Universitat Politècnica de València (UPV)

Escuela Politécnica Superior de Alcoy

Paralelización del problema de las N-Reinas con OpenMP

PRÁCTICA 1

Computación Paralela
Curso 2024-2025

Autor: Marcos del Amo Fernández

Docente: Adolfo Ferre Vilaplana

22 de enero de 2025

Resumen

En este documento se presenta la paralelización del **problema de las N-Reinas** usando la tecnología `OpenMP`. Primero, se describe el problema y se enlaza con el editorial disponible en *LeetCode*, donde se plantea la solución de backtracking y la devolución de una lista de configuraciones (en formato filas con 'Q' y '.'). Finalmente, se propone la versión paralela con `OpenMP` que distribuye la búsqueda entre varios hilos y se muestran resultados de rendimiento comparándola con la versión secuencial.

DECLARACIÓN

En la siguiente declaración, explicitaré las partes del documento sobre los que no me corresponde la autoría:

1. Introducción, enunciado del problema de N-Reinas. [1]
2. Datos sobre los detalles del Hardware [6].
3. Toda la explicación del algoritmo secuencial, simplemente ha sido traducido, y la implementación secuencial ha sido muy ligeramente modificada, tan solo para que devuelva el número de soluciones en vez de las soluciones. [2].
4. En relación a la IA generativa, se ha utilizado ChatGPT, en tareas repetitivas o visuales, como lo es ayuda para manipulación de una plantilla de LateX, tablas, pies de página, cabeceras, y conjuntamente, enumeraciones y pseudocódigo. Siendo el resultado, manipulado últimamente por mí, a mi gusto.
5. Se ha usado material de la asignatura, así como otros recursos, principalmente como consulta [5][3][4]

Índice

1. Introducción

El **problema de las N-Reinas** consiste en colocar N reinas en un tablero de ajedrez $N \times N$ de forma que ninguna se ataque entre sí.

Dos reinas se atacan si comparten la misma fila, la misma columna o cualquiera de las dos diagonales (principal y secundaria). Se trata del ejemplo por excelencia cuando nos introducimos al *backtracking*.

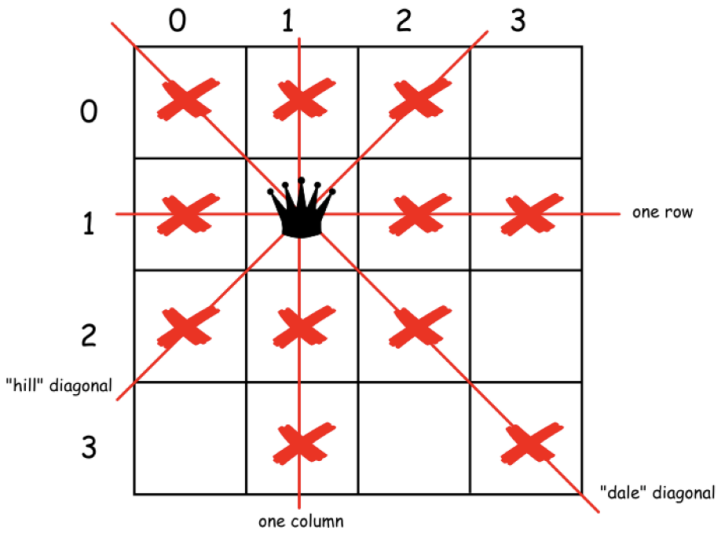


Figura 1: Fuente Leetcode

Por ejemplo, en la plataforma de programación competitiva LeetCode, se muestra el problema con el siguiente enunciado:

> *El juego de las n-reinas es el problema de colocar n reinas en un tablero de ajedrez de n x n de modo que ninguna de las dos reinas se ataquen entre sí.*
>
> *Dado un entero n, devuelve todas las soluciones distintas al rompecabezas de las n-reinas. Puedes devolver la respuesta en cualquier orden.*
>
> *Cada solución contiene una configuración de tablero distinta de la colocación de las n-reinas, donde 'Q' y '.' indican tanto una reina como un espacio vacío, respectivamente.*
>
> *Ejemplo:*
>
> *Input:* $n = 4$
>
> *Output:* [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]

51. N-Queens

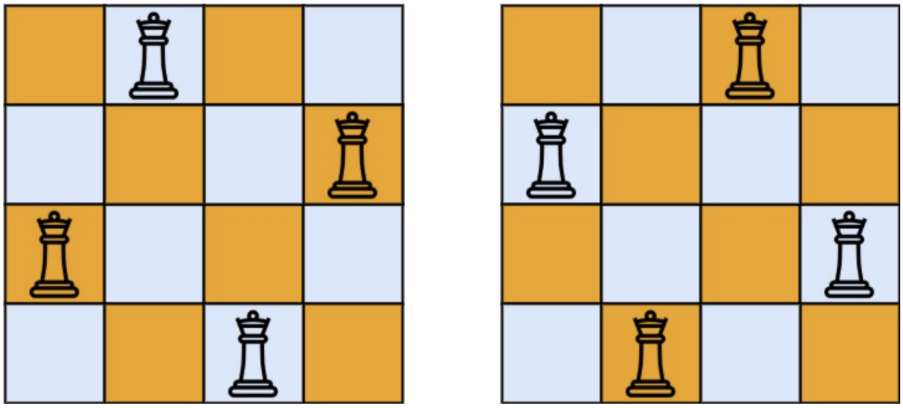
Hard | Topics | Companies

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above
```

Example 2:

