THE PERFORMANCE OF CRYPTOGRAPHIC ALGORITHMS IN THE AGE OF PARALLEL COMPUTING

By

Osama Khalifa

In partial fulfilment of the requirements for the degree of Master of Science

MSc. Computer Services Management

Supervised by:

Dr. Hans-Wolfgang Loidl

HERIOT WATT UNIVERSITY

SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCE

AUGUST 2011

Declaration

I “Osama Khalifa” confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form e.g., ideas, equations, figures, text, tables, programs etc are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: .....

Date: .....

Abstract

This project addresses the problem of enhancing the performance of strong cryptographic algorithms, which are widely used and executed by almost all internet users. The project uses parallel computing as a means to improve performance. Especially nowadays that multi-core computer machines have become commonly available. Since the security level provided by most cryptographic algorithms depends on the difficulty of solving some computational problems, the developments in computer systems manufacturing will threaten people's security. Thus, it is very important to cope with this development and increase the security level by using stronger cryptographic algorithms with longer keys which in return will take longer to encrypt and decrypt data but also a much longer time to hack the cipher text. This project makes it possible for several internet users to use longer keys without being hampered by the bad performance. The resulted parallel algorithm(s) will be assessed by measuring the scalability and speedup features, moreover, it will be able to adapt to the increasing number of cores in a dynamic way.

Acknowledgment

I would like to express my appreciation for all the efforts of my supervisor, Dr. Hans-Wolfgang Loidl, for all the help, suggestions and guidance that he offered to me. I am thankful to him for his advice and the time that he spent to make this work better. I would also like to thank Dr. Ewen Maclean for his support and his willingness to help at any time. I am truly grateful.

Table of Contents

Table of Figures

Chapter 1. Introduction

Nowadays the typical desktop computer contains a multi-core processor. This parallel hardware makes the software designers reconsider the software design to get the most possible computational power from these powerful processors. This can be done by exploiting parallel programming techniques and using them to make the execution of the software components concurrent. Some of the most commonly executed algorithms by computer users nowadays are cryptographic algorithms, which are used to encrypt and decrypt data in order to send it safely and securely over an unsafe environment like the internet.

The project focuses on enhancing the performance of these algorithms in order to speed up the encryption/decryption process and use the new multi-core processors efficiently. The approach adopted to enhance the performance is the use of parallel programming to speed-up the strong cryptographic algorithms. The parallel language technology that is used is OpenMP. Because it uses thread as a main executing unit, and works perfectly in a multi-core computer environment. The cryptographic algorithms which are nominated are RSA and AES. RSA is an asymmetric algorithm or public key algorithm that is considered to be the most secure one available; however, it is slow and needs high computational power. AES is the new standard after DES (the previous encryption standard); AES is a symmetric algorithm and nowadays it is widely used to encrypt and send data via internet, however it is very fast compared with RSA.

By enhancing the performance of these algorithms, they can be easily configured to provide a higher level of security by using longer keys or increasing the computations inside them. This will enhance the security of every internet user all over the world.

1.1. Context

Cryptographic algorithms are widely used to maintain a level of security between parties interacting over the internet. Public key encryption (asymmetric cryptography) and symmetric cryptography are collaborating to make the internet a place people can trust. Trust is essential, especially in the e-Commerce field, where sometimes a whole business is conducted from the internet. Consequently, some protocols like SSL (Secure Sockets Layer), TLS (Transport Layer Security) and HTTPs (Hypertext Transfer Protocol Secure) have been identified, which use cryptographic algorithms without even requiring user interaction. Those protocols make some other technologies appear, such as VPN (Virtual Private Network), which enables a secure network to be made over the internet medium so that only the authenticated people can login to the network. Moreover, all the traffic between the participated computers is encrypted as well.

1.2. Motivation

One approach used to enhance the performance of commonly-used algorithms is to devise a dedicated piece of hardware to solve the problem. However, there are a number of limitations that can be faced, such as:

- This is a very expensive approach that can be adopted by the big companies and organizations, but not by normal computer users.
- Usually there is a scalability limitation of the dedicated hardware, so after a period of time another device will be required.

On the other hand, another approach can be used which exploits parallel programming to solve the problem. This approach may not reach the same performance enhancement level of a dedicated hardware; however, it addresses the first approach limitations by providing a relatively cheap alternative to the dedicated hardware, can be applied even to normal computer users' machines. Furthermore, the scalability is much easier, especially when the parallel algorithm is well-designed. Moreover, a typical computer system nowadays contains a multi-core processor which has eight cores or more, thus, it is a wise decision to make use of all of them to enhance the overall performance.

The RSA algorithm is a very secure, but slow, as it contains heavy mathematical computations, thus parallelising this algorithm will make it perform faster and may increase the security level provided by this algorithm by increasing the key size. RSA now uses keys in 1024 bit length which are not considered secure enough. However, the longer the key used the more security and the slower the performance is obtained. This leads to the need to use parallel computing to increase the security level without adversely affecting the performance. AES encryption/decryption computation performs fairly fast; however, enhancing the performance can be essential for internet servers that provide secure channels with client, since the bottleneck there may be the encryption/decryption of the outgoing stream.

1.3. Technology

The technology, that is required to run the parallel implementation of the cryptographic algorithms, is any multi-core or multi-processor computer. This platform is chosen because it is becoming wide spread among the computer systems. The processors' manufacturing companies have changed the approach of improving the performance of their products. Nowadays, the adopted approach is to increase the number of sub-processing units (cores) inside the processor. This choice makes it very easy to replace the current sequential algorithms, which exist in the security protocols SSL, TLS and HTTPs, with the high performance parallel version without any change to other aspects of the protocol.

The choice of hardware makes some restriction on choosing the parallel technology that will be used in the project in order to implement the parallel version of the cryptographic algorithms. The chosen parallel technology is OpenMP because it is high-level parallel technology and suitable for a shared memory environment. Moreover, this technology can be learnt easily because plenty of resources are available online.

1.4. Aims & Objectives

The overall aim of the project is to speed-up the encryption/decryption process, and to increase the security level of the internet end-user. This can be done by exploiting parallel programming techniques and providing a parallel version of strong cryptographic algorithm(s) which are used nowadays in the security protocols. This overall aim can be identified by several specific aims:

- Increase the security level provided by the strong cryptographic algorithm(s).
- Figure out the best tuned and strongest cryptographic algorithm(s).
- Develop a parallel version of the previous algorithm(s).
- Tune the performance of the parallel algorithm(s), so it will give the best performance over multi-core computer machine.
- Show the performance of the parallel algorithm(s) over a multi-core parallel hardware.

In order to achieve the previous aims, there are some activities provided like:

- Use longer keys to encrypt and decrypt using the strong cryptographic algorithm(s).
- Implement some efficient sequential cryptographic algorithms various versions from recent papers. And run performance benchmark to discover the version with the best performance over multi-core computer machine.
- Develop a prototype of the selected algorithm(s) and run a performance benchmark to discover the problems of this version.
- Use of specialist parallel tools in order to discover the problems more efficiently. And try to solve the show up problems.
- Display the scalability and the speed-up graphs for the resulted algorithms.

1.5. Professional, legal, and ethical issues

The project will be done professionally, by implementing the best performance sequential versions of the algorithms, then prototyping the parallel version. This will guarantee development speed and facing the problems in the parallel design very early in the project life, so dealing with these problems will be easier and more efficient. Then, using the parallel tools available to discover bottlenecks in the algorithms' performance will significantly enhance the results. Finally, after running some practical experiments, some important factors can be decided such as the granularity level of the parallelism. Thus, the resulting algorithm(s) will be highly tuned and guarantee the best possible performance.

The project aims to improve the performance of cryptographic algorithms, and consequently increasing their security level. No part of the project is illegal; on the contrary, this project may be considered to have a positive effect on law application as it provides an opportunity to replace the encryption/decryption algorithm in the internet security protocol SSL, TLS and HTTPs with another more secure one, by adopting longer keys for example. As a result the project does not break any law.

The project does not contain any acts of hate or human rights violations, as the main goal of the project is to increase the security and enhance the performance of the cryptographic security algorithms. Furthermore, all the information provided in this dissertation is referenced in a proper way and any help in the project is acknowledged.

1.6. Project Requirements List

The project has a set of requirement that can be categorised as mandatory and optional requirements.

1.6.1. Mandatory Requirements

The mandatory requirements of the project are:

1. *Identify strong cryptographic algorithms to be parallelised*: in the cryptography field there are two major kinds of algorithms symmetric and asymmetric. The RSA is the most used and the strongest asymmetric algorithm. On the other hand, AES is the new standard (after DES algorithm cracking). Thus, these two algorithms are chosen to be parallelised. However, the final project report should contain description of the choice reasons.
2. *Select parallel technology:* this selection of the parallel technology should be based on selected parallel hardware that will execute the parallel code. This requirement is almost done by selecting OpenMP. However the final project report should contain description of the reasons of this choice.
3. *Sequential version of the selected algorithms:* a sequential implementation of the selected algorithms RSA and AES should be written. This implementation will be used in two ways firstly, as a starting point to implement the parallel version of the algorithms. Secondly, in the evaluation process its execution time will play as a reference to the parallel version speedup figure.
4. *Prototype for all algorithms:* This includes implementing a parallel prototype for all algorithms that are chosen to parallelise. The prototype can give valuable information about the most efficient way to design the concurrent code.
5. *Implement Parallel versions of chosen algorithms:* This requirement includes building initial parallel version of the chosen algorithms RSA and AES. The encryption and decryption algorithms are required for each one.
6. *Tuned parallel version of the chosen algorithms:* A tuned final version depending on the initial parallel version of the selected algorithms. This can be done by using

the parallel performance evaluation tools to discover the bottlenecks in the initial parallel design and then fix them. Eventually, a well tuned parallel version is identified.

7. *Speedup figures for all parallel algorithms:* This comprises gathering the execution time for each parallel algorithm and forming the speedup figures from these data. By measuring the execution time when using specific number of cores, and using these data to draw the speedup figure for a certain algorithm.
8. *Scalability features of parallel algorithms:* It is very important to check the scalability of the parallel code. One important clue can be used by calculating the efficiency of the parallel code using specific number of cores.
9. *Granularity level of each parallel algorithm:* This includes figures of speedup level by fixing the number of cores and changing the number of working threads. This can be used to decide about the best granularity level.
10. *Performance figures against the key size:* This comprises fixing the number of cores and measuring the execution times of a specific algorithm by changing the key size value in the RSA case, or the round number in AES case. With the same figure the performance can be compared with the sequential versions.
11. *The cryptographic algorithms should support the operation modes ECB, CTR:* to provide more security the operation modes should be supported as well. It is important to mention here that these modes support the parallelism.
12. *Final report:* Illustrate the overall process of developing the parallel algorithms in a detailed way.

1.6.2. Optional Requirements

1. *The cryptographic algorithms should support the operation mode CBC:* This involve extending the supported modes and include CBC. This mode does not support parallelism because the processing of the next data block depends on the result of the previous data block. However, the algorithms can still be parallelised but in a different way.
2. *The code should be well documented:* This comprises that the code should be clear and can be understood easily due to the sufficient comments.

Chapter 2. Literature Reviews

2.1. Computer Security

Many years ago, the term *"computer security"* was used to describe the process of keeping the secret information stored in computers secret. However, nowadays with the massive spread of computers all over the world and with increased internet access, computer security now deals with other issues such as protecting users' privacy and intellectual property, which have become serious issues because of the escalating number of people who are using e-commerce technologies and digital banking, as well as due to the more organized online crimes.

This will lead to information security in terms of computers, so information security principles should be satisfied in every organization or even every computer system. Thus, information security is responsible for protecting data from unauthorized access. Information security comprises three main pillars: confidentiality, integrity and availability; some people refer to these pillars as the acronym *"CIA"*.

Confidentiality means preventing unauthorized parties from obtaining information, while integrity is the guarantee that the data has not been changed after it had been sent and before receiving it, i.e. it has not been changed by a third party. Availability guarantees that the data is available at any time it is requested, as long as the requesting party is authorized.

Nowadays, implementing confidentiality and integrity is being done using cryptography, however, achieving availability needs another technique.

In this section, firstly some mathematical background is identified and then a full description of cryptographic algorithms will be presented.

2.2. Mathematical Background

Here, there is some mathematical background which is important for the cryptographic algorithms that are described later in this report in detail. Understanding this mathematical background is essential for understanding these algorithms.

2.2.1. The Field $GF(2^8)$

According to (1), for each prime p to the power n, there is one finite field or Galois field which contains a limited number of elements which equals p^n.

Since a byte contains 7 bits, the representation will be using polynomials as follows:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

E.g. the hexadecimal byte value 65 (01100101_b) can be represented by:

$$x^6 + x^5 + x^2 + 1$$

2.2.1.1. Addition and Subtraction

Addition in the polynomials representation can be achieved by a sum of all coefficients with the same power but by calculating the module 2 for the result i.e. $(1 + 1 = 0)$.

E.g. the sum of 98_h and $6F_h$ is $F7_h$.

$$(x^7 + x^4 + \boldsymbol{x^3}) + (x^6 + x^5 + \boldsymbol{x^3} + x^2 + x + 1) = x^7 + x^6 + x^5 + x^4 + x^2 + x + 1$$

In binary system: $10011000_b + 01101111_b = 11110111_b$

It is clear that addition is applying bitwise XOR operation. However, the subtraction operation has the same rules of addition (1).

2.2.1.2. Multiplication

The multiplication of two polynomials $a(x)$ and $b(x)$ is the normal algebraic products of them modulo to the polynomial $m(x)$ which is called a *"reduction polynomial"*. The modulo operation is done in order to keep the result inside the values of $GF(2^8)$.

The polynomial $m(x)$ should be chosen carefully. One good example can be:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

E.g. the multiplication of 65_h and 43_h is $B7_h$ and evaluated as follows:

$$\begin{aligned}(x^6 + x^5 + x^2 + 1) \cdot (x^6 + x + 1) \\ &= (x^{12} + x^7 + \boldsymbol{x^6}) + (x^{11} + \boldsymbol{x^6} + x^5) + (x^8 + x^3 + x^2) + (x^6 + x + 1) \\ &= x^{12} + x^{11} + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 + x + 1\end{aligned}$$

$$\begin{aligned}x^{12} + x^{11} + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 + x + 1 \; modulo \; x^8 + x^4 + x^3 + x + 1 \\ &= x^7 + x^2 + x + 1\end{aligned}$$

The "modulo" could be calculated in the binary way:

```
1100111101111
100011011    (xor)
0100001011111
 100011011   (xor)
0000010000111
```

Hence, the result is $10000111_b = 87_h$

The neutral element is 01.

And with a suitable choice of $m(x)$ this equation will be always correct:

$$a(x) \times b(x) \equiv 1 (mod \; m(x))$$

This means that there is always an inversion of $a(x)$, which is $b(x) = a^{-1}(x)$

$$a(x) \cdot b(x) = 1$$

(1 pp. 14,15)

2.2.2. Extended Euclidean algorithm

This algorithm is useful to solve the equation

$$a.x + b.y = GCD(a,b), \qquad a,b \in Z^{+}$$

Thus, the inputs are a and b, whereas the outputs are x, y and d where $d = GCD(a,b)$.

2.2.2.1. Algorithm Pseudo code:

```
if b == 0 then
    d = a, x = 1, y = 0, return (x, y, d)
x2 = 1, x1 = 0, y2 = 0, y1 = 1

while b > 0 do
   q = a div b, r = a - qb, x = x2 - qx1, y = y2 - qy1
   a = b, b = r, x2 = x1, x1 = x, y2 = y1, y1 = y

d = a, x = x2, y = y2, return (x, y, d)
```

List 1 The Extended Euclidean Algorithm, (3 p. 67)

2.2.3. Euler's Function

The Euler's function $\emptyset(n)$ for a positive number n is the number of positive integers that are less than the number n and co-prime with it i.e. has with n no common divisors but the number 1. According to (2 p. 7) the $\emptyset(n)$ function is given by the formula:

$$\emptyset(n) = \prod_{j=1}^{r} p_j^{j-1}.(p_j - 1), \qquad n > 1, n = p_1^{k_1}.p_2^{k_2}..p_r^{k_r}$$

Where, the values $p_1^{k_1}.p_2^{k_2}..p_r^{k_r}$ are the prime factorization of n.

In special cases when n is a prime number, the value $\emptyset(n) = n - 1$. Furthermore, for two prime numbers p and q, the Euler's function is:

$$\emptyset(p.q) = (p-1)(q-1)$$

2.2.4. Congruencies and Fermat's Theorem

2.2.4.1. Congruence

When two numbers a and b leave the same remainder after they are divided by another number n, it can be said that a and b are congruent modulo n and is written like:

$$a \equiv b \bmod n$$

A congruent relation has some properties such as: it is possible to multiply both sides of the congruent relation by a fixed number c, thus:

$$a.c \equiv b.c \bmod n$$

Furthermore, it is possible to multiply two congruent relations side by side, so if this equation is true $c \equiv d \bmod n$, then it implies:

$$a.c \equiv b.d \bmod n$$

This leads to:

$$a^k \equiv b^k \bmod n, \qquad k \geq 0$$

Finally, if $a \equiv b \bmod m_1$ and $a \equiv b \bmod m_2$, then:

$$a \equiv b \bmod lcm(m_1, m_2)$$

Where, LCM means the least common multiple. However, if m_1 and m_2 were prime numbers, then $lcm(m_1, m_2) = m_1.m_2$, and in this special case:

$$a \equiv b \bmod m_1.m_2$$

2.2.4.2. Fermat's Theorem

Let p be a prime number then $\forall a \in Z$ and p are two relatively prime numbers, which means that the greatest common divisor for a and p equals 1 ($GCD(a,p) = 1$), this immediately implies:

$$a^{p-1} \equiv 1 \bmod p$$

This is called *"Fermat's Little Theorem"*.

However, Fermat's Little Theorem was generalized by Euler, as follows:

Let m and a be two relatively prime integer numbers and $\emptyset(m)$ be Euler's function value of m; this yields:

$$a^{\emptyset(m)} \equiv 1 \bmod m$$

This is called *"Euler's generalization of Fermat's little Theorem"* or *"Generalized Fermat's Theorem"*.(2 pp. 141,142)

2.3. Cryptography

2.3.1. Definition

"Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication" (3 p. 4). Thus, cryptography is a technique to hide or represent information so nobody but authorized parties can decode it. Encryption is hiding the information whereas decryption is retrieving the information back from the encrypted data. The oldest encrypted text or cipher text was found in ancient Egypt. Cryptography was used in India, Mesopotamia and Greece. Furthermore, it was used a lot in wars e.g. Julius Caesar used to encrypt messages and send them to his troops and also a more recent example is the Enigma machine used by Germany in World War II.(4)(5)

2.3.2. Cryptography Algorithms

There are a lot of algorithms which can be used to encrypt and decrypt data. In the past, cryptographic strength was based on the secrecy of the algorithm; however, a lot of scenarios have shown that this is incorrect, because using the same algorithm all the time, no matter how strong it is, will reduce system security, since as time passes the possibility for cracking the algorithm will increase. Nowadays, the strongest algorithms are well known; however, the secret part is the encryption/decryption key, which obviously can be changed easily on a regular basis.(4)

Modern algorithms can be categorized into two main categories: symmetric key algorithms and public key algorithms.

Symmetric Key Algorithms

The main feature of this type of algorithm is that the decryption key can be calculated from the encryption key, however, they are equal in most practical applications. The major problem here is the key distribution issue.

Symmetric key algorithms can be categorized into two classes: block cipher scheme and stream cipher scheme.

The block cipher encryption scheme breaks the message down into blocks with a specific length, and encrypts each one separately. In its simplest form, when block size equals one, the block cipher scheme becomes a stream cipher scheme. A stream cipher scheme has an advantage over a block cipher scheme, which is that in error prone transmission the sender can resend the wrong data without affecting the decryption process, plus it can

be used easily with telephone conversations and wireless network communications. However, it is considered less secure than the block cipher scheme (3 pp. 16-20).

The most famous symmetric key algorithms are: DES, AES, Blowfish, and 3DES.

2.3.2.1. AES (Rijndael)

In 1997, The US National Institute of Standards and Technology (NIST) announced the new selection of a new encryption standard to replace the old standard DES. After an open competition, the Rijndael algorithm won and became the Advanced Encryption Standard (AES). However, there is a slight difference between AES and Rijndael, which is in the supported values of block and key lengths. Rijndael accepts any multiple of 32 between 128 and 256 for its key or block lengths, whereas in AES the block length is 128 and supported key lengths are 128, 192 or 256 bits only.(1 p. 31)

2.3.2.1.1. Structure of Rijndael

Rijndael is a byte level algorithm, and the bytes are represented using the finite field $GF(2^8)$.

According to (1), Rijndael is a block cipher algorithm; it consists of repetition of an application of a transformation over the "State". Each one transformation application is called round, and the number of rounds N_r is a vital factor in the security level provided by this algorithm. So increasing N_r will make the algorithm more immune to attacks.

The state that was mentioned before, is the block itself represented in a two dimensional matrix, the number of its rows is four, whereas the number of its columns N_b equals the block length divided by 32. In AES N_b is always equals 4.

The key is represented in another matrix with four rows and the number of columns N_k equals the key length divided by 32. In AES N_k can be 4, 6 and 8.

p0	**p4**	**p8**	**p12**
p1	p5	p9	p13
p2	p6	p10	p14
p3	p7	p11	p15

k0	**k4**	**k8**	**k12**	**k16**	**k20**
k1	k5	k9	k13	k17	k21
k2	k6	k10	k14	k18	k22
k3	k7	k11	k15	k19	k23

Figure 1: State and cipher key layout in case of block length 128 bit and key length 192 bit (1 p. 33)

Hence, if the state is represented by $a_{i,j}, 0 \leq i < 4, 0 \leq j < N_b$, then it could be formed from the block bytes using:

$$a_{i,j} = p_{i+4 \cdot j}, 0 \leq i < 4, 0 \leq j < N_b$$

At the end of encryption the cipher text can be extracted using:

$$c_i = a_{i \bmod 4, \frac{i}{4}}, 0 \leq i < 4 \cdot N_b$$

For the decryption the input will be represented to a state using:

$$a_{i,j} = c_{i+4 \cdot j}, 0 \leq i < 4, 0 \leq j < N_b$$

And at the end of decryption the plain text can be taken using:

$$p_i = a_{i \bmod 4, \frac{i}{4}}, 0 \leq i < 4 \cdot N_b$$

Encryption in Rijndael comprises expansion of the key, so all round keys can be taken from the expanded key, initial key addition then $N_r - 1$ application of the round transformation, and finally application of the final round (1 pp. 32,33).

```
Encrypt(State, CipherKey){
  KeyExpansion(CipherKey, ExpandedKey);
  AddRoundKey(State, ExpandedKey[0]);
  for (i = 1; i < Nr; i++){
    RoundTransformation(State, ExpandedKey[i]);
  }
  FinalRound(State, ExpandedKey[Nr]);
}
```

List 2: High-level Encryption algorithm in Rijndael, (Daernen & Rijrnen, 2002, p. 34)

2.3.2.1.1.1. Key Expansion

At this stage the cipher key is expanded to an expanded array which consists of 4 rows and $N_b \cdot (N_r + 1)$ columns. This array will be referred to as $W_{4,N_b \cdot (N_r+1)}$, and in round i key is the sub array from the column i to the column $i \cdot N_b$.

The value of N_r is different depending on the key length and block length, however, in AES the possible N_r values are 10, 12 and 14 for key lengths 128, 192 and 256 respectively.

Furthermore, the key expansion takes the original cipher key as input and applies the following:

- The first N_k columns are copied from the cipher key
- The rest of the columns will be evaluated using recursive calculation. Generally by applying the bitwise xor $\oplus$, between the previous byte and the byte before N_k columns. However, to eliminate similarity if the evaluated byte index is divisible by N_k or satisfies the condition $i \bmod N_k = 4$ when $N_k > 6$ then the

calculation will be done by using a non-linear function $f(x)$.

$$K_i = \begin{cases} K_{i-N_k} \oplus f(K_{i-1}), & i = N_k.n \; or \; (i \; mod \; N_k = 4 \; and \; N_k > 6) \\ K_{i-N_k} \oplus \mathrm{K_{i-1}}, & i \neq N_k.n \end{cases}$$

The non-linear function $f(x)$ is realized by applying a substitution function S_{RD} and addition $\oplus$ with a round constant defined by recursion rule in $GF(2^8)$ as follows:

$$RC(i) \begin{cases} x^0 = 1, & i = 1 \\ x^1 = 2, & i = 2 \\ x.RC(x-1) = x^{i-1}, & i > 2 \end{cases}$$

(1 pp. 43-45)

2.3.2.1.1.2. The Round Transformation

Each typical round comprises four transformations called steps: SubBytes, ShiftRows, MixColumns and AddRoundKey. However, encryption consists of N_r times application of round transformation and eventually application of the FinalRound which is a typical round but without the MixColumns step.

(1 p. 36)

2.3.2.1.1.2.1. SubBytes

Includes application of a non-linear function S_{RD} over the State, this function should satisfy the following criteria:

1. ***Non-Linearity:*** The maximum correlation between input and output and the maximum difference propagation should be as small as possible.
2. ***Algebraic Complexity:*** This means that the algebraic formula of the function in $GF(2^8)$ should be complex.

InvSubBytes is the inverse operation of SubBytes which include applying the inverse function S_{RD}^{-1}.

