NOMBRES

Líder - Rafael Alberto Pérez Aybar

Enyel Leonardo Medrano García

Luis Elier Pujols Rosa

Joel De Jesús Oseliz Reynoso

MATRÍCULAS

2023-1241

2023-1215

2023-1667

2023-1132

CARRERA

Desarrollo de Software

MATERIA

Programación Paralela

NOMBRE DEL DOCENTE

Erick Leonardo Pérez Veloz

TEMA

Proyecto final

FECHA

22/08/2025

Introducción

Este proyecto desarrolla un sistema de recomendaciones para una plataforma de streaming implementado en C#. El núcleo es un motor de recomendación especulativo que ejecuta múltiples estrategias en paralelo (basado en contenido, colaborativo y tendencias), selecciona la respuesta más rápida, y fusiona los resultados disponibles para mejorar la calidad sin aumentar la latencia.

El sistema incluye una interfaz de consola que permite al usuario visualizar su top-k de recomendaciones, explorar detalles de cada título, simular la reproducción, gestionar "Mi Lista" y buscar por título o género. Para asegurar estabilidad y buen rendimiento, se emplean estructuras concurrentes y cancelación cooperativa, además de una capa de métricas que mide tiempos por estrategia, latencia al primer resultado y throughput.

¿El por qué de este proyecto?

La recomendación personalizada reduce el tiempo de decisión y mejora la retención en catálogos extensos. Este proyecto, además de su valor práctico, funciona como banco de pruebas académico-técnico: aplica programación paralela y patrones de concurrencia modernos (Tasks, cancelación y estructuras thread-safe), equilibra precisión y latencia mediante paralelismo especulativo y fusión de resultados, integra métricas operativas (latencia y throughput) para medir el impacto real en la experiencia, y ofrece una arquitectura extensible que admite nuevas estrategias, persistencia y calibración de pesos sin reescribir el núcleo.

Objetivo general

Diseñar e implementar un sistema de recomendación paralelo que entregue resultados relevantes y de baja latencia, combinando varias estrategias en un motor especulativo medible y extensible.

Objetivos específicos

1 - Implementar un motor especulativo que lance en paralelo al menos tres estrategias de recomendación y seleccione la primera en completar, cancelando el resto de manera cooperativa.

2 - Diseñar las estrategias:

- Contenido (afinidad por género/etiquetas y similitud por Jaccard con el último visto).
- Colaborativo (refuerzo por géneros compartidos y señales de co-visualización simplificadas).
- Tendencias (ranking por calificación con ruido controlado para diversidad).

3 - Evitar duplicados y fusionar resultados por identificador, conservando la mejor puntuación y recortando al top-k.

4 - Garantizar seguridad de concurrencia con colecciones thread-safe y caché de similitud concurrente para minimizar recomputo.

5 - Incorporar cancelación cooperativa mediante CancellationToken en los bucles paralelos y tareas asíncronas.

6 - Construir una interfaz de consola que permita: ver recomendaciones, abrir detalles, simular reproducción, gestionar "Mi Lista" y buscar por título o género con validaciones robustas.

7 - Instrumentar métricas para medir tiempos por estrategia, latencia al primer resultado y throughput, facilitando la comparación de enfoques.

8 - Documentar el diseño, supuestos y posibles mejoras (p. ej., ajuste de pesos, índices por Id, aleatoriedad segura por hilo, persistencia en JSON/SQLite) para futuras iteraciones.

Descripción del Problema - Contexto del problema

En plataformas con catálogos extensos como películas o series, los usuarios suelen enfrentar sobrecarga de opciones: tardan en decidir, abandonan la navegación o terminan eligiendo contenidos poco relevantes. Por eso, un sistema de recomendación debe personalizar según gustos (géneros, etiquetas, historial), responder con baja latencia para que la experiencia sea fluida, actualizarse al instante con nuevas señales como lo último visto, búsquedas o "Mi Lista", y escalar conforme crece tanto el catálogo como la cantidad de usuarios concurrentes. Los retos principales son equilibrar relevancia y costo de cómputo, manejar señales heterogéneas como calificación, similitud de contenido y patrones colaborativos, adaptarse a la constante variación de tendencias y preferencias, y controlar la latencia de cola, ya que una sola estrategia lenta puede elevar el tiempo total de respuesta.

Aplicación del problema en un escenario real

En una plataforma de streaming, cada interacción del usuario —abrir la app o la pantalla de inicio, terminar un título, buscar por género o nombre, o gestionar "Mi Lista"— requiere que el sistema genere un top-k de recomendaciones actualizado. Para lograrlo, se combinan varias señales:

- Basado en contenido: afinidad por géneros y etiquetas del usuario, más similitud con lo último que vio.
- Colaborativo: lo que consumen usuarios con patrones similares y refuerzos por co-visualización.
- Tendencias/popularidad: lo más visto o mejor valorado, aportando diversidad y reduciendo riesgo.

En el proyecto en versión consola, esto se refleja en tres componentes:

1. Motor de recomendación, que procesa catálogo y perfil del usuario para devolver un top-k.

2. Interfaz de usuario, que permite mostrar recomendaciones, ver detalles, simular reproducción, buscar y gestionar "Mi Lista".

3. Métricas de rendimiento, que registran tiempos por estrategia y throughput para evaluar el balance entre calidad y rapidez.

Sin una arquitectura diseñada para estos flujos, la aplicación presentaría problemas clave: demoras en cargar la pantalla inicial, incapacidad para reaccionar a nuevas señales (recomendaciones desactualizadas) y poca escalabilidad a medida que crece la base de usuarios y el catálogo.

Importancia del paralelismo en la solución

El paralelismo permite que el sistema de recomendaciones logre a la vez rapidez y calidad. Al ejecutar varias estrategias en paralelo, usamos paralelismo especulativo: se entrega la primera que responde (reduciendo la latencia percibida), y si otras terminan casi al mismo tiempo se fusionan los resultados para mejorar la precisión sin retrasar la interfaz. Con Parallel.ForEach y Tasks, el cálculo de puntuaciones se distribuye entre núcleos, aprovechando al máximo el hardware y acelerando el procesamiento frente a una ejecución secuencial. Además, se evita el impacto de colas lentas: si una estrategia se atasca por datos atípicos o cachés frías, no bloquea la respuesta, ya que la especulación cancela las tareas restantes y libera recursos. La fusión por Id permite sumar parcialmente señales de otras estrategias sin perder velocidad, equilibrando relevancia y rapidez. Finalmente, la cancelación cooperativa reduce trabajo innecesario y aumenta el throughput, escalando mejor con el mismo nivel de infraestructura.

Cumplimiento de los Requisitos del Proyecto

1. **Ejecución Simultánea de múltiples tareas.**
   ParallelFlix ejecuta varias tareas de forma concurrente: el motor lanza las tres estrategias de recomendación en paralelo y espera la primera respuesta (Task.WhenAny), mientras que cada estrategia procesa su conjunto de candidatos en paralelo con Parallel.ForEach; este diseño combina paralelismo a nivel de tareas y a nivel de datos para reducir latencia y aprovechar todos los núcleos disponibles (implementación en Servicios/MotorRecomendacion.cs y Recomendadores/*).

2. **Necesidad de compartir datos entre tareas.**
   Las estrategias comparten cálculos y resultados usando estructuras seguras para hilos: una caché concurrente (ConcurrentDictionary) almacena similitudes entre pares de películas para evitar recomputaciones costosas y ConcurrentBag recoge resultados parciales; esto permite que múltiples tareas accedan y reutilicen datos comunes sin condiciones de carrera ni bloqueos manuales (ver Nucleo/Similitud.cs y usos en los recomendadores).

3. **Exploración de diferentes estrategias de paralelización.**
   El proyecto permite experimentar con distintas estrategias: task-level (ejecutar varias heurísticas en paralelo y escoger o fusionar la mejor) y data-level (paralelizar el scoring de ítems internamente), pudiendo activarlas por separado o conjuntamente; ese doble nivel facilita comparar modos (solo tareas, solo datos, ambos) y ajustar el grado de paralelismo (ParallelOptions.MaxDegreeOfParallelism) para estudiar comportamiento y trade-offs.

