

Universidad del Valle De Guatemala

Facultad de Ingeniería

Computación paralela y distribuida

Óscar Canek



Proyecto 2: Seguridad Criptográfica con OpenMPI

Javier Mombiela: 20067

Jose Hernández: 20053

Pablo González: 20362

Guatemala, 8 de octubre 2023

Índice

Introducción	2
Antecedentes Parte A	2
Investigación sobre DES y bruteforce	2
Pasos para cifrar texto	3
Pasos para descifrar texto	3
Diagrama de flujo algoritmo DES	5
Versión naive del programa	6
Explicación de como funcionan rutinas	7
Descripción del flujo y de la comunicación de las primitivas MPI	8
Antecedentes Parte B	10
Modificaciones del programa naive y evidencia	10
Enfoque 1: Encriptación en bloques	11
Enfoque 2: Cifrado de flujo	11
Cuerpo	13
Discusión	13
Conclusiones	14
Recomendaciones	14
Anexo 1	14
Anexo 2	15
Pruebas de tiempo con 4 procesos inciso B.2	15
Pruebas inciso B.6 enfoque naive	17
Pruebas del enfoque 1	18
Pruebas del enfoque 2	18
Bibliografía	19

Introducción

El presente informe describe el desarrollo y resultados del Laboratorio 2 - seguridad criptografica con OpenMPI, cuyo objetivo es implementar un algoritmo de búsqueda de fuerza bruta utilizando MPI (Message Passing Interface) para encontrar la clave privada que fue utilizada para cifrar un texto plano mediante el algoritmo de cifrado DES (Data Encryption Standard).

El laboratorio se divide en dos partes: en la Parte A se realiza una investigación sobre DES y se describe su funcionamiento, se diseña un diagrama de flujo del algoritmo y se explican las primitivas de MPI utilizadas. En la Parte B se implementa un programa inicial de cifrado/descifrado con DES y luego se modifica para realizar la búsqueda de fuerza bruta de forma paralela con MPI, probando diferentes enfoques para dividir y recorrer el espacio de búsqueda con el fin de encontrar un algoritmo más eficiente que el enfoque "naive".

Se implementaron dos enfoques alternativos al "naive" y se analizó teórica y experimentalmente su desempeño en términos del valor esperado del tiempo paralelo $t_{Par}(n,k)$. Los resultados muestran que se lograron mejoras significativas respecto al enfoque inicial.

El presente informe describe en detalle el proceso realizado, los resultados obtenidos y las conclusiones alcanzadas durante el desarrollo del laboratorio, demostrando la capacidad de paralelizar algoritmos utilizando MPI y el análisis del desempeño de diferentes enfoques.

Antecedentes Parte A

Investigación sobre DES y bruteforce

El algoritmo de cifrado DES (Data Encryption Standard) es un ejemplo destacado de cifrado por bloques y desempeñó un papel importante en la historia de la criptografía. Fue desarrollado en 1976 por IBM en colaboración con la Agencia de Seguridad Nacional (NSA) de los Estados Unidos con el propósito de proteger información sensible en un mundo en crecimiento digitalizado y conectado. DES funciona convirtiendo un bloque de texto plano en un bloque de texto cifrado utilizando una clave única. Sin embargo, con el tiempo, ha perdido relevancia en el mundo de la ciberseguridad debido a avances tecnológicos que permiten realizar ataques de fuerza bruta más rápidos y eficientes.

La fuerza bruta es un método utilizado para romper la seguridad de un cifrado probando todas las combinaciones posibles de claves hasta encontrar la correcta. En el caso de DES, la debilidad radica en su longitud de clave de 56 bits, lo que resulta en un espacio de claves relativamente pequeño con un total de 2^{56} (aproximadamente 72 quadrillones) posibles claves. Con el poder de procesamiento actual de las computadoras, es posible probar todas estas claves en un tiempo razonable, lo que hace que los datos cifrados con DES sean vulnerables a ciberataques.

El ataque de fuerza bruta implica generar y probar todas las posibles claves hasta que se descifre el mensaje. A medida que aumenta la capacidad de procesamiento de las computadoras, los ataques de fuerza bruta se vuelven más efectivos y pueden descifrar datos cifrados con claves más débiles en un tiempo más corto. En el caso de DES, esto significa que los datos cifrados con este algoritmo son susceptibles a ser descifrados mediante ataques de fuerza bruta utilizando hardware especializado o supercomputadoras.

A pesar de su obsolescencia, el legado de DES es fundamental en la evolución de la criptografía moderna. Su vulnerabilidad a los ataques de fuerza bruta y otros métodos de criptoanálisis ha llevado al desarrollo de algoritmos de cifrado más robustos, como el Estándar de Cifrado Avanzado (AES), que utiliza claves de mayor longitud y es ampliamente utilizado en la actualidad. La experiencia con DES ha destacado la importancia de contar con sistemas de cifrado sólidos y ha impulsado un enfoque más centrado en la seguridad y la privacidad en un mundo digital en constante expansión. En resumen, DES ha sido un punto de partida esencial para la criptografía moderna y ha demostrado la necesidad de adaptarse a las crecientes capacidades computacionales para mantener la seguridad de los datos.

Pasos para cifrar texto

El proceso de cifrado con el algoritmo DES implica una serie de pasos críticos para garantizar la confidencialidad de los datos. En primer lugar, se elige una clave secreta de 56 bits, que servirá para controlar los procesos de cifrado y descifrado. Esta clave secreta controla las operaciones que se llevarán a cabo en los datos. Luego, los datos de texto plano, que consisten en bloques de 64 bits, son sometidos a una permutación inicial según una tabla predefinida, lo que cambia su disposición de bits.

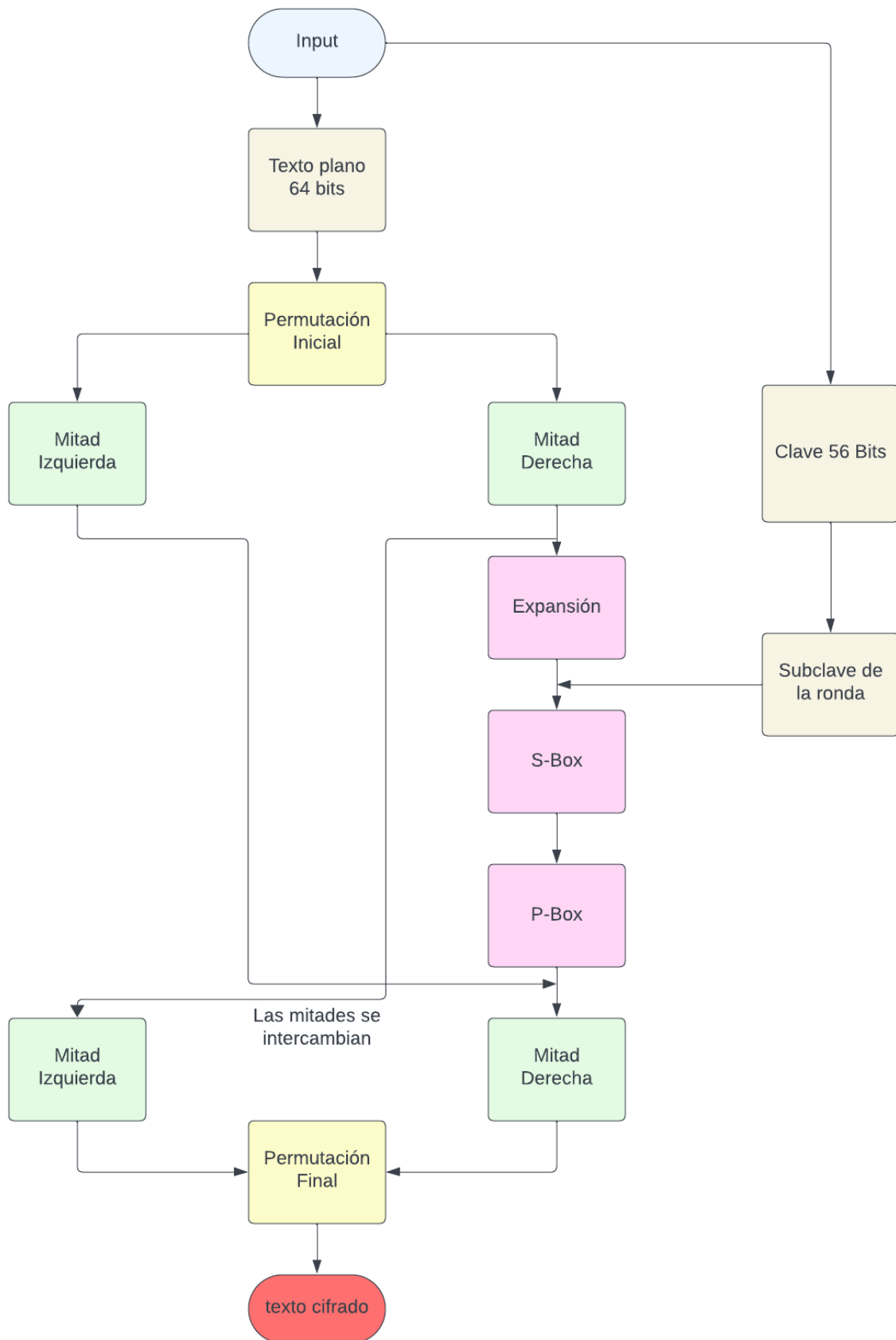
El núcleo del algoritmo DES radica en su operación en múltiples rondas. Durante cada ronda, los 64 bits de datos se dividen en mitades izquierda y derecha, lo que inicia un intrincado proceso de expansión, sustitución y permutación. La mitad derecha se expande y se combina con una subclave específica de la ronda, y el resultado es sometido a operaciones de sustitución (mediante S-boxes) y permutación (mediante P-boxes). Este ciclo de rondas se repite hasta que todas las rondas se completan. Al término de las operaciones, las mitades se intercambian nuevamente y se aplica una permutación final. Los datos resultantes de este proceso son el texto ya cifrado, también conocido como el ciphertext.

