UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO
FACULTAD DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA, INFORMÁTICA Y MECÁNICA
ESCUELA PROFESIONAL DE INGENIERÍA INFORMÁTICA Y DE SISTEMAS

Proyecto N° 1

Algoritmos de Ordenamiento Paralelos y Secuenciales

Asignatura:
Algoritmos Paralelos y Distribuidos

Docente:
Mgt. Dueñas Jimenez, Ray

Estudiantes:
Castro Pari, Rayneld Fidel
Mayhuire Chacon, Brenda Lucia
Mendoza Quispe, Jose Daniel
Perez Cahuana, Gabriel
Zevallos Yanqui, Andy Jefferson

Cusco - Perú
2025

Índice

Índice de figuras

1. Resumen

La **cerradura transitiva booleana** (también conocida como **Warshall lógico**) es un algoritmo clásico de álgebra booleana y teoría de grafos que permite determinar la **alcanzabilidad** entre todos los pares de vértices de un grafo dirigido. A partir de una matriz de adyacencia $A \in \{0,1\}^{N\times N}$, el algoritmo produce una matriz T donde $T_{ij} = 1$ si existe un camino (de longitud ≥ 0 o ≥ 1 según convención) desde i hasta j.

Este trabajo propone Warshall lógico como candidato para la **paralelización con CUDA** debido a su estructura matricial, su costo computacional $O(N^3)$ y su verificación directa comparando contra la versión secuencial. Se describe el algoritmo, su representación matricial, su complejidad y un diseño de implementación secuencial en C/C++ con medición de tiempo para diferentes tamaños de N (incluyendo $N \geq 1024$) y verificación para matrices pequeñas.

2. Introducción

En problemas reales de computación científica e ingeniería, es frecuente trabajar con **matrices masivas**: modelos de conectividad, relaciones de dependencia, grafos de estados, análisis de flujos, rutas, etc. Cuando el tamaño N crece (por ejemplo $N \geq 1024$), el costo de los algoritmos cúbicos puede volverse prohibitivo en CPU, lo cual motiva el uso de paralelización en GPU (CUDA).

El algoritmo de Warshall lógico pertenece a la familia de algoritmos **All-Pairs Reachability** y es una variante booleana del cálculo de caminos. Su núcleo consiste en actualizar A_{ij} usando operaciones booleanas con un índice intermedio k, manteniendo una dependencia por iteración de k que es relevante para el diseño paralelo.

3. Objetivo General

Diseñar, implementar y evaluar la aceleración obtenida al paralelizar el algoritmo de **cerradura transitiva booleana** (Warshall lógico) para matrices cuadradas masivas ($N \geq 1024$), comparando:

- Versión secuencial (CPU).
- Versión paralela (CUDA) en Fase 2.
- (Opcional en Fase 2) Versión CPU multi-core (OpenMP) para comparación.

4. Warshall Lógico: Versión Secuencial

El algoritmo de Warshall lógico calcula la **cerradura transitiva** de un grafo dirigido usando su matriz de adyacencia. Sea A la matriz de entrada, donde:

$$A_{ij} = \begin{cases} 1, & \text{si existe un arco directo } i \to j \\ 0, & \text{caso contrario} \end{cases}$$

La idea es permitir, progresivamente, que los caminos utilicen vértices intermedios del conjunto $\{0, 1, \ldots, k\}$.

4.1. Requisitos del Algoritmo

(a) Operación con matrices masivas: El algoritmo trabaja con una matriz cuadrada booleana $N \times N$. Para el proyecto se requiere $N \geq 1024$.

(b) Complejidad mínima: Warshall lógico presenta complejidad temporal:

$$T(N) = O(N^3),$$

lo cual es un excelente candidato para paralelización porque el costo crece rápidamente con N.

(c) Verificabilidad: La versión paralela puede validarse comparando su salida con la versión secuencial (CPU) usando:

- Comparación elemento a elemento para matrices pequeñas (por ejemplo $N \leq 128$).
- Pruebas aleatorias (random testing) con semillas fijas.
- Casos controlados (grafos con estructura conocida: cadena, ciclo, completo, vacío).

