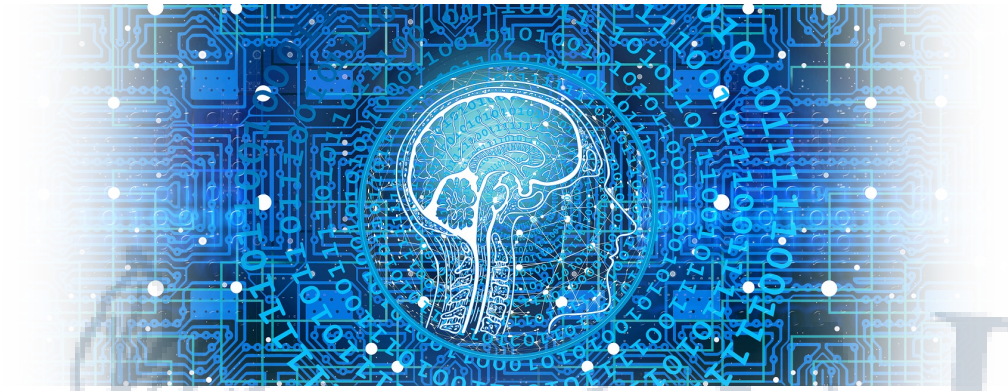

Trabajos de la asignatura

Programación Avanzada en Bionformática

Grado en Ingeniería de la Salud

Curso 2012-13

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

Prólogo

Este libro, editado por el Prof. Alberto G. Salguero Hidalgo, contiene los trabajos realizados por los alumnos de la asignatura Programación Avanzada en Bioinformática, del Grado en Ingeniería de la Salud de la Universidad de Málaga.

Los alumnos son plenos responsables de su contribución al libro y conservan todos los derechos de autoría del contenido de sus respectivos capítulos.

Índice general

Capítulo 1

Paralelización en GPU de algorítmos de búsqueda de caminos óptimos.

Diego De Pablo

1.1. Resumen

Este trabajo explora la viabilidad de la paralelización para la búsqueda de caminos óptimos en grafos, utilizando como caso de estudio el algoritmo de Dijkstra. El objetivo principal es evaluar la eficiencia de una implementación paralela en Java comparandola con su versión secuencial original, centrándose en el análisis del tiempo de ejecución y la precisión de los resultados.

La recurrencia, inherente a la naturaleza de los grafos, presenta un desafío computacional significativo al tratar con problemas de gran escala. La paralelización en GPU emerge como una alternativa prometedora para abordar este reto, aprovechando la arquitectura altamente paralela de las GPU para acelerar el proceso de búsqueda de caminos. Si bien en este trabajo se centrará tanto en la parte teórica como la implementación de lo visto, la amplitud y complejidad del tema han hecho necesario enfocarse en el algoritmo de Dijkstra, dejando otros aspectos para futuras investigaciones.

Los resultados obtenidos revelaron que la implementación paralela en Java ofrece mejoras significativas en el tiempo de ejecución al hablar de grafos grafos completos o con un número elevado de nodos. En el caso de grafos con menor densidad o complejidad, la implementación paralela no mostró una ventaja significativa sobre la versión secuencial. No obstante, los resultados generales confirman el potencial de la paralelización para optimizar la búsqueda de caminos óptimos en grafos.

En esencia, la paralelización en grafos consiste en dividir el grafo original en subgrafos más pequeños y manejables. Esto permite distribuir la carga computacional entre múltiples procesadores, lo que puede traducirse en una reducción significativa del tiempo de ejecución, especialmente para problemas de gran escala.

Palabras clave: Paralelismo, Dijkstra, Búsqueda de caminos más cortos, Grafo, Java, GPU.

1.2. Introducción

Los grafos, estructuras matemáticas que representan relaciones entre entidades, han trascendido su rol tradicional en las matemáticas para convertirse en un lenguaje universal capaz de modelar y analizar sistemas complejos en una amplia gama de disciplinas. Desde la logística y la planificación de rutas hasta la optimización de diversas ramas de la bioinformática, su versatilidad los convierte en una herramienta invaluable para abordar problemas desafiantes en la era de la información.

En el ámbito de la optimización, una de las tareas más relevantes es la búsqueda de caminos óptimos entre pares de nodos en un grafo. Estos caminos, que representan las rutas más eficientes para conectar dos puntos, la literatura científica ofrece diversos algoritmos para la búsqueda de caminos óptimos en grafos. Entre los más conocidos se encuentran los algoritmos de Dijkstra, a* (Estrella), Floyd y Dantzig, descritos y comparados por Minieka (1978). Estos algoritmos proporcionan soluciones eficientes para encontrar los caminos más cortos entre pares de nodos, lo que los convierte en herramientas valiosas para resolver problemas de optimización en diversos campos.

A pesar de ello, como indica Harish and Narayanan (2007), la aplicación de estos algoritmos secuenciales sobre grafos de gran tamaño presenta un reto computacional significativo. La complejidad computacional de estos algoritmos crece exponencialmente con el número de nodos y aristas del grafo, lo que puede traducirse en tiempos de ejecución prohibitivamente largos para problemas de gran escala. En este contexto, la computación paralela surge como una alternativa prometedora para abordar este desafío. La computación paralela, que distribuye la carga computacional entre múltiples procesadores, permite obtener mejoras significativas en el rendimiento de algoritmos computacionalmente intensivos, como los algoritmos de búsqueda de caminos óptimos en grafos.

Las Unidades de Procesamiento Gráfico (GPU), inicialmente concebidas para la rápida generación de imágenes en gráficos 3D, han experimentado una notable evolución, transformándose en coprocesadores altamente paralelos con un enorme potencial para la computación general. A diferencia de los procesadores centrales tradicionales (CPU), las GPU poseen una arquitectura altamente paralela que les permite ejecutar múltiples tareas de forma simultánea. Esta característica las convierte en herramientas ideales para acelerar algoritmos que se benefician de la paralelización, como los de búsqueda de caminos óptimos en grafos. Tal como lo expresa Che et al. (2008) demostraron que el uso de GPU para la búsqueda de caminos óptimos en grafos puede proporcionar mejoras significativas de rendimiento en comparación con las implementaciones secuenciales en CPU. Esta capacidad de aceleración se debe a la arquitectura paralela de las GPU, que permite distribuir la carga computacional entre miles de núcleos de procesamiento, lo que reduce significativamente el tiempo de ejecución de los algoritmos.

Al igual que otros capítulos de este libro, este se centra en evaluar la eficacia de la paralelización en un área específica de estudio: la búsqueda de caminos óptimos. Para ello, analizaremos en profundidad el algoritmo de Dijkstra y otros algoritmos de búsqueda de caminos óptimos, explorando las posibilidades de paralelización en cada uno de ellos.

Nuestro objetivo no solo es evaluar la viabilidad de la paralelización en este campo, sino también servir de inspiración y contribuir al avance de la comprensión de esta técnica en el contexto de la computación con GPU y el uso de grafos. Aspiramos a proporcionar información valiosa que impulse el desarrollo de nuevas técnicas de optimización más eficientes y escalables.

1.3. Tema de Estudio

Según el autor Wilson (1996)), al modelar un sistema como un grafo, las entidades que lo componen se representan como nodos y las relaciones entre estas entidades se representan como arcos que conectan los nodos. Esta representación gráfica permite analizar las propiedades del sistema de manera eficiente y estudiar las interacciones entre sus componentes.

Una forma de emplear los grafos son los algoritmos de búsqueda de caminos más cortos entre pares de nodos. Estos algoritmos son capaces de identificar la ruta más eficiente para conectar dos puntos en un grafo. El significado de çamino óptimo"puede variar según el sistema que se modele. Para una comerciante, por ejemplo, Minieka (1978) señala que este camino se traduce como el conjunto de transacciones que genera el mayor beneficio económico. En caso de ser un vendedor ambulante, se convierte en la ruta de ventas que maximiza sus ganancias. Y para una aerolínea es el camino que se traduce en la ruta de vuelo que minimiza el tiempo en el aire.

En el vasto universo de la teoría de grafos, el algoritmo de Dijkstra se erige como un faro que guía a través del complejo laberinto de nodos y arcos, iluminando el camino más corto entre dos puntos. Esta herramienta matemática, desarrollada por el informático holandés Edsger Dijkstra en 1956, se ha convertido en un pilar fundamental para la resolución de problemas de búsqueda y optimización en diversos campos.

El algoritmo de Dijkstra, también conocido como algoritmo de etiquetas, Cassingena (2022) explica como el algoritmo opera de manera sistemática, explorando el grafo paso a paso. Comienza seleccionando un nodo inicial (punto de partida) y lo marca como visitado. A continuación, calcula la distancia más corta entre el nodo actual y cada uno de los nodos no visitados, utilizando una función de peso asignada a cada arco. El nodo no visitado con la distancia más corta se convierte en el nuevo nodo actual, y el proceso se repite. Se actualizan las distancias de los nodos no visitados adyacentes al nuevo nodo actual, tomando en cuenta la distancia mínima encontrada hasta el momento. Esta exploración continua hasta que se alcanza el nodo final (punto de destino).

En este trabajo se desea abordar temas aunque sea de manera superficial que estan estrechamente relacionados con la paralelización de algorítmos de búsqueda de caminos óptimos, si bien cada algoritmo abarca tema suficiente para una tesís, en este trabajo se plantea iniciar el camino aunque sea de manera teorica de otros algoritmos además del Dijkstra para una futura implementación paralela, se mencionarán Floyd, Dantzig y A* (estrella), entre otros. Además de considerarse las distintas herramientas para paralelizar como podría ser el concepto de procesamiento por lotes, paralelizar el propio algoritmo, entre otros.

1.4. Objetivo

- Evaluar la eficiencia de la paralelización de los algoritmos de búsqueda de caminos más cortos entre los nodos de un grafo, comparando su versión secuencial con su versión paralela.

- Considerar teóricamente las herramientas capaces de paralelizar un algoritmo de búsqueda de caminos más cortos, buscando el mejor método en cuanto a facilidad de implementación, reutilización y resultados.

- Comparar tiempos de ejecución y exactitud de los resultados para el algoritmo de Dijkstra en su implementación secuencial y paralela.

1.5. marco teórico

La concurrencia, en el ámbito de la computación, se asemeja a una coreografía precisa donde múltiples procesos comparten el escenario del procesador, ejecutando sus pasos de manera simultánea. A diferencia de la ejecución secuencial, donde cada proceso espera su turno pacientemente, la concurrencia permite que los procesos avancen en paralelo, aprovechando al máximo los recursos disponibles. Como resume Blancarte (2017) cada núcleo representa un espacio independiente donde un proceso puede desplegar su coreografía. De esta manera, un procesador con cuatro núcleos puede albergar hasta cuatro procesos ejecutando sus instrucciones de forma simultánea, como si se tratara de cuatro bailarines independientes en un mismo escenario.

No obstante, la concurrencia no implica que todos los procesos reciban la misma atención del procesador de manera uniforme. La planificación del sistema operativo determina el orden en que los procesos acceden a los recursos compartidos, como la memoria y la unidad central de procesamiento. Es decir, los procesos comparten el escenario, pero no necesariamente al mismo tiempo.

Como sigue explicando Blancarte (2017) el paralelismo, en el ámbito de la computación, adopta la filosofía de "divide y vencerás"para abordar problemas complejos de manera más eficiente. En lugar de tratar un problema como una entidad única, el paralelismo lo descompone en subproblemas más pequeños y manejables, asignando cada uno a un procesador o unidad de procesamiento independiente.

Podrías extrapolarse en un ejemplo de un rompecabezas gigante. En un enfoque secuencial, se tendría que armar cada pieza una a una, lo que podría llevar mucho tiempo. Sin embargo, el paralelismo nos permite dividir el rompecabezas en secciones más pequeñas, enfoca elementos de poco nivel y esfuerzo en unir una porción de las piezas. Una vez que cada elemento ha completado su sección, al unir todas las soluciones coincidirá con la solución final de manera mucho más rápida.

La clave del paralelismo reside en la descomposición efectiva del problema en subproblemas independientes. Si la descomposición no es adecuada, la comunicación entre las unidades de procesamiento puede convertirse en un cuello de botella, ralentizando el proceso general.

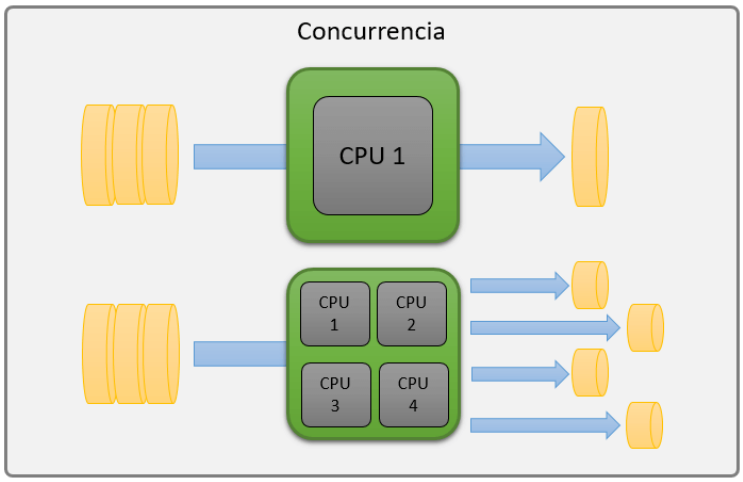

Figura 1.1: Secuencial contra Paralelo by Oscar Blancarte

La posibilidad de dividir un problema en tareas más pequeñas que se ejecutan simultáneamente, conocida como paralelismo, abre la puerta a un significativo aumento en el rendimiento computacional. Esto resulta especialmente ventajoso cuando se trabaja con equipos que cuentan con múltiples procesadores Stallings (2009)

Una vez conocido el concepto de paralelismo, el siguiente tópico que se debe resaltar son los grafos, es el problema principal que se plantea en este capitulo. En el libro Ïntroduction to Graph Theory"Wilson (1996) se introduce al concepto de grafos como una poderosa herramienta para modelar y analizar sistemas complejos. Estos entes matemáticos, compuestos por nodos (entidades del sistema) y arcos (relaciones entre entidades), nos permiten representar una amplia gama de escenarios, desde redes de transporte y comunicación hasta estructuras moleculares y relaciones sociales.

Según Wilson (1996), la versatilidad de los grafos radica en su capacidad para capturar la esencia de un sistema, abstraer sus elementos fundamentales y revelar las interacciones que los definen. Un mismo grafo puede representar diversas realidades: una red eléctrica (Figura 2.2), un mapa de caminos con nodos como ciudades y aristas como caminos (Figura 2.3), entre muchas otras, estos escenarios pueden ser representados en un grafo aún más simple en el que trabajar comodamente(Figura 2.4).

