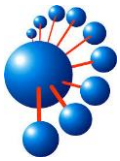


Proyecto Semestral:
Paralelizar Algoritmo Floyd-Warshall

Introducción a la Computación Paralela (2023-1)

Nombre: Dazhi Feng Zong
Profesora: Cecilia Hernández R.
Fecha: 17 de julio, 2023

Introducción

El problema por resolver en este proyecto consiste en la paralelización del algoritmo de Floyd-Warshall con **vectorización**, **OpenMP** y **CUDA**. El algoritmo es utilizado para encontrar todos los caminos más cortos en un grafo ponderado dirigido o no dirigido. Este algoritmo es ampliamente utilizado en diversos campos, como en redes de comunicaciones, rutas de navegación, optimización de rutas de entrega, entre otros. La solución óptima del problema tiene una complejidad de $O(V^3)$, donde V es el número de vértices en el grafo.

El algoritmo de Floyd-Warshall es relevante debido a su uso en problemas del mundo real, donde se requiere encontrar caminos más cortos en grafos grandes. Sin embargo, su implementación secuencial puede ser un poco ineficiente para grafos de gran tamaño debido a su complejidad. Por lo tanto, la paralelización de este algoritmo puede ser beneficioso para reducir el tiempo de ejecución y mejorar la eficiencia.

Entre otras alternativas para encontrar los caminos más cortos en un grafo, existe:

- **Algoritmo de Dijkstra:** Este algoritmo encuentra el camino más corto desde un nodo origen a todos los demás nodos en un grafo ponderado dirigido o no dirigido. A diferencia de Floyd-Warshall, que encuentra todos los caminos más cortos entre todos los pares de nodos, Dijkstra encuentra los caminos más cortos desde un solo nodo origen.
- **Algoritmo de Bellman-Ford:** Similar a Dijkstra, encuentra el camino más corto desde un nodo origen a todos los demás nodos en un grafo ponderado dirigido o no dirigido. Sin embargo, a diferencia de Dijkstra, funciona con grafos con pesos negativos.

Otras alternativas para paralelizar el algoritmo:

- MPI
- Cilk
- Frameworks de programación paralela como Intel TBB y OpenCL
- Otros

Objetivos

- Paralelizar con vectorización, OpenMP y CUDA.
- Comparar y evaluar el rendimiento de cada paralelización.
- Comprobar la reducción de tiempo que se puede lograr con la paralelización del algoritmo.

Desarrollo

Secuencial: El algoritmo secuencial (normal) consiste en un bucle triple anidado con las variables k, i y j. Aquí, k representa un nodo intermedio, i representa el nodo de origen y j representa el nodo de destino. En cada iteración, se verifica si el recorrido desde i hasta k y luego de k a j es más corto que ir directamente de i a j. Luego, se actualiza la matriz de distancias con la ruta más corta para ese nodo y se repite este proceso para todos los nodos.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

Vectorización: Se utiliza instrucciones SIMD para generar vectores que contienen las distancias para 8 o 16 nodos de destino correspondientes a un nodo de origen específico. Es decir, para cada valor de i, se verifican simultáneamente 8 o 16 valores de j. Para ello, se crean vectores de tamaño 8 o 16 para la fila i y la fila k, y después se realizan operaciones entre estos vectores como sumar y mínimo. Se hace lo mismo para los 8 o 16 elementos próximos, y así, hasta que termine el bucle j. Si el tamaño de la matriz no es divisible por 8 o 16, quedan elementos restantes que no pueden ser vectorizados y en estos casos se procesan en forma secuencial.

Paralelización con OpenMP: Se paraleliza el bucle i, es decir, cada nodo de origen se procesa de manera simultánea, y se verifica su correspondiente conjunto de nodos destino de forma concurrente.

Paralelización con OpenMP y vectorización: Se paraleliza el bucle i y se vectorizan 8 elementos de esa fila. Cada nodo de origen se procesa de manera simultánea, y se verifica con 8 nodos destinos a la vez.

