UNIVERSITAT POLITÈCNICA DE VALÈNCIA
CAMPUS D'ALCOI

Universitat Politècnica de València (UPV)

Escuela Politécnica Superior de Alcoy

Paralelización de la Búsqueda en Anchura para el 8-Puzzle usando MPI

PRÁCTICA 2

Computación Paralela
Curso 2024-2025

Autor: Marcos del Amo Fernández

Docente: Adolfo Ferre Vilaplana

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

22 de enero de 2025

Resumen

En este trabajo se estudia cómo paralelizar el algoritmo de búsqueda en anchura (BFS) aplicado al problema del 8-Puzzle, utilizando la interfaz de paso de mensajes (MPI). Se implementa un modelo maestro-esclavo y se evalúa cómo afecta la paralelización a métricas como el tiempo de ejecución, speedup y escalabilidad a nivel de cantidad de procesos. Los resultados muestran que esta estrategia tiene problemas, como los cuellos de botella en el maestro, el aumento de ineficiencias con más procesos y dificultades para coordinar la exploración del espacio de estados. Aunque la versión paralelizada no supera a la secuencial, este trabajo ayuda a entender mejor las limitaciones de paralelizar utilizando este paradigma en un modelo de programación distribuido, especialmente en problemas que requieren la compartición de variables globales.

DECLARACIÓN

En la siguiente declaración, explicitaré las partes del documento sobre los que no me corresponde la autoría:

1. Introducción, enunciado del problema del 8-puzzle. [1][4].
2. Datos sobre los detalles del Hardware [7].
3. Algoritmo Búsqueda en Anchura secuencial, realizado por mí implementación y explicación; solución de Búsqueda en Anchura secuencial para el 8-puzzle, extraido de [1] tal cual, soluciones paralelas realizadas por mí, aunque realicé una consulta a la IA generativa sobre posibles estrategias, me listo varias y elegí maestro-esclavo porque era en la que estaba más familiarizado.
4. En relación a la IA generativa, se ha utilizado ChatGPT, en tareas repetitivas o visuales, como lo es ayuda para manipulación de una plantilla de LateX, tablas, pies de página, cabeceras, y conjuntamente, enumeraciones. Obviamente siendo el resultado, manipulado últimamente por mí, a mi gusto.
5. Se ha usado material de la asignatura, así como otros recursos, principalmente como consulta [5][3][4][6]

Índice

1. Introducción

El *8-Puzzle* es un problema clásico de la literatura de la inteligencia artificial y teoría de la búsqueda. Consiste en un tablero de 3×3 que contiene 8 fichas numeradas del 1 al 8 y una casilla vacía o ficticia (representada con un 0).

El objetivo es, dado un estado inicial, llegar a un estado objetivo, con la restricción de únicamente poder mover la casilla vacía.

Un ejemplo sencillo sería:

1	2	3
4	5	6
0	7	8

Estado Inicial

$\longrightarrow$

1	2	3
4	5	6
7	8	0

Estado Objetivo

Para este ejemplo en particular, bastará con mover la casilla vacía dos veces hacia la derecha para alcanzar el estado objetivo:

1	2	3
4	5	6
0	7	8

$\xrightarrow{\text{Paso 1}}$

1	2	3
4	5	6
7	0	8

$\xrightarrow{\text{Paso 2}}$

1	2	3
4	5	6
7	8	0

El jugador (o el algoritmo) puede mover la casilla vacía hacia arriba, abajo, izquierda o derecha, siempre que el movimiento sea válido. Si la casilla vacía está en el centro, se podrán realizar los 4 movimientos. Si está en una esquina, entonces el número de movimientos válidos es 2; en cualquier otro caso, existen 3 posibles movimientos para el espacio vacío. Esto da lugar a un **factor de ramificación medio**:

$$b = \frac{1 \cdot 4 + 4 \cdot 2 + 4 \cdot 3}{9} = 2{,}67.$$

El número total de estados o configuraciones del problema que se puede generar en el juego del 8-Puzzle es:

$$9! = 362{,}880 \text{ estados.}$$

Sin embargo, debido a restricciones de paridad, sólo la mitad de estas configuraciones son alcanzables desde un estado inicial válido, dejando aproximadamente 181.440 configuraciones posibles para explorar.

A lo largo de las últimas décadas, el 8-Puzzle ha sido ampliamente investigado en diversas áreas de la computación. Dentro de la teoría de búsqueda, ha servido como un problema de referencia para evaluar y comparar la eficiencia de algoritmos como *Búsqueda en Amplitud* (*Breadth-First Search*, BFS),

Búsqueda en Profundidad (*Depth-First Search*, DFS), y enfoques heurísticos como *A** y *IDA**. En particular, su estructura de grafo implícito y su espacio de estados limitado permiten realizar análisis detallados del desempeño de estos algoritmos.

En el ámbito de la optimización, el 8-Puzzle se utiliza como un modelo simplificado para estudiar problemas más complejos en áreas como logística, robótica y sistemas de planeación. Algoritmos metaheurísticos, como *simulated annealing* o *genetic algorithms*, también han sido aplicados para resolver variantes del problema.

Además, el 8-Puzzle se emplea en la docencia de inteligencia artificial y computación para ilustrar conceptos fundamentales como el diseño de heurísticas, la búsqueda óptima y las limitaciones computacionales asociadas al crecimiento exponencial del espacio de búsqueda.

2. Motivación personal

El **8-Puzzle** es un problema que se ha trabajado previamente en la asignatura de Sistemas Inteligentes, donde se exploraron múltiples soluciones secuenciales, como las mencionadas anteriormente. En esta práctica, quise aprovechar la oportunidad para lograr un entendimiento más profundo del problema y tratar de realizar su versión paralelizada. Además, siento curiosidad por comparar el rendimiento de algoritmos secuenciales altamente optimizados con algoritmos paralelos, para evaluar hasta qué punto la paralelización que yo pueda realizar es capaz de superar un diseño secuencial muy eficiente.

Soy consciente de que, dado el tipo de problema; talla reducida, con un espacio fijo de 9!, es altamente probable que el diseño secuencial resulte más eficiente en términos absolutos. Sin embargo, considero que esta práctica es una buena oportunidad para explorar los principios de la paralelización bajo el paradigma que impone MPI, incluso si no se alcanzan mejoras significativas.

