primes with length $n$
$d$, $m$ private and public keys
$d_P$, $d_Q$, $q_{Inv}$ pre-computed values for decryption with CRT

(0) *Run parallel prime number sieve to find primes up to* $10^3$
$primes \leftarrow$ makePrimeList$(10^3)$

$e \leftarrow 65537$
**while** $\gcd(e, \phi(m) \neq 1)$
**do**
(1) *Generate random primes of length n*
$p \leftarrow$ primeGen$(n, p_s)$
$q \leftarrow$ primeGen$(n, p_s)$
(2) *Calculate the totient function* $\phi(m)$
$\phi(m) = (q-1)(p-1)$

(3) *Compute the keys*
$m \leftarrow qp$
$d \leftarrow e^1 \ mod \ \phi(m)$
$d_P \leftarrow d \ mod \ (p-1)$
$d_Q \leftarrow d \ mod \ (q-1)$
$q_{Inv} \leftarrow q^{-1} \ mod \ p$
**return** $(q, p, d, m, d_P, d_Q, q_{Inv})$

The public key pair consists of the modulus $m$ and the public exponent $e$. The private key pair consists of the private exponent $d$, which must be kept secret, and the modulus $m$. e is set to a prime 65537 = 0x10001, as it the number in binary notation has only 2 bits set and thus will reduce the computational effort of the modulo exponentiation during encryption.

## 3.3 Encryption and Decryption

Encryption is being performed by means of the modular exponentiation, where a message $w$ is being translated into a ciphertext $c = w^e \pmod m$. Decryption on the other hand is a different story. As already mentioned previously, $e$ has been set to a special value with only two bits set in binary representation to reduce the computational effort. The bits in the decryption exponent $d$, however, will not be that convenient and therefore decryption using the standard method of modular exponentiation will take longer than encryption. That is the reason to use a more efficient way to decrypt a message, namely with the help of the chinese remainder theorem as described in [6]. $p$ and $q$ are the primes from the key generation,

$$d_P = d \pmod{(p-1)}$$

$$d_Q = d \pmod{(q-1)}$$

and

$$q_{Inv} = q^{-1} \pmod{p}$$

These values allow to compute the exponentiation $w = c^d \mod m$ more efficiently computed as follows:

$$w_1 = c^{d_P} \pmod{p}$$
$$w_2 = c^{d_Q} \pmod{q}$$
$$h = q_{Inv} * (w_1 - w_2) \pmod{p}$$
$$w = w_2 + h * q$$

This is more efficient than computing $w = c^d \pmod{m}$ even though two modular exponentiations have to be computed. The reason is that these two modular exponentiations both use a smaller exponent and a smaller modulus.

Compared to the non-CRT based decryption CRT-based one is about twice faster. This speed improvement is due to the length reduction of both by two, the exponent and the modulus.

# 4 Experimental Results

Key generation is only carried out occasionally and thus computational efficiency is less of an issue. Therefore, this section will deal with timings for modular exponentiation, as this routine is the most time expensive part of the encryption and decryption. The runtime will be measured on the Huygens supercomputer.

In order to compare the measured timings with theoretically predicted complexity the characteristic parameters of the Huygens architecture were obtained. For that, a benchmark from the *BSPedupack*[1] by Rob Bisseling was used and the parameters as shown in Tab. 1 were obtained.

Table 1: Characteristics of the Huygens supercomputer

| **# of processors** | **r [flops]** | **g [flop units]** | **l [flop units]** |
|---|---|---|---|
| 1 | 195 | 58 | 570 |
| 2 | 194 | 56 | 2007 |
| 4 | 187 | 62 | 4288 |
| 8 | 195 | 57 | 8316 |
| 16 | 194 | 60 | 39138 |
| 32 | 195 | 59 | 42821 |

