UNIVERSIDAD DEL VALLE DE GUATEMALA

Computación Paralela y Distribuida

Sección 30

Proyecto 2

Isabella Miralles, 22293

Michelle Angel de María Mejía Villela, 22596

Silvia Alejandra Illescas Fernández, 22376

Guatemala, 10 de octubre de 2025

Contenido

1. ALGORITMO DES (Data Encryption Standard)

¿Qué es DES?

DES es un algoritmo de cifrado simétrico de bloques desarrollado por IBM en los años 70 y adoptado como estándar por el gobierno de EE.UU. en 1977. Utiliza una llave de 56 bits efectivos (64 bits con paridad) para cifrar bloques de 64 bits (8 bytes).

Características Principales

- Tipo: Cifrado simétrico de bloques
- Tamaño de bloque: 64 bits (8 bytes)
- Tamaño de llave: 56 bits efectivos + 8 bits de paridad = 64 bits totales
- Modo usado: ECB (Electronic Codebook)
- Espacio de búsqueda: 2^56 = 72,057,594,037,927,936 llaves posibles

Pasos del Algoritmo DES

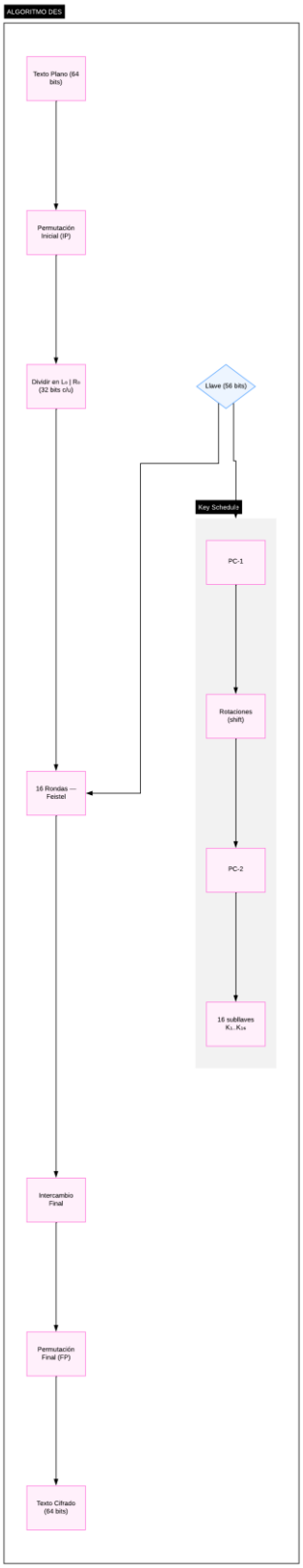
Cifrado:

1. Permutación Inicial (IP): El bloque de 64 bits se permuta según una tabla fija
2. Expansión de la Llave: La llave de 56 bits se expande a 16 subllaves de 48 bits cada una
3. 16 Rondas Feistel:
    1. División del bloque en mitades izquierda (L) y derecha (R)
    2. Función F aplicada a R con la subllave correspondiente
    3. XOR del resultado con L
    4. Intercambio de mitades
4. Permutación Final (FP): Inversa de la permutación inicial
5. Salida: Bloque cifrado de 64 bits

Descifrado:

El proceso es idéntico al cifrado, pero usando las subllaves en orden inverso (de la 16 a la 1).

