



Introducción a la Programación Paralela: Actividad Java

Bernardo Zapico (62318)

bzapico@itba.edu.ar

Índice

Índice.....	1
Introducción.....	2
Multiplicación de Matrices.....	2
Explicación del código.....	2
Comparación Secuencial y Paralelo con Threads de procesador.....	4
Threads de procesador y Threads virtuales.....	6
NQueens.....	9
Explicación del código.....	9
Comparación Secuencial y Paralelo con Threads de procesador.....	10
Threads virtuales.....	12
Conclusiones.....	13
Anexo.....	14
Tipos de paralelización.....	14
Código útil para ambos problemas.....	14
Multiplicación de Matrices.....	14
MatrixMain.....	14
MatrixMultiplication.....	15
NQueens.....	17
NQueensMain.....	17
NQueens.....	18

Introducción

En este proyecto se tomaron dos problemas y se buscó paralelizarlos mediante el uso de la API de concurrencia de Java y los cores del sistema donde se realizaron las pruebas, en mi caso contaba con ocho cores y una velocidad de CPU base de 2.80GHz. El objetivo del primer problema es, dado dos matrices aleatorias cuadradas de tamaño $N \times N$, calcular su producto matricial y el objetivo del segundo problema es, dado un número N , calcular el número de soluciones para el problema de las reinas en un tablero de $N \times N$ para N reinas. El código fuente donde se implementó las soluciones se encuentra en <https://github.com/berni-245/IPP-TP2> junto con su README con las instrucciones para correr el código. Los códigos también se encuentran en el [Anexo](#) y algunos serán referenciados durante las explicaciones.

Se utilizaron 5 distintas formas de paralelización como se observa implementado en el [Cod. 6](#) y serán las utilizadas para ambos problemas. Estas son la forma Secuencial (sin ninguna paralelización) llamada `SEQUENTIAL`, la forma paralelizada con Threads de procesador utilizando ExecutorService llamada `PARALLEL`, la forma paralelizada con Threads de procesador usando el framework de ForkJoin llamada `FORK_JOIN`, la forma paralelizada con Threads virtuales lanzado un Thread virtual por cada fila llamada `VIRTUAL_PER_ROW` y finalmente la forma paralelizada con Threads virtuales lanzado un Thread virtual para un rango de filas llamada `VIRTUAL_PER_CHUNK`.

Multiplicación de Matrices

Explicación del código

Para este problema se tuvo que implementar una versión del código del [enunciado](#) acerca de multiplicación de matrices cuadradas en java. Vamos a desglosar la implementación realizada.

Partiendo desde el Main ([Cod. 8](#)), se recibe por argumento el N de la matriz (se llamará habitualmente `size`), el tipo de paralelización (`type`), la cantidad de Threads (`numThreads`), el número de corridas que se va a realizar (`times`) y un boolean por si se desea imprimir el contenido de la celda `[0][0]` como en el enunciado (`showFirstCell`).

En base a los argumentos, se instancia la clase `MatrixMultiplication` con una `Seed` fija de 6834723, que será utilizada para generación de matrices iniciales determinísticas que son las que serán utilizadas para la multiplicación. Luego, sabiendo el tipo de paralelización solicitado, se llama a su método correspondiente en `MatrixMultiplication`.

En el main se hace un paso adicional en un método privado donde se llama `times` veces al método del producto matricial para la paralelización solicitada, imprimiendo cada una de esas iteraciones para futuro análisis desde un Jupyter

Notebook y también se setea el **Locale** en inglés para que se utilice el punto como separador decimal.

Yendo al corazón de la implementación, vamos a explicar la implementación de **MatrixMultiplication** (Cod. 4), donde se realizan todas las versiones de paralelización del producto matricial. Si bien era posible paralelizar también la inicialización de las matrices que se utilizarán en el producto, se decidió paralelizar únicamente la parte del producto, dejando valores constantes en las matrices iniciales para los distintos algoritmos y así buscar obtener los mismos resultados siempre.

```
public void multiplySequential() {
    multiplyRowInRange(0, size);

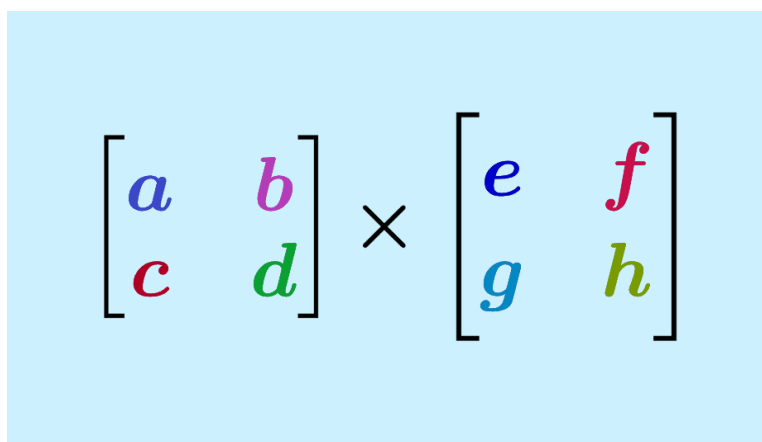
    showFirstCellIfNeeded();
}

// ...

// endIdx exclusive
private void multiplyRowInRange(int startIdx, int endIdx) {
    for (int rowIdx = startIdx; rowIdx < endIdx; rowIdx++) {
        for (int colIdx = 0; colIdx < size; colIdx++) {
            for (int k = 0; k < size; k++) {
                result[rowIdx][colIdx] += left[rowIdx][k] * right[k][colIdx];
            }
        }
    }
}
```

Código 1: Implementación secuencial de la multiplicación de matrices.

Empezaremos por entender la solución secuencial (Cod. 1), para así poder evaluar formas de paralelizar el problema. En el método **multiplySequential** se observa que solo hay dos líneas importantes, una llama a otro método **multiplyRowInRange** y le pasa el rango de todas las filas de la matriz **left** [0, size), y otro método que se encarga de imprimir la primera celda al realizar el producto matricial si se solicitó en el constructor.



