

énez19

Proyecto #2: Programación paralela

Algoritmo de Dijkstra

Estructura de datos y algoritmos II

Cabello, Sofía López, Ricardo Núñez, Diego

18 de enero de 2021

1. Introducción

El campo de la arquitectura digital de los computadores ha crecido exponencialmente en los últimos años. De manera simultánea, la investigación sobre diferentes formas de concurrencia, desde la segmentación estándar hasta paralelismo puro y la creación de estructuras que puedan soportar modelos de programación para estos sistemas han permitido que la capacidad y productividad del hardware continúe creciendo también. Los computadores modernos se vuelven cada vez más eficientes al contar con más procesadores, núcleos y memoria. Esta es una tendencia que se espera que continúe en los próximos años.

La ventaja de esto es que múltiples procesadores pueden cumplir una tarea o pueden dividirla para que más de un hilo trabaje sobre ella, cuyo resultado óptimo sería ejecutarla en menos tiempo que si sólo se hiciera de manera secuencial. Una de las formas de medir este rendimiento es a través de la complejidad que tiene, en el caso de un algoritmo paralelo se mide por el tiempo y el número de procesadores que se requieren. El nuevo reto es orientar el diseño computacional a la simplicidad y alto rendimiento, pero conservando un bajo costo y necesidad de poder, el paralelismo juega un papel importante en esto.

El problema de camino más corto es uno de los problemas fundamentales de optimización y uno de los más notorios en modelos computacionales paralelos debido a su amplio campo de aplicaciones. En este proyecto, abordaremos este problema haciendo uso del algoritmo de Dijkstra con la objetivo

de analizar primero la definición y funcionamiento del algoritmo, tomando en cuenta sus características, pseudocódigo, ventajas y analizando en particular la implementación paralela. Finalmente se hará una comparación entre las dos versiones para encontrar cuál resulta más eficiente al hacer pruebas con OpenMP en el lenguaje C.

2. Antecedentes

Aunque hay una gran cantidad de estudios teóricos reportados sobre el procesamiento de grafos como Arjomandi en 1975 [1], Bentley en 1980[2], en el pasado no muchos de ellos consideraron el algoritmo para la ruta más corta. Uno de los pocos a encontrar fue realizado en 1980 por Deo et al. [3] en el que se diseñó, codificó y se ejecutó un algoritmo paralelo en una computadora con arquitectura MIMD.

En los últimos 20 años, sin embargo, se ha explorado mucho más. Una de estas implementaciones fue realizada por Fang Zhou Lin y Nan Zhao [4] en 1999. Respecto a aplicaciones, en 2012, Songmin Jia, Xiaolin Yin, y Xiuzhi Li [5] propusieron una técnica de Localización y Construcción de Mapas Simultánea (SLAM por sus siglas en inglés *Simultaneous Localization and Map Building*) que resulta eficiente para la navegación de robots móviles interiores basada en OpenMP. El código implementado para realizar las pruebas fue tomado de Programming on Parallel Machines de Matloff [6].

3. Programando en OpenMP

OpenMP es una API (interfaz de programa de aplicación) que se utiliza para dirigir explícitamente el paralelismo de memoria compartida. OpenMP se introdujo en 1997 para estandarizar las extensiones de programación para máquinas de memoria compartida. OpenMP se ha convertido en el entorno elegido por muchos, de los practicantes de este tipo de programación. Consiste en un conjunto de directivas que se agregan al código C / C ++ / FORTRAN que manipula los subprocesos, sin que el programador tenga que lidiar con los subprocesos directamente. De esta manera obtenemos "lo mejor de ambos mundos": el verdadero paralelismo de los hilos (no preventivos) y el placer de evitar las molestias de la programación de hilos.

OpenMP proporciona una API "basada en directivas". Esto significa que hay instrucciones especiales de preprocesador conocidas como pragmas. Los

pragmas generalmente se agregan a un sistema para permitir comportamientos que no son parte de la especificación C básica. Los compiladores que no apoyan los pragmas son libres de omitirlos. Esto permite que un programa que utiliza los pragmas se ejecute en cualquier plataforma. Entonces, en principio, si tiene un programa OpenMP cuidadosamente escrito, se puede compilar y ejecutar en cualquier sistema con un compilador C, independientemente de si el compilador es compatible con OpenMP.

Esta directiva le dice al sistema de tiempo de ejecución que ejecute el siguiente bloque estructurado de codificar en paralelo. Puede bifurcar o iniciar varios subprocesos para ejecutar el bloque estructurado, que es un bloque de código con un único punto de entrada y una única salida. El número de subprocesos iniciados depende del sistema, pero la mayoría de los sistemas comienza un hilo para cada núcleo disponible.

