

Universidad del Valle de Guatemala
Facultad de Ingeniería



Proyecto 2 - MPI
Paralela

Mariana David 201055
Angel Higueros 20460
Pablo Escobar

Guatemala 27 de octubre del 2023

Parte A

1. **Investigue sobre DES y describa los pasos requeridos para cifrar/descifrar un texto. Incluya esto en su reporte.**

Data Encryption Standard (DES):

El Estándar de Cifrado de Datos (Data Encryption Standard o DES) es un algoritmo de cifrado simétrico que fue adoptado como un estándar federal en los Estados Unidos en la década de 1970. DES es un algoritmo de bloque, lo que significa que cifra y descifra datos en bloques fijos de tamaño predeterminado. En el caso de DES, los bloques tienen una longitud de 64 bits. A continuación, se describen los pasos requeridos para cifrar y descifrar un texto utilizando el algoritmo DES:

Cifrado con DES:

1. **Clave de Cifrado:** El proceso de cifrado con DES comienza con la selección de una clave de 56 bits. Inicialmente, la clave de 64 bits se somete a una permutación para generar una clave de 56 bits. Esto se hace para eliminar 8 bits de paridad y fortalecer la seguridad de la clave.
2. **Generación de Subclaves:** A partir de la clave de 56 bits, se generan un conjunto de subclaves para cada una de las 16 rondas de cifrado. Estas subclaves se derivan de la clave maestra utilizando una combinación de rotaciones y permutaciones.
3. **División del Texto en Bloques:** El texto original que se va a cifrar se divide en bloques de 64 bits. Si el texto no es un múltiplo de 64 bits, se realiza el relleno apropiado para que sea divisible.
4. **Cifrado por Rondas:** El texto en bloque se somete a una serie de 16 rondas de cifrado. En cada ronda, el bloque de texto se mezcla con una de las subclaves generadas. Cada ronda de cifrado implica operaciones como la sustitución de bits, la permutación y la función Feistel, que involucra la

división del bloque en dos mitades, una de las cuales se mezcla con la otra usando una función no lineal y operaciones XOR.

5. Finalización: Después de las 16 rondas, el bloque de texto se ha cifrado. El texto cifrado resultante es el resultado del último bloque después de todas las rondas de cifrado.

Descifrado con DES:

El proceso de descifrado DES es esencialmente el proceso de cifrado en reversa, utilizando las mismas subclaves pero en orden inverso. Los pasos son los siguientes:

1. Clave de Descifrado: Se utiliza la misma clave de cifrado para el proceso de descifrado. Las subclaves generadas a partir de la clave maestra se utilizan en orden inverso.
2. División del Texto en Bloques: Al igual que en el cifrado, el texto cifrado se divide en bloques de 64 bits.
3. Descifrado por Rondas: El texto cifrado se somete a las mismas 16 rondas de descifrado, pero las subclaves se utilizan en orden inverso al cifrado. En cada ronda, se realiza la operación inversa de las utilizadas en el cifrado.
4. Finalización: Después de las 16 rondas de descifrado, se obtiene el texto original en bloque.

Cabe destacar que DES se considera actualmente inseguro debido a su longitud de clave corta, lo que lo hace vulnerable a los ataques de fuerza bruta modernos. Por lo tanto, se ha recomendado el uso de algoritmos de cifrado más fuertes, como AES (Advanced Encryption Standard). DES se ha retirado gradualmente en muchas aplicaciones en favor de algoritmos de cifrado más seguros.