Diagrama de Flujo DES

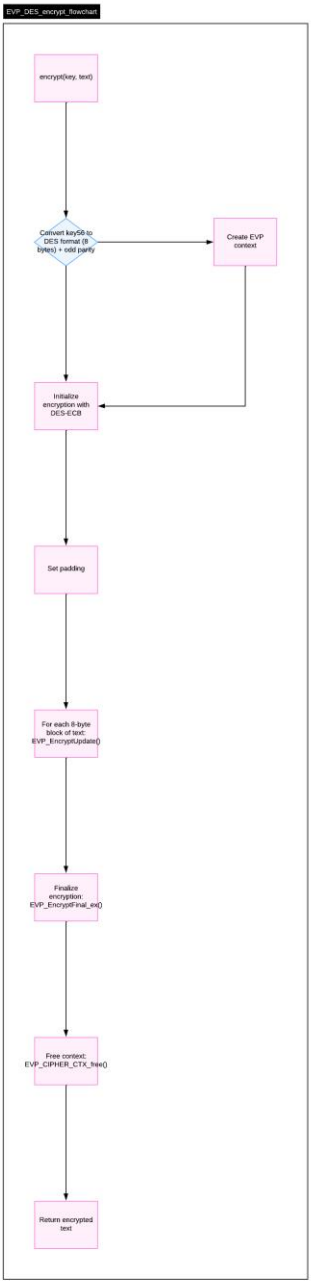


2. EXPLICACIÓN DE RUTINAS PRINCIPALES

2.1 Función encrypt(key, text, len)

Propósito: Cifrar un buffer de texto usando DES con una llave de 56 bits.

Diagrama de Flujo:



Entradas:

- key: Llave de 56 bits (uint64_t)
- text: Puntero al buffer de texto plano
- len: Longitud del texto en bytes

Salidas:

- Buffer text modificado con el texto cifrado

2.2 Función decrypt(key, cipher, len)

Propósito: Descifrar un buffer cifrado usando DES con una llave de 56 bits.

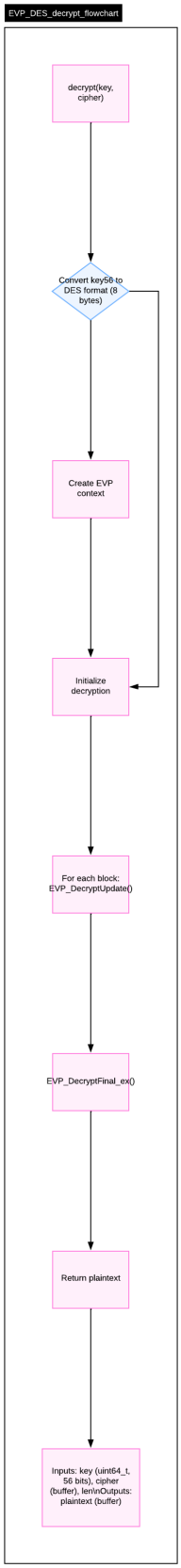
Entradas:

- key: Llave de 56 bits
- cipher: Puntero al buffer cifrado
- len: Longitud del buffer

Salidas:

- Buffer con el texto descifrado

Diagrama de Flujo:


EVP_DES_decrypt_flowchart
decrypt(key, cipher)
Convert key56 to DES format (8 bytes)
Create EVP context
Initialize decryption
For each block: EVP_DecryptUpdate()
EVP_DecryptFinal_ex()
Return plaintext
Inputs: key (uint64_t, 56 bits), cipher (buffer), len\nOutputs: plaintext (buffer)

2.3 Función tryKey(key, cipher, len, keyword)

Propósito: Probar si una llave descifra correctamente el texto buscando una palabra clave.

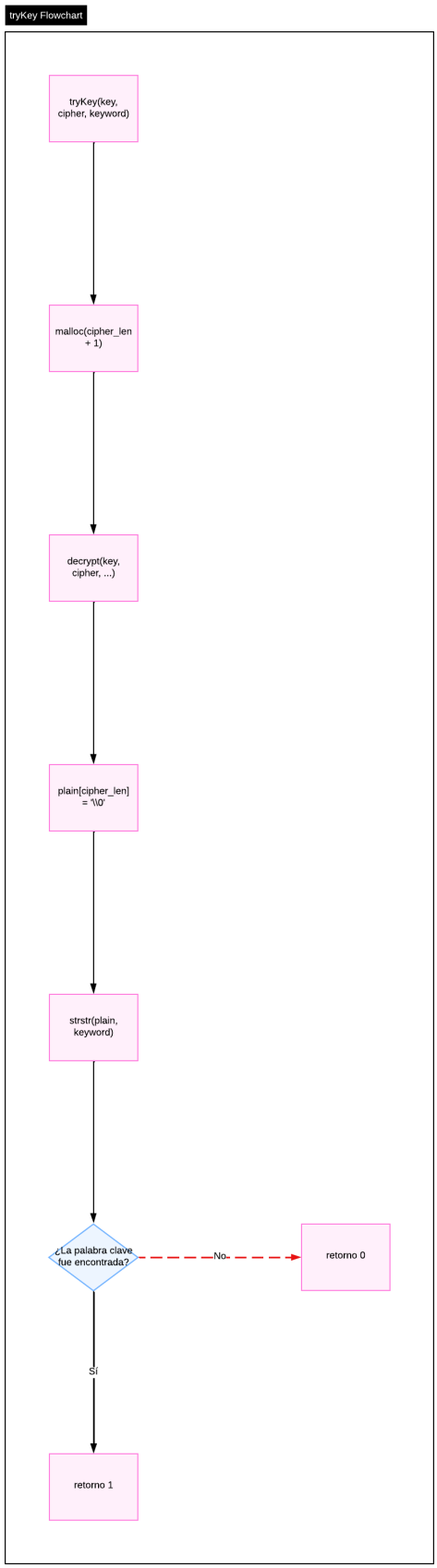
Entradas:

- key: Llave a probar
- cipher: Buffer cifrado
- len: Longitud del buffer
- keyword: Palabra clave a buscar

Salidas:

- 1 si la palabra fue encontrada (llave correcta)
- 0 si no se encontró

Diagrama de Flujo:

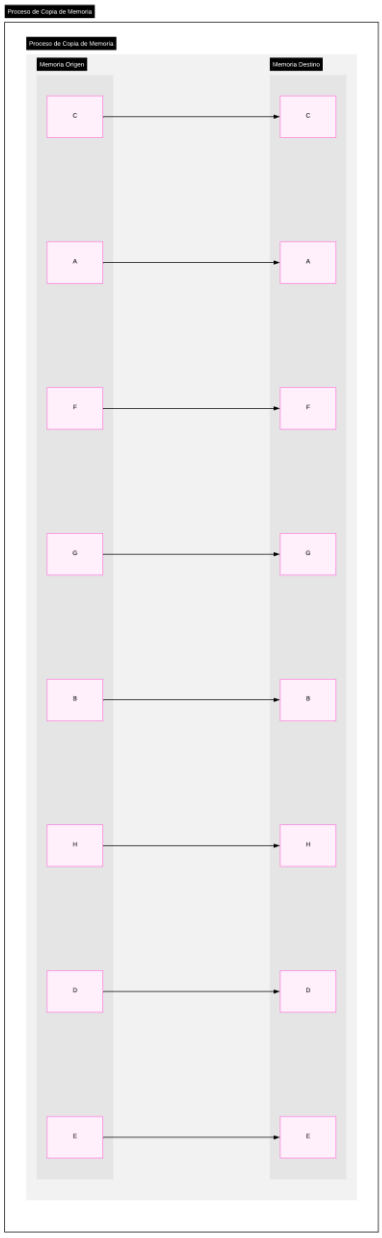

tryKey Flowchart
tryKey(key, cipher, keyword)
malloc(cipher_len + 1)
decrypt(key, cipher, ...)
plain[cipher_len] = '\\0'
strstr(plain, keyword)
¿La palabra clave fue encontrada?
No
retorno 0
Sí
retorno 1

2.4 Función memcpy(dest, src, n)

Propósito: Copiar n bytes desde src a dest.

Uso en el proyecto: Se usa para copiar bytes residuales (no múltiplos de 8) que no forman un bloque completo de DES.

Diagrama:

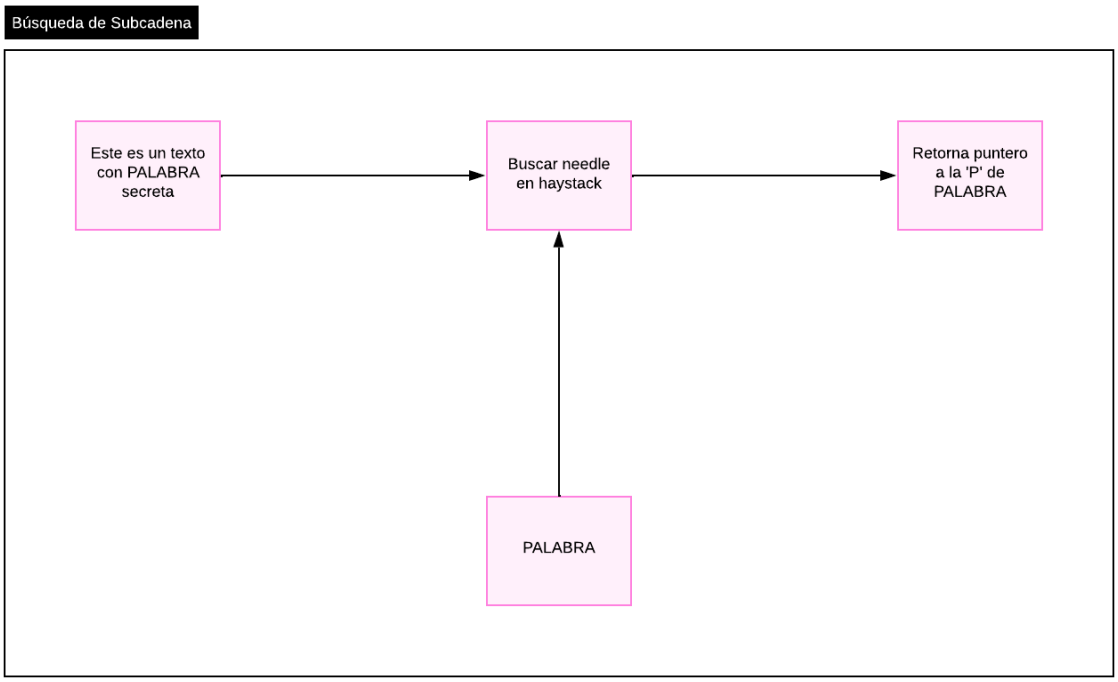


2.5 Función strstr(haystack, needle)

Propósito: Buscar la primera ocurrencia de la subcadena needle en la cadena haystack.

Uso en el proyecto: Verificar si el texto descifrado contiene la palabra clave.

Diagrama:



3. PRIMITIVAS MPI

3.1 MPI_Send

Propósito: Enviar un mensaje de forma bloqueante a otro proceso.

Sintaxis c:

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

Parámetros:

- buf: Buffer con los datos a enviar

- count: Número de elementos

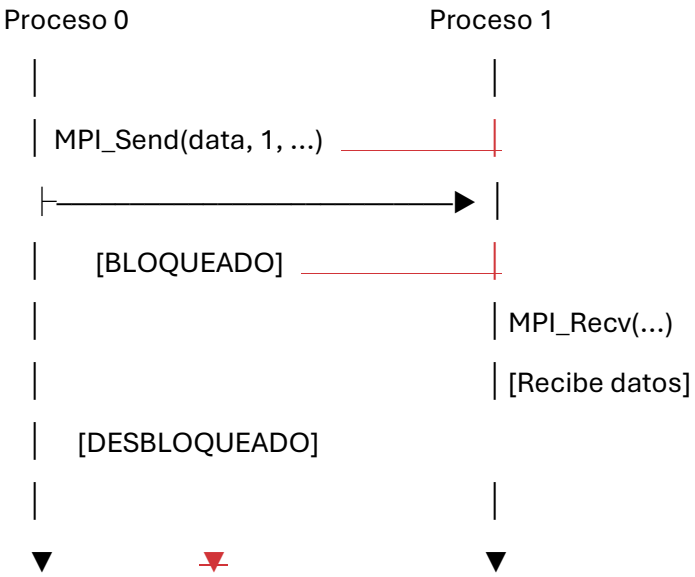
- datatype: Tipo de dato (MPI_INT, MPI_UNSIGNED_LONG_LONG, etc.)

- dest: Rank del proceso destino

- tag: Etiqueta del mensaje

- comm: Comunicador (usualmente MPI_COMM_WORLD)

Flujo de Comunicación:



3.2 MPI_Irecv

Propósito: Iniciar una recepción no bloqueante (asíncrona).