2.3.2.1.1.2.2. ShiftRows

In this step the bytes of the state are shifted over different values, Row0 is shifted by $C0$ bytes, Row1 by $C1$ bytes, etc. So the byte with position j in the row i is shifted by $j - C_i \; mod \; N_b$ bytes. The values of C_i differ depending on the values of N_b according to the table.

N_b	C_0	C_1	C_2	C_3
4	0	1	2	3
6	0	1	2	3
8	0	1	3	4

Figure 2 Block lengths and corresponding shift offsets, (1 p. 38)

InvShiftRows is the inverse operation of ShiftRows and this is achieved by performing the opposite offsets, so the j_{th}-byte of the row i is shifted by $j + C_i \ mod \ N_b$ bytes.

2.3.2.1.1.2.4. MixColumns

In this step the column's values, which are represented as polynomials in $GF(2^8)$, are recomputed by multiplying with the polynomial $C(x)$.

$$C(x) = 3.x^3 + x^2 + x + 2$$

This transformation is linear in $GF(2^8)$, with a good diffusion power and good performance on various processors. Furthermore, it is invertible in $GF(2^8)$; the polynomial multiplication can be represented as matrix multiplication $b(x) = c(x).a(x) \ (mod \ x^4 + 1)$, then

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

InvMixColumns is the inverse operation of MixColumns and this is achieved by multiplying by the inverse polynomial $c(x)^{-1} = 0\text{B}.x^3 + 0\text{D}.x^2 + 09.x + 0\text{E}$. Thus

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0\text{E} & 0\text{B} & 0\text{D} & 09 \\ 09 & 0\text{E} & 0\text{B} & 0\text{D} \\ 0\text{D} & 09 & 0\text{E} & 0\text{B} \\ 0\text{B} & 0\text{D} & 09 & 0\text{E} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

2.3.2.1.1.2.5. AddRoundKey

In this step an XOR operation is performed between the State and the round key, the round key is taken from the expanded key in a previous step.

2.3.2.1.2. Modes of Operation

The role of the operation modes is to determine how a sequence of input text blocks is encrypted. However, according to (6) the recommended modes are: Electronic Codebook

(ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). Most of these modes cannot be parallelized or can be parallelized in decryption but not encryption, so in this paper only the modes that can be parallelized are discussed.

2.3.2.1.2.1. Electronic Codebook (ECB)

In this mode, the encryption algorithm is applied independently to each block on the input text to calculate a block of the cipher text. The decryption process happens the opposite way. According to (7) this mode is the simplest and the most insecure one, because it produces the same cipher text for the same input text and does not completely blur patterns in the input text.

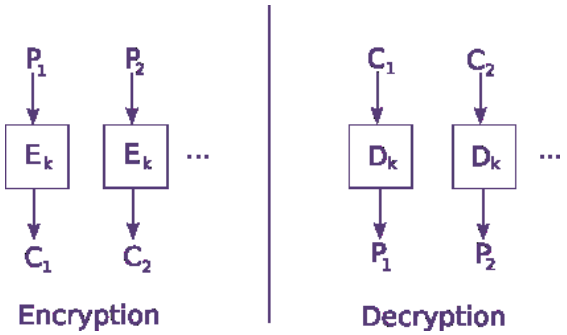

Figure 3 Ciphering in Electronic Codebook Mode

2.3.2.1.2.2. Counter (CTR)

In this mode, input blocks called counters are encrypted to produce output blocks which are then exclusive-ORed with the input text to produce the cipher text, for the last block just the most significant bytes are exclusive-ORed with the output block. The counters T_i should be distinct; however, according to (8) this can be achieved by providing a unique nonce which usually comes from the message itself, and a counter to guarantee the uniqueness of the blocks.

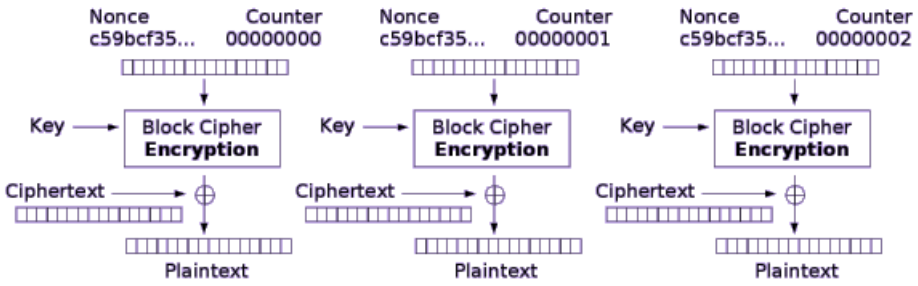

Figure 4 Counter (CTR) mode encryption

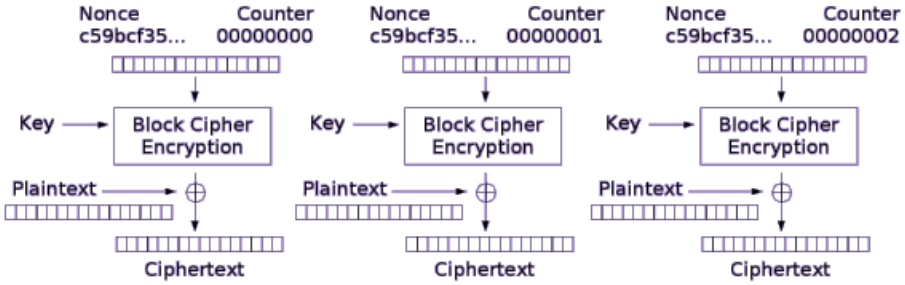

Figure 5 Counter (CTR) mode decryption

2.3.2.1.3. Implementation Aspects

To implement the algorithm in an efficient way let us consider the output of all its steps.

The output of the SubBytes step is $b_{i,j}$, hence:

$$b_{i,j} = S_{RD}[a_{i,j}], \qquad 0 \le i < 4, 0 \le j < N_b$$

The output of the ShiftRows step, which takes as input $b_{i,j}$, is $c_{i,j}$, hence:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} + C_0 \\ b_{1,j} + C_1 \\ b_{2,j} + C_2 \\ b_{3,j} + C_3 \end{bmatrix}, \qquad 0 \le j < N_b$$

Thus, the output of the step MixColumns is $d_{i,j}$, hence:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}, \qquad 0 \le j < N_b$$

Or by substitution each $c_{i,j}$ with its value; and by taking in the consideration that adding C_i must be in modulo of N_b:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} S_{RD}[a_{0,j}+C_0] \\ S_{RD}[a_{1,j}+C_1] \\ S_{RD}[a_{2,j}+C_2] \\ S_{RD}[a_{3,j}+C_3] \end{bmatrix}, \qquad 0 \le j < N_b$$

This can be represented by a combination of four column vectors:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 3 \end{bmatrix} . S_{RD}[a_{0,j}+C_0] \oplus \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix} . S_{RD}[a_{1,j}+C_1] \oplus \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix} . S_{RD}[a_{2,j}+C_2] \oplus \begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix} . S_{RD}[a_{3,j}+C_3],$$

$$0 \leq j < N_b$$

Now by defining the four tables T_i:

$$T_0[a] = \begin{bmatrix} 2.S_{RD}[a] \\ 1.S_{RD}[a] \\ 1.S_{RD}[a] \\ 3.S_{RD}[a] \end{bmatrix}, \quad T_1[a] = \begin{bmatrix} 3.S_{RD}[a] \\ 2.S_{RD}[a] \\ 1.S_{RD}[a] \\ 1.S_{RD}[a] \end{bmatrix}, \quad T_2[a] = \begin{bmatrix} 1.S_{RD}[a] \\ 3.S_{RD}[a] \\ 2.S_{RD}[a] \\ 1.S_{RD}[a] \end{bmatrix},$$

$$T_3[a] = \begin{bmatrix} 1.S_{RD}[a] \\ 1.S_{RD}[a] \\ 3.S_{RD}[a] \\ 2.S_{RD}[a] \end{bmatrix}$$

This leads to the equation that represents the result of SubBytes, ShiftRows and MixColumns:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = T_0[a_{0,j} + C_0] \oplus T_0[a_{1,j} + C_1] \oplus T_0[a_{2,j} + C_2] \oplus T_0[a_{3,j} + C_3],$$

$$0 \leq j < N_b$$

To represent a typical AES round, the state has to be exclusive-ORed with the round key. Thus, storing the four tables makes the performance better, and needs 4KB. However, the last round does not contain the MixColumns step so it can be calculated using the S_{RD} table. The decryption process can be done using look-up tables in exactly the same way. (1 pp. 57-59)

Public Key Algorithms

The main feature of this type of algorithms is that the encryption/decryption key consists of two parts a public key e and a private key d. The public key is shared with other parties so they can use this key to send encrypted messages to the first party, however, only the first party, who knows the private key, can decrypt the encrypted message. This solves the key distribution problem that existed in symmetric key algorithms, as the first party can use the unsecure channel to share its public key.

According to (3 pp. 31,32) there are some advantages for public key algorithms over symmetric key algorithms. Firstly, only the private key needs to be kept secret. The number of necessary keys in a large network may be smaller than the equivalent

symmetric key structure. Furthermore, sometimes the public/private key pair may be kept unchanged for a fairly long time e.g. several years, whereas in symmetric key algorithms the secret key should be changed frequently. However, there are some disadvantages of public key algorithms like the fact that their throughput rate is much slower than the throughput rate of symmetric key algorithms, and their key sizes are larger.

The most well-known public key algorithms are RSA, ElGamal and McEliece etc.

2.3.2.2. RSA

RSA is the most popular asymmetric cryptography algorithm, published in 1978 and named after its inventors Rivest, Shamir, and Adleman by making acronym with their initials. This algorithm depends on the problem of factoring very large integers (more than 300 decimal digits) by using a one-way function principle. RSA is considered the first algorithm which provides asymmetric cryptography along with digital certifying. (7) (3).

2.3.2.2.1. Structure of RSA

In order to use RSA or any other public key algorithm, the party should have a public/private key pair; this key pair should be calculated and cannot be chosen randomly. The key length that is used nowadays is 1024, however the longer the key, the more security obtained. According to (9), the RSA key size 1024 does not provide a sufficient level of security in the period between 2011 and 2019 and the key size should be extended to be 2048 bits in length.

2.3.2.2.1.1. Generating Keys

According to (3 p. 286), in order to generate a key pair the following steps should be adopted:

1. Generate two big unique prime numbers p and q, which have almost the same digit size.
2. Compute the numbers $n = p.q$, and $\emptyset(n) = (p-1).(q-1)$.
3. Choose a random number $1 < e < \emptyset(n)$, which is relatively prime to $\emptyset(n)$, according to (7 p. 183), the value of e is usually 3, 17 or $2^{16} + 1$ because they will make the encryption faster.
4. By using an extended Euclidean algorithm, calculate the value d using the equation $e.d \equiv 1 \ mod \ \emptyset(n)$, or $e.d - \emptyset(n).k = GCD\big(e, \emptyset(n)\big) = 1$
5. The public key is (n, e) whereas the private key is (n, d).

The numbers e and d are called the *"encryption exponent"* and *"decryption exponent"* respectively, whereas the number n is called the *"modulus"*.

Example:

In this example the public and private keys are generated using small numbers to make the process easier to understand, however, in the real world very big numbers are used.

$p = 7, q = 11$

Thus, $n = p.q = 77$ and $\emptyset(n) = (p-1).(q-1) = 60$.

Let $e = 17$, then by using an extended Euclidean algorithm to solve the equation $e.d \equiv 1\ mod\ \emptyset(n)$, or $17.d \equiv 1\ mod\ 60$, yields:

$$d = 53$$

Thus, the public key is (77, 17) and the private key is (77, 53).

2.3.2.2.1.2. Encryption/Decryption

To encrypt a message using RSA, firstly the message should be segmented into integers $0 \leq m \leq 2^{n-1}$, i.e. $n-1$ bits length, then performing the equation:

$$\boldsymbol{c = m^e mod\ n}$$

Thus, c represents the encrypted message which is n bits in length. However, to retrieve the original message m from the encrypted message c, this calculation should be performed:

$$\boldsymbol{m = c^d mod\ n}$$

(3 p. 286)

Example:

By using the previous public and private keys a random message $m = 7$ is encrypted then decrypted using the public key (77, 17) and private key (77, 53):

$$c = m^e mod\ n = 7^{17} mod\ 77 = 28$$

$$m = c^d mod\ n = 28^{53} mod\ 77 = 7$$

2.3.2.2.1.3. RSA mathematical Proof

The two numbers m and p are relative primes, thus applying generalized Fermat's theorem yields:

$$m^{\emptyset(p)} \equiv 1 mod\ p$$

Or

$$m^{p-1} \equiv 1 mod\ p$$

Raising both sides to the power $k(q-1)$ yields:

$$m^{k(p-1)(q-1)} \equiv 1\ mod\ p$$

Multiplying the both sides with m yields:

$$m^{k(p-1)(q-1)+1} \equiv m\ mod\ p$$

Or

$$m^{ed} \equiv m\ mod\ p$$

Using the same strategy we can conclude that:

$$m^{ed} \equiv m\ mod\ q$$

Since p and q are distinct primary numbers, and $p.q = n$, then:

$$m^{ed} \equiv m\ mod\ n$$

Hence,

$$\boldsymbol{c^d \equiv (m^e)^d \equiv m\ mod\ n}$$

2.3.2.2.2. Implementation Aspects

The main challenge in implementing RSA in a computer system is calculating the modular multiplication operations on very large integer numbers $c = m^e mod\ n$. Where m, e and n are very big integers. Because m^e is huge the previous equation can be simplified to be suitable for computer systems

$$c = \Big(\Big(\big((m.m) mod\ n\big).m\Big) mod\ n\Big) \ldots \Big) mod\ n$$

Unfortunately, the division operation to calculate each modulo remains. One method which can be used to calculate the modular multiplication is Montgomery's Algorithm, which will replace the division operation by n to division by the power of 2(10). This will enhance the performance significantly, because the division by power of 2 in the binary system can be very simple. According to (10), Montgomery Modular Algorithm calculates:

$$MM(a,b) = a.b.r^{-1} mod\ n, \qquad a,b < n, GCD(n,r) = 1$$

The value of r can equal 2^k where k is the number of bits in n or the key size, thus, the condition $GCD(n,r) = 1$ will always be true because n is always odd, since it is the product of two big primes(10).

So, the implementation should use Montgomery's Algorithm to enhance the performance. However, to make the calculation of the big integers i.e. multiplying two integers there are two implementation options:

- Using one of the existing libraries that perform this kind of calculation.
- Implementing the data structure of the big integer and any required operations from scratch.

The decision will be taken after surveying the capabilities of these libraries and their relevance to solving the problem.

2.3.2.2.2.1. Montgomery Multiplication

Let N be an integer number represented by k binary digits i.e. $2^k > N \geq 2^{k-1}$, a, b are two integers where $a, b < N$, and radix $R \in Z^+$ where $R > N$ and R is relatively-prime to N, i.e. $GCD(R, N) = 1$.

The integer $i, 0 \leq i < N$, where $iR^{-1} mod\ N$ represents a complete residue system, which means it contains the whole numbers $0..N-1$. The Montgomery reduction exploits this by providing an efficient way to calculate the value $TR^{-1} mod\ N$, where $0 \leq T < N$ using the following algorithm (11).

Function $REDC(T)$

$m \leftarrow (T\ mod\ R)N'\ mod\ R$

$t \leftarrow (T + m.N)/R$

if $(t \geq N)$ **return** $t - N$ **else return** t

List 3 The Montgomery Reduction Algorithm (11).

According to (12) and (13), integers N' and R^{-1} can be calculated using the Extended Euclidean Algorithm from the equation $R.R^{-1} - N.N' = 1$, or can be calculated using the formula $N' = -N^{-1} mod\ R$

So, the residue of the product of the numbers $0 \leq a,\ b < N$ can be calculated easily using the Montgomery reduction.

$$c = REDC(a.b) \equiv a.b.R^{-1}\ mod\ N$$

$$c.R^{-1} \equiv (a.R^{-1}).(b.R^{-1}) \; mod \; N$$

Thus, c is the product of a and b in the residue representation.

It is possible to convert to the N-residue system, so let $a,\ b$ are two integers where $0 \leq a, b < N$. Then, the N-residue of a and b are respectively:

$$\bar{a} = a.R \; mod \; N$$

$$\bar{b} = b.R \; mod \; N$$

Yields:

$$REDC(\bar{a}.\bar{b}) = \bar{a}.\bar{b}.R^{-1} mod \; N = a.R.b.R.R^{-1} mod \; N = a.b.R \; mod \; N$$

Let $c = a.b$, then

$$\bar{c} = a.b.R \; mod \; N$$

Thus,

$$\bar{c} = REDC(\bar{a}.\bar{b})$$

This leads to the Montgomery multiplication algorithm:

```
Function monPro(ā, b̄)
    T ← ā.b̄
    m ← (T mod R)N' mod R
    t ← (T + m.N)/R
    if (t ≥ N) return t − N else return t
```

List 4 The Montgomery Multiplication, (14).

So, if it is required to calculate the modulus of the product of two numbers a and b, the required steps are:

1. Calculate N' and R^{-1} using an extended Euclidian algorithm.
2. Calculate the N-residue $\bar{a} = monPro(a, R^2 mod \; N)$.
3. The result is $c = monPro(\bar{a}, b) = a.R.b.R^{-1} mod \; N = a.b \; mod \; N$.

For performing one modulus multiplication operation, it is not a good idea to apply this method, as it will be more time consuming compared with the simple division, due to the required pre-computations like calculating N' and $R^2 mod \; N$. However, if the multiplication operations are repeated with the same modulus N and radix R,

Montgomery multiplication will be more efficient, and this is the case in RSA encryption and decryption system.

If the radix R is 2^k, then it will be relatively-prime to any odd number N. However in RSA N is always odd as it is the result of multiplication of two big odd prime numbers p and q. Furthermore, the previous mod and div operations can be performed in the computer system by simple bitwise operations.

2.3.2.2.2.1.1. Montgomery Multiplication Algorithms

There are different versions of Montgomery multiplication algorithm. In addition, in (14) there is a comparison between five various implementations of Montgomery multiplication, which are: Separated Operand Scanning (SOS), Coarsely Integrated Operand Scanning (CIOS), Finely Integrated Operand Scanning (FIOS), Finely Integrated Product Scanning (FIPS), and Coarsely Integrated Hybrid Scanning (CIHS). All these versions require the same number of multiplication operations $(2.s^2 + s)$, where s is the number's word count. However, they vary in the additions, memory write and read operations. According to (14), these versions can be arranged in descending order in terms of efficiency as follows: CIOS, SOS, FIOS, FIPS finally CIHS. These versions are classified depending on two factors: the first is whether or not the multiplication is separated from the reduction so there are two classifications "*Separated*" and "*Integrated*"; furthermore the integration can be "*Coarse-grained*" or "*Fine-grained*" depending on how often the alternation between the multiplication and the reduction is happening. The second factor is based on the way that the multiplication and reduction go, which can be "*Operand Scanning*" if the outer loop moves through the words of one of the operands, or "*Product Scanning*" when the outer loop moves throw the words of the multiplication result.

There is one major improvement used in all previous versions. This improvement is using the number n_0' instead of N'. This can be done because in all previous versions the numbers are processed word by word, so it is possible to use the first word n_0 from the number N and calculate the value $n_0' = -n_0^{-1} mod\ n$, where $n = 2^w$ where w is the word length(13). This value can be calculated efficiently using an extended Euclidian algorithm or in a more efficient way using the following algorithm:

```
Function modInv(x, n)
    r ← 1
    for i ← 2 to lg(n)
        if (x.r mod 2^i > 2^(i−1))
            r ← r + 2^(i−1)
    return r
```

List 5 Algorithm for calculating $x^{-1} mod\ n$ when x is odd and n is power to 2, (13).

This algorithm works only when n_0 is odd number and modulus is a power of 2. In this context n_0 is always odd as it is the least significant word of the odd number N, and $n = 2^w$ where w is the word length.

According to (14), CIOS version achieves the best performance compared with other versions. The multiplication step of the two input numbers a and b is integrated using the same loop along with the reduction step. This can be done because the value of $m[i]$ depends only on the value of $T[i]$ in the i_{th} iteration, furthermore, the value $T[i]$ will be calculated completely by the i_{th} iteration of the outer loop. This leads to the following algorithm:

Function $monProCIOS(a, b, n)$
  for $i \leftarrow 0$ **to** $s - 1$
    $C \leftarrow 0$
    for $j \leftarrow 0$ **to** $s - 1$
      $(C, S) \leftarrow t_j + a_j . b_i + C$
      $t_j \leftarrow S$
    $(C, S) \leftarrow t_s + C$
    $t_s \leftarrow S$
    $t_{s+1} \leftarrow C$
    $C \leftarrow 0$
    $m \leftarrow t_0 . n_0' \ \mathbf{mod}\ 2^w$
    $(C, S) \leftarrow t_0 + m . n_0$
    for $j \leftarrow 1$ **to** $s - 1$
      $(C, S) \leftarrow t_j + m . n_j + C$
      $t_{j-1} \leftarrow S$
    $(C, S) \leftarrow t_s + C$
    $t_{s-1} \leftarrow S$
    $t_s \leftarrow t_{s+1} + C$
  if $(\frac{t}{R} \neq 0)$
    $B \leftarrow 0$
    for $i \leftarrow 0$ **to** $s - 1$
      $(B, D) \leftarrow t_i - n_i - B$
      $t_i \leftarrow D$
  return t

List 6 The CIOS Montgomery multiplication algorithm, (14), (13), (15).

The previous algorithm employs two improvements: the first one is using n_0' instead of N', and the second improvement is getting rid of the final comparison by checking the condition $\frac{t}{R} \neq 0$ which can be done easily by looking at the last word of the result t_s, so the condition will be $t_s \neq 0$ (15). These improvements can be applied to all other versions of Montgomery multiplication.

Finely Integrated Operand Scanning (FIOS) is a variation of CIOS so that the calculation of the multiplication and the reduction is integrated in one internal loop. After applying the same previous improvements the resulting algorithm is:

```
Function monProFIOS(a, b, n)
    for i ← 0 to s − 1
        (C, S) ← t_0 + a_0 . b_i
        ADD(t_1, C)
        m ← S . n_0' mod 2^w
        (C, S) ← S + m . n_0
        for j ← 1 to s − 1
            (C, S) ← t_j + a_j . b_i + C
            ADD(t_{j+1}, C)
            (C, S) ← S + m . n_j
            t_{j−1} ← S
        (C, S) ← t_s + C
        t_{s−1} ← S
        t_s ← t_{s+1} + C
        t_{s+1} ← 0
    if (t/R ≠ 0)
        B ← 0
        for i ← 0 to s − 1
            (B, D) ← t_i − n_i − B
            t_i ← D
    return t
```

List 7 FIOS Montgomery Multiplication Algorithm, (14), (13), (15).

The function ADD adds the carry to the number t starting from specified word and propagates the new resulted carry. Although FIOS seems simpler than CIOS with just one internal loop, the cost of calling ADD function make the overall performance worse (14).

2.3.3. Summary

The previous section provides an overview of all the security issues that must be considered for the project. Firstly, there was a description of general security aspects with illustration of the main classification of cryptographic algorithms. Then there was illustration of the cryptographic operation modes, which play a major role in the security side of the produced cipher code, followed by a detailed description of AES, the new encryption standard and which is the pioneer among the symmetric cryptographic algorithms. Finally, a detailed description of RSA, the pioneer algorithm among asymmetric cryptographic algorithms, was introduced.

2.4. Parallel Programming

2.4.1. Definition

Parallel programming is a form of computing which depends on concurrent execution of the program components; this enhances the performance of the program and reduces the execution time. The role of concurrency in increasing the performance has been realised as a means to increase computing performance, especially recently. At the beginning of processors development, Moore's Law was adopted as a way to increase the performance by increasing the number of transistors in the integrated chip, then another technique was adopted which tried to implement implicit parallelism by executing more than one instruction in one cycle, pipelining and increasing the frequency of the processors. However, these techniques reached their limit as well when problems related to power consumption and heat began to appear. Nowadays, multi-core techniques are adopted to achieve the goal with an increased number of cores inside one processor. However, to get the most benefit of these processors, the application that is being executed should be designed in a concurrent way; this is the need of explicit parallel techniques and high-level parallel programming languages to make the mission easier to the programmer in order to develop multi-threaded or multi-process parallel applications(16),(17).

2.4.2. Parallel Programming Languages

"Parallel Programming is programming in a language that allows you to explicitly indicate how different portions of the computation may be executed concurrently by different processors." (18). A lot of techniques (libraries, API, parallel models and languages) have been evolved to assist the development of parallel programs.

2.4.2.1. MPI

MPI stands for Message Passing Interface, which is a standardised and portable message passing library with wide acceptance among academia and industry. The first MPI version was released in 1994 by the MPI Forum group which involves more than 80 people from 40 organizations mainly from US and Europe (19).

The message passing technique is usually designed for the distributed-memory parallel computer context; however, the same code works well on shared-memory parallel computer context. Thus it can run in a heterogeneous network of computers or as multiple processes inside one computer. Consequently, the user should not worry about whether the messages are sent between unalike process architecture. Another feature of MPI is hiding the details of how the MPI object is represented so MPI objects are free to get the best implementation under specific circumstances. Furthermore, well-designed MPI eliminates complex messages header encoding or decoding and reduces the need for extra copying and buffering of the message (19 pp. 1-5). It is suitable for multiprocessors, multicomputer platforms with explicit memory hierarchy control (18).

