ISSN 2229-5518

Parallel: One Time Pad using Java

Bala Dhandayuthapani Veerasamy, Dr. G.M. Nasira

Abstract— Parallel program allows most efficient use of processors. The efficient processors utilization is the key to maximizing performance of computing systems. This research paper described the computations to be parallelized One Time Pad (OTP) in the form of a sequential program. To transform the sequential computations into a parallel program, the control and data dependencies have to be taken into consideration to ensure that the parallel program produces the same results as the sequential program for all possible input values. The main goal is usually to reduce the program execution time as much as possible by using multiple processors or cores.

Index Terms— Asymmetric encryption, Chip Multiprocessing, Cryptography, Multi-core, Multithreading, One Time Pad, Simultaneous multi-threading, Symmetric encryption

1 INTRODUCTION

CRYPTOGRAPHY [2] refers to the art of protecting transmitted information from unauthorized interception or tampering. The other side of the coin, cryptanalysis, is the art of breaking such secret ciphers and reading the information, or perhaps replacing it with different information. Sometimes the term cryptology is used to include both of these aspects.

Cryptography is closely related to another part of communication theory, namely coding theory. This involves translating information of any kind (text, scientific data, pictures, sound, and so on) into a standard form for transmission, and protecting this information against distortion by random noise. There is a big difference though, between interference by random noise, and interference by a purposeful enemy, and the techniques used are quite different.

Cryptography [7] is probably the most important aspect of communications security and is becoming increasingly important as a basic building block for computer security. The increased use of computer and communications systems by industry has increased the risk of theft of proprietary information.

1.1 Cryptographic Algorithms

In general, cryptographic algorithms [2] are often grouped into two broad categories symmetric and asymmetric. But in practice, today's popular cryptosystems [2] use a hybrid combination of symmetric and asymmetric algorithms. Symmetric and asymmetric algorithms can be distinguished by the types of keys they use for encryption and decryption operations.

An original message is known as the plaintext, while the coded message is called the ciphertext [2]. The process of converting from plaintext to ciphertext is known as enciphering or encryption; restoring the plaintext from the ciphertext is deciphering or decryption. The many schemes used for encryption constitute the area of study known as cryptography. Such a scheme is known as a cryptographic system or a cipher. Techniques used for deciphering a message without any knowledge of the enciphering details fall into the area of cryptanalysis [2]. Cryptanalysis is what the layperson calls "breaking the code." The areas of cryptography and cryptanalysis together are called cryptology.

Symmetric Encryption:

A method of encryption that requires the same secret key to encipher and decipher the message is known as private key encryption or symmetric encryption [1]. Symmetric encryption methods use mathematical operations that can be programmed into extremely fast computing algorithms so that the encryption and decryption processes are done quickly by even small computers. There are a number of popular symmetric encryption cryptosystem. One of the most widely known is the Data Encryption Standards (DES), which was developed by IBM and is based on the company's Lucifer algorithm, this uses a key length of 128 bits. A symmetric encryption scheme has five ingredients

Plaintext: This is the original intelligible message or data that is fed into the algorithm as input.

Encryption algorithm: The encryption algorithm performs various substitutions and transformations on the plaintext.

Secret key: The secret key is also input to the encryption algorithm. The key is a value independent of the plaintext and of the algorithm. The algorithm will produce a different output depending on the specific key being used at the time. The exact substitutions and transformations performed by the algorithm depend on the key.

Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and, as it stands, is unintelligible.

Decryption algorithm: This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

- *Bala Dhandayuthapani Veerasamy is currently a Research Scholar in Information Technology, Manonmaniam Sundaranar University, India. E-mail: dhanssoft@gmail.com*
- *Dr. G.M. Nasira is currently working as an Assistant professor / Computer Science, Chikkanna Govt Arts College, Tirupur -2, Tamilnadu, India. E-mail: nasiragm99@yahoo.com*

Asymmetric Encryption:

Another category of encryption techniques is asymmetric encryption [1]. Whereas the symmetric encryption systems are based on using a single key to both encrypt and decrypt a message, asymmetric encryption uses two different but related keys, and either key can be used to encrypt or decrypt the message. If, however, Key A is used to encrypt the message, only Key B can decrypt it, and if Key B is used to encrypt a message, only Key A can decrypt it. Asymmetric encryption can be used to provide elegant solutions to problems of secrecy and verification. This technique has its highest value when one key is used as a private key, which means that it is kept secret (much like the key of symmetric encryption), known only to the owner of the key pair, and the other key serves as a public key, which means that it is stored in a public location where anyone can use it. This is why the more common name for asymmetric encryption is public key encryption.