[1] http://www.staff.science.uu.nl/~bisse101/Software/software.html

For the RSA encryption the following results have been obtained as depicted in Figure 1. It shows the theoretical and real runtimes of the encryption routine $c = msg^e \pmod m$, where $e = 65537$ and with different keysizes $size_{m2} = (128, 256, 512, 1024, 2048)$ in binary representation, or $size_{m10} = (39, 77, 154, 308, 617)$ in decimal representation. The computational precision of the high-precision mathematical library has been set to $PRECISION = (128, 256, 512, 1024, 2048)$ for numbers in decimal notation. This was needed, because during modular exponentiation big numbers with sizes up to $2 * size_m10$ occur, due to a multiplication of the modulo $m$ with itself. To be able to run the encryption and decryption routines a test message $w = 1234$ has been used. The prime numbers $p$ and $q$ for key calculation $m = pq$ have been generated using the previously described prime number generating routine with the Miller–Rabin method for primality tests. The measured runtimes are in a good correspondence with theoretical results.

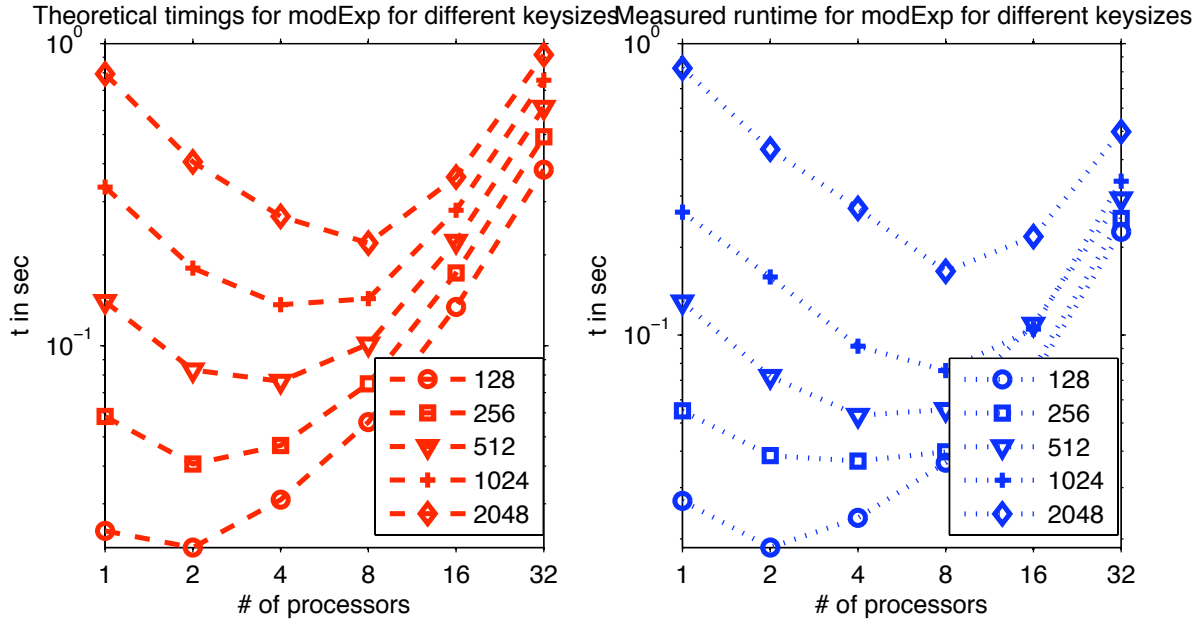


Figure 1: Theoretical and measured runtimes of the RSA encryption of the message $w = 1234$ using modular exponentiation for exponent $e = 65537$ and different keysizes $m$ in binary representation with $size_{m2} = (128, 256, 512, 1024, 2048)$ on up to 32 processors.

Concerning the speed-up, the following conclusion can be made. The algorithm proposed here does not scale well for tested keysizes as Figure 2 shows. It can be seen that the algorithm needs even more time when more than 1 processor is being used. The explanation for this behavior lies in the fact that the the communication introduces additional cost to the algorithm and even outweighs the gain in the computation time when the work is being split among several processors. Although, it can be observed that some gain could be made, but just for sufficiently long keys. I assume that a perfect speed-up for up to 32 processors could be observed when longer keysizes would be tested. However, the use of very long keysizes $> 4096$ bits is not feasible, as the decryption even with CRT then takes too long (i.e., $> 560$ seconds for 1 processor). In order to justify the use of CRT, the runtime of the RSA decryption without CRT modification and CRT modified version has been compared. Figure 3 proves

that indeed by using CRT, half the time is needed due to smaller exponent and modulus compared to non-CRT decryption.