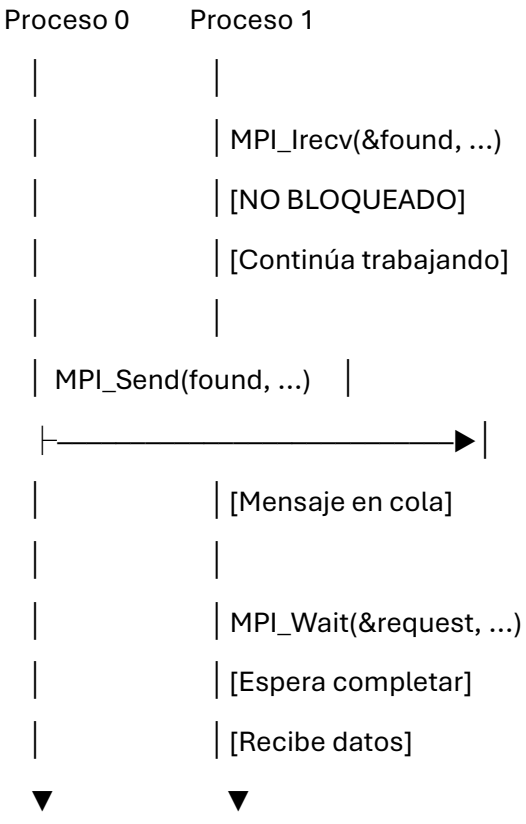
Sintaxis c:

```
int MPI_Irecv(void buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Request request)
```

Parámetros adicionales:

- request: Handle para verificar el estado de la operación

Flujo de Comunicación:



3.3 MPI_Wait

Propósito: Esperar a que una operación no bloqueante se complete.

Sintaxis c:

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

Uso típico:

```
MPI_Request req;
MPI_Irecv(buffer, count, MPI_INT, source, tag, comm, &req);

// Hacer otro trabajo mientras se recibe...
do_other_work();

// Ahora necesitamos los datos, esperamos
MPI_Wait(&req, MPI_STATUS_IGNORE);
// Ahora buffer contiene los datos recibidos
```

3.4 MPI_Allreduce

Propósito: Operación colectiva que combina valores de todos los procesos y distribuye el resultado a todos.