Furthermore, According to (20 p. 277) MPI-2 was released in 1997 with a lot of new technology such as: dynamic process management, parallel I/O, complete thread interface and interrupt-driven receives etc.

MPI still does not include explicit shared-memory operations, real-time computing support or wide area networks support (20 p. 278). In addition, MPI does not support incremental parallelization, and the resulting code is fairly big (18), and considered Mid-Level parallel technique.

2.4.2.2. OpenMP

OpenMP stands for Open Multi-Processing, which is a portable shared memory application programming interface (API) to make shared memory parallel programming easier. OpenMP provides an easy to learn and apply approach to incrementally parallelise portions of an existing code, by adding compiler directives, runtime library routines, or environment variables to the code in order to explain to the compiler how to execute the code using a collection of cooperated threads. Eventually, the parallelised code and the original sequential one exist in the same set of files, thus the maintenance of the code is much easier than other parallel techniques.

The main idea of OpenMP is developing multiple threads to solve the problem; all the threads will share the same resources that correspond to the main process. So, OpenMP provides an opportunity to write such multithreaded programs easily by adopting the fork-join programming model. In this model the execution starts as a single thread, however, OpenMP creates a team of threads when a parallel construct is encountered. At the end of the construct only the main thread continues while the others are terminated - this is the join. All the parallel constructs in the program are called parallel regions.(21 pp. 23,24)

According to (21 p. 31) OpenMP encourages heavily parallelising the loops using multiple-threads in a structured way, however, if loop-level parallelism is limited in the program, the programmer can use another approach, which is assigning work to each thread explicitly. This approach will produce a high quality code; however, the programmer should manually inset synchronizations, which may lead to programming errors like deadlock error when all the threads are waiting for each other forever. OpenMP provides sufficient tools to support this programming approach which is called a low-level programming approach. However, debugging in OpenMP may be very difficult, especially data race condition that may be very difficult to be detected.

2.4.2.3. Parallel Haskell

Haskell is a general purpose functional programming language named after the logician Haskell Curry. As a functional language the function is the primary element in the source code and adopts the functional style. The Haskell code is concise with two to ten times

shorter than other programming languages equivalent code because Haskell uses lists as basic concept in the language and uses recursive function notion as a basic mechanism from looping. Furthermore, it has powerful type system which requires little type information from the programmer but supports a large class of incompatibility errors. Haskell has high-order and pure functions, which means that the functions can take functions as parameters and all parameters are input and the all the returned values are output. In addition, Haskell adopts lazy computation, which means that the expression will not be evaluated until its result is actually required(22 pp. 1-5).

According to (23 p. 438) Haskell does not provide parallelism directly, because it is general purpose and high level programming language. However, according to (24) there are two well-developed parallel extensions to Haskell which are GpH and Concurrent Haskell.

2.4.2.3.1. GpH

According to (24) GpH is an extension to Haskel'98, in which the program still has one main I/O thread but new pure threads can be defined to evaluate parallel parts. The creation of the threads depends on the machine state; however, the threads cannot be manipulated by the programmer.

2.4.2.3.2. Concurrent Haskell:

This is Haskell'98 extension as well, in which the program has one main I/O thread, however, more I/O threads can be created and controlled by programmer because each one is identified by id. The creation of new I/O threads is mandatory regardless of the state of machine (24).

2.4.3. Designing Parallel Encryption Algorithms

Modern encryption algorithms usually rely heavily on maths to produce the cipher text. Consequently, those algorithms can contain heavy mathematical calculation repetition on the text data segments (bytes, blocks etc). So the SPMD mode can be the most suitable mode to parallelize these algorithms. However, according to (25) the general stages of designing a parallel program from sequential code are: analyse the sequential algorithm to identify the potential concurrency, design and implement the parallel version (in this stage the prototyping techniques may used), test to discover the concurrency errors that may be caused by multi-threads and tune the algorithm for better performance by removing all performance bottlenecks.

In the security section of this report, two of the most frequently used cryptographic algorithms have been described and identified. The previous steps will be adopted when the parallel algorithms are designed; however, here it is important to illustrate some factors which will be used in the project.

2.4.3.1. Prototyping

"Prototype is an initial version of a software system that is used to demonstrate concepts, try out design options and, generally, to find out more about the problem and its possible solutions" (26). According to (27), prototyping is a common activity during the design stage of systems. It concentrates on the main ideas and core requirements, and results in decreasing in the overall coding which has been dedicated to solve the problem. Furthermore, prototyping allows exploration of various solutions and critical resource aspects associated with the problem. The prototypes should be built rapidly, and modified rapidly as well in order to get the most benefits from the prototyping process. To achieve this, the language should provide very high-level semantic constructs which enable the designer to model any data structure or idea.

In the parallel context prototyping can be used in the parallel algorithm design stage to produce the initial parallel version of the software which contains the main parallel features. Later, the designer can learn from this version and improve it until he/she reaches the final version. So, the main goal of prototyping is not achieving the perfect speedup but making the designer experience the ideas before mapping the algorithm to a specific parallel architecture in order to achieve the perfect speedup as a final result. The prototyping language should provide a powerful means to dynamically create and coordinate the parallel processes (28).

Functional programming languages like Haskell are very suitable for prototyping because of the high level of abstraction that they provide (27). With the concise code that these languages provide the prototype version can be developed quickly. Furthermore, Haskell can make the coding of expensive computations algorithms easier due to its lazy and strict high-level semantic features.

2.4.3.2. Parallel Implementation

According to (25), there are eight rules which should be adopted in designing multithreaded parallel algorithms which are:

1. Identify independent components of the sequential algorithm, as some of the algorithm components may not be parallelised due to the dependencies between them.
2. Implement the concurrency using the highest level of parallelism, there are two approaches to define the highest level top-down and bottom-up approaches.
3. Plan for scalability in the early design stage and take into consideration the increasing number of processing units.
4. Use thread-safe libraries as much as possible, however, when it is not possible synchronization should be added to protect the shared resources.
5. Use the implicit threading model over the explicit one, if the implicit model provides the required functionality.

6. Do not assume that the algorithm components should be executed in a predefined specific order.
7. Use local thread variables as much as possible, and provide locks on the shared data to guarantee synchronization.
8. Change the algorithm if this can provide more concurrency, even if this leads to increasing the complexity of the algorithm.

A first glance at the cryptographic algorithms, which have been illustrated before in this report, can show that the main potential parallel regions in the algorithms can be in the AES algorithm case dividing the state itself between some threads which can perform the transformation. AES does not need a great deal of computational power; however, by using parallelism the number of rounds can be increased and performance will not be affected. In the case of RSA, the situation is different as it needs a lot of computation power, as it deals with a huge numbers and performs very expensive multiplication and modular operations by using the Montgomery algorithm(10). Thus, the potential parallelism can be done by dividing the overall modular computation depending on the mathematical properties of modulo operation. However, both AES and RSA can be parallelised to a higher level, if the used encryption mode was ECB or CTR modes, which may provide higher performance. If the CBC mode is used, only the first criteria can be used.

2.4.3.3. Performance Benchmarking

Performance benchmarking is a very important step which allows the programmer to identify the bottlenecks in his/her algorithm and try to get rid of them in order to enhance the overall performance. Moreover, the programmer can decide about the best suitable level of parallelism or the granularity level for the algorithm and the parallel environment. And know more about the scalability factor in his/her algorithm, when the programmer increases the number of processing units that are co-operating to solve the problem.

2.4.3.3.1. Speedup

According to (18), the speedup of the parallel algorithm is the ratio between the delay time of sequential execution and the delay time of the parallel version execution:

$$S = \frac{T_{seq}}{T_{par}}$$

Where the value T_{seq} is the execution delay time of the sequential version, and T_{par} is the execution delay of the parallel version. However, the value of the speedup can be represented as a multiple as well, e.g. 2x, which means that the speed will be doubled (25). The speedup should be changed by changing the number of processing units; this

will give a feeling of the scalability of the algorithm by measuring how much the speedup is changing when the number of processing units is changed.

Sometimes it is very important that the designer knows the expected speedup before he/she writes any concurrent code. Amdahl's Law can be used to give the maximum speedup value that the designer can achieve after implementing the parallel algorithm. The designer of course should analyse the sequential algorithm to know the amount of potential parallel portions and the sequential portions in the algorithm, consequently, the percentage of execution time of the parallel portions in the algorithm, and then apply Amdahl's Law (25):

$$Speedup \leq \frac{1}{\frac{Pct_{par}}{P} + \left(1 - Pct_{par}\right)}$$

The value Pct_{par} is the percentage of the parallel parts execution time, and P is the number of processing units which are employed to execute the code. The perfect case is when the percentage of the parallel parts execution time $Pct_{par} = 1$. Thus Amdahl's formula will be $Speedup \leq P$, thus the speedup represented in the multiple representation will be equal to the number of processing units.

According to (25), one criticism of Amdahl's Law is that it assumes that the amount of data that is processed is fixed for any number of processing units. Gustafson-Barsis's Law addresses this problem and takes into consideration the increased proportion of processed data as the number of processing units is increasing. The formula of Gustafson-Barsis's Law is:

$$Speedup \leq P + (1 - P).Pct_{seq}$$

The value Pct_{seq} is the percentage of the execution time of the sequential part of a parallel code.

2.4.3.3.2. Efficiency

Efficiency tells the designer how well the computational resources are being utilized (25). Efficiency can be measured by dividing the Speedup value by the number of processing units.

$$Efficiency = \frac{Speedup}{P}$$

The number of processing units is a major factor in the efficiency equation; moreover, in some programs the efficiency may be negatively affected when the number of processing units is increased. Although reducing the number of processing units may enhance efficiency, the major question remains *"why can the parallel program not get the most*

benefit from the processing units?" In addition, increasing the number of threads and assigning more than one thread to each processing unit may slightly enhance the performance of the program (25). However, the values of Speedup and Efficiency can provide clues to the designer about the granularity level for best performance.

2.4.4. Summary

The literature review covers the main aspects of the project, which can be divided into two essential parts: security and parallel parts. The main focus was on the description of the cryptographic algorithms AES and RSA as they rely on maths and need mathematical understanding in order to be able to program them in the best way. This is especially true when the target mission is to parallelise these algorithms, which need a very deep understanding not only for their steps and stages, but also how they work mathematically. The Parallel part focuses on the technologies which are available to build a parallel program. Furthermore, it focuses on the performance measures that can be applied to improve the parallel algorithm.

Chapter 3. Sequential Implementation and Performance

In this chapter, the sequential versions of the cryptographic algorithm RSA will be implemented by adopting the work in the most recent papers and the best techniques that are known nowadays. All these versions will be implemented and integrated in the same program and then the performance benchmark will be performed in order to help choose the version with the best performance.

As a result of this chapter, there should be one chosen sequential version which represents the best tuned version of RSA algorithm, so it will be possible to work on this version later and use it as a reference for the RSA parallel version.

3.1. Algorithm(s) to be parallelized

From the previous literature survey it can be seen that *RSA* and *AES* are the strongest algorithms and the most frequently used ones nowadays. Consequently, those two algorithms are very strong candidates to be implemented and parallelized. *RSA* is very strong and very slow, especially when long keys are used. *RSA* complexity is $O(n^3)$ where $(n = key\ size)$. *AES* is a strong and very fast symmetric algorithm with complexity $O(1)$ (29), because AES takes only three key lengths 128, 192 and 256 bit, i.e. a finite set of numbers.

Consequently, due to time limitations the *"symmetric algorithm AES"*, is not implemented or parallelized. Moreover, it would be more interesting to parallelize *RSA* algorithm, thus only *RSA* algorithm will be parallelized and as a result of this project, a very finely-tuned parallel version of *RSA* algorithm should be provided.

3.2. RSA Overview

RSA is one of public key cryptographic algorithms; its encryption and decryption involve computation of the modular exponentiations by accomplishing the computation $c = m^d mod\ n$ in the encryption and $m = c^e mod\ n$ in the decryption, where m is the message, d is the private key, e is the public key, and n is the modulus.

In RSA applications, the numbers m, d, e, c and n are very big integers and represented by 1024 bits and more. In this case, the modulus operation which contains a division operation is very time consuming. However, there are many different ways to implement the *"modulo operation"* or even the *"modular exponentiation"* in an efficient way by combining Montgomery modular multiplication with binary method (12), common multiplicand multiplication method (30), or signed-digit recoding method (31).

In this project, Montgomery modular multiplication with binary exponentiation, and Montgomery modular multiplication with signed-digit recording exponentiation methods will be used to implement RSA.

3.3. Designing the Data Structure

It is essential to design a data structure holding arbitrary-precision integers in order to implement RSA encryption and decryption functions. One possibility is to use the existing libraries to deal with such numbers like GMP with C programming language. However, the library's structure may restrict the programmer and make the implementations very complicated. So, designing a different structure to represent arbitrary-precision integers can be a better solution.

The new structure should provide a very fast and simple way to move back and forth over the parts of the number. Furthermore, it should make use of C language capabilities by employing the biggest possible positive integer type to represent each part of the number; however, there should still be a bigger integer type (typically double length) to implement the multiplication and addition of the parts. This can be satisfied by using these types:

```
typedef unsigned int uInt;
typedef unsigned long long bigInt;
```

And by using simple dynamic C arrays the first condition can be satisfied as well, thus the resulting data structure that can be used easily and efficiently will be an array of uInt type:

```
typedef uInt *hugeInt;
```

Each number now can be separated into parts with 32 bit length (the length of uInt), and because all the methods that will be used to process the basic operations on the numbers deal with the number from right to left or from the least significant part to the most significant part, the least significant part will go at the very beginning at index 0; which will make implementing the basic arithmetic operation easier. Figure 6 shows the suggested structure to represent a $(32 * s$ bit) Integer.

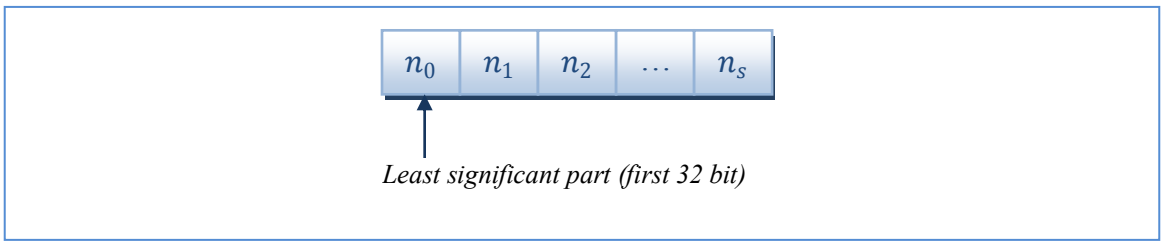

Figure 6 Data structure to represent the arbitrary-precision integers.

3.4. Designing the Experiment

Based on the previous data structure there are two main units: the HuInt unit and the Montgomery unit. The HuInt unit contains all the basic arithmetic operations on arbitrary-precision integers plus the functionality to generate prime numbers for RSA keys. The Montgomery unit contains implementation of various versions of Montgomery multiplication and exponentiation.

3.4.1. HuInt Unit

This unit provides the functionality to perform mathematical operations on arbitrary-precision integers. Furthermore, it provides some functions to generate prime numbers which can be used to generate RSA keys.

HuInt provides the function *"newNum"* which allocates memory for new arbitrary-precision integer, and sets the new allocated space to zero.

```
int newNum(hugeInt *t, size_t size) {
    if (size == 0) {
        t = NULL;
        return 0;
    }
    *t = (hugeInt) calloc(size, sizeof (uInt));
    if (*t == NULL) {
        puts("Cannot allocate memory!!");
        exit(-1);
    }
    return 1;
}
```

List 8 newNum function to create new number

This unit provides the basic arithmetic functions as well which are: *"ADD", "SUB", "MUL", "DIV", "MOD",* to perform addition, subtraction, multiplication, division and normal modulus for two arbitrary-precision integers. The modulus function simply calls the division function and returns the division reminder. So, it is very slow and cannot be used in the RSA implementation, however, this function can be used for validation purposes. There is also another set of functions *"Add", "Sub"* and *"Mul"* which are used to add, subtract and multiply scalar to an arbitrary-precision integer, whereas the function *"add"* performs the addition between arbitrary-precision integer and scalar from a specified index and propagate the carry.

The *"ASSIGN"* function copies one arbitrary-precision integer to another quickly by calling the function *"memcpy"* from some C library. There is another function called *"Assign"* which assigns a scalar to the least significant part of the arbitrary-precision integer and puts zero in all other parts. *"COMP"* function compares two arbitrary-precision integers and returns a 1, 0, or -1 if the first number was bigger, equal to or smaller than the second number.
The *"MOD_INV"* function calculates the reverse modulus for a given arbitrary-precision integer x and a given modulus n, so the result will be $x^{-1} mod\ n$. This function calls the *"EGCD"* function which applies Extended Euclidian Algorithm to calculate the reverse modulus. However, there is another function called *"EBGCD",* which performs the Binary Extended Euclidian Algorithm, this can perform faster on computer systems as it needs only right shift, left shift, addition and subtraction operations. However, the algorithm *"EBGCD"* needs right and left shift so these three functions *"SHIFT_LEFT",*

"SHIFT_RIGHT" and *"SHIFT_RIGHT_SIGNED"* are implemented. The function *"minModInv"* returns the negative reverse modulus in a very efficient way for an integer when the modulus is 2^w i.e. $r = -x^{-1} mod\ 2^w$.

The functions *"toUDR"* and *"toSDR"* are used to convert the number to a different format; *"toUDR"* converts the number to an array of bytes which contain the binary format of the number i.e. 0s and 1s, whereas *"toSDR"* converts the number to the signed digit record so the resulted array contains 0s, 1s, and -1s. List 9 shows the code of the signed digit record method.

```
void toSDR(binary res, hugeInt e, uInt rLen) {
    unsigned int i;
    byte c0 = 0, c1;
    byte r [rLen * W + 2];
    toUDR(r, e, rLen);

    r[rLen * W] = 0;
    r[rLen * W + 1] = 0;
    for (i = 0; i <= rLen * W; i++) {
        c1 = (byte) ((c0 + r[i] + r[i + 1]) / 2);
        res[i] = c0 + r[i] - 2 * c1;
        c0 = c1;
    }
}
```

List 9 Signed Digit Recording Method

```
void toUDR(binary res, hugeInt e, uInt rLen){
    uInt et;
    unsigned int i, j;
    for (i = 0; i < rLen; i++) {
        et = e[i];
        for (j = 0; j < W; j++) {
            res[i * W + j] = et & 1;
            et = et >> 1;
        }
    }
}
```

List 10 Binary Digit Recording Method

The HuInt unit contains some functions to generate random and prime numbers, it contains a *"getRandom"* function which generates a random number by providing the length of the number. The function *"createPrime"* generates a prime number by generating random numbers and testing them by calling the function *"rabinMiller"* which returns true when the input number is prime number. List 11 shows the Rabin Miller primality test method.

```
int rabinMiller(hugeInt n, uInt checkNo, uInt rLen) {
    uInt s = 0;  int result = 0;  unsigned int i, j;
    hugeInt one = NULL, d = NULL, n_1 = NULL, b = NULL, x = NULL;
    newNum(&one, rLen);  newNum(&d, rLen);  newNum(&n_1, rLen);
    newNum(&b, rLen);    newNum(&x, rLen);
    assign(one, 1, rLen);  SUB(n_1, n, one, rLen);  ASSIGN(d, n_1, rLen);
    while (!IS_ODD(d, rLen)) {
        SHIFT_RIGHT(d, d, 1, rLen);
        s++;
    }

    for (i = 0; i < checkNo; i++) {
        do {
            getRandom(b, rLen);
            if (b[rLen - 1] > n_1[rLen - 1]) b[rLen - 1] -= n_1[rLen - 1];
        } while ((COMP(b, one, rLen) == 0) || (COMP(b, n_1, rLen) >= 0));

        POW_MOD(x, b, d, n, rLen);
        if (COMP(x, one, rLen) == 0 || COMP(x, n_1, rLen) == 0) continue;

        for (j = 0; j < s - 1; j++) {
            MOD_MUL(x, x, x, n, rLen);
            if (COMP(x, one, rLen) == 0) goto FREE_VARS_RET;
            if (COMP(x, n_1, rLen) == 0) break;
        }
        if (j < s - 1) continue;
        goto FREE_VARS_RET;
    }
    result = 1;
FREE_VARS_RET:
    free(one);  free(n_1);  free(d);  free(b);  free(x);
    return result;
}
```

List 11 Rabin Miller primality test method

Finally, the functions *"toBytes"* and *"toInvBytes"* are used to convert the number to an array of bytes so it will be ready to be written in a file. On the other hand, the functions *"fromBytes"* and *"fromInvBytes"* are used to convert a byte array to an arbitrary-precision integer. The difference between *"toBytes"* and *"toInvBytes"* is that the *"toBytes"* function converts the arbitrary-precision integer to a byte array so that the first index in the resulted array will contain the least significant byte. The function *"toInvBytes"* puts the most significant byte in the first index in the resulted byte array. The same difference exists between *"fromBytes"* and *"fromInvBytes"*. The functions *"fprintHugeInt"* and *"fscanHugeInt"* are used to write and read the number from file.

3.4.2. Montgomery Unit

This unit contains two groups of functions, all of them based on Montgomery reduction; the first group is the Montgomery multiplication functions and the second group is Montgomery exponentiation functions that are based on the first group of functions. There are multiple versions which vary in their performance. The strategy is to choose the best performance. For Montgomery multiplication SOS, CIOS, FIOS versions are

implemented, and with regard to the exponentiation algorithms the Binary and SDR versions are implemented. Each version of the Montgomery exponentiation can use any version of multiplication, so it is essential to use the best combination.

3.4.2.1. Montgomery Multiplication

Mainly from (14), and with the improvements from (15) and (13) SOS, CIOS, FIOS versions are implemented. Regarding to the FIOS version some other improvements are made to the algorithm in order to achieve better performance.

