



# Universidad Nacional Autónoma de México

FACULTAD DE INGENIERÍA  
DIVISIÓN DE CIENCIAS BÁSICAS

## PROYECTO PROGRAMACIÓN PARALELA: PROBLEMA DE LAS N REINAS

PROFESOR:  
Edgar Tista García

INTEGRANTES:  
Troncoso González Carlos Andrés  
Vences Santillán Carlos Eduardo  
Villagrán Segura Denisse Abril

Semestre 2023-1

8 de enero de 2023

# Índice

|   |           |
|---|-----------|
| <b>Introducción</b>                             | <b>2</b>  |
| <b>Objetivo</b>                                 | <b>3</b>  |
| <b>Antecedentes</b>                             | <b>4</b>  |
| <b>Descripción del algoritmo</b>                | <b>8</b>  |
| <b>Paralelización del algoritmo</b>             | <b>11</b> |
| Tipo de paralelismo . . . . .                   | 11        |
| Métricas de desempeño . . . . .                 | 12        |
| Formas de comunicación . . . . .                | 13        |
| Granularidad . . . . .                          | 14        |
| Balance de carga . . . . .                      | 14        |
| <b>Implementación del algortimo paralelo</b>    | <b>15</b> |
| <b>Pruebas realizadas</b>                       | <b>17</b> |
| Versión secuencial . . . . .                    | 17        |
| Versión concurrente . . . . .                   | 22        |
| Paralelizando solo la primera columna . . . . . | 22        |
| <b>Conclusiones</b>                             | <b>25</b> |
| <b>Autoevaluación general</b>                   | <b>28</b> |
| <b>Refrencias</b>                               | <b>28</b> |

## Introducción

La programación paralela es una gran herramienta cuando se busca realizar un algoritmo de manera más eficiente, de modo que en muchas ocasiones nos permite tener una mejor complejidad al ejecutar un algoritmo, aunque esto no siempre será así.

Que se presente una mejora en la complejidad de un algoritmo al pasar de una versión secuencial de este a una versión paralela, depende de muchos factores relacionados a la programación paralela y como es que la parallelización de un algoritmo se lleve a cabo. En este caso se trabajará sobre uno de los problemas matemáticos más estudiados en la programación, es cual es el problema de las  $n$  reinas. Este consiste en acomodar una cantidad  $n$  de reinas en un tablero de  $n$  filas por  $n$  columnas, sin que estas se toquen entre sí, es decir, que no se encuentren en posiciones contiguas, en una misma fila, en una misma columna o en una misma diagonal.

Lo que se realiza en primera instancia es realizar el código que dé una solución a este problema para un tablero de tamaño  $n$ , este tamaño será probado con diferentes números, teniendo en cuenta que el tamaño mínimo donde se puede resolver este problema es de cuatro. En cada una de las pruebas se medirá el tiempo que le toma resolver el problema.

Posteriormente se tomarán en cuenta todos los conceptos vistos en clase con respecto a la programación paralela para realizar ahora una implementación paralela, pero buscando que esta tenga un mejor rendimiento en cuanto a la complejidad.

Una vez se tengan ambos códigos se realiza un análisis de diferentes aspectos de código paralelo, como lo son que se realizó para hacer la versión paralela, el tipo de paralelismo, sus métricas, granularidad y el balance de carga. Además se realizarán gráficas del tiempo con respecto al tamaño para distintos tamaños de cada código, se modo que podamos observar la complejidad de cada algoritmo.

# **Objetivo**

## **Objetivo Proyecto**

Que el alumno ponga en práctica los conceptos de la programación paralela a través de la implementación de un algoritmo paralelo, así mismo desarrolle su capacidad para responder preguntas acerca de un concepto analizado a profundidad

## **Objetivo Personal del equipo**

El principal objetivo de este proyecto es realizar un algoritmo tanto en su versión secuencial como en su versión paralela. Esto con el objetivo de aumentar su rendimiento y hacerlo más eficiente, manteniendo al mismo tiempo la exactitud de los resultados que produce. Esta puede ser una tarea compleja y desafiante, pero puede valer la pena el esfuerzo, especialmente para los algoritmos que son críticos para el rendimiento de un programa o sistema.

Pero para realizar esta tarea debemos tener en cuenta el diseño de programas paralelos, partiendo de buscar sus puntos paralelizables, buscando la manera adecuada de realizar la partición del algoritmo, como es que se va a realizar las comunicaciones para coordinar las ejecuciones de las tareas y cómo es que estas nos darán el resultado esperado.

Para posteriormente analizar los diferentes aspectos del algoritmo paralelo obtenido, como son qué tipo de paralelismo se aplicó, analizar las métricas de desempeño, realizar gráficas probando con diferentes tamaños de entrada, qué formas de comunicación existen entre los procesos, la granularidad del algoritmo, el balance de carga, entre otros.

De modo que ya con el análisis realizado ver qué tanto aumentó el rendimiento en cuanto a la complejidad del algoritmo, y poder buscar un algoritmo que muestre una mejora en la complejidad, requiriendo menos tiempo para poder llegar a la solución del problema.

## Antecedentes

El problema de las  $n$  reinas es un problema clásico en informática y matemáticas. Se trata de colocar  $n$  reinas en un tablero de ajedrez de  $n \times n$  de manera que dos reinas no puedan atacarse entre sí. Una reina puede atacar cualquier casilla en la misma fila, columna o diagonal. El objetivo del problema es encontrar una ubicación de las reinas tal que no haya dos reinas que se ataquen entre sí, o determinar que no existe tal ubicación.

El primer algoritmo para resolverlo aparece en el libro "Solving Problems in Automata, Languages and Complexity" que fue publicado por Ian Miguel y Peter Nightingale en 1998. Los investigadores proponen un algoritmo basado en programación dinámica y una matriz de almacenamiento que asegura un tiempo  $O(N^4)$  para resolver el problema.

Los problemas NP son un tipo específico de problemas que se caracterizan porque todo aquel que pertenece a este grupo se puede resolver con algoritmos polinomiales. La versión clásica del problema consistía en colocar  $n$  reinas en un tablero de  $n \times n$ , de manera que las reinas no se ataquen mutuamente. Es decir, no pueden estar en la misma fila, columna ni diagonal.

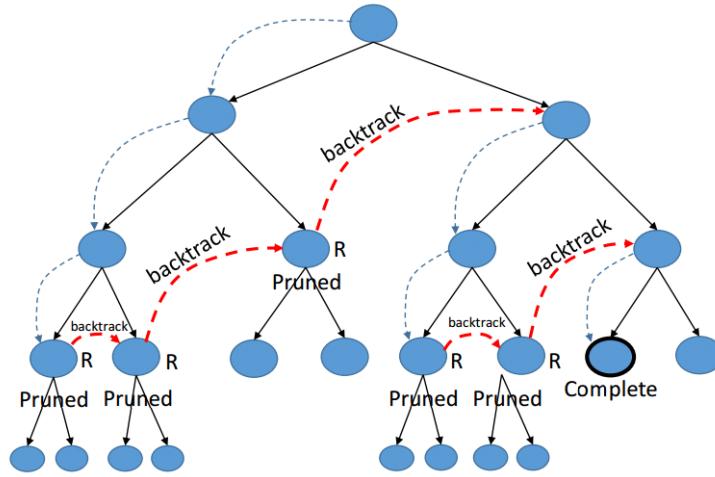
De manera computacional el problemas de las  $N$  reinas se puede resolver utilizando backtracking, o bien programación dinámica. En el caso de backtracking, intentamos colocar la primera reina en la primera columna, después en la segunda, hasta encontrar una posición que no amenace a las demás reinas, si llegamos a un momento donde no podemos colocar una reina pero ya no tenemos columnas disponibles para las siguientes reinas, entonces hemos fallado y regresamos una reina hacia atrás para intentar otra posición distinta. Esto se puede realizar con un algoritmo secuencial.

|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |
| 3 |   |   |   |   |   |   |
| 4 |   |   |   |   |   |   |
| 5 |   |   |   |   |   |   |
| 6 |   |   |   |   |   |   |

A 6x6 grid representing a chessboard for the N-Queens problem. Queens are placed at (1,4), (2,1), (3,5), (4,2), (5,6), and (6,3). Queens are represented by crowns with eight points.

Backtracking es un algoritmo general que funciona para muchos tipos de problemas. Se puede utilizar para resolver el problema de las  $n$  reinas en tiempo  $O(n!)$ , que se ha denominado uno de los problemas sin resolver más difíciles de la informática.

El algoritmo backtrack toma como entrada un conjunto de estados y reglas que describen cómo estos estados interactúan entre sí. El objetivo es hacer una secuencia de movimientos de modo que cada movimiento mejore su posición después de que se hayan ejecutado todos los movimientos anteriores. En otras palabras, desea encontrar una solución óptima independientemente de lo que se haya hecho antes o de lo que suceda a continuación.



Existen algunos algoritmos que buscan solucionar el tema de la complejidad del algoritmo, esto a través de la programación paralela, concurrente o distribuida. Dando un programa el cual llegue a una solución para un tablero de tamaño  $n \times n$  en menor tiempo que el programa que trabaja de manera secuencial.

En este caso se buscó realizar un algoritmo a través de la programación concurrente. La programación concurrente es una forma de programación en la que múltiples subprocesos o procesos pueden ejecutarse simultáneamente dentro de un solo sistema informático. Esto contrasta con la programación secuencial tradicional, en la que las instrucciones se ejecutan una a la vez en el orden en que están escritas.



La programación concurrente se utiliza a menudo para mejorar el rendimiento y la escalabilidad de los programas aprovechando los múltiples procesadores y núcleos que están disponibles en los sistemas informáticos modernos. También se puede usar para mejorar la capacidad de respuesta de los programas al permitirles realizar tareas en segundo plano mientras el programa principal continúa ejecutándose.

