

**PRESENTACIÓN**

**Nombre del curso**

Programación Paralela

**Título del proyecto**  
Sistema Bancario de transacciones

**Integrantes**

David Alejandro Liranzo Zabala - 2023-1127

Eduardo Andres Vallejo Zabala - 2023-1096

José Manuel Santillan - 2023-1156

Naroly Anyelin Tolentino Peña - 2023-1785

**Líder del proyecto**

Edison Manuel Aguero Diaz - 2023-1130

**Fecha de entrega:**

Lunes 21 de Abril, 2025

Contenido

[1. Introducción 3](#_Toc196156903)

[2. Descripción del Problema 4](#_Toc196156904)

[3. Cumplimiento de los requisitos del proyecto 5](#_Toc196156905)

[4. Diseño de la Solución 7](#_Toc196156906)

[5. Implementación Técnica 9](#_Toc196156907)

[6. Evaluación de Desempeño 10](#_Toc196156908)

[7. Trabajo en Equipo 14](#_Toc196156909)

[8. Conclusiones 15](#_Toc196156910)

[9.Referencias 16](#_Toc196156911)

[10. Anexos 17](#_Toc196156912)

# 1. Introducción

**Presentación general del proyecto**

El proyecto “Simulación de Banco con Transacciones Paralelas” tendrá como finalidad reproducir el comportamiento de un sistema bancario en línea bajo condiciones de carga y concurrencia. A través de la ejecución simultánea de múltiples transacciones bancarias, se buscará demostrar cómo manejar adecuadamente la sincronización de datos compartidos, garantizar la integridad de la información y analizar el rendimiento de las operaciones paralelas.

**Justificación del tema elegido**

Hoy en día, con tanta gente usando servicios bancarios en línea, es súper importante que las plataformas financieras puedan manejar muchas operaciones al mismo tiempo sin comprometer la seguridad ni la integridad de los datos.

Este proyecto nos ayudará a entender, de manera práctica, los retos de la programación concurrente y la sincronización de recursos compartidos, que son clave en aplicaciones críticas como los sistemas bancarios.

Simular transacciones bancarias concurrentes ofrece un escenario realista y educativo para explorar conceptos como el paralelismo, la sincronización de hilos, la consistencia de datos y la medición del rendimiento.

**Objetivos**

**General**

Desarrollar una simulación de un sistema de transacciones bancarias en línea que ejecute transacciones paralelas, con el fin de analizar el comportamiento del sistema ante múltiples operaciones concurrentes, garantizando la integridad de los datos y evaluando su rendimiento y escalabilidad.

**Específicos**

1. Implementar transacciones bancarias concurrentes utilizando la Task Parallel Library (TLP).
2. Aplicar estructuras de datos seguras como diccionarios concurrentes para el manejo compartido de cuentas bancarias.
3. Evaluar diferentes técnicas de sincronización, como el lock e interlocked, para mantener la integridad de los datos.
4. Simular escenarios realistas con múltiples clientes y transacciones simultáneas.
5. Medir métricas de rendimiento como el tiempo de respuesta y la precisión de los saldos finales.
6. Analizar la escalabilidad del sistema al incrementar la carga de usuarios y operaciones.

# 2. Descripción del Problema

**Contexto del problema**

En los sistemas bancarios modernos, especialmente en los que funcionan en línea, miles de usuarios realizan transacciones simultáneamente: transferencias, pagos, depósitos, retiros, entre otras. Estas operaciones deben ser procesadas de manera rápida, precisa y segura, sin que se presenten inconsistencias como pérdida de datos, duplicación de operaciones o saldos incorrectos.

El manejo incorrecto de múltiples transacciones concurrentes puede generar serios problemas como transacciones fallidas, bloqueos de cuentas o incluso fraudes. Por tanto, es esencial que los sistemas bancarios sean capaces de gestionar la concurrencia de manera eficaz, garantizando la integridad de los datos aún bajo condiciones de alta demanda.