Asimismo, reconozco que este problema podría ser demasiado ambicioso para mis conocimientos actuales. Por ello, trataré de simplificarlo al máximo, centrándome exclusivamente en la paralelización del algoritmo *BFS* y comparándolo con su alternativa secuencial, así como, potencialmente, con otras versiones más eficientes.

3. Consideraciones

Para la realización de este trabajo, se han empleado los siguientes recursos y configuraciones:

3.1. Detalles del Hardware

Las pruebas se han llevado a cabo en un sistema con las siguientes características:

- **Modelo del equipo:** MacBook Air (Mac15)
- **Chip:** Apple M2
- **Cantidad de núcleos:** 8 (4 de rendimiento y 4 de eficiencia)
- **Memoria RAM:** 16 GB

Cuadro 1: Especificaciones técnicas del chip Apple M2

Característica	**Especificación**
Procesador	
Número de núcleos de CPU	8 (4 de rendimiento y 4 de eficiencia)
Frecuencia máxima	No especificado (dinámico)
Caché L2 compartida núcleos AR	16 MB
Caché L2 compartida núcleos AE	4 MB
Memoria	
Memoria máxima unificada	24 GB
Ancho de banda de memoria	100 GB/s
Fabricación y Fotolitografía	
Tecnología de fabricación	5 nm (segunda generación)
Fabricante	TSMC
Gráficos y Neural Engine	
Número de núcleos gráficos	10
Rendimiento FP32	3,6 TFLOPS
Número de núcleos Neural Engine (NE)	16
Rendimiento Neural Engine	15,8 TOPS

3.2. Compilación y Ejecución

El código fue compilado utilizando el compilador de `C++` incluido con OpenMPI (`mpicxx`), y ejecutado mediante el comando `mpirun` desde terminal para lanzar múltiples procesos paralelos. Todos los experimentos se han realizado en el mismo entorno, con la configuración mencionada anteriormente y mediante el editor de código de Visual Studio Code.

3.3. Limitaciones y Alcance

- Es importante subrayar que todo el código presentado en este documento está escrito exclusivamente en `C++`, elegido para facilitar la implementación al utilizar la librería `std::`; que nos ofrece estructuras de datos útiles como el vector, la cola y el conjunto en su versión `unordered-set` (buscamos coste constante en las operaciones).
- Se explicará con detalle únicamente aspectos de la paralelización, y la implementación en `MPI`, y se omitirán explicaciones de la implementación relacionadas con la semántica de `c++` como lenguaje, ya que se consideran que no aportan valor a una práctica de Computación Paralela.
- Se es consciente de que, aunque OpenMPI proporciona un entorno portable, los resultados obtenidos podrían variar en sistemas con diferentes arquitecturas de hardware o implementaciones de `MPI`.
- Se es consciente de que el problema estudiado (8-Puzzle) tiene una talla fija y un espacio de búsqueda limitado, lo cual condiciona los resultados de las pruebas de rendimiento.

4. Algoritmo de Búsqueda en Anchura (BFS) Secuencial

El algoritmo de búsqueda en anchura, en inglés *Breadth-First Search (BFS)*, por definición recorre un grafo nivel a nivel, utilizando una cola FIFO (First-In First-Out) para replicar este comportamiento de forma natural.

```cpp
// BFS
void BFS (const vector<vector<int>> &graph, int start)
{
    queue<int> queue;
    unordered_set<int> visited;

    queue.push(start);
    visited.insert(start);

    while (!queue.empty()) {
        int node = queue.front();
        queue.pop();

        // PROCESAR NODO ACTUAL

        for (int adj : graph[node]) {
            if (visited.find(adj) != visited.end()) continue;
            visited.insert(adj);
            queue.push(adj);
        }
    }
}
```

Listing 1: Implementación del algoritmo BFS en C++

Se muestra el algoritmo genérico para recorrer grafos por nivel, donde la parte de *procesar* el nodo actual dependerá de la lógica que queramos aplicar.

La implementación del BFS para el 8-puzzle no se aleja mucho de esta idea, ya que el conjunto de estados es esencialmente un grafo implícito y en nuestro caso *procesar* será el equivalente a comprobar si el estado actual corresponde con el estado objetivo, es decir, si hemos finalizado la búsqueda.

BFS para el 8-Puzzle se implementa típicamente como:

1. Insertar el estado inicial en una cola.
2. Mientras la cola no esté vacía:
   - Extraer el estado.
   - Si es el estado objetivo, retornar la solución.
   - Generar todos los sucesores (los alcanzables con un movimiento válido).
   - Añadir aquellos sucesores no visitados a la cola.

- Marcar como visitados.

```
        // BFS (8-puzzle)
    queue<Node> queue;
    unordered_set<string> visited;

    queue.push({startState, 0});
    visited.insert(stateToString(startState));
    max_stored = queue.size();

    bool found = false;


    while (!queue.empty() && !found) {
        Node curr = queue.front();
        queue.pop();
        expanded++;

                // PROCESAR NODO ACTUAL (COMPROBAR SI ES LA SOLUCION)
        if (is_solution(curr.st)) {
            found = true;
            solution_cost = current.depth;
            break;
        }

        vector<State> successors = getSuccessors(curr.st);
        generated += successors.size();

        for (auto &succ : successors) {
            string key = stateToString(succ);
            if (visited.find(key) != visited.end()) continue;
            visited.insert(key);
            frontier.push({succ, current.depth + 1});

        }
    }
```

Listing 2: Implementación del algoritmo BFS en C++

Se muestra una solución BFS secuencial para el problema del 8-puzzle, que compone la base de la estrategia de la solución BFS secuencial utilizada para la comparación de rendimientos. Ver anexo para implementación completa.

