

# Taller 5: Optimización de Matriz de Costos con Backtracking Paralelo

Cristian Bravo  
Bruno Bastidas  
Silvio Villagra  
Daniella Lecanda

**Grupo investigativo Los Catalizadores**

25/04/2025

## Introducción

En el presente taller, nos enfrentamos al desafío de encontrar el camino de costo mínimo entre dos nodos en un grafo representado mediante una matriz de adyacencia. Para abordar este problema, implementamos un algoritmo de **backtracking** con paralelización selectiva utilizando OpenMP. El backtracking es una técnica de búsqueda exhaustiva que explora todas las posibles soluciones mediante recursión, retrocediendo cuando una solución parcial no puede extenderse a una solución completa válida.

Nuestra implementación se desarrolló en el lenguaje de programación C++, conocido por su eficiencia y control sobre los recursos del sistema. C++ nos proporciona las herramientas necesarias para manejar estructuras de datos complejas y realizar operaciones de bajo nivel, lo cual es esencial para optimizar el rendimiento de algoritmos intensivos en cómputo como el backtracking. Además, la compatibilidad de C++ con bibliotecas de paralelización como **OpenMP** facilita la implementación de soluciones concurrentes que aprovechan los recursos de hardware disponibles.

Para mejorar la eficiencia y escalabilidad de nuestra solución, aplicamos las siguientes estrategias:

- **Paralelización selectiva:** Implementamos OpenMP en los primeros niveles del árbol de recursión (profundidad  $< 2$ ) para equilibrar el overhead de crear hilos frente al beneficio del paralelismo. Esta decisión se basa en que los niveles más profundos tienen menos trabajo por hilo, y la creación de hilos adicionales podría no ser eficiente.
- **Estructura de datos mejorada:** Diseñamos la estructura **Camino**, que almacena tanto el costo total como los nodos visitados. A diferencia del código base que solo rastreaba el costo mínimo, esta estructura permite visualizar la ruta óptima encontrada, facilitando el análisis y la comprensión de los resultados.
- **Generación realista de matrices:** Incorporamos una probabilidad del 5% de tener aristas infinitas (inaccesibles) para simular redes de transporte reales donde no todos los nodos están conectados directamente. Esta característica añade realismo al problema y permite evaluar la robustez del algoritmo frente a grafos con conectividad parcial.
- **Control de visitados:** Optimizamos el manejo del vector **visited** utilizando la cláusula **firstprivate** en OpenMP para evitar condiciones

de carrera, creando copias independientes para cada hilo. Esta medida asegura la correcta ejecución concurrente del algoritmo sin interferencias entre hilos.

El objetivo final de este estudio es determinar empíricamente el **punto de inflexión** donde la paralelización justifica su overhead, comparando los tiempos de ejecución secuencial y paralela para matrices desde 2x2 hasta 14x14. Este análisis nos permite validar en qué casos el backtracking paralelo es viable y eficiente para problemas de optimización de rutas, y proporciona una comprensión más profunda de las ventajas y limitaciones de la paralelización en algoritmos de búsqueda exhaustiva.

## Código Base del Taller

Según lo solicitado en el documento del taller, se presentan primero los códigos base proporcionados:

```
1  #include <iostream>
2  #include <vector>
3  #include <random>
4
5  using namespace std;
6
7  int main() {
8      int n = 5;
9      vector<vector<int>> matrix(n, vector<int>(n));
10
11      random_device rd;
12      mt19937 gen(rd());
13      uniform_int_distribution<> dis(1, 10);
14
15      for(int i = 0; i < n; i++) {
16          for(int j = 0; j < n; j++) {
17              matrix[i][j] = dis(gen);
18              cout << matrix[i][j] << " ";
19          }
20          cout << endl;
21      }
22      return 0;
```

```
23 }
```