La ventaja de tener varios núcleos es que podríamos usar estos núcleos para extraer el paralelismo a nivel de subprocesos en un programa y, por lo tanto, aumentar el rendimiento de un solo programa. Se han realizado muchas investigaciones en esta área. Algunas de las ventajas de OpenMP incluyen: buen rendimiento, portátil (es compatible con una gran cantidad de compiladores), requiere muy poco esfuerzo de programación y permite que el programa sea paralelos incrementalmente.

4. El algoritmo de Dijkstra

El algoritmo de Dijkstra es uno de los principales usados para encontrar las rutas más cortas en un gráfico ponderado por bordes y / o vértices. Dado un vértice de inicio en particular, encuentra el camino más corto desde todos los demás vértices del gráfico, incluido el destino deseado.

Suponiendo que el camino más corto de s a t en la gráfica G pasa a través del nodo intermedio x . Es claro que esta ruta debe contener la ruta más corta de s a x como un prefijo, de lo contrario, podríamos acortar la ruta s a t usando el prefijo s a x . Por lo tanto tenemos que encontrar primero la ruta más corta de s a x antes de encontrar la más corta entre s y t .

Este algoritmo procede a hacer ciclos en los que en cada uno de ellos calcula la ruta más corta de x a un nuevo vértice. Esto sugiere una estrategia dinámica. La ruta más corta de s a sí mismo es trivial a menos que hayan

pesos negativos. Una vez que se determina la ruta a un nodo x , se revisan todos los adyacentes a x para ver si hay un ,mejor camino desde s a un otro vértice atravesando a x .

Esto se puede explicar como referencia los siguientes pasos:

1. Seleccionar vértice de partida, es decir un origen.
2. Marcar el punto de partida como el punto de inicio.
3. Determinar los caminos especiales desde el nodo de partida, es decir, el de inicio.
4. Camino especial es aquel que solo puede trazarse a través de los nodos o vértices ya marcados.
5. Para cada nodo no marcado, se debe determinar si es mejor usar el camino especial antes calculado o si es mejor usar el nuevo camino especial que resulte al marcar este nuevo nodo.
6. Para seleccionar un nuevo nodo no marcado como referencia, deberá tomarse aquel cuyo camino especial para llegar a él es el mínimo, por ejemplo si anteriormente marqué el nodo o vértice 2, el cual tiene dos nodos adyacentes 3 y 4 cuyo peso en la arista corresponde a 10 y 5 respectivamente, se tomará como nuevo nodo de partida el 4, ya que el peso de la arista o camino es menor.
7. Cada camino mínimo corresponde a la suma de los pesos de las aristas que forman el camino para ir del nodo principal al resto de nodos, pasando únicamente por caminos especiales, es decir nodos marcados.

La complejidad del algoritmo de Dijkstra en su versión serial es de $O(n^2)$ si no se usa una cola de prioridad, como es el caso de las implementaciones realizadas para este proyecto. Es igualmente costos calcular la ruta más corta de un vértice s a cualquier otro vértice como lo es calcular la ruta más corta de cualquier vértice a s . Por esto, se puede decir que el nodo origen no afecta el rendimiento.

```

ShortestPath-Dijkstra( $G, s, t$ )
     $known = \{s\}$ 
    for  $i = 1$  to  $n$ ,  $dist[i] = \infty$ 
    for each edge  $(s, v)$ ,  $dist[v] = w(s, v)$ 
     $last = s$ 
    while ( $last \neq t$ )
        select  $v_{next}$ , the unknown vertex minimizing  $dist[v]$ 
        for each edge  $(v_{next}, x)$ ,  $dist[x] = \min[dist[x], dist[v_{next}] + w(v_{next}, x)]$ 
         $last = v_{next}$ 
         $known = known \cup \{v_{next}\}$ 

```

Figura 1: Algoritmo de Dijkstra.

Hay muchas aplicaciones para el algoritmo de Dijkstra, algunas de las más relevantes las podemos encontrar en los mapas, por ejemplo Google Maps u otros servicios de navegación. Una aplicación particularmente relevante es en la epidemiología. Una versión de este tipo de algoritmo puede ser usada en el cálculo del tamaño esperado del brote de un simple contagio en una red de contactos conocida, generalmente esta operación se lleva a cabo mediante una simulación computacionalmente intensiva que permite modelar todas las redes que cambian dinámicamente cuando el contagio solo puede viajar hacia adelante en el tiempo.

5. Algoritmo utilizado y su implementación

Primero que nada, es importante aclarar que nos basamos en código encontrado en [6] con algunas modificaciones, ya que se generaba una condición de carrera que será explicada más adelante.

