PROFESORES:

Sandra Julieta Rueda Rodriguez
Jesus David Borre Ordosgoitia

INTEGRANTES:

Kevin Steven Gamez Abril (201912514)
Sergio Julian Zona Moreno (201914936)

Tabla de contenido

1 Introducción

En este documento se presenta la solución del caso 3 de estudio del curso Infraestructura Computacional. Basados en el modelo presentado durante el caso 2, se añadirán diversos Threads que ejecuten las pruebas con el fin de analizar el rendimiento de la máquina en cada uno de los diversos escenarios; y, a partir de estos, generar una conclusión al respecto de acuerdo a lo aprendido durante el módulo 3 del curso.

1.1 Aclaraciones preliminares

- Se recomienda al calificador/lector de este documento visualizarlo con un Zoom de 150% para evitar forzar la vista. Además, este hecho permite ver con nitidez los gráficos y las tablas presentadas.
- Dada la cantidad de datos obtenidos durante las pruebas, se procede a omitir muchos de los mismos en el texto (para evitar generar una extensión innecesaria). Sin embargo, toda la recolección de datos se encuentra anexa en la carpeta ".docs".

2 Especificaciones de la máquina utilizada

A pesar de que se nos fue aprovisionada una máquina virtual para el desarrollo de este ciclo. Mi compañero y yo optamos por utilizar otra máquina virtual de mayor potencia. Esta máquina es perteneciente a AdmonSis y cuenta con las siguientes características:

- Procesador Intel(R) Xeon(R) CPU ES-2640 v3 @ 2.60GHz 2.59 GHz, con 8 núcleos.
- 16 GB de memoria RAM disponible.
- Arquitectura de 64 bits.
- Espacio asignado para la JVM (Java Virtual Machine) de 15 GB.

3 Explicación de la implementación realizada.

Nuestra aplicación consta de 5 clases principales donde se ejecutan los procesos para poder cumplir con los requerimientos. Estas son:

- **Main:**
  Clase principal de aplicación, donde se lanzan los procedimientos. Ejecuta una interfaz en la consola interna del entorno de desarrollo y por medio de interacciones con el usuario ejecuta solicita la ejecución de los requerimientos a la clase `Hash.java.`
- **Hash**:
  Clase que tiene todos los métodos necesarios de ejecución. Allí se obtiene el código criptográfico de Hash según un algoritmo y una entrada (además realiza el proceso inverso). Desde esta clase inicializan los threads de combinaciones que identificarán una entrada por fuerza bruta.
  Para el método de `identificar_entrada` se implementaron dos factores importantes: un ataque por diccionario y un ataque por fuerza bruta. El ataque por diccionario es el primer intento por obtener la entrada, sin embargo, en la mayoría de casos esto será insuficiente, por lo que se procede a comprobar todas las combinaciones posibles de valores y comparar sus códigos de Hash para obtener el resultado deseado. ¿Por qué implementamos un diccionario si ya teníamos un ataque por fuerza bruta? La explicación a esta pregunta radica en la complejidad, el ataque de diccionario que utilizamos permite adquirir cadenas de manera constante sin la necesidad de recorrer todas las combinaciones; si bien es poco probable que un conjunto de letras se encuentre en el diccionario, su uso facilita el reconocimiento de alrededor de 300.000 palabras (y solamente pesa 8 MB).
- **Ataque diccionario**:
  Es la clase que almacena todos los métodos necesarios para el manejo de la base de datos embebida. En este caso, mi compañero y yo optamos por utilizar Apache Derby. Una librería de Software libre que permite almacenar datos por medio de sentencias .SQL. Dentro de esta clase, se encuentran métodos para obtener valor, e incluso métodos que permiten añadir más diccionarios a la base de datos.
- **Combinaciones**:

Es la clase que confirmará por fuerza bruta todas las posibles cadenas dada una longitud. El código principal de recorrido fue tomado de GeekForGeeks, se da autoría total a las personas que lo desarrollaron. Es en esta clase donde son lanzados los threads. Por este motivo, se maneja un atributo estático denominado `encontrado` que vigila constantemente los threads para detenerlos cuando uno de ellos encuentre la coincidencia necesaria. Para poder devolver el valor, se utiliza un modelo `Observable-Observador`. Donde la clase `Hash.java` observa todos los threads de `Combinaciones.java` a la espera de una notificación del valor.

- **Simétrico**:
  Para garantizar la integridad de los datos, se ejecuta una clase simétrico que cifra el código criptográfico de hash con el fin de evitar su modificación. Esta clase es llamada en el `Main.java` para la ejecución del requerimiento.