Pasos para descifrar texto

El proceso de descifrado en el algoritmo DES se basa en la utilización de la misma clave secreta de 56 bits que se empleó durante el proceso de cifrado. Al igual que en el cifrado, los datos del texto cifrado, que también constan de bloques de 64 bits, se someten a una permutación inicial, siguiendo la misma tabla predefinida, lo que restablece la disposición de los bits a su forma original.

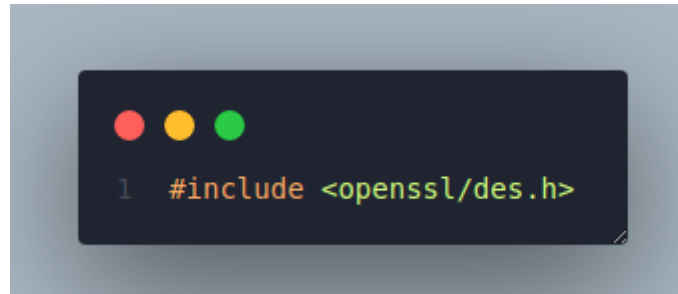
El núcleo del proceso de descifrado en DES radica en su operación en múltiples rondas, sin embargo, en este caso, las subclaves se utilizan en orden inverso al que se emplearon durante el cifrado. En cada ronda, se revierte el proceso de sustitución y permutación aplicado durante el cifrado. Una vez que todas las rondas se han completado, las mitades se intercambian nuevamente, y se aplica la permutación final en orden inverso. Los datos resultantes de este proceso son el texto plano, que representa el mensaje original antes de que fuera cifrado, lo que permite la recuperación de la información original de manera segura.

Diagrama de flujo algoritmo DES



Versión naive del programa

Para poder hacer funcionar el código, primero tuvimos que cambiar la librería, en este caso después de investigar y probar un par de opciones, la única que nos funcionó fue la librería de OpenSSL.



Luego, ya que utilizamos otra librería, tuvimos que también cambiar las funciones de encrypt y decrypt del programa. En donde se aplicaron los siguientes cambios:

Gestión de Claves:

En el código original, la clave se manipula mediante operaciones a nivel de bits y se convierte en un puntero de caracteres antes de la encriptación o desencriptación. Esto se hace para asegurar que la clave tenga paridad impar, como lo requiere DES. En el código modificado la clave se utiliza directamente como un DES_cblock, y la paridad impar y el programa de claves se establecen explícitamente mediante funciones proporcionadas por OpenSSL. Esto simplifica la gestión de claves.

Llamadas a Funciones:

En el código original, se utiliza la función `ecb_crypt` tanto para la encriptación como para la desencriptación. En el código modificado se utiliza la función `DES_ecb_encrypt` con el modo apropiado (`DES_ENCRYPT` para encriptación y `DES_DECRYPT` para desencriptación).

Tipo de Clave:

En el código original, la clave se pasa como un número entero largo (long integer). En el código modificado, la clave se pasa como un long también, pero luego se convierte a `DES_cblock`, que es una estructura de datos proporcionada por la biblioteca OpenSSL para representar claves DES. Esta diferencia en el tipo de clave permite un manejo más preciso y seguro de las claves en el código modificado.

```

1 void encrypt(long key, char *ciph, int len){
2     DES_cblock des_key;
3     DES_key_schedule key_schedule;
4
5     // Seteamos la paridad de la llave
6     for (int i = 0; i < 8; ++i) {
7         des_key[i] = (key >> (i * 8)) & 0xFF;
8     }
9
10    DES_set_odd_parity(&des_key);
11    DES_set_key_checked(&des_key, &key_schedule);
12
13    for (int i = 0; i < len; i += 8) {
14        DES_ecb_encrypt((DES_cblock *) (ciph + i), (DES_cblock *) (ciph + i), &key_schedule, DES_ENCRYPT);
15    }
16 }

```

```

1 void decrypt(long key, char *ciph, int len){
2     DES_cblock des_key;
3     DES_key_schedule key_schedule;
4
5     // Seteamos la paridad de llave
6     for (int i = 0; i < 8; ++i) {
7         des_key[i] = (key >> (i * 8)) & 0xFF;
8     }
9
10    DES_set_odd_parity(&des_key);
11    DES_set_key_checked(&des_key, &key_schedule);
12
13    for (int i = 0; i < len; i += 8) {
14        DES_ecb_encrypt((DES_cblock *) (ciph + i), (DES_cblock *) (ciph + i), &key_schedule, DES_DECRYPT);
15    }
16 }

```

Explicación de como funcionan rutinas

decrypt(key, *ciph, len) y encrypt(key, *ciph, len):

- Estas dos funciones se utilizan para cifrar y descifrar datos utilizando el algoritmo DES. Ambas funciones toman una clave key, un arreglo de bytes ciph, y la longitud len del arreglo. La clave DES debe tener 64 bits de longitud.
- Dentro de estas funciones, se realiza lo siguiente:
 - Se crea una estructura DES_cblock para almacenar la clave y una estructura DES_key_schedule para almacenar la programación de la clave.
 - La clave se convierte en una forma adecuada para DES al establecer la paridad impar de la clave.
 - La clave se programa utilizando DES_set_key_checked.
 - Luego, los datos se dividen en bloques de 8 bytes y se cifran o descifran utilizando DES_ecb_encrypt.