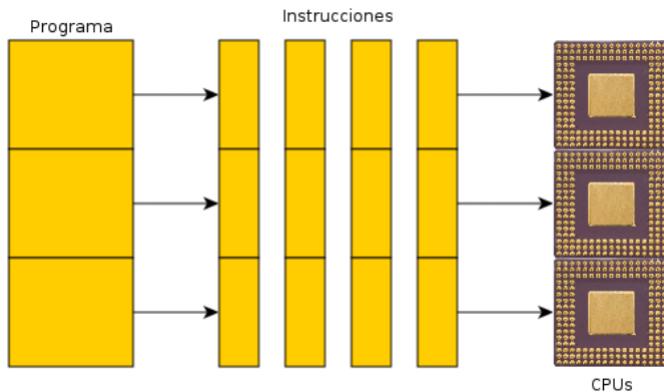
Hay muchas técnicas y enfoques diferentes para la programación concurrente, incluidos bloqueos, semáforos, monitores y paso de mensajes. Estas técnicas se utilizan para sincronizar el acceso a los recursos compartidos y para garantizar que se accede a los datos y se modifican de manera consistente y predecible.

En Java, la programación concurrente se admite mediante el uso de subprocessos. Un subprocesso es una ruta de ejecución separada dentro de un programa Java, y varios subprocessos pueden ejecutarse simultáneamente dentro del mismo proceso Java. Para crear un nuevo subprocesso en Java, puede definir una clase que amplíe la clase Subproceso y anule el método run(). Luego puede crear una instancia de esta clase y llamar al método start() para comenzar a ejecutar el hilo.

Java también proporciona una serie de otras funciones para admitir la programación concurrente, como el marco Executor, que le permite administrar un conjunto de subprocessos y ejecutar tareas al mismo tiempo, y el paquete java.util.concurrent, que incluye una serie de clases para realizar tareas de programación concurrentes comunes, como sincronizar el acceso a recursos compartidos y coordinar la ejecución de múltiples subprocessos.

Es importante tener en cuenta que la programación concurrente puede ser un desafío y es fácil introducir errores o condiciones de carrera en su programa si no tiene cuidado. Para evitar estos problemas, es importante utilizar técnicas de sincronización adecuadas y diseñar cuidadosamente su programa concurrente para minimizar la posibilidad de conflictos.

Pero también se podría crear un algoritmo que funcione a partir de la programación paralela. La programación paralela es una forma de programación informática en la que se utilizan varios subprocessos o procesos para ejecutar tareas al mismo tiempo, en lugar de secuencialmente. El objetivo de la programación paralela es aumentar el rendimiento y la capacidad de respuesta de un programa aprovechando múltiples núcleos de CPU o unidades de procesamiento. La programación paralela se puede usar para acelerar la ejecución de un programa al permitirle realizar múltiples tareas simultáneamente, en lugar de una a la vez.



Hay muchas formas de realizar la programación paralela, incluido el uso de subprocessos, procesos y sistemas distribuidos. Cada enfoque tiene sus propias ventajas y desventajas, y el mejor enfoque para un programa en particular dependerá de los requisitos y limitaciones específicos del problema.

Sin embargo, la programación en paralelo puede ser un desafío porque requiere que el programador considere detenidamente cuestiones como las condiciones de carrera, los interbloqueos y la sincronización. También puede ser más complejo escribir y depurar programas paralelos que programas secuenciales, porque el comportamiento de un programa paralelo puede depender del orden en que se ejecutan las tareas, lo que puede ser difícil de predecir.

La programación concurrente y la programación paralela son similares en el sentido de que ambas implican la ejecución de múltiples tareas simultáneamente. Sin embargo, hay una distinción importante entre los dos:

Programación concurrente: esto implica la ejecución de múltiples tareas simultáneamente dentro del contexto de un solo proceso. En la programación concurrente, las tareas pueden ejecutarse simultáneamente, pero comparten un espacio de memoria común y están sujetas a las mismas restricciones.

Programación paralela: esto implica la ejecución de múltiples tareas simultáneamente en múltiples procesos o núcleos de CPU. En la programación paralela, las tareas se ejecutan simultáneamente y tienen su propio espacio de memoria, lo que les permite ejecutarse independientemente unas de otras.

En resumen, la programación concurrente implica la ejecución de múltiples tareas al mismo tiempo dentro de un solo proceso, mientras que la programación paralela implica la ejecución de múltiples tareas al mismo tiempo en múltiples procesos o núcleos de CPU.

## Descripción del algoritmo

Antes de comenzar la paralelización del algoritmo debemos tener un algoritmo secuencial del cual partiremos para realizar un algoritmo paralelo.

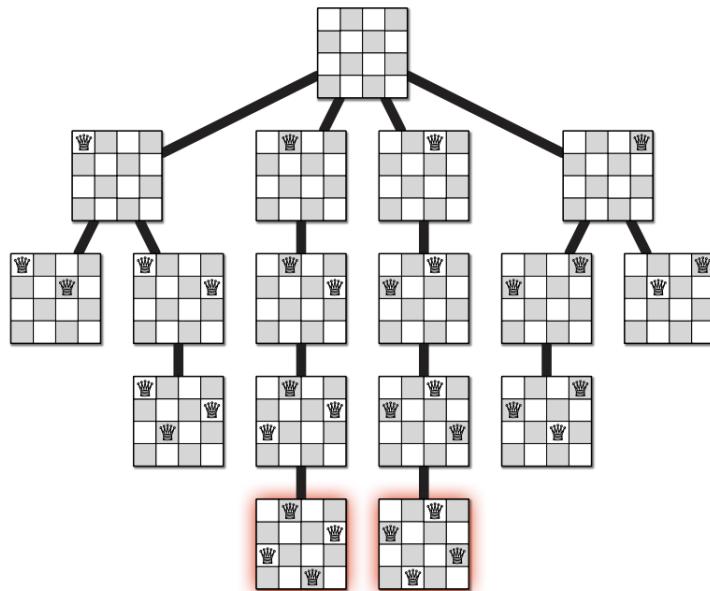
Hay que tomar en cuenta las reglas para la solución del problema de las n reinas, es decir, la manera en la que se acomoden las reinas debe cumplir con que estas no se encuentre en una casilla contigua a donde ya exista una reina, y no puede haber más de una reina en una misma columna, en una misma fila o en una misma diagonal.

Para poder ir colocando las reinas se realizará un algoritmo de backtracking. Un algoritmo de retroceso(backtracking) funciona construyendo una solución de forma incremental, una pieza a la vez, y abandonando una solución parcial tan pronto como determina que la solución no se puede completar.

Algunas consideración que existen en un algoritmo de retroceso son:

- Comience considerando la primera parte de la solución.
- Si no hay más piezas a considerar, el algoritmo termina.
- Si hay más piezas a considerar, el algoritmo intenta colocar la siguiente pieza en la posición actual.
- Si la pieza se puede colocar en la posición actual, el algoritmo pasa a la siguiente pieza.
- Si la pieza no se puede colocar en la posición actual, el algoritmo retrocede a la posición anterior y prueba una opción diferente.

Por ejemplo, considere el problema de encontrar un camino a través de un laberinto. El algoritmo comenzaría en la entrada del laberinto y trataría de moverse a través del laberinto paso a paso. Cada vez que llega a un callejón sin salida, retrocede hasta el último punto en el que tenía una opción y prueba una opción diferente. Este proceso continúa hasta que el algoritmo encuentra la salida o determina que no hay salida.



En analogía al ejemplo anterior trabajaremos colocando reinas en el tablero y cuando lleguemos a un punto en el que ya no se puedan colocar en un lugar las reinas faltantes nos regresamos al movimiento anterior y colocaremos la reina anterior en otra posición para posteriormente intentar colocar las reinas faltantes.

Si aún se realiza un solo retroceso varias veces y se llega a otro caso donde no se puede colocar las reinas se hará un segundo retroceso.

Al querer organizar n reinas en un tablero con n columnas donde ninguna de ellas se puede encontrar en la misma fila que otra, significa que cada una de las reinas debe colocarse en una columna distinta.

Entonces el algoritmo secuencial funcionara partiendo de colocar una primera reina en la posición (0,0), es decir, la primera columna y primera fila. y posteriormente buscará la primera fila de la siguiente columna donde podrá colocar la siguiente reina y así con todas. Si en un punto se realiza backtrack lo que hará será mover la reina a la que se regresó a la siguiente posición de fila por debajo.

A grandes rasgos es como funciona nuestro código, por lo que, en el algoritmo que se realizó en Java, se tiene tres clase, la primera es “Casillas” la cual nos servirá para que por medio de sus atributos podamos ver si la casilla se encuentra ocupada por una reina no está disponible según el valor que tenga, y su otro atributo nos indicará por cuantas reinas se encuentra bloqueada la casilla, esto último ya que a menos de que ninguna reina la esté bloqueando la casilla se encontrará bloqueada.

Otra clase es “Tablero”, esta contendrá como tal el tablero con su casillas, esto por medio de un arreglo bidimensional de Casillas que es una de sus atributos, el otro atributo es el tamaño del tablero. Esta clase posee los métodos que permiten agregar una reina bloqueando las casillas correspondientes tras agregarla, además de quitar una reina desbloqueando las casillas que corresponden.

En el método que permite agregar una reina se reciben como parámetros la fila donde será agregada y la columna, para primero revisar si la casilla indica por la fila y la columna se encuentra bloqueada por alguna reina, esto hace revisando si ArrayList está vacío, en caso de que este vacío se coloca la reina, en caso contrario no se coloca. Según se haya podido colocar o no de devuelve un valor booleano indicandolo.

En caso de que si se haya colocado la reina se bloquearán todas las casillas que corresponde a las casillas por debajo en la misma columna, por delante en la misma fila, en los lados derechos de las diagonales descendentes y ascendentes. Esto por medio de ciclos for y ciclo while para las diagonales, donde cada casilla que se va a bloquear se le agrega a su ArrayList el número de la reina que la bloqueo.

El otro método mencionado es para quitar una reina, lo que es necesario para el backtracking o retroceso, se reciben como parámetros la fila y columna donde se quitara a la reina, y se verifica si en esa casilla hay una reina, esto se sabe con que el atributo de la casilla tenga como valor un uno, de lo contrario no existe una reina en esa casillas.

En caso de que si haya una reina se desmarca la casilla como ocupada cambiando el valor a menos unos, luego se comenzará a quitar a la reina del ArrayList de las casillas casillas por debajo en la misma columna, por delante en la misma fila, en los lados derechos de las diagonales descendentes y ascendentes, accediendo a estas de la misma manera que se realizó en el método anterior; pero con la diferencia que en esta ocasión en lugar de agregar la reina al ArrayList la vamos a quitar.

Es importante indicar que el número de de reina está marcado por la columna donde se pretende agregar o dónde está agregada.

Por último se tiene la clase “BackTrack”, la cual es la que contiene el método main además de un método que es el que lleva a cabo el backtrack para dar solución, el cual es el método probarColumna(), en el se recibe como parámetros la columna donde se va a posicionar a la reina. Este método funciona de manera iterativa al mismo tiempo que realiza procesos recursivos; retorna un valor boolean, en caso de que la columna esté llena retorna true.

Se tiene una variable que nos indicará si la columna se encuentra llena, la cual comienza con el valor true, y otra variable que nos indica si alguna posición en la columna se encuentra indirectamente bloqueada, es decir, bloqueada por la reina en la columna anterior. Durante cada movimiento que se realice en las columnas al posicionar o quitar alguna reina se mostrará en la pantalla.

Se comienza un ciclo for que irá desde cero hasta n, de modo que nos permitirá recorrer todas las filas en la columna. En cada iteración se intentara agregar la reina en cada fila, de modo que hasta que se logre agregar la reina se pasará realizar las instrucciones dentro de un if, al lograr agregar una reina se indica que la columna no está llena así que se modifica el valor de la variable a false.

Aun dentro del if se comienza el proceso recursivo, donde si la columna donde nos encontramos no sea la última la variable bloqueo indirecto adquirirá el valor boolean que retorno de la llamada recursiva del método probarColumna() donde se realiza con la siguiente columna del tablero.

En caso de que esta variable tenga como valor false, significa que no hubo ninguna casilla disponible donde colocar la siguiente reina en la siguiente columna o alguna reinas posteriores, por lo que ya se logró seguir ese camino así que se debe realizar backtrack.

Para realizar el backtrack se quita la reina que se acaba de colocar, ya que en esa posición no existe ninguna solución posible para el problema, y se marca la variable que indica si la columna está llena como true nuevamente, debido a que como se quito esa reina aún la posición estaria bloqueda indirectamente ya que bloquea las posibles soluciones para el resto de reinas faltantes.

Al terminar las iteraciones se retorna el valor que nos indica si la columna está llena, ya que con este sabremos si se logró colocar una reina o no en esa columna. De este modo se llegará a un resultado por medio de backtrack, el cual debido a cómo está estructurado el algoritmo siempre será el mismo para un mismo tamaño, ya que las posibilidades se van probando de manera secuencial siguiendo un mismo orden en las instrucciones.

Por último para poder hacer uso de este algoritmo y medir el tiempo en el main() se da valor al tamaño del tablero y se crea un objeto de la clase “BackTrack”, mandando como argumento el tamaño. Posteriormente se registra el tiempo actual de la ejecución, se manda a llamar al método probarColumna() comenzando desde la columna cero(la primera) y una vez se termine su ejecución ya que se halló una solución se vuelve a registrar el tiempo de ejecución, restando el tiempo final menos el tiempo inicial sabemos cuánto tiempo tardó en hallar una solución, este tiempo está dado en milisegundos.

## Paralelización del algoritmo

Para implementar la paralelización en este algoritmo tipo BackTrack, se nos ocurrieron varias hipótesis sobre cómo podríamos implementar la programación paralela.

Tomando en cuenta que ya sabemos cómo funciona el algoritmo secuencial, nos dimos cuenta de que la forma más lógica de optimizar todas las iteraciones era por medio de dividir los hilos del procesador. Como se revisó anteriormente, para que el algoritmo BackTrack funcione se requiere saber en qué casilla de la siguiente columna se puede agregar una nueva reina; en caso de no haber ninguna posición disponible, entonces se quita la última reina y se vuelve a intentar en otra casilla. Esto quiere decir que cada que se encuentre un “mal camino en el laberinto” va a tener que regresar a probar por uno nuevo.

La primera hipótesis que creamos fue la de dividir los hilos cada que se cambie de columna, esto quiere decir que para cada una de las posibilidades, se genere un nuevo hilo que siga el proceso de forma paralela. Llamemos a este algoritmo el “100 paralelo”.

Al notar algunas desventajas muy notorias en esta primera implementación (se explica más adelante), la segunda que realizamos fue una especie de paralelización híbrida en la cual la generación de hilos son “limitados” por el programador para que en tableros de tamaño n grandes no sufran problemas de complejidad. Llamemos a este algoritmo el “Híbrido limitado”.

Después de seguir haciendo pruebas, descubrimos una última forma que realmente funcionaba para cualquier tamaño n de tablero. Esta forma de programarlo es todavía más concurrente que la versión anterior, pero a su vez es la que mejor paraleliza nuestra versión secuencial de BrackTrack original (esto se comenta más a detalle en la implementación). En pocas palabras lo que hace es simplemente dividir los hilos en n partes distintas, después de realizar este proceso, cada hilo trabaja con el algoritmo secuencial hasta que uno de estos halla una solución. Llamaremos a este algoritmo el “Concurrente”.

Para estas 3 versiones analizaremos un poco de sus características para ver cuál es el teóricamente nos convendría al momento de pasarlo a código.

### Tipo de paralelismo

Versión “100 paralelo”: Pensábamos que el tipo de paralelismo en esta versión podía ser algorítmico, ya que ésta reparte los hilos cada que se encuentre con una nueva columna llena de diferentes posibilidades (siendo cada hilo independiente). Esto quiere decir que la manera de repartir el trabajo depende puramente del algoritmo BackTrack que programamos para la versión secuencial, solo que en vez de regresar y probar una nueva opción, se divide para ver todas las posibles soluciones a la vez.

Sin embargo, una idea que nos puso a pensar, es que el tipo de paralelismo Farm trabaja de forma muy similar a la estructura de ramas que se genera al dividirse. Podemos decir que el hilo principal que genera java es el Manager mientras que los demás hilos que se van invocando son Workers, el problema de afirmar esto es que cada Worker al final se volvería un Manager de otro Worker, por lo que al final la “jerarquía” que establece este concepto no se cumpliría como tal.

Versión “Híbrido limitado”: En este algoritmo creemos que también se usa el tipo Farm/Manager-Workers, ya que se hará el mismo procedimiento que en el “100 paralelo” hasta que se acaben los hilos disponibles (recordemos que en esta versión el programador elige cuál es el límite de hilos que se pueden

usar al mismo tiempo). Cuando se terminen los hilos, empezará a ejecutarse la versión secuencial para cada uno de éstos, por lo que a partir de ese punto podríamos decir que se vuelve algorítmico, ya que se dejan de dividir las tareas y cada uno va buscando la mejor solución de manera independiente.

**Versión “Concurrente”:** Siendo nuestro mejor algoritmo, podemos decir que éste trabaja con paralelismo geométrico, ya que (como vamos a ver en el siguiente punto) el algoritmo funciona dividiendo cada fila del tablero entre los hilos del procesador. Podemos verlo como si estuviéramos buscando un muñeco en la rosca de reyes; esta implementación lo que hace es dividir la rosca en  $n$  cantidades para que cada parte se encargue de encontrar al muñeco. Con esta analogía podemos decir que la instrucción de “encontrar al muñeco” en realidad es hacer BackTrack hasta encontrar el resultado, por lo que la paralelización en realidad solamente se da al inicio al ingresar el tamaño del tablero (“dividir el pastel”). Así entonces, cada hilo se encarga de encontrar una solución paralelamente de forma secuencial.

### Métricas de desempeño

Para realizar las métricas de desempeño posibles, vamos a utilizar el algoritmo que realmente nos funcionó (el que llamamos “Concurrente”), ya que estos criterios fueron los que nos permitieron llegar a la conclusión de que realmente esta solución era la mejor para parallelizar el problema de las n-reinas.

**Tiempos de procesamiento:** Como veremos más adelante en las pruebas realizadas, el tiempo de ejecución de este algoritmo es realmente mucho más veloz a su versión secuencial, llegando incluso a ser proporcional a la cantidad de hilos en los que se divide: esto quiere decir que si un tablero de 5x5 tarda 25 milisegundos en ejecutarse, esta versión paralela solo tardará 5 milisegundos, debido a la forma en la que repartieron las tareas con los hilos (un hilo para cada fila).

**Speedup:** Para realizar un ejemplo de speedup con este algoritmo lo que haremos será tomar algunos de los datos recaudados en las pruebas para poner a prueba esta relación. Ésta en esencia es el tiempo de ejecución de un programa ejecutándose en un solo procesador sobre el tiempo de ejecución ejecutándose en  $n$  procesadores. Lo que quiere decir: Speedup = Tiempo con un procesador / Tiempo con  $n$  procesador.

Entonces como ejemplo pongamos uno de los datos.

$n = 9$  en versión secuencial: 0.118 segundos

$n = 9$  en versión paralela: 0.053 segundos

Tomando este ejemplo, tenemos que el SpeedUp de esta muestra es de: 2.2264

**Eficiencia:** La eficiencia se define como que  $n$  procesadores deben hacer el trabajo en una fracción  $1/n$  del tiempo que le lleva a un solo procesador. Por lo que la fórmula para calcular esto sería:  $S(n) / n$ .

Tomando en cuenta nuestro ejemplo, entonces la eficiencia sería igual a:  $2.2264/9 = 0.2473$

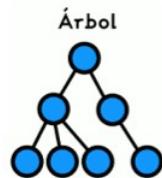
Fracción serial: Este concepto relaciona el speedup y la eficiencia. Se obtiene dividiendo el inverso de speedup menos  $1/n$  y todo eso dividido entre  $1 - 1/n$ . Viéndose la fórmula igual a  $((1/S) - (1/n)) / (1 - (1/n))$ .

