

# 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.

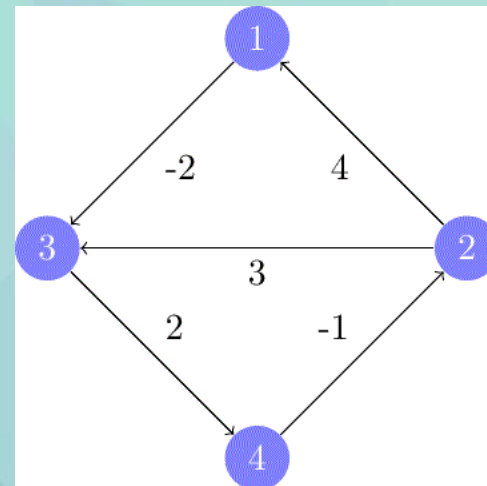
Utilizado en diversas aplicaciones, como en redes de comunicación, sistemas de navegación, optimización de rutas logísticas y análisis de redes sociales.

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**.



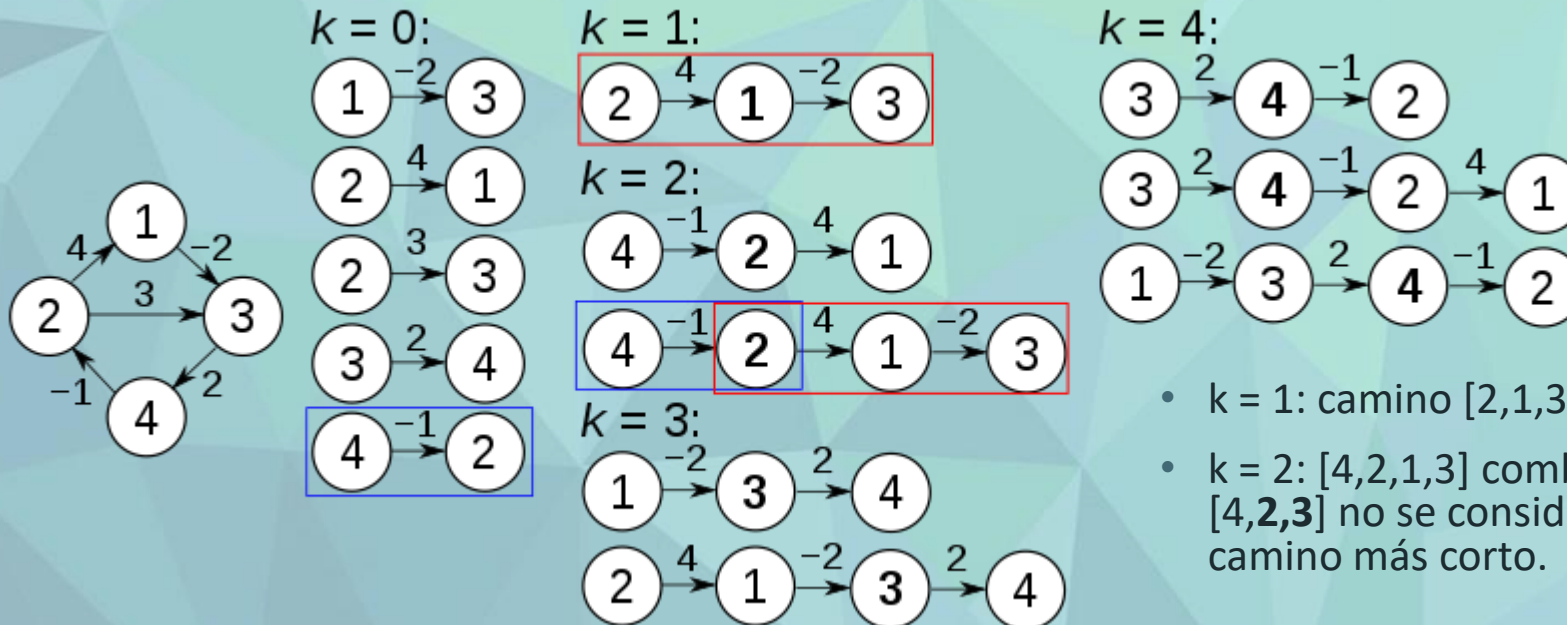
# Algoritmo

```

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.



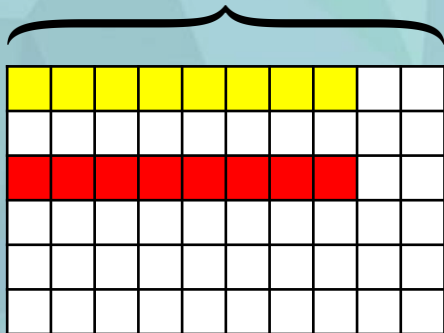
# Avance 1

- Floyd Warshall Vectorizado
- Floyd Warshall Paralelizado con OpenMP

# Floyd-Warshall Vectorizado

- Bucle **k** no es paralelizable. k depende del k-1 anterior.
- Operaciones **SIMD** de floats.
- Se cargan **8** elementos de una fila a un vector.
- Operaciones entre vectores SIMD.
- Si quedan menos de **8** elementos: secuencial.

8 elementos de una fila