4. **Escalabilidad con más recursos.**
   El diseño escala verticalmente al aprovechar TPL y Parallel para utilizar más núcleos y reducir tiempos; además, su arquitectura modular facilita la escala horizontal: el corpus puede particionarse y distribuirse entre nodos (map/merge), o migrarse a frameworks como Spark para procesamiento masivo, por lo que la misma lógica barre bien en máquinas con más CPU o en un clúster distribuido (Recomendadores/* y MotorRecomendacion).

5. **Métricas de evaluación del rendimiento.**
   ParallelFlix incorpora medición explícita: tiempos por estrategia mediante Stopwatch, tiempo total y cálculo de throughput (resultados/segundo) en la clase Metricas; estos datos permiten calcular speedup y eficiencia al variar núcleos o configuración y sirven para análisis de percentiles de latencia (P50/P90) y comparación de modos de paralelización (ver Servicios/Metricas.cs y la recolección en el motor).

6. **Aplicación a un problema del mundo real.**
   Aunque es una simulación, la arquitectura reproduce patrones reales de motores de recomendación —combinación de señales, caché compartida, paralelismo y descomposición especulativa para bajar latencia— y puede extenderse con persistencia, matrices reales de usuario-ítem o despliegue distribuido para convertirse en un prototipo aplicable a plataformas de streaming y otros sistemas que requieran recomendaciones rápidas y escalables.