3.4.2.1.1. SOS Montgomery Multiplication

The Separated Operand Scanning (SOS) version separates between the multiplication and reduction steps. This version is implemented directly from published papers. List 12 shows the C implementation of this algorithm.

```
void monProSOS(hugeInt r, hugeInt a, hugeInt b, hugeInt n, uInt n0Prime,
uInt rLen) {
    uInt tLen = 2 * rLen + 1;
    hugeInt t;  newNum(&t, tLen);
    bigInt cs = 0; uInt c, m;  unsigned int i, j;
    for (i = 0; i < rLen; i++) {
        c = 0;
        for (j = 0; j < rLen; j++) {
            cs = (bigInt) t[i + j] + (bigInt) a[j] * (bigInt) b[i] + (bigInt) c;
            t[i + j] = (uInt) cs;
            c = cs >> W;
        }
        t[i + rLen] = c;
    }
    for (i = 0; i < rLen; i++) {
        c = 0;
        m = t[i] * n0Prime;
        for (j = 0; j < rLen; j++) {
            cs = (bigInt) t[i + j] + (bigInt) m * (bigInt) n[j] + (bigInt) c;
            t[i + j] = (uInt) cs;
            c = cs >> W;
        }
        add(t, c, rLen + i, tLen);
    }
    hugeInt tm = t + rLen; //tm = t / R
    if (tm[rLen] != 0) {   //t/R != 0
        c = 0;
        for (i = 0; i < rLen; i++) {
            cs = (bigInt) tm[i] - (bigInt) n[i] - (bigInt) c;
            r[i] = (uInt) cs;
            c = (cs >> W) & 1;
        }
    } else {
        ASSIGN(r, tm, rLen);
    }
    free(t);
}
```

List 12 SOS Montgomery Multiplication method

3.4.2.1.2. CIOS Montgomery Multiplication

The Coarsely Integrated Operand Scanning (CIOS) version integrates the multiplication with the reduction so it has only one outer loop. However, it has two internal loops - one for partial multiplication and the other one for partial reduction. This version is implemented directly from published papers. List 13 shows the C implementation of this algorithm.

```
void monProCIOS(hugeInt r, hugeInt a, hugeInt b, hugeInt n, uInt n0Prime,
uInt rLen) {
    uInt tLen = 2 * rLen + 1;
    hugeInt t; newNum(&t, tLen);
    bigInt cs = 0;  uInt c, m;  unsigned int i, j;
    for (i = 0; i < rLen; i++) {
        c = 0;
        for (j = 0; j < rLen; j++) {
            cs = (bigInt) t[j] + (bigInt) a[j] * (bigInt) b[i] + (bigInt) c;
            t[j] = (uInt) cs;
            c = cs >> W;
        }
        cs = (bigInt) t[rLen] + (bigInt) c;
        t[rLen] = (uInt) cs;
        c = cs >> W;
        t[rLen + 1] = c;
        c = 0;
        m = t[0] * n0Prime;
        cs = (bigInt) t[0] + (bigInt) m * (bigInt) n[0];
        c = cs >> W;
        for (j = 1; j < rLen; j++) {
            cs = (bigInt) t[j] + (bigInt) m * (bigInt) n[j] + (bigInt) c;
            t[j - 1] = (uInt) cs;
            c = cs >> W;
        }
        cs = (bigInt) t[rLen] + (bigInt) c;
        t[rLen - 1] = (uInt) cs;
        c = cs >> W;
        t[rLen] = t[rLen + 1] + c;
    }
    if (t[rLen] != 0) {
        c = 0;
        for (i = 0; i < rLen; i++) {
            cs = (bigInt) t[i] - (bigInt) n[i] - (bigInt) c;
            r[i] = (uInt) cs;
            c = (cs >> W) & 1;
        }
    } else {
        ASSIGN(r, t, rLen);
    }
    free(t);
}
```

List 13 CIOS Montgomery Multiplication method

3.4.2.1.3. FIOS Montgomery Multiplication (Paper)

The Finely Integrated Operand Scanning (FIOS) version is very similar to the CIOS version; however, the main difference is that the two internal loops in CIOS are

integrated in one loop to calculate the partial multiplication and the partial reduction at the same time. This version is implemented directly from published papers. List 14 shows its implementation.

```
void monProFIOS(hugeInt r, hugeInt a, hugeInt b, hugeInt n, uInt n0Prime,
uInt rLen) {
    hugeInt t = NULL;  newNum(&t, rLen + 2);
    int i, j;  bigInt cs;  uInt m, c;
    for (i = 0; i < rLen; i++) {
        cs = (bigInt)t[0] + (bigInt)a[0] * (bigInt)b[i];
        c = cs >> W;
        add(t, c, 1, rLen + 2);
        m = (uInt)cs * n0Prime;
        cs = (uInt)cs + (bigInt)m * (bigInt)n[0];
        c = cs >> W;
        for (j = 1; j < rLen; j++) {
            cs = (bigInt) a[j] * (bigInt) b[i] + (bigInt) t[j] + (bigInt) c;
            c = cs >> W;
            add(t, c, j + 1, rLen + 2);
            cs = (bigInt) m * (bigInt) n[j] + (uInt) cs;
            c = cs >> W;
            t[j - 1] = (uInt) cs;
        }
        cs = (bigInt)t[rLen] + (bigInt)c;
        c = cs >> W;
        t[rLen - 1] = (uInt) cs;
        t[rLen] = t[rLen + 1] + c;
        t[rLen + 1] = 0;
    }
    if (t[rLen] == 0) {
        ASSIGN(r, t, rLen);
    } else {
        c = 0;
        for (i = 0; i < rLen; i++) {
            cs = (bigInt) t[i] - (bigInt) n[i] - (bigInt) c;
            r[i] = (uInt) cs;
            c = (cs >> W) & 1;
        }
    }
    free(t);
}
```

List 14 FIOS Montgomery Multiplication method

3.4.2.1.4. FIOS Montgomery Multiplication (Modified)

According to (14), The FIOS algorithm performs slower than CIOS because of the internal call of the function *"add"*, which adds and propagates the resulting carry to the partial result. However, the carry addition can be postponed by saving the carry in a two-word long variable and in the next iteration the least significant part is added to the partial result whereas the most significant part becomes the least significant for the carry to the next iteration. Thus the new modified FIOS algorithm can be:

```
Function monProFIOS(a, b, n)
    for i ← 0 to s − 1
        (C1, C0, S) ← t0 + a0. bi
        m ← S. n0 mod 2^w
        (C, S) ← S + m. n0
        (C1, C0) ← (C1, C0) + C
        for j ← 1 to s − 1
            (C, S) ← tj + aj . bi + C0
            (C1, C0) ← C1 + C
            (C, S) ← S + m. nj
            (C1, C0) ← (C1, C0) + C
            tj−1 ← S
        (C1, C0) ← ts + (C1, C0)
        ts−1 ← C0
        ts ← C1
    if (t/R ≠ 0)
        B ← 0
        for i ← 0 to s − 1
            (B, D) ← ti − ni − B
            ti ← D
    return t
```

List 15 Enhanced FIOS Montgomery Multiplication

The step $(C_1, C_0) \leftarrow t_s + (C_1, C_0)$ does not produce overflow because the number (C_1, C_0) can easily hold four half size additions, i.e. can hold the result of $C_1 + C + C + t_s$.

The previous algorithm can be implemented in C as follows:

```
void monProFIOS(hugeInt r, hugeInt a, hugeInt b, hugeInt n, uInt n0Prime, uInt
rLen) {
    hugeInt t = NULL;  newNum(&t, rLen + 1);
    int i, j;  bigInt cs, cc;  uInt m;
    for (i = 0; i < rLen; i++) {
        cs = (bigInt) a[0] * (bigInt) b[i] + (bigInt) t[0];
        m = (uInt) cs * n0Prime;
        cc = cs >> W;
        cs = (uInt) cs + (bigInt) m * (bigInt) n[0];
        cc += cs >> W;
        for (j = 1; j < rLen; j++) {
            cs = (bigInt) a[j] * (bigInt) b[i] + (bigInt) t[j] + (uInt) cc;
            cc = (cc >> W) + (cs >> W);
            cs = (bigInt) m * (bigInt) n[j] + (uInt) cs;
            cc += cs >> W;
            t[j - 1] = (uInt) cs;
        }
        cc += (bigInt) t[rLen];
        t[rLen - 1] = (uInt) cc;
        t[rLen] = cc >> W;
    }
    if (t[rLen] == 0) {
        ASSIGN(r, t, rLen);
    } else {
        uInt c = 0;
        for (i = 0; i < rLen; i++) {
            cs = (bigInt) t[i] - (bigInt) n[i] - (bigInt) c;
            r[i] = (uInt) cs;
            c = (cs >> W) & 1;
        }
    }
    free(t);
}
```

List 16 the Enhanced FIOS Montgomery Multiplication method.

3.4.2.2. Montgomery Exponentiation

The Montgomery unit contains another set of functions to calculate the modular exponentiation using Montgomery Multiplication. There are two algorithms to calculate the modular exponentiation, Binary Algorithm and Signed Digit Recording (SDR) Algorithm.

3.4.2.2.1. Binary Montgomery Exponentiation

In order to calculate the modular exponentiation of the given numbers, the Binary Montgomery exponentiation algorithm works depending on Montgomery Multiplication. This algorithm is implemented in Montgomery unit directly from (32) and (33).

```
void monExpBin(hugeInt res, hugeInt m, PreCalcs pc, hugeInt n, uInt rLen, int
monMode) {
    hugeInt c = NULL;  hugeInt s = NULL;
    newNum(&c, rLen);  newNum(&s, rLen);
    ASSIGN(c, pc.modRN, rLen);
    monPro(s, m, pc.modR2N, n, pc.n0Prime, rLen, monMode);
    unsigned int i;
    for (i = 0; i < pc.eudLen; i++) {
        if (pc.eud[i] == 1) {
            monPro(c, s, c, n, pc.n0Prime, rLen, monMode);
        }
        monPro(s, s, s, n, pc.n0Prime, rLen, monMode);
    }
    monPro(res, c, pc.one, n, pc.n0Prime, rLen, monMode);
    free(c); free(s);
}
```

List 17 the Binary Montgomery Modular Exponentiation, (32).

3.4.2.2.2. Signed Digit Recording Montgomery Exponentiation

SDR Montgomery Exponentiation is another algorithm to calculate the modular exponentiation depending on Montgomery Multiplication. Signed digit recording represents the binary number using three symbols 1, 0, -1; this representation may be called redundant number representation. According to (31) this algorithm should be faster than standard Binary Exponentiation by about 11.11%.

```
void monExpSdr(hugeInt res, hugeInt m, PreCalcs pc, hugeInt n, uInt rLen, int
monMode) {
    hugeInt d, c, s;  newNum(&d, rLen);  newNum(&c, rLen);  newNum(&s, rLen);
    ASSIGN(c, pc.modRN, rLen);  ASSIGN(d, pc.modRN, rLen);
    monPro(s, m, pc.modR2N, n, pc.n0Prime, rLen, monMode);
    unsigned int i;
    for (i = 0; i < pc.esdLen; i++) {
        if (pc.esd[i] == 1)
            monPro(c, c, s, n, pc.n0Prime, rLen, monMode);
        else if(pc.esd[i] == -1)
            monPro(d, d, s, n, pc.n0Prime, rLen, monMode);
        monPro(s, s, s, n, pc.n0Prime, rLen, monMode);
    }
    monPro(c, c, pc.one, n, pc.n0Prime, rLen, monMode);
    monPro(d, d, pc.one, n, pc.n0Prime, rLen, monMode);
    MOD_INV(d, d, n, rLen); MOD_MUL(res, c, d, n, rLen);
    free(d);  free(c);  free(s);
}
```

List 18 the Signed Digit Recording Montgomery modular exponentiation, (31).

3.4.3. CryptoSEQ Program:

The main role of this program is to help to decide on the best RSA tuned sequential algorithm, thus, this program is the sequential experiment in the project. It mainly contains *"RSA Class"* in order to perform RSA encryption and decryption.

3.4.3.1. RSA Class

This class provides the main functionality to encrypt and decrypt files or buffers using the RSA algorithm by providing four methods *"encrypt", "decrypt", "encryptFile",* and *"decryptFile"*. The methods *"encrypt"* and *"decrypt"* are for encrypting and decrypting buffers of bytes, whereas the methods *"encryptFile"* and *"decryptFile"* can be used for encrypting and decrypting files. In case of encrypting and decrypting buffers, the maximum buffer length should equal the key length in bytes minus 12 bytes. These 12 free bytes are used to encrypt and decrypt the block using PKCS#1 standard. However, when encrypting and decrypting files, the *"encryptFile"* and *"decryptFile"* methods segment the files into blocks with suitable lengths and then apply the PKCS#1 standard to each block.

The PKCS#1 standard can be applied by formatting the message block before encryption in a special way, and then encrypting the block using RSA to produce the cipher text. When it is time to decrypt the cipher text and after decrypting using RSA, the format of the message is validated to the original PKCS#1 format. This process makes sure that the original message is retrieved.

By using PKCS#1 the encryption block is formatted as follows:

- At the very beginning there is a leading zero byte. This byte will represent the most significant byte in the encryption block, so the condition that the message as a number should be less than the modulus will always be satisfied no matter what the message is.
- The next byte should have a value which represents the current cryptographic operation, whether it is signing or encryption. For signing, the second byte should have the value 0x01, whereas for encryption this byte should have the value 0x02.
- If it is a signing mode, all the next bytes are set to have the value 0xFF, the number of these bytes should be at least 8 bytes, whereas, in the encryption mode these bytes are set to have some random numbers.
- After it there is a separate byte with value zero.
- Finally, the original message bytes start to be recorded.

List 19 shows the C code of the PKCS#1 format method.

```
uByte* format_pkcs1(uByte* EB, uInt LenMod, uByte BlockType, const uByte*
data, uInt LenData) {
    int lps = LenMod - 3 - LenData;
    if (lps < 8) {
        return NULL;
    }
    uByte* hlp = EB;
    *hlp++ = 0x00;
    switch (BlockType) {
        case BLOCKTYPE_SIGN:
            *hlp++ = 0x01;
            while (lps-- > 0) {
                *hlp++ = 0xff;
            }
            break;
        case BLOCKTYPE_ENCR:
            *hlp++ = 0x02;
            uInt iseed = (uInt) time(NULL);
            srand(iseed);
            while (lps-- > 0) {
                do {
                    *hlp = rand() * 0x786532 * rand() * rand();
                } while (*hlp == 0);
                ++hlp;
            }
    }
    *hlp++ = 0x00;
    for (unsigned int l = 1; l <= LenData; l++) {
        *hlp++ = *data++;
    }
    return (uByte*) EB;
}
```

List 19 the PKCS#1 format method.

Thus, the *"encrypt"* method formats the block then calculates the value $c = m^e mod\ n$ by calling the Montgomery Modular Exponentiation method. Since, this class is for doing the experiment there is an extra parameter for the *"encrypt"* method to define the Montgomery Modular Exponentiation version that should be used in the encryption. The same logic is applied to the *"decrypt"* function. List 20 shows the implementation of the encrypt function.

A parameter of type *"RsaKey"* should be provided to the constructor of RSA class in order to create a new instance. The *"RsaKey"* variable contains two numbers: the exponent and the modulus. This means that for a full encryption/decryption process, two "Rsa" instances are required; the first one holds the public exponent and the modulus, while the other holds the private exponent and the modulus. Once the new instance is created, all the pre-calculations for the Montgomery methods are performed. This means that the SDR format of the exponent is calculated and stored, the numbers $n_0^{'}$, $R\ mod\ N$ and $R^2 mod\ N$ are calculated and stored in a local structure variable called *"pCals"*.

```
int Rsa::encrypt(hugeInt ciph, uByte *mess, uInt mLen, int mode) {
    int BLen = key.sLen * (W >> 3);
    uByte* encBlock = new uByte[BLen];
    if (format_pkcs1(encBlock, BLen, BLOCKTYPE_ENCR, mess, mLen) == NULL) {
        delete[] encBlock;
        return false;
    }
    fromInvBytes(ciph, key.sLen, encBlock);
    delete[] encBlock;
    switch (mode) {
        case BIN_FIOS:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, BIN, FIOS);
            break;
        case BIN_SOS:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, BIN, SOS);
            break;
        case BIN_CIOS:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, BIN, CIOS);
            break;
        case BIN_FIOS_MOD:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, BIN, FIOS_MOD);
            break;
        case SDR_FIOS:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, SDR, FIOS);
            break;
        case SDR_SOS:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, SDR, SOS);
            break;
        case SDR_CIOS:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, SDR, CIOS);
            break;
        case SDR_FIOS_MOD:
            monExp(ciph, ciph, pCalcs, key.n, key.sLen, SDR, FIOS_MOD);
            break;
        default:
            return false;
    }

    return true;
}
```

List 20 RSA encrypt method implementation

3.4.4. Key Generator (Keygen) Program:

The role of this program is to generated valid RSA key pairs of a specified length. Although, there are a lot of key generation tools, it is important to have another one due to restrictions on the key lengths. In addition, key generation tools store the keys in files with special formats. So, because most of the requirements are already implemented it will be not very difficult to write this program. There is an extra class responsible for generating RSA key pairs called the *"RsaKeyPair"* class.

3.4.4.1. RsaKeyPair Class

The main role of this class is to generating a new RSA key pair. This can be done by passing the required key size in bits and creating a new instance. The new instance will

generate the key pair directly, so it will be possible to get the keys using the methods *"getPublicKey"* and *"getPrivateKey"* and then write them to some external files. The protection of the generated private key is outside the scope of the project, since the main purpose is developing a parallel version of RSA encryption and decryption in order to enhance the performance in the desktop computers of today.

The job of this program will be completed after generating all the keys that are required in the experiment. Using this program, some RSA key pairs are generated and stored in external files. The generated key lengths are: 512, 1024, 1536, 2048, 2560, 3072, 3584, 4096, 4608, 5120, 5632, 6144 and 6656 bit length. Each key pair is stored in three files for public exponent, private exponent and the modulus. Now with all the requirements ready, the sequential experiment can take place to discover the best tuned sequential RSA encryption/decryption version.

3.5. Sequential Performance Benchmark

It is possible to benchmark the performance of the various Montgomery Modular Exponentiation algorithms by running the program *CryptoSEQ* on a computer system. This program executes all currently implemented sequential versions for RSA encryption and decryption which are: *Binary FIOS, Binary SOS, Binary CIOS, Binary FIOS_Modified, SDR FIOS, SDR SOS, SDR CIOS and SDR FIOS_Modified*, and writes the execution time to an external text file. The key sizes that are used in the experiment are: 512 bit, 1024 bit, 1536 bit, 2048 bit, 2560 bit, 3072 bit, 3584 bit, 4096 bit, 4608 bit, 5120 bit, 5632 bit, 6144 bit, and 6656 bit which are generated previously and stored in external files.

All the sequential versions have been measured on multi-core machine, comprising *two Intel(R) Xeon(R) CPU E5506 quad-core processors* running at *2.13GHz* and *12000 MB RAM*, with C compiler *gcc version 4.1.2 20080704* running under *"redhat-linux 2.6.18-238.19.1.el5 x86_64"* operating system. So, the CryptoSEQ program was executed to compare the performance of the various Montgomery Modular Exponentiation algorithms. To obtain reliable readings the measurement is repeated three times and the median is taken. There are two experiments - one using a compiler optimization feature i.e. turning the flag –O3 on, whereas the other is without code optimization.

First Experiment (E1) with code optimization:

key	BIN FIOS	BIN SOS	BIN CIOS	BIN FIOS Mod	SDR FIOS	SDR SOS	SDR CIOS	SDR FIOS Mod
512	0	0	0	0	0	0	0	0
1024	0.04	0.02	0.01	0.01	0.03	0.01	0.01	0.01
1536	0.13	0.06	0.05	0.04	0.12	0.05	0.05	0.04
2048	0.31	0.14	0.12	0.1	0.28	0.13	0.11	0.09
2560	0.61	0.33	0.25	0.2	0.55	0.3	0.23	0.19
3072	1.06	0.57	0.43	0.35	0.96	0.52	0.4	0.33
3584	1.71	0.89	0.67	0.55	1.51	0.82	0.62	0.52

4096	2.5	1.34	1	0.83	2.25	1.22	0.92	0.77
4608	3.57	1.92	1.43	1.18	3.21	1.74	1.31	1.1
5120	4.94	2.59	1.93	1.61	4.37	2.37	1.77	1.49
5632	6.59	3.47	2.59	2.16	5.83	3.16	2.36	1.98
6144	8.44	4.53	3.37	2.8	7.53	4.06	3.03	2.54
6656	10.73	5.76	4.26	3.57	9.66	5.21	3.87	3.3

Table 1 Montgomery Modular Exponentiation Algorithms execution time using various key sizes after optimizing the code.

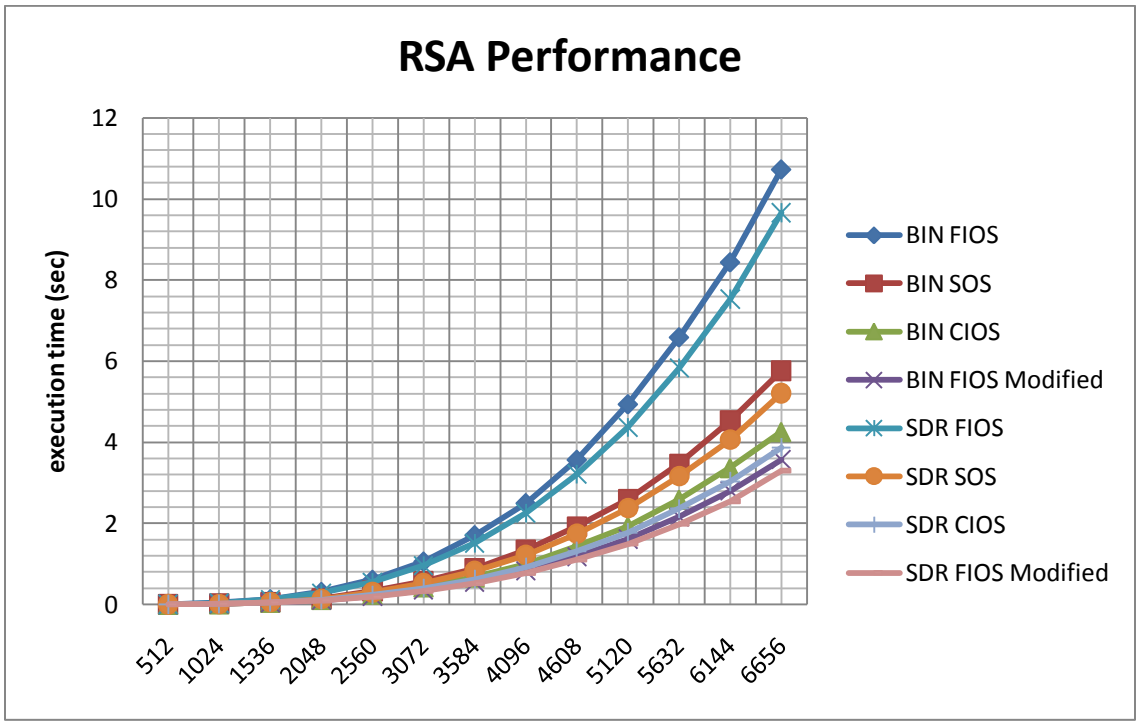

Figure 7 The performance of various Montgomery Modular Exponentiation versions using various key sizes and optimized code

Second Experiment (E2) without code optimization:

Key	BIN FIOS	BIN SOS	BIN CIOS	BIN FIOS Mod	SDR FIOS	SDR SOS	SDR CIOS	SDR FIOS Mod
512	0.01	0.01	0	0	0.01	0.01	0	0
1024	0.11	0.06	0.06	0.05	0.1	0.07	0.06	0.05
1536	0.38	0.25	0.21	0.17	0.35	0.24	0.21	0.17
2048	0.85	0.54	0.49	0.39	0.79	0.51	0.47	0.38
2560	1.65	1.05	0.94	0.76	1.52	0.98	0.88	0.73
3072	2.89	1.84	1.64	1.33	2.55	1.75	1.56	1.29
3584	4.56	2.84	2.56	2.08	4.14	2.63	2.38	1.96
4096	6.8	4.26	3.86	3.14	6.17	4.07	3.55	2.91
4608	9.73	6.01	5.51	4.48	8.87	5.72	5.11	4.21
5120	13.37	8.44	7.57	6.17	11.98	7.54	6.84	5.61
5632	17.71	11.06	10.01	8.16	15.98	10.15	9.13	7.48
6144	23.06	14.37	13.47	10.6	20.88	13.23	11.78	9.65
6656	29.94	18.64	16.83	13.64	26.47	16.7	15.26	12.35

Table 2 The execution time of Montgomery Modular Exponentiation Algorithms using various key sizes, (no code optimizing).

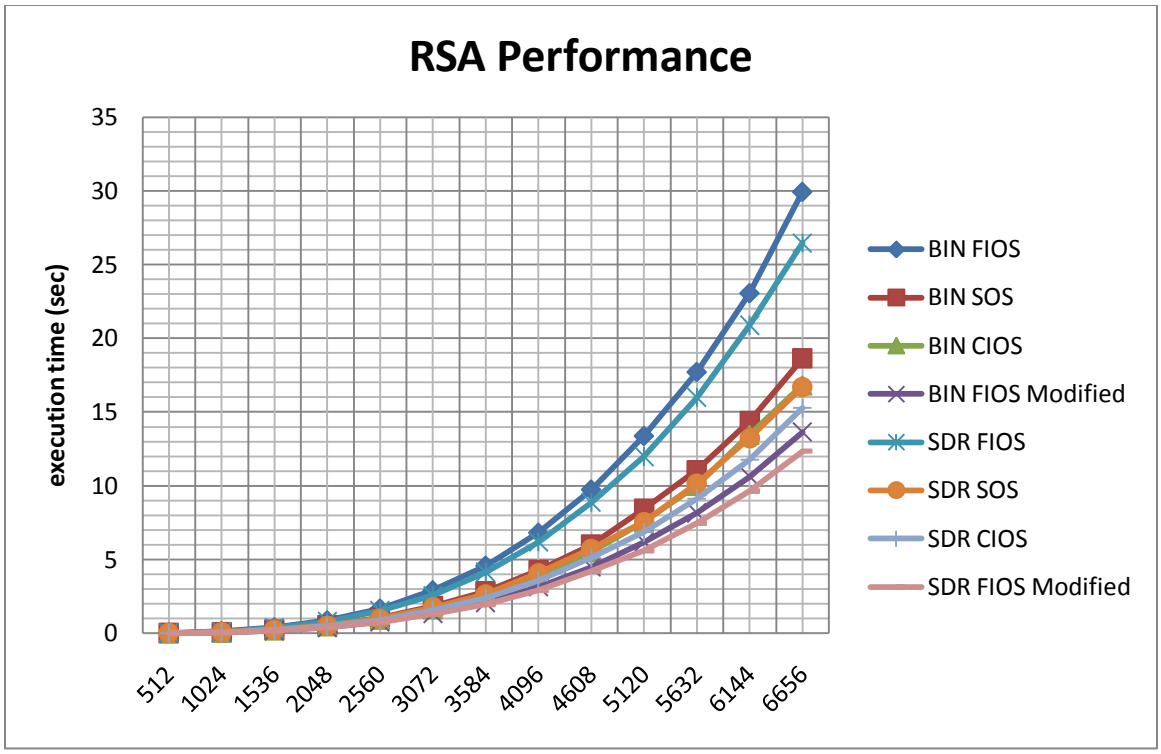

Figure 8 The performance of various Montgomery Modular Exponentiation versions using various key sizes, (no code optimization)

From the previous graphs, it is obvious that whether or not the code is optimized, the best performance was achieved by the algorithm *"SDR FIOS Modified"* which employs *"SDR binary modular exponentiation"* with *"FIOS Modified Montgomery Multiplication"* algorithm. The worst performance was achieved by *"Binary FIOS"* which employs *"Binary modular exponentiation"* with *"FIOS Montgomery Multiplication"* directly implemented from (14). The algorithm *"SDR FIOS Modified"* is about 3.25 times faster than *"Binary FIOS"* when the code is optimized by the compiler, while it is about 2.4 times faster when the code is not optimized. This is expected because the new FIOS Montgomery Multiplication version is very simple with just two interleaved loops, thus it can be optimized better against the more complicated ones.

To conclude, it is obvious that SDR with FIOS Modified can be the best tuned sequential version to perform RSA encryption and decryption. So, it will act as a reference for the parallel implementations that will be designed later. However, for simplicity and because it is not very different from the original one, *"FIOS Modified Montgomery Multiplication"* will be called *"FIOS Montgomery Multiplication"*.