4 Modificaciones realizadas y análisis del peor caso

Desde el caso 2 nuestro grupo tenía desarrollada la implementación de los Threads. Sin embargo, enviaba un thread por cada letra del alfabeto y fue necesario modificar este factor para permitir una solución con 1, 2, 4 y 8 threads. Por estos motivos, los métodos modificados para cumplir con el requerimiento fueron:

- **identificar_entrada() en la clase Hash.java:**
  En este método se quitó el recorrido que comprobaba todas las cadenas de caracteres menores a 7 caracteres. Y simplemente se dejó fijo para verificar longitudes de 7.

```
//Se implementa un ataque de diccionario primero.
resultado = diccionario.obtenerValor(algoritmo, codigoHash);

//Fuerza bruta en caso de que el ataque por diccionario falle.
if (resultado.compareTo("") == 0)
{
    int numCaracteres=7;
    init(algoritmo, codigoHash, numCaracteres, numThreads);
}
```

  Por otro lado, para poder registrar el uso de CPU, se implementó un Timer que realizaba una tarea cada 5 minutos. Esta tarea era la ejecución del método que fue entregado durante el enunciado del caso. La implementación del Timer fue obtenida del canal de YouTube “Profe Javier” (se anexa enlace del vídeo al final del documento).

```
//Timer que obtiene el porcentaje de uso de CPU cada 5 minutos.
Timer timer = new Timer();
TimerTask tarea = new TimerTask() {
    @Override
    public void run() {
        try {
            System.out.println("Porcentaje de uso de CPU: "+getSystemCpuLoad());
        } catch (Exception e) {

        }
    }
};
timer.schedule(tarea, 0, 300000);

while(this.darResultado().compareTo("")==0)
{
    //Método que realiza una espera activa hasta que llegue el resultado de búsqueda en los Threads.
}
timer.cancel();
timer.purge();
System.out.println("Porcentaje de uso de CPU: "+getSystemCpuLoad());
return resultado;


    /**
     * Método que devuelve el uso del CPU.
     * @return Porcentaje de uso del CPU.
     * @throws Exception En caso de que ocurra un error.
     */
    public double getSystemCpuLoad() throws Exception {
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        ObjectName name = ObjectName.getInstance("java.lang:type=OperatingSystem");
        AttributeList list = mbs.getAttributes(name, new String[]{ "SystemCpuLoad" });
        if (list.isEmpty()) return Double.NaN;
        Attribute att = (Attribute)list.get(0);
        Double value = (Double)att.getValue();
        // usually takes a couple of seconds before we get real values
        if (value == -1.0) return Double.NaN;
        // returns a percentage value with 1 decimal point precision
        return ((int)(value * 1000) / 10.0);
    }
```

- **init() en la clase Hash.java:**
  En el método init, se añadió un nuevo parámetro que indica el número de threads, y la "letra" del prefijo, ahora es implementada como letra identificadora del thread. La implementación que terminó fue la siguiente:

```
/**
 * Método que devuelve el resultado de la palabra buscada.
 * @param codigo. Código criptográfico ingresado.
 * @param algoritmo   Algoritmo que será utilizado.
 * @param pNumThreads Número de threads que serán ejecutados para la búsqueda.
 * @param pNumCaracteres Número de caracteres.
 */
public synchronized void init(String algoritmo, byte[] codigo, int pNumCaracteres, int pNumThreads) {
    encontrado =false;
    hilos = new ArrayList<Thread>();
    classHilos = new ArrayList<Combinaciones>();
    for(int i=0; i<pNumThreads; ++i)
    {
        Combinaciones hilo = new Combinaciones((i+1)+"", pNumCaracteres-1, codigo, algoritmo, this, pNumThreads);
        classHilos.add(hilo);
        Thread t = new Thread(hilo);
        hilos.add(t);
        t.start();
    }
}
```

- **darListaCombinaciones() en la clase Combinaciones.java:**
  Este fue el método que tuvo un cambio más drástico, debido a que su implementación se modificó para que cada thread recorriera una parte específica del alfabeto. Por ejemplo, si eran 2 threads los que iban a ser lanzados, entonces el primer thread recorría todas las combinaciones que empezaron por las letras "a-n". Mientras que el segundo thread recorrería todas las letras que fuesen desde la "n-z". Por obvios motivos, 1 thread va a acabar primero que el otro, y esto se debe a que un thread tiene que revisar una letra más que el otro thread. Por este motivo, **los casos para la peor cadena ya no son por orden lexicográfico, sino que dependen del número de threads que sean lanzados.**

Con el contexto del método `darListaCombinaciones()`, se procede a presentar los peores casos según el número de threads:

- Para 1 thread, el peor caso es una cadena con "zzzzzzz". Debido a que este thread realizará una revisión lexicográfica.
- Para 2 threads, el peor caso es una cadena con "nnnnnn". Debido a que el thread 1, revisa desde la "a-n", mientras que el thread 2 desde la "ñ-z". Entonces, el thread 1 debe revisar una letras más comparativamente hablando respecto al thread 2.
- Para 4 threads, el peor caso es una cadena con "ttttttt" (o cualquier cadena final correspondiente a un thread diferente del último, depende de las colisiones). Debido a que la cadena es revisada por el penúltimo thread. Y el último thread revisa desde la "u-z" (tiene una letra menos de revisión que los otros threads).
- Para 8 threads, el peor caso es una cadena con "zzzzzzz". Debido a que el último thread revisa desde la "u-z", es decir, revisa 3 letras más que el resto de threads, por lo que su ejecución tardará más tiempo.

