Programación de Sistemas Distribuidos

Curso 2024/2025

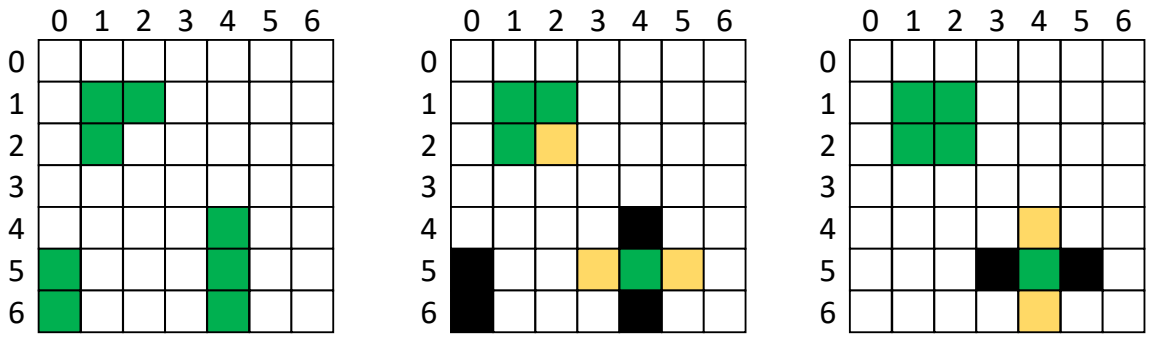
Práctica 3

Paralelizando “el Juego de la Vida” utilizando MPI

Esta práctica consiste en paralelizar la implementación de "El juego de la vida", diseñado por el matemático británico John Horton Conway en 1970. Esencialmente, se trata de un juego de 0 jugadores donde existe un tablero dividido en cuadrados. En cada cuadrado puede existir, o no, una célula, de forma que el estado de un cuadrado puede ser *estar vacío* o *contener una célula viva*. La idea del juego consiste en calcular los estados del tablero aplicando reglas sencillas. Estas reglas son:

- Si un cuadrado vacío (sin célula) tiene exactamente tres células vivas a distancia 1 – también consideradas vecinas – generará una célula viva en el siguiente estado.
- Si un cuadrado tiene una célula viva, y ésta tiene – exactamente – 2 o 3 células vivas vecinas (a distancia 1), entonces esta célula seguirá viva. En otro caso, la célula muere, dejando el cuadrado vacío.

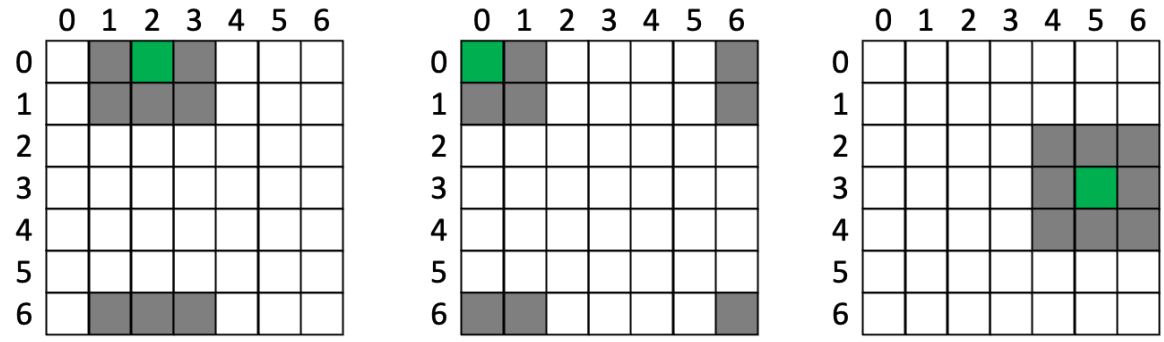
El siguiente ejemplo muestra tres iteraciones de un tablero de 7 filas y 7 columnas. En este caso, representamos los cuadrados vacíos en blanco. Los cuadrados verdes representan una célula viva, los cuadrados negros representan que en la siguiente iteración esa célula se elimina, ya que ha muerto, y los cuadrados amarillos representan que ha nacido una célula nueva.