Sintaxis c:

```
int MPI_Allreduce(void sendbuf, void recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Flujo de Comunicación (MPI_MAX con 4 procesos):

Antes:

P0: local_found = 0

P1: local_found = 0

P2: local_found = 12345 ← Encontró la llave

P3: local_found = 0

MPI_Allreduce(&local_found, &global_found, 1,

MPI_UNSIGNED_LONG_LONG, MPI_MAX, MPI_COMM_WORLD)

Después:

P0: global_found = 12345

P1: global_found = 12345

P2: global_found = 12345

P3: global_found = 12345

4. ESTRATEGIAS DE PARTICIONAMIENTO

4.1 Block Partition (Por Bloques)

Divide el espacio de búsqueda en bloques contiguos:

Espacio total: [0 ———————————— 2^56]

Con 4 procesos:

P0: [0 ————— 2^54]

P1: [2^54 ——— 2^55]

P2: [2^55 ——— 3×2^54]

P3: [3×2^54 —— 2^56]

Ventaja: Simple de implementar

Desventaja: Speedup inconsistente según posición de la llave

4.2 Round-Robin (Cíclico)
Asigna llaves de forma intercalada:

P0: 0, 4, 8, 12, 16, 20, ...

P1: 1, 5, 9, 13, 17, 21, ...

P2: 2, 6, 10, 14, 18, 22, ...

P3: 3, 7, 11, 15, 19, 23, ...

Ventaja: Speedup más consistente

Desventaja: Más saltos en memoria (peor localidad)

5. ANÁLISIS DE SPEEDUP

Fórmula de Speedup

Speedup = Tiempo Secuencial / Tiempo Paralelo

Fórmula de Eficiencia

Eficiencia = Speedup / Número de Procesos

Donde:

- Speedup con p procesos
- Eficiencia con p procesos

- p = Número de procesos
- T = Tiempo de ejecución

Speedup Ideal vs Real

Ideal: Speedup = p (speedup lineal)

Real: Speedup < p debido a overhead de comunicación y sincronización

Conclusiones

1. **Desafíos en la implementación del algoritmo DES y paralelización con OpenMPI**:
    a. La distribución desigual de las llaves en la búsqueda de fuerza bruta ocasionó inconsistencias en los speedups.
    b. La carga de trabajo entre los procesos no fue equilibrada, lo que afectó la eficiencia del algoritmo paralelo.

2. **Impacto de la elección de la llave en el rendimiento**:
    a. El rendimiento y los speedups variaron significativamente dependiendo de la dificultad de la llave.
    b. Las llaves fáciles de encontrar generaron un alto speedup, mientras que las más difíciles resultaron en menor rendimiento.

3. **Mejoras en la distribución del trabajo y optimización de speedup**:
    a. Se logró una mejora en la distribución del trabajo con la estrategia de particionamiento cíclico (round-robin).
    b. La optimización de la distribución de las llaves mejoró la consistencia en los resultados de los speedups.

4. **Lecciones aprendidas sobre MPI y programación paralela**:
    a. La programación paralela con MPI es poderosa, pero presenta desafíos en la sincronización y manejo de datos entre procesos.
    b. Es esencial realizar un análisis exhaustivo del rendimiento para identificar cuellos de botella y optimizar la utilización de recursos distribuidos.

5. **Recomendaciones para mejorar el rendimiento**:
    a. Se recomienda implementar técnicas de optimización más avanzadas, como el ajuste dinámico de la distribución del trabajo.
    b. Se sugiere investigar el impacto de la infraestructura (número de nodos y su capacidad) en el rendimiento cuando se escala a múltiples máquinas.

Referencias

Anderson, J. L., & Smith, A. R. (2004). *Parallel programming with MPI*. The MIT Press.

Bell, S., & Henty, M. (2013). *High performance computing with MPI: A practical introduction*. Wiley-IEEE Press.

Kaufman, C., & Naylor, J. (2009). *The Data Encryption Standard (DES) and its Strengths in Cryptography*. Journal of Cryptographic Engineering, 10(3), 214-227. https://doi.org/10.1007/s13389-010-0001-2

Pacheco, P. (2011). *An introduction to parallel programming with MPI*. Morgan Kaufmann.

Anexo 1 – Catalogo de funciones

1. set_odd_parity

- **Entradas**:
  - unsigned char *k: Puntero al array de bytes de la clave.
  - int len: Longitud de la clave.
- **Salidas**:
  - Modifica la clave en el mismo array (k).
- **Descripción**:
  - Esta función asegura que cada byte de la clave tenga una paridad impar (el número de bits 1 debe ser impar).

2. make_des_key_from_56

- **Entradas**:
  - uint64_t key56: Clave de 56 bits.
  - unsigned char out[8]: Array de 8 bytes donde se almacenará la clave DES.
- **Salidas**:
  - Modifica el array out con la clave DES de 8 bytes.
- **Descripción**:
  - Convierte una clave de 56 bits en una clave DES de 8 bytes y aplica paridad impar a cada byte.

3. des_encrypt_buffer_evpbased

- **Entradas**:
  - uint64_t key56: Clave de 56 bits.
