Informe Laboratorio 3

Alondra Acosta, Rol: 202273524-4
Sergio Cárcamo, Rol: 202273512-0

4 de noviembre de 2025

1. Introducción

El presente informe detalla el desarrollo del Laboratorio 3 para el curso de Sistemas Operativos **(INF246)**. El problema central consiste en encontrar el camino más corto dentro de un grafo ponderado, que simula la planificación de rutas en un volcán. Dado que analizar todas las combinaciones de rutas puede ser un proceso computacionalmente costoso, se requiere el uso de paralelismo.

El objetivo de la tarea es implementar dos versiones del programa para resolver este problema y comparar su eficiencia. La primera versión se debe desarrollar en **C++**, utilizando la llamada al sistema `fork()` para la creación de procesos hijos y `pipes` para la comunicación entre ellos. La segunda versión se debe implementar en **Java**, empleando `threads` para la paralelización.

Para ambas soluciones se utilizó el algoritmo de **Bellman-Ford**, como fue sugerido por el enunciado, permitiendo así una comparación justa del rendimiento de ambas estrategias. A continuación, se detallará el desarrollo de cada implementación y se presentará un análisis comparativo de los resultados obtenidos.

2. Desarrollo

2.1. Implementación en C++

Tal como se sugirió en el enunciado, se usó el algoritmo de **Bellman-Ford**. Este algoritmo sirve para encontrar las distancias más cortas desde un vértice origen a todos los demás en un grafo ponderado, incluso si hay pesos negativos (siempre que no existan ciclos negativos alcanzables).

La paralelización se implementó usando la llamada al sistema `fork()` para crear procesos hijos, como se especificaba en los requisitos. Se genera un número de hijos igual al total de procesadores lógicos del sistema, obtenido mediante `sysconf(_SC_NPROCESSORS_ONLN)`, en nuestro caso, 12.

La comunicación y sincronización entre el padre y los hijos se gestiona íntegramente con `pipes`. Por cada hijo, el padre crea dos canales de comunicación (pipes) dedicados:

- Un pipe "Padre-a-Hijo"(P→H), donde el padre escribe y el hijo lee.
- Un pipe "Hijo-a-Padre"(H→P), donde el hijo escribe y el padre lee.

El algoritmo opera en fases sincronizadas que se repiten V veces (donde V es el número de nodos):

1. **Fase 1: Broadcast (Padre)**: El padre envía (escribe) el estado *completo* y actual del arreglo de `distancias` a *todos* sus hijos a través de sus respectivos pipes P→ H.

2. **Fase 2: Trabajo (Hijos)**: Cada hijo se bloquea (`read()`) esperando recibir el arreglo. Una vez recibido, itera sobre **su propio** subconjunto de aristas (el pedazo que el padre le asignó. Si un hijo encuentra una relajación (una ruta más corta), no la aplica; en su lugar, la añade a una lista local de propuestas (el struct `Actualizacion`).

3. **Fase 3: Reporte (Hijos)**: Al terminar de revisar sus aristas, cada hijo le informa al padre cuántas actualizaciones encontró (escribiendo un `int`). Si este número es mayor a cero, procede a escribir el arreglo de sus `Actualizacion` propuestas en el pipe H-¿P.

4. **Fase 4: Consolidación (Padre)**: El padre, en un bucle *serial*, se bloquea leyendo (`read()`) los resultados de cada hijo. Recolecta todas las propuestas de todos los hijos y es el **único** responsable de consolidarlas, aplicando solo las mejores (las de menor distancia) a su arreglo maestro de `distancias`.

Este ciclo de cuatro fases se repite hasta completar las V iteraciones (necesario para detectar ciclos negativos) o hasta que una iteración completa no produce ningún cambio global (early exit).

Al finalizar, el padre cierra los extremos de escritura de los pipes. Esto envía una señal EOF a los hijos que están bloqueados leyendo, indicándoles que deben terminar su bucle y salir. Finalmente, el padre ejecuta `waitpid()` por cada hijo para limpiar los procesos zombie y escribe el resultado final en `salidaFork.txt`.