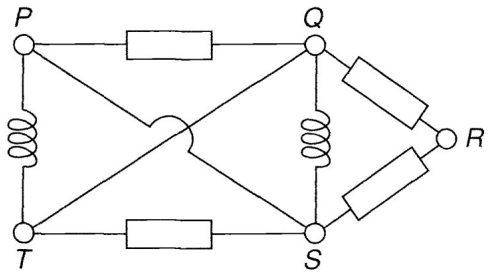

Figura 1.2: Secuencial contra Paralelo by Robin J. Wilson

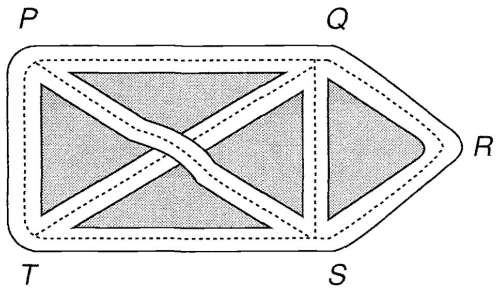

Figura 1.3: Mapa de caminos by Robin J. Wilson

En el vasto universo de la teoría de grafos, los algoritmos de búsqueda de caminos más cortos se erigen como brújulas infalibles para navegar por el laberinto de nodos y arcos. Estos algoritmos, como su nombre lo indica, tienen como objetivo encontrar la secuencia de arcos que conecta dos

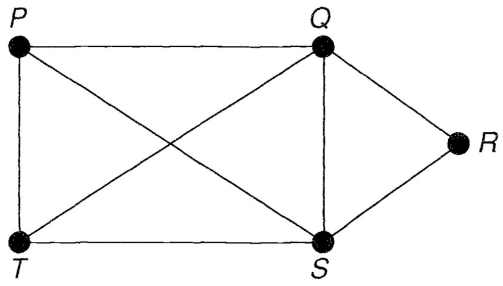

Figura 1.4: grafo simplificicado by Robin J. Wilson

nodos en un grafo, minimizando el costo total del recorrido.

En palabras de Deo (1974), la búsqueda de caminos más cortos trasciende su propósito original y se convierte en el equivalente matemático de una amplia gama de problemas de optimización. La naturaleza del grafo que se modela otorga a estas rutas óptimas un significado específico, haciéndolas útiles en diversos contextos.

La versatilidad de estos algoritmos radica en su capacidad para adaptarse a diversos escenarios y problemáticas. Desde redes de transporte y comunicación hasta sistemas de distribución y logística, la búsqueda de caminos más cortos se convierte en una herramienta indispensable para la toma de decisiones óptimas.

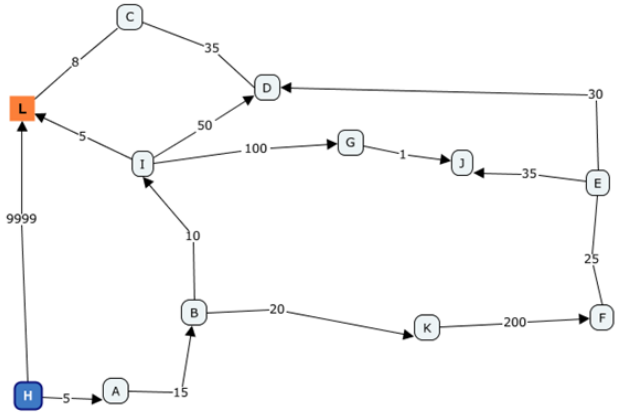

Figura 1.5: Ejemplo de un grafo ponderado no dirigido by Diego De Pablo (Elaboración propia)

El algoritmo de Dijkstra se utiliza para encontrar el camino más corto entre un nodo inicial A a un nodo final B. Para ello evalúa las distancias conocidas desde el nodo s a los otros nodos, empezando por aquellos con la menor distancia y actualizando la información cada vez que se evalúa un nuevo nodo. De esta forma se recorre el grafo hasta hallar la menor distancia al nodo t. A continuación se detalla el funcionamiento del algoritmo, conforme a la explicación de Minieka (1978). A continuación se ejemplifica su ejecución, en la figura 2.5 se observa un grafo ponderado cuyo nodo inicial es H y nodo final es L:

Paso 1: Inicialización

Se marcan todos los nodos como no visitados y se asigna una distancia inicial de $d(s) = 0$ al nodo origen s, y $d(x) = \infty$ para todos los demás nodos $x \neq s$. El nodo s se marca como visitado y se almacena en la variable y.

Paso 2: Actualización de distancias

Para cada nodo no visitado x, se actualiza $d(x)$ como el mínimo entre su distancia actual y la suma de $d(y)$ más el peso del arco (y, x), si existe. Si no hay arco, la distancia permanece infinita.

Paso 3: Condición de parada

Si $d(x) = \infty$ para todos los nodos no visitados, el algoritmo termina, indicando que no hay más caminos accesibles desde el nodo origen.

Paso 4: Iteración

Se selecciona el nodo no visitado x con la menor distancia $d(x)$, se marca como visitado, y se actualiza y con el nodo x para la próxima iteración.

Paso 5: Comprobación del nodo destino

Si el nodo destino t ha sido visitado, el algoritmo finaliza, habiendo encontrado el camino más corto de s a t.

Paso 6: Repetición

Si el nodo destino t no ha sido visitado, se vuelve al Paso 2 hasta que se cumpla la condición de parada o se encuentre el camino al nodo destino.

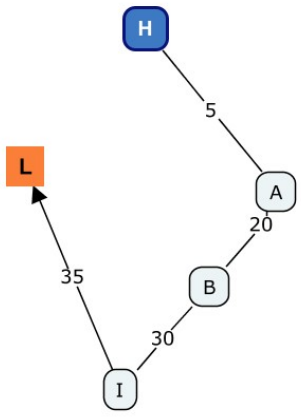

Figura 1.6: Ejemplo de camino óptimo by Diego De Pablo (Elaboración propia)

Deo (1974) propone una interpretación del algoritmo de Dijkstra en la que el grafo se representa como una matriz D = [dij], donde dij representa la distancia entre el nodo i y el nodo j. Esta matriz simplifica la programación del algoritmo, pero los pasos y el funcionamiento subyacente siguen siendo los mismos.

Minieka (1978) señala que el algoritmo de Dijkstra en su forma original no funciona correctamente si el grafo contiene arcos con pesos negativos. Esto se debe a que al marcar un nodo, este nunca se vuelve a evaluar, lo que puede llevar a resultados erróneos. Existen modificaciones al algoritmo para manejar este tipo de casos, pero no se profundizarán en este trabajo.

Un termino importante a destacar es la relajación de aristas es un concepto fundamental en el algoritmo de Dijkstra para encontrar el camino más corto entre dos nodos en un grafo. Cormen et al. (2009) Explica como esta técnica se basa en la actualización iterativa de las distancias estimadas a cada nodo, tomando en cuenta los pesos de las aristas y las distancias ya calculadas.

Formalmente para , la relajación de una arista (u, v) implica comparar la distancia actual estimada a v (denotada como d_v) con la suma de la distancia estimada actual a u (denotada como d_u) y el peso de la arista (u, v), representado por w_{uv}. Si esta suma es menor que d_v, se actualiza el valor de d_v a la nueva distancia estimada.

En otras palabras, la relajación de una arista verifica si existe un camino más corto a un nodo a través de la arista en cuestión. Si se encuentra un camino más corto, se actualiza la distancia estimada al nodo para reflejar esta nueva información.

La relajación de aristas se repite iterativamente hasta que se hayan considerado todas las aristas del grafo. Una vez finalizado el proceso, las distancias estimadas almacenadas en cada nodo representan las distancias más cortas reales entre el nodo origen y todos los demás nodos del grafo.

Esta técnica es crucial para el funcionamiento eficiente del algoritmo de Dijkstra, ya que permite identificar y actualizar de manera incremental los caminos más cortos a medida que se exploran las aristas del grafo.

Deo (1974) analiza la complejidad temporal del algoritmo de Dijkstra, concluyendo que el tiempo de ejecución es independiente de la cantidad de arcos en el grafo. Esto se debe a que, implícitamente, se asume que los arcos inexistentes sí existen, pero con pesos muy altos. El mismo autor demuestra que el tiempo de ejecución es proporcional a n^2, donde n es el número de nodos en el grafo.

A continuación una listas de algoritmos similares que podremos paralelizar de manera muy similar al algoritmo de dijkstra.

- El algoritmo de Floyd tiene como objetivo encontrar los caminos más cortos entre todos los pares de nodos en un grafo. Es considerado uno de los algoritmos más eficientes para esta tarea, junto con el algoritmo de Dantzig, según Dreyfus (1969). Su tiempo de ejecución es proporcional a n^3, donde n es el número de nodos en el grafo, como indica Deo (1974).