```
Input: n = 1
Output: [["Q"]]
```

Figura 2: Captura del enunciado original de la plataforma LeetCode

Importante. Nosotros nos basaremos en este enunciado para la realización de la práctica, si bien es cierto que para simplificar la paralelización, haremos que nuestro algoritmo devuelva, dado un entero n, el número de distintas soluciones para el **problema de las n-reinas**, en vez de devolver una lista con dichas soluciones.

2. Motivación personal

En la práctica anterior de MPI, centré mis esfuerzos en el **8-Puzzle**, intentando paralelizar un algoritmo tan aparentemente simple y familiar como el *Búsqueda en Anchura*, y esta práctica iba a ser una extensión de la misma, pero en OpenMP. Sin embargo, me encontré con dificultades intrínsecas que no había previsto, lo que se tradujo en mucha complejidad a la hora de lograr una implementación paralela que tuviese sentido.

Precisamente por ello, en esta nueva práctica he querido cambiar de enfoque y basarme en un problema también clásico que, al menos en teoría, pienso que debería de resultar *más naturalmente paralelizable*: el de las **N-Reinas**. Se trata de un problema en el que cada rama de la búsqueda trabaja casi de forma independiente, encajando mucho mejor con la idea de *divide y vencerás*.

Además, el problema de las N-Reinas me resulta muy interesante desde que lo descubrí el año pasado al iniciarme en la *programación competitiva* a través de la plataforma LeetCode. Aquella vez, abordé su solución secuencial con `backtracking`. Por ello, ahora veo el momento idóneo para repasarlo e intentar una versión en la que, gracias a OpenMP, cada rama del *backtracking* se distribuya entre varios hilos y así comprobar el beneficio que puede aportar el paralelismo.

3. Consideraciones

Para la realización de este trabajo, se han empleado los siguientes recursos y configuraciones:

3.1. Detalles del Hardware

Las pruebas se han llevado a cabo en un sistema con las siguientes características:

- **Modelo del equipo:** MacBook Air (Mac15)
- **Chip:** Apple M2
- **Cantidad de núcleos:** 8 (4 de rendimiento y 4 de eficiencia)
- **Memoria RAM:** 16 GB

Cuadro 1: Especificaciones técnicas del chip Apple M2

Característica	**Especificación**
Procesador	
Número de núcleos de CPU	8 (4 de rendimiento y 4 de eficiencia)
Frecuencia máxima	No especificado (dinámico)
Caché L2 compartida núcleos AR	16 MB
Caché L2 compartida núcleos AE	4 MB
Memoria	
Memoria máxima unificada	24 GB
Ancho de banda de memoria	100 GB/s
Fabricación y Fotolitografía	
Tecnología de fabricación	5 nm (segunda generación)
Fabricante	TSMC
Gráficos y Neural Engine	
Número de núcleos gráficos	10
Rendimiento FP32	3.6 TFLOPS
Número de núcleos Neural Engine (NE)	16
Rendimiento Neural Engine	15.8 TOPS

3.2. Compilación y Ejecución

El código fue compilado haciendo llamadas desde la terminal de IDE Visual Studio Code tales como `g++ -fopenmp -std=c++17 -o fichero fichero.cpp`.

3.3. Limitaciones y Alcance

- Es importante subrayar que todo el código presentado en este documento está escrito exclusivamente en `C++`, elegido para facilitar la implementación al utilizar la librería `std::`, que ofrece estructuras de datos como `vector`, `queue` o `unordered_set`.
- Se explicará con detalle solamente la parte de la paralelización y la implementación en `OpenMP`, omitiendo explicaciones sobre la semántica de `C++` en general.
- Se es consciente de que los resultados obtenidos pueden variar en sistemas con hardware o implementaciones de `OpenMP` diferentes.
- Se es consciente de que el problema estudiado (N-Reinas) tiene pocas tallas que son razonables de calcular, ya que el coste crece exponencialmente y a partir de talla 16 hablamos de ejecuciones de más de 30 minutos, por lo que se aclara desde aquí que no se van a probar tales tallas para simplificar.

4. Algoritmo secuencial

En esta sección se describe el algoritmo secuencial para resolver el problema de las N-Reinas utilizando backtracking. Este algoritmo explora todas las posibles configuraciones del tablero fila por fila, verificando que las reinas colocadas no se ataquen entre ellas. La solución presentada ha sido extraida de la explicación del Editorial de la plataforma Leetcode.

4.1. Intuición

Una solución de fuerza bruta sería generar todos los tableros posibles de tamaño $N \times N$ con N reinas. Sin embargo, habría N^2 posibles ubicaciones para la primera reina, (N^2-1) para la segunda y así sucesivamente, alcanzando una complejidad de $O\left(N^{2N}\right)$. En la práctica, el número de configuraciones realmente válidas es muchísimo menor, así que conviene descartar las posiciones inválidas tan pronto como sea posible. Por ejemplo, imaginemos que colocamos la primera reina en la esquina superior izquierda de un tablero de ajedrez estándar de 8 reinas (coordenadas $(0, 0)$). Luego, si intentáramos poner la segunda reina justo a su derecha (coordenadas $(0, 1)$), ya sabemos que esas dos reinas se atacan en la misma fila. Nos ahorraríamos explorar cualquiera de las miles de millones de posibilidades para las reinas restantes, ya que estarían todas invalidadas.

Figura 3: Fuente Leetcode

Aún podemos seguir la estrategia de generar estados del tablero, pero nunca deberíamos colocar una reina en una casilla en la que otra reina pueda atacarla. Este es un problema perfecto para el *backtracking*: coloca las reinas una por una y, cuando se agoten todas las posibilidades, retrocede eliminando una reina y colocándola en otro lugar.

