UNIVERSIDAD DEL VALLE DE GUATEMALA

Computación Paralela

Sección 10

Ing. Juan Luis García Zarceño

Proyecto #2

Maria Ramirez #21342

Gustavo González #21438

Parte A

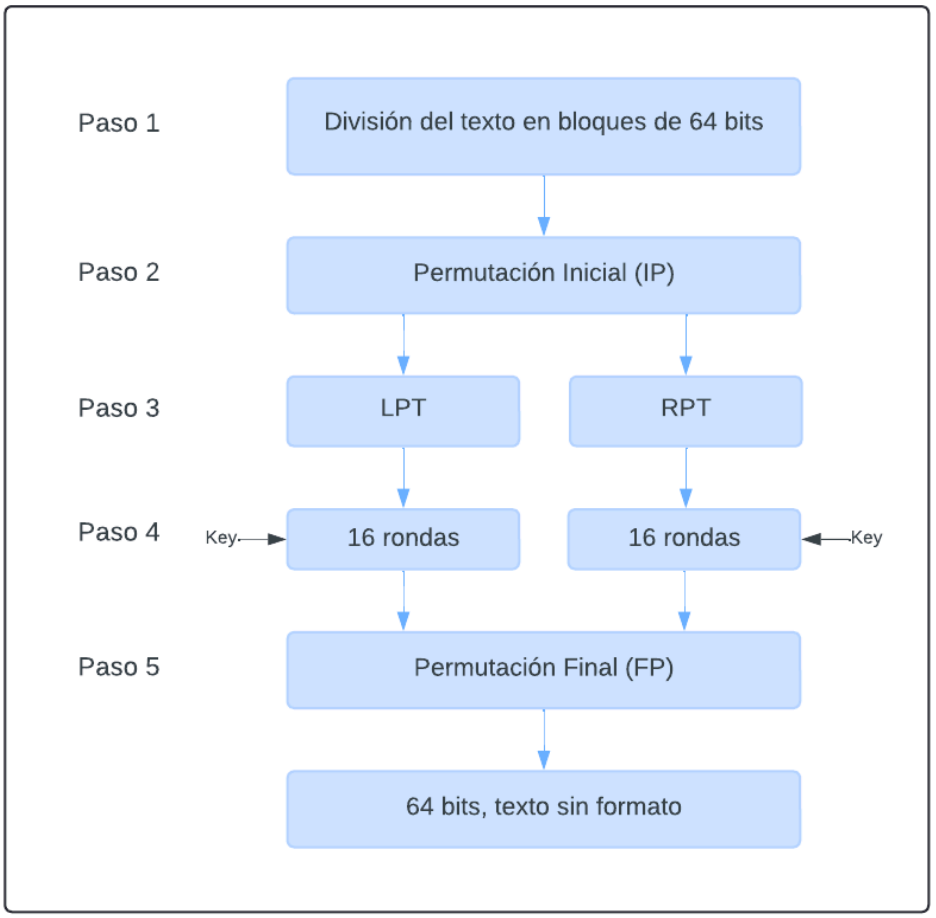
Investigación

El algoritmo *DES* (Data Encryption Standard) es un estándar de cifrado simétrico que utiliza una clave de 56 bits para cifrar bloques de texto plano de 64 bits. Los pasos para cifrar/descifrar con DES son:

1. División del texto en bloques de 64 bits.
2. Inicialización del proceso con una clave de 56 bits (generada por una serie de permutaciones).
3. 16 rondas de sustitución y permutación mediante una función F que utiliza la clave y el texto del bloque. Cada ronda incluye:
    a. Expansión del bloque de 32 bits a 48 bits.
    b. Mezcla con la clave (XOR).
    c. Uso de cajas S (S-boxes) para reducir el tamaño del bloque de nuevo a 32 bits.
    d. Permutación del bloque resultante.
4. Intercambio de mitades de los bloques (swap) y aplicación de la permutación final.
5. El descifrado sigue el mismo proceso, pero aplicando las claves de forma inversa.