- El algoritmo de Dantzig, al igual que el algoritmo de Floyd, busca encontrar los caminos más cortos entre todos los pares de nodos en un grafo. Conforme Minieka (1978), ambos algoritmos realizan los mismos cálculos, pero en un orden diferente. Deo (1974) indica que su tiempo de ejecución es proporcional a n^3 y su eficiencia es comparable al algoritmo de Floyd.

- El algoritmo A* (estrella) muy parecido al algoritmo de Dijkstra con la diferencia de aplicar una función heurística, Dreyfus (1969) lo define como un algoritmo de búsqueda heurística que busca el camino más corto entre un nodo origen y un nodo destino en un grafo. A diferencia de otros algoritmos de búsqueda como la búsqueda en anchura o en profundidad, A* utiliza una función heurística para estimar la distancia restante al nodo destino, lo que le permite enfocarse en los caminos más prometedores y mejorar su eficiencia.

Ya conociendose los conceptos de paralelismo, grafos y sus algoritmos, es importante destacar que los algoritmos tradicionales suelen estar diseñados para una ejecución secuencial, donde las operaciones se realizan una tras otra. Como dicta Stallings (2009) para aprovechar las ventajas del paralelismo, es necesario rediseñarlos y adaptarlos, dividiéndolos en fracciones de trabajo independientes que puedan ejecutarse en paralelo sin generar conflictos.

A mayor paralelismo, es decir, a mayor cantidad de tareas que se ejecutan simultáneamente, mayor será la reducción en el tiempo de ejecución total. Pese a todo, es importante considerar que la ejecución paralela requiere de hardware compatible.

Independientemente de la cantidad de subtareas en que se divida un programa, solo podrán ejecutarse en paralelo la cantidad que permita el procesador o los procesadores del equipo. Por lo tanto, el costo del hardware también debe ser un factor a considerar en el diseño de soluciones.

Las Unidades de Procesamiento Gráfico (GPU), originalmente diseñadas para la generación rápida de imágenes para su visualización en pantalla, han demostrado ser altamente paralelas y útiles para la decodificación de vídeo y la creación de gráficos 3D Randima (2005).

De todas formas, su potencial no se limita a estas áreas, ya que también pueden ser empleadas para resolver problemas en otros ámbitos, dando lugar a lo que se conoce como Programación de Propósito General en Unidades de Procesamiento Gráfico (GPGPU).

Como establece Cormen et al. (2009) el algoritmo de Dijkstra, por su naturaleza secuencial, presenta un desafío a la hora de paralelizarlo. Pero no es imposible, existen diversas estrategias que pueden ser empleadas para mejorar su rendimiento en arquitecturas con múltiples procesadores, entre ella destaca la partición de grafo y paralelización del cálculo dentro de un subgrafo.

En cuanto a la partición de grafo, dividir el grafo original en subgrafos más pequeños y manejables.Asignar el cálculo de las rutas más cortas dentro de cada subgrafo a un procesador diferente. Combinar los resultados parciales obtenidos de cada subgrafo para determinar las rutas más cortas entre todos los pares de nodos del grafo original.

Por parte de la paralelización del cálculo dentro de un subgrafo, en un subgrafo específico, paralelizar el cálculo de las distancias mínimas desde un nodo origen a todos los demás nodos del subgrafo. Esto puede implicar la exploración simultánea de múltiples rutas potenciales desde el nodo origen. Se pueden emplear algoritmos paralelos como BFS (Búsqueda en anchura) o DFS (Búsqueda en profundidad) modificados para aprovechar el paralelismo.

Esta última fue evaluada por Luo and Dong (2013) Los autores proponen un algoritmo de Dijkstra paralelo que aprovecha la arquitectura altamente paralela de las GPUs. El algoritmo divide el cálculo de las rutas más cortas en tareas independientes que pueden ejecutarse simultáneamente en los múltiples núcleos de la GPU. Esto permite un procesamiento más rápido en comparación con el algoritmo de Dijkstra secuencial.

El artículo describe la implementación del algoritmo en CUDA, un lenguaje de programación paralelo para GPUs de NVIDIA. Además, los autores evalúan el rendimiento del algoritmo propuesto en diferentes escenarios y comparan su desempeño con el algoritmo de Dijkstra secuencial. Los resultados demuestran que el algoritmo paralelo de Dijkstra basado en GPU puede lograr mejoras significativas en la velocidad de cálculo, especialmente para grafos de gran tamaño.

Otra investigación interesante nace del grupo A. et al. (1998) El artículo titulado en español "Paralelización del algoritmo de camino más corto de Dijkstra"se centra en la paralelización del algoritmo de Dijkstra. El algoritmo original de Dijkstra es secuencial, Como ya se mencionó esto puede ser ineficiente en arquitecturas con múltiples procesadores.

Los autores del artículo proponen un algoritmo PRAM (Parallel Random Access Machine) para paralelizar el algoritmo de Dijkstra. El algoritmo PRAM divide el trabajo en fases que pueden

ejecutarse en paralelo. Los autores analizan el rendimiento del algoritmo PRAM y muestran que puede mejorar significativamente el rendimiento del algoritmo de Dijkstra original en arquitecturas con múltiples procesadores.

El algoritmo PRAM (Parallel Random Access Machine) es un modelo de computación teórico que describe una máquina paralela con acceso aleatorio a una memoria compartida. Se utiliza para analizar la complejidad paralela de algoritmos y evaluar su eficiencia en arquitecturas con múltiples procesadores.

El artículo también presenta extensiones del algoritmo PRAM para computación en memoria externa. Esto es útil para grafos grandes que no caben en la memoria principal. Las simulaciones muestran que el enfoque propuesto es aplicable incluso en grafos no aleatorios.

1.6. Resultados y análisis de código

Tras revisar las herramientas teóricas para paralelizar el algoritmo de búsqueda de caminos más cortos, se ha observado que el algoritmo de Dijkstra, diseñado originalmente para un procesamiento secuencial, ha sido objeto de diversas modificaciones para su paralelización. Entre estas soluciones se encuentran el uso de múltiples hebras para aumentar la velocidad de cálculo y la división del grafo en subgrafos.

Sin embargo, en este trabajo se propone un enfoque alternativo que centra el paralelismo en la creación de tareas concurrentes para relajar las aristas de los nodos vecinos. Esta estrategia permite procesar simultáneamente múltiples aristas, aprovechando los múltiples núcleos de la CPU para acelerar la ejecución.

Para demostrar la utilidad de la paralelización en este contexto, en lugar de aplicar el algoritmo una sola vez buscando un nodo origen y un nodo destino, se planteará una tarea más compleja: la búsqueda de todos los caminos óptimos que parten del nodo origen hacia todos los demás nodos del grafo. De esta manera, se pondrá a prueba la capacidad del algoritmo paralelo para manejar una tarea computacionalmente más exigente.

Antes de comparar el rendimiento de las versiones secuencial y paralela del algoritmo de Dijkstra, es crucial verificar el correcto funcionamiento de ambas implementaciones. Para ello, se desarrollarán dos versiones del algoritmo: una secuencial y otra paralela. Ambas versiones estarán diseñadas para resolver el problema de búsqueda de caminos óptimos en el grafo presentado en la Figura 2.5, tomando como nodo inicial el nodo H y considerando todos los nodos restantes como destinos.

El código implementado aparecerá detalladamente en el apartado de apéndice, pero haré un breve resumen de cada código, en el caso del código 1, es una implementación del algoritmo de Dijkstra en Java para encontrar el camino más corto entre dos nodos en un grafo ponderado. Comienza verificando la existencia de los nodos especificados en el grafo y luego inicializa las estructuras de datos necesarias para el algoritmo. Utiliza un bucle para recorrer iterativamente los nodos no visitados, actualizando las distancias mínimas a los nodos adyacentes. Una vez que se visitan todos los nodos, se construye el camino óptimo y se devuelve como una cadena de salida. Además, el código mide el tiempo de ejecución del algoritmo y lo imprime junto con el camino óptimo encontrado.