¿Cómo correr el programa? El programa recibe dos parámetros desde la terminal, uno para indicar el número de nodos que tendrá el grafo aleatorio (llamado nv dentro del programa) y otro para indicar si se deben imprimir los resultados. Aunque esta última opción parece un poco extra o tonta, fue muy útil para las pruebas de rendimiento.

El programa comienza llenando la matriz de adyacencia del grafo. Como el programa crea grafos ponderados bidireccionales, no se ingresan valores negativos a la matriz. Primero se llena una mitad y después los valores son copiados a la otra, ya que la matriz debe ser simétrica.

Después, se inicializan los valores de un arreglo de nv espacios que indica si un nodo todavía no ha sido visitado. Al nodo 0 le corresponde el espacio 0 de este arreglo, al 1 el espacio 1 y así sucesivamente hasta $nv-1$. Cuando existe un valor de 1, entonces el nodo todavía no ha sido visitado. A este

arreglo se le conoce como `notDone`.

Junto al arreglo anterior se inicializan los valores de otro arreglo llamado `mind`, que almacena las distancias mínimas almacenadas hasta el momento hacia cada nodo. Por defecto, estas distancias se inicializan con el valor de la arista que conecta directamente a cualquier nodo con el nodo 0. Además, este arreglo almacena datos de tipo `unsigned`, ya que esta versión del algoritmo que no utiliza una cola de prioridad no funciona con pesos negativos.

Teniendo todo listo comienza el algoritmo en sí. Se encuentra el constructor `parallel`, lo que significa que a partir de este punto se utilizarán varios hilos. Primero, se inicializan tres variables, `step`, `mymv` y `mymd`. La primera variable indica que nodo se está revisando en un momento. Cada nodo es revisado una vez, por esta razón `step` va desde 0 hasta `nv-1`. La segunda y la tercera sirven para almacenar de manera local para los hilos cual es el nodo más cercano al nodo inicial que han encontrado de manera individual.

Después, en un bloque `single`, se inicializa los valores de `mv` y `md`. Estas variables almacenan el nodo más cercano al nodo inicial de entre todos los encontrados por todos los hilos. La variable `md` se inicializa en -1 (Que, como las distancias son de tipo `unsigned`, en realidad representa el mayor entero posible) y `mv` en 0. Esto es así ya que la distancia mínima encontrar al inicio del algoritmo es presuntamente infinita y el vértice hacia el que hay esa distancia es presuntamente inexistente hasta que se indique lo contrario.

Después, en una estructura `omp for`, cada hilo trabaja con un subconjunto de los nodos adyacentes al nodo 0, cada uno determina cual es el más cercano de su subconjunto. Luego, en una estructura `critical`, cada hilo pasa uno por uno a revisar si su menor valor encontrado es menor al ya existente. Si es así, se actualizan `mv` y `md` para referir al nodo más cercano en la iteración.

En una estructura `single`, se marca a este nuevo nodo más cercano como revisado, así que en la siguiente iteración ya no podrá ser seleccionado como el nodo más cercano al inicial.

Finalmente, en un ciclo `for` también paralelizado, se calculan distancias hacia todos los demás nodos y si es menor a la distancia más corta ya existente, se actualiza.

En este punto es donde se generaba la condición de carrera. OpenMP no asegura la manera ni el orden en cómo se ejecutarán las iteraciones de una estructura `for`. Por esta razón, un hilo podía terminar su parte primero y llegar a esta sección antes de que el otro, actualizando las distancias antes de que los demás hilos propusieran un nodo para ser el más pequeño en general. Para arreglar esto, se agregó una barrera después del `critical`, de esta manera se espera que todos los hilos actualicen `mv` y `md` si lo requieren

antes de continuar.

En la siguiente iteración se repite el proceso, solo que esta vez ya estará marcado el nodo más cercano que se utilizó en la iteración anterior, por lo que ahora se ocupará el segundo más cercano y así sucesivamente hasta que no haya más nodos adyacentes al nodo inicial.

