ParallelFlix — Documentación ampliada y práctica

Resumen rápido

ParallelFlix es una simulación en consola de un motor de recomendaciones estilo Netflix construido en C# (.NET 8). Está pensado para **mostrar técnicas reales de paralelismo** (task-level y data-level) y la técnica de **descomposición especulativa** (lanzar varias estrategias y usar la que responda primero). El proyecto incluye el dataset (100 ítems), tres recomendadores, un motor que coordina la ejecución, una UI de consola con las interacciones esperadas (reproducir 10s, Mi Lista, búsqueda) y utilidades para medir rendimiento. Todo se ejecuta en memoria (sesión volátil).

Estructura del proyecto

```
ParallelFlix/
└─src/ParallelFlix/
|   ├─ ParallelFlix.csproj
|   ├─ Program.cs
|   ├─ Datos/Movies.json
|   ├─ Modelos/
|   ├─ Nucleo/
|   ├─ Recomendadores/
|   ├─ Servicios/
|   └─ Utilidades/
├─ test/
└─ metrics/
```

¿Por qué esta separación?

- Mantener **Modelos** puros facilita tests y cambios de formato.
- **Núcleo** centraliza lógica común (similitud) evitando duplicar código en los recomendadores.
- **Recomendadores** son intercambiables; se pueden añadir o quitar sin tocar motor o UI.
- **Servicios** contienen la orquestación y IO, lo que hace la UI simple y el motor testable.
- **Utilidades** agrupan pequeños helpers que limpiarán la UI.

Descripción detallada — archivo por archivo

ParallelFlix.csproj

- **Qué hace:** define el proyecto .NET (output, target framework net8.0) y copia `Datos/Movies.json` al binario.
- **Por qué importa:** sin esto, el ejecutable no encontraría el JSON cuando se publique o ejecute desde otra carpeta.

Program.cs

- **Qué hace:** solicita el nombre de usuario, carga el dataset (con `ConjuntoDatos.CargarPeliculas`), crea `SesionUsuario`, instancia `MotorRecomendacion` e `InterfazUsuario` y ejecuta la UI.
- **Valor práctico:** punto único para iniciar la app; fácil de modificar para tests (aquí podrías aceptar argumentos para modo bench o modo demo).

Datos/Movies.json

- **Qué contiene:** 100 objetos con campos `Id`, `Titulo`, `Genero`, `Etiquetas`, `Calificacion`, `Duracion`, `Ano`, `Director`.
- **Uso en el proyecto:** se parsea a `List<Pelicula>` y alimenta el motor. Se usa para listas iniciales, búsqueda, y para calcular similitudes.
- **Consejo:** si quieres probar variaciones, crea otra versión (por ejemplo `Movies_small.json`) y modifica `Program.cs` para apuntar a ella.

Modelos/

Pelicula.cs

- **Descripción:** `record` que modela la película con sus metadatos.
- **Por qué `record`?:** inmutabilidad por defecto (buena práctica cuando los objetos son datos) y sintaxis compacta.

Recomendacion.cs

- **Descripción:** envuelve una `Pelicula` con un `Puntuacion` (double) para ranking.
- **Uso:** las estrategias devuelven `List<Recomendacion>`; el motor las fusiona.

PerfilUsuario.cs

- **Descripción:** contiene nombre, géneros/etiquetas preferidas, `IdsVistos` y `IdsMiLista`.
- **Decisión de diseño:** mantener listas/sets en memoria y simples. La UI escribe solo desde el hilo principal; los recomendadores solo leen.

Nucleo/Similitud.cs

- **Qué hace:** calcula similitud entre dos películas con una combinación simple: 60% género (comparación simple) + 40% Jaccard de etiquetas. Además mantiene `_cacheSim: ConcurrentDictionary<(int,int),double>`.
- **Por qué:** calcular similitudes repetidas puede ser costoso; la caché evita work duplicado cuando varias estrategias o hilos consultan los mismos pares.
- **Puntos prácticos:** `GetOrAdd` es atómico y seguro para escenarios concurrentes.

Recomendadores/

IRecomendador.cs

- **Qué define:** contrato `Task<List<Recomendacion>> RecomendarAsync(PerfilUsuario usuario, List<Pelicula> corpus, int k, CancellationToken ct)` y `string Nombre`.
- **Ventaja:** permite añadir nuevas heurísticas sin cambiar el motor.

