Parallel approximation of RSA encryption

Saint-Jean A.O. Djungu[1]

[1] **Mathematics and Informatics Department, Faculty of Sciences, University of Kinshasa**

Kinshasa, R.D. Congo

Abstract

RSA is the most popular public key cryptography algorithm. Therefore, one of the major weaknesses of RSA is that it requires a lot of computing time compared to secret hey cryptography algorithms. For example in hardware, some authors claim that RSA is around 1000 times slower than DES (Data Encryption Standard).

In addition, current consumer architectures embed several computing units, distributed on processors and possibly on graphics cards. These resources are now easily exploitable thanks to parallel programming interfaces like OpenMP or CUDA. This article proposes parallel versions of RSA allowing to take advantage of the whole of the computing resources, in particular on multi-core architectures with shared memory.

***Keywords**: Cryptography, encryption, RSA, parallel algorithm*

1. Introduction

An encryption algorithm transforms a message, called plain text, into an encrypted text that will only be readable by its legitimate recipient [5, 11]. As indicated in Fig. 1, this transformation is carried out by an encryption function parameterized by an encryption key. A privileged interlocutor can then decrypt the message using the decryption function if he knows the corresponding decryption key.

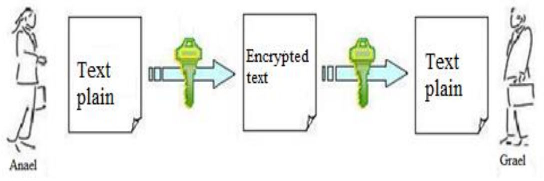

Fig. 1 : Communication principle

Described in 1978 by R.L. Rivest, A. Shamir and L.M. Adleman [10], RSA is the most widely used public key system. It is not strictly speaking a standard but its use is described and recommended in a large number of official standards, in particular for banking applications.

RSA is an asymmetric cryptography algorithm, widely used in electronic commerce, and more generally for exchanging confidential data on the Internet.

With RSA, each user has a couple of keys, a public key (or encryption), which he generally makes available to everyone in a directory, for example, and a secret key (or decryption), known to him alone. So, to send a confidential message to Grael, Anael encrypts the clear message using Grael's public key. Only the latter, using the corresponding secret key, can decrypt the received message.

In many communications, data confidentiality matters little but it is necessary to ascertain their origin and their integrity, that is to say to verify that they have not been modified during transmission.

The RSA algorithm can be used to provide:

- confidentiality: only the owner of the private key can read the message encrypted with the corresponding public key.
- non-alteration and non-repudiation: only the owner of the private key can sign a message (with the private key). A signature decrypted with the public key will therefore prove the authenticity of the message.

This article is organized as follows: section 2 presents the various basic operations related to modular arithmetic. In section 3, the procedure for generating the RSA parameters is shown. The mechanisms for stuffing and decomposing messages into blocks are given in section 4. The parallel version of modular exponentiation is the subject of section 5. Next, in section 6, the declination of parallel versions of encryption and RSA decryption. The signing of messages with RSA constitutes the essential of section 7. Finally we conclude and present perspectives for the implementation of our parallel RSA algorithms.

2. Modular arithmetic

Let be a positive integer *n*. The operations of modular arithmetic are defined in the ring of integers $\mathbb{Z}/n\mathbb{Z}$ between elements of this ring. When *n* is prime, $\mathbb{Z}/n\mathbb{Z}$ is a finite field which we will denote by $\mathbb{Z}_n$. In the following, we represent the elements of $\mathbb{Z}/n\mathbb{Z}$ by integers in [*0, n -1*].

By definition, a modular operation consists in calculating the remainder of the Euclidean division of the whole result of the operation by *n* to guarantee that the final result belongs to the interval [*0, n - 1*], that is to say : $\forall \otimes \in \{+, -, \times, /\}$ [7]:

$\otimes$: *[0, n -1] × [0, n -1] → [0, n -1]*

$$a,b \rightarrow a \otimes b - qn \text{ avec } q = \left\lfloor \frac{a \otimes b}{n} \right\rfloor$$

where $\lfloor \dots \rfloor$ is the bottom integer. In the case of an addition, the division is useless because if *a, b <n* then *a + b <2n*. The reduction is thus achieved by a single subtraction.