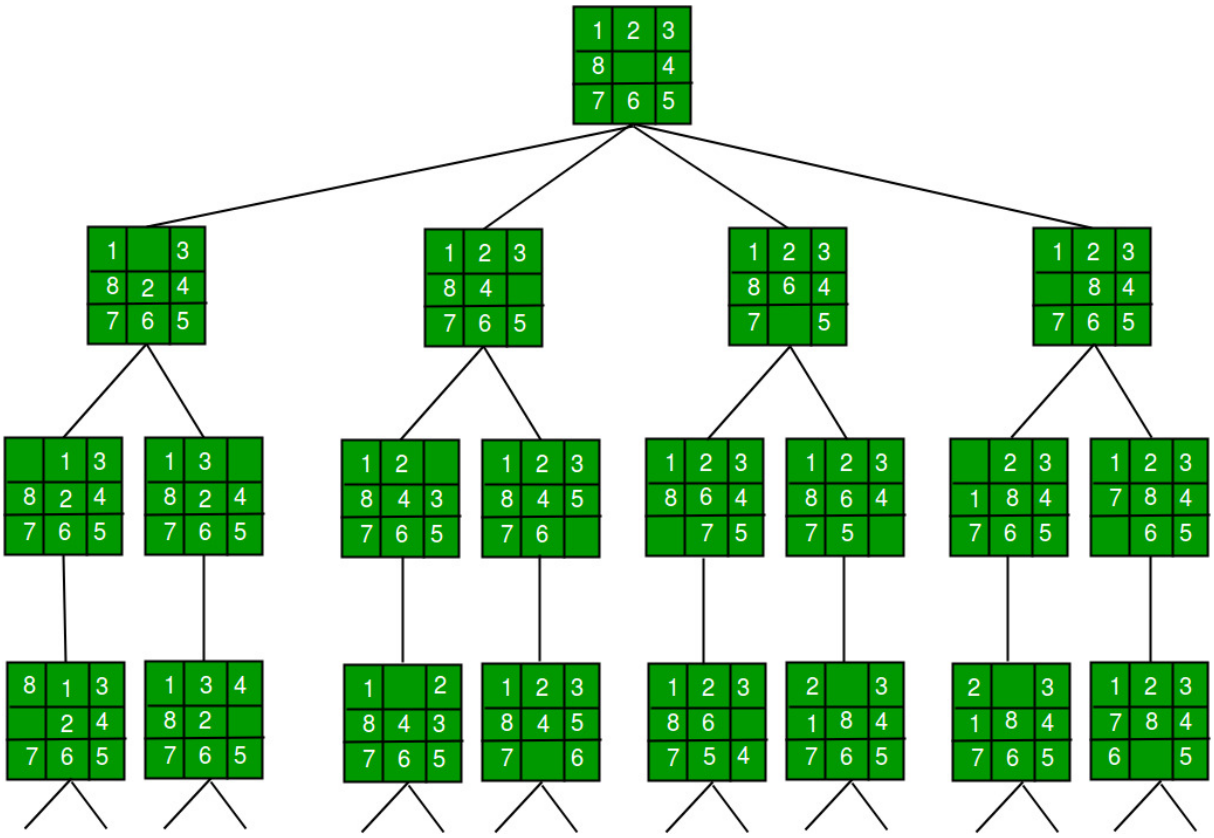

Figura 1: Ejemplo visual ramificación de estados del 8-puzzle

La complejidad temporal es $\mathcal{O}(b^d)$, donde b es el factor de ramificación (en el 8-Puzzle típicamente 2 a 4) y d la profundidad de la solución. La complejidad espacial es similar, ya que tanto la cola como el conjunto de nodos visitados almacenan en memoria los estados.

5. Paralelización del BFS con MPI

A continuación, se presentará una posible estrategia de paralelización del algoritmo BFS para el problema del 8-puzzle, aunque antes de eso, se presenta el proceso que se ha llevado a cabo para alcanzar a dicha solución. Análogamente a la explicación del algoritmo secuencial BFS, empezaremos con una solución paralelización BFS genérica, que luego se aplicará al problema del 8-puzzle.

Dada la naturaleza del modo de programación que plantea MPI, resulta intuitivo seguir una estrategia en la que un proceso se encarga del procesamiento de los nodos, y los demás procesos expanden los nodos en paralelo. Este paradigma es muy común en modelos de programación de memoria distribuida, y se conoce como *paradigma maestro-esclavo*, y se ha visto aplicado en los apuntes de asignaturas como Computación Paralela y Concurrencia y Sistemas Distribuidos.

La estrategia de paralelización del algoritmo BFS MPI utilizada sigue precísamente este paradigma, donde un proceso maestro coordina la exploración del grafo y varios esclavos realizan paralelamente las operaciones de expansión de nodos.

Es importante señalar que dado que estamos en un paradigma de paralelismo explícito, somos nosotros, los programadores, que decidimos qué y cómo se paraleliza y, en pocas palabras, paralelizaremos las expansiones de nodos utilizando una estrategia *maestro-esclavo*.

5.1. Consideraciones de la implementación

- Se han utilizado etiquetas (`tags`) para facilitar la comprensión de la comunicación en MPI:

```
1    // TAGS DE COMUNICACION
2    enum Tag {
3        EXPAND = 1,     // MAESTRO -> ESCLAVO: Nodo a expandir
4        STOP = 2,       // MAESTRO -> ESCLAVO: Mensaje de parada
5        SUCCESORS = 3   // ESCLAVO -> MAESTRO: Sucesores generados
6    };
```

Listing 3: Tags utilizados para la comunicación MPI

- Para esta estrategia no se van a utilizar primtivas MPI colectivas, ya que se va a dividir por un lado las tareas que realiza el maestro y por el otro, las tareas que realizan los esclavos. Y en MPI, las funciones colectivas han de ser invocadas por todos los procesos de la comunicación.

5.2. Tareas del maestro

El maestro es responsable de las siguientes acciones:

1. Inicializar las estructuras de datos y distribuir los nodos a los esclavos.
2. Recibir resultados de los esclavos, procesar los sucesores y verificar si se alcanzó la solución.
3. Reasignar nodos a los esclavos o enviar mensajes de parada cuando no quedan más nodos por explorar.

5.2.1. Inicialización del maestro y distribución de nodos

```
// INICIALIZACION MAESTRO
queue<int> queue;
unordered_set<int> visited;
bool found = false;

queue.push(start);
visited.insert(start);

int active_slaves = 0;

// DISTRIBUCION DE NODOS
for (int i = 1; i < size; i++) {
    if (!queue.empty()) {
        int node = queue.front();
        queue.pop();
        MPI_Send(&node, 1, MPI_INT, i, EXPAND, MPI_COMM_WORLD);
        active_slaves++;
    } else break;
}
```

Listing 4: Inicialización del maestro y distribución de nodos

Descripción:

- `MPI_Send`: Se utiliza para enviar nodos desde el maestro a los esclavos con la etiqueta `EXPAND`.

