UNIVERSIDAD DEL VALLE DE GUATEMALA
CC3069 - Computación Paralela y Distribuida
Sección 10
Ing. Miguel Novella

Proyecto 2 - Cifrar y descifrar DES con MPI

Diego Arredondo 19422
Julio Herrera 19402
Diego Álvarez 19498
Grupo 9

Link al repositorio: [Proyecto2 MPI DES]

GUATEMALA, 14 de mayo de 2023

Índice

Introducción

En el presente trabajo en el que se presenta un programa de encripción y desencripción utilizando el algoritmo de DES (Estándar de Encripción de Datos, por sus siglas en inglés) y 3 enfoques distintos para aplicar fuerza bruta y encontrar las llaves necesarias para la encripción y desencripción de datos.

El algoritmo de DES es uno de los más utilizados para la encripción de datos debido a su seguridad y eficiencia, pero su principal limitación es que puede ser vulnerable a los ataques de fuerza bruta. Para superar esta limitación, en este trabajo se presentan dos algoritmos adicionales para encontrar las llaves necesarias para la encripción y desencripción de datos, que se compararán con el enfoque naive utilizado originalmente en el programa.

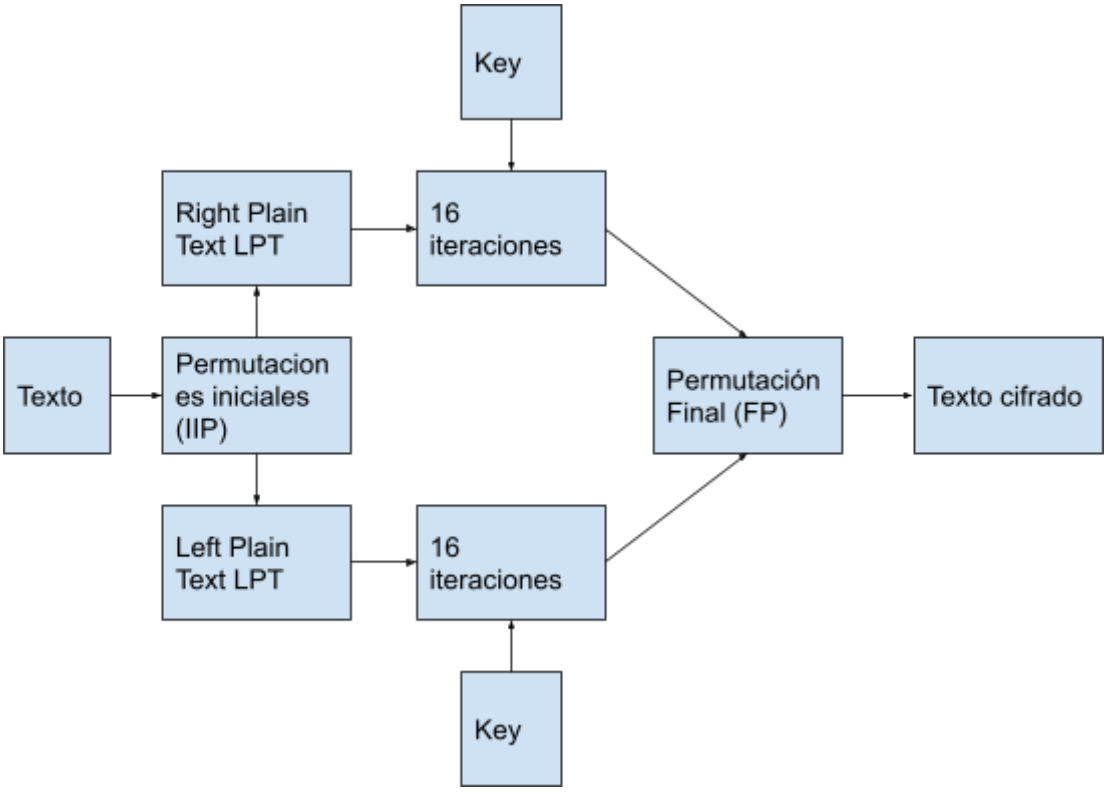
Además, se realizará una comparación del rendimiento y speedup de los tres algoritmos para encontrar la llave. El objetivo es evaluar cuál de los tres métodos proporciona la mejor combinación de seguridad y eficiencia para el programa de encripción y desencripción.

Cuerpo

Parte A

1. Investigue sobre DES y describa los pasos requeridos para cifrar/descifrar un texto.
    a. El método de codificación conocido como DES fue creado por IBM en los años setenta y fue adoptado como un estándar por el Instituto Nacional de Estándares y Tecnología de los Estados Unidos en 1977. Este fue el primer método de encriptación utilizado en la informática y, aunque se consideraba muy seguro en su época, ahora se considera anticuado debido a que las claves eran cortas, de solo 64 bits (56 bits útiles). Aunque no tiene patente, el DES fue descifrado en 1994 en un criptoanálisis experimental que tomó 50 días utilizando 12 estaciones HP-9735. La tecnología actual ha hecho que una clave DES sea muy vulnerable a los ataques de fuerza bruta, lo que significa que puede descifrarse en solo unos pocos segundos.

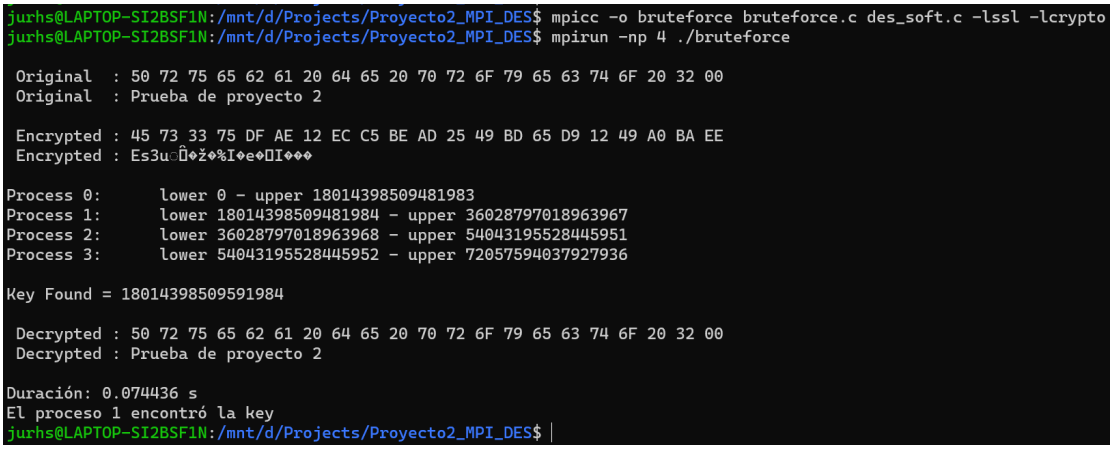
2. Dibuje un diagrama de flujo describiendo el algoritmo DES



3. Luego de trabajar un poco con DES, haremos una prueba de código como introducción y base para el resto del desarrollo. Haga funcionar el programa bruteforce.c Es probable que necesite una biblioteca que reemplace a “rpc/des_crypt.h” en caso su computadora/instalación/sistema operativo lo requiera, para ello puede utilizar cualquier librería que desee/encuentre.(shubhamupadhyay, 2023)

   Se reemplazó la librería “rpc/des_crypt.h” y se utilizó “openssl/des.h” para realizar la encriptación y desencriptación DES, sin embargo, se utilizó el código de la librería “rpc/des_crypt.h” para utilizar la función “des_setparity” y aplicar la paridad a la llave.