Siguiendo nuestro ejemplo de  $n=9$  tendríamos lo siguiente:  $((1/2.2264) - (1/9)) / (1 - (1/9))$ ; lo que es igual a 0.3803

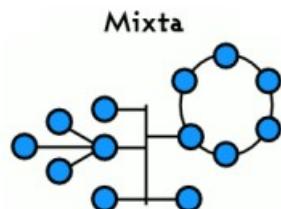
### Formas de comunicación

Podemos decir que la forma de comunicación de estos algoritmos es parecida, ya que todos trabajan con memoria distribuída. Esto quiere decir que cada procesador utiliza su propia memoria privada comunicándose con los demás mediante una red basada en una “topología”.

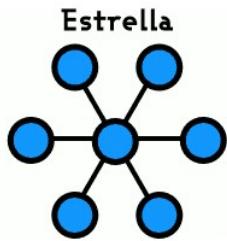
Versión “100 paralelo”: La topología de esta red podemos decir que es de tipo árbol, ya que la información de cada hilo va dependiendo de su “padre”, cada una de estas ramas es una ruta que está siguiendo, por lo que se podría llegar a representar como un grafo de tipo árbol.



Versión “Híbrido limitado”: Para explicar esta forma de comunicación podríamos decir que tiene una topología mixta, ya que al inicio se comunican de la misma forma que el 100 paralelo, por lo que tendríamos inicialmente un árbol, pero cuando se acaban los hilos este pasa a ser un tipo estrella, ya que los procesos no tienen que seguirse dividiendo, y simplemente dependen de un solo parente (para cada uno).



Versión “Concurrente”: La comunicación del mejor algoritmo es muy fácil; es una topología tipo estrella (nuestro algoritmo estrella). Esto lo podemos deducir fácilmente ya que la memoria de cada hilo solo depende del hilo principal, como si cada parte de la rosca de reyes solo necesitará la información de la rosca original. Aquí absolutamente todos los hilos que salen del principal son independientes, y no necesitan comunicación con nadie más que con el único parente.



### Granularidad

La granularidad de la propuesta de nuestros algoritmos fue algo que también tomamos en cuenta al momento de escoger el mejor. Tal como vimos en clase, la granularidad supone un papel importante al momento de querer parallelizar un algoritmo, ya que si es muy fina puede presentar graves problemas de complejidad que simplemente empeoran el problema principal, y si es muy gruesa, la parallelización puede que no se complete de manera adecuada.

**Versión “100 paralelo”:** Al ser un algoritmo que va creciendo con llamadas de nuevos hilos exponencialmente, podemos decir que la granularidad es demasiado fina. La comunicación entre los procesadores va aumentando cada vez más y más, por lo que en un tablero medianamente grande, ya tendríamos problemas al ejecutar tantas instrucciones entre hilos.

**Versión “Híbrido limitado”:** Para esta versión ocurre algo muy especial; al poder nosotros controlar la cantidad de hilos que se están ocupando durante la ejecución del programa, podemos “ajustar” la granularidad que deseemos, ya que la versión secuencial del algoritmo es la que se ejecuta inmediatamente después de que se acaben los hilos y esta trae consigo la granularidad “gruesa”.

**Versión “Concurrente”:** En este algoritmo se puede suponer que tiene granularidad medianamente gruesa pero no tanto como para no ser considerado paralelo. Podemos ver que la única vez que se comunican entre sí los hilos es ejecutar el programa y después de ahí todo sigue con la versión secuencial. Esta implementación resultó óptima, ya que al usar el mínimo número de división de procesadores conseguimos una granularidad que funciona muy bien comparándola con todas las demás, incluida la versión secuencial.

### Balance de carga

El balance de carga se refiere a la manera en la cual nuestro algoritmo reparte el total de tareas a realizar entre los procesadores disponibles.

**Versión “100 paralelo” e “Híbrido limitado”:** Estas dos implementaciones del algoritmo trabajan con balance de carga dinámico, podemos afirmar esto debido a la forma en la que están programados los hilos. Cada vez que uno de estos toma una ruta equivocada para encontrar la solución, el hilo muere y se invoca otro para otra ruta diferente en caso de no haber acabado. Para el algoritmo híbrido hay una peculiaridad, ya que el balance de carga es dinámico solo hasta que se acaban los hilos disponibles.

**Versión “Concurrente”:** Para la versión concurrente se usa balance de carga estático, ya que aquí los hilos no se vuelven a crear, simplemente se generan al hacer la partición del tablero en filas. Podemos decir que las tareas son definidas previo a la ejecución del algoritmo, ya que como comentamos anteriormente, el mero funcionamiento de esta versión es igual al secuencial, la única diferencia es que repartimos las tareas desde el comienzo.

## Implementación del algoritmo paralelo (cocurrente)

Para implementar nuestros primeros algoritmos paralelos de prueba en Java, vamos a usar la biblioteca Thread, esta va a ser la base de todo el paralelismo que le apliquemos al algoritmo secuencial. Para hacer esto vamos a agregar un paquete a nuestro proyecto llamado “BackTrackParalelo” en el cual agregaremos nuevos métodos y funciones al algoritmo original que explicamos al inicio del documento.

Este nuevo paquete va a ser la cuna del manejo paralelo. Lo primero que hicimos fue crear la clase que contendrá como tal el manejo de los hilos, llamada “HiloCasilla”; ésta heredará todos los métodos que nos proporciona Thread para hacer uso libre de los hilos del procesador (que en realidad viene de la máquina virtual de Netbeans). Para este punto es importante resaltar que se modifica el método run de los hilos, para que cada vez que se inicialice uno, se genere un objeto tipo “Tablero” que a su vez contiene otro atributo suyo llamado “casillas”. Esto quiere decir que cada que se genere un hilo nuevo, tendrá su propio tablero el cual contendrá modificaciones PARALELAS a todos los demás al momento de ejecutarse.

Como atributos, esta clase contendrá parte de la información que la clase “BackTrack”, los cuales son atributos como: tablero, fila, columna, y un booleano que nos indicará si la reina se puede poner en la posición a elegir.

Entonces, al igual que el algoritmo original, este método se ejecutará recurrentemente para así saber si la siguiente columna tiene espacio para una reina más que no choque con las casillas ya “marcadas”. Hasta este momento todo funciona al igual que la versión secuencial, la parallelización viene cuando en vez de regresar al BackTrack cada que se encuentra con un camino sin salida, el programa indica que se genere un nuevo hilo con un tablero que sea igual al “original” cuando vea más de un posible camino. Esto quiere decir que cada que encuentre múltiples espacios disponibles en la siguiente columna se genera un hilo para cada posible camino, copiando así la información del tablero previa a su creación.

**Características y desventajas de versión “100 paralela”:** Como estuvimos analizando a lo largo del proyecto, esta primera implementación consistía en parallelizar por completo el algoritmo que inicialmente era BackTrack. Para hacer eso, se modificó el código para que cada vez que se tuvieran múltiples caminos a evaluar, se crearán nuevos hilos y se dividieran la tarea de buscar la solución.

Teóricamente suena una buena idea, ya que al tener más hilos “buscando” el camino correcto, se llegaría a la solución más rápidamente. El verdadero y grotesco problema yace en que cada vez que se genera un hilo, se genera un Tablero el cual tiene que copiar la información del punto en el que se va a dividir, en ese caso, si se tiene un tablero de 8 x 8, hay muchas más posibilidades de que haya un mayor número de casos disponibles, de los cuales solo uno va a ser el que contenga la solución. Viéndolo desde la complejidad computacional, sería generar un montón de tableros al mismo tiempo que estén buscando el camino paralelamente, lo que supone una cantidad ridícula de memoria ocupada por cada tablero; sin hablar del procesador y de los hilos que pueda manejar a la vez. Sabiendo esto, la posibilidad de crear un algoritmo completamente paralelo quedó descartada.

**Características y desventajas de versión “Híbrida limitada”:** Para esta versión intentamos arreglar el problema del algoritmo anterior, lo primero que se nos vino a la cabeza fue controlar la cantidad de veces que se puede parallelizar el problema. Después de investigar en la api de Java, encontramos un método llamado “activeCount()” el cual devuelve la cantidad de hilos activos en ese momento. Gracias a esta útil herramienta, agregamos una condición con la siguiente instrucción “if(activeCount();30);”;

esta instrucción básicamente nos dice que ejecute todo el código paralelo (la generación de hilos) solo si la cantidad de hilos activos no pasa de 30. En caso de que ya se hayan completado todos los permitidos, entonces el algoritmo vuelve a usar la forma secuencial para terminar así de buscar la solución en uno de sus tantos hilos repartidos.

Una de las posibles ventajas de esta implementación es que podemos asignar la cantidad de hilos que queramos limitar, de esta forma la complejidad no se vuelve tan horriblemente factorial como en el algoritmo 100 paralelo. La principal desventaja de esta versión es que nunca vamos a estar seguros del número que se ingrese como tamaño del tablero, por lo que puede llegar a crear problemas si no se sabe la cantidad de hilos con la que trabaja nuestro procesador. Esto en consecuencia nos trae problemas de complejidad al tener muchos hilos generando tableros al mismo tiempo.

Características de versión “Concurrente”: Como ya mencionamos varias veces, este fue el algoritmo final que implementamos como “paralelización” de las n reinas.

El motivo es el siguiente: después de hacer las pruebas con las primeras dos propuestas de algoritmos, llegamos a la hipótesis de que entre menos paralelismo haya para cada columna, menos complejo va a ser. Al inicio dudamos de si entonces el algoritmo no se podía parallelizar, pero después de analizarlo llegamos a la conclusión que nos daría el resultado deseado.

La forma más básica de poder parallelizar el algoritmo BackTrack que generamos, sería simplemente hacer que un hilo revise cada fila para saber en cuál de éstas el camino es más fácil. Con esta simple lógica nos pusimos manos a la obra y pudimos implementar una instrucción que, con ayuda de la variable n (el tamaño del tablero) genere la cantidad de hilos igual al número de filas; por lo que en un tablero 4x4 se generarían 4 hilos y cada uno de estos trabajaría con la versión secuencial del BackTrack.

El programa anteriormente se enfocaba en probar todas las combinaciones para la fila uno y luego regresar a hacer la fila dos y de esa manera esperar hasta encontrar la fila que tuviera la solución. En esta versión paralela, los hilos se dividen en n cantidades y cada uno genera su propio tablero en el cual hará las pruebas (usando la versión secuencial) hasta que el primero devuelva un tablero con la solución. Gracias a esto podemos ver que el tiempo de ejecución mejora drásticamente.

Siguiendo con la analogía de la rosca de reyes, podemos decir que la versión secuencial se trata de que una sola persona esté buscando a lo largo de toda la rosca al muñeco. En cambio, la versión “Concurrente” divide la rosca en pedazos iguales y se las da a n personas para que busquen al muñeco; entonces el primero que lo encuentre le avisa al “repartidor” (que en este caso es el hilo principal del main) y finalmente se imprime esa versión.

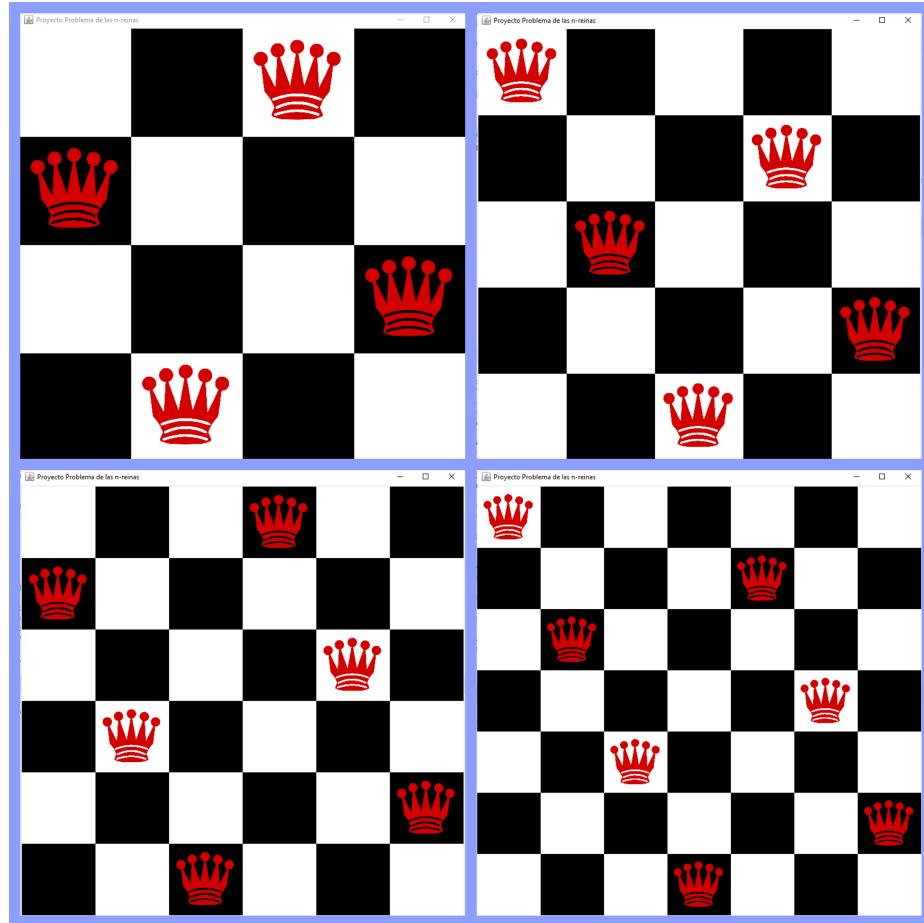
Dentro de este paquete también tenemos una clase la cual se llama “BackTrackParalelo”, está lo único que hace es buscar el hilo ganador proveniente de la clase “HiloCasilla”. Del mismo caso esta clase se encarga de mostrarnos en pantalla el tablero del hilo con la solución correcta. Una cosa a resaltar sobre esto, es que en la versión secuencial se nos va mostrando el proceso del backtracking en pantalla hasta que llega a una solución final, en cambio, en esta versión simplemente se va a imprimir el tablero que contenga la solución final. Gracias a esto podemos decir que el hilo que comenzó con la reina en la posición de la fila x es el ganador.

## Pruebas realizadas

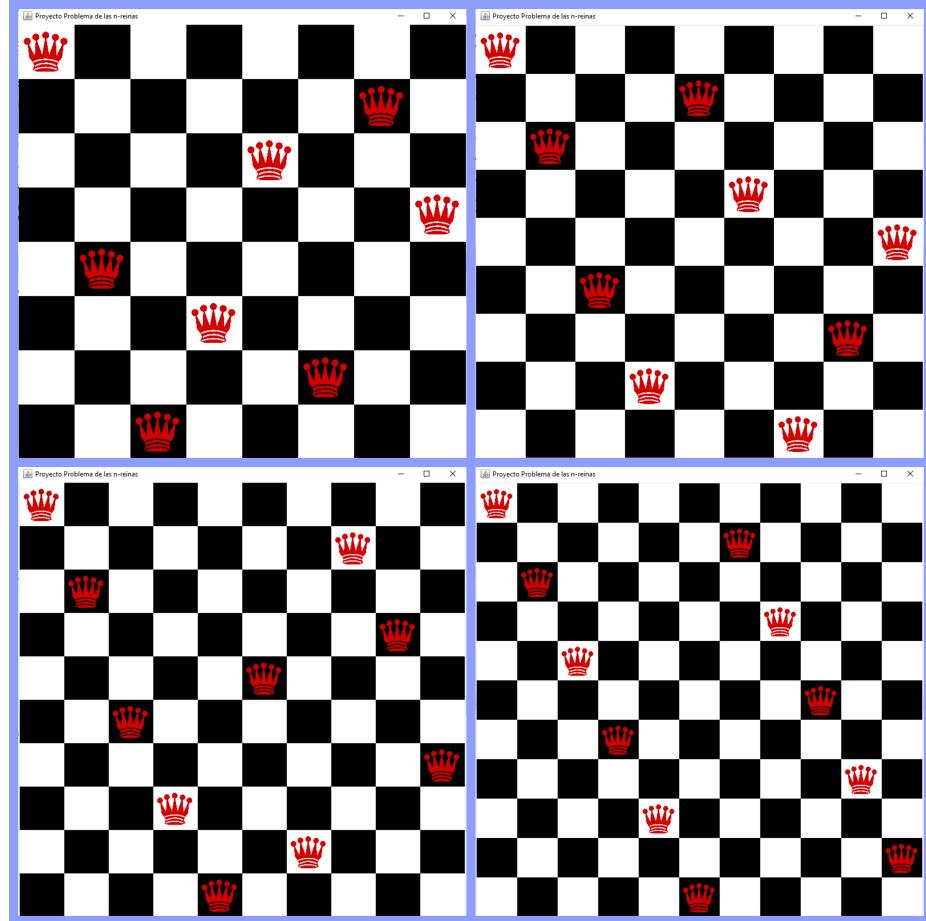
### Versión secuencial

Pruebas para tableros de distintos tamaños:

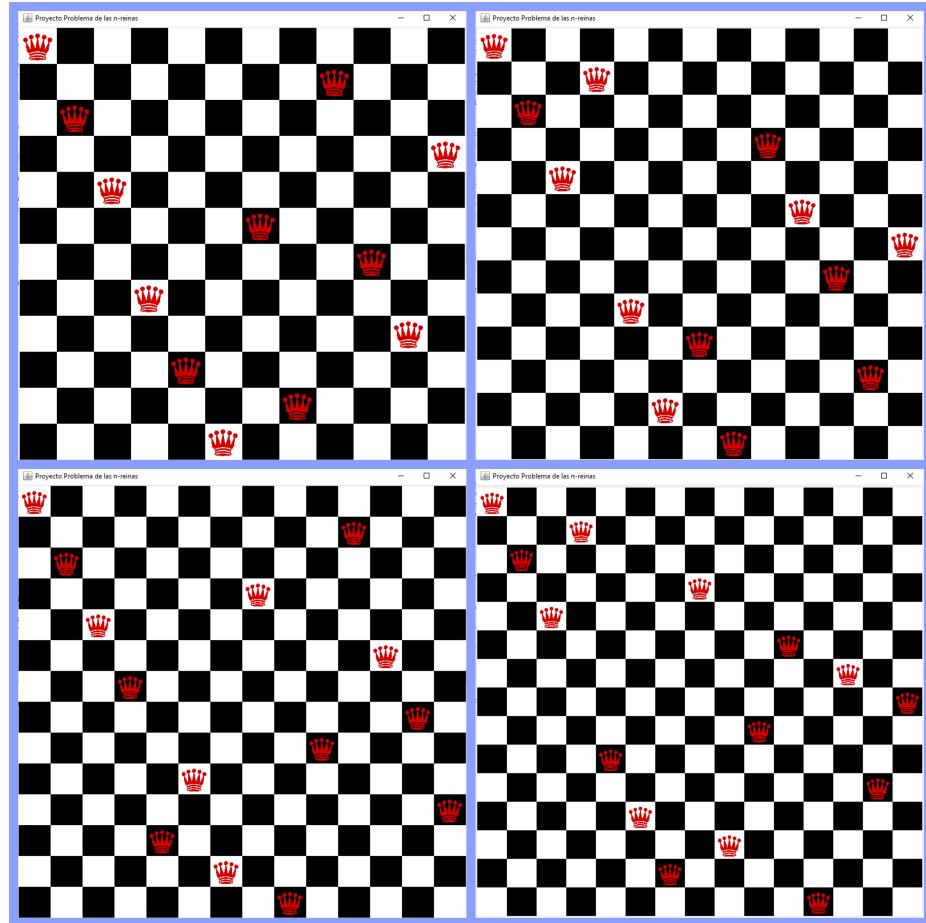
- Tamaño de tablero 4x4 a 7x7.



- Tamaño de tablero 8x8 a 11x11.



- Tamaño de tablero 12x12 a 15x15.



- Tamaño de tablero 16x16 a 19x19.