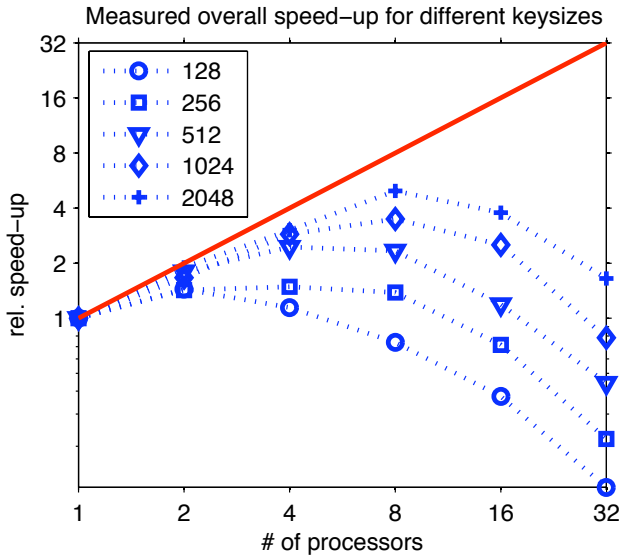


Figure 2: Relative speed-up for encrypting a $msg = 1234$ to get a ciphered value $c = w^e \pmod m$, where $e = 65537$ and $m$ in binary representation with $size_{m2} = (128, 256, 512, 1024, 2048)$ on up to 32 processors.

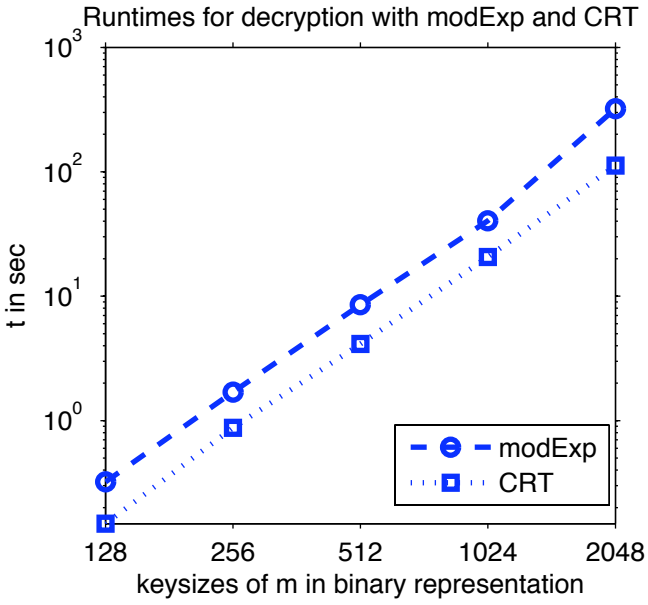


Figure 3: Measured runtimes for the decryption routine using the Chinese Remainder Theorem (CRT) and straight forward modular exponentiation for different keysizes of $m$. Decryption with paraModExp has twice as long runtime compared to decryption with CRT on 1 processing unit.

# 5 Discussion

To parallelize the RSA transformation routines I have implemented the high-precision library for mathematical operations. The basic set of parallel operations has been described in a preceding report on parallel $\pi$ calculation with the Brent-Salamin algorithm [2]. In this project I have extended this library by modular exponentiation, modularization, greatest common divisor (GCD), modular inverse, and additional functions. GCD and modular inverse have been implemented using the Eucledean algorithm and the extended Euclidean algorithm, respectively. These additional operations were needed to calculate the public and private key pairs and also to generate random prime numbers $p$ and $q$ of desired length $n$. To do so, the Miller–Rabin primality test has been used. Also the previous work on parallelization of the Sieve of Eratosthenes has been utilized. Before applying the Miller–Rabin primality test to a random number, it has been tested for compositeness against a list of primes up to $10^3$. By this means many numbers already have been declared as composites, before running a computationally expensive Miller–Rabin test, which uses modular exponentiation.