5 Monitores utilizados y su implementación

Para la solución del proyecto se utilizaron dos monitores: uno para el uso de la CPU, y el otro para conocer el tiempo de los threads.

Como fue indicado en la sección 4 de este documento, para medir el uso de la CPU fue implementado un `Timer` que ejecutaba la acción cada 5 minutos. Se solicita al evaluador revisar nuevamente dicha sección para observar el código correspondiente.

Por otro lado, para obtener el tiempo de ejecución de los threads, se utilizaron medidas del sistemas con el método `System.currentTimeMillis()`. Se utilizó un flag inicial, y flag final, y se restó la diferencia de tiempos para obtener la tardanza. Esto fue realizado en el método `run()` de la clase `Combinaciones.java`. A continuación, se presenta la implementación:

```
/**
 * Método que inicia la ejecución de un Thread.
 */
public synchronized void run()
{
    long inicio = System.currentTimeMillis();
    while(!encontrado) {
        palabra = "";
        darListaCombinaciones(caracteres, letra, numThreads);
    }
    long fin = System.currentTimeMillis();
    System.out.println("El Thread "+letra+ " tardó en ejecutarse "+ (fin-inicio)+ " milisegundos.");
}
```

6 Gráficas y análisis de los resultados obtenidos

Si bien algunos datos serán presentados en esta sección, se recomienda al evaluador visualizar el documento de Excel que se encuentra en la carpeta ".docs". Allí se encuentran al completo todos los datos obtenidos durante la ejecución de cada uno de los escenarios.

6.1 Evaluación de la calidad de los datos

Para este inciso analizaremos los escenarios de "Prueba 1" obtenidos durante la ejecución de cada uno de los algoritmos, y veremos que tan congruentes son nuestros datos con respecto a los valores esperados. Nótese que el **tiempo** se encuentra en **minutos**. Además, también se debe aclarar que parte del aumento en el tiempo de ejecución (por ejemplo, casi 4 horas con 1 thread SHA-512), se debe a la implementación de los monitores que añaden complejidad al código.

Se observa que entre mayor es la cantidad de threads, existe un aumento en el porcentaje del CPU utilizado. Al parecer, nuestra máquina ubica un tope alrededor del 65% como umbral de uso (pero aún con lo anterior, sigue siendo una potencia demasiado elevada). Al ejecutar los 8 threads, se puede observar que en la parte final de ejecución solamente encontramos 1 thread ejecutando, por lo que el consumo es igual al de la primera columna (debido a nuestra implementación, el último thread cuando lanzamos 8 tendrá que revisar 3 letras adicionales comparativamente respecto a los demás threads).

Otro factor importante es el hecho de que los tiempos de ejecución disminuyen a medida que se añaden más threads (la inconsistencia en el último thread se debe a nuestra implementación, sin embargo, sigue un patrón similar). Al enviar un solo thread, el tiempo de ejecución es bastante largo, mientras que al ejecutar 4 threads se obtiene un nivel óptimo. Eso puede ser visualizado en cualquiera de los 3 escenarios.

Por los anteriores motivos, se considera que los datos son válidos y cumplen con los criterios mínimos de experimentación. Esto se debe a que el uso de la máquina fue exclusivo para la ejecución de este programa y no se alteraron variables externas que pudiesen generar perturbaciones en los resultados.

Para MD5:

Prueba 1	Threads			
Tiempo	1	2	4	8
5	25,7%	39,7%	62,4%	64,5%
10	25,9%	38,8%	61,6%	64,5%
15	25,4%	38,2%	62,1%	64,7%
20	25,4%	38,0%	62,2%	64,6%
25	25,4%	38,0%	62,3%	65,0%
30	25,7%	37,8%	61,7%	64,5%
35	25,7%	39,1%	63,5%	64,7%
40	25,9%	37,9%		64,6%
45	25,4%	37,8%		64,5%
50	25,4%	37,8%		51,6%
55	25,3%	37,9%		25,7%
60	25,3%	39,7%		25,3%
65	25,8%	39,3%		25,6%
70	25,5%	38,5%		
75	25,5%	49,7%		
80	25,4%	39,4%		
85	25,3%	37,9%		
90	25,4%	37,8%		
95	25,8%			
100	25,2%			
105	25,1%			
110	25,6%			

Para SHA-256:

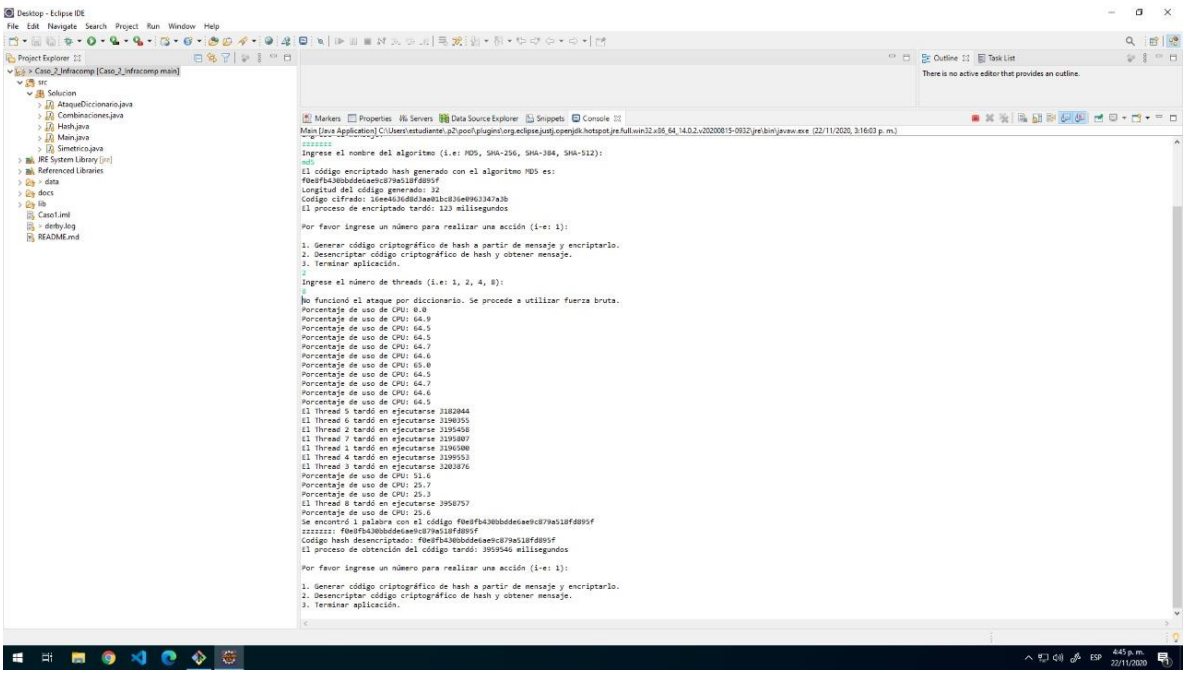
Prueba 1	Threads			
Tiempo	1	2	4	8
5	37,7%	37,7%	69,0%	65,9%
10	37,5%	37,9%	67,1%	65,5%
15	34,0%	38,5%	63,8%	65,5%
20	36,9%	37,8%	62,9%	66,0%
25	37,8%	37,8%	63,1%	65,9%
30	36,2%	37,8%	62,9%	65,7%
35	29,6%	37,8%	62,7%	67,2%
40	32,1%	37,9%		65,6%
45	25,7%	38,6%		65,4%
50	25,4%	37,8%		65,2%
55	25,4%	38,0%		65,4%
60	26,4%	37,9%		65,5%
65	26,9%	37,8%		32,3%
70	25,4%	38,0%		25,2%
75	25,5%	38,6%		25,3%
80	25,4%	37,8%		25,1%
85	25,4%	38%		
90	26,3%			
95	25,4%			
100	25,5%			
105	25,5%			
110	26,2%			
115	25,6%			
120	26,4%			
125	25,8%			
130	25,7%			

Para SHA-512:

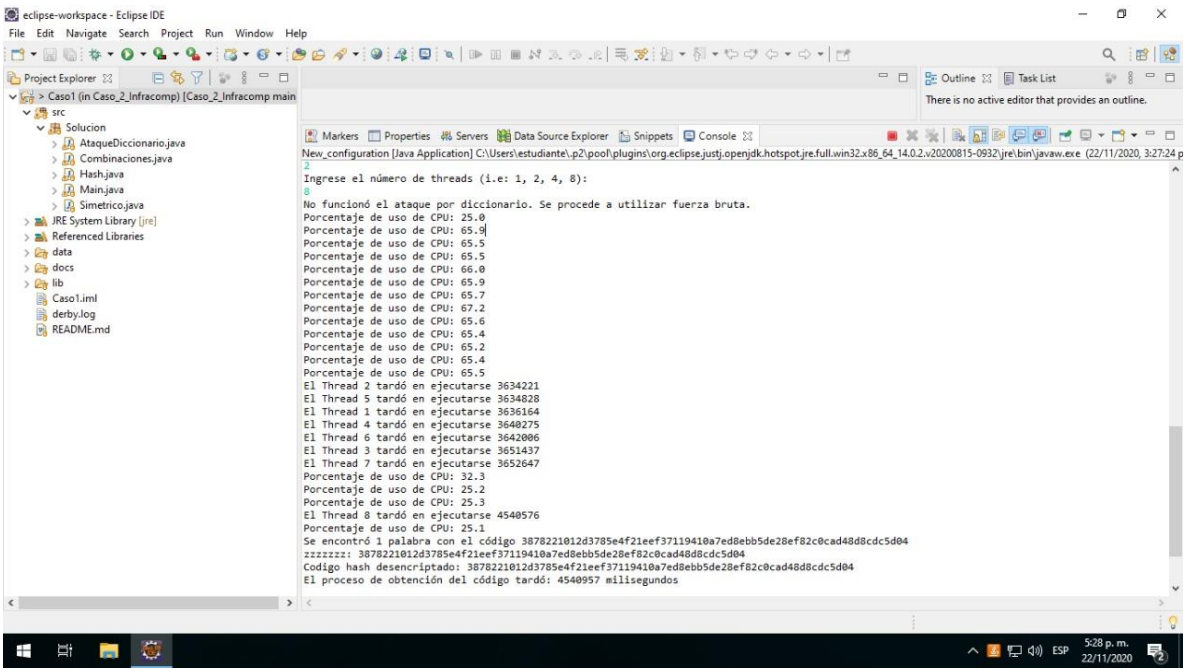
Prueba 1	Threads			
Tiempo	1	2	4	8
5	26,0%	38,5%	67,3%	65,8%
10	25,4%	38,3%	69,3%	65,7%
15	25,4%	38,9%	62,9%	65,6%
20	26,4%	37,9%	62,4%	65,9%
25	25,5%	38,0%	63,1%	66,2%
30	26,3%	38,0%	62,4%	66,3%
35	25,6%	37,9%	62,4%	65,8%
40	25,4%	38,1%	63,1%	65,9%
45	25,4%	38,6%	63,5%	65,9%
50	26,4%	37,8%	62,4%	65,7%
55	25,5%	38,2%		65,5%
60	25,4%	39,6%		66,2%
65	25,5%	40,5%		65,9%
70	25,4%	39,7%		66,7%
75	25,3%	39,5%		31,2%
80	26,5%	39,9%		25,7%
85	25,4%	38,4%		25,6%
90	25,4%	39,0%		33,8%
95	25,4%	39%		
100	25,5%	38%		
105	25,4%	39%		
110	26,3%	39%		
115	25,4%	40%		
120	25,4%	38%		
125	25,4%	40%		
130	25,4%	38%		
135	25,4%			
140	26,5%			
145	25,4%			
150	25,8%			
155	27,1%			
160	27,4%			
165	27,1%			
170	26,8%			
175	25,5%			
180	25,4%			
185	25,7%			
190	25,4%			
195	25,4%			
200	26,4%			
205	25,4%			
210	25,4%			
215	25,4%			
220	25,4%			
225	25,4%			
230	26,2%			
235	25,9%			
240	28,9%			
245	25,5%			
250	25,4%			
255	25,4%			

Para demostrar que estos datos no fueron alterados, se presentan algunas capturas de pantalla que demuestran su veracidad:

MD5 --- 8 Threads



SHA-256 --- 8 Threads

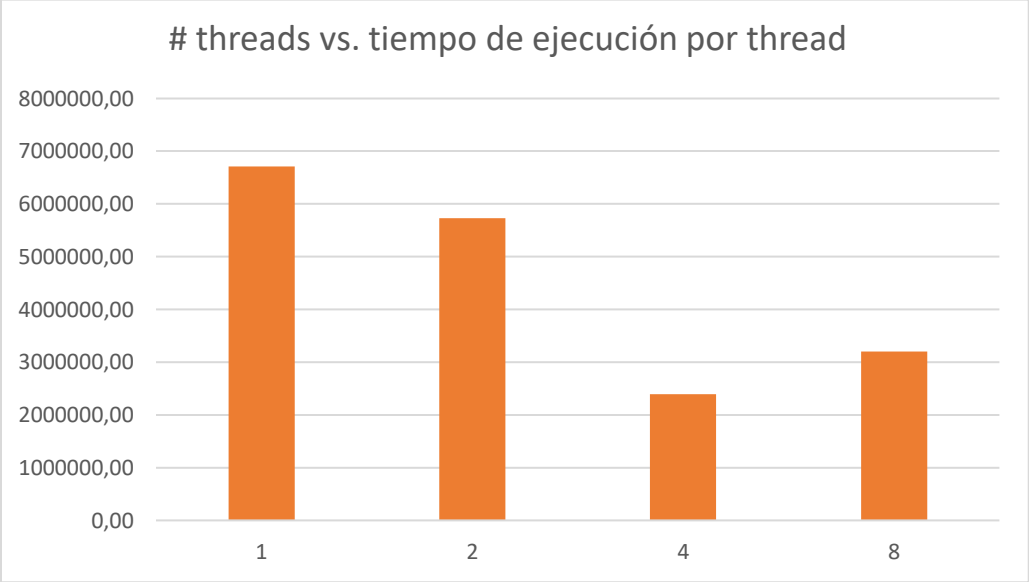


Para las demás ejecuciones se obtuvieron los valores directamente de la consola de comandos y se consignaron en el Excel.