   Aplicando los cambios para utilizar la librería “openssl/des.h” con el mismo approach se logró ejecutar correctamente el programa donde se encripta un texto y utilizando fuerza bruta de manera paralela (Naive) se obtiene la llave original (número de tipo long). El resultado de la ejecución del programa es el siguiente:



4. Una vez funcionando su programa base, explique, mediante diagramas, texto, dibujos, etc., cómo funcionan las rutinas (o la equivalente si uso otra librería en caso de decrypt/encrypt):

   a. decrypt (key, *ciph, len) y encrypt (key, *ciph, len)
      i. Para nuestro caso, usamos la librería openssl, el proceso que hace el método de decrypt es:
         1. La función toma cuatro argumentos:
            a. Una clave de cifrado de tipo long (key)
            b. Un texto cifrado (cipher)
            c. La longitud del texto cifrado (len)
            d. Un puntero al buffer donde se almacenará el texto descifrado (text)
         2. El vector de inicialización (iv) es un valor aleatorio que se utiliza para encriptar el primer bloque de texto. En este código, se crea un segundo vector de inicialización (iv) y se establece su paridad para asegurarse de que tiene un número impar de bits.

3. A continuación, se genera una clave de cifrado DES a partir de la clave proporcionada utilizando la función set_key(). La función también establece la paridad de la clave para asegurarse de que tiene un número impar de bits.
4. Finalmente, se utiliza la función DES_ncbc_encrypt() en modo DES_DECRYPT para descifrar el texto cifrado. La función toma el texto cifrado, la longitud del texto cifrado, la clave de cifrado triple DES, el vector de inicialización (iv) y la dirección del buffer de texto descifrado como argumentos. La función realiza la operación de descifrado utilizando el modo CBC y almacena el resultado en el buffer de texto descifrado.
   Es importante destacar que el vector de inicialización (iv) es un valor crítico en la seguridad del cifrado, y que debe ser diferente para cada bloque de texto cifrado para evitar ataques. En este código, sin embargo, el vector de inicialización (iv) se inicializa con ceros en lugar de ser generado aleatoriamente, lo que puede comprometer la seguridad del cifrado.

ii. No existe una función encrypt ya que solo se realizan los pasos al iniciar el programa, pero el proceso para encriptar es muy similar al explicado anteriormente para desencriptar.
1. Primero se crea un vector de inicialización (iv) con ceros y se le aplica paridad.
2. Se crea la llave para DES usando la función set_key enviándole el valor de tipo long que es la llave que queremos encontrar con la fuerza bruta (dentro de la función set_key se aplica paridad a la llave).
3. Se lee el archivo de texto donde se almacena el mensaje a encriptar y se guarda en una variable así como el tamaño del texto.
4. Se crea una variable que almacenará el resultado del cifrado (cipher) y se llama a la función DES_ncbc_encrypt con el modo DES_ENCRYPT en su último parámetro, así como el texto a cifrar, dónde se guardará el cifrado, el tamaño del mensaje, la llave de tipo DES_key_schedule y el vector de inicialización.

b. tryKey (key, *ciph, len)
   i. La función intenta descifrar un mensaje cifrado utilizando una clave proporcionada. Después de descifrar el mensaje, busca una subcadena en el texto descifrado y devuelve “true" si la subcadena se encuentra y “false" si no se encuentra.

c. memcpy
   i. Para nuestro caso, se usó la función memset de openssl
      1. Los argumentos de la función son:

a. Un puntero al bloque de memoria que se debe inicializar.
b. El valor a establecer en cada byte del bloque de memoria.
c. El número de bytes que se deben establecer en el bloque de memoria.

2. Se utiliza para establecer un bloque de memoria en un valor específico. En el caso de OpenSSL, esta función se utiliza para inicializar estructuras de datos que se utilizan para el cifrado y descifrado de datos.

d. strstr
   i. Los argumentos de la función son:
      1. Haystack: la cadena en la que se busca la subcadena.
      2. Needle: la subcadena que se busca.
   ii. La función devuelve un puntero a la primera aparición de la subcadena Needle en la cadena Haystack. Si la subcadena no se encuentra en la cadena, la función devuelve un puntero nulo (NULL).

5. Describa y explique el uso y flujo de comunicación de las primitivas de MPI:

   a. MPI_Irecv: Este permite realizar una recepción asíncrona de mensajes en un programa paralelo. Los argumentos que reciben son:
      i. buf es un puntero al búfer donde se almacenará el mensaje.
      ii. count es el número de elementos en el búfer.
      iii. datatype es el tipo de datos que se espera recibir.
      iv. source es el identificador del proceso que envía el mensaje. Si se establece en MPI_ANY_SOURCE, la función aceptará un mensaje de cualquier proceso.
      v. tag es una etiqueta que se utiliza para identificar el mensaje. Si se establece en MPI_ANY_TAG, la función aceptará un mensaje con cualquier etiqueta.
      vi. comm es el comunicador que se utilizará para recibir el mensaje.
      vii. request es un objeto que se utiliza para controlar la operación asíncrona.

         1. Una vez que se llama a MPI_Irecv, la función devuelve inmediatamente y el programa puede continuar ejecutándose mientras espera la llegada del mensaje. El objeto request se utiliza para comprobar el estado de la operación de recepción asíncrona. Esto se puede hacer con la función MPI_Test, que comprueba si el mensaje ha llegado al búfer.

2. Una vez que el mensaje ha sido recibido, se puede acceder a los datos en el búfer y el objeto request se puede liberar con la función MPI_Request_free.

b. MPI_Send: permite enviar un mensaje de un proceso a otro en un programa paralelo. Los argumentos que reciben son:
   i. buf es un puntero al búfer que contiene los datos que se van a enviar.
   ii. count es el número de elementos en el búfer.
   iii. datatype es el tipo de datos que se está enviando.
   iv. dest es el identificador del proceso destino. El proceso destino debe estar dentro del comunicador especificado en comm.
   v. tag es una etiqueta que se utiliza para identificar el mensaje.
   vi. comm es el comunicador que se utilizará para enviar el mensaje.
      1. Una vez que se llama a MPI_Send, el proceso se bloquea hasta que el mensaje se haya enviado con éxito al proceso destino. Esto significa que el proceso no puede realizar ninguna otra tarea hasta que la operación de envío se haya completado.
      2. Si el proceso destino no está preparado para recibir el mensaje cuando se llama a MPI_Send, el proceso se bloqueará hasta que el proceso destino esté listo para recibirlo. Esto se conoce como un envío sincrónico, ya que el proceso de envío espera la confirmación de que el mensaje ha sido recibido antes de continuar.

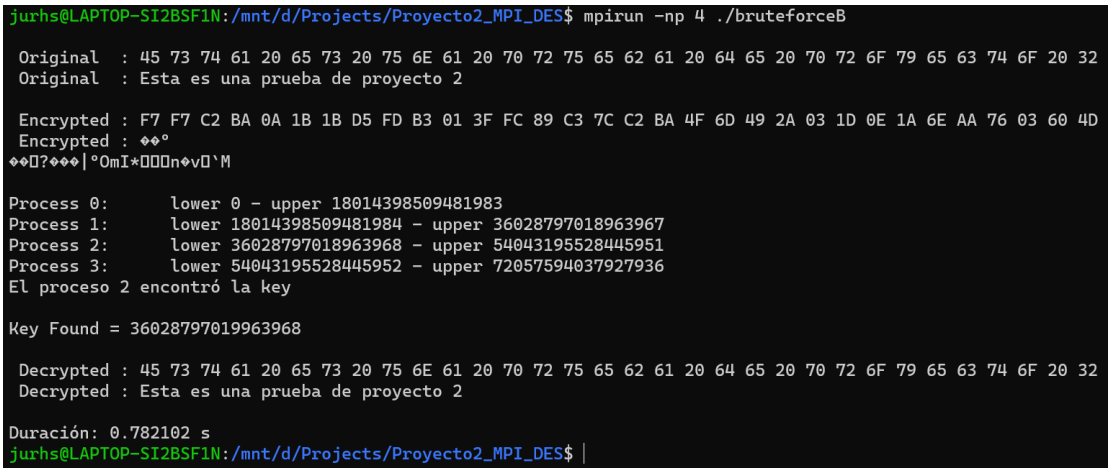
   (Open MPI, 2021)

c. MPI_Wait: permite esperar hasta que se complete una operación asíncrona de comunicación en un programa paralelo. Los argumentos que reciben son:
   i. request: es un objeto que se utiliza para controlar la operación asíncrona.
   ii. status: es un objeto que se utiliza para devolver información sobre el estado de la operación asíncrona.
      1. MPI_Wait se utiliza después de llamar a una función asíncrona de MPI, como MPI_Isend o MPI_Irecv. Estas funciones devuelven un objeto de solicitud (MPI_Request) que se utiliza para comprobar el estado de la operación asíncrona.
      2. MPI_Wait bloquea el proceso hasta que se complete la operación asíncrona.
      3. Una vez que se completa la operación, MPI_Wait devuelve un código de éxito y se puede acceder a los datos recibidos en el búfer. El objeto status también se puede utilizar para obtener información sobre la operación, como el identificador del proceso que envió el mensaje y la etiqueta del mensaje.