Paralelización con CUDA: Se divide la matriz de distancias en bloques cuadrados de igual tamaño, creando submatrices. Estos bloques pueden ejecutarse secuencialmente o en paralelo, según los recursos disponibles. Para cada valor de k, se recorren estos bloques y cada bloque ejecuta el kernel con el algoritmo. Cada bloque en CUDA está compuesto por múltiples hilos que se ejecutan en paralelo. El tamaño máximo para un bloque es de 32x32, lo que equivale a 1024 hilos en paralelo.

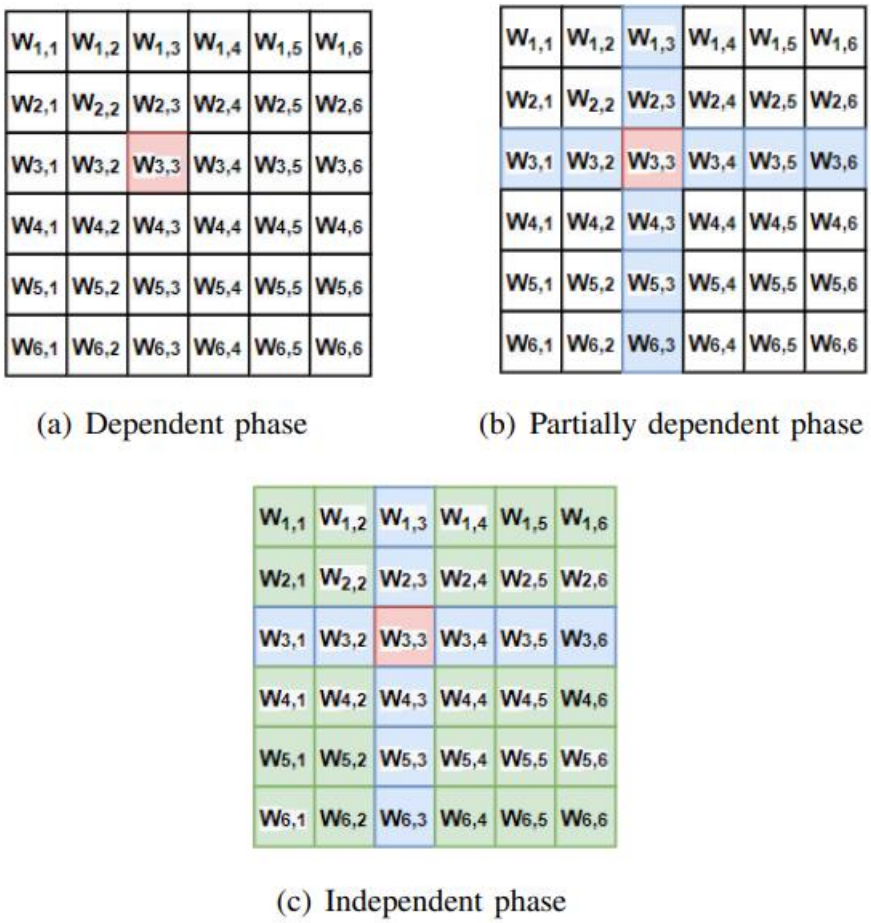
Paralelización en bloques con OpenMP: Se divide la matriz de distancias en bloques cuadrados de igual tamaño para crear submatrices. A diferencia del algoritmo CUDA, en este caso **se recorre cada bloque de la diagonal** de la nueva matriz que contiene los bloques.

Para cada bloque perteneciente a la diagonal, se realiza lo siguiente en orden:

1. Se calcula el camino más corto para el **bloque actual k**, que forma parte de la diagonal.
2. Se calcula el camino más corto para los **bloques de la fila k y columna k**, es decir, los bloques que se encuentran en la misma fila o columna que el bloque actual k.
3. Se calcula el resto de los bloques.

