Parcial 1 - Paralelismo en Acción - Creación de un Juego Paralelizado.

Integrantes:
Juan Manuel Perea - 1926462.
Erika García Muñoz - 202259395-3743

Presentado a:
PhD. Manuel Alejandro Pastrana.

Universidad del Valle.
Escuela de Ingeniería de Sistemas y Computación.
Infraestructuras Paralelas y Distribuidas.
Octubre de 2025.

INFORME

Implementación

El objetivo principal de esta actividad consiste en paralelizar el cálculo de las iteraciones o generaciones del Juego de la Vida de John Conway para reducir su tiempo de ejecución, manteniendo la corrección del algoritmo; donde cada celda se actualiza usando el estado de la generación previa completa.

Se implementaron tres versiones del Juego de la Vida, siendo la primera la secuencial, la segunda donde se implementaron hilos usando la librería Threading y la última con procesos implementando la librería Multiprocessing y un Pool de datos. La estructura algorítmica es la misma en las tres estrategias. Se inicia con la generación aleatoria de una matriz binaria y luego se calcula, por célula, el número de vecinos vivos con el que se define la actualización de su valor en cada generación.

Cómo se podría inferir de lo anterior y detallando un poco, la tarea principal por cada generación se basa en calcular una nueva matriz llamada “nueva_matriz” a partir de la matriz actual (Que en un principio es la inicial, generada aleatoriamente con 0s y 1s - binaria). La actualización de cada celda se calcula de forma independiente donde todas las lecturas se hacen de la matriz vieja (Actual o inicial) y las escrituras se hacen en la matriz nueva, permitiendo paralelizar por particiones.

Se consideraron distintas formas de particionar la matriz al momento de paralelizar. Dos de ellas consistían en hacerlo por filas o por columnas, donde cada hilo recibe un conjunto de filas o columnas completas; según el caso. Las implementaciones realizadas particionan la matriz por filas, ya que es una forma sencilla sencilla y fácil de implementar, y se puede tener un buen manejo de fronteras; es decir que, el calcular las celdas que están cerca de los bordes de cada partición es mucho más fácil de resolver que por otras alternativas.

Para obtener el resultado correcto del Juego de la Vida, todas las celdas deben evaluarse con el estado de la generación anterior, para esto usamos dos matrices (Tal y como se ha mencionado antes), una matriz para leer el estado actual del juego y otra para actualizar el estado a través de los hilos o procesos (Según sea el caso). Gracias a lo anterior, no hay una escritura concurrente en la misma ubicación de memoria, por lo que no es necesario usar locks y por tanto se evitan condiciones de carrera. También se usaron barreras o join() para los hilos, de tal forma que no se actualice el estado del juego sin que todos los hilos hayan terminado, y en el caso de los procesos se logra un resultado similar a través del Pool de datos,

garantizando que no se actualice el estado global del juego hasta que todos los procesos hayan completado su tarea.

Hablando de forma más detallada sobre la paralelización que se llevó a cabo, se implementaron dos enfoques distintos (Tal y como se viene mencionando), uno utilizando hilos con la librería Threading y otro basado en procesos mediante la librería Multiprocessing. Inicialmente se implementaron hilos por distintas razones, destacando que son más sencillos de implementar, que hay menos overhead en la comunicación, ya que la memoria está compartida de forma natural, y que es una de las opciones que explícitamente se pide explorar dentro del enunciado de la actividad.

La implementación con hilos presentó algunas limitaciones inherentes a Python, ya que el Global Interpreter Lock (GIL) impide que múltiples hilos ejecuten código puro de Python en paralelo dentro de un mismo proceso, haciendo que; a pesar de los esfuerzos para que el código estuviese correctamente sincronizado y libre de condiciones de carrera; el rendimiento obtenido fuera incluso menor al de la versión secuencial. Por otro lado, la versión con multiprocessing resultó ser más eficiente al aprovechar múltiples núcleos del procesador mediante procesos independientes, donde cada proceso tiene su propio intérprete de Python y su propia memoria, eliminando así los límites impuestos por el GIL (Como ocurre en el caso de los hilos).

Al igual que con el otro enfoque, este presenta una consideración importante relacionada a la comunicación entre procesos, y es que puede ser muy costosa debido a la serialización que se le hace a los datos para poder ser enviados (En este caso, se refiere a los bloques de la matriz). Sin embargo, el beneficio de la ejecución paralela compensa ampliamente dicho overhead, especialmente en matrices grandes.

Ambas implementaciones comparten ciertos retos técnicos. Uno de ellos fue el balance de carga, ya que cuando el número de filas de la matriz no es divisible entre el número de hilos o procