AN ANALYSIS OF KEY GENERATION EFFICIENCY OF RSA CRYPTOSYSTEM IN DISTRIBUTED ENVIRONMENTS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Software

by
Gökhan ÇAĞRICI

October 2005
İZMİR

We approve the thesis of **Gökhan ÇAĞRICI**

Date of Signature

……. **19 October 2005**
Assoc. Prof. Dr. Ahmet KOLTUKSUZ
Supervisor
Department of Computer Engineering
İzmir Institute of Technology

……. **19 October 2005**
Asst. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering
İzmir Institute of Technology

……. **19 October 2005**
Prof. Dr. Şaban EREN
Department of Computer Engineering
Ege University

……. **19 October 2005**
Prof. Dr. Kayhan ERCİYEŞ
Head of Department
Department of Computer Engineering
İzmir Institute of Technology

…….
Assoc. Prof. Dr. Semahat ÖZDEMİR
Head of the Graduate School

ACKNOWLEDGEMENTS

I would like to thank Assoc Prof. Ahmet KOLTUKSUZ, PhD., for his intimacy and for his great support during all the steps of this thesis. I would also like to thank Asst. Prof. Tuğkan TUĞLULAR, PhD. and Prof. Şaban EREN, PhD. for their valuable suggestions.

If I had the chance to append an acknowledgement for my life, it would give me immense pleasure to express my deep gratitude towards Deniz TOLGAY for her existence and being the reason for mine thereupon.

ABSTRACT

As the size of the communication through networks and especially through Internet grew, there became a huge need for securing these connections. The symmetric and asymmetric cryptosystems formed a good complementary approach for providing this security. While the asymmetric cryptosystems were a perfect solution for the distribution of the keys used by the communicating parties, they were very slow for the actual encryption and decryption of the data flowing between them. Therefore, the symmetric cryptosystems perfectly filled this space and were used for the encryption and decryption process once the session keys had been exchanged securely.

Parallelism is a hot research topic area in many different fields and being used to deal with problems whose solutions take a considerable amount of time. Cryptography is no exception and, computer scientists have discovered that parallelism could certainly be used for making the algorithms for asymmetric cryptosystems go faster and the experimental results have shown a good promise so far.

This thesis is based on the parallelization of a famous public-key algorithm, namely RSA.

ÖZET

Ağlar arasında ve de özellikle Internet üzerinde gerçekleşen veri iletişiminin boyutları arttıkça bu bağlantıları güvenli bir hale getirme ihtiyacı da önem kazanmıştır. Simetrik ve asimertik kriptosystemler ise bu soruna tümleşik bir çözüm sunmaktadırlar. Asimetrik kriptosistemler, anahtarların değişimi ile ilgili sorunlara mükemmel çözümler getirmesine karşın, veri iletişimi gerçekleşirken kullanılan şifreleme ve deşifreleme işlemleri için yavaş kalmaktadırlar. İşte simetrik kriptosistemler de, bağlantı için kullanılacak anahtarın değişiminin tamamlanmasını takiben bu boşluğu mükemmel bir şekilde doldururlar.

Paralelizm bir çok alan için sıcak bir araştırma konusu olup çözümleri uzun zaman dilimleri gerektiren problemler için kullanılmaktadır. Kriptografi de bunlardan biridir ve asimetrik kriptosistemlerin daha hızlı çalıştırılması için paralel algoritmaların kullanılabileceği farkına varılmış ve deneysel sonuçlar da gayet umut verici sonuçlar ortaya koymuştur.

Bu tez, sıkça kullanılan bir açık anahtar kriptosistemi olan RSA'nın paralelizasyonu ile ilgilidir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

PRIMES

1.1. Primes and Cryptography

Complexity theory classifies a problem according to the minimum time and space needed to solve the hardest instances of the problem on a Turing Machine (or some other abstract model of computation). A Turing Machine (TM) is a finite state machine with an infinite read-write tape. A *TM* is a "realistic" model of computation in that problems that are polynomial solvable on a *TM* are also polynomial solvable on real systems and vice versa.

Problems that are solvable in polynomial time are called **tractable** because they can usually be solved for reasonable size inputs. Problems that cannot be systemically solved in polynomial time are called **intractable** or simply "hard" because as the size of the input increases, their solution becomes infeasible on even the fastest computers. Turing proved that some problems are so hard they are **undecidable** in the sense that it is impossible to write an algorithm to solve them. In particular, he showed the problem of determining whether an arbitrary *TM* (or program) halts is undecidable. Many other problems have been shown to be undecidable by providing that if they could be solved, then the "halting problem" could be solved.

There are several important complexity classes and they have several relationships with others.

The class **P** consists of all problems solvable in polynomial time.

The class **NP** (nondeterministic polynomial) consists of all problems solvable in polynomial time on a nondeterministic *TM*. This means if the machine guesses the solution, it can check its correctness in polynomial time. Of course, this does not really "solve" the problem, because there is no guarantee the machine will guess the right answer; it depends on the capability of the computer to guess the answer correctly.

To systematically (deterministically) solve certain problems in **NP** seems to require exponential time. An example of such a problem is the "knapsack problem": given a set of n integers $A = \{a_1, ..., a_n\}$ and an integer S, determine whether there exists a subset of A that sums to S. The problem is clearly in **NP** because for any given subset,

it is easy to check whether it sums to *S*. Finding a subset that sums to *S* is much harder, however, as there are 2^n possible subsets; trying all of them has time complexity $T = O(2^n)$. Another example of a problem that seems to have exponential time complexity is the "satisfiability problem", which is to determine whether there exists an assignment of values to a set of *n* boolean variables $v_1, ..., v_n$ such that a given set of clauses over the variables is true.

The class **NP** includes the class **P** because any problem polynomial solvable on a deterministic *TM* is polynomial solvable on a nondeterministic one. If all **NP** problems are polynomial solvable on a deterministic *TM*, then **P = NP**. Although many problems in **NP** seem much "harder" then the problems in **P** (e.g., the knapsack problem and satisfiability) no one has yet proved **P ≠ NP**.

It has been shown that the satisfiability problem has the property that every other problem in **NP** can be reduced to it in polynomial time. This means that if the satisfiability problem is polynomial solvable, then every problem in **NP** is polynomial solvable, and if some problem in **NP** is intractable, then satisfiability must also be intractable. Since then, other problems (including the knapsack problem) have been shown to be equivalent to satisfiability in the preceding sense. This set of equivalent problems is called the **NP-complete** problems, and has the property that if any one of the problems is in **P**, then all **NP** problems are in **P** and **P = NP**. Thus, the **NP-complete** problems are the "hardest" problems in **NP**. The fastest known algorithms for systematically solving these problems have worst-case time complexities exponential in the size *n* of the problem. Finding a polynomial-time solution to one of them would be a major breakthrough in computer science.

A problem is shown to be **NP-complete** by proving it is **NP-hard** and in **NP**. A problem is **NP-hard** if it cannot be solved in polynomial time unless **P = NP**. To show a problem *A* is **NP-Hard**, it is necessary to show that some **NP-Complete** problem *B* is polynomial-time reducible to an instance of *A*, and a polynomial-time algorithm for solving *A* would also solve *B*. To show *A* is in *NP*, it is necessary to prove that a correct solution can be proved correct in polynomial time.

The class **CoNP** consists of all problems that are the complement of some problem in **NP**. Intuitively, problems in **NP** are of the form "determine whether a solution exists," whereas the complementary problems in **CoNP** are of the form "show there are no solutions." It is now known whether **NP=CoNP**, but there are problems that fall in the intersection **NP** ∩ **CoNP**. An example of such a problem is the "composite

numbers problem": given an integer n, determine whether n is composite (i.e., there exist factors p and q such that $n = pq$ or prime (i.e., there are no such factors). The problem of finding factors, however, may be harder than showing their existence.

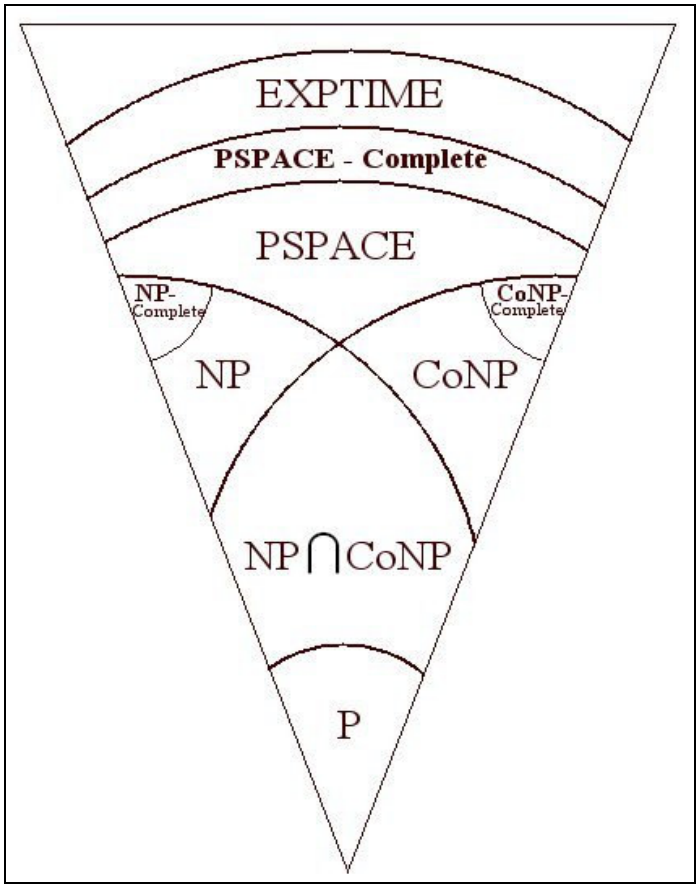

Figure 1.1. Complexity Classes.

The class **PSPACE** consists of those problems solvable in polynomial space, but not necessarily polynomial time. It includes **NP** and **CoNP**, but there are problems in **PSPACE** that are thought by some to be harder than problems in **NP** and **CoNP**. The **PSPACE-complete** problems have the property that if any one of them is in **NP**, then **PSPACE = NP**, or if any one is in **P**, then **PSPACE = P**. The class **EXPTIME** consists of those problems solvable in exponential time, and includes **PSPACE**.

1.1.1. Ciphers Based on Computationally Hard Problems

In their 1976 paper, Diffie and Hellman suggested applying computational complexity to the design of encryption algorithm. They noted that **NP-complete**

problems might make excellent candidates for ciphers because they cannot be solved in polynomial time by any known techniques. Problems that are computationally more difficult than the problems in **NP** are not suitable for encryption because the enciphering and deciphering transformations must be fast (i.e., computable in polynomial time). But this means the cryptanalyst could guess a key and check the solution in polynomial time (e.g., by enciphering known plaintext). Thus, the cryptanalytic effort to break any polynomial-time encryption algorithm must be in **NP**.

Diffie and Hellman speculated that cryptography could draw from the theory of **NP** complexity but examining ways in which **NP-complete** problems could be adapted to cryptographic use. Information could be enciphered by encoding it in an **NP-complete** problem in such a way that breaking the cipher would require solving the problem in the usual way. With the deciphering key, however, a shortcut solution would be possible.

To construct such a cipher, secret "trapdoor" information is inserted into a computationally hard problem that involves inverting a one-way function. A function f is a **one-way function** if it is easy to compute $f(x)$ for any x in the domain of f, while, for almost all y in the range of f, it is computationally infeasible to compute $f^{-1}(y)$ even if y is known. It is a **trapdoor one-way function** if it is easy to compute f^{-1} given certain additional information. This additional information is the secret deciphering key.

Public-key systems are based on this principle. The trapdoor knapsack schemes are based on the knapsack problem. The RSA scheme forming the basis of this thesis work is based on factoring composite numbers.

The strength of such a cipher depends on the computational complexity of the problem on which it is based. A computationally difficult problem does not necessarily imply a strong cryptosystem; however, Shamir gives three reasons:

1. Complexity theory usually deals with single isolated instances of a problem. A cryptanalyst often has a large collection of statistically related problems to solve (e.g., several ciphertexts generated by the same key).
2. The computational complexity of a problem is typically measured by its worst-case or average-case behavior. To be useful as a cipher, the problem must be hard to solve in almost all cases.
3. An arbitrarily difficult problem cannot necessarily be transformed into a cryptosystem, and it must be possible to insert trapdoor information into the

problem in such a way that a shortcut solution is possible with this information and only with this information (Denning 1982).[1]

1.2. Prime Number

Definition 1.1. *A **prime number** is an integer greater than 1 that can only be divided by 1 and itself with no remainder.*

Here are the first few prime numbers:

$$2,3,5,7,11,13,17,19,23,29 \ldots \tag{1.1}$$

As proceeded in the set of natural numbers $N = \{1,2,3, \ldots\}$, primes are seen less and less frequent in general. However, there is no largest prime number. As the Greek mathematician Euclid stated in the ancient times, for every prime number p, there exists a prime number p' such that p' is greater than p.

Prime numbers are important since they form the building blocks of the multiplicative structure on integers, which means that every integer can be factorized in terms of prime numbers and this assertion is known as *The Fundamental Theorem of Arithmetic*. This property is very useful in many application areas of mathematics (WEB_1 2002).

1.3. Strong Prime

The security of the RSA cryptosystem relies heavily upon the characteristics of the modulus chosen for the encryption and decryption process.

[1] Denning, E. R. D., 1982. "Cryptography and Data Security", Purdue University, (Addison-Wesley, USA), pp. 31-35.

1.3.1. Strong Prime Definition

It is desired that the modulus is not to be factorized by any special-purpose factoring algorithm within any feasible amount of time. There are four basic factoring attacks that can be applied to an RSA modulus.

The ***first*** factoring technique is related with the polynomial running time passed until the smallest prime factor of the product is found. One simple technique is to divide the product by increasing numbers starting from a small number until a factor is found, which is called as *trial division*. If a small factor p is found, then it means that the running time is $O(p)$. There are also more complex algorithms that can perform the same task faster than $O(p)$. To prevent this attack, the prime numbers must be chosen approximately the same size during the formation of the RSA modulus to make this attack infeasible for the attacker.

The ***second*** factoring technique developed by Fermat is similar to the previous technique, however, it starts searching starting from the square root of the product. The running time of the algorithm is directly proportional to the difference between the two prime factors of the RSA modulus. It means that if the two primes are chosen very close to each other, then this factoring technique will be able to find them easily using the product. Therefore, there must be a substantial difference between the primes.

The first and the second factoring technique have a contradictory nature. To avoid the RSA modulus from being factored by these two techniques, the primes must not only be approximately the same size but also be far from each other significantly. For example, if the primes with two hundred decimal digits will be used, the first prime might be $2x10^{100}$ whereas the second might be $4x10^{100}$. Since they have the same size and there is a substantial difference between them (which is $2x10^{100}$), then the two requirements have been satisfied. In fact, if the two primes are selected very large with the same size, these requirements can be fulfilled more easily.