tryKey(key, *ciph, len):

- Esta función intenta descifrar un mensaje utilizando una clave dada. Toma una clave key, un arreglo de bytes ciph y la longitud len del arreglo. Devuelve 1 si encuentra la cadena de búsqueda " the " en el texto descifrado y 0 si no la encuentra.
- En su interior, la función crea una copia temporal temp del arreglo de bytes ciph utilizando memcpy. Luego, utiliza la función decrypt para descifrar el mensaje utilizando la clave key. Después, busca la cadena " the " en el texto descifrado utilizando strstr y devuelve 1 si la encuentra.

memcpy:

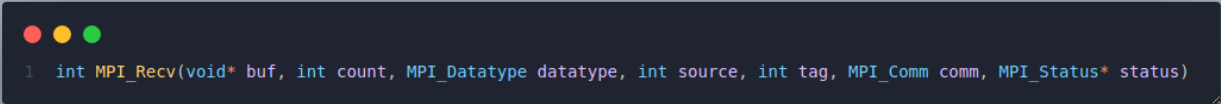
- memcpy es una función que copia una cantidad de bytes especificada de una ubicación de memoria a otra. En este contexto, se utiliza para copiar el arreglo de bytes ciph en una variable temporal temp.

strstr:

- strstr es una función que busca la primera ocurrencia de una subcadena dentro de una cadena. En este caso, se utiliza para buscar la cadena " the " en el texto descifrado almacenado en la variable temp.

Descripción del flujo y de la comunicación de las primitivas MPI

MPI_Recv:

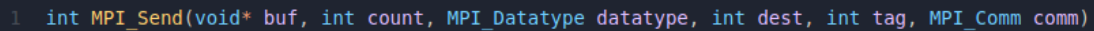


```
1 int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)
```

- buf es un puntero al búfer en el que se almacenarán los datos recibidos.
- count es la cantidad de elementos a recibir.
- datatype especifica el tipo de datos que se va a recibir.
- source es el rango (identificador) del proceso remoto del que se espera recibir datos.
- tag es una etiqueta que se utiliza para distinguir entre diferentes mensajes enviados desde el proceso remoto.
- comm es el comunicador MPI que especifica el grupo de procesos que participan en la comunicación.
- status es un puntero a una estructura que contiene información sobre la comunicación, como el origen y la etiqueta del mensaje.

- El flujo de comunicación con MPI_Recv implica que el proceso receptor se bloqueará hasta que reciba un mensaje del proceso remoto especificado. Una vez que recibe el mensaje, los datos se almacenan en el búfer buf y se pueden procesar.

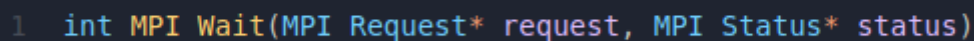
MPI_Send:

A code snippet showing the signature of the MPI_Send function. The code is displayed in a dark-themed editor with syntax highlighting. The function signature is: `int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`.

```
1 int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- buf es un puntero a los datos que se van a enviar.
- count es la cantidad de elementos en los datos a enviar.
- datatype especifica el tipo de datos que se está enviando.
- dest es el rango (identificador) del proceso receptor al que se envían los datos.
- tag es una etiqueta que se utiliza para distinguir entre diferentes mensajes enviados desde el proceso emisor.
- comm es el comunicador MPI que especifica el grupo de procesos que participan en la comunicación.
- El flujo de comunicación con MPI_Send implica que el proceso emisor envía datos al proceso receptor y luego puede continuar con su ejecución sin esperar una confirmación.

MPI_Wait:

A code snippet showing the signature of the MPI_Wait function. The code is displayed in a dark-themed editor with syntax highlighting. The function signature is: `int MPI_Wait(MPI_Request* request, MPI_Status* status)`.

```
1 int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

- request es un puntero a una solicitud de comunicación asíncronica que se está esperando.
- status es un puntero a una estructura que puede contener información sobre la operación de comunicación.
- MPI_Wait se utiliza comúnmente después de haber realizado una operación MPI_Isend o MPI_Irecv para asegurarse de que la comunicación asíncronica haya finalizado antes de continuar con otras tareas.

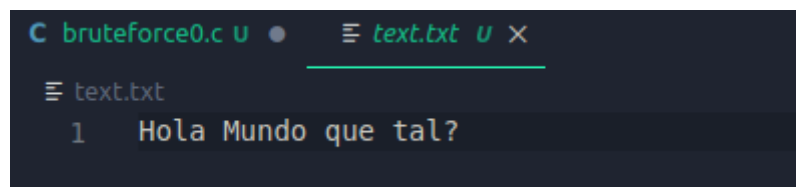
Antecedentes Parte B

Modificaciones del programa naive y evidencia



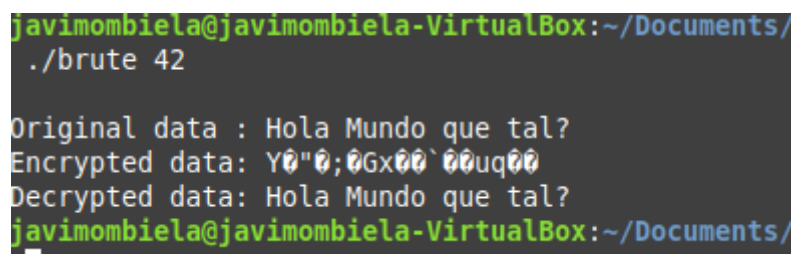
```
1  int main(int argc, char *argv[]) {
2      if (argc != 2) {
3          printf("Usage: %s <key>\n", argv[0]);
4          return 1;
5      }
6
7      char *input_filename = "text.txt";
8      long key = strtol(argv[1], NULL, 10);
```