2. Dibuje un diagrama de flujo describiendo el algoritmo DES

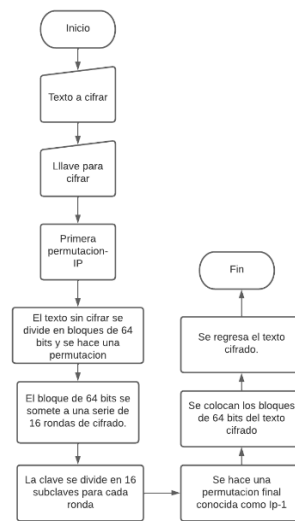


Figura1. Diagrama de flujo de algoritmo DES

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.

```

(base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./bruteforce00
Process 0 lower 0 upper 18014398509481983
Process 1 lower 18014398509481984 upper 36028797018963967
Process 2 lower 36028797018963968 upper 54043195528445951
Process 3 lower 54043195528445952 upper 72057594037927936
Process 0 found the key
Key = 123456

Esta es una prueba de proyecto 2
Process 0 exiting
Process 1 exiting
Process 3 exiting
Process 2 exiting
(base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$
  
```

Figura1. Prueba de funcionamiento del archivo dado

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

La función **"decrypt"** tiene como propósito descifrar un texto utilizando el algoritmo DES (Data Encryption Standard) en modo ECB (Electronic Code Book), proporcionando una capa adicional de seguridad a la información. En su proceso inicial, la función se encarga de establecer la paridad de la llave. Esto se logra mediante un bucle que desplaza un bit a la izquierda en cada iteración, aplicando una máscara para asegurar que el bit más significativo sea cero. El resultado, denotado como "k", representa una nueva llave con paridad par, garantizando que la cantidad de "1" en la representación binaria sea par.

Posteriormente, se inicia la clave DES utilizando la llave ajustada y se establece una paridad impar a través de la función "DES_set_odd_parity". Una vez obtenida esta llave modificada, se procede al análisis del mensaje. Se calcula la longitud del mensaje y se realiza el relleno necesario para que la longitud sea un múltiplo de 8 bytes, asegurando un procesamiento eficiente.

Finalmente, la función procede con el proceso de descifrado del mensaje mediante el algoritmo DES en modo ECB, empleando la función "DES_ecb_decrypt". Este proceso se realiza en bloques de 8 bytes mediante un bucle, culminando así el descifrado del mensaje de manera segura y eficaz. Finalmente, se realiza una verificación del relleno en el proceso.

La función **"encryptText"** despliega la capacidad de cifrar un texto mediante una llave utilizando el algoritmo DES (Data Encryption Standard) en modo ECB (Electronic Code Book). El inicio de su ejecución consiste en establecer la paridad de la llave. Este proceso se materializa a través de un bucle que desplaza un bit a la izquierda en cada iteración, seguido de la aplicación de una máscara para garantizar que el bit más significativo sea cero. La variable resultante, denominada "k", representa una nueva llave con paridad par, lo que implica que la cantidad de "1" en su representación binaria es par.

Posteriormente, se inicia la generación de la clave DES utilizando la llave ajustada, y se establece una paridad impar mediante la función "DES_set_odd_parity". Con la clave DES preparada, se procede al análisis del mensaje. Se calcula la longitud del

mismo y se añade el relleno necesario para que su extensión sea un múltiplo de 8 bytes, asegurando así un procesamiento eficiente.

Finalmente, la función cifra el mensaje utilizando el algoritmo DES en modo ECB mediante la función "DES_ecb_encrypt". Esta operación se lleva a cabo en bloques de 8 bytes mediante un bucle, reemplazando cada bloque con el bloque cifrado correspondiente.

Luego tenemos la función **tryKey (key, *ciph, len)**. Esta función asume la responsabilidad de probar la llave proporcionada y crear una copia del puntero del texto cifrado para evitar posibles condiciones de carrera que podrían ocasionar que un proceso intente descifrar un texto cifrado duplicado o incluso previamente descifrado. Finalmente, devuelve la indicación de si fue posible o no descifrar el mensaje. Este método recibe los siguientes parámetros:

1. key: Tipo long, representa la llave a probar.
2. ciph: Puntero de tipo char, es la ubicación del texto cifrado.
3. len: Tipo int, denota el número de caracteres en el texto cifrado.

El pseudocódigo de este método es el siguiente:

- Se crea un puntero de tipo char denominado "temp" con un tamaño igual a "len" + 1.
- A continuación, se lleva a cabo la copia del contenido de "ciph" en la variable "temp".
- La función "decrypt" se invoca con los argumentos "key", "temp", y "len", con el propósito de realizar el proceso de descifrado.
- Finalmente, la función retorna el valor 1 si logra romper el programa y 0 en caso contrario.