5.2.2. Procesamiento de resultados de los esclavos

```
while (!found && active_slaves > 0) {
    int slave;
    MPI_Status status;

    int n_successors;
    MPI_Recv(&n_successors, 1, MPI_INT, MPI_ANY_SOURCE, SUCCESORS,
        MPI_COMM_WORLD, &status);
    slave = status.MPI_SOURCE;

    if (n_successors > 0) {
```

```
10          vector<int> buffer(n_successors);
11          MPI_Recv(buffer.data(), n_successors, MPI_INT, slave, SUCCESORS,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
12
13          for (int i = 0; i < n_successors; i++) {
14              int successor = buffer[i];
15              if (visited.find(successor) != visited.end()) continue;
16              visited.insert(successor);
17
18              // PROCESAMIENTO DE RESULTADO
19              if (successor == SOLUTION) {
20                  found = true;
21                  for (int w = 1; w < size; w++) {
22                      MPI_Send(NULL, 0, MPI_INT, w, STOP, MPI_COMM_WORLD);
23                  }
24                  break;
25              } else {
26                  queue.push(successor);
27              }
28          }
29      }
30  }
```

Listing 5: Procesamiento de resultados recibidos de los esclavos

Descripción:

- `MPI_Recv`: Recibe el número de sucesores generados por un esclavo y posteriormente los sucesores.
- Si se encuentra la solución (`successor == SOLUTION`), se envía un mensaje de parada (`STOP`) a todos los esclavos.
- Si no es la solución, los sucesores se añaden a la cola para su posterior procesamiento.

5.2.3. Reasignación de nodos o finalización de esclavos

```
1   if (!found) {
2       if (!queue.empty()) {
3           int next_node = queue.front();
4           queue.pop();
5           MPI_Send(&next_node, 1, MPI_INT, slave, EXPAND, MPI_COMM_WORLD);
6       } else {
7           MPI_Send(NULL, 0, MPI_INT, slave, STOP, MPI_COMM_WORLD);
8           active_slaves--;
9       }
10  }
```

Listing 6: Reasignación de nodos o finalización de esclavos

Descripción:

- Si quedan nodos en la cola, se asigna un nuevo nodo al esclavo inactivo.

- Si no quedan nodos, se envía un mensaje de parada (`STOP`) al esclavo y se decrementa el contador `active_slaves`.

5.3. Tareas del esclavo

Cada esclavo realiza las siguientes acciones:

1. Recibir nodos desde el maestro para expandir.
2. Generar sucesores del nodo actual y enviarlos de vuelta al maestro.
3. Escuchar mensajes de parada y finalizar su ejecución cuando sea necesario.

```
bool stop = false;
while (!stop) {
    int current_node;
    int n_successors;
    MPI_Status status;

    MPI_Recv(&current_node, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
       &status);
    int tag = status.MPI_TAG;

    if (tag == STOP) {
        stop = true;
    } else if (tag == EXPAND) {
        const vector<int> &adj_nodes = graph[current_node];
        n_successors = adj_nodes.size();

        MPI_Send(&n_successors, 1, MPI_INT, 0, SUCCESORS, MPI_COMM_WORLD
           );

        if (n_successors > 0) {
            MPI_Send(adj_nodes.data(), n_successors, MPI_INT, 0,
               SUCCESORS, MPI_COMM_WORLD);
        }
    }
}
```

Listing 7: Implementación del esclavo

Descripción:

- `EXPAND`: El esclavo recibe un nodo, genera sus sucesores y los envía al maestro.
- `STOP`: Detiene la ejecución del esclavo.

6. Paralelización del BFS en el 8-puzzle con MPI

Una vez que ya tenemos bien definida la estrategia de paralelización BFS, implementarla para nuestro problema del 8-puzzle sólo implicará considerar una serie de ajustes.

6.1. Consideraciones de la implementación

- Adición de métricas adicionales para evaluar el rendimiento del algoritmo, como el número de nodos generados, expandidos, el tamaño máximo de la cola (`max_queue_size`), y la profundidad máxima alcanzada (`max_depth`).
- La diferencia principal con respecto a una implementación genérica de BFS es que en esta versión, el buffer que se envía entre maestro y esclavos está basado en un `struct` que representa el estado del tablero:

```
1    struct State {
2        int cells[9];
3    };
```

Listing 8: Definición del `struct` para representar el estado

6.2. Tareas del maestro

De nuevo, el maestro realiza tres acciones principales:

1. Inicializa las estructuras de datos y distribuye los nodos iniciales a los esclavos.
2. Procesa los resultados de los esclavos y verifica si se encontró la solución.
3. Reasigna nodos a los esclavos o envía mensajes de parada cuando no hay más nodos por explorar.

6.2.1. Inicialización del maestro y distribución de nodos

```
// INICIALIZACION DEL MAESTRO
queue<Node> queue;
unordered_set<string> visited;

queue.push({start, 0});
visited.insert(state_to_string(start));
if ((long long)queue.size() > max_queue_size) max_queue_size = queue.
    size();

int active_slaves = 0;

// DISTRIBUCION INICIAL DE NODOS
for (int i = 1; i < size; i++) {
    if (!queue.empty()) {
        Node node = queue.front();
        queue.pop();
        states_expanded++;

        int buffer[10];
        for (int j = 0; j < 9; j++) buffer[j] = node.state.cells[j];
        buffer[9] = node.depth;
        MPI_Send(buffer, 10, MPI_INT, i, EXPAND, MPI_COMM_WORLD);
        active_slaves++;
    } else break;
}
```