Referencia de imagen: https://www.techtarget.com/rms/onlineimages/encryption_operation-f.png

Diagrama de Flujo



Cómo funcionan las rutinas

a. **decrypt** (key, *ciph, len) y encrypt (key, *ciph, len):
Ambas rutinas se encargan de la encriptación y desencriptación de datos usando el algoritmo DES (Data Encryption Standard). El flujo para ambas funciones es muy similar, con la única diferencia de que una desencripta y la otra encripta.

Proceso:
Entrada: Una clave de 56 bits (key), un bloque de texto cifrado (*ciph), y la longitud del bloque (len).
Conversión de clave: Se convierte la clave de 56 bits en un bloque DES (DES_cblock), dividiéndola en 8 bytes.
Paridad y programación de la clave: Se establece la paridad impar de la clave usando DES_set_odd_parity y se configura el programador de la clave con DES_set_key_checked.
Cifrado/Descifrado: Se realiza la operación de cifrado o descifrado con DES_ecb_encrypt. Este método usa el modo ECB (Electronic Codebook), que cifra o descifra bloques de 64 bits individualmente.

b. **tryKey** (key, *ciph, len):
La función tryKey se encarga de probar si una clave dada (key) puede descifrar correctamente el texto cifrado (*ciph) y verificar si contiene una cadena específica (search = " the ").

Proceso:
Copiar el cifrado: Utiliza memcpy para hacer una copia temporal del texto cifrado en el buffer temp.
Descifrar: Llama a decrypt para descifrar el texto usando la clave proporcionada.
Buscar la cadena: Utiliza strstr para buscar si el texto descifrado contiene la palabra " the ".
Resultado: Retorna 1 si encuentra la cadena y 0 si no la encuentra.

c. **memcpy**:
memcpy copia bloques de memoria de una ubicación a otra. En este caso, se utiliza para hacer una copia temporal del texto cifrado antes de realizar la operación de descifrado.

Proceso:
Origen: Recibe el puntero al texto cifrado (*ciph).
Destino: Copia el texto cifrado a un buffer temporal temp.
Longitud: Copia exactamente len bytes.

d. **strstr**:
strstr es una función de la biblioteca estándar de C que busca una subcadena en una cadena más larga. En este caso, busca la subcadena " the " en el texto descifrado.

Proceso:
Entrada: Recibe el buffer de texto descifrado temp y la cadena a buscar " the ".
Búsqueda: Escanea el texto descifrado para ver si contiene la subcadena.
Resultado: Retorna un puntero a la primera aparición de la subcadena si es encontrada, o NULL si no se encuentra.

<u>Explicación del el uso y flujo de comunicación de las primitivas de MPI</u>

a. MPI_Irecv
Esta es una función no bloqueante que se utiliza para recibir mensajes de otros procesos. Esta función permite que el proceso que la invoca continúe su ejecución sin esperar a que se complete la recepción del mensaje.

Uso:
MPI_Irecv(buffer, count, datatype, source, tag, comm, &request);

Primero está el puntero al búfer donde se almacenarán los datos recibidos. Además de count que son los números de elementos que se recibirán. datatype es el tipo de dato de los elementos que se recibirán (MPI_INT, MPI_FLOAT, etc.). source es el identificador del proceso fuente desde el que se espera recibir el mensaje (puede ser MPI_ANY_SOURCE). tag que etiqueta del mensaje para diferenciar tipos de mensajes (puede ser MPI_ANY_TAG) comm es el Comunicador que agrupa los procesos (normalmente MPI_COMM_WORLD). Y por último request que es un objeto de tipo MPI_Request que se usará para manejar la operación no bloqueante.

b. MPI_Send

MPI_Send es una primitiva de envío bloqueante en MPI. Cuando un proceso llama a MPI_Send, se detiene hasta que los datos han sido copiados fuera del buffer, ya sea al sistema de comunicación o al proceso receptor. En muchos casos, esto significa que el proceso queda bloqueado hasta que el receptor haya llamado a una función de recepción (como MPI_Recv o MPI_Irecv).