Asymmetric [2] algorithms are based on one-way functions. A one-way function is simple to compute in one direction, but complex to compute in the opposite. This is the foundation of public-key encryption. Public-key encryption is based on a hash value, is calculated from an input number using a hashing algorithm. This hash value is essential summary of the original input values. It is virtually impossible to derive the original values without knowing how the values were used to create the hash value. One of the most popular public key cryptosystems is RSA, whose name is derived from Rivest-Shamir-Adleman, the algorithm's developers. The RSA algorithm was the first public key encryption algorithm developed (in 1977) and published for commercial use. It is very popular and has been embedded in both Microsoft's and Netscape's Web browsers to enable them to provide security for e-commerce applications. The patented RSA algorithm has in fact become the de facto standard for public use encryption applications.

2 ONE TIME PADS

One Time Pad [2] is one of the methods in symmetric encryption. The method was presented in 1918 by Gilbert Vernam and Joseph Mauborgne. An Army Signal Corp officer, Joseph Mauborgne, proposed an improvement to the Vernam cipher that yields the ultimate in security. Mauborgne suggested using a random key [7] that is as long as the message, so that the key need not be repeated. In addition, the key is to be used to encrypt and decrypt a single message, and then is discarded. Each new message requires a new key of the same length as the new message. Such a scheme, known as a one-time pad, is unbreakable [7]. It produces random output that bears no statistical relationship to the plaintext. Because the ciphertext contains no information whatsoever about the plaintext, there is simply no way to break the code.

Methods on One Time Pad:

To perform the Vernam cipher encryption operation, the pad values are added to numeric values that represent the plaintext that needs to be encrypted. So, each character of the plaintext is turned into a number and a pad value for that position is added to it. The resulting sum for that character is then converted back to a ciphertext letter for transmission.

There are two way to produce random key. They are additive key and subtractive key. In the additive key, we just add modulo 26 each letter of the message (plaintext) with its corresponding key. If the sum of the two values exceeds 26, then 26 is subtracted from the total. The process of keeping a computed number within a specific range is called modulo; thus, requiring that all numbers be in the range 1-26 is referred to as Modulo 26. In Modulo 26, if a number is larger than 26, then 26 is repeatedly subtracted from it until the number is in the proper range. Here we used 25 (0-25=26), because we used arrary values in the following program 1.

Encryption:

```
      i     j      s      e      r  (plaintext)
   (8)i  (9)j (18)s   (4)e (17)r  (plaintextkey)
   (7)h  (4)e (11)l  (11)l  (14)o  (secrettextkey)
+(15) (13)   (29)   (19)   (31)   (plaintextkey+secrettextkey)
 ---------------------------------------
(15)p (13)n  (4)e (15)p   (6)g  (ciphertextkey-25)
     p    n    e    p    g (ciphertext)
```

Decryption:

```
     p    n    e    p    g (ciphertext)
 (15) p (13)n  (4)e (15)p  (6)g  (ciphertextkey)
  (7) h  (4) e (11)l (11)l  (14)o  (secrettextkey)
- (11)    (9)    (-7)  (4)   (-8)   (ciphertextkey- ciphertextkey)
   ---------------------------------------
 (11)i   (9)j  (18)s   (4)e (17)r  (25-plaintextkey)
     i     j      s      e      r  (plaintext)
```

And the subtractive key works in a similar way to additive key, but here the value of the key is subtracted modulo 26 for the value of the plaintext. The one time pad has a unique cryptographic quality in that it cannot be broken. It is not that it is difficult to crack, nor that it takes a long time, but rather that the cipher text contains no information about the key other than its length.

Advantages:

One-time pads have a number of advantages over traditional cryptography systems. They are extremely computationally efficient both in terms of encryption and decryption. They are also unbreakable under the assumptions that

1. The pad is truly random. Generating truly random one-time-pads is a complex task. Choosing the output of a random number generator, for example, is not a desirable mechanism for generating the pad. Good pads are generated by sampling some random physical event such as radioactive decay.

2. The pad is never reused. Reusing the pad permits an adversary to guess the pad and the two plaintext messages. The basic strategy is to guess that the plaintext version of one message is a particular sequence, and to infer the pad from this.

Elements of the pad are then verified by trying to decode the second message with this pad. Parts of the second message that appear to have been correctly decoded can then be used to guess more of the pad.

3. The pad is secret to the two parties. Having an adversary obtain a copy of the pad permits the adversary to decode all pervious messages, as well as future communications. Destruction of used pads is therefore critical. It is also interesting to note that breaking part of the message does not provide any advantage in term of breaking the rest of the message. There is no short decrypting key to guess. The messages are also self authenticating.

Disadvantages:

An interloper cannot generate fictitious messages as they will not properly decrypt. The primary disadvantage with using a one-time only pad is that the pad must be as large as the message to be transmitted. While this has inhibited the use of one-time pads until now, recently we have seen a significant increase in the number of bits that can be efficiently stored, and easily transported and made available to a mobile user. The one-time pad offers complete security, but in practice, has two fundamental difficulties:

1. There is the practical problem of making large quantities of random keys. Any heavily used system might require millions of random characters on a regular basis. Supplying truly random characters in this volume is a significant task.