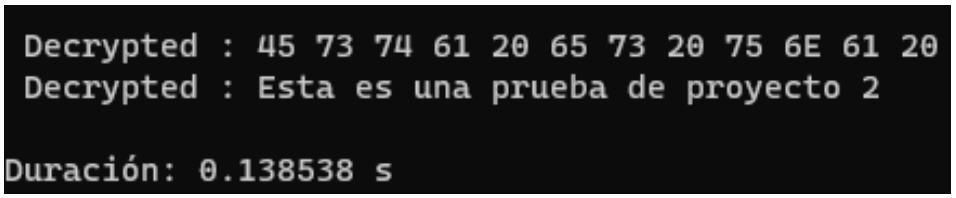
Parte B

1. Ya que estamos familiarizados con el temario, vamos a analizar el problema más a fondo. Modifique su programa para que cifre un texto cargado desde un archivo (.txt) usando una llave privada arbitraria (como parámetro). Muestra una captura de pantalla evidenciando que puede cifrar y descifrar un texto sencillo (una oración) con una clave sencilla (por ejemplo 42).

   Se modificó el programa para leer un archivo .txt, por lo que se necesita crear el archivo, en este caso contiene el texto “Esta es una prueba de proyecto 2”. El nuevo código se guardó en el archivo bruteforceB.c, el cual se compiló y da como resultado el correcto funcionamiento del programa:



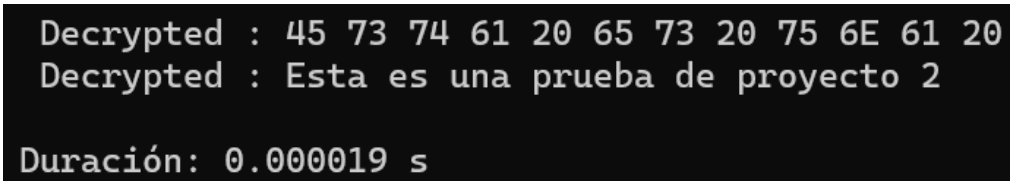
2. Una vez listo el paso anterior, proceder a hacer las siguientes pruebas, evidenciando todo en su reporte. Para todas ellas utilice 4 procesos (-np 4). El texto a cifrar/descifrar: “Esta es una prueba de proyecto 2”. La palabra clave a buscar es: “es una prueba de”:
   a. Mida el tiempo de ejecución en romper el código usando la llave 123456L
      La llave fue encontrada en un tiempo relativamente corto ya que el número a buscar es alcanzado rápidamente por el proceso 0, esto se puede traducir como que se tardó 0.138538 segundos en probar 123456 llaves.



   b. Mida el tiempo de ejecución en romper el código usando la llave 18014398509481983L. [spoiler: se tardará mucho, si es que(256/4) termina, no se ofusquen si no termina].
      i. Para este caso, la llave es muy complicada, por lo mismo el programa nunca terminó.

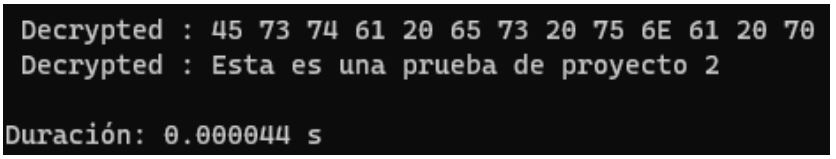
   c. Mida el tiempo de ejecución en romper el código usando la llave 18014398509481984L.

Para esta llave, el tiempo que tardó en encontrarla fue menor que el del inciso a, esto ya que la llave es igual al número desde el que empieza a probar el proceso 1, por lo tanto la primera llave que prueba es la correcta.

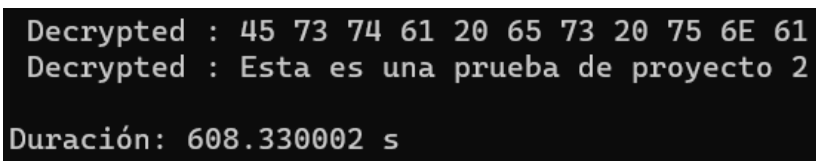


d. Reflexione lo observado y el comportamiento del tiempo en función de la llave.
    i. El comportamiento del tiempo en función de la llave es proporcional a la distancia desde el punto en el que empieza a probar cada proceso hasta llegar a probar dicha llave. Con las pruebas, el tiempo para poder encontrar una llave muy cerca del punto de inicio de cada proceso $((Nº Proceso \cdot (2^{56}/4)) + 1)$, era muy bajo, pero cada vez que la llave se alejaba más, el tiempo para encontrarla aumentaba significativamente hasta el punto que el programa no logró terminar de ejecutarse.

e. Una llave fácil de encontrar, por ejemplo, con valor de (2^56) / 2 + 1
    i. Llave: 36028797018963969L, con 4 procesos. Ésta igualmente es el número desde el que empieza a probar, en este caso el proceso 2, por lo tanto la encuentra instantáneamente.



f. Una llave medianamente difícil de encontrar, por ejemplo, con valor de (2^56) / 2 + (2^56) / 8
    i. La llave mencionada (7,205,759,403,792,793) no llega a encontrarse debido a que se aleja mucho del punto inicial del proceso 3. Se utilizó en cambio la llave: 599481984L, con 4 procesos, para comparar cuánto se tarda una llave más grande, el resultado es el siguiente:



g. Una llave difícil de encontrar, por ejemplo, con valor de (2^56) / 7 + (2^56) / 13 aproximados al entero superior
    i. Llave: 10428198925439226L
        1. Con esta llave, con 4 procesos, no terminó el programa por la complejidad de la misma. Considerando que esta llevaría 10,428,198,925,439,226 iteraciones hasta encontrar el

resultado y el tiempo que tomó la llave anterior, encontrar esta llave tomaría aproximadamente 8.889 días con el poder computacional utilizado para este proyecto.

3. (opcional, fuertemente sugerido ya que les tocará hacer el proceso 2+ veces en siguientes pasos) Demuestre que el valor esperado para el tiempo paralelo es lo indicado en la Ecuación 1 del párrafo anterior.

$$E[tPar(n,k)] = \sum_{i}^{\Pi} x_i p_i = \frac{2^{55}}{n} + 1/2 \quad \textbf{(1)}$$

4. Como podemos ver, el approach “naive” no es el mejor posible. Proponga, analice, e implemente 2 opciones alternativas al acercamiento “naive”. Tenga como objetivo en mente encontrar un algoritmo que tenga mejor “tiempo paralelo esperado” que la versión “naive” demostrada en ecuación (1). Para cada una no olvide:

Algoritmo N1:

a. Describir el acercamiento propuesto, se puede apoyar con diagramas de flujo, pseudocódigo, o algoritmo descriptivo.
Este algoritmo parte de la idea naive de distribución de trabajo, dividiendo en partes la carga de trabajo entre los procesos existentes. La diferencia recae en la manera de comunicarse entre los procesos, el proceso cero distribuye la carga y envía a cada proceso la cantidad que le corresponde. Cada uno opera y en el momento en que es encontrada la solución se avisan que ya deben detenerse. En lugar de esperar al proceso cero para mostrar el valor, el proceso que ha encontrado la llave es el que comunica.

b. Derivar el valor esperado de tPar(n,k) de ese acercamiento y compararlo con el del acercamiento “naive”. Además del valor esperado, discuta su procedimiento y razonamiento. Mencione cómo se comporta el speedup en este acercamiento.