Listing 1: Generación de matriz aleatoria (código base del taller)

```
1  #include <iostream>
2  #include <vector>
3  #include <random>
4  #include <climits>
5
6  using namespace std;
7
8  void backtracking(vector<vector<int>>& matrix, int
   current, int end,
9      int dist, int& minDist, vector<bool>
      & visited) {
10     if(current == end) {
11         minDist = min(minDist, dist);
12         return;
13     }
14
15     for(int i = 0; i < matrix.size(); i++) {
16         if(matrix[current][i] != 0 && !visited[i]) {
17             visited[i] = true;
18             backtracking(matrix, i, end, dist +
               matrix[current][i],
19                 minDist, visited);
20             visited[i] = false;
21         }
22     }
23 }
24
25 int main() {
26     int n = 5;
27     vector<vector<int>> matrix(n, vector<int>(n));
28
29     // ... (generación de matriz igual al código
30     anterior)
31
31     int start = 0;
32     int end = 4;
```

```

33     int dist = 0;
34     int minDist = INT_MAX;
35     vector<bool> visited(n, false);
36     visited[start] = true;
37
38     backtracking(matrix, start, end, dist, minDist,
39                 visited);
40
41     cout << "La distancia m nima desde " << start
42           << " hasta " << end << " es: " << minDist
43           << endl;
44
45     return 0;
46 }

```

Listing 2: Backtracking secuencial (código base del taller)

## Implementación Mejorada con Paralelismo

Basados en los códigos proporcionados, desarrollamos una versión mejorada que incluye:

```

1  #include <iostream>
2  #include <vector>
3  #include <random>
4  #include <chrono>
5  #include <mutex>
6  #include <climits>
7  #include <algorithm>
8  #include <iomanip>
9  #include <omp.h>
10
11 using namespace std;
12 using namespace std::chrono;
13
14 const int INF = 9999;
15 mutex mtx;
16
17 struct Camino {

```

```

18     vector<int> nodos;
19     int costo;
20 };
21
22 vector<vector<int>> generarMatriz(int n) {
23     vector<vector<int>> matrix(n, vector<int>(n));
24     random_device rd;
25     mt19937 gen(rd());
26     uniform_int_distribution<> dis(1, 10);
27     uniform_real_distribution<> prob(0.0, 1.0);
28
29     for(int i = 0; i < n; i++) {
30         for(int j = 0; j < n; j++) {
31             if(i == j) {
32                 matrix[i][j] = 0;
33             } else {
34                 if(prob(gen) < 0.05) {
35                     matrix[i][j] = INF;
36                 } else {
37                     matrix[i][j] = dis(gen);
38                 }
39             }
40         }
41     }
42     return matrix;
43 }
44
45 void backtracking_paralelo_omp(vector<vector<int>>&
46     matrix, int current, int end,
47     int dist, vector<int>&
48     camino_actual, Camino
49     & mejor_camino,
50     vector<bool>& visited,
51     int depth = 0) {
52     camino_actual.push_back(current);
53
54     if(current == end) {
55         #pragma omp critical
56         {
57             if(dist < mejor_camino.costo) {

```

```

54         mejor_camino.costo = dist;
55         mejor_camino.nodos = camino_actual;
56     }
57 }
58 camino_actual.pop_back();
59 return;
60 }
61
62 if(depth < 2) {
63     #pragma omp parallel for shared(mejor_camino
64     ) firstprivate(visited, dist,
65     camino_actual) schedule(dynamic)
66     for(int i = 0; i < matrix.size(); i++) {
67         if(matrix[current][i] != 0 && matrix[
68         current][i] != INF && !visited[i]) {
69             vector<bool> new_visited = visited;
70             new_visited[i] = true;
71             vector<int> new_camino =
72                 camino_actual;
73             backtracking_paralelo_omp(matrix, i,
74                 end, dist + matrix[current][i],
75                 new_camino,
76                 mejor_camino
77                 ,
78                 new_visited
79                 , depth +
80                 1);
81         }
82     }
83 } else {
84     for(int i = 0; i < matrix.size(); i++) {
85         if(matrix[current][i] != 0 && matrix[
86         current][i] != INF && !visited[i]) {
87             visited[i] = true;
88             backtracking_paralelo_omp(matrix, i,
89                 end, dist + matrix[current][i],
90                 camino_actual
91                 ,
92                 mejor_camino
93                 , visited,

```

```

79                                     depth +
80                                     1);
81                                     visited[i] = false;
82                                 }
83                             }
84     camino_actual.pop_back();
}