Diseño de la Solución - Arquitectura general del sistema

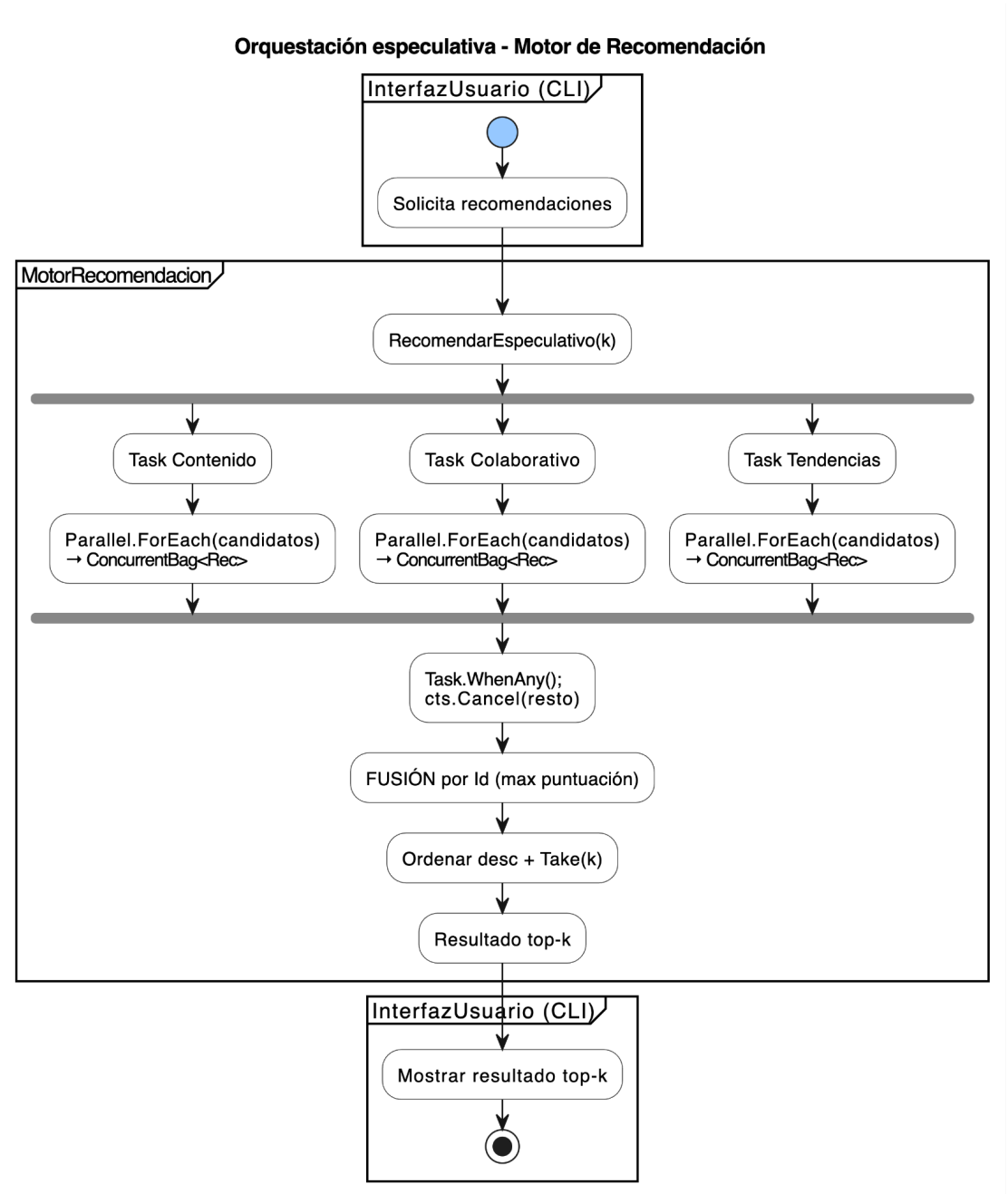
Capas / módulos principales

- Datos: ConjuntoDatos (carga JSON → List<Pelicula>).
- Dominio: PerfilUsuario, SesionUsuario (historial, "Mi Lista", preferencias).
- Estrategias de recomendación (implementan IRecomendador):
  RecomendadorContenido (géneros/etiquetas + similitud al último visto).
  RecomendadorColaborativo (refuerzo por géneros de vistos + aleatoriedad controlada).
  RecomendadorTendencias (calificación + ruido para diversidad).
- Orquestador: MotorRecomendacion (paralelismo especulativo entre estrategias + fusión de resultados).
- Utilidades:
  Similitud (Jaccard + caché concurrente).
  Metricas (Stopwatch, tiempos, throughput).
  TemaConsola, AyudanteEntrada (UI/validación consola).
  SimuladorReproduccion (mock de reproducción 10s).

Flujo

1. La UI pide top-k al Motor.
2. El Motor lanza en paralelo las estrategias y toma la primera que termina (reduce latencia).
3. Cancela las restantes; si alguna alcanzó a completar, fusiona por Id (máxima puntuación).
4. Devuelve top-k a la UI; el usuario explora, reproduce (simulado), busca, gestiona "Mi Lista".

Diagrama de componentes / tareas paralelas



La estrategia de paralelización

Por un lado está el **paralelismo especulativo**, donde el motor lanza de forma simultánea las estrategias de Contenido, Colaborativo y Tendencias. La primera en terminar se usa para responder rápido y, al mismo tiempo, se cancelan las demás para no desperdiciar recursos. Si alguna alcanza a finalizar casi en paralelo, sus resultados se fusionan con la estrategia ganadora, agrupando por Id y conservando la mejor puntuación. Así se mantiene la calidad sin añadir latencia percibida.

Por otro lado, cada estrategia aplica **paralelismo de datos**. Esto significa que el cálculo de las puntuaciones sobre el catálogo se distribuye entre los núcleos del procesador con `Parallel.ForEach`, almacenando resultados en estructuras seguras para hilos (`ConcurrentBag`) y usando cancelación cooperativa con `CancellationToken`. Una vez procesados los candidatos, los resultados se ordenan y se toma el top-k.

El diseño responde a necesidades claras: la latencia se reduce porque siempre se entrega el primer resultado disponible, el CPU se aprovecha mejor al repartir cargas, y el sistema se vuelve robusto frente a estrategias lentas que de otro modo bloquean la respuesta. Además, la fusión de resultados permite enriquecer la recomendación final con señales de más de una estrategia, siempre que estas se completen a tiempo.

Los **trade-offs** principales son un mayor uso temporal de CPU al correr varias estrategias a la vez y ciertos riesgos técnicos como la aleatoriedad en paralelo o el consumo de memoria en cachés de similitud. Estos efectos se mitigan con cancelación temprana, uso controlado de generadores de números aleatorios por hilo, pre-indexación del catálogo y límites en las cachés.

Herramientas y tecnologías empleadas

- C# / .NET (proyecto consola).