Uso:
*int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);*

- buf: Puntero al buffer con los datos a enviar.
- count: Número de elementos a enviar.
- datatype: Tipo de datos a enviar.
- dest: ID del proceso destino.
- tag: Etiqueta del mensaje (permite identificar distintos mensajes).
- comm: Comunicador (ej. MPI_COMM_WORLD).

c. MPI_Wait

MPI_Wait se utiliza para esperar a que una operación no bloqueante (como MPI_Irecv o MPI_Isend) se complete. Es útil cuando se ha iniciado una operación asíncrona y el proceso necesita asegurarse de que dicha operación ha terminado antes de continuar.

Uso:
*int MPI_Wait(MPI_Request *request, MPI_Status *status);*

- request: El objeto MPI_Request asociado con la operación no bloqueante.
- status: Objeto que contiene información sobre el estado de la operación (ej., si fue exitosa, el origen del mensaje, etc.).

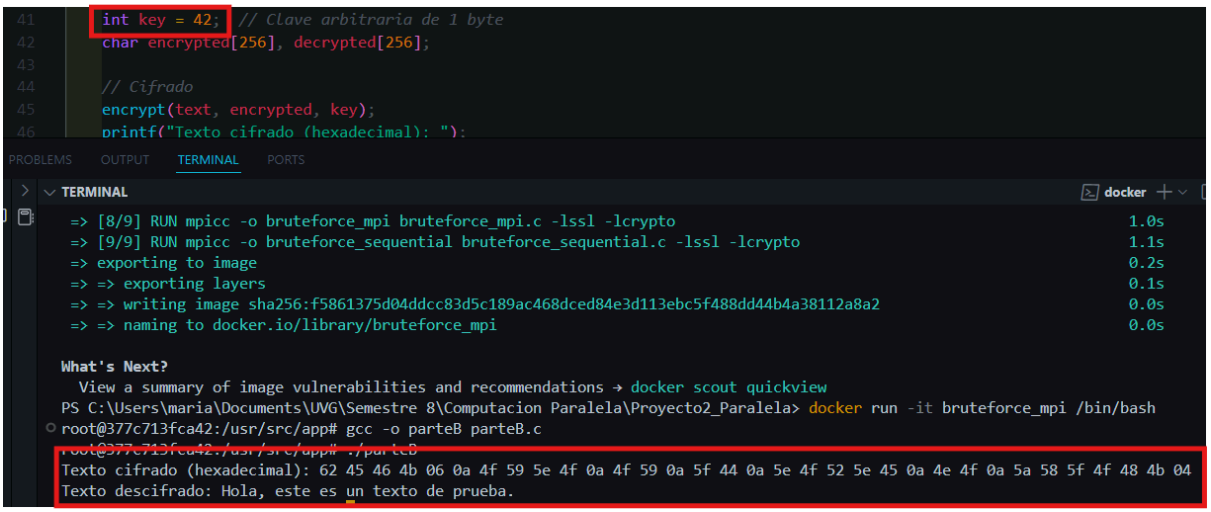
<u>Flujo de Comunicación:</u>

1. Operación No Bloqueante: Se inicia una operación de recepción o envío no bloqueante usando MPI_Irecv o MPI_Isend.

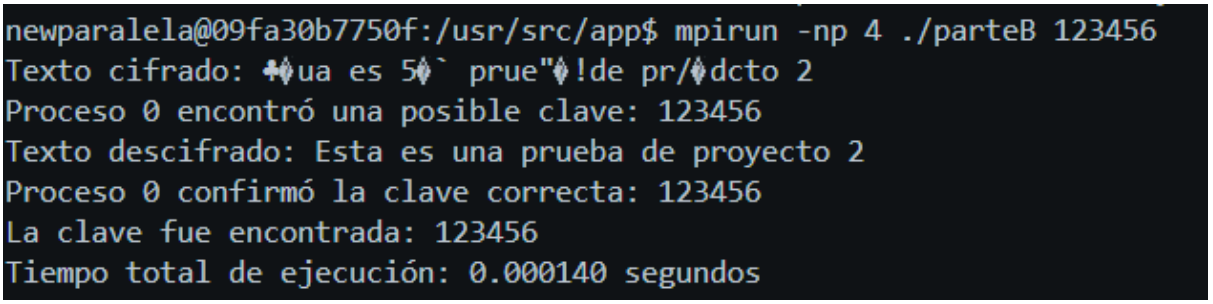
2. Esperar Completitud: El proceso llama a MPI_Wait, que bloquea la ejecución hasta que la operación especificada por request ha terminado.
3. Continuar: Una vez que la operación ha terminado, MPI_Wait retorna y el proceso continúa su ejecución.

