

Universidad del Valle de Guatemala  
Facultad de ingeniería



Computación paralela y distribuida  
Proyecto 2  
Open MPI y la criptografía

Gabriela Contreras  
Diego Cordova  
Andres de la Roca

Guatemala, 2023

# Índice

<b>Índice.....</b>	<b>1</b>
<b>Introducción.....</b>	<b>2</b>
<b>Antecedentes.....</b>	<b>2</b>
<b>Objetivos.....</b>	<b>6</b>
<b>Metodología.....</b>	<b>6</b>
<b>Resultados.....</b>	<b>6</b>
<b>Discusión.....</b>	<b>9</b>
<b>Conclusiones.....</b>	<b>10</b>
<b>Apéndice.....</b>	<b>11</b>
Anexo 1.....	11
Anexo 2.....	11
Anexo 3.....	13
Anexo 4.....	18
<b>Referencias bibliográficas.....</b>	<b>19</b>

## Introducción

Este proyecto busca enfrentarse a uno de los problemas fundamentales en seguridad de la información y de la criptografía: La recuperación de llaves privadas a través de un enfoque de fuerza bruta. En esta ocasión para realizar los diferentes algoritmos requeridos de la manera mas eficiente posible se implementarán temas de computación distribuida, en específico el uso de Open MPI, herramienta poderosa utilizada para realizar programación paralela con memoria distribuida. La premisa detrás de este esfuerzo es la determinación de la llave utilizada para cifrar una cierta cadena de texto, a partir de la fuerza bruta en un espacio de búsqueda masivo de posibles combinaciones de llaves de diferentes longitudes.

Este trabajo detalla el proceso de diseño e implementación del programa de descifrado paralelo con memoria distribuida, así como el análisis de los speedups y el rendimiento en general de las diferentes iteraciones que se realizará del algoritmo. El objetivo no es solo comprender cómo se puede implementar el algoritmo de fuerza bruta sino que también examinar el impacto de la llave de solución en el rendimiento y en los speedups del algoritmo, así como su consistencia en relación con diferentes llaves posibles.

A través de este trabajo se proporcionará una visión general al enfoque tomado durante el desarrollo de los algoritmos y programas, detallando los métodos utilizados para mapear los algoritmos, los desafíos encontrados y las conclusiones alcanzadas a través del análisis de resultados.

## Antecedentes

En un mundo cada vez más digitalizado, la seguridad de la información se ha convertido en un pilar fundamental de nuestra sociedad. La criptografía, la ciencia detrás de la protección de la confidencialidad de la información, es esencial para salvaguardar datos sensibles. Sin embargo, la criptografía también plantea desafíos fundamentales en cuanto a la seguridad de las claves utilizadas para cifrar y descifrar la información. La búsqueda de una llave privada mediante un enfoque de fuerza bruta es un problema clásico, y este proyecto se sitúa en el contexto de abordar esta tarea mediante la programación paralela con memoria distribuida, aprovechando las capacidades de computación distribuida de Open MPI.

En el contexto de la seguridad de la información, el algoritmo de Cifrado de Datos Estándar (DES en inglés) ha sido un pilar fundamental desde su desarrollo en la década de 1970.

El proceso de cifrado en el algoritmo DES implica los siguientes pasos:

- Permutación Inicial (IP): El texto original se somete a una permutación inicial que reorganiza los bits. Esta permutación tiene como objetivo aleatorizar el texto de entrada.
- Rondas de Feistel: DES utiliza un enfoque de Feistel, donde el bloque de texto se divide en dos mitades. Luego, se realizan múltiples rondas de transformaciones en estas mitades de manera alternada. Estas rondas incluyen operaciones de expansión, sustitución y permutación.

- Clave de Ronda: En cada ronda, se utiliza una subclave derivada de la clave de cifrado principal para realizar operaciones en las mitades del texto. La subclave se genera a partir de la clave de cifrado principal utilizando un proceso de generación de subclaves.
- Permutación Final (FP): Después de las rondas de Feistel, se aplica una permutación final para reorganizar los bits del texto cifrado.

El proceso de descifrado en DES es esencialmente el inverso del proceso de cifrado. Las subclaves se utilizan en orden inverso para deshacer las transformaciones realizadas en el cifrado.

El uso de una clave de cifrado adecuada en DES es crucial para garantizar la seguridad de la información. Sin embargo, el cifrado DES ha sido superado por estándares más modernos debido a su vulnerabilidad a ataques de fuerza bruta.

Como referencia para el desarrollo de los algoritmos de este proyecto de tiene un programa desarrollado en el lenguaje C con el uso de Open MPI, que cumple con el mismo objetivo general del proyecto, algunas de sus rutinas son más útiles son:

- **decrypt:**  
Esta rutina se encarga de descifrar datos cifrados utilizando una clave proporcionada.  
  
Se toma la clave proporcionada (key) y se ajusta su paridad. Esto implica asegurarse de que la clave cumple con ciertas restricciones de paridad, lo que es necesario para el funcionamiento adecuado de DES.  
Se inicializa un programa de horario de clave (keysched) y una clave DES (des\_key) a partir de la clave ajustada.  
Luego, la rutina procede a descifrar los datos en bloques de 8 bytes (len representa la longitud total de los datos). Para cada bloque, se utiliza el modo de cifrado DES en modo ECB (Electronic Codebook) para realizar el proceso de descifrado. El programa de horario de clave se utiliza para gestionar las subclaves necesarias en cada ronda de cifrado.
- **encrypt:**  
Esta rutina se utiliza para cifrar datos utilizando una clave proporcionada. Su funcionamiento es similar al de decrypt, pero en sentido inverso.  
  