Further, the runtimes of the decryption routine with straight forward modular exponentiation versus exponentiation with CRT has been evaluated. As expected, the use of the CRT accelerates the RSA decryption by a factor of two. There is a possibility to increase the efficiency of this routine even further by dynamically adjusting the precision of the used numbers, and thus decreasing the length of the arrays where the numbers are being stored and decreasing the number of iterations during addition, subtraction, and multiplication, respectively. This adjustment can only be made, because the numbers involved are smaller in size compared to the standard modular exponentiation approach.

It could be shown that the proposed algorithm for modular exponentiation with the LSB binary exponentiation or binary right-to-left method is not well suited for parallel implementation. The algorithm does not only have a bad scalability, but the runtime even increases, when more processors are used. This is easily explained by the fact that a costly division is generously used and the numbers involved are not large enough to ensure that the computational gain compensates for the communication overhead.

To overcome the above mentioned problem, a different modular exponentiation routine can be used, as this operation is the 'heart' of the RSA transformations. The Montgomery modular exponentiation might be a suitable candidate, as recently described by S. Fleissner [5]. The Montgomery method also uses division, but this division is reduced to the division by the base $B$ of the high-precision number. This division is realized by just adjusting the exponent, thus without any computational cost at all. Although, Montgomery exponentiation method requires a transformation to Montgomery representation (Montgomery reduction) in the beginning and in the end of the routine, and the same number of iterations will be needed, the method should be still faster as each iteration is less time consuming.

## References


[1] Maryna Babayeva. Parallelizing the Sieve of Eratosthenes. Technical report, Utrecht University, November 2010.

[2] Maryna Babayeva. Parallel Computation of $\pi$ Using High Precision Arithmetics. Technical report, Utrecht University, January 2011.

[3] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[5] Sebastian Fleissner. GPU-Accelerated Montgomery Exponentiation. In *Proceedings of the 7th international conference on Computational Science, Part I: ICCS 2007*, ICCS '07, pages 213–220, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] J. Grosschädel. The chinese remainder theorem and its application in a high-speed rsa crypto chip. *Computer Security Applications Conference, Annual*, 0:384, 2000.

[7] Marc Joye, Pascal Paillier, and Serge Vaudenay. Efficient generation of prime numbers, 2000.

[8] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1981.

[9] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley Professional, 3 edition, July 1997.

[10] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126, February 1978.

# 6 Appendix

## 6.1 Modulo

**Algorithm 6.1:** paraMod$(x, y, res, s, p)$

*input* : $x$ $n$-digit number
$y$ $n$-digit number
$s$ for the processor ID
$p$ for the number of processors
*output* : $res = x \mod y$

(0) *Determine correct operation according to the signs*
$res \leftarrow x \ddot{/} y$
**if** $res.sign =$ **true**
**then**
- **if** $res \ddot{<} 1$
  - **then** $res \leftarrow x \ddot{+} y$
    **return** $(res)$
  - **else** $res \leftarrow \ddot{\lfloor} res \ddot{\rfloor} \ddot{+} 1$

**if** $res.sign \neq$ **true**
**then**
- **if** $res \ddot{<} 1$
  - **then** $res \leftarrow x$
    **return** $(res)$
  - **else** $res \leftarrow \ddot{\lfloor} res \ddot{\rfloor}$

(1) *Multiply the result with modulus $y$ and subtract the product from $x$ to obtain the remainder*
$res \leftarrow res \ddot{*} y$
$res \leftarrow x \ddot{-} res$
**return** $(res)$

The theoretical cost of the modulo operation is dominated by the parallel division operation. Thus the total cost can be approximated by $T_{paraMod}(n) = T_{paraDiv}(n) + T_{paraMult}(n) + T_{paraSub}(n)$. The mentioned complexities for parallel division, multiplication and subtraction can be found in [2].