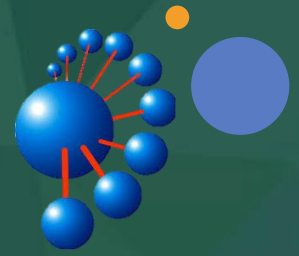



# Paralelizar Algoritmo Floyd-Warshall

**Nombre:** Dazhi Feng Z.

**Profesora:** Cecilia Hernández R.

*Introducción a la Computación Paralela (2023-1)*

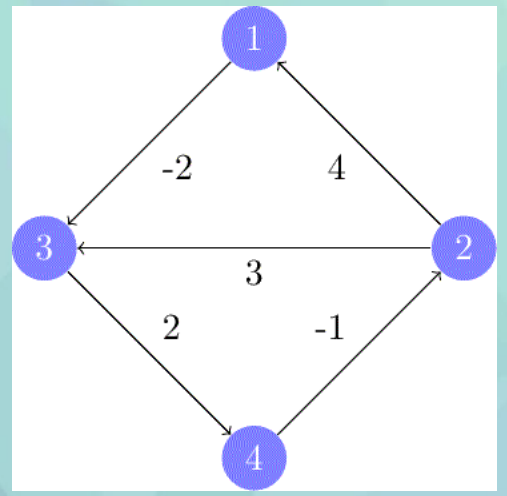
# Introducción

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.

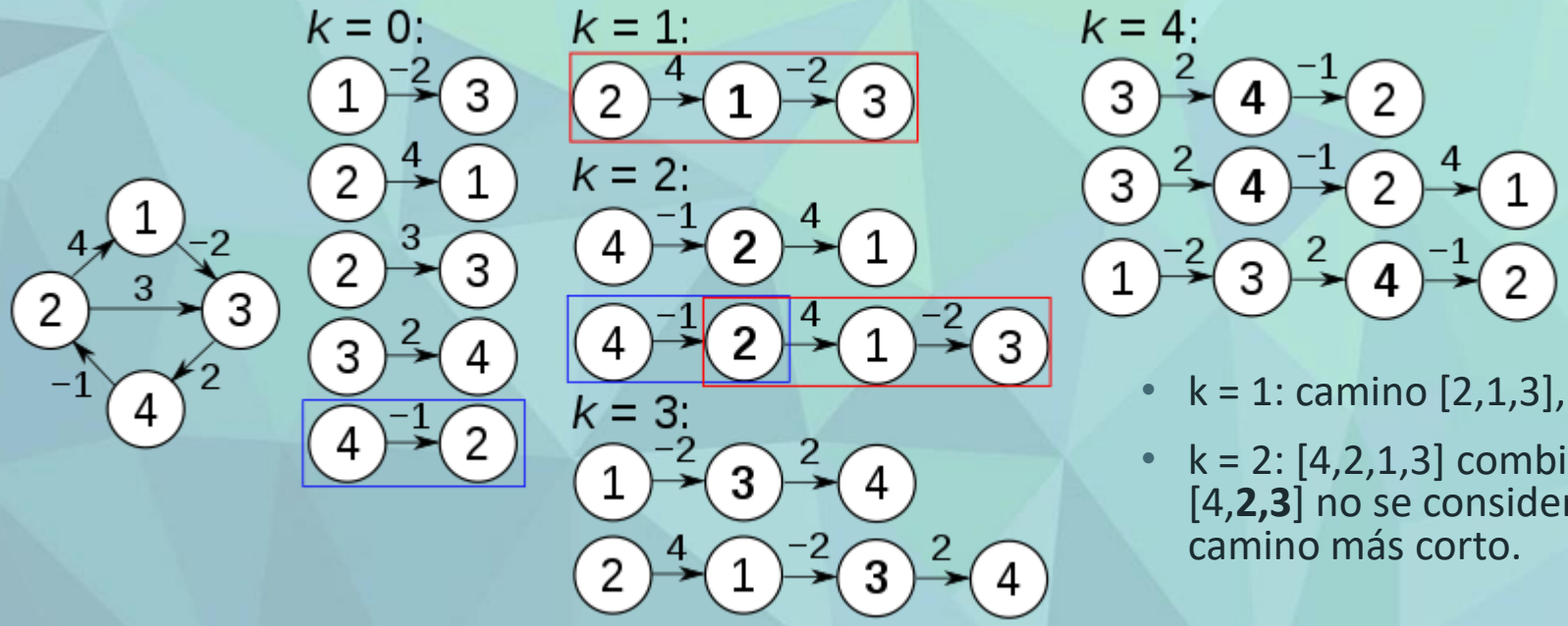
**Dijkstra:** Camino más corto desde **un** vértice a todos los demás vértices en un grafo.
**Bellman-Ford:** Camino más corto desde **un** vértice a todos los demás vértices en un grafo, **incluye aristas negativas.**
**Floyd-Warshall:** Caminos más cortos entre **todos** los pares de vértices en un grafo, **incluye aristas negativas.**



# Algoritmo

```
let dist be a |V| × |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
```

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.