5. Descripción del Algoritmo

5.1. Campo de aplicación

La cerradura transitiva se usa en:

- **Teoría de grafos**: alcanzabilidad, componentes, análisis de conectividad.
- **Bases de datos**: consultas de tipo *reachability* sobre relaciones (p.ej., jerarquías, dependencias).
- **Compiladores y análisis de programas**: grafos de flujo, dependencias entre módulos.
- **Redes y sistemas**: análisis de rutas y acceso entre nodos.

5.2. Descripción detallada del funcionamiento

Warshall lógico aplica la recurrencia:

$$A_{ij} \leftarrow A_{ij} \ \vee \ (A_{ik} \wedge A_{kj}) \quad \text{para } k = 0, \ldots, N-1.$$

Intuición:

- $A_{ik} = 1$ indica que i puede llegar a k.
- $A_{kj} = 1$ indica que k puede llegar a j.
- Si ambos son verdaderos, entonces i puede llegar a j usando a k como intermedio.

Pseudocódigo (Warshall lógico)

Algorithm 1 Cerradura transitiva booleana (Warshall lógico)

Matriz booleana $A \in \{0,1\}^{N \times N}$ Matriz booleana A actualizada con su cerradura transitiva

for $k \leftarrow 0$ $N-1$ **do**

  for $i \leftarrow 0$ $N-1$ **do**

    for $j \leftarrow 0$ $N-1$ **do** $A[i][j] \leftarrow A[i][j] \ \vee \ (A[i][k] \wedge A[k][j])$

Ejemplo pequeño

Considere $N = 4$ y la matriz de adyacencia:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Esto representa la cadena $0 \to 1 \to 2 \to 3$. La cerradura transitiva resultante debe permitir: 0 llega a 2 y 3, y 1 llega a 3. Por tanto:

$$T = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

6. Representación Matricial

6.1. Representación de los datos de entrada

La entrada se modela como una matriz booleana A:

$$A \in \{0,1\}^{N \times N}.$$

Cada fila i representa los destinos alcanzables desde i en un paso (aristas directas).

Memoria: almacenar A como `uint8_t` (1 byte) requiere N^2 bytes. Para $N = 1024$, $N^2 =$ 1,048,576 entradas $\approx$ 1 MB.

6.2. Representación de la operación núcleo

La actualización elemento a elemento es:

$$A_{ij}^{(k)} = A_{ij}^{(k-1)} \vee \left(A_{ik}^{(k-1)} \wedge A_{kj}^{(k-1)} \right).$$

Donde el superíndice (k) indica que se permiten vértices intermedios hasta k.

6.3. Representación de los datos de salida

La salida es la misma matriz A ya actualizada (in-place) o una matriz T separada. En ambos casos:

$$T_{ij} = 1 \iff \exists \text{ un camino de } i \text{ a } j.$$

7. Complejidad Computacional

7.1. Parámetros relevantes

- N: dimensión de la matriz cuadrada ($N \times N$).
- Operación booleana básica: $\vee$ y $\wedge$ (tiempo constante en CPU).
- Accesos a memoria: lecturas de $A[i][j]$, $A[i][k]$, $A[k][j]$ y escritura de $A[i][j]$.

7.1.1. Complejidad temporal

El algoritmo ejecuta tres bucles anidados de tamaño N, por lo tanto:

$$T(N) = \sum_{k=1}^{N} \sum_{i=1}^{N} \sum_{j=1}^{N} O(1) = O(N^3).$$

7.1.2. Complejidad espacial

La memoria principal está dominada por la matriz:

$$S(N) = O(N^2).$$

7.1.3. Implicancias para el rendimiento

Para N grande, el tiempo está fuertemente influenciado por:

- **Localidad de memoria**: el acceso repetido a filas/columnas puede causar fallos de caché.
- **Ancho de banda**: $O(N^3)$ operaciones implican múltiples lecturas/escrituras.
- **Dependencias**: existe una dependencia fuerte por iteración de k, lo cual define el diseño paralelo.

8. Análisis del Algoritmo

8.1. Funcionamiento del Algoritmo

La iteración externa (k) se interpreta como: "permitir que k sea vértice intermedio". Para cada k, se actualizan todos los pares (i, j) en la matriz.