La función **"memcpy"** se encarga fundamentalmente de realizar la copia de una cantidad específica, denotada por 'n', de elementos presentes en el área de memoria del objeto fuente hacia el área de memoria del objeto destino.

Y por último, la función **"strstr"** se emplea para identificar la primera aparición de un subcadena que se especifica como un parámetro en un texto determinado.

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

a. MPI_Recv

- La función "MPI_Irecv" se utiliza para realizar una recepción no bloqueante. Esto significa que se inicia una operación de recepción, pero el programa no espera hasta que se complete la recepción antes de continuar con otras tareas.
- En el código, se inicia una recepción no bloqueante de un valor largo ("MPI_LONG") desde cualquier fuente ("MPI_ANY_SOURCE") y con cualquier etiqueta ("MPI_ANY_TAG"). Esto se hace en un bucle para verificar si otro proceso ha encontrado la clave.
- La función "MPI_Test" se utiliza para verificar si la recepción ha tenido éxito y si se ha encontrado la clave por parte de otros procesos. Si se encuentra la clave, se establece la variable "ready" en verdadero, y el bucle se rompe.

b. MPI_Send

- La función "MPI_Send" se utiliza para enviar datos desde un proceso a otro en el entorno MPI. En el código, se utiliza para avisar a otros procesos cuando se encuentra la clave.
- Cuando un proceso encuentra la clave, utiliza "MPI_Send" para enviar el valor de la clave ("found") a todos los demás procesos en el comunicador ("comm").

c. MPI_Wai

- La función "MPI_Wait" se utiliza para esperar a que una operación no bloqueante se complete. En este caso, se utiliza para esperar a que la

operación de recepción ("MPI_Irecv") se complete antes de continuar con la ejecución del programa.

- En el código, solo el proceso con ID 0 ("id == 0") espera a que se complete la operación de recepción antes de descifrar el texto y mostrar el resultado.

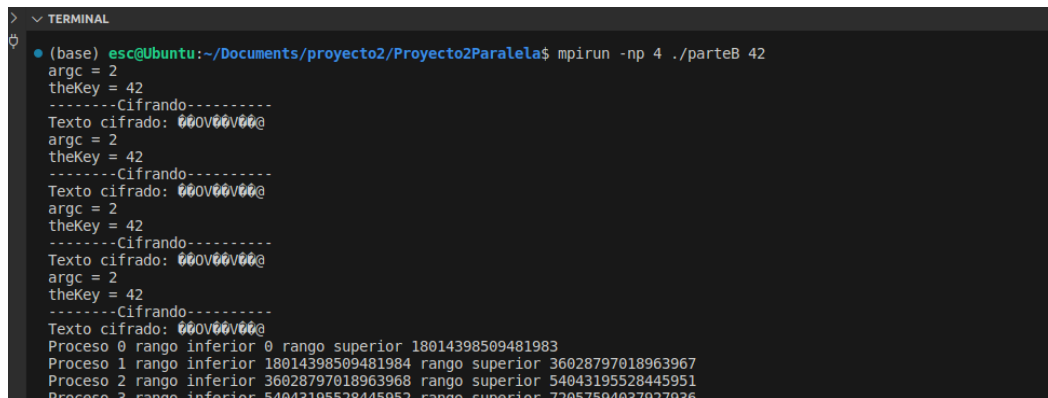
Flujo de Comunicación

1. Los procesos inician la comunicación MPI y obtienen su ID de proceso.
2. El texto original se cifra con una clave predefinida ("the_key") y se almacena en el búfer "cipher".
3. Los procesos dividen el espacio de búsqueda de claves ("upper") en partes iguales y se asignan tareas para buscar la clave en su rango.
4. Cada proceso inicia una operación de recepción no bloqueante para escuchar si otros procesos encuentran la clave.
5. Cada proceso realiza una búsqueda de fuerza bruta en su rango de claves utilizando la función "tryKey".
6. Si un proceso encuentra la clave, envía la clave a todos los demás procesos mediante la función "MPI_Send".
7. Si el proceso con ID 0 recibe una clave a través de "MPI_Recv", espera a que se complete la operación utilizando "MPI_Wait".
8. Una vez que se ha completado la operación de recepción, el proceso con ID 0 descifra el texto utilizando la clave encontrada y muestra el resultado.
9. Todos los procesos finalizan la comunicación MPI y terminan su ejecución.