```

Listing 3: Implementación mejorada con paralelismo OpenMP

- Paralelización con OpenMP
- Creación de estructura de datos para almacenar camino
- Generación mejorada de matrices
- Control de profundidad para los hilos de la maquina debido a la existencia de los costos
- Medición de tiempos
- Visualización de resultados
- Manejo de memoria

## Compilación y Ejecución

```

1  g++ -std=c++11 -fopenmp -o Taller5 Taller5.cpp
2  ./Taller5

```

Listing 4: Comandos para compilar y ejecutar sugerido

```

1  g++ -o taller_5_con_caminos -fopenmp
   taller_5_con_caminos.cpp
2  ./taller_5_con_caminos

```

Listing 5: Comandos para compilar y ejecutar utilizado



## Resultados y Análisis

Cuadro 1: Tiempos de ejecución ( $\mu$ s) para diferentes tamaños de matriz

Tamaño	Secuencial	Paralelo	Speedup
2x2	2	5624	0.000356x
3x3	1	19281	0.000052x
4x4	6	7880	0.000761x
5x5	10	11877	0.000842x
6x6	37	11770	0.003144x
7x7	183	20422	0.008961x
8x8	4167	14612	0.285177x
9x9	17910	12091	1.481270x
10x10	76963	61094	1.259750x
11x11	684560	643304	1.064130x
12x12	9720973	10040253	0.968200x
13x13	90302272	90018521	1.003150x
14x14	947564067	948699787	1.00315x