Dado un estado del tablero y una posible ubicación para una reina, necesitamos una forma para determinar si colocarla ahí hará que la reina sea atacada. Una reina puede ser atacada si hay otra reina en la misma fila, columna, diagonal o anti-diagonal.

Recordemos que para implementar el *backtracking*, implementamos una función `backtrack` que hace algunos cambios en el estado, se llama a sí misma de nuevo, y luego, cuando esa llamada retorna, deshace esos cambios (esta última parte es la razón por la que se llama "backtracking").

Cada vez que se llama a nuestra función `backtrack`, podemos codificar el estado de la siguiente manera:

- Para asegurarnos de colocar solo 1 reina por fila, pasaremos un argumento entero `row` a `backtrack` y solo colocaremos una reina durante cada llamada. Cada vez que coloquemos una reina, avanzaremos a la siguiente fila llamando otra vez a `backtrack` con el valor de parámetro `row + 1`.
- Para asegurarnos de colocar solo 1 reina por columna, usaremos un conjunto. Cada vez que coloquemos una reina, podemos añadir el índice de su columna a dicho conjunto.

Las diagonales son un poco más complicadas, pero tienen una propiedad que podemos aprovechar:

- Para cada casilla de una diagonal dada, la diferencia entre los índices de fila y de columna (`row - col`) será constante. Piensa en la diagonal que empieza en $(0, 0)$: la casilla i-ésima tiene las coordenadas (i, i), así que la diferencia siempre es 0.

Every square has value (row - col). Diagonals share the same values

	0	1	2	3	4	5	6	7
0	0	-1	-2	-3	-4	-5	-6	-7
1	1	0	-1	-2	-3	-4	-5	-6
2	2	1	0	-1	-2	-3	-4	-5
3	3	2	1	0	-1	-2	-3	-4
4	4	3	2	1	0	-1	-2	-3
5	5	4	3	2	1	0	-1	-2
6	6	5	4	3	2	1	0	-1
7	7	6	5	4	3	2	1	0

Figura 4: Fuente Leetcode

- Para cada casilla de una anti-diagonal dada, la suma de los índices de fila y columna (`row + col`) será constante. Si comenzaras en la casilla más alta de una anti-diagonal y fueras hacia abajo, el índice de fila se incrementa en 1 (`row + 1`) y el índice de columna se decrementa en 1 (`col - 1`). Estos cambios se anulan mutuamente.

Every square has value (row + col).
Anti-diagonals share the same values

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14

Figura 5: Fuente Leetcode

Cada vez que coloquemos una reina, deberíamos calcular la diagonal y la

anti-diagonal a la que pertenece. Del mismo modo que usamos un conjunto para realizar un seguimiento de qué columnas se han utilizado, también deberíamos contar con un conjunto para llevar el control de qué diagonales y anti-diagonales se han usado. Luego, podemos añadir los valores de esta reina a los conjuntos correspondientes.

4.2. Algoritmo

Crearemos una función recursiva `backtrack` que reciba 4 argumentos para mantener el estado del tablero. El primer parámetro es la fila en la que vamos a colocar la siguiente reina, y los otros 3 son conjuntos que registran qué columnas, diagonales y anti-diagonales ya tienen reinas. La función funciona de la siguiente manera:

1. Si la fila actual que estamos considerando es mayor que `n`, significa que hemos encontrado una solución. En ese caso, devolvemos `1`.
2. Declaramos una variable local `solutions = 0`, que representará todas las soluciones posibles que se pueden obtener desde el estado actual del tablero.
3. Recorremos las columnas de la fila actual. En cada columna, intentamos colocar una reina en la casilla (`row`, `col`). (Recuerda que estamos considerando la fila actual a través de los argumentos de la función).
4. Calculamos la diagonal y la anti-diagonal a la que pertenece esa casilla. Si no hay ninguna reina todavía en esa columna, diagonal o anti-diagonal, entonces podemos colocar una reina en esa columna, en la fila actual.
   - Si no se puede colocar la reina, se omite esa columna (y pasamos a la siguiente).
   - Si logramos colocarla, entonces actualizamos nuestros 3 conjuntos (`cols`, `diagonals` y `antiDiagonals`) y llamamos de nuevo a la función, pero con `row + 1`.
5. La llamada realizada en el paso anterior explora todos los estados de tablero válidos con la reina que pusimos en el paso 3. Cuando terminamos de explorar ese camino, retrocedemos (*backtrack*) retirando la reina de la casilla, lo que incluye eliminar los valores añadidos en nuestros conjuntos.

4.3. Análisis de la complejidad

Complejidad temporal: $O(N!)$. Donde N es el número de reinas (y coincide con la anchura y altura del tablero). A diferencia del enfoque de fuerza bruta, solo colocamos reinas en casillas que no estén atacadas. Para la primera reina, tenemos N opciones. Para la siguiente, no intentaremos colocarla en la misma columna que la primera reina, y habrá al menos una casilla atacada diagonalmente por la primera. Por lo tanto, el número máximo de casillas que podemos considerar para la segunda reina es $(N - 2)$. Para la tercera reina, no la colocaremos en 2 columnas ya ocupadas por las primeras 2 reinas, y habrá al menos dos casillas más atacadas en diagonal. De este modo, el número máximo de casillas para la tercera reina es $(N - 4)$. Siguiendo este patrón, la complejidad temporal aproximada es $N!$.

Complejidad espacial: $O(N)$. Donde N es el número de reinas (y coincide con la anchura y altura del tablero). La memoria adicional incluye los 3 conjuntos que guardan el estado del tablero, además de la pila de llamadas de la recursión. Todo ello crece de forma lineal con N.

4.4. Pseudocódigo: Algoritmo Secuencial