Este orden es necesario, ya que para calcular el camino más corto de la fila o columna k, depende del bloque k,k (bloque actual en la diagonal). Asimismo, para calcular el resto de los bloques, depende de los bloques de la fila y columna k.

En el ejemplo mostrado en la imagen, se realiza el cálculo del camino más corto siguiendo los pasos: Se calcula el camino más corto para el bloque actual que pertenece a la diagonal (marcado en rojo en la imagen). Después, se calcula el camino más corto para los bloques que comparten la misma fila o columna que el bloque actual (marcados en azul). Finalmente, se realiza el cálculo del camino más corto para el resto de los bloques que no fueron incluidos en los pasos anteriores (marcados en verde).



(a) Dependent phase (b) Partially dependent phase

(c) Independent phase

Evaluación

Análisis Teórico

Secuencial: La complejidad del algoritmo secuencial es de $\boldsymbol{O(n^3)}$ (donde n es el número de vértices), debido al triple bucle for que se tiene que recorrer.

Vectorización: La complejidad del algoritmo vectorizado sigue siendo $O(n^3)$, ya que el último ciclo for sigue siendo lineal, debido a que solo se paralelizan 8 o 16 elementos a la vez.

$$O\left(\frac{n}{8}\right) = O(n) \Rightarrow O(n) \times O(n^2) = \boldsymbol{O(n^3)}$$

Paralelización con OpenMP: Dado que se paraliza solo el bucle i, por Teorema de Brent:

$$\mathrm{Tp} \leq \mathrm{W/p} + \mathrm{D}$$

$$\mathrm{Tp} \leq O(n^3)/p + O(n^2)$$

$$\boldsymbol{p = n} \implies \mathrm{Tp} \leq O(n^2) + O(n^2) \Leftrightarrow \mathrm{Tp} \leq \boldsymbol{2O(n^2)} \implies \boldsymbol{O(n^2)}$$

Paralelización con OpenMP y vectorización: Como se paraliza el bucle i y se escogen 8 elementos del bucle j:

$$\mathrm{Tp} \leq O(n^3)/8p + O(n^2)/8$$

$$\boldsymbol{8p = n} \implies \mathrm{Tp} \leq O(n^2) + O(n^2)/8 \Leftrightarrow \mathrm{Tp} \leq \boldsymbol{\frac{9}{8}O(n^2)} \implies \boldsymbol{O(n^2)}$$

Paralelización con CUDA: Al paralelizar el bucle i y bucle j, la complejidad temporal del algoritmo CUDA depende del tamaño del bloque,

$$\frac{\boldsymbol{O(n^3)}}{\boldsymbol{B^2}}, con\ \boldsymbol{B = n} \implies \frac{O(n^3)}{n^2} \Leftrightarrow \boldsymbol{O(n)}$$

donde B es el tamaño del bloque y entonces, B^2 es el número de hebras ejecutándose en paralelo. Si el tamaño de bloque no es igual al tamaño del grafo, se tiene que recorrer el bucle k y todos los bloques, entonces:

$$\frac{\boldsymbol{O(n^3)}}{\boldsymbol{(n/B)^2}} \Leftrightarrow O(nB^2) \Longrightarrow \boldsymbol{O(n)}$$

donde $(n/B)^2$ es el número de bloques. Como CUDA también paraleliza el recorrido de los bloques (B^2), si hay recursos disponibles, la complejidad también es lineal.