2.1. Addition and subtraction

Addition and subtraction are the two simplest modular operations. Let *a < n* and *b < n*, then:

$$a + b \bmod n = \begin{cases} a+b & if\ a+b < n \\ a+b-n & \text{otherwise} \end{cases}$$

and

$$a - b \bmod n = \begin{cases} a-b & si\ a-b < n \\ a-b+n & otherwise \end{cases}$$

2.2. Inversion

The *modular inverse* of a relative integer *a* for multiplication modulo *n* is an integer *u* satisfying the equation:

$$au \equiv 1 \bmod n$$

In other words, it is the reverse in the ring of integers modulo *n*. Once thus defined, *u* can be noted a^{-1}, it being implicitly understood that the inversion is modular and is done modulo *n*. The definition is therefore equivalent to:

$$u \equiv a^{-1} \bmod n$$

The inverse of a modulo n exists if and only if a and n are prime to each other, (i.e. if *pgcd(a, n) = 1*). If this inverse exists, the operation of division by *a* modulo *n* is equivalent to multiplication by its inverse. The algorithm 1 allows to calculate the inverse of a [11]:

```
Algorithm 1: Inverse (a, n: integers)
$(a_0, b_0, u_0, u) \leftarrow (n, a, 0, 1)$
$q \leftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
$r \leftarrow a_0 - qb_0$
as long as (r> 0) do {
        $temp \leftarrow (u_0 - qu) \bmod n$
        $u_0 \leftarrow u$
        $u \leftarrow temp$
        $a_0 \leftarrow b_0$
        $b_0 \leftarrow r$
        $q \leftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
        $r \leftarrow a_0 - qb_0$
}
if $b_0 \neq 1$ then
        a has not inverse modulo n
else
      return u
```

2.3. Multiplication of Montgomery

In 1985, Montgomery introduced a very efficient method to perform modular multiplication by defining a new system for representing integers.

Let *n* be the modulo involved in the operation. We will now assume that *n* is an odd number. The number of bits of *n* is the integer *k* such that:

$$2^{k-1} \leq n < 2^k$$

Denote $r = 2^k$. Since *n* is odd, then *r* is prime with n and therefore invertible modulo *n*. We will denote r^1 the inverse of *r* modulo *n*.

Let θ (Montgomery transformation) be the application of $I_n = \{0, 1, \dots, n-1\}$in itself defined by:

$$\theta(a) = a.r \bmod n.$$

This application *θ* (multiplication by *r* modulo *n*) is a bijection of I_n in itself since *r* is invertible modulo *n* and we can write:

$$a = \theta(a).r^{-1} \bmod n.$$

Let *c=a.b mod n*. Then $\theta(c) = \theta(a).\theta(b).r^{-1} \bmod n$. This leads us to calculate *c = a.b mod n* to calculate *θ(a)* and *θ(b)*, to deduce *θ(c)* by the previous formula and finally to find *c* by the inverse Montgomery transformation.

Let us note by $\otimes$ the operation on {*0, 1, …, n-1*} defined by:

$$a \otimes b = a.b.r^{-1} \bmod n.$$

The calculation $a \otimes b$, Montgomery multiplication of a and b, can be performed as follows [3, 8, 9]:

```
Algorithm 2: MultiMontgomery (a, b, n: integers)
Find an integer k such that $2^{k-1} \leq n < 2^k$
$r \leftarrow 2^k$
$r^{-1} \leftarrow Inverse(r, n)$
$s \leftarrow - r^{-1} \bmod r$
```

```
$t \leftarrow ab$
$q \leftarrow s.t \bmod r$
$t \leftarrow (t + q.n)/r$
**if** $t \geq n$ **then** $t \leftarrow t - n$
**return t**
```

In algorithm 2, we can take $k = \lceil log_2\, n \rceil$, where $\lceil \dots \rceil$ denotes the function "upper integer which rounds a number up" [1].

