Universidad Del Valle de Guatemala
Computación Paralela
Miguel Novella

# Proyecto # 2

29/04/2022
Augusto Alonso #181085
Esteban Cabrera #17781
Mario Sarmientos #17055

# Antecedentes

# DES

DES o Data Encryption Standard es un algoritmo de cifrado simétrico, es un prototipo del cifrado por bloques. Se basa en la estructura de red de Feistel que tiene 16 rondas. Utiliza una clave criptográfica para modificar la transformación, así el descifrado se realiza solo para quienes tienen la llave. El proceso de criptografía se basa en modelos matemáticos que al ser computables requieren de recursos para desarrollarlo. La data que se encripta se convierte en un cifrado, el cual muestra la misma de una manera ilegible para su protección. El proceso de descifrado es todo lo contrario, sin embargo, se requiere la llave mencionada anteriormente. DES se estableció en el año 1977, constituyéndose de nuevo en el año 1993 para mantener así una normativa a nivel federal en Estados Unidos. Sin embargo, se convirtió en un estándar mundial.

**Algoritmo**

El algoritmo está diseñado para cifrar y descifrar bloques de datos que consisten en 64 bits bajo control de una clave de 64 bits. El descifrado debe realizarse utilizando la misma clave que para el cifrado, pero con el programa de direccionamiento de los bits clave alterado para que el proceso de descifrado sea el inverso de el proceso de cifrado. Un bloque a cifrar se somete a una permutación inicial IP, luego a un cálculo complejo dependiente de la clave y finalmente a una permutación que es la inversa de la permutación inicial IP'-1. El cálculo dependiente de la clave se puede definir simplemente en términos de una función / llamada función de cifrado, y una función KS, llamada programa clave. Una descripción primero proporciona el cálculo, junto con detalles sobre cómo se utiliza el algoritmo para el cifrado. A continuación, se describe el uso del algoritmo para el desciframiento.Finalmente, una definición del cifrado se da en términos de funciones primitivas que se denominan funciones de selección S, y la función f de permutación-

**Aplicaciones**

El cifrado de datos se utiliza en diversas aplicaciones y entornos. La utilización específica del cifrado y la implementación del DES se basarán en muchos factores particulares al sistema informático y sus componentes asociados. En general, la criptografía se utiliza para proteger los datos mientras está siendo comunicado entre dos puntos o mientras está almacenado en un medio vulnerable al robo físico. La seguridad de las comunicaciones brinda protección a los

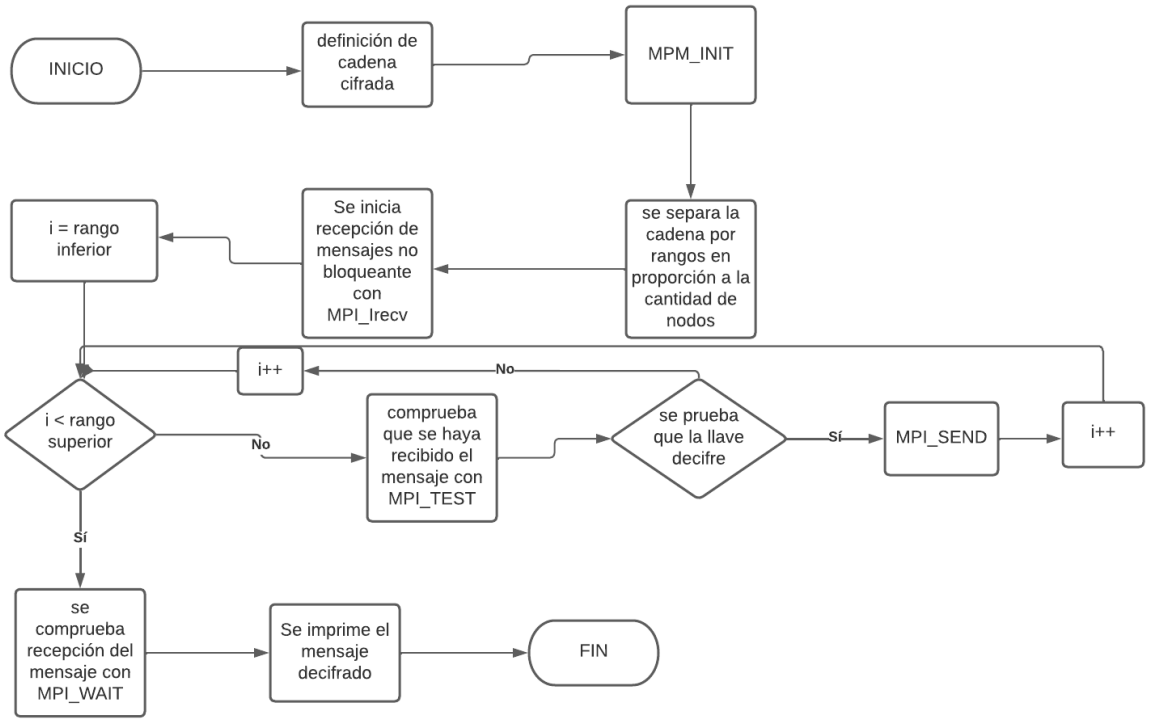
datos al cifrarlos en el punto de transmisión y descifrarlos en el punto de recepción. La seguridad de archivos brinda protección a los datos cifrándolos cuando se registran en un medio de almacenamiento y descifrando cuando se vuelve a leer desde el medio de almacenamiento. En el primer caso, la clave debe estar disponible en el transmisor y el receptor simultáneamente durante la comunicación. En el segundo caso, la clave debe mantenerse y estar accesible durante el período de almacenamiento.