El diagrama muestra la multiplicación de dos matrices 2x2. La primera matriz tiene elementos a (azul), b (púrpura), c (rojo) y d (verde). La segunda matriz tiene elementos e (azul), f (rojo), g (azul) y h (verde). Los elementos de la misma fila o columna en matrices diferentes comparten el mismo color. Un símbolo de multiplicación \times se encuentra entre las matrices.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Figura 1: Idea visual de un producto matricial para la explicación del código.

El verdadero producto matricial se encuentra en `multiplyRowInRange`, donde dado el rango de filas que se recibió por argumento, para cada índice de ese rango de filas de la primera matriz `left` se la asocia **todos** los índice de columna `[0, size)` de la matriz `right`. En el caso de la Fig. 1 el índice cero sería la fila `[a b]` asociada a la columna `[e ; g]` y luego a la columna `[f ; h]`, formando la primera fila en la matriz resultante mediante la cuenta:

$$\text{result}[0][0] = a * e + b * g \quad \text{y} \quad \text{result}[0][1] = a * f + b * h,$$

que se consigue para cada celda de la fila algorítmicamente dejando fijo el índice de fila de `left` y de columna de `right` y se varía el `k` entre `[0, size)`. Lo mismo sucede para la fila de índice uno.

Notar como la fila en `left` de índice cero, formó la fila en `result` de índice cero, es decir, por cada rango de filas que le paso, obtendré el rango de filas equivalente en `result`, esto va a ser **muy utilizado** para la paralelización.

Comparación Secuencial y Paralelo con Threads de procesador

```
public void multiplyParallel() {
    if (numThreads > size) {
        multiplySequential();
        return;
    }

    ExecutorService executor = Executors.newFixedThreadPool(numThreads);
    List<Future<?>> futures = new ArrayList<>();

    distributeChunksForAllThreads(
        (start, end) -> futures.add(executor.submit(() -> multiplyRowInRange(start, end)))
    );

    Utils.waitForAll(futures);
    Utils.shutdownExecutor(executor);
    showFirstCellIfNeeded();
}

// ...

private void distributeChunksForAllThreads(BiConsumer<Integer, Integer> chunkForThreadHandler) {
    int rowsPerThread = size / numThreads;
    int remainder = size % numThreads;

    int start = 0;
    for (int i = 0; i < numThreads; i++) {
        int end = start + rowsPerThread + (i < remainder ? 1 : 0); // handle remainder
        chunkForThreadHandler.accept(start, end);
        start = end;
    }
}
```

Código 2: Implementación paralela de la multiplicación de matrices.

Para la paralelización del código con el `ExecutorService`, se buscó distribuir los rangos de filas de la matriz `left` en partes iguales para cada `Future` (Cod. 2). Por ejemplo, si tenemos 4 `Future` y 20 filas, cada `Future` tendría un rango de 5 filas. En caso de que la división `filas/numFutures` no sea entera, se le otorga una fila del resto secuencialmente a cada `Future` hasta quedarse sin resto. Para cada uno de esos rangos de filas se calculan sus filas equivalentes de manera secuencial en `result` utilizando

`multiplyRowInRange`. Finalmente, se espera que todos los Future hayan terminado y luego se cierra el `ExecutorService`.

```
public void multiplyForkJoin() {
    ForkJoinPool pool = new ForkJoinPool(numThreads);
    int threshold = Math.max(20, size/numThreads);
    pool.invoke(new MultiplyTask(0, size, threshold));

    Utils.shutdownExecutor(pool);
    showFirstCellIfNeeded();

    // ...

    private class MultiplyTask extends RecursiveAction {
        private final int startRow;
        private final int endRow;
        private final int threshold;

        MultiplyTask(int startRow, int endRow, int threshold) {
            this.startRow = startRow;
            this.endRow = endRow;
            this.threshold = threshold;
        }

        @Override
        protected void compute() {
            if (endRow - startRow <= threshold) {
                multiplyRowInRange(startRow, endRow);
                return;
            }
            int middleRow = (startRow + endRow) / 2;
            invokeAll(new MultiplyTask(startRow, middleRow, threshold), new MultiplyTask(middleRow, endRow,
            threshold));
        }
    }
}
```

Código 3: Implementación paralela con ForkJoin de la multiplicación de matrices.

Mientras que para la paralelización con el `ForkJoin` de Java (Cod. 3) se creó un pool `ForkJoin`, permitiendo a tareas que estén en waiting tomar otras tareas distintas de manera similar a como hacía `OpenMP`. Luego, se buscó dividir los rangos de filas a la mitad de manera recursiva hasta llegar a cierto `threshold` en la longitud del rango para no tomar una recursión tan amplia por dividir mucho los rangos. El `threshold` se calcula usando: $threshold = \max(20, size/numThreads)$. El `invoke/invokeAll` ya espera que termine la tarea entonces se utiliza eso para garantizar que se haya terminado el producto matricial antes de cortar.

Después de haber definido las implementaciones y haber corroborado que funcionen correctamente, se quiso probar el número de Threads para usar en las pruebas paralelas. Se probó distintos números de Threads para distintos valores de N , tres veces para cada par (`numThreads`, N) para lograr una media y desvío acorde.

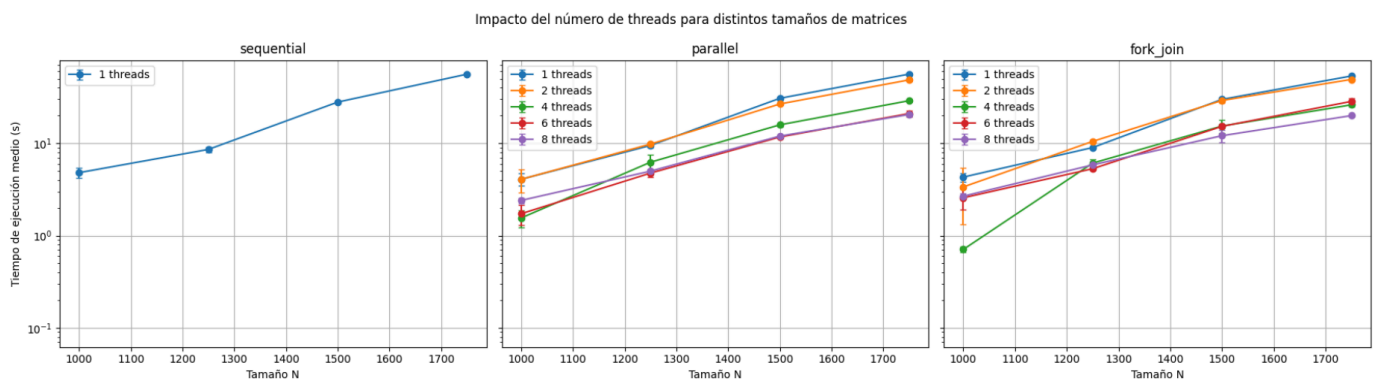


Figura 2: Comparación de tiempos de ejecución para distintos números de Threads para cada N entre 1000 y 1750 con step de 250.

