Programación Concurrente
Segundo semestre 2020
Trabajo Práctico
Monitores

Autores

Nombres: Cristian Ariel Gonzalez
E-mail: cristian.gonzalez.unq@gmail.com
Número de legajo: 31420

Nombres: Luis Alejandro Mamani Jatabe
E-mail: alejandromamanijtabe@gmail.com
Número de legajo: 35327

Autores: Alejandro Mamani,
Cristian Gonzalez

Informe

Introducción

El dominio de nuestro trabajo corresponde al mundo de las criptomonedas. Nuestra tarea consiste en simular minería en la red *BITCOIN*, desarrollamos un programa concurrente para aprovechar el uso eficiente de los recursos de nuestras computadoras. Se divide la tarea en unidades de trabajo y las mismas son ejecutadas en concurrencia por un pool de **PowWorkers**.

Al usuario de nuestro programa se le solicita ingresar: una cantidad de threads para trabajar, una dificultad (que es la cantidad de los primeros n bytes que deben cumplir una condición) y una cadena de strings que puede ser vacía, esta última es transformada en bytes.

Ya con estos datos ingresados el objetivo de nuestro programa es el de encontrar un "nonce" dentro un rango de 2^{32} posibilidades que representan los 4 bytes del nonce buscado, la asignación equitativa a cada unidad de trabajo varía según la cantidad de threads que se hayan indicado.

Se instancia un **Buffer** que funciona como monitor.

Luego se instancia un **ThreadPool** que se encarga de crear la cantidad indicada por el usuario de **PowWorkers** (estos reciben el mismo buffer como parámetro) y luego se los inicia.

El programa se encarga de llenar el buffer con las unidades de trabajo, estas comprenden una tupla con la dificultad, la cadena y el rango antes mencionado.

La clase **Buffer** tiene un constructor con una capacidad parametrizable. Tiene dos métodos *synhronized* llamados "*write*" y "*read*". Tanto el **PowWorker** que utiliza el método read como el programa main que utiliza el método write toman el "lock" del monitor al momento de ejecutarlos.

Ambos métodos aseguran con un bucle (while) que estén dadas las condiciones para hacer uso del buffer. En caso negativo el thread se encola en la *variable de condición* esperando a ser despertado para volver a competir. En caso afirmativo y antes de finalizar la ejecución del método correspondiente despierta a todos los threads de la cola mediante el método notifyAll() (ya que al tener diferentes roles encolados en la variable de condición un simple notify() podría producir un deadlock).

El thread **PowWorker** en su método run() toma una unidad de trabajo del buffer setea las variables antes mencionadas y mediante un bucle (while) evalúa que el nonce todavía no haya sido hallado y que el rango no haya sido recorrido en su totalidad. En dicho caso emprende el proceso de evaluación descrito en el próximo punto.

Evaluación

Cada **PowWorker** obtiene de forma sincrónica una unidad de trabajo del buffer, entre otros datos, recibe: una cadena de bytes, un rango de búsqueda y una dificultad. Para la evaluación se toman valores individuales de dicho rango (comenzando por el mínimo e incrementándose hasta llegar a su máximo)

Dicho valor se transforma a cadena de bytes y se concatena de la siguiente forma (“cadena” + ”piezaIndividualDentroDelRangoAsignado”). A este nuevo valor se le aplica la función de hash SHA-256.

El siguiente paso es el de evaluar que los primeros n bytes sean 0 "cero" (según la dificultad informada). El resultado de esta función es un booleano que determina si el proceso continúa o si termina su ejecución y dá aviso al programa main para detener la ejecución del resto de los threads.

En este último caso también se informa el nonce encontrado y el tiempo transcurrido desde el inicio de la ejecución.

Existe otra condición booleana que determina la finalización de un proceso, la misma corresponde a si ya se han evaluado todos los valores del rango de búsqueda asignado al **PowWorker**, en dicho caso además de finalizar se informa esta situación al programa main.