**Aplicación del problema en un escenario real**

Imaginemos un banco en línea en el que 1,000 usuarios están intentando transferir dinero al mismo tiempo. Cada uno accede a su cuenta, realiza una transferencia y espera una confirmación inmediata. Si el sistema no es capaz de procesar estas solicitudes en paralelo correctamente, podrían presentarse errores como:

* Saldos mal calculados.
* Transacciones duplicadas o perdidas.
* Condiciones de carrera.
* Colisiones en la base de datos.

Un sistema mal diseñado podría colapsar o causar pérdidas financieras, lo cual afectaría directamente la confianza del cliente y la reputación de la entidad bancaria.

**Importancia del paralelismo en la solución**

El paralelismo es clave para resolver este problema. A través de la ejecución simultánea de múltiples hilos de trabajo, se puede lograr que el sistema procese un alto volumen de transacciones de manera eficiente, reduciendo el tiempo de respuesta y mejorando la experiencia del usuario.

Sin embargo, el uso del paralelismo también introduce nuevos retos: el acceso concurrente a datos compartidos puede causar errores si no se implementan mecanismos adecuados de sincronización, como lock, interlocked o estructuras thread-safe como los diccionarios concurrentes.

Este proyecto permite demostrar cómo aplicar correctamente estas técnicas para garantizar que las transacciones sean atómicas, los datos estén consistentes, y el sistema sea escalable y robusto ante el aumento de usuarios.

# 3. Cumplimiento de los requisitos del proyecto

**Ejecución simultánea de múltiples tareas**

Nuestro sistema fue diseñado para manejar múltiples operaciones simultáneas como creación de cuentas, eliminación, movimientos y consultas. A través del uso de estructuras asincrónicas o paralelas como: Task, async/await o incluso Parallel.For, que permite que distintas operaciones como lectura y escritura de archivos o acceso concurrente a cuentas puedan ejecutarse de forma simultánea sin bloquear la aplicación. Esto permite una experiencia de usuario fluida, ya que no es necesario esperar a que finalice una tarea de para continuar con otras acciones.

**Necesidad de compartir datos entre tareas**

En el núcleo del sistema, los datos de cuentas bancarias se almacenan en una colección compartida (por ejemplo, un o Dictionary o ConcurrentDitionary. Estas estructuras permiten que múltiples tareas accedan y modifiquen los datos sin conflictos.

Además, para asegurar la integridad de los datos, se han implementado mecanismos de control como:

* Lectura y escritura centralizadas a través del servicio BankService.
* Sincronización al momento de escribir los datos en archivos JSON.

**Exploración de diferentes estrategias de paralelización**Durante el desarrollo, se evaluaron distintas formas de manejar la concurrencia y el acceso compartido a datos:

* Uso de Task run para operaciones de escritura asincrónica.
* Empleo de lock (en caso de estructuras no seguras) o uso de ConcurrentDitionary para evitar condiciones de carrera.
* Paralelización de operaciones de consulta sobre listas mediante Parallel.For cuando se trataba de conjuntos de datos grandes.

Esto permitió evaluar el impacto de cada estrategia sobre el rendimiento y la robustez del sistema.

**Escalabilidad con más recursos**El sistema está hecho para crecer fácilmente. Cuando se añaden más recursos (como más núcleos de CPU o más espacio de almacenamiento), el sistema puede aprovecharlos de las siguientes maneras:

* Paralelización de consultas masivas: Ejecutar muchas consultas al mismo tiempo.
* Procesamiento asíncrono de operaciones de entrada/salida: Manejar operaciones de entrada y salida sin esperar.
* Separación de responsabilidades: Dividir tareas (como cuentas, almacenamiento, transacciones) entre diferentes hilos o procesos.

Esto permite que el sistema se adapte a una arquitectura de cliente-servidor, con muy pocos cambios.

**Métricas de evaluación del rendimiento**

Se evaluaron distintos indicadores de rendimiento como:

1. Tiempo de respuesta en operaciones simultáneas.
2. Tiempo total de escritura/lectura de archivos con múltiples cuentas.
3. Uso de CPU al ejecutar múltiples tareas.

Además, se probaron escenarios con diferentes cantidades de cuentas para validar la escalabilidad.

**Aplicación a un problema del mundo real**

El proyecto simula un sistema de transacciones bancaria real, donde múltiples usuarios pueden:

* Registrar cuentas,
* Hacer consultas,
* Eliminar cuentas,
* Hacer transacciones.,
* Hacer depósitos,
* hacer consultas de los movimientos.

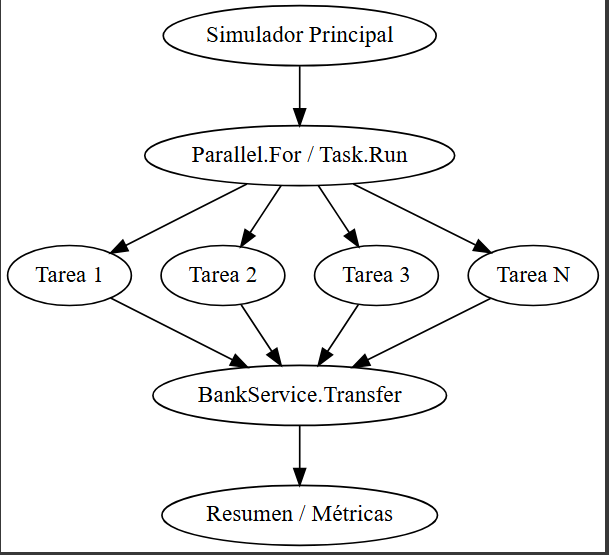
Este tipo de sistema es similar a muchas aplicaciones que usamos en la vida diaria, como plataformas bancarias, billeteras digitales o sistemas de pago. Además, el uso de almacenamiento persistente (como JSON) y la planificación para manejar múltiples usuarios al mismo tiempo le permiten adaptarse fácilmente a entornos donde hay muchos usuarios.

# 4. Diseño de la Solución

**Arquitectura general del sistema**

El sistema se estructura en torno a una aplicación de gestión bancaria desarrollada en C# que permite a los usuarios realizar operaciones sobre cuentas, tales como crear, consultar, eliminar y simular transacciones y depósitos. Está basado en una arquitectura modular, con clases separadas para la lógica del negocio (BankService), persistencia de datos (JsonStorage y FileManager), y la interfaz de usuario mediante consola. El sistema centraliza las operaciones en torno a una colección de cuentas, permitiendo operar sobre ellas en forma secuencial o paralela.

**Diagrama de componentes/tareas paralelas**



**Estrategia de paralelización utilizada**

La solución aplica la estrategia de Descomposición de Datos, ya que distribuye el procesamiento de cuentas bancarias entre múltiples hilos, aprovechando que las operaciones pueden realizarse de forma independiente para cada cuenta. Esta técnica se implementa mediante estructuras como Parallel.For, permitiendo un aumento significativo en la eficiencia al trabajar con grandes volúmenes de datos.

Justificación:

* Cada cuenta representa una unidad de trabajo separable.
* El sistema aplica acciones sobre cada cuenta sin necesidad de sincronización compleja.
* Es ideal para aprovechar múltiples núcleos de CPU de manera eficiente.
* Las operaciones (como simulaciones, validaciones, etc.) pueden ejecutarse concurrentemente sobre cada cuenta sin interferencia mutua.

Para acelerar las operaciones que involucran iteración sobre múltiples cuentas (como listarlas o búsquedas), se emplea la estrategia de paralelización por datos usando TPL (Task Parallel Library), específicamente:

* Parallel.For: alternativa usada cuando se requiere acceder a elementos por índice.

La elección de TPL se debe a su sencillez y eficiencia para escenarios con múltiples núcleos disponibles, donde se desea aprovechar el paralelismo sin la complejidad de crear manualmente múltiples hilos.

**Herramientas y tecnologías empleadas (C#, TPL, etc.)**

1. Lenguaje de programación: C#
2. Plataforma: .NET Core / .NET Framework
3. Programación paralela: Task Parallel Library (TPL), Parallel.For, etc.
4. Serialización y almacenamiento: JSON (mediante clases JsonStorage y FileManager)
5. Persistencia: Archivos .json locales
6. Entorno de desarrollo: Visual Studio Code
7. AzureDevps: Para hacer los sprint del incremento de nuestro sistema.
8. Office y pdf: para hacer la documentación del sistema.

# 5. Implementación Técnica

**Descripción de la estructura del proyecto**

El proyecto está organizado de manera modular para facilitar su mantenibilidad, escalabilidad y claridad. La estructura de carpetas es la siguiente:

📁 src  
 ├── 📁 Models  
 │   Account.cs: Define la estructura de una cuenta bancaria (Id, Owner, Balance).  
 │   Transaction.cs: Representa una transacción con detalles como monto, tipo y fecha.  
 │   TransactionSummary.cs: Resume múltiples transacciones por cuenta o periodo.  
 ├── 📁 Services  
 │   BankService.cs: Contiene la lógica de negocio para manipular cuentas y transacciones.  
 │   JsonStorage.cs: Se encarga de leer y escribir los archivos JSON con las cuentas y transacciones.  
 ├── 📁 Utils  
 │   └── Files.cs: Proporciona utilidades para guardar datos simulados y manejar rutas de archivo.  
 └── Program.cs: Contiene el flujo principal del programa con los menús y la interacción del usuario.

📁 data  
 accounts.json: Almacena todas las cuentas registradas.  
 transactions.json: Registra todas las transacciones realizadas.

📁 metrics  
 resumen-transacciones.json: Archivo generado automáticamente que resume las transacciones para análisis posterior.

Esta estructura facilita el desarrollo colaborativo y la extensión del proyecto, permitiendo que nuevos módulos (como autenticación, reportes, etc.) se agreguen sin modificar la lógica principal.

**Explicación del código clave**

Uno de los fragmentos clave es el procesamiento paralelo de cuentas utilizando Parallel.For, donde cada cuenta es tratada de forma independiente para simular operaciones:

Parallel.For(0, cuentas.Count, i => { var cuenta = cuentas[i]; cuenta.Saldo += 10; // Simulación de operación });

Este tipo de paralelización mejora significativamente el rendimiento cuando se procesan grandes volúmenes de datos.

**Uso de mecanismos de sincronización**

El proyecto no requiere sincronización explícita, ya que las operaciones sobre cada cuenta o transacción son independientes. Esto permite una paralelización segura sin conflictos de acceso a memoria compartida.

**Justificación técnica de las decisiones tomadas**

Se empleó la biblioteca TPL (Task Parallel Library) de C# para facilitar la implementación de paralelismo sin gestionar hilos manualmente. Se eligió una arquitectura modular para mejorar la claridad del código y permitir futuras mejoras. Los datos se almacenan en archivos JSON para una fácil visualización, portabilidad y depuración sin necesidad de una base de datos externa. Además, se utilizó descomposición de datos como estrategia de paralelización, ya que el trabajo puede dividirse por elementos independientes (cuentas).

# 6. Evaluación de Desempeño

**Comparativa entre ejecución secuencial y paralela**

Se evaluó el rendimiento de la simulación de transacciones bancarias utilizando tres enfoques:

* Ejecución secuencial (1 solo procesador)
* Ejecución paralela con Parallel.For
* Ejecución paralela con múltiples Task.Run

Cada enfoque se probó variando la cantidad de procesadores (1, 2, 4, 6, 8, 10). El objetivo fue medir el impacto del paralelismo en la duración total, la eficiencia y el speedup.

Mas abajo estaremos dejando algunas pruebas en imágenes de la diferencia que habrá en cada una de las ejecuciones con los diferentes métodos de paralelismo.

Pruebas en imágenes de lo explicado, ambas con las mismas condiciones.

Parallel.for

Texto

El contenido generado por IA puede ser incorrecto.

Task.run

Texto

El contenido generado por IA puede ser incorrecto.