```
1  for (int k = 0; k < V; k++)
2      {
3          for (int i = 0; i < V; i++)
4              {
5                  __m256 ik = _mm256_set1_ps(dist[i][k]);
6                  for (int j = 0; j < V - 7; j += 8)
7                      {
8                          __m256 kj = _mm256_loadu_ps(&dist[k][j]);
9                          __m256 ij = _mm256_loadu_ps(&dist[i][j]);
10                         __m256 ikj = _mm256_add_ps(ik, kj);
11                         __m256 result = _mm256_min_ps(ij, ikj);
12                         _mm256_storeu_ps(&dist[i][j], result);
13                     }
14                 for (int j = V - V % 8; j < V; j++)
15                     if (dist[i][j] > dist[i][k] + dist[k][j])
16                         dist[i][j] = dist[i][k] + dist[k][j];
17             }
18     }
```



# Floyd-Warshall Paralelizado con OpenMP

```
1 for (k = 0; k < V; k++)
2 {
3     #pragma omp parallel for private(i, j) schedule(static)
4     for (i = 0; i < V; i++)
5         for (j = 0; j < V; j++)
6             dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
7 }
```

- Paraleliza el **for i**
- **i** y **j** como variables privadas. Evita condiciones de carrera.
- Asignación estática (carga de trabajo más equilibrada).

Secuencial

```
1 for (int k = 0; k < V; k++)
2     for (int i = 0; i < V; i++)
3         for (int j = 0; j < V; j++)
4             // si dist de: i->k->j < i->j
5             dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
```

# Paralelizaciones adicionales

## Vectorizado con 16

Utiliza instrucciones `__mm512`

```
1 for (int i = 0; i < V; i++)
2     {
3         __m512 ik = _mm512_set1_ps(dist[i][k]);
4         for (int j = 0; j < V - 15; j += 16)
5             {
6                 __m512 kj = _mm512_loadu_ps(&dist[k][j]);
```

## Vectorizado + paralelización OpenMP

Se paraleliza el bucle i

```
1 for (int k = 0; k < V; k++)
2     {
3         #pragma omp parallel for
4         for (int i = 0; i < V; i++)
5             {
6                 __m256 ik = _mm256_set1_ps(dist[i][k]);
7                 for (int j = 0; j < V - 7; j += 8)
8                     {
```



# Avance 2

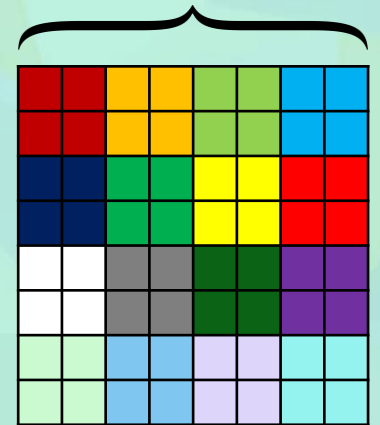
- Floyd-Warshall Paralelizado con CUDA



# Floyd-Warshall Paralelizado con CUDA

- Dividir la matriz en bloques (submatrices) cuadrados con tamaños iguales.
- Cada bloque se ejecuta en paralelo (si hay suficientes recursos).
- Para cada  $k$ :
  - Cada **bloque** ejecuta el kernel con el algoritmo.
  - Los **hilos** de los bloques se ejecutan en paralelo.

16 bloques de 2x2



$V = 8$

$B = 2$

$\text{Grid} = 8/2 = 4$

$4 \times 4$  (16) bloques

```
1 // Dividir la matriz en bloques de tamaño B
2 // Grid es el numero de bloques
3 int B = 32;
4 dim3 block(B, B);
5 dim3 grid((V + B - 1) / B, (V + B - 1) / B);
6
7 // Para cada k, se recorre todos los bloques del grid, y se ejecuta el kernel FW
8 for (int k = 0; k < V; k++)
9     floydKernel<<grid, block>>>(thrust::raw_pointer_cast(device.data()), V, k);
```

# Kernel

Secuencial

```
1  __global__ void floydKernel(float *dist, int V, int k)
2  {
3      int i = blockIdx.y * blockDim.y + threadIdx.y;
4      int j = blockIdx.x * blockDim.x + threadIdx.x;
5
6      __shared__ float ik[32], kj[32];
7
8      if (i < V && j < V)
9      {
10         if (!threadIdx.x)
11             ik[threadIdx.y] = dist[i * V + k];
12         if (!threadIdx.y)
13             kj[threadIdx.x] = dist[k * V + j];
14         __syncthreads();
15
16         float ikj = ik[threadIdx.y] + kj[threadIdx.x];
17         dist[i * V + j] = fminf(dist[i * V + j], ikj);
18     }
19 }
```