Note that in the description of algorithm 2, we do not divide by *n*, but by *r*, which changes everything since *r* is a power of 2. Of course, for a single operation, this is not interesting since there are several preparatory calculations to do. But if you chain a large number of operations then it becomes a winner.

2.4. Exponentiation

Exponentiation is a crucial operation in the RSA algorithm. To do this, the fast exponentiation algorithm uses a binary decomposition of the exponent $e = (e_{k-1}, \dots, e_0)_2$ where $e_i = 0$ or 1, $0 \leq i \leq k$-1. The algorithm 3 below makes it possible to calculate $z = x^e \bmod n$.

```
**Algorithm 3 : Exponentiation(x, e, n : integers)**
$z \leftarrow 1$
$e \leftarrow (e_{k-1}, \dots, e_0)_2$ where $e_i = 0$ or 1, $0 \leq i \leq k - 1$
for i ← (k-1) up to 0 do{
    $z \leftarrow$ MultiMontgomery(z, z, n)
    **if** $e_i = 1$ **then**
        $z \leftarrow$ MultiMontgomery(z, x, n)
}
**return z**
```

3. Generation of RSA parameters

If encryption is to be performed by Anael, then the key creation step is the responsibility of Grael. It does not intervene with each encryption because the keys can be reused. The primary difficulty, which encryption does not solve, is that Anael is very certain that the public key she holds is that of Grael. The renewal of the keys occurs only if the private key is compromised, or as a precaution after a certain time (which can be counted in years).

Grael must choose p and q, two distinct odd prime numbers and use algorithm 4 to determine the encryption keys:

```
**Algorithm 4 : RSA(p, q : prima numbers)**
$n \leftarrow p*q$
$\varphi \leftarrow (p-1)*(q-1)$
Determine c (encryption exponent) such that $3 < c < \varphi$
$d \leftarrow Inverse(e, \varphi)$
if (d exist) then
    **return (The public encryption key (n, e) and the private decryption key (n, d)**
```

4. Message jam and cutting

Let *m* be the plain text and *l(m)* the length of *m* expressed in bytes. Let *b* be the *block size* of the block encryption in bytes [5].

1. Determine a number *n* such that $1 \leq n \leq b$ and $n + l(m)$ is a multiple of *b*.
2. Complete the plain text by adding *n* bytes, all of value *n*.

When the length of the completed message is a multiple of the block size, the completed plain text is cut into blocks. The plain text *m* is thus transformed into a sequence of blocks $m_1, \dots, m_k$. The number of blocks *k* can be calculated by $\lceil (l(m) + 1)/b \rceil$. In the following, we will assume that the plain text m consists of an integer number of blocks $m_1, \dots, m_k$. After deciphering, the jam must be removed.

5. Parallel arithmetic

In this section, we are interested in the parallelization of exponentiation, the most expensive operation in the RSA algorithm. Note that the parallel implementation under OpenMP or CUDA of Montgomery's multiplication has been the subject of several researches [1, 4, 6, 7].

5.1. Parallel multiplications

It is a question, using a multiprocessor machine or a GPU card, of finding the product of two numbers *a* and *b* [12, 13]. As shown in algorithm 5, we start with the parallelization of the whole multiplication.

```
**Algorithm 5: MultiParallel(a, b: integers)**
$a \leftarrow (a_{k-1}, \dots, a_0)_2$ where $a_i = 0$ or $1$, $0 \leq i \leq k - 1$
$t \leftarrow 0$
**for i ← (k – 1) up to 0 calculate in parallel**
```

```
        $t \leftarrow 2t + a_i*b$
return t
```

For modular multiplication, the parallel version of algorithm 2 is as follows:

```
Algorithm 6: MultiMontgomeryParallel(a, b, n: integers)
$m \leftarrow \lceil \log_2 n \rceil$
$r \leftarrow 2^m$
$r^{-1} \leftarrow Inverse(r, n)$
$n^* \leftarrow -r^{-1} \bmod r$
$t \leftarrow MultiParallele(a,b)$
$q \leftarrow MultiParallele(t, n) \bmod r$
$u \leftarrow MultiParallel(q,n)$
$t \leftarrow (t + u)/r$
if $t \geq n$ then $t \leftarrow t - n$
return t
```

5.2. Parallel exponentiation

By exploiting the parallel version of Montgomery's multiplication, the parallel version of algorithm 3 is then presented as follows:

```
Algorithme 7 : ExponentiationParallel(x, e, n : entiers)
$z \leftarrow 1$
$e \leftarrow (e_{k-1}, \ldots, e_0)_2$ where $e_i = 0$ ou 1, $0 \leq i \leq k - 1$
for $i \leftarrow (k-1)$ up to 0 do{
    $z \leftarrow MultiMontgomeryParallel(z, z, n)$
    if $e_i = 1$ then
      $z \leftarrow MultiMontgomeryParallel(z, x, n)$
}
return z
```

6. Encryption and decryption

6.1. Encryption

Anael begins by transforming the message m into a series of numbers, for example by replacing the letters and the various symbols used with numbers (from 0 to 255 in the case of the ASCII code). Starting from the message *m* and the pair *(n, e)*, the public encryption key of Grael (cf. Fig. 2), then the message encrypted by Anael will be provided by algorithm 8 below:

```
Algorithm 8 : EncryptionRSA(m, n, e)

Jam of m

$m \leftarrow m_1 || \ldots || m_k$ where the block $m_i$ is of the size n

for $i \leftarrow 1$ to k calculate in parallel

    $c_i \leftarrow ExponentiationParallel(m_i, e, n)$

end

The encrypted message is $c \leftarrow c_1 || \ldots || c_k$ (where || is the symbol of concatenation)
```

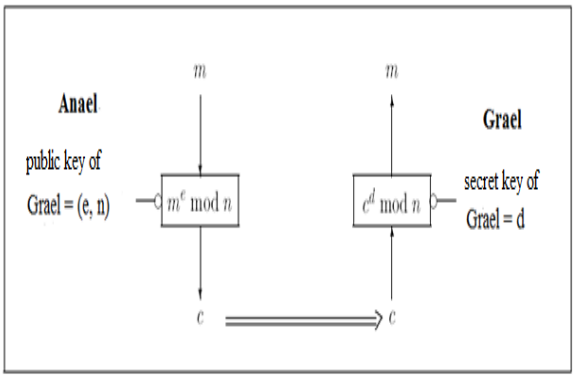

Fig. 2 : RSA public key encryption

6.2. Decryption

Starting from an encrypted message *c* and the pair (*n*, *d*), the secret decryption key, then the unencrypted message will be provided using algorithm 6.

```
Algorithm 9 : DecryptionRSA(c, n, d)

$c \leftarrow c_1 || \ldots || c_k$

for $i \leftarrow 1$ to k calculate in parallel

    $m_i \leftarrow ExponentiationParallel(c_i, d, n)$

fin

The unencrypted message is $m \leftarrow m_1 || \ldots || m_k$
```

7. Digital signature

The digital signature (sometimes called electronic signature) is a mechanism allowing to guarantee the integrity of an electronic document and to authenticate the author, by analogy with the handwritten signature of a paper document [2].

7.1. Principle of the signature

A digital signature process consists in adding to the plain text a small number of bits which depend simultaneously on the message and its author. To obtain the same functionality as the signature that is affixed to the bottom of a text in paper form, it is necessary that everyone can verify a signature but that nobody can imitate it.

A *signature scheme* is therefore composed of a signature function and a verification function. The signature function is configured by a secret key specific to the signatory; it associates a clear message with a signature. The verification function does not require knowledge of any secrets. It allows from the clear message and the signature to verify the authenticity of the latter.

A signature scheme must therefore have a certain number of properties. In particular, it must in practice be impossible to forge a signature: only the holder of the secret key can sign in his name. The signature should no longer be valid if the clear message has been modified; it should be impossible to reuse a signature. Finally, the signatory should not be able to deny having signed a message.