Para realizar esta práctica, partimos del código del juego ya implementado en modo secuencial, esto es, la ejecución se lleva a cabo con un único proceso. El objetivo de esta práctica consiste en distribuir el cálculo entre las CPUs de un sistema distribuido, de forma que cada proceso calcula una porción del tablero, aumentando el rendimiento de la aplicación. Podéis encontrar el código de la implementación secuencial en el C.V.

Con el fin de observar la evolución del juego, se ha utilizado SDL para dibujar gráficamente el tablero. Como código de partida para realizar esta práctica, se proporcionan varios ficheros con algunos subprogramas ya implementados. Es importante remarcar que todo el código necesario para pintar el tablero se proporciona en los ficheros iniciales de la práctica.

Como particularidad, simularemos un tablero proyectado sobre una superficie toroidal, es decir, haremos que la primera y última columna sean adyacentes, así como la primera y última fila, de forma que habrá que tener esto en cuenta para calcular las células vecinas. La siguiente figura muestra tres ejemplos de los cuadrados vecinos – en gris – de uno que contiene una célula viva – en color verde – utilizando la representación descrita para el tablero.



El fichero `lifeGame.c` contiene el punto de entrada (*main*) al programa. En la función `main` se realiza la lectura de los parámetros de entrada y se inicializa el entorno SDL. La ejecución del programa requiere los siguientes argumentos de entrada:

```
lifeGame worldWidth worldHeight iterations [step|auto] outputImage [static|dynamic grainSize]
```

donde `lifeGame` es el ejecutable, `worldWidth` es el ancho del tablero (en cuadrados), `worldHeight` es el alto del tablero (en cuadrados), `iterations` es el número de iteraciones que se calcularán en la ejecución, `step|auto` es el modo de ejecución, de forma que `step` representa la ejecución paso a paso (el usuario deberá pulsar la tecla *Enter* para comenzar la siguiente iteración), mientras que `auto` realiza una ejecución continuada sin requerir la intervención del usuario, `outputImage` es el nombre del fichero donde se guardará el resultado de la última iteración (en una imagen BMP) y, finalmente, `static` representa una distribución estática, mientras que `dynamic` representa una distribución dinámica con un tamaño de grano `grainSize` filas.

El fichero `types.h` contiene algunas constantes y estructuras que se utilizarán en el resto del programa. Por ejemplo, `SEED` representa la semilla utilizada para inicializar el generador de números aleatorios, `MATRIX_SIZE` define la cantidad de cómputo a realizar por los procesos *worker* al encontrar un cuadrado **sin células vecinas vivas**, `INITIAL_CELLS_PERCENTAGE` es el porcentaje de células creadas en el estado inicial del tablero (iteración 0), y `CELL_SIZE` es el tamaño – en pixels – que tendrá cada cuadrado cuando se pinte.

El estado de las células se representa con las siguientes constantes:

- `CELL_LIVE` representa una célula que estaba viva en la iteración anterior y sigue viva en la iteración actual.
- `CELL_NEW` representa una célula que nace.
- `CELL_DEAD` representa una célula que muere.
- `CELL_EMPTY` representa un cuadrado vacío.

La estructura `tCoordinate` se utiliza para representar una coordenada en el tablero, de forma que el campo `row` hace referencia a la fila, mientras que `col` hace referencia a la columna. Finalmente, utilizaremos la constante `MASTER` para identificar al proceso *master*, es decir, el proceso con *rank* 0, y `END_PROCESSING` como *flag* que indica el fin de ejecución a los procesos *worker*.

La siguiente porción de código muestra la inicialización del entorno SDL. Para esta práctica utilizaremos dos componentes: `window` y `renderer`. El primero representa la ventana donde se mostrará visualmente el tablero, mientras que `renderer` es donde se realizará el pintado del mismo.

```
// The window to display the cells
SDL_Window* window = NULL;

// The window renderer
SDL_Renderer* renderer = NULL;

…

// Init video mode
if(SDL_Init(SDL_INIT_VIDEO) < 0){
        showError ("Error initializing SDL\n");
        exit (0);
}

// Create window
window = SDL_CreateWindow( "Práctica 3 de PSD", 0, 0, worldWidth * CELL_SIZE,
                              worldHeight * CELL_SIZE, SDL_WINDOW_SHOWN);

// Check if the window has been successfully created
If (window == NULL ){
        showError( "Window could not be created!\n" );
        exit (0);
}

// Create a renderer
renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
```

