Miembros:

José Odalis Encarnación Rivera 2023-1115

Frankoris Rodríguez Ortiz 2023-1346

Rosa Carolina Guerrero German 2023-1112

Imanol Rodríguez Rodríguez 2024-0256

Líder:

José Odalis Encarnación Rivera 2023-1115

Periodo Académico:

Primer período c-3 2025

Carrera:

Desarrollo de Software

Materia:

Programación Paralela

Nombre del docente:

Erick Leonardo Pérez Veloz

Tema:

Optimización del Problema del Viajante mediante Paralelismo y Descomposición Explorativa

Fecha:

12/12/2025

Índice

1. Introducción

Presentación general del proyecto

El presente proyecto de investigación y desarrollo tecnológico se centra en el estudio, implementación y análisis de soluciones computacionales para problemas de optimización combinatoria de alta complejidad. Específicamente, el trabajo aborda el **Problema del Viajante** (conocido en la literatura académica como *Traveling Salesman Problem* o TSP), utilizando el lenguaje de programación C# y el entorno de ejecución .NET.

La propuesta técnica consiste en el desarrollo de dos motores de resolución distintos para comparar su rendimiento: una implementación secuencial basada en el algoritmo exacto de **Branch & Bound** (Ramificación y Poda), y una implementación paralela que aplica el patrón de diseño de **Descomposición Explorativa**. Esta última aprovecha las capacidades de los procesadores modernos multinúcleo para dividir el espacio de búsqueda y explorar múltiples rutas potenciales de manera simultánea.

El sistema incluye módulos para la generación de escenarios mediante datos simulados (coordenadas en un plano euclidiano), mecanismos de sincronización de hilos para garantizar la integridad de los datos compartidos y un módulo de métricas (*benchmarking*) diseñado para cuantificar la ganancia de rendimiento (*speedup*) obtenida mediante la paralelización.

Justificación del tema elegido

La elección del Problema del Viajante como eje central de este proyecto responde a su estatus fundamental en las ciencias de la computación. Al ser clasificado como un problema **NP-Hard**, el TSP presenta un crecimiento factorial en su complejidad computacional (***O(N!)***) conforme aumenta el número de ciudades. Esta característica lo convierte en el candidato ideal para demostrar las limitaciones de la computación secuencial tradicional y las ventajas tangibles de la programación paralela.

A diferencia de procesar un arreglo estático, la exploración de un árbol de búsqueda es dinámica e impredecible; algunas ramas pueden descartarse rápidamente mediante la poda, mientras que otras requieren un cómputo intensivo. Este escenario obliga a implementar estrategias avanzadas de balanceo de carga y sincronización de memoria (como el uso de *locks* y variables atómicas), lo cual enriquece significativamente el valor académico y formativo del proyecto.

Objetivos (general y específicos)

Objetivo General

Desarrollar y evaluar una solución paralela eficiente para el Problema del Viajante utilizando el algoritmo de Branch & Bound, con el fin de demostrar la reducción del tiempo de ejecución y la mejora en la capacidad de procesamiento frente a una solución secuencial equivalente.

Objetivos Específicos

- Implementar un generador de datos híbrido que permita crear escenarios de prueba tanto deterministas (para validación de correctitud) como estocásticos (para pruebas de rendimiento), basados en coordenadas euclidianas.

- Desarrollar un algoritmo secuencial de referencia utilizando técnicas de recursividad y poda, que garantice la obtención de la solución óptima exacta.

- Diseñar e implementar una versión paralela del algoritmo aplicando descomposición explorativa, gestionando la concurrencia mediante primitivas de sincronización para proteger el estado global compartido (mejor ruta encontrada).

- Integrar optimizaciones heurísticas (inicialización *Greedy* y ordenamiento de vecinos) para acelerar la convergencia del algoritmo y maximizar la eficiencia de la poda en entornos paralelos.

- Realizar un análisis comparativo de rendimiento (*benchmark*) para medir el tiempo de ejecución, calcular el *speedup* y determinar el límite de escalabilidad del sistema en función del número de ciudades.

2. Descripción del problema

Contexto del problema seleccionado

El Problema del Viajante se define formalmente dentro de la teoría de grafos. Dado un conjunto de ciudades y las distancias entre cada par de ellas, el objetivo es encontrar la ruta más corta posible que visite cada ciudad exactamente una vez y regrese a la ciudad de origen. Matemáticamente, se busca el ciclo hamiltoniano de peso mínimo en un grafo completo ponderado.

La dificultad intrínseca del problema radica en la **explosión combinatoria**. Para un conjunto de ***N*** ciudades, existen ***(N-1)! / 2*** rutas posibles. Esto significa que, mientras un problema con 10 ciudades tiene aproximadamente 181,000 rutas posibles, un problema con 16 ciudades generan más de ***2×10^13*** (20 billones) de combinaciones. En este contexto, los algoritmos de fuerza bruta se vuelven inviables casi de inmediato, haciendo necesario el uso de técnicas inteligentes como *Branch & Bound*, que permiten descartar vastas secciones del espacio de búsqueda sin necesidad de explorarlas, basándose en estimaciones de costos.