Como se observa en la Fig. 2, cuando se usa un número bajo de Threads, el resultado tiende a tardar lo mismo que el algoritmo secuencial, mientras que a mayor cantidad de Threads parecida a la cantidad de cores, se logra un mejor tiempo de ejecución. Por eso se eligió utilizar 8 Threads para el resto de pruebas de multiplicación de matrices de manera paralela.

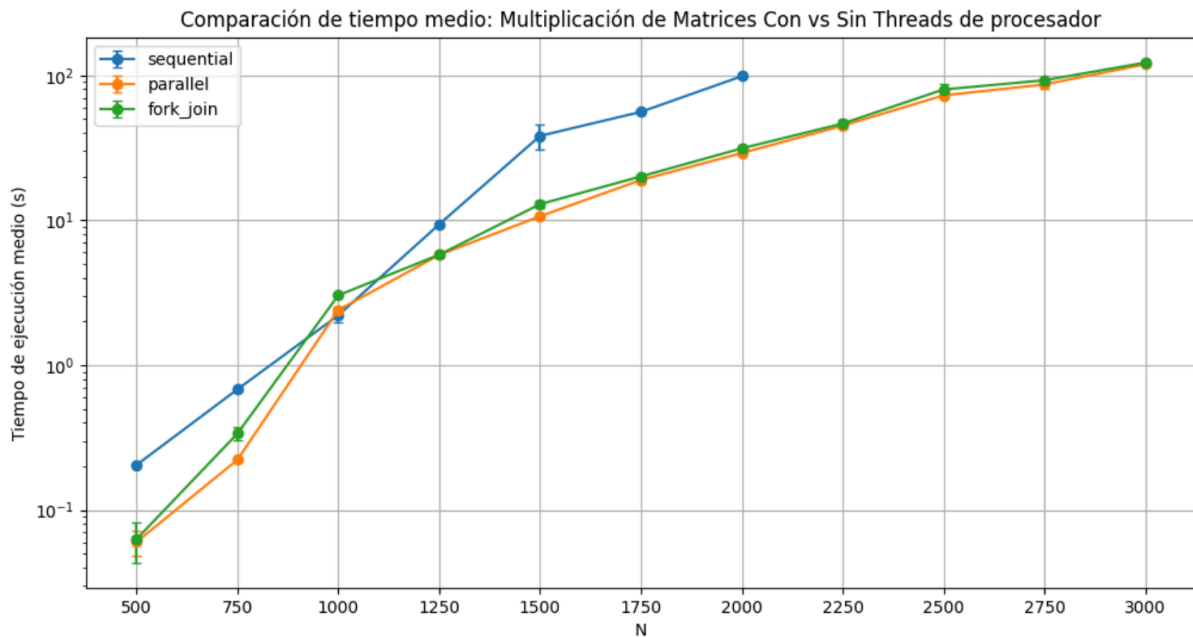


Figura 3: Comparación de tiempos de ejecución Secuencial vs Paralelo para cada N entre 500 y 3000 con step de 250.

Luego, se comparó los tiempos de ejecución para un rango de N mucho mayor en un promedio de 3 intentos por cada N (Fig. 3). Se observa que el secuencial rápidamente empieza a tardar mucho, comparativamente se ve que en N=2000 el secuencial tarda más o menos lo mismo que el resto en N=3000. También se observa que entre el algoritmo de paralelización propuesto con ExecutorService y el algoritmo propuesto con ForkJoin, no hay mucho cambio de tiempos de ejecución y dan tendencias similares.

Threads de procesador y Threads virtuales

Hasta ahora se estaba resolviendo el paralelismo con Threads de Procesador, es decir, Threads a nivel Kernel. A continuación se probó resolverlo el producto matricial utilizando Threads virtuales, estos son muchos más livianos, fáciles de crear y de realizar context switch, haciéndolos más escalables y eficientes. Por lo que se decidió probar con estos para paralelizar.

```

public void multiplyVirtualThreadsPerRow() {
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
    List<Future<?>> futures = new ArrayList<>();

    for (int row = 0; row < size; row++) {
        final int rowIdx = row;
        futures.add(executor.submit(() -> multiplyRowInRange(rowIdx, rowIdx + 1)));
    }

    Utils.waitForAll(futures);
    Utils.shutdownExecutor(executor);
    showFirstCellIfNeeded();
}

public void multiplyVirtualThreadsPerChunk() {
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
    List<Future<?>> futures = new ArrayList<>();

    distributeChunksForAllThreads(
        (start, end) -> futures.add(executor.submit(() -> multiplyRowInRange(start, end)))
    );

    Utils.waitForAll(futures);
    Utils.shutdownExecutor(executor);
    showFirstCellIfNeeded();
}

```

Código 4: Implementación paralela con Threads virtuales de la multiplicación de matrices.

Se implementó dos estrategias de lanzamientos de tareas en Threads virtuales (Cod. 4), la primera consta en lanzar una tarea por cada fila y la segunda consta en lanzar una tarea por cada rango (o chunk) de filas, de manera similar al primer algoritmo paralelo mencionado.

Estos Threads se lanzan utilizando la librerías de **Executors** de la forma: **Executors.newVirtualThreadPerTaskExecutor()** y posteriormente con **submit** por cada tarea en Thread virtual wrappeado en la interfaz Future. Una vez lanzadas todas las tareas, se espera a que terminen.