A signature scheme therefore guarantees:

- the identity of the person sending the message;
- the integrity of the data received, ie the assurance that the message was not modified during its transmission;
- non-repudiation of the message, which means that the sender of the message cannot deny being the author.

This is why digital signature processes constitute proof in the same way as handwritten signatures. Their legal value is now recognized by the law of certain countries.

7.2. RSA signature

For the RSA signature scheme, a user signs a message m by applying the RSA decryption function to it with his secret key *d* (cf. Fig. 3). To verify the signature, simply apply the RSA encryption function configured by the associated public key *(n, e)*, and verify that the result of this calculation corresponds to the clear message sent. The conditions imposed on the size of the integers *p* and *q* are the same in the context of the signature as in that of the encryption.

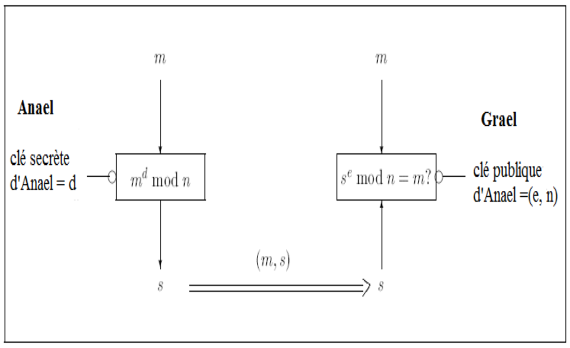

Fig. 3 : RSA signature

Fig. 3 shows that Anael signs a message *m* using the RSA decryption function. Anael is the only person who can create the *s* signature because the *d* key is private. The verification uses the RSA encryption function. Anyone can verify the signature since the RSA encryption function is public. Thus, RSA can be used to encrypt or to sign.

8. Conclusion

To make practical use of the famous RSA encryption algorithm, we have proposed in this article a mechanism which consists of encrypting or deciphering in a concurrent manner the different parts of the message. This same approach can be used to sign messages.

As an extension to this article, we propose the implementation of the different algorithms using the OpenMP directives or the CUDA language and then evaluate the speedup of each of them compared to the sequential version.

References

[1] S. Baktir and E. Sava, *Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors*, 2012.
[2] M. Baudet, *Sécurité des protocoles cryptographiques : aspects logiques et calculatoires*, PhD thesis, l'École Normale Supérieure de Cachan2, 2007.
[3] D. J. Bernstein, *Fast multiplication and its applications*, *Algorithmic number theory*, 44:325–384, 2008.
[4] Z. Chen and P. Schaumont. *A parallel implementation of Montgomery multiplication on multicore systems*: *Algorithm, analysis, and prototype*, IEEE Trans. Comput., 60:1692_1703, December 2011.
[5] N. Ferguson et B. schneier, *Cryptographie en pratique*, éditions Vuibert 2^{e} édition, 2004.
[6] P. Giorgi, L. Imbert and T. Izard, *Parallel modular multiplication on multi-core processors*, 2013.
[7] T. Izard, Opérateurs *arithmétiques parallèles pour la cryptographie asymétrique*, PhD thesis, Universite Montpellier 2, 2011.
[8] P. Montgomery, *Modular multiplication without trial division*, *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
[9] T. Plantard, *Arithmétique modulaire pour la cryptographie*, PhD thesis, Université Montpellier 2, 2005.
[10] R.L. Rivest, A. Shamir and L.M. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, 21, pp. 120-126, 1978
[11] D. R. Stinson, *Cryptographie : théorie et pratique*, éditions Vuibert 2^{e} édition, 2003.
[12] M. Sus and C. Leopold, *Common mistakes in OpenMP and how to avoid them a collection of best*

practices, In *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 312–323. Springer-Verlag, 2008.

[13] R. Szerwinski and T. Guneysu. *Exploiting the power of GPUs for asymmetric cryptography*, In Springer, editor, *Proc. Cryptographic Hardware and Embedded Systems*, volume 5154, pp. 79–99, 2008.