For the ***third*** factoring technique, there are two special-purpose algorithms that are very similar indeed. The first algorithm is called *Pollard's $p - 1$ method*. Considering a product with a prime factor p, if $p-1$ has only small prime factors, then this algorithm can factor the product in a reasonable amount of time. The second algorithm, *Williams' $p + 1$ method*, differs from the first one by taking $p + 1$ into consideration instead of $p - 1$. As a consequence, to avoid suffering from this kind of

attacks, for each prime p factor of the RSA modulus, $p + 1$ and $p - 1$ must contain at least one large prime factor, which means that $p - 1$ and $p + 1$ must be a multiple of large primes r and s, respectively. Mathematically;

$$p \equiv 1 \pmod{r} \tag{1.2}$$

$$p \equiv s - 1 \pmod{s} \tag{1.3}$$

The ***last*** kind of attacks is related with the repeatedly application of the encryption algorithm to the ciphertext produced by the RSA encryption. This method will get a large percentage of the plaintext after the application of the algorithm for a small number of times if the chosen RSA modulus is poor. There are two approaches proposed to decrease the probability of success of these attacks:

1. Each prime factor p of the RSA modulus should be chosen such that $p - 1 = 2r$ where r is a large prime, which is a case of $p \equiv 1 \pmod{r}$ described in the previous paragraph.
2. Each prime factor p of the RSA modulus should be chosen such that $p - 1$ is a multiple of r where r is a large prime and $r - 1$ is also a multiple of another large prime t.

Among these two approaches, the second one will be preferred since it is much easier to satisfy its requirements than the first one.

Combining all the requirements described above, the definition of a strong prime comes to the scene:

Definition 1.2. *A **strong prime** is a prime that satisfies all the conditions shown below:*

$$p \equiv 1 \pmod{r} \tag{1.4}$$

$$p \equiv s - 1 \pmod{s} \tag{1.5}$$

$$r \equiv 1 \pmod{t} \tag{1.6}$$

where p, r, s, and t are all large primes.

1.3.2. Strong Prime Construction

Some primes possess special characteristics over the others and they are given certain names. Other than strong prime, there is also another kind of prime called as **double prime**.

Definition 1.3. *A prime r is a **double prime** if it satisfies the requirement of*

$$r \equiv 1 \pmod{t} \tag{1.7}$$

where t is another prime.

Definition 1.4. *A prime p is a **simple prime** if it does not possess any special characteristics.*

The definitions of these primes are not mutually exclusive. In fact, a strong prime is also a double prime and thus a simple prime in turn. However, the reverse is not always true. It means, a simple prime may be a double prime or not until it is looked for if it satisfies the requirements.

To generate a strong prime, first, two simple primes, s and t must be generated. After this, a double prime r should be chosen such that $r - 1$ is a multiple of t. To search for such a double prime r, an integer k can be used to calculate $kt + 1$ and iterated until $kt + 1$ is a prime.

Using Theorem 1.1, a strong prime p can be calculated.

Theorem 1.1. *If r and s are odd primes, then prime p satisfies*

$$p \equiv 1 \pmod{r} \equiv s - 1 \pmod{s} \tag{1.8}$$

if p is of the form

$$p = u(r, s) + krs \tag{1.9}$$

where

$$u(r, s) = s^{r-1} - r^{s-1} \bmod rs \tag{1.10}$$

and k is an integer.

Since r and s are both odd, rs is odd and krs is alternately odd and even while iterating k. The initial value of p (denoted as p_0) should be chosen so that p_0 is odd, which means that if $u(r, s)$ is odd then $p_0 = u(r, s)$, otherwise $p_0 = u(r, s) + rs$.

Then, $2krs$ will be added to p_0 continuously until a prime is found since the sum of an odd number by an even number is always odd. Consequently, the prime p takes the form of $p_0 + 2krs$.

The pseudocodes related to simple, strong and double prime generation are given in Algorithm 1.1, Algorithm 1.2, and Algorithm 1.3.

```
procedure StrongPrime(seed1,seed2:positive integers)
{seed1 and seed2 will be used to feed the random number generator}

s:=SimplePrime(seed1)
t:=SimplePrime(seed2)
r:=DoublePrime(t)
rs:=r*s
u:=(rs+modpower(s,r-1,rs)-modpower(r,s-1,rs)) mod rs
/*  modpower(a,b,c) calculates aˆb mod c  */
if even(u) then u:=u+rs
while not certify(u)
begin
    u:=u+2rs
end
{u is the value of the strong prime}
```

Algorithm 1.1. Generating Strong Primes.

```
procedure DoublePrime(x:positive integer)
{x is assumed to be a prime that will be used to find a double prime}

do
begin
    y:=k*x+1
    k:=k+2
end while(not certify(y))
{y is the value of the double prime}
```

Algorithm 1.2. Generating Double Primes.

```
procedure SimplePrime(x:positive integer)
{x is the starting point for searching a prime}

if even(start) then start:=start+1
while not certify(start)
begin
    start:=start+2
end
{start is the value of the simple prime}
```

Algorithm 1.3. Generating Simple Primes.

1.4. Prime Certification

Primality testing is the process of distinguishing primes from composites (products of more than one prime). In the last two decades, the importance of primality testing gained much importance after the introduction of the public-key cryptography that is now the standard form for encryption for electronic commerce. The security of this type of cryptography primarily relies on the difficulty involved in factoring a large integer into its prime factors. Certification of a prime candidate is accomplished by using mathematical methods some of which will be described soon (Greenfield 1994).[2]

[2] Greenfield, J. S., 1994. "Distributed Programming Paradigms with Cryptography Applications", *Distributed Computing Environments Group*, M/S B272, Los Alamos, New Mexico, USA, pp. 37-44.

Both the ancient Greeks and the ancient Chinese independently developed primality tests. One of the simplest and most famous primality test is *the Sieve of Eratosthenes*.

The Sieve of Eratosthenes. Eratosthenes lived in Greece circa 200 B.C. His method works as follows. A number n is chosen to test for primality. A list of all integers up to the largest integer that is less than or equal to $\sqrt{n}$ is made. Next, 2 is circled and all its multiples are crossed off. Then, 3 is circled and all its multiples are crossed off. This process is repeated by circling the next least integer that has not been crossed off yet and again, crossing off its multiples. Each of these circled numbers is tested to see if any divides n. If the list of circled numbers is exhausted and no divisor is found, then n is prime. This algorithm is based on the simple observation that, if n is composite, then n has a prime factor less than or equal to $\sqrt{n}$. Although the algorithm itself is very straightforward and easy to implement, it is not efficient. In the application of cryptography, most primality testing is concerned with large numbers, usually in excess of 100 digits and often much larger. For example, if a number with 100 digits is to be tested for primality, then at least all the primes up to 10^{50} must be found, which makes the method inefficient.

There are other approaches to the problem of finding a faster algorithm for primality testing. One way is to find a pattern among primes, and then determine if a given integer n follows that pattern. So far, no complete and easily implementable pattern has been found. Another method is to find a pattern among all composites, and then determine if a given n follows that pattern. In fact, these two methods are the same, since if an integer is not prime, then it is composite and vice versa.

The two more approaches discussed below are all based on finding patterns that are unique to composites. However, these approaches are not perfect. The patterns that they rely on fit most, but not all, composites. In other words, the numbers fitting the patterns are always composite, but there are some exceptions. These exceptions are called as *pseudoprimes*. Despite this flaw, test with almost perfect accuracy are quite useful in many applications.

There are some tests such as the Sieve of Erastosthenes that fully establish primality and do so even faster. One such test is the *Elliptic Curve Primality Procedure (ECPP)*. However, whereas composite-based tests can digest a 500-digit number in only a few minutes, ECPP takes several hours. Therefore, though ECPP is much more

efficient than the Sieve of Eratosthenes, it is not nearly as efficient as other tests. Generally, ECPP is used to verify results given by the other tests.

Two composite-based tests will be discussed below: *Fermat Primality Test* and *Strong Pseudoprimality Test (Miller-Rabin Test).*

1. **Fermat Primality Test.** In 1640, Fermat rediscovered what the ancient Chinese had known nearly 200 years before him. The result of his work is now known as *Fermat's Little Theorem.*

   Theorem 1.2. (Fermat's Little Theorem). *Let p be a prime, and a any positive integer. If gcd(p, a) = 1, then*

   $$a^{p-1} \equiv 1 \pmod{p} \tag{1.11}$$

   Fermat's primality test is another form of this theorem:

   Theorem 1.3. (Fermat's Primality Test). *An odd positive whole number n is composite if there exists a positive whole number a such that*

   $$gcd(a, n) = 1 \text{ and } a^{n-1} \neq 1 \pmod{n} \tag{1.12}$$

   Unfortunately, some composites are pseudoprimes.

   Definition 1.5. *Let n be a composite. If* $a^{n-1} \equiv 1 \pmod{n}$ *for every positive integer a with gcd(a, n) = 1, then n is called a* ***Carmichael number****.*

   Carmichael numbers are few and far between. Richard Pinch (unpublished) has recently found that there are 246,683 Carmichael numbers below 10^{16}. Below 10^{16}, there are 279,238,341,033,925 primes; so there is less than a one-in-a-billion chance that a number is a Carmichael number. Fermat's Test could be a perfect test if there were some way to easily distinguish a prime from a Carmichael number, however, nobody has succeeded that.

2. **Strong Pseudoprimality Test.** Fermat's Test can be improved with an algorithm based on the following theorem given by G. Miller.

Theorem 1.4. *Let n be an odd prime, and write n in the form* $1 + 2^s * d$ *where d is odd. Then the Miller-Rabin sequence*

$$a^d, a^{2d}, \ldots, a^{2^{s-1}*d}, a^{2^s*d} \pmod{n} \qquad (1.13)$$

ends with 1; moreover, if a^d *is not congruent to 1 (mod n), then the value directly preceding the first appearance of 1 is* $n - 1$.

This theorem also suggests a concept of pseudoprime:

Definition 1.6. *If a composite n has the characteristics described in the previous theorem for some base a, then n is called a* ***strong pseudoprime to the base a****. If n is either a prime or a pseudoprime, then n is called* ***probably prime****.*

To implement this theorem, it can be restated as follows:

Proposition 1.1. *(Strong Pseudoprimality Test). If* $n - 1 = 2^s * d$ *with d odd and s nonnegative, then n is probably prime if* $a^d \equiv 1 \pmod{n}$ *or* $a^{d^{2^r}} \equiv n - 1 \pmod{n}$ *for some nonnegative r less than s.*

This test is even stronger than Fermat's Test; in fact, it reduces the number of pseudoprimes by half. If $S(x)$ is the probability that x is a strong pseudoprime, as x goes to infinity, $S(x)$ goes to zero. So, the larger x is, the better the Strong Test is. More important, there is no strong pseudoprime equivalent to the Carmichael number.

Since Miller-Rabin Test is a probabilistic test, the accuracy is really important, which is stated as follows:

Proposition 1.2. *(Miller-Rabin Probabilistic Primality Test). While testing an odd integer n for k randomly selected bases, if n is prime, then the result*

of the test is always correct. If n is composite, then the probability that n passes all k tests is at most $\left(\frac{1}{4}\right)^k$ (McGregor-Dorsey 1991).[3]

Miller-Rabin test is the prime certification algorithm for this thesis since it is fast when compared to nonprobabilistic prime certification methods (which is the main reason for the existence of probabilistic prime certification methods), highly reliable since the probability of the test to erroneously certify a composite number as a prime number is equal to $\left(\frac{1}{4}\right)^k$ as stated in Proposition 1.2 where *k* is number of tests applied, and does not have a flaw caused by Carmichael numbers. The pseudocode for the Miller-Rabin Test is given in Algorithm 1.4.

```
procedure miller rabin test(p,factor:integer)
{p is the prime candidate and factor is to identify how many times to repeat
the test to decrease the probability of an undetected composite selected as a
prime}

composite:=false
write p in the form of p = 1 + 2^s * t where t is an odd number
while factor > 0
begin
    select a random base b that is between 1 and p-1
    if b^t (mod n) := 1 or b^t (mod n) := n-1 then continue with the next base to test
    else
    begin
        for i:=1 to s
        begin
            t:=t*2
            if b^t (mod n) := 1 composite := true
            else if b^t (mod n) := n -1 then continue with the next base to test
        end
        composite:=true
    end
factor := factor - 1
end
{composite is the return value}
```

Algorithm 1.4. Miller-Rabin Test

[3] McGregor-Dorsey Z. S., 1991. "Methods of Primality Testing", pp. 133-141.

To decrease the time passed while certifying a prime candidate, trial division will be performed with small prime numbers to test if the candidate can be divided by them or not before using the other slower test, that is, Miller- Rabin Test. The complete *certify* algorithm is now complete and shown in Algorithm 1.5.

```
procedure certify(p:integer)
{p is the prime candidate}

passed:=false
if not SmallFactor(p) then
passed:=miller rabin test(p,40)
{passed is the return value}
```

Algorithm 1.5. Certification Procedure.

The trial division algorithm shown in Algorithm 1.6 uses the first *n* small primes to test if the candidate is divisible by any of them or not.

```
procedure SmallFactor(p:integer)
{p is the prime candidate}
found:=false
while not found and j < NumPrimes do begin
    found:=(p mod primes[j]=0)
    j:=j+1
end
{found is the return value}
```

Algorithm 1.6. Testing for Small Factors.

CHAPTER 2

RSA

2.1. Cryptosystems and Cryptanalysis

Cryptography deals with the transformation of ordinary text (plaintext) into coded form (ciphertext) by encryption, and transformation of ciphertext into plaintext by decryption. Normally these transformations are parameterized by one or more keys. The reason for encrypting text is security for transmissions over insecure channels (Simmons 1992).[4]

Until the advent of computers, one of the main constraints on cryptography had been the ability of the code clerk to perform the necessary transformations, often on a battlefield with little equipment. An additional constraint has been the difficulty in switching over quickly from one cryptographic method to another one, since this entails retraining a large number of people. However, the danger of a code clerk being captured by the enemy has made it essential to be able to change the cryptographic method instantly if need be. These conflicting requirements have given rise to the model shown below.

The messages to be encrypted, known as the **plaintext**, are transformed by a function that is parameterized by a key. The output of the encryption process, known as the **ciphertext**, is then transmitted, often by messenger or radio. The so-called enemy, or **intruder**, hears and accurately copies down the complete ciphertext. However, unlike the intended recipient, he does not know what the decryption key is and so cannot decrypt the ciphertext easily. Sometimes the intruder can not only listen to the communication channel (passive intruder) but can also record messages and play them back later, inject his own messages, or modify legitimate messages before they get to the receiver (active intruder). The art of breaking ciphers, called **cryptanalysis**, and the art devising them (cryptography) is collectively known as **cryptology** (Tanenbaum 2003).

[4] Simmons, G. J., 1992. "Contemporary Cryptology", (IEEE Press, New York, USA), p. 180.

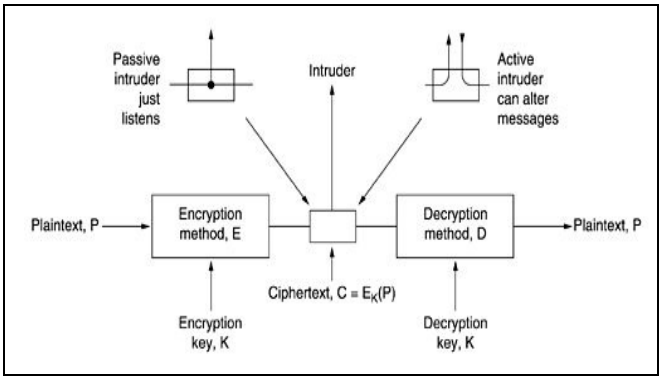

Figure 2.1. The Encryption Model (Symmetric Case).
(Source: Tanenbaum 2003)

Three of the most important services provided by cryptosystems are **secrecy**, **authenticity**, and **integrity**. *Secrecy* refers to denial of access to information by unauthorized individuals. *Authenticity* refers to validating the source of a message; that is, that it was transmitted by a properly identified sender and is not a replay of a previously transmitted message. *Integrity* refers to assurance that a message was not modified accidentally or deliberately in transit, by replacement, insertion, or deletion. A fourth service that may be provided is nonrepudiation of origin, that is, protection against a sender of a message later denying transmission.