Finalmente se buscó una media de tiempos de 3 intentos para cada algoritmo:

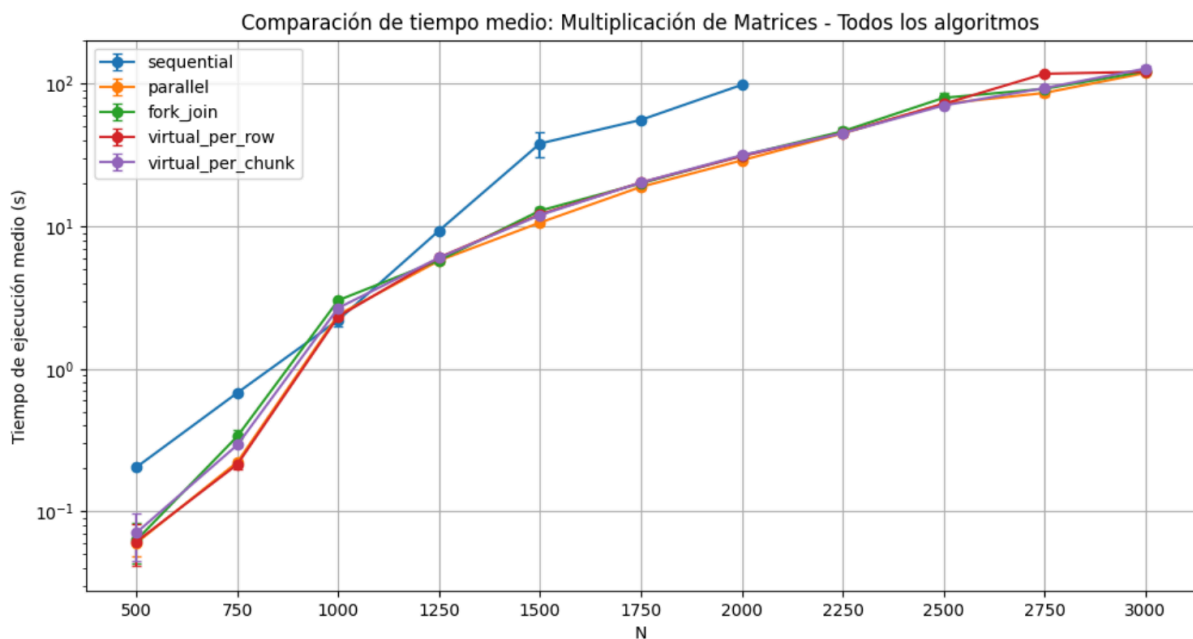


Figura 4: Comparación de tiempos de ejecución de todos los algoritmos para cada N entre 500 y 3000 con step de 250.

Se observa en la Fig. 4 que si bien el secuencial sigue siendo el que más tarda, esto es especialmente notable a partir de $N=1250$. Ninguno de los algoritmos paralelos destaca realmente por sobre los otros. Tal vez no varía mucho porque no hay mucho costo de context switch ni Locks, es decir, no hay mucho overhead por la sincronización ya que se dividen en tareas no relacionadas entre sí. También podría deberse a que se necesita N más grandes para notar alguna diferencia en tiempo entre algoritmos.

NQueens

Explicación del código

En el problema de las N-reinas se tienen un tablero $N \times N$ y se busca distribuir las N reinas en dicho tablero sin que ninguna se esté amenazando entre sí. El código busca contar la cantidad de soluciones para un tablero $N \times N$, mediante un N que recibe por argumento ([Cod. 10](#)). El código es muy similar a la implementación realizada para la [actividad de OpenMP](#).

Tomando un $N=8$ arbitrario, imaginemos un tablero de ajedrez 8×8 .

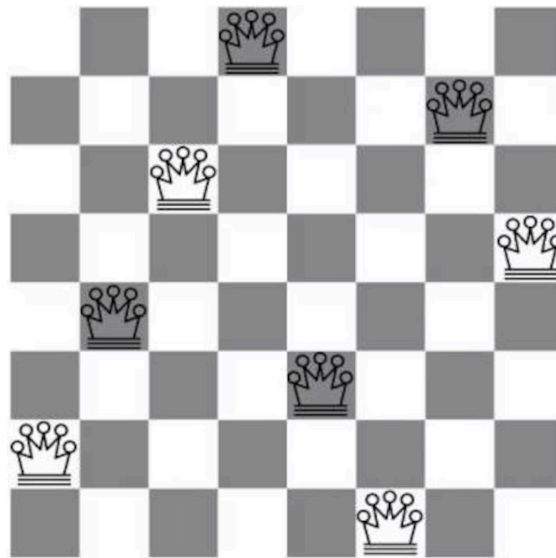


Figura 5: Representación visual del tablero para un problema de las reinas con $N=8$.

La idea del código secuencial es:

- Pararse en la primera fila.
- Ir por cada columna en la primera fila y busco el primer espacio (`row, col`) que pueda colocar una reina sin que se amenace con otra reina, la coloco (se guarda en un arreglo) y paso a la siguiente fila.
- En la siguiente fila voy columna por columna hasta encontrar el primer espacio que pueda color una reina sin conflictos.
- Y así sucesivamente hasta que pueda colocar una reina en la última fila y, de poder, sumo un contador con la cantidad de soluciones.
- En caso de haber completado el tablero o de no poder poner ninguna ficha en toda una fila, vuelvo al stack de la fila anterior y sigo avanzado por las columnas desde donde me quedé la última vez en esa fila anterior y sigo, se va a volver en la pila tanto como sea necesario para poder continuar.
- Finalmente, cuando recorrí todas las posiciones, termina el algoritmo y devuelvo la cantidad de soluciones.

Comparación Secuencial y Paralelo con Threads de procesador

Primero se implementó el paralelismo en **NQueens** mediante un **ExecutorService** utilizando un **ThreadPool** normal. En este, se lanza un **Future** por cada columna de la primera fila en el tablero NxN para que individualmente cuenten la cantidad de soluciones que se desprenden de empezar en esa posición. Esta operación es válida ya que independientemente de dónde coloque la reina en la primera fila, nunca tendrá oposición.