2. Even more daunting is the problem of key distribution and protection. For every message to be sent, a key of equal length is needed by both sender and receiver. Thus, a mammoth key distribution problem exists.

Because of these difficulties, the one-time pad is of limited utility, and is useful primarily for low-bandwidth channels requiring very high security.

3 Parallel Programming

Parallel computers have been used for many years, and many different architectural alternatives have been proposed and used. In order to achieve parallel execution in software, hardware must provide a platform that supports the simultaneous execution of multiple threads. Generally speaking, computer architectures can be classified by two different dimensions. The first dimension is the number of instruction streams that particular computer architecture may be able to process at a single point in time. The second dimension is the number of data streams that can be processed at a single point in time. In this way, any given computing system can be described in terms of how instructions and data are processed. This classification system is known as Flynn's taxonomy [5] [8](Flynn, 1972). This taxonomy characterizes parallel computers according to the global control and the resulting data and control flows. Four categories are distinguished:

1. Single-Instruction, Single-Data (SISD)[8]: There is one processing element which has access to a single program and data storage. In each step, the processing element loads an instruction and the corresponding data and executes the instruction. The result is stored back in the data storage. Thus, SISD is the conventional sequential computer according to the von Neumann model.

2. Multiple-Instruction, Single-Data (MISD)[8]: There are multiple processing elements each of which has a private program memory, but there is only one common access to a single global data memory. In each step, each processing element obtains the same data element from the data memory and loads an instruction from its private program memory. These possibly different instructions are then executed in parallel by the processing elements using the previously obtained (identical) data element as operand. This execution model is very restrictive and no commercial parallel computer of this type has ever been built.

3. Single-Instruction, Multiple-Data (SIMD)[8]: There are multiple processing elements each of which has a private access to a (shared or distributed) data memory, but there is only one program memory from which a special control processor fetches and dispatches instructions. In each step, each processing element obtains from the control processor the same instruction and loads a separate data element through its private data access on which the instruction is performed. Thus, the instruction is synchronously applied in parallel by all processing elements to different data elements. For applications with a significant degree of data parallelism, the SIMD approach can be very efficient.

4. Multiple-Instruction, Multiple-Data (MIMD)[8]: There are multiple processing elements each of which has a separate instruction and data access to a (shared or distributed) program and data memory. In each step, each processing element loads a separate instruction and a separate data element, applies the instruction to the data element, and stores a possible result back into the data storage. The processing elements work asynchronously with each other. Multi-core [5] processors or cluster systems are examples for the MIMD model. The core of a processor is the part of the chip responsible for executing instructions.

A physical processor [6] is made up of a number of different resources, including the architecture state—the general purpose CPU registers and interrupt controller registers, caches, buses, execution units, and branch prediction logic. However, in order to define a thread, only the architecture state is required. A logical processor can thus be created by duplicating this architecture space. The execution resources are then shared among the different logical processors. This technique is known as Simultaneous MultiThreading, or SMT[6]. Intel's implementation of SMT is known as Hyper-Threading Technology, or HT Technology. HT Technology makes a single processor appear, from software's perspective, as multiple logical processors. This allows operating systems and applications to schedule multiple threads to logical processors as they would on multiprocessor [5] systems. From a microarchitecture perspective, instructions from logical processors are persistent and execute simultaneously on shared execution resources. In other words, multiple threads can be scheduled, but since the execution resources are shared, it's up to the microarchitecture to determine how and when to interleave the execution of the two threads. When one thread stalls, another

thread is allowed to make progress. These stall events include handling cache misses and branch mispredictions. The next logical step from simultaneous multi-threading is the multi-core processor. Multi-core processors use Chip MultiProcessing (CMP)[6]. Rather than just reuse select processor resources in a single-core processor, processor manufacturers take advantage of improvements in manufacturing technology to implement two or more "execution cores" within a single processor. These cores are essentially two individual processors on a single die. Execution cores have their own set of execution and architectural resources. Depending on design, these processors may or may not share a large on-chip cache. In addition, these individual cores may be combined with SMT; effectively increasing the number of logical processors by twice the number of execution cores. The different processor architectures [6] are highlighted in Fig. 1.