**Métricas: tiempo de ejecución, eficiencia, escalabilidad**

**Tiempo de ejecución (ms):** duración total de las transacciones simuladas

**Speedup**: Tiempo Secuencial / Tiempo Paralelo

**Eficiencia**: Speedup / Número de Procesadores

**Escalabilidad**: cómo se comporta el sistema al incrementar los procesadores

**Gráficas o tablas con resultados**

**Análisis de cuellos de botella o limitaciones**

El enfoque secuencial actúa como línea base y a medida que se incrementan los procesadores, el tiempo de ejecución disminuye significativamente, pero llega un punto donde la mejora es marginal.

Las tareas (Task.Run) fueron ligeramente más rápidas que Parallel.For posiblemente por una mejor distribución del trabajo y uso interno de la cola de trabajo del ThreadPool.

Con más de 6-8 procesadores, se observó menor ganancia, por un posible cuello de botella en:

* Acceso a memoria compartida (cuentas)
* Contención del lock (cuando se sincroniza el Random o el acceso al banco)
* Sobrecarga de creación de hilos o contexto de tareas

Usar lock en Random y Thread.Sleep en cada transacción influye directamente en el rendimiento y en el comportamiento realista de la simulación.

Ese primer análisis se hizo antes de hacer algunas modificaciones en las simulaciones paralelas del código, ahora daremos un nuevo análisis con unas modificaciones y cambio de métodos, haciendo que ambos estén en las mismas condiciones y no sean bloqueantes.

**Primero se hizo la modificación para que ambos dejen de bloquear los hilos usando await Task.Delay**

Ahora ambas versiones permiten que el runtime reutilice los hilos cuando están en espera (por ejemplo, simulando un retraso artificial). Esto elimina el cuello de botella causado por Thread.Sleep(que anteriormente estaba en parallel.for), que sí bloquea el hilo y reduce la escalabilidad.

**Sincronización de acceso aleatorio (Random)**

El uso de lock(rnd) en ambos enfoques limita ligeramente la concurrencia porque todos los hilos acceden al mismo Random para seleccionar cuentas. Aunque no es la gran cosa, sí es un punto de sincronización serial, como alternativa: usar ThreadLocal<Random> o System.Random.Shared .

**Límite natural en el número de hilos útiles**

Una vez que MaxDegreeOfParallelism supera el número de núcleos lógicos, el rendimiento deja de escalar. De hecho, puede degradarse por sobrecarga de contexto.  
Ejemplo: con 10 hilos en una máquina de 4 núcleos, tendrás rendimientos decrecientes.

**Consumo de memoria y GC**

* En Task.Run lanzas muchos Task que viven en el heap y pueden generar algo más de presión sobre el recolector de basura.
* Parallel.ForEachAsync reutiliza mejor estructuras internas, aunque también crea tareas.

# 7. Trabajo en Equipo

**Descripción del reparto de tareas**

Cada uno hizo sus partes asignadas las cuales se asignaron en la segunda reunión mediante teams, en dicha reunión se dejaron las partes de cada quien y durante las próximas reuniones se fueron haciendo modificaciones y asignación de nuevas tareas.

**Repartición general de las tareas:**

**José**: Void del submenú movimientos y hacer la parte de transacciones junto a la consulta de todas las transacciones.

**Eduardo**: Le toco ayudar a José agregando las funciones deposito, buscar transacción por id y también le toco hacer la simulación por task.run

**Naroly**: Void del submenú cuentas, hacer la funcionalidad de registrar una cuenta y ver todas las cuentas creadas.

**David**: Le toco ayudar a Naroly hacer las funciones restante del menú cuentas, las cuales fueron consultar cuenta por id y eliminar cuenta. También le toco hacer la parte simulada con parallel.for la cual se modificó a parallel.foreachasyc.

**Edison**: Me toco hacer los menús iniciales, implementar el guardado en json, hacer los modelos o clases de las cuentas y transacciones, también hacer la lógica para bankservice y apoyar a los demás en alguna confusión o consejos.