2.2. Implementación en Java

Para la versión en Java, se siguió la especificación de usar Threads para la paralelización. Se mantuvo el mismo algoritmo de **Bellman-Ford** para asegurar una comparación consistente.

A diferencia de C++ (que usa memoria separada para cada proceso), en Java todas las hebras comparten el mismo espacio de memoria, lo que presenta un desafío de concurrencia para acceder a los arreglos compartidos de `distancias` y `predecesores`.

La estrategia de paralelización fue similar a la de C++: el hilo principal ('main') divide la lista total de aristas y asigna un subconjunto a cada hilo 'Worker'. Se creó una clase interna `Worker` que implementa `Runnable`. El número de hebras se define dinámicamente según los procesadores disponibles (`Runtime.getRuntime().availableProcessors()`), en nuestro caso, 12.

Para evitar condiciones de carrera, se implementó un mecanismo de sincronización usando barreras (CyclicBarrier), tal como se sugería en el enunciado. El algoritmo opera en fases sincronizadas por dos barreras:

- **Fase de Sincronización (Inicio)**: Todas las hebras (workers y main) deben llegar a la `startBarrier.await()`. Esto asegura que ningún hilo comience a trabajar hasta que todos estén listos para la iteración.

- **Fase de Trabajo (Paralela)**: Una vez liberados, cada hilo 'Worker' revisa su subconjunto de aristas. Leen del arreglo `distancias` compartido (lo cual es seguro) y, si encuentran una mejora, la guardan en una lista *local* `(misActualizaciones)`, sin modificar aún el arreglo global.
- **Fase de Sincronización (Fin)**: Al terminar de revisar sus aristas, todas las hebras (workers y main) esperan en la `endBarrier.await()`.
- **Fase de Consolidación (Serial)**: Una vez que todos los workers han llegado a la barrera final, el hilo 'main' se activa. Este hilo es el **único** que tiene permiso para escribir en los arreglos compartidos. Recorre las listas `misActualizaciones` de todos los workers y aplica las mejoras en `distancias` y `predecesores`.

Este ciclo (Inicio → Trabajo → Fin → Consolidación) se repite V veces (o se detiene antes si no hay cambios). Este enfoque garantiza que nunca ocurran escrituras concurrentes en la memoria compartida.

Finalmente, el hilo principal espera que todos los workers terminen (join()) y escribe los resultados en el archivo `salidaThread.txt`.

2.3. Entorno de implementación y pruebas

Las implementaciones y pruebas de los programas se realizaron bajo dos equipos distintos. Siendo la especificación tecnica de ellos, detallada a continuación:

Computador 1 - Implementación Java

- Procesador: 13th Gen Intel(R) Core(TM) i5-1335U, 1300 MHz (1.30 GHz) base, 10 procesadores principales, 12 procesadores lógicos.
- Memoria RAM: 16,0 GB.
- Almacenamiento: SSD NVMe, marca SAMSUNG MZVL41T0HBLB-00B, con una capacidad total de 954 GB.
- Sistema operativo: Ubuntu 22.04.5 LTS
- Compilador: g++ 11.4.0
- Arquitectura x64 bits

Computador 2 - Implementación C++

- Procesador: 13th Gen Intel(R) Core(TM) i5-1335U, 1300 MHz (1.30 GHz) base, 10 procesadores principales, 12 procesadores lógicos.
- Memoria RAM: 16,0 GB.
- Almacenamiento: SSD NVMe, marca Micron MTFDKBA512QGN-1BN1AABGA, con una capacidad total de 512 GB.
- Sistema operativo: Ubuntu 24.04.3 LTS (WSL2) bajo Microsoft Windows 11 Enterprise IoT LTSC
- Compilador: g++ 11.4.0
- Arquitectura x64 bits

2.4. Comparación de Resultados

Una vez obtenidos los datos, podemos generar gráficos usando el script **script_plots.py**.