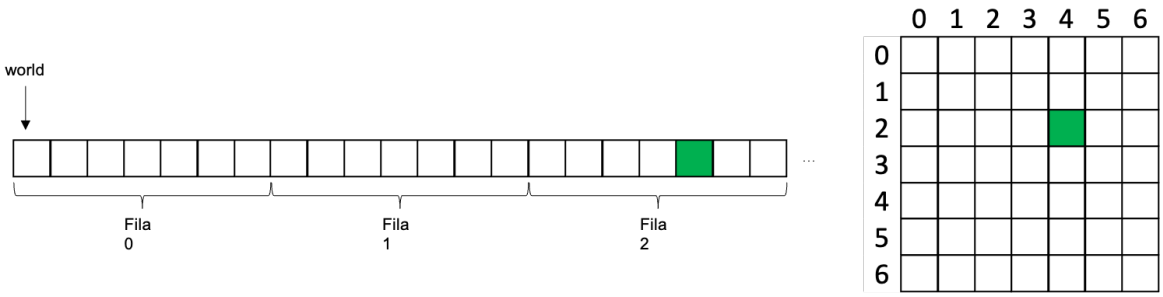
Durante la ejecución del programa podemos distinguir dos tipos de procesos. Tendremos un proceso *master*, el cual se encargará de distribuir los datos y realizar el pintado del tablero, y los procesos *worker*, que se encargarán de calcular el estado siguiente del tablero. Tened en cuenta que en el algún punto de la función `main`, se deberá invocar a los subprogramas que contengan el comportamiento del proceso *master* y de los procesos *worker*. Además, se deberá pasar, entre otros parámetros, `window` y `renderer` al subprograma que contenga la lógica del proceso *master*.

En esta práctica, representaremos un tablero con un array de `unsigned short`. Tened en cuenta que utilizaremos un array de una dimensión y, por ello, para acceder a cada posición del tablero, deberemos calcular el desplazamiento de las filas previas.

Por ejemplo, el siguiente código reserva memoria para un tablero (`world`) y lo inicializa.

```
unsigned short * world;
world = (unsigned short*) malloc (worldWidth * worldHeight * sizeof (unsigned short));
initRandomWorld (world, worldWidth, worldHeight);
```

Si quisiéramos acceder al cuadrado en la fila 2, columna 4, tendríamos que calcular el desplazamiento de dos filas completas, para luego acceder a la columna 4 de la fila 2. La siguiente imagen muestra cómo se almacenan en memoria los elementos del tablero (parte izquierda) y la representación gráfica que mostraremos por pantalla (parte derecha).



Para acceder a la casilla (2,4) podremos acceder de la siguiente forma:

```
world[(2*worldWidth)+4]
```

El fichero `world.h` contiene las cabeceras de los subprogramas utilizados para gestionar los tableros. Por ejemplo, las siguientes funciones calculan la coordenada de las células situadas arriba, abajo, a la izquierda y a la derecha de la coordenada pasada como primer parámetro. El valor devuelto (en el parámetro de salida) es la coordenada destino, el cual puede utilizarse posteriormente para obtener el valor de cada cuadrado del tablero.

```
void getCellUp (tCoordinate* c, tCoordinate* destCell);
void getCellDown (tCoordinate* c, tCoordinate* destCell);
void getCellLeft (tCoordinate* c, int worldWidth, tCoordinate* destCell);
void getCellRight (tCoordinate* c, int worldWidth, tCoordinate* destCell);
```

Los subprogramas `getCellAtWorld` y `setCellAt` devuelven y establecen, respectivamente, el valor de una célula en la coordenada pasada como parámetro:

```
unsigned short int getCellAtWorld (tCoordinate* c, unsigned short* world,
                                   int worldWidth);

void setCellAt (tCoordinate* c, unsigned short* world, int worldWidth,
                unsigned short int type);
```

Para crear un tablero de forma aleatoria se puede utilizar el subprograma `initRandomWorld`, el cual utilizará la constante `INITIAL_CELLS_PERCENTAGE` para crear células en un tablero de forma aleatoria. De forma similar, se puede utilizar el subprograma `clearWorld` para vaciar un tablero, dejando todos sus cuadrados vacíos.

```
void initRandomWorld (unsigned short *w, int worldWidth, int worldHeight);

void clearWorld (unsigned short *w, int worldWidth, int worldHeight);
```

Finalmente, el subprograma `calculateLonelyCell` deberá invocarse cuando se encuentre una casilla que **no tenga ninguna célula viva a distancia 1**. Esencialmente, este programa realiza la multiplicación de dos matrices cuadradas de dimensión `MATRIX_SIZE`.