Paralelización en bloques con OpenMP: En este caso, se paraleliza el recorrido de los bloques, pero cada bloque de la diagonal se tiene que seguir ejecutando de forma secuencial.

$$O(B) \times [O(b^3) + O\left(\frac{B}{p} \times b^3\right) + O\left(\frac{B}{p} \times (b^3 + B \times b^3)\right)]$$

donde $\boldsymbol{B = \frac{n}{b}}$ es el número de bloques y $\boldsymbol{b = \frac{n}{B}}$ tamaño de cada bloque. Entonces:

$$O\left(\frac{n}{b}\right) \times [O(b^3) + O\left(\frac{n/b}{p} \times b^3\right) + O\left(\frac{n/b}{p} \times (b^3 + \frac{n}{b} \times b^3)\right)]$$

Al paralelizar B, nos conviene tener un b bajo y un B alto (se "elimina" b):

$$O(n) \times \left[O\left(\frac{n}{p}\right) + O\left(\frac{n}{p} \times n\right)\right], con\ \boldsymbol{p = n} \implies O(n) \times [O(1) + O(n)] \implies \boldsymbol{O(n^2)}$$

De lo contrario, si b es un valor muy alto cercano a n:

$$O(1) \times [O(n^3) + O\left(\frac{1}{p} \times n^3\right) + O\left(\frac{1}{p} \times 2n^3\right)] \Longrightarrow 4\boldsymbol{O(n^3)}$$

Resultados

Para el análisis experimental se utilizaron matrices de https://sparse.tamu.edu/
Se realizaron 10 pruebas para las ejecuciones rápidas (menor a 30 min.) y solo 2 pruebas para las ejecuciones más lentas. Luego se calculó el promedio para las 10 o 2 ejecuciones.
Las pruebas se realizaron en:

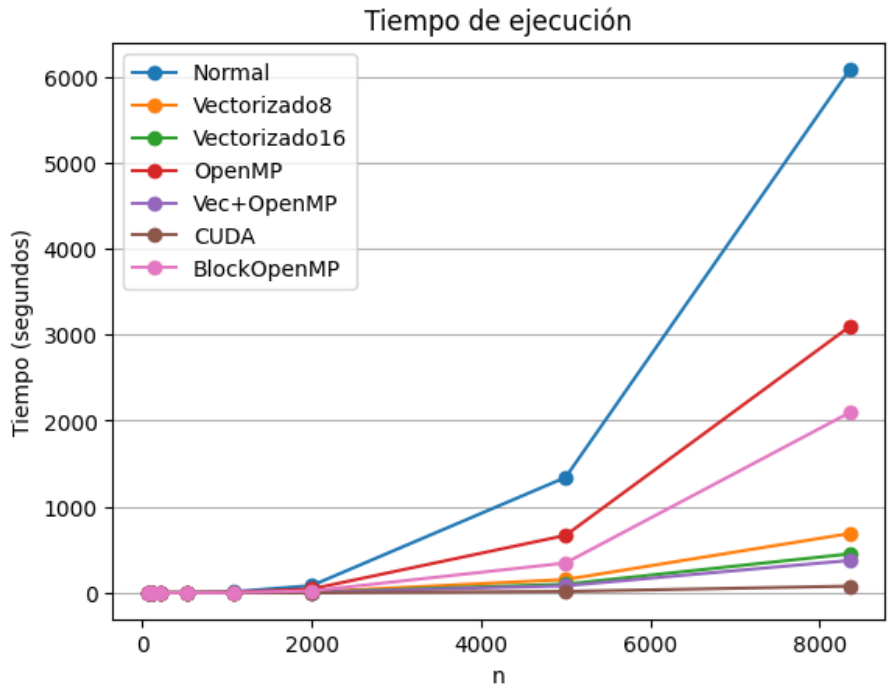
- Procesador con 6 procesadores principales, 12 procesadores lógicos
- RAM 8,00 GB
- GPU NVIDIA GeForce GTX 1650

Tabla con los resultados (**milisegundos**)

Se aproximaron los decimales. Valores en amarillo el mejor rendimiento y en rojo el peor.

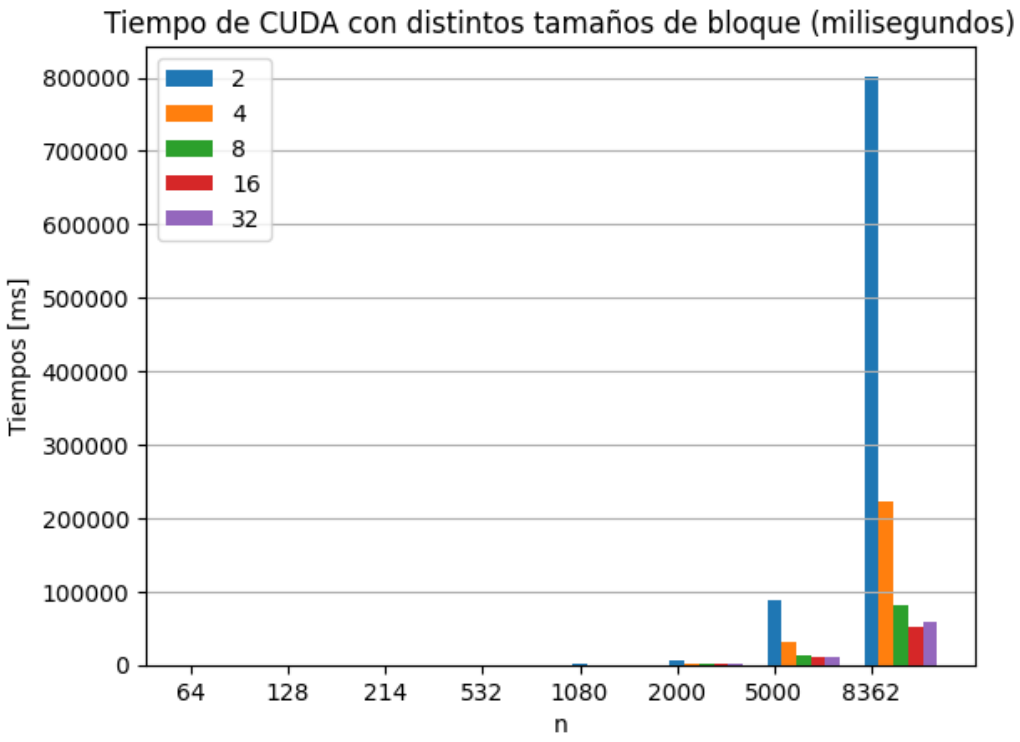
Tabla 1: Número de vertices y tiempo en milisegundos [ms]							
n	Normal	Vec8	Vec16	OpenMP	Vec+OpenMP	CUDA	BlockOpenMP
64	4	0	0	3	0	8	5
128	22	2	1	18	1	13	17
214	112	12	8	70	7	15	55
532	1.577	183	111	860	108	36	486
1.080	13.463	1.482	920	7.179	766	154	3.292
2.000	83.954	10.135	6.239	44.833	5.210	862	22.001
5.000	1.341.107	154.406	100.318	667.092	81.281	15.459	346.436
8.362	6.085.038	689.809	452.683	3.096.147	376.512	77.726	2.097.657

Gráfico de los resultados (**segundos**)