Mientras que para el ForkJoin se buscó implementar un algoritmo recursivo mediante tareas y thresholding.

```
public void solveForkJoin() {
    solutions.set(0);
    ForkJoinPool pool = new ForkJoinPool(numThreads);
    int threshold = 6;
    pool.invoke(new NQueensTask(0, new int[N], threshold));
    Utils.shutdownExecutor(pool);
    showResultIfNeeded();
}

private class NQueensTask extends RecursiveAction {
    private final int row;
    private final int[] board;
    private final int threshold;

    NQueensTask(int row, int[] board, int threshold) {
        this.row = row;
        this.board = board;
        this.threshold = threshold;
    }

    @Override
    protected void compute() {
        if (row == N) {
            solutions.incrementAndGet();
            return;
        }
        if (row >= threshold) {
            solve(row, board);
            return;
        }
        List<NQueensTask> subtasks = new ArrayList<>();
        for (int col = 0; col < N; col++) {
            if (isSafe(board, row, col)) {
                int[] newBoard = board.clone();
                newBoard[row] = col;
                subtasks.add(new NQueensTask(row + 1, newBoard, threshold));
            }
        }
        if (!subtasks.isEmpty()) {
            invokeAll(subtasks);
        }
    }
}
```

Código 5: Implementación ForkJoin para el cálculo de soluciones de N-Queens.

Se extiende en una **inner class** la clase **RecursiveAction** pues no hay ningún valor de retorno, el contador de soluciones se guarda en una variable de instancia atómica de la clase **NQueens** y todas las **RecursiveAction** modifican directamente esa variable. El threshold se definió arbitrariamente como 6, obligando a que cualquier fila de índice mayor o igual ese threshold, tenga que resolverse de

manera secuencial. Se tomó un threshold arbitrario ya que el problema de NQueens [no tiene mucha posibilidad crecer mucho su input de N](#).

Para ambos paralelismos es importante clonar los arreglos `board` para evitar que todos estén modificando la misma referencia y haya variables compartidas con estado inconsistente.

Se probó estos algoritmos paralelos con distintos números de Threads:

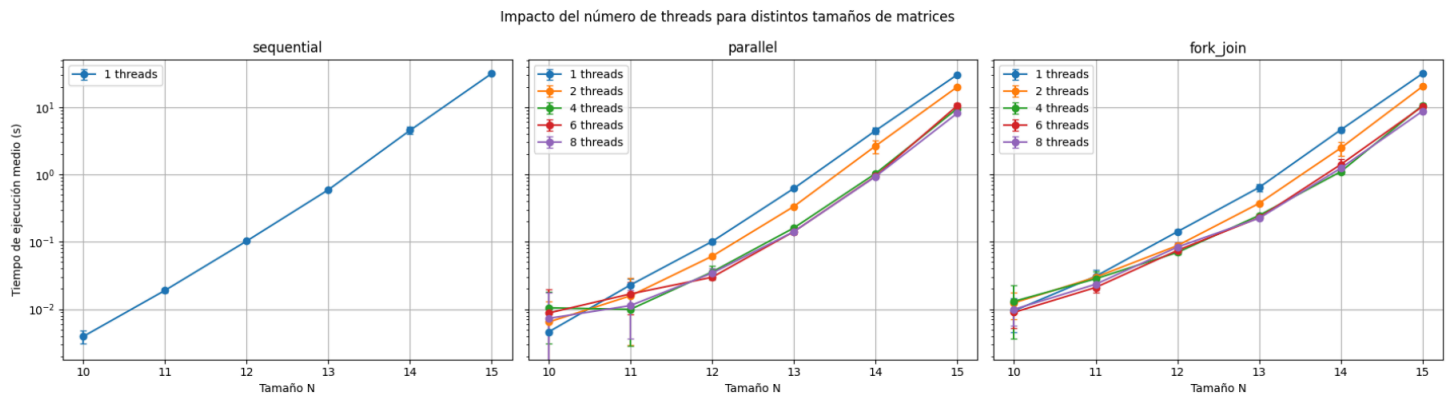


Figura 6: Comparación de tiempos de ejecución para distintos números de Threads para cada N entre 10 y 15.

Donde se observó (Fig. 6) nuevamente que a mayor cantidad de Threads que sea parecida al número de cores, se consigue un resultado en menor tiempo.

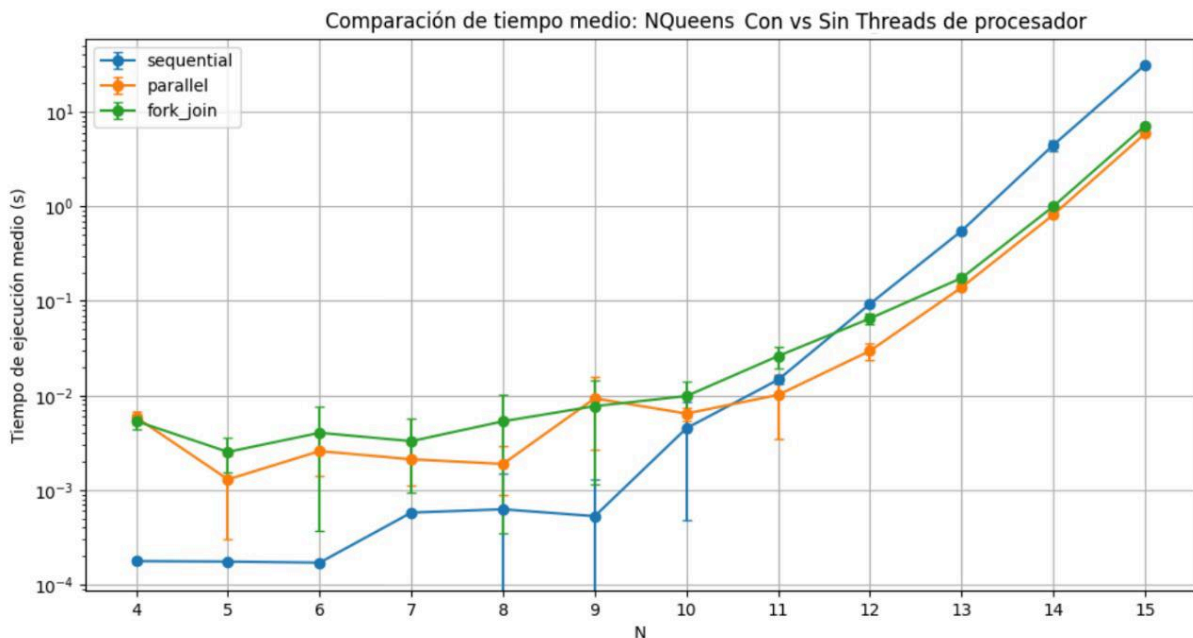


Figura 7: Comparación de tiempos de ejecución Secuencial vs Paralelo para cada N entre 4 y 15.