**Implementaciones**

Las implementaciones que se le han dado a los modelos de encriptación han sido con mayor frecuencia para la seguridad informática. Se pretende utilizarlos para proteger información sensible que se pueda encontrar en cualquier sistema informático o red. En la cultura informática actual ya se ha convertido en un estándar de manejo de datos debido a la importancia de mantener la seguridad de la misma. Dentro de instituciones o compañías importantes, la encriptación de los datos es obligatoria debido a la ética y legalidad que deben mantener para ser confiables. Estos establecimientos usan la encripción de datos para múltiples propósitos como:

- Manejo de información sensible
    - Credenciales
    - Datos demográficos de personas
    - Información bancaria o económica
    - Información confidencial gubernamental o empresarial
    - Material audiovisual privado
- Protocolos de envío de información en una red
    - Para que la información sea verídica.
    - Que sea envíada o recibida por el destinatario o receptor correcto.

# Diagrama de flujo




Una vez funcionando su programa base, explique mediante diagramas como funcionan las rutinas:

a. decrypt (key, *ciph, len) y encrypt (key, *ciph, len)

Decrypt es un método que recibe la llave a utilizar, el mensaje cifrado como siguiente parámetro y por último, la longitud de la cadena recibida. Su función es desencriptar un mensaje.

Ecrypt es un método que recibe la llave a utilizar, el mensaje cifrado como siguiente parámetro y por último, la longitud de la cadena recibida. Su función es encriptar un mensaje.

b. tryKey (key, *ciph, len)

Recibe la llave, el mensaje cifrado y la longitud del mensaje. Prueba si la llave es válida para poder descifrar el mensaje.

c. memcpy

Copia los valores de num bytes desde la ubicación a la que apunta el origen directamente al bloque de memoria al que apunta el destino. El resultado es una copia binaria de los datos.

d. strstr

Devuelve un puntero a la primera aparición de str2 en str1, o un puntero nulo si str2 no forma parte de str1. El proceso de coincidencia no incluye los caracteres nulos finales, pero se detiene ahí.

## Describa el uso y flujo de comunicación de las primitivas de MPI:

a. **MPI_Irecv:** Esta función es para comenzar el recibimiento de un mensaje. Lo que hace es bloquear el proceso hasta que se le notifique la llegada de un mensaje. Cuando esto suceda, pedirá que se comience a recibir el mensaje, a la vez que continúa la ejecución del resto del proceso.

b. **MPI_Send:** Función de envío de mensaje bloqueante de un proceso de origen a uno de destino. Al ser bloqueante significa que hasta que el mensaje no haya sido enviado (que salga del buffer de salida) no se continúa la ejecución.

c. **MPI_Wait:** Este método bloquea el proceso que lo invoca hasta que la operación indicada en request se complete.