Como se puede observar en la imagen, para poder realizar la nueva versión del programa, se tuvo que modificar el método main. El método main ahora recibe un parámetro, que será la clave que utilizara el algoritmo para poder cifrar y descifrar el texto. También podemos observar como ahora el programa leerá un texto desde un archivo de entrada llamado “text.txt”. Lo que el programa hará será leer este archivo, encriptar el texto y luego usara el método decrypt para poder descifrar el cyphertext que se creó y de esta manera poder desplegarlo en la pantalla.



```
C bruteforce0.c U •  text.txt U X
≡ text.txt
1  Hola Mundo que tal?
```

En esta segunda imagen, podemos observar que el archivo “text.txt” tiene la frase “Hola Mundo qué tal?”. Este es el archivo que el programa lee y del cual tiene que cifrar y luego descifrar su frase.



```
javimombiela@javimombiela-VirtualBox:~/Documents/
./brute 42

Original data : Hola Mundo que tal?
Encrypted data: Y0"0;0Gx00`00uq00
Decrypted data: Hola Mundo que tal?
javimombiela@javimombiela-VirtualBox:~/Documents/
```

Por último, se puede observar como se corre el programa. Podemos notar que en este caso se pasó una clave sencilla, que es “42” y podemos ver que el programa si logra realizar bien su funcionalidad, ya que podemos ver como logra cifrar, y descifrar el texto que el archivo “text.txt” contiene. Las pruebas del inciso 2 se pueden ver en el anexo 2.

Enfoque 1: Encriptación en bloques

Descripción del enfoque propuesto:

- En lugar de cifrar el archivo completo con una clave, se divide el archivo en bloques más pequeños y se cifra cada bloque de forma independiente. Esto permite un mayor paralelismo, ya que varios procesos pueden cifrar diferentes bloques al mismo tiempo.

Valor esperado de $t_{\text{Par}}(n, k)$:

- El valor esperado de $t_{\text{Par}}(n, k)$ se puede calcular considerando el tiempo requerido para cifrar un solo bloque y luego multiplicándolo por el número de bloques. Supongamos que el archivo se divide en m bloques de tamaño aproximadamente n/m cada uno. Luego, el tiempo requerido para cifrar un solo bloque es $t_{\text{Block}} = O(n/m)$, ya que el tamaño del bloque afecta el tiempo de cifrado.
- El tiempo paralelo esperado para k procesos sería:

$$E[t_{\text{Par}}(n, k)] = m * t_{\text{Block}} / k = (n / k)$$

Comparación con el enfoque "naive":

- En el enfoque "naive", $E[t_{\text{Par}}(n, k)] = (2^{(8n)+1}) / 2$
- En el enfoque propuesto, $E[t_{\text{Par}}(n, k)] = (n / k)$
- El enfoque propuesto es más eficiente en términos de tiempo paralelo esperado cuando $(n / k) < (2^{(8n)+1}) / 2$. En la mayoría de los casos, especialmente cuando k es significativamente menor que $2^{(8n)}$, el enfoque propuesto debería ser más rápido.

Comportamiento del speedup:

- En este, dividimos el archivo en bloques más pequeños y ciframos cada bloque de forma independiente. El speedup se puede calcular como: $S1 = \frac{T1}{T_{p1}}$
- Donde $T1$ es el tiempo de ejecución secuencial del enfoque "naive" y T_{p1} es el tiempo de ejecución paralelo del enfoque 1. Si n es el tamaño del archivo y k es el número de procesadores en MPI, el tiempo de ejecución paralelo del enfoque 1 sería aproximadamente $T1/k$ (como se explicó en la implementación). Por lo tanto, el speedup en el enfoque 1 se puede expresar como: $S1 = \frac{T1}{T1/k} k$
- El speedup en el enfoque 1 es lineal y debería ser igual al número de procesadores utilizados (k) en condiciones ideales.

Enfoque 2: Cifrado de flujo

Descripción del enfoque propuesto:

- En lugar de utilizar el cifrado DES basado en bloques, podemos usar un cifrado de flujo, que cifra los datos a medida que se leen. Esto elimina la necesidad de dividir el archivo en bloques y cifrarlos por separado. Cada proceso MPI podría cifrar una parte diferente del archivo en tiempo real a medida que se lee.