Classic cryptography deals mainly with the secrecy aspect. It also treats keys as secret. In the past 15 years, two new trends have emerged:

1. Authenticity as a consideration that rivals and sometimes exceeds secrecy in importance.
2. The notion that some key material need not be secret.

The first trend has arisen in connection with applications such as electronic mail systems and electronic funds transfers. In such settings, an electronic equivalent of the handwritten signature may be desirable. Also, intruders into a system often gain entry by masquerading as legitimate users; cryptography presents an alternative to password systems for access control.

The second trend addresses difficulties that have traditionally accompanied the management of secret keys. This may entail the use of couriers or other costly,

inefficient, and not really secure methods. In contrast, if keys are public, the task of key management may be substantially simplified.

An ideal system might solve all of these problems concurrently, that is, using public keys, providing secrecy, and providing authenticity. Unfortunately, no single technique proposed to date has met all three criteria. Conventional systems such as the *Data Encryption Standard (DES)* or the *Advanced Encryption Standard (AES)* require management of secret keys; systems using public key components may provide authenticity but are inefficient for bulk encryption of data due to low bandwidths.

Fortunately, conventional and public key systems are not mutually exclusive; in fact, they can complement each other. Public key systems can be used for signatures and also for distribution of keys used in systems such as DES or AES. Thus, it is possible to construct hybrids of conventional and public key systems that can meet all of the above goals: secrecy, authenticity, and ease of key management.

In the following, E and D represent encryption and decryption transformations, respectively. It is always required that $D(E(M)) = M$. It may also be the case that $E(D(M)) = M$; in this event E or D can be employed for encryption. Normally, D is assumed to be secret, but E may be public. In addition, it may be assumed that E and D are relatively easy to compute when they are known.

2.1.1. Requirements for Secrecy

Secrecy requires that a cryptanalyst (i.e., an intruder) should not be able to determine the plaintext corresponding to given ciphertext, and should not be able reconstruct D by examining ciphertext for known plaintext. This translates into two requirements for a cryptosystem to provide secrecy:

1. A cryptanalyst should not be able to determine M from $E(M)$; that is, the cryptosystem should be immune to ciphertext-only attacks.
2. A cryptanalyst should not be able to determine D given $E(M_i)$ for any sequence of plaintexts $M_1,M_2,...$; that is, the cryptosystem should be immune to known-plaintext attacks. This should remain true even the cryptanalyst can choose M_i (chosen-plaintext attack), including the case in which the

cryptanalyst can inspect $E(M_i),...,E(M_j)$ before specifying M_{j+1} (adaptive chosen-plaintext attack).

To illustrate the difference between these two categories, two examples will be used. First, it is supposed that $E(M) = M^3 \ mod N$, $N = p * q$, where p and q are large secret primes. Then, it is infeasible for a cryptanalyst to determine D, even after inspecting numerous pairs of the form M, $E(M)$. However, an eavesdropper who intercepts $E(M) = 8$ can conclude $M = 2$. Thus, a ciphertext-only attack may be feasible in an instance where known- or chosen-plaintext attack is not useful.

On the other hand, it is supposed that $E(M)=5M \ mod \ N$ where N is secret. Then, interception of $E(M)$ would not reveal M or N; this would remain true even if several ciphertexts were intercepted. However, an intruder who learns that $E(12) = 3$ and $E(16) = 4$ could conclude $N = 19$. Thus, a known- or chosen-plaintext attack may succeed where a ciphertext-only attack fails.

Deficiencies in (1), that is, vulnerability to ciphertext-only attack, can frequently be corrected by slight modifications of the encoding scheme, as in the $M^3 \ mod \ N$ encoding above. Adaptive chosen-plaintext is often regarded as the strongest attack.

Secrecy ensures that decryption of messages is infeasible. However, the enciphering transformation E is not covered by the above requirements; it could even be public. Thus, secrecy leaves open the possibility that an intruder could masquerade as a legitimate user, or could compromise the integrity of a message by altering it. That is, secrecy does not imply authenticity/integrity.

2.1.2. Requirements for Authenticity and Integrity

Authenticity requires that an intruder should not be able to masquerade as a legitimate user of a system. Integrity requires that an intruder should not be able to substitute false ciphertext for legitimate ciphertext. Two minimal requirements should be met for a cryptosystem to provide these services:

1. It should be possible for the recipient of a message to ascertain its origin.
2. It should be possible for the recipient of a message to verify that it has not been modified in transit.

These requirements are independent of secrecy. For example, a message *M* could be encoded by using *D* instead of *E*. Then, assuming *D* is secret, the recipient of *C = D(M)* is assured that this message was not generated by an intruder. However, *E* might be public; *C* could then be decoded by anyone intercepting it.

A related service which may be provided is nonrepudiation; that is, a third requirement may be added if desired:

3. A sender should not be able to deny later that he sent a message.

It may also be added that:

4. It should be possible for the recipient of a message to detect whether it is a replay of a previous transmission (Simmons 1992).[5]

2.2. Public-Key Cryptography

2.2.1. The Need for Public-Key Cryptography

Before going on to the concept of public-key systems, the reason for their existence must be argued. Before these systems, there was symmetric ones and their weaknesses resulted in a need for the asymmetric ones. One of the problems is the key distribution problem. Since each party must have the key to encrypt or decrypt a message, it soon became a problem to distribute this key to everyone as the number of the parties increased. Furthermore, if the conversations are distinct, then each one must have a unique key.

If sharing keys is not an option, a *trusted third party (TTP)* can be used. In this scheme, the trusted third party shares a key with each individual. Actually, the keys are *key-encrypting keys*, or *KEK*s. When one individual wants to communicate with the other side, he requests a session key from the TTP. To fulfill the request, the TTP generates a new session key, encrypts it with the KEK they are sharing and sends it to the requester. The TTP also sends the session key to the other side by encrypting it with the other KEK they are sharing this time. Now, each side can exchange messages securely without using the TTP. However, the main disadvantage with this solution is

[5] Simmons, G. J., 1992. "Contemporary Cryptology", (IEEE Press, New York, USA), pp. 180-183.

that the TTP can read all the messages and some people may not want to trust such a TTP (Burnett and Paine 2001).[6]

2.2.2. Historical Background

In the mid-1970s, Stanford University graduate student Whitfield Diffie and professor Martin Hellman investigated cryptography in general and the key distribution problem in particular. The two came up with a scheme whereby two people could create a shared secret key by exchanging public information. They could communicate over public lines, sending information back and forth in a form readable by eavesdroppers, at the same time generating a secret values not made public. The two correspondents would then be able to use that secret value as a symmetric session key. The name given to this is **Diffie-Hellman**, or **DH**.

DH solves a problem –sharing a key– but it is not encryption. That does not make it unusable; in fact, DH is in use to this day. But it was not the "ultimate" algoritm, one that could be used for encryption. Diffie and Hellman published their result in 1976. That paper outlined the idea of public-key cryptography (one key encrypts, the other descrpts), pointed out that the authors did not yet have such an algorithm, and described what they had so far.

Ron Rivest, a professor at MIT, liked Diffie and Hellman's idea of public-key cryptography and decided to create the ultimate algorithm. He recruited two colleagues –Adi Shamir and Len Adleman– to work on the problem. In 1977, the trio developed an algorithm that could indeed encrypt data. They published the algorithm in 1978, and it became known as **RSA**, the initials of its inventors.

In 1985, working independently, two men Neal Koblitz of the University of Washington and Victor Miller of IBM's Watson Research Center– proposed that an obscure branch of math called elliptic curves could be used to perform public-key cryptography. By the late 1990s, this class of algorithms had begun to gain momentum.

Since 1977 (and 1985), many researchers have invented many public-key algorithms.

To this day, however, the most commonly used public-key algorithm for solving the key distribution problem is RSA. In second place is DH, followed by elliptic curves.

[6] Burnett, S. and Paine, S. 2001. "RSA Security's Official Guide to Cryptography", (Osborne, Berkeley, California, USA), p. 85.

2.2.3. Public-Key Cryptosystems

As explained before, a cryptosystem is an algorithm that can convert input data into something unrecognizable (encryption), and convert the unrecognizable data back to its original form (decryption). The idea behind any public-key cryptosystem is that a user can release a public key which can be used only for encryption; in other words, that key cannot be used to decipher encrypted text without performing a lot of work. Private and public keys are associated by a function. In the RSA cryptosystem, the private and public keys are linked by the factorization of prime numbers.

Public-key cryptosystems are based on trap-door one-way functions. A *one-way function* is a function for which forward computation is easy, while the backward computation is very hard. In other words, a function $f : X \rightarrow Y$ (where X and Y are arbitrary sets) is one-way if it is easy to compute $f(x)$ for every $x \in X$, while it is hard for most $y \in Y$ to find any $x \in X$ such that $f(x) = y$.

A function is said to be trap-door one-way if it is possible to easily perform both forward and backward computation; however, the algorithm for backward computation cannot be easily determined without certain secret information, even with knowledge of the complete algorithm for forward computation. In a public-key cryptosystem, an individual makes the algorithm public to allow anyone encipher a message by using it. All the messages directed to a specific receiver are encrypted using a key that is known by every individual, but the key which will be used to do the backward computation for getting the original message is known only by the receiver.

It has not been proven that any one-way functions exist, but some known functions are candidates. For example, integer multiplication is very easy, while the inverse function, integer factoring, is currently considered to be very hard. The existence of one-way functions is related to the **P = NP** question (Greenfield 1994).[7]

2.2.3.1. Secrecy and Authenticity

To support secrecy, the transformations of a public key system must satisfy $D(E(M)) = M$. For example, if A wishes to send a secure message M to B. Then, A must

[7] Greenfield, J. S., 1994. "Distributed Programming Paradigms with Cryptography Applications", *Distributed Computing Environments Group*, M/S B272, Los Alamos, New Mexico, USA, p. 11.

have access to E_B, the public transformation of B. Now, A encrypts M via $C = E_B(M)$ and sends C to B. On receipt, B employs his private transformation DB for decryption; that is, B computes $D_B(C) = D_B(E_B(M)) = M$. If A's transmission is overheard, the intruder cannot decrypt C since D_B is private. Thus, secrecy is ensured. However, presumably anyone can access E_B; B has no way knowing the identity of the sender. Also, A's transmission could have been altered. Thus, authenticity and integrity are not assured.

To support authentication and integrity, the transformations in a public key system must satisfy $E(D(M)) = M$. If A wished to send an authenticated message M to B, that is, B will be able to verify that the message was sent by A and was not altered; in this case, A could use his private transformation D_A to compute $C = D_A(M)$ and send C to B. That is, A employs D_A as a de facto encryption function. Now B can use A's public transformation E_A to find $E_A(C) = E_A(D_A(M)) = M$; that is, E_A acts as a de facto decryption function. Assuming M is valid plaintext, B knows that C was in fact sent by A, and was not altered in transit. This follows from the one-way nature of E_A: if a cryptanalyst, starting with a message M, could find C' such that $E_A(C') = M$, this would imply that he can invert E_A, a contradiction.

If M, or any portion of M, is a random string, then it may be difficult for B to ascertain that C is authentic procedure and unaltered merely by examining $E_A(C)$. Actually, however, a slightly more complex procedure is generally employed: an auxiliary public function H is used to produce a much smaller message $S = D_A(H(M))$ that A sends to B along with M. On receipt, B can compute $H(M)$ directly. The latter may be checked against $E_A(S)$ to ensure authenticity and integrity, since once again the ability of a cryptanalyst to find S' for a given M would violate the one-way nature of E_A. Actually, H must also be one-way.

Sending C or S above ensures authenticity, but secrecy is nonexistent. In the second scheme, M was sent in the clear along with S; in the first scheme, an intruder who intercepts $C = D_A(M)$ presumably has access to E_A and hence can compute $M = E_A(C)$. Thus in either case, M is accessible to an eavesdropper.

It may be necessary to use a combination of systems to provide secrecy, authenticity, and integrity. However, in some cases, it is possible to employ the same public-key system for these services simultaneously. It has been noted that for authenticity and integrity purposes, D is regarded as an encryptor; for secrecy, E is the encryptor. If the same public key system is to be used in both cases, then $D(E(M)) = M$

and $E(D(M)) = M$ must both hold; that is, D and E are inverse functions. A requirement is that the plaintext space (i.e., the domain of E) must be the same as the ciphertext space (i.e., the domain of D).

In addition to E and D being inverses for each user, for each pair of users A and B, the functions E_A, D_A, E_B, and D_B all have a common domain. Then, both secrecy and authenticity can be accomplished with a single transmission: A sends $C = E_B(D_A(M))$ to B; then B computes $E_A(D_B(C)) = E_A(D_A(M)) = M$. An intruder cannot decrypt C since he lacks D_B; hence secrecy is assured. If the intruder sends C' instead of C, C' cannot produce a valid M since D_A is needed to produce a valid C. This assures authenticity.

2.2.3.2. Applicability and Limitations

The range of applicability of public key systems is limited in practice by the relatively low bandwidths associated with public key ciphers, compared to their conventional counterparts. It has not been proven that time or space complexity must necessarily be greater for public key systems than for conventional systems. However, the public key systems that have withstood cryptanalytic attacks are all characterized by relatively low efficiency. For example, some are based on modular exponentiation, a relatively slow operation. Others are characterized by high data expansion (ciphertext much larger than plaintext). This inefficiency, under the conservative assumption that it is in fact inherent, seems to preclude the use of public key systems as replacements for conventional systems utilizing fast encryption techniques such as permutations and substitutions. That is, using public key systems for bulk data encryption is not feasible, at least for the present.

On the other hand, there are two major application areas for public key cryptosystems:

1. Distribution of secret keys
2. Digital signatures

The first involves using public key systems for secure and authenticated Exchange of data-encrypting keys (DEKs) between two parties as explained before. DEKs are secret shared keys connected with a conventional system used for bulk data

encryption. This permits users to establish common keys for use with a system such as DES. Classically, users have had to rely on a mechanism such as a courier service or a central authority for assistance in the key exchange process. Use of a public key system permits users to establish a common key that does not need to be generated by or revealed to any third party, providing both enhanced security and greater convenience and robustness.

Digital signatures are a second major application. They provide authentication, nonrepudiation, and integrity checks. As noted above, in some settings, authentication is a major consideration; in some cases, it is desirable even when secrecy is not a consideration. Moreover, nonrepudiation is another property desirable for digital signatures. Public key cryptosystems provide this property as well.