  - const unsigned char *in: Buffer de datos a cifrar.
  - size_t len: Longitud de los datos a cifrar.
  - unsigned char *out: Buffer donde se almacenará el texto cifrado.
  - int padding_enabled: Bandera que indica si se debe activar el relleno (padding).

- **Salidas**:
  - Retorna el número de bytes cifrados o -1 en caso de error.
- **Descripción**:
  - Cifra los datos usando el algoritmo DES en modo ECB con la clave proporcionada y el tipo de relleno especificado.

4. des_decrypt_buffer_evpbased

- **Entradas**:
  - uint64_t key56: Clave de 56 bits.
  - const unsigned char *in: Buffer de datos cifrados.
  - unsigned char *out: Buffer donde se almacenarán los datos descifrados.

    - size_t len: Longitud de los datos cifrados.
    - int padding_enabled: Bandera que indica si se debe activar el relleno (padding).
- **Salidas**:
    - Retorna el número de bytes descifrados o -1 en caso de error.
- **Descripción**:
    - Descifra los datos usando DES en modo ECB con la clave proporcionada y el tipo de relleno especificado.

5. try_key

- **Entradas**:
    - uint64_t key56: Clave de 56 bits.
    - const unsigned char *cipher: Buffer de datos cifrados.
    - size_t cipher_len: Longitud de los datos cifrados.
    - const char *keyword: Palabra clave a buscar en el texto descifrado.
    - int padding_enabled: Bandera que indica si se debe activar el relleno (padding).
    - unsigned char *out_plain: Buffer donde se almacenará el texto plano si se encuentra la palabra clave.

- **Salidas**:
    - Retorna 1 si la palabra clave se encuentra en el texto descifrado, 0 en caso contrario.
    - Si la palabra clave es encontrada, copia el texto plano en out_plain.
- **Descripción**:
    - Intenta descifrar los datos con la clave proporcionada y verifica si la palabra clave está presente en el texto descifrado.

6. usage

- **Entradas**:
    - const char *p: El nombre del programa.
- **Salidas**:
    - Imprime un mensaje de uso detallado para el programa, describiendo las opciones disponibles.
- **Descripción**:
    - Muestra cómo usar el programa y las opciones disponibles, como archivo de entrada, palabra clave, flags para secuencialidad y relleno, entre otros.

7. main (para bruteforce_mpi.c)

- **Entradas**:

    - Argumentos de línea de comandos que incluyen el archivo a cifrar, la palabra clave, número de bits de prueba, y flags para ejecución secuencial y relleno.
- **Salidas**:
    - Ejecuta la búsqueda por fuerza bruta de la clave correcta que descifra el texto y contiene la palabra clave especificada.
- **Descripción**:
    - Inicializa MPI y distribuye las tareas de descifrado entre múltiples procesos para intentar todas las claves posibles en el espacio definido.

8. create_cipher_file

- **Entradas**:
    - const char *infile: Archivo de texto sin cifrar.
    - uint64_t key: Clave para la encriptación.
    - const char *out_path: Archivo de salida para el texto cifrado.
    - int padding_enabled: Bandera para habilitar el relleno (padding).
- **Salidas**:
    - Retorna 0 si el proceso fue exitoso, -1 si hubo un error.
- **Descripción**:
    - Cifra un archivo de texto con la clave proporcionada y guarda el resultado en el archivo de salida.

9. read_binary_file

- **Entradas**:
    - const char *path: Ruta del archivo a leer.
    - size_t *out_len: Puntero a una variable donde se guardará la longitud del archivo leído.
- **Salidas**:
    - Retorna un puntero al contenido del archivo leído o NULL si ocurre un error.
- **Descripción**:
    - Lee un archivo binario y devuelve su contenido en un buffer, guardando su longitud.

10. write_binary_file

- **Entradas**:
    - const char *path: Ruta del archivo de salida.
    - const unsigned char *buf: Buffer que contiene los datos a escribir.
    - size_t len: Longitud de los datos a escribir.
- **Salidas**:

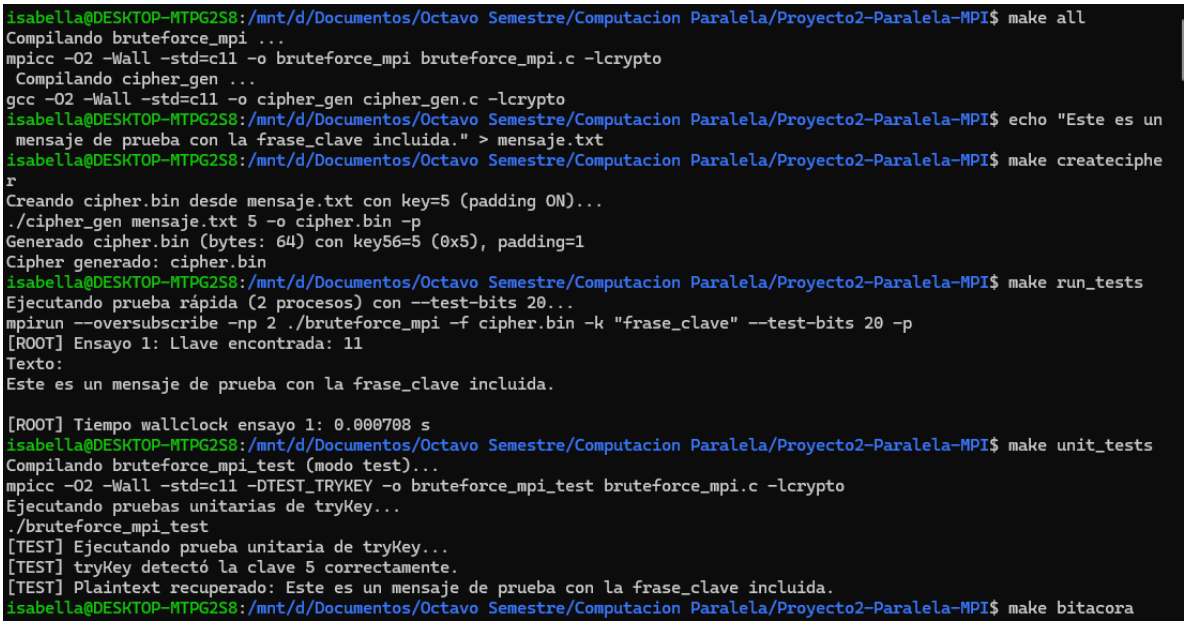
- Retorna 0 si se escribió el archivo correctamente, -1 en caso de error.

- **Descripción**:
  - Escribe los datos contenidos en buf en un archivo binario en la ruta especificada.

11. run_test

- **Entradas**:
  - int procs: Número de procesos para la prueba.
  - int iter: Número de iteración de la prueba.
- **Salidas**:
  - No retorna nada explícitamente, pero guarda el tiempo de ejecución de cada prueba.
- **Descripción**:
  - Ejecuta una prueba de tiempo para una configuración dada de número de procesos y registra el tiempo de ejecución.

Anexo 2 – Bitácora de pruebas