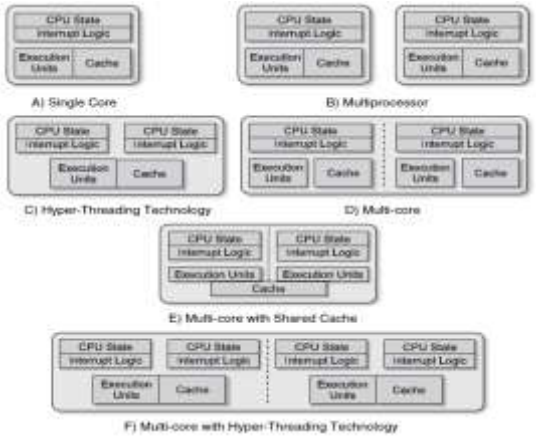

Fig. 1. Simple Comparison of Single-core, Multi-processor, and Multi-Core Architectures

Multi-core platforms allow developers to optimize applications by intelligently partitioning different workloads on different processor cores. Application code can be optimized to use multiple processor resources, resulting in faster application performance. Multi-threaded applications running on multi-core platforms have different design considerations than do multi-threaded applications running on single-core platforms. On single-core platforms, assumptions may be made by the developer to simplify writing and debugging a multi-threaded application. These assumptions may not be valid on multi-core platforms. Two areas that highlight these differences are memory caching and thread priority.

4 Identifying Problem

One Time Pad is resolved in many programming languages such as C, C++. This is not new to the world. But we have not noticed one time pad resolved in java programming language. The following program 1 resolved One Time Pad in sequential way using java language, this itself is a research findings. This program can be exected on single processor environment.

Since we discussed earliy in this paper, there are two ways to produce random key, they are additive key and subtractive key. We used addictive key to produce ciphertext. This One Time Pad can be used various application such as password access of any system, message transfer, etc.

Program 1: sequential One Time Pad

```
package onepad;
class Sequentialonepad {
   //one pad text
   String pad="abcdefghijklmnopqrstuvwxyz";
    char onepad[]=pad.toCharArray();
    //plaing text
    String planintxt="ijser";
    char plain[]=planintxt.toCharArray();
    //secret text
    String enctxt="hello";
    char enc[]=enctxt.toCharArray();

    char cipher[]=new char[5]; //for ciphertext
    static int key1[]=new int[5];// for plaintext key
    static int key2[]=new int[5];//for secrettext key

    int k=0;
    Sequentialonepad(){}

   public void encript(){
      try{

       for(int j=0;j<plain.length;j++){
      //find location of plaintext and placed in key1 array
          for(int i=0;i<onepad.length;i++)
             if(plain[j]==onepad[i])
               key1[j]=i;

    //find location of secrettext and placed in key2 array
          for(int i=0;i<onepad.length;i++)
             if(enc[j]==onepad[i])
               key2[j]=i;

//summation of plaintext and secrettext location
         k=key1[j]+key2[j];
//if key (k) value existe more than 25 and then subtract by 25
             if(k>25)
               k=k-25;
//assign ciphertext locaton by key (k) from onepad array
             cipher[j]=onepad[k];
  System.out.println("(key1) " + key1[j] + " + (key2) " + key2[j]
+" = " + k +" (cipher) " + cipher[j]);
          }
        }catch(Exception e) { System.out.println("Error" + e);}
    }
   public void decryipt(){
        System.out.println("Decrypt...");
        try{

          for(int j=0;j<plain.length;j++){
             for(int i=0;i<onepad.length;i++)
              if(cipher[j]==onepad[i]){
                key1[j]=i;
                break;
```

```
                }

                k=key1[j]-key2[j];
                if(k<0)
                 k=25+k;
                cipher[j]=onepad[k];
                System.out.println("(key1) " + key1[j] + " + (key2) "
+ key2[j] +" = " + k +" (cipher) " + cipher[j]);
              }
           }catch(Exception e) { System.out.println("Error" + e);}
       }
       public static void main(String str[]){
         try{
            Sequentialonepad sp=new Sequentialonepad();
            sp.encript();
         }catch(Exception e){ System.out.println("Error: " +e);}
       }
    }
```

Here we have used onepad[] array to have all alphabets, plain[] array to have plaintext, enc[] array to have secrettext, key1[] array to have location of plaintext on the array, key2[] array to have location of secrettext and cipher[] array used for holding the encrypted text. We assigned plaintext[] array as "ijser" and enc[] array as "hello"; hence the output of the program is

```
Encripty...
(key1) 8 + (key2) 7 = 15 (cipher) p
(key1) 9 + (key2) 4 = 13 (cipher) n
(key1) 18 + (key2) 11 = 4 (cipher) e
(key1) 4 + (key2) 11 = 15 (cipher) p
(key1) 17 + (key2) 14 = 6 (cipher) g
Decrypt...
(key1) 15 + (key2) 7 = 8 (cipher) i
(key1) 13 + (key2) 4 = 9 (cipher) j
(key1) 4 + (key2) 11 = 18 (cipher) s
(key1) 15 + (key2) 11 = 4 (cipher) e
(key1) 6 + (key2) 14 = 17 (cipher) r
```

In the above program, there are two processes we identified. The first process is converting plaintext to plaintext key, and the second process is converting secrettext to secrettext key. When we execute this two process as sequential way, would take some more time. The problem we identified here, to execut the above two processes either in Simultaneous Multi-Threading (SMT) (discussed in section 5.1) or in multi-core processors (Chip MultiProcessing - CMP) (discussed in section 5.2) in parallel way on, would concretely reduce the execution time. Hence, we will have better performance than the sequential One Time Pad.