En resumen, el código proporciona una implementación secuencial eficiente del algoritmo de Dijkstra para encontrar el camino más corto en un grafo. Utiliza estructuras de datos como mapas y conjuntos para realizar los cálculos y devuelve el camino óptimo junto con el tiempo de ejecución. A continuación una imagen de la salida resultante para el grafo de la figura 2.5 tomando como nodo

de origen el nodo H:

```
Nodo H: 0 (H(0))
Nodo I: 30 (H(0) -> A(5) -> B(20) -> I(30))
Nodo K: 40 (H(0) -> A(5) -> B(20) -> K(40))
Nodo C: 43 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43))
Nodo D: 78 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43) -> D(78))
Nodo L: 35 (H(0) -> A(5) -> B(20) -> I(30) -> L(35))
Nodo E: 108 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43) -> D(78) -> E(108))
Nodo F: 133 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43) -> D(78) -> E(108) -> F(133))
Nodo J: 131 (H(0) -> A(5) -> B(20) -> I(30) -> G(130) -> J(131))
Nodo G: 130 (H(0) -> A(5) -> B(20) -> I(30) -> G(130))
Tiempo de ejecución: 18 milisegundos
```

Figura 1.7: Salida por pantalla del código 1 (implementación secuencial de dijkstra)

La figura muestra la salida del algoritmo, donde se presentan los pesos de los caminos óptimos desde el nodo H hasta todos los demás nodos del grafo. Para este grafo relativamente pequeño, la implementación secuencial del algoritmo de Dijkstra tarda 18 milisegundos en ejecutarse. Un análisis detallado del grafo confirma que los caminos mostrados son efectivamente los caminos óptimos.

El código 2 implementa el algoritmo de Dijkstra para encontrar el camino más corto Utilizando paralelismo para acelerar el proceso de relajación de aristas, aprovecha un ForkJoinPool y la creación de tareas paralelas como componentes clave que permite al algoritmo paralelo procesar múltiples aristas simultáneamente, potencialmente mejorando el rendimiento en grafos grandes y complejos.

La lógica detrás de este enfoque paralelo es que la relajación de las aristas del grafo es una operación que se puede realizar de manera independiente para diferentes aristas. Por lo tanto, al ejecutar estas operaciones en paralelo, se pueden aprovechar múltiples núcleos de CPU para acelerar el proceso.

Este enfoque paralelo puede predecir resultados peores en grafos de pequeño tamaño debido al costo adicional asociado con la creación y gestión de múltiples hilos, que puede superar el beneficio del paralelismo en grafos pequeños. Sin embargo, en grafos de gran tamaño con muchas aristas, el paralelismo puede conducir a mejoras significativas en el tiempo de ejecución, ya que permite aprovechar la capacidad de procesamiento de múltiples núcleos para procesar aristas simultáneamente, lo que puede compensar el costo adicional de la paralelización.

```
Nodo H: 0 (H(0))
Nodo I: 30 (H(0) -> A(5) -> B(20) -> I(30))
Nodo K: 40 (H(0) -> A(5) -> B(20) -> K(40))
Nodo C: 43 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43))
Nodo D: 78 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43) -> D(78))
Nodo L: 35 (H(0) -> A(5) -> B(20) -> I(30) -> L(35))
Nodo E: 108 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43) -> D(78) -> E(108))
Nodo F: 133 (H(0) -> A(5) -> B(20) -> I(30) -> L(35) -> C(43) -> D(78) -> E(108) -> F(133))
Nodo J: 131 (H(0) -> A(5) -> B(20) -> I(30) -> G(130) -> J(131))
Nodo G: 130 (H(0) -> A(5) -> B(20) -> I(30) -> G(130))
Tiempo de ejecución: 24 milisegundos
```

Figura 1.8: Salida por pantalla del código 2 (implementación paralela de dijkstra)

En esta ocasión la salida más común es que tarde 24 milisegundos, un poco más que nuestra implementación secuencial, y aunque en ocasiones alguna compilación puede bajar hasta los 19 segundos en la mayoría de ocasiones para un grafo tan pequeño dará peores resultados que la versión secuencial.

Aspecto	**SequentialDijkstra**	**ParallelDijkstra**
Ejecución	Secuencial, un solo hilo	Paralelo, múltiples hilos
Recorrido	Secuencialmente, una arista a la vez	Paralelamente, múltiples aristas a la vez
Eficiencia	Depende del número de nodos y aristas	Puede ser más eficiente en grafos grandes con muchas aristas
Uso de recursos	Menos uso de recursos, menor complejidad	Mayor uso de recursos debido a la creación y gestión de múltiples tareas
Complejidad de implementación	Relativamente simple	Más compleja debido a la necesidad de sincronización y gestión de tareas paralelas

Tabla 1.1: Comparación entre SequentialDijkstra y ParallelDijkstra

Los códigos 3 y 4 se han desarrollado para evaluar la escalabilidad del rendimiento de las implementaciones secuencial y paralela del algoritmo de Dijkstra. Estos códigos modifican el método main para generar nodos de forma aleatoria, incluyendo tanto letras como números. Esto permite crear grafos de diferentes tamaños, desde 24 nodos hasta un número mucho mayor, con sus correspondientes aristas.

El objetivo de esta estrategia es observar cómo la diferencia de velocidad entre las implementaciones secuencial y paralela se comporta al aumentar el tamaño del grafo. Se pretende determinar el punto a partir del cual la implementación paralela comienza a superar significativamente a la implementación secuencial en términos de rendimiento.

La gráfica 2.9 muestra el tiempo de ejecución promedio, en milisegundos, del algoritmo de Dijkstra para diferentes tamaños de grafo. Cada punto de datos representa el promedio de 10 compilaciones del algoritmo para un número específico de nodos

Si bien la gráfica 2.9 que compara el algoritmo secuencial contra el paralelo a través del aumento de nodos, esta aunque sugiere una cierta superioridad del algoritmo paralelo, es importante destacar una limitación en los resultados presentados. Durante la evaluación del rendimiento en escenarios con un número elevado de nodos, se observaron errores de manejo de valores nulos en la implementación paralela. Estos errores, resultado de la complejidad de los cálculos y la necesidad de información sobre nodos intermedios, ocasionaron la interrupción del proceso y la necesidad de recompilaciones adicionales para obtener una media precisa.

Para abordar este problema, se propone la implementación de mecanismos de concurrencia, como semáforos, que permitan gestionar el acceso a los recursos y evitar la aparición de errores. Si bien en este caso la implementación de semáforos no arrojó resultados positivos, debido a un "speedup"negativo para configuraciones con más de 20.000 nodos (posiblemente relacionado con limitaciones de memoria en Java ya que cada nodo tiene de 1 a 4 aristas), la exploración de otras técnicas de concurrencia sigue siendo una línea de investigación prometedora para optimizar el rendimiento de la implementación paralela.