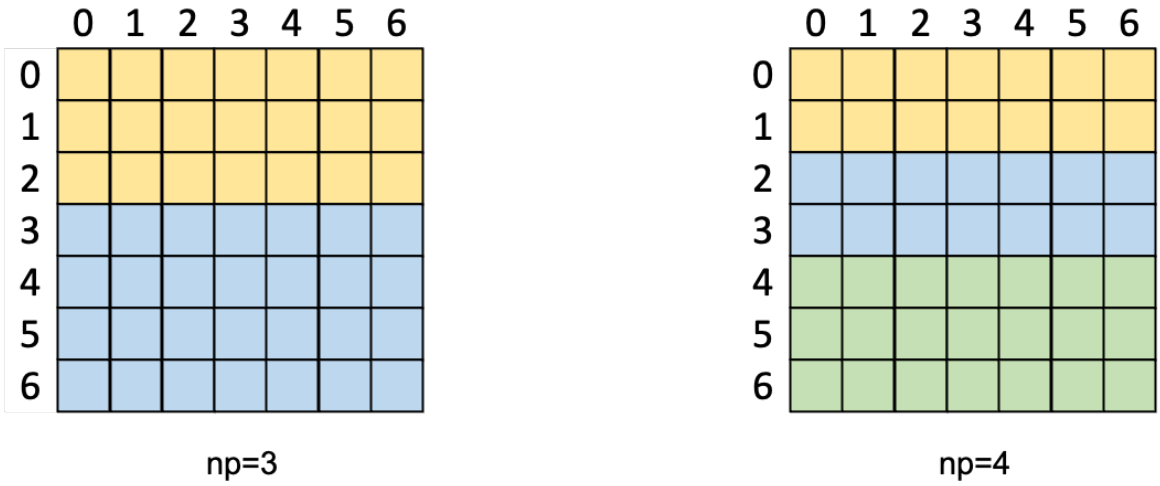
```
void calculateLonelyCell ();
```

A diferencia del programa secuencial, donde todo el procesamiento lo realiza un único proceso, en esta práctica vamos a repartir las tareas entre el proceso *master* y los procesos *worker*. En particular, para una ejecución de *n* procesos, tendremos siempre un proceso *master* y *n-1* procesos *worker*. En cualquier caso, el número mínimo de procesos con los que se puede ejecutar la aplicación es 3.

Las tareas que realizará el proceso *master* son, esencialmente, repartir la carga de trabajo a los procesos *worker* y pintar el tablero. En ningún caso este proceso realizará ningún cálculo relativo al estado del tablero. Sí podrá, por ejemplo, crear el primer estado del tablero, generado de forma aleatoria mediante el subprograma `initRandomWorld`.

Para cada iteración sobre el estado del tablero, los procesos *worker* realizarán el cálculo del siguiente estado, mandarán el resultado al proceso *master*, y será éste el encargado de pintar el tablero en pantalla. La ejecución del programa finaliza cuando se han calculado `iterations` iteraciones.

El proceso *master* podrá distribuir la carga de dos formas distintas, estática y dinámicamente. En la distribución estática, la cantidad de datos a procesar por cada proceso *worker* – en cada iteración – se calcula antes de enviar los mismos. De esta forma, no se tiene en cuenta la velocidad del procesador donde se ejecuta cada proceso. Por ejemplo, para una ejecución con 3 y 4 procesos, un reparto de carga estático puede ser el que se muestra en la siguiente figura.