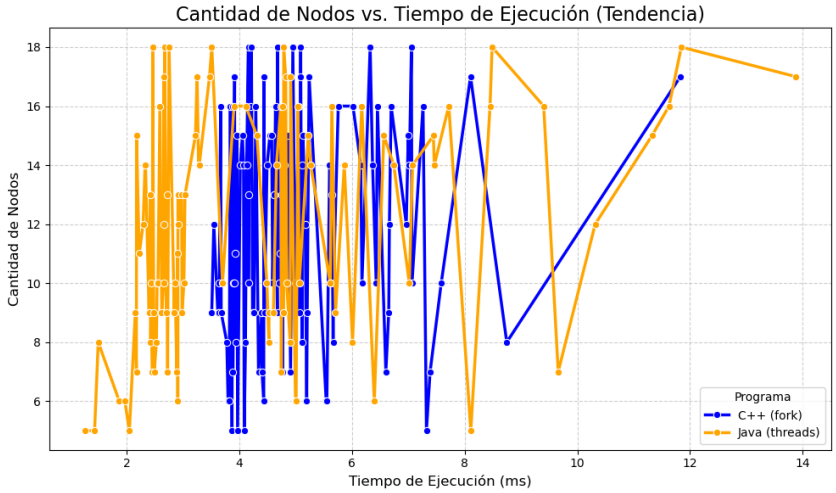

Figura 1: Comparación de Nodos vs Tiempo de ejecución

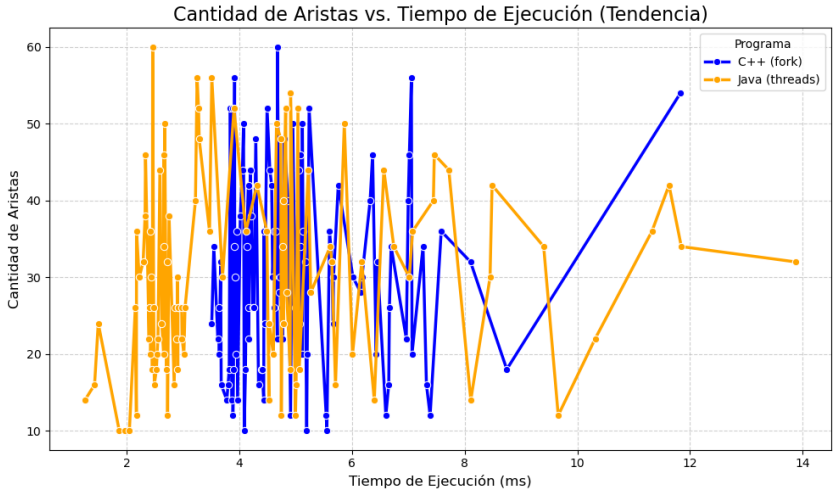

Figura 2: Comparación de Aristas vs Tiempo de ejecución

Como podemos ver en los gráficos 1 y 2, que comparan el tiempo de ejecución (eje X) contra el tamaño del problema, emerge una tendencia clara a pesar de la variabilidad en los datos.

En ambos gráficos, la línea azul (**C++ (fork)**) se mantiene consistentemente a la izquierda de la línea naranja (**Java (threads)**) para un tamaño de problema similar. Esto indica que, para un grafo con una cantidad dada de nodos o aristas, la implementación en C++ con `fork()` logra completar el cálculo en un **tiempo de ejecución menor** que la implementación en Java con `threads`.