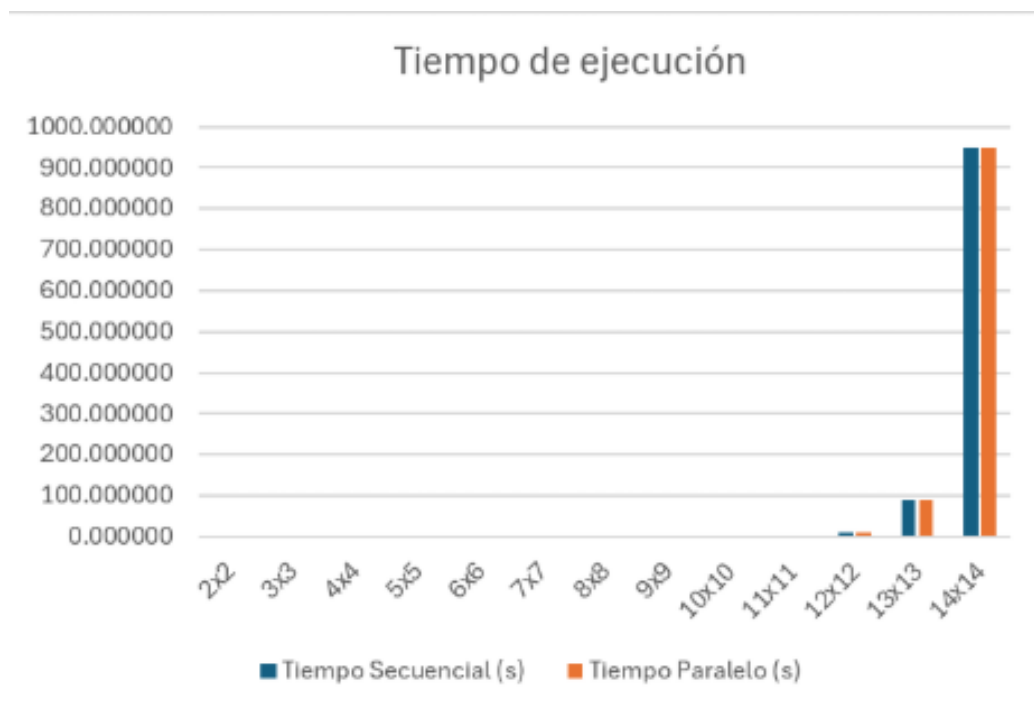


Figura 1: Grafico hasta matriz 14x14

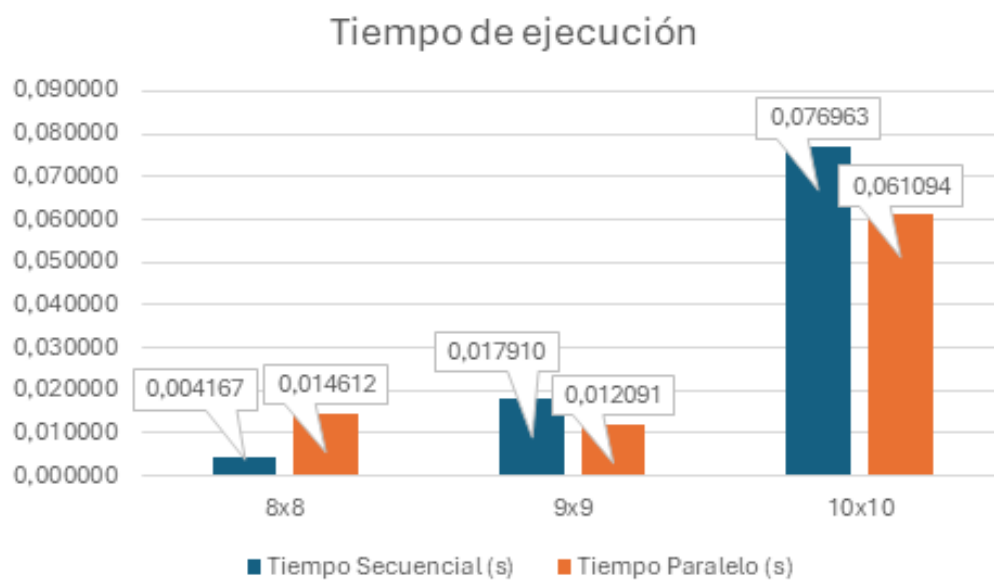


Figura 2: Grafico matriz de 8x8 a 10x10

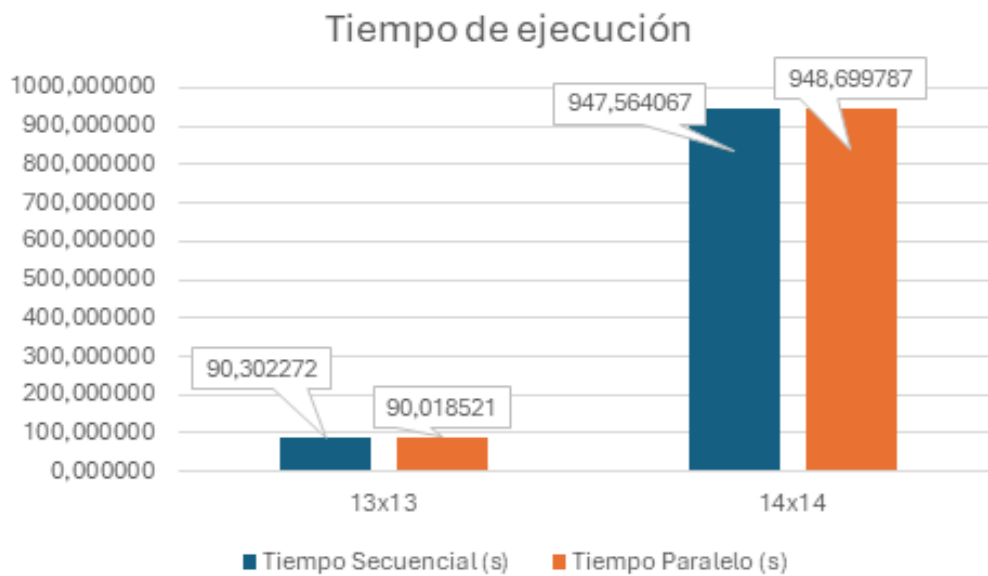


Figura 3: Grafico matriz 13x13 y 14x14

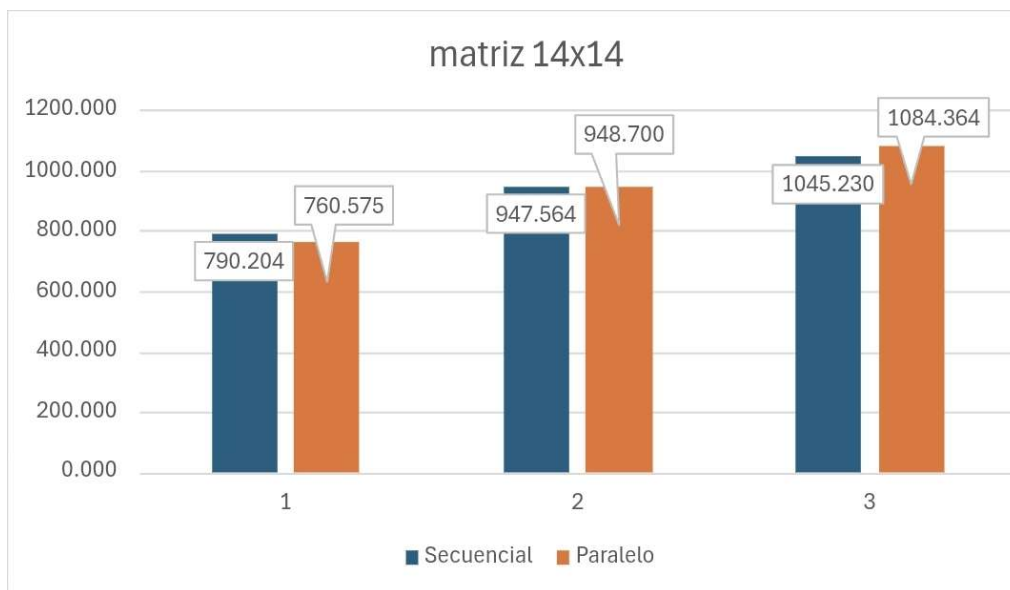


Figura 4: Comparación de 3 ejecuciones de codigo matriz 14x14

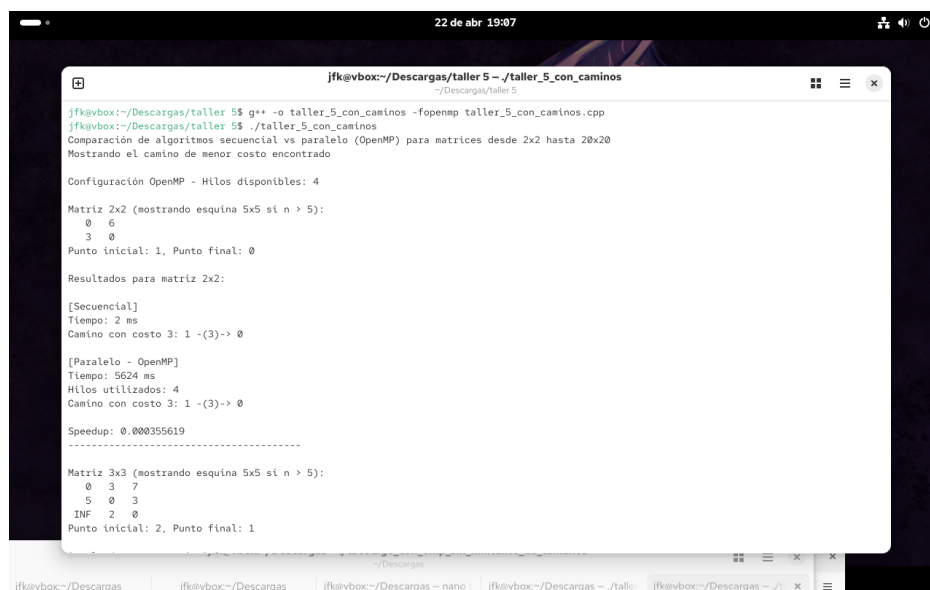
## Conclusiones

Los resultados demuestran que la paralelización comienza a ser efectiva a partir de matrices 9x9, alcanzando un speedup máximo de aproximadamente 1.48x. Este comportamiento se explica porque:

- **Para matrices pequeñas (2x2 a 8x8):** La ejecución secuencial resulta más eficiente debido al overhead que introduce la paralelización. Al realizar las mismas tareas, la versión paralela requiere más pasos e instrucciones adicionales para la gestión de hilos, lo que incrementa su complejidad y la hace menos eficiente para problemas de menor tamaño.
- **Para matrices grandes (9x9 en adelante):** La implementación paralela muestra mejor escalabilidad, ya que el beneficio de distribuir la carga computacional supera el costo inicial de la paralelización. La mayor cantidad de operaciones permite aprovechar mejor los recursos del sistema.
- **Punto de inflexión:** El cambio de comportamiento ocurre alrededor de matrices 9x9, donde el tiempo adicional de gestión de hilos se compensa con la ganancia en velocidad de procesamiento.