Luego, se compararon los tiempos de ejecución para un rango de N mucho mayor en un promedio de 3 intentos por cada N (Fig. 7). El algoritmo secuencial rápidamente empieza a tardar mucho, se puede notar que el algoritmo paralelo es

ligeramente más rápido que el algoritmo ForkJoin, esto puede deberse a la sincronización de RecursiveAction que tiene que hacer extra.

Threads virtuales

Finalmente se implementó la misma versión paralela sobre las columnas de la primera fila, lanzando una tarea por cada columna y llamando recursivamente con esa configuración a la siguiente fila, pero utilizando Threads virtuales en lugar de Threads de procesador. Para este problema no se usó chunks ya que el número N de columnas máximo que se puede calcular sin tardar demasiado tiempo, no es muy alto.

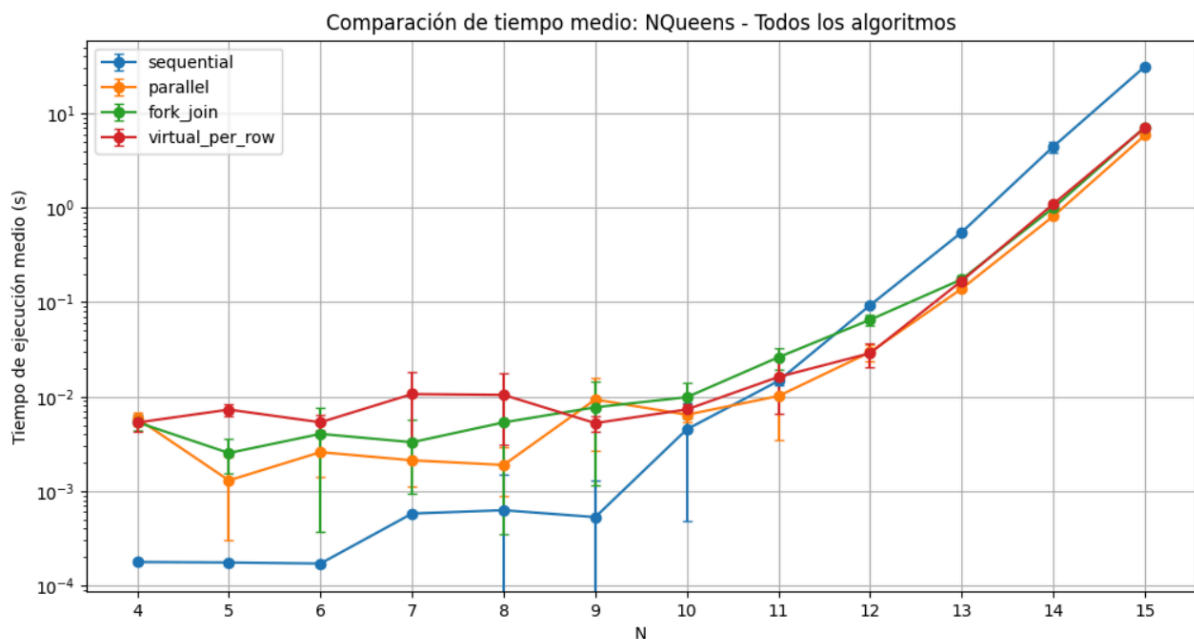


Figura 8: Comparación de tiempos de ejecución de todos los algoritmos para cada N entre 4 y 15.

En los resultados ilustrados en la Fig. 8, se observa que el algoritmo más lento fue el secuencial y el más rápido fue el paralelo, también se observa que el paralelo empieza a ser más rápido cerca de $N = 12$, indicando que ante un posible N mayor podría llegar a notarse más esa distinción entre qué algoritmo es el que consigue un menor tiempo.

Conclusiones

El trabajo permitió evaluar dos problemas clásicos de paralelización y distintas estrategias para dividir tareas. En ambos problemas se buscó siempre una paralelización de tareas que manejen particiones disjuntas. En el primero, se dividió en rangos disjuntos para evitar tener un orden en las tareas. Mientras que en el segundo se lo dividió casos con distinto inicio, haciendo que sus trayectorias no tengan orden y sean independientes. Los resultados particulares muestran que:

- Multiplicación de matrices: la versión secuencial escala rápidamente en tiempo de ejecución a medida que crece N mientras que las versiones paralelas logran reducir el tiempo, en especial al usar un número de threads cercano al número de cores disponibles. No se observaron diferencias significativas entre `ExecutorService` y `ForkJoin`, lo que indica que en un problema altamente `CPU-bound` como este, el beneficio proviene principalmente de la división del trabajo más que del framework de concurrencia elegido. El uso de `Virtual Threads` no ofreció ventajas claras, se estima que es debido a que este tipo de hilos se verá más beneficiado en escenarios con tareas intensivas en `I/O` o con mucha espera, y no en cálculos puramente computacionales.
- Problema de N -reinas: la paralelización mostró mejoras frente a la versión secuencial a partir de valores de N moderadamente grandes (alrededor de 12), donde la complejidad del problema hace que el trabajo distribuido entre threads amortice el overhead de coordinación. El `ExecutorService` se comportó ligeramente mejor que `ForkJoin`, posiblemente debido al costo adicional de las tareas recursivas. Aún así, `ForkJoin` se ejecuta en un buen tiempo para lo que suele costar el nivel de recursión, esto se consigue gracias al `thresholding`.

En términos generales, puede concluirse que la paralelización en Java mejora los tiempos de ejecución en problemas de gran tamaño, siempre que el tamaño de entrada justifique el overhead de gestionar múltiples threads y de sincronizarlos.

Anexo

Tipos de paralelización

```
package common;

public enum ParallelizationType {
    SEQUENTIAL,
    PARALLEL,
    FORK_JOIN,
    VIRTUAL_PER_ROW,
    VIRTUAL_PER_CHUNK
;

    public static ParallelizationType fromString(String string) {
        return ParallelizationType.valueOf(string.toUpperCase());
    }
}
```

Código 6: Enum con todos los tipos de paralelización que se usarán en los dos problemas.