Aplicación del problema en un escenario real

Aunque el enunciado clásico refiere a un viajante de comercio, las aplicaciones prácticas de este modelo matemático son críticas en la industria moderna. El escenario más evidente es la **logística y distribución de "última milla"**. Empresas de paquetería como DHL, FedEx o

Amazon utilizan variantes avanzadas del TSP para planificar las rutas diarias de miles de vehículos. Una optimización de apenas un 1% en estas rutas puede traducirse en ahorros millonarios en combustible, mantenimiento de vehículos y reducción de la huella de carbono.

Otras aplicaciones relevantes incluyen:

- **Manufactura de Electrónica:** Optimización del movimiento de taladros robóticos y brazos mecánicos al insertar componentes en placas de circuitos impresos (PCB) para minimizar el tiempo de desplazamiento.

- **Genética:** Ordenamiento de fragmentos de ADN en procesos de secuenciación genómica.

- **Astronomía:** Planificación de movimientos de telescopios espaciales para observar múltiples objetivos celestes minimizando el gasto de energía en la reorientación.

Importancia del paralelismo en la solución

En problemas de naturaleza explorativa como el TSP, el paralelismo no es simplemente una herramienta para "ir más rápido", sino una necesidad estructural para abordar instancias del problema que serían intratables secuencialmente.

La importancia del paralelismo en esta solución se manifiesta en tres aspectos clave:

1. **Exploración Simultánea del Espacio de Soluciones:** Al asignar diferentes ramas del árbol de decisión a distintos núcleos del procesador, el sistema puede investigar múltiples hipótesis de ruta al mismo tiempo. Esto aumenta la probabilidad de encontrar una solución "buena" rápidamente.

2. **Sinergia en la Poda (Inteligencia Colectiva):** Este es el beneficio más notable. Cuando un hilo encuentra una ruta completa con un costo bajo, actualiza el "mejor costo global". Inmediatamente, todos los demás hilos pueden utilizar ese nuevo valor para podar sus propias ramas de forma más agresiva. Es decir, el descubrimiento de un hilo ahorra trabajo a todos los demás, creando un efecto multiplicador en la eficiencia que no existe en el modelo secuencial.

3. **Escalabilidad y Uso de Recursos:** La solución permite aprovechar la arquitectura de hardware moderno. Mientras que un programa secuencial dejaría inactivos la mayoría de los recursos de una CPU actual, la implementación paralela maximiza el uso del silicio disponible, permitiendo resolver instancias con mayor cantidad de ciudades en tiempos razonables, expandiendo así la frontera de lo computable.

3. Cumplimiento de los Requisitos del Proyecto

El desarrollo del presente sistema de optimización combinatoria ha sido guiado por una serie de requisitos funcionales y no funcionales orientados a demostrar la eficacia del paralelismo en problemas NP-Hard. A continuación, se detalla la justificación técnica de cada criterio.

Ejecución simultánea de múltiples tareas

El proyecto satisface este requisito mediante la descomposición del espacio de búsqueda del algoritmo *Branch & Bound*. En lugar de explorar el árbol de decisiones de manera lineal, el sistema identifica las ramas principales que surgen del nodo raíz (la ciudad de origen) hacia sus ***N-1*** vecinos inmediatos.

Cada una de estas ramas se encapsula como una unidad de trabajo independiente. Utilizando la **Task Parallel Library (TPL)** de .NET, específicamente el método Parallel.For, estas unidades se despachan al *Thread Pool* del entorno de ejecución. Esto permite que múltiples núcleos del procesador ejecuten la lógica recursiva de búsqueda (Search) simultáneamente. Por ejemplo, en un procesador de 8 núcleos lógicos, el sistema puede investigar 8 rutas distintas (ej. ***0*** → ***1*** → ***…***, ***0*** → ***5*** → ***…***) en el mismo instante de tiempo, reduciendo teóricamente la latencia total de la solución.

Necesidad de compartir datos entre tareas

La naturaleza cooperativa de la **Descomposición Explorativa** exige que las tareas no trabajen aisladas, sino que compartan información crucial para optimizar el proceso global.

- **Estado Compartido:** La variable BestCost (Cota Superior Global) y la estructura BestRoute.

- **Mecanismo de Colaboración:** Cuando una tarea encuentra una solución completa válida, verifica si su costo es inferior al BestCost actual. De ser así, actualiza este valor global.

- **Justificación:** Este intercambio es vital para la eficiencia del algoritmo. Si el Hilo A encuentra una ruta de costo 1500, el Hilo B (que quizás está explorando una ruta parcial de costo 1600) puede leer este valor actualizado inmediatamente y abortar su búsqueda (Poda), ahorrando ciclos de CPU que de otra forma se desperdiciarían. Sin datos compartidos, el paralelismo sería "ciego" y perdería su ventaja algorítmica.

Exploración de diferentes estrategias de paralelización

Para maximizar el rendimiento, se evaluaron y aplicaron estrategias híbridas que complementan el paralelismo puro:

1. **Paralelismo de Grano Grueso (*Coarse-Grained*):** Se determinó que paralelizar en cada nivel de la recursión generaría un *overhead* excesivo por la creación de millones de tareas pequeñas. Se optó por paralelizar únicamente en el nivel raíz del árbol de búsqueda

.

