Conway's Game Of Life

Garavano Lautaro - Regolo Giuliano

Abril 2021

1 Introducción

En este trabajo realizamos una implementación en c del juego de la vida de Conway, optimizándola utilizando paralelismo. Describiremos en este informe el algoritmo implementado para realizarlo, así como las dificultades y oportunidades de optimización que nos encontramos durante el desarrollo del mismo. Adjunto a este informe se encuentra el código fuente del programa, el cual se puede compilar y ejecutar siguiendo las indicaciones a continuación.

2 Aclaraciones del programa

2.1 Compilación

Para compilar el programa, contamos con un archivo `makefile`. De esta manera, tan solo llamar al comando `make` creará en el directorio raíz un archivo `main.exe`. De encontrarse una versión antigua o simplemente de querer volver a generar este archivo, se puede utilizar el comando `make clean`.

2.2 Ejecución

Para ejecutar el programa, se ejecuta el archivo `main.exe`. Este archivo toma como argumentos los nombres de los archivo de entrada y de salida. De no ser especificados, se tomará como entrada el archivo `Ejemplo.game` y como salida `Salida.out`.

Los archivos de entrada deberán tener el siguiente formato:

- 3 dígitos en una misma linea, que serán el número de **ciclos**, **columnas**, y **filas**, en ese orden.
- Una cantidad de líneas igual a la cantidad de filas, que contengan el tablero de entrada, representado en formato RLE.

Los archivos de salida, por su parte, mostrarán el tablero resultante codificado utilizando también RLE.

3 Algoritmo

Idear un algoritmo para simular una iteración del juego de la vida de Conway es relativamente simple sin paralelismo. Una primer implementación (errónea) del algoritmo se movería celda por celda, contaría los vecinos vivos a su alrededor, y actualizaría su estado, repitiendo este proceso para cada celda. Esto no funciona, debido a que al actualizar una celda c_{ij} perdemos la información del estado que tenía anteriormente. Esto hace que las celdas que están alrededor de ella calculen su cantidad de vecinos erróneamente. La solución a este problema es crear una nueva tabla "borrador" en la que se escriben los estados actualizados.

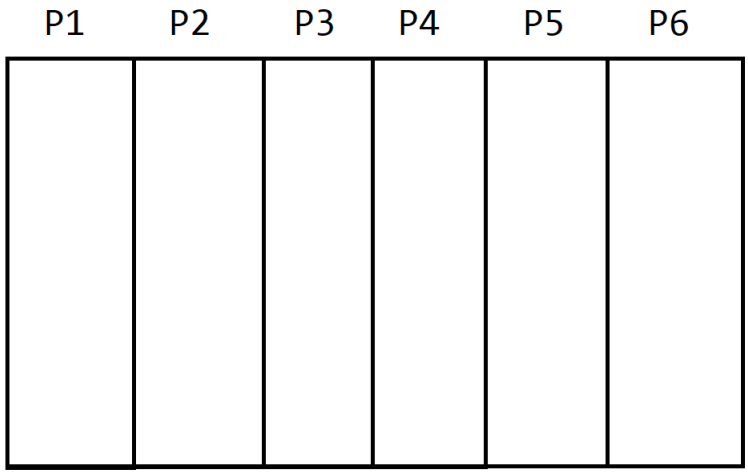
Al terminar de iterar por toda la tabla, se copian los datos de la tabla borrador a la tabla original, o directamente se devuelve la tabla borrador.

Este algoritmo es también fácilmente paralelizable. Lo que debemos hacer es dividir el tablero en tantas partes como unidades de procesamiento tengamos, y darle a cada proceso una tabla borrador del tamaño de su porción del tablero. Cada proceso escribirá su borrador en base a las celdas del tablero original, al cual todos pueden acceder pero ninguno puede escribir. Cuando un proceso termina su borrador, espera a que el resto de procesos también terminen con los suyos. Una vez que se tienen los borradores, se sobreescribe el tablero con los borradores, y comienza el nuevo ciclo.

Si observamos atentamente el algoritmo anterior, veremos que aún hay lugar para mayor optimización. Si un subproceso p_n terminó de escribir su borrador, y todos los procesos contiguos (cuyas porciones del tablero son limítrofes) terminaron de leer las celdas de p_n que necesitan para actualizar sus celdas, no hay ninguna razón para que p_n no escriba su borrador en la tabla. De esta manera podemos tener varias porciones de la tabla terminando de escribirse simultáneamente.

3.1 Posible mejora (no implementada)

Luego de la implementación del algoritmo descrito, nos encontramos con que había más posibilidad de optimización. La optimización posible aparece cuando consideramos un programa que calcula varios ciclos del juego. Es posible, siendo que hay porciones del tablero que escriben antes que otras, que coordinemos los procesos de manera tal que distintos procesos puedan estar calculando ciclos distintos en un determinado momento. Por ejemplo, si dividimos nuestro tablero de la siguiente forma:



Es posible que los procesos p_2, p_3, y p_4 terminen de escribir antes que el resto. De pasar esto, p_3 podría empezar a calcular el siguiente ciclo, puesto que ya tiene toda la información que necesita.

Esta mejora no fue implementada ya que su implementación requería una reescritura bastante grande del código existente y hubiera tomado demasiado tiempo.

4 Implementación

Pasando en limpio nuestro algoritmo para cada proceso, la lista de pasos a seguir es:

1. Calcular los estados nuevos de nuestras celdas y escribirlos en nuestro borrador
2. Esperar a que los procesos adyacentes hayan terminado de leer las celdas que necesitan de nuestra porción de la tabla.
3. Reemplazar nuestra porción de la tabla con nuestro borrador.