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)

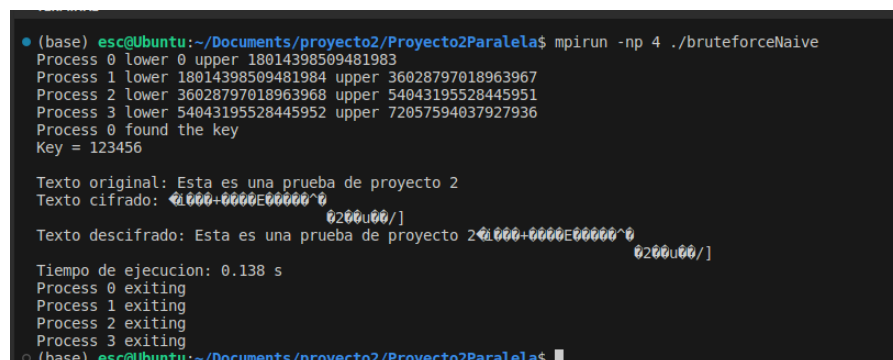
Clave que se usó de ejemplo: 42



```
> v TERMINAL
• (base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./parteB 42
argc = 2
theKey = 42
-----Cifrando-----
Texto cifrado: 000V00V00@
argc = 2
theKey = 42
-----Cifrando-----
Texto cifrado: 000V00V00@
argc = 2
theKey = 42
-----Cifrando-----
Texto cifrado: 000V00V00@
argc = 2
theKey = 42
-----Cifrando-----
Texto cifrado: 000V00V00@
Proceso 0 rango inferior 0 rango superior 18014398509481983
Proceso 1 rango inferior 18014398509481984 rango superior 36028797018963967
Proceso 2 rango inferior 36028797018963968 rango superior 54043195528445951
Proceso 3 rango inferior 54043195528445952 rango superior 72057594037927936
```

Figura2. Cifrando texto con clave 42.

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/decifrar: “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



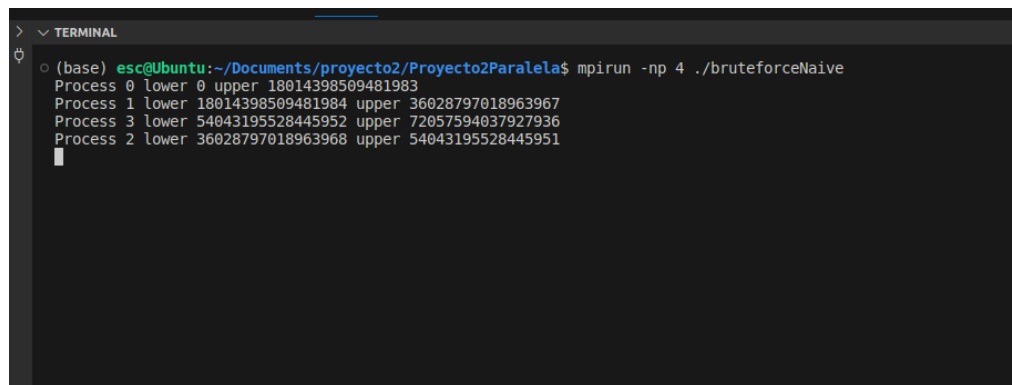
```
• (base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./bruteforceNaive
Process 0 lower 0 upper 18014398509481983
Process 1 lower 18014398509481984 upper 36028797018963967
Process 2 lower 36028797018963968 upper 54043195528445951
Process 3 lower 54043195528445952 upper 72057594037927936
Process 0 found the key
Key = 123456

Texto original: Esta es una prueba de proyecto 2
Texto cifrado: 0000+0000E00000^0
0200u00/]
Texto descifrado: Esta es una prueba de proyecto 2 0000+0000E00000^0
0200u00/]

Tiempo de ejecución: 0.138 s
Process 0 exiting
Process 1 exiting
Process 2 exiting
Process 3 exiting
• (base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$
```

Figura3. Realización de prueba cifrando texto

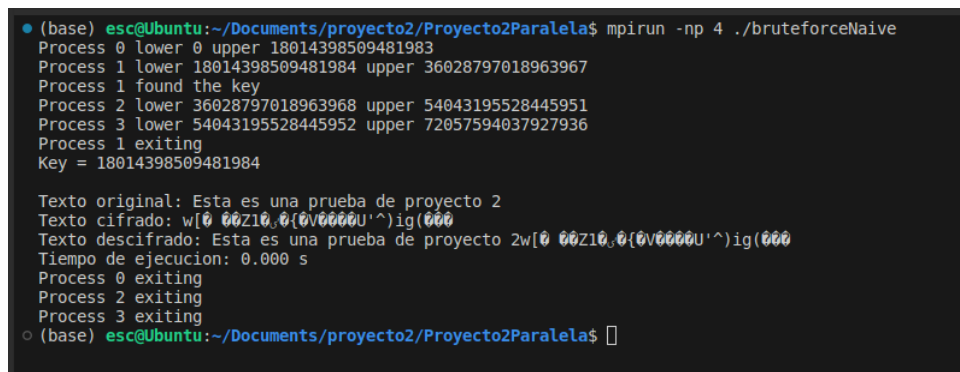
- b. Mida el tiempo de ejecución en romper el código usando la llave: $(2 \cdot 56/4)$. O sea, 18014398509481983L. [spoiler: se tardará mucho, si es que termina, no se ofusquen si no termina].



```
(base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./bruteforceNaive
Process 0 lower 0 upper 18014398509481983
Process 1 lower 18014398509481984 upper 36028797018963967
Process 3 lower 54043195528445952 upper 72057594037927936
Process 2 lower 36028797018963968 upper 54043195528445951
█
```

Figura4. Medición de tiempos con una llave específica

- c. Mida el tiempo de ejecución en romper el código usando la llave: $(2 \cdot 56/4) + 1$. O sea, 18014398509481984L.



```
• (base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./bruteforceNaive
Process 0 lower 0 upper 18014398509481983
Process 1 lower 18014398509481984 upper 36028797018963967
Process 1 found the key
Process 2 lower 36028797018963968 upper 54043195528445951
Process 3 lower 54043195528445952 upper 72057594037927936
Process 1 exiting
Key = 18014398509481984

Texto original: Esta es una prueba de proyecto 2
Texto cifrado: w[0 00Z10,0(0V0000U'^)ig(000
Texto descifrado: Esta es una prueba de proyecto 2w[0 00Z10,0(0V0000U'^)ig(000
Tiempo de ejecución: 0.000 s
Process 0 exiting
Process 2 exiting
Process 3 exiting
(base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ █
```

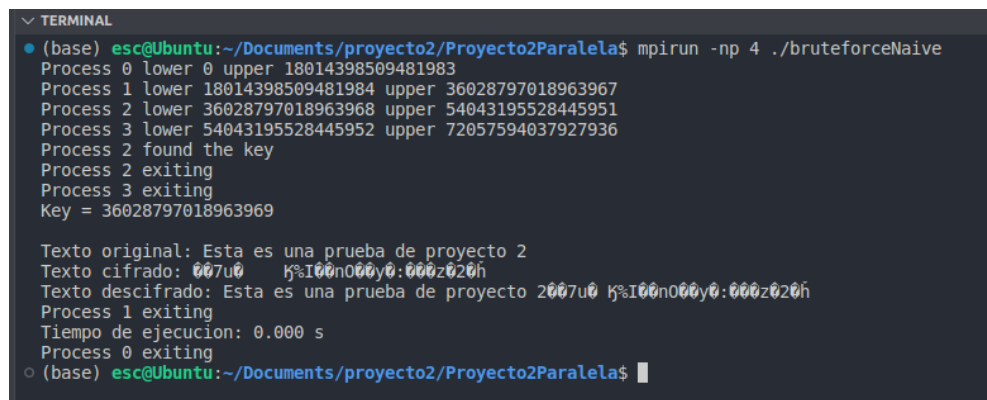
Figura5. Tiempo de ejecución con llave específica

- d. Reflexione lo observado y el comportamiento del tiempo en función de la llave.

Se evidencia que el tiempo de ejecución está vinculado a la ubicación del bloque que contiene la llave seleccionada. Por ejemplo, en el caso de la llave especificada en el inciso B, el programa sigue en ejecución después de cierto tiempo, ya que la llave está en la última posición del primer proceso, lo que implica que será encontrada en la última iteración. En contraste, en el inciso C, la llave está en la primera posición del bloque del segundo proceso, lo que permite encontrarla en la primera iteración y, en consecuencia, el tiempo de ejecución es prácticamente nulo.

Un aspecto interesante del presente problema/temario es que los speedups y tiempos paralelos obtenidos son sumamente inconsistentes y dependientes de la llave elegida. Ello debido a que estamos recorriendo el espacio de datos de forma “naive” (incremental y en orden y dividiendo equitativamente los segmentos). Por ejemplo (tomando como base el inciso anterior), asumiendo 4 procesos, y eligiendo como llave $(2^{56})/4 + 1$, podemos ver que el proceso #2 (al que se le asigna el segundo segmento de datos) encontrará la llave en el primer intento ($T_{par}=1$ iter). Un algoritmo secuencial le hubiera tomado $(2^{56})/4 + 1$ iteraciones, por lo que el speedup es de $(2^{56})/4 + 1$ (algo sumamente alto y que nos puede dar falsa confianza en nuestro algoritmo). Caso contrario, si la llave fuera $(2^{56})/4$ el proceso #1 la encontraría en su última iteración. Podemos notar que un programa secuencial le tomaría la misma cantidad de iteraciones encontrar esa llave, por lo que no obtendremos speedup alguno en este caso ($speedup=1$). Para poder ver y comprender tal fenómeno podemos realizar cambios a la llave privada y analizar el desempeño en función de ellas; asumiendo 4 procesos (ojo, adaptar dependiendo de los procesos que usen):

d. Una llave fácil de encontrar, por ejemplo, con valor de $(2^{56}) / 2 + 1$



```

▼ TERMINAL
• (base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./bruteforceNaive
Process 0 lower 0 upper 18014398509481983
Process 1 lower 18014398509481984 upper 36028797018963967
Process 2 lower 36028797018963968 upper 54043195528445951
Process 3 lower 54043195528445952 upper 72057594037927936
Process 2 found the key
Process 2 exiting
Process 3 exiting
Key = 36028797018963969

Texto original: Esta es una prueba de proyecto 2
Texto cifrado: 007u0 K%I00n000y0:000z020h
Texto descifrado: Esta es una prueba de proyecto 2007u0 K%I00n000y0:000z020h
Process 1 exiting
Tiempo de ejecucion: 0.000 s
Process 0 exiting
• (base) esc@Ubuntu:~/Documents/proyecto2/Proyecto2Paralela$

```

Figura6. Prueba con una llave fácil

e. Una llave medianamente difícil de encontrar, por ejemplo, con valor de $(2^{56}) / 2 + (2^{56}) / 8$

```

156
157 char the_key[] = "45035996273704960";
158
159
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
TERMINAL
(base) esc@ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./bruteforceNaive
Process 0 lower 0 upper 18014398509481983
Process 1 lower 18014398509481984 upper 36028797018963967
Process 2 lower 36028797018963968 upper 54043195528445951
Process 3 lower 54043195528445952 upper 72057594037927936
[]
(base) esc@ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ python
Python 3.11.5 (main, Sep 11 2023, 13:54:46) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = ((2 ** 56)/2) + ((2 ** 56)/8)
>>> print(a)
4.503599627370496e+16
>>> '{:f}'.format(a)
'45035996273704960.000000'
>>> aInt = int(a)
>>> print(str(aInt))
45035996273704960
>>> []