Parte B

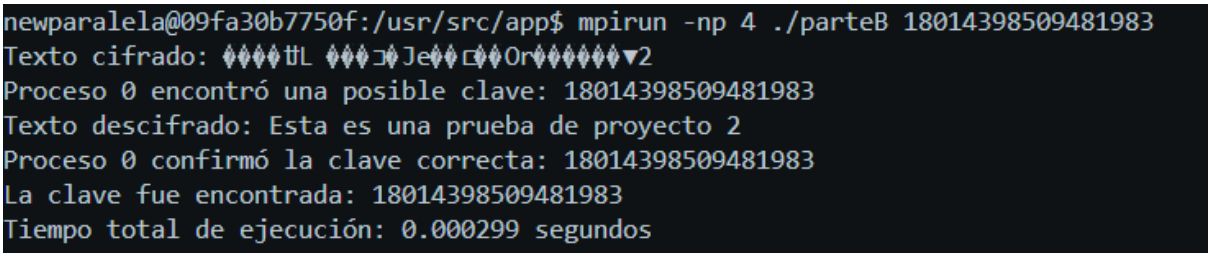
Programa que cifra un texto (.txt) usando una llave arbitraria:



Clave 123456L



Clave 18014398509481983L



Clave 18014398509481984

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 18014398509481984
Texto cifrado: Esta e3 una pr5eba de 0royecto`2
Proceso 0 encontró una posible clave: 18014398509481984
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 18014398509481984
La clave fue encontrada: 18014398509481984
Tiempo total de ejecución: 0.000217 segundos
```

Reflexión:
Durante la ejecución del programa con diferentes claves, se puede observar un comportamiento interesante respecto al tiempo de ejecución en función de la longitud y complejidad de la clave utilizada. Sin embargo en nuestro caso no es una diferencia muy grande.

En primer lugar, la clave inicial probada fue 123456L, que es una clave relativamente corta y sencilla. El programa encontró la clave correcta de forma casi inmediata, registrando un tiempo total de ejecución de 0.000140 segundos. Esto indica que, para claves pequeñas, el proceso de cifrado y descifrado es extremadamente eficiente, ya que el espacio de búsqueda no es muy extenso.

Luego, al probar con la clave 18014398509481983, que es considerablemente más grande, el tiempo de ejecución aumentó ligeramente a 0.000299 segundos. Aunque el incremento es mínimo, muestra que, a medida que la clave se hace más grande y compleja, el programa necesita un poco más de tiempo para completar el proceso. Este resultado es razonable, dado que claves más largas pueden generar secuencias de cifrado más complejas, aunque en este caso el tiempo de búsqueda sigue siendo bastante reducido.

Finalmente, la prueba con la clave 18014398509481984, que es solo una unidad más que la clave anterior, resultó en un tiempo de ejecución intermedio de 0.000217 segundos. Esto sugiere que el tiempo no aumenta de manera estrictamente lineal con el tamaño de la clave, sino que puede variar dependiendo de la estructura interna de la clave y cómo esta interactúa con los algoritmos de cifrado y descifrado. Además se puede ver como encripta el mensaje, es de una manera muy simple por lo que puede ser que al usar (256/4) + 1 el encriptar y desencriptar los mensajes se vuelvan más fácil en este caso.

Proponer 2 alternativas

Alternativa 1. Rango adaptativo basada en heurísticas:

En lugar de dividir el espacio de claves de manera equitativa, se basaría en la suposición de que algunas claves pueden ser más probables de ser correctas que otras, dependiendo de las características del cifrado. Este enfoque utiliza heurísticas para identificar rangos de claves con una mayor probabilidad de contener la clave correcta y asignar más recursos de procesamiento a esos rangos.

Descripción del algoritmo:

1. Estimación de rangos probables: Analizar el patrón de cifrado para determinar qué rangos de claves son más probables. Utilizando estadísticas sobre las claves usadas o basándose en el análisis de la secuencia cifrada.
2. Asignación de procesos: Asignar más procesos a los rangos que tienen una mayor probabilidad de contener la clave correcta.
3. Reevaluación dinámica: Si un proceso durante la ejecución se encuentra que su rango es improbable, entonces se reasigna el procesamiento a un rango más probable. Esto se puede hacer redistribuyendo el trabajo a través de mensajes MPI.

Pseudocódigo:

```
def adaptive_key_search(start_key, end_key, keyword):
    ranges = estimate_likely_ranges(start_key, end_key)  # Dividir en rangos basados en probabilidad
    for range in ranges:
        assign_more_processes(range)  # Asignar más procesos a rangos más probables
    while not found_key:
        check_ranges()  # Revisar los rangos asignados y continuar búsqueda
        redistribute_processes_if_needed()  # Reasignar procesos si un rango no parece prometedor
```

Ventajas:

- Mejor distribución del trabajo: Las claves más probables reciben más atención, lo que reduce su tiempo priorizando claves.
- Reevaluación dinámica: Si un proceso se queda sin trabajo, puede contribuir en otro rango.
- Velocidad mejorada: Reduce el tiempo total de búsqueda si las claves probables se encuentran al principio.

Alternativa 2. Búsqueda Binaria Distribuida

Con esta alternativa proponemos dividir el espacio de búsqueda utilizando un algoritmo similar a la búsqueda binaria. En lugar de asignar un rango secuencial de claves a cada proceso, el espacio de búsqueda se divide iterativamente por la mitad, permitiendo que los procesos busquen en diferentes partes del espacio clave.

Descripción del algoritmo:

1. División binaria: El proceso maestro divide el espacio de claves en dos mitades. Un proceso busca en la primera mitad y otro en la segunda.
2. Búsqueda recursiva: Si no se encuentra la clave en la primera iteración, se vuelve a dividir el espacio restante, asignando los nuevos subrangos a los procesos disponibles.
3. Detección temprana de la clave: Cuando un proceso encuentra la clave, todos los procesos detienen su búsqueda.

Pseudocódigo:

```
def binary_key_search(start_key, end_key, keyword):
   while not found_key:
      mid_key = (start_key + end_key) // 2
      if key_in_range(start_key, mid_key):
         assign_process_to_range(start_key, mid_key)
      if key_in_range(mid_key + 1, end_key):
         assign_process_to_range(mid_key + 1, end_key)
      wait_for_result()  # Detener la búsqueda si un proceso encuentra la clave
```

Ventajas:

- Exploración rápida del espacio: Al dividir iterativamente el espacio de búsqueda, se evita el sesgo hacia el inicio o fin del espacio.
- Paralelismo: Cada proceso explora un subrango con un enfoque balanceado.
- Detección rápida: Si la clave está cerca de la mitad del espacio, se encuentra rápidamente

<u>Pruebas y evaluación final</u>

Llave fácil: $(2^{56}) / 2 + 1$

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 36028797018963969
Texto cifrado: Dsta e tna pr eca de rnyecto2
Proceso 0 encontró una posible clave: 36028797018963969
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 36028797018963969
La clave fue encontrada: 36028797018963969
Tiempo total de ejecución: 0.001629 segundos
```