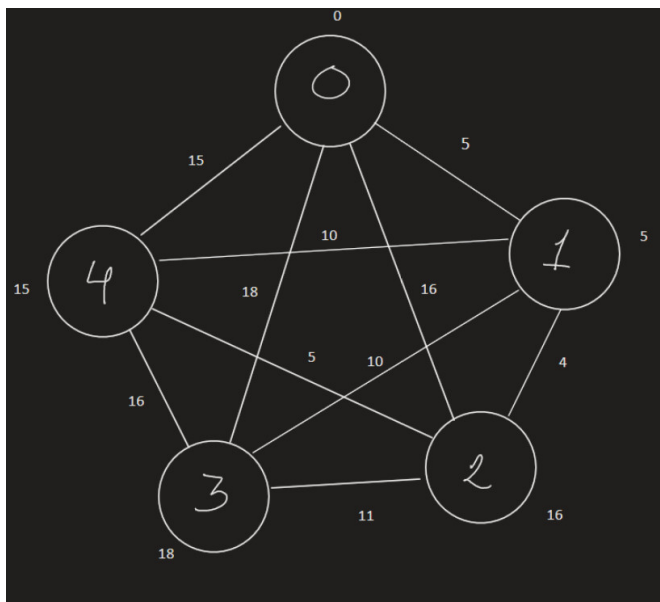


Figura 2: Grafo inicial

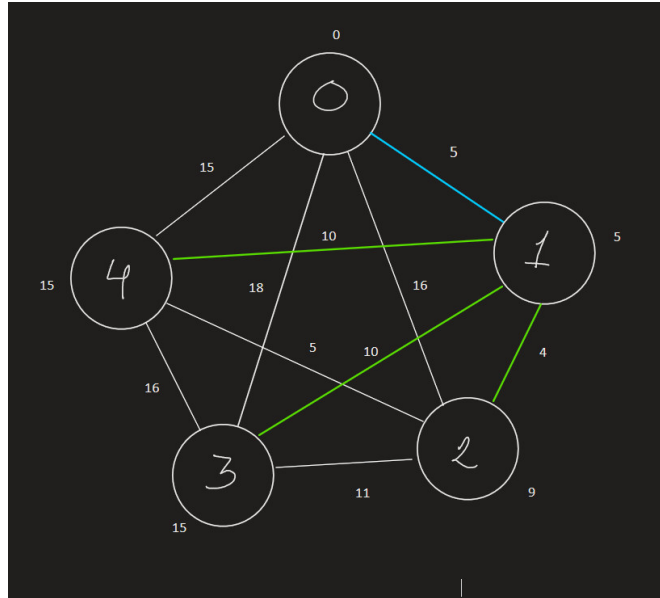


Figura 3: Primero se visita al nodo al nodo adyacente más cercano al origen.

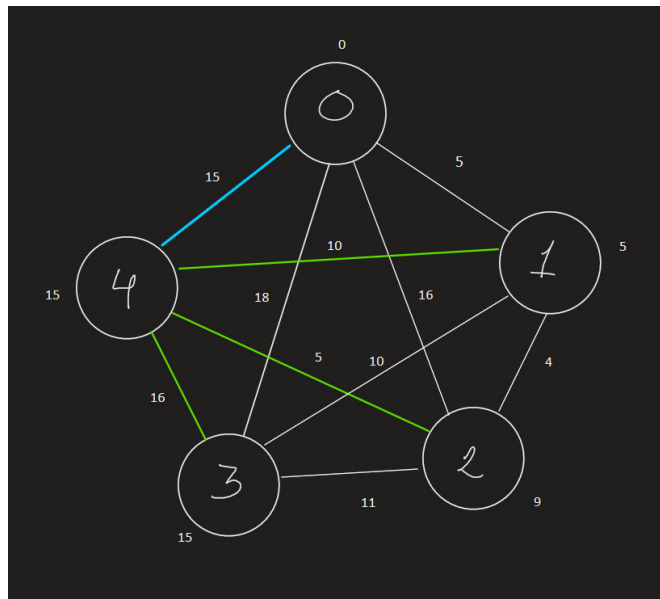


Figura 4: Después se visita el segundo más cerca.

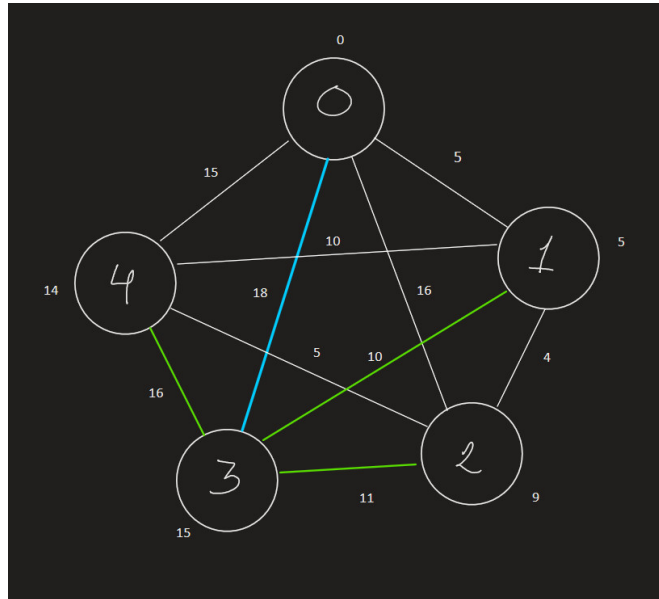


Figura 5: El algoritmo finaliza una vez que se visitó el más lejano.