- TPL (Task Parallel Library):Task, Task.WhenAny, CancellationToken / CancellationTokenSource. Parallel.ForEach con ParallelOptions.

- Colecciones concurrentes: ConcurrentBag<T> (acumulación de recomendaciones por múltiples hilos). ConcurrentDictionary<TKey,TValue> (caché de similitud con (minId,maxId) como clave).

- Medición de rendimiento: System.Diagnostics.Stopwatch (tiempos por estrategia, latencia). Módulo Metricas (acumulación de tiempos, throughput).

- Serialización: System.Text.Json (carga de catálogo desde JSON, tolerante a mayúsc./minúsc.).

- UI de consola / helpers: TemaConsola (títulos, separadores, etiqueta-valor), AyudanteEntrada (validación robusta), SimuladorReproduccion (barra de progreso 10s).

5. Implementación Técnica

- Descripción de la estructura del proyecto

```
ParallelFlix/
│   ParallelFlix.sln
│   parallelflix.db
│   README.md
│
├──docs/                # Documentación del proyecto
├──metrics/             # Gráficas y resultados de métricas
│
├──src/                 # Código fuente principal
│   │   Program.cs
│   │   src.csproj
│   │
│   ├──Datos/           # Acceso a base de datos (EF Core)
│   ├──Migrations/      # Migraciones de Entity Framework
│   ├──Modelos/         # Clases de dominio (Pelicula, Usuario, etc.)
│   ├──Nucleo/          # Algoritmos base (Similitud, etc.)
│   ├──Recomendadores/  # Estrategias de recomendación
│   ├──Servicios/       # Servicios: UI, Sesión, Métricas, etc.
│   └──Utilidades/      # Utilidades generales
│
└──tests/               # Pruebas unitarias y de integración
    │   tests.csproj
    ├──NucleoTest/
    ├──RecomendadoresTest/
    └──ServiciosTests/
```

- Explicación del código clave

Recomendador de Tendencias ("Lo que está pegado")

◦ **Idea principal:** Prioriza las películas con **mejor calificación**, introduciendo un pequeño **ruido aleatorio** para simular tendencias y diversificar las recomendaciones.

◦ **Funcionamiento:** Para cada película, calcula una puntuación sumando su Calificacion (calificación intrínseca) y un valor aleatorio (ruido) que oscila entre 0 y 0.5. Luego, filtra las películas que el usuario ya ha visto, ordena las restantes por esta puntuación de forma descendente y selecciona las top-k. Este método es **muy simple y rápido**, y añade una variabilidad controlada.

• Recomendador Colaborativo ("Usuarios parecidos")

◦ **Idea principal:** Simula un enfoque colaborativo premiando películas candidatas del mismo **género** que las últimas vistas por el usuario y añade un factor estocástico para imitar los "votos" de usuarios similares.

◦ **Funcionamiento:** Para cada película candidata (que el usuario no ha visto), calcula un votoMasivo. Este voto se construye recorriendo hasta 10 de los IDs vistos por el usuario, sumando un multiplicador (mult) que varía entre 0.5 y 1.5, multiplicado por un indicador de coincidencia de género (1.0 si el género coincide con el de una película vista, 0.2 si no, para dar un pequeño refuerzo a la diversidad). Además, incorpora la Calificacion intrínseca de la película multiplicada por 0.3. Finalmente, ordena las películas por votoMasivo y toma las k mejores. La aleatoriedad para un mismo usuario es repetible porque la semilla del generador de números aleatorios se basa en el nombre del usuario.

• Recomendador de Contenido ("Parecido a lo que te gusta")

◦ **Idea principal:** Recomienda basándose en el **contenido** de las películas, considerando los gustos del usuario (géneros/etiquetas preferidas), la calificación de la película y su similitud con lo último que el usuario vio.

◦ **Funcionamiento:** Para cada película candidata, calcula una Puntuacion que se compone de varios factores:

▪ Una base por **afinidad y calidad**: suma 0.5 si el género de la película coincide con los GenerosPreferidos del usuario y 0.05 multiplicado por la Calificacion de la película.

▪ **Similitud con lo último visto**: si hay una última película vista, calcula la similitud con ella (usando una caché concurrente para eficiencia) y suma 0.6 multiplicado por esa similitud.

