

Paralelizar Algoritmo Floyd-Warshall

Nombre: Dazhi Feng Z.

Profesora: Cecilia Hernández R.

Introducción a la Computación Paralela (2023-1)

Introducción

El algoritmo Floyd-Warshall se utiliza para encontrar los caminos más cortos entre todos los pares de vértices en un grafo.

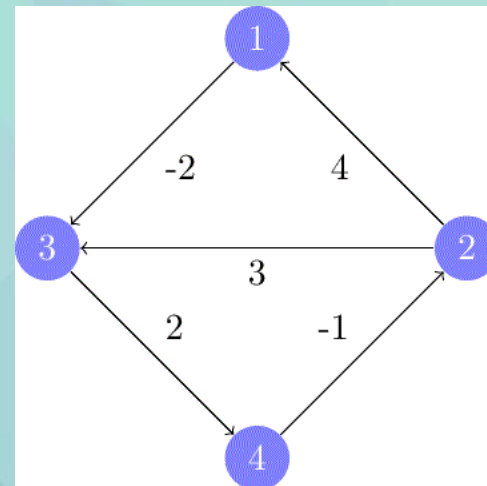
Utilizado en diversas aplicaciones, como en redes de comunicación, sistemas de navegación, optimización de rutas logísticas y análisis de redes sociales.

El algoritmo es secuencial y no aprovecha las capacidades de los sistemas multiprocesador o paralelos.

Dijkstra: Camino más corto desde **un** vértice a todos los demás vértices en un grafo.

Bellman-Ford: Camino más corto desde **un** vértice a todos los demás vértices en un grafo, **incluye aristas negativas**.

Floyd-Warshall: Caminos más cortos entre **todos** los pares de vértices en un grafo, **incluye aristas negativas**.



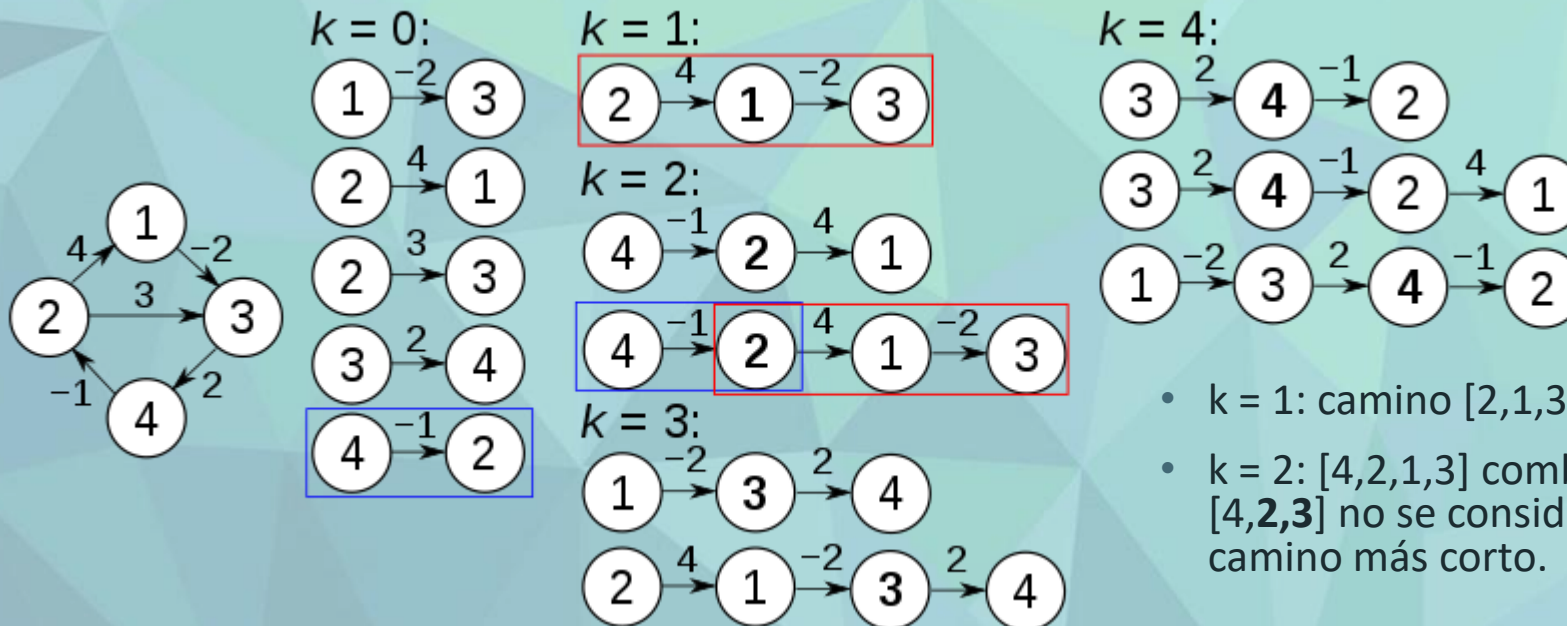
Algoritmo

```

let dist be a  $|V| \times |V|$  array of minimum distances initialized to  $\infty$  (infinity)
for each edge  $(u, v)$  do
     $\text{dist}[u][v] \leftarrow w(u, v)$  // The weight of the edge  $(u, v)$ 
for each vertex  $v$  do
     $\text{dist}[v][v] \leftarrow 0$ 
for  $k$  from 1 to  $|V|$ 
    for  $i$  from 1 to  $|V|$ 
        for  $j$  from 1 to  $|V|$ 
            if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$ 
                 $\text{dist}[i][j] \leftarrow \text{dist}[i][k] + \text{dist}[k][j]$ 
            end if