Al igual que en la rutina de descifrado, se ajusta la paridad de la clave proporcionada. Se inicializa el programa de horario de clave y la clave DES a partir de la clave ajustada.

Luego, la rutina procede a cifrar los datos en bloques de 8 bytes utilizando el modo de cifrado DES en modo ECB. El programa de horario de clave se encarga de administrar las subclaves necesarias en cada ronda de cifrado.

- tryKey:

Es una rutina que se utiliza para probar una clave específica en un texto cifrado con el algoritmo DES y determinar si el resultado del descifrado contiene una palabra o frase clave específica que estamos buscando.

Se crea una copia temporal de los datos cifrados en un nuevo arreglo llamado temp. Esto se hace para evitar modificar los datos originales en el proceso de descifrado. La copia temporal tiene una longitud un byte mayor que la original y se establece en 0 al final, lo que convierte la copia en una cadena de caracteres nula.

Luego, la función llama a la rutina decrypt que se explicó previamente, utilizando la clave key para descifrar los datos almacenados en temp. El resultado del descifrado se almacena nuevamente en el arreglo temp.

Después, la función busca si la palabra o frase clave que estamos buscando, representada por la variable search, se encuentra en el texto descifrado. Esto se hace utilizando la función strstr, que busca una subcadena en el texto descifrado (temp). Si la función strstr encuentra una coincidencia y devuelve un puntero diferente de NULL, significa que se ha encontrado la palabra clave en el texto descifrado.

Por último, la función tryKey retorna un valor booleano. Si la palabra clave se encuentra en el texto descifrado, la función devuelve 1, lo que indica que se ha tenido éxito en encontrar la palabra clave con la clave probada. Si no se encuentra, la función devuelve 0, lo que significa que la clave no es la correcta.

- memcpy:

Función cuya utilidad principal es copiar un bloque de memoria desde una ubicación de origen hasta una ubicación de destino, copia la cantidad específica de bytes.

- strstr:

Función que se utiliza para buscar la primera ocurrencia de una subcadena dentro de una cadena más grande.

Las primitivas de comunicación en Open MPI son fundamentales para permitir la comunicación entre procesos en un programa paralelo. A continuación se explicarán tres primitivas principales de esta herramienta:

- MPI\_Irecv

Se utiliza para iniciar una operación de recepción de datos de otro proceso. La principal característica de esta función es que es no bloqueante, lo que permite al receptor continuar su ejecución sin esperar a que los datos lleguen. Aquí está el flujo de uso y comunicación:

1. El proceso receptor (el que llama a `MPI_Irecv`) inicia una operación de recepción especificando el búfer donde se almacenarán los datos recibidos, el tamaño de los datos esperados, el tipo de dato y el identificador del proceso remitente.
2. El receptor continúa su ejecución, sin bloquearse, lo que le permite realizar otras tareas mientras espera la llegada de datos.
3. El proceso emisor (otro proceso en el programa MPI) utiliza `MPI_Send` para enviar los datos al proceso receptor.
4. Cuando los datos se envían, se almacenan en el búfer especificado en `MPI_Irecv` en el proceso receptor.
5. Para asegurarse de que los datos se han recibido por completo, se utiliza la función `MPI_Wait`, que se explica a continuación.

- **MPI\_Send**

Se utiliza para enviar datos desde un proceso a otro. Esta función es bloqueante por defecto, lo que significa que el proceso emisor espera hasta que los datos se hayan transmitido con éxito antes de continuar. El flujo de comunicación es el siguiente:

1. El proceso emisor (el que llama a `MPI_Send`) especifica los datos que se deben enviar, el tamaño de los datos, el tipo de dato y el identificador del proceso receptor.
2. El emisor inicia la transmisión de datos.
3. La función `MPI_Send` bloquea el proceso emisor hasta que los datos se hayan transmitido por completo al proceso receptor.
4. Una vez que los datos se han transmitido con éxito, el proceso emisor puede continuar su ejecución.

- **MPI\_Wait**

Se utiliza para garantizar que las operaciones de recepción iniciadas con `MPI_Irecv` se completen correctamente antes de continuar la ejecución del proceso receptor. El flujo de uso es el siguiente:

1. El proceso receptor inicia una operación de recepción de datos con `MPI_Irecv`.

2. El proceso receptor continúa su ejecución y realiza otras tareas.
3. Cuando es necesario asegurarse de que la recepción de datos se haya completado, se llama a MPI\_Wait con la solicitud correspondiente. MPI\_Wait bloquea el proceso receptor hasta que los datos se hayan recibido por completo.
4. Una vez que los datos se han recibido con éxito, MPI\_Wait permite que el proceso receptor continúe su ejecución.

## Objetivos

- Implementar y diseñar programas para la paralelización de procesos con memoria distribuida usando Open MPI.
- Optimizar el uso de recursos distribuidos y mejorar el speedup de un programa paralelo
- Descubrir la llave privada usada para cifrar un texto, usando el método de fuerza bruta.
- Comprender y analizar el comportamiento del speedup de forma estadística.

## Metodología

Para lograr alcanzar la propuesta y objetivos del proyecto presente se desarrolló una metodología que combina la implementación secuencial inicial del programa con iteraciones paralelas utilizando Open MPI con un enfoque “naive” y dos iteraciones realizadas con un enfoque diferente al “naive”.