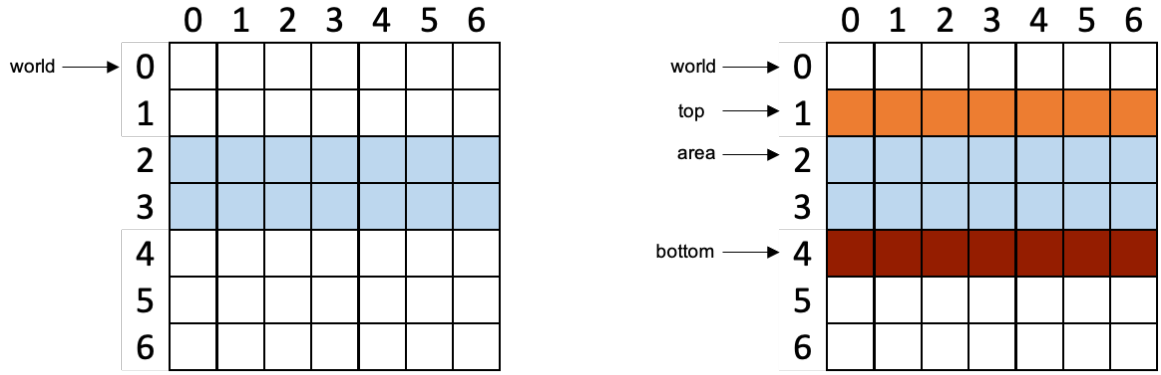
Tened en cuenta que, al ejecutar el programa con 3 procesos, sólo dos de ellos serán procesos *worker*. De esta forma, las tres primeras filas – resaltadas en amarillo – se asignan al primer *worker*, mientras que las cuatro siguientes – resaltadas en azul – se asignan al segundo *worker*. En el siguiente caso (np=4), el reparto cambia, asignando dos filas a los dos primeros procesos *worker* y tres al último. Es importante remarcar que el reparto de carga debe realizarse de forma equitativa entre los procesos *worker*.

Para poder asignar la carga de trabajo de forma dinámica, es necesario establecer el tamaño de grano, o lo que es lo mismo, el tamaño de datos máximo que cada proceso *worker* recibirá para realizar el cálculo del siguiente estado del tablero. Por ejemplo, el grano podrá hacer referencia al número de filas que un proceso deberá procesar, cada vez que el proceso *master* le asigne una porción del tablero. Consecuentemente, cada proceso *worker* puede procesar más de una porción del mismo.

De esta forma, cada vez que un proceso *worker* haya finalizado el procesado de una determinada porción del tablero, el proceso *master* le podrá asignar una nueva porción que aún no haya sido procesada. Así, la carga se distribuye de forma dinámica, pudiendo asignar más cantidad de trabajo a los procesos ejecutados en las máquinas con mayor capacidad de cómputo.

Se tendrán en cuenta todas aquellas consideraciones que tengan un impacto positivo en el rendimiento del programa.

Para realizar el envío de las filas desde el proceso *master* a los procesos *worker* tendremos que utilizar aritmética de punteros. Además, hay que tener en cuenta que, para procesar un área determinada del tablero, nos harán falta también la fila anterior y siguiente de esa zona. La siguiente figura muestra un ejemplo.



Suponed que tenemos el estado del tablero de juego en una variable llamada `world`. El área del tablero que queremos enviar al proceso *worker* está resaltada en azul. En este caso, *grainseSize*=2. Sin embargo, para poder procesar esa zona del tablero, debemos enviar, además, las filas 1 (naranja) y 4 (marrón) al proceso *worker*. Para ello, podemos utilizar punteros auxiliares, de forma que podamos desplazarlos tomando como punto de partida la variable `world`. En este ejemplo, los punteros los podemos asignar de la siguiente forma:

```
top = world + (worldWidth*1);
area = world + (worldWidth*2);
bottom = world + (worldWidth*4);
```

El proceso *master* podrá realizar el pintado del tablero mediante el subprograma `drawWorld`:

```
void drawWorld (unsigned short *currentWorld, unsigned short *newWorld,
                SDL_Renderer* renderer,
                int firstRow, int lastRow,
                int worldWidth, int worldHeight);
```

donde `currentWorld` contiene el tablero en su estado actual, `newWorld` contiene el tablero con el siguiente estado, `renderer` es la estructura sobre la cual se realizará el pintado del tablero, `firstRow` y `lastRow` representan la primera y última fila del tablero que se pintará sobre `renderer`, respectivamente, y `worldWidth` y `worldHeight` representan, respectivamente, el ancho y alto del tablero (en cuadrados).