```
1  for (int k = 0; k < V; k++)
2      for (int i = 0; i < V; i++)
3          for (int j = 0; j < V; j++)
4              // si dist de: i->k->j < i->j
5              dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
```

- Se obtiene **i** y **j**
- Se crean 2 memorias compartidas.
- La memoria compartida **ik**, almacena la columna **dist[i][k]** del bloque.
- La memoria compartida **kj**, almacena la fila **dist[k][j]** del bloque.
- Se compara **dist[i][j]** con **ik + kj**.



# Avance 3

- Floyd-Warshall Paralelizado en bloques con OpenMP

# 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.

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

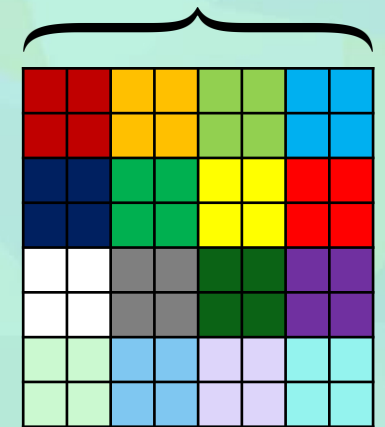


$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	$W_{1,4}$	$W_{1,5}$	$W_{1,6}$
$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	$W_{2,4}$	$W_{2,5}$	$W_{2,6}$
$W_{3,1}$	$W_{3,2}$	$W_{3,3}$	$W_{3,4}$	$W_{3,5}$	$W_{3,6}$
$W_{4,1}$	$W_{4,2}$	$W_{4,3}$	$W_{4,4}$	$W_{4,5}$	$W_{4,6}$
$W_{5,1}$	$W_{5,2}$	$W_{5,3}$	$W_{5,4}$	$W_{5,5}$	$W_{5,6}$
$W_{6,1}$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



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

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 <sub>1,1</sub>	W <sub>1,2</sub>	W <sub>1,3</sub>	W <sub>1,4</sub>	W <sub>1,5</sub>	W <sub>1,6</sub>
W <sub>2,1</sub>	W <sub>2,2</sub>	W <sub>2,3</sub>	W <sub>2,4</sub>	W <sub>2,5</sub>	W <sub>2,6</sub>
W <sub>3,1</sub>	W <sub>3,2</sub>	W <sub>3,3</sub>	W <sub>3,4</sub>	W <sub>3,5</sub>	W <sub>3,6</sub>
W <sub>4,1</sub>	W <sub>4,2</sub>	W <sub>4,3</sub>	W <sub>4,4</sub>	W <sub>4,5</sub>	W <sub>4,6</sub>
W <sub>5,1</sub>	W <sub>5,2</sub>	W <sub>5,3</sub>	W <sub>5,4</sub>	W <sub>5,5</sub>	W <sub>5,6</sub>
W <sub>6,1</sub>	W <sub>6,2</sub>	W <sub>6,3</sub>	W <sub>6,4</sub>	W <sub>6,5</sub>	W <sub>6,6</sub>

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