```

Figura7. Prueba con una llave de dificultad intermedia

f. Una llave difícil de encontrar, por ejemplo, con valor de $(2^{56}) / 7 + (2^{56}) / 13$ aproximados al entero superior

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
TERMINAL
(base) esc@ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ mpirun -np 4 ./bruteforceNaive
Process 0 lower 0 upper 18014398509481983
Process 1 lower 18014398509481984 upper 36028797018963967
Process 2 lower 36028797018963968 upper 54043195528445951
Process 3 lower 54043195528445952 upper 72057594037927936
[]
(base) esc@ubuntu:~/Documents/proyecto2/Proyecto2Paralela$ python
Python 3.11.5 (main, Sep 11 2023, 13:54:46) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = ((2 ** 56)/2) + ((2 ** 56)/8)
>>> print(a)
4.503599627370496e+16
>>> '{:f}'.format(a)
'45035996273704960.000000'
>>> aInt = int(a)
>>> print(str(aInt))
45035996273704960
>>> b = ((2 ** 56)/7) + ((2 ** 56)/13)
>>> print(str(b))
1.5836833854489656e+16
>>> bInt = int(b)
>>> print(str(bInt))
15836833854489656
>>> []

```

Figura8. Prueba con una llave de dificultad alta

En este punto nos podemos percatar de algo: el tiempo paralelo es una función del número de procesos y la llave utilizada - $tPar(n,k)$. De ello sigue una observación importante: nos interesa saber el valor esperado de $tPar$ para poder tener una mejor medida del desempeño del algoritmo. Nótese que cada posible llave $[0-2^{56}]$ tiene la misma probabilidad de ser elegida ($1/ 2^{56}$). Se puede demostrar que el valor esperado para el approach “naive” mencionado es (tip: graficar ayuda, fórmula de Gauss suma números consecutivos también):

$$E[tPar(n, k)] = \sum_i^n x_i p_i = \frac{2^{55}}{n} + 1/2$$

6. (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.

7. 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:

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

b. Derivar el valor esperado de $t_{Par}(n,k)$ 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.

c. Implementelo en código y pruébelo con 3-4 llaves (fáciles, medianas, difíciles). Compare el tiempo medido con el tiempo pronosticado por su función $t_{Par}(n,k)$

Conclusiones

Descifrar un algoritmo mediante fuerza bruta y hallar la clave correspondiente puede ser un proceso computacionalmente desafiante. Este procedimiento requiere un análisis cuidadoso para segmentar la búsqueda, evitando inconsistencias en las mediciones de rendimiento. Además, la paralelización de esta tarea en conjuntos de datos extensos demuestra ser una estrategia efectiva para reducir significativamente el tiempo necesario para obtener la clave de encriptación. Cada proceso opera en un rango específico de números, optimizando así la solución global.

Por otra parte, la viabilidad y eficacia del enfoque de fuerza bruta están intrínsecamente ligadas al tamaño del espacio de claves y la longitud del texto cifrado. Es crucial considerar estos factores al evaluar la aplicabilidad de este método.

Recomendaciones

La modularización del código mediante distintas funciones facilita un análisis más rápido de posibles inconsistencias, especialmente en los procesos de encriptación y desencriptación. Esta práctica contribuye a la legibilidad y mantenimiento del código. Paralelamente, la implementación de MPI (Message Passing Interface) demuestra ser beneficiosa en procesos de fuerza bruta. Proporciona optimización en operaciones y procesos, mejorando la eficiencia general del código.

Por último, antes de la programación, se recomienda realizar una investigación exhaustiva para evitar errores de ejecución por falta de comprensión teórica. Por ejemplo, la limpieza del buffer se destaca como una práctica esencial para prevenir la presencia de caracteres inesperados al final del mismo.