En concreto primero se diseñó la base secuencial con la que se cumplieron ciertos requisitos especificados que se deben cumplir para poder realizar las iteraciones paralelas con Open MPI.

Para la medición de la eficiencia y las mejoras en el rendimiento se calculó el speedup de las versiones paralelas del programa, a través de esto se logró medir el impacto de la paralelización distribuida en el rendimiento de este programa.

Adicionalmente se optó por documentar extensivamente los programas para poder tener un mejor control y mayor entendimiento del proyecto.

## Resultados

Para las pruebas secuenciales se realizaron pruebas con diferentes tamaños de llave cifrada para observar cómo se comportaba el programa en estas condiciones. Se encontró lo siguiente tras realizar 3 conjuntos de pruebas para cada longitud de llave:

### *Pruebas secuenciales con llaves aleatorias*

Longitud de la llave	Tiempo Promedio de ejecución
1	0.001 ms
2	0.003 ms
3	0.097 ms
4	1.459 ms
5	46.426 ms
6	2061.41 ms

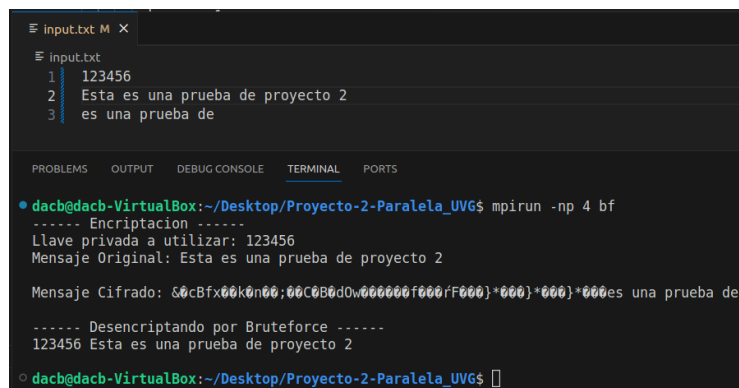
Ahora bien para los próximos programas se realizaron tres pruebas con las siguientes llaves, nivel fácil con 42, medio con 123456 y difícil con 123456789

### *Pruebas brute force secuenciales*

No. Prueba	Tiempo ejecución
1	48 ms
2	628569 ms
3	1523148 ms

### *Incisos programa naive*

- Tiempo de ejecución usando la llave 123456L



```
input.txt M X
input.txt
1 123456
2 Esta es una prueba de proyecto 2
3 es una prueba de

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

• dach@dach-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$ mpirun -np 4 bf
----- Encriptacion -----
Llave privada a utilizar: 123456
Mensaje Original: Esta es una prueba de proyecto 2

Mensaje Cifrado: S6cBfx06k0n00,00C0B0d0w000000f000rF000}*000}*000}*000es una prueba de

----- Descriptando por Bruteforce -----
123456 Esta es una prueba de proyecto 2
o dach@dach-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$
```



- b. Tiempo de ejecución usando la llave 18014398509481983L

```

andres@andres-VirtualBox:~/Desktop/Universidad/Paralela/Proyecto2$ mpirun -np 4 bf_txt
----- Encriptacion -----
Llave privada a utilizar: 18014398509481983
Mensaje Original: Esta es una prueba de proyecto 2
Mensaje Cifrado: 0+0r"VuK000_ ++^00V 0)000;H000-0)00^000U0t7H000{L:[0D0/es una prueba de
----- Desencriptando por Bruteforce -----

```

*En esta iteración el programa corrió por más de 1 hora. Pero, no se logró encontrar la llave privada*

- c. Tiempo de ejecución usando la llave 18014398509481984L

```

input.txt M x
input.txt
1 18014398509481984L
2 Esta es una prueba de proyecto 2
3 es una prueba de

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Writing objects: 100% (4/4), 61.85 KiB | 20.62 MiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/andresdlRoca/Proyecto-2-Paralela_UVG.git
3ded18d..bb1c381 main -> main
dadb@dadb-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$ ./drivers/runbf.sh
----- Encriptacion -----
Llave privada a utilizar: 18014398509481984
Mensaje Original: Esta es una prueba de proyecto 2
Mensaje Cifrado: 8+h?0u0000p000;b
----- Desencriptando por Bruteforce -----
^Cdadb@dadb-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$

```

*En este caso se dejó correr al programa por más de 1 hora. Sin embargo, no se logró encontrar la llave privada.*

- d. Reflexión

- Este programa es funcional puesto que es capaz de encriptar y decodificar una cadena de texto en base a una llave otorgada por el usuario. Se puede resaltar que este es bastante efectivo cuando la llave que se está utilizando no es muy larga ya que al momento de ser muy grande el algoritmo puede llegar a tardar mucho tiempo puesto a que debe de probar todas las posibles combinaciones haciendo que este sea poco efectivo, esto es algo que logramos apreciar al momento de realizar el inciso a contra el inciso b y c ya que en los últimos dos el programa se dejó corriendo por más de una hora por lo que se deduce que el algoritmo no ha sido capaz de encontrar la solución.