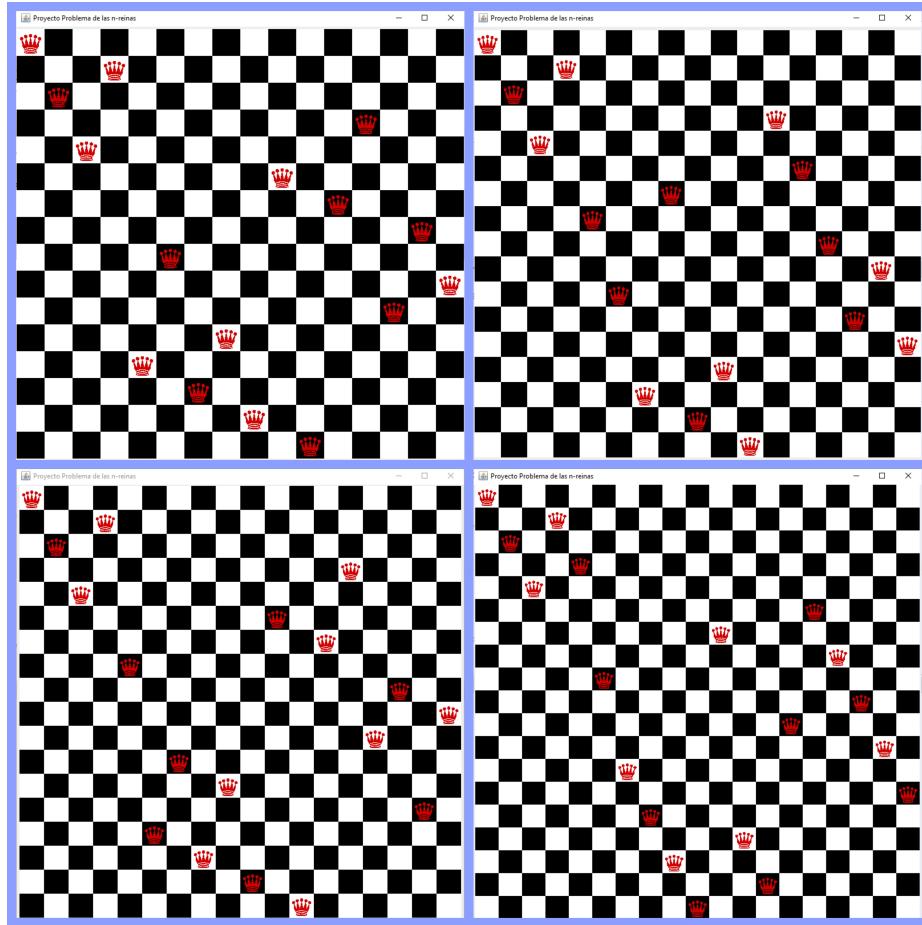
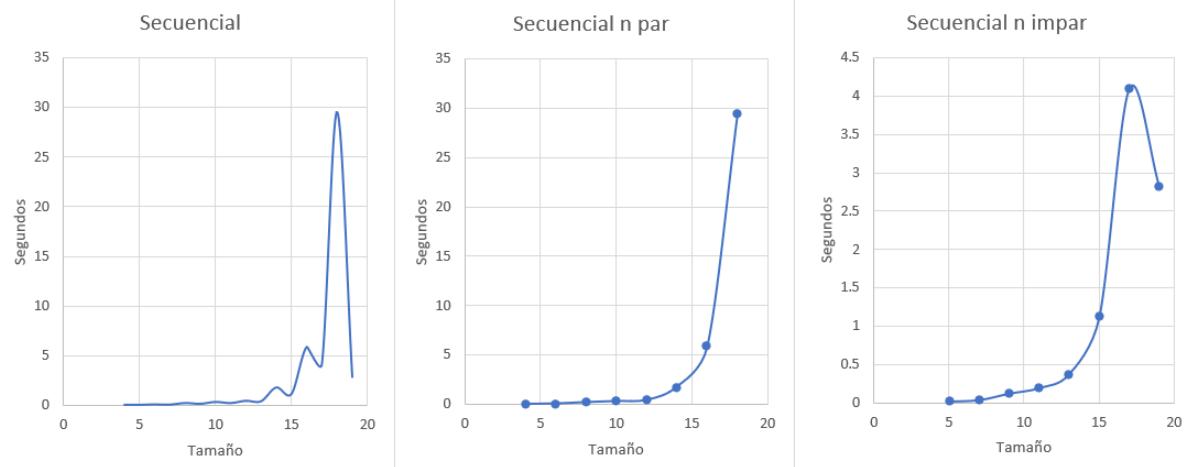


Tabla del tiempo en segundos con varias pruebas para cada tamaño, obteniendo un promedio:

| Tamaño | Segundos |        |        |        |        | Promedio |
|--------|----------|--------|--------|--------|--------|----------|
|        | 1er      | 2do    | 3er    | 4to    | 5to    |          |
| 4      | 0.024    | 0.006  | 0.007  | 0.005  | 0.06   | 0.0204   |
| 5      | 0.006    | 0.026  | 0.007  | 0.006  | 0.033  | 0.0156   |
| 6      | 0.021    | 0.017  | 0.12   | 0.111  | 0.023  | 0.0584   |
| 7      | 0.042    | 0.017  | 0.036  | 0.059  | 0.015  | 0.0338   |
| 8      | 0.307    | 0.109  | 0.116  | 0.247  | 0.2    | 0.1958   |
| 9      | 0.043    | 0.035  | 0.037  | 0.185  | 0.29   | 0.118    |
| 10     | 0.427    | 0.103  | 0.334  | 0.322  | 0.336  | 0.3044   |
| 11     | 0.29     | 0.257  | 0.047  | 0.267  | 0.091  | 0.1904   |
| 12     | 0.311    | 0.281  | 0.661  | 0.627  | 0.232  | 0.4224   |
| 13     | 0.441    | 0.185  | 0.293  | 0.394  | 0.554  | 0.3734   |
| 14     | 2.359    | 1.168  | 1.291  | 1.599  | 2.426  | 1.7686   |
| 15     | 1.023    | 1.106  | 1.187  | 1.162  | 1.095  | 1.1146   |
| 16     | 4.905    | 5.286  | 4.355  | 7.836  | 6.782  | 5.8328   |
| 17     | 2.949    | 2.838  | 4.398  | 4.88   | 5.43   | 4.099    |
| 18     | 37.066   | 17.536 | 43.602 | 17.938 | 31.321 | 29.4926  |
| 19     | 2.621    | 2.923  | 3.684  | 2.394  | 2.462  | 2.8168   |

Graficas del tiempo en segundos con respecto al tamaño:

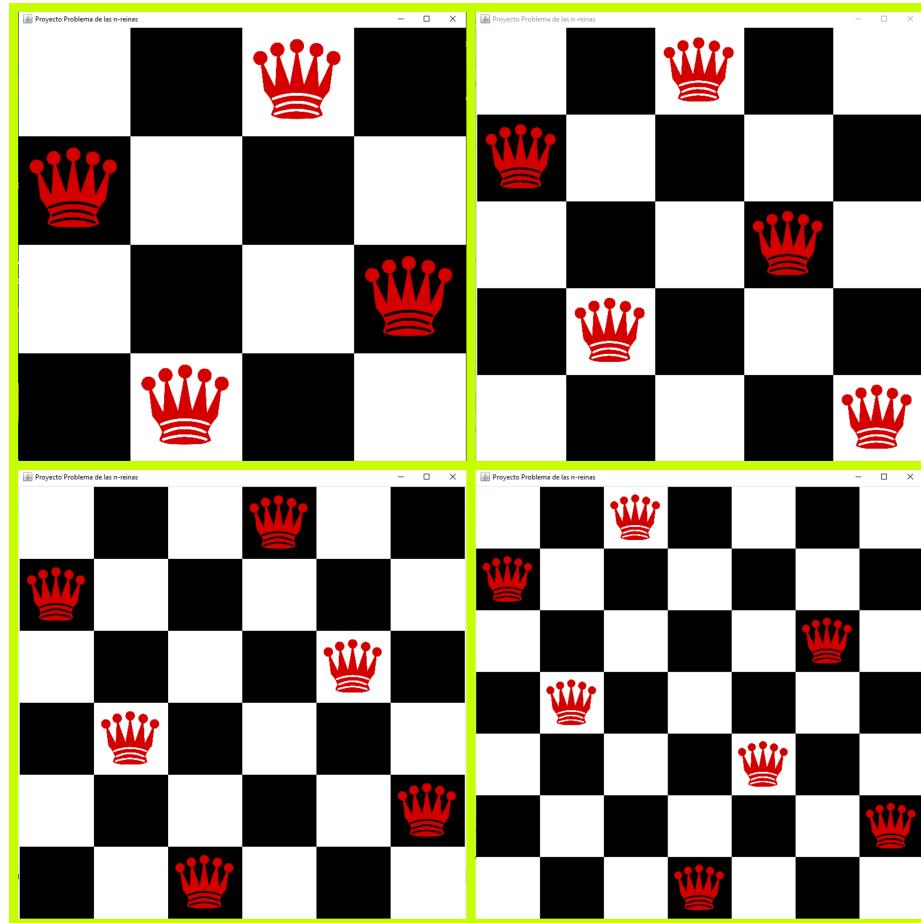


## Versión concurrente

Paralelizando solo la primera columna

Pruebas para tableros de distintos tamaños:

- Tamaño de tablero 4x4 a 7x7.



- Tamaño de tablero 8x8 a 11x11.