Además, se deberá tener en cuenta la probabilidad `PROB_CATACLYSM` de que ocurra un cataclismo cada `ITER_CATACLYSM` iteraciones. Estas constantes están definidas en el fichero `master.h`. Si en una determinada iteración se genera un cataclismo, éste afectará únicamente a la fila central y a la columna central. Si el mundo tiene un número par de filas y/o columnas, se elegirá la mayor de ellas. Las celdas afectadas deben eliminarse del nuevo estado del mundo, es decir, pasarán a tener el valor `CELL_EMPTY`. Para facilitar el pintado, se puede establecer el valor de estas celdas a `CELL_CATACLYSM` en el estado actual del mundo, de forma que al invocar el subprograma *drawWorld*, éste pueda pintar las celdas afectadas por el cataclismo de color azul.

Adicionalmente, se puede utilizar el subprograma `saveImage` para guardar el estado del tablero en una imagen con formato BMP,

```
void saveImage (SDL_Renderer* renderer, char* fileName,
                int screenWidth, int screenHeight);
```

donde `renderer` es la estructura sobre la cual se realizará el pintado del tablero, `fileName` es el nombre del fichero de la imagen BMP, y `screenWidth` y `screenHeight` representan, respectivamente, el ancho y alto de la imagen en pixels.

En la representación gráfica se utilizan distintos colores para representar el estado de las células. En particular, las células que nacen en la iteración actual se representan en verde lima, las células que están vivas en el estado actual y permanecen vivas en el estado siguiente se representan en verde oscuro, y las células que mueren, en rojo. Los cuadrados vacíos se muestran en negro.

En el proceso *master* deberán invocarse varias llamadas a la biblioteca SDL para realizar el pintado del tablero. Tal y como se realiza en el programa secuencial, este proceso deberá invocar las llamadas `SDL_SetRenderDrawColor` y `SDL_RenderClear` para limpiar el `renderer` en cada iteración. De esta forma, todos los pixels se inicializarán con el color negro. El siguiente código muestra cómo deben invocarse.

```
SDL_SetRenderDrawColor(renderer, 0x00, 0x00, 0x00, 0x00);
SDL_RenderClear(renderer);
```

De forma similar, para mostrar en la ventana el estado actual del tablero, se deberán invocar las llamadas `SDL_RenderPresent` **y** `SDL_UpdateWindowSurface`.

```
SDL_RenderPresent(renderer);
SDL_UpdateWindowSurface(window);
```

Consideraciones de implementación:

- No se permitirá, en ningún caso, el envío completo del tablero a los procesos *worker*.
- El único proceso encargado de realizar el pintado del tablero y de generar la imagen final es el proceso *master*.
- Las estructuras `window` y `renderer` no pueden enviarse a los procesos *worker*.
- Cada proceso *worker* que encuentre una casilla con 0 células vecinas, deberá invocar el subprograma `calculateLonelyCell`.
- Se tendrá en cuenta la eficiencia de los algoritmos desarrollados, por ejemplo, evitando copias innecesarias de zonas del tablero.

Ficheros a entregar

Esta práctica puede desarrollarse utilizando un código fuente inicial, el cual contiene parte de la implementación y los tipos de datos utilizados. Este código fuente se encuentra en el fichero `PSD_Prac3_MPI.zip`.

Es importante matizar que el fichero .zip entregado debe contener los ficheros necesarios para realizar la compilación del programa, el cual debe poder ejecutarse utilizando una distribución estática y dinámica. En caso de que cualquiera de las partes entregadas no compile, se tendrá en cuenta la penalización correspondiente.

Entrega de la práctica

Para entregar esta práctica se habilitará un entregador en la página de la asignatura del Campus Virtual. La fecha límite para entregarla será el **día 10 de diciembre de 2024 a las 17:55.**

No se permitirá la entrega de prácticas fuera del plazo establecido.

La entrega se realizará mediante un **único fichero con extensión .zip**, el cual deberá incluir los ficheros necesarios para ejecutar y compilar la aplicación pedida, además del fichero `nombres.txt`, que contendrá el nombre y apellidos de los integrantes del grupo.

Se van a perseguir las copias y plagios de prácticas, aplicando con rigor la normativa vigente.