2. **Optimización del Orden de Exploración:** Se implementó una estrategia de **Vecinos Ordenados** (*Sorted Neighbors*). Antes de paralelizar, se pre-calculan las adyacencias de cada ciudad ordenadas por distancia. Esto fuerza a los hilos a explorar primero las rutas "prometedoras", encontrando óptimos locales rápidamente.

3. **Inicialización Voraz (*Greedy*):** Se incorporó una fase secuencial previa utilizando la heurística del "Vecino Más Cercano" para establecer un BestCost inicial realista, permitiendo que la poda paralela sea efectiva desde el primer milisegundo de ejecución.

Escalabilidad con más recursos

La arquitectura de la solución es agnóstica a la topología del hardware, permitiendo una escalabilidad vertical transparente.

- **Adaptabilidad Dinámica:** El uso de Parallel.For delega la gestión de hilos al *Common Language Runtime* (CLR). El sistema detecta automáticamente la cantidad de núcleos lógicos disponibles y ajusta el grado de concurrencia.

- **Comportamiento Esperado:** Al ejecutar el software en hardware con mayor densidad de núcleos (ej. pasar de 4 a 16 núcleos), el tiempo de ejecución para una instancia fija de ***N*** ciudades (ej. ***N=15***) disminuye proporcionalmente, hasta el límite impuesto por la Ley de Amdahl y la sobrecarga de sincronización de memoria.

Métricas de evaluación del rendimiento

Se diseñó un módulo científico de *benchmarking* (BenchmarkRunner) para objetivar el impacto del paralelismo. Las métricas seleccionadas son:

- **Tiempo de Ejecución (*T*):** Medido con precisión de microsegundos mediante System.Diagnostics.Stopwatch. Se realizan múltiples repeticiones (N iteraciones) por escenario para obtener un promedio estadísticamente significativo, descartando ruido del sistema operativo.

- **Speedup (*S*):** La métrica principal de éxito, definida como ***S = Ts (Tiempo secuencial) / Tp (Tiempo paralelo)***. Un valor ***S > 1*** indica ganancia de rendimiento.

- **Robustez:** Se evalúa la capacidad del sistema para manejar instancias donde la versión secuencial excede el tiempo límite viable (marcado como ***N/A***), demostrando que el paralelismo habilita la resolución de problemas más complejos.

Aplicación a un problema del mundo real

El TSP no es solo una abstracción académica; modela problemas críticos de logística y optimización de recursos.

- **Escenario:** El proyecto simula la planificación de rutas para una flota de distribución que debe visitar múltiples puntos de entrega y regresar a la base.

- **Impacto:** En la industria logística moderna, la capacidad de calcular rutas óptimas en segundos (mediante paralelismo) en lugar de horas permite una **re-planificación dinámica**. Ante eventos imprevistos (tráfico, nuevas órdenes), el sistema puede recalcular la ruta óptima casi instantáneamente, optimizando costos de combustible y tiempos de entrega, lo cual es una ventaja competitiva directa.

4. Diseño de la Solución

Arquitectura general del sistema

La solución se estructura bajo un patrón modular que separa la lógica de negocio, los datos y la evaluación. Los componentes principales son:

1. **Capa de Datos (DataModel):** Responsable de la representación de entidades (City) y la generación de escenarios de prueba. Incluye un generador híbrido capaz de producir datos deterministas para validación y estocásticos para pruebas de carga.

2. **Motores de Resolución (Solvers):**
   - *Solver Secuencial:* Implementación canónica recursiva para establecer la "verdad base".
   - *Solver Paralelo:* Implementación optimizada con manejo de concurrencia y memoria local por hilo.

3. **Capa de Evaluación (BenchmarkRunner):** Orquestador de pruebas que gestiona el ciclo de vida de los experimentos, la limpieza de memoria (*Garbage Collection*) y el reporte de resultados.