# 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.

# Código algoritmo vectorizado

Secuencial

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

```
for (int k = 0; k < V; k++)
    {
        for (int i = 0; i < V; i++)
        {
            __m256 ik = _mm256_set1_ps(dist[i][k]);
            for (int j = 0; j < V - 7; j += 8)
            {
                __m256 kj = _mm256_loadu_ps(&dist[k][j]);
                __m256 ij = _mm256_loadu_ps(&dist[i][j]);
                __m256 ikj = _mm256_add_ps(ik, kj);
                __m256 result = _mm256_min_ps(ij, ikj);
                _mm256_storeu_ps(&dist[i][j], result);
            }
            for (int j = V - V % 8; j < V; j++)
                if (dist[i][j] > dist[i][k] + dist[k][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
```

- Cargar **8** elementos de la fila **k** a un vector SIMD, **kj**
- Cargar **8** elementos de la fila **i** (mismas columnas j), **ij**
- Sumar los 2 vectores de **8** elementos **ik + kj = ikj**
- Comparar la suma (**ikj**) con vector **ij**
- Guardar los resultados.

# Floyd-Warshall Paralelizado con OpenMP

```
for (k = 0; k < V; k++)
    {
        #pragma omp parallel for private(i, j) schedule(static)
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
    }
```

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

Secuencial

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

# Paralelizaciones adicionales

**Vectorizado con 16**

Utiliza instrucciones **__mm512**

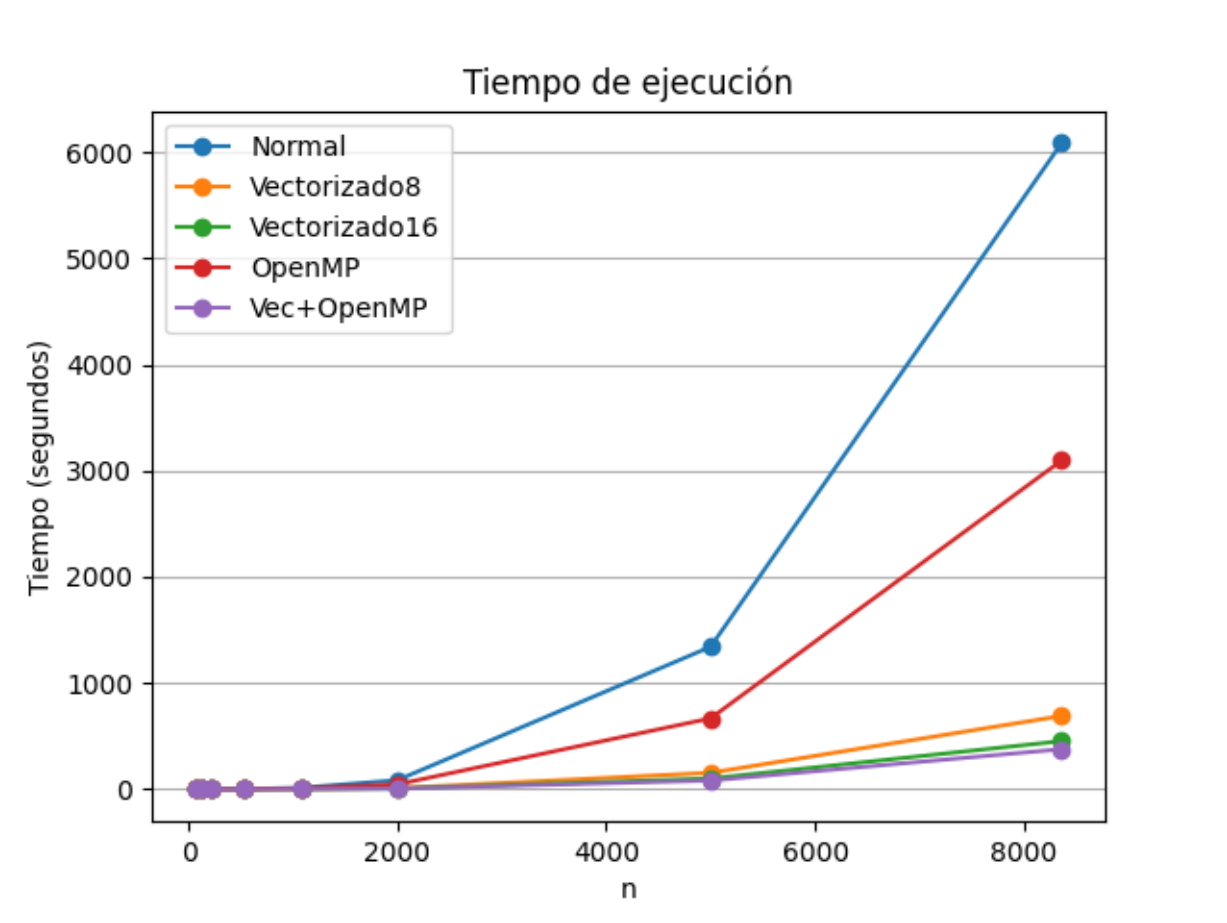
```
for (int i = 0; i < V; i++)
        {
            __m512 ik = _mm512_set1_ps(dist[i][k]);
            for (int j = 0; j < V - 15; j += 16)
            {
                __m512 kj = _mm512_loadu_ps(&dist[k][j]);
```

