Enhancing of Encryption Algorithms (RSA/RC5) by Parallelization

A PROJECT REPORT

Submitted by

Sl No	Reg No	Name
1	19BCE2279	ANCHIT AGARWAL
2	19BCE2679	NAVDEEP SUREKA

Course Code: CSE4001

Course Title: PARALLEL AND DISTRIBUTED COMPUTING

Under the guidance of

Dr. Siva. Shanmugam

Associate Professor, SCOPE,

VIT , Vellore.

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

April, 2022

Table of Contents

ABSTRACT

Today, encryption algorithms are a key component of information transmission since every bit of data requires encryption to be sent from one point to another without being damaged or leaking data. Every bit of data lost jeopardizes the system.

Encryption algorithms are made up of needed stages that involve a key that aids in the conversion of plaintext to ciphertext. Until it is transformed to plain text, the generated ciphertext should be comprehensible. Also, in order to obtain the exact same plain text again, we must carefully follow all of the necessary procedures.

Various methods for converting plaintext to ciphertext have been developed throughout the years by researchers. AES (Advanced Encryption Standard), DES (Data Encryption Standard), TDES, RSA, Blowfish Algo, HMAC, and others are examples of these algorithms. Because most of these algorithms operate serially, it is possible that they will take a long time to accept input, process it, and deliver a response.

By parallelizing the technique, we hope to minimise the overall computing time necessary to finish the encryption process. We want to parallelize methods such as RSA, RC5 (Rivest Cipher 5), and the International Data Encryption Algorithm (IDEA).

1. Introduction

Today, encryption methods are a critical component of data sharing since every item of data must be encrypted before it can be sent from one location to another without risk of data leaking. Every bit of data lost jeopardizes the system. To transform plain text to encrypted text, encryption algorithms consist of stages that need the usage of a key. To guarantee that the encrypted data is retrieved, all processes must be scrupulously followed. As a result, many algorithms are programmed to run in a sequential and sequential order. As a result, computing the method takes a lengthy time. We expect to reduce the overall computation time necessary to complete the procedure by using the notion of multi-threading to parallelize it.

2. Literature Survey

2.1. Survey

Paper Name and Author	Idea Methodology	Drawbacks

INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA) – A TYPICAL ILLUSTRATION By Sandipan Basu (2011)	In this paper the author tries to make the IDEA encryption algorithm more efficient by preventing the same set of operations again and again. Instead he has identified those operations and performs those operation once.	This paper follows the approach of another researcher who faced a weak key problem, i.e. detecting key used would not require a significant amount of effort. This issue has yet to be resolved.
Research and Implementation of RSA Algorithm for Encryption and Decryption By Xin Zhou, Xiaofei Tang (2011)	The authors of this paper have designed a complete and practical RSA encoding solution. They have combined the symmetric key algorithms and public key cryptography algorithms, the symmetric key is used to encrypt the confidential information needed to be sent, and the RSA asymmetric key cryptosystem is used to send the DES key.	The algorithm used in this approach has a very slow data transfer rate.

Application of RC5 for IoT devices in Smart Transportation System By Nawal Alsaffar, Wael Elmedany, Hayat Ali (2019)	This paper focuses on the use of the RC5 encryption technique to secure the vehicle's user data. It has been used as a security technique in smart transportation to protect users' privacy. The author used Quartus Prime Lite Edition version 18 to simulate RC5 with FPGA.	Because the algorithm's resource usage is not optimised, it can only work with a limited number of IoT devices. Few devices may require the use of a low-resource algorithm to protect users' privacy.
Chaos Based Enhanced RC5 Algorithm for Security and Integrity of Clinical Images in Remote Health Monitoring By ROMANA SHAHZADI, SYED MUHAMMAD ANWAR, FARHAN QAMAR (2019)	This paper focuses on the use of RC5 for providing security in health monitoring, The encryption key is iterated based on feedback mode. Due to the large keyspace the proposed scheme provides more security to the patient's data	Although the author was successful in securing the patient's data but there is still more enhancements to be done using higher chaotic maps & security of clinical signals through optical chaos
A Novel and Efficient Design for RSA Cryptosystem with Very Large Key Size By Wei Wang, Xinming Huang (2015)	In this paper the authors proposed a FFT based modular multiplication and exponentation algorithm for RSA. The proposed meathos has more advantages in terms of throughput and performance when the key size is larger	Although the authors were successful in increasing the performance, The SRAMs store alot of pre-computed data which basically occupies 80% of the chip area

Variants of RSA and their Cryptanalysis By Kannan Balasubramanian (2014)	In this paper the authors introduced the variants of RSA which overcome the cost of implementing RSA algorithm. The attacks on the variants are based on the choice of prime numbers chosen from the modulus and the choice of encryption & decryption exponents	Even though the cost of implementing RSA, but there are some factors like the process of computation, plain text being too big and value of public exponent being too big results in the same cost as that of original RSA algorithm.