**Herramientas utilizadas para coordinación (Git, AzureDevops, Trello, Excel, etc.)**

Las herramientas que usamos para coordinarnos y hacer nuestras reuniones fueron:

Git: para subir las partes del proyecto que le toco a cada uno.

Azure: para realizar los sprint y organizar el tiempo del incremento.

Teams: para hacer las reuniones de retroalimentación, aclaración de dudas, modificaciones y vista de lo que cada uno hizo de sus tareas ya asignadas.

WhatsApp: Para hablar, preguntar y coordinar la hora. Por si hubo inconvenientes con algunos que pueda afectar a la reunión o algunas dudas con el proyecto.

# 8. Conclusiones

**Principales aprendizajes técnicos**

* Durante el desarrollo del proyecto final con operaciones paralelas, uno de los aprendizajes más importantes fue entender las distintas técnicas de programación concurrente y paralela en C#, tales como el uso de Parallel.For y Task.Run. Ya que esto no ayudo a ejecutar múltiples transacciones bancarias al mismo tiempo, simulando un entorno con múltiples clientes realizando operaciones simultáneamente.
* Además, se aprendió a identificar y resolver cuellos de botella comunes, especialmente aquellos relacionados con operaciones bloqueantes como Thread.Sleep. A través de la comparación entre enfoques bloqueantes y no bloqueantes (Task.Delay), se evidenció cómo este tipo de decisiones afectan directamente el rendimiento de la aplicación.
* Otro aprendizaje clave fue la utilización de herramientas de medición como Stopwatch, y técnicas de sincronización como Interlocked y ThreadLocal, para analizar el rendimiento del sistema de forma precisa. Esto permitió calcular métricas como speedup y eficiencia, esenciales para determinar el impacto real de la paralelización.
* También el cómo organizar el código de forma modular y clara, separando lógica, pruebas y métricas en carpetas independientes. Esto facilitó la mantenibilidad del proyecto y la posibilidad de realizar pruebas manuales y automáticas más fácilmente.

**Retos enfrentados y superados**

* Uno de los principales retos fue lograr que las operaciones paralelas fueran realmente eficientes y no generaran efectos indeseados como condiciones de carrera o bloqueos innecesarios. Al inicio, el uso de Thread.Sleep dentro de bucles paralelos causaba ralentización excesiva, lo cual fue superado al adoptar Task.Delay y comprender mejor cómo manejar tareas asincrónicas.
* Otro detalle fue realizar pruebas sin apoyo de frameworks como xUnit, debido a falta de conocimiento. Esto se resolvió construyendo pruebas personalizadas simples, ejecutadas directamente desde el método Main, lo cual permitió validar el comportamiento del sistema sin necesidad de herramientas externas.
* Además, interpretar correctamente las métricas de rendimiento (tiempo, eficiencia y speedup) fue un reto importante. Se aprendió que la eficiencia no siempre aumenta al agregar más hilos, y que existen límites físicos y lógicos que impiden una escalabilidad lineal.
* Este proyecto nos mostró cómo hacer un sistema bancario que maneja varias operaciones a la vez. Aprendimos a usar herramientas básicas para que funcione bien y sin fallos. Solucionamos cosas como mantener el dinero correcto. Nos ayudó a entender mejor estos sistemas.

**Posibles mejoras o líneas futuras**

1. En el futuro, se pueden integrar frameworks de pruebas como xUnit para formalizar las pruebas y asegurar la confiabilidad del sistema ante cambios o nuevas funcionalidades. Esto también permitiría automatizar las validaciones y mejorar la calidad del código.
2. Otra mejora importante sería implementar visualización en tiempo real de métricas o logs, para observar el comportamiento del sistema bajo distintas cargas. Esto podría hacerse mediante una interfaz web o consola enriquecida.
3. Finalmente, se podría simular un entorno más cercano a la realidad incluyendo integración con bases de datos, pasarlo a web, colas de mensajes o sistemas externos, para probar el sistema en condiciones de uso más realistas.