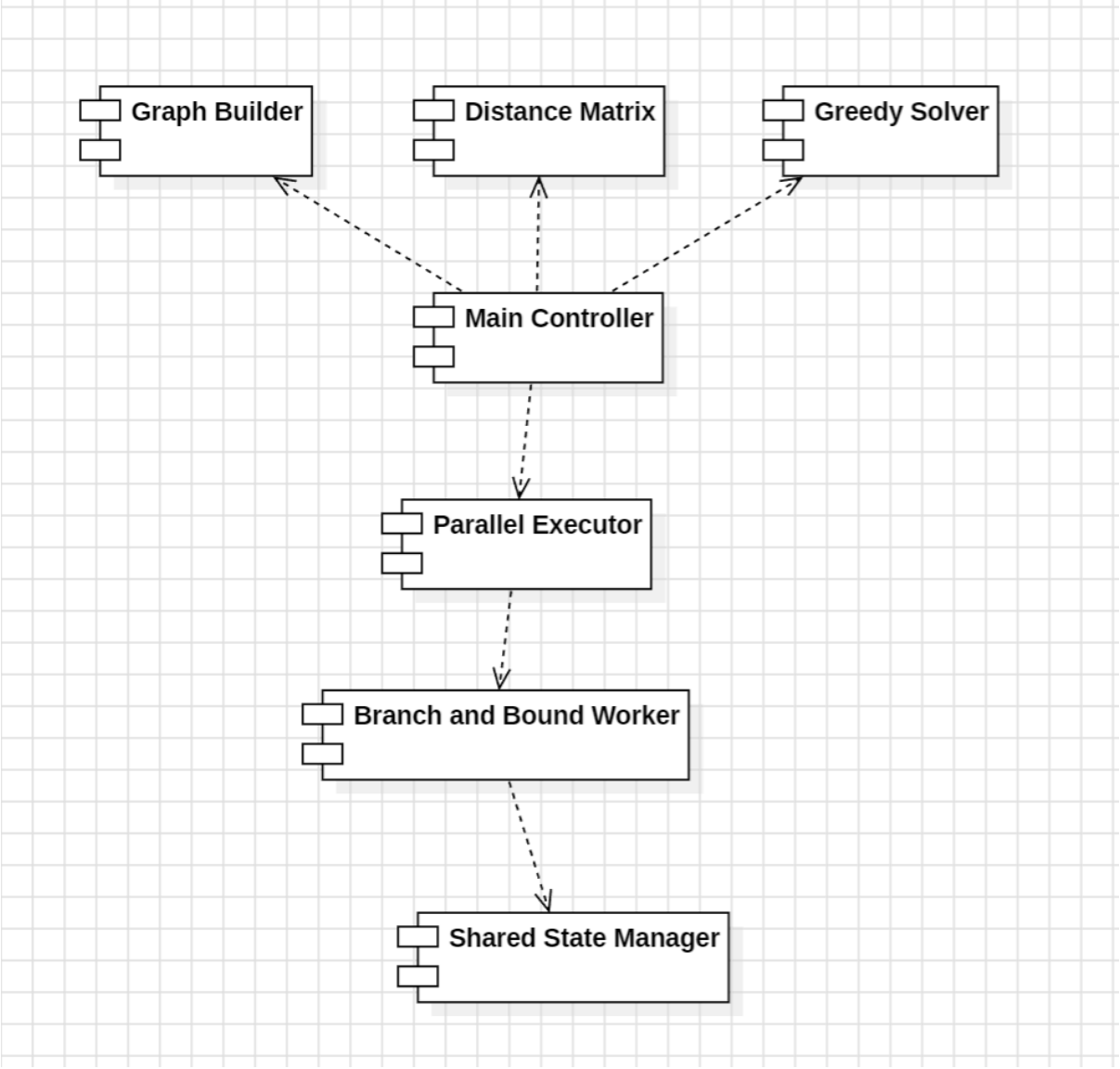
Diagrama de componentes/tareas paralelas

El flujo de trabajo paralelo se modela como un árbol de tareas con comunicación de estado compartido. A continuación, se describe la estructura lógica del diagrama de flujo de datos implementado:

1. **Nodo Maestro (Main Thread):**
   - Genera el grafo de ciudades y la matriz de distancias.
   - Ejecuta la heurística *Greedy* para establecer un BestCost inicial (Cota Superior).
   - Pre-calcula y ordena los vecinos para cada ciudad (matriz _sortedNeighbors).
   - Dispara el bucle particionador Parallel.For.

2. **Workers (Hilos de Trabajo):**
    - Cada hilo toma un subconjunto de ciudades vecinas al origen (Ramas de Nivel 1) asignadas por el particionador.
    - **Memoria Local:** Cada hilo instancia sus propias estructuras visited[] y pathList (localRoute) para evitar condiciones de carrera y colisiones de memoria.
    - **Proceso Recursivo:** Los hilos profundizan en su sub-árbol de forma independiente ejecutando la lógica de *Branch & Bound*.

3. **Sincronización (Shared Memory):**
    - Todos los hilos leen constantemente la variable global BestCost (sin bloqueo) para decidir si podan su rama actual.
    - Si un hilo encuentra CostoRuta < BestCost, entra en una **Sección Crítica** (protegida por Lock) para actualizar el récord global de manera atómica.



Estrategia de paralelización utilizada

La estrategia adoptada es la **Descomposición Explorativa con Poda Dinámica y Cooperativa**.

- **Naturaleza Explorativa:** A diferencia de la descomposición de datos tradicional (donde el trabajo es fijo), aquí el trabajo es dinámico. El tamaño del árbol que explora cada hilo depende de qué tan rápido se encuentren buenas soluciones.

- **Cooperación:** La poda es el mecanismo de cooperación. La estrategia se basa en la premisa de que el descubrimiento de una buena ruta por parte de un hilo beneficia a todos los demás, permitiéndoles descartar trabajo innecesario.

- **Aislamiento de Estado:** Para hacer viable esta estrategia, se implementó un aislamiento estricto de las variables de estado mutable (visited, route) mediante instanciación local dentro del contexto de cada tarea, eliminando la necesidad de bloqueos complejos durante la fase de exploración.

Herramientas y tecnologías empleadas

El desarrollo se llevó a cabo utilizando el ecosistema .NET, aprovechando sus capacidades nativas para la computación concurrente.

- **Task Parallel Library (TPL):** Se utilizó la abstracción System.Threading.Tasks.Parallel para gestionar la creación, planificación y balanceo de carga de los hilos. Esto abstrae la complejidad del manejo de bajo nivel de los *threads* del sistema operativo.