```

Utiliza una matriz para almacenar las distancias mínimas entre los pares de nodos del grafo. El algoritmo realiza una serie de iteraciones para actualizar la matriz y encontrar las distancias mínimas.



- $k = 1$: camino [2,1,3], reemplaza el camino [2,3]
- $k = 2$: [4,2,1,3] combinación de [4,2] y [2,1,3]. [4,2,3] no se considera, porque [2,1,3] es el camino más corto.

FW Paralelizado en bloques con OpenMP

- Dividir la matriz en bloques (submatrices) cuadrados con tamaños iguales.
- Se recorre **cada bloque de la diagonal** secuencialmente.
- Para cada bloque de la diagonal:
 1. Calcular FW para el bloque actual.
 2. Calcular FW para los bloques con la misma fila o columna que el bloque actual.
 3. Calcular FW para el resto de los bloques.

$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	$W_{1,4}$	$W_{1,5}$	$W_{1,6}$
$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	$W_{2,4}$	$W_{2,5}$	$W_{2,6}$
$W_{3,1}$	$W_{3,2}$	$W_{3,3}$	$W_{3,4}$	$W_{3,5}$	$W_{3,6}$
$W_{4,1}$	$W_{4,2}$	$W_{4,3}$	$W_{4,4}$	$W_{4,5}$	$W_{4,6}$
$W_{5,1}$	$W_{5,2}$	$W_{5,3}$	$W_{5,4}$	$W_{5,5}$	$W_{5,6}$
$W_{6,1}$	$W_{6,2}$	$W_{6,3}$	$W_{6,4}$	$W_{6,5}$	$W_{6,6}$

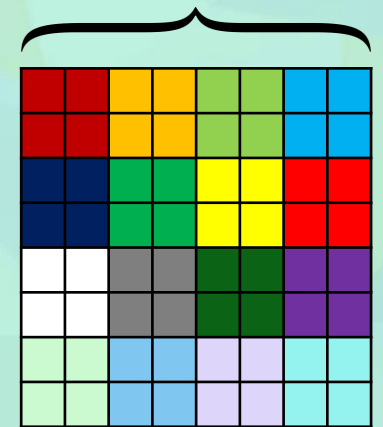


$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	$W_{1,4}$	$W_{1,5}$	$W_{1,6}$
$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	$W_{2,4}$	$W_{2,5}$	$W_{2,6}$
$W_{3,1}$	$W_{3,2}$	$W_{3,3}$	$W_{3,4}$	$W_{3,5}$	$W_{3,6}$
$W_{4,1}$	$W_{4,2}$	$W_{4,3}$	$W_{4,4}$	$W_{4,5}$	$W_{4,6}$
$W_{5,1}$	$W_{5,2}$	$W_{5,3}$	$W_{5,4}$	$W_{5,5}$	$W_{5,6}$
$W_{6,1}$	$W_{6,2}$	$W_{6,3}$	$W_{6,4}$	$W_{6,5}$	$W_{6,6}$



$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	$W_{1,4}$	$W_{1,5}$	$W_{1,6}$
$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	$W_{2,4}$	$W_{2,5}$	$W_{2,6}$
$W_{3,1}$	$W_{3,2}$	$W_{3,3}$	$W_{3,4}$	$W_{3,5}$	$W_{3,6}$
$W_{4,1}$	$W_{4,2}$	$W_{4,3}$	$W_{4,4}$	$W_{4,5}$	$W_{4,6}$
$W_{5,1}$	$W_{5,2}$	$W_{5,3}$	$W_{5,4}$	$W_{5,5}$	$W_{5,6}$
$W_{6,1}$	$W_{6,2}$	$W_{6,3}$	$W_{6,4}$	$W_{6,5}$	$W_{6,6}$