6.2 Gráficas

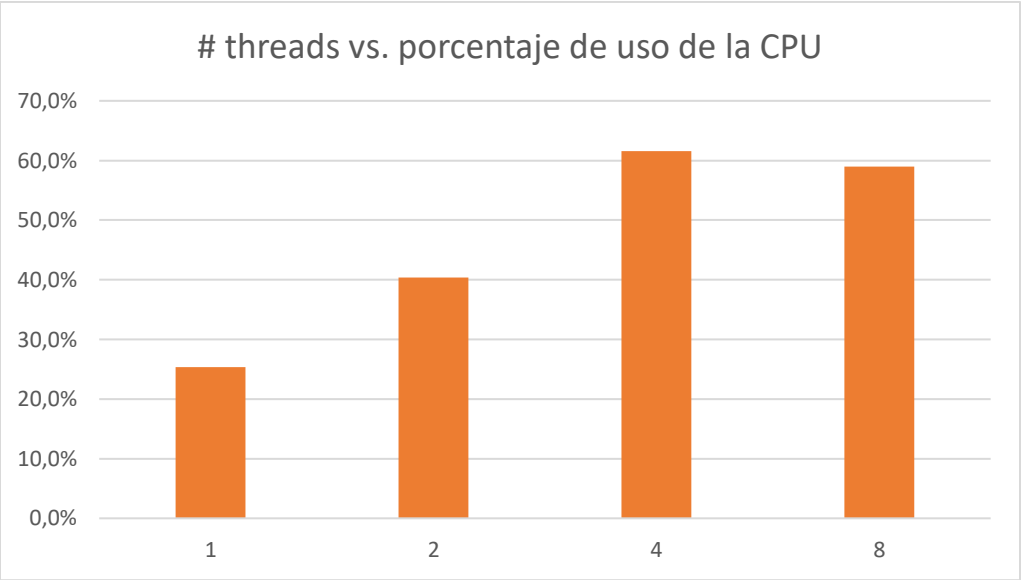
MD5:

Unificado	Tiempo de ejecución Thread de las 3 pruebas			
Datos estadísticos	**1**	**2**	**4**	**8**
Media de los escenarios	6709721,00	5731829,33	2394675,83	3199042,38



Datos estadísticos	Valores
Media	4508817,14
Mediana	4465435,85
Desviación estándar	2043268,63

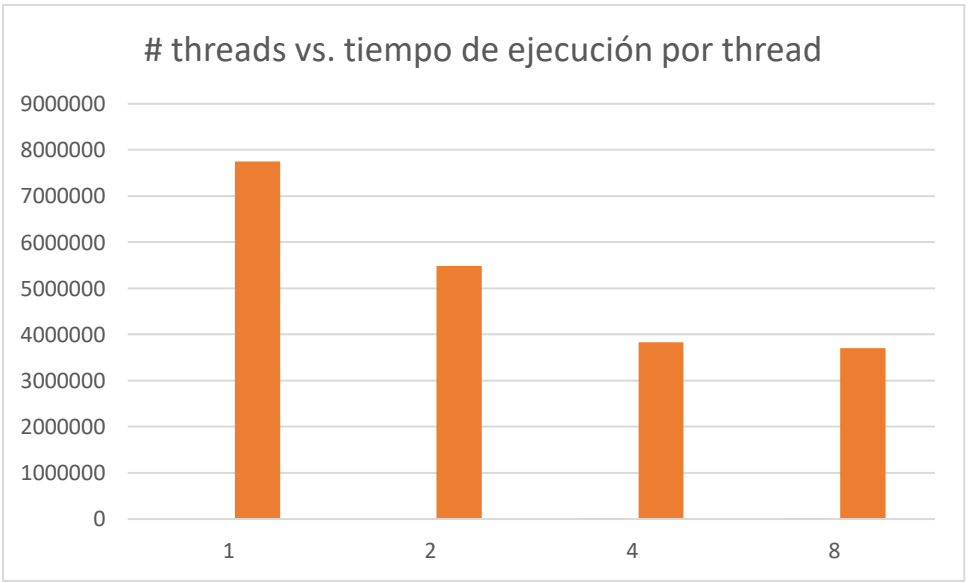
Unificado	Porcentaje de uso del procesador de las 3 pruebas			
Datos estadísticos	**1**	**2**	**4**	**8**
Media de los escenarios	25,3%	40,3%	61,6%	59,0%