No bulk encryption is needed when public key cryptography is used to distribute keys, since the latter are generally short. Also, digital signatures are generally applied only to outputs of hash functions. In both cases, the data to be encrypted or decrypted are restricted in size. Thus, the bandwidth limitation of public key is not a major restriction for either application (Simmons 1992).[8]

2.3. RSA Cryptosystem

Three natural numbers $< e, d, M >$ define a particular instance of an RSA cryptosystem, where e is the public enciphering component, d is the secret deciphering exponent, and M is the modulus. A plaintext message m (assumed to be in the form of an integer that is greater than 1 and less than M) is enciphered into a cryptogram $c = E(m)$, where $E(m) = m^e \ mod \ M$. The cryptogram may subsequently be deciphered to retrieve the plaintext message $m = D(c)$, where $D(c) = c^d \ mod \ M$. $< e, M >$ is called the *public-key* and $< d, M >$ is called the *secret-key*. The modulus M is chosen to be the product of two large primes, p and q. This fact allows a cryptographer to publish an RSA public-key without revealing the secret-key.

[8] Simmons, G. J., 1992. "Contemporary Cryptology", (IEEE Press, New York, USA), pp. 185-187.

2.4. The Mathematics of RSA

The RSA cryptosystem exploits a property of modular arithmetic described by *Euler's Generalization of Fermat's Theorem*. Fermat made the following conjecture, now known as Fermat's Theorem:

Theorem 2.1. ***Fermat's Theorem.*** *If p is prime, and gcd(a, p) = 1 then* $a^{p-1} \equiv 1 \pmod{p}$.

The term *gcd(a, p)* is used to mean the *greatest common divisor* of *a* and *p*. When the greatest common divisor of any two numbers is equal to one, then they are said to be relatively prime to each other. Euler generalized Fermat's theorem into a form that applied to both prime and non-prime moduli:

Theorem 2.2. *Euler's Generalization. If gcd(a, n) = 1 then* $a^{\Phi(n)} \equiv 1 \pmod{n}$.

Here, $\Phi(n)$ is known as *Euler's totient function*, and is defined to be the number of non-negative integers less than *n* that are relatively prime to *n*. For a prime *p*, every integer from 1 to $p - 1$ is relatively prime to *p*. Accordingly, $\Phi(p)$ is defined to be $p - 1$.

For an RSA modulus $M = pq$, $\Phi(M)$ is easily computed from *p* and *q*. There are M −1 positive integers less than *M*. Of those integers, $p-1$ are divisible by *q*:

$$q, 2q, 3q, ..., (p - 1)q \tag{2.1}$$

and *q*-1 are divisible by *p*:

$$p, 2p, 3p, ..., (q - 1)p \tag{2.2}$$

Therefore,

$$\begin{aligned} \Phi(M) &= (M - 1) - (p - 1) - (q - 1) \\ &= pq - p - q + 1 \\ &= (p - 1)(q - 1) \end{aligned} \tag{2.3}$$

Using RSA cryptosystem, a message m is enciphered into a cryptogram

$$c = m^e \bmod M \tag{2.4}$$

and similarly, the cryptogram is deciphered to reveal the message

$$m = c^d \bmod M \tag{2.5}$$

Substituting 2.4 into 2.5 gives

$$\begin{aligned} m &= c^d \bmod M \\ &= (m^e \bmod M)^d \bmod M \\ &= m^{ed} \bmod M \end{aligned} \tag{2.6}$$

This specifies a relationship between e,d, and M necessary for the cryptosystem to function. The keys must be chosen so that

$$m = m^{ed} \bmod M \tag{2.7}$$

If the two prime factors of M have approximately 100 digits each, the number of possible messages that can be enciphered is approximately 10^{200}. Since $M = pq$, $(p-1)+(q-1) \approx 10^{100}$ integers between 1 and M are not relatively prime to M. So the probability that any given message shares a factor with M is approximately $10^{100}/10^{200} = 10^{-100}$. Consequently, even if an RSA modulus were used to encipher a million different messages, the probability that any of the messages would share a factor with the modulus would be negligibly small.

Therefore, it may be assumed that an arbitrary message m is relatively prime to modulus M. By Euler's Generalization,

$$m^{\Phi(M)} \equiv 1 \pmod M \tag{2.8}$$

It can be shown that 2.7 is satisfied when the keys are chosen so that

$$ed \equiv 1 \pmod{\Phi(M)} \tag{2.9}$$

which is equivalent to

$$ed = u\Phi(M) + 1 \tag{2.10}$$

for some positive integer u.
When 2.10 is true:

$$\begin{aligned} M^{ed} \bmod M &= m^{u\Phi(M)+1} \bmod M \\ &= (m^{u\Phi(M)} \bmod M)(m^1 \bmod M) \\ &= (m^{u\Phi(M)} \bmod M)m \\ &= ((m^{\Phi(M)} \bmod M)^u \bmod M)m \\ &= (1^u \bmod M)m \\ &= m \end{aligned} \tag{2.11}$$

and requirement 2.7 is satisfied.

A set of RSA keys is created by first constructing the modulus as the product of two primes ($M = pq$). $\Phi(M)$ is easily computed from the prime factors p and q, as described.

Next, an enciphering exponent e, relatively prime to $\Phi(M)$, is chosen. Finally, the deciphering component d is computed so that $ed \equiv 1 \pmod{\Phi(M)}$. Such a d is called the *inverse* of e modulo $\Phi(M)$. When $\Phi(M)$ and e are known, this inverse can be found by means of a fast extended gcd algorithm.

There is no known method for computing $\Phi(M)$ without knowing the factors of M. Furthermore, there is no known algorithm to compute an inverse modulo $\Phi(M)$ without $\Phi(M)$.

RSA security requires, therefore, that modulus M be chosen so that it cannot be factored. If a cryptanalyst could factor the modulus, $\Phi(M)$ and d could be easily computed, and the system would be broken.

To prevent a factoring attack, the modulus is chosen to be the product of two large primes. By making the prime factors sufficiently large, general-purpose factoring

algorithms are useless. In addition, the prime factors are chosen to have special properties that make the modulus safe from special-purpose factoring attacks.

The choice of the size of the prime factors represents a balance between security from factoring attacks and speed of enciphering and deciphering. The execution times for general-purpose factoring algorithms are exponential in the number of digits of the modulus. On the other hand, the running time for RSA enciphering and deciphering, using a standard modular exponentiation algorithm, is cubic.

As a result, the time required to factor a modulus grows much faster with modulus size than does the time required to perform enciphering and deciphering operations (Greenfield 1994).[9]

2.5. Modes of Operation

While applying many encryption techniques to long messages, there is a need to break the plaintext message into short blocks for enciphering. For example, an RSA enciphered message is encoded modulo *M*. Such an RSA enciphering can encode no more than *M* distinct messages. An arbitrary plaintext message to be enciphered, however, may contain considerably more information than could be encoded without loss, by a single application of the RSA enciphering algorithm.

To prevent any information loss, a long plaintext message is broken into short blocks, each of a size that may be encoded using the desired enciphering technique, without loss. The plaintext message becomes a sequence of plaintext message blocks and the enciphered message becomes a sequence of ciphertext blocks.

A variety of modes of operation exist for breaking a large message into short blocks and each will be described below.

2.5.1. Electronic Code Book (ECB)

The simplest mode of operation is known as *Electronic Code Book (ECB)*. ECB consists of splitting the plaintext message into a sequence of short blocks, $m_0, m_1, m_2, \ldots$,

[9] Greenfield, J. S., 1994. "Distributed Programming Paradigms with Cryptography Applications", *Distributed Computing Environments Group*, M/S B272, Los Alamos, New Mexico, USA, pp. 12-15.

and subsequently enciphering each of the short blocks, independently, to form a sequence of ciphertext blocks, $c_0 = E(m_0)$, $c_1 = E(m_1)$, $c_2 = E(m_2)$, ...

While this mode of operation is particularly simple to understand and implement, it suffers from a number of security problems when used for certain applications. In particular, ECB enciphering has the undesirable property that duplicate portions of plaintext may be encoded into duplicate ciphertext portions, with a relatively high probability. This can make cryptanalysis of the ciphertext considerably easier. In addition, since the ECB enciphering of a block is independent of its position within the plaintext message, a cryptanalyst may be able to cut and paste segments of ciphertext, in order to forge a message (Greenfield 1994).[10]

To see how this monoalphabetic substitution cipher property can be used to partially defeat the cipher, (triple) DES will be used because it is easier to depict 64-bit blocks than 128-bit blocks, but AES has exactly the same problem (and other cryptosystems). The straightforward way to use DES to encrypt a long piece of plaintext is to break it up into consecutive 8-byte (64-bit) blocks and encrypt them one after another with the same key. The last piece of plaintext is padded out to 64 bits, if need be.

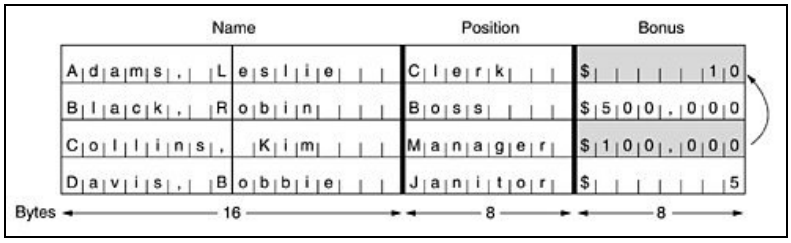

Figure 2.2. The Plaintext of a File Encrypted as 16 DES Blocks.

In Figure 2.2, there is the start of a computer file listing the annual bonuses a company has decided to award to its employees. This file consists of consecutive 32-byte records, one per employee, in the format shown: 16 bytes for the name, 8 bytes for the position, and 8 bytes for the bonus. Each of the sixteen 8-byte blocks (numbered from 0 to 15) is encrypted by (triple) DES.

Leslie can get access to the file after it is encrypted but before it is sent to the bank. All Leslie has to do is make a copy of the 12th ciphertext block (which contains

[10] Greenfield, J. S., 1994. "Distributed Programming Paradigms with Cryptography Applications", *Distributed Computing Environments Group*, M/S B272, Los Alamos, New Mexico, USA, p. 84.

Kim's bonus and Leslie can guess that Kim has a higher bonus) and use it to replace the 4th ciphertext block (which contains Leslie's bonus) (Tanenbaum 2003).

2.5.2. Cipher Block Chaining (CBC)

A second mode of operation is known as *Cipher Block Chaining*. This is the mode of operation normally used for RSA enciphering. Like ECB, cipher block chaining starts with splitting the plaintext message into a sequence of short blocks, m_0, m_1, m_2, ... The first plaintext block is then encoded to form ciphertext block $c_0 = E(m_0)$. The remaining blocks are encoded using a chaining technique, so that the second plaintext block is encoded to form ciphertext block $c_1 = E(c_0 \oplus m_1)$, where *oplus* represents *bitwise exclusive-or*. The third plaintext block is enciphered to form ciphertext block $c_2 = E(c_{i-1} \oplus m_i)$, for $i > 0$.

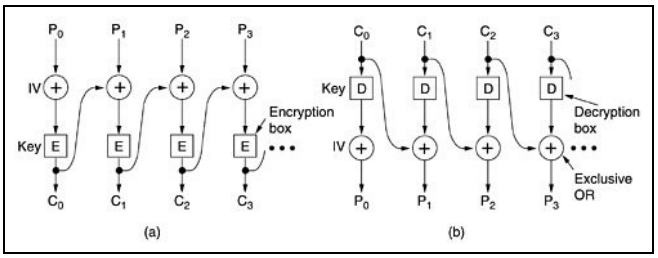

Figure 2.3. Cipher Block Chaining. (a) Encryption. (b) Decryption.

Cipher text block chaining possesses a number of properties that make it a desirable mode of operation. Duplicate portions of plaintext, encoded using this technique, have a very low probability of being encoded into duplicate portions of ciphertext. In addition, due to the chaining of ciphertext blocks, each ciphertext block depends upon all previous blocks of the message. This makes it virtually impossible for a cryptanalyst to simply cut and paste blocks of ciphertext in order to form a forged message.

At the same time, arbitrary blocks of ciphertext can be deciphered individually, without the need to decipher the entire ciphertext message. For $i > 0$, $m_i = c_{i-1} \oplus D(c_i)$, so only one deciphering operation is required.

Similarly, cipher block chaining allows the deciphering algorithm to recover from temporary transmission or enciphering errors. For example, if ciphertext block c_i is

improperly transmitted, only message blocks $m_i = c_{i-1} \oplus D(c_i)$ and $m_{i+1} = c_i \oplus D(c_{i+1})$, depend on the value of c_i. Therefore, only those two blocks are affected by the transmission error.

As a result, cipher block chaining maintains much of its simplicity of ECB, while providing more secure encoding (Greenfield 1994).[11] There are also other modes of operation that may be used. The sequential algorithm of RSA encryption and decryption with cipher block chaining is shown below:

```
procedure encrypt(e,N:positive integers)
{e is the encryption key and N is the modulus}

ulast:=0
do
begin
    ReadBlock(mi)
    chain(ui,ulast,mi)
    ci:=modpower(ui,e,M)
    WriteBlock(ci)
end while not eof ci is the encrypted form of the message mi

procedure chain(ui,ulast,mi:positive integers)
{ui is the value of the message block mi after chained, ulast is the value of the last
chained block and mi is the message to be chained}

ui:=xor(ulast,mi)
ulast:=ui
{ui is the chained value of mi}
```

Algoritm 2.1. RSA Encryption and Cipher Block Chaining.

[11] Greenfield, J. S., 1994. "Distributed Programming Paradigms with Cryptography Applications", *Distributed Computing Environments Group*, M/S B272, Los Alamos, New Mexico, USA, pp. 84-85.

```
procedure decrypt(d,N:positive integers)
{d is the decryption key and N is the modulus}

ulast:=0
do
begin
    ui:=modpower(ci,d,M)
    unchain(mi,ulast,ui)
    WriteBlock(ci)
end while not eof
{mi is the decrypted form of the cipherblock ci}

procedure unchain(mi,ulast,ui:positive integers)
{ui is the value of the message block mi after chained, ulast is the value of the last
chained block and mi is the message that is unchained}

mi:=xor(ulast,ui)
ulast:=ui
{mi is the unchained value of ui}
```

Algoritm 2.2. RSA Decryption and Cipher Block Unchaining.

2.5.3. Cipher Feedback Mode (CFM)

Cipher block chaining requires an entire 64-bit block to arrive before decryption can begin. For use with interactive terminals, where people can type lines shorter than eight characters and then stop, waiting for a response, this mode is unsuitable. For byte-by-byte encryption, cipher feedback mode, using (triple) DES is used, as shown in Figure 2.4. For AES, the idea is exactly the same, only a 128-bit shift register is used. In this figure, the state of the encryption machine is shown after bytes 0 through 9 have been encrypted and sent. When plaintext byte 10 arrives, as illustrated in the left part, the DES algorithm operates on the 64-bit shift register to generate a 64-bit ciphertext. The leftmost byte of that ciphertext is extracted and XORed with P10. That byte is transmitted on the transmission line. In addition, the shift register is shifted left 8 bits, causing C2 to fall off the left end, and C10 is inserted in the position just vacated at the right end by C9. It should be noted that the contents of the shift register depend on the entire previous history of the plaintext, so a pattern that repeats multiple times in the plaintext will be encrypted differently each time in the ciphertext. As with cipher block chaining, an initialization vector is needed to start the process.