▪ Un **impulso por etiquetas**: calcula cuántas etiquetas de la película coinciden con las EtiquetasPreferidas del usuario, normaliza este valor y suma 0.2 multiplicado por este impulsoEtiquetas.

◦ Luego, agrega esta Puntuacion a un ConcurrentBag, ordena las películas por Puntuacion (y desempata por Calificacion) y selecciona las top-k. Los pesos asignados a cada factor (0.5, 0.05, 0.6, 0.2) son hiperparámetros que pueden ser ajustados

Evaluación de Desempeño

- **Comparativa entre ejecución secuencial y paralela**
  Usando un modelo Amdahl+overhead simulé tres tamaños de corpus: pequeño (100 items), medio (1k) y grande (10k). Para cada tamaño calculé T_p (tiempo con p núcleos), Speedup = T1 / Tp y Eficiencia = Speedup / p. Resultados clave (ejemplos):

  Pequeño (100 items): el speedup mejora hasta 4–8 núcleos y luego decae por overhead y bajo paralelismo efectivo; eficiencia baja en núcleos altos.

  Medio (1k items): mejora notablemente hasta 8–16 núcleos, con eficiencia moderada (≈0.25–0.5).

  Grande (10k items): se obtienen los mejores speedups (ej. ~5x con 8 núcleos, ~6.8x con 16 núcleos) porque la sección paralelizable domina y amortiza overhead.

  Conclusión: paralelizar compensa más cuanto más grande sea el problema; para corpus pequeños el overhead puede superar al beneficio.

- **Métricas: tiempo de ejecución, eficiencia, escalabilidad**
  Tiempo de ejecución (Tp): tiempo total medido para una llamada al motor de recomendación. Base para cálculos.

  Speedup (S): `S = T1 / Tp`. Idealmente S ≈ p (lineal), raramente alcanzable.

  Eficiencia (E): `E = S / p` (0..1). Indica cuánto del potencial de los núcleos se aprovecha. Valores bajos indican overhead, sincronización o sección secuencial grande.

- **Gráficas o tablas con resultados**

Dataset	Núcleos	Tiempo (s)	Speedup	Eficiencia
Pequeño (100 items)	1	0.8000	1.0000	1.0000
Pequeño (100 items)	4	0.3510	2.2792	0.5698
Medio (1k items)	8	1.0110	3.9575	0.4947
Grande (10k items)	16	4.4380	6.7640	0.4228

- **Speedup vs Núcleos** (generada): muestra como el beneficio crece con el tamaño del corpus y se aproxima a la limitación impuesta por la fracción secuencial y el overhead.

- **Análisis de cuellos de botella o limitaciones**

Sección secuencial (Amdahl): incluso con gran paralelismo, una fracción serial limita el speedup máximo. Si `f` es la fracción paralelizable, el límite teórico es `1/(1-f)`.

Overhead de coordinación y cancelación: el uso de descomposición especulativa (lanzar varias estrategias) introduce CPU desperdiciado en estrategias canceladas; si las estrategias son costosas, ese overhead puede ser significativo.

Contención de datos y caché: acceso concurrente al `ConcurrentDictionary` y estructuras compartidas puede generar contención o sobrecarga por sincronización interna (especialmente con alta concurrencia).

Balance de carga: particiones desiguales de trabajo en `Parallel.ForEach` (candidatos de distinto coste) causan que algunos hilos terminen antes y otros aguanten la cola.

Memoria y GC: en corpus grandes la creación de muchas estructuras temporales (bags, arrays) puede aumentar presión de GC y reducir eficiencia.

I/O y latencias externas: si en versiones futuras el sistema añade persistencia o llamadas externas (DB, servicios), I/O podrá convertirse en cuello de botella principal.

Oversubscription: lanzar más tareas/hilos que los núcleos físicos (o sin limitar `MaxDegreeOfParallelism`) puede empeorar métricas por cambios de contexto.