```
funcion colocarReina(tablero, fila):
    si fila = N entonces
        soluciones += 1
        devolver
    fin si

    para cada columna desde 0 hasta N-1 hacer:
        si esPosicionValida(tablero, fila, columna) entonces
            tablero[fila] = columna
            colocarReina(tablero, fila + 1)
        fin si
    fin para
fin funcion

funcion principal():
    inicializar tablero como un arreglo de tamano N con -1
    llamar colocarReina(tablero, 0)
    reportar el numero de soluciones y el tiempo de ejecucion
fin funcion
```

Listing 1: Pseudocodigo de N-Reinas (version secuencial)

Véase el anexo para implementación completa.

5. Algoritmo Paralelo con OpenMP

A continuación, se presenta una estrategia de paralelización muy sencilla, utilizando OpenMP. Simplemente trataremos de distribuir la exploración de las ramas del árbol de búsqueda desde el nivel más alto. Esto funcionaría tal que se distribuye el trabajo en la fila 0 y a partir de ahí continúa de manera secuencial en las demás filas.

5.1. Estrategia: Algoritmo Paralelo

1. La colocación de la reina en la primera fila (`fila = 0`) se paraleliza con la directiva `#pragma omp parallel for`. Cada columna en la primera fila representa una rama independiente.
2. Cada hilo procesa una de estas ramas haciendo el backtracking de manera secuencial para las filas restantes.
3. Se utilizan variables locales para las métricas de cada hilo que se combinan al final con variables globales, usando `omp atomic` o `omp critical`.

5.2. Pseudocódigo: Algoritmo Paralelo

```
funcion principalOMP():
    #pragma omp parallel for schedule(dynamic)
    para cada columna desde 0 hasta N-1 hacer:
        inicializar tablero como un arreglo de tamano N con -1
        tablero[0] = columna
        llamar colocarReina(tablero, 1)
    fin para

    reportar el numero de soluciones y el tiempo de ejecucion
fin funcion
```

Listing 2: Pseudocodigo de N-Reinas (version OpenMP)

Véase el anexo para implementación completa.

5.3. Consideraciones: Algoritmo Paralelo

- Se usa `schedule(dynamic)` para equilibrar la carga entre hilos.
- El uso de directivas como `atomic` o `critical` garantiza consistencia en las variables compartidas.

6. Algoritmo de Paralelismo Anidado con OpenMP

En la solución paralela previa, paralelizábamos principalmente la primera fila del tablero. Sin embargo, cada hilo exploraba un subárbol relativamente grande, pero a medida que N crece, la profundidad del árbol de búsqueda también crece.

La idea del **paralelismo anidado** es abrir sucesivas regiones paralelas dentro de otras regiones paralelas, de modo que el backtracking también se ejecute en paralelo en filas más allá de la primera o segunda. Pero no siempre resulta conveniente anidar indefinidamente, ya que, crear muchas regiones paralelas puede potencialmente acarrear sobrecostes. Debemos, por tanto, fijar un `nivelMaximoAnidamiento` que indique hasta qué fila vamos a seguir abriendo nuevas regiones paralelas, y, a partir de ese momento, continuaremos secuencialmente.

6.1. Estrategia: Paralelismo Anidado

Mantenemos la misma lógica de *backtracking* descrita en apartados anteriores. Para cada fila, se intenta colocar una reina en cada columna (verificando mediante `esPosicionValida` que no esté atacada) y, de ser posible, se avanza recursivamente a la fila siguiente; en caso contrario, se retrocede (*backtrack*).

En concreto, gestionamos tres estructuras lógicas:

- `colUsada`: Indica si una columna concreta está ocupada por alguna reina.
- `diag1Usada`: Rastrea las diagonales principales (habitualmente indexadas como $(\texttt{fila} - \texttt{col} + N - 1)$).
- `diag2Usada`: Rastrea las diagonales secundarias ($(\texttt{fila} + \texttt{col})$).

El pseudocódigo describe la función `colocarReinaAnidado`, que:

- Comprueba si `fila` == `N`: en tal caso, todas las reinas han sido colocadas y se incrementa el contador de soluciones.
- Si la fila actual es menor que `nivelMaximoAnidamiento`, activa una región paralela con `#pragma omp parallel for`, procesando cada columna en un hilo distinto y generando copias locales (*colLocal*, *d1Local*, *d2Local*) para no interferir con el estado de otros hilos.
- De otro modo, itera las columnas en un bucle secuencial, marcando y desmarcando la posición de la reina mediante `marcar` y `desmarcar`.

6.2. Pseudocódigo: Paralelismo Anidado

A continuación se muestra la definición resumida de `colocarReinaAnidado`, que ejemplifica el paralelismo anidado:

```
funcion colocarReinaAnidado(fila, N, colUsada, diag1Usada, diag2Usada,
    nivelMaximoAnidamiento):
    si fila == N entonces
        totalSolucionesGlobal += 1
        devolver
    fin si

    si fila < nivelMaximoAnidamiento entonces
        #pragma omp parallel for
        para col desde 0 hasta N-1 hacer:
            colLocal  = copia(colUsada)
            d1Local   = copia(diag1Usada)
            d2Local   = copia(diag2Usada)

            si esPosicionValida(fila, col, colLocal, d1Local, d2Local, N
                ) entonces
                marcar(fila, col, colLocal, d1Local, d2Local, N)
                colocarReinaAnidado(fila + 1, N, colLocal, d1Local,
                    d2Local, nivelMaximoAnidamiento)
            fin si
        fin para
    si no
        para col desde 0 hasta N-1 hacer:
            si esPosicionValida(fila, col, colUsada, diag1Usada,
                diag2Usada, N) entonces
                marcar(fila, col, colUsada, diag1Usada, diag2Usada, N)
                colocarReinaAnidado(fila + 1, N, colUsada, diag1Usada,
                    diag2Usada, nivelMaximoAnidamiento)
                desmarcar(fila, col, colUsada, diag1Usada, diag2Usada, N
                    )
            fin si
        fin para
    fin si
fin funcion
```

Listing 3: Pseudocódigo función recursiva con paralelismo anidado

Véase el anexo para implementación completa.

6.3. Consideraciones:

`nivelMaximoAnidamiento:` Gracias a esta variable, decidimos cuántas filas iniciales se beneficiarán de una región paralela. Si es muy pequeño (por ejemplo, `nivelMaximoAnidamiento` = 1), se parecerá mucho a la solución previa que sólo paralelizaba la fila 0. Si es demasiado grande (por ejemplo, `nivelMaximoAnidamiento` = N), podría generar gran cantidad de hilos a niveles profundos, causando un overhead con la creación y sincronización de regiones paralelas en cada recursión.

7. Resultados

A continuación se muestran resultados de tiempo de ejecución (en segundos) de todas las versiones comentadas

(Por defecto, se ha decidido `nivelMaximoAnidamiento = 2`):

7.1. Tiempos de ejecución

N	Secuencial(s)	Paralelo(s)	P.Anidado(s)	Speedup Paralelo	Speedup P.Anidado
4	0,000006	0,000159	0,000166	0,037	0,036
6	0,000016	0,000160	0,000197	0,10	0,08
8	0.000492	0.000194	0.000205	2.54	2.40
10	0.004389	0.000852	0.000747	5.15	5.87
12	0.203299	0.051275	0.013983	3.96	14.53
14	2.276710	0.534589	0.349806	4.26	6.51
16	107.470000	21.698900	13.242000	4.95	8.12

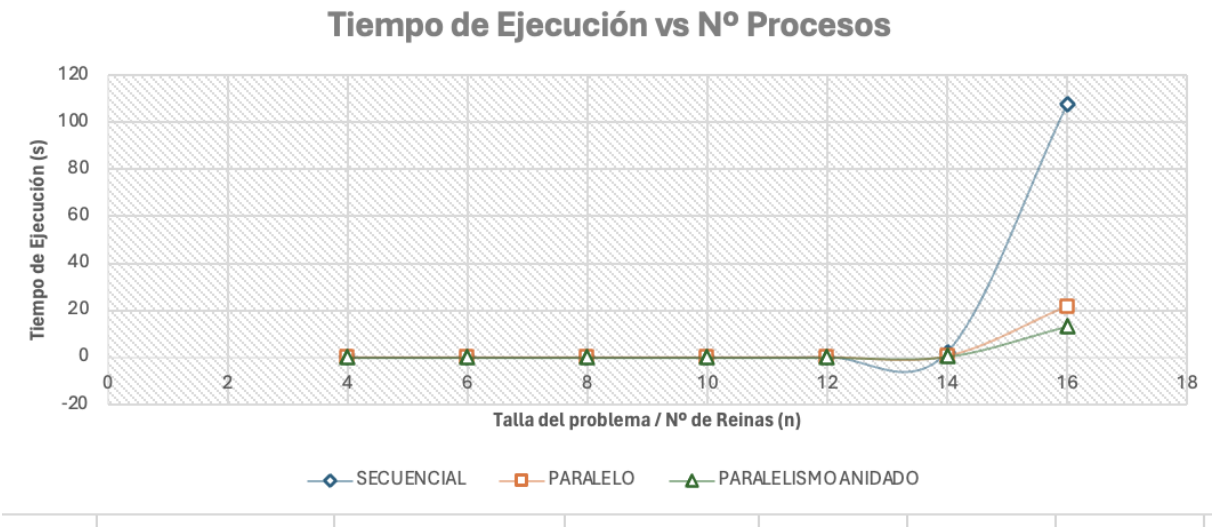


Figura 6: Elaboración propia

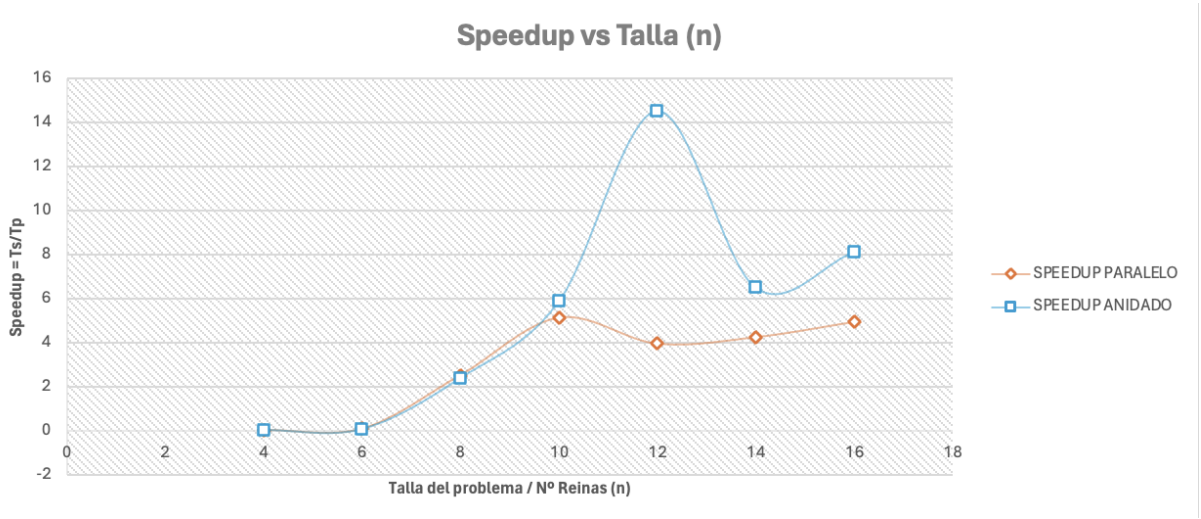


Figura 7: Elaboración propia

8. Discusión

Los resultados presentados en la tabla muestran los tiempos de ejecución para cada versión (Secuencial, Paralela y Paralelismo Anidado) en distintos tamaños de N. Al comparar las columnas, se observa que:

Para N pequeños (4, 6, 8), la versión secuencial ya es muy rápida en términos absolutos, por lo que las mejoras del paralelismo no son muy pronunciadas, de hecho para las tallas 4 y 6, el speedup es inferior a uno. De $N = 10$ en adelante, la solución paralela consigue acelerar la búsqueda respecto a la secuencial, y la versión con paralelismo anidado consigue superarla sin mucha holgura. En los casos $N = 12$ y especialmente $N = 16$, el paralelismo anidado alcanza una mayor ventaja sobre la versión secuencial que el paralelismo simple.

Al visualizar gráficamente (p.ej., trazando tiempo vs. N o *speedup* vs. N), se confirma que la estrategia anidada alcanza mayor rendimiento conforme crece el tamaño del tablero, distribuye mejor la carga de trabajo en las filas profundas del *backtracking*.

En la solución previa, al paralelizar solo la fila 0 (o filas iniciales), podemos saturar algunos hilos rápido pero, conforme avanzamos en filas internas, la expansión vuelve a ser secuencial en cada subárbol. Para tableros grandes, esto puede desaprovechar parte de la potencia de múltiples núcleos en niveles más profundos del árbol de búsqueda.

Sin embargo, el paralelismo anidado en demasiados niveles puede ser contraproducente, ya que creación y sincronización de hilos es sinónimo de overhead. El parámetro que controla esta situación ya lo hemos introducido antes, `nivelMaximoAnidamiento`, y debemos tener en cuenta de que las

pruebas se han realizado con un valor de anidamiento por defecto de 2, por lo que, aunque no se ha profundizado en valores críticos de este parámetro, creo que ajustándolo a la talla se podría alcanzar un rendimiento todavía mayor, como creo que es lo que ha pasado para la talla 12 reinas, que se alcanza el Speedup máximo, `14,53`.

9. Conclusiones

En conclusión, el problema de N-Reinas es un escenario ejemplar para profundizar en la programación paralela: pone de manifiesto cómo decisiones como "hasta dónde anidar" o "qué directivas de sincronización usar" pueden marcar diferencias abismales en la eficiencia. Haber obtenido speedups cercanos a 14 (según la configuración y el tamaño de N) refuerza la idea de que este problema sí puede sacarle partido al paralelismo, a diferencia de otros como el BFS sobre el 8-Puzzle, que plantean complicaciones intrínsecas y no son tan agradecidos en términos de mejora de tiempos.

Referencias

[1] **LeetCode Problem #51: N-Queens**. `https://leetcode.com/problems/n-queens/` (Ultimo acceso: 2025).

[2] *LeetCode Editorial for N-Queens (problem #51)*, `https://leetcode.com/problems/n-queens/editorial/` (Ultimo acceso: 2025).

[3] *Recursion II Explore Card, Backtracking section (LeetCode)*, `https://leetcode.com/explore/learn/card/recursion-ii/`

[4] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.2*, `https://www.openmp.org/specifications/`

[5] *Material de la asignatura Sistemas Inteligentes*, Universitat Politecnica de Valencia, 2024-2025.

[6] Xataka, *Microarquitectura del procesador M2 de Apple explicada: así sube la apuesta por el rendimiento y la eficiencia*. Disponible en: `https://www.xataka.com/componentes/microarquitectura-procesador-m2-apple-explicada-asi-sube-apuesta-rend` Última consulta: 13 de diciembre de 2024.

10. Anexos

10.1. Algoritmo Secuencial