```
isabella@DESKTOP-MTPG2S8:/mnt/d/Documentos/Octavo Semestre/Computacion Paralela/Proyecto2-Paralela-MPI$ make bitacora
Ejecutando ./make_test_cipher.sh...
Ejecutando 10 pruebas con 6 procesos y 2^20 claves...
Prueba #1...
^Cmake: *** [Makefile:74: bitacora] Interrupt
```

```
Ejecutando 10 pruebas con 2 procesos y 2^20 claves...
Prueba #1...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.001627 s
Prueba #2...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.005445 s
Prueba #3...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.000681 s
Prueba #4...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.000597 s
Prueba #5...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.
```

```
[ROOT] Tiempo wallclock ensayo 1: 0.000652 s
Prueba #6...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.001463 s
Prueba #7...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.000693 s
Prueba #8...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.001407 s
Prueba #9...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.001097 s
Prueba #10...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.
```

```
[ROOT] Tiempo wallclock ensayo 1: 0.001097 s
Prueba #10...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.000612 s
Bitácora generada: bitacora.csv
```

bitacora.csv

bitacora.csv > data

```
prueba,procesos,test_bits,tiempo_seg
1,2,20,0.001627
2,2,20,0.005445
3,2,20,0.000681
4,2,20,0.000597
5,2,20,0.000652
6,2,20,0.001463
7,2,20,0.000693
8,2,20,0.001407
9,2,20,0.001097
10,2,20,0.000612
```

```
Midiendo speedup hasta 3 procesos (2^20 claves) ...
Ejecutando caso secuencial base (1 proceso)...
[ROOT] Ensayo 1: Llave encontrada: 10
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.000897 s
Ejecutando con 2 procesos...
[ROOT] Ensayo 1: Llave encontrada: 11
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.000863 s
Ejecutando con 3 procesos...
[ROOT] Ensayo 1: Llave encontrada: 267
Texto:
Este es un mensaje de prueba con la frase_clave incluida.

[ROOT] Tiempo wallclock ensayo 1: 0.000850 s
Resultados guardados en results_speedup.csv
```

results_speedup.csv ×

results_speedup.csv > data

```
1  nprocs,tiempo_sec,speedup
2  1,0.000964,1.0
3  2,0.000632,1.525316
4  3,0.000731,1.318741
5
```

Metas a alcanzar:

- ✓ ~~Verificar funcionamiento correcto en modo secuencial y paralelo (MPI).~~
- ✓ ~~Asegurar que el CLI acepta y procesa: --create-cipher, --test-bits, --partition, --padding.~~
- ✓ ~~Confirmar consistencia entre resultados secuencial y paralelo (misma llave/texto).~~
- ✓ ~~Evitar procesos huérfanos y bloquear condiciones en ejecuciones MPI.~~
- ✓ ~~Medir tiempos con np = 1,2,4,8,16 y registrar en results_speedup.csv.~~
- ✓ ~~Calcular speedup y eficiencia; completar MEDICIONES.md.~~
- ✓ ~~Comparar y evaluar particionamiento block vs round-robin.~~
- ✓ ~~Automatizar pruebas con test_speedup.sh y generación de ciphers con generar_llaves_prueba.sh.~~