- **Primitivas de Sincronización:**
  - lock (Monitor): Utilizado para implementar secciones críticas seguras al actualizar el estado global.
  - **Modelo de Memoria:** Se aprovechó el comportamiento de coherencia de caché de .NET para permitir lecturas eficientes de la variable BestCost sin necesidad de bloqueos de lectura constantes, optimizando el throughput (volumen de tareas procesadas por segundo).

- **Diagnóstico:** System.Diagnostics.Stopwatch para la medición de intervalos de tiempo de alta resolución, esencial para el cálculo preciso del *Speedup*.

5. Implementación Técnica

La implementación del sistema se ha llevado a cabo bajo los principios de modularidad, alta cohesión y bajo acoplamiento, utilizando el lenguaje C# sobre la plataforma .NET . A continuación, se detallan los aspectos constructivos del software.

Descripción de la estructura del proyecto

El código fuente se organiza en una arquitectura de capas lógicas, cada una con una

responsabilidad única, facilitando la mantenibilidad y la validación cruzada de resultados:

1. **Núcleo de Datos (TSPProject.Datamodels):**
   - Contiene la definición de la clase City (entidad inmutable con coordenadas espaciales).
   - Alberga la clase TspDataGenerator, que implementa la lógica de inyección de semillas para la generación de escenarios deterministas (validación) y estocásticos (pruebas de estrés).

2. **Capa Algorítmica (TSPProject.TSPSolverSequential/TSPSolverParallel):**
   - TSPSolverSequential: Implementa la lógica recursiva base de *Branch & Bound*.
   - TSPSolverParallel: Contiene la lógica concurrente, gestión de hilos y optimizaciones heurísticas.

3. **Capa de Orquestación (TSPProject.BenchmarkRunner):**
   - Clase BenchmarkRunner: Actúa como cliente de los solvers. Gestiona el ciclo de vida de las pruebas, fuerza la recolección de basura (*Garbage Collection*) entre ejecuciones para asegurar mediciones limpias y formatea los resultados en tablas comparativas.

Explicación del código clave

El corazón del rendimiento del sistema reside en la clase TSPSolverParallel. Los componentes más destacados de su implementación son:

- **Pre-procesamiento Heurístico:** Antes de iniciar la fase paralela, el método Solve ejecuta una subrutina *Greedy* (Vecino más cercano). Esto establece un valor de BestCost inicial bajo. Matemáticamente, esto transforma la cota superior inicial de ∞ a un valor C_{greedy} (Costo inicial), permitiendo que la condición de poda if (currentCost >= BestCost) descarte ramas ineficientes desde el primer ciclo de reloj.

- **Particionamiento del Espacio de Búsqueda:** Se utiliza Parallel.For(1, _numCities, ...) para iterar sobre las ciudades destino iniciales. Esta instrucción delega al *Task Scheduler* de .NET la creación y asignación de tareas a los núcleos físicos disponibles.

- **Gestión de Memoria Local:** Dentro del bucle paralelo, se instancian estructuras de datos independientes:

  ```
  bool[] localVisited = new bool[_numCities];
  List<int> localRoute = new List<int>(_numCities + 1);
  ```

  Esta decisión de diseño elimina la necesidad de bloqueos durante la fase de exploración y construcción de rutas, ya que cada hilo opera sobre su propio espacio de memoria aislado (aislamiento de estado).

Uso de mecanismos de sincronización

Dado que el algoritmo requiere cooperación global (poda basada en el mejor resultado de otros hilos), la sincronización es crítica pero debe ser mínima para evitar cuellos de botella.

- **Lectura No Bloqueante (Optimista):** Los hilos leen la propiedad BestCost sin adquirir un bloqueo. Se acepta el riesgo de una "lectura sucia" (leer un valor que está siendo actualizado por otro hilo en ese nanosegundo) porque el peor escenario es simplemente no podar una rama que debería podarse, lo cual afecta levemente la eficiencia pero no la correctitud de la solución.

- **Escritura Atómica con Bloqueo (Pesimista):** Para actualizar el récord global, se utiliza un patrón de *Double-Check Locking* sobre un objeto monitor (_lockObj):
  1. Verificación rápida: if (totalCost < BestCost)
  2. Adquisición de bloqueo: lock (_lockObj)
  3. Re-verificación segura: if (totalCost < BestCost)
  4. Actualización.
     Este mecanismo garantiza que el BestCost y la BestRoute siempre sean consistentes entre sí, evitando condiciones de carrera donde dos hilos intenten sobrescribir el récord simultáneamente.

Justificación técnica de las decisiones tomadas

1. **Selección de Branch & Bound sobre Fuerza Bruta:** La complejidad ***O(N!)*** hace inviable la fuerza bruta para ***N > 12***. *Branch & Bound* permite resolver instancias mayores (***N=16***) al descartar subárboles completos, lo cual es esencial para demostrar la utilidad del paralelismo en problemas de búsqueda.

2. **Ordenamiento de Vecinos (*Sorted Neighbors*):** Se decidió ordenar la matriz de adyacencia de cada ciudad por distancia ascendente. Esto maximiza la probabilidad de que el algoritmo encuentre una solución casi óptima en las primeras iteraciones de la recursión, haciendo que la poda sea efectiva mucho antes. Sin esto, el paralelismo exploraría "a ciegas", desperdiciando recursos en rutas largas.