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

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 40532396646334464
Texto cifrado: Esta e♦ una pr♦eba de ♦royecto♦2
Proceso 0 encontró una posible clave: 40532396646334464
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 40532396646334464
La clave fue encontrada: 40532396646334464
Tiempo total de ejecución: 0.000447 segundos
```

Llave difícil: $(2^{56}) / 7 + (2^{56}) / 13$

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 20503144928365875
Texto cifrado: v♦O♦♦; F♦ZN♦♦=eQ♦

♦♦h2♦^
Proceso 0 encontró una posible clave: 20503144928365875
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 20503144928365875
La clave fue encontrada: 20503144928365875
Tiempo total de ejecución: 0.000193 segundos
```

Anexo 1 - Catálogo de funciones y librerías

Funciones:

1. decrypt

   Entradas:
   - long key: Clave utilizada para desencriptar (tipo: long).
   - char *ciph: Puntero al texto cifrado que se va a desencriptar (tipo: char*).
   - int len: Longitud del texto cifrado (tipo: int).

   Salidas:
   No tiene una salida explícita. Modifica ciph en su lugar.

   Descripción:
   Esta función desencripta un texto cifrado utilizando el algoritmo DES (Data Encryption Standard) en modo ECB (Electronic Codebook). La clave es convertida a un formato adecuado y se aplica el proceso de desencriptación.

2. tryKey

   Entradas:
   - long key: Clave a probar para desencriptar el texto (tipo: long).

- char *ciph: Puntero al texto cifrado (tipo: char*).
- int len: Longitud del texto cifrado (tipo: int).

Salidas:

- int: Devuelve 1 si se encuentra la palabra clave en el texto descifrado, 0 de lo contrario (tipo: int).

Descripción:
Esta función intenta desencriptar el texto cifrado con una clave dada y verifica si el texto descifrado contiene una palabra clave predefinida. Utiliza decrypt para el proceso de desencriptación.

3. encrypt

Entradas:

- char *input: Texto plano que se va a cifrar (tipo: char*).
- char *output: Buffer donde se almacenará el texto cifrado (tipo: char*).
- unsigned long long key: Clave utilizada para cifrar el texto (tipo: unsigned long long).

Salidas:
No tiene salida explícita. Modifica output en su lugar.

Descripción:
Esta función cifra un texto utilizando un simple algoritmo de cifrado XOR basado en la clave proporcionada. Cada carácter del texto de entrada se combina con un byte de la clave para producir el texto cifrado.

4. contains_keyword

Entradas:

- char *text: Texto en el que se busca la palabra clave (tipo: char*).
- const char *keyword: Palabra clave que se busca en el texto (tipo: const char*).

Salidas:

- int: Devuelve 1 si se encuentra la palabra clave, 0 de lo contrario (tipo: int).

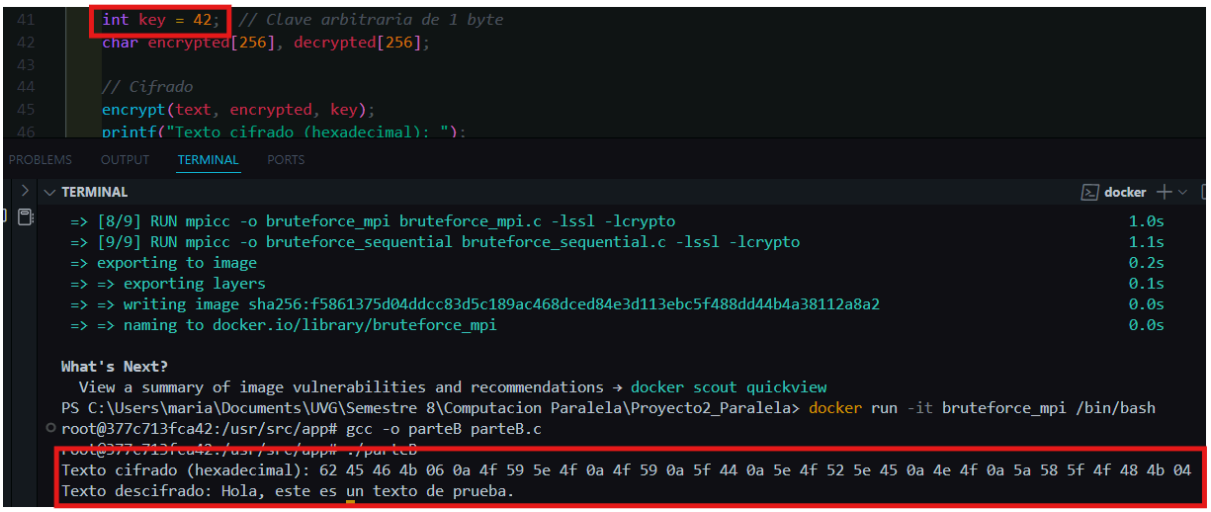
Descripción:
Esta función verifica si una palabra clave dada está presente en un texto. Utiliza la función strstr para realizar la búsqueda.