6. Paralelización del Algoritmo

6.1. Nivel de paralelismo

Muchos algoritmos realizan cálculos atravesando iterativamente una gran estructura de datos por medio de un bucle. Por lo general, un bucle se ejecuta secuencialmente, lo que significa que los cálculos de la i -ésima iteración no se inician antes de que se completen todos los cálculos de la $(i - 1)$ ésima iteración. Este esquema de ejecución se denomina ciclo secuencial.

Si no hay dependencias entre las iteraciones de un bucle, las iteraciones se pueden ejecutar en un orden arbitrario y también se pueden ejecutar en paralelo por diferentes procesadores. Entonces, dicho bucle se denomina bucle paralelo.

En el caso del programa del Algoritmo de Dijkstra, cada hilo ejecuta una o más de las iteraciones, es decir, asume la responsabilidad de uno o más valores de i . Es por esto que podemos afirmar que el nivel de paralelismo que se maneja es a nivel de ciclo.

6.2. Métricas de desempeño

El principal propósito de escribir programas paralelos es aumentar el rendimiento con respecto a su contraparte secuencial. Para poder medir esto se establecieron las métricas de desempeño, que proporcionan elementos para decidir si los programas valen la pena o no.

El tiempo de procesamiento del programa del Algoritmo de Dijkstra fue tomado en tres distintos procesadores con el mismo grafo para todos los casos. Los procesadores utilizados fueron los siguientes:

Procesador 1: Intel i5, 2.50 GHz, 4 núcleos

Procesador 2: Intel i5, 1.4 GHz, 4 núcleos

Procesador 3: Intel i7, 2 GHz, 8 núcleos

Los resultados obtenidos se muestran a continuación:

| Procesador | Hilos | Tiempo en segundos |
|------------|-------|--------------------|
| 1 | 4 | 2.805 |
| 2 | 4 | 0.8661996 |
| 3 | 4 | 0.6124 |

| Procesador | Hilos | Tiempo en segundos |
|------------|-------|--------------------|
| 1 | 1 | 1.7194 |
| 2 | 1 | 0.5101114 |
| 3 | 1 | 0.4622 |

Figura 6: Para la toma del tiempo se utilizó la función `omp_get_wtime()` de la biblioteca `omp.h`

6.2.1. Speedup

Para el análisis de programas paralelos, una comparación con el tiempo de ejecución de una implementación secuencial es especialmente importante para ver el beneficio del paralelismo.

El speedup de una implementación paralela expresa el ahorro de tiempo de ejecución que se puede obtener utilizando una ejecución paralela en n procesadores en comparación con la mejor implementación secuencial.

Para el programa del Algoritmo de Dijkstra se obtuvo lo siguiente:

En los tres procesadores se observa que el valor del speedup es menor que la unidad. Esto quiere decir que el algoritmo ejecutado en forma paralela

| Procesador | Speedup |
|------------|---------|
| 1 | 0.6129 |
| 2 | 0.5889 |
| 3 | 0.7547 |

no presenta una mejora en cuanto al tiempo de ejecución, al contrario, hace que el programa se ejecute más lento.

6.2.2. Eficiencia

La eficiencia se define como la medida de la fracción de tiempo en la que cada procesador es usado para resolver el problema en cuestión de forma útil. En otras palabras, refleja el aprovechamiento de los recursos de hardware del sistema.

Los cálculos de eficiencias obtenidos fueron los siguientes:

| Procesador | Eficiencia |
|------------|------------|
| 1 | 0.1532 |
| 2 | 0.1472 |
| 3 | 0.1886 |

Si nuestros procesadores se utilizan de forma eficiente se espera que el tiempo de ejecución dedicado por cada procesador por el número de procesadores sea igual al tiempo de ejecución en un sólo procesador. Esto quiere decir que se requiere un valor cercano a la unidad. En base a los resultados obtenidos, se puede observar que están más cercanos a 0 que a 1, por lo que el programa no tiene una buena eficiencia.

6.2.3. Fracción serial

La fracción serial relaciona el speedup y la eficiencia con el propósito de tomar en cuenta otros factores además del tiempo.

Los resultados se muestran a continuación:

La fracción serial contenida en un código paralelo degrada el rendimiento. Si sumamos el valor de la fracción serial más el valor de la fracción paralela del código debemos de obtener un valor de uno. Esto quiere decir que el valor de la fracción serial debe ser menor que la unidad, cosa que no pasa con los resultados obtenidos.

| Procesador | Fracción serial |
|------------|-----------------|
| 1 | 1.8421 |
| 2 | 1.9307 |
| 3 | 1.4333 |

Después de obtener las métricas de desempeño podemos afirmar que el programa no presenta una mejora en cuanto a su versión secuencial. En un apartado posterior se hará un análisis más profundo de esto.