Listing 9: Inicialización del maestro y distribución inicial de nodos

6.2.2. Procesamiento de resultados de los esclavos

```
while (!found && active_slaves > 0) {
    int slave;
    MPI_Status status;

    int n_successors;
    MPI_Recv(&n_successors, 1, MPI_INT, MPI_ANY_SOURCE, SUCCESORS,
       MPI_COMM_WORLD, &status);
    slave = status.MPI_SOURCE;

    if (n_successors > 0) {
        vector<int> buffer(n_successors * 10);
        MPI_Recv(buffer.data(), n_successors * 10, MPI_INT, slave,
           SUCCESORS, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        states_generated += n_successors;

        for (int i = 0; i < n_successors; i++) {
            State new_state;
            for (int j = 0; j < 9; j++) new_state.cells[j] = buffer[i *
               10 + j];
            int new_depth = buffer[i * 10 + 9];
            string key = state_to_string(new_state);

            if (visited.find(key) == visited.end()) {
                visited.insert(key);
                //PROCESAMIENTO DE RESULTADO
                if (is_solution(new_state)) {
                    found = true;
                    cost = new_depth;
                    for (int w = 1; w < size; w++) {
                        MPI_Send(NULL, 0, MPI_INT, w, STOP,
                           MPI_COMM_WORLD);
                    }
                    break;
                } else {
                    queue.push({new_state, new_depth});
                    if (new_depth > max_depth) max_depth = new_depth;
                    if ((long long)queue.size() > max_queue_size)
                        max_queue_size = queue.size();
                }
            }
        }
    }
}
```

Listing 10: Procesamiento de resultados recibidos de los esclavos

6.2.3. Reasignación de nodos o finalización de esclavos

```
if (!found) {
    if (!queue.empty()) {
        Node next_node = queue.front();
        queue.pop();
        states_expanded++;

        int buffer[10];
        for (int j = 0; j < 9; j++) buffer[j] = next_node.state.cells[j
            ];
        buffer[9] = next_node.depth;
        MPI_Send(buffer, 10, MPI_INT, slave, EXPAND, MPI_COMM_WORLD);
    } else {
        MPI_Send(NULL, 0, MPI_INT, slave, STOP, MPI_COMM_WORLD);
        active_slaves--;
    }
}
```

Listing 11: Reasignación de nodos o finalización de esclavos

6.3. Tareas del esclavo

De nuevo, cada esclavo realiza las siguientes acciones:

1. Recibir nodos desde el maestro para expandir.
2. Generar los sucesores del nodo actual y enviarlos al maestro.
3. Escuchar mensajes de parada y finalizar su ejecución cuando sea necesario.

```
bool stop = false;
while (!stop) {
    int buffer[10];
    int n_successors;
    MPI_Status status;
        // RECIBIR NODOS
    MPI_Recv(buffer, 10, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
        status);
    int tag = status.MPI_TAG;

    if (tag == STOP) {
        stop = true; // FINALIZAR
    } else if (tag == EXPAND) {
        State current_state;
        for (int i = 0; i < 9; i++) current_state.cells[i] = buffer[i];
        int current_depth = buffer[9];
                // GENERAR SUCESORES
        vector<State> successors = get_successors(current_state);
        n_successors = successors.size();
                // ENVIAR SUCESORES
        MPI_Send(&n_successors, 1, MPI_INT, 0, SUCCESORS, MPI_COMM_WORLD
            );
```

```
21
22          if (n_successors > 0) {
23              vector<int> send_buffer(n_successors * 10);
24              for (int i = 0; i < n_successors; i++) {
25                  for (int j = 0; j < 9; j++) {
26                      send_buffer[i * 10 + j] = successors[i].cells[j];
27                  }
28                  send_buffer[i * 10 + 9] = current_depth + 1;
29              }
30              MPI_Send(send_buffer.data(), n_successors * 10, MPI_INT, 0,
                    SUCCESORS, MPI_COMM_WORLD);
31          }
32      }
33  }
```

Listing 12: Implementación del esclavo

7. Resultados

Los estados inicial y final utilizados para las pruebas de rendimiento son los siguientes:

0	8	7
6	5	4
3	2	1

Estado Inicial

$\longrightarrow$

1	2	3
4	5	6
7	8	0

Estado Objetivo

Sobre este escenario de ejecución, obtenemos los siguientes resultados:

Número de Procesos	Nodos Generados	Nodos Expandidos	Tiempo (s)
Secuencial	476,080	178,224	0.551378
2	460,298	172,569	0.606374
3	460,298	172,569	0.622449
4	460,298	172,569	0.630995
5	460,298	172,569	0.791588
6	460,298	172,569	0.896145
7	460,298	172,569	0.976613
8	460,298	172,569	1.14333

7.1. Tiempo de Ejecución y Speedup

En la tabla, se presenta una comparación entre la versión secuencial (1 proceso) y la versión paralela con distintos números de procesos. La métrica de *speedup* se calcula como $T_{secuencial}/T_{paralelo}$.

Cuadro 2: Comparación de tiempos de ejecución (ejemplo) y speedup

Núm. de procesos	Tiempo (s)	Speedup
2	0.606374	0.91
3	0.622449	0.89
4	0.630995	0.87
5	0.791588	0.70
6	0.896145	0.62
7	0.976613	0.56
8	1.14333	0.48