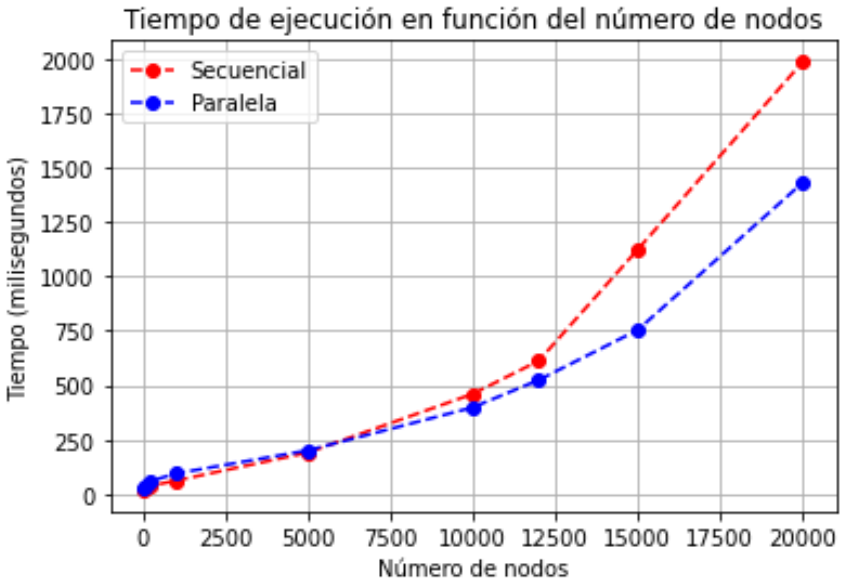

Figura 1.9: Comparación a través del número de nodos, secuencial vs paralelo

1.7. Conclusiones y recomendaciones

En este trabajo se ha evaluado la eficacia de los algoritmos de Dijkstra para la búsqueda de caminos más cortos entre todos los pares de nodos de un grafo, tanto en sus implementaciones secuencial como paralela. Los resultados obtenidos revelan un panorama alentador con algunas consideraciones importantes a tomar en cuenta que no dejan tan bien el paralelismo.

Teóricamente, se ha resaltado la naturaleza secuencial del algoritmo de Dijkstra, lo que se ha corroborado durante la implementación de su versión paralela. En este sentido, sería de gran interés extender el trabajo comparando la versión secuencial y paralela con otros algoritmos de búsqueda de caminos óptimos, como A* (Estrellas), Floyd y otros mencionados en el marco teórico.

La implementación desarrollada ha eliminado la posibilidad de errores asociados a los solapes de escrituras y lecturas que presentan muchas versiones paralelas, gracias a la concurrencia en el cálculo de los vecinos de un nodo sin tener alteraciones en una misma cuenta.

No obstante, los resultados sugieren que el algoritmo de Dijkstra no es el método más eficiente para la búsqueda de todos los caminos más cortos en un grafo en lo que paralelismo respecta. Además, su eficiencia presenta una gran variabilidad en función de las propiedades del grafo evaluado, dependiendo en gran medida del punto de inicio y la cantidad de aristas. En este sentido, se considera que un enfoque más preciso que fije un número exacto de aristas por nodo podría generar resultados más robustos y menos variables por iteración.

El capítulo concluye con un mensaje crucial: el paralelismo no es una mejora superior, sino una herramienta que ofrece una perspectiva diferente y permite alcanzar soluciones con resultados ligeramente distintos. Si bien se trata de una herramienta poderosa y valiosa, no es una solución mágica que garantice un mejor rendimiento. Este trabajo ha demostrado que su implementación

puede ser compleja, especialmente en el contexto de grafos. Además, para problemas sencillos, implementar paralelismo, a pesar del esfuerzo adicional, puede generar resultados adversos. En ocasiones, dividir una tarea simple en múltiples tareas simples no se alinea con los objetivos del paralelismo. Se busca eliminar la dificultad de las tareas, no aumentarlas.

Bibliografía

A., C., K., M., U., M. and P, S. (1998). A parallelization of dijkstra's shortest path algorithm., *Lecture notes in computer science* **1**(10): 722–731.

Blancarte, O. (2017). Concurrencia vs. paralelismo, *Oscar Blancarte Blog* .
URL: *https://www.oscarblancarteblog.com/2017/03/29/concurrencia-vs-paralelismo/*

Cassingena, E. (2022). Algoritmo de la ruta más corta de dijkstra: Introducción gráfica.
URL: *https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/*

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W. and Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using cuda., *Journal of parallel and distributed computing,* **68**(10): 1370–1380.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009). *Introduction to Algorithms (3rd ed.)*, MIT Press, Cambridge, Massachusetts, Estados Unidos.

Deo, N. (1974). *Graph theory with applications to engineering and computer science.*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Dreyfus, S. E. (1969). An appraisal of some shortest-path algorithms., *Operations research* **17**(3): 372–426.

Harish, P. and Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda. en international conference on highperformance computing, *Springer* **1**(1): 197–208.

Luo, Q. and Dong, Y. (2013). A parallel dijkstra's algorithm for gpu-based computing., *Journal of Parallel and Distributed Computing* **73**(11): 1716–1724.

Minieka, E. (1978). *Optimization Algorithms for Networks and Graphs.*, Marcel Dekker, New York, NY.

Randima, F. (2005). *GPU Gems 2, Programming techniques for HighPerformance Graphics and General-Purpose Computation*, NVIDIA, AddisonWesley, Massachusetts.

Stallings, W. (2009). *Operating systems: internals and design principles (6ta edición).*, Pearson Education, Inc., Upper Saddle River, NJ.

Wilson, R. J. (1996). *Introduction to graph theory.*, Longman Group, Harlow.

1.8. Apendice (código)

Código 1 para la construcción del grafo y aplicación de dijkstra secuencial:

```
1  package dijkstra_paralelo;
2
3  import java.util.*;
4
5  public class SequentialDijkstra {
6
7      static class Graph {
8          private final Map<String, Integer> nodeIndexMap;
9          private final List<String> nodes;
10         private final int[][] adjMatrix;
11
12         public Graph(int maxNodes) {
13             nodeIndexMap = new HashMap<>();
14             nodes = new ArrayList<>();
15             adjMatrix = new int[maxNodes][maxNodes];
16         }
17
18         public void addNode(String node) {
19             if (!nodeIndexMap.containsKey(node)) {
20                 int index = nodes.size();
21                 nodes.add(node);
22                 nodeIndexMap.put(node, index);
23             }
24         }
25
26         public void addEdge(String src, String dest, int weight) {
27             addNode(src);
28             addNode(dest);
29             int srcIndex = nodeIndexMap.get(src);
30             int destIndex = nodeIndexMap.get(dest);
31             adjMatrix[srcIndex][destIndex] = weight;
32             adjMatrix[destIndex][srcIndex] = weight;
33             // Si el grafico es no dirigido
34         }
35
36         public Result dijkstra(String startNode) {
37             int V = nodes.size();
38             int[] dist = new int[V];
39             int[] pred = new int[V];
40             Arrays.fill(dist, Integer.MAX_VALUE);
41             Arrays.fill(pred, -1);
42             int srcIndex = nodeIndexMap.get(startNode);
```

```
43              dist[srcIndex] = 0;
44
45              boolean[] visited = new boolean[V];
46              PriorityQueue<Integer> pq = new PriorityQueue<>
47              (Comparator.comparingInt(node -> dist[node]));
48              pq.add(srcIndex);
49
50              while (!pq.isEmpty()) {
51                  int u = pq.poll();
52                  if (visited[u]) continue;
53                  visited[u] = true;
54
55                  for (int v = 0; v < V; v++) {
56                      if (!visited[v] && adjMatrix[u][v] != 0) {
57                          int weight = adjMatrix[u][v];
58                          if (dist[u] + weight < dist[v]) {
59                              dist[v] = dist[u] + weight;
60                              pred[v] = u;
61                              pq.add(v);
62                          }
63                      }
64                  }
65              }
66              return new Result(dist, pred);
67          }
68
69          public void printResult(Result result, String startNode) {
70              for (int i = 0; i < result.dist.length; i++) {
71                  if (result.dist[i] == Integer.MAX_VALUE) {
72                      System.out.println
73                      ("Nodo␣" + nodes.get(i) + ":␣Inalcanzable");
74                      continue;
75                  }
76                  StringBuilder path = new StringBuilder();
77                  printPath(result.pred, i, path, result.dist);
78                  System.out.println
79                  ("Nodo␣" + nodes.get(i) + ":␣" +
80                  result.dist[i] + "␣(" + path + ")");
81              }
82          }
83
84          private void printPath(int[] pred, int i,
85          StringBuilder path, int[] dist) {
86              if (i == -1) return;
87              printPath(pred, pred[i], path, dist);
88              if (path.length() > 0) {
```

```
89                  path.append("␣->␣");
90              }
91              path.append(nodes.get(i)).append
92              ("(").append(dist[i]).append(")");
93          }
94
95          static class Result {
96              int[] dist;
97              int[] pred;
98
99              Result(int[] dist, int[] pred) {
100                 this.dist = dist;
101                 this.pred = pred;
102             }
103         }
104     }
105
106     public static void main(String[] args) {
107         // Creacion del grafo
108         Graph graph = new Graph(12);
109         // Numero maximo de nodos en el grafo
110
111         graph.addEdge("A", "B", 15);
112         graph.addEdge("A", "H", 5);
113         graph.addEdge("B", "I", 10);
114         graph.addEdge("B", "K", 20);
115         graph.addEdge("C", "D", 35);
116         graph.addEdge("C", "L", 8);
117         graph.addEdge("D", "E", 30);
118         graph.addEdge("D", "I", 50);
119         graph.addEdge("E", "F", 25);
120         graph.addEdge("E", "J", 35);
121         graph.addEdge("F", "K", 200);
122         graph.addEdge("G", "I", 100);
123         graph.addEdge("G", "J", 1);
124         graph.addEdge("H", "L", 9999);
125         graph.addEdge("I", "L", 5);
126
127         // Obtener el tiempo de inicio
128             long inicio = System.nanoTime();
129         Graph.Result result = graph.dijkstra("H");
130         System.out.println("Distancias␣desde␣el␣nodo␣H:");
131         graph.printResult(result, "H");
132
133      // Obtener el tiempo de finalizacion
134             long fin = System.nanoTime();
```

```
135
136             // Calcular la duracion de la ejecucion
137             long t = (fin - inicio) / 1_000_000;
138             // Convertir de nanosegundos a milisegundos
139
140             System.out.println
141      ("Tiempo de ejecucion: " + t + " milisegundos");
142     }
143 }
```