## Cronograma de actividades

| Día / Task | Implementacion de MPI | Paralelizacion en los metodos | Mediciones de tiempos nuevos | Pruebas de cifrado y decifrado | Trabajo en documento | Trabajo en documento |
|---|---|---|---|---|---|---|
| lunes | | | | | | |
| martes | | | | | | |
| miércoles | | | | | | |
| jueves | | | | | | |
| viernes | | | | | | |

## Problema con la distribución actual de las keys en los procesos

1. **Una llave fácil de encontrar, por ejemplo, con valor de (2^56) / 2  + 1**
   **0.04 segundos**

2. **Una llave medianamente difícil de encontrar, por ejemplo, con valor de (2^56) / 2 + (2^56) / 8**
   **4 horas**

3. **Una llave difícil de encontrar, por ejemplo, con valor de (2^56) / 7 + (2^56) / 13 aproximados al entero superior**

   **Nunca termino**

# Primeras Mediciones

## Formato

Texto Cifrado: "operating systems is fun"
Palabra clave: systems
Key: 20000000L
Número de cores a utilizar: 4

### Mediciones de el código secuencial

```
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ ./build/sec

Key encontrada: 20000000
Texto cifrado operating systems is fun
That took 8.538701 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ ./build/sec

Key encontrada: 20000000
Texto cifrado operating systems is fun
That took 8.496298 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ ./build/sec

Key encontrada: 20000000
Texto cifrado operating systems is fun
That took 8.538808 seconds
```

### Mediciones de el código en bloques de 2^56/N

Es el formato original que se tenía en el código con el único cambio que se utiliza MPI_Test para cada 250 iteraciones quitando un poco del overhead.

```
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto

20000000 operating systems is fun
That took 10.459021 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto

20000000 operating systems is fun
That took 10.168583 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto

20000000 operating systems is fun
That took 10.457465 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$
```

### Mediciones de el código como si fuera con un schedule static 4 de omp

Se va recorriendo de 4 en 4 cada thread.

```
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto_block

Key encontrada: 20000000
Valor del texto: operating systems is fun
That took 2.322079 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto_block

Key encontrada: 20000000
Valor del texto: operating systems is fun
That took 2.380352 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto_block

Key encontrada: 20000000
Valor del texto: operating systems is fun
That took 2.618224 seconds
```

## Formato

Con este formato se tendra
Texto Cifrado: "operating systems is fun"
Palabra clave: systems
Key:18014398509482484 (más cercano a 2^56/4) = 2^56/4
Número de cores a utilizar: 4

### Mediciones de el código secuencial

```
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ ./build/sec
^C
```

Luego de estar corriendo 3 minutos aún no había terminado

### Mediciones de el código en bloques de 2^56/N

```
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto

18014398509482484 operating systems is fun
That took 0.000698 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto

18014398509482484 operating systems is fun
That took 0.000344 seconds
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto

18014398509482484 operating systems is fun
That took 0.000478 seconds
```

### Mediciones de el código como si fuera con un schedule static 4 de omp

```
augusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$ mpirun -np 4 ./build/proyecto_block
^Caugusto@augusto-VirtualBox:~/proyecto2-computacion-paralela$
```

Mediciones de el código como si fuera con un schedule static 2 de omp

```
augusto@augusto-VirtualBox:~/shared/proyecto2-computacion-paralela$ mpirun -np 2
 ./build/proyecto_block

Key encontrada: 10000000
Valor del texto: operating systems is fun
That took 2.010035 seconds
augusto@augusto-VirtualBox:~/shared/proyecto2-computacion-paralela$ mpirun -np 2
 ./build/proyecto_block

Key encontrada: 10000000
Valor del texto: operating systems is fun
That took 1.998090 seconds
augusto@augusto-VirtualBox:~/shared/proyecto2-computacion-paralela$ mpirun -np 2
 ./build/proyecto_block

Key encontrada: 10000000
Valor del texto: operating systems is fun
That took 2.051622 seconds
```

```
augusto@augusto-VirtualBox:~/shared/proyecto2-computacion-paralela$ mpirun -np 2
 ./build/proyecto_block

Key encontrada: 10000000
Valor del texto: operating systems is fun
That took 1.962123 seconds
augusto@augusto-VirtualBox:~/shared/proyecto2-computacion-paralela$ mpirun -np 2
 ./build/proyecto_block

Key encontrada: 10000000
Valor del texto: operating systems is fun
That took 2.102561 seconds
augusto@augusto-VirtualBox:~/shared/proyecto2-computacion-paralela$ mpirun -np 2
 ./build/proyecto_block

Key encontrada: 10000000
Valor del texto: operating systems is fun
That took 2.039705 seconds
```