RecomendadorContenido.cs

- **Cómo funciona:** puntúa candidatos por coincidencia de género (boost), calificación, similitud con el último visto y coincidencia de etiquetas preferidas.
- **Paralelismo:** `Parallel.ForEach` con `ConcurrentBag` para acumular resultados.
- **Uso práctico:** es la estrategia que se enfoca en las propiedades del contenido.

RecomendadorColaborativo.cs

- **Cómo funciona (simulado):** genera "votos" de usuarios sintéticos basados en `IdsVistos` del usuario actual y suma puntajes; no usa una base real de usuarios para no complicar el scope.
- **Paralelismo:** `Parallel.ForEach` y `ConcurrentBag`.
- **Nota:** es una versión simplificada — para producción se sustituiría por matrix factorization o nearest neighbors.

RecomendadorTendencias.cs

- **Cómo funciona:** mezcla `Calificacion` con ruido aleatorio para simular tendencia global.
- **Uso:** aporta diversidad y permite que el motor tenga opciones rápidas (útil para la descomposición especulativa).

Servicios/

ConjuntoDatos.cs

- **Qué hace:** lee `Movies.json` con `System.Text.Json` y devuelve `List<Pelicula>`.
- **Consejo:** aquí puedes añadir validación del schema o carga incremental.

MotorRecomendacion.cs

- **Corazón del paralelismo especulativo.**
- **Flujo:** crea `CancellationTokenSource`, lanza cada `IRecomendador` con un wrapper que mide tiempo; espera `Task.WhenAny`; cancela remainder; recoge resultados completados y fusiona top-k por puntuación máxima por id.
- **Por qué así:** reduce latencia (tomar la estrategia más rápida) y aprovecha resultados parciales de las otras si terminan a tiempo.
- **Cuidado:** consumirás CPU extra por las estrategias que arrancaste y cancelaste — aceptable en prototipo, hay que medir en producción.

Metricas.cs

- **Qué registra:** tiempos por estrategia (`Dictionary<string, TimeSpan>`), tiempo total acumulado y throughput.
- **Uso práctico:** alimenta experimentos de benchmarking (calcular speedup, eficiencia, percentiles).

SimuladorReproduccion.cs

- **Qué hace:** simula 10 segundos de reproducción con barra de progreso (sleep 250ms en bucle). También es donde registramos la película como vista.
- **Por qué:** requisito explícito del proyecto y útil para probar la lógica posterior (mostrar similares, actualizar `IdsVistos`).

SesionUsuario.cs

- **Qué guarda:** el objeto `PerfilUsuario` y una lista rápida `Vistos` para la sesión.
- **Diseño:** sesión en memoria y volátil por requisito: todo se borra al cerrar la app.

InterfazUsuario.cs

- **Qué hace:** menú principal, impresión de recomendaciones, manejo de detalle (R/P/A), búsqueda, Mi Lista, y el ciclo principal de la app.
- **Valor práctico:** aquí vive toda la UX — si quieres cambiar textos, órdenes, o agregar atajos, edita este archivo.

Utilidades/

TemaConsola.cs

- **Qué hace:** funciones para dibujar líneas y títulos en consola para que la UI luzca ordenada.
- **Por qué:** separar presentación para no ensuciar el flujo.

AyudanteEntrada.cs

- **Funciones:** `LeerInt`, `LeerNoVacio` con validaciones.
- **Por qué:** evitar duplicar validaciones de input por todo el código.

tests/

Aquí se concentran las pruebas automáticas realizadas con xUnit para asegurar que el sistema funcione como se espera. Se crearon diferentes archivos de pruebas unitarias y de integración.

metrics/

Esta carpeta contiene los **resultados de las mediciones de rendimiento** obtenidas al ejecutar el sistema en modo secuencial y paralelo. Se almacenan reportes, tablas y gráficas que muestran cómo el uso de paralelismo mejora la eficiencia.

Flujo de ejecución (qué ocurre cuando el usuario corre `dotnet run`)

1. `Program.cs` pide el usuario y carga `Movies.json`.
2. Crea `SesionUsuario`, `MotorRecomendacion`, `InterfazUsuario`.
3. `InterfazUsuario.EjecutarAsync()` entra al loop principal:
    - Pide recomendaciones: llama `_motor.RecomendarEspeculativoAsync(...)`.
    - El motor lanza las 3 estrategias y espera la primera; fusiona resultados y devuelve top-10.
    - La UI imprime las 10 y espera input del usuario (selección, Mi Lista, Buscar o Salir).
    - Si el usuario ve/da play a una película: `SimuladorReproduccion` corre 10 segundos; la película se marca en `IdsVistos` y se muestran 10 similares.
    - Si agrega a Mi Lista, se añade el id a `IdsMiLista`.
4. Cuando el usuario presiona `S`, el programa termina y la sesión desaparece (no hay persistencia).