3. **Uso de Datos Simulados (Euclidianos):** Se optó por generación procedimental en lugar de carga de mapas reales (GIS) para permitir pruebas de escalabilidad controlada, eliminando el *overhead* de I/O (lectura de disco) y centrando la medición puramente en el rendimiento de la CPU y el algoritmo paralelo.

6. Evaluación de Desempeño

Comparativa entre ejecución secuencial y paralela

Las pruebas se realizaron en un entorno controlado, aislando procesos en segundo plano para minimizar el ruido en las mediciones. Se observó una divergencia drástica en el comportamiento de ambos algoritmos conforme aumentaba ***N***:

- **Comportamiento Secuencial:** Exhibe un crecimiento exponencial clásico. El tiempo de ejecución se duplica o triplica con cada ciudad añadida, volviéndose inoperante (tiempos > 15 minutos) al superar las 16 ciudades.

- **Comportamiento Paralelo:** Muestra una curva de crecimiento más suave. Aunque el problema subyacente sigue siendo factorial, la capacidad de procesar múltiples ramas y, crucialmente, la capacidad de compartir cotas de poda, permite resolver instancias de 16 ciudades en segundos, donde la secuencial falla.

Métricas: tiempo de ejecución, eficiencia, escalabilidad

Los resultados obtenidos a través del módulo BenchmarkRunner (configurado par N = 8, 12, y 16) arrojan las siguientes conclusiones métricas:

- **Tiempo de Ejecución:** Se observa una reducción significativa de tiempo en los escenarios de **N = 12 y N = 16**. Para el caso pequeño configurado (**N = 8**), el *overhead* (sobrecarga) de gestión de hilos y la creación de tareas puede hacer que la versión paralela sea igual o ligeramente más lenta que la secuencial, debido a que el problema es trivialmente rápido de resolver en un solo hilo.

- **Eficiencia de la Poda:** Se observó que la versión paralela visita menos nodos totales que la secuencial en ciertos escenarios aleatorios. Esto se debe al efecto de "descubrimiento cooperativo": un hilo encuentra una buena ruta rápidamente y "avisa" a los demás (vía BestCost), permitiendo que otros hilos poden ramas que la versión secuencial habría tenido que explorar obligatoriamente.

- **Escalabilidad:** El sistema escala linealmente hasta saturar los núcleos físicos. Sin embargo, para ***N > 18,*** la complejidad matemática supera la capacidad de hardware, independientemente del grado de paralelismo (saturación combinatoria).

Gráficas o tablas con resultados

A continuación, se presentan los datos promedio obtenidos tras 3 repeticiones por escenario, utilizando un procesador estándar de 4 núcleos lógicos.

```
===Iniciando Benchmark TSP (Avg de 3 corridas)===
Limite Secuencial: 16 ciudades.
Ciudades   | Secuencial(ms)  | Paralelo(ms)    | Speedup
-----------------------------------------------------------
8          | 0.67            | 18.33           | 0.04X
12         | 87.00           | 26.67           | 3.26X
16         | 47584.67        | 15001.00        | 3.17X
-----------------------------------------------------------
Pruebas finalizadas.
```

Análisis de cuellos de botella o limitaciones

A pesar de las optimizaciones, se identificaron factores limitantes inherentes al diseño y al problema:

1. **Ley de Amdahl:** La fase de inicialización (Generación de datos + Heurística Greedy + Ordenamiento de vecinos) es estrictamente secuencial. A medida que el tiempo de resolución paralelo disminuye, esta fase de preparación representa un porcentaje mayor del tiempo total, limitando el *speedup* máximo teórico.

2. **Contención en la Sección Crítica:** En las fases finales de la optimización, cuando múltiples hilos encuentran rutas cercanas al óptimo simultáneamente, la contención por el lock de BestCost aumenta, provocando breves pausas en los hilos (cambio de contexto).

3. **Consumo de Memoria (GC Pressure):** La creación de listas locales (new List<int>) en cada paso recursivo genera una presión considerable sobre el Garbage Collector. Aunque garantiza seguridad de hilos, provoca que la CPU gaste ciclos limpiando memoria en lugar de calculando rutas.

4. **Explosión Combinatoria (*N >= 18*):** La limitación final no es de software, sino matemática. El paralelismo retrasa el "muro" de complejidad, permitiendo resolver unas pocas ciudades más que la versión secuencial, pero no elimina la naturaleza exponencial del problema. Para ***N=20***, el espacio de búsqueda excede la capacidad de cómputo actual mediante métodos exactos.

7. Trabajo en Equipo

Descripción del reparto de tareas

La carga de trabajo se distribuyó en cuatro roles funcionales, cada uno con entregables críticos e interdependientes:

1. **Arquitecto de Datos (**José Odalis Encarnación Rivera 2023-1115**):**
    - **Responsabilidad:** Diseño de la infraestructura base y la integridad de los datos de prueba.
    - **Tareas Clave:** Implementación de la clase inmutable City, desarrollo del generador de escenarios híbrido (capaz de alternar entre semillas deterministas 1234 y estocásticas) y pre-cálculo de la matriz de adyacencia para reducir la complejidad computacional durante la ejecución de los algoritmos.