Chapter 4. Parallel Implementation and Performance

In this chapter, a naive parallel FIOS version will be developed using OpenMP and the parallel SHS version will be implemented directly from (34) using MPI. The performance of these two versions will be compared with the best tuned sequential performance of Montgomery Modular Exponentiation. As a result of this chapter there will be some analysis for the naive parallel FIOS with problem identification in order to try to provide solutions in the next chapter.

4.1. Why OpenMP

OpenMP technology is chosen for the RSA parallel version because of various reasons:

1. OpenMP adopts a shared memory model which is very convenient for multi-core computer systems and performs faster than other models.
2. It uses threads as a computational unit, so creating new processing units i.e. creating new threads does not take much time compared with creating new processes.
3. Learning and understanding OpenMP is very easy because of plenty of tutorials which are easily available.
4. OpenMP encourages parallelizing loops in a structured way. However, RSA encryption/decryption provides loops which can be parallelized.

4.2. Naive Parallel FIOS OpenMP

It is obvious that the simplest Montgomery Multiplication algorithm is the FIOS version with two interleaved for loops. This algorithm became even simpler and much faster after removing the call of the function *"add"* in the internal loop. This makes this algorithm the strongest candidate to be parallelized. A first glance at the algorithm will give two possibilities for parallelism. The first is execute the outer loop in parallel whereas the second one is parallelising the internal loop. The decision will be taken depending on the data dependency that exists in each method.

4.2.1. Data Dependency Analysis

List 21 shows the main structure of FIOS Montgomery Multiplication.

$$
\begin{aligned}
&\textbf{for } i \leftarrow 0 \textbf{ to } s-1 \\
&\quad (C_1, C_0, S) \leftarrow t_0 + a_0 . b_i \\
&\quad m \leftarrow S.n_0 \textbf{ mod } 2^w \\
&\quad \ldots \\
&\quad \textbf{for } j \leftarrow 1 \textbf{ to } s-1 \\
&\qquad (C, S) \leftarrow t_j + a_j . b_i + C_0 \\
&\qquad \ldots \\
&\qquad t_{j-1} \leftarrow S \\
&\quad \ldots
\end{aligned}
$$

List 21 the main structure of FIOS Montgomery Multiplication

From this main structure it can be seen that there are carries between each of t_j values, furthermore, the calculation of t_j cannot be done before calculating the value t_{j-1}. Moreover, the value m_i cannot be calculated before calculation of the value $t_{i-1\ 0}$.

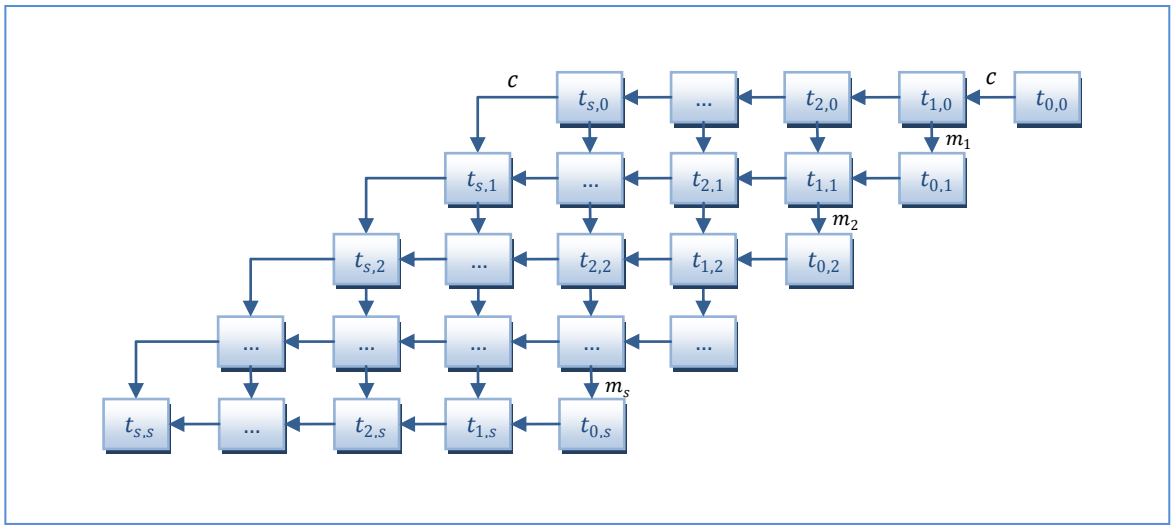

Figure 9 Data dependency diagram in FIOS Montgomery Multiplication Algorithm.

Hence, the calculation of m requires the value t_0, which is changing after every iteration of the outer loop. In other words, m_i depends directly on the value t_{i-1_0} from the previous outer loop iteration, however, the value t_0 will be calculated fully in the previous iteration's first internal loop iteration. Another important thing is that the value $t_{i,j}$ cannot be calculated before calculating the value $t_{i-1,j+1}$.

4.2.2. Method 1: Outer Loop Parallelising

The advantage here is that the carry c is always processed in the thread itself and there is no need to send the carry to other threads except the final carry, which is considered acceptable. However, when multiple threads execute the outer loop this means -by examining the Figure 9- that the first thread should start executing and all other threads should wait until it processes the value $t_{1,1}$ then thread 2 can start, whereas the others still waiting until thread 2 processes the value $t_{1,2}$ and so on. There should still be synchronization between all the threads in order to avoid a *"data race condition"*. Data race can happen when thread 2 for example reads $t_{2,0}$ before thread 1 writes this value. Hence, each thread should wait for $t_1..t_s$ values along with the final carry c, in other words each thread should wait S times for a variable. By taking the overall number of threads in consideration the number of waiting times can be calculated using the formula:

$$S^2 \cdot (1 - \frac{1}{threads\ number})$$

This means that when the number of threads is decreased the total number of waits decreases as well until it becomes 0 when the algorithm is executed sequentially.

It is obvious that the number of synchronization instructions required is huge and this will make the OpenMP performance very poor.

4.2.3. Method 2: Internal Loop Parallelising

The advantage here is that threads are no longer needed to wait the t values because all of them are calculating different parts of the t array at the same time using the same m_i. However, the carry can cause problems since each thread needs the final carry of the previous thread neighbour. This problem can be solved by postponing the carry addition until all the threads finish the calculation of the array t. So this step can make the result correct when each one adds and propagates the carry produced from the previous thread to the values that it has processed in array t.

Threads should enter the internal loop at the same time and finish at the same time. For the next i_{th} iteration the value t_{i-1_0} is ready for all threads so they can calculate the value m_i and enter the loop. In OpenMP there will be a need for a *"Barrier"* before the internal loop to guarantee that all threads enter the internal loop at the same time. The cost of this *"Barrier"* will be less than the cost of synchronization in the method 1, especially as OpenMP does not support waiting on a status, so the synchronization in Method 1 should be done either by busy waiting or by using POSIX semaphores and Locks which is alien to OpenMP. However, the *"Barrier"* before internal loop with the implicit barrier after the internal loop means that there will be $2 * S$ barrier synchronization which can still adversely affect performance, Furthermore, the critical section in the outer loop will make the performance worse.

4.2.4. Naive Parallel FIOS Implementation

It is obvious that Method 2 is better than Method 1 because it needs less thread synchronization. However, regarding to the carry dependency problem, the solution which is suggested before will be used. This means that threads will calculate the values t_j and each thread will end up with a carry of two words. This carry should be added to t array starting from the right index. This index can be calculated using the formula:

$$index = (thread\ id + 1) \cdot \frac{S}{threads\ number}$$

To make the segmentation of the array t easier, the initial length of array t become $2.S + 1$ so that every thread works on his chunk which equals $\frac{S}{threads\ number}$ and adds the resulting carry to the right place of array t and then starts the new iteration to process the array t_i starting from index i.

List 22 shows the implementation of the Naive Parallel FIOS algorithm.

```
void monProPFIOS(hugeInt r, hugeInt a, hugeInt b, hugeInt n, uInt n0Prime,
uInt rLen) {
    hugeInt t = NULL;  newNum(&t, 2 * rLen + 1);
    int i, j;
    bigInt cs, cc;
    omp_set_dynamic(0);
    omp_set_num_threads(thdsNo);
#pragma omp parallel default(none) private(i, j, cs, cc) firstprivate(rLen,
n0Prime, a, b, n, t, thdsNo)
    {
        int tid = omp_get_thread_num();
        int chk = rLen / thdsNo;
        for (i = 0; i < rLen; i++) {
            uInt m = (t[i] + a[0] * b[i]) * n0Prime;
            //at this point all threads should be synchronized to calculate m
#pragma omp barrier
            cc = 0;
#pragma omp for schedule (static, chk)
            for (j = 0; j < rLen; j++) {
                cs = (bigInt)a[j] * (bigInt)b[i] + (bigInt)t[i + j]+ (uInt)cc;
                cc = (cc >> W) + (cs >> W);
                cs = (bigInt) m * (bigInt) n[j] + (uInt) cs;
                cc += cs >> W;
                t[i + j] = (uInt) cs;
            }
            //the thread's carry is added to t starting from ji
            int ji = (tid + 1) * chk + i;
            //every thread adds its carry to t,one thread at a time is allowed
#pragma omp critical
            {
                cs = (bigInt) t[ji] + (bigInt) cc;
                t[ji] = (uInt) cs;
                add(t, cs >> W, ji + 1, 2 * rLen + 1);
            }
        }
    }
    if (t[2 * rLen] == 0) { // equivalent to (t < R)
        ASSIGN(r, t + rLen, rLen);
    } else {
        cc = 0;
        for (i = 0; i < rLen; i++) {
            cs = (bigInt) t[rLen + i] - (bigInt) n[i] - (bigInt) cc;
            r[i] = (uInt) cs;
            cc = (cs >> W) & 1;
        }
    }
    free(t);
}
```

List 22 Naive Parallel FIOS C + OpenMP Implementation

4.3. Parallel Separated Hybrid Scanning (pSHS)

In (34), there is a design for a very efficient parallel algorithm for Montgomery Multiplication. The authors made use of various Montgomery Multiplication versions and developed a Hybrid Separated version. They provide an algorithm based on message

passing schema as a general solution. Their algorithm is separated, which means that the multiplication and reduction stages are performed separately. In their paper they discriminate between multiplication calculations and reduction calculations by depicting these calculations as white boxes and shaded boxes respectively.

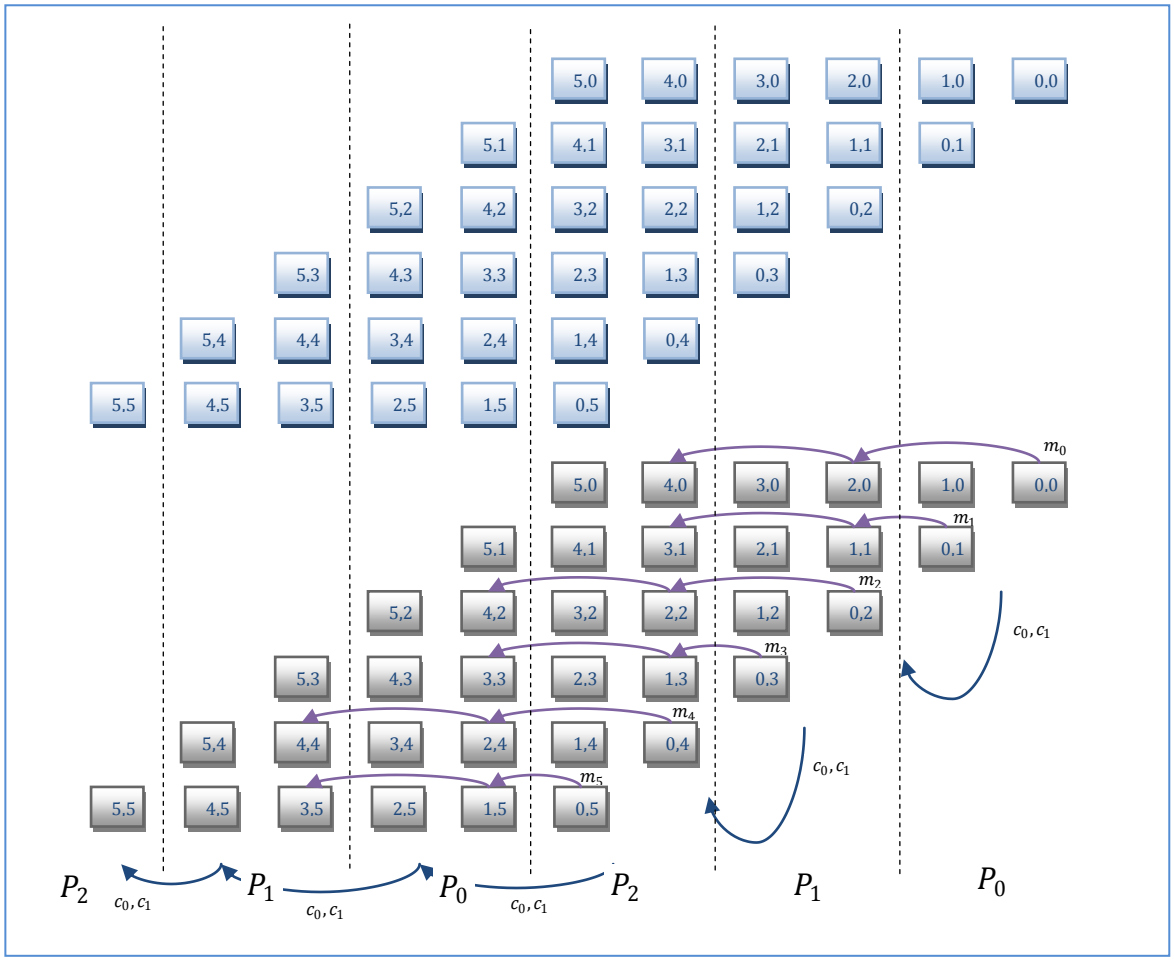

Figure 10 Parallel Separated Hybrid Scaning Example $S = 6$, $Processes\ Number = 3$ (34).

From Figure 10 it can be seen that they divide the work by processing - at least every two adjacent columns as a one group. Inside each group they are using SOS Montgomery Multiplication to calculate the sub-result whereas overall, it can be seen that they are using the Product Scanning method. This is the reason why the algorithm called Hybrid. In order to work in parallel the input numbers size should be divisable by the number of processes.

Regarding to the synchronization between processes, each process waits for m_i value except the process which calculates the value. So always and for each m_i there will be $processes\ number - 1$ waiting times. Furthermore, each process waits for the carry from its neighbour which adds $2 \cdot processes\ number - 1$ waiting times. The overall

number of waiting times can be calculated using the formula: $(processes\ number - 1) \cdot (S + 2)$

This formula does not calculate the intermediate carry (in Figure 10 from box 0,5 to box 2,5), so if there is more than one process the total waiting times should be

$$(processes\ number - 1) \cdot (S + 2) + 1$$

In the algorithm there are two numbers p and q which are used mainly to divide the work. Furthermore, the multiplication $p \cdot q$ should divide the input size S. However, according to (34), the best performance can be achieved when $p = processes\ num$.

The pSHS algorithm is benchmarked in (34) using an FPGA device which is a special device. This means that implementing this algorithm on a general computer system may give different results.

4.3.1. MPI pSHS

Depending on (34), parallel Separated Hybrid Scanning can be implemented easily using MPI or using any message passing library because this algorithm in this paper is written using a message passing schema. List 23 shows the implementation of pSHS algorithm using C + MPI.

```
void monProPSHS(hugeInt r, hugeInt a, hugeInt b, hugeInt n, uInt n0Prime, uInt
rLen, int p) {
    int q, pid, nextPid, prevPid, k, jLimit, iLimit, cTag;
    q = rLen / p; //best performance [34], page(11)
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    hugeInt rt = NULL, m = NULL, c = NULL;  int rtLen = q + 2;
    newNum(&rt, rtLen);  newNum(&m, rLen);  newNum(&c, 2);
    MPI_Status status;
    nextPid = (pid + 1) % p;    prevPid = (pid > 0) ? pid - 1: p - 1;
    uInt ca;    bigInt cs;    int i, j, l, h;
    for (l = 0; l < (2 * rLen) / (p * q); l++) {
        k = pid * q + l * p * q;
        assign(rt, 0, rtLen);      iLimit = min(k + q, rLen);
        for (i = max(0, k - rLen + 1); i < iLimit; i++) {
            ca = 0;
            jLimit = min(k - i + q, rLen);
            for (j = max(0, k - i); j < jLimit; j++) {
                cs = (bigInt)a[j] * (bigInt)b[i] +
                                    (bigInt)rt[j - k + i] + (bigInt)ca;
                rt[j - k + i] = (uInt)cs;
                ca = cs >> W;
            }
            h = min(q, rLen - k + i);
            for (j = h; j < rtLen; j++){
                if (ca == 0) break;
                cs = (bigInt)rt[j] + (bigInt)ca;
                rt[j] = (uInt)cs;
                ca = cs >> W;
            }
        }
```

```
        iLimit = min(k + q, rLen);
        for (i = max(0, k - rLen + 1); i < iLimit; i++) {
            if (i >= k) {
                m[i] = n0Prime * rt[i - k];
                MPI_Send(&m[i], 1, MPI_INT, nextPid, i, MPI_COMM_WORLD);
            } else {
                if (i >= k - (p - 1) * q) {
                    MPI_Recv(&m[i], 1, MPI_INT, prevPid, i,
                                                MPI_COMM_WORLD, &status);
                }
                if (i >= k - (p - 2) * q) {
                    MPI_Send(&m[i], 1, MPI_INT, nextPid, i, MPI_COMM_WORLD);
                }
            }
            ca = 0;
            jLimit = min(k - i + q, rLen);
            for (j = max(0, k - i); j < jLimit; j++) {
                cs = (bigInt)n[j] * (bigInt)m[i] +
                                    (bigInt)rt[j - k + i] + (bigInt)ca;
                rt[j - k + i] = (uInt)cs;
                ca = cs >> W;
            }
            h = min(q, rLen - k + i);
            for (j = h; j < rtLen; j++){
                if (ca == 0) break;
                cs = (bigInt)rt[j] + (bigInt)ca;
                rt[j] = (uInt)cs;
                ca = cs >> W;
            }
            if ((i == min(k - 1, rLen - 1)) && (k != 0)) {
                cTag = max(0, k - rLen + 1);
                cTag += max(0, k - cTag) + rLen;
                MPI_Recv(c,2, MPI_INT, prevPid, cTag, MPI_COMM_WORLD,&status);
                ca = 0;
                for (j = 0; j < 2; j++){
                    cs = (bigInt)c[j] + (bigInt)rt[j] + (bigInt)ca;
                    rt[j] = (uInt)cs;
                    ca = cs >> W;
                }
                for (j = 2; j < rtLen; j++) {
                    if (ca == 0) break;
                    cs = (bigInt)rt[j] + (bigInt)ca;
                    rt[j] = (uInt)cs;
                    ca = cs >> W;
                }
            }
        }
        cTag = min(k + q, rLen) - 1;
        cTag += min(k - cTag + q, rLen) + rLen;
        MPI_Send(&rt[q], 2, MPI_INT, nextPid, cTag, MPI_COMM_WORLD);
        if (k >= rLen) {
            for (i = 0; i < q; i++) {
                r[k - rLen + i] = rt[i];
            }
        }
    }
```

```
    if (pid == 0) {
        MPI_Recv(c,2, MPI_INT, prevPid, 3 * rLen - 1, MPI_COMM_WORLD,&status);
    }
    MPI_Bcast(c, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Send(&r[pid * q], q, MPI_INT, nextPid, 3 * rLen, MPI_COMM_WORLD);
    MPI_Recv(&rt[0], q, MPI_INT, prevPid, 3 * rLen, MPI_COMM_WORLD, &status);
    for (i = 0; i < q; i++) {
        r[i + prevPid * q] = rt[i];
    }
    k = (prevPid > 0) ? prevPid - 1 : p - 1;
    for (i = 0; i < p - 2; i++) {
        MPI_Send(&rt[0],q,MPI_INT,nextPid, 3 * rLen + i + 1, MPI_COMM_WORLD);
        MPI_Recv(&rt[0], q, MPI_INT, prevPid, 3 * rLen + i + 1,
                                              MPI_COMM_WORLD, &status);
        for (j = 0; j < q; j++) {
            r[j + k * q] = rt[j];
        }
        k = (k > 0) ? k - 1 : p - 1;
    }
    if (c[0] != 0){
        ca = 0;
        for (i = 0; i < rLen; i++) {
            cs = (bigInt) r[i] - (bigInt) n[i] - (bigInt) ca;
            r[i] = (uInt) cs;
            ca = (cs >> W) & 1;
        }
    }
    free(rt);  free(m);  free(c);
}
```

List 23 pSHS C + MPI Implementation.

4.4. Parallel Performance Benchmark

In this section, the naive FIOS Montgomery Multiplication and pSHS Montgomery Multiplication will be benchmarked. Firstly, the multiplication and exponentiation performance will be measured by fixing the length of the processed numbers and increasing the number of processes. Secondly, the multiplication and exponentiation performance will be measured by fixing the number of processes and increasing the number length.

These two experiments will be executed on the same machine that has been used to benchmark the sequential versions of Montgomery Modular Exponentiation. Which is multi-core machine; comprising two *"Intel(R) Xeon(R) CPU E5506 quad-core processors"* running at *2.13GHz* and *12000 MB RAM*, with C compiler *gcc version 4.1.2 20080704*, *MPICH2*, running under *"redhat-linux 2.6.18-238.19.1.el5 x86_64"* operating system. Furthermore, the code is compiled with turning the flag *–O3* on i.e. code is optimized by the compiler. As a result this performance benchmark can expose some problems in the naive FIOS Montgomery Multiplication algorithm. However, any problems which show up can be solved in the next chapter.

4.4.1. Montgomery Multiplication Performance

To benchmark the multiplication performance, two experiments will be executed. Firstly, experiment 1 will be run by fixing the length of the input numbers and changing the number of cores that are executing the algorithms. Experiment 1 will result a speedup graph. Secondly, the execution times will be measured by fixing the number of cores which execute the algorithms and by changing the input numbers sizes.

4.4.1.1. Experiment 1: Speedup & Scalability

The length of the processed input numbers is fixed, which will be 645,120 bit. This length is suitable as it will take the computer machine a considerable amount of time to perform the multiplication while at the same time not exceeding the MPI buffer limit.