Naive: tPar(4,56) = (2^56)/4 + 1
Algoritmo N1: tPar(4,56) = (2^56)/4 + 1

Ambos tienen el mismo valor esperado de tiempo, ya que se distribuye el trabajo equitativamente y de la misma manera. El speedup con este acercamiento es de 0.97 lo que nos indica que no hubo mejora en el tiempo de ejecución. Concluyendo de esta manera que este acercamiento no funcionó.

Este valor es correcto debido a que todas las llaves posibles son combinaciones de números entre 0 y 2^56. En este algoritmo se inicia desde

el primer valor del rango y aumenta en uno cada iteración, teniendo así en total (1 / 2^56) probabilidad de encontrar el número en todos los rangos creados. Teniendo así el mismo comportamiento que en el naive, por lo tanto se puede decir que el tiempo que toma esta implementación es tPar(4,56) = (2^56) / 4 + 1.

c. Impleméntelo en código y pruébelo con 3-4 llaves (fáciles, medianas, difíciles). Compare el tiempo medido con el tiempo pronosticado por su función tPar(n,k).
Tiempo pronosticado: (2^56) / 4 + 1
   i. Fácil, llave: 360279
      Tiempo 1.46s

```
Key Found 1 = 360279 rank = 0

Key Found = 360279

 Decrypted : 45 73 74 61 20 65 73 20 75 6E 61
 Decrypted : Esta es una prueba de proyecto 2

Duración: 1.466858 s
```

   ii. Mediana, llave: 36028797019963968
      Tiempo: 2.87s

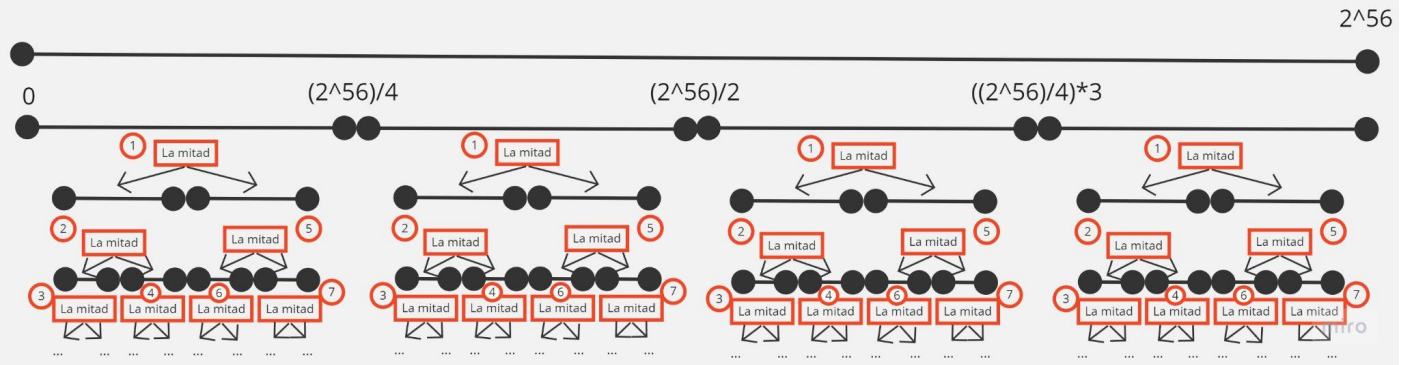
```
Key Found 1 = 36028797019963968 rank = 2

Key Found = 36028797019963968

 Decrypted : 45 73 74 61 20 65 73 20 75 6E 61
 Decrypted : Esta es una prueba de proyecto 2

Duración: 2.870343 s
```

   iii. Difícil, llave: 18014398509481983

Algoritmo N2: Árbol secuencial

d. Describir el acercamiento propuesto, se puede apoyar con diagramas de flujo, pseudocódigo, o algoritmo descriptivo.

Este acercamiento consiste en la misma división de trabajo inicial entre los procesos, es decir que si hay 4 procesos cada proceso probará (2^56 / 4) números, sin embargo, ahora cada proceso empezará probando el número que está a la mitad de ese rango y mientras el número probado no sea la llave correcta, dividirá el rango en 2, lo que queda al lado izquierdo y lo que queda al lado derecho, de cada rango restante volverá a probar la mitad y por lo tanto el rango se volverá a dividir en 2. Así se irá probando cada mitad de rango hasta llegar a un rango donde haya solo un número. Algo importante a resaltar es que cada proceso irá probando siempre cada rango que queda a la izquierda, tal y como se recorre un árbol binario de izquierda a derecha.



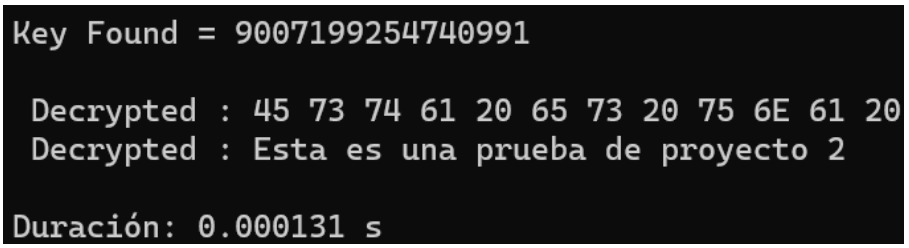
Esta implementación requiere recursividad por lo tanto se crea la función que recibe un rango, prueba la llave y vuelve a llamar a la misma función con cada restante de lado izquierdo primero y luego el derecho. Si ya no hay nada que probar retorna o comunica que ya encontró la llave en cada proceso. Al inicio de la función, es decir antes de probar la mitad de un rango, se verifica si alguien ya encontró la llave.

e. Derivar el valor esperado de tPar(n,k) de ese acercamiento y compararlo con el del acercamiento "naive". Además del valor esperado, discuta su procedimiento y razonamiento. Mencione cómo se comporta el speedup en este acercamiento.

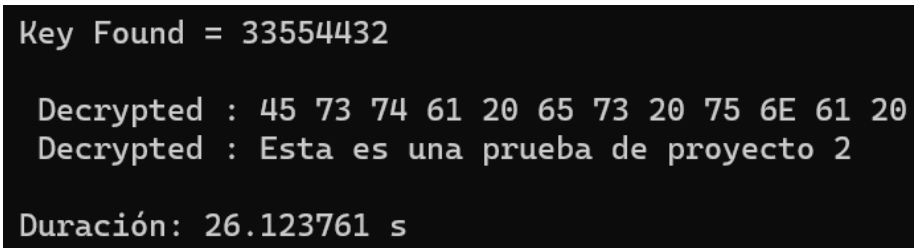
Este acercamiento también divide el trabajo inicial igualitariamente entre el número de procesos así que podríamos empezar diciendo que $\frac{2^{55}}{n}$. Pero ahora cada proceso en lugar de revisar desde el inicio del rango e ir incrementando uno a cada prueba, toma el número a la mitad del rango, igualmente se tiene la misma probabilidad de que el número escogido sea la llave, es decir (1 / 2^56). Como prueba siempre el lado izquierdo de los rangos restantes, el tiempo que tardará en llegar a descifrar si la llave escogida es el número al final del rango inicial de cada proceso, es el mismo tiempo que el naive. Sin embargo esta implementación beneficia a cada x/2 número de cada $(Nº Proceso \cdot (2^{56}/n)$, es decir la secuencia de mitades, o como se mencionó antes, siguiendo el orden en el que se recorre un árbol binario de izquierda a derecha siendo este el comportamiento del speedup en este acercamiento. Por lo tanto se puede decir que el tiempo que toma esta implementación es tPar(4,56) = (2^56) / 4 + 1.

f. Impleméntelo en código y pruébelo con 3-4 llaves (fáciles, medianas, difíciles). Compare el tiempo medido con el tiempo pronosticado por su función tPar(n,k).
Tiempo pronosticado: (2^56) / 4 + 1
   i. Fácil, llave: 9007199254740991L.
      Este es el número que se encuentra a la mitad del rango inicial del proceso 1, es decir el primer número que prueba. En realidad el número que está a la mitad es el mismo + .5 pero como agarra el

entero se consideró este. Por lo tanto se espera que se haga instantáneo.



ii. Mediana, llave: 1L. Siguiendo la forma en la que se recorre este árbol, una llave medianamente compleja de encontrar sería 536870912. Este número es ((2^56)/4)/2 19 veces, más uno, esto nos asegura que será uno de los números aproximadamente a la izquierda y centro del árbol. El tiempo e iteraciones esperadas para que llegue a dicho número es 33554485t.