5 Problem Solving Method

One Time Pad java program using sequential execution on single processor will give normal execution time. Solving any problem in parallel way, encompass three steps. They are discussed bellow.

1. Decomposition of the computations [8]: The computations of the sequential algorithm are decomposed into tasks, and dependencies between the tasks are determined. The tasks are the smallest units of parallelism. Depending on the target system, they can be identified at different execution levels: instruction level, data parallelism, or functional parallelism. In principle, a task is a sequence of computations executed by a single processor or core. Depending on the memory model, a task may involve accesses to the shared address space or may execute message-passing operations. Depending on the specific application, the decomposition into tasks may be done in an initialization phase at program start (static decomposition), but tasks can also be created dynamically during program execution. In this case, the number of tasks available for execution can vary significantly during the execution of a program. At any point in program execution, the number of executable tasks is an upper bound on the available degree of parallelism and, thus the number of cores that can be usefully employed. The goal of task decomposition is therefore to generate enough tasks to keep all cores busy at all times during program execution. But on the other hand, the tasks should contain enough computations such that the task execution time is large compared to the scheduling and mapping time required to bring the task to execution. The computation time of a task is also referred to as granularity: Tasks with many computations have a coarse-grained granularity; tasks with only a few computations are fine-grained. If task granularity is too fine-grained, the scheduling and mapping overhead is large and constitutes a significant amount of the total execution time. Thus, the decomposition step must find a good compromise between the number of tasks and their granularity. According to our sequential One Time Pad program, there are two dissimilar loops used to get keys for plaintext and secrettext. We identified, this two loops are two different task or thread. All other rest of the program will be executed as sequential way only. The actual fraction of code is given bellow.

```
for(int i=0;i<onepad.length;i++) // loop one
   if(plain[j]==onepad[i])
     key1[j]=i;
  for(int i=0;i<onepad.length;i++)//loop two
   if(enc[j]==onepad[i])
     key2[j]=i;
```

2. Assignment of tasks to processes or threads [8]: A process or a thread represents a flow of control executed by a physical processor or core. A process or thread can execute different tasks one after another. The number of processes or threads does not necessarily need to be the same as the number of physical processors or cores, but often the same number is used. The main goal of the assignment step is to assign the tasks such that a good load balancing results, i.e., each process or thread should have about the same number of computations to perform. But the number of memory accesses (for shared address space) or communication operations for data exchange (for distributed address space) should also be taken into consideration. For example, when using a shared address space, it is useful to assign two tasks which work on the same data set to the same thread, since this leads to a good cache usage. The assignment of tasks to processes or threads is also

called scheduling. For a static decomposition, the assignment can be done in the initialization phase at program start (static scheduling). But scheduling can also be done during program execution (dynamic scheduling). According to the above fraction of code, we decided to have the following named threads.

```
Thread1:
        for(int i=0;i<onepad.length;i++)
            if(plain[j]==onepad[i])
              key1[j]=i;
Thread2:
            for(int i=0;i<onepad.length;i++)
             if(enc[j]==onepad[i])
               key2[j]=i;
```

This Thread1 considered as task1 and Thread2 considered as task2, executing on multi processor environment.

3. Mapping of processes or threads to physical processes or cores [8]: In the simplest case, each process or thread is mapped to a separate processor or core, also called execution unit in the following. If fewer cores than threads are available, multiple threads must be mapped to a single core. This mapping can be done by the operating system, but it could also be supported by program statements. The main goal of the mapping step is to get an equal utilization of the processors or cores while keeping communication between the processors as small as possible. According to our program, we have identified first loop as "plain" thread and second loop as "secret" thread. The fraction of code exposed bellow.

```
if(t.getName().equals("plain")){
      for(int i=0;i<onepad.length;i++)
       if(plain[j]==onepad[i])
         key1[j]=i;
   }

 if(t.getName().equals("secret")){
      for(int i=0;i<onepad.length;i++)
       if(enc[j]==onepad[i])
         key2[j]=i;
```

1.1 Parallel One Time Pad on SMT

The following program 2 illustates Simultaneous Multi-Threading (SMT), it can execute without scheduling the thread on the single or multiprocessing environment.