## Primeras Impresiones

Sin sacar aún speedups notamos que de primero con un número menor que 2^56/4 el formato donde se recorre como si fuera static es más rápido vs el secuencial y el primer formato. Notamos el tiempo de recorrer de 1-2^56/4 el primer bloque y el segundo 2^56/4-2^56/2 y así consecuentemente fue hasta más lento que el secuencial tardando aprox 10 segundos vs los 8 del secuencial vs los 2.32. Mientras que al correr con una key significativamente mucho más grande al segundo thread iniciar en 2^56/4 prácticamente encontró el key de manera inmediata. Mientras que los otros dos no lo encontraron ni pasando minutos enteros dándonos que fuera un speed up gigante pero engañoso ya que no será consistente.

# Mediciones

Tabla 1: Tiempos con key = 10000000 L y 4 cores

| CON Key = 10000000L | | Tiempo en segundos |
|---|---|---|
| **Secuencial** | **Tiempo schedule static 4 (compartamiento)** | **Tiempo 4 blocks** |
| 14.5495 | 2.514771 | 13.012125 |
| 12.824756 | 2.5765 | 10.835046 |
| 12.658079 | 2.32775 | 8.247481 |
| 7.432927 | 2.368534 | 7.2021 |
| 5.658958 | 2.517491 | 9.521566 |
| 6.559268 | 2.747923 | 6.657698 |
| 11.166622 | 2.267725 | 6.760089 |
| 8.322659 | 2.585728 | 5.88873 |
| 12.811318 | 2.623135 | 17.45175 |
| 11.692481 | 2.683424 | 13.334573 |
| 10.3676568 | 2.5212981 | 9.8911158 |

Tabla 2: Speedup y eficiencia con key = 10000000 L y 4 cores

| | SpeedUp | Efficency |
|---|---|---|
| Tiempo 4 cores Scheduled static = 4 | 4.112031338 | 1.028007835 |
| Tiempo 4 cores | 1.04817869 | 0.08734822415 |

Tabla 3: Tiempos con key = 10000000 L y 2 cores

| CON Key = 10000000L | | Tiempo en segundos |
|---|---|---|
| **Secuencial** | **Tiempo schedule static 2 (compartamiento)** | **Tiempo 2 blocks** |
| 14.5495 | 2,01 | 18,30352448 |
| 12.824756 | 1,99 | 15,63807648 |
| 12.658079 | 2,05 | 17,21306304 |
| 7.432927 | 1,99 | 17,67960670 |
| 5.658958 | 2,09 | 9,67501335 |
| 6.559268 | 1,97 | 8,80944582 |
| 11.166622 | 2,03 | 9,09955575 |
| 8.322659 | 1,96 | 15,26273165 |
| 12.811318 | 1,96 | 10,80167684 |
| 11.692481 | 2,10 | 9,72159193 |
| 10.3676568 | 2,015 | 13,2204286 |

Tabla 4: Speedup y eficiencia con key = 10000000 L y 2 cores

| | SpeedUp | Efficency |
|---|---|---|
| Tiempo 2 cores Scheduled static = 2 | 6,56100675 | 3,280503375 |
| Tiempo 2 cores | 1,27516071 | 0,6375803548 |