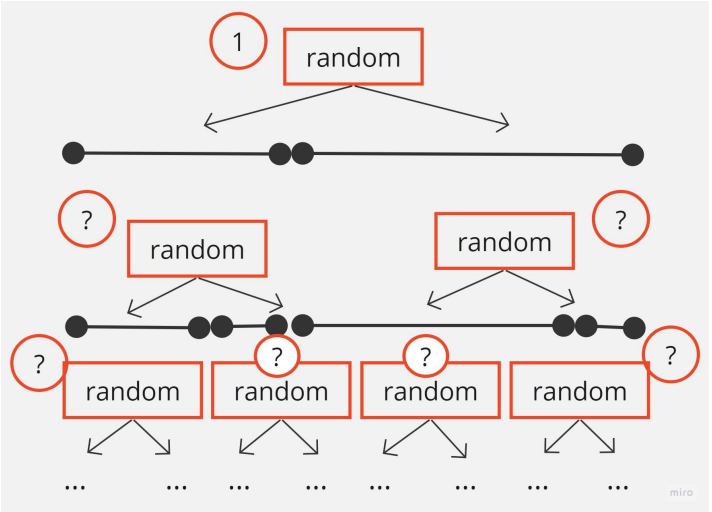


iii. Difícil, llave: 18014398509481983L. Cómo se mencionó, el número al final de cada rango inicial de cada proceso es el último en probarse, por lo tanto si se elige esta llave, el tiempo sería el mismo que tomaría en el naive. Para esta prueba no se llegó a encontrar la llave.

Algoritmo N3: Árbol Random

g. Describir el acercamiento propuesto, se puede apoyar con diagramas de flujo, pseudocódigo, o algoritmo descriptivo.

El acercamiento de este algoritmo es muy parecido al anterior “árbol secuencial” en la forma en la que se divide el trabajo, sin embargo, ahora ya no se elegirá la mitad del rango para probar dicho número, sino que se elegirá un número random de entre todo el rango. Igualmente de forma recursiva se probará luego el rango restante de la izquierda y el de la derecha pero ahora también se elegirá de forma aleatoria cuál se prueba primero. La forma en la que se divide cada árbol puede verse como la siguiente:



Ahora no se sabe realmente el orden en el que se recorrerá el árbol, empieza desde la raíz el cual es un random pero no se sabe si recorrerá primero el subnodo (rango restante) izquierdo o derecho. También se agregaron validaciones por si el número random elegido es igual al inicio o fin del rango, prácticamente sólo se hará recursividad al único rango restante en este caso.

De esta forma cada número tendrá la misma probabilidad de salir "teóricamente" pero en la práctica cada probabilidad depende de la anterior, es decir que dependiendo de qué rango resulta por recorrer primero (izquierdo o derecho) los números tendrán más probabilidad de salir si están dentro del rango que se eligió (aleatoriamente).

Este acercamiento hace que ya no se dependa de la llave elegida para conocer el tiempo aproximado, ya que al inicio cada número tendrá la misma probabilidad de salir que los otros.

h. Derivar el valor esperado de tPar(n,k) de ese acercamiento y compararlo con el del acercamiento "naive". Además del valor esperado, discuta su procedimiento y razonamiento. Mencione cómo se comporta el speedup en este acercamiento.

   Como se mencionó, ahora ya no se depende de la llave y sólo sabemos que cada número tiene las mismas probabilidades debido a que se elige aleatoriamente, se puede decir que tPar(n) = 1 / (2^56) / n. El speedup en este acercamiento comparado con naive, beneficia a los números que estaban al final del rango, pero ahora se tiene una reducción a los números que estaban al inicio del rango.

i. Impleméntelo en código y pruébelo con 3-4 llaves (fáciles, medianas, difíciles). Compare el tiempo medido con el tiempo pronosticado por su función tPar(n,k).
   Tiempo pronosticado: (2^56) / 4 + 1

Para cada llave probada se esperará un máximo de 1 hora.

i. Llave 1: 134227728L. Luego de 1 hora, no se encontró la llave.
ii. Llave 2: 18014398509481984L. Luego de 1 hora, no se encontró la llave.
iii. Llave 3: 10L. Luego de 1 hora, no se encontró la llave.

Discusión

El proceso de encontrar la llave con la que se encripta un texto utilizando fuerza bruta es un algoritmo que puede aprovechar muy bien del paralelismo. Pero uno de los primeros cuestionamientos que surgen al pensar en este tipo de "ataque" es qué tanto tardará en encontrar la llave original; sin tener ninguna pista de la llave original esto podría tomar mucho tiempo, cuando se utiliza fuerza bruta para descifrar contraseñas se puede intuir cuales son las contraseñas más utilizadas o probar combinaciones comunes de lenguaje primero para intentar dar lo antes posible con la original, pero en este proyecto la llave que se quiere encontrar es un número de tipo long, esto hace que no tengamos ninguna inferencia sobre qué podemos probar primero, lo único que sabemos es que la llave es un número entre 0 y 2^56.

La versión naive es lo "primero" que se nos puede ocurrir, probar todos los números en orden, esperando que la llave utilizada no sea de los últimos. Desde aquí podemos ver la gran ventaja de utilizar paralelismo ya que se divide el trabajo entre los procesos, si tenemos 4 procesos el rango total se dividirá en 4 y cada proceso empezará a probar desde el número $(Nº Proceso \cdot (2^{56}/4)) + 1$, por lo que no necesariamente los números más cercanos a 0 tendrán más probabilidad de ser probados. Esta probabilidad de obtener el número aumenta linealmente según la cantidad de procesadores que usemos.

Como ya se mencionó, no tenemos ninguna información sobre la llave, por lo que no importa el enfoque que tomemos o cómo recorramos el rango total de números, todos tienen la misma probabilidad de salir, pudiendo tomar menos de 1 milisegundo en encontrarse o hasta varios días.

Parte importante de este proyecto es conocer cómo interactúa cada acercamiento sabiendo cuál es la llave, por lo tanto si conocemos la llave original podemos saber cómo se comportará cada implementación. En el acercamiento naive y el algoritmo 1, sabemos que se benefician todas aquellas llaves más cercanas a $Nº Proceso \cdot (2^{56}/n)$ donde n es el número de procesos utilizados y así se verificó en las pruebas, que mientras más se aleja de cada uno de estos números, más se tarda en encontrar la llave.

Por otro lado, la implementación más acorde si no conocemos nada de información sobre la llave original es la de ir probando aleatoriamente cada número en el rango original, por lo que cada uno tiene la misma probabilidad de salir. Con este algoritmo, aunque tengamos información de la llave, no se podría saber si la encontrará antes que otras ya que es aleatorio, por lo que este es el mejor enfoque si no conocemos absolutamente nada de la llave original.

Por último tenemos el algoritmo N2: Árbol secuencial. Este acercamiento puede ser utilizado cuando se tiene cierta información sobre la llave original ya que la forma en la que se recorre es como se recorrería un árbol binario de manera pre-order desde la raíz, donde cada nodo es la mitad de un rango. Por lo que sí sabemos que la llave original está dentro de cierto rango de números, este beneficia a los que están más a la izquierda pero no de manera secuencial, si bien lleva siempre el mismo orden tiene es más amplio en los valores que prueba primero.