Program 2 – Parallel One Time Pad

```
package onepad;
class Parallelonepad implements Runnable{
   Thread t;
   // one pad text
   String pad="abcdefghijklmnopqrstuvwxyz";
    char onepad[]=pad.toCharArray();
   // plaintext
    String planintxt="ijser";
    char plain[]=planintxt.toCharArray();
   //secrettext
    String enctxt="hello";
      char enc[]=enctxt.toCharArray();
      //ciphertext
      static char cipher[]=new char[5];
      static int key1[]=new int[5];
      static int key2[]=new int[5];
      static int key3[]=new int[5];

      int k=0;
      Parallelonepad(){}
      Parallelonepad(String name){
        t=new Thread(this,name);
        t.start();
      }

      public void run(){
 try{
            for(int j=0;j<plain.length;j++){
             if(t.getName().equals("plain")){   //thread plain
               for(int i=0;i<onepad.length;i++)
                if(plain[j]==onepad[i])
                  key1[j]=i;
              }
              t.sleep(10);
             if(t.getName().equals("secret")){   //thread secret
                for(int i=0;i<onepad.length;i++)
                 if(enc[j]==onepad[i])
                   key2[j]=i;
              }
             }
 }catch(Exception e) { System.out.println("Thread Error" + e);}
}
public void encript(){ //encryption method
    System.out.println("Encript...");
    for(int j=0;j<plain.length;j++){
         k=key1[j]+key2[j];
         if(k>25)
            k=k-25;
         cipher[j]=onepad[k];
         System.out.println("(key1) " + key1[j] + " + (key2) " +
key2[j] +" = " + k +" (cipher) " + cipher[j]);
         }
       }
public void decryipt(){ // decryption method
    System.out.println("Decrypt...");
    try{
          for(int j=0;j<plain.length;j++){
              for(int i=0;i<onepad.length;i++)
               if(cipher[j]==onepad[i]){
                  key3[j]=i;
                  break;
                }

              k=key3[j]-key2[j];
              if(k<0)
               k=25+k;
              cipher[j]=onepad[k];
              System.out.println("(key3) " + key3[j] + " + (key2) "
+ key2[j] +" = " + k +" (cipher) " + cipher[j]);
```

```
            }
          }catch(Exception e) { System.out.println("Error" + e);}
       }

public static void main(String str[]){
try{
Parallelonepad Parallelpad1 = new Parallelonepad("plain");
Parallelonepad Parallelpad2 = new Parallelonepad("secret");
Parallelpad1.t.join();
Parallelpad2.t.join();
if(Parallelpad1.t.isAlive()==false                                    &&
Parallelpad2.t.isAlive()==false){
     new Parallelonepad().encript();
     new Parallelonepad().decryipt();}
}catch(Exception e){ System.out.println("Error: " +e);}
  }
}
```

As like previous sequential One Time Pad program, here we have used onepad[] array to have all alphabets, plain[] array to have plaintext, enc[] array to have secrettext, key1[] array to have location of plaintext on the array, key2[] array to have location of secrettext and cipher[] array used for holding the encrypted text. But all the arrays declared as static, because during thread running time, the value of array content might be changed. Since, we cannot reassign array values. Therefore, key3[] array used to hold cipher key values while decreption process occured. We assigned plaintext[] array as "ijser" and enc[] array as "hello"; the output of the program is as like the same like previous sequential program. But plaintext keys and secrettext keys were identified by two different threads. Since this model presents with Symutaneous MultiThreading, it cannot be mapped in which processor core the 'plain thread' and 'secret thread' were run on.

```
Encript...
(key1) 8 + (key2) 7 = 15 (cipher) p
(key1) 9 + (key2) 4 = 13 (cipher) n
(key1) 18 + (key2) 11 = 4 (cipher) e
(key1) 4 + (key2) 11 = 15 (cipher) p
(key1) 17 + (key2) 14 = 6 (cipher) g
Decrypt...
(key3) 15 + (key2) 7 = 8 (cipher) i
(key3) 13 + (key2) 4 = 9 (cipher) j
(key3) 4 + (key2) 11 = 18 (cipher) s
(key3) 15 + (key2) 11 = 4 (cipher) e
(key3) 6 + (key2) 14 = 17 (cipher) r
```

5.2 Parallel One Time Pad on CMP

In general, a scheduling algorithm is a method to determine an efficient execution order for a set of tasks of a given duration on a given set of execution units. Typically, the number of tasks is much larger than the number of execution units. There may be dependencies between the tasks, leading to precedence constraints. Since the number of execution units is fixed, there are also capacity constraints. Both types of constraints restrict the schedules that can be used. Usually, the scheduling algorithm considers the situation that each task is executed sequentially by one processor or core (single-processor tasks). But in some models, a more general case is also considered which assumes that several execution units can be employed for a single task (parallel tasks), thus leading to a smaller task execution time. The overall goal of a scheduling algorithm is to find a schedule for the tasks which defines for each task a starting time and an execution unit such that the precedence and capacity constraints are fulfilled and such that a given objective function is optimized. Often, the overall completion time (also called makespan) should be minimized. This is the time elapsed between the start of the first task and the completion of the last task of the program.