With length 645,120 bits or $S = 20{,}160$, Table 3 shows the execution time.

cores	pFIOS Naïve	pSHS
1	1.6	2.68
2	0.83	1.57
3	0.57	0.91
4	0.45	0.67
5	0.37	0.51
6	0.32	0.43
7	0.305	0.37
8	0.29	0.32

Table 3 pFIOS Naive vs. pSHS Montgomery Multiplication Execution Time

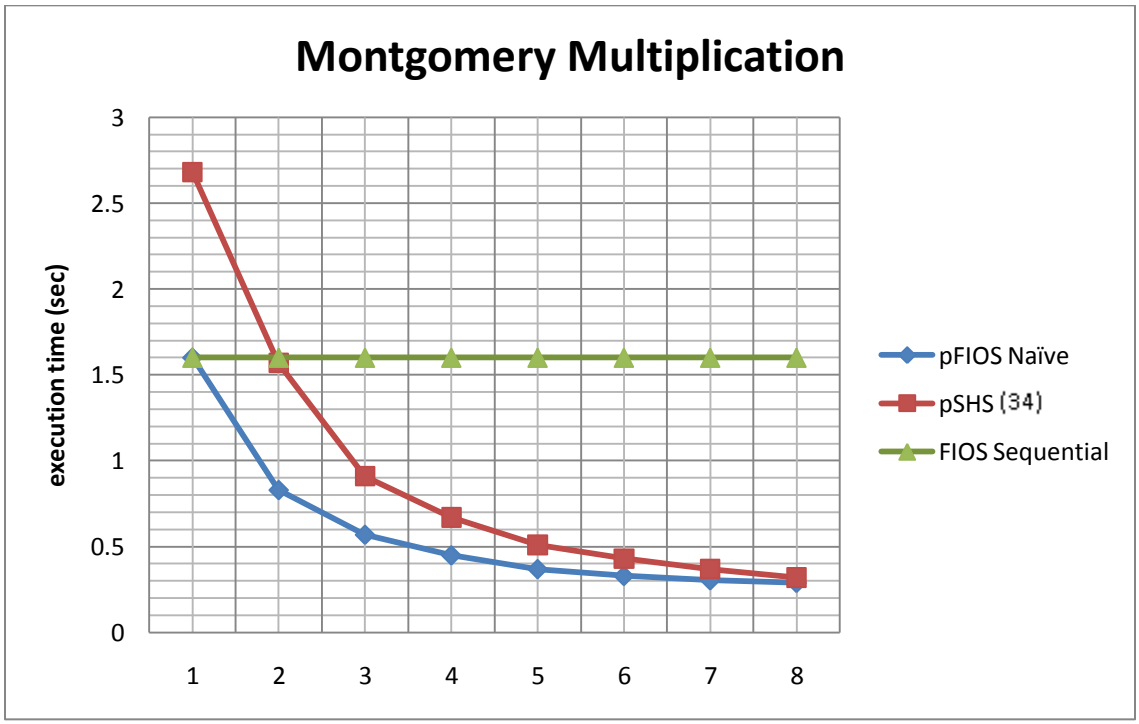

Figure 11 pFIOS naive vs. pSHS Montgomery Multiplication Execution time

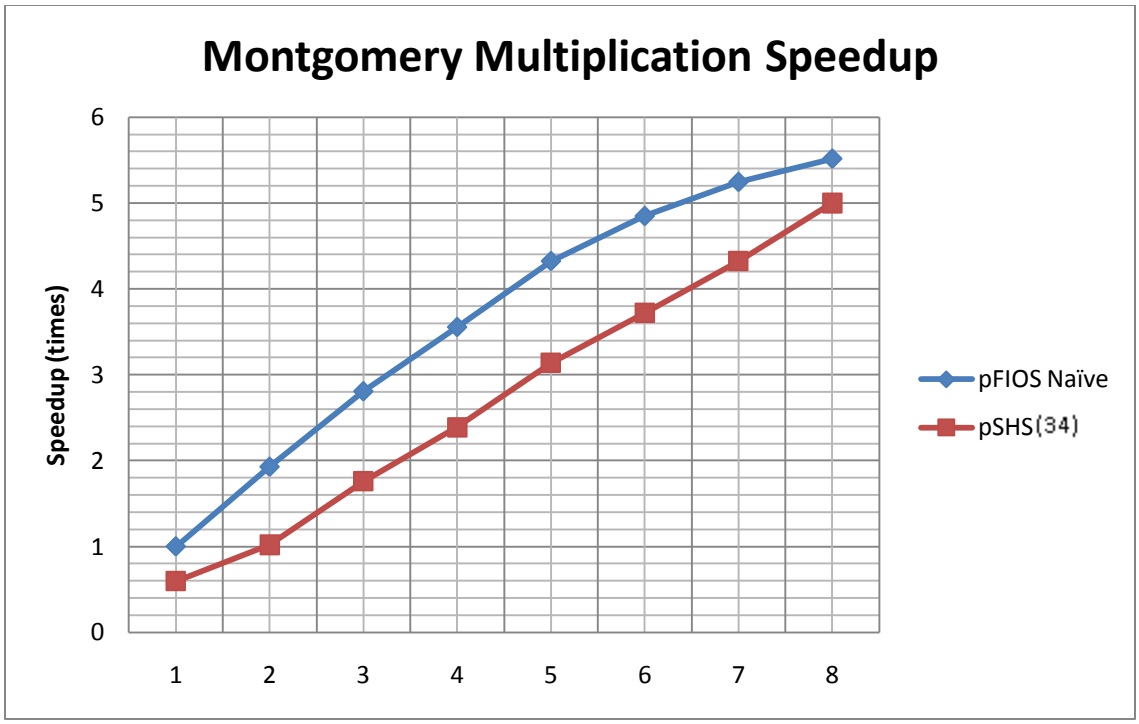

Figure 12 pFIOS naive vs. pSHS Montgomery Multiplication Speedup

Execution Time and speedup graphs show that pFIOS naïve has better performance than pSHS in multiplying very big numbers. However, as the number of cores increases (the processes number increases), pFIOS naïve performance begins to become worse and worse. Although pFIOS naïve has better performance, pSHS has better scalability. These graphs show that by increasing the number of threads pFIOS naïve will become worse than pSHS and this is the case in the exponentiation, where the Montgomery Multiplication algorithm is called again and again for much smaller numbers than the number that both algorithms are tested with now.

4.4.1.2. Experiment 2: Various Numbers Lengths

In experiment 2 the number of cores that execute the algorithm is fixed; and the input numbers' lengths are increased. This experiment will show how much each algorithm is sensitive to the amount of given work. The starting number length is $S = 512$ and the ending length is $S = 20096$ the difference between two adjacent lengths is 1224. All the lengths are divisible by 8 which is the fixed number of cores or the fixed number of processes. The table below shows the execution time associated with each input number size.

Number Size	pFIOS naïve	pSHS
512	0.005714	0.00162
1736	0.011696	0.004799
2960	0.019283	0.009775
4184	0.028543	0.015482

5408	0.040825	0.025219
6632	0.054359	0.0371
7856	0.06931	0.051271
9080	0.085042	0.067002
10304	0.10226	0.087178
11528	0.120689	0.107863
12752	0.141079	0.132104
13976	0.162869	0.158339
15200	0.187674	0.180949
16424	0.210456	0.210541
17648	0.236414	0.24231
18872	0.262722	0.276801
20096	0.285881	0.31435

Table 4 pFIOS naive vs. pSHS Montgomery Multiplication with different input numbers sizes

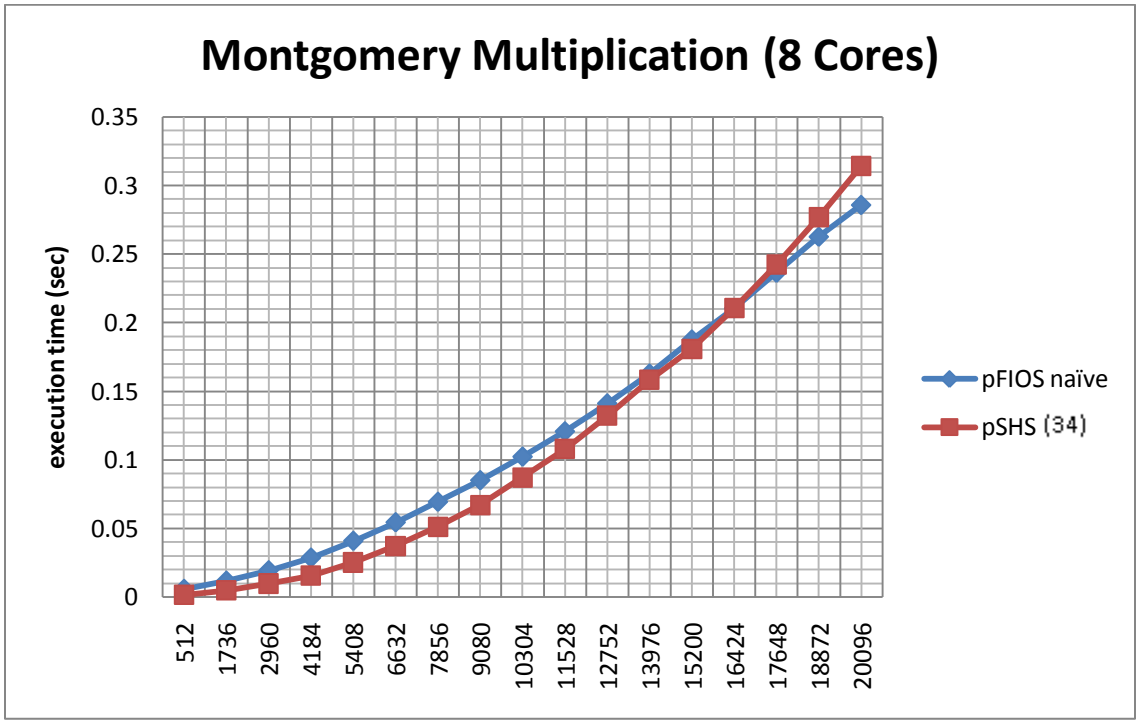

Figure 13 pFIOS naive vs. pSHS Montgomery Multiplication with different input numbers sizes using 8-core machine.

The graph shows that pFIOS naïve performs slower than pSHS when the input is relatively small. With very big numbers pFIOS naïve performs faster. This gives a clue that pSHS has less overhead than pFIOS naïve. However, in case of big numbers, pSHS performs worse because of the huge messages sent between processes and the delay encountered. Also here pSHS is better than pFIOS naïve, because in practical applications the number sizes are not very big. This is the case in the exponentiation where the Montgomery Multiplication input numbers are relatively small. Algorithm pSHS is very likely to perform faster than pFIOS naïve in the exponentiation modular.

4.4.2. Montgomery Modular Exponentiation Performance

To benchmark the modular exponentiation performance, the speedup and scalability graph will be shown by fixing the length of the input numbers and changing the number of cores that are executing the algorithms.

4.4.2.1. Speedup & Scalability

The length of the input numbers is fixed in this experiment. However the algorithm will be executed by a different number of cores. The length of number will be $S = 840$. This number is divisible by 1, 2, 3... 8. The exponentiation algorithm that will be adopted is SDR Montgomery Modular Exponentiation, because this algorithm performs faster than the Binary Montgomery Modular Exponentiation algorithm. Table 5 shows the execution time of Montgomery Modular Exponentiation based on pFIOS naïve and pSHS.

cores	pFIOS naïve	pSHS	FIOS
1	117.36	160.38	99.7
2	99.2	116.57	99.7
3	97.05	78.63	99.7
4	100.5	56.28	99.7
5	104.63	49.36	99.7
6	133.7	44.09	99.7
7	165.73	42.5	99.7
8	165.83	41.54	99.7

Table 5 pFIOS naive vs. pSHS Montgomery Modular Exponentiation Execution time

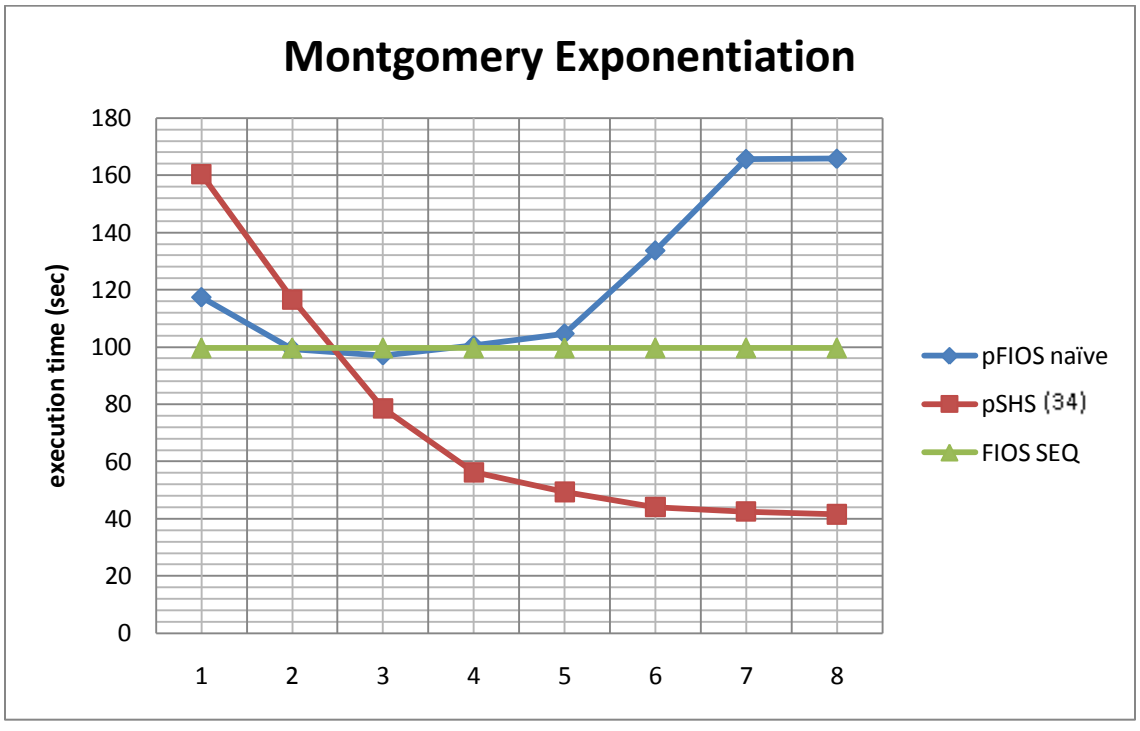

Figure 14 pFIOS naive vs. pSHS Montgomery Modular Exponentiation execution time.

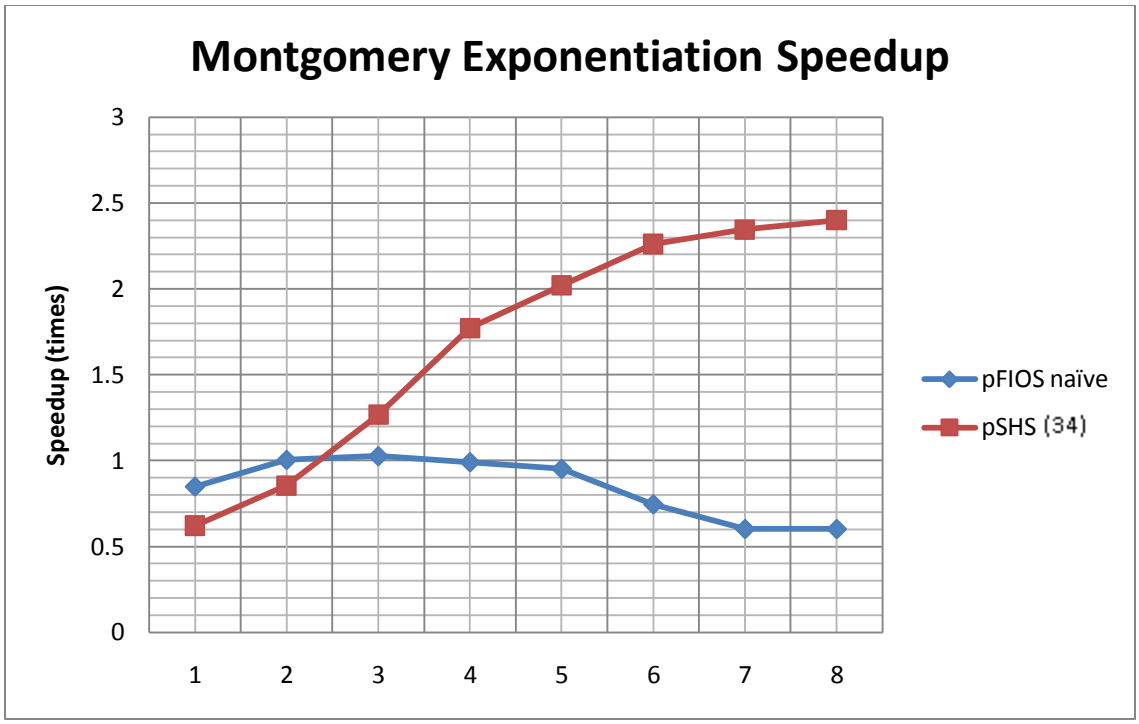

Figure 15 pFIOS naive vs. pSHS Montgomery Modular Exponentiation speedup.

Figure 14 and Figure 15 above show that pFIOS naïve performs very badly when it is used in a modular exponentiation algorithm. This happened because the exponentiation algorithm calls Montgomery Multiplication algorithm $S * 32$ times where S is the input numbers' length. So, the amount of work that is done by each thread is too small compared with the pure multiplication with very big input. The synchronization problems appeared now, which are caused by the two barriers (the implicit and explicit ones) and the critical section. However, pSHS shows consistent behavior even when it called for small input numbers many times.

To make the problem clearer, the *ompP* tool is used to show the actual threads' execution time and the overhead. The number of threads which execute the parallel algorithm is 8 using 8 cores machine. With previous conditions and after compiling and running the pFIOS naïve algorithm using this tool, the report is generated. The report shows that when 8 threads execute pFIOS naïve, there is a 46.42% overhead which is considered high and needs to be reduced. The same report shows that this synchronization overhead is caused by the Barrier, critical section, and for loop itself. List 24 shows the overhead report generated by the *ompP* tool.

```
------------------------------------------------------------------
----     ompP Overhead Analysis Report     --------------------------
------------------------------------------------------------------
Total runtime (wallclock)   : 1109.85 sec [8 threads]
Number of parallel regions  : 1
Parallel coverage           : 1109.08 sec (99.93%)

Parallel regions sorted by wallclock time:
            Type                          Location      Wallclock (%)
R00009  PARALLEL          montgomery_mp.c (129-153)    1109.08 (99.93)
                                              SUM    1109.08 (99.93)
Overheads wrt. each individual parallel region:
        Total   Ovhds    (%)   = Synch   (%)   + Imbal   (%)   + Limpar(%) + Mgmt   (%)
R00009  8872.61 4118.90 (46.42) 2699.75 (30.43) 1105.62 (12.46) 0.00 (0.00) 313.53 (3.53)

Overheads wrt. whole program:
        Total   Ovhds    (%)   = Synch   (%)   + Imbal   (%)   + Limpar (%) + Mgmt   (%)
R00009  8872.61 4118.90 (46.39) 2699.75 (30.41) 1105.62 (12.45) 0.00 (0.00) 313.53 (3.53)
   SUM  8872.61 4118.90 (46.39) 2699.75 (30.41) 1105.62 (12.45) 0.00 (0.00) 313.53 (3.53)
```

List 24 The ompP tool overhead report for pFIOS naïve Montgomery Modular Exponentiation Algorithm.

To conclude, pFIOS naïve shows acceptable performance in multiplying two big numbers with a good scalability and small overhead. However, when it is used in the SDR Montgomery Modular Exponentiation algorithm, synchronization overhead becomes very big because the input numbers are smaller and the threads are much more often waiting on the barriers and the critical section. This result is supported by *ompP* tool which is a specialized *OpenMP* tool to display valuable information about the execution time for each thread along with the overhead percentage.

On the other hand, pSHS shows very good performance in multiplication and exponentiation with a good scalability. However, it tends to perform slower when multiplying very big numbers. This can happen due to the huge buffer sizes sent and received between the processes. This is okay because in practical application like RSA encryption and decryption the numbers are not very big; furthermore, the Montgomery Modular Exponentiation algorithm is used.

Chapter 5. Experimental Results and Performance Tuning

The speedup graphs and *ompP* report from the previous chapter have shown that there is too much synchronization overhead in pFIOS naïve. This problem appears in the exponentiation algorithm more than in the multiplication algorithm where the input numbers are smaller and waiting on the barriers and critical section is more frequent. However, it is essential to get a better performance in the exponentiation algorithm because it is the algorithm that is used in the RSA encryption and decryption process.

To reduce the synchronization overhead, the barriers and critical section should be removed or at least make their execution less frequent. This solution needs algorithm code restructuring so that it is possible to eliminate all currently existing data races which will show up when the barriers are removed.

5.1. Parallel FIOS (pFIOS) Synchronization Reduction

After removing all the critical section and the barriers (the explicit and implicit) some data race conditions appear. They mainly occur after the loop when the threads try to make the sub-result correct by adding their carry to the right place in the array. Furthermore, the value of t_i is always changing after each iteration of the internal loop and the thread which changes this value is always the $thread_0$. So if other threads were faster than $thread_0$ they would read an invalid value of t_0 the thing that will produce wrong m_i and eventually a wrong result.

Regarding the first data race problem, this can be solved when each thread works on its part of the temporary array result t and postponed the carry adding until it is really important to proceed. When all threads postponed their carry addition this can reduce the much-needed synchronization. However, the carry addition will be crucial after each thread processes $\frac{S}{threads\ number}$ outer loop iterations. In the full algorithm execution this stage happens as many times as the number of threads which execute the algorithm. This means that in case of one thread, there will be just one carry addition stage and the algorithm will perform almost like the sequential one with just a slight difference. Figure 16 shows an example of execution when $S = 6$ and three threads are executing the algorithm. In this example the carry addition should be done three times because the number of threads is three.

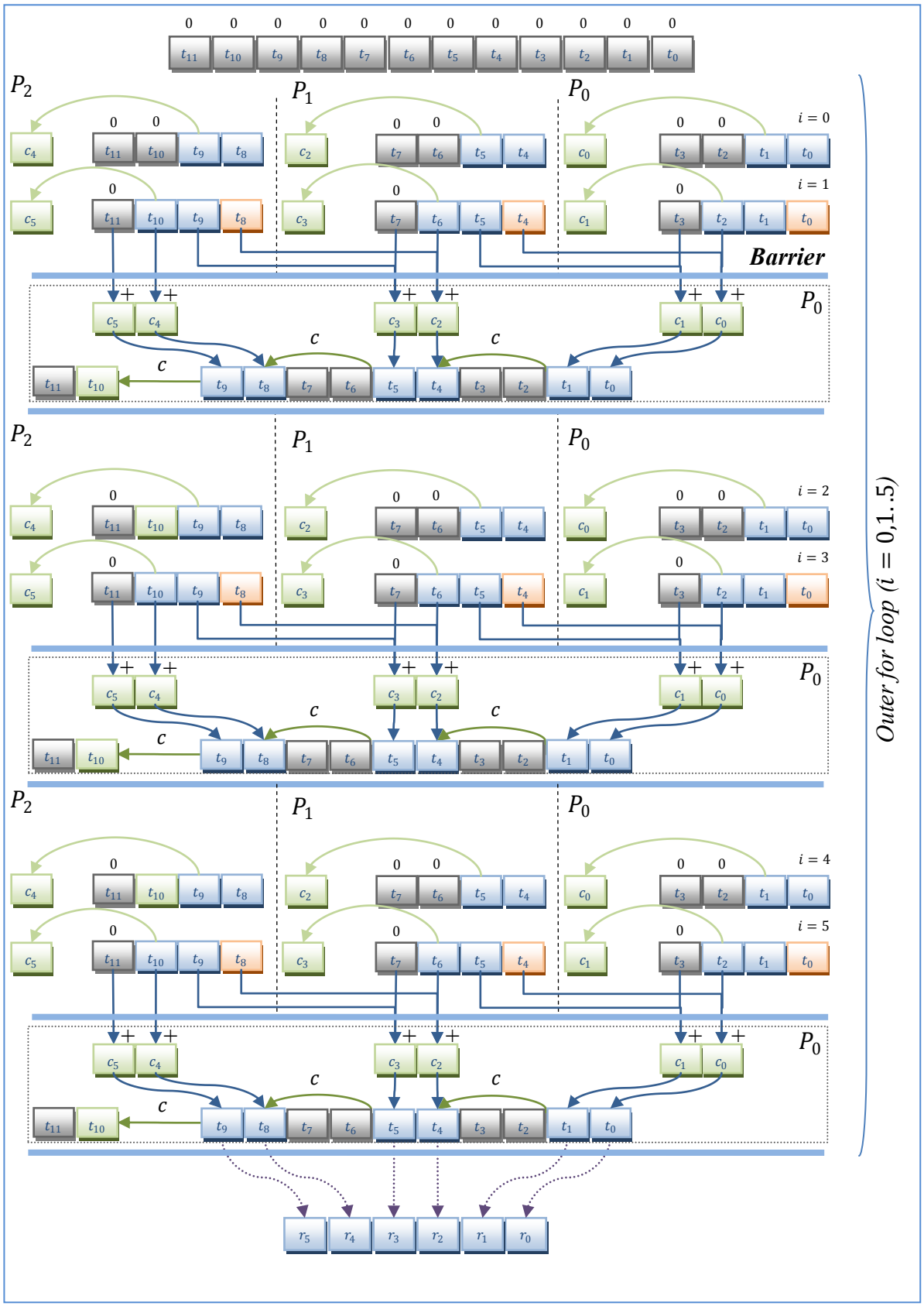

Figure 16 pFIOS full example $S = 6$, $threads = 3$.

Regarding the synchronization on t_i value, the data race here cannot be eliminated or reduced, so access to t_i should be protected by checking a condition and waiting until this condition is satisfied. This can be done by using a shared variable managed by $thread_0$ and which holds the $thread_0$'s currently processed index i. Thus, the other threads will read this variable and compare their local index i with this variable; if their index is smaller or equals the variable's value, it is safe to read the t_i value and calculate the right m_i otherwise, they should wait until the previous condition is satisfied.