Código 2 para la creación de un grafo y llamada del algoritmo de dijkstra versión con concurrencia:

```
1  package dijkstra_paralelo;
2
3  import java.util.concurrent.*;
4  import java.util.*;
5
6  public class ParallelDijkstra {
7
8      static class Graph {
9          private final Map<String, Integer> nodeIndexMap;
10         private final List<String> nodes;
11         private final int[][] adjMatrix;
12
13         public Graph(int maxNodes) {
14             nodeIndexMap = new HashMap<>();
15             nodes = new ArrayList<>();
16             adjMatrix = new int[maxNodes][maxNodes];
17         }
18
19         public void addNode(String node) {
20             if (!nodeIndexMap.containsKey(node)) {
21                 int index = nodes.size();
22                 nodes.add(node);
23                 nodeIndexMap.put(node, index);
24             }
25         }
26
27         public void addEdge(String src, String dest, int weight) {
28             addNode(src);
29             addNode(dest);
30             int srcIndex = nodeIndexMap.get(src);
31             int destIndex = nodeIndexMap.get(dest);
32             adjMatrix[srcIndex][destIndex] = weight;
33             adjMatrix[destIndex][srcIndex] = weight;
34             // Si el grafico es no dirigido
```

```
35          }
36
37          public Result parallelDijkstra(String startNode) throws
38          InterruptedException, ExecutionException {
39              int V = nodes.size();
40              int[] dist = new int[V];
41              int[] pred = new int[V];
42              Arrays.fill(dist, Integer.MAX_VALUE);
43              Arrays.fill(pred, -1);
44              int srcIndex = nodeIndexMap.get(startNode);
45              dist[srcIndex] = 0;
46
47              boolean[] visited = new boolean[V];
48              PriorityQueue<Integer> pq = new
49              PriorityQueue<>
50              (Comparator.comparingInt(node -> dist[node]));
51              pq.add(srcIndex);
52
53              ForkJoinPool forkJoinPool = new ForkJoinPool();
54              while (!pq.isEmpty()) {
55                  int u = pq.poll();
56                  if (visited[u]) continue;
57                  visited[u] = true;
58
59                  List<Callable<Void>> tasks = new ArrayList<>();
60                  for (int v = 0; v < V; v++) {
61                      if (!visited[v] && adjMatrix[u][v] != 0) {
62                          final int node =
63                          v;
64                          //Definir node como final
65                          int weight = adjMatrix[u][node];
66                          tasks.add(() -> {
67                              if (dist[u] + weight < dist[node]) {
68                                  dist[node] = dist[u] + weight;
69                                  pred[node] = u;
70                                  pq.add(node);
71                              }
72                              return null;
73                          });
74                      }
75                  }
76                  forkJoinPool.invokeAll(tasks);
77              }
78              forkJoinPool.shutdown();
79              return new Result(dist, pred);
80          }
```

```
81
82          public void printResult(Result result, String startNode) {
83              for (int i = 0; i < result.dist.length; i++) {
84                  if (result.dist[i] == Integer.MAX_VALUE) {
85                      System.out.println
86                      ("Nodo " + nodes.get(i) + ": Inalcanzable");
87                      continue;
88                  }
89                  StringBuilder path = new StringBuilder();
90                  printPath(result.pred, i, path, result.dist);
91                  System.out.println
92                  ("Nodo " + nodes.get(i) + ": " +
93                  result.dist[i] + " (" + path + ")");
94              }
95          }
96
97          private void printPath
98          (int[] pred, int i, StringBuilder path, int[] dist) {
99              if (i == -1) return;
100             printPath(pred, pred[i], path, dist);
101             if (path.length() > 0) {
102                 path.append(" -> ");
103             }
104             path.append(nodes.get(i)).append
105             ("(").append(dist[i]).append(")");
106         }
107
108         static class Result {
109             int[] dist;
110             int[] pred;
111
112             Result(int[] dist, int[] pred) {
113                 this.dist = dist;
114                 this.pred = pred;
115             }
116         }
117     }
118
119     public static void main(String[] args)
120     throws InterruptedException, ExecutionException {
121         // Creacion del grafo
122         Graph graph = new Graph(12);
123         // Numero maximo de nodos en el grafo
124
125         graph.addEdge("A", "B", 15);
126         graph.addEdge("A", "H", 5);
```

```
127         graph.addEdge("B", "I", 10);
128         graph.addEdge("B", "K", 20);
129         graph.addEdge("C", "D", 35);
130         graph.addEdge("C", "L", 8);
131         graph.addEdge("D", "E", 30);
132         graph.addEdge("D", "I", 50);
133         graph.addEdge("E", "F", 25);
134         graph.addEdge("E", "J", 35);
135         graph.addEdge("F", "K", 200);
136         graph.addEdge("G", "I", 100);
137         graph.addEdge("G", "J", 1);
138         graph.addEdge("H", "L", 9999);
139         graph.addEdge("I", "L", 5);
140 
141         // Obtener el tiempo de inicio
142             long inicio = System.nanoTime();
143 
144         Graph.Result result = graph.parallelDijkstra("H");
145         System.out.println("Distancias desde el nodo H:");
146         graph.printResult(result, "H");
147 
148         // Obtener el tiempo de finalizacion
149             long fin = System.nanoTime();
150 
151             // Calcular la duracion de la ejecucion
152             long t = (fin - inicio) / 1_000_000;
153             // Convertir de nanosegundos a milisegundos
154 
155             System.out.println
156      ("Tiempo de ejecucion: " + t + " milisegundos");
157     }
158 }
```