Análisis

- Cualquier algoritmo es más eficiente que el secuencial, especialmente en grafos/matrices grandes.
- Buena escalabilidad, al aumentar el “n”, los algoritmos paralelos siguen siendo más rápidos.
- En el caso de los algoritmos de bloques, se aprovecha la localidad para mejorar la eficiencia del acceso a la memoria caché. Como no es posible almacenar toda la matriz de distancias en la caché debido a su tamaño, se utilizan bloques más pequeños. Esto permite que se carguen y almacenen en la caché solo los bloques en uso. Al hacerlo, se mejora la localidad de referencia, lo que implica que las lecturas y escrituras en la memoria caché se realizan de manera más rápida y eficiente.
- Para grafos más pequeños, la elección del algoritmo no es muy importante, ya que la diferencia en tiempos de ejecución es mínima. Pero, es importante considerar el tiempo necesario para inicializar componentes como la GPU o los procesadores en OpenMP.
- Dado el alto poder de procesamiento que ofrece la GPU, con la capacidad de ejecutar 1024 operaciones en paralelo, su rendimiento es muy superior en comparación a otros algoritmos.
- En el caso de los algoritmos vectorizados, las diferencias no son muy significativas. Sin embargo, existe una notable diferencia entre los algoritmos con vectorización y los paralelizados solo con OpenMP.
- Esta diferencia se debe al número limitado de procesadores disponibles en OpenMP, ya que los algoritmos paralelizados requieren una mayor cantidad de procesadores para alcanzar su máximo rendimiento teórico. Por lo tanto, estos algoritmos tienen potencial de mejora al aumentar la cantidad de procesadores, en comparación con los algoritmos vectorizados, que no mostrarán mejoras significativas (ya que siempre se paralelizarán 8 o 16 elementos).
- Además, existe una gran diferencia entre los dos algoritmos paralelizados con OpenMP, y esta diferencia puede incrementarse o decrementarse dependiendo del tamaño de bloque escogido.
- La elección del tamaño de bloque también afecta al algoritmo de CUDA. En el siguiente gráfico se muestra la diferencia en tiempos de ejecución con distintos tamaños de bloque (2, 4, 8, 16, 32). A partir de aproximadamente 1080 elementos, se comienza a notar una diferencia, y esta diferencia se vuelve aún más grande al aumentar el tamaño de la matriz. Dependiendo del tamaño de la matriz, un tamaño de bloque específico puede ser más efectivo y eficiente que otros. (No se realizó una comparación con la implementación de bloques OpenMP, debido a que el tamaño de la matriz debe ser necesariamente divisible por el tamaño del bloque, y en este caso solo el valor 2 cumple con ese criterio.)



Conclusión

En este proyecto, se paralelizó el algoritmo de Floyd-Warshall de distintas formas con vectorización, OpenMP y CUDA. El objetivo principal era reducir el tiempo de ejecución y mejorar la eficiencia de este algoritmo que es utilizado en la búsqueda de caminos más cortos en grafos con pesos.

Se implementaron distintos algoritmos paralelos que incluyen vectorización, OpenMP o CUDA. Cada uno de ellos presentó mejoras significativas en comparación con la implementación secuencial.

En cuanto al análisis teórico, se determinó que la paralelización con CUDA presenta la mejor complejidad temporal, seguida por la paralelización con OpenMP. Por otro lado, los algoritmos de vectorización mostraron complejidades similares al algoritmo secuencial.

En los resultados experimentales, se observó que todos los algoritmos paralelizados superaron el secuencial, especialmente en grafos o matrices de mayor tamaño. Además, se demostró una buena escalabilidad, ya que los algoritmos paralelos mantuvieron su rendimiento a medida que se aumentaba el tamaño del grafo.

En general, todos los algoritmos paralelizados demostraron ser más eficientes que la implementación secuencial, especialmente para grafos grandes. La elección de la estrategia de paralelización depende del contexto y de los recursos disponibles, como el procesador y la GPU. Sin embargo, entre todas las estrategias evaluadas, se observó que el uso de CUDA obtuvo los mejores resultados. Por lo tanto, si es posible utilizar CUDA, se recomienda debido a su alto poder y eficiencia.

Referencias

"Floyd-Warshall Algorithm." Wikipedia, 2023, https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
"Parallel All-Pairs Shortest Path Algorithm." Wikipedia, 2023, https://en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm
Pozder, N., Ćorović, D., Herenda, E., & Divjan, B. (2021). Towards performance improvement of a parallel Floyd-Warshall algorithm using OpenMP and Intel TBB. University of Sarajevo. https://www.researchgate.net/publication/350236410_Towards_performance_improvement_of_a_parallel_Floyd-Warshall_algorithm_using_OpenMP_and_Intel_TBB
Chauhan, Munesh. (2017). CUDA Analysis of Parallelization in Large Graph Algorithms. https://www.researchgate.net/publication/315489843_CUDA_Analysis_of_Parallelization_in_Large_Graph_Algorithms