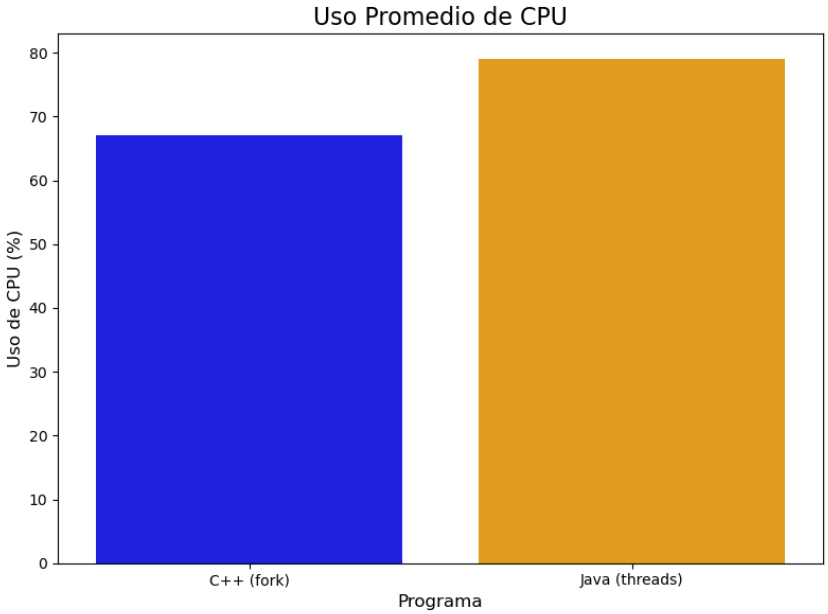

Figura 3: Comparación uso de CPU

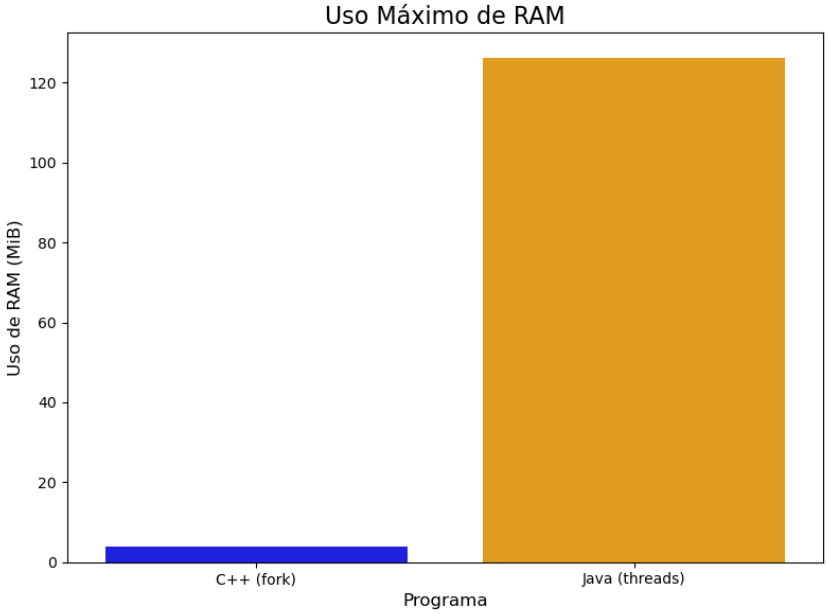

Figura 4: Comparación uso de memoria RAM

Luego, al analizar el uso de recursos, las diferencias se vuelven aún más evidentes.

El gráfico 3 muestra el uso promedio de CPU. Aquí, Java (threads) registra un uso de CPU ligeramente *mayor* (aprox. 79 %) que C++ (fork) (aprox. 67 %). Esto es notable porque, a pesar de usar más recursos de CPU, Java obtuvo un rendimiento en tiempo *inferior*. Esto sugiere que una parte significativa de ese uso de CPU en Java se debe a la sobrecarga (overhead) del propio entorno de ejecución (la JVM), su recolector de basura (Garbage Collector) y la gestión de la sincronización de hebras (barreras).

Finalmente, el gráfico 4 presenta la diferencia más drástica: el **Uso Máximo de RAM**. La versión de **Java (threads) consume** ≈ 125 **MiB** de RAM, mientras que la versión de **C++ (fork) utiliza una fracción de eso,** $\approx 4 - 5$ **MiB**.