2.2. Problem Definition

As the volume of data and connections has expanded, so has data exchange. Every day, around 15 million communications are transmitted, and everything we type, talk about, or command is logged in our network. We want our talks and actions to remain safeguarded in such a place, rather than ending up in a data market to be exploited by certain persons or groups. The value of data has now surpassed that of oil. This creates worries about the security of data storage and transmission, which is where encryption comes into play. Humans have made major contributions to the field of encryption, developing ways that make data virtually hard to decipher by unauthorized eyes.

3. Overview of the Work

3.1. Objectives of the Project

Encryption algorithms are made up of steps that need the use of a key to transform plaintext to ciphertext. Until the ciphertext is translated to plain text, it should be understandable. We must also complete all of the necessary steps in order to extract the same simple text.

Over the years, researchers have devised a number of algorithms for converting plain text to ciphertext. AES (Advanced Encryption Standard), DES (Data Encryption Standard), TDES, RSA, Blowfish Algo, HMAC, and others are examples of these algorithms. Because most of these algorithms are serial, it's possible that taking the input, processing it, and providing the answer will take a long time.

The goal of this product is to parallelize these algorithms in order to reduce their execution time.

3.2. Software Requirements

To run our code in a system, it should complete the following requirements

1. openmp package
2. Any linux distribution
3. GCC 7.4 and above

3.3. Hardware Requirements

The processor can be single-core or multi-core. The processor should not be single-threaded. The performance of all the programs has been tested on an Intel i5 9th Generation processor.

4. System Design

4.1. Algorithms

4.1.1 RSA

- Choose two distinct prime numbers p and q

  For security purposes, the integers p and q should be chosen at random and should be similar in magnitude but differ in length by a few digits to make factoring harder. Prime integers can be efficiently found using a primality test.

- Compute n = pq

  is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length. n is released as part of the public key.

- Compute $\lambda(n)$, where λ is Carmichael's totient function. Since $n = pq$, $\lambda(n) = \text{lcm}(\lambda(p), \lambda(q))$, and since p and q are prime, $\lambda(p) = \varphi(p) = p - 1$, and likewise $\lambda(q) = q - 1$. Hence $\lambda(n) = \text{lcm}(p - 1, q - 1)$.

- Choose an integer e such that $1 < e < \lambda(n)$ and $\gcd(e, \lambda(n)) = 1$; that is, e and $\lambda(n)$ are coprime.

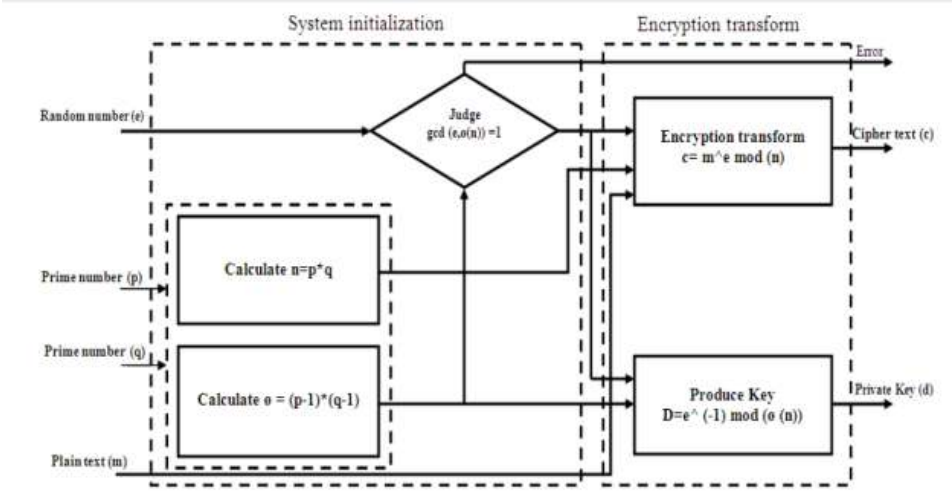
- Determine d as $d \equiv e^{-1} \pmod{\lambda(n)}$; that is, d is the modular multiplicative inverse of e modulo $\lambda(n)$.