Librerías:

- <string.h>: Proporciona funciones para manejar cadenas, como strstr y memcpy.
- <stdio.h>: Proporciona funciones de entrada/salida, como printf, fgets, y fopen.

- <stdlib.h>: Proporciona funciones de utilidad, como malloc, free, y strtoull.
- <openssl/des.h>: Proporciona las funciones y tipos necesarios para trabajar con el algoritmo DES.
- <mpi.h>: Proporciona funciones y tipos para la programación paralela utilizando MPI.

Anexo 2 - Bitácora de Pruebas




```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 123456
Texto cifrado: ♣♦ua es 5♦` prue"♦!de pr/♦dcto 2
Proceso 0 encontró una posible clave: 123456
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 123456
La clave fue encontrada: 123456
Tiempo total de ejecución: 0.000140 segundos
```

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 18014398509481983
Texto cifrado: ♦♦♦♦tL ♦♦♦⊐♦Je♦♦□♦♦Or♦♦♦♦♦♦▼2
Proceso 0 encontró una posible clave: 18014398509481983
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 18014398509481983
La clave fue encontrada: 18014398509481983
Tiempo total de ejecución: 0.000299 segundos
```

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 18014398509481984
Texto cifrado: Esta e3 una pr5eba de 0royecto`2
Proceso 0 encontró una posible clave: 18014398509481984
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 18014398509481984
La clave fue encontrada: 18014398509481984
Tiempo total de ejecución: 0.000217 segundos
```

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 36028797018963969
Texto cifrado: Dsta e tna pr eca de rnyecto2
Proceso 0 encontró una posible clave: 36028797018963969
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 36028797018963969
La clave fue encontrada: 36028797018963969
Tiempo total de ejecución: 0.001629 segundos
```

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 40532396646334464
Texto cifrado: Esta e una pr eba de royecto2
Proceso 0 encontró una posible clave: 40532396646334464
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 40532396646334464
La clave fue encontrada: 40532396646334464
Tiempo total de ejecución: 0.000447 segundos
```

```
newparalela@09fa30b7750f:/usr/src/app$ mpirun -np 4 ./parteB 20503144928365875
Texto cifrado: v O ; F ZN =e

 h2 ^
Proceso 0 encontró una posible clave: 20503144928365875
Texto descifrado: Esta es una prueba de proyecto 2
Proceso 0 confirmó la clave correcta: 20503144928365875
La clave fue encontrada: 20503144928365875
Tiempo total de ejecución: 0.000193 segundos
```

```
newparalela@35a297667f74:/usr/src/app$ mpirun -np 4 ./adaptive_search
```

```
newparalela@029bc731bbb5:/usr/src/app$ mpirun -np 4 ./binary_search
```

Referencias

Peter, L. (2024). *Data Encryption Standard (DES)*. TechTarget. Recuperado de https://www.techtarget.com/searchsecurity/definition/Data-Encryption-Standard

Christine S.. (2012). *Heuristic optimization algorithms*. Utah State University. Recuperado de: https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1048&context=ecstatic_all

Universidad Politécnica de Madrid (2024). *Búsqueda binaria*. Escuela Técnica Superior de Ingenieros de Telecomunicación, Universidad Politécnica de Madrid. Recuperado de: https://dcain.etsin.upm.es/~carlos/bookCNP/02.04_BusquedaBinaria.html