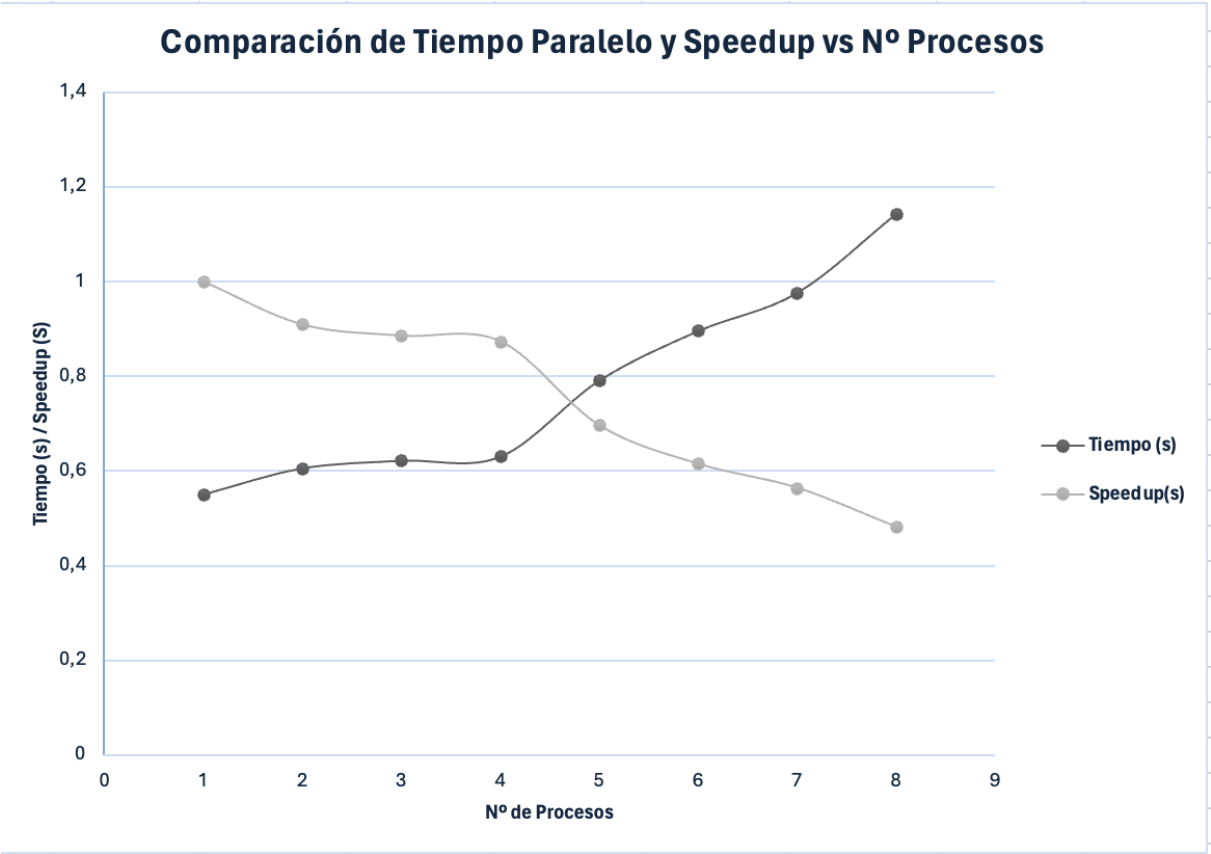

Figura 2: Comparación del tiempo de ejecución y speedup según el número de procesos.

8. Discusión

La implementación del BFS paralelizado con el modelo maestro-esclavo demuestra ser ineficiente en el contexto del 8-puzzle, el speedup disminuye y el tiempo de ejecución amuenta a medida que ejecutamos el algoritmo con más procesos. El principal problema radica en el cuello de botella del maestro, que limita la escalabilidad y provoca tiempos muertos en los esclavos a medida que aumenta el número de procesos. Esto genera una sobrecarga de comunicación que empeora el rendimiento con más procesos, y el uso de primitivas bloqueantes, especialmente el tener que recibir dinámicamente los sucesores (con dos primitivas de recepción bloqueante, una para la cantidad y otra para los estados), no facilita la paralelización en absoluto.

Además, el BFS puede no ser adecuado para problemas con espacios de búsqueda grandes, debido a la explosión combinatoria de estados y su alto consumo de memoria, que encima conceptualmente debe ser compartida (cola y conjunto de visitados). La estrategia maestro-esclavo no es una solución efectiva para este problema y existe la necesidad de investigar mejores estrategias de paralelización.

Sin embargo, bajo ninguna circunstancia se pretende sugerir que estos resultados demuestran que no haya otra posible solución que, utilizando la misma estrategia, consiga satisfacer los objetivos planteados en esta práctica.

9. Conclusiones

Los resultados obtenidos dejan claro que la solución de paralelización del BFS en el 8-Puzzle no ha sido adecuada. A pesar de distribuir la exploración del espacio de estados entre múltiples procesos, el algoritmo paralelo no ha sido capaz de superar al secuencial en términos de eficiencia.

Todo indica que para el problema del 8-Puzzle en concreto, la comparación con otros algoritmos y heurísticas como A* o IDA* habría sido aún más dramática en favor de las versiones secuenciales. Esto evidencia las limitaciones del BFS como enfoque base para la paralelización, especialmente en un modelo de programación distribuida, ya que se utilizan estructuras de datos como la cola de expandidos, y conjunto de visitados, que han de ser compartidas entre los procesos y dificultan la implementación.

En conclusión, aunque el BFS paralelo desarrollado no ha demostrado ser efectivo en este caso, esta segunda práctica de la asignatura ha servido para al menos comprender las limitaciones de paralelización en estrategias con similares características.

Referencias

[1] *El juego del 8-Puzzle*, material de la asignatura Sistemas Inteligentes, Universitat Politècnica de València, Escuela Politécnica Superior de Alcoy. Disponible en: `https://dsic.gitbook.io/sin/p1/el-juego-del-8-puzzle`. Última consulta: 13 de diciembre de 2024.

[2] Open MPI Documentation, *Open MPI v5.0.x*. Disponible en: `https://docs.open-mpi.org/en/v5.0.x/`. Última consulta: 13 de diciembre de 2024.

[3] Notas de clase de Computación Paralela, Universitat Politècnica de València, Escuela Politécnica Superior de Alcoy. Última consulta: 13 de diciembre de 2024.

[4] Notas de clase de Sistemas Inteligentes, Universitat Politècnica de València, Escuela Politécnica Superior de Alcoy. Última consulta: 13 de diciembre de 2024.

[5] GeeksforGeeks, *8-Puzzle Problem using Branch and Bound*. Disponible en: `https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/`. Última consulta: 13 de diciembre de 2024.

[6] TheCodest, *Arquitectura maestro-esclavo*. Disponible en: `https://thecodest.co/es/diccionario/arquitectura-maestro-esclavo/#:~:text=La%20arquitectura%20maestro%2Desclavo%20es,los%20resultados%20al%20nodo%20maestro.`. Última consulta: 13 de diciembre de 2024.

[7] Xataka, *Microarquitectura del procesador M2 de Apple explicada: así sube la apuesta por el rendimiento y la eficiencia*. Disponible en: `https://www.xataka.com/componentes/microarquitectura-procesador-m2-apple-explicada-asi-sube-apuesta-rend` Última consulta: 13 de diciembre de 2024.

