Paralelismo del Juego de la vida de Conway

Franco Ignacio Vallejos Vigier y Facundo Emmanuel Messulam

April 19, 2021

Abstract

El problema de paralelismo suele ser muy complejo en situaciones donde se debe obtener información creada en otro hilo. Para solucionar este tipo de problemas proponemos un sistema de manejo de información donde están claramente indicadas las secciones de memoria que fueron actualizadas, esto permite trabajar en paralelo, simplificar el código y reducir costos en términos de tiempos de espera a otros hilos.
Usamos el juego de la vida de Conway para mostrar este sistema en una aplicación que requiere tener una copia anterior del tablero para pasar al siguiente ciclo. Nuestro algoritmo mejora el estado del arte actual sobre tableros densamente poblados por células vivas. Mostramos que permitiendo el desfase entre los hilos, es decir, cuando los hilos pueden trabajar sobre diferentes ciclos, se puede lograr mayor eficiencia del uso del tiempo de computo, logrando menores tiempos de ejecución.

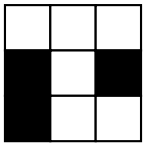
1 Introducción

1.1 Juego de la vida

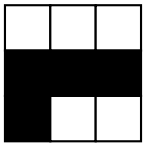
El juego de la vida toma un tablero de celdas llamadas "celulas", cada célula puede estar viva o muerta. Las células pasan a estar vivas o muertas de acuerdo a una serie de reglas y un estado anterior. Las reglas son las siguientes:

1. Toda célula viva con 2 o 3 vecinos vivos sobrevive
2. Toda célula muerta con exactamente con 3 vecinos vivo revive
3. El resto de las células vivas mueren, como a su vez el resto de las células muertas se mantienen muertas

Es importante remarcar que cada célula esta determinada en el ciclo siguiente solo por las 8 células aledañas:



Produce:



La célula central va a estar viva en este ejemplo, sin importar lo que rodee a estas 9 celdas dibujadas.

1.2 Consideraciones de trabajo

Proponemos una solución basada únicamente en unidades de procesamiento generales, CPU, y no CUDA o TPU. Tampoco hacemos uso de mecanismos automáticos de paralelización como OpenMP.
Usamos como referencia un procesador Ryzen 3200G, con 4 núcleos y 4 hilos, con una linea de cache L1 de 64bytes (`LEVEL1_DCACHE_LINESIZE`).

2 Implementación del tablero y el juego

En una primera instancia uno puede pensar que el problema se puede implementar con otro algoritmo, que no sigue las reglas directamente, pero es más óptimo y permite calcular el tablero final más rápido. Sin embargo, el problema de computar un tablero final a partir de uno inicial es NP-completo (Sutner 1994). Como vamos a ver, decidimos tomar una serie de operaciones que son más rápidas que una implementación ingenua.

2.1 Implementación del tablero

La implementación del tablero esta dada por `Board.h`. Creamos 2 estructuras: ellas son `board_t` y `board_subdivider_t`:

- La primera se encarga de representar a la tabla mediante el número de columnas y filas acompañado por un `atomic_u_int8_t` array siendo este la representación de los estados de cada célula.

- La segunda estructura hace referencia a porciones de la tabla original a la cual fraccionamos para paralelizar su cálculo de manera cooperativa. Para ello a cada porción la represento por sus límites superior, inferior y laterales, izquierdo y derecho.

En compañía de estas declaraciones, se hacen uso de funciones primitivas para tablas como su inicialización, indexación y destrucción.

2.1.1 ¿Por qué elegimos un atomic_u_int8_t?

Porque objetos de tipo atómico son los únicos que están libres de condición de carrera, es decir, pueden ser modificados por dos hilos de forma concurrente o modificados por uno y leídos por otro. Para ello en nuestro trabajo, se necesita de ciertas garantías del tipo de dato 'atomic':

Coherencia de lectura Si la lectura de un valor A de un objeto atómico M ocurre antes de la lectura de un valor B de M, se mantiene el orden de lectura leyendo siempre B después que A en M.

Coherencia de escritura Si una operación A que modifica un objeto atómico M ocurre antes que una operación B que modifica M, entonces A aparece antes que B en el orden de modificación de M.

En nuestro trabajo estas invariantes van a ser de suma importancia porque requerimos que lecto-escrituras simultaneas no devuelvan valores incoherentes.

Cada célula en el tablero tiene 3 estados, viva, muerta, y no determinada. El uso de este sistema se hace claro en la sección "Cada sección rectangular un hilo rehusado con desfase permisivo".

