Informe Técnico. Implementación Concurrente del Juego de la Vida

El **Juego de la Vida** es un autómata celular que simula la evolución de un tablero donde cada celda cambia su estado en función de sus vecinas. Su implementación secuencial es simple, pero **ineficiente para tableros grandes** o simulaciones largas.

El objetivo del proyecto fue:

- Acelerar la simulación usando técnicas de programación paralela.
- Aprovechar los núcleos múltiples del procesador (CPU).
- Mantener una interfaz gráfica interactiva fluida.
- Recoger estadísticas detalladas de rendimiento.
- Aplicar los conceptos de sincronización, concurrencia y comunicación interprocesos aprendidos en la asignatura.

2. Solución implementada

División del trabajo (paralelismo por dominio)

Se divide la matriz global de celdas en **bloques horizontales** (filas) y se asigna a múltiples procesos hijos, cada uno encargado de calcular su parte de la generación actual.

filas_por_proceso = FILAS // PROCESOS

Cada proceso ejecuta la función actualizar_bloque, que evalúa las reglas del juego sobre su porción de la matriz.

Compartición de memoria (shared memory)

Para evitar el coste de copiar matrices entre procesos, se usan estructuras compartidas:

multiprocessing.Array('b', FILAS * COLUMNAS)

Se utilizan dos arrays compartidos:

- matriz_compartida: estado actual
- nueva_matriz_compartida: siguiente estado

Se accede a estos como numpy.ndarray usando np.frombuffer(...), lo que permite operaciones vectorizadas sin copias de datos.

Sincronización (Barrier)

Se usa un objeto multiprocessing.Barrier para asegurar que todos los procesos terminen su bloque antes de avanzar a la siguiente generación.

Esto garantiza consistencia y elimina condiciones de carrera.

barrera.wait()

Además, el **proceso con inicio_fila == 0** copia la nueva matriz al buffer principal después de cada iteración

Comunicación interprocesos (Queue)

Cada proceso envía al proceso principal el número de células vivas en su bloque usando una cola:

queue.put(celulas_vivas_bloque)

Esto permite acumular estadísticas sin acceso directo a memoria no sincroniza da

Coordinación con GUI (Tkinter)

El uso del método root.after(...) permite que el bucle de la interfaz gráfica no se bloquee y coopere con la simulación, permitiendo pausas y reanudaciones sin congelar la ventana.

Carga de patrones desde archivo

Se implementó una funcionalidad para cargar patrones personalizados desde archivos externos en formato .npy o .csv. Esto permite:

- Reutilizar configuraciones personalizadas.
- Compartir patrones con otros usuarios.
- Inyectar automáticamente el patrón en el centro del tablero si su tamaño es compatible.

El programa verifica que:

- El patrón contenga solo 0 y 1.
- El tamaño del patrón no exceda el tablero (20x20).
- El patrón se ajuste correctamente al centro.

Además, se informa al usuario del éxito o del error correspondiente mediante cuadros de diálogo (messagebox).

Exportación del patrón actual

Se proporciona también la opción de **guardar el estado actual del tablero** en formato .npy (para uso interno) o .csv (para edición o uso externo). Esto es útil para:

- Documentar configuraciones iniciales.
- Generar datasets de prueba.
- Realizar análisis o validación externa de patrones complejos.

Análisis de paralelismo y eficiencia

Speedup teórico

Con n procesos y una carga de trabajo equilibrada, se espera un speedup lineal ideal:

nginx

Speedup ≈ T_sec / T_paralelo ≈ n (ideal)

En la práctica, hay costes de:

- Sincronización (barrier)
- Comunicación (queue)
- Overhead de multiprocessing

Núcleos (procesos)	Tiempo total aproximado	Speedup aproximado
1 (secuencial)	20s	1.0
2	12s	1.6
4	9s	2.2

4. Generación de resultados

Al finalizar cada simulación, se genera:

- estadisticas.csv: estadísticas de rendimiento por generación.
- estado.npy: estado final del tablero.

Estas métricas permiten analizar:

- Evolución de la complejidad (actividad celular)
- Estabilidad de patrones (convergencia o caos)
- Coste computacional por generación