Program 3: JThreadCore.java

```
//this is from my earliar research; see reference # [3]
package onepad;
// inherited with Thread class
public class JThreadCore extends Thread{
// get the available system processor
private int Processors ;
// select the processor to execute the thread
private int ProcessorNum;
//assign the name of the thread
private static  String name;
Thread t;
public JThreadCore(){

}
public JThreadCore(String name){
//interchange the thread name value to class variable
this.name=name;
}
public JThreadCore(String name,int ProcessorNum){
//interchange the thread name value to class variable
this.name=name;
//interchange the processor number to class variable
this.ProcessorNum=ProcessorNum;
//call setAffinity method
setAffinity(ProcessorNum);
}
//synchronized method to select the processor
public synchronized void setAffinity(int ProcessorNum){
// interchange processor number to class variable
this.ProcessorNum=ProcessorNum;
//get the available processor in the system
Processors  = Runtime.getRuntime().availableProcessors();
//check selected processor is greater than equal to the selected
//processor
if (ProcessorNum>=Processors )
throw new IllegalArgumentException("This processor is not
available");
//create threads using loop
for(int i=0; i < Processors ; i++)
//check i value is equal to user selected processor
  if (i==ProcessorNum){
//create thread
 t=new Thread(name);  }
 }
//get the affinity of the thread
```

```
public int getAffinity(){
//return the core  number, in which current thread is running
 return ProcessorNum;
 }
}
```

Program 4: ParallelonepadCore.java

```
package onepad;
import onepad.JThreadCore;
class ParallelonepadCore extends JThreadCore{
private static String name;
private static int wait;
private static int pri,afi;
  String pad="abcdefghijklmnopqrstuvwxyz";
    char onepad[]=pad.toCharArray();

    String planintxt="ijser";
    char plain[]=planintxt.toCharArray();

    String enctxt="hello";
    char enc[]=enctxt.toCharArray();

    static char cipher[]=new char[5];
    static int key1[]=new int[5];
    static int key2[]=new int[5];
    static int key3[]=new int[5];

    int k=0;
ParallelonepadCore(){
}

ParallelonepadCore(String name,int afi){
super(name);
this.name=name;
this.afi=afi;
setAffinity(afi);
start();
}
public void run(){
  try{

          for(int j=0;j<plain.length;j++){
            if(t.getName().equals("plain")){
              for(int i=0;i<onepad.length;i++)
               if(plain[j]==onepad[i]){
                  key1[j]=i;
                 System.out.println(t.getName() +" thread on core "
+ getAffinity() + " with the plaintext key " + key1[j]);
                }
            }
          t.sleep(10);
          if(t.getName().equals("secret")){
              for(int i=0;i<onepad.length;i++)
               if(enc[j]==onepad[i]){
                  key2[j]=i;
                 System.out.println(t.getName() + " thread on core
" + getAffinity() + " with the secrettext key " + key2[j]);
                }
            }
          }
        }catch(Exception e) { System.out.println("Thread Error"
+ e);}
    }

    public void encript(){
      System.out.println("Encript...");
       for(int j=0;j<plain.length;j++){
        k=key1[j]+key2[j];
        if(k>25)
           k=k-25;
        cipher[j]=onepad[k];
        System.out.println("(key1) " + key1[j] + " " + (key2) " +
key2[j] +" = " + k +" (cipher) " + cipher[j]);
        }
            }

      public void decryipt(){
        System.out.println("Decrypt...");
        try{

          for(int j=0;j<plain.length;j++){
              for(int i=0;i<onepad.length;i++)
               if(cipher[j]==onepad[i]){
                  key3[j]=i;
                  break;
                }

              k=key3[j]-key2[j];
              if(k<0)
               k=25+k;
              cipher[j]=onepad[k];
              System.out.println("(key3) " + key3[j] + " " + (key2) "
+ key2[j] +" = " + k +" (cipher) " + cipher[j]);
            }
          }catch(Exception e) { System.out.println("Error" + e);}
       }

  public static void main(String s[]){
  try{
  ParallelonepadCore a1= new ParallelonepadCore("plain",0);
  ParallelonepadCore a2= new ParallelonepadCore("secret",1);
  a1.join(); a2.join();
  if(a1.t.isAlive()==false && a2.t.isAlive()==false){
            new ParallelonepadCore().encript();
            new ParallelonepadCore().decryipt();}
   }catch(Exception e){System.out.println(e);                    }
  }
}
```