Este análisis confirma el principio fundamental de la computación paralela: mientras mayor sea el problema, mayor será el beneficio obtenido al paralelizar, siempre que se supere el umbral mínimo donde el overhead de la paralelización se justifique.

## Anexo: Evidencias de Máquina Virtual



```
jfk@vbox:~/Descargas/taller 5 - /taller_5_con_caminos
~/Descargas/taller 5
jfk@vbox:~/Descargas/taller 5$ g++ -o taller_5_con_camnos -fopenmp taller_5_con_camnos.cpp
jfk@vbox:~/Descargas/taller 5$ ./taller_5_con_camnos
Comparación de algoritmos secuencial vs paralelo (OpenMP) para matrices desde 2x2 hasta 20x20
Mostrando el camino de menor costo encontrado

Configuración OpenMP - Hilos disponibles: 4

Matriz 2x2 (mostrando esquina 5x5 si n > 5):
0 6
3 0
Punto inicial: 1, Punto final: 0

Resultados para matriz 2x2:

[Secuencial]
Tiempo: 2 ms
Camino con costo 3: 1 -(3)-> 0

[Paralelo - OpenMP]
Tiempo: 5624 ms
Hilos utilizados: 4
Camino con costo 3: 1 -(3)-> 0

Speedup: 0.000355619
-----
Matriz 3x3 (mostrando esquina 5x5 si n > 5):
0 3 7
5 0 3
INF 2 0
Punto inicial: 2, Punto final: 1
```

Figura 5: Compilación y ejecución del código en la máquina virtual (la unidad esta en microsegundos)

```
24 de abr 15:42
jfk@vbox:/home/jfk/Descargas/taller 5 - ./taller_5_con_caminos
~/Descargas/taller 5
jfk@vbox:/home/jfk/Descargas/taller 5 - ./taller_5_con_caminos x
jfk@vbox:~/Descargas/taller 5
Punto inicial: 2, Punto final: 1

Resultados para matriz 9x9:

[Secuencial]
Tiempo: 10753 ms
Camino con costo 3: 2 -(3)-> 1

[Paralelo - OpenMP]
Tiempo: 20511 ms
Hilos utilizados: 4
Camino con costo 3: 2 -(3)-> 1

Speedup: 0.524255
-----

Matriz 10x10 (mostrando esquina 5x5 si n > 5):
 0  2  7  9 10 ...
 1  0  9  5 10 ...
 8  4  0  9  6 ...
 3  3  8  0  4 ...
10  9  8  4  0 ...
...
Punto inicial: 7, Punto final: 6

Resultados para matriz 10x10:

[Secuencial]
Tiempo: 127417 ms
Camino con costo 5: 7 -(1)-> 0 -(4)-> 6

[Paralelo - OpenMP]
Tiempo: 107471 ms
Hilos utilizados: 4
Camino con costo 5: 7 -(1)-> 0 -(4)-> 6
```

Figura 6: Visualización aparte matrices de otra ejecución



```
GNU nano 8.1 taller_5_con_caminos.cpp
// Backtracking paralelo con OpenMP y registro del camino
void backtracking_paralelo_omp(vector<vector<int>>& matrix, int current, int end,
                             int dist, vector<int>& camino_actual, Camino& mejor_camino,
                             vector<bool>& visited, int depth = 0) {
    camino_actual.push_back(current);
    if(current == end) {
        #pragma omp critical
        {
            if(dist < mejor_camino.costo) {
                mejor_camino.costo = dist;
                mejor_camino.nodos = camino_actual;
            }
        }
        camino_actual.pop_back();
        return;
    }
    if(depth < 2) {
        #pragma omp parallel for shared(mejor_camino) firstprivate(visited, dist, camino_actual) schedule(dynamic)
        for(int i = 0; i < matrix.size(); i++) {
            if(matrix[current][i] != 0 && matrix[current][i] != INF && !visited[i]) {
                vector<bool> new_visited = visited;
                new_visited[i] = true;
                vector<int> new_camino = camino_actual;
                backtracking_paralelo_omp(matrix, i, end, dist + matrix[current][i],

```

Figura 7: Visualización del código en el editor nano