Esta diferencia abismal se debe a dos factores fundamentales:

- **Java (threads)**: Las hebras se ejecutan dentro de la Máquina Virtual de Java (JVM). El alto consumo de memoria no proviene de las hebras en sí, sino de la propia JVM, que debe cargar el entorno de ejecución, el compilador JIT (Just-In-Time) y el recolector de basura, reservando un espacio de memoria (heap) considerable antes de ejecutar la primera línea del programa.
- **C++ (fork)**: Los procesos creados con `fork()` en sistemas operativos modernos (como Linux) se benefician enormemente de la optimización **Copy-on-Write (CoW)**. Esto significa que, al hacer `fork()`, el hijo no duplica inmediatamente toda la memoria del padre. En su lugar, comparte las páginas de memoria del padre. Solo si el hijo intenta *escribir* en una de esas páginas, el kernel crea una copia privada de esa página específica. Dado que nuestros hijos pasan la mayor parte del tiempo *leyendo* los datos del grafo y solo escriben pequeños mensajes de actualización a través de los `pipes`, la duplicación de memoria real es mínima, resultando en un uso de RAM extremadamente eficiente.

En resumen, los datos muestran que la implementación de C++ (fork) superó a Java (threads) en todas las métricas clave: fue **significativamente más rápida** y consumió **drásticamente menos memoria RAM**.

3. Preguntas de análisis

- **¿Cuál de las dos implementaciones tuvo un mejor rendimiento en términos de tiempo de ejecución? ¿A qué crees que se debe esto?**

  Respuesta: La implementación en **C++ (fork)** tuvo un rendimiento en tiempo de ejecución significativamente mejor. Como se observa en las Figuras 1 y 2, para un mismo tamaño de grafo (misma cantidad de nodos o aristas), la línea azul de C++ se ubica consistentemente en un tiempo de ejecución menor (más a la izquierda) que la línea naranja de Java.

  Esto se debe a varias razones:

  - **Sobrecarga (Overhead) de Ejecución**: C++ se compila a código máquina nativo, que se ejecuta directamente por el SO. Java se ejecuta sobre la Máquina Virtual de Java (JVM), la cual introduce una capa de abstracción que consume más tiempo tiempo (compilación JIT), gestión de memoria (Garbage Collector) y arranque.
  - **Eficiencia del Modelo de Procesos**: Aunque `fork()` es conceptualmente "pesado", los sistemas operativos modernos como Linux (que es el que nos piden en enunciado) usan **Copy-on-Write (CoW)**. Esto significa que la memoria no se duplica al crear un hijo, sino que se comparte. La duplicación solo ocurre si un hijo *escribe* en una página de memoria. Dado que nuestro algoritmo es principalmente de lectura (los hijos solo leen el grafo y escriben pequeñas actualizaciones en un pipe), la penalización de `fork()` es mínima.

- **Sincronización**: La implementación de Java utiliza `CyclicBarrier`, que fuerza a todas las hebras a detenerse y esperar en dos puntos por cada iteración del algoritmo. Esta sobrecarga de sincronización, aunque necesaria para la correctitud, suma un costo de espera en cada una de las V iteraciones.

- **¿Se observó alguna diferencia significativa en el uso de recursos del sistema (CPU, RAM) entre ambas soluciones?**

  Respuesta: Sí, las diferencias fueron extremadamente significativas, como se ilustra en las Figuras 3 y 4.

  - **RAM**: La diferencia es drástica. Java (threads) consumió ≈ 125 MiB de RAM, mientras que C++ (fork) usó menos de 5 MiB. Esto se debe a que el consumo de Java está dominado por la **JVM**, que reserva un gran espacio de memoria (heap) para su funcionamiento y el Garbage Collector. En contraste, el bajo consumo de C++ demuestra la eficiencia de la optimización **Copy-on-Write (CoW)** explicada en la pregunta anterior.
  - **CPU**: Curiosamente, Java utilizó *más* CPU (aprox. 79 %) que C++ (aprox. 67 %), a pesar de ser más lento. Esto sugiere que un porcentaje considerable del uso de CPU de Java se destina a tareas de *overhead* (como el GC y la compilación JIT) y no directamente al cómputo del algoritmo Bellman-Ford.