A. Anexos

A.1. Implementación Paralela: BFS 8-puzzle con MPI

```
#include <mpi.h>
#include <iostream>
#include <queue>
#include <unordered_set>
#include <vector>
#include <algorithm>
#include <cstring>
#include <string>

using namespace std;

enum Etiqueta {
    EXPANDIR  = 1,
    PARAR     = 2,
    SUCESORES = 3
};

struct Estado {
    int celdas[9];
};

string estadoACadena(const Estado &est) {
    string cadena;
    for (int i = 0; i < 9; ++i) {
        cadena += to_string(est.celdas[i]) + ",";
    }
    return cadena;
}

bool esObjetivo(const Estado &est) {
    Estado meta = {{1, 2, 3, 4, 5, 6, 7, 8, 0}};
    for (int i = 0; i < 9; ++i) {
        if (est.celdas[i] != meta.celdas[i]) {
            return false;
        }
    }
    return true;
}

vector<Estado> obtenerSucesores(const Estado &est) {
    vector<Estado> sucesores;
    int posVacia = 0;

    for (int i = 0; i < 9; ++i) {
        if (est.celdas[i] == 0) {
            posVacia = i;
            break;
        }
    }

    int fila = posVacia / 3;
    int col  = posVacia % 3;
    int movimientos[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

```
54
55          for (auto &mov : movimientos) {
56              int nuevaFila = fila + mov[0];
57              int nuevaCol  = col + mov[1];
58              if (nuevaFila >= 0 && nuevaFila < 3 && nuevaCol >= 0 &&
                    nuevaCol < 3) {
59                  int nuevaPos = nuevaFila * 3 + nuevaCol;
60                  Estado nuevo = est;
61                  swap(nuevo.celdas[posVacia], nuevo.celdas[nuevaPos]);
62                  sucesores.push_back(nuevo);
63              }
64          }
65          return sucesores;
66      }
67
68      struct Nodo {
69          Estado est;
70          int    profundidad;
71      };
72
73      void MAESTRO(const Estado &estadoInicial, int numProcesos) {
74          long long generados = 0;
75          long long expandidos = 0;
76          long long maximoCola = 0;
77          int profundidadMax = 0;
78          int costeSolucion = -1;
79          bool encontrado = false;
80
81          double tiempoInicio = MPI_Wtime();
82
83          queue<Nodo> cola;
84          unordered_set<string> visitados;
85
86          cola.push({estadoInicial, 0});
87          visitados.insert(estadoACadena(estadoInicial));
88          if ((long long)cola.size() > maximoCola) {
89              maximoCola = cola.size();
90          }
91
92          int esclavosActivos = 0;
93
94          for (int i = 1; i < numProcesos; ++i) {
95              if (!cola.empty()) {
96                  Nodo nd = cola.front();
97                  cola.pop();
98                  expandidos++;
99
100                 int buffer[10];
101                 for (int j = 0; j < 9; ++j) {
102                     buffer[j] = nd.est.celdas[j];
103                 }
104                 buffer[9] = nd.profundidad;
105
106                 MPI_Send(buffer, 10, MPI_INT, i, EXPANDIR, MPI_COMM_WORLD
                        );
107                 ++esclavosActivos;
108             } else {
109                 break;
```

```
110             }
111         }
112 
113         while (!encontrado && esclavosActivos > 0) {
114             MPI_Status estadoMPI;
115             int numSucesores;
116 
117             MPI_Recv(&numSucesores, 1, MPI_INT, MPI_ANY_SOURCE, SUCESORES
                    , MPI_COMM_WORLD, &estadoMPI);
118             int esclavo = estadoMPI.MPI_SOURCE;
119 
120             if (numSucesores > 0) {
121                 vector<int> buffer(numSucesores * 10);
122                 MPI_Recv(buffer.data(), numSucesores * 10, MPI_INT,
                        esclavo, SUCESORES, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
                        ;
123 
124                 generados += numSucesores;
125 
126                 for (int i = 0; i < numSucesores; ++i) {
127                     Estado nuevoEstado;
128                     for (int j = 0; j < 9; ++j) {
129                         nuevoEstado.celdas[j] = buffer[i * 10 + j];
130                     }
131                     int nuevaProfundidad = buffer[i * 10 + 9];
132                     string clave = estadoACadena(nuevoEstado);
133 
134                     if (visitados.find(clave) != visitados.end()) {
135                         continue;
136                     }
137                     visitados.insert(clave);
138 
139                     if (esObjetivo(nuevoEstado)) {
140                         encontrado = true;
141                         costeSolucion = nuevaProfundidad;
142                         for (int w = 1; w < numProcesos; ++w) {
143                             MPI_Send(NULL, 0, MPI_INT, w, PARAR,
                                    MPI_COMM_WORLD);
144                         }
145                         break;
146                     } else {
147                         cola.push({nuevoEstado, nuevaProfundidad});
148                         if (nuevaProfundidad > profundidadMax) {
149                             profundidadMax = nuevaProfundidad;
150                         }
151                         if ((long long)cola.size() > maximoCola) {
152                             maximoCola = cola.size();
153                         }
154                     }
155                 }
156             }
157 
158             if (!encontrado && !cola.empty()) {
159                 Nodo siguiente = cola.front();
160                 cola.pop();
161                 expandidos++;
162 
163                 int buffer[10];
```