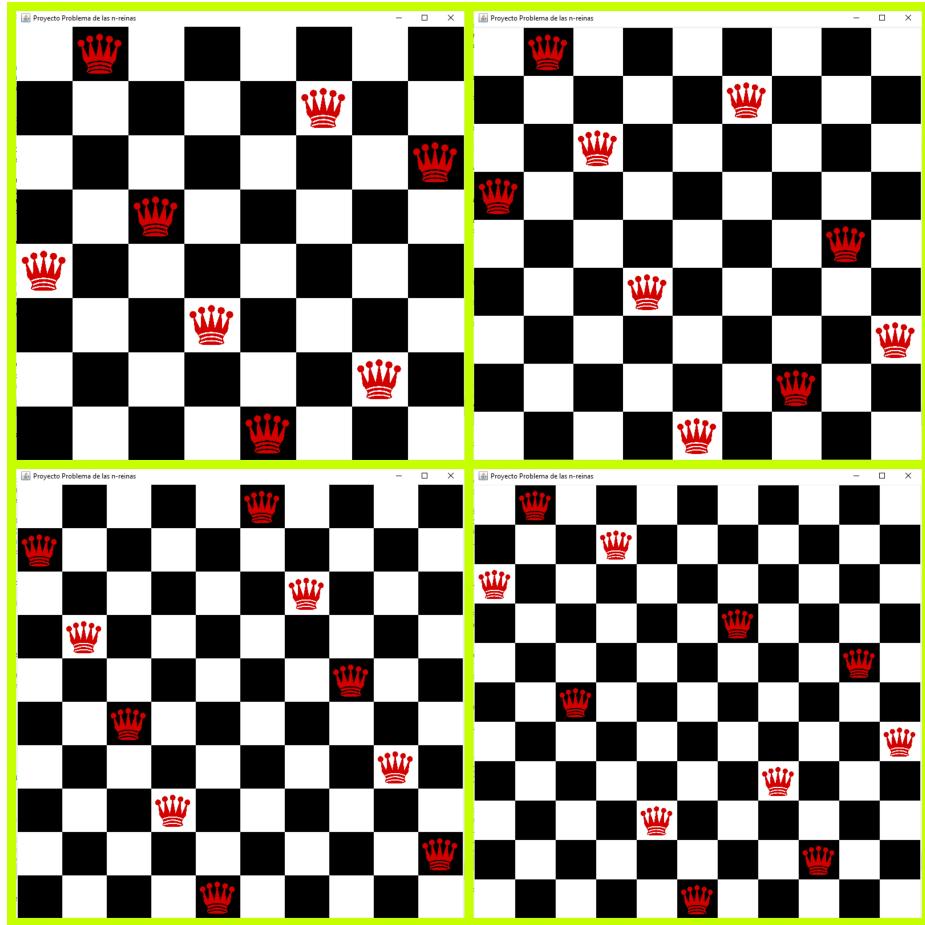
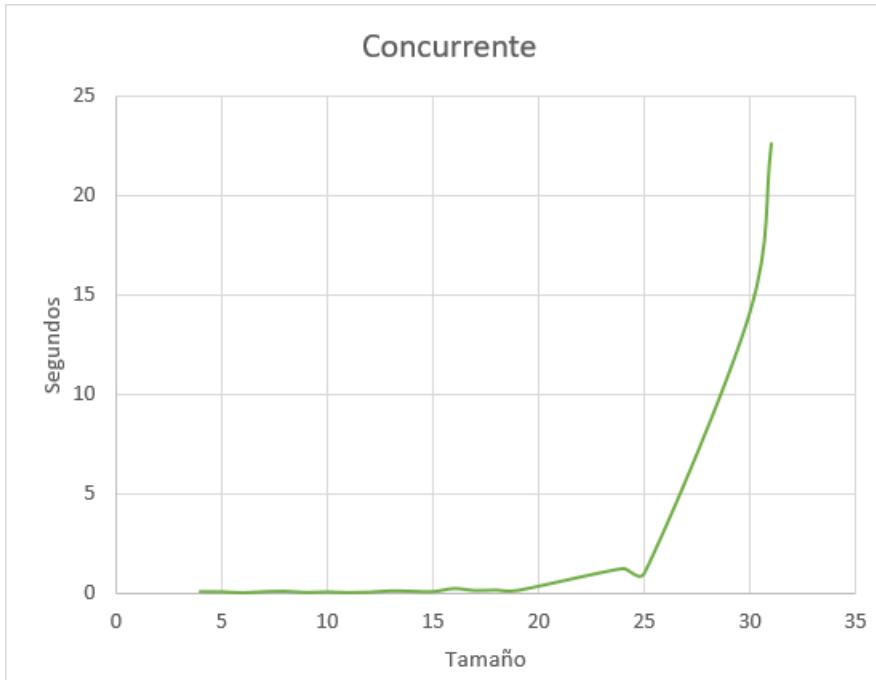


Tabla del tiempo en segundos con varias pruebas para cada tamaño, obteniendo un promedio:

| Tamaño | Segundos |        |        |        |        | Promedio |
|--------|----------|--------|--------|--------|--------|----------|
|        | 1er      | 2do    | 3er    | 4to    | 5to    |          |
| 4      | 0.069    | 0.105  | 0.09   | 0.097  | 0.098  | 0.0918   |
| 5      | 0.091    | 0.098  | 0.09   | 0.045  | 0.097  | 0.0842   |
| 6      | 0.045    | 0.036  | 0.046  | 0.036  | 0.034  | 0.0394   |
| 7      | 0.093    | 0.106  | 0.097  | 0.109  | 0.042  | 0.0894   |
| 8      | 0.113    | 0.109  | 0.134  | 0.101  | 0.098  | 0.111    |
| 9      | 0.051    | 0.05   | 0.056  | 0.069  | 0.039  | 0.053    |
| 10     | 0.048    | 0.062  | 0.122  | 0.11   | 0.056  | 0.0796   |
| 11     | 0.044    | 0.03   | 0.041  | 0.043  | 0.075  | 0.0466   |
| 12     | 0.086    | 0.041  | 0.07   | 0.067  | 0.069  | 0.0666   |
| 13     | 0.158    | 0.119  | 0.116  | 0.146  | 0.12   | 0.1318   |
| 14     | 0.194    | 0.056  | 0.049  | 0.203  | 0.063  | 0.113    |
| 15     | 0.071    | 0.091  | 0.181  | 0.059  | 0.055  | 0.0914   |
| 16     | 0.152    | 0.111  | 0.68   | 0.17   | 0.118  | 0.2462   |
| 17     | 0.094    | 0.092  | 0.174  | 0.076  | 0.314  | 0.15     |
| 18     | 0.12     | 0.096  | 0.12   | 0.173  | 0.354  | 0.1726   |
| 19     | 0.43     | 0.11   | 0.043  | 0.104  | 0.077  | 0.1528   |
| 24     | 0.771    | 1.027  | 1.465  | 1.488  | 1.53   | 1.2562   |
| 25     | 1.683    | 1.349  | 0.474  | 1.526  | 0.093  | 1.025    |
| 30     | 12.246   | 13.437 | 14.176 | 9.107  | 22.002 | 14.1936  |
| 31     | 22.246   | 24.529 | 22.516 | 21.959 | 21.893 | 22.6286  |

Grafica del tiempo en segundos con respecto al tamaño:



# Conclusiones

## Troncoso González Carlos Andrés

La elección del problema de las N-reinas para este proyecto fue de mi particular interés, ya que no solo el ajedrez es un pasatiempo que me ha acompañado desde chico, sino que también se trata de un problema con el que ya había tenido contacto.

El semestre pasado, mientras estudiábamos cuáles son las distintas estrategias para construir algoritmos, se nos presentaron diversos problemas y se nos solicitó imaginar cómo podrían funcionar algoritmos de cada estrategia, de forma que solucionen uno de estos problemas. En esa ocasión, para la forma backtrack de resolver el problema de las N-reinas, yo imaginé el algoritmo secuencial que presentamos a lo largo de este proyecto. Por lo que implementarlo en este proyecto tiene un fuerte significado para mí.

Sin embargo, algo que en esa ocasión nunca imaginé que sucedería, es paralelizar ese algoritmo; así que el primer conflicto con el que nos encontramos fue determinar si el algoritmo es paralelizable, y si sí lo es, ¿cuáles son las regiones paralelizables?. La respuesta a esta pregunta me llegó al volver a analizar lo que significa que un algoritmo sea Backtrack; y eso es, que cada vez que se presenten diversas alternativas para continuar, el algoritmo escogerá una de ellas. De manera que si llega a un punto en el que "descubre" que el "camino" tomado no es el correcto, regresa a la última decisión tomada para probar las otras alternativas existentes. Por lo tanto, las regiones paralelizables son justamente las zonas en las que se "toma una decisión", ya que las demás alternativas podrían ser "probadas".<sup>a</sup>l mismo tiempo, y simplemente monitorear cuál es el primer "camino".<sup>en</sup>contrado que lleva a la solución del problema.

De esta manera, podemos analizar al problema de las N-reinas como un "laberinto", en el cual repetidamente se tiene que tomar una decisión de cuál camino tomar para intentar salir de él; relacionándolo con la implementación paralela, esto sería como si la persona en el laberinto pudiera dividirse en  $n+1$  personas cada vez que se encuentre con  $n$  caminos alternativos; de forma que las  $n$  personas siguieran cada una una alternativa diferente, y la persona original se quedara en ese lugar a esperar que uno de ellos regrese con la solución para salir del laberinto; y a su vez cada uno de esos "clones" también podrán "dividirse" para una futura decisión. Analizándolo así, es fácil visualizar que con el crecimiento del tablero, las alternativas crecen sumamente rápido, por lo que también lo harán la cantidad de hilos generados.

Una vez más, imaginando que cada hilo está probando solucionar el problema de las n-reinas a partir de un cierto avance, también se hace evidente que cada hilo necesitará trabajar con un tablero personal; ya que si todos quisieran probar sus alternativas sobre el mismo tablero, constantemente surgirían confusiones. Esto implica un problema bastante fuerte, ya que el tablero es un objeto complejo que necesitará crearse a partir de otro, lo que combinado con el aumento abrupto de alternativas, implica la creación de demasiados tableros. Además, los tableros se "utilizarán" menos mientras más se paraleliza el problema, ya que el hilo que se "queda en el lugar" (volviéndose Manager al crear Workers) no volverá a modificar su tablero. Esto supone una primera pista de que el paralelizar demasiado el problema puede no ser la mejor idea.

Una segunda pista de porqué no es buena idea paralelizar todo, es el aumento en lo fina que es la Granularidad, ya que mientras más hilos más Managers, y estos implican gastar recursos en la comunicación entre hilos; esto llega incluso al punto de que la mayor parte de los hilos activos podrían ser Managers, y que los Workers sean minoría.

Analizando este caso de paralelizar cada decisión que se toma en el "laberinto", llegamos a la conclusión de que esto sería equivalente a intentar probar todos los caminos al mismo tiempo, por lo que el

algoritmo pasaría de ser BackTrack, a ser un BruteForce paralelizado al extremo.