```cpp
/* nqueens_secuencial.cpp */

#include <iostream>
#include <vector>
#include <chrono>

using namespace std;

static const int N = 4;

long long generados = 0;
long long expandidos = 0;
long long maximoAlmacenado = 0;
long long soluciones = 0;

bool esPosicionValida(const vector<int> &tablero, int fila, int columna) {
    for (int f = 0; f < fila; f++) {
        int c = tablero[f];
        if (c == columna) return false;
        if ((f - fila) == (c - columna)) return false;
        if ((f - fila) == -(c - columna)) return false;
    }
    return true;
}

void colocarReina(vector<int> &tablero, int fila) {
    expandidos++;
    if ((long long)(fila + 1) > maximoAlmacenado) {
        maximoAlmacenado = fila + 1;
    }
    if (fila == N) {
        soluciones++;
        return;
    }
    for (int columna = 0; columna < N; columna++) {
        generados++;
        if (esPosicionValida(tablero, fila, columna)) {
            tablero[fila] = columna;
            colocarReina(tablero, fila + 1);
        }
    }
}

int main() {
    auto inicio = chrono::high_resolution_clock::now();
    vector<int> tablero(N, -1);
    colocarReina(tablero, 0);
    auto fin = chrono::high_resolution_clock::now();
    double tiempo = chrono::duration<double>(fin - inicio).count();

    cout << "# estrategia generados expandidos maximo_almacenado
       soluciones tiempo\n";
```

```
52      cout << "N-Reinas(Seq)␣"
53           << generados << "␣"
54           << expandidos << "␣"
55           << maximoAlmacenado << "␣"
56           << soluciones << "␣"
57           << tiempo << "\n";
58
59      return 0;
60  }
```

10.2. Algoritmo Paralelo (OpenMP)

```
1   /* nqueens_omp.cpp */
2
3   #include <iostream>
4   #include <vector>
5   #include <chrono>
6   #include <omp.h>
7
8   using namespace std;
9
10  static const int N = 4;
11
12  long long generadosGlobal = 0;
13  long long expandidosGlobal = 0;
14  long long maximoAlmacenadoGlobal = 0;
15  long long solucionesGlobal = 0;
16
17  bool esPosicionValida(const vector<int> &tablero, int fila, int columna)
        {
18      for (int f = 0; f < fila; f++) {
19          int c = tablero[f];
20          if (c == columna) return false;
21          if ((f - fila) == (c - columna)) return false;
22          if ((f - fila) == -(c - columna)) return false;
23      }
24      return true;
25  }
26
27  void colocarReina(vector<int> &tablero, int fila,
28                    long long &generados,
29                    long long &expandidos,
30                    long long &maximoAlmacenado,
31                    long long &soluciones) {
32      expandidos++;
33      if ((long long)fila > maximoAlmacenado) {
34          maximoAlmacenado = fila;
35      }
36      if (fila == N) {
37          soluciones++;
38          return;
39      }
40      for (int columna = 0; columna < N; columna++) {
41          generados++;
42          if (esPosicionValida(tablero, fila, columna)) {
43              tablero[fila] = columna;
```