6.3. Formas de comunicación

Una región paralela es ejecutada por varios hilos que acceden a los mismos datos compartidos, por lo que es necesaria la sincronización para proteger las regiones críticas o evitar la condición de carrera. OpenMP ofrece varios constructores que se pueden utilizar para la sincronización y coordinación de hilos dentro de una región paralela. El constructor critical especifica una región crítica que solo puede ser ejecutada por un único hilo a la vez.

Dentro del código del Algoritmo de Dijkstra encontramos distintas partes en donde se presentan regiones críticas, es por ello que se utiliza el constructor critical y otros constructores que ayudan a la sincronización de hilos como single y barrier.

6.4. Granularidad

El tiempo de cálculo de una tarea se denomina granularidad: las tareas con muchos cálculos tienen una granularidad de grano grueso, las tareas con solo unos pocos cálculos son de grano fino.

En el caso del programa del Algoritmo de Dijkstra se presenta una granularidad de tipo fina, ya que en cada región paralela hay pocas instrucciones. Lamentablemente, esto afecta directamente al desempeño del programa debido a que hay una gran sobrecarga de comunicación y sincronización de los hilos.

7. Pruebas

Se realizaron múltiples pruebas para analizar el rendimiento del programa. Para ellos fue necesario ejecutar en tres diferentes procesadores las diferentes versiones y analizar su comportamiento.

Empezaremos comparando a la versión paralela consigo misma. En la Figura 2 se puede ver que sin importar el procesador que se use, la tendencia es que al usar más hilos, aumenta el tiempo de ejecución del programa. Esto se hace más evidente dependiendo del número de núcleos con los que cuenta la computadora.

Tiempo de Ejecución 1 vs 4 Hilos

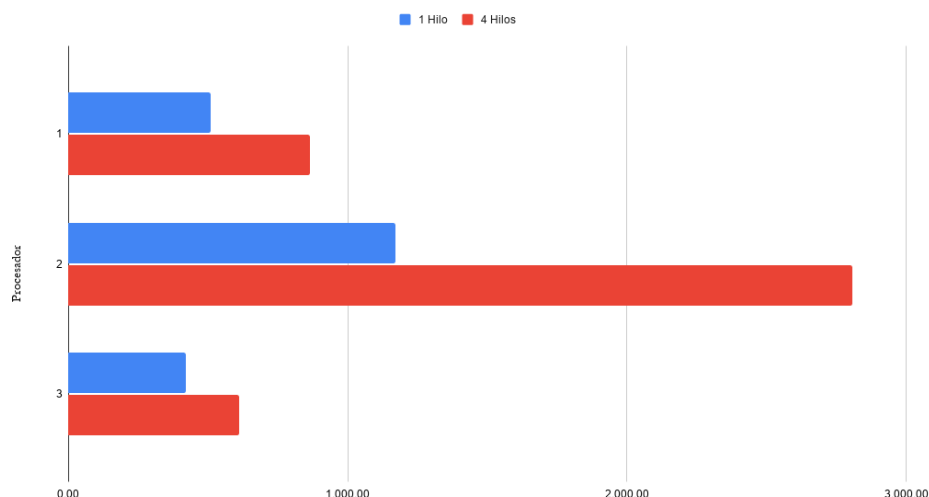


Figura 7: Ejecución en segundos usando 3 diferentes procesadores.

Ahora, si comparamos el rendimiento de la versión serial contra la paralela (usando 4 hilos, es claro que el algoritmo lineal es más eficiente. La diferencia crece conforme aumenta el tamaño de los datos.

Algo importante a notar aquí es que el tiempo de ejecución depende del tamaño del problema, es decir, cuando tenemos pocos nodos resulta poco conveniente implementar paralelismo. Mientras más crece la cantidad de nodos con la que se va a trabajar, la versión paralela del algoritmo se vuelve más eficiente, como lo indican las tablas de la Figura 9.

Hay diferentes factores que pudieran estar favoreciendo este comportamiento. El primero de ellos es la sección crítica presente en el código. Es necesaria por que en cada iteración, los hilos calculan sus valores de distancia mínima local a un cierto vértice, y luego actualizan los valores globales.