Problemas reales que enfrentamos y cómo los resolvimos

1) Condiciones de carrera al principio

Qué pasó: varias estrategias leían y escribían estructuras compartidas que inicialmente no eran seguras en hilos, por ejemplo una memoización casera con `Dictionary` y checks de `ContainsKey` + `Add`. **Efecto:** excepción `InvalidOperationException` o datos corruptos (dos hilos intentaron `Add` de la misma clave). **Solución:** cambiamos a `ConcurrentDictionary<(int,int), double>` y usamos `GetOrAdd`. También reemplazamos listas compartidas por `ConcurrentBag` cuando hacían push desde múltiples hilos. **Lección:** evitar `Dictionary`/`List` sin sincronización cuando múltiples hilos escriben.

2) Paralelizar demasiado (oversubscription)

Qué pasó: corrimos `Parallel.ForEach` en cada recomendador sin limitar `MaxDegreeOfParallelism` y además ejecutamos múltiples estrategias en paralelo; en CPUs con pocos núcleos el rendimiento empeoró. **Efecto:** tiempos mayores por context switching y pérdida de CPU por thrashing. **Solución:** añadimos `ParallelOptions` con posibilidad de configurar `MaxDegreeOfParallelism` (por defecto `Environment.ProcessorCount`) y documentamos la necesidad de calibrarlo según la máquina. **Lección:** paralelismo sin control puede empeorar rendimiento.

3) Overhead de la descomposición especulativa

Qué pasó: al lanzar 3 estrategias costosas, el ahorro de latencia en algunos escenarios fue nulo porque las otras estrategias consumían recursos y generaban overhead de cancelación. **Efecto:** en datasets pequeños la versión especulativa era más lenta que ejecutar una sola estrategia bien optimizada. **Solución:** implementamos la posibilidad de desactivar la ejecución especulativa para tests y sugerimos en la documentación probar ambos modos (y medir `T1` real). También agregamos `CancellationToken` para minimizar el trabajo que siguen haciendo las tareas canceladas. **Lección:** la descomposición especulativa es útil para latencia, pero tiene coste — medir y elegir cuándo usarla.

4) Sesgo y ruido en recomendaciones

Qué pasó: el recomendador colaborativo simulado y el de tendencias introducían ruido que, con ciertos seeds, producía resultados poco coherentes con el historial. **Efecto:** usuarios veían recomendaciones poco relevantes en algunas ejecuciones. **Solución:** ajustamos los pesos (p. ej., el `baseScore`, multiplicadores) y el seed para `Random` para estabilizar resultados durante la sesión. Recomendamos reemplazar el colaborativo simulado por una implementación real si se desea calidad. **Lección:** en sistemas recomendadores la calibración de parámetros impacta directamente la percepción de calidad.

5) Presión del GC y objetos temporales

Qué pasó: al trabajar con corpus grande y crear muchas `Recomendacion` temporales en `ConcurrentBag`, vimos aumentos en GC en pruebas intensivas. **Efecto:** latencia de respuesta aumentó y eficiencia cayó. **Solución:** pre-seed pools (si se necesitara), reusar objetos o reducir allocations temporales; también sugerimos medir GC con `dotnet-counters` y reducir creación excesiva de objetos. **Lección:** en .NET, al paralelizar, las allocations pueden convertirse en cuello de botella.

Consejos prácticos para el equipo (qué hacer si continuamos el proyecto)

1. **Instrumentar la caché de similitud** con contadores de hits/misses para justificar su uso.
2. **Agregar pruebas automatizadas** para `Similitud.JaccardEtiquetas` y para `MotorRecomendacion` (mockear recomendadores rápidos/lentos).
3. **Añadir un modo `bench` en `Program.cs`** que corra el motor N veces, varíe `MaxDegreeOfParallelism` y exporte CSV para análisi.
4. **Reemplazar el colaborativo simulado** por una implementación real si el objetivo cambia de demostración a prototipo funcional.
5. **Controlar degree-of-parallelism**: exponer variable de entorno o argumento de línea de comandos para `ParallelOptions.MaxDegreeOfParallelism`.
6. **Agregar registro (logging)**: usar `ILogger` para rastrear inicio/fin de estrategias (útil para debugging y mediciones reales).