Ejercicio 3

Nombre REDACTED

1 Pruebas

Las pruebas realizadas tienen dos objetivos principales: (1) verificar la correctitud de la implementación paralela y (2) evaluar su desempeño frente a la versión secuencial. A continuación se describe el procedimiento seguido en cada etapa.

1. **Verificación de correctitud**

   Antes de evaluar el rendimiento, se debe confirmar que la versión paralela produce exactamente los mismos resultados que la versión secuencial. Para ello, las pruebas ejecutan el siguiente procedimiento:

   - **Generación de matrices de entrada**: Se crean dos matrices A y B de tamaño $N \times N$ con valores aleatorios generados mediante una semilla fija (`Random(314)`) con el fin de garantizar reproducibilidad.
   - **Cálculo del resultado de referencia**: Se utiliza la implementación secuencial `seqMatrixMultiply`, incluida dentro de la clase de prueba, para obtener la matriz `refC`, considerada como el resultado correcto.
   - **Ejecución de la versión paralela**: Se invoca el método `MatrixMultiply.parMatrixMultiply(A, B, C, N)`, obteniendo la salida paralela en la matriz C.
   - **Comparación elemento a elemento**: Mediante la función `checkResult`, cada posición de la matriz C se compara con su valor correspondiente en `refC`. Si existe alguna discrepancia, la prueba falla indicando el índice exacto donde ocurrió.

   Esta etapa garantiza que el paralelismo no introduce errores de sincronización, condiciones de carrera ni diferencias numéricas respecto a la versión secuencial.

2. **Evaluación del desempeño**

   Una vez verificada la correctitud, las pruebas se enfocan en medir el rendimiento comparando los tiempos de ejecución de ambas versiones. El procedimiento es el siguiente:

   (a) **Medición del tiempo secuencial**: La función `seqMatrixMultiply` se ejecuta `REPEATS` veces, obteniendo un tiempo promedio estable.

   (b) **Medición del tiempo paralelo**: De manera análoga, `MatrixMultiply.parMatrixMultiply` se ejecuta también `REPEATS` veces y se obtiene su tiempo promedio.

   (c) **Cálculo del speedup**

   (d) **Validación del speedup mínimo esperado**: El número de núcleos disponibles se obtiene mediante `Runtime.getRuntime().availableProcessors()`. Para que una prueba sea considerada satisfactoria, el speedup debe ser al menos:

   $$0.6 \times (\text{número de núcleos}).$$

   Este umbral asegura que la implementación paralela aprovecha efectivamente el hardware multicore.

3. **Pruebas con tamaños grandes (512×512 y 768×768)**

   Se realizan dos pruebas principales:

   - `testPar512_x_512()`
   - `testPar768_x_768()`

   El uso de matrices grandes es fundamental para evaluar el paralelismo real. En problemas pequeños, el costo del overhead del paralelismo (creación de hilos, sincronización, acceso compartido a memoria) puede ser comparable o incluso superior al trabajo computacional útil, produciendo speedups bajos o negativos. En cambio, con matrices grandes, la computación domina sobre el overhead, permitiendo observar mejoras significativas en el tiempo de ejecución.

2 Código

La multiplicación matricial presenta una propiedad fundamental: **cada fila de la matriz resultado puede calcularse de manera completamente independiente de las demás**.

Recordemos la definición del producto matricial:

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] \cdot B[k][j].$$

Dado que el cálculo de la fila i únicamente depende de la fila i de A y de todas las columnas de B, es posible asignar distintas filas a diferentes hilos sin riesgo de condiciones de carrera. A partir de esta observación, se desarrolla la implementación presentada a continuación:

```
public static void parMatrixMultiply(final double[][] A, final double[][] B,
                                      final double[][] C, final int N) {
    forallChunked(0, N - 1, i -> {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    });
}
```

En esta versión paralela se emplea la función `forallChunked` de la librería **PCDP**. Dicha función divide el rango de iteración $[0, N-1]$ en *chunks* o bloques, que posteriormente son distribuidos entre los hilos disponibles. Cada hilo procesa uno o varios bloques ejecutando la función lambda proporcionada.

La elección de `forallChunked` frente a `forall` responde a razones de eficiencia:

- reduce el **overhead** asociado a manejar un número excesivo de tareas pequeñas,
- mejora el **balance de carga** entre hilos,
- y permite alcanzar una **escalabilidad más cercana al ideal lineal**.

Dentro de cada bloque, el primer bucle recorre las columnas j de la fila asignada, mientras que el segundo bucle realiza la suma acumulada para obtener el valor de $C[i][j]$.

De este modo, cada hilo construye de forma autónoma las filas que le corresponden, sin interferir con los demás, aprovechando plenamente la independencia de las operaciones por fila que caracteriza al producto matricial.

3 Análisis de resultados

Dado que los tests que fallan son `testPar512_x_512` y `testPar768_x_768`, ambos asociados exclusivamente al desempeño y no a la correctitud, se concluye que la implementación paralela genera resultados matemáticamente correctos, pero no alcanza la mejora de rapidez exigida por el enunciado. En las pruebas, ambos tiempos se estiman ejecutando cada versión 20 veces, lo que permite reducir el ruido introducido por el sistema operativo o el entorno de ejecución.

Los tests `testPar512_x_512` y `testPar768_x_768` establecen como criterio de éxito que el speedup cumpla:

$$\text{speedup} \;\geq\; 0.6 \times \text{ncores},$$

El fallo de ambas pruebas indica que, aunque la ejecución paralela es correcta, el speedup obtenido es *menor* que este umbral. Esto refleja que la implementación paralela no está aprovechando de forma suficientemente eficiente el paralelismo disponible en el hardware.

Este comportamiento puede deberse a varias razones:

- **Overhead del paralelismo.** El uso de `forallChunked` introduce cierto coste asociado a la creación, planificación y sincronización de tareas. Aunque este overhead se amortiza en problemas grandes, puede seguir siendo apreciable y reducir el rendimiento efectivo, especialmente si el entorno limita el número de hilos ejecutables simultáneamente.

- **Granularidad no óptima.** Aunque `forallChunked` reparte filas en bloques, el tamaño de dichos bloques puede no ser el más adecuado para la arquitectura subyacente. Chunks demasiado pequeños aumentan el overhead; chunks demasiado grandes pueden generar desbalance de carga entre hilos.
- **Limitaciones del entorno de ejecución.** El valor reportado por `availableProcessors()` puede sobreestimar el número efectivo de núcleos asignados a la máquina virtual donde corre el corrector automático. Si los recursos están compartidos o se ejecutan varias pruebas simultáneamente, el criterio speedup $\geq 0.6 \times$ ncores puede ser difícil de alcanzar.
- **Restricciones de memoria y jerarquía de caché.** La multiplicación de matrices es intensiva en memoria. Aunque la computación esté paralelizada, el ancho de banda de memoria y los niveles de caché pueden convertirse en un cuello de botella que limite la escalabilidad. A partir de cierto número de hilos, el acceso concurrente a memoria reduce el speedup alcanzable.

En resumen, los resultados experimentales muestran que la versión paralela es completamente correcta, pero que el speedup obtenido para matrices de tamaño 512×512 y 768×768 no alcanza el 60% del número de núcleos disponibles. Por ello, no se cumple el criterio de *escalabilidad casi lineal* requerido por las pruebas automáticas. Para mejorar estos resultados, sería necesario ajustar la estrategia de paralelización (por ejemplo, explorando `forall2dChunked` o diferentes granularidades) o incorporar optimizaciones adicionales que reduzcan el overhead y mejoren la eficiencia en el uso de la memoria.