### *Pruebas paralelas naive*

No. Prueba	Tiempo ejecución	Speed Up
1	0.1119 ms	428.95
2	203.2521 ms	3092.55
3	154418.9558 ms	9.86

### *Pruebas paralelas segunda implementación*

No. Prueba	Tiempo ejecución	Speed Up
1	0.050139 ms	957.33
2	1065.1475 ms	590.12
3	2405236.7 ms	0.63

### *Pruebas paralelas tercera implementación*

No. Prueba	Tiempo ejecución	Speed Up
1	0.085361 ms	562.31
2	151.27303 ms	4155.19
3	171066.48 ms	8.90

## **Discusión**

El objetivo del proyecto fue diseñar e implementar programas paralelos haciendo uso de memoria distribuida, es importante destacar que en cada programa desarrollado se buscó la optimización de recursos con la finalidad de mejorar el speed up. Para ello se realizaron cinco programas con diferentes enfoques pero siempre con un mismo objetivo el cual era encriptar y desencriptar un texto.

Primeramente se realizó un programa secuencial el cual recibe un texto el cual es cifrado por medio del algoritmo DES y luego por medio de la fuerza bruta intenta adivinar la clave original intentando todas las combinaciones posibles de las letras mayúsculas de A a la Z. Cuando encuentra la llave esta es desencriptada para luego mostrarle al usuario el resultado.

Ahora bien el segundo programa cuenta con un enfoque algo diferente en el sentido de que ahora este hace uso de la biblioteca MPI para buscar la clave. En este se define una nueva función la cual intenta descifrar un mensaje cifrado con una clave DES específica y busca una cadena específica en el mensaje descifrado. Ahora bien al momento de descifrar primero se calcula el rango de claves que cada proceso, cabe resaltar que la búsqueda de fuerza bruta se distribuye entre los procesos ya que cada proceso prueba un rango de claves específico. Si este encuentra un resultado este le avisa a los demás procesos para luego poder decodificar esta con el respectivo mensaje.

Por otro lado el tercer programa cuenta con la misma lógica que el segundo programa pero ahora lo que se le cambio a este es que en vez de ingresar manualmente la cadena de texto y

la clave que se desea utilizar esta es brindada en un txt el cual es leído y procesado por el programa en donde se establece que línea hace referencia a que variable. En este se puede observar que los tiempos de ejecución son algo elevados lo cual se puede relacionar al tamaño de las llaves y a que la fuerza bruta debe de probar con todas las posibles combinaciones para asegurarse de que la llave sea la correcta

Así mismo el cuarto programa está basado en el programa anterior por lo que en esencia este realiza lo mismo, lo único que varía es la forma en la que está buscando las llaves se ha optimizado ya que ahora en lugar de usar una comunicación de punto a punto utiliza MPI\_allReduce para encontrar las claves lo cual representa una mejora ya que de esta manera se evita recopilar y sincronizar de forma “manual” entre cada nodo. Por último, al momento de buscar las claves ahora se itera desde mylower hasta myupper lo cual hace que la búsqueda se realice más rápido

Finalmente, el quinto programa está igualmente basado en el tercer programa por lo que es casi lo mismo, solo que ahora la forma en la que busca las llaves se realiza de diferente forma ya que este realiza la distribución de llaves como lo hace bruteforce. Sin embargo, al llegar al límite, este algoritmo no continúa buscando llaves una por una, sino que realiza saltos de exponenciales. Es decir al llegar a un limite  $x$ , se saltará al valor  $x^2$  y al llegar  $x^4$ , se saltará a  $x^6$ . Esta decisión se tomó debido a que la mayoría de claves (o por lo menos las más seguras) son números extremadamente grandes, por lo que se toma como objetivo el llegar “más rápido” a dichos números. Cabe mencionar que el número  $x$  con el que se calcula el exponente, es el id del proceso en cuestión. Además, se aísla un proceso para que siga realizando un bruteforce para no dejar ninguna llave posible sin revisar.

Como se puede observar en los resultados obtenidos en algunos casos se contó con una mejora significativa en cuanto al tiempo de ejecución y esto se puede ver reflejado en el speed up, no obstante esto no fue constante para todos los casos ya que en algunas ocasiones el tiempo del secuencial fue menor al paralelo. Esto fue algo que logramos ver con el cuarto programa realizado ya que este en un inicio contó con una mejor significativa pero al momento de realizar las otras pruebas esta carece de rapidez. Ahora bien con el último algoritmo implementado podemos ver que la mejora fue constante únicamente decreció poco en la última prueba pero aun su mejora sigue siendo significativa.