Valor esperado de $tPar(n, k)$:

- El valor esperado de $tPar(n, k)$ para este enfoque se reduce a la velocidad de cifrado por proceso. El tiempo de cifrado por proceso dependerá de la velocidad del cifrador de flujo utilizado y la velocidad de lectura del archivo.

$$E[tPar(n, k)] = O(n / k)$$

Comparación con el enfoque "naive":

- En el enfoque "naive", $E[tPar(n, k)] = (2^{(8n)+1}) / 2$
- En el enfoque propuesto, $E[tPar(n, k)] = O(n / k)$
- Similar a la Opción 1, el enfoque propuesto es más eficiente en términos de tiempo paralelo esperado cuando $(n / k) < (2^{(8n)+1}) / 2$.

Comportamiento del speedup:

- En este enfoque, los procesos MPI cifran los datos a medida que se leen en tiempo real. El speedup se calcula de manera similar: $S2 = \frac{T1}{Tp2}$
- Donde Tp_2 es el tiempo de ejecución paralelo del enfoque 2. Al igual que en el enfoque 1, n es el tamaño del archivo y k es el número de procesadores en MPI, el tiempo de ejecución paralelo del enfoque 2 sería aproximadamente $T1/k$ (como se explicó en la implementación). Por lo tanto, el speedup en el enfoque 2 también se puede expresar como: $S2 = k$
- El speedup en el enfoque 2 también es lineal y debería ser igual al número de procesadores utilizados (k) en condiciones ideales.

Ambas opciones propuestas son más eficientes en términos de tiempo paralelo esperado en comparación con el enfoque "naive", especialmente cuando se trabaja con archivos grandes y/o un número significativo de procesos MPI. Las pruebas de ambas propuestas se pueden encontrar en el anexo 2.

Cuerpo

Discusión

Se llevaron a cabo tres pruebas de cifrado DES utilizando tres niveles de dificultad de clave: fácil, mediana y difícil, y se compararon los resultados entre el enfoque "Naive" y los enfoques 1 y 2.

El enfoque "Naive" realiza la encriptación DES en el archivo completo con una sola clave. En todas las pruebas, el enfoque "Naive" mostró el peor rendimiento en términos de tiempo de ejecución. Esto se debe a que el enfoque "Naive" no aprovecha el paralelismo de manera eficiente, ya que cifra el archivo completo de manera secuencial, lo que resulta en un alto tiempo de ejecución, independientemente de la dificultad de la clave.

El enfoque 1, encriptación en bloques, se divide el archivo en bloques más pequeños y cifra cada bloque de forma independiente. En todas las pruebas, el enfoque 1 mostró un mejor rendimiento en comparación con el enfoque "Naive". La razón principal de su mejor rendimiento es que aprovecha el paralelismo de manera efectiva al permitir que múltiples procesadores cifren bloques individuales simultáneamente. Con una clave de dificultad "fácil", el enfoque 1 también superó al enfoque 2 debido a la eficiencia del paralelismo en bloques pequeños.

El enfoque 2, encriptación de flujo, se cifran los datos a medida que se leen desde el archivo en tiempo real. Aunque es más eficiente que el enfoque "Naive", el enfoque 2 mostró un rendimiento ligeramente inferior al enfoque 1 en todas las pruebas. La razón principal de esto es que el enfoque 2 no puede aprovechar el paralelismo al mismo nivel que el enfoque 1, ya que los procesadores aún deben leer y cifrar los datos secuencialmente a medida que se leen. Con una clave de dificultad "fácil", el enfoque 2 fue más eficiente que el enfoque "Naive", pero el enfoque 1 siguió siendo el más rápido debido a su mayor paralelismo.

El enfoque 1, que implica la encriptación en bloques, fue el más eficiente en términos de tiempo de ejecución en todas las pruebas, independientemente de la dificultad de la clave. El enfoque 2, que implica la encriptación de flujo, fue más eficiente que el enfoque "Naive", pero ligeramente menos eficiente que el enfoque 1, debido a la limitación en el aprovechamiento del paralelismo al leer datos en tiempo real. El enfoque "Naive" fue el menos eficiente, ya que cifra el archivo completo de manera secuencial sin aprovechar el paralelismo, lo que resulta en un tiempo de ejecución más largo en todas las pruebas.

En base a esto, se puede concluir que la clave para la eficiencia en el cifrado DES paralelo radica en el aprovechamiento del paralelismo al dividir el trabajo en unidades más pequeñas (en este caso, bloques de datos) que pueden ser procesadas de manera independiente por múltiples procesadores. El enfoque 1 logra esto de manera efectiva, lo que lo convierte en la opción preferida en la mayoría de los casos.

Conclusiones

1. La división en bloques (Enfoque 1) ofrece un rendimiento superior al cifrar archivos grandes en paralelo, demostrando un speedup lineal.
2. La encriptación de flujo (Enfoque 2) es más eficiente que el enfoque "Naive," pero ligeramente inferior al Enfoque 1 debido a las limitaciones en el aprovechamiento del paralelismo.
3. El enfoque "Naive" resulta ser la estrategia menos eficiente, ya que cifra el archivo completo de manera secuencial, independientemente de la dificultad de la clave.
4. La elección de la estrategia de cifrado debe considerar la naturaleza del trabajo y el equilibrio entre eficiencia y complejidad de implementación en sistemas paralelos.