Conclusiones

- Se concluye que el tiempo tomado para encontrar una llave, es proporcional a la complejidad de la misma.
- La forma en la que se comunicaron los procesos y cómo estos comparten sus datos no fue un factor por el cual el tiempo haya sido reducido.

Recomendaciones

- La distribución de trabajo pudo tomar una diferente implementación. Alojar globalmente las posibilidades de llaves y que cada proceso tome una porción aleatoria de esta (no muy grande) y cuando haya terminado de procesar volver a tomar datos si no ha encontrado la llave.

Anexo 1 − Catálogo de funciones y librerías

- print_result:
    - Entrada:
        - header (const char *): Un puntero a una cadena de caracteres que se utiliza para imprimir un encabezado antes de mostrar los datos.
        - data (const void *): Un puntero genérico a los datos que se deben imprimir.
        - datalen (int): La longitud de los datos que se deben imprimir.
    - Salida:
        - Primero, se imprime el encabezado seguido de los datos en formato hexadecimal.
        - Después, se imprime el encabezado seguido de los datos en formato ASCII.
    - Descripción:
        - Este método se encarga de imprimir en la consola la información de los datos que se le pasan como argumentos
        - Para imprimir los datos en formato hexadecimal, se utiliza un bucle que recorre los bytes de los datos uno por uno, los convierte a su

valor hexadecimal correspondiente y los imprime en la consola separados por un espacio.
        - Para imprimir los datos en formato ASCII, se utiliza otro bucle similar que recorre los bytes de los datos uno por uno, los convierte en su correspondiente caracter ASCII y los imprime en la consola sin separación alguna. Finalmente, se imprime un salto de línea para separar la salida de este método de la siguiente salida que se pueda imprimir.
- Trykey:
    - Entrada:
        - key: representa la clave a probar.
        - ciph: representa el texto cifrado que se desea descifrar.
        - len: indica la longitud de la cadena ciph.
    - Salida:
        - Un entero que indica si se ha encontrado la cadena search_text en el texto descifrado. Si la cadena se encuentra, el valor de retorno es 1. De lo contrario, el valor de retorno es 0.
    - Descripción:
        - Este método intenta descifrar el texto cifrado ciph utilizando la clave key. El resultado del descifrado se almacena en un arreglo de caracteres llamado text. A continuación, se busca la cadena search_text en el texto descifrado utilizando la función strstr(). Si se encuentra la cadena, el valor de retorno es 1. De lo contrario, el valor de retorno es 0.
- decrypt:
    - Entrada:
        - Una llave de cifrado (key) en formato long.
        - Un texto cifrado (ciph) en formato char*.
        - La longitud (len) del texto cifrado.
        - Un puntero a un array de unsigned char (text) donde se almacenará el texto descifrado.
    - Salida:
        - No hay un valor de retorno. El texto descifrado se almacenará en el array de unsigned char apuntado por el parámetro "text".
    - Descripción:
        - Este método implementa el algoritmo de cifrado Triple DES (3DES) en modo CBC (Cipher Block Chaining) para descifrar un texto cifrado utilizando la llave proporcionada. El texto cifrado se encuentra en el array de caracteres "ciph", mientras que el texto descifrado se almacenará en el array de unsigned char apuntado por el parámetro "text".
        - Primero, se inicializa un vector de inicialización (IV) para el modo CBC con ceros y se establece su paridad impar utilizando la función "DES_set_odd_parity()".

        - A continuación, se establece la llave de cifrado utilizando la función "set_key()" y se crea una estructura de clave de cifrado (SchKey2) utilizando la función "DES_key_schedule()".
        - Por último, se utiliza la función "DES_ncbc_encrypt()" para descifrar el texto cifrado utilizando la llave y el vector de inicialización previamente establecidos, y el texto descifrado se almacena en el array de unsigned char apuntado por el parámetro "text".
- set_key:
    - Entrada
        - key de tipo long: Es la clave que se utilizará para cifrar o descifrar los datos.
        - SchKey de tipo DES_key_schedule *: Es un puntero al objeto DES_key_schedule, que es una estructura utilizada por el algoritmo de cifrado DES para almacenar la clave.
        - original de tipo int: Si se establece en 1, se mostrará un mensaje de advertencia en caso de que la clave sea una "weak key" (clave débil).
    - Salida:
        - Este método no tiene un valor de retorno explícito, ya que está diseñado para modificar el valor de SchKey directamente.
    - Descripción:
        - Este método toma una clave key de 64 bits, la ajusta para cumplir con el esquema de paridad de clave de DES, la convierte en una clave de tipo DES_cblock y la almacena en la estructura SchKey. También verifica si la clave es débil o no, y muestra un mensaje de advertencia si es necesario.
        - La función des_setparity se utiliza para ajustar la paridad de la clave de 64 bits k (que se obtiene de la variable key). A continuación, la clave se convierte en una clave de tipo DES_cblock, que es una estructura de 8 bytes utilizada por el algoritmo de cifrado DES. Esto se hace mediante la función memcpy, que copia 8 bytes desde la dirección de memoria de k a la dirección de memoria de Key.
        - A continuación, se utiliza la función DES_set_odd_parity para asegurarse de que la clave tenga una paridad impar, que es un requisito del algoritmo de cifrado DES. Después, se llama a la función DES_set_key_checked para verificar si la clave es débil o no. Si la clave es débil y el parámetro original se establece en 1, se imprimirá un mensaje de advertencia en la consola. La función DES_set_key_checked devuelve -2 si la clave es débil.
- main:
    - Entrada: nada
    - Salida: int (exit del programa).
    - Descripción:
        - Indica el flujo de MPI para correr el programa, y las configuraciones de lo que se va a utilizar. Indica las funciones que se ejecutarán al momento de encriptar o desencriptar datos. Por último inicia el loop principal y retorna 0 al terminar.

- test_range:
    - Entradas:
        - Range: estructura que contiene un rango de valores a probar como posibles claves para descifrar el mensaje.
        - ciph: arreglo de caracteres que representa el mensaje cifrado.
        - int len: longitud del mensaje cifrado.
        - int id: identificador único del proceso actual en el grupo MPI.
        - int N: número total de procesos en el grupo MPI.
        - MPI_Comm *comm: puntero a la comunicador MPI utilizado para la comunicación entre procesos.
        - MPI_Request *req: puntero a la solicitud de recepción MPI utilizada para verificar si algún otro proceso ha encontrado ya la clave para descifrar el mensaje.
        - int *ready: puntero a una variable booleana que indica si algún otro proceso ya ha encontrado la clave para descifrar el mensaje.
        - long *found: puntero a una variable que almacenará la clave encontrada, si es que se encuentra.
    - Salidas:
        - No tiene una salida explícita, pero actualiza los valores de las variables ready y found si se encuentra la clave para descifrar el mensaje.
    - Descripción:
        - Este método utiliza un enfoque recursivo para dividir el rango de valores a probar en sub-rangos más pequeños y probar cada uno de ellos. En cada llamada recursiva, se elige un valor aleatorio dentro del rango actual y se prueba si es la clave correcta para descifrar el mensaje. Si se encuentra la clave, se actualizan los valores de ready y found y se envía la clave a todos los demás procesos en el grupo MPI. Si no se encuentra la clave, se divide el rango actual en dos sub-rangos y se prueba cada uno de ellos en llamadas recursivas separadas. En cada llamada recursiva se comprueba si algún otro proceso ha encontrado ya la clave, utilizando MPI_Test para verificar la solicitud de recepción req. Si se ha encontrado la clave, se sale inmediatamente de la función sin realizar más pruebas.
- processTask:
    - Entrada:
        - Task task: es una estructura que contiene la tarea específica que se debe procesar. Contiene un límite inferior y un límite superior, que indican el rango de claves a probar.
        - int id: es el identificador único del proceso actual en el sistema distribuido.
        - long *found: es un puntero a una variable de tipo long que se utiliza para almacenar la clave encontrada. Si otro proceso ya ha encontrado la clave, esta variable se establece en el valor encontrado por ese proceso.