## Conclusiones

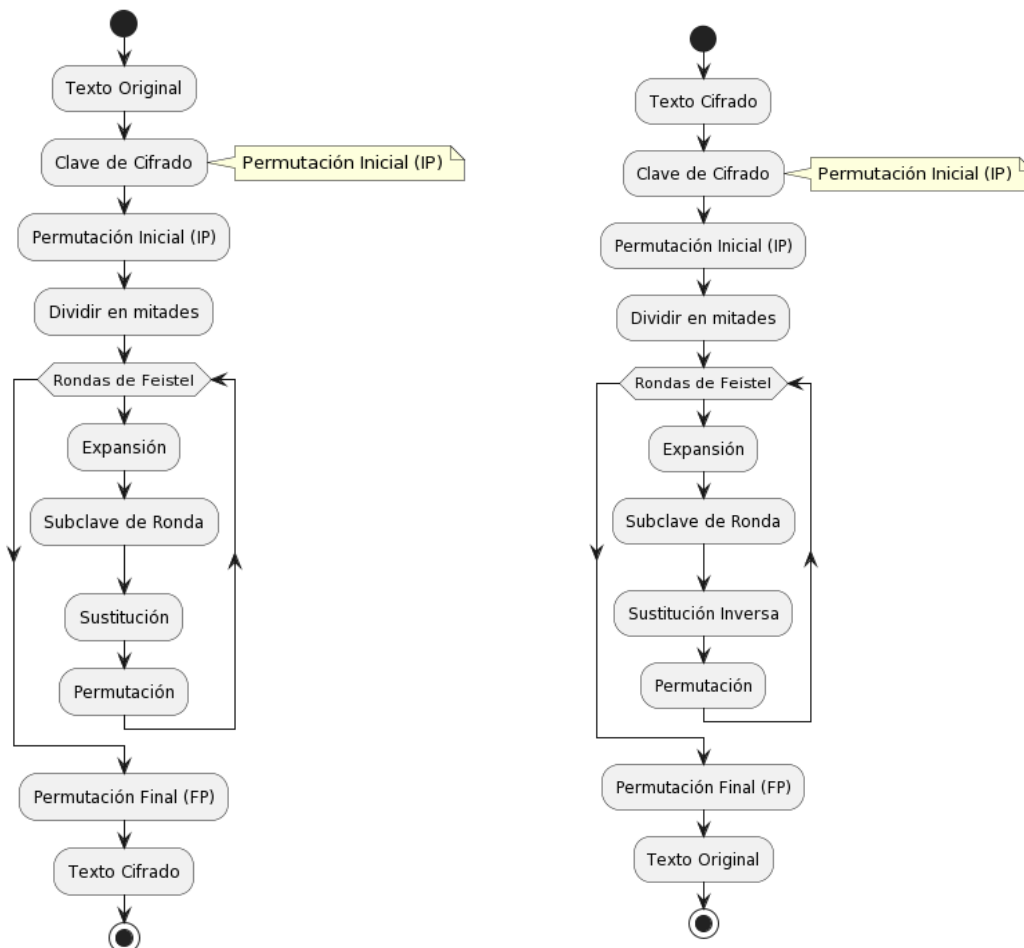
- Se debe de tener un conocimiento claro acerca de las diferentes directivas que se van a utilizar pues de esta manera se pueden optimizar sus usos y al mismo tiempo se puede ahorrar tiempo probando diferentes directivas.
- Al hablar de descryptación de mensajes, es importante conocer el contexto que se aplica ya que esto puede dar una pauta para implementar un algoritmo que se adapte a dicho contexto y por tanto sea más eficiente. Aunque no sea posible en todos los escenarios.

- La distribución de cargas entre procesos es una de las estrategias más efectivas para conseguir un buen rendimiento al momento de desarrollar un programa con paralelismo. Utilizar las directivas y comandos correctos para alcanzar esto permitió acelerar el proceso de búsqueda de claves.
- En algunos casos, la mejora en el tiempo de ejecución y el speedup se mantuvo constante a medida que se realizaban pruebas con claves de mayor complejidad, lo que sugiere que utilizar la programación paralela aumentó la eficiencia durante la búsqueda de claves.