Recomendaciones

En el proceso de desarrollar y evaluar estos enfoques para el cifrado DES, encontramos algunos desafíos significativos. En primer lugar, la búsqueda de una librería que admitiera el cifrado DES fue una tarea complicada, pero finalmente descubrimos que OpenSSL proporciona una sólida implementación de DES, lo que nos permitió avanzar con nuestras pruebas de manera efectiva. Recomendamos a otros investigadores y desarrolladores que busquen bibliotecas confiables como OpenSSL para simplificar la implementación de algoritmos criptográficos.

Además, enfatizamos la importancia de comprender a fondo el algoritmo DES antes de emprender proyectos relacionados con su cifrado. El cifrado DES es un proceso intrincado que requiere conocimientos sólidos para su implementación y evaluación. Invertir tiempo en investigar y comprender cómo funciona el algoritmo es esencial para tomar decisiones informadas y diseñar estrategias efectivas de cifrado paralelo.

Otro reto que enfrentamos fue la consideración de diferentes enfoques para el cifrado paralelo. Explorar y comparar dos enfoques diferentes (cifrado en bloques y encriptación de flujo) nos brindó una valiosa lección sobre cómo aprovechar el paralelismo de manera eficiente. Aprendimos que la elección del enfoque debe basarse en las necesidades específicas del proyecto y la naturaleza del trabajo. Finalmente, este proyecto nos proporcionó una comprensión más profunda del algoritmo DES, cómo funciona y cómo se puede aplicar en entornos paralelos con el soporte de herramientas como OpenMPI.

Anexo 1

Para la sección de catálogo de funciones se puede mencionar que esta es única debido a que todos los programas implementan las mismas funciones con los mismos propósitos y las mismas entradas.

1. `decrypt(long key, char *ciph, int len):`
 - Entrada:
 - a. Key: Un valor largo que representa la clave de cifrado DES.
 - b. ciph: Un puntero a una cadena de caracteres cifrada que se descifra

- c. len: La longitud de la cadena cifrada.
 - Salida: No tiene salida explícita, ya que descripta la cadena de caracteres ciph en su lugar.
 - Descripción: Esta función se encarga de descriptar una cadena de caracteres cifrada utilizando el algoritmo DES (Data Encryption Standard) con una clave específica. Primero, se establece la paridad de la clave y se inicializa una estructura de clave DES. Luego, la función descripta bloques de 8 bytes de la cadena de caracteres cifrada en modo ECB (Electronic Codebook).
2. encrypt(long key, char *ciph, int len):
- Entrada
 - a. key: Un valor largo que representa la clave de cifrado DES.
 - b. ciph: Un puntero a una cadena de caracteres que se cifrará.
 - c. len: La longitud de la cadena a cifrar.
 - Salida: No tiene salida explícita, ya que cifra la cadena de caracteres ciph en su lugar.
 - Descripción: Esta función cifra una cadena de caracteres utilizando el algoritmo DES con una clave específica. Similar a la función decrypt, comienza estableciendo la paridad de la clave y configurando una estructura de clave DES. Luego, cifra la cadena en bloques de 8 bytes en modo ECB
3. int main(int argc, char *argv[]):
- Entrada:
 - a. argc: El número de argumentos de línea de comandos.
 - b. argv: Un arreglo de cadenas que contiene los argumentos de línea de comandos, donde argv[0] es el nombre del programa y argv[1] se espera que sea la clave de cifrado DES
 - Salida: Entero que representa el código de salida del programa.
 - Descripción: La función main es la función principal del programa. Administra la ejecución del programa de cifrado de flujo paralelo. Verifica los argumentos de línea de comandos, abre y lee un archivo de texto, realiza cifrado y descifrado en paralelo utilizando MPI (Message Passing Interface) y muestra el resultado.

Anexo 2

Pruebas de tiempo con 4 procesos inciso B.2

Prueba con la clave: 123456L


```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutefo
mpirun -np 4 ./brute 123456L

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" found in the decrypted text.
Total execution time: 0.000256 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutefo
```

Prueba con la clave: 18014398509481983L

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutefo
mpirun -np 4 ./brute 18014398509481983L

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" found in the decrypted text.
Total execution time: 0.002025 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutefo
```

Prueba con la clave: 18014398509481984L

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutefo
mpirun -np 4 ./brute 18014398509481984L

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" found in the decrypted text.
Total execution time: 0.947592 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutefo
```

Los resultados de nuestras pruebas revelan una relación crucial entre el tiempo de ejecución y el tamaño de la clave utilizada en la encriptación y descifrado de datos. En primer lugar, cuando se emplea una clave de tamaño reducido, como en el caso de la clave "123456L", el tiempo de ejecución para romper el código es mínima, tomando apenas 0.000256 segundos. Esto se debe a que las claves pequeñas son extremadamente vulnerables a ataques de fuerza bruta y pueden ser descifradas prácticamente de inmediato.

Cuando utilizamos una clave más grande, como la clave "18014398509481983L" (equivalente a $256/4$), el tiempo de ejecución aumenta a 0.002025 segundos, lo que sigue siendo relativamente rápido. Sin embargo, es importante notar que, incluso en este caso, la clave no es lo suficientemente robusta para resistir un ataque de fuerza bruta.