Descubrir esto nos llevó a probar con "híbridos." entre la versión completamente paralelizada y la versión secuencias; llegando así a algoritmos con diversas granularidades, balances de carga, tipos de paralelismo, etc.

Al final, las versiones más útiles fueron dos; en la mayoría de casos la más útil es una que paraleliza únicamente la ejecución de la primer columna (como si en el algoritmo solo se "dividieran." la primer decisión del "laberinto", y que a partir de ahí los clones" trabajaran con el algoritmo secuencial), siendo esta una solución con paralelismo geométrico y un balance de carga estático. Y la otra versión, mejor en algunos casos particulares, que se basa en limitar la cantidad de hilos activos a un valor especificado, por lo que en cada decisión el programa decidirá "dividirse." proseguir secuencialmente de acuerdo a dicha cantidad; representando así un balance de carga dinámico.

### **Vences Santillán Carlos Eduardo**

Como conclusión de este proyecto puedo decir que se cumplieron con los objetivos planteados al inicio, ya que fuimos capaces de generar un algoritmo paralelo al problema inicial; que en este caso fue el problema de las n-reinas. Una de las cosas que me pareció más importante de este proyecto fueron los conceptos que tocamos para saber la diferencia entre un algoritmo secuencial y uno paralelo; a lo largo de la teoría en clase fuimos analizando las características de cada uno y su funcionamiento, pero creo que la mejor manera de entender esto al cien por ciento es aplicándolo en un programa real.

Lo primero que hicimos para realizar este trabajo fue buscar un problema que cumpliera con las características que pidió el profesor. Una vez definido esto, lo que hicimos fue comprender a detalle las variables que se debían tomar en cuenta para generar una solución a este problema. Inicialmente por intuición hicimos varias pruebas de escritorio a mano en las que desarrollamos el problema y todo lo que se consideraba al agregar una reina, por lo que llegamos a la conclusión de que debíamos bloquear las casillas que no estuvieran disponibles para agregar una más.

Este concepto fue la base para generar el algoritmo secuencial, ya que el tablero podemos simularlo como un arreglo bidimensional. La lógica a seguir fue: al bloquear todas las casillas que estuvieran tomadas por una reina, solo quedarían las disponibles, por lo que al insertar una nueva también se bloquearían los nuevos espacios. Entonces, la solución radicaba en encontrar un algoritmo que hallara la manera de encontrar una combinación de posiciones en las que las reinas no se tocasen. Por lo que recurrimos al clásico prueba y error "Backtracking".

El verdadero objetivo del proyecto salió a la luz cuando debíamos paralelizar el algoritmo secuencial que implementamos, por lo que hacer esto tomó varias horas en pensar una forma lógica en la que se pudieran implementar los hilos. Como resultado de esta lluvia de ideas, nacieron las tres versiones que generamos de paralelización, pero después de hacer pruebas nos dimos cuenta que solo una no tenía una complejidad tan mala y genuinamente era más veloz que la versión secuencial. El hacer la medida de los tiempos fue la mejor parte, ya que cuando agregamos este algoritmo y vimos la diferencia cronometrada vimos que efectivamente cumplimos con la paralelización.

### **Villagrán Segura Denisse Abril**

Realizar un algoritmo paralelo del programa de las n reinas presentó un cierto grado de dificultad, ya que para poder realizar un algoritmo paralelo, o en este caso concurrente ya que se decidió utilizar el lenguaje de programación de Java que únicamente nos permite trabajar con programación concurrente, fue necesario tener muy presente los conceptos teóricos y prácticos visto en clase.

Lo anterior debido a que hay que tener un orden, comenzando por analizar qué parte del código es paralelizable y que tipo de paralelización es la que nos conviene, como en este se trabajo con programación concurrente se tuvo que analizar qué procesos podríamos dividir entre una cierta cantidad de hilos.

Al dividir el código en subprocessos surgieron diferentes opciones de cómo realizar dicha división y diversas cantidades de hilos a utilizar. Por lo que se realizaron diversas pruebas para analizar cual era más conveniente, es decir cual requería un menor tiempo de ejecución para distintos tamaños del tablero.

La opción que se seleccionó como la más conveniente fue dividir entre una cantidad  $n$  de hilos, del mismo tamaño que las filas de tablero, al realizar distintas pruebas con tamaños de 4 a 20 filas x columnas si se obtenían mejores tiempos de solución al problema en comparación al algoritmo secuencial.

Un problema respecto a esta división donde cada hilo probaba con una fila distinta de la primera columna es que al momento de que el tamaño del tablero crecía, la cantidad hilos también aumentaba, esto implica a su vez un granularidad fina en el código, por lo que la comunicaciones que había que realizar entre hilos para saber el momento en que alguno ya logró llegar a una solución es muy grande, lo que hacía que el algoritmo en un punto fuera menos eficiente. Aunque sí lo resolvió en menor tiempo que la versión secuencial la computadora se comenzaba a trabar por la cantidad de comunicaciones. Situación que no pasaba con el algoritmo secuencial aunque este se tardaba más en llegar al resultado.

La otra manera de dividir en subprocessos que se trabajó en fue tener una cierta cantidad de hilos predefinida sin importar el tamaño del tablero, el problema de esta implementación es que si se define una cantidad muy grande de hilos puede suceder lo mismo que paso con la solución anterior, por lo que al momento de realizar las pruebas si no es muy grande la cantidad de hilos si hay una cierta mejora en la complejidad sin que se llegue a atorar la computadora.

Al ver distintas versiones para realizar el algoritmo de manera concurrente resultó en una versión que mantiene un cierto límite de subprocessos, mientras que otra puede ser desde una división mínima hasta una división muy grande, que incluso puede hacer que al algoritmo deje de ser backtrack y pase a trabajar como un algoritmo de fuerza bruta.

Al momento de realizar las gráficas fue interesante visualizar cómo es que en el algoritmo secuencial tiene comportamientos distintos para tableros con tamaños pares que los que son de un tamaño impar, esto puede ser debido a que el acomodo con el que se puede llegar a un solución y con la manera en la que se van colocando las reinas resulta más sencillo que se llegue a la solución de una tablero impar.

Otra situación curiosa al momento de analizar los resultados de las tablas es que en ocasiones para algunos tamaños del tablero el tiempo en promedio que les tomó hallar la solución es algo menor a lo que se esperaría, ya que es menor a lo que lo tomó a su tamaño anterior, sobretodo con los primeros tamaños del algoritmo concurrente.

Con todo lo anterior considero que se logró gran parte del objetivo propuesto para este proyecto ya que no solo se logró realizar una versión concurrente del algoritmo propuesto, sino que se analizaron varias, viendo en como es el comportamiento de estas versiones y analizando a su vez por que es así el comportamiento. Dicho análisis teniendo en cuenta los conceptos visto en clase sobre programación paralela como lo son el balance de carga, la granularidad, la comunicación entre los hilos, y el tipo de paralelismo.

## Autoevaluación general

Como equipo creemos que trabajamos de una buena manera, todos nosotros nos esforzamos por dar ideas al comienzo cuando no teníamos ningún algoritmo en mente. Con ayuda de todos los integrantes nos fuimos apoyando para plasmar las distintas opiniones hasta llegar a una en común.

Las cosas a mejorar o que no salieron como pensábamos fueron las reuniones para compartir parte del progreso del proyecto. Una de las cosas que nos costaron trabajo fue sincronizar nuestros horarios para estar en llamada frecuentemente, ya que no todos tenían la oportunidad de acoplarse a determinada hora. Nuestra principal forma de comunicación fue por mensajería instantánea lo cual podríamos decir que no es tan fluido como hablar en una llamada o presencialmente.

Otra cosa importante que consideramos que realizamos de buena manera fue la “nivelación de conocimientos” entre los integrantes, ya que cada uno contaba con experiencias distintas en el entorno de java, por lo que las habilidades en este lenguaje estaban un poco disparejas. Pero con la retroalimentación y paciencia del equipo, todos pudimos llegar a comprender todo lo necesario para cumplir con los objetivos del proyecto. Consideramos que este último punto es fundamental para cada equipo de trabajo, ya que así todos pueden contribuir en algo para la elaboración de un trabajo formal.

## Referencias

- [1] ALFREDO C. & CESAR A., *Algoritmos para el problema de las n-reinas*. 30ma Conferencia Latinoamericana de Informática (CLEI2004), pages 160–167. Sociedad Peruana de Computación, September 2004. ISBN 9972-9876-2-0.
- [2] JUAN P., *Programación Concurrente con Java*. Universidad Complutense Madrid. Dep. Sistemas Informáticos y Programación, 2001. <http://grasia.fdi.ucm.es/jpavon/docencia/dso/programacionconcurrentejava.pdf>
- [3] CARLOS V., MIGUEL ANGEL M. & JOSÉ LUIS R., *EL ESQUEMA ALGORÍTMICO DEL BACKTRACKING. INTRODUCCIÓN A ESTE MÉTODO DE RESOLUCIÓN DE PROBLEMAS, ESTUDIO DE VARIANTES POSIBLES Y EJEMPLOS DE IMPLEMENTACIÓN*. <http://www.it-docs.net/ddata/3619.pdf>
- [4] SANTIAGO P. & CAMILO M., *Programación Paralela*. Universidad Nacional de Colombia. [http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela\\_teoria/presentacion\\_p.pdf](http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/presentacion_p.pdf)
- [5] ALICIA C., *Conceptos, algoritmo y aplicación al problema de las N – reinas* . Capítulo4. Un ejemplo y su implementación computacional. Universidad Nacional MAYOR DE SAN MARCOS. 2005. Lima, Peru. <https://sisbib.unmsm.edu.pe/bibvirtualdata/monografias/basic/riojas.ca/cap4.pdf>
- [6] ETSISI-UPM, *Problema de las N Reinas*. Arquitecturas Avanzadas. [https://www.etsisi.upm.es/sites/default/files/asigs/arquitecturas\\_avanzadas/practicas/MPI/nreinas.pdf](https://www.etsisi.upm.es/sites/default/files/asigs/arquitecturas_avanzadas/practicas/MPI/nreinas.pdf)