5.2. New Parallel FIOS (pFIOS) Algorithm

The new pFIOS algorithm now has less synchronization overhead because the two barriers are executed less frequently (there are still two barriers before and after the carry addition to the sub-result t). The previous explicit barrier has been replaced with a busy waiting loop until a condition satisfied. Furthermore, the critical section has been removed completely. List 25 shows pFIOS C implementation after applying the previous changes. This version improves the pFIOS naive version implemented in previous chapter in List 22. It is obvious that the changes that have been done are algorithm structure changes. Mainly, involves postponing the carry addition until it is really important to carry on the computation. Furthermore, now each thread is working on its own t segment and stores all resulted carry to carry array (c.f. List 22 Naive Parallel FIOS C + OpenMP Implementation).

```
void monProPFIOS(hugeInt r, hugeInt a, hugeInt b, hugeInt n, uInt n0Prime,
uInt rLen) {
    hugeInt t = NULL;
    hugeInt t0s = NULL;
    bigInt* cc = NULL;
    newNum(&t, 2 * rLen);
    newNum(&t0s, rLen + 1);
    cc = calloc(rLen, sizeof(bigInt));
    int i, j, id;  bigInt cs;  uInt mi = 0, c = 0;
    omp_set_dynamic(0);
    omp_set_num_threads(thdsNo);
#pragma omp parallel default(none) private(i, j, id, cs)
        firstprivate(rLen, n0Prime, a, b, n, t, t0s, cc, thdsNo) shared(c, mi)
    {
        int tid = omp_get_thread_num();
        int chk = rLen / thdsNo;

        for (i = 0; i < rLen; i++) {
        //the thread should wait while mi is smaller than the
        //index that he is processing
#pragma omp flush(mi)
            while (i > mi){
                wait(0);
#pragma omp flush (mi)
            }
           //all threads calculates the m value based on t0s written by master
            uInt m = (t0s[i] + a[0] * b[i]) * n0Prime;
            int ci = tid * chk + i % chk;
            cc[ci] = 0;
            int jst = 2 * chk * tid + i % chk;
            int ji = 0;
            //This loop is processed in parallel, every thread works on his
            //part. The implicit barrier at the end of the loop is removed.
#pragma omp for schedule (static, chk) nowait
            for (j = 0; j < rLen; j++) {
                cs = (bigInt)a[j]*(bigInt)b[i]+(bigInt)t[jst+ji]+(uInt)cc[ci];
                cc[ci] = (cc[ci] >> W) + (cs >> W);
                cs = (bigInt) m * (bigInt) n[j] + (uInt) cs;
                cc[ci] += cs >> W;
                t[jst + ji++] = (uInt) cs;
            }
            //the thread 0 is responsible for writing the t0s values, and
            //managing the shared variable mi which tells other threads about
            //the master's current index.
#pragma omp master
            {
                t0s[i + 1] = t[jst + 1];
                mi = i + 1;
            }
```

```
            if ((i + 1) % chk == 0){ //Carry addition cannot be postponed more
#pragma omp barrier
#pragma omp master
                {
                    bigInt cb = 0;
                    for (id = 0; id < thdsNo; id++){
                        int js1 = chk * (2 * id + 1);
                        int js2 = chk * (2 * (id + 1));
                        int jd = chk * (2 * id);

                        for (j = 0; j < chk; j++){
                            cs = (js2 < 2 * rLen) ? t[js2 + j] : 0;
                            cs += (bigInt) t[js1 + j] + (uInt)cb;
                            cb = (bigInt)(cs >> W) + (bigInt)(cb >> W);
                            cs = (bigInt)((uInt)cs) +  (uInt)cc[id * chk + j];
                            cb += (bigInt)(cs >> W) +
                                                (bigInt)(cc[id*chk + j] >> W);
                            t[jd + j] = (uInt) cs;
                            t[js1 + j] = 0;
                            if (js2 < 2 * rLen) t[js2 + j] = 0;
                        }
                    }
                    c = (uInt)cb;
                    t0s[i + 1] = t[0]; t[chk * (2 * thdsNo - 1)] = c;
                }
#pragma omp barrier
            }
        }
    }
    int chk = rLen / thdsNo;
    if (c == 0) { //equivalent to (t < R)
        //collect the result from the right indices.
        for (id = 0; id < thdsNo; id++) {
            int js = chk * (2 * id);
            for (j = 0; j < chk; j++) {
                r[id * chk + j] = t[js + j];
            }
        }
    } else {  //collect result and subtract n from the it.
        c = 0;
        for (id = 0; id < thdsNo; id++) {
            int js = chk * (2 * id);
            for (j = 0; j < chk; j++) {
                cs = (bigInt)t[js + j] - (bigInt)n[id * chk + j] - (bigInt) c;
                r[id * chk + j] = (uInt) cs;
                c = (cs >> W) & 1;
            }
        }
    }
    free(t);    free(cc);    free(t0s);
}
```

List 25 pFIOS Montgomery Multiplication Algorithm C implementation, this improves pFIOS naïve in List 22

After we have an enhanced pFIOS Montgomery Multiplication Algorithm, it is essential to see the *ompP* report and compare the performance with pSHS Montgomery Multiplication Algorithm.

5.3. The ompP Tool Report

The new pFIOS is run after it is compiled using the *ompP* tool in order to see the overhead of the pFIOS Montgomery Multiplication Algorithm. The number of threads that executed the algorithm is 8 with input number length $S = 840$. The algorithm pFIOS is tested by executing the SDR Montgomery Modular Exponentiation algorithm. List 26 shows the report produced by *ompP* tool after running SDR Montgomery Modular Exponentiation with pFIOS. This report shows that the current total overhead now is 3.21% down from 46.39% in the pFIOS naive version. This overhead value is mostly synchronization overhead. However, this value is accepted and does not affect the performance very much.

```
------------------------------------------------------------------------
----     ompP Overhead Analysis Report     ----------------------------
------------------------------------------------------------------------
Total runtime (wallclock)   : 874.52 sec [8 threads]
Number of parallel regions  : 1
Parallel coverage           : 873.73 sec (99.91%)

Parallel regions sorted by wallclock time:
           Type                            Location      Wallclock (%)
R00001  PARALLEL            montgomery_mp.c (30-92)     873.73 (99.91)
                                               SUM      873.73 (99.91)
Overheads wrt. each individual parallel region:
        Total    Ovhds  (%)  =  Synch  (%)  +  Imbal   (%)+   Limpar  (%)+   Mgmt (%)
R00001  6989.85  224.46 (3.21)  221.71 (3.17)  0.97 (0.01)    0.00 (0.00)    1.78 (0.03)

Overheads wrt. whole program:
        Total    Ovhds  (%)  =  Synch  (%)  +  Imbal   (%)+   Limpar  (%)+   Mgmt (%)
R00001  6989.85  224.46 (3.21)  221.71 (3.17)  0.97 (0.01)    0.00 (0.00)    1.78 (0.03)
   SUM  6989.85  224.46 (3.21)  221.71 (3.17)  0.97 (0.01)    0.00 (0.00)    1.78 (0.03)
```

List 26 pFIOS Montgomery Modular Exponentiation ompP tool overhead report

5.4. Parallel Performance Benchmark

In this section pFIOS Montgomery Multiplication and pSHS Montgomery Multiplication algorithms will be benchmarked. Their performance will be evaluated when using them to multiply two big numbers and when they are used in SDR Montgomery Modular Exponentiation in order to calculate the modular exponentiation.

The same machine which was previously used to run the benchmark will be used here. Furthermore, the code is compiled with turning the flag *–O3* on i.e. code is optimized by the compiler. The purpose of this benchmark is to see how well the algorithm pFIOS performs compared with a well tuned algorithm like pSHS.

5.4.1. Montgomery Multiplication Performance

In order to benchmark the multiplication performance, two experiments are run. Firstly, the size of input numbers is fixed and the number of executing cores is changed. As a result of this experiment, the speedup and scalability graphs can be shown. Secondly, the number of executing cores is fixed and the input number size is changed. This experiment

will give an overview of how sensitive the parallel algorithm is to the input size or to the amount of work.

5.4.1.1. Experiment 1: Speedup & Scalability

In this experiment the size of the input numbers is fixed to the value 645,120 bit or $S = 20{,}160$. This length is acceptable since the computer machine takes a considerable amount of time to perform the multiplication and at the same time does not exceed the MPI buffers limit.

Regarding the algorithm pSHS, the previous readings will be taken, only pFIOS Montgomery Multiplication will be run. Table 6 shows the execution time of pFIOS and pSHS Montgomery Multiplication Algorithms.

cores	pFIOS	pSHS	FIOS Sequential
1	1.56	2.68	1.6
2	0.79	1.57	1.6
3	0.53	0.91	1.6
4	0.39	0.67	1.6
5	0.32	0.51	1.6
6	0.27	0.43	1.6
7	0.23	0.37	1.6
8	0.20	0.32	1.6

Table 6 pFIOS vs. pSHS Montgomery Multiplication execution time

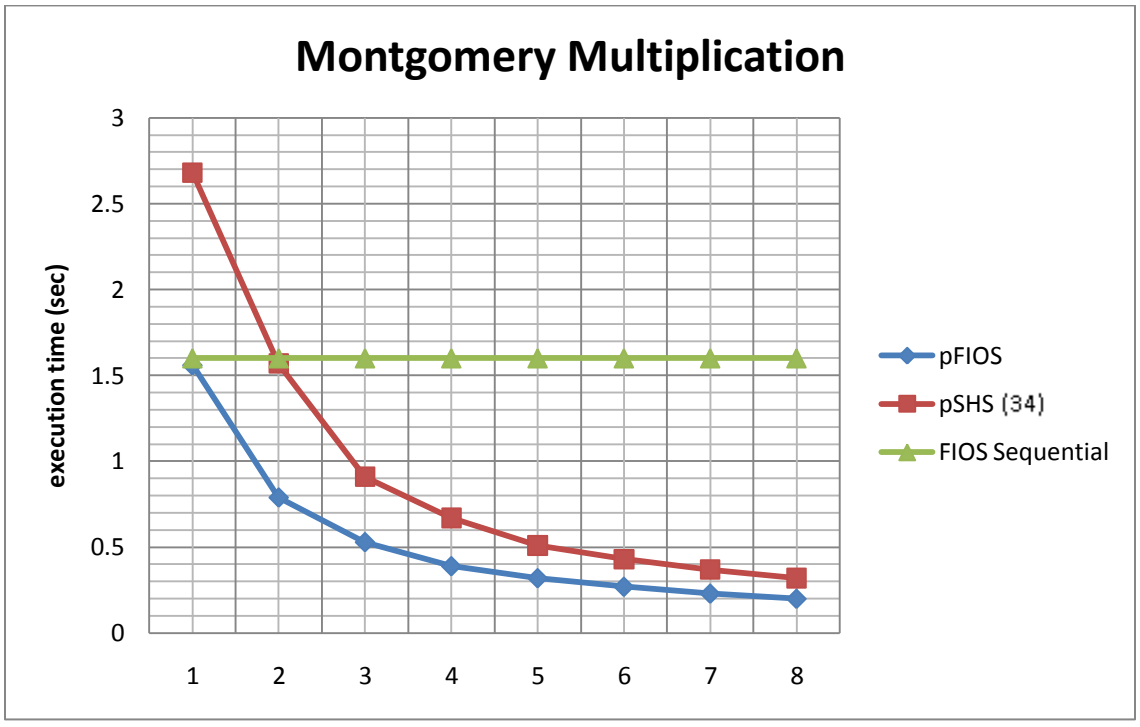

Figure 17 pFIOS vs. pSHS Montgomery Multiplication execution time, $S = 20{,}160$.

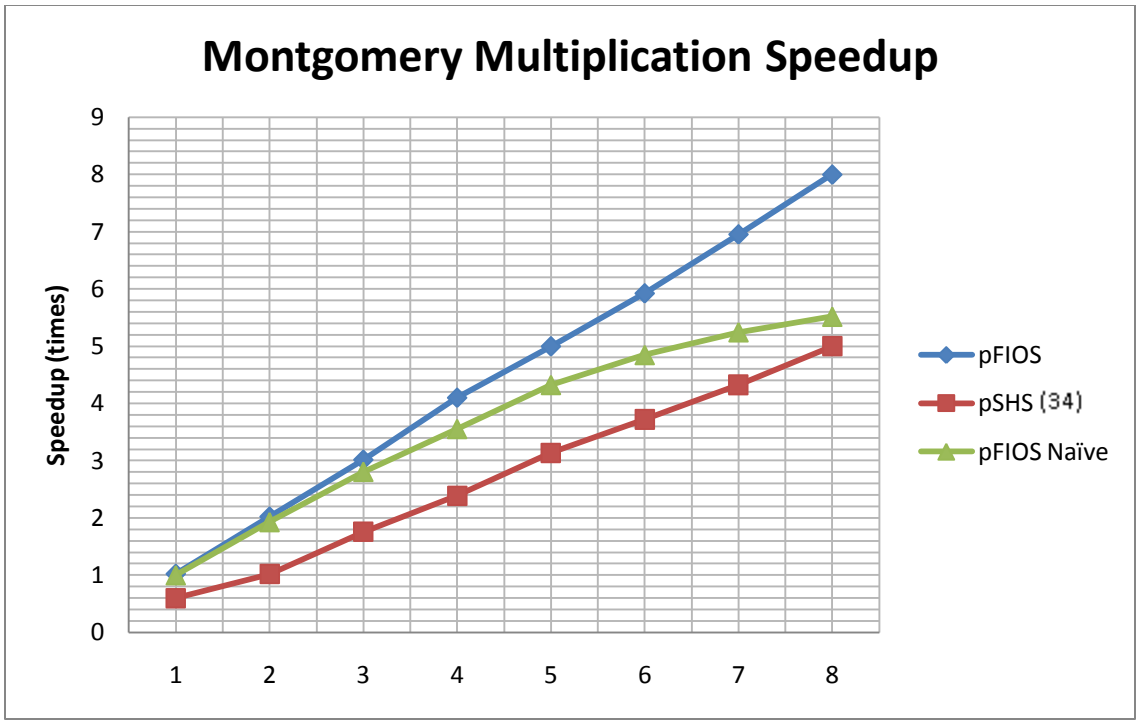

Figure 18 pFIOS vs. pSHS Montgomery Multiplication speedup

The execution time and speedup graphs show that pFIOS has a very good scalability over up to 8 cores and better performance than pSHS when both algorithms are used for multiply big numbers. Furthermore, pFIOS algorithm6 which is executed by one core or by one thread performs as fast as the sequential version. This happens because when just one thread executes pFIOS there will be just one execution of the carry addition so there will be no overhead. This makes the relative speedup of pFIOS equal to its absolute speedup.

5.4.1.2. Experiment 2: Various Number Lengths

In this experiment the number of cores that execute the algorithms is fixed to 8 cores, however, the size of the input numbers will be changed. This experiment can give a clue about how sensitive each algorithm is to various workloads. The starting number size is $S = 512$ and the ending size is $S = 20096$ the difference between two adjacent sizes is 1224. All the lengths are divisible by 8 which is the fixed number of cores or the fixed number of processes.

Number Size	pFIOS	pSHS
512	0.005347	0.00162
1736	0.002407	0.004799
2960	0.006241	0.009775
4184	0.011971	0.015482
5408	0.015052	0.025219
6632	0.021945	0.0371

7856	0.030574	0.051271
9080	0.040333	0.067002
10304	0.051644	0.087178
11528	0.064325	0.107863
12752	0.078695	0.132104
13976	0.09389	0.158339
15200	0.111033	0.180949
16424	0.129521	0.210541
17648	0.149407	0.24231
18872	0.170386	0.276801
20096	0.193759	0.31435

Table 7 pFIOS vs. pSHS Montgomery Multiplication execution time with various input number sizes.

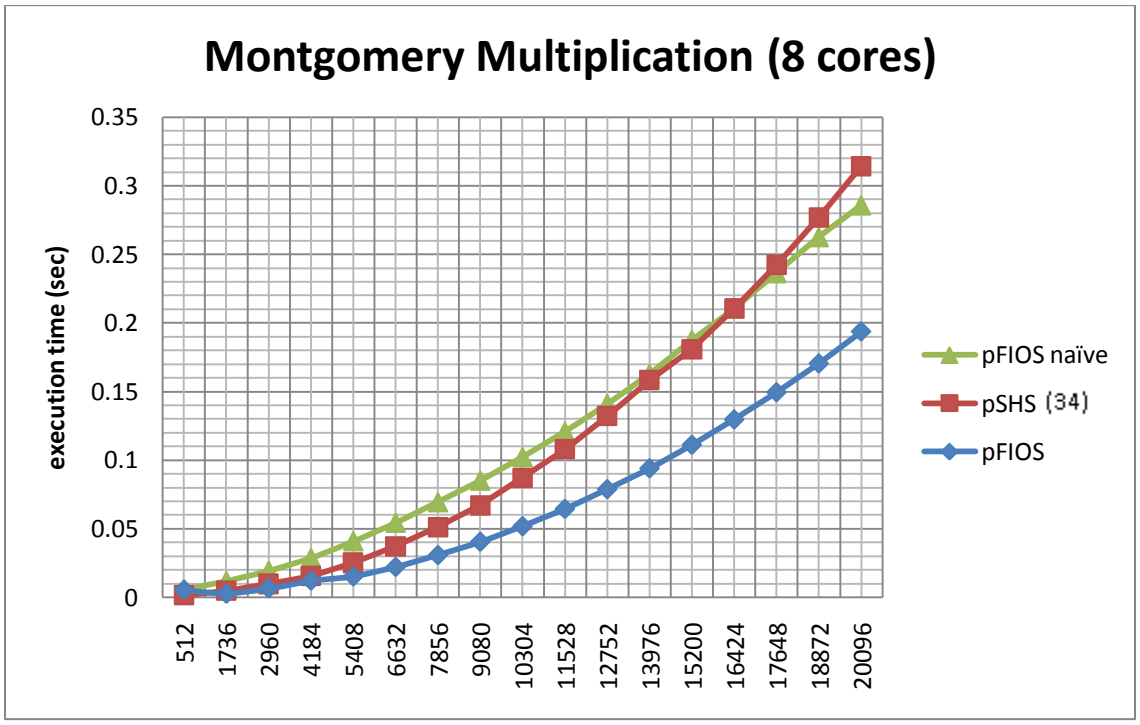

Figure 19 pFIOS vs. pSHS Montgomery Multiplication execution time with various input size and 8-core machine

Again, pFIOS shows better performance for small and large inputs. This happened after the synchronization overhead has been reduced. However, this graph does not give details about the very small input sizes. It is likely to have worse performance for the very small input where the overhead of creating the threads is bigger than the time consumed to process the input. Thus, there should be a threshold of input size, when the input size is bigger than this threshold the algorithm is executed in parallel.

5.4.2. Montgomery Modular Exponentiation Performance

In order to benchmark the pFIOS Montgomery Modular Exponentiation performance, two experiments will be run. Firstly, the input numbers' size is fixed and the SDR Montgomery Modular Exponentiation with pFIOS and with pSHS will be executed using

a variable number of cores. As a result, the speedup and scalability graphs will be displayed. Secondly, the number of executing cores is fixed and both algorithms will be executed using variable input numbers sizes - this will show how well the algorithm is performing for a smaller input size.

5.4.2.1. Experiment 1: Speedup & Scalability

In this experiment the input numbers' sizes are fixed to be $S = 840$. This length is chosen because it is divisible by the numbers 1, 2, 3, 4, 5, 6, 7, 8 so both algorithms can be executed in parallel using up to 8 cores. Table 8 shows the execution time for both pFIOS and pSHS when they are used inside SDR Montgomery Modular Exponentiation algorithm.

Cores	pFIOS			pSHS			FIOS SEQ
	Runtime	Speedup	Efficiency	Runtime	Speedup	Efficiency	Runtime
1	97.32	1.024455	1.024455	160.38	0.621649	0.621649	99.7
2	56.21	1.773706	0.886853	116.57	0.85528	0.42764	99.7
3	42.59	2.340925	0.780308	78.63	1.267964	0.422655	99.7
4	34.87	2.859191	0.714798	56.28	1.7715	0.442875	99.7
5	29.28	3.405055	0.681011	49.36	2.019854	0.403971	99.7
6	24.98	3.991193	0.665199	44.09	2.261284	0.376881	99.7
7	22.46	4.439003	0.634143	42.5	2.345882	0.335126	99.7
8	20.41	4.88486	0.610608	41.54	2.400096	0.300012	99.7

Table 8 pFIOS & pSHS SDR Montgomery Modular Exponentiation execution time, speedup and efficiency

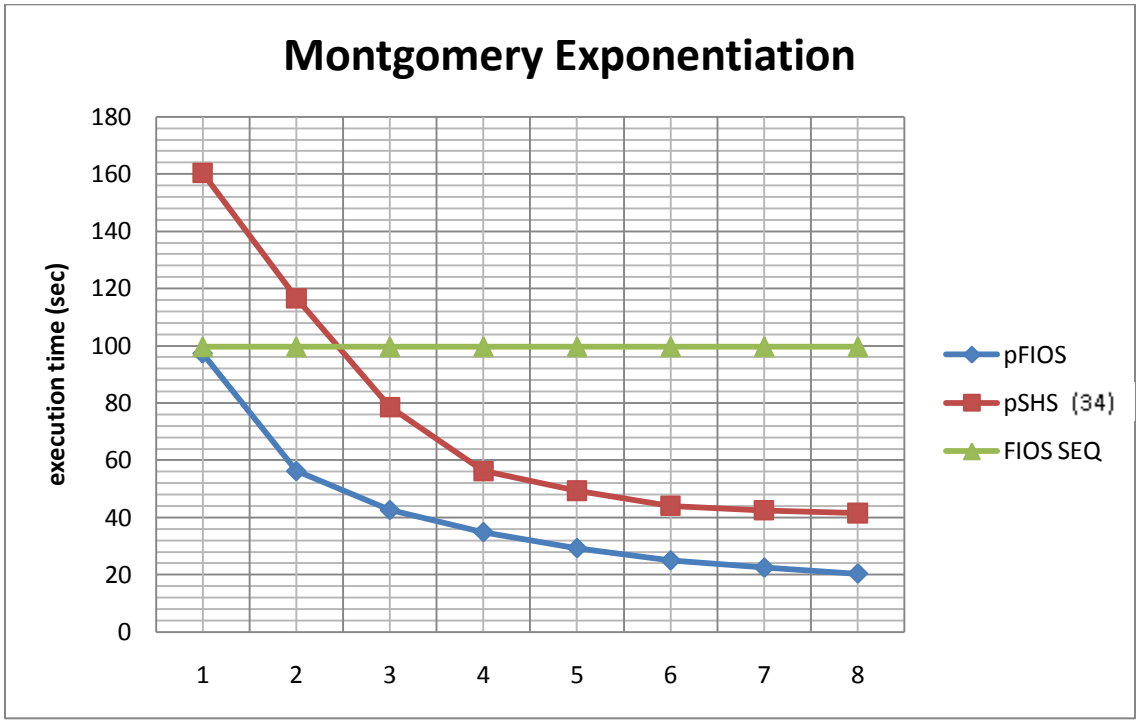

Figure 20 pFIOS vs. pSHS Monrgomery Modular Exponentiation execution time

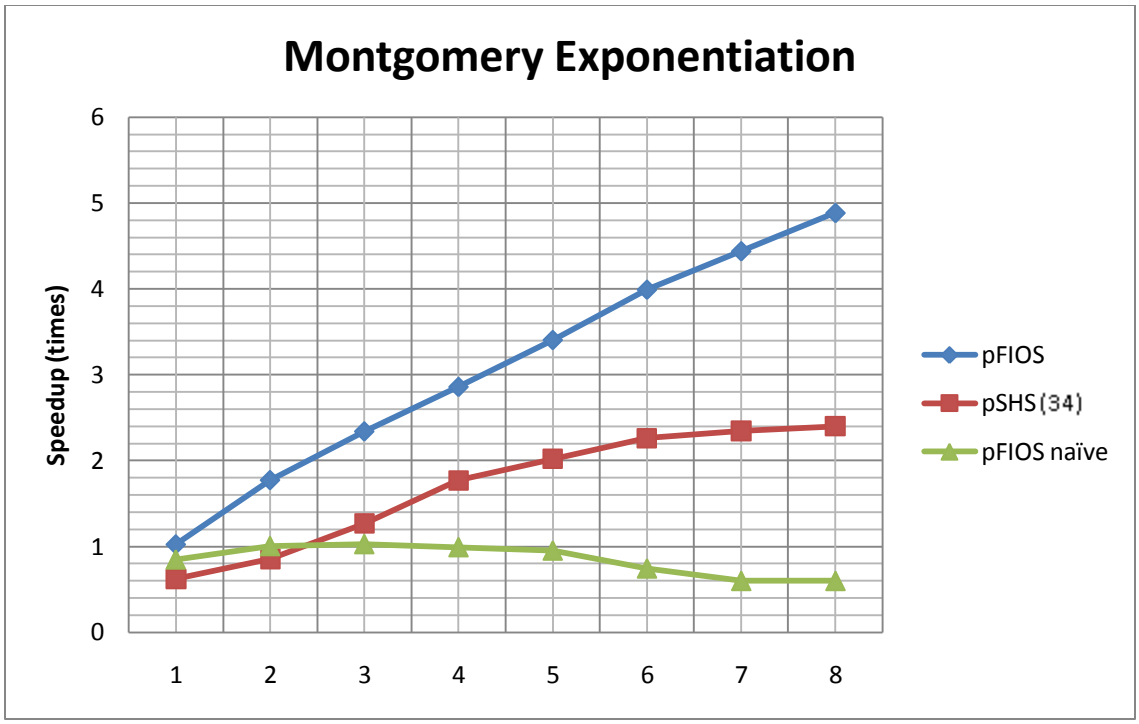

Figure 21 pFIOS vs. pSHS Montgomery Modular Exponentiation speedup

Figure 20 and Figure 21 show that pFIOS performs very well when it is used with SDR Montgomery Modular Exponentiation algorithm as well. It shows speedup of up to 5 times when it is executed by 8 cores machine, whereas the maximum speedup achieved by pSHS is about 2.5 times using the same number of cores. However, the size $S = 840$ or 26880 bit is fairly big and it is much bigger than the sizes that are used in the practical applications like RSA encryption/decryption algorithm nowadays. Thus, it is important to benchmark the performance using various input number sizes.

5.4.2.2. Efficiency & Granularity

Figure 22 shows the efficiency for both pFIOS and pSHS Montgomery Modular Exponentiation Algorithms. It can be seen that pFIOS has better efficiency than pSHS. Regarding the granularity level, increasing the number of threads will make the performance worse because when the number of threads increases, the number of carry addition steps will increase as well. Moreover, the carry addition step is very expensive as it contains two barriers and should be executed by only one thread. That is why the best performance is achieved when the number of threads is as small as possible i.e. one for each core.