Datos estadísticos	Valores
Media	0,47
Mediana	0,50
Desviación estándar	0,17

SHA-256:

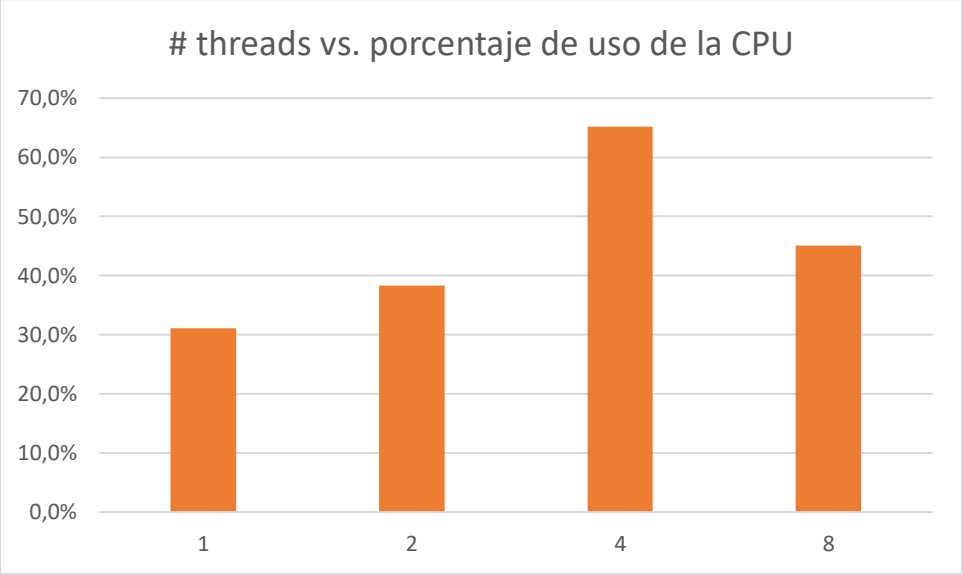
Unificado	Tiempo de ejecución Thread de las 3 pruebas			
Datos estadísticos	**1**	**2**	**4**	**8**
Media de los escenarios	7748904,33	5482956,17	3831126,50	3703055,04



Datos estadísticos	Valores

Media	5191510,51
Mediana	4657041,33
Desviación estándar	1887798,37

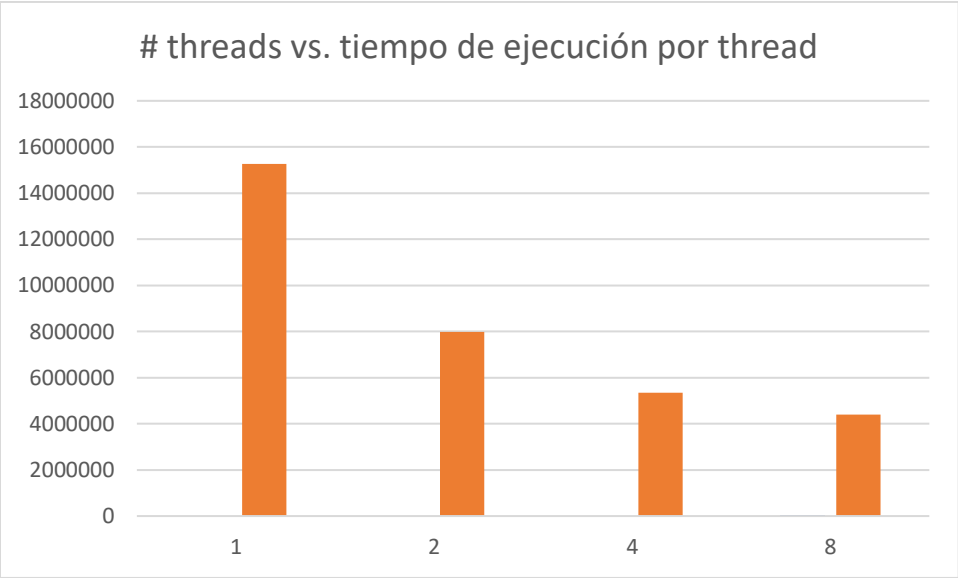
Unificado	**Porcentaje de uso del procesador de las 3 pruebas**			
Datos estadísticos	**1**	**2**	**4**	**8**
Media de los escenarios	31,1%	38,3%	65,2%	45,0%