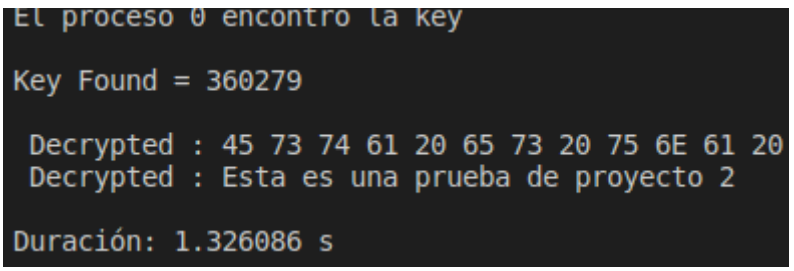
        - int *flag: es un puntero a una variable de tipo int que se utiliza para indicar si la clave ha sido encontrada. Si otro proceso ya ha encontrado la clave, esta variable se establece en 1.
        - char *ciph: es un puntero a una matriz de caracteres que contiene el texto cifrado.
        - int len: es el tamaño de la matriz de caracteres ciph.
        - DES_cblock *iv: es un puntero a un bloque de cifrado DES que se utiliza como vector de inicialización.
        - int datalen: es el tamaño del vector de inicialización.
        - MPI_Request *req: es un puntero a una variable de tipo MPI_Request que se utiliza para realizar operaciones de comunicación no bloqueantes.
    - Salida:
        - No retorna nada porque es un método void.
    - Descripción:
        - El método primero imprime un mensaje en la consola indicando la tarea que se está procesando y el identificador del proceso que la está procesando. Luego, se inicializa una variable req_recv para realizar una recepción no bloqueante utilizando MPI_Irecv, esperando el valor de la clave encontrada de cualquier proceso que esté enviando el mensaje. A continuación, se ejecuta un bucle for que prueba cada clave dentro del rango especificado por la tarea.
        - Dentro del bucle, se utiliza MPI_Test para verificar si el proceso ha recibido una respuesta de otro proceso. Si otro proceso ya ha encontrado la clave, el método establece la variable flag en 1 y sale del bucle. Si la clave se encuentra durante la iteración actual del bucle, se establece la variable found en el valor encontrado y se establece la variable flag en 1 para indicar que la clave ha sido encontrada. Luego, el método imprime un mensaje en la consola indicando que el proceso actual ha encontrado la clave.

Anexo 2 − Bitácora de pruebas y speedups

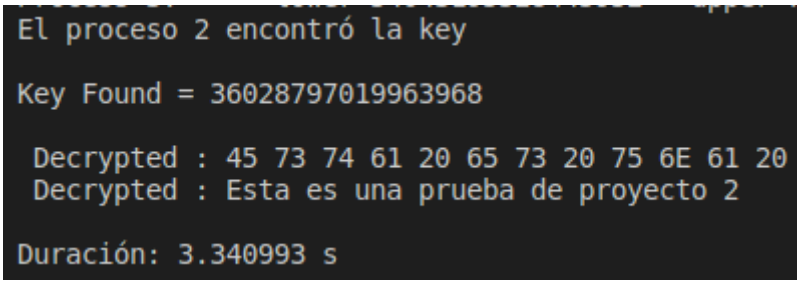
Algoritmo Naive:

El Speed Up de este algoritmo no tiene con quien compararse.

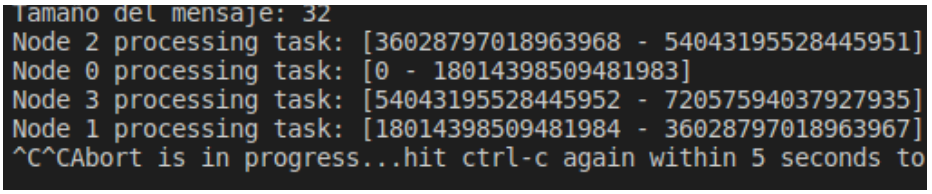
- **Llave** 360279 **Tiempo** 1.32s



- **Llave** 36028797019963968 **Tiempo** 3.34s



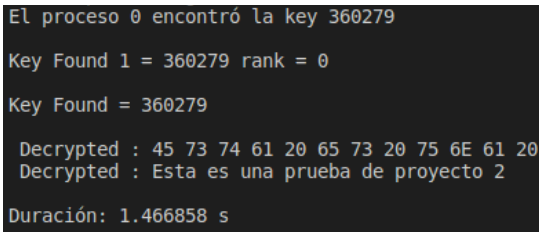
- **Llave** 18014398509481983 **Tiempo** nunca termino despues de 2 horas



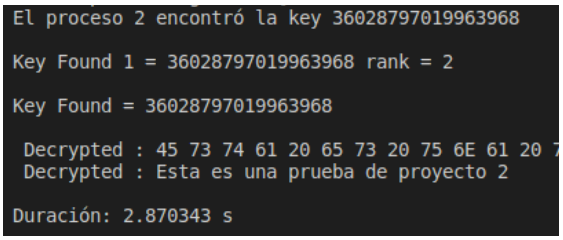
Algoritmo N1:

Speed Up de 0.97 indica que no hubo mejora.

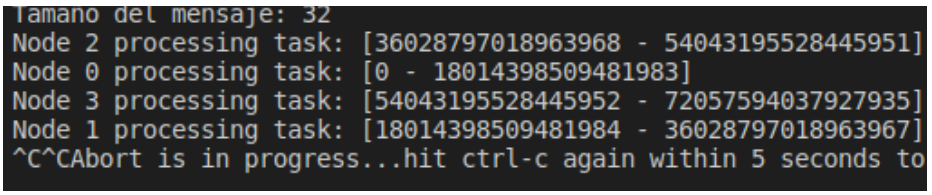
- **Llave** 360279 **Tiempo** 1.46s



- **Llave** 36028797019963968 **Tiempo** 2.87s



- **Llave** 18014398509481983 **Tiempo** nunca terminó después de 2 horas



Algoritmo N2: Árbol secuencial

- Llave: 11722020, tiempo: 9.09 s.

```
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpicc -o bruteforceLT bruteforceLT.c des_soft.c -lssl -lcrypto
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpirun -np 4 ./bruteforceLT

 Original  : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Original  : Esta es una prueba de proyecto 2

 Encrypted : 95 3E 8E D6 4C B0 E5 D1 0B E2 39 8A E2 15 D7 11 B4 1F D2 5A BF 42 44 28 72 83 F2 1B ED 53 61 C7
 Encrypted : ◆>◆◆L◆◆◆
                   ◆9◆◆□◆□◆□◆Z◆BD(r◆◆Sa◆

Process 0:      lower 0 - upper 18014398509481983
Process 1:      lower 18014398509481984 - upper 36028797018963967
Process 2:      lower 36028797018963968 - upper 54043195528445951
Process 3:      lower 54043195528445952 - upper 72057594037927936

Key Found = 11722020

 Decrypted : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Decrypted : Esta es una prueba de proyecto 2

Duración: 9.091537 s
```

- Llave 33554432, tiempo: 26.31 s.

```
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpicc -o bruteforceLT bruteforceLT.c des_soft.c -lssl -lcrypto
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpirun -np 4 ./bruteforceLT

 Original  : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Original  : Esta es una prueba de proyecto 2

 Encrypted : 1E 4C 94 4E AC 88 94 99 09 34 4E 72 C6 65 B5 A5 D1 5F 0C 89 22 79 25 8E 17 07 F9 A6 D2 AF F3 57
 Encrypted : □L◆N◆◆◆◆    4Nr◆e◆◆◆_
                                ◆"y%◆□◆◆y◆W

Process 0:      lower 0 - upper 18014398509481983
Process 1:      lower 18014398509481984 - upper 36028797018963967
Process 2:      lower 36028797018963968 - upper 54043195528445951
Process 3:      lower 54043195528445952 - upper 72057594037927936
count = 33554485

Key Found = 33554432

 Decrypted : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Decrypted : Esta es una prueba de proyecto 2

Duración: 26.307796 s
```