16 bloques de 2x2



$$V = 8$$

$$b = 2$$

$$B = 8/2 = 4$$

4x4 (16) bloques


Código

```
1 void floydb(vdd &C, const vdd &A, const vdd &B, const int b)
2 {
3     for (int k = 0; k < b; k++)
4         for (int i = 0; i < b; i++)
5             for (int j = 0; j < b; j++)
6                 C[i][j] = min(C[i][j], A[i][k] + B[k][j]);
7 }
```

floydb()


Recorrer los bloques de la diagonal

```
1 for (int k = 0; k < B; k++)
2 {
3     floydb(blocks[k][k], blocks[k][k], blocks[k][k], b);
4
5     #pragma omp parallel for
6     for (int j = 0; j < B; j++)
7         if (j != k)
8             floydb(blocks[k][j], blocks[k][k], blocks[k][j], b);
9
10    #pragma omp parallel for
11    for (int i = 0; i < B; i++)
12    {
13        if (i != k)
14            floydb(blocks[i][k], blocks[i][k], blocks[k][k], b);
15        for (int j = 0; j < B; j++)
16            if (j != k)
17                floydb(blocks[i][j], blocks[i][k], blocks[k][j], b);
18    }
19 }
```




W _{1,1}	W _{1,2}	W _{1,3}	W _{1,4}	W _{1,5}	W _{1,6}
W _{2,1}	W _{2,2}	W _{2,3}	W _{2,4}	W _{2,5}	W _{2,6}
W _{3,1}	W _{3,2}	W _{3,3}	W _{3,4}	W _{3,5}	W _{3,6}
W _{4,1}	W _{4,2}	W _{4,3}	W _{4,4}	W _{4,5}	W _{4,6}
W _{5,1}	W _{5,2}	W _{5,3}	W _{5,4}	W _{5,5}	W _{5,6}
W _{6,1}	W _{6,2}	W _{6,3}	W _{6,4}	W _{6,5}	W _{6,6}

1. Calcular FW para el bloque actual (el bloque pertenece a la diagonal).



W _{1,1}	W _{1,2}	W _{1,3}	W _{1,4}	W _{1,5}	W _{1,6}
W _{2,1}	W _{2,2}	W _{2,3}	W _{2,4}	W _{2,5}	W _{2,6}
W _{3,1}	W _{3,2}	W _{3,3}	W _{3,4}	W _{3,5}	W _{3,6}
W _{4,1}	W _{4,2}	W _{4,3}	W _{4,4}	W _{4,5}	W _{4,6}
W _{5,1}	W _{5,2}	W _{5,3}	W _{5,4}	W _{5,5}	W _{5,6}
W _{6,1}	W _{6,2}	W _{6,3}	W _{6,4}	W _{6,5}	W _{6,6}

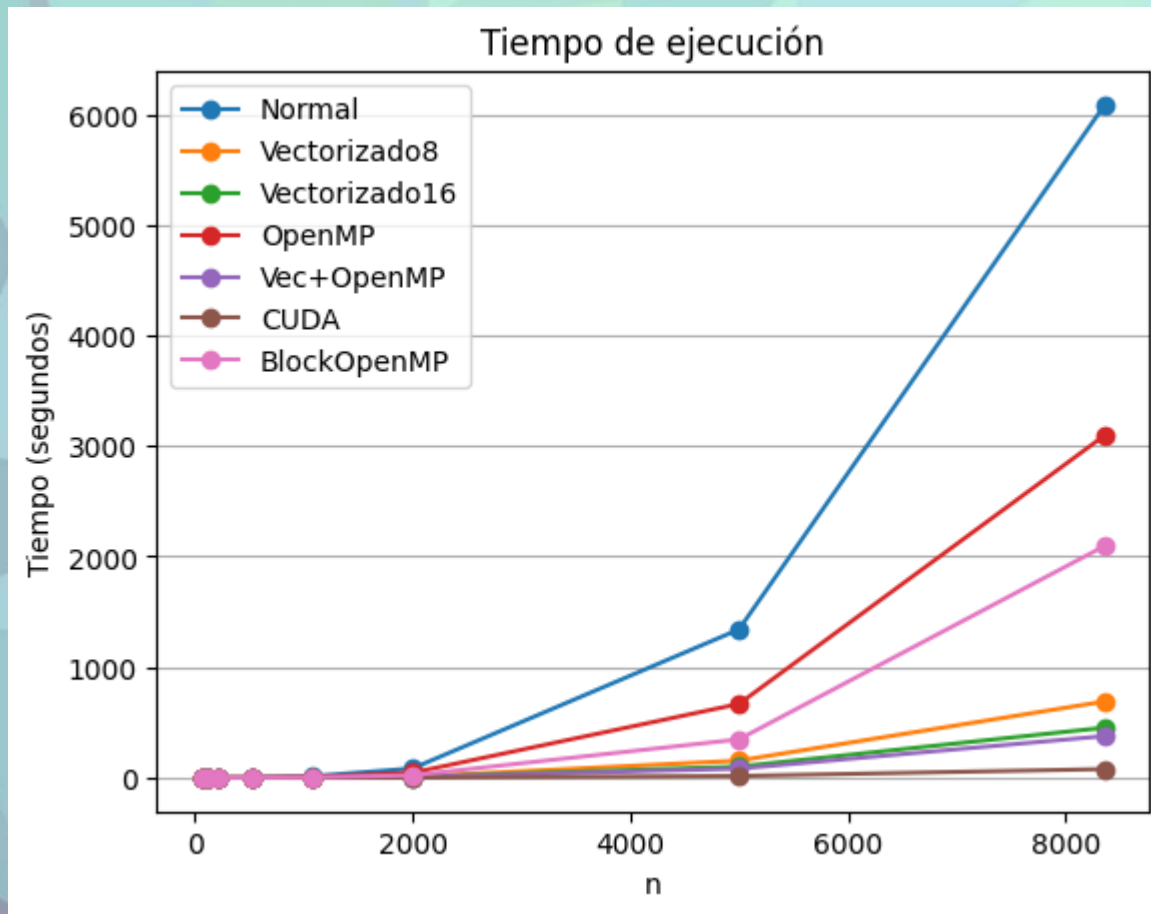
2. Calcular FW para los bloques con la misma fila o columna que el bloque actual.



W _{1,1}	W _{1,2}	W _{1,3}	W _{1,4}	W _{1,5}	W _{1,6}
W _{2,1}	W _{2,2}	W _{2,3}	W _{2,4}	W _{2,5}	W _{2,6}
W _{3,1}	W _{3,2}	W _{3,3}	W _{3,4}	W _{3,5}	W _{3,6}
W _{4,1}	W _{4,2}	W _{4,3}	W _{4,4}	W _{4,5}	W _{4,6}
W _{5,1}	W _{5,2}	W _{5,3}	W _{5,4}	W _{5,5}	W _{5,6}
W _{6,1}	W _{6,2}	W _{6,3}	W _{6,4}	W _{6,5}	W _{6,6}

3. Calcular FW para el resto de los bloques.

Resultados



- Se puede ver una mejora de tiempo con respecto a OpenMP sin bloques.
- Pero sigue siendo peor que los algoritmos vectorizados y CUDA.

Tabla 1: Número de vértices 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

• 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.
- Mejor localidad al dividir la matriz en submatrices.
- 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 $B = \frac{n}{b}$ es el número de bloques y b tamaño de cada bloque. Entonces:

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

Con un b bajo:

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

Demo

1. Ejecutar el programa con el grafo en un archivo.
2. Luego imprimirá los tiempos de ejecución de cada algoritmo.

```
1 ./a.exe 4.mtx -p
2 Normal: 0 [ms]
3 Vectorizado8: 0 [ms]
4 Vectorizado16: 0 [ms]
5 Paralelizado OpenMP: 0 [ms]
6 Vectorizado + Paralelizado OpenMP: 0 [ms]
7 Paralelizado en bloques OpenMP: 0 [ms]
```

3. Si se agrega “-p” al ejecutar el programa, imprimirá la matriz original y resuelta. Después lee infinitamente 2 vértices para imprimir el camino más corto y su costo.



```
dist:
  1  2  3  4
-----
1|  0 INF -2 INF
2|  4  0  3 INF
3| INF INF  0  2
4| INF -1 INF  0

dist:
  1  2  3  4
-----
1|  0 -1 -2  0
2|  4  0  2  4
3|  5  1  0  2
4|  3 -1  1  0

Vertices: 1 2
Camino: 1 -> 3 -> 4 -> 2
Costo:      -2 +  2 + -1 = -1

Vertices: 3 1
Camino: 3 -> 4 -> 2 -> 1
Costo:      2 + -1 +  4 = 5
```


Referencias

- "Floyd-Warshall Algorithm." Wikipedia, 2023, https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_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