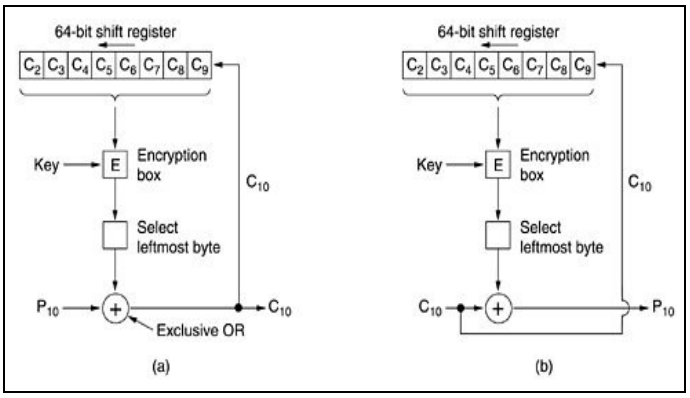

Figure 2.4. Cipher Feedback Mode. (a) Encryption. (b) Decryption.

Decryption with cipher feedback mode just does the same thing as encryption. In particular, the content of the shift register is encrypted, not decrypted, so the selected byte that is XORed with C10 to get P10 is the same one that was XORed with P10 to generate C10 in the first place. As long as the two shift registers remain identical, decryption works correctly. It is illustrated in the right part.

A problem with cipher feedback mode is that if one bit of the ciphertext is accidentally inverted during transmission, the 8 bytes that are decrypted while the bad byte is in the shift register will be corrupted. Once the bad byte is pushed out of the shift register, correct plaintext will once again be generated. Thus, the effects of a single inverted bit are relatively localized and do not ruin the rest of the message, but they do ruin as many bits as the shift register is wide.

2.5.4. Stream Cipher Mode (SCM)

Nevertheless, applications exist in which having a 1-bit transmission error mess up 64 bits of plaintext is too large an effect. For these applications, a fourth option, stream cipher mode, exists. It works by encrypting an initialization vector, using a key to get an output block. The output block is then encrypted, using the key to get a second output block. This block is then encrypted to get a third block, and so on. The (arbitrarily large) sequence of output blocks, called the keystream, is treated like a one-time pad and XORed with the plaintext to get the ciphertext, as shown in the left part of the related figure. It should be noted that the IV is used only on the first step. After that, the output is encrypted. Also it should be noted that the keystream is independent of the

data, so it can be computed in advance, if need be, and is completely insensitive to transmission errors. Decryption is shown in the right part.

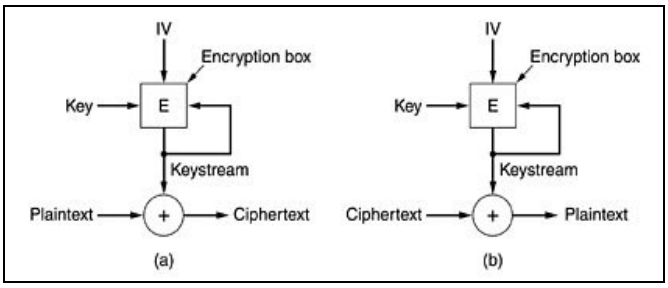

Figure 2.5. Stream Cipher Mode. (a) Encryption. (b) Decryption.

Decryption occurs by generating the same keystream at the receiving side. Since the keystream depends only on the IV and the key, it is not affected by transmission errors in the ciphertext. Thus, a 1-bit error in the transmitted ciphertext generates only a 1-bit error in the decrypted plaintext.

It is essential never to use the same (key, IV) pair twice with a stream cipher because doing so will generate the same keystream each time. Using the same keystream twice exposes the ciphertext to a keystream reuse attack. For example, the plaintext block, P_0, is encrypted with the keystream to get P_0 XOR K_0. Later, a second plaintext block, Q_0, is encrypted with the same keystream to get Q_0 XOR K_0. An intruder who captures both of these ciphertext blocks can simply XOR them together to get P_0 XOR Q_0, which eliminates the key. The intruder now has the XOR of the two plaintext blocks. If one of them is known or can be guessed, the other can also be found. In any event, the XOR of two plaintext streams can be attacked by using statistical properties of the message. For example, for English text, the most common character in the stream will probably be the XOR of two spaces, followed by the XOR of space and the letter “e”, etc. In short, equipped with the XOR of two plaintexts, the cryptanalyst has an excellent chance of deducing both of them.

2.5.5. Counter Mode

One problem that all the modes except electronic code book mode have is that random access to encrypted data is impossible. For example, suppose a file is transmitted over a network and then stored on disk in encrypted form. This might be a

reasonable way to operate if the receiving computer is a notebook computer that might be stolen. Storing all critical files in encrypted form greatly reduces the damage due to secret information leaking out in the event that the computer falls into the wrong hands.

However, disk files are often accessed in nonsequential order, especially files in databases. With a file encrypted using cipher block chaining, accessing a random block requires first decrypting all the blocks ahead of it, an expensive proposition. For this reason, yet another mode has been invented, counter mode, as illustrated in the figure. Here, the plaintext is not encrypted directly. Instead, the initialization vector plus a constant is encrypted, and the resulting ciphertext XORed with the plaintext. By stepping the initialization vector by 1 for each new block, it is easy to decrypt a block anywhere in the file without first having to decrypt all of its predecessors.

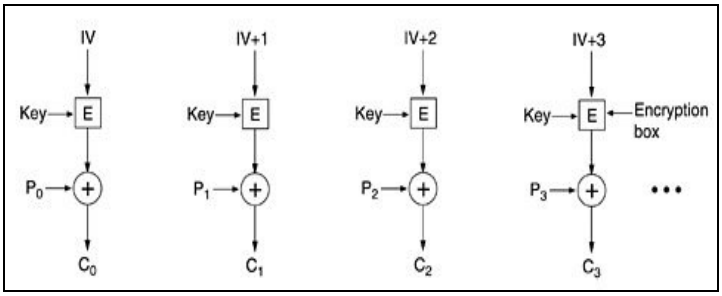

Figure 2.6. Encryption Using Counter Mode.

Although counter mode is useful, it has a weakness that is worth pointing out. For example, the same key, K, may be used again in the future (with a different plaintext but the same IV) and an attacker may acquire all the ciphertext from both runs. The keystreams are the same in both cases, exposing the cipher to a keystream reuse attack of the same kind we saw with stream ciphers. All the cryptanalyst has to do is to XOR the two ciphertexts together to eliminate all the cryptographic protection and just get the XOR of the plaintexts. This weakness does not mean counter mode is a bad idea. It just means that both keys and initialization vectors should be chosen independently and at random. Even if the same key is accidentally used twice, if the IV is different each time, the plaintext is safe (Tanenbaum 2003).

CHAPTER 3

PARALLEL COMPUTING

3.1. Introduction

Parallel computing is accomplished by splitting up a large computational problem into smaller tasks that may be performed simultaneously by multiple processors. For example, the addition of two very long vectors of numbers, A and B, can be performed by two processors if one of them adds the first half of vector A to the first half of vector B, while the second adds the second half of vector A to the second half of vector B. While this theoretically halves the time needed to solve the problem, the resulting vector, C, is now split across two different processor memories.

Because of this split, communication must occur to get the entire solution in one place. The distribution of the initial data, A and B, and the collection of the result, C, adds overhead to the computational problem and reduces the actual speedup of the entire task to less than double.

Symmetric multi-processor (SMP) and other shared memory computers can be used to reduce the amount and cost (in terms of time) of this communication; however, these systems typically have only a small number of processors or are very expensive, custom-built supercomputers. On the other hand, distributed memory platforms -- including Beowulf clusters -- are relatively inexpensive and can be scaled to hundreds or thousands of processors.

Most clusters built today are hybrids; they consist of many nodes (i.e., individual computers), each having two or more processors.

3.1.1. Decomposition and Granularity

Computational problems may be parallelized in a variety of ways. Parallelization may be accomplished by decomposing the data (as in the vector addition example above), by decomposing functionality so that one processor performs one type of

operation while other processors simultaneously perform different operations, or by decomposing both the data and the functionality.

This decomposition may be established a priori or, more often, is performed dynamically once the program is running. Good parallel code most often automatically decomposes the problem at hand and allows the processors to communicate with each other when necessary -- this is called "*message passing*" -- while performing individual tasks.

Not all computational problems are amenable to parallel computing. If an algorithm cannot be restructured so that sub-tasks can be performed simultaneously or if the model components are highly interdependent, attempts to parallelize these codes may result in increased time-to-solution. Such "fine grained" problems do not scale well as more processors are applied to the computation. Performance of the finest-grained problems is limited by the speed of the fastest single CPU that is available.

Other computational problems, such as image processing where which each pixel may be manipulated independently, are "*coarse grained*". Image processing is a good example of this type of problem. These problems are generally easier to parallelize and tend to benefit the most from parallel processing, particularly in distributed memory environments. The coarsest grained problems are often referred to as "*embarrassingly parallel*".

Fortunately, most complex scientific problems may be decomposed by performing separate tasks independently and simultaneously on multiple processors or by splitting up the space and/or time coordinates of the system being modeled. These problems tend to fall somewhere between coarse and fine granularity and usually require a moderate amount of interprocessor communication to coordinate activities and share data.

For example, values for cells on a map may depend on neighboring cell values. If the map is decomposed into two pieces, each being processed on a separate CPU, the processors must exchange cell values along the adjacent edges of the map.

Problem decomposition is very important for successful parallel processing. A balance must be struck between computation and communication so that what was a computational problem on a single processor does not become a communications problem in a parallel environment. Writing good parallel code is actually more of an art than a science; practitioners must be able to think about algorithms in novel ways (WEB_3 2002).

3.1.2. Historical Background

Until relatively recently, the standard architecture model for most digital computers was that introduced by von Neumann. The von Neumann model assumes that the program and data are held in the store of the machine and that a central processing unit (CPU) fetches instructions from the store and executes them. The instructions result in either data in the store being manipulated or information being input or output. Machines based on this model are entirely sequential in operation – one instruction is executed in each time interval.

The first general-purpose electronic digital computer, called ENIAC, was developed at the University of Pennsylvania, USA, in 1946. Programming of the machine was achieved by special wiring, and rewiring was necessary if the operations were to be modified. The first stored-program computers, and thus the first realizations of the von Neumann model, were the EDSAC and the EDVAC prototype machines, leading to commercial systems such as the UNIVAC 1. These early machines used bit-serial access to memory and consequently performed bit-serial arithmetic. By 1953 the IBM 701, the first commercial computer with bit-parallel access to memory and bit-parallel arithmetic, was available; thus from the earliest days, parallelism was introduced into digital computers.

The four decades that have elapsed since the development of the first commercial systems have witness to dramatic improvements in computer technology. Between 1950 and 1980, every five-year period has seen an approximately tenfold increase in the achievable performance of computers. These improvements were partly the result of advances in semiconductor technology, but also arose from an evolution of the original von Neumann model in which the requirement of purely serial processing was abandoned.

The development of parallel computers has been at two levels, both of which have attempted to enhance the performance of a particular class of machine. Early efforts centered on the development of high performance *supercomputers* to solve very large scientific problems such as are encountered in accurate weather forecasting. The most popular supercomputer of the mid-1970s was the Cray-1, a *vector processor* capable of 130 Mflops (*megaflops*, or millions of *flops*, floating-point operations per second). The eight-processor Cray Y-MP and the four-processor Cray-2 of the late

1980s are capable of Gflops (*gigaflops*, or thousands of Mflops) performance. These machines exhibit limited parallelism (as far as the user is concerned) with just a small number of powerful processing units operating in parallel. The race is on for the development of Tflops (*teraflops*, or millions of Mflops) machines.

The second strand of parallel computer development has centered around the desire to produce machines (*minisupercomputers*, or *superminicomputers*) which are capable of performances approaching that of a supercomputer, but at considerably reduced cost. These machines achieve their performance either by using vector processing capabilities, or by including a number of parallel processing units, or both. Thus, parallelism in computing is not only present in supercomputers, but also increasingly common less powerful machines.

It is instructive to relate these developments in computer architecture to the computer solution of numerical problems. Throughout the decades 1950-1980, the architectural developments were, for the most part, invisible to the user. Programs which worked optimally on one machine were likely also to work well on some other system. The only hardware feature of which the programmer might need to be aware was the available memory. Virtual memory aided portability; programs simply had to minimize the amount of data transfer between main and secondary storage. The development of units possessing vector processing capabilities meant that the way a user chose to express his algorithm could have a significant affect on the performance of an implementation. By making the vector structure of the algorithm visible to the compiler, significant improvements could be obtained over a corresponding code for which this structure was not apparent. The impact of the computer architecture on the user is greater in the case of a multiprocessor system and is compounded if its memory is distributed over the individual processors. Such architectural considerations crucially affect the choice of algorithm and the way that an algorithm is expressed.

The demand for increased performance may result from a desire

- to decrease the execution time of certain programs so that results can be obtained in a reasonable time (possibly even in real time), or
- to run programs for the same amount of time but obtain higher accuracy or more accurate modeling of the underlying physical problem by increasing the problem size in some way, or
- to solve problems previously considered too large for existing architectures.

In the history of computing, it has been largely possible to achieve these requirements by improved technology aimed at a uniprocessor. However, there are physical limits to just how far such a machine can be improved and the use of parallelism is seen to be one of the most attractive avenues to explore in the quest for increased performance (Freeman and Phillips 1992).[12]

3.2. Parallel Programming Platforms

The traditional logical view of a sequential computer consists of a memory connected to a processor via a datapath. All three components – processor, memory, and datapath – present bottlenecks to the overall processing rate of a computer system. A number of architectural innovations over the years have addressed these bottlenecks. One of the most important innovations is multiplicity – in processing units, datapaths, and memory units. This multiplicity is either entirely hidden from the programmer, as in the case of implicit parallelism, or exposed to the programmer in different forms.

3.2.1. Implicit Parallelism: Trends in Microprocessor Architectures

While microprocessor technology has delivered significant improvements in clock speeds over the past decade, it has also exposed a variety of other performance bottlenecks. To alleviate these bottlenecks, microprocessor designers have explored alternate routes to cost-effective performance gains.

Clock speeds of microprocessors have posted impressive gains - two to three orders of magnitude over the past 20 years. However, these increments in clock speed are severely diluted by the limitations of memory technology. At the same time, higher levels of device integration have also resulted in a very large transistor count, raising the obvious issue of how best to utilize them. Consequently, techniques that enable execution of multiple instructions in a single clock cycle have become popular. Indeed, this trend is evident in the current generation of microprocessors such as the Itanium, Sparc Ultra, MIPS, and Power4.

[12] Freeman, T.L. and Phillips, C. 1992. “Parallel Numerical Algorithms”, (Prentice Hall, UK), pp. 5-6.

3.2.1.1. Pipelining and Superscalar Execution

Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, store, among others), pipelining enables faster execution. The assembly-line analogy works well for understanding pipelines. It should be noted that the speed of a single pipeline is ultimately limited by the largest atomic task in the pipeline. Furthermore, in typical instruction traces, every fifth to sixth instruction is a branch instruction. Long instruction pipelines therefore need effective techniques for predicting branch destinations so that pipelines can be speculatively filled. The penalty of a misprediction increases as the pipelines become deeper since a larger number of instructions need to be flushed. These factors place limitations on the depth of a processor pipeline and the resulting performance gains.

An obvious way to improve instruction execution rate beyond this level is to use multiple pipelines. During each clock cycle, multiple instructions are piped into the processor in parallel. These instructions are executed on multiple functional units.