W <sub>1,1</sub>	W <sub>1,2</sub>	W <sub>1,3</sub>	W <sub>1,4</sub>	W <sub>1,5</sub>	W <sub>1,6</sub>
W <sub>2,1</sub>	W <sub>2,2</sub>	W <sub>2,3</sub>	W <sub>2,4</sub>	W <sub>2,5</sub>	W <sub>2,6</sub>
W <sub>3,1</sub>	W <sub>3,2</sub>	W <sub>3,3</sub>	W <sub>3,4</sub>	W <sub>3,5</sub>	W <sub>3,6</sub>
W <sub>4,1</sub>	W <sub>4,2</sub>	W <sub>4,3</sub>	W <sub>4,4</sub>	W <sub>4,5</sub>	W <sub>4,6</sub>
W <sub>5,1</sub>	W <sub>5,2</sub>	W <sub>5,3</sub>	W <sub>5,4</sub>	W <sub>5,5</sub>	W <sub>5,6</sub>
W <sub>6,1</sub>	W <sub>6,2</sub>	W <sub>6,3</sub>	W <sub>6,4</sub>	W <sub>6,5</sub>	W <sub>6,6</sub>

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

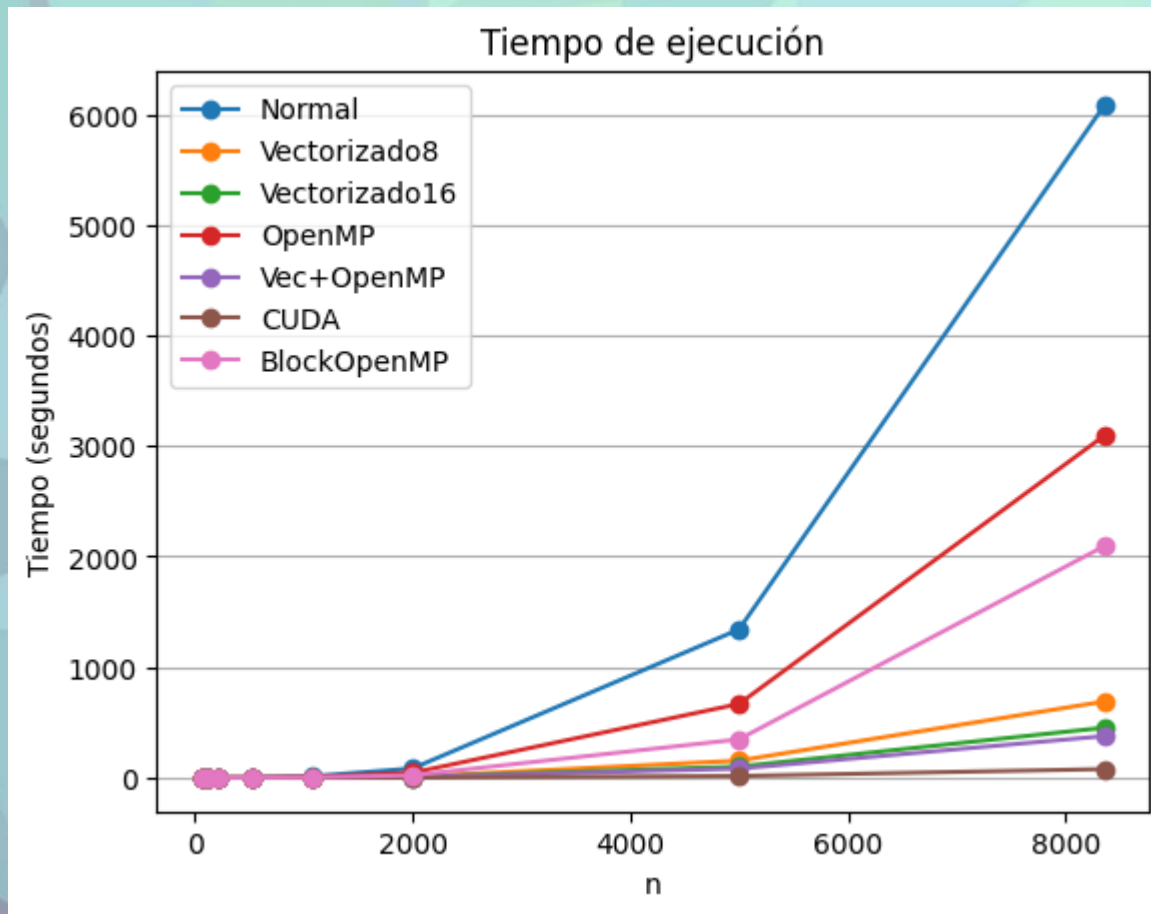


W <sub>1,1</sub>	W <sub>1,2</sub>	W <sub>1,3</sub>	W <sub>1,4</sub>	W <sub>1,5</sub>	W <sub>1,6</sub>
W <sub>2,1</sub>	W <sub>2,2</sub>	W <sub>2,3</sub>	W <sub>2,4</sub>	W <sub>2,5</sub>	W <sub>2,6</sub>
W <sub>3,1</sub>	W <sub>3,2</sub>	W <sub>3,3</sub>	W <sub>3,4</sub>	W <sub>3,5</sub>	W <sub>3,6</sub>
W <sub>4,1</sub>	W <sub>4,2</sub>	W <sub>4,3</sub>	W <sub>4,4</sub>	W <sub>4,5</sub>	W <sub>4,6</sub>
W <sub>5,1</sub>	W <sub>5,2</sub>	W <sub>5,3</sub>	W <sub>5,4</sub>	W <sub>5,5</sub>	W <sub>5,6</sub>
W <sub>6,1</sub>	W <sub>6,2</sub>	W <sub>6,3</sub>	W <sub>6,4</sub>	W <sub>6,5</sub>	W <sub>6,6</sub>

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
```

# Conclusiones

- En este proyecto, se paralelizó el algoritmo de Floyd-Warshall de distintas formas con vectorización, OpenMP y CUDA.
- Teóricamente, 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. CUDA fue el más rápido seguido de vectorización y finalmente paralelización con OpenMP.
- 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](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](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](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](https://www.researchgate.net/publication/315489843_CUDA_Analysis_of_Parallelization_in_Large_Graph_Algorithms)