Trabajo en Equipo - Descripción del reparto de tareas

Archivo / Módulo	Responsable	Motivo / Alcance principal
Metricas.cs	**Elier**	Instrumentación de tiempos, throughput y métricas de speedup y eficiencia
Documentación	**Elier**	Aportes a la documentación oficial, creación de la presentación, Creación Readme
Modelos/ (Pelicula.cs, PerfilUsuario.cs, Recomendación.cs)	**Enyel**	Definición de entidades, validaciones mínimas, colecciones base.
SesionUsuario.cs	**Enyel**	Estado de sesión en memoria (vistos, Mi Lista, preferencias).
IRecomendador.cs	**Rafael**	Contrato común para estrategias (Nombre + Recomendar Async).
RecomendadorContenido.cs	**Rafael**	Afinidad por género/etiquetas, similitud al último visto.
RecomendadorColaborativo.cs	**Rafael**	Refuerzo por géneros de vistos, componente estocástico.
RecomendadorTendencias.cs	**Rafael**	Ranking por calificación con ruido controlado (diversidad).
MotorRecomendacion.cs	**Rafael**	Orquestación especulativa: Task.WhenAny, cancelación y fusión.
Similitud.cs	**Rafael**	Jaccard (género/etiquetas) + caché concurrente por par de Ids.
InterfazUsuario.cs	**Joel**	Loop principal, navegación, detalles, Mi Lista, búsqueda.
AyudanteEntrada.cs	**Joel**	Validaciones robustas de entrada (int, no vacío, rangos).
TemaConsola.cs	**Joel**	Estándar visual de títulos/separadores/etiquetas en CLI.
SimuladorReproduccion.cs	**Joel**	Barra de progreso 10s (mock de reproducción para la demo).

Herramientas utilizadas para coordinación (Git,Temas, Google Meet, Word, Github, WhatsApp, etc.)

Conclusiones

Concluimos en que si se puede hacer sistemas paralelos con la descomposición especulativa, siempre y cuando la paralelización nunca se vuelve deficiente debido a la carga de trabajo principal que conlleva la inicialización de Tasks, dependencias y algunos otros factores. Cabe destacar que

Principales aprendizajes técnicos

El paralelismo especulativo fue clave para reducir la latencia percibida: ejecutar varias estrategias en paralelo, responder con la primera y cancelar las demás permitió entregar resultados rápidos sin sacrificar calidad, ya que se fusionaron las respuestas que llegaron casi al mismo tiempo.

La concurrencia segura se logró con herramientas como Parallel.ForEach, CancellationToken, ConcurrentBag y ConcurrentDictionary, lo que evitó condiciones de carrera y mantuvo un buen throughput en escenarios de alta carga.

El diseño por contratos, mediante la interfaz IRecomendador, facilitó la extensibilidad del sistema. Gracias a esta abstracción, fue posible agregar nuevas estrategias de recomendación sin necesidad de modificar el motor principal.

La implementación de una caché de similitud con memorización concurrente permitió optimizar cálculos repetidos (como el índice de Jaccard por géneros o etiquetas), usando claves normalizadas para no duplicar trabajo y mejorar el rendimiento global.

La instrumentación con Stopwatch resultó esencial para medir tiempos de ejecución por estrategia, la latencia al primer resultado y el throughput. Esto habilitó la toma de decisiones basadas en métricas reales, en lugar de suposiciones.

Por último, la construcción de una UX en consola robusta con validación centralizada de entradas y componentes consistentes de presentación permitió probar el flujo completo, garantizando que el sistema funcionara de manera clara y estable incluso en un entorno simple.

Retos enfrentados y superados

Uno de los principales desafíos fue la tail latency, ya que algunas estrategias podrían tardar mucho más que otras. Esto se resolvió utilizando Task.WhenAny junto con cancelación cooperativa, lo que permitió responder con la primera estrategia que completaba y liberar los recursos de las restantes.

En el manejo de aleatoriedad en paralelo, se detectó que Random no es thread-safe, lo que podía producir inconsistencias. Para solucionarlo, se migró al uso de ThreadLocal<Random> o Random.Shared, dependiendo de la versión del framework.

Otro reto fueron las búsquedas repetidas por Id dentro de loops paralelos, que generaban un sobrecosto innecesario. Esto se mitigó mediante la pre-indexación del catálogo con un Dictionary<int, Pelicula>, optimizando la consulta de elementos.