3.2.1.2. Very Long Instruction Word Processors

The parallelism extracted by superscalar processors is often limited by the instruction look-ahead. The hardware logic for dynamic dependency analysis is typically in the range of 5-10% of the total logic on conventional microprocessors (about 5% on the four-way superscalar Sun UltraSPARC). This complexity grows roughly quadratically with the number of issues and can become a bottleneck. An alternate concept for exploiting instruction-level parallelism used in very long instruction word (VLIW) processors relies on the compiler to resolve dependencies and resource availability at compile time. Instructions that can be executed concurrently are packed into groups and parceled off to the processor as a single long instruction word (thus the name) to be executed on multiple functional units at the same time.

The VLIW concept, first used in Multiflow Trace (circa 1984) and subsequently as a variant in the Intel IA64 architecture, has both advantages and disadvantages compared to superscalar processors. Since scheduling is done in software, the decoding and instruction issue mechanisms are simpler in VLIW processors. The compiler has a

larger context from which to select instructions and can use a variety of transformations to optimize parallelism when compared to a hardware issue unit. Additional parallel instructions are typically made available to the compiler to control parallel execution. However, compilers do not have the dynamic program state (e.g., the branch history buffer) available to make scheduling decisions. This reduces the accuracy of branch and memory prediction, but allows the use of more sophisticated static prediction schemes. Other runtime situations such as stalls on data fetch because of cache misses are extremely difficult to predict accurately. This limits the scope and performance of static compiler-based scheduling.

Finally, the performance of VLIW processors is very sensitive to the compilers' ability to detect data and resource dependencies and read and write hazards, and to schedule instructions for maximum parallelism. Loop unrolling, branch prediction and speculative execution all play important roles in the performance of VLIW processors. While superscalar and VLIW processors have been successful in exploiting implicit parallelism, they are generally limited to smaller scales of concurrency in the range of four- to eight-way parallelism.

3.2.2. Dichotomy of Parallel Computing Platforms

The increasing gap in peak and sustainable performance of current microprocessors, the impact of memory system performance, and the distributed nature of many problems present overarching motivations for parallelism. Now, the elements of parallel computing platforms that are critical for performance oriented and portable parallel programming will be produced at a high level. The logical organization refers to a programmer's view of the platform while the physical organization refers to the actual hardware organization of the platform. The two critical components of parallel computing from a programmer's perspective are ways of expressing parallel tasks and mechanisms for specifying interaction between these tasks. The former is sometimes also referred to as the control structure and the latter as the communication model.

3.2.2.1. Control Structure of Parallel Platforms

Parallel tasks can be specified at various levels of granularity. At one extreme, each program in a set of programs can be viewed as one parallel task. At the other extreme, individual instructions within a program can be viewed as parallel tasks. Between these extremes lie a range of models for specifying the control structure of programs and the corresponding architectural support for them.

Processing units in parallel computers either operate under the centralized control of a single control unit or work independently. In architectures referred to as **single instruction stream, multiple data stream (SIMD)**, a single control unit dispatches instructions to each processing unit. Figure 3.1(a) illustrates a typical SIMD architecture. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units. Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines. More recently, variants of this concept have found use in co-processing units such as the MMX units in Intel processors and DSP chips such as the Sharc. The Intel Pentium processor with its SSE (Streaming SIMD Extensions) provides a number of instructions that execute the same instruction on multiple data items. These architectural enhancements rely on the highly structured (regular) nature of the underlying computations, for example in image processing and graphics, to deliver improved performance.

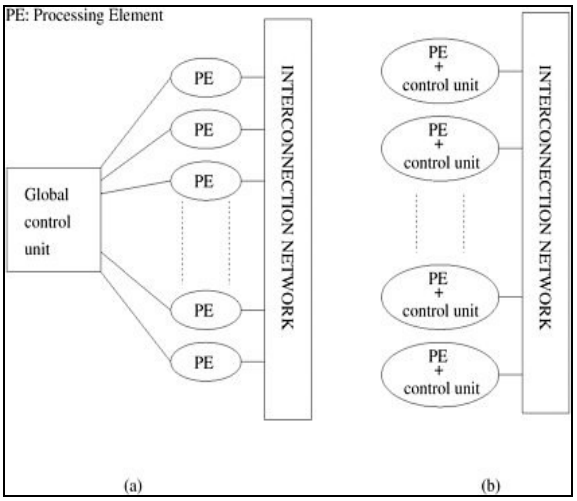

Figure 3.1. A typical SIMD architecture (a) and a typical MIMD architecture (b).

While the SIMD concept works well for structured computations on parallel data structures such as arrays, often it is necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an "activity mask". This is a binary mask associated with each data item and operation that specifies whether it should participate in the operation or not. Primitives such as *where (condition) then* <*stmnt*> <*elsewhere stmnt*> are used to support selective execution. Conditional execution can be detrimental to the performance of SIMD processors and therefore must be used with care.

In contrast to SIMD architectures, computers in which each processing element is capable of executing a different program independent of the other processing elements are called **multiple instruction stream, multiple data stream (MIMD)** computers. Figure 3.1(b) depicts a typical MIMD computer. A simple variant of this model, called the **single program multiple data (SPMD)** model, relies on multiple instances of the same program executing on different data. It is easy to see that the SPMD model has the same expressiveness as the MIMD model since each of the multiple programs can be inserted into one large *if-else* block with conditions specified by the task identifiers. The SPMD model is widely used by many parallel platforms and requires minimal architectural support. Examples of such platforms include the Sun Ultra Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. However, the relative unpopularity of SIMD processors as general purpose compute engines can be attributed to their specialized hardware architectures, economic factors, design constraints, product life-cycle, and application characteristics. In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time. SIMD computers require extensive design effort resulting in longer product development times. Since the underlying serial processors change so rapidly, SIMD computers suffer from fast obsolescence. The irregular nature of many applications also makes SIMD architectures less suitable.

3.2.2.2. Communication Model of Parallel Platforms

There are two primary forms of data exchange between parallel tasks – accessing a shared data space and exchanging messages.

3.2.2.2.1. Shared-Address-Space Platforms

The "shared-address-space" view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared-address-space. Shared-address-space platforms supporting SPMD programming are also referred to as **multiprocessors**. Memory in shared-address-space platforms can be local (exclusive to a processor) or global (common to all processors). If the time taken by a processor to access any memory word in the system (global or local) is identical, the platform is classified as a **uniform memory access (UMA)** multicomputer. On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a **non-uniform memory access (NUMA)** multicomputer. Figures 3.2(a) and (b) illustrate UMA platforms, whereas Figure 3.2(c) illustrates a NUMA platform. An interesting case is illustrated in Figure 3.2(b). Here, it is faster to access a memory word in cache than a location in memory. However, we still classify this as a UMA architecture. The reason for this is that all current microprocessors have cache hierarchies. Consequently, even a uniprocessor would not be termed UMA if cache access times are considered. For this reason, we define NUMA and UMA architectures only in terms of memory access times and not cache access times. Machines such as the SGI Origin 2000 and Sun Ultra HPC servers belong to the class of NUMA multiprocessors. The distinction between UMA and NUMA platforms is important. If accessing local memory is cheaper than accessing global memory, algorithms must build locality and structure data and computation accordingly.

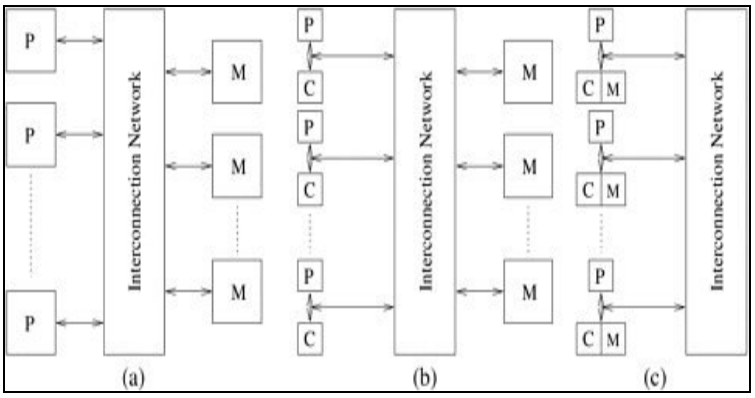

Figure 3.2. Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

The presence of a global memory space makes programming such platforms much easier. All read-only interactions are invisible to the programmer, as they are coded no differently than in a serial program. This greatly eases the burden of writing parallel programs. Read/write interactions are, however, harder to program than the read-only interactions, as these operations require mutual exclusion for concurrent accesses. Shared-address-space programming paradigms such as threads (POSIX, NT) and directives (OpenMP) therefore support synchronization using locks and related mechanisms.

The presence of caches on processors also raises the issue of multiple copies of a single memory word being manipulated by two or more processors at the same time. Supporting a shared-address-space in this context involves two major tasks: providing an address translation mechanism that locates a memory word in the system, and ensuring that concurrent operations on multiple copies of the same memory word have well-defined semantics. The latter is also referred to as the **cache coherence** mechanism. Supporting cache coherence requires considerable hardware support. Consequently, some shared-address-space machines only support an address translation mechanism and leave the task of ensuring coherence to the programmer. The native programming model for such platforms consists of primitives such as get and put. These primitives allow a processor to get (and put) variables stored at a remote processor. However, if one of the copies of this variable is changed, the other copies are not automatically updated or invalidated.

It is important to note the difference between two commonly used and often misunderstood terms – shared-address-space and shared-memory computers. The term shared-memory computer is historically used for architectures in which the memory is physically shared among various processors, i.e., each processor has equal access to any memory segment. This is identical to the UMA model we just discussed. This is in contrast to a distributed-memory computer, in which different segments of the memory are physically associated with different processing elements. The dichotomy of shared- versus distributed-memory computers pertains to the physical organization of the machine. Either of these physical models, shared or distributed memory, can present the logical view of a disjoint or shared-address-space platform. A distributed-memory shared-address-space computer is identical to a NUMA machine.

3.2.2.2.2. Message-Passing Platforms

The logical machine view of a message-passing platform consists of p processing nodes, each with its own exclusive address space. Each of these processing nodes can either be single processors or a shared-address-space multiprocessor – a trend that is fast gaining momentum in modern message-passing parallel computers. Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers. On such platforms, interactions between processes running on different nodes must be accomplished using messages, hence the name **message passing**. This exchange of messages is used to transfer data, work, and to synchronize actions among the processes. In its most general form, message-passing paradigms support execution of a different program on each of the *p* nodes.

Since interactions are accomplished by sending and receiving messages, the basic operations in this programming paradigm are *send* and *receive* (the corresponding calls may differ across APIs but the semantics are largely identical). In addition, since the send and receive operations must specify target addresses, there must be a mechanism to assign a unique identification or ID to each of the multiple processes executing a parallel program. This ID is typically made available to the program using a function such as *whoami*, which returns to a calling process its ID. There is one other function that is typically needed to complete the basic set of message-passing operations – *numprocs*, which specifies the number of processes participating in the ensemble.

With these four basic operations, it is possible to write any message-passing program. Different message-passing APIs, such as the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM), support these basic operations and a variety of higher level functionality under different function names. Examples of parallel platforms that support the message-passing paradigm include the IBM SP, SGI Origin 2000, and workstation clusters.

It is easy to emulate a message-passing architecture containing p nodes on a shared-address-space computer with an identical number of nodes. Assuming uniprocessor nodes, this can be done by partitioning the shared-address-space into p disjoint parts and assigning one such partition exclusively to each processor. A processor can then "send" or "receive" messages by writing to or reading from another processor's partition while using appropriate synchronization primitives to inform its communication partner when it has finished reading or writing the data. However, emulating a shared-address-space architecture on a message-passing computer is costly, since accessing another node's memory requires sending and receiving messages (Grama et al. 2003).

3.3. Message Passing Interface (MPI)

Of the many parallel programming languages, the three main ones are HPF (High Performance Fortran), OpenMP (Open Message Passing) and MPI (Message Passing Interface). The choice of which one to use is simply governed by the following key issues:

- Portability
- Ease of use
- Efficiency
- Cost/Effort

HPF was the first widely supported, portable parallel programming language. It is basically a set of directive-based extensions to Fortran 90, available on both shared and distributed memory machines from workstation clusters to massively parallel supercomputers. One of the advantages of HPF is that the interactions between

processors do not have to be specified explicitly. Unfortunately, HPF has not been adopted by many developers.

OpenMP is a set of compiler directives and callable runtime libraries that extend Fortran and C to allow the development of scalable parallel programs on shared memory machines. OpenMP takes account of developments in the programming languages (e.g. can handle the implementation of pointers in F90) and has been designed to be extensible. Shared memory usage appears to be growing rapidly through the use of OpenMP, perhaps because it is targeted at programmers who need to quickly parallize existing programs without rewriting. OpenMP provides access to the strengths of shared memory parallel computation without excessive programming effort. For example, a single loop can be parallelized by simply inserting standard directives, facilitating incremental parallelism – a real bonus if most of the program execution time is dominated by a single, simple do loop.

Despite the promise of simplicity and scalability, developers have resisted adopting the shared memory programming model because of portability concerns. Previously, every vendor of shared memory systems had created its own extensions to Fortran and C for developers to produce parallel code. The absence of portability has encouraged many developers to adopt a portable message passing model like MPI or PVM. In Message Passing, a distributed memory machine holds all variables in local memory. Work shared across processes requires communication and message passing is the context in which this communication takes place.

Currently, there are a number of Message Passing standards, the main ones being summarized in Table 3.1.

Table 3.1. Message Passing Libraries.

Standard	**Acronym**	**Usage**	**Portability**
Message Passing Interface	MPI	Common	All vendors
Paralel Virtual Machine	PVM	Used less often	Most vendors
Shared Memory	SHMEM	Machine specific	SGI/CRAY
Portable SHMEM library	BSP	Uncommon	Unknown

From the table, it is perhaps clear that MPI would be the library of choice if one wanted to write a portable message passing program. Its popularity is partly due to the fact that it was developed by an international consortium that involved virtually every

parallel computing vendor. As a consequence of the almost universal acceptance of MPI as standard message passing library, many manufacturers have invested a great deal of effort into the performance of MPI. Furthermore, programs can be written in several dialects of FORTRAN and in C. In short, one can use MPI on virtually any computer, even a serial one. This last comment may seem rather pointless, but its worth pointing out that some researchers do not have access to parallel machines and have published research in which the parallel computation was simulated.

The reasons for choosing MPI among the alternatives may be summarized as follows:

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** - Over 115 routines are defined.
- **Availability** - A variety of implementations are available, both vendor and public domain (WEB_4 2003).

3.3.1. Introduction to MPI

Message passing is a programming paradigm used widely on parallel computers, especially Scalable Parallel Computers (SPCs) with distributed memory, and on Networks of Workstations (NOWs). Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications into this paradigm. Each vendor has implemented its own variant. More recently, several public-domain systems have demonstrated that a message-passing system can be efficiently and portably implemented. It is thus an appropriate time to define both the syntax and semantics of a standard core of library routines that will be useful to a wide range of

users and efficiently implementable on a wide range of computers. This effort has been undertaken over the last three years by the Message Passing Interface (MPI) Forum, a group of more than 80 people from 40 organizations, representing vendors of parallel systems, industrial users, industrial and national research laboratories, and universities.