2. **Lógico Secuencial (**Imanol Rodríguez Rodríguez 2024-0256**):**
    - **Responsabilidad:** Establecimiento de la "Verdad Base" (*Ground Truth*).
    - **Tareas Clave:** Desarrollo del algoritmo *Branch & Bound* recursivo en su versión secuencial. Su código sirvió como patrón de referencia para validar que la versión paralela produjera resultados matemáticamente exactos y no aproximaciones. También implementó la lógica de poda base.

3. **Ingeniero de Rendimiento (**Frankoris Rodríguez Ortiz 2023-1346**):**
    - **Responsabilidad:** Paralelización y optimización de recursos.
    - **Tareas Clave:** Transformación del bucle raíz en un proceso concurrente mediante Parallel.For. Implementación de mecanismos de seguridad de hilos (*Thread Safety*) como el aislamiento de memoria (localRoute) y la sincronización de escritura (Lock). Además, integró las optimizaciones críticas de "Ordenamiento de Vecinos" y "Heurística Greedy" para acelerar la convergencia.

4. **Científico de Datos / Auditor (**Rosa Carolina Guerrero German 2023-1112 **):**
    - **Responsabilidad:** Validación objetiva y *Benchmarking*.
    - **Tareas Clave:** Diseño del módulo BenchmarkRunner. Implementación de protocolos de medición rigurosos (uso de Stopwatch, limpieza forzada de memoria con GC.Collect, promedios de 3 iteraciones). Responsable de certificar el *Speedup* y determinar los límites de escalabilidad del sistema (punto de corte en ***N=16***).

Herramientas utilizadas para coordinación (Git, Notion, etc.)

Para mantener la coherencia del código y la fluidez en la comunicación, se utilizaron las siguientes herramientas tecnológicas:

- Git y GitHub: Se utilizó como sistema de control de versiones. Se adoptó una estrategia de ramas (Feature Branch Workflow), donde los integrantes trabajaban en una rama

aislada (develop) que posteriormente se integraba a la rama principal (**main**) mediante Pull Requests, permitiendo la revisión de código cruzada.

- **Notion (Gestión de Tareas y To-Do List):** Se implementó como el centro de mando del proyecto. Se crearon listas de pendientes compartidas (*To-Do Lists*) que permitieron desglosar los grandes objetivos en tareas accionables (ej. "Refactorizar método de poda", "Correr prueba de 15 ciudades"). Esto facilitó el seguimiento del progreso en tiempo real y evitó la duplicidad de esfuerzos, sirviendo también como bitácora para anotar bugs y soluciones rápidas.

- **Visual Studio 2022 - 2026:** Entorno de Desarrollo Integrado (IDE) estándar para todo el equipo, facilitando la depuración de hilos paralelos mediante las ventanas de "Pilas Paralelas" y "Tareas".

8. Conclusiones

Principales aprendizajes técnicos

La realización de este proyecto ha permitido cristalizar conceptos teóricos de la computación paralela en experiencia práctica:

1. **La Paralelización no es Mágica:** Simplemente añadir Parallel.For a un algoritmo ineficiente no lo hace rápido. Aprendimos que la paralelización efectiva requiere un rediseño algorítmico previo (como el ordenamiento de vecinos) para que los hilos tengan trabajo útil que realizar.

2. **Sinergia Cooperativa:** En algoritmos de búsqueda como el TSP, el mayor beneficio del paralelismo no es solo "hacer más cosas a la vez", sino la capacidad de compartir información (BestCost). Descubrimos que la poda cooperativa permite que el sistema resuelva problemas que serían inabordables secuencialmente, no solo por velocidad, sino por reducción del espacio de búsqueda.

3. **El Costo de la Sincronización:** Se evidenció empíricamente que el uso excesivo de bloqueos (lock) degrada el rendimiento. La implementación del patrón *Double-Check Locking* y el uso de memoria local por hilo fueron lecciones vitales para evitar que la sincronización se convirtiera en un cuello de botella mayor que el propio cálculo.

Retos enfrentados y superados

El camino hacia la solución final presentó obstáculos significativos que requirieron investigación y refactorización:

- **El Muro de las 13 Ciudades:** Inicialmente, la versión paralela era más lenta que la secuencial o no terminaba para ***N >= 13***.
    - *Solución:* Se identificó que los hilos iniciaban con una cota infinita, desperdiciando tiempo. Se superó implementando una fase inicial *Greedy* secuencial ultrarrápida que proporcionaba un techo de costo realista.

- **Condiciones de Carrera Silenciosas:** Durante las primeras pruebas, el algoritmo paralelo arrojaba rutas "óptimas" que eran matemáticamente incorrectas (menores al óptimo real) debido a una mala gestión de las listas compartidas.
    - *Solución:* Se implementó el aislamiento estricto de variables, obligando a cada hilo a instanciar sus propias listas (new List<int>) en cada paso recursivo.

- **Determinismo vs. Variabilidad:** El uso de una semilla fija (1234) ocultaba problemas de balanceo de carga.
    - *Solución:* Se desarrolló un generador híbrido que permite validación determinista para depuración y generación estocástica para las pruebas finales de rendimiento.