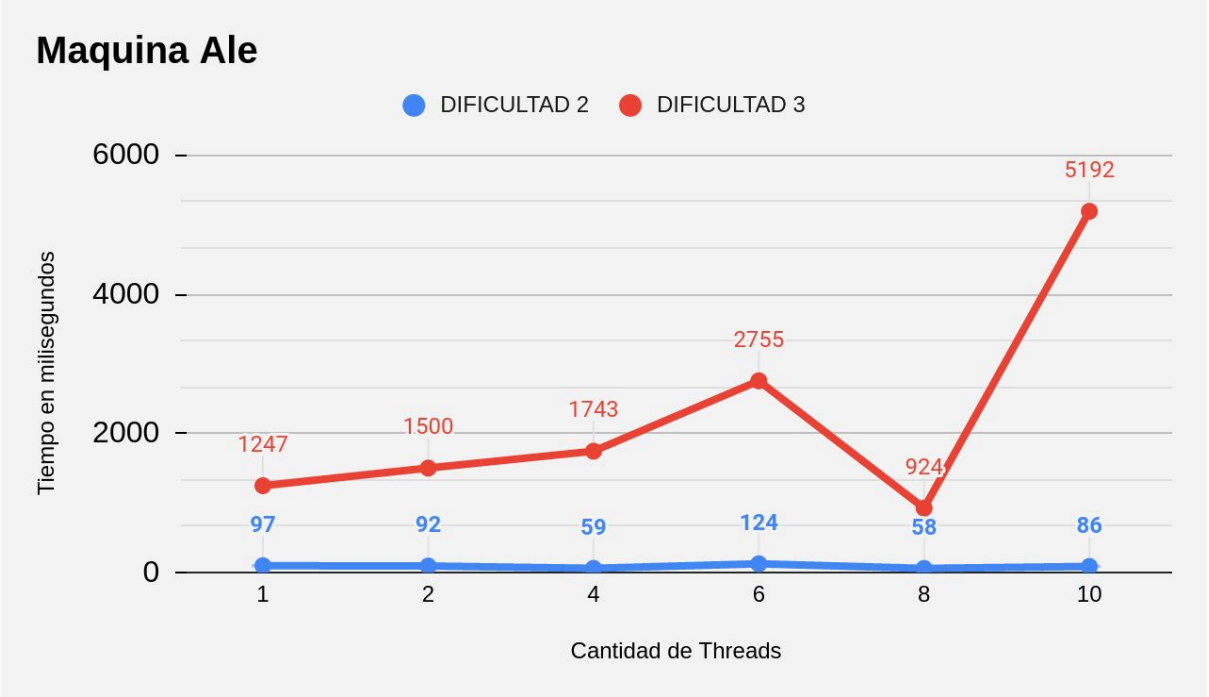
A continuación se detallan las características de los equipos donde se llevaron a cabo los sets experimentales:

Máquina	**Alejandro**	**Cristian**
Procesador	*Intel(R) Core(TM)i7-8750H CPU @2.20.GHz 2.21 GHz*	*AMD® A12-9720p radeon r7, 12 compute cores 4c+8g x4*
Memoria	*16 GB*	*7,3 GB*
Nombre del SO	*Windows 10 Home Single Language*	*Ubuntu 20.04.1 LTS*
Tipo de SO	*64 bits*	*64 bits*

En las siguientes páginas podrán observarse de manera independiente los resultados sobre cada una de las experiencias en las maquina que fueron corridas:

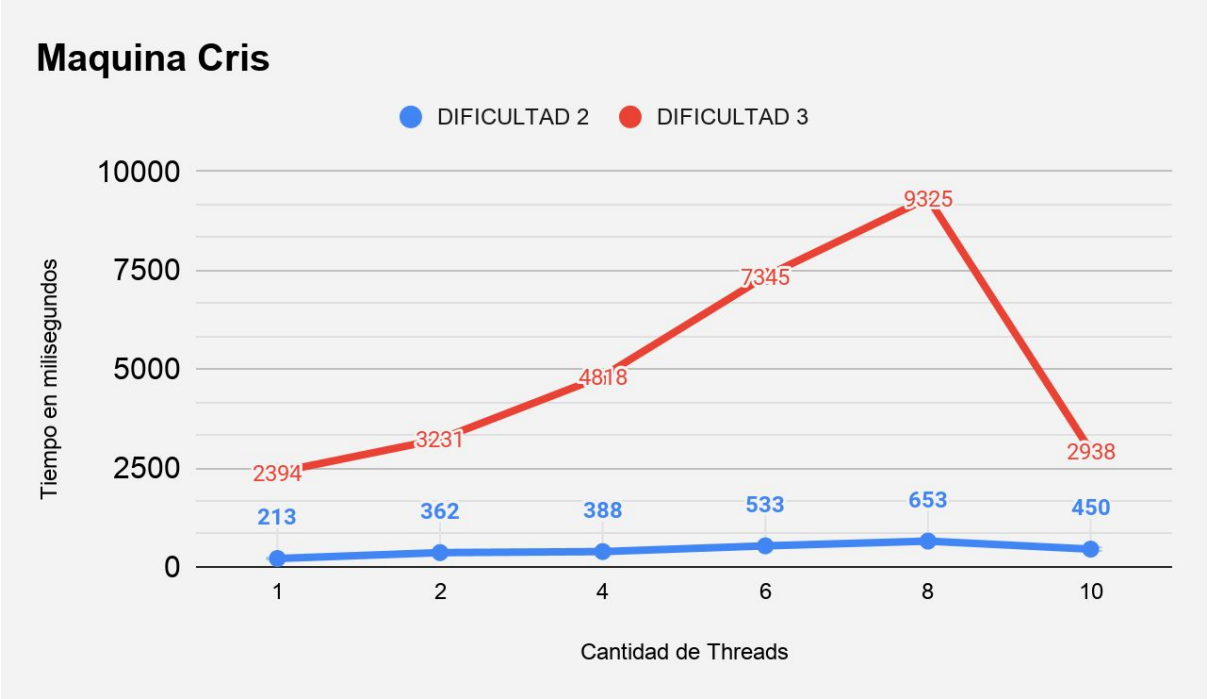
- Set de pruebas sobre la Máquina de Ale

