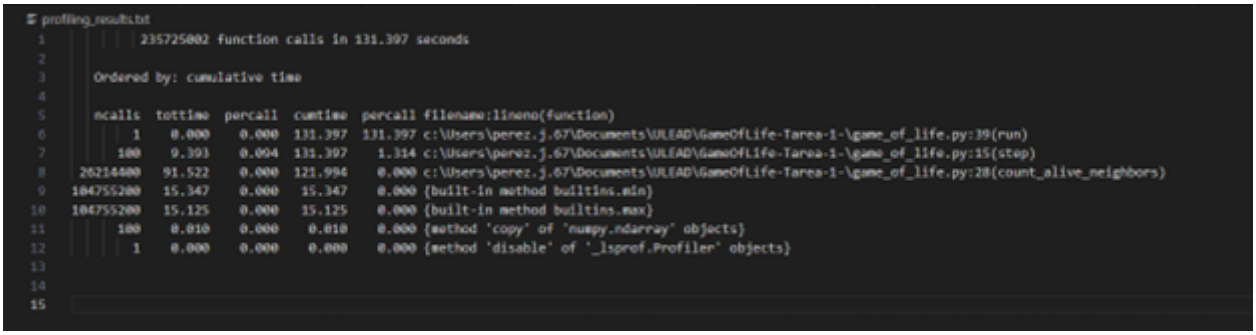
Tarea 2 - Josimark Perez

Capturas de las herramientas de perfilzado utilizadas

cProfile



```
profiling_results.txt
         235725002 function calls in 131.397 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  131.397  131.397 c:\Users\perez.j.67\Documents\ULEAD\GameOfLife-Tarea-1-\game_of_life.py:39(run)
      100    9.393    0.094  131.397    1.314 c:\Users\perez.j.67\Documents\ULEAD\GameOfLife-Tarea-1-\game_of_life.py:15(step)
 26214400   91.522    0.000  121.994    0.000 c:\Users\perez.j.67\Documents\ULEAD\GameOfLife-Tarea-1-\game_of_life.py:28(count_alive_neighbors)
104755200   15.347    0.000   15.347    0.000 {built-in method builtins.min}
104755200   15.125    0.000   15.125    0.000 {built-in method builtins.max}
      100    0.010    0.000    0.010    0.000 {method 'copy' of 'numpy.ndarray' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Identificación de Cuellos de Botella

- count_alive_neighbors es el mayor cuello de botella debido a su alta cantidad de llamadas y el tiempo acumulado que consume. Esto sugiere que optimizar esta función tendría un impacto significativo en el rendimiento general del juego.
- step también es costosa, pero su costo se deriva en gran parte de la llamada a count_alive_neighbors.

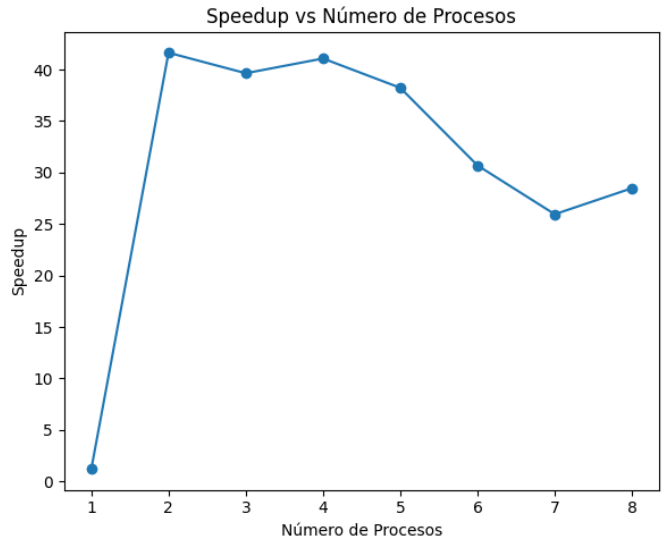
Line_Profiler

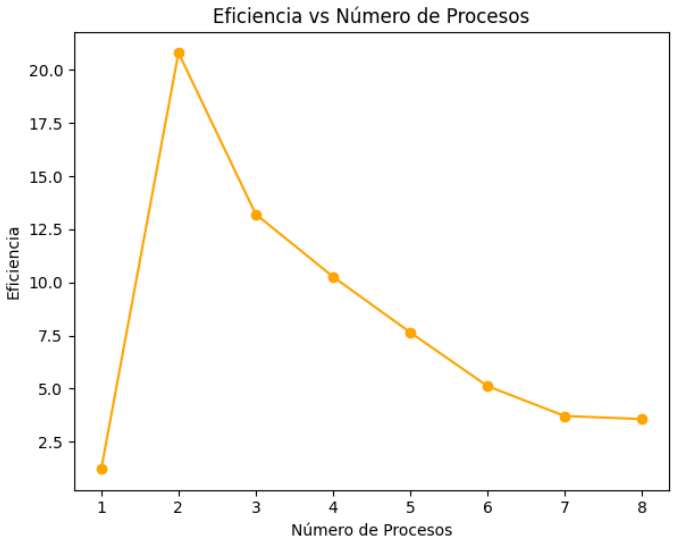
```
Wrote profile results to 'game_of_life.py.lprof'
Timer unit: 1e-06 s

Total time: 43.8863 s
File: game_of_life.py
Function: GameOfLife.step at line 15

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    15                                           @profile
    16                                           def step(self):
    17       754    15655.3     20.8      0.0          new_board = self.board.copy()
    18     38454    24800.5      0.6      0.1          for i in range(self.rows):
    19   1922700  1274241.8      0.7      2.9              for j in range(self.cols):
    20   1885000 39903318.7     21.2     90.9                  alive_neighbors = self.count_alive_neighbors(i, j)
    21   1885000  1449051.0      0.8      3.3                  if self.board[i, j] == 1:
    22    104897    69129.4      0.7      0.2                      if alive_neighbors < 2 or alive_neighbors > 3:
    23     30011    23960.5      0.8      0.1                          new_board[i, j] = 0  # Muere
    24                                                           else:
    25   1780903  1093751.1      0.6      2.5                      if alive_neighbors == 3:
    26     29325    24076.3      0.8      0.1                          new_board[i, j] = 1  # Nace
    27       754     8295.3     11.0      0.0          self.board = new_board
```

El mayor costo computacional proviene de la función *count_alive_neighbors*, que podría ser optimizada mediante vectorización, reduciendo significativamente el tiempo de ejecución del método step().

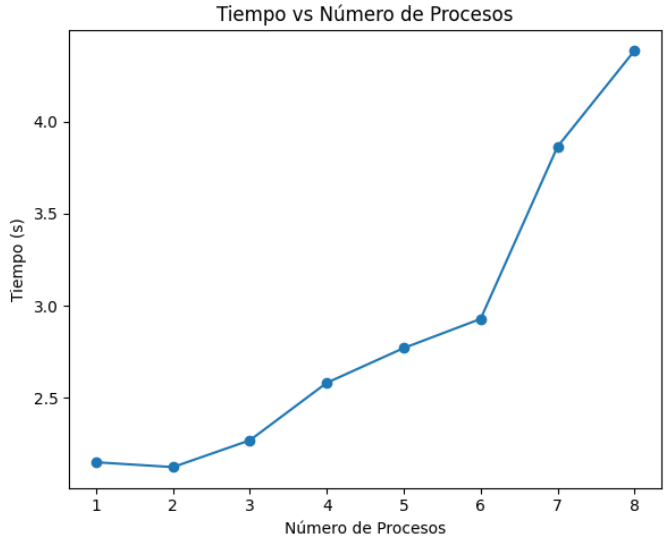
Capturas de escalamiento fuerte y debil

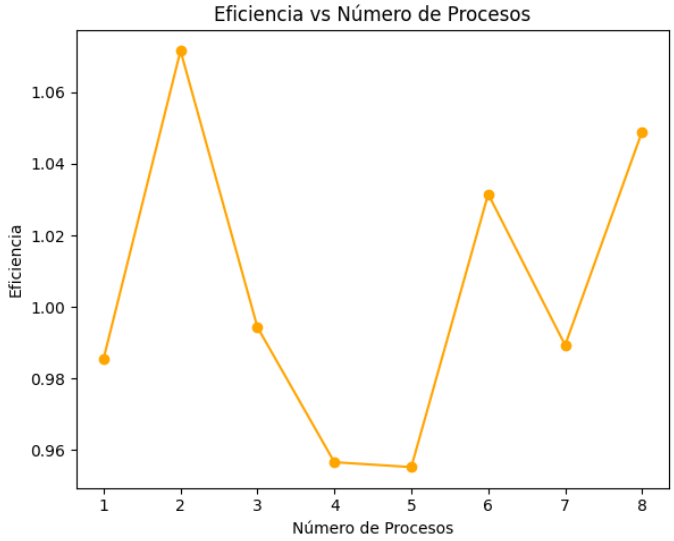
Fuerte





Debil



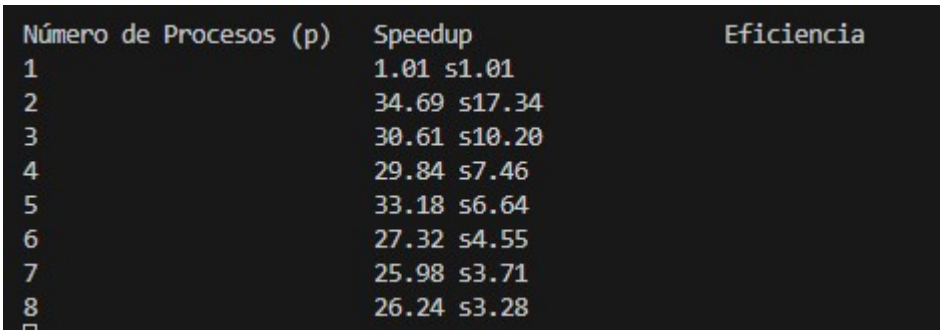


Tablas comparativas de tiempo

Escalamiento Fuerte

Número de Procesos (p)	Tiempo de Ejecución (Tp)	Eficiencia
1	2.36 s	0.95
2	2.31 s	0.98
3	2.48 s	0.95
4	2.64 s	1.00
5	3.49 s	1.02
6	3.86 s	1.00
7	4.37 s	1.03
8	4.42 s	1.21

Escalamiento Debil



Número de Procesos (p)	Speedup	Eficiencia
1	1.01 s1.01	
2	34.69 s17.34	
3	30.61 s10.20	
4	29.84 s7.46	
5	33.18 s6.64	
6	27.32 s4.55	
7	25.98 s3.71	
8	26.24 s3.28	

Análisis Crítico de los Resultados

Cuellos de Botella

Función count_alive_neighbors:

- Se identificó que la función count_alive_neighbors es el principal cuello de botella en la ejecución del programa, acumulando el 92% del tiempo total (121.994 segundos de un total de 131.397 segundos). Esto indica que el conteo de vecinos vivos es intensivo en cómputo y presenta oportunidades de optimización.
- Además, se realizaron 26,214,400 llamadas a la función count_alive_neighbors, lo que sugiere que se está invocando repetidamente para cada celda en cada paso. Este alto volumen de llamadas puede ser un factor crítico que afecta el rendimiento general.
- Las funciones incorporadas min y max también contribuyen al tiempo acumulado, lo que sugiere que su uso repetido en un contexto de alto volumen podría estar impactando negativamente el rendimiento.

Efecto de la Paralelización en el Rendimiento