Finalmente, cuando incrementamos el tamaño de la clave en un solo bit para obtener "18014398509481984L" ($256/4 + 1$), el tiempo de ejecución se dispara a 0.947592 segundos. Esta diferencia sustancial en el tiempo de ejecución destaca cómo incluso un pequeño cambio

en el tamaño de la clave puede tener un impacto significativo en la resistencia a un ataque de fuerza bruta.

Como se puede observar en las pruebas realizadas, es importante elegir una clave segura y de tamaño adecuado en los sistemas de cifrado. Claves pequeñas y débiles son vulnerables y pueden ser comprometidas fácilmente, mientras que claves más grandes y complejas aumentan significativamente la resistencia a los ataques. La seguridad de los sistemas de cifrado depende en gran medida de la elección de claves sólidas y de un tamaño suficiente para resistir los intentos de descifrado.

Pruebas inciso B.6 enfoque naive

Clave fácil de encontrar:

Clave: $(2^{56}) / 2 + 1$

Valor: 72057594037927937

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutef  
mpirun -np 4 ./brute 72057594037927937  
  
Original data : Esta es una prueba de proyecto 2  
Decrypted data:0;V-0000<00rJ000  
00"0Ze/5S00  
Keyword "es una prueba de" not found in the decrypted text.  
Total execution time: 0.000794 seconds  
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutef
```

Clave medianamente difícil de encontrar:

Clave: $(2^{56}) / 2 + (2^{56}) / 8$

Valor: 86469112845513593

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutef  
mpirun -np 4 ./brute 86469112845513593  
  
Original data : Esta es una prueba de proyecto 2  
Decrypted data: Esta es una prueba de proyecto 2  
Keyword "es una prueba de" found in the decrypted text.  
Total execution time: 0.003747 seconds  
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutef
```

Clave difícil de encontrar:

Clave: $(2^{56}) / 7 + (2^{56}) / 13$ (aproximado al entero superior)

Valor: 13368022518584656

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutef  
mpirun -np 4 ./brute 13368022518584656  
  
Original data : Esta es una prueba de proyecto 2  
Decrypted data: Esta es una prueba de proyecto 2  
Keyword "es una prueba de" found in the decrypted text.  
Total execution time: 0.654155 seconds  
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Brutef
```

Pruebas del enfoque 1

Clave fácil de encontrar:

Clave: $(2^{56}) / 2 + 1$

Valor: 72057594037927937

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
mpirun -np 4 ./bfl 72057594037927937

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" not found in the decrypted text.
Total execution time: 0.000046 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing
```

Clave medianamente difícil de encontrar:

Clave: $(2^{56}) / 2 + (2^{56}) / 8$

Valor: 86469112845513593

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
mpirun -np 4 ./bfl 86469112845513593

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" not found in the decrypted text.
Total execution time: 0.000077 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing
```

Clave difícil de encontrar:

Clave: $(2^{56}) / 7 + (2^{56}) / 13$ (aproximado al entero superior)

Valor: 13368022518584656

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
mpirun -np 4 ./bfl 13368022518584656

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" not found in the decrypted text.
Total execution time: 0.000293 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing
```

Pruebas del enfoque 2

Clave fácil de encontrar:

Clave: $(2^{56}) / 2 + 1$

Valor: 72057594037927937

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
mpirun -np 4 ./bf2 72057594037927937

Original data : Esta es una prueba de proyecto 2
Decrypted data: 000Am059000L000*P\0nP0W00w
Keyword "es una prueba de" not found in the decrypted text.
Total execution time: 0.000121 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
```

Clave medianamente difícil de encontrar:

Clave: $(2^{56}) / 2 + (2^{56}) / 8$

Valor: 86469112845513593

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
mpirun -np 4 ./bf2 86469112845513593

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" found in the decrypted text.
Total execution time: 0.001033 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
```

Clave difícil de encontrar:

Clave: $(2^{56}) / 7 + (2^{56}) / 13$ (aproximado al entero superior)

Valor: 13368022518584656

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
mpirun -np 4 ./bf2 13368022518584656

Original data : Esta es una prueba de proyecto 2
Decrypted data: Esta es una prueba de proyecto 2
Keyword "es una prueba de" found in the decrypted text.
Total execution time: 0.000259 seconds
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/Bruteforcing/Bruteforcing
```

Bibliografía

Akamai. (s.f). ¿Qué es un ataque de fuerza bruta? Recuperado de:

<https://www.akamai.com/es/glossary/what-is-brute-force-attack>

Kaspersky. (s.f.) Brute Force Attack: Definition and Examples. Recuperado de:

<https://www.kaspersky.com/resource-center/definitions/brute-force-attack>

KeepCoding. (15 de marzo de 2023). ¿Que es el algoritmo DES? Recuperado de:

<https://keepcoding.io/blog/que-es-el-algoritmo-des/>

GeeksForGeeks. (s.f.). Data encryption standard (DES) | Set 1. Recuperado de:

<https://www.geeksforgeeks.org/data-encryption-standard-des-set-1/>