Código 3 método que crea grafos con N + 64 nodos y se aplica el dijkstra secuencial

```
1  package dijkstra_paralelo;
2 
3  import java.util.*;
4  import java.util.concurrent.ExecutionException;
5 
6  public class SequentialDijkstraWithLargeGraph {
7 
8      public static void main(String[] args) {
9          Scanner scanner = new Scanner(System.in);
10 
11         System.out.print
12         ("Ingrese el numero maximo de nodos en el grafo: ");
```

```
13          int n = scanner.nextInt();
14
15          SequentialDijkstra.grafo graph = new
16          SequentialDijkstra.grafo(n + 64);
17          // Numero maximo de nodos en el grafo
18
19          // Generar aristas con valores aleatorios entre 1 y 100
20          Random random = new Random();
21          for (char c1 = 'A'; c1 <= 'Z'; c1++) {
22              for (char c2 = 'A'; c2 <= 'Z'; c2++) {
23                  if (c1 != c2) { // Evitar lazos
24                      int weight = random.nextInt(100) + 1;
25                      // Valor aleatorio entre 1 y 100
26                      graph.addEdge(String.valueOf(c1),
27                      String.valueOf(c2), weight);
28                  }
29              }
30          }
31          for (char c1 = 'a'; c1 <= 'z'; c1++) {
32              for (char c2 = 'A'; c2 <= 'Z'; c2++) {
33                  if (c1 != c2) { // Evitar lazos
34                      int weight = random.nextInt(100) + 1;
35                      // Valor aleatorio entre 1 y 100
36                      graph.addEdge(String.valueOf(c1),
37                      String.valueOf(c2), weight);
38                  }
39              }
40          }
41
42          // Agregar 120 aristas mas
43          for (int i = 0; i < 120; i++) {
44              char c1 = (char) ('A' + random.nextInt(26));
45              char c2 = (char) ('A' + random.nextInt(26));
46              int weight = random.nextInt(100) + 1;
47              // Valor aleatorio entre 1 y 100
48              graph.addEdge(String.valueOf(c1),
49              String.valueOf(c2), weight);
50          }
51
52          // Agregar nodos con nombres del 1 al n
53          for (int i = 1; i <= n; i++) {
54              graph.addEdge(String.valueOf(i),
55              String.valueOf(i), 0); // Agregar nodo con peso 0
56          }
57
58          // Agregar n aristas adicionales entre nodos
```

```
59          for (int i = 0; i < n; i++) {
60              int node1 = random.nextInt(n) + 1;
61              // Nodo aleatorio del 1 al n
62              int node2 = random.nextInt(n) + 1;
63              // Nodo aleatorio del 1 al n
64              int weight = random.nextInt(200) + 1;
65              // Valor aleatorio entre 1 y 200
66              graph.addEdge(String.valueOf(node1),
67              String.valueOf(node2), weight);
68          }
69
70          // Agregar aristas que conecten nodos
71          //numericos con nodos de letras minusculas
72          for (int i = 1; i <= n; i++) {
73              char letter = (char) ('a' + random.nextInt(26));
74              // Letra minuscula aleatoria
75              int weight = random.nextInt(200)
76              + 1; // Valor aleatorio entre 1 y 200
77              graph.addEdge(String.valueOf(i),
78              String.valueOf(letter), weight);
79          }
80
81          // Obtener el tiempo de inicio
82          long inicio = System.nanoTime();
83          SequentialDijkstra.grafo.Result
84          result = graph.dijkstra("H");
85          System.out.println("Distancias␣desde␣el␣nodo␣H:");
86          graph.printResult(result, "H");
87
88          // Obtener el tiempo de finalizacion
89          long fin = System.nanoTime();
90
91          // Calcular la duracion de la ejecucion
92          long t = (fin - inicio) / 1_000_000;
93          // Convertir de nanosegundos a milisegundos
94
95          System.out.println
96          ("Tiempo␣de␣ejecucion:␣" + t + "␣milisegundos");
97
98          scanner.close();
99      }
100 }
```

Código 4 método que crea grafos con N + 64 nodos y se aplica el dijkstra paralelo

```
1  package dijkstra_paralelo;
2
```

```
3  import java.util.*;
4  import java.util.concurrent.ExecutionException;
5
6  public class ParallelDijkstraWithLargeGraph {
7
8      public static void main(String[] args) {
9          Scanner scanner = new Scanner(System.in);
10
11         System.out.print
12         ("Ingrese el numero maximo de nodos en el grafo: ");
13         int n = scanner.nextInt();
14
15         ParallelDijkstra.Graph graph = new
16         ParallelDijkstra.Graph(n + 64);
17
18 // Grafo con letras desde A hasta Z (mayusculas y minusculas)
19 //y numeros del 1 al n
20
21         // Agregar aristas con valores aleatorios entre 1 y 100
22         Random random = new Random();
23         for (char c1 = 'A'; c1 <= 'Z'; c1++) {
24             for (char c2 = 'A'; c2 <= 'Z'; c2++) {
25                 if (c1 != c2) { // Evitar lazos
26                     int weight = random.nextInt(100) + 1;
27 // Valor aleatorio entre 1 y 100
28                     graph.addEdge(String.valueOf(c1),
29                     String.valueOf(c2), weight);
30                 }
31             }
32         }
33         for (char c1 = 'a'; c1 <= 'z'; c1++) {
34             for (char c2 = 'A'; c2 <= 'Z'; c2++) {
35                 if (c1 != c2) { // Evitar lazos
36                     int weight = random.nextInt(100) + 1;
37                     // Valor aleatorio entre 1 y 100
38                     graph.addEdge(String.valueOf(c1),
39                     String.valueOf(c2), weight);
40                 }
41             }
42         }
43
44         // Agregar 120 aristas mas
45         for (int i = 0; i < 120; i++) {
46             char c1 = (char) ('A' + random.nextInt(26));
47             char c2 = (char) ('A' + random.nextInt(26));
48             int weight = random.nextInt(100) + 1;
```

```
49              // Valor aleatorio entre 1 y 100
50              graph.addEdge(String.valueOf(c1),
51              String.valueOf(c2), weight);
52          }
53
54          // Agregar nodos con nombres del 1 al n
55          for (int i = 1; i <= n; i++) {
56              graph.addEdge(String.valueOf(i),
57              String.valueOf(i), 0);
58              // Agregar nodo con peso 0
59          }
60          // Agregar n aristas adicionales entre nodos
61          for (int i = 0; i < n; i++) {
62              int node1 = random.nextInt(n) + 1;
63              // Nodo aleatorio del 1 al n
64              int node2 = random.nextInt(n) + 1;
65              // Nodo aleatorio del 1 al n
66              int weight = random.nextInt(200) + 1;
67              // Valor aleatorio entre 1 y 200
68              graph.addEdge(String.valueOf(node1),
69              String.valueOf(node2), weight);
70          }
71
72          // Agregar aristas que conecten nodos
73          //numericos con nodos de letras minusculas
74          for (int i = 1; i <= n; i++) {
75              char letter = (char) ('a' + random.nextInt(26));
76              // Letra minuscula aleatoria
77              int weight = random.nextInt(200) + 1;
78              // Valor aleatorio entre 1 y 200
79              graph.addEdge(String.valueOf(i),
80              String.valueOf(letter), weight);
81          }
82          // Obtener el tiempo de inicio
83          long inicio = System.nanoTime();
84          ParallelDijkstra.Graph.Result result = null;
85          try {
86              result = graph.parallelDijkstra("H");
87          } catch (InterruptedException e) {
88              e.printStackTrace();
89          } catch (ExecutionException e) {
90              e.printStackTrace();
91          }
92          System.out.println("Distancias␣desde␣el␣nodo␣H:");
93          graph.printResult(result, "H");
94
```

```
95          // Obtener el tiempo de finalizacion
96          long fin = System.nanoTime();
97
98          // Calcular la duracion de la ejecucion
99          long t = (fin - inicio) / 1_000_000;
100         // Convertir de nanosegundos a milisegundos
101
102         System.out.println
103         ("Tiempo de ejecucion: " + t + " milisegundos");
104
105         scanner.close();
106     }
107 }
```