2.2 Implementación del juego

Como primera optimización decidimos implementar las reglas como una serie de indexaciones a una tabla. Dada una configuración de bits de la siguiente forma: `000000000`, donde los primeros 3 son la primera fila, los segundos 3 son la segunda fila y los terceros 3 son la ultima fila. Esto forma un entero, entre 0 y $2^9 - 1$ inclusive, que es el índice en la tabla, el resultado de `table[index]` es si la célula central esta viva o muerta en la siguiente iteración. La tabla se crea una solo vez durante la ejecución del programa en `init_table()`. Luego por cada porción de ella vamos aplicando la serie de comparaciones y reemplazándola por su correcta imagen en la serie de indexaciones que representan a las reglas de Conway.

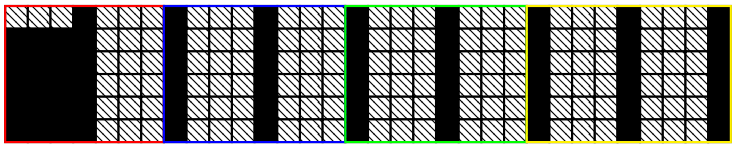
Para la ejecución del juego se crean n hilos, que son explicados en "Implementación del paralelismo".

3 Implementación del paralelismo

3.1 Manejo de memoria y compartición

Para la paralelización implementamos un sistema que reparte equitativamente la carga sobre tantos hilos como sea necesario. 'necesario' es, sin embargo, difícil de definir en este caso, ya que hay que hacer uso de tanto tiempo de procesamiento como sea necesario, y a su vez, no caer en los costos agregados de una implementación que sobreparaleliza.

La solución consta de implementar un hilo por cada sección rectangular más grande que una constante `MIN_AREA`, pero sin crear más hilos que unidades de procesamiento paralelo se hayan implementado. Puede verse usamos franjas verticales, y no rectangulos, como propone Gropp. Para hacer esto dividimos el tablero en secciones rectangulares menores (parecido a lo hecho en Gropp 2015):



Sin embargo no utilizamos mecanismos complejos de pasaje de datos entre hilos, como propone Gropp, sino que usamos una sección de memoria que solo se lee y una sección de memoria que solo se escribe. Cuando una iteración está completada, todos los hilos pueden progresar a la siguiente, y además, los hilos solo requieren de la iteración anterior para computar la siguiente, esto permite, que no existan problemas de bloqueo, como deadlock, a pesar de que se usan mecanismos bloqueantes.

La optimización del posicionamiento de cada rectángulo de forma que lineas de cache no se invaliden innecesariamente se deja como trabajo futuro, pero se intento trabajar con una cantidad mínima de 64 bytes de ancho por sección rectangular, lo que debería impedir que las columnas sean mas chicas que una linea de cache de ancho (Drepper 2007). Los efectos de hacer esto se consideran fuera del estudio actual.

3.2 Mediciones

Como métrica de la optimización del paralelismo, se fuerza la cantidad de hilos en los que se ejecuta y usamos el promedio de 10 mediciones de
`/usr/bin/time -f "%P" ./simulador aspas.game` para medición "a)" (ver anexo A) y
`/usr/bin/time -f "%P" ./simulador creciente.game` para medición "b)" (ver anexo B) con una compilación sin optimizaciones:
`gcc main.c Board.c Board.h Game.c Game.h -o simulador -lpthread -lm`.
Es importante no comparar entre programas o entre archivos de entrada, ya que esta métrica depende del tiempo real de ejecución. Esta métrica mide uso del

tiempo de usuario y tiempo de kernel sobre tiempo de reloj, es decir, tiempo usado sobre tiempo haciendo trabajo. En la maquina de testeo usada tiene una cantidad máxima de 400%, 100% por hilo por 4 hilos.

Para medir velocidad absoluta de una ejecución nominal usamos el tiempo de reloj de ejecución del programa, calculando la cantidad de hilos con el sistema explicado en "Manejo de memoria y compartición". Para esto se usaron los comandos `time ./simulador aspas.game` y
`time ./simulador creciente.game` y se tomó el tiempo marcado como "real".

3.3 Tres algoritmos y sus tiempos

3.3.1 Cada sección rectangular un hilo nuevo

Por cada sección rectangular se crea un hilo, que procesa esa sección, cuando el tablero itera al siguiente paso, todos los hilos son destruidos, se espera a que mueran, y luego se empieza con la siguiente iteración.

Resultados:

a) Una ejecución con 1 hilo da un puntaje de 95,5%, mientras que una ejecución con 2 hilos da 145,2% y con 4 hilos da 158,9%. La ejecución del programa con la cantidad de hilos automáticamente determinada tomó 5,510s.

b) Una ejecución con 1 hilo da un puntaje de 98,00% , mientras que una ejecución con 2 hilos da 173,70% y con 4 hilos da 214,70%. La ejecución del programa con la cantidad de hilos automáticamente determinada tomó 29,160s.

3.3.2 Cada sección rectangular un hilo reusado

Por cada sección rectangular se crea un hilo, que procesa esa sección, cuando el tablero itera al siguiente paso, todos los hilos deben llegar a una barrera, y luego se empieza con la siguiente iteración. Esto evita crear y destruir hilos repetidamente.