Los desafíos para implementar este algoritmo son:

- ¿Cómo deberíamos dividir la tabla según la cantidad de procesos?
- ¿Cómo coordinamos los procesos de manera que ninguno escriba antes de que sus vecinos puedan leer?

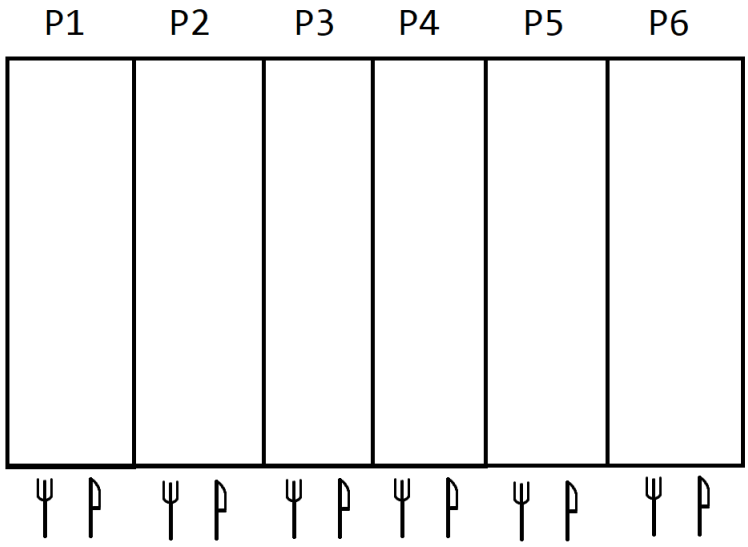
Nuestra implementación es tan sólo una de las formas en las que estas preguntas pueden contestarse. Cuenta con sus fortalezas y debilidades, las cuales ilustraremos a continuación.

Nuestro algoritmo divide el tablero según sus columnas. Para saber cuántas columnas le corresponden a cada proceso, utilizamos el siguiente algoritmo:

- Dividimos la cantidad de columnas por la cantidad de unidades de procesamiento n, redondeando hacia abajo. Esta será la cantidad mínima de columnas asignadas a cada proceso.
- Calculamos el resto de la división. Este número necesariamente será menor que la cantidad de unidades de procesamiento. Esta será la cantidad de procesos a los cuales se les asignará una columna de más.

Este algoritmo divide el tablero en porciones verticales de la manera más equitativa posible. Esta división cuenta con una debilidad, que es que puede ocurrir que la cantidad de columnas sea menor a la cantidad de procesos. En este caso, según nuestra implementación, los procesos a los cuales no se les asigna ninguna columna no correrán. Esto puede ser un problema en un caso en el que tenemos un tablero con muy pocas columnas pero con muchas filas, ya que estamos desaprovechando completamente los núcleos restantes. Esta división del tablero sin embargo, cuenta también con varias ventajas, que veremos más adelante.

Respecto a la sincronización de los procesos a la hora de leer y escribir, nos decidimos por utilizar el siguiente modelo:



Como ilustra el gráfico, podemos ver que a cada proceso le asignamos dos "cubiertos", analogía en honor al programa de la cena de los filósofos. Cada cubierto es un mutex que se utilizará para coordinar cada proceso con sus dos vecinos. A la hora de leer el tablero y escribir el borrador, un proceso toma el cuchillo de su vecino izquierdo, y el tenedor de su vecino derecho. Esto señaliza que el proceso lee datos de esas dos porciones[1]. Una vez que terminó con su borrador, el proceso intenta pasar a la fase de escritura. Para hacer esto, deberá contar con tanto su tenedor como su cuchillo.[2] Esto quiere decir que los procesos que tiene al lado, quienes robaron sus cubiertos para leer sus datos, tienen que habérselos devuelto. De no ser el caso, el proceso se queda esperando. Una vez que cuente con sus cubiertos, el proceso escribe su borrador en el tablero. Luego de escribir queremos que el proceso espere a que se haya terminado el ciclo en todo el tablero, por lo que lo hacemos esperar a los otros procesos. Esto lo logramos mediante la utilización de una barrier.

Una de las dudas que puede surgir al ver este algoritmo es por qué usamos dos mutex para cada proceso en lugar de uno sólo. La respuesta es simple. Cuando el proceso p_3 quiere empezar a generar su borrador, este necesita tomar el mutex de p_2. p_1, por su parte, también necesita tomar este mutex. Sin embargo, no podría hacerlo ya que estaría lockeado por p_3, haciéndolo esperar innecesariamente.

Este algoritmo es relativamente robusto y funciona la mayoría de las veces, pero hay un caso de error que no hemos considerado (el cual arreglamos en la

[1] Aquí se evidencia una de las ventajas de la división en columnas, que es que cada proceso solo tiene 2 vecinos, en vez de si dividiéramos, por ejemplo, en una cuadrícula, donde cada proceso tendría 8 vecinos.

[2] Si no hiciéramos la división en columnas, habría que esperar a muchos más procesos, lo cual podría ser ineficiente

implementación provista). El error es el siguiente: Es posible que p_1 complete su lectura incluso antes de que p_2 haya tenido tiempo para agarrar el cuchillo de p_1. Entonces, p_1 será capaz de escribir antes de que p_2 lea, modificando el resultado de p_2. La solución que elegimos para este problema es colocar otra barrier justo después de cuando cada proceso le roba los cubiertos a sus vecinos. Así nos aseguramos de que este escenario no pueda ocurrir.

5 Bibliografía

r/learnprogramming: Algoritmo para dividir un array equitativamente
qnx.com: Guía para usar pthread_barrier en c
stackoverflow.com: Valgrind detecta erróneamente leaks de memoria
geeksforgeeks.com: Guía sobre pthread
geeksforgeeks.com: Guía sobre mutex