```
164                 for (int j = 0; j < 9; ++j) {
165                     buffer[j] = siguiente.est.celdas[j];
166                 }
167                 buffer[9] = siguiente.profundidad;
168                 MPI_Send(buffer, 10, MPI_INT, esclavo, EXPANDIR,
                        MPI_COMM_WORLD);
169             } else if (!encontrado) {
170                 --esclavosActivos;
171             }
172         }
173 
174         double tiempoFin = MPI_Wtime();
175         double tiempoTotal = tiempoFin - tiempoInicio;
176 
177         cout << "#␣estrategia␣␣␣␣␣␣␣␣generados␣␣expandidos␣␣max_cola␣␣␣
                coste␣␣prof_max␣␣␣tiempo(s)\n";
178         cout << "BFS␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
179              << generados << "␣␣␣␣␣␣␣"
180              << expandidos << "␣␣␣␣␣␣␣␣␣␣"
181              << maximoCola << "␣␣␣␣␣␣␣"
182              << costeSolucion << "␣␣␣␣␣␣␣"
183              << profundidadMax << "␣␣␣␣␣␣␣"
184              << tiempoTotal << "\n";
185     }
186 
187     void ESCLAVO() {
188         bool parar = false;
189 
190         while (!parar) {
191             int buffer[10];
192             int numSucesores;
193             MPI_Status estadoMPI;
194 
195             MPI_Recv(buffer, 10, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
                    &estadoMPI);
196             int etiq = estadoMPI.MPI_TAG;
197 
198             if (etiq == PARAR) {
199                 parar = true;
200             } else if (etiq == EXPANDIR) {
201                 Estado estadoActual;
202                 for (int i = 0; i < 9; ++i) {
203                     estadoActual.celdas[i] = buffer[i];
204                 }
205                 int profundidadActual = buffer[9];
206 
207                 vector<Estado> sucesores = obtenerSucesores(estadoActual)
                        ;
208                 numSucesores = sucesores.size();
209 
210                 MPI_Send(&numSucesores, 1, MPI_INT, 0, SUCESORES,
                        MPI_COMM_WORLD);
211 
212                 if (numSucesores > 0) {
213                     vector<int> enviar(numSucesores * 10);
214                     for (int i = 0; i < numSucesores; ++i) {
215                         for (int j = 0; j < 9; ++j) {
216                             enviar[i * 10 + j] = sucesores[i].celdas[j];
```

```
                        }
                        enviar[i * 10 + 9] = profundidadActual + 1;
                    }
                    MPI_Send(enviar.data(), numSucesores * 10, MPI_INT,
                        0, SUCESORES, MPI_COMM_WORLD);
                }
            }
        }
    }

    int main(int argc, char **argv) {
        MPI_Init(&argc, &argv);
        int rango, numProcesos;
        MPI_Comm_rank(MPI_COMM_WORLD, &rango);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcesos);

        Estado estadoInicial = {{0, 8, 7, 6, 5, 4, 3, 2, 1}};

        if (rango == 0) {
            MAESTRO(estadoInicial, numProcesos);
        } else {
            ESCLAVO();
        }

        MPI_Finalize();
        return 0;
    }
```

Listing 13: Implementación paralela del BFS 8-puzzle con MPI

B. Anexo: Versión Secuencial BFS (8-Puzzle)

```

#include <iostream>
#include <queue>
#include <unordered_set>
#include <vector>
#include <string>
#include <chrono>
using namespace std;

struct Estado {
    int celdas[9];
};

string estadoACadena(const Estado &est) {
    string cadena;
    for (int i = 0; i < 9; i++) {
        cadena += to_string(est.celdas[i]) + ",";
    }
    return cadena;
}

bool esObjetivo(const Estado &est) {
    Estado meta = {{1,2,3,4,5,6,7,8,0}};
    for (int i = 0; i < 9; i++) {
```

```
25          if (est.celdas[i] != meta.celdas[i]) return false;
26      }
27      return true;
28  }
29
30  vector<Estado> obtenerSucesores(const Estado &est) {
31      vector<Estado> sucesores;
32      int posCero = 0;
33      for (int i = 0; i < 9; i++) {
34          if (est.celdas[i] == 0) {
35              posCero = i;
36              break;
37          }
38      }
39      int fila = posCero / 3;
40      int col = posCero % 3;
41      int movimientos[4][2] = {{-1,0},{1,0},{0,-1},{0,1}};
42      for (auto &mov : movimientos) {
43          int nf = fila + mov[0];
44          int nc = col + mov[1];
45          if (nf >= 0 && nf < 3 && nc >= 0 && nc < 3) {
46              int nuevaPos = nf*3 + nc;
47              Estado nuevo = est;
48              swap(nuevo.celdas[posCero], nuevo.celdas[nuevaPos]);
49              sucesores.push_back(nuevo);
50          }
51      }
52      return sucesores;
53  }
54
55  struct Nodo {
56      Estado est;
57      int profundidad;
58  };
59
60  int main() {
61      Estado inicial = {{0,8,7,6,5,4,3,2,1}};
62      long long generados = 0;
63      long long expandidos = 0;
64      long long maximoAlmacenado = 0;
65      int profundidadMax = 0;
66      int costoSolucion = -1;
67
68      auto inicio = chrono::high_resolution_clock::now();
69
70      queue<Nodo> frontera;
71      unordered_set<string> visitados;
72
73      frontera.push({inicial, 0});
74      visitados.insert(estadoACadena(inicial));
75      maximoAlmacenado = frontera.size();
76      bool encontrado = false;
77
78      while (!frontera.empty() && !encontrado) {
79          Nodo actual = frontera.front();
80          frontera.pop();
81          expandidos++;
82          if (esObjetivo(actual.est)) {
```

```
            encontrado = true;
            costoSolucion = actual.profundidad;
            break;
        }
        vector<Estado> sucesores = obtenerSucesores(actual.est);
        generados += sucesores.size();
        for (auto &suc : sucesores) {
            string clave = estadoACadena(suc);
            if (visitados.find(clave) == visitados.end()) {
                visitados.insert(clave);
                frontera.push({suc, actual.profundidad+1});
                if (actual.profundidad+1 > profundidadMax)
                    profundidadMax = actual.profundidad+1;
            }
        }
        if ((long long)frontera.size() > maximoAlmacenado)
            maximoAlmacenado = frontera.size();
    }

    auto fin = chrono::high_resolution_clock::now();
    double tiempo = chrono::duration<double>(fin - inicio).count();

    cout << "# estrategia generados expandidos maximo_almacenado costo
       max_profundidad tiempo\n";
    cout << "BFS ";
    cout << generados << " "
         << expandidos << " "
         << maximoAlmacenado << " "
         << costoSolucion << " "
         << profundidadMax << " "
         << tiempo << "\n";

    cout << "Estado inicial: ";
    for (int i = 0; i < 9; i++) cout << inicial.celdas[i] << " ";
    cout << "\n";
    if (encontrado) {
        cout << "Solucion con coste = " << costoSolucion << "\n";
    } else {
        cout << "No se encontro solucion.\n";
    }
    return 0;
}
```