The designers of MPI sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI has been strongly influenced by work at the IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, and PARMACS. Other important contributions have come from Zipcode, Chimp, PVM, Chameleon, and PICL. The MPI Forum identified some critical shortcomings of existing message-passing systems, in areas such as complex data layouts or support for modularity and safe communication. This led to the introduction of new features in MPI.

The MPI standard defines the user interface and functionality for a wide range of message-passing capabilities. Since its completion in June of 1994, MPI has become widely accepted and used. Implementations are available on a range of machines from SPCs to NOWs. A growing number of SPCs have an MPI supplied and supported by the vendor. Because of this, MPI has achieved one of its goals - adding credibility to parallel computing. Third party vendors, researchers, and others now have a reliable and portable way to express message-passing, parallel programs.

The major goal of MPI, as with most standards, is a degree of portability across different machines. The expectation is for a degree of portability comparable to that given by programming languages such as Fortran. This means that the same message-passing source code can be executed on a variety of machines as long as the MPI library is available, while some tuning might be needed to take best advantage of the features of each system. portability Though message passing is often thought of in the context of distributed-memory parallel computers, the same code can run well on a shared-memory parallel computer. It can run on a network of workstations, or, indeed, as a set of processes running on a single workstation. Knowing that efficient MPI implementations exist across a wide variety of computers gives a high degree of flexibility in code development, debugging, and in choosing a platform for production runs.

Another type of compatibility offered by MPI is the ability to run transparently on heterogeneous systems, that is, collections of processors with distinct architectures.

It is possible for an MPI implementation to span such a heterogeneous collection, yet provide a virtual computing model that hides many architectural differences. The user need not worry whether the code is sending messages between processors of like or unlike architecture. The MPI implementation will automatically do any necessary data conversion and utilize the correct communications protocol. However, MPI does not prohibit implementations that are targeted to a single, homogeneous system, and does not mandate that distinct implementations be interoperable. Users that wish to run on an heterogeneous system must use an MPI implementation designed to support heterogeneity. heterogeneous interoperability

Portability is central but the standard will not gain wide usage if this was achieved at the expense of performance. For example, Fortran is commonly used over assembly languages because compilers are almost always available that yield acceptable performance compared to the non-portable alternative of assembly languages. A crucial point is that MPI was carefully designed so as to allow efficient implementations. The design choices seem to have been made correctly, since MPI implementations over a wide range of platforms are achieving high performance, comparable to that of less portable, vendor-specific systems.

An important design goal of MPI was to allow efficient implementations across machines of differing characteristics. efficiency For example, MPI carefully avoids specifying how operations will take place. It only specifies what an operation does logically. As a result, MPI can be easily implemented on systems that buffer messages at the sender, receiver, or do no buffering at all. Implementations can take advantage of specific features of the communication subsystem of various machines. On machines with intelligent communication coprocessors, much of the message passing protocol can be offloaded to this coprocessor. On other systems, most of the communication code is executed by the main processor. Another example is the use of opaque objects in MPI. By hiding the details of how MPI-specific objects are represented, each implementation is free to do whatever is best under the circumstances.

Another design choice leading to efficiency is the avoidance of unnecessary work. MPI was carefully designed so as to avoid a requirement for large amounts of extra information with each message, or the need for complex encoding or decoding of message headers. MPI also avoids extra computation or tests in critical routines since this can degrade performance. Another way of minimizing work is to encourage the reuse of previous computations. MPI provides this capability through constructs such as

persistent communication requests and caching of attributes on communicators. The design of MPI avoids the need for extra copying and buffering of data: in many cases, data can be moved from the user memory directly to the wire, and be received directly from the wire to the receiver memory.

MPI was designed to encourage overlap of communication and computation, so as to take advantage of intelligent communication agents, and to hide communication latencies. This is achieved by the use of nonblocking communication calls, which separate the initiation of a communication from its completion.

Scalability is an important goal of parallel processing. MPI allows or supports scalability through several of its design features. For example, an application can create subgroups of processes that, in turn, allows collective communication operations to limit their scope to the processes involved. Another technique used is to provide functionality without a computation that scales as the number of processes. For example, a two-dimensional Cartesian topology can be subdivided into its one-dimensional rows or columns without explicitly enumerating the processes. scalability

Finally, MPI, as all good standards, is valuable in that it defines a known, minimum behavior of message-passing implementations. This relieves the programmer from having to worry about certain problems that can arise. One example is that MPI guarantees that the underlying transmission of messages is reliable. The user need not check if a message is received correctly.

3.4. Inclusions of MPI

The goal of the Message Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

- Design an applciation programming interface. Although MPI is currently used as a run-time for parallel compilers and for various libraries, the design of MPI primarily reflects the perceived needs of applications programmers.
- Allow efficient communication. Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to a communication coprocessor-processor, where available.

- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface. Also, the semantics of the interface should be language independent.
- Provide a reliable communication interface. The user need not cope with communication failures.
- Define an interface not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- The interface should be designed to allow for thread-safety.

3.5. MPI Standard

The standard includes:

- Point-to-point communication
- Collective operations
- Process groups
- Communication domains
- Process topologies
- Environmental management and inquiry
- Profiling interface
- Bindings for Fortran 77 and C (WEB_2 1995)

MPI standard supports point-to-point, multicast, and broadcast communication styles. From computer networks, it is well known that to use multicast communication, there must be a mechanism to create/destroy groups. MPI provides this mechanism via process groups and communication domains. The communicating parties can also collaborate with each other effectively to reach a single solution collectively. Among many popular programming languages, C and Fortran 77 can be preferred to code parallel programs using MPI.

In fact, MPI has a huge library, however, it has been documented and designed in a good fashion; therefore it does not require enormous effort to get accustomed to it.

CHAPTER 4

PARALLEL ALGORITHMS

The parallel platform chosen for this thesis is not a shared-address-space platform but message-passing platform instead, which means that each process is executed in a completely-isolated memory space and the communication is only available through the passing of messages between. Therefore, the algorithms will be illustrated suited for those platforms.

There are two classes of nodes used to perform a task in parallel. One is the master process and the other is the slave one. There is probably more than one slave process, however, there is usually only one master process coordinating the others. The algorithms are also grouped that way: some algorithms are only for the master process, whereas some are for the others.

4.1. Simple Prime Generation

To generate a prime candidate *y*, again a seed value *x* should be selected to define a starting point for the search process. The parameter *id* is the process number. The servers are numbered from 0 to $p - 1$.

The servers must examine disjoint portions of the search space. To achieve this, each server starts searching at $x + 2 * id$ using increments of $2p$. This interleaves the numbers examined by each server, and allows the node array, as a whole, to examine odd numbers greater than or equal to *x* until a prime candidate is found.

```
procedure SimplePrimeGenerationMaster(x:positive integer)
{x is the starting point for searching a prime}

if even(x) x:=x+1
do
begin
     broadcast the value of x to all the other nodes
     wait until the slave nodes find a candidate and certify it
     get the collective result found
     x:=x+2
end until not found
get the simple prime p
```

Algorithm 4.1. Simple Prime Generation Algorithm at Master.

```
procedure SimplePrimeGeneration(x:positive integer)
{x will be used to calculate the starting point for this node}

x:=x+2*id
do
begin
     start := start + 2*p
     if not SmallFactor(start) then
          found:=miller rabin test(start,1)
end while not found and any other node has not found a prime candidate yet
share the candidate with other nodes
collectively certify the candidate
return the result found to the master
```

Algorithm 4.2. Simple Prime Generation Algorithm at Slave.

4.2. Double Prime Generation

While finding a double prime, the servers will again search disjoint search places. The only difference from the simple prime generation algorithm is that a search is initiated for double prime $r = kt + 1$, for some $k \geq 0$. If received candidate r is not prime, the process is repeated with a new search starting at $k = (r \text{ div } t) + 2$.

```
procedure DoublePrimeGenerationMaster(t:positive integer)
{x is the simple prime to be used for finding a double prime}

k:=0
do
begin
     broadcast the value of t and k to all the other nodes
     wait until the slave nodes find a candidate r and certify it
     get the collective result found
     k:=(r div t) + 2
end until not found
get the double prime r
```

Algorithm 4.3. Double Prime Generation Algorithm at Master.

```
procedure DoublePrimeGeneration(x,k:positive integer)
{t is the simple prime to calculate a double prime using the formula kt+1 by iterating
k }

k:=k+2*(id+1)
do
begin
     y := k*start + 1
     if not SmallFactor(y) then found:=miller rabin test(y,1)
     k:=k+2*p
end while not found and any other node has not found a prime candidate yet
share the candidate with other nodes and master
collectively certify the candidate
return the result found to the master
```

Algorithm 4.4. Double Prime Generation Algorithm at Slave.

4.3. Strong Prime Generation

While finding a strong prime, the servers will again search disjoint search places. The only difference from the two previous algorithms is that a search is initiated for strong prime $p = p_0 + krs$, for some $k \geq 0$. If received candidate p is not prime, the process is repeated with a new search starting at $k = (p \text{ div } rs) + 2$.

```
procedure StrongPrimeGenerationMaster(r,s:positive integer)
{r is the double prime whereas s is the simple prime to be used while generating the
strong prime}

rs:=r*s
u:=(rs+modpower(s,r-1,rs)-modpower(r,s-1,rs)) mod rs;
if odd(u) then p0:=u else p0:=u+rs
k:=0
do
begin
     broadcast the value of p0, rs and k to all the other nodes
     wait until the slave nodes find a candidate p and certify it
     get the collective result found
     k:=(p div rs) + 2
end until not found
get the double prime r
```

Algorithm 4.5. Strong Prime Generation Algorithm at Master.

```
procedure StrongPrimeGeneration(p0,rs,k:positive integer)
{p0 is the starting point to be used in the formula p = p0+krs, s is the simple prime
and r is the double prime. k is the value to be iterated until a prime p is found}

k:=k+2*(id+1)
do
begin
     y := p0+k*rs
     if not SmallFactor(y) then found:=miller rabin test(y,1)
     k:=k+2*p
end while not found and any other node has not found a prime candidate yet
share the candidate with other nodes and master
collectively certify the candidate
return the result found to the master
```

Algorithm 4.6. Strong Prime Generation Algorithm at Slave.

4.4. Parallel Prime Certification

The parallelization of the Miller-Rabin Test is quite straightforward: if the primality test is applied *m* times to a prime candidate *n* in a serial algorithm; in the parallel case, each slave applies the test for *m/p* times where *p* is the number of total nodes. Then, the results are combined to see if any of these nodes has found that the candidate is composite. If not, the candidate is prime, otherwise it is composite.

4.5. RSA in Operation

After generating the strong primes to compute the encryption and generation keys, the process is again distributed among the slaves. The pseudocode will only be shown for the encryption process since the decryption process is the symmetric of it.

```
procedure RSAEncryptionMaster(p,q:positive integer; srcFile, dstFile:
character array)
{p and q are the strong primes that will be used for generating the keys; srcFile and
dstFile are the source file to be encrypted and the destination file where the
encrypted data will be output, respectively}

Compute M := p*q
Compute N := (p-1)*(q-1)
Find an exponent e that is relatively prime to N
Find an exponent d where e*d := 1 mod N
Share e, d and N with the slaves
open srcFile
open dstFile
begin
    Read n times the BLOCKSIZE from srcfile to be encrypted
    Apply the chaining operation to the multiple block
    Send each partition to a distinct slave
    Collect the results and write to dstFile
end until there is any remaining block with a size of n * BLOCKSIZE
if there is any data left in the file then
    encrypt it yourself
    write the result to the file end and the remaining data size to the header field of
        dstFile
end if
close srcFile
close dstFile
```

Algorithm 4.7. RSA Encryption at Master.

procedure RSAEncryption(e,N:positive integer; buf[]:character array)
{*e is the encryption key, N is the modulus and buf is the array holding the partition of the file to be encrypted*}

Encrypt the data found in buf using the key and the modulus by using the equation of
    c = m^e modM
Send c to the master

Algorithm 4.8. RSA Encryption at Slave.

CHAPTER 5

EXPERIMENTAL RESULTS

Since approximately one in every Ln(n) numbers around n is prime, testing can be averaged over O(Ln(n)) trials (Greenfield 1994).[13] However, to see the probabilistic distribution of primes clearly, another approach has been selected in this thesis. That is, two modes have been used to collect the measures. In one mode, the prime searching process for the serial and parallel case has been initiated from randomly chosen origins; and in the other mode, the same origin has been selected for both. It is expected that when the origins are randomly chosen, the speedup measures will have a higher variability than the ones which are taken when the same origin is used.

After the pseudocodes were converted into source code representations and then compiled, the executable program was run against a sample file with a size of 9820 bytes that was used for the encryption/decryption process using different key sizes and different modes. As explained before, these two modes differ in the selection of using either different seeds or the same seed for the random number generator which is supplied by a multiprecision arithmetic library, namely MIRACL. The measures taken for this work are the times elapsed for the strong prime generation (p and q), the creation of the keys, the encryption process and the decryption process; and the speedup values related to those measures. The measures were entered in two tables one set for each seeding mode; namely, different seeds and the same seed for the random number generator. To be able to make a sound judgement based on the measures shown in Table 5.1 and Table 5.2 more easily, the measures related to strong prime generation are plotted in Figure 5.1.

Figure 5.1 illustrates the speedup measures in two modes. On average, the speed up value is greater than one for all key lengths in each mode, however, for the case of using the same seed, it can be observed schematically that the speedup measures have a smaller variability (it does not have sharp edges like the other one). Mathematically, the standard deviation of speedup values for the case of using different seeds is *3.67439*, where as the standard deviation of speedup values for the other case is *1.473729*. This

[13] Greenfield, J. S., 1994. "Distributed Programming Paradigms with Cryptography Applications", *Distributed Computing Environments Group*, M/S B272, Los Alamos, New Mexico, USA, p. 108.

result can be explained as such: if the sequential and the parallel algorithm starts searching the prime numbers randomly, then the starting point for one of them may be very close to a prime number, and moreover, if that algorithm is the sequential one, the results may show wrongly that the parallel one is less efficient than the sequential one. The luck comes into the play in this scenario and may add very much noise to the measures. To overcome this problem, the algorithms should be started from the same origin for searching and the noise may be decreased as much as possible. Using that method, they will run in the same search space starting from the same origin and the comparison will be more accurate. The standard deviations calculated also justifies this fact.

Table 5.1. The Measured Values Using Different Seeds.