Resultados:

a) Una ejecución con 1 hilo da un puntaje de 80,60%, mientras que una ejecución con 2 hilos da 109,90% y con 4 hilos da 104,10%. La ejecución del programa con la cantidad de hilos automáticamente determinada tomo 4,217s.

b) Una ejecución con 1 hilo da un puntaje de 98,00%, mientras que una ejecución con 2 hilos da 163,60% y con 4 hilos da 178,30%. La ejecución del programa con la cantidad de hilos automáticamente determinada tomo 16,190s.

3.3.3 Cada sección rectangular un hilo rehusado con desfase permisivo

Un mismo hilo puede seguir calculando su región del tablero en ciclos posteriores aún si los ciclos anteriores no se llegaron a terminar de computar. Este algoritmo opera de la siguiente forma:

1 Cada hilo primero va a calcular lo que se pueda de las fronteras internas de su región izquierda y derecha en ese orden, guardando la posición del primer elemento que necesita información pero todavía no fue computada.

2 Luego calcula del área interna de la sección rectangular

3 Para terminar, computa el resto de las fronteras, de forma bloqueante

Para determinar si una posición dada es computable, las 8 celdas aledañas deben estar determinadas. Esto constituye un evento, análogo a una barrera con broadcast, pero tiene la ventaja de ser más simple de codificar. El paso uno simplemente lee si la celda frontera es computable, en cambio el paso tres lee repetidamente las celdas hasta que están determinadas.

Es importante remarcar que nunca escribe dos veces la misma posición en memoria.

Manejo de memoria Aplicación y uso de una lista simplemente enlazada, con invariantes, para manejar los tableros de diferentes ciclos.

Invariantes:

1 La lista enlazada no puede ser vacía

2 Cada elemento de la lista va a ser liberado N veces, una liberación por hilo

Definimos a la iteración M como aquella en la que los hilos leen sus correspondientes secciones de la tabla M-1 y escriben en la M, por lo tanto la tabla resultado va a ser la última tabla. De esta forma todos los hilos en una iteración M van a escribir su resultado en una tabla M en la lista a partir de la lectura de la formada en el ciclo M-1.
Dado que cada hilo lee una porción de las R determinadas en la tabla, si una tabla es descartada R veces significa que ninguno de los hilos va a hacer uso de la misma por lo que el ciclo M ya terminó de calcularse totalmente y se libera la memoria de la tabla M-1 que está en desuso. Como resultado final en nuestra lista enlazada después de calcular los M ciclos solo vamos a obtener un solo elemento, el final, ya que habremos descartado M-1 tablas anteriores a la final, recordemos que toda tabla con índice menor a M es intermedia a la resultante.

Resultados:

a) Una ejecución con 1 hilo da un puntaje de 99,00%, mientras que una ejecución con 2 hilos da 197,10% y con 4 hilos da 306,40%. La ejecución del programa con la cantidad de hilos automáticamente determinada tomo 2,939s.

b) Una ejecución con 1 hilo da un puntaje de 98,70%, mientras que una ejecución con 2 hilos da 197,40% y con 4 hilos da 322,70%. La ejecución del programa con la cantidad de hilos automáticamente determinada tomo 13,014s.

4 Generalizabilidad

El algoritmo dado puede ser adaptado para procesar autómatas celulares de muchos tipos, estos se usan para procesar simulaciones de arena y similares (Bittker 2019), estas se usan en juegos, que comúnmente se ejecutan en múltiples hilos y son sensibles a el tiempo de ejecución de simulaciones físicas.

References

[Sut94] Klaus Sutner. "On The Computational Complexity of Finite Cellular Automata". In: *Journal of Computer and System Sciences 1995* 50 (1994), pp. 87–97.

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*. `https://www.akkadia.org/drepper/cpumemory.pdf`. 2007.

[MCM12] Longfei Ma, Xue Chen, and Zhouxiang Meng. *A performance Analysis of the Game of Life based on parallel algorithm*. 2012. arXiv: `1209.4408 [cs.DC]`.

[Gro15] William D. Gropp. *Mlife2d – a simple, parallel, implementation of Conway's "Game of Life"*. 2015.

[Bit19] Max Bittker. *Making Sandspiel*. `https://maxbittker.com/making-sandspiel`. 2019.

5 Anexo A

Archivo `aspas.game`:

```
50000 25 25
25X
2X1O22X
2X1O22X
2X1O22X
25X
25X
25X
25X
22X1O2X
22X1O2X
22X1O2X
25X
25X
25X
25X
25X
10X1O14X
10X1O14X
10X1O14X
25X
25X
25X
25X
25X
25X
```

6 Anexo B

Archivo `creciente.game`:

Este archivo es demasiado largo, se provee la linea 250, el resto son `500X`.

```
1000 500 500
500X
...
500X
230X801X503X306X701X50231X
500X
...
500X
```