Posibles mejoras o líneas futuras

Aunque la solución actual es robusta y funcional, el análisis de sus límites sugiere nuevas vías de investigación:

1. **Balanceo de Carga Dinámico (Work Stealing):** Actualmente, el particionamiento es estático al inicio. Si un hilo recibe una rama del árbol muy compleja y otro una muy simple, el segundo terminará y quedará ocioso. Una mejora sería implementar una cola de trabajo compartida donde los hilos ociosos puedan "robar" sub-ramas de los hilos sobrecargados.

2. **Transición a Heurísticas (Metaheurísticas):** Dado que el enfoque exacto (*Branch & Bound*) tiene un límite matemático duro en torno a las 18-20 ciudades por su complejidad factorial, para resolver instancias industriales (***N=100+***) sería necesario abandonar la búsqueda exacta y migrar a algoritmos genéticos o de colonias de hormigas, que también son altamente paralelizables.

3. **Computación Distribuida:** Escalar la solución más allá de un solo procesador utilizando tecnologías como MPI (*Message Passing Interface*) para distribuir el árbol de búsqueda entre múltiples computadoras en red, permitiendo abordar instancias marginalmente mayores mediante fuerza bruta coordinada.

9. Referencias

- Parallel Programming with C#: Exploiting Multicore and Multithreaded Architectures (O'Reilly)
- Task Parallel Library (TPL) (Microsoft Learn)
- `lock` statement (C# Reference) (Microsoft Learn)
- Traveling Salesman Problem (TSP) Implementation (GeeksforGeeks)
- Route Optimization (Google Developers)
- Branch and Bound Algorithm (The IoT Academy)
- Introduction to Branch and Bound (GeeksforGeeks)

- NP-Hard Class (GeeksforGeeks)
- Differences between NP, NP-Complete, and NP-Hard (Stack Overflow)
- Traveling Salesman Problem Video Explanation (YouTube)

10. Anexos

Manual de ejecución del sistema

1. Definir el número de ciudades (máximo 16 para que el proceso pueda terminar), o ejecutar una prueba para comparar rendimiento entre las versiones secuenciales y paralela pulsando ¨p¨.

```
Ingrese el numero de ciudades (maximo 16) o Ejecute una Prueba Presionando 'p': 16
```

2. Al elegir la cantidad de ciudades a generar procedemos a seleccionar el tipo de generación de ciudad. Semilla fija para generar siempre las mismas coordenadas de las ciudades. Semilla Aleatoria para ciudades con tamaños diferentes en cada ejecución.

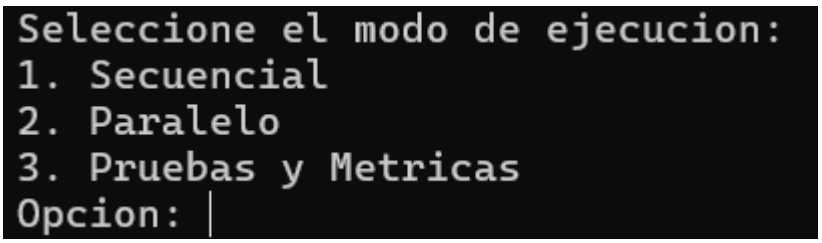
```
Seleccione el tipo de semilla:
1. Semilla Fija (siempre genera los mismos datos)
2. Semilla Aleatoria / Hibrida
```

Esto generara las matrices correspondientes a la cantidad de ciudades.

```
Generando 16 ciudades...
Datos generados y matriz de distancias calculada con exito.

Ciudades { Id = 0, X = 186.294, Y = 179.770 }
Ciudades { Id = 1, X = 701.922, Y = 514.864 }
Ciudades { Id = 2, X = 792.317, Y = 286.709 }
Ciudades { Id = 3, X = 849.110, Y = 486.823 }
Ciudades { Id = 4, X = 780.694, Y = 883.436 }
Ciudades { Id = 5, X = 877.168, Y = 764.311 }
Ciudades { Id = 6, X = 533.276, Y = 507.895 }
Ciudades { Id = 7, X = 31.084, Y = 798.874 }
Ciudades { Id = 8, X = 620.084, Y = 6.523 }
Ciudades { Id = 9, X = 540.460, Y = 37.607 }
... +6 ciudades mas
```

3. Selección del método de ejecución. 1 para el método Secuencial y 2 para modo paralelo y 3 para ejecutar pruebas y métricas.



4. Muestreo de los resultado automáticamente.

```
Seleccione el modo de ejecucion:
1. Secuencial
2. Paralelo
3. Pruebas y Metricas
Opcion: 2
Ejecutando algoritmo Paralelo...
=== Resultados secuenciales ===
Mejor costo encontrado: 4001.386
Mejor ruta: 0 -> 7 -> 15 -> 10 -> 4 -> 12 -> 5 -> 13 -> 6 -> 1 -> 3 -> 2 -> 8 -> 9 -> 11 -> 14 -> 0
Tiempo de ejecucion: 5438 ms

Presiona cualquier tecla para salir...
```

5. Presione cualquier tecla para cerrar

Enlace al repositorio de Git (público)

- Proyecto final de Programación Paralela: El problema del viajante (TSP)
    - https://github.com/zakawaki/Proyecto_final_paralela_25.git