DIFFERENT SEEDS (k = 4)			TIME ELAPSED (secs)				SPEEDUP			
			Strong Prime Gen. (p , q)	Key Gen.	Encryption	Decryption	Strong Prime Gen. (p , q)	Key Gen.	Encryption	Decryption
KEY LENGTH (bits)	1024	*Serial*	7,806	0,034	3,013	3,202	6,720	1,028	3,665	3,984
		Parallel	1,162	0,033	0,822	0,804				
	2048	*Serial*	11,829	0,069	9,345	9,410	2,031	0,938	3,161	3,160
		Parallel	5,823	0,074	2,956	2,977				
	3072	*Serial*	50,241	0,116	19,788	19,920	4,448	0,957	3,245	3,278
		Parallel	11,295	0,121	6,099	6,077				
	4096	*Serial*	376,63	0,148	34,069	34,770	8,363	0,932	2,470	2,524
		Parallel	45,036	0,159	13,795	13,774				
	5120	*Serial*	1079,663	0,197	51,712	52,286	8,944	1,001	2,254	2,279
		Parallel	120,716	0,197	22,941	22,938				
	6144	*Serial*	1978,144	0,286	71,538	71,772	12,918	1,002	3,226	3,226
		Parallel	153,130	0,285	22,176	22,245				
	7168	*Serial*	1445,904	0,351	93,451	93,964	2,978	0,617	2,178	2,155
		Parallel	485,526	0,568	42,904	43,611				
	8192	*Serial*	3797,425	0,399	124,936	126,250	3,206	0,885	2,467	2,495
		Parallel	1184,646	0,450	50,645	50,601				
	9216	*Serial*	2641,749	0,627	159,090	160,244	2,192	1,246	2,977	2,961
		Parallel	1205,353	0,503	53,438	54,125				
	10240	*Serial*	2448,290	0,542	193,214	193,330	2,453	1,092	1,588	1,583
		Parallel	998,131	0,496	121,686	122,144				

Table 5.2. The Measured Values Using the Same Seed.

SAME SEED (k = 4)			TIME ELAPSED (secs)				SPEEDUP			
			Strong Prime Gen. (p , q)	Key Gen.	Encryption	Decryption	Strong Prime Gen. (p , q)	Key Gen.	Encryption	Decryption
KEY LENGTH (bits)	1024	*Serial*	3,322	0,031	2,934	3,011	3,170	0,969	3,407	3,570
		Parallel	1,048	0,032	0,861	0,843				
	2048	*Serial*	22,147	0,070	9,765	9,894	6,171	1,071	3,341	3,350
		Parallel	3,589	0,065	2,923	2,953				
	3072	*Serial*	62,914	0,148	19,743	19,645	2,716	1,135	3,230	3,170
		Parallel	23,161	0,130	6,113	6,198				
	4096	*Serial*	179,319	0,167	34,020	34,341	2,636	1,068	2,589	2,593
		Parallel	68,038	0,156	13,140	13,244				
	5120	*Serial*	367,050	0,237	51,436	52,658	4,426	1,198	2,259	2,280
		Parallel	82,935	0,197	22,770	23,090				
	6144	*Serial*	972,797	0,249	70,565	71,726	4,681	0,995	3,210	3,242
		Parallel	207,798	0,250	21,985	22,121				
	7168	*Serial*	2812,432	0,307	94,817	96,474	5,493	0,686	2,210	2,194
		Parallel	511,983	0,448	42,896	43,978				
	8192	*Serial*	1376,377	0,551	124,575	125,200	1,692	0,950	2,464	2,427
		Parallel	813,666	0,580	50,552	51,596				
	9216	*Serial*	4408,174	0,504	160,328	161,161	4,336	1,167	2,990	2,984
		Parallel	1016,627	0,431	53,627	54,002				
	10240	*Serial*	11644,961	0,830	196,065	194,632	5,548	1,546	1,618	1,584
		Parallel	2098,976	0,537	121,142	122,839				

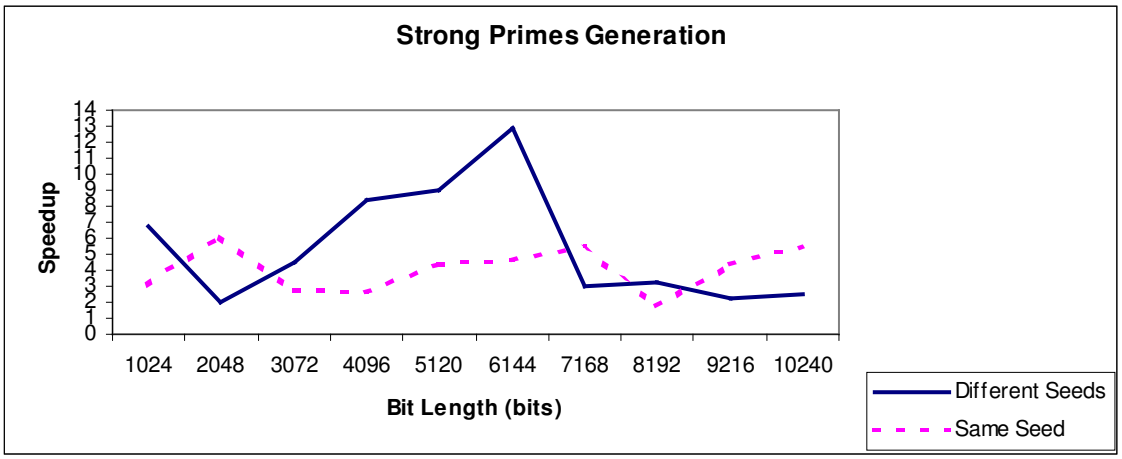

Figure 5.1. Strong Primes Generation.

CHAPTER 6

CONCLUSION AND RECOMMENDED FURTHER WORKS

As seen from the experimental results, there is little doubt that the parallelism can be used for speeding up the operation of several cryptosystems. However, it should be noted that the amount of speedup achieved has a strong relationship with the choice of the platform used for parallelism. If the data to be exchanged between nodes are large, then the shared memory approach will be better. On the other hand, if the data to be exchanged can be limited to a small amount, then message-passing platforms will certainly be more useful. Each method has its own advantages and disadvantages. Shared-memory approach makes the communication faster since all the nodes share a common place for data storage; however, it is more difficult to code and debug due to synchronization and contention problems, and to scale when new nodes are added to increase the speedup value. Message-passing platforms make the programming and maintenance task easier, but they are less efficient when there needs to be heavy communication between the nodes since each data exchanged means a message going between the parties.

It is certain that parallelism can not only be used on the cryptographers' side for forming an unbreakable cryptosystem but also on the cryptanalysts' side to try to break a cryptosystem as soon as possible. In this thesis, the impact of parallelism on generation and certification of primes numbers and the encryption and decryption processes were discussed, but this does not mean that the parallelism can only be used in a constructive manner; in fact, it can also be used in destructive manner – cryptanalysis. This means that the usage of parallelism is not limited to any specific area but has a general usability.

This work may certainly be extended by parallelizing other cryptosystems such as El-Gammal and several techniques used for cryptanalysis such as factoring large primes, which can be used for testing the strength of any new cryptosystem before it is standardized for public usage. In fact, the parallelism is used much more by cryptanalysts than cryptographers in real life.

It should never be overlooked that the hardest work is not just to set up a parallel environment for running parallel algorithms but to find the logic for devising effective parallel solutions; otherwise, it is inevitable to code a parallel program that is inefficient or even worse, that executes slower than its serial counterpart.

REFERENCES

Burnett, S. and Paine, S. 2001. "RSA Security's Official Guide to Cryptography", (Osborne, Berkeley, California, USA).

Denning, E. R. D., 1982. "Cryptography and Data Security", Purdue University, (Addison-Wesley, USA).

Freeman T.L. and Phillips C. 1992. "Parallel Numerical Algorithms", (Prentice Hall, UK).

Grama, A., Gupta, A., Karypis, G. and Kumar, V. 2003. "Introduction to Parallel Computing", Second Edition, E-Book, (Addison-Wesley, USA).

Greenfield, J. S., 1994. "Distributed Programming Paradigms with Cryptography Applications", *Distributed Computing Environments Group*, M/S B272, Los Alamos, New Mexico, USA.

McGregor-Dorsey, Z. S., 1991. "Methods of Primality Testing".

Simmons, G. J., 1992. "Contemporary Cryptology", (IEEE Press, New York, USA).

Tanenbaum, A. S., 2003. "Computer Networks", Fourth Edition, E-Book, (Prentice Hall).

WEB_1, 2002. Prime Numbers, 15/10/2003. http://odin.mdacc.tmc.edu/~krc/numbers/prime.html.

WEB_2, 1995. MPI: The Complete Reference, 15/10/2003. http://www.netlib.org/utk/papers/ mpi-book/node2.html.

WEB_3, 2002. An Introduction to Parallel Programming, 17/10/2003. http://www.linux-mag.com/2002-03/extreme_01.html.

WEB_4, 2003. Message Passing Interface, 18/10/2003. http://www.llnl.gov/computing/tutorials/workshops/workshop/mpi/MAIN.html.

APPENDIX A

DISTRIBUTED ENVIRONMENT

The distributed environment used in this thesis is formed by four identical PCs each with Pentium IV 2 GHz processor and 512 MB RAM connected via a 100 Mbps Ethernet LAN. The operating system preferred is RedHat Linux 9 which is the popular version of RedHat during the time of writing this thesis. Although some parts of the environment setup is a must (such as a multiprecision integer arithmetic library), some other parts are optional (such as NFS) just for increasing the efficiency. All of the environment work will be discussed.

Since the programs are all written in C programming language, the standard C compiler, *gcc*, is preferred and the parallel platform is constructed with MPI (Message Passing Interface). There are many implementations of MPI, and among them, MPICH1 is used which is just a library that can be linked with other C programs. The usual registers used in a standard PC nowadays are at most 64 bits long which forces the users dealing with cryptography to use a multiprecision integer arithmetic library. MIRACL2 is a popular one in this area and it is also another library just like MPICH. To allow each PC send messages to each other, there must be a remote login mechanism used by them, which is Secure Shell (SSH) in this work. SSH allows the remote login process to be done with public-key cryptographic techniques.

The components mentioned in the previous paragraph are the must-part of the work. Besides those, there are also some optional parts to improve the efficiency. NFS and NIS are two techniques used for sharing any directory or system-wide file (such as the shadow file) between several computers, respectively. Using them, there is no need to compile a program in one computer and then copying it to others or to create several accounts on each of them to allow login from terminals.

A.1. Network Setup

The four PCS are connected to each other via an Ethernet LAN using a switch. The PCS are given virtual IPS starting from 192.168.0.1 up to 192.168.0.4 using a netmask of 255.255.255.0.

A.2. MIRACL Setup

After downloading MIRACL to a temporary location, it can be extracted by using a command like

```
% unzip –j –aa –L miracl.zip
```

Next, a tailored build of MIRACL for the underlying system can be created by typing

```
% bash linux
```

All C programs must be linked with miracl.a and during compilation, the MIRACL header files must be included by using specific gcc parameters. For example, a typical compilation and linking command can be

```
% gcc –c <filename>.c –o <obj>.o I/location/to/miracl/include
% gcc –o <output> <objname>.o /location/to/miracl/miracl.a
```

A.3. MPICH Setup

The downloaded TAR.GZ file can be extracted with

```
% tar –zxvf mpich.X.X.X.tar.gz
```

Then, the unzipped and untarred MPICH source must be configured by giving the target location using **–prefix** option, and the login procedure between PCs using **-RSHCOMMAND** option and it must be compiled and installed using **make** utility. A typical example may be

```
% ./configure –-prefix=/location/to/targetdir –RSHCOMMAND=ssh
% make
% make install
```

Successfully completing all of these, the source program can be compiled and linked with MPICH (and also MIRACL) by typing

```
% gcc -c <filename>.c -o <obj>.o -I/location/to/mpich
        -I/location/to/miracl/include
% gcc -o <output> <objname>.o -L/location/to/mpich/lib
        -lmpich /location/to/miracl/miracl.a
```

A.4. SSH Setup

sshd is the name of the server daemon and ssh is the name of the client program. Both of them are set up in each PC to allow each of them to communicate with each other. To use SSH, a user must create a public and private key and save the public key in the server he/she wants to connect to, whereas save the rivate key in the client system. The command for creating the key is

```
% ssh-keygen -t rsa
```

This command creates the keys using RSA as the public-key cryptosystem and puts them under the **.ssh** directory under the user's home as default.

There are two configuration files for the server daemon and the client program: **sshd_config** and **ssh_config**, respectively, residing under **/etc/ssh** directory. The contents of them are shown in Configuration A.1 and Configuration A.2.

```
Port 22
Protocol 2
ListenAddress 0.0.0.0
HostKey /etc/ssh/ssh host rsa key
SyslogFacility AUTHPRIV
PermitRootLogin no
RSAAuthentication yes
PubkeyAuthentication yes
AuthorizedKeysFile .ssh/id rsa.pub
RhostsAuthentication no
IgnoreRhosts yes
RhostsRSAAuthentication no
PasswordAuthentication no
PermitEmptyPasswords no
X11Forwarding yes
PrintMotd yes
Subsystem sftp /usr/libexec/openssh/sftp-server
```

Configuration A.1. /etc/ssh/sshd config.

```
ForwardX11 yes
RhostsAuthentication no
RhostsRSAAuthentication no
RSAAuthentication yes
PasswordAuthentication yes
IdentityFile /.ssh/identity
IdentityFile /.ssh/id rsa
IdentityFile /.ssh/id dsa
Port 22
Protocol 2
Host *
ForwardX11 yes
```

Configuration A.2. /etc/ssh/ssh config.

A.5. NFS Setup

NFS is used to share a directory between hosts. To avoid the executable file from copying each PC individually, the user's home directory has been shared between each one.

The steps for sharing a directory is a two-step process:

At the server-side, the directory to be shared is defined in a configuration file, namely /etc/exports. The content is shown below:

```
/user/home/dir 192.168.0.0/24(rw)
# This configuration allows the user directory to be shared with
# the hosts in the network of 192.168.0.0/24 with read/write
# access
```

Configuration A.3. /etc/exports

At the client-side, the shared directory can be mounted at startup by entering the necessary directive in the /etc/fstab configuration file which is shown below:

```
<serverIP or hostname>:<remotePathToBeMounted> </local/mount/point>
        nfs soft 0 0
```

Configuration A.4. /etc/fstab.

A.6. NIS Setup

NIS is used to share system-wide files between hosts. In this work, it is preferred to ease the management of login accounts. NIS setup is again a two-step process like NFS. The tool to be used is **yp** that comes with standard RedHat distributions.

At the server-side, the configuration file /etc/ypserv.conf is used to define what kind of system-wide information will be shared (passwd info, shadow info etc.) and among which hosts it can be shared. The content of it, which is shown below, says that all system-wide info can be shared with any host. Since the LAN is a virtual one, there is no need to think about security restrictions to avoid any attacks.

```
# * : * : * : none
```

Configuration A.5. /etc/ypserv.conf.

Then, the database that will be used by yp tool must be created by the ypinit command:

```
% /usr/lib/yp/ypinit –m
```

At the client-side, the /etc/yp.conf file is used to direct the host to which server it must bind for getting the system-wide information. Its content is shown below:

```
domain test server <serverIP>
```

Configuration A.6. /etc/yp.conf.

Then, another configuration file, namely **/etc/nsswitch.conf**, must be adjusted to define the order of the lookups for an account, service or other system-wide information when needed. The content shown below directs the client to search its own shadow, group, or passwd file and then the server's if it cannot find the necessary information in its own one.

```
passwd: files nis
shadow: files nis
group: files nis
hosts: files dns
bootparams: nisplus [NOTFOUND=return] files
ethers: files
netmasks: files
networks: files
protocols: files
rpc: files
services: files
netgroup: files
publickey: nisplus
automount: files
aliases: files nisplus
```

Configuration A.7. /etc/nsswitch.conf.

There is one important point left to be mentioned: all the hosts must share a common NIS domain which can be set in /etc/sysconfig/network. An example is shown:

```
NETWORKING=yes
HOSTNAME=<hostname>
NISDOMAIN=<nisdomain>
```

Configuration A.8. /etc/sysconfig/network.