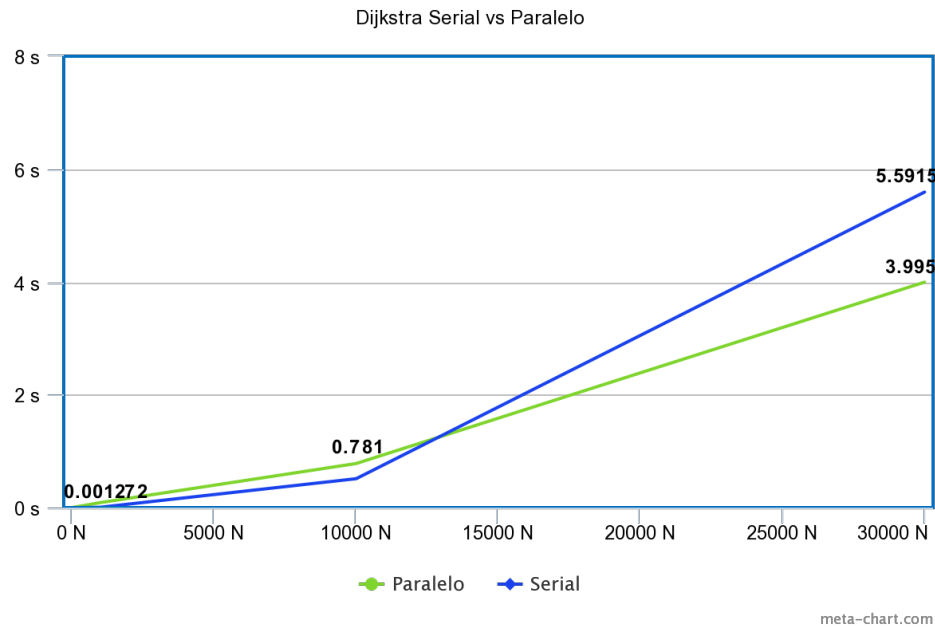


Figura 8: Rendimiento de programa serial vs paralelo.

Dado que la actualización debe ser atómica (sólo puede ser realizada por un hilo), esto provoca cierta serialización del programa.

Otro factor podría que es posible que ocurra False Sharing durante la ejecución. Esto se refiere a que cuando una parte del programa intenta acceder en el ciclo a datos que no serán alterados pero que comparten bloques de caché con datos que sí lo son, como en este caso las variables de distancia, el protocolo de caché puede forzar al programa a recargar todo el bloque a pesar de no ser estrictamente necesario, lo cual ocasiona esperas en la memoria además de perder eficiencia de procesador.

Por último, se puede considerar el balance de carga, particularmente en el ciclo. En realidad no se está definiendo una partición específica de las iteraciones que le corresponden a cada hilo del programa. Por un lado, las particiones grandes son buenas, debido a que hay menos sobrecarga: cada vez que un hilo termina un fragmento, debe pasar por la sección crítica, que serializa nuestro programa paralelo y, por lo tanto, ralentiza las cosas. Por otro lado, si los tamaños de las particiones son grandes, entonces hacia el final del trabajo, algunos hilos pueden estar trabajando en sus últimos fragmentos mientras que otros han terminado y ahora están inactivos, renunciando así

| Nodos | Hilos | Tiempo (s) |
|-------|-------|------------|
| 3000 | 1 | 0.048 |
| 3000 | 2 | 0.118 |
| 3000 | 4 | 0.193 |

| Nodos | Hilos | Tiempo (s) |
|-------|-------|------------|
| 30000 | 1 | 4.2 |
| 30000 | 2 | 3.82 |
| 30000 | 4 | 3.09 |

Figura 9: Rendimiento de programa paralelo con diferente cantidad de nodos.

a una posible mejora de la velocidad.

Por lo tanto, sería beneficioso tener trozos grandes al comienzo de la ejecución, para reducir la sobrecarga, pero trozos más pequeños al final. Esto se puede hacer usando la cláusula `guided` o `schedule`.

En consecuencia, se puede decir que este algoritmo no es óptimo en su versión paralela dependiendo de qué se busque, a pesar de segmentar las tareas entre hilos, la versión serial resulta ser la mejor opción cuando se tienen pocos nodos. Todo lo mencionado anteriormente contribuye a que el rendimiento del programa no sea óptimo con pocos nodos, cuando esta cantidad aumenta, la distribución de tareas a través del paralelismo es finalmente suficiente para compensar el tiempo perdido en crear y sincronizar todos los hilos. Algo importante a notar aquí es que esto fue probado en programa de OpenMP y que el lenguaje, así como API, de implementación juega un rol en el rendimiento final. Además, sería posible hacer modificaciones al código utilizado con el fin de hacer su ejecución más rápida.

8. Conclusiones

Como suele ser el caso de la programación paralela, la mera paralelización de un programa no necesariamente lo hará más rápido. Operaciones como regiones críticas, barreras, etc. vuelven secuencial a un programa que de otro modo sería paralelo, le quita gran parte de su velocidad. Además, existen problemas de transacciones de coherencia de caché, false sharing, etc.