As like previous parallel One Time Pad program, here we have used onepad[] array to have all alphabets, plain[] array to have plaintext, enc[] array to have secrettext, key1[] array to have location of plaintext on the array, key2[] array to have location of secrettext and cipher[] array used for holding the

encrypted text, key3[] array used to hold cipher key values while decreption process occured. We assigned plaintext[] array as "ijser" and enc[] array as "hello";the output of the program shown bellow. The main advantage of this programming model can be identified the core processing with, in which processor core the 'plain thread' and 'secret thread' were run on.

```
plain thread on core 0 with the plaintext key 8
secret thread on core 1 with the secrettext key 7
plain thread on core 0 with the plaintext key 9
secret thread on core 1 with the secrettext key 4
plain thread on core 0 with the plaintext key 18
secret thread on core 1 with the secrettext key 11
plain thread on core 0 with the plaintext key 4
secret thread on core 1 with the secrettext key 11
plain thread on core 0 with the plaintext key 17
secret thread on core 1 with the secrettext key 14
Encript...
(key1) 8 + (key2) 7 = 15 (cipher) p
(key1) 9 + (key2) 4 = 13 (cipher) n
(key1) 18 + (key2) 11 = 4 (cipher) e
(key1) 4 + (key2) 11 = 15 (cipher) p
(key1) 17 + (key2) 14 = 6 (cipher) g
Decrypt...
(key3) 15 + (key2) 7 = 8 (cipher) i
(key3) 13 + (key2) 4 = 9 (cipher) j
(key3) 4 + (key2) 11 = 18 (cipher) s
(key3) 15 + (key2) 11 = 4 (cipher) e
(key3) 6 + (key2) 14 = 17 (cipher) r
```

6 RESULT AND DISCUSION

At this point one may wonder how we measure the performance benefit of parallel programming. Intuition tells us that if we can subdivide disparate tasks and process them simultaneously, we are likely to see significant performance improvements. In the case where the task is completely independent, the performance benefit is obvious, but most cases are not so simple. How does one quantitatively determine the performance [4] benefit of parallel programming? One metric is to compare the elapsed run time of the best sequential algorithm versus the elapsed run time of the parallel program. This ratio is known as the speedup and characterizes how much faster a program runs when parallelized.

Speedup is defined in terms of the number of physical threads used in the parallel implementation. Based on the parallel One Time Pad program solving methods, we exctely acquired the actual performance improvements on, where plaintext key and secret key loop is occurred. When we have huge amont of plaintext expected to be encrypted, this required to execute the loop based on the length of the plaintext. And the securettext is also expeted to have same length what plaintext length has. If two different loops executed in a sequential One Time Pad as one after the other loop based on the length of the plaintext and secrettext, diffinately the performance of the program will be degraded. When this situation occurred in parallel One Time Pad program, there is possiblitis of getting more performance than sequential One Time Pad program. In this program, we assigned static array size as 5, because the plaintext is "ijser" and secrettext is "hello". If suppose you need to have more length plaintext, we sugget to get inputs for the plaintext in addition to secrettext and assign them with the dynamic array.

7 CONCLUSION

Working with multiple threads on multiprocessor is very natural to improve the performance based on number of threads on CPUs. We implemented One Time Pad cryptography technique with excellent programs in a sequaltial as well as parallel way. Especially, parallel One Time Pad programs have only two threads that can utilize the parallel processing machine. Parallel One Time Pad program cannot be mapped the threads on the execution cores of the CPUs. However the parallel One Time Pad with core program can be mapped the threads on the execution cores of the CPUs. With respect to the result of the actual performace only based on the length of the plaintext/secrettext. If the length of the plaintext/secrettext is less, there is less performance in parallel executon. However, if the length of the plaintext/secrettext is more, there is great performance in parallel One Time Pad programs.

REFERENCES

[1] Ajit Singh and Rimple Gilhotra, International Journal of Network Security & Its Applications (IJNSA), Vol.3, No.3, May 2011, ISSN: 09752307.
[2] Alan g. Konheim, Computer Security and Cryptography, John Wiley & Sons, 2007, ISBN-10: 0-471-94783-0.
[3] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, "Setting CPU Affinity in Windows Based SMP Systems Using Java", International Journal of Scientific & Engineering Research, USA, Volume 3, Issue 4, April 2012, ISSN 2229-5518.
[4] Darryl Gove, Multicore Application Programming For Windows, Linux, and Oracle® Solaris, Pearson Education, 2011, ISBN-10: 0-321-71137-8.
[5] Numerics, Applications, and Trends, Parallel Computing, Springer, 2009, ISBN 978-1-84882-408-9
[6] Shameem Akhter, Jason Roberts, Multi-Core Programming Increasing Performance through Software Multi-threading, Intel Corporation, ISBN 0-9764832-4-6.
[7] Sharad Patil, Manoj Devare & Ajay Kumar, International Journal of Computer Science and Security (IJCSS), Volume (3): Issue (2), Feb 2011, ISSN (online): 1985-1553.
[8] Thomas Rauber, Gudula Runger, Parallel Programming For Multicore and Cluster Systems, Springer, 2010, ISBN 978-3-642-04817-3.