Código útil para ambos problemas

```
package common;

import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class Utils {
    public static void waitForAll(List<Future<?>> futures) {
        for (Future<?> f : futures) {
            try {
                f.get(); // blocks until the task is finished
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            } catch (ExecutionException e) {
                throw new RuntimeException(e.getCause());
            }
        }
    }

    public static void shutdownExecutor(ExecutorService executor) {
        // Note: I don't use try-with-resources because that calls only shutdown() and I want
        // to make sure the platform threads are free between methods using shutdownNow() as last
        resource
        executor.shutdown();
        try {
            if (!executor.awaitTermination(1, TimeUnit.MINUTES)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}
```

Código 7: Código general utilizado en ambos problemas.

Multiplicación de Matrices

MatrixMain

```

package matrix;

import common.ParallelizationType;

import java.time.Duration;
import java.time.Instant;
import java.util.Locale;

public class MatrixMain {
    public static void main(String[] args) {
        final int size = Integer.parseInt(System.getProperty("size"));
        final int seed = 6834723;
        final int numThreads = Integer.parseInt(System.getProperty("numThreads"));
        final int times = Integer.parseInt(System.getProperty("times"));
        final ParallelizationType type = ParallelizationType.fromString(System.getProperty("type"));
        final boolean showFirstCell = Boolean.parseBoolean(System.getProperty("showFirstCell"));

        MatrixMultiplication m = new MatrixMultiplication(size, seed, numThreads, showFirstCell);

        switch (type) {
            case SEQUENTIAL -> runNTimesAndPrint(times, m, m::multiplySequential);
            case PARALLEL -> runNTimesAndPrint(times, m, m::multiplyParallel);
            case FORK_JOIN -> runNTimesAndPrint(times, m, m::multiplyForkJoin);
            case VIRTUAL_PER_ROW -> runNTimesAndPrint(times, m, m::multiplyVirtualThreadsPerRow);
            case VIRTUAL_PER_CHUNK -> runNTimesAndPrint(times, m, m::multiplyVirtualThreadsPerChunk);
        }

        private static void runNTimesAndPrint(int nTimes, MatrixMultiplication m, Runnable multiply) {
            for (int i = 0; i < nTimes; i++) {
                Instant start = Instant.now();
                multiply.run();
                Instant end = Instant.now();
                Duration elapsedTime = Duration.between(start, end);
                Locale.setDefault(Locale.ENGLISH);
                System.out.printf("%.10f\n", elapsedTime.toNanos() / 1000000000.0);
                m.resetResultMatrix();
            }
        }
    }
}

```

Código 8: Clase *MatrixMain* que funcionará como punto de partida para el ejercicio de multiplicación de matrices.

MatrixMultiplication

```

package matrix;

import common.Utils;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.*;
import java.util.function.BiConsumer;

public class MatrixMultiplication {
    private final int size;
    private final int numThreads;
    private final boolean showFirstCell;
    private final double[][] left;
    private final double[][] right;
    private final double[][] result;

    public MatrixMultiplication(int size, int seed, int numThreads, boolean showFirstCell) {
        this.size = size;
        this.numThreads = numThreads;
        this.showFirstCell = showFirstCell;
        left = new double[size][size];
        right = new double[size][size];
        result = new double[size][size];
        Random rand = new Random(seed);
        for (int rowIdx = 0; rowIdx < size; rowIdx++) {
            for (int colIdx = 0; colIdx < size; colIdx++) {
                left[rowIdx][colIdx] = rand.nextDouble();
                right[rowIdx][colIdx] = rand.nextDouble();
                result[rowIdx][colIdx] = 0;
            }
        }
    }
}

```



```

public void multiplySequential() {
    multiplyRowInRange(0, size);

    showFirstCellIfNeeded();
}

public void multiplyParallel() {
    if (numThreads > size) {
        multiplySequential();
        return;
    }

    ExecutorService executor = Executors.newFixedThreadPool(numThreads);
    List<Future<?>> futures = new ArrayList<>();

    distributeChunksForAllThreads(
        (start, end) -> futures.add(executor.submit(() -> multiplyRowInRange(start, end)))
    );

    Utils.waitForAll(futures);
    Utils.shutdownExecutor(executor);
    showFirstCellIfNeeded();
}

public void multiplyForkJoin() {
    ForkJoinPool pool = new ForkJoinPool(numThreads);
    int threshold = Math.max(20, size/numThreads);
    pool.invoke(new MultiplyTask(0, size, threshold));

    Utils.shutdownExecutor(pool);
    showFirstCellIfNeeded();
}

public void multiplyVirtualThreadsPerRow() {
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
    List<Future<?>> futures = new ArrayList<>();

    for (int row = 0; row < size; row++) {
        final int rowIdx = row;
        futures.add(executor.submit(() -> multiplyRowInRange(rowIdx, rowIdx + 1)));
    }

    Utils.waitForAll(futures);
    Utils.shutdownExecutor(executor);
    showFirstCellIfNeeded();
}

public void multiplyVirtualThreadsPerChunk() {
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
    List<Future<?>> futures = new ArrayList<>();

    distributeChunksForAllThreads(
        (start, end) -> futures.add(executor.submit(() -> multiplyRowInRange(start, end)))
    );

    Utils.waitForAll(futures);
    Utils.shutdownExecutor(executor);
    showFirstCellIfNeeded();
}

private class MultiplyTask extends RecursiveAction {
    private final int startRow;
    private final int endRow;
    private final int threshold;

    MultiplyTask(int startRow, int endRow, int threshold) {
        this.startRow = startRow;
        this.endRow = endRow;
        this.threshold = threshold;
    }

    @Override
    protected void compute() {
        if (endRow - startRow <= threshold) {
            multiplyRowInRange(startRow, endRow);
            return;
        }
        int middleRow = (startRow + endRow) / 2;
        invokeAll(new MultiplyTask(startRow, middleRow, threshold), new MultiplyTask(middleRow, endRow,
threshold));
    }
}

// endIndex exclusive
private void multiplyRowInRange(int startIdx, int endIdx) {
    for (int rowIdx = startIdx; rowIdx < endIdx; rowIdx++) {