- Llave 1048576, tiempo: 0.80 s

```
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpicc -o bruteforceLT bruteforceLT.c des_soft.c -lssl -lcrypto
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpirun -np 4 ./bruteforceLT

 Original  : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Original  : Esta es una prueba de proyecto 2

 Encrypted : C3 D7 C8 AC 63 AD AC 55 41 2D 4F 11 7A 32 88 FB 2B 1F A4 12 20 B4 E6 41 89 3E AE 13 25 B6 D3 E3
 Encrypted : ◆◆Ôc◆◆UA-O□z2◆◆+□◆□ ◆◆A◆>◆□%◆◆◆

Process 0:      lower 0 - upper 18014398509481983
Process 1:      lower 18014398509481984 - upper 36028797018963967
Process 2:      lower 36028797018963968 - upper 54043195528445951
Process 3:      lower 54043195528445952 - upper 72057594037927936

Key Found = 1048576

 Decrypted : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Decrypted : Esta es una prueba de proyecto 2

Duración: 0.801816 s
```

- Llave 1048576, tiempo: 0.78 s.

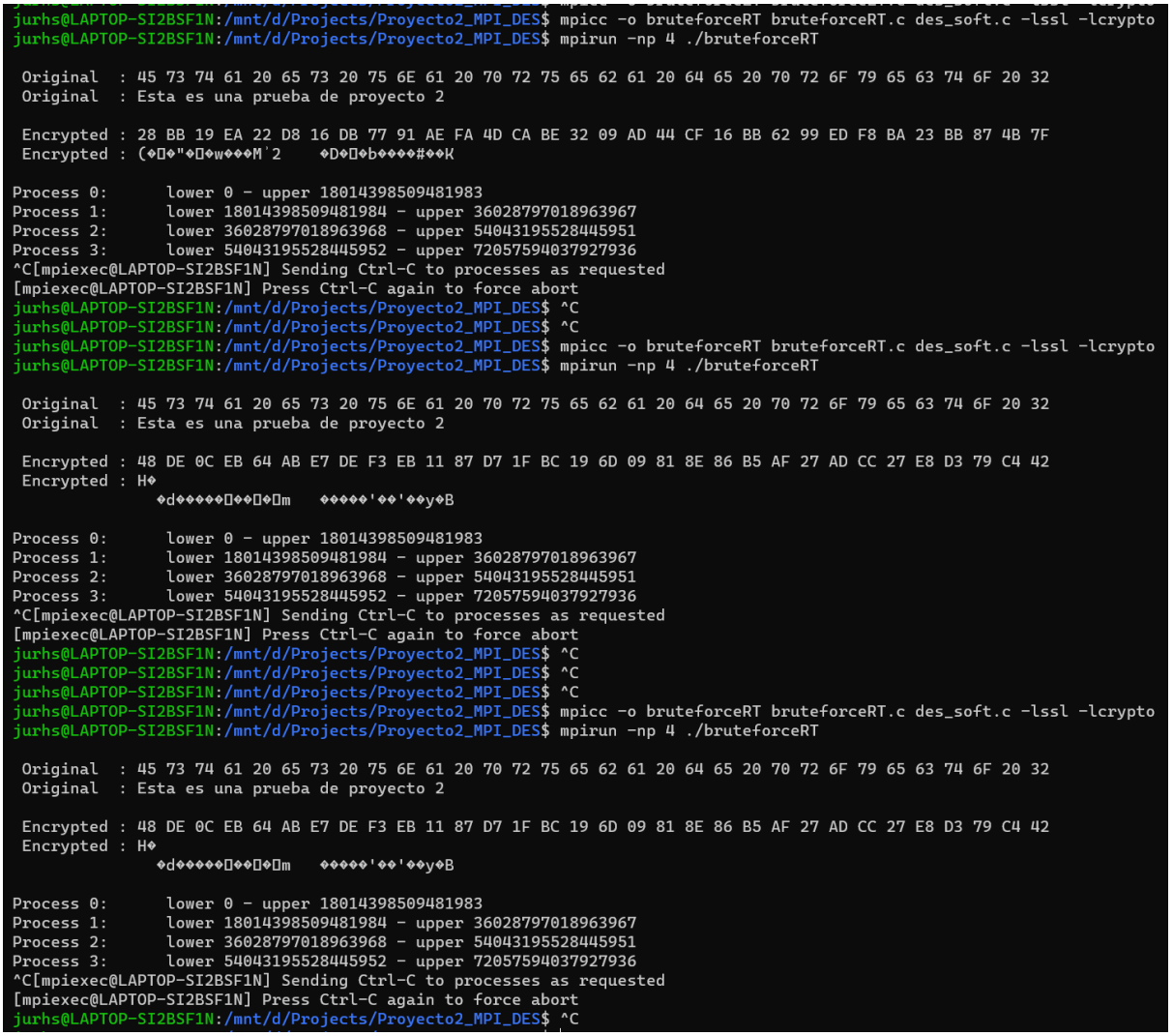
```
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpicc -o bruteforceLT bruteforceLT.c des_soft.c -lssl -lcrypto
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpirun -np 4 ./bruteforceLT

 Original  : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Original  : Esta es una prueba de proyecto 2

 Encrypted : C3 D7 C8 AC 63 AD AC 55 41 2D 4F 11 7A 32 88 FB 2B 1F A4 12 20 B4 E6 41 89 3E AE 13 25 B6 D3 E3
 Encrypted : ◆◆Åc◆◆UA-O□z2◆◆+□◆□ ◆◆A◆>◆□%◆◆◆

Process 0:      lower 0 - upper 18014398509481983
Process 1:      lower 18014398509481984 - upper 36028797018963967
Process 2:      lower 36028797018963968 - upper 54043195528445951
Process 3:      lower 54043195528445952 - upper 72057594037927936

Key Found = 1048576

 Decrypted : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Decrypted : Esta es una prueba de proyecto 2

Duración: 0.780472 s
```

- Llave 16777215, tiempo: 0.000046 s.

```
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpicc -o bruteforceLT bruteforceLT.c des_soft.c -lssl -lcrypto
jurhs@LAPTOP-SI2BSF1N:/mnt/d/Projects/Proyecto2_MPI_DES$ mpirun -np 4 ./bruteforceLT

 Original  : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Original  : Esta es una prueba de proyecto 2

 Encrypted : D6 A6 6D 09 06 DF 34 EA 42 F5 C7 DB 31 8D A1 3C E6 5B A4 46 94 C0 65 EE 27 75 E2 F9 5F CB 9C CA
 Encrypted : „m □◆4◆B◆◆◆1◆◆<◆[◆F◆◆e◆'u◆◆_˜◆

Process 0:      lower 0 - upper 18014398509481983
Process 1:      lower 18014398509481984 - upper 36028797018963967
Process 2:      lower 36028797018963968 - upper 54043195528445951
Process 3:      lower 54043195528445952 - upper 72057594037927936

Key Found = 16777215

 Decrypted : 45 73 74 61 20 65 73 20 75 6E 61 20 70 72 75 65 62 61 20 64 65 20 70 72 6F 79 65 63 74 6F 20 32
 Decrypted : Esta es una prueba de proyecto 2

Duración: 0.000046 s
```

Algoritmo N3: Árbol Random

Pruebas de llave 1, 2 y 3. (no terminaron luego de 1 hora):



Bibliografía

shubhamupadhyay. (2023). Data encryption standard DES Set 1. Extraido de: https://www.geeksforgeeks.org/data-encryption-standard-des-set-1/#article-meta-div

Open MPI. (2021). What is MPI? Extraido de: https://www.open-mpi.org/faq/?category=general#whatismpi