- **¿En qué escenarios consideras más adecuado el uso de procesos (forks) frente a hilos (threads)?**

  Respuesta: La elección depende de la necesidad de aislamiento frente a la necesidad de comunicación:

  Usar Procesos (forks) es más adecuado para:

  - **Aislamiento y Robustez**: Cuando las tareas son independientes y la falla de una (un crasheo) no debe afectar a las demás. Un ejemplo clásico es un servidor web que crea un proceso por cada solicitud de cliente.
  - **Seguridad**: Cuando se necesita ejecutar código que no es de confianza o que debe tener privilegios separados, ya que el espacio de memoria aislado del proceso actúa como una barrera de seguridad.
  - **Tareas de Lectura-Intensiva)**: Como se demostró en este laboratorio, si las tareas paralelas principalmente leen de un gran conjunto de datos común, la optimización Copy-on-Write hace que `fork()` sea extremadamente eficiente en memoria.

  Usar Hilos/Hebras (threads) es más adecuado para:

  - **Comunicación Intensiva**: Cuando las tareas paralelas necesitan compartir y *modificar frecuentemente* grandes estructuras de datos. La memoria compartida por defecto de las hebras es mucho más rápida para esto que los pipes o sockets.
  - **Alto Paralelismo de E/S**: En escenarios donde se necesita gestionar miles de conexiones simultáneas (ej. un servidor de chat) que pasan la mayor parte del tiempo bloqueadas esperando datos. Crear un proceso por cada conexión sería inviable.

- **Creación y Destrucción Rápida**: Las hebras son más ligeros y rápidos de crear y destruir que los procesos (incluso con CoW).

- **¿Qué estrategias de optimización aplicarías al programa con menor rendimiento para que iguale o supere la eficiencia del programa mejor evaluado?**

  Respuesta: El programa con menor rendimiento fue el de **Java (threads)**. El principal cuello de botella es el modelo de sincronización: (trabajo paralelo) → (barrera) → (consolidación serial por main) → (barrera). Para optimizarlo, podríamos:

  - **Usar Operaciones Atómicas (Lock-Free)**: En lugar de usar barreras y una fase de consolidación serial, podríamos usar un `AtomicLongArray` para las `distancias`. Las hebras intentarían actualizar las distancias de forma concurrente usando operaciones `compareAndSet()`. Esto eliminaría el cuello de botella serial, ya que todas las hebras podrían leer y escribir al mismo tiempo sin bloquearse, aunque complicaría significativamente la lógica para detectar el fin de una iteración y los ciclos negativos.
  - **Reducir la Presión sobre el Garbage Collector (GC)**: En la implementación actual, se crea un nuevo objeto `Actualizacion` (un `record`) por cada mejora encontrada en cada iteración. En grafos densos, esto podría generar millones de objetos pequeños, forzando al GC a trabajar constantemente (aumentando el uso de CPU y pausas). Una optimización sería usar "pools"de objetos o arrays de tipos primitivos para comunicar las actualizaciones, evitando la creación de nuevos objetos dentro del bucle principal.
  - **Tuning de la JVM**: A un nivel más avanzado, se podría experimentar con los parámetros de la JVM, como el tipo de Garbage Collector (ej. usar G1GC o ZGC para pausas más cortas) o los tamaños de memoria (-Xms, -Xmx), para ajustar el rendimiento a la carga de trabajo específica del algoritmo Bellman-Ford.

4. Conclusión

Los resultados de las pruebas fueron concluyentes: la implementación en **C++ utilizando procesos (`fork`)** demostró ser superior a la implementación en **Java con hebras (`threads`)** en todas las métricas evaluadas. La versión en C++ fue significativamente más rápida en tiempo de ejecución y consumió una fracción de la memoria RAM.

La eficiencia de C++ se atribuye a su compilación a código nativo y, fundamentalmente, a la optimización **Copy-on-Write (CoW)** del sistema operativo al gestionar los procesos hijos. Por el contrario, el rendimiento de Java se vio penalizado por la sobrecarga inherente de la Máquina Virtual de Java (JVM), su recolector de basura, y la gestión de la sincronización de hebras.

Concluimos que, para este problema de cómputo intensivo con datos compartidos que son principalmente de lectura, el modelo de paralelismo basado en procesos con `fork()` resultó ser una solución mucho más eficiente que el modelo basado en hebras.