## Discusión

Al ver el flujo del código notamos las utilidades que tienen las funciones como "MPI_Irecv" que nos permite recibir un valor sin parar la ejecución y como se complementa con "MPI_Test" al entender este flujo procedimos a quitar que se verificará si ya se recibio el valor (con MPI_Test) en cada iteración y en lugar de eso revisamos cada 250 iteraciones en los threads para quitar un poco de overhead y vimos una mejora en el código en un segundo aproximadamente. Pero nos topamos con el siguiente problema: el código es 100% dependiente de su key. Es decir el key determina que tanto se tardará el código en resolver el cifrado y no de una manera consistente es decir en nuestro flujo de pruebas que se mantuvo corriendo con 4 cores siempre que se escogiera una key dentro de 2^56/4-1 para abajo no representaba prácticamente ni un speed up es decir ¼ de nuestras keys no tendrá speed up con este código pero otras como (3*2^56)/4 representan que se encuentran prácticamente de manera inmediata teniendo un speed up absurdo.
Con esto mencionado nuestro primer approach fue como quitar esa dependencia a la key y se optó por escoger una key de manera aleatoria sin repetición ya que con repetición representa que cada intento tendría una oportunidad de 1/2^56 lo cúal nos podría tardar para siempre, se considero tener un array con todas las keys y hacer un shuffle en el array pero por el rango de las keys no hay suficiente memoría dejándonos con la segunda posibilidad imitar el comportamiento o de OpenMP schedule static con un bloque más pequeño. Con este approach se tomó como valor de bloque 4 y se observó que siempre hay un speed up sobre el código secuencial más bien no siempre tiene mejor performance que recorriendo en 4 bloques iguales para cada hijo.

De hecho únicamente la mitad de keys tendrían un mejor speed up vs el código actual entonces ¿por qué elegirlo? Se eligió esta última opción por su consistencia de siempre mantener un speed up más "saludable" no teniendo picos y considerando que sin importar la key siempre será mejor opción que encontrarlo que de manera secuencial como observamos en la tabla 2. donde el speed up fue de 4 lo cual tiene sentido por los 4 cores con los que se corre teniendo una eficiencia perfecta prácticamente mientras que por la naturaleza de la key con el código paralelo por bloques notamos que no hubo speed up con una pésima eficiencia. Pero si miramos en las primeras mediciones cuando escogimos una key más cercano a 2^56/4 notamos que el speedup de este código sería con un valor irreal ya que ni el secuencial ni el paralelo con un tipo de schedule como el estático de OpenMP  no terminaron en un tiempo razonable.

Como se puede observar en la tabla 3, en donde se hacen las 10 mediciones pero con 2 núcleos, si aumenta el tiempo de ejecución a comparación de las ejecuciones con 4 núcleos. Esto se debe a que no se tiene la misma cantidad de recursos para poder computar los procesos emergentes. Sin embargo, sigue siendo de manera paralela y eso permite que los tiempos de ejecución sean más bajos aún que los tiempos de las ejecuciones en secuencial. Básicamente se puede observar que a mayor núcleos, el cifrado o descifrado de una cadena de texto será más rápido. Sin embargo, la eficiencia disminuirá debido a que la cantidad de núcleos es directamente proporcional al tiempo de ejecución en paralelo y su resultante es inversamente proporcional al tiempo de ejecución secuencial.

Al intentar probar con el código de otros compañeros nos encontramos con los siguientes retos. Al utilizar la mayoría keys de strings no se sabe realmente qué tan grande es teniendo que pasar horas para cada medición se intentó hacer una medición y luego de más de una hora no se había terminado además que la utilización de diferentes librerías afecto y la manera de guardar el texto cifrado optando nosotros por guardarlo como sea retornado por el algoritmo mientras que se vio que algunos grupos decidieron guardarlo en un formato HEX decidiendo mejor tomar speed ups con keys en un dato de tipo LONG controlando así nosotros que tan tardados pueden ser los tiempos. Se sugiere para futuros grupos intentando el proyecto hablar con la clase para que todos tengan un estándar de que tamaño de key utilizar y que librería utilizar.

**Conclusiones**

- La cantidad de núcleos en el cifrado y descifrado afecta positivamente los tiempos de ejecución del programa en paralelo
- Para la implementación de este algoritmo la separación de tareas por bloques no resultó tan efectiva, aún así, se mantiene por debajo de los tiempos secuenciales.
- Los diferentes tipos de datos asignados para los keys pueden variar el tiempo de ejecución de los programas.

**Bibliografía**

U.S. Department of Commerce. (1993). Federal Information Processing Standards Publication: data encryption standard (DES). Extraído de: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub46-2.pdf

Biryukov, A. (2011). Data Encryption Standard. Extraído de: https://scholar.google.com/scholar?hl=es&as_sdt=0%2C5&q=DES+encryption&oq=des+e

Coppersmith, D. (1994). The Data Encryption Standard (DES) and its strength against attacks. Extraído de: https://ieeexplore.ieee.org/abstract/document/5389567