Datos estadísticos	**Valores**
Media	45%
Mediana	42%
Desviación estándar	15%

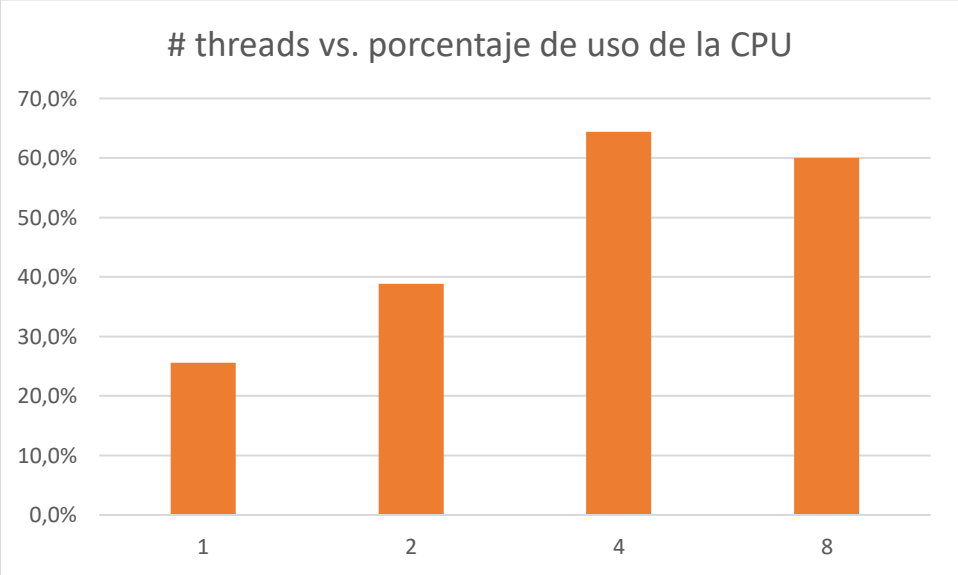
SHA-512:

Unificado	**Tiempo de ejecución Thread de las 3 pruebas**			
Datos estadísticos	**1**	**2**	**4**	**8**
Media de los escenarios	15267162,33	7982312,33	5356503,83	4392025,96



Datos estadísticos	Valores
Media	8249501,11
Mediana	6669408,08
Desviación estándar	4918282,45

Unificado	Porcentaje de uso del procesador de las 3 pruebas			
Datos estadísticos	**1**	**2**	**4**	**8**
Media de los escenarios	25,6%	38,8%	64,4%	60,0%



Datos estadísticos	Valores
Media	47%

Mediana	49%
Desviación estándar	18%

Como fue denotado anteriormente, en cada una de las gráficas se puede presenciar un pico óptimo de tiempo con un total de 4 threads en el tiempo de ejecución (esto se debe a nuestra implementación, ya que con 8 threads se genera una complejidad extra por el último thread). Este comportamiento es replicado en cada uno de los 3 algoritmos, pero tienen una variación importante, y es que, de manera global, los tiempos de ejecución del algoritmo MD5 son menores que los de SHA-256 y de manera equivalente, los de SHA-256, son menores a los de SHA-512. Esto es por la longitud de los códigos en cada uno de los algoritmos. También, el uso del procesador es mayor en un total de 4 threads para todos los algoritmos, aquí llega al pico del 65% de ejecución

Al analizar cada conjunto de datos, puede ser observado que tienen una media irregular, es decir, con un thread se demora mucho en ejecutar el requerimiento, mientras que cuatro threads lo hacen relativamente rápido. De manera subsecuente, la mediana tampoco es un indicador adecuado para un análisis. Por lo anterior, la desviación estándar es elevada e indica que los datos se encuentran demasiado dispersos con respecto a la media.

Por último, se resalta que los tiempos de ejecución empeoran entre los 4 y los 8 threads, y el uso del procesador no incrementa. Esto indica que el procesador ha llegado a su "tope" y no está maximizando todos sus recursos por lo que los 8 threads terminan haciendo un trabajo peor que los 4.

7 Conclusión

Al realizar este caso, obtuvimos los resultados esperados a partir de la teoría impartida a lo largo del curso. Se concluyó que a mayor número de threads se genera un menor tiempo de ejecución y un mayor uso del CPU, pero, que esto tiene un "tope", un máximo, por lo que no se pueden generar infinitos threads, dado que el procesador no los usará todos y por el contrario terminará gastando más recursos.

8 Bibliografía y referencias

- Universidad de los Andes. (2020). *Material del curso: Infraestructura computacional*. Disponible en SICUAPLUS para estudiantes de la Universidad.
- GeeksforGeeks. (2019, 21 febrero). *Print all possible strings of length k that can be formed from a set of n characters*. https://www.geeksforgeeks.org/print-all-combinations-of-givenlength/
- Profe Javier. (2017, 4 octubre). *Cómo usar Timers en Java*. YouTube. https://www.youtube.com/watch?v=6Uan34Whx3o&ab_channel=ProfeJavier

- StackOverflow. Foro utilizado para consulta frente a errores durante implementación, especialmente los siguientes links:
    - https://stackoverflow.com/questions/18489273/how-to-get-percentage-of-cpu-usage-of-os-from-java/21962037)
    - https://stackoverflow.com/questions/4044726/how-to-set-a-timer-in-java
    - https://stackoverflow.com/questions/1565388/increase-heap-size-in-java