Dependencia de datos clave:

- Para un k fijo, cada celda (i, j) se puede calcular de manera independiente **si se leen** $A[i][k]$ y $A[k][j]$ consistentes con la iteración k.
- Sin embargo, pasar de k a $k + 1$ requiere que todas las actualizaciones de k estén completadas.

Esto sugiere un patrón típico para CUDA:

- Mantener el bucle k en CPU o en GPU de forma secuencial.
- Para cada k, paralelizar la actualización de $A[i][j]$ sobre una grilla 2D de hilos.

8.2. Justificación del Algoritmo como candidato CUDA

Warshall lógico cumple los requisitos del proyecto:

- **Matrices masivas:** A es $N \times N$ y puede evaluarse con $N \geq 1024$.
- **Alta complejidad:** $O(N^3)$ (categoría "Alto").
- **Paralelizable:** para un k fijo, las celdas (i, j) son paralelizables.
- **Verificación directa:** comparación bit a bit contra la versión secuencial.

Nota de optimización para Fase 2 (opcional): En GPU suele ser ventajoso aplicar una versión **bloqueada (tiled)** para aprovechar memoria compartida y mejorar coalescencia, similar a Floyd-Warshall bloqueado.

8.3. Ejecución y Medición del Tiempo (CPU)

La medición debe contabilizar únicamente el **núcleo del algoritmo** (los bucles k, i, j), excluyendo:

- Generación/lectura de la matriz.
- Impresión de matrices.
- Verificación (cuando se mida rendimiento).

Se recomienda usar `clock_gettime(CLOCK_MONOTONIC, ...)` para medir en segundos.

8.4. Medición del rendimiento (diferentes tamaños)

La ejecución debe repetirse para varios tamaños:

$$N \in \{128, 256, 512, 1024, 2048, \dots\}$$

y registrar el tiempo $t(N)$.

Cuadro 1: Plantilla de resultados (CPU secuencial)

N	Entradas (N^2)	Tiempo (s)	Observaciones
128	16384	–	Verificación habilitada
256	65536	–	–
512	262144	–	–
1024	1048576	–	Caso masivo requerido

9. Conclusiones

- Warshall lógico permite calcular la cerradura transitiva booleana de una matriz de adyacencia y resolver alcanzabilidad all-pairs en grafos dirigidos.
- Su complejidad $O(N^3)$ lo vuelve un candidato fuerte para acelerar con CUDA cuando $N \geq 1024$.
- La dependencia por iteración k impone un esquema paralelo por "fases" (un kernel por k o un enfoque bloqueado), pero dentro de cada k el paralelismo sobre (i, j) es masivo.
- La validación es clara: la salida paralela debe coincidir con la salida secuencial.

10. ANEXOS

A. Implementación secuencial en C (con medición de tiempo y verificación)

Estructura propuesta:

- `alloc_matrix`: reserva memoria contigua N^2.
- `init_random`: genera matriz booleana (densidad controlable).
- `warshall_logical`: núcleo $O(N^3)$.
- `verify_small`: comparación con versión alternativa o pruebas controladas.
- `timing`: medición del tiempo sólo del núcleo.

Listing 1: warshall$_l$ogico$_s$ecuencial.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>

static inline double seconds_now() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec + (double)ts.tv_nsec * 1e-9;
}

uint8_t* alloc_matrix(int N) {
    uint8_t* A = (uint8_t*)malloc((size_t)N * (size_t)N * sizeof(uint8_t));
    if (!A) {
        fprintf(stderr, "Error: no se pudo asignar memoria para N=%d\n", N);
        exit(EXIT_FAILURE);
    }
    return A;
}

void init_random(uint8_t* A, int N, double p, unsigned seed) {
    // p = probabilidad de 1 (densidad). Ej: p=0.05 -> matriz dispersa
    srand(seed);
    for (int i = 0; i < N*N; i++) {
        double r = (double)rand() / (double)RAND_MAX;
        A[i] = (r < p) ? 1 : 0;
    }
```

```
}

void init_chain_example(uint8_t* A, int N) {
    // Ejemplo controlado: cadena 0->1->2->...->N-1
    for (int i = 0; i < N*N; i++) A[i] = 0;
    for (int i = 0; i < N-1; i++) A[i*N + (i+1)] = 1;
}

void print_matrix(const uint8_t* A, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", (int)A[i*N + j]);
        }
        printf("\n");
    }
}

void warshall_logical(uint8_t* A, int N) {
    // A[i][j] = A[i][j] OR (A[i][k] AND A[k][j])
    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            uint8_t aik = A[i*N + k];
            if (!aik) continue; // peque o ahorro: si A[i][k]==0, no aporta
            for (int j = 0; j < N; j++) {
                A[i*N + j] = (uint8_t)(A[i*N + j] | (aik & A[k*N + j]));
            }
        }
    }
}

int matrices_equal(const uint8_t* A, const uint8_t* B, int N) {
    for (int i = 0; i < N*N; i++) {
        if (A[i] != B[i]) return 0;
    }
    return 1;
}

int main(int argc, char** argv) {
```

```
66      int N = 256;
67      double p = 0.05;
68      unsigned seed = 1234;
69      int verify = 0;
70
71      if (argc >= 2) N = atoi(argv[1]);        // ./a.out N
72      if (argc >= 3) p = atof(argv[2]);        // ./a.out N p
73      if (argc >= 4) seed = (unsigned)atoi(argv[3]); // ./a.out N p seed
74      if (argc >= 5) verify = atoi(argv[4]);   // ./a.out N p seed verify
75
76      uint8_t* A = alloc_matrix(N);
77      init_random(A, N, p, seed);
78
79      // Verificaci n  s lo para N peque o  (opcional)
80      // Aqu  se muestra un ejemplo de verificaci n con un caso controlado "cadena"
81      if (verify && N <= 64) {
82          uint8_t* Atest = alloc_matrix(N);
83          uint8_t* Aref  = alloc_matrix(N);
84          init_chain_example(Atest, N);
85          for (int i = 0; i < N*N; i++) Aref[i] = Atest[i];
86
87          warshall_logical(Atest, N);
88
89          // Propiedad esperada para cadena: si i<j entonces alcanzable (1) para j>i
90          for (int i = 0; i < N; i++) {
91              for (int j = i+1; j < N; j++) {
92                  Aref[i*N + j] = 1;
93              }
94          }
95
96          if (!matrices_equal(Atest, Aref, N)) {
97              fprintf(stderr, "Verificacion FALLIDA para N=%d\n", N);
98              fprintf(stderr, "Salida obtenida:\n"); print_matrix(Atest, N);
99              fprintf(stderr, "Salida esperada:\n"); print_matrix(Aref, N);
100             return EXIT_FAILURE;
101         } else {
102             printf("Verificacion OK para N=%d (caso cadena)\n", N);
103         }
```

```
        free(Atest); free(Aref);
    }

    // Medici n del n cleo
    double t0 = seconds_now();
    warshall_logical(A, N);
    double t1 = seconds_now();

    printf("N=%d, p=%.3f, tiempo_nucleo=%.6f s\n", N, p, (t1 - t0));

    free(A);
    return 0;
}
```

B. Comandos de compilación y ejecución (Linux/Colab)

```
# Compilar (GCC)
gcc -O3 -std=c11 warshall_logico_secuencial.c -o warshall

# Ejecutar: ./warshall N p seed verify
# N: tamaño, p: densidad, seed: semilla, verify: 1/0
./warshall 256 0.05 1234 1
./warshall 1024 0.02 1234 0
```

C. Cómo se evaluará en Fase 2 (referencia del proyecto)

Para la Fase 2 (entrega: 7 de enero 2026), se implementará el núcleo en CUDA y se comparará contra CPU (y opcionalmente OpenMP) con métricas:

$$\text{Speedup } S = \frac{T_{\text{secuencial}}}{T_{\text{paralelo}}} \qquad \text{Efficiency (OpenMP) } E = \frac{S}{\#\text{núcleos}}.$$

Además, se analizará en CUDA:

- Configuración de *blocks* e hilos por bloque.
- Coalescencia de memoria global.
- Uso de memoria compartida (tiling).