```

```

        for (int colIdx = 0; colIdx < size; colIdx++) {
            for (int k = 0; k < size; k++) {
                result[rowIdx][colIdx] += left[rowIdx][k] * right[k][colIdx];
            }
        }
    }

    public void resetResultMatrix() {
        for (int rowIdx = 0; rowIdx < size; rowIdx++) {
            for (int colIdx = 0; colIdx < size; colIdx++) {
                result[rowIdx][colIdx] = 0;
            }
        }
    }

    private void distributeChunksForAllThreads(BiConsumer<Integer, Integer> chunkForThreadHandler) {
        int rowsPerThread = size / numThreads;
        int remainder = size % numThreads;

        int start = 0;
        for (int i = 0; i < numThreads; i++) {
            int end = start + rowsPerThread + (i < remainder ? 1 : 0); // handle remainder
            chunkForThreadHandler.accept(start, end);
            start = end;
        }
    }

    private void showFirstCellIfNeeded() {
        if (showFirstCell)
            System.out.printf("The content of [0][0] in the matrix is %.2f\n", result[0][0]);
    }
}

```

Código 9: Clase MatrixMultiplication con los distintos tipos de paralelización implementados.

NQueens

NQueensMain

```

package nqueens;

import common.ParallelizationType;

import java.time.Duration;
import java.time.Instant;
import java.util.Locale;

public class NQueensMain {
    public static void main(String[] args) {
        final int size = Integer.parseInt(System.getProperty("N"));
        final int numThreads = Integer.parseInt(System.getProperty("numThreads"));
        final int times = Integer.parseInt(System.getProperty("times"));
        final ParallelizationType type = ParallelizationType.fromString(System.getProperty("type"));
        final boolean showResult = Boolean.parseBoolean(System.getProperty("showResult"));

        NQueens nQueens = new NQueens(size, numThreads, showResult);

        switch (type) {
            case SEQUENTIAL -> runNTimesAndPrint(times, nQueens::solveSequential);
            case PARALLEL -> runNTimesAndPrint(times, nQueens::solveParallel);
            case FORK_JOIN -> runNTimesAndPrint(times, nQueens::solveForkJoin);
            case VIRTUAL_PER_ROW -> runNTimesAndPrint(times, nQueens::solveVirtualThreadsPerRow);
            case VIRTUAL_PER_CHUNK -> throw new UnsupportedOperationException("This parallelization type is not implemented");
        }
    }

    private static void runNTimesAndPrint(int nTimes, Runnable countSolutions) {
        for (int i = 0; i < nTimes; i++) {
            Instant start = Instant.now();
            countSolutions.run();
            Instant end = Instant.now();
            Duration elapsedTime = Duration.between(start, end);
        }
    }
}

```

```

        Locale.setDefault(Locale.ENGLISH);
        System.out.printf("%.10f\n", elapsedTime.toNanos()/1000000000.0);
    }
}

```

Código 10: Clase NQueensMain que funcionará como punto de partida para el ejercicio de NQueens.

NQueens

```

package nqueens;

import common.Utills;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

public class NQueens {
    private final int N;
    private final int numThreads;
    private final boolean showResult;
    private final AtomicInteger solutions;

    public NQueens(int N, int numThreads, boolean showResult) {
        this.N = N;
        this.numThreads = numThreads;
        this.showResult = showResult;
        solutions = new AtomicInteger(0);
    }

    public void solveSequential() {
        solutions.set(0);
        solve(0, new int[N]);
        showResultIfNeeded();
    }

    public void solveParallel() {
        solutions.set(0);
        ExecutorService executor = Executors.newFixedThreadPool(numThreads);
        List<Future<?>> futures = new ArrayList<>();

        launchFutureInFirstRow(executor, futures);

        Utills.waitForAll(futures);
        Utills.shutdownExecutor(executor);
        showResultIfNeeded();
    }

    public void solveForkJoin() {
        solutions.set(0);
        ForkJoinPool pool = new ForkJoinPool(numThreads);
        int threshold = 6;
        pool.invoke(new NQueensTask(0, new int[N], threshold));
        Utills.shutdownExecutor(pool);
        showResultIfNeeded();
    }

    private class NQueensTask extends RecursiveAction {
        private final int row;
        private final int[] board;
        private final int threshold;

        NQueensTask(int row, int[] board, int threshold) {
            this.row = row;
            this.board = board;
            this.threshold = threshold;
        }

        @Override
        protected void compute() {
            if (row == N) {
                solutions.incrementAndGet();
                return;
            }
            if (row >= threshold) {
                solve(row, board);
                return;
            }
            List<NQueensTask> subtasks = new ArrayList<>();

```

```

        for (int col = 0; col < N; col++) {
            if (isSafe(board, row, col)) {
                int[] newBoard = board.clone();
                newBoard[row] = col;
                subtasks.add(new NQueensTask(row + 1, newBoard, threshold));
            }
        }
        if (!subtasks.isEmpty()) {
            invokeAll(subtasks);
        }
    }
}

public void solveVirtualThreadsPerRow() {
    solutions.set(0);
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
    List<Future<?>> futures = new ArrayList<>();

    launchFutureInFirstRow(executor, futures);

    Utils.waitForAll(futures);
    Utils.shutdownExecutor(executor);
    showResultIfNeeded();
}

private boolean isSafe(int[] board, int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col ||
            board[i] - i == col - row ||
            board[i] + i == col + row)
            return false;
    }
    return true;
}

private void solve(int row, int[] board) {
    if (row == N) {
        solutions.incrementAndGet();
        return;
    }
    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col)) {
            board[row] = col;
            solve(row + 1, board);
        }
    }
}

private void launchFutureInFirstRow(ExecutorService executor, List<Future<?>> futures) {
    for (int col = 0; col < N; col++) {
        final int firstCol = col;
        futures.add(executor.submit(() -> {
            int[] board = new int[N];
            board[0] = firstCol;
            solve(1, board);
        }));
    }
}

private void showResultIfNeeded() {
    if (showResult)
        System.out.printf("Number of solutions %d\n", solutions.get());
}
}

```

Código 11: Clase NQueen con los distintos tipos de paralelización implementados.