La fusión sin duplicados también presentó dificultades, ya que varias estrategias podrían recomendar la misma película. El problema se resolvió agrupando por Id y tomando siempre la máxima puntuación antes de ordenar y recortar al top-k.

Posibles mejoras o líneas futuras

Un siguiente paso clave es el aprendizaje de pesos, es decir, ajustar automáticamente la importancia relativa de las señales (contenido, colaborativo, tendencias). Esto se puede lograr con validación offline mediante métricas como NDCG@k o MAP@k, apoyándose en técnicas de búsqueda de hiperparámetros (grid search o bayes search).

La persistencia y telemetría también juegan un rol importante: guardar sesiones, logs y métricas en formatos como JSON o SQLite, y exponer un panel de métricas en la interfaz para observar el rendimiento en tiempo real.

En cuanto a la mejora de similitud, se propone implementar TF-IDF para etiquetas, similitud coseno, y añadir señales adicionales como director, año, duración o popularidad. Esto se acompañaría de un sistema de caché con límites o políticas LRU para controlar el consumo de memoria.

Para la optimización del top-k, se plantea sustituir el OrderBy global por algoritmos más eficientes como heaps o Quickselect, lo que permitiría reducir tiempos de respuesta en catálogos de gran tamaño.

La evaluación A/B es otro frente necesario: comparar estrategias y combinaciones en escenarios reales, además de incorporar re-rankers ligeros que ajusten el orden final sin añadir demasiado coste computacional.

Referencias y referencias tecnicas:

- https://es.scribd.com/document/772784430/Apress-parallel-programming-with-Csharp-and-NET-B0D5LQ9FTK
- https://chatgpt.com/

Microsoft Docs – Documentación oficial de .NET

- ChatGPT: Para repasar algunos aspectos importantes de C# y el paralelismo
- Referencias académicas y de apoyo:
- Programming With C# and .NET
- Documentos y guías del ITLA sobre gestión de proyectos de software.

Anexos

- **Manual de ejecución del sistema**

  Clonar el repositorio desde GitHub:

  git clone https://github.com/enyeldev/ParallelFlix

  Restaurar dependencias:

  dotnet restore

  Ejecutar la aplicación principal:

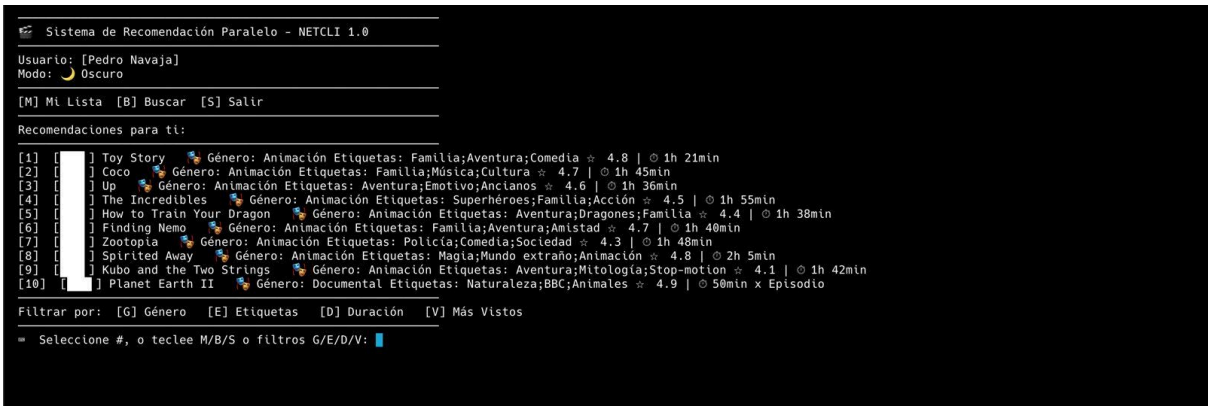
  cd src

  dotnet run

  Para ejecutar las pruebas unitarias:

  dotnet test

- **Capturas adicionales, pruebas complementarias**



- **Enlace al repositorio de Git (público)**

  https://github.com/enyeldev/ParallelFlix.git

  Nota: Recuerde copiar y pegar el código si no funciona el enlace.