El algoritmo en versión secuencial es más rápido cuando se tiene un número pequeño de nodos, pero si aumentamos este mismo número, la versión paralela empieza a tener mejores resultados y se obtiene un speedup de 1.3.

La razón de esto se debe a diversos factores. Primero hay que recordar que la complejidad del algoritmo secuencial de Dijkstra es de $O(n^2)$; esto quiere decir que mientras más nodos tengamos, el algoritmo tarda más en su ejecución. Por otro lado, en la versión paralela, al aumentar el tamaño del grafo cada hilo trabaja de forma más independiente con sus nodos asignados, por lo que el tiempo de comunicación y sincronización entre hilos se vuelve algo más insignificante.

Referencias

- [1] ARJOMANDI, E.(1975). *A study of parallelism in graph theory*, Doctoral thesis, TR 86, Dept. of Computer Science, University of Toronto.
- [2] BENTLEY, J. L.(1979). *A tree machine for searching problems.*, Proc. International Conference on Parallel Processing, pp. 257-266.
- [3] DEO, N., PANG, C. Y.(1980).. *Shortest path algorithms: taxonomy and annotation*, Computer Science Department, Washington State University, Pullman, Wash., Technical Report No. CS-80-057.
- [4] LIN F. Z., ZHAO N.(1999). *Parallel Implementation of Dijkstra's Algorithm*, pp COP5570.
- [5] JIA S., YIN X., LI X.(2012) *Mobile Robot Parallel PF-SLAM*, IEEE, International Conference on Robotics and Biomimetics.
- [6] MATLOFF, N. *Programming on Parallel Machines*, University of California, Davis, <http://heather.cs.ucdavis.edu/matloff/158/PLN/ParProcBook.pdf>.

- [7] CAOLA, H., WANGB, F., XIN FANG, TU H. L. (2012). *OpenMP Parallel Optimal Path Algorithm and Its Performance Analysis*, Jun Shi World Congress on Software Engineering, DOI 10.1109.
- [8] JASIKA N., ALISPAHIC N., ELMA A., ILVANA K., ELMA L. Y NOSOVIC, N. (2012). *Faster Parallel Algorithm for Approximate Shortest Path*, Proceedings of the 35th International Convention MIPRO, Opatija, pp. 1811-1815.
- [9] LI J. (2020) *Programming on Parallel Machines*, Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), Chicago, USA, 14 pages. <https://doi.org/10.1145/3357713.3384268>.
- [10] PANITANARAK, T. MADDURI, K. *Performance Analysis of Single-source Shortest Path Algorithms on Distributed-memory Systems*, Department of Computer Science and Engineering, The Pennsylvania State University University, USA.
- [11] STEFANO G., PETRICOLA A., ZAROLIAGIS C.(2006) *On the implementation of parallel shortest path algorithms on a supercomputer*, Proceedings of ISPA'06, pp. 406-417.
- [12] MATETI, P., DEO, N. (1982) *Parallel algorithms for the single source shortest path problem*, Computing 29, 31-49, <https://doi.org/10.1007/BF02254849>.
- [13] PARHAMI B. (2002) *Introduction to Parallel Processing, Algorithms and Architectures*, Springer US, <https://doi.org/10.1007/b116777>.
- [14] THOMAS H. SPENCER. (1991) *More time-work tradeoffs for parallel graph algorithms*, Proceedings of the third annual ACM symposium on Parallel algorithms and architectures (SPAA '91). Association for Computing Machinery, New York, NY, USA, pp 81-93, <https://doi.org/10.1145/113379.113387>.
- [15] CRAUSER A., MEHLHORN K., MEYER U., SANDERS P. (1998). *A parallelization of Dijkstra's shortest path algorithm*, Mathematical Foundations of Computer Science 1998, Lecture Notes in Computer Science, vol 1450. Springer, Berlin, Heidelberg. <https://doi.org.pbbidi.unam.mx:2443/10.1007/BFb0055823>.

- [16] KULKARNI P. ,PATHARE S. (2014) *Performance Analysis of Parallel Algorithm over Sequential using OpenMP*, IOSR Journal of Computer Engineering. 16. 58-62. 10.9790/0661-162105862.
- [17] RAUBER, T. , RÜNGER, G. (2010) *Parallel Programming: for Multi-core and Cluster Systems*, Heidelberg: Springer.
- [18] PARDO,F. (2004) *Arquitectura e ingeniería de los computadores*, Valencia: Universidad de Valencia.
- [19] JIMÉNEZ, D. (2013) *Introducción a las arquitecturas paralelas*, Universitat Oberta de Catalunya.
- [20] PACHECO, P. (2011) *An Introduction to Parallel Programming*, MA, USA: Morgan Kaufmann.