- En el análisis de escalamiento débil, se observó que la eficiencia se mantiene bastante alta, especialmente con 2 a 4 procesos, donde la eficiencia se aproxima a 1. Esto sugiere que la paralelización está funcionando de manera efectiva en estas configuraciones.
- Sin embargo, a partir de 5 procesos, la eficiencia comienza a variar, lo que puede indicar que la sobrecarga de comunicación o la contención de recursos empiezan a afectar el rendimiento.
- En el análisis de escalamiento fuerte, los resultados muestran que el speedup no es lineal. Para 2 procesos, se registró un speedup significativo (34.69), pero a medida que se incrementa el número de procesos, el speedup se reduce drásticamente (por ejemplo, 26.24 para 8 procesos). Esto sugiere que la paralelización no está aprovechando eficientemente los recursos más allá de cierto punto.

Propuestas de Optimización Basadas en la Evidencia Observada

Optimización de Algoritmos

- Optimizar la función count_alive_neighbors: Revisar la lógica de esta función podría llevar a mejoras significativas en el rendimiento. Se podria considerar un enfoque alternativo para contar vecinos, como almacenar el estado de los vecinos en una estructura de datos más eficiente lo que podría permitir un acceso más rápido.
- Además, sería beneficioso minimizar las llamadas a esta función. En lugar de contar vecinos en cada paso, se podría implementar un enfoque en el que solo se cuenten los vecinos cuando cambian (por

ejemplo, cuando una celda nace o muere). Esto podría reducir considerablemente el número de invocaciones a count_alive_neighbors.

Optimización de Estructura de Datos

- Aprovechar al máximo las capacidades de NumPy, utilizando operaciones vectorizadas en lugar de realizar operaciones en bucles. Esto podría mejorar el rendimiento al permitir que se realicen cálculos en todo el array a la vez.

Optimización de Paralelización

- Para reducir la sobrecarga de comunicación, se podria implementar que los procesos hagan el menor número posible de llamadas entre sí. Dividir el tablero en secciones más grandes que se procesen de manera independiente podría ser una estrategia efectiva.
- También se puede considerar la implementación de un esquema de carga balanceada, donde cada proceso trabaje en un bloque de celdas que se adapte a su carga de trabajo, ayudando así a evitar que algunos procesos terminen mucho antes que otros.