Cantidad de Threads	Dificultad 2	Dificultad 3
	Tiempo (en milisegundos)	
1	97	1247
2	92	1500
4	59	1743
6	124	2755
8	58	924
10	86	5192



- Set de pruebas sobre la Maquina de Cris

Cantidad de Threads	Dificultad 2	Dificultad 3
	Tiempo (en milisegundos)	
1	*213*	*2394*
2	*362*	*3231*
4	*388*	*4818*
6	*533*	*7345*
8	*653*	*9325*
10	*450*	*2938*



Análisis

El objetivo de nuestro programa (encontrar un nonce dentro de un rango de posibilidades) es más efectivo cuando la tarea es dividida en varias unidades, porque cada thread puede realizar la búsqueda de forma concurrente al resto.

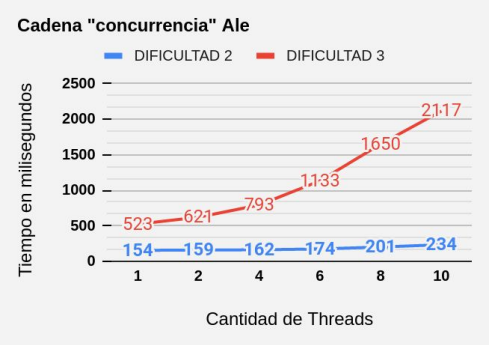
La cantidad de threads influyen en el orden en que se analizan los datos. Al distribuir el rango en partes equitativas cada thread evalúa cada valor del rango que se asigna hasta encontrar su objetivo o agotar las posibilidades de búsqueda.

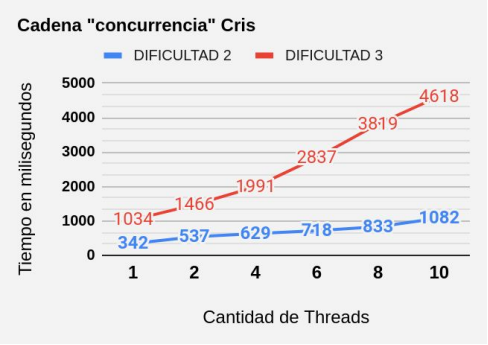
Obs:

Estimamos que al finalizar la ejecución de un thread, existe mayor capacidad del CPU para atender solicitudes del resto.

Para el caso hipotético que compitan 2^{32} threads el tiempo de acceso a CPU condiciona la eficiencia del programa.

El mismo set de pruebas, pero ingresando la cadena “concurrencia” en todos los casos nos arroja los siguientes resultado:





Verificamos que en comparación con los resultados en los cuales no se introdujo una cadena los tiempos con dificultad 2 son mayores, mientras que con dificultad 3 son notablemente menores.

Notamos que la tendencia alcista se mantiene uniforme cuando ingresamos una cadena. En el caso que no se ingrese nada puede producirse un efecto random que produzca el quiebre de la misma.

Luego verificamos con la búsqueda del "golden nonce" que independientemente de la performance del programa concurrente, con una tendencia a acompañar el rendimiento al hardware, existe un componente aleatorio que puede determinar que un nonce sea encontrado antes en el equipo con menos recursos computacionales.

- Set de pruebas “Buscando golden nonce” DIFICULTAD 4

DIFICULTAD 4		
Cantidad de Threads	Maquina Ale	Maquina Cris
	Tiempo (en milisegundos)	
10	*-*	*2025731*
8	*2570671*	*-*