## Apéndice

### Anexo 1

*Se adjunta diagramas de flujo describiendo el algoritmo DES*



### Anexo 2

*A continuación se adjuntan los screenshots de las pruebas realizadas para el programa secuencial.*

```

• andres@andres-VirtualBox:~/Desktop/Universidad/Paralela/Proyecto2$ ./secuencial
Key Length: 1, Found Key: A, Elapsed Time: 0.000001 seconds
Key Length: 2, Found Key: GL, Elapsed Time: 0.000002 seconds
Key Length: 3, Found Key: TBS, Elapsed Time: 0.000220 seconds
Key Length: 4, Found Key: VQDR, Elapsed Time: 0.003281 seconds
Key Length: 5, Found Key: ZQZUU, Elapsed Time: 0.107519 seconds
Key Length: 6, Found Key: ZAVZVT, Elapsed Time: 2.630661 seconds

• andres@andres-VirtualBox:~/Desktop/Universidad/Paralela/Proyecto2$ ./secuencial
Key Length: 1, Found Key: O, Elapsed Time: 0.000001 seconds
Key Length: 2, Found Key: XC, Elapsed Time: 0.000006 seconds
Key Length: 3, Found Key: HKY, Elapsed Time: 0.000042 seconds
Key Length: 4, Found Key: FJTN, Elapsed Time: 0.000902 seconds
Key Length: 5, Found Key: HARVG, Elapsed Time: 0.028448 seconds
Key Length: 6, Found Key: NEPZWR, Elapsed Time: 1.394627 seconds

• andres@andres-VirtualBox:~/Desktop/Universidad/Paralela/Proyecto2$ ./secuencial
Key Length: 1, Found Key: C, Elapsed Time: 0.000001 seconds
Key Length: 2, Found Key: FC, Elapsed Time: 0.000002 seconds
Key Length: 3, Found Key: FFB, Elapsed Time: 0.000030 seconds
Key Length: 4, Found Key: BHTV, Elapsed Time: 0.000196 seconds
Key Length: 5, Found Key: AVEWU, Elapsed Time: 0.003312 seconds
Key Length: 6, Found Key: VHKZXC, Elapsed Time: 2.158961 seconds

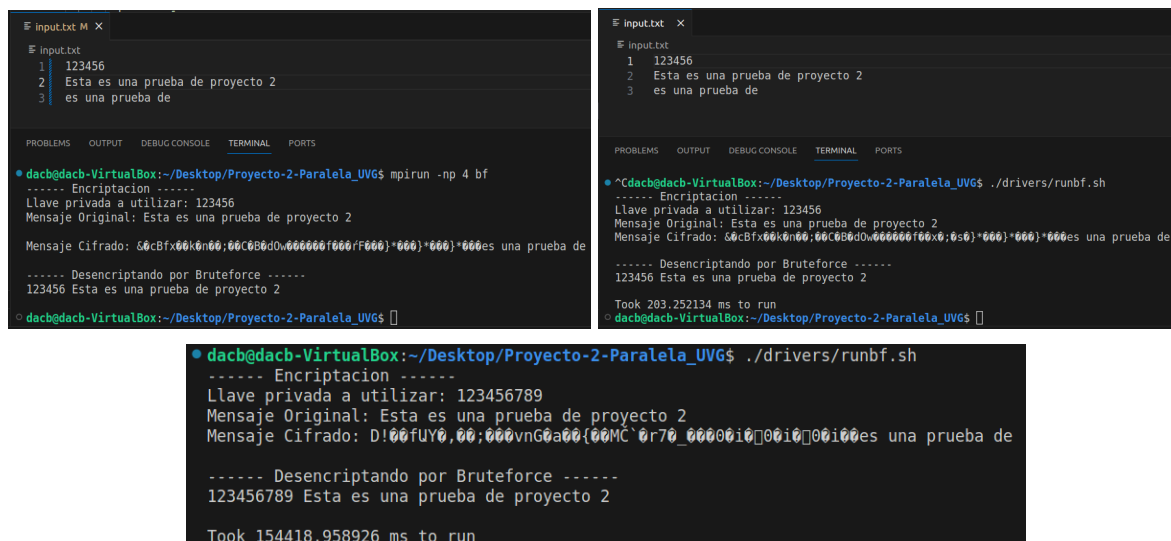
• andres@andres-VirtualBox:~/Desktop/Universidad/Paralela/Proyecto2$ ./secuencial
Ingrese la key: 42
Key Length: 2, Found Key: 42, Elapsed Time: 0.000048 seconds

• andres@andres-VirtualBox:~/Desktop/Universidad/Paralela/Proyecto2$
Ingrese la key: 123456
Key Length: 6, Found Key: 123456, Elapsed Time: 628.569 seconds

• andres@andres-VirtualBox:~/Desktop/Universidad/Paralela/Proyecto2$
Ingrese la key: 123456789
Key Length: 9, Found Key: 123456789, Elapsed Time: 1523.148943 seconds