```
            colocarReina(tablero, fila + 1,
                         generados,
                         expandidos,
                         maximoAlmacenado,
                         soluciones);
        }
    }
}

int main() {
    auto inicio = chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(dynamic)
    for (int columna = 0; columna < N; columna++) {
        long long generadosLocal = 0;
        long long expandidosLocal = 0;
        long long maximoAlmacenadoLocal = 0;
        long long solucionesLocal = 0;
        vector<int> tablero(N, -1);
        tablero[0] = columna;
        colocarReina(tablero, 1,
                     generadosLocal,
                     expandidosLocal,
                     maximoAlmacenadoLocal,
                     solucionesLocal);
        #pragma omp atomic
        generadosGlobal += generadosLocal;
        #pragma omp atomic
        expandidosGlobal += expandidosLocal;
        #pragma omp critical
        {
            if (maximoAlmacenadoLocal > maximoAlmacenadoGlobal) {
                maximoAlmacenadoGlobal = maximoAlmacenadoLocal;
            }
            solucionesGlobal += solucionesLocal;
        }
    }

    auto fin = chrono::high_resolution_clock::now();
    double tiempo = chrono::duration<double>(fin - inicio).count();

    cout << "# estrategia generados expandidos maximo_almacenado
       soluciones tiempo\n";
    cout << "N-Reinas(OMP) "
         << generadosGlobal << " "
         << expandidosGlobal << " "
         << maximoAlmacenadoGlobal << " "
         << solucionesGlobal << " "
         << tiempo << "\n";

    return 0;
}
```

10.3. Algoritmo con Paralelismo Anidado

```
/* nqueens_nested.cpp */
```

```
2
3  #include <iostream>
4  #include <vector>
5  #include <cstdlib>
6  #include <omp.h>
7
8  static long long solucionesTotales = 0;
9  #pragma omp threadprivate(solucionesTotales)
10
11 bool esPosicionValida(int fila, int columna,
12                        const std::vector<bool> &colUsada,
13                        const std::vector<bool> &diag1Usada,
14                        const std::vector<bool> &diag2Usada,
15                        int n) {
16     if (colUsada[columna]) return false;
17     if (diag1Usada[fila - columna + n - 1]) return false;
18     if (diag2Usada[fila + columna]) return false;
19     return true;
20 }
21
22 void colocarReinaAnidado(int fila, int n,
23                          std::vector<bool> &colUsada,
24                          std::vector<bool> &diag1Usada,
25                          std::vector<bool> &diag2Usada,
26                          int nivelMaximoAnidamiento) {
27     if (fila == n) {
28         solucionesTotales++;
29         return;
30     }
31     if (fila < nivelMaximoAnidamiento) {
32         #pragma omp parallel for default(none) \
33         shared(colUsada, diag1Usada, diag2Usada, n, fila,
               nivelMaximoAnidamiento)
34         for (int columna = 0; columna < n; columna++) {
35             std::vector<bool> colLocal(colUsada);
36             std::vector<bool> diag1Local(diag1Usada);
37             std::vector<bool> diag2Local(diag2Usada);
38             if (esPosicionValida(fila, columna, colLocal, diag1Local,
                   diag2Local, n)) {
39                 colLocal[columna] = true;
40                 diag1Local[fila - columna + n - 1] = true;
41                 diag2Local[fila + columna] = true;
42                 colocarReinaAnidado(fila + 1, n,
43                                     colLocal,
44                                     diag1Local,
45                                     diag2Local,
46                                     nivelMaximoAnidamiento);
47             }
48         }
49     } else {
50         for (int columna = 0; columna < n; columna++) {
51             if (esPosicionValida(fila, columna, colUsada, diag1Usada,
                   diag2Usada, n)) {
52                 colUsada[columna] = true;
53                 diag1Usada[fila - columna + n - 1] = true;
54                 diag2Usada[fila + columna] = true;
55                 colocarReinaAnidado(fila + 1, n,
56                                     colUsada, diag1Usada, diag2Usada,
```

```
57                                      nivelMaximoAnidamiento);
58                 colUsada[columna] = false;
59                 diag1Usada[fila - columna + n - 1] = false;
60                 diag2Usada[fila + columna] = false;
61             }
62         }
63     }
64 }
65
66 int main(int argc, char* argv[]) {
67     if (argc < 3) {
68         return 1;
69     }
70     int n = 0;
71     int nivelMaximoAnidamiento = 2;
72     for (int i = 1; i < argc; ++i) {
73         if (std::string(argv[i]) == "-n") {
74             n = std::atoi(argv[++i]);
75         } else {
76             nivelMaximoAnidamiento = std::atoi(argv[i]);
77         }
78     }
79     std::vector<bool> colUsada(n, false);
80     std::vector<bool> diag1Usada(2*n - 1, false);
81     std::vector<bool> diag2Usada(2*n - 1, false);
82     double inicio = omp_get_wtime();
83     colocarReinaAnidado(0, n, colUsada, diag1Usada, diag2Usada,
           nivelMaximoAnidamiento);
84     double fin = omp_get_wtime();
85     long long soluciones = 0;
86     #pragma omp parallel reduction(+:soluciones)
87     {
88         soluciones += solucionesTotales;
89     }
90     std::cout << "Numero␣de␣soluciones␣para␣N=" << n
91               << "␣(paralelismo␣anidado)␣=␣" << soluciones << std::endl;
92     std::cout << "Tiempo␣transcurrido:␣" << (fin - inicio) << "␣segundos
           .\n";
93     return 0;
94 }
```