4.1.2 RC5

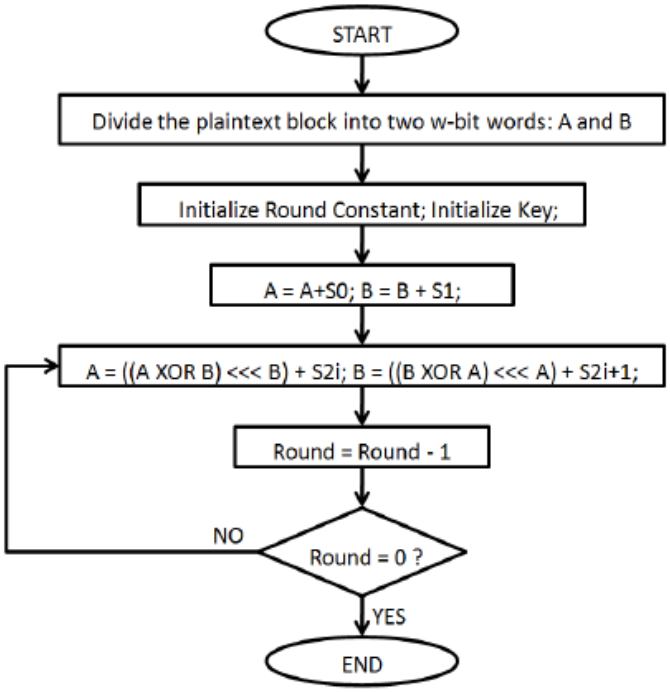
- RC5 encryption and decryption both expand the random key into 2(r+1) words that will be used sequentially (and only once each) during the encryption and decryption processes
- Initialize the constant p & q
- Convert the secret key k from bytes to the word, the secret key k of size in “b” bytes is used to initialize the array consisting of the instance of RC5.
- Initialize the subkey S, The subkey is of the size t=2(r+1) and is done by using constant p and q
- Sub key mixing, a temporary array is formed on the basis of the secret key that is entered by the user
- For the encryption; after the One-time initialization of plain text blocks A and B by adding S[0] and S[1] to A and B respectively. These operations are mod2^w. XOR A and B. A=A^B and then the Cyclic left shift is done to the new value of A by B bits.
- Add S[2*i] to the output of the previous step. and this is the new value of A.
- XOR B with the new value of A and store it in B.
- Cyclic left shifts the new value of B by A bits.
- Add S[2*i+1] to the output of the previous step. This is the new value of B.
- Repeat the entire procedure (except one-time initialization) r times.

Block Diagrams

- **RSA**



- **RC5**



5. Implementation

5.1. Description of Modules/Programs

5.2. Source Code

5.2.1. RSA

a. Serial Code

```
//Program for RSA asymmetric cryptographic algorithm
//for demonstration values are relatively small compared to
practical application
#include<stdio.h>
#include<math.h>
#include <time.h>

//to find gcd
int gcd(int a, int h)
{
   int temp;
   while(1)
   {
       temp = a%h;
       if(temp==0)
       return h;
       a = h;
       h = temp;
   }
}
int main()
{
   double time_spent = 0.0;
   clock_t begin = clock();
   //2 random prime numbers
   double p = 3;
   double q = 7;
   double n=p*q;
   double count;
   double totient = (p-1)*(q-1);
   //public key
   //e stands for encrypt
   double e=2;
   //for checking co-prime which satisfies e>1
   while(e<totient)
   {
       count = gcd(e,totient);
```

```
        if(count==1)
            break;
        else
            e++;
    }
    //private key
    //d stands for decrypt
    double d;
    //k can be any arbitrary value
    double k = 2;
    //choosing d such that it satisfies d*e = 1 + k * totient
    d = (1 + (k*totient))/e;
    double msg = 12;
    double c = pow(msg,e);
    double m = pow(c,d);
    c=fmod(c,n);
    m=fmod(m,n);
    printf("Message data = %lf",msg);
    printf("\np = %lf",p);
    printf("\nq = %lf",q);
    printf("\nn = pq = %lf",n);
    printf("\ntotient = %lf",totient);
    printf("\ne = %lf",e);
    printf("\nd = %lf",d);
    printf("\nEncrypted data = %lf",c);
    printf("\nOriginal Message Sent = %lf",m);

    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

    printf("\n\tTime taken is: %f seconds \n", time_spent);
    return 0;
}
```