```

*A continuación se adjuntan los screenshots de las pruebas realizadas para el programa paralelo naive.*



```

• dach@dach-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$ mpirun -np 4 bf
----- Encriptacion -----
Llave privada a utilizar: 123456
Mensaje Original: Esta es una prueba de proyecto 2
Mensaje Cifrado: 50cBfx00k0n00;00C0B0d0w00000f000;F000}*000}*000}*000es una prueba de
----- Desencriptando por Bruteforce -----
123456 Esta es una prueba de proyecto 2

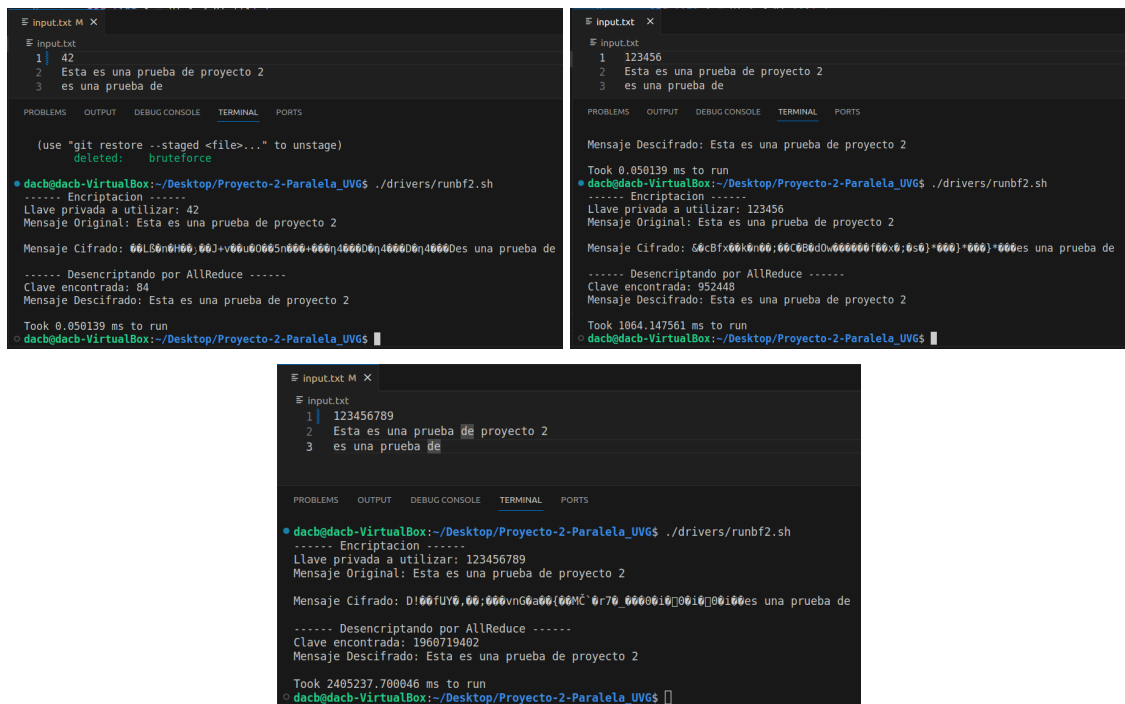
• dach@dach-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$

• ^Cdach@dach-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$ ./drivers/runbf.sh
----- Encriptacion -----
Llave privada a utilizar: 123456
Mensaje Original: Esta es una prueba de proyecto 2
Mensaje Cifrado: 50cBfx00k0n00;00C0B0d0w00000f000;0s0}*000}*000}*000es una prueba de
----- Desencriptando por Bruteforce -----
123456 Esta es una prueba de proyecto 2
Took 203.252134 ms to run

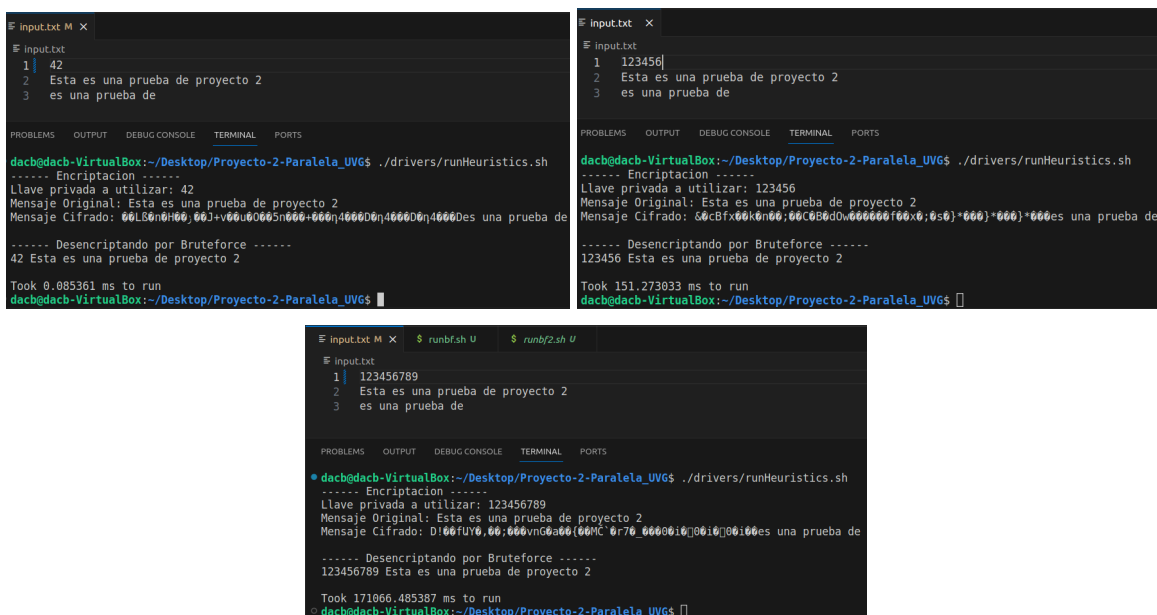
• dach@dach-VirtualBox:~/Desktop/Proyecto-2-Paralela_UVG$ ./drivers/runbf.sh
----- Encriptacion -----
Llave privada a utilizar: 123456789
Mensaje Original: Esta es una prueba de proyecto 2
Mensaje Cifrado: D!00fUY0;00;000vnG0a00{00MC`0r70_00006i000i000i000es una prueba de
----- Desencriptando por Bruteforce -----
123456789 Esta es una prueba de proyecto 2
Took 154418.958926 ms to run

```

*A continuación se adjuntan los screenshots de las pruebas realizadas para el programa paralelo segunda implementación.*



*A continuación se adjuntan los screenshots de las pruebas realizadas para el programa paralelo tercera implementación.*



## Anexo 3

### - Variables locales

Nombre	Tipo	Ubicación	Descripción
Secuencial.c			

textLen	int	encrypt	almacena la longitud del texto original
keyLen	int	encrypt	almacena la longitud de la clave generada
text	char	main	almacena el texto original
ciphertext	char	main	arreglo de caracteres que almacena el texto cifrado
key	char	main	arreglo de caracteres que almacena la llave
start_time	clock_t	main	almacena el tiempo en el que comienza a buscar la clave
foundKey	char	main	arreglo que almacena la clave
end_time	clock_t	main	almacena el tiempo en el que se completa la búsqueda de la clave
elapsed_time	double	main	almacena el tiempo transcurrido durante la ejecución
<b>bruteforce.c</b>			
temp	int	trykey	arreglo que almacena una copia de los datos cifrados
N	int	main	almacena el número total de los procesos MPI
id	int	main	almacena el identificador del proceso actual
upper	long	main	almacena el límite superior del rango de búsqueda
mylower	long	main	almacena la clave inferior del rango de búsqueda
myupper	long	main	almacena la clave superior del rango de búsqueda
ciphlen	int	main	almacena la longitud de los datos cifrados
cipher	char	main	arreglo de bytes que almacena los datos cifrados
range per node	int	main	almacena la cantidad de claves asignadas a cada nodo

