Informe de S.O.

Agustín Gurvich Santiago Wirzt

30/04/2020

1 Cambios al código provisto

En la función congwayGoL realizamos un cambio en el cual pasa a recibir un game_t* y la cantidad de hilos a crear, y ahora no tiene valor de retorno pues al estar modificando el tablero usando un puntero ya tenemos el cambio guardado en memoria.

2 Idea de la resolución

Nuestra idea para crear el simulador fue comenzar planteando una solución lineal. Para ello definimos nuestro propio tablero con la estructura board_t:

```
typedef struct board_ {
  tupla** tab;
  size_t n;  // Columnas
  size_t m;  // Filas
} * board_t;
```

Donde tupla es una estructura que tiene la forma:

```
typedef struct _tupla {
  char estado;
  int futuro;
} tupla;
```

Que contiene un caracter indicando el estado de la celda (ALIVE o DEAD) y un entero (1 o 0) que indica si la celda tiene que cambiar su estado en la siguiente iteración o no.

La idea es recorrer el tablero y por cada celda contar la cantidad de vecinos. Luego chequear, según el estado de la celda actual, qué condición cumple y y así indicar si deberá cambiar la celda en la próxima iteración o no. Después realiza otra vuelta en la cual revisa en cada celda el valor que almacena su futuro, y de ser necesario cambia el valor de la celda y asigna 0 a futuro.

Finalmente, mientras no se hayan realizado la cantidad de ciclos indicados, se repite el proceso.

3 Solución con hilos

Para resolver el ejercicio utilizando hilos, decidimos dividir el tablero en secciones y otorgarle una a cada hilo. Para marcar los límites de las secciones utilizamos una estructura intervalo que tiene la forma:

```
typedef struct intervalo_ {
  size_t inicio;
  size_t fin;
} intervalo;
```

La cual almacena en inicio el inicio del intervalo, y en fin su final.

En la función main calculamos la cantidad de hilos utilizando la función get_nuprocs() y a partir de ello creamos las zonas de la siguiente forma:

1. Decidimos cuál es la dimensión más grande para dividir eficientemente el tablero
2. Llamamos a la función dividir() que genera los intervalos
3. Reparte los intervalos en los tablero_h correspondientes
4. Devuelve la lista de tableros_h para ser pasada a los hilos

En este proceso, la función dividir funciona de la siguiente manera:

1. Decide un tamaño inicial de intervalos que se calcula como $tam = max/procs$ donde max es el tamaño de la dimensión máxima y procs es la cantidad de hilos. Notamos que esta es una división entera por lo tanto puede no ser exacta.
2. Calcula el resto de la división $sobra = max\%procs$.
3. Por cada intervalo a llenar, si $sobra \neq 0$ al final del intervalo le agrego una unidad. Si no sobra nada, entonces lleno normalmente.

Esto nos asegura que cada hilo tendrá a lo sumo una fila(o columna) extra.
Por otro lado, la estructura tableros_h tiene la forma:

```
typedef struct tablero_hilo {
  tupla** tab;
  intervalo intN;  // Intervalo de columnas
  intervalo intM;  // Intervalo de filas
  unsigned int ciclos;
  size_t m;
  size_t n;
} * tablero_h;
```

Destacamos que la definición de la estructura ya se crea como un puntero, pues nos pareció apropiado ya que en todo momento se trata como puntero en las funciones. Donde cada parte cumple una función particular:

- tab es el tablero que estamos utilizando
- intN es el intervalo de columnas a recorrer
- intM es el intervalo de filas a recorrer
- ciclos es la cantidad de iteraciones del juego a ejecutar
- m es la cantidad de filas
- n es la cantidad de columnas

Con esto ya listo podemos ejecutar cada hilo por separado utilizando la función hiloworker, que recibirá un tablero_h y dentro de los límites que se le indican realizará los siguientes pasos:

1. Chequea en su sección qué debería sucederle a cada celda
2. Espera en una barrera que todos los hilos ya hayan chequeado sus celdas
3. Recorre su sección modificando las celdas correspondientes
4. Espera en una barrera que todos los hilos ya hayan modificado sus celdas
5. Si aún quedan iteraciones por hacer, se repiten los pasos anteriores

4 ¿Se presentan problemas de concurrencia?

Utilizando esta implementación no se pueden generar problemas de concurrencia.

Por un lado, cuando realizamos el primer recorrido para chequear el futuro de cada celda no realizamos asignaciones en otros lugares por lo cual no hay posibilidad de problemas de escritura. Tampoco estamos repartiendo recursos ni esperando que se cumplan ciertas condiciones durante el primer chequeo.

Por otro lado, al realizar las modificaciones de cada tupla solamente se tocan las celdas correspondientes a cada zona indicada en tablero_h, las cuales nos aseguramos que no se superponen. Así no hay problemas ni de escritura ni de reparto de recursos, por lo que evitamos los deadlocks completamente.

Entre cada sección del hilo, tenemos una barrera que lo detiene y espera que todos los hilos lleguen a ese punto para poder estar sincronizados y evitar inconsistencias (por ejemplo, que un hilo cambie una celda que otro hilo esta chequeando).