b. Parallel Code

```
//Program for RSA asymmetric cryptographic algorithm
//for demonstration values are relatively small compared to
practical application
```

```c
#include<stdio.h>
#include<math.h>
#include <omp.h>
//to find gcd
int gcd(int a, int h)
{
   int temp;
   #pragma omp parallel num_threads(1)
   {
       while(1)
       {
           #pragma omp for
           {
               temp = a%h;
               if(temp==0)
               return h;
               a = h;
               h = temp;
           }
       }
   }
}
int main()
{
   double t1 = omp_get_wtime();
   //2 random prime numbers
   double p = 3;
   double q = 7;
   double n=p*q;
   double count;
   double totient = (p-1)*(q-1);
   //public key
   //e stands for encrypt
   double e=2;
   //for checking co-prime which satisfies e>1
   #pragma omp parallel num_threads(1)
   {
       #pragma omp
       while(e<totient)
       {
```

```
            count = gcd(e,totient);
            if(count==1)
                break;
            else
                e++;
        }
    }
    //private key
    //d stands for decrypt
    double d;
    //k can be any arbitrary value
    double k = 2;
    //choosing d such that it satisfies d*e = 1 + k * totient
    d = (1 + (k*totient))/e;
    double msg = 12;
    double c = pow(msg,e);
    double m = pow(c,d);
    c=fmod(c,n);
    m=fmod(m,n);
    printf("Message data = %lf",msg);
    printf("\np = %lf",p);
    printf("\nq = %lf",q);
    printf("\nn = pq = %lf",n);
    printf("\ntotient = %lf",totient);
    printf("\ne = %lf",e);
    printf("\nd = %lf",d);
    printf("\nEncrypted data = %lf",c);
    printf("\nOriginal Message Sent = %lf",m);

    double t2 = omp_get_wtime();
    printf("\nTime taken is: %f seconds", (t2 - t1));
    printf("\n\n\n");
    return 0;
}
```

5.2.2. RC5

a. Serial Code

```
#include <stdio.h>
```

```c
#include <string.h>
#include <stdlib.h>
#include <time.h>

typedef unsigned int WORD;
#define w 32 /* word size in bits */
#define r 12 /* number of rounds*/
#define b 16 /* number of bytes in key */
#define c 4  /* number words in key = ceil(8*b/w)*/
#define t 26 /* size of table S = 2*(r+1) words */

WORD S[t];
WORD P = 0xb7e15163, Q = 0x9e3779b9;
#define ROTL(x, y) (((x) << (y & (w - 1))) | ((x) >> (w - (y &
(w - 1)))))
#define ROTR(x, y) (((x) >> (y & (w - 1))) | ((x) << (w - (y &
(w - 1)))))

void RC5_ENCRYPT(WORD *pt, WORD *ct) /* 2 WORD input pt/output
ct */
{
   WORD i, A = pt[0] + S[0], B = pt[1] + S[1];
   for (i = 1; i <= r; i++)
   {
       A = ROTL(A ^ B, B) + S[2 * i];
       B = ROTL(B ^ A, A) + S[2 * i + 1];
   }
   ct[0] = A;
   ct[1] = B;
}

void RC5_DECRYPT(WORD *ct, WORD *pt) /* 2 WORD input ct/output
pt */
{
   WORD i, B = ct[1], A = ct[0];
   for (i = r; i > 0; i--)
   {
       B = ROTR(B - S[2 * i + 1], A) ^ A;
       A = ROTR(A - S[2 * i], B) ^ B;
   }
```

```
    pt[1] = B - S[1];
    pt[0] = A - S[0];
}

void RC5_SETUP(unsigned char *K) /* secret input key K[0...b-1]
*/
{
    WORD i, j, k, u = w / 8, A, B, L[c];
    /* Initialize L, then S, then mix key into S */
    for (i = b - 1, L[c - 1] = 0; i != -1; i--)
        L[i / u] = (L[i / u] << 8) + K[i];

    for (S[0] = P, i = 1; i < t; i++)
        S[i] = S[i - 1] + Q;

    for (A = B = i = j = k = 0; k < 3 * t; k++, i = (i + 1) % t,
j = (j + 1) % c) /* 3*t > 3*c */
    {
        A = S[i] = ROTL(S[i] + (A + B), 3);
        B = L[j] = ROTL(L[j] + (A + B), (A + B));
    }
}

void main()
{
    double time_spent = -0.0;
    clock_t begin = clock();

    WORD i, j, pt1[2], pt2[2], ct[2] = {0, 0};
    unsigned char key[b];
    if (sizeof(WORD) != 4)
        printf("RC5 error: WORD has %ld bytes.\n", sizeof(WORD));
    printf("RC5-32/12/16 encryption and decryption:\n");

    for (i = 0; i < 3; i++)
    {
        if (i == 0)
        {
            pt1[0] = "Hello";
            pt1[1] = "Anchit";
```

```
        }

        if (i == 1)
        {
            pt1[0] = "Hello";
            pt1[1] = "Agarwal";
        }

        if (i == 2)
        {
            pt1[0] = "Hello";
            pt1[1] = "Navdeep";
        }

        for (j = 0; j < b; j++)
            key[j] = rand() % 10;

        RC5_SETUP(key);
        RC5_ENCRYPT(pt1, ct);
        RC5_DECRYPT(ct, pt2);
        printf("\n key = ");

        for (j = 0; j < b; j++)
            printf("%.2X ", key[j]);
          printf("\nEncryption\n plaintext %d %d --- > ciphertext
%.8X %.8X\n", pt1[0], pt1[1], ct[0], ct[1]);
             printf("\nDecryption\n  ciphertext  %.8X  %.8X  --- >
plaintext %d %d\n", ct[0], ct[1], pt2[0], pt2[1]);

        if (pt1[0] != pt2[0] || pt1[1] != pt2[1])
            printf("Decryption Error!");
    }

    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

    printf("\n\tTime taken is: %f seconds \n", time_spent);
}
```