found	long	main	indica si se ha encontrado la clave o no
<b>bf_txt.c</b>			
temp	int	trykey	arreglo que almacena una copia de los datos cifrados
N	int	main	almacena el número total de los procesos MPI
id	int	main	almacena el identificador del proceso actual
upper	long	main	almacena el límite superior del rango de búsqueda
mylower	long	main	almacena la clave inferior del rango de búsqueda
myupper	long	main	almacena la clave superior del rango de búsqueda
ciphlen	int	main	almacena la longitud de los datos cifrados
chiperLine	char	main	arreglo de bytes que almacena el texto cifrado y leído desde un archivo
search	char	main	arreglo que almacena la cadena que se busca en el texto de entrada
inputFileName	char	main	almacena el nombre del archivo de entrada
key	long	main	almacena la conversión de la llave a long
buffer	char	main	arreglo que se usa para almacenar información temporal
found	long	main	indica si se ha encontrado la clave o no
tstart	clock_t	main	almacena el tiempo en el que se completa la búsqueda de la clave
tend	clock_t	main	almacena el tiempo en el que se completa la búsqueda de la clave
<b>bf2.c</b>			



temp	int	trykey	arreglo que almacena una copia de los datos cifrados
N	int	main	almacena el número total de los procesos MPI
id	int	main	almacena el identificador del proceso actual
upper	long	main	almacena el límite superior del rango de búsqueda
mylower	long	main	almacena la clave inferior del rango de búsqueda
myupper	long	main	almacena la clave superior del rango de búsqueda
cipherLine	char	main	almacena la longitud de los datos cifrados
KeyLine	char	main	almacena la llave leída en el archivo de entrada
inputFileName	char	main	almacena el nombre del archivo de entrada
search	char	main	arreglo que almacena la cadena que se busca en el texto de entrada
ciphlen	int	main	almacena la longitud de los datos cifrados
key	long	main	almacena la conversión de la llave a long
buffer	char	main	arreglo que se usa para almacenar información temporal
found	long	main	indica si se ha encontrado la clave o no
range_per_node	int	main	almacena la cantidad de claves asignadas a cada nodo
tstart	clock_t	main	almacena el tiempo en el que se completa la búsqueda de la clave
tend	clock_t	main	almacena el tiempo en el que se completa la búsqueda de la clave
<b>fases.c</b>			

temp	int	trykey	arreglo que almacena una copia de los datos cifrados
N	int	main	almacena el número total de los procesos MPI
id	int	main	almacena el identificador del proceso actual
upper	long	main	almacena el límite superior del rango de búsqueda
mylower	long	main	almacena la clave inferior del rango de búsqueda
myupper	long	main	almacena la clave superior del rango de búsqueda
cipherLine	char	main	almacena la longitud de los datos cifrados
keyLine	char	main	almacena la llave leída en el archivo de entrada
inputFileName	char	main	almacena el nombre del archivo de entrada
search	char	main	arreglo que almacena la cadena que se busca en el texto de entrada
key	int	main	almacena la conversión de la llave a long
buffer	char	main	arreglo que se usa para almacenar información temporal
range_per_node	int	main	almacena la cantidad de claves asignadas a cada nodo
found	long	main	indica si se ha encontrado la clave o no
temp_index	int	main	calcula dinámicamente el valor de la clave que se está probando en cada iteración
limit	long	main	almacena el límite superior actual para la búsqueda de las claves
exp	long	main	almacena la potencia actual a la que se eleva upper

start	long	main	almacena el valor inicial de la búsqueda en cada fase
differ	ling	main	almacena la diferencia entre el índice actual y el límite superior actual
local_exp	int	main	utilizada para calcular el valor de temp_index
tstart	clock_t	main	almacena el tiempo en el que se completa la búsqueda de la clave
tend	clock_t	main	almacena el tiempo en el que se completa la búsqueda de la clave

#### - Funciones

Nombre	Tipo	Descripción
Encrypt	void	Esta función es utilizada para cifrar datos mediante el algoritmo DES y sobrescribe los datos en el mismo búfer de datos originales
Decrypt	void	Esta función descripta datos utilizando el algoritmo de DES, esta sobrescribe los datos descriptados en el mismo buffer de datos cifrados .
TryKey	int	Esta función es utilizada para probar una clave en un conjunto de datos cifrados descriptar los mismos para así poder buscar una cadena en los datos descriptados
Main	int	Esta función es la encargada de realizar toda la lógica principal del programa pues dependiendo del archivo que se esté probando encripta y descripta por medio de la fuerza bruta un conjunto de datos.

#### Anexo 4

Link al repositorio utilizado

- [https://github.com/andresdlRoca/Proyecto-2-Paralela\\_UVG](https://github.com/andresdlRoca/Proyecto-2-Paralela_UVG)

## Referencias bibliográficas

Bianchi, O. M., & Repetto, A. J. M. (2012). Computación distribuida para seguridad informática (CoDiSe). In XIV Workshop de Investigadores en Ciencias de la Computación.

Castro Lechtaler, A., Repetto, A. J. M., Bianchi, O. M., Cipriano, M., Arroyo Arzubi, A., Cicerchia, C. D., & Malvacio, E. (2014). ULTRACOM: Computación de alto rendimiento para criptoanálisis. In XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014).

Bianchi, O. M., Ariznabarreta Fossati, J. I., & Repetto, A. J. M. (2013, June). Construyendo un sistema de cómputo distribuido multipropósito. In XV Workshop de Investigadores en Ciencias de la Computación.

Feigenbaum, J., & Merritt, M. (Eds.). (1991). Distributed computing and cryptography: proceedings of a DIMACS Workshop, October 4-6, 1989 (Vol. 2). American Mathematical Soc..

Grabbe, J. O. (2010). The DES algorithm illustrated.