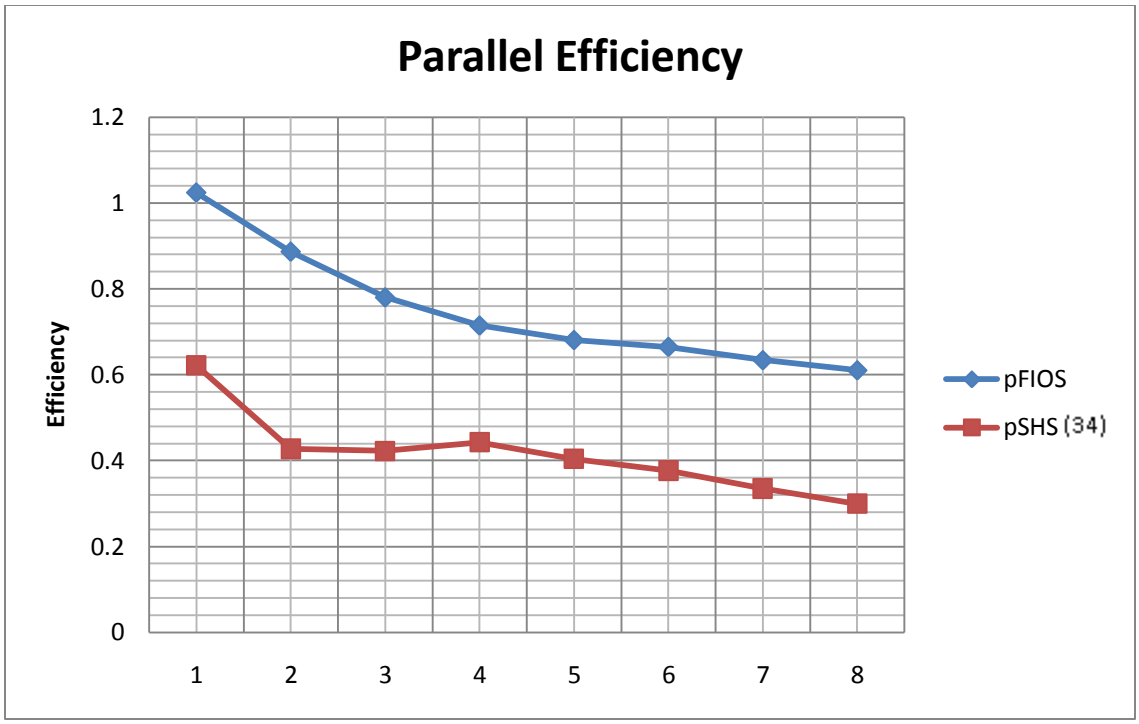

Figure 22 pFIOS vs. pSHS Montgomery Modular Exponentiation Efficiency

5.4.2.3. Experiment 2: Various Numbers Lengths

In this experiment the number of cores that are executing the algorithm is fixed to 8, and the input numbers size is changed in order to see how pFIOS performs with smaller numbers' lengths. The lengths that will be used are practical lengths, the smallest length is $S = 32$ or 1024 bit and the biggest one is $S = 256$ or 8192 bit. Table 9 shows the execution time for pFIOS, pSHS and the best tunes sequential version FIOS after executing them by an 8-core machine.

Length	pFIOS	pSHS	FIOS SEQ
32	0.119662	0.080482	0
64	0.307214	0.248997	0.04
96	0.374782	0.525218	0.16
128	0.609957	0.897865	0.37
160	0.989254	1.390228	0.71
192	1.249087	1.987458	1.23
224	1.647229	2.710583	1.95
256	1.758552	3.535123	2.88

Table 9 pFIOS, pSHS and FIOS Montgomery Modular Exponentiation execution time (8 Cores)

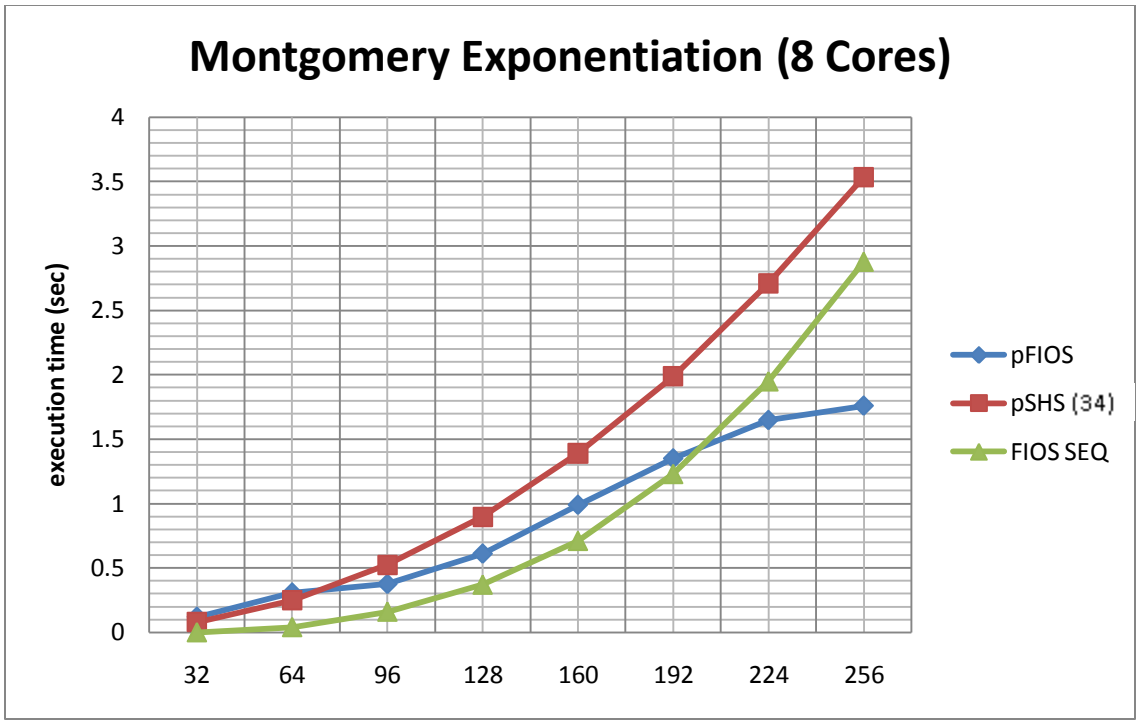

Figure 23 pFIOS, pSHS and FIOS Montgomery Modular Exponentiation execution time (8 Cores).

Figure 23 shows that when 8 cores are executing the algorithm, the performance of pFIOS version is worse than sequential version performance when the input number size is smaller than 192. This is because of the overhead of creating the threads every time pFIOS is called. When the input size is bigger than 192, the pFIOS performance starts to become better. However, pSHS still performs slower than the sequential version for such small input, because of the latency of sending and receiving messages between the processes. When the input is very small (less than or equal to 64) the performance of pSHS is better that pFIOS performance. Consequently, it is essential to adjust the number of threads that execute pFIOS depending on the length of the input.

By repeating the same experiment using only 2 cores (two threads), the same scenario happened but with different thresholds. So the sequential version still performs faster than pFIOS when the input length is less than or equal to 96. In this case it is better to run the sequential version. However, pFIOS performs faster than pSHS for all the lengths when two threads are executing the algorithm.

To conclude, OpenMP pFIOS Montgomery Multiplication Algorithm shows very good performance and scalability compared with MPI pSHS Montgomery Multiplication Algorithm in the computer systems. However, for the very small input sizes it is essential to adjust the granularity level of pFIOS i.e. the numbers of threads, because for such a small input the overhead of creating the threads and the synchronization overhead is too

time-consuming compared with the calculation time, which negatively affects the performance.

Chapter 6. Conclusions

This chapter provides a summary of all the work that has been done and the corresponding results. Then the research contributions will be listed, so all the project's deliverables are mentioned. Furthermore, there will be a section for checking and matching the research requirements and a section for future work that can be done based on this project in order to enhance the achieved results.

6.1. Summary

In this dissertation, the work for parallelizing RSA encryption and decryption is presented. This has been achieved by parallelizing Montgomery Multiplication algorithm and using it by Montgomery Modular Exponentiation which is used directly to perform encryption or decryption. Firstly, there is an introduction illustrating the contexts in which this work can be applied, along with the motivation and the technology that is used to develop the work. Then, most recent and all corresponding previous work that is related to this project is surveyed and summarised, so there is no need for any further external reading. After it, several sequential versions of RSA encryption and decryption algorithms are developed and integrated in one program. This program contains some of the most efficient implementations for Montgomery Multiplication and exponentiation algorithms. Moreover, the sequential version is benchmarked in order to discover the algorithm that has the best performance in the computer systems. As a result of this benchmark *"FIOS Modified"* version had the best sequential performance, so it is adopted as a reference for the parallel version. Next, a parallel version of *"FIOS Modified"* developed and implemented using *OpenMP* technology. However, this naive version contains too much overhead. This can be discovered by performance benchmarking and comparison with a very tuned MPI parallel version of Montgomery Multiplication, which has been published in an official paper (34). This algorithm is called *"Parallel Separated Hybrid Scanning (pSHS)"*. Moreover, the cause of the overhead is discovered by consulting *ompP* profiling tool, which is an OpenMP specialized profiling tool which shows the time consumed by each thread when they are executing OpenMP components. Some solutions have been provided and integrated in the naive parallel version in order to provide the final version *"pFIOS Montgomery Multiplication"*. Thus, to see how the final parallel version performs, it is benchmarked and compared with *pSHS* MPI implementation. The algorithm *pFIOS Montgomery Multiplication* showed very good performance and it was about 1.6 times faster than *pSHS Montgomery Multiplication* when it is executed by an 8-core machine, and about 8 times faster than the sequential version using the same machine. However, when *pFIOS* is used in *SDR Montgomery Modular Exponentiation* algorithm it was about 5 times faster than the sequential version and about twice as faster as pSHS. The algorithm is tested with a very small input as well and achieved good results. However, pFIOS was inefficient when the input numbers were less than 96 words or 3072 bit, so it is essential to check the input size and depending on this value the parallel or sequential version is

used. The pSHS algorithm, which is developed and tuned by running it by FPGA parallel hardware, was inefficient for such small input as well and worse than the pFIOS version when the experiment is run under a multi-core computer system.

6.2. Evaluation

To evaluate the resulting parallel algorithm pFIOS, three types of evaluations will be discussed here. Firstly, performance evaluation of OpenMP pFIOS using multi-core machine is discussed. Then *OpenMP* vs. *MPI* software evaluation using multi-core parallel machine is taking place. Eventually, General Purpose vs. FPGA hardware evaluation.

6.2.1. Performance Evaluation of pFIOS

The performance of pFIOS algorithm is benchmarked very well; firstly its multiplication performance is benchmarked in 5.4.1 and compared with the best tuned sequential algorithm FIOS Montgomery Multiplication and with pSHS(34) algorithm. It had a linear speedup (up to 8 times using 8 cores) when it is tested using a very big input numbers (c.f. Figure 18). However, pSHS had a linear speedup as well but up to 5 times when it is tested under the same conditions. Secondly, pFIOS exponentiation performance is benchmarked in 5.4.2 and compared with the best tuned sequential algorithm SDR-FIOS Montgomery Modular Exponentiation and with pSHS algorithm. It had a linear speedup up to 5 times using 8 cores. Whereas pSHS had speedup up to 2.5 times under the same conditions (c.f. Figure 21). The numbers size used in the exponentiation performance benchmark is relatively big and not practical as nowadays RSA keys are only 1024 bit or $S = 32$ words. So, another benchmark is run in 5.4.2.3 where the input numbers' lengths are small and practical, pFIOS exponentiation performance was better than pSHS exponentiation performance but worse than the performance of the sequential algorithm for such small input (c.f. Figure 23).

6.2.2. OpenMP vs. MPI on Multi-Core Machine (Software Evaluation)

On multi-core general purpose computer machine OpenMP and MPI technologies can be evaluated in terms of the performance and many other factors. Regarding the performance OpenMP is suitable more than MPI for such parallel hardware and can give better performance than MPI. Furthermore, OpenMP is very easy to be learnt technology and would not take from the programmer much time to parallelise an algorithm compared with MPI which requires from the programmer entire code restructure, whereas, OpenMP preserve the structure of the code. However, when there is too much data dependency and in order to obtain a good performance, it is essential to restructure the whole code to minimize the data dependency and reduce the synchronization. So, in this case OpenMP parallel programming style will not be easier than MPI parallel programming style. Furthermore, OpenMP still needs to support extra constructs like wait on a variable status which is important for the parallel algorithm in this project. Now in the parallel FIOS

algorithm there is a busy waiting loop to solve this problem. However, waiting on a status support should be provided by OpenMP because using POSIX semaphores and locks to solve this problem adverse the performance badly.

6.2.3. General Purpose vs. FPGA (Hardware Evaluation)

According to (34), parallel Separated Hybrid Scanning pSHS is developed and tuned using FPGA hardware which is *"Xilinx Virtex 5 FPGA, the authors implemented three multi-core prototypes with 2, 4, and 8 MicroBlaze cores (w=32, running at 100MHz) respectively"*. The authors provide a message passing implementation of this algorithm and benchmark the performance by taking into consideration the latency of the messages. However, in the exponentiation case, they could achieve a very good speedup up to 6.29 using 8-cores when the messages latency is 32 cycles and input numbers length is 2048. Using the same numbers length and number of cores the speedup is up to 5.02 when the latency is 1100 cycles. The same algorithm is tested using a general purpose computer system with multi-core processors the performance was worse. This can be because more latency in sending and receiving the messages. Although pFIOS has less speedup using general purpose computer system compared with the pSHS speedup using FPGA hardware, its performance is better than pSHS over general purpose computer system. The speedups of both algorithms are 5 and 2.5 times for pFIOS and pSHS respectively using 8-core machine. FPGA hardware can be very good means to thorough examine an algorithm and discover the various parameters effect. However, eventually the algorithm will be run using general purpose computer system. So developing and tuning the algorithm using such system is more practical.

6.3. Contributions

This project produces a list of deliverables:

- *Implementation and performance evaluation of 8 different sequential algorithms:* In Chapter 3 there is implementation of 4 different versions of Montgomery Multiplication: SOS, CIOS, FIOS and FIOS Modified, plus, two different versions of Montgomery Modular Exponentiation: SDR and BIN. So, 8 Montgomery Modular Exponentiation versions will be produced by using each exponentiation algorithm with the different Multiplication versions. The purpose of implementing all these versions is to choose the best tuned sequential version of RSA algorithm when it is run using multi-core computer machine. Thus, at the end of same chapter there is performance benchmark for the various versions. All the algorithms are implemented from recent papers. However, FIOS Modified is a modified version from FIOS Montgomery Multiplication. The modification is postponing the carry addition in the internal loop by saving it in a two-word variable. The new algorithm shows better performance when it is compiled with *gcc v1.4* C compiler with and without turning on the flag –O3.

- *Development and performance evaluation of efficient parallel FIOS Montgomery Multiplication Algorithm:* This is a parallel version of the best tuned sequential Montgomery Multiplication algorithm (FIOS). The new parallel algorithm has a very good scalability tested up to 8 cores and speedup of up to 8 times when it used to multiply very large numbers. However, when it is used with SDR Montgomery Modular Exponentiation, the resulting algorithm is efficient parallel version of RSA encryption/decryption. This resulting algorithm has a good scalability tested up to 8 cores and good speedup up to 5 times faster when the numbers are relatively large.
- *Implementation of parallel Separated Hybrid Scanning Algorithm (pSHS) using C+MPI:* which is parallel Montgomery Multiplication Algorithm existed in Chapter 4: Parallel Implementation and Performance. Implemented directly from (34) in order to be used as a reference for pFIOS Montgomery Multiplication Algorithm.

6.4. Requirements Matching

In this section all the project requirements are listed and matched with the already existing work in this report.

1. *Identify strong cryptographic algorithms to be parallelised:* This requirement was met by nominating RSA and AES as very strong candidates. Then RSA was chosen because sequential AES is very fast. In addition, more time was devoted to developing a better tuned RSA parallel version (c.f. 3.1 Algorithm(s) to be parallelized).
2. *Select parallel technology:* The parallel technology that has been chosen in OpenMP because of various reasons (c.f. 4.1 Why OpenMP).
3. *Sequential version of the selected algorithms:* Eight different sequential versions of RSA algorithm are provided in Chapter 3: Sequential Implementation and Performance. Moreover, there is a performance benchmark at the end of this chapter to determine the best-tuned sequential version (c.f. 3.5 Sequential Performance Benchmark).
4. *Prototype for all algorithms:* A prototype or naive version of RSA algorithm is provided in Chapter 4: Parallel Implementation and Performance. There is a performance benchmark and ompP tool analysis for this prototype in the same chapter in order to discover the problems (c.f. 4.2 Naive Parallel FIOS OpenMP, 4.4 Parallel Performance Benchmark).
5. *Implement Parallel versions of chosen algorithms:* A full final version is implemented in Chapter 5: Experimental Results and Performance Tuning (c.f. 5.2 New Parallel FIOS (pFIOS) Algorithm).
6. *Tuned parallel version of the chosen algorithms:* The new parallel version is tuned in the Chapter 5: Experimental Results and Performance Tuning. A report produced by ompP tool is provided in the same chapter as well (c.f. 5.3 The ompP Tool Report).

7. *Speedup figures for all parallel algorithms:* The speedup figures are provided in Chapter 5: Experimental Results and Performance Tuning. There are speedup figures for the plain multiplication and for the exponentiation which is the RSA encryption or decryption. Furthermore, the speedup is compared with the pSHS (34) algorithm (c.f. 5.4 Parallel Performance Benchmark).
8. *Scalability features of parallel algorithms:* A discussion about the scalability of the new parallel version took place in Chapter 5: Experimental Results and Performance Tuning. Furthermore, the new algorithm's scalability is compared with pSHS (34) scalability (c.f. 5.4.1 Montgomery Multiplication Performance, 5.4.2 Montgomery Modular Exponentiation Performance).
9. *Granularity level of each parallel algorithm:* A discussion of the best granularity level value took place in Chapter 5: Experimental Results and Performance Tuning (c.f. 5.4.2.2 Efficiency & Granularity).
10. *Performance figures against the key size:* A figure showing the execution time consumed by new parallel algorithm pFIOS is provided in chapter: Experimental Results and Performance Tuning (c.f. 5.4.2.3 Experiment 2: Various Numbers Lengths).
11. *The cryptographic algorithms should support the operation modes ECB and CTR:* The developed parallel algorithm to compute the modular exponentiation SDR with pFIOS supports all the cryptographic modes ECB and CTR because the parallelism is achieved by parallelising the Montgomery Multiplication algorithm. In other words, parallelising internal multiplication function which is called by the RSA encryption or decryption functions. That is, this way of parallelizing would not contradict with applying these encryption modes.
12. *Final report:* Done.
13. *The cryptographic algorithm should support the operation mode CBC:* This optional requirement is achieved by parallelising the internal Montgomery Multiplication algorithm. That is, this way is parallelising would not contradict with applying this encryption mode.

6.5. Future Work

For future work that can be done based on the interesting results obtained from this project, it would be useful to:

- Develop a parallel version of common multiplicand multiplication algorithm (30), and use the parallel version along with pFIOS in order to further enhance the performance of RSA encryption and decryption.
- Make more experiments for a very small input size and write a handy algorithm in order to decide the level of granularity based on the input size and the number of machine cores or processing units.

- Implement pFIOS using MPI technology and compare the MPI version with pSHS, which will give a more accurate decision about the better algorithm.
- The scalability in this research is checked for up to 8-cores only, it would be interesting to run the experiments using a computer machine with a higher number of cores or processing units.
- Regarding the pSHS MPI version, it would be interesting to benchmark its performance over *Beowulf Cluster* machines along with MPI version of pFIOS and carry on the experiments.

References

1. **Daernen, Joan and Rijrnen, Vincent.** *The Design of Rijndael, AES The Advanced Encryption Standard.* Berlin : Springer, 2002.

2. **Knapp, Anthony W.** *Basic Algebra, Along with a companion volume Advanced Algebra.* Boston, Basel, Berlin : Birkhauser, 2006.

3. **Menezes, Alfred J., van Oorschot, Paul C. and Vanstone, Scott A.** *Handbook of Applied Cryptography.* s.l. : CRC Press, 2001.

4. **Lehtinen, Rick and Russell, Deborah.** *Computer Security Basics, 2nd Edition.* s.l. : O'Reilly, 2006.

5. **Churchhouse, Robert F.** *Codes and Ciphers: Julius Caesar, the Enigma, and the Internet.* Cambridge : Cambridge University Press, 2001.

6. **Dworkin, Morris.** *Recommendation for Block Cipher Modes of Operation.* s.l. : National Institute of Standards and Technology, 2001.

7. **Wobst, Reinhard.** *Cryptology Unlocked.* Chichester : John Wiley & Sons Ltd, 2007.

8. **Ferguson, Niels, Schneier, Bruce and Kohno, Tadayoshi.** *Cryptography Engineering: Design Principles and Practical Applications.* Indianapolis : Wiley Publishing, Inc, 2010.

9. **Barker, Elaine, et al.** *Recommendation for Key Management – Part 1: General (Revised).* s.l. : National Institute of Standards and Technology (NIST) Special Publication, 2007. 800-57.

10. *Very Fast Pipelined RSA Architecture Based on Montgomery's Algorithm.* **Heri K, Iput, et al.** Selangor, Malaysia : IEEE, 2009. 978-1-4244-4913-2.

11. **Montgomery, Peter L.** *Modular Multiplication Without Trial Division.* s.l. : American Mathematical Society, 1985. 0025-5718.

12. **Knuth, Donald E.** *The Art of Computer Programming, Vol. II: Semi-numerical Algorithms, 3rd edition.* Reading, Massachusetts : Addison-Wesley, 1997. 0-201-89684-2.

13. **Dusse, Stephen R. and Kaliski Jr, Burton S.** *A Cryptographic Library for the Motorola DSP56000.* Redwood City, CA : Sprigner-Verlag, 1998.

14. **Koc, Cetin Kaya, Acar, Tolga and Kalisk, Burton S.** *Analyzing and Comparing Montgomery Multiplication Algorithms.* s.l. : IEEE Micro, 1996. 16(3):26-33.

15. **Pu, Qiong and Zhao, Xiuying.** *Montgomery Exponentiation with no Final Comparisons: Improved Results.* Zhengzhou, He'nan, China : IEEE Computer Society, 2009. 978-0-7695-3614-9/09.

16. **Grama, Ananth, et al.** *Introduction to Parallel Computing, Second Edition.* Boston : Addison Wesley, 2003.

17. **Keckler, Stephen W., Olukotun, Kunle and Hofstee, H. Peter.** *Multicore Processors and Systems, Integrated Circuits and Systems.* Austin : Springers, 2009.

18. **Quinn, Michael J.** *Parallel Programming in C with MPI and OpenMP.* New York : McGraw-Hill, 2004. 0072822562.

19. **Snir, Marc, et al.** *MPI: The Complete Reference.* London : The MIT Press, Cambridge, 1996.

20. **Gropp, William, Lusk, Ewing and Skjellum, Anthony.** *Using MPI : Portable Parallel Programming With the Message-passing Interface Scientific and Engineering Computation.* Boston : Massachusetts Institute of Technology, 1999.

21. **Chapman, Barbara, Jost, Gabriele and der Pas, Ruud van.** *Using OpenMP, Portable Shared Memory Parallel Programming.* Cambridge : The MIT Press, 2008.

22. **Hutton, Graham.** *Programming in Haskell.* New York : Cambridge University Press, 2007.

23. **Thompson, Simon.** *Haskell The Craft of Functional Programming, Second Edition.* London : Addison-Wesley, 1999.

24. **Pointon, Robert F., Trinder, Philip W. and Loidl, Hans-Wolfgang.** *The Design and Implementation of Glasgow Distributed Haskell.* London : Springer-Verlag, 2000.

25. **Breshears, Clay.** *The Art of Concurrency, A Thread Monkey's Guide to Writing Parallel Applications.* Sebastopol, CA : O'Reilly Media, Inc, 2009. 978-0-596-52153-0.

26. **Sommerville, Ian.** *Software Engineering 8th edition.* Edinburgh : Addison-Wesley Professional, 2008. 7111197704.

27. *A Scientific Rapid Prototyping Model for the Haskell Language.* **d'Auriol, Brian J., Lee, Sungyoung and Lee, Young-Koo.** s.l. : IEEE Computer Society, 2008. 978-0-7695-3407-7/08.

28. **Hasselbring, W., Jodeleit, P. and Kirsch, M.** *Implementing Parallel Algorithms based on Prototype Evaluation and Transformation.* 1998.

29. **ORHANOU, Ghizlane, EL HAJJI, Saïd and BENTALEB, Youssef.** *EPS AES-based confidentiality and integrity algorithms: Complexity study.* Rabat, Morocco : IEEE, 2010. 978-1-61284-732-0.

30. **Yen, S. M. and Laih, C. S.** *Common-multiplicand multiplication and its applications to public key cryptography.* s.l. : IEE Electronics Letters, 1993. Vol. 29 No. 17.

31. **Joye, Marc and Yen, Sung-Ming.** *Optimal Left-to-Right Binary Signed-Digit Recoding.* s.l. : IEEE Transactions on Computers, 2000. 0018-9340.

32. **WU, Chia-Long, LOU, Der-Chyuan and CHANG, Te-Jen.** An Efficient Montgomery Exponentiation Algorithm for Cryptographic Applications. *INFORMATICA.* 3, 2005, Vol. 16, 449–468.

33. **Wu, Chia-Long Wu, Lou, Der-Chyuan and Chang, Te-Jen.** *An Efficient Montgomery Exponentiation Algorithm for Public-Key Cryptosystems.* Taipei, Taiwan : IEEE, 2008. 1-4244-2415-3.

34. **Chen, Zhimin and Schaumont, Patrick.** *A Parallel Implementation of Montgomery Multiplication on Multi-core Systems: Algorithm, Analysis, and Prototype.* s.l. : IEEE, 2010. 0018-9340/10.