b. Parallel Code

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h>
typedef unsigned int WORD;
#define w 32 /* word size in bits */
#define r 12 /* number of rounds*/
#define b 16 /* number of bytes in key */
#define c 4  /* number words in key = ceil(8*b/w)*/
#define t 26 /* size of table S = 2*(r+1) words */
#define numThreads 1
WORD S[t];
WORD P = 0xb7e15163, Q = 0x9e3779b9;
#define ROTL(x, y) (((x) << (y & (w - 1))) | ((x) >> (w - (y &
(w - 1)))))
#define ROTR(x, y) (((x) >> (y & (w - 1))) | ((x) << (w - (y &
(w - 1)))))
void RC5_ENCRYPT(WORD *pt, WORD *ct) /* 2 WORD input pt/output
ct */
{
   WORD i, A = pt[0] + S[0], B = pt[1] + S[1];
   #pragma section("rc5_encrypt")
   #pragma omp parallel for
   for (i = 1; i <= r; i++)
   {
       A = ROTL(A ^ B, B) + S[2 * i];
       B = ROTL(B ^ A, A) + S[2 * i + 1];
   }
   ct[0] = A;
   ct[1] = B;
}

void RC5_DECRYPT(WORD *ct, WORD *pt) /* 2 WORD input ct/output
pt */
{
   WORD i, B = ct[1], A = ct[0];
   #pragma section("rc5_decrypt")
```

```
    #pragma omp parallel for
    for (i = r; i > 0; i--)
    {
        B = ROTR(B - S[2 * i + 1], A) ^ A;
        A = ROTR(A - S[2 * i], B) ^ B;
    }
    pt[1] = B - S[1];
    pt[0] = A - S[0];
}
void RC5_SETUP(unsigned char *K) /* secret input key K[0...b-1]
*/
{
    #pragma omp parallel num_threads(numThreads)
    {
        #pragma omp
        WORD i, j, k, u = w / 8, A, B, L[c];
        /* Initialize L, then S, then mix key into S */
        for (i = b - 1, L[c - 1] = 0; i != -1; i--)
            L[i / u] = (L[i / u] << 8) + K[i];

        #pragma omp for
        for (S[0] = P, i = 1; i < t; i++)
            S[i] = S[i - 1] + Q;

        for (A = B = i = j = k = 0; k < 3 * t; k++, i = (i + 1) %
t, j = (j + 1) % c) /* 3*t > 3*c */
        {
            A = S[i] = ROTL(S[i] + (A + B), 3);
            B = L[j] = ROTL(L[j] + (A + B), (A + B));
        }
    }

}

void main()
{
    double t1 = omp_get_wtime();
    WORD i, j, pt1[2], pt2[2], ct[2] = {0, 0};
    unsigned char key[b];
    if (sizeof(WORD) != 4)
```

```
        printf("RC5 error: WORD has %ld bytes.\n", sizeof(WORD));
    printf("RC5-32/12/16 encryption and decryption:\n");
#pragma omp parallel num_threads(1)
    {
#pragma omp
        for (i = 0; i < 3; i++)
        {
            if (i == 0)
            {
                pt1[0] = "Hello";
                pt1[1] = "Anchit";
            }
            if (i == 1)
            {
                pt1[0] = "Hello";
                pt1[1] = "Agarwal";
            }
            if (i == 2)
            {
                pt1[0] = "Hello";
                pt1[1] = "Navdeep";
            }
            for (j = 0; j < b; j++)
                key[j] = rand() % 10;
            RC5_SETUP(key);
            RC5_ENCRYPT(pt1, ct);
            RC5_DECRYPT(ct, pt2);
            printf("\n key = ");
            for (j = 0; j < b; j++)
                printf("%.2X ", key[j]);
                  printf("\nEncryption\n plaintext %d %d --- >
ciphertext %.8X %.8X\n", pt1[0], pt1[1], ct[0], ct[1]);
               printf("\nDecryption\n ciphertext %.8X %.8X --- >
plaintext %d %d\n", ct[0], ct[1], pt2[0], pt2[1]);
            if (pt1[0] != pt2[0] || pt1[1] != pt2[1])
                printf("Decryption Error!");
        }
    }
    double t2 = omp_get_wtime();
    printf("\n\tTime taken is: %f seconds", (t2 - t1));
```

```
    printf("\n\n\n");
}
```

5.3. Test cases

5.3.1. RSA

Not all numbers are getting encrypted and decrypted properly with the help of our code. For the execution, we will be encrypting numbers like 16, 12, 20, etc.

5.3.2. RC5

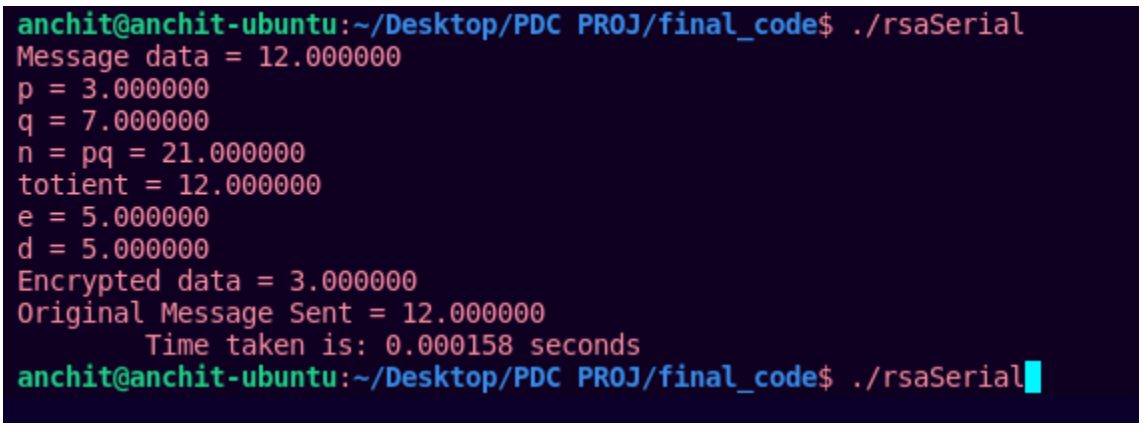
Using the RC5 algorithm we have encrypted the plain text “Hello Anchit”, “Hello Agarwal”, and “Hello Navdeep”. These are the 3 character arrays that we will be encrypting.

6. Output and Performance Analysis

6.1. Execution snapshots

6.1.1. RSA

a. Serial Code



b. Parallel Code

```
anchit@anchit-ubuntu:~/Desktop/PDC PROJ/final_code$ ./rsa
Message data = 12.000000
p = 3.000000
q = 7.000000
n = pq = 21.000000
totient = 12.000000
e = 5.000000
d = 5.000000
Encrypted data = 3.000000
Original Message Sent = 12.000000
Time taken is: 0.000043 seconds


anchit@anchit-ubuntu:~/Desktop/PDC PROJ/final_code$
```

6.1.2. RC5

a. Serial Code

```
anchit@anchit-ubuntu:~/Desktop/PDC PROJ/final_code$ ./series
RC5-32/12/16 encryption and decryption:

 key = 03 06 07 05 03 05 06 02 09 01 02 07 00 09 03 06
Encryption
 plaintext 119484464 119484470 --- > ciphertext D48B93EC 6B6D7DD6

Decryption
 ciphertext D48B93EC 6B6D7DD6 --- > plaintext 119484464 119484470

 key = 00 06 02 06 01 08 07 09 02 00 02 03 07 05 09 02
Encryption
 plaintext 119484464 119484476 --- > ciphertext 662E16D8 2783C880

Decryption
 ciphertext 662E16D8 2783C880 --- > plaintext 119484464 119484476

 key = 02 08 09 07 03 06 01 02 09 03 01 09 04 07 08 04
Encryption
 plaintext 119484464 119484484 --- > ciphertext 442F0983 EECDD7B4

Decryption
 ciphertext 442F0983 EECDD7B4 --- > plaintext 119484464 119484484

        Time taken is: 0.000237 seconds
```

b. Parallel Code

```
anchit@anchit-ubuntu:~/Desktop/PDC PROJ/final_code$ ./rc5test
RC5-32/12/16 encryption and decryption:

 key = 03 06 07 05 03 05 06 02 09 01 02 07 00 09 03 06
Encryption
 plaintext 21467369 21467392 --- > ciphertext CDA5E9DB D98E68F1

Decryption
 ciphertext CDA5E9DB D98E68F1 --- > plaintext 21467369 21467392

 key = 00 06 02 06 01 08 07 09 02 00 02 03 07 05 09 02
Encryption
 plaintext 21467369 21467384 --- > ciphertext 64CAF6B7 B02645E0

Decryption
 ciphertext 64CAF6B7 B02645E0 --- > plaintext 21467369 21467384

 key = 02 08 09 07 03 06 01 02 09 03 01 09 04 07 08 04
Encryption
 plaintext 21467369 21467375 --- > ciphertext 866CF70D EA4B3E17

Decryption
 ciphertext 866CF70D EA4B3E17 --- > plaintext 21467369 21467375

        Time taken is: 0.000062 seconds


anchit@anchit-ubuntu:~/Desktop/PDC PROJ/final_code$
```

6.2. Performance comparison with existing works

6.2.1. RSA

Serial implementation of it took **0.000158 seconds**.

Parallel implementation of it took **0.000043 seconds**.

Hence, a time **decrease of 72.78481%.**

6.2.2. RC5

Serial implementation of it took **0.000237 seconds**.

Parallel implementation of it took **0.000062 seconds**.

Hence, a time **decrease of 73.8396%.**

7. Conclusion and Future Directions

Advantages of parallelization

· Less execution times

· Efficient code

· Distribution of instructions to threads automated

Disadvantages

· Overhead cost outweighs the time saved in case of small strings

· Memory usage multiplies by the number of threads/processors used

· Weight of performance on processor increases exponentially with the increase in number of processors

Further Study

With the hope of better parallelization algorithms and improvement in the encryption algorithms, this process can be accelerated much faster than its usual speed. Data dependency among the steps must be made a little less complicated so that they are easy to be modularised into various components. Certain algorithms must be developed that actually parallelize the core components of these algorithms.

8. References

[1] Sandipan Basu, "INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA) – A TYPICAL ILLUSTRATION" *Journal of Global Research in Computer Science*, July 2011

[2] Xin Zhou, Xiaofei Tang , "Research and implementation of RSA algorithm for encryption and decryption", *IEEE*, August 2011

[3] Nawal Alsaffar, Wael Elmedany, Hayat Ali, " Application of RC5 for IoT devices in Smart Transportation System", *IEEE, October 2019*

[4] ROMANA SHAHZADI, SYED MUHAMMAD ANWAR, FARHAN QAMAR, "Chaos Based Enhanced RC5 Algorithm for Security and Integrity of Clinical Images in Remote Health Monitoring", *IEEE,* April 2019

[5] Wei Wang, Xinming Huang, "A Novel and Efficient Design for RSA Cryptosystem with Very Large Key Size", *IEEE,* October 2015

[6] Kannan Balasubramanian, "Variants of RSA and their cryptanalysis", *IEEE,* December 2014