**Vectorizado + paralelización OpenMP**

Se paraleliza el bucle **i**

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

# Resultados




- **Vectorización** tiene mejor rendimiento.
- **Vectorización de 8** tarda aprox. **11,5%** del tiempo normal.
- Paralelización con **OpenMP** tarda con valores cercanos al **50%** del tiempo normal.
- **Vectorización + paralelización** tiene el mejor rendimiento y tarda un poco más de la mitad del tiempo de la **vectorización de 8**.

| n | Normal | Vectorizado8 | Vectorizado16 | OpenMP | Vec+OpenMP |
|---|---|---|---|---|---|
| 64 | 4 | 0 | 0 | 3 | 0 |
| 128 | 22 | 2 | 1 | 18 | 1 |
| 214 | 112 | 12 | 8 | 70 | 7 |
| 531 | 1.577 | 183 | 111 | 860 | 108 |
| 1.080 | 13.463 | 1.482 | 920 | 7.179 | 766 |
| 2.000 | 83.954 | 10.135 | 6.239 | 44.833 | 5.210 |
| 5.000 | 1.341.107 | 154.406 | 100.318 | 667.092 | 81.281 |
| 8.361 | 6.085.038 | 689.809 | 452.683 | 3.096.147 | 376.512 |

# 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.
- La complejidad del algoritmo **vectorizado** es $O(n^3)$, ya que en último ciclo *for* sigue siendo lineal.
  - $O\left(\frac{n}{8}\right) = O(n) \Rightarrow O(n) \times O(n^2) = \boldsymbol{O(n^3)}$
- La **paralelización con OpenMP** mejora a medida que se aumenten los procesadores utilizados, pero la complejidad va a seguir siendo $O(n^3)$, a menos que p = n.
  - $O\left(\frac{n}{p}\right) = O(n) \Rightarrow O(n) \times O(n^2) = \boldsymbol{O(n^3)}$
- Teorema de Brent:
  - $\mathrm{Tp} \leq \mathrm{W/p} + \mathrm{D}$
  - $\mathrm{Tp} \leq O(n^3)/\mathrm{p} + O(n^2)$ , si $\mathbf{p = n}$
  - $\mathrm{Tp} \leq O(n^2) + O(n^2) \Leftrightarrow \mathbf{Tp} \leq \boldsymbol{2O(n^2)}$ (en teoría)

# Demo

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

```
./a.exe 4.mtx -p
Normal: 0 [ms]
Vectorizado8: 0 [ms]
Vectorizado16: 0 [ms]
Paralelizado OpenMP: 2 [ms]
Vectorizado + Paralelizado 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
```

# Planificación restante

- **Avance 2:**
  - Implementar Floyd-Warshall paralelo en bloques con OpenMP.
  - Implementar Floyd-Warshall paralelo con CUDA.
- **Entrega final:**
  - Terminar/optimizar algoritmos paralelos.
  - Comparar resultados, realizar nuevamente un análisis y escribir informe.

# 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
- Sao, Piyush, Lu, Hao, Kannan, Ramakrishnan, Thakkar, Vijay, Vuduc, Richard, and Potok, Thomas. Scalable All-pairs Shortest Paths for Huge Graphs on Multi-GPU Clusters. United States: N. p., 2020. Web. https://www.osti.gov/servlets/purl/1814306
- Weiss, Stewart. CSci 493.65 Parallel Computing: Chapter 5 Floyd's Algorithm. s. f. http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci493.65/lecture_notes/chapter05.pdf
- 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
- Gautam, Asmita. "Parallel Implementation of Floyd-Warshall Algorithm." Presentación. CSE 633: Parallel Algorithms. Guía: Dr. Russ Miller (UB Distinguished Professor). https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Asmita-Gautam-Spring-2019.pdf
- Students of the Parallel Processing Systems course. "Parallelizing the Floyd-Warshall Algorithm on Modern Multicore Platforms: Lessons Learned." Informe. School of Electrical & Computer Engineering, National Technical University of Athens, 2011. http://www.cslab.ntua.gr/courses/pps/files/fall2011/paper_sfhmmy.pdf
- Hou, Kaixi, et al. "Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study." Department of Computer Science, Virginia Tech, Blacksburg, Virginia, U.S.A. s. f. https://kaixih.github.io/assets/papers/floyd-camera-ready.pdf
- Nagavalli, Sasanka. "Using Hardware Parallelism for Faster Search via Dynamic Programming." Robotics Institute, Carnegie Mellon University, May 17, 2013. https://www.andrew.cmu.edu/user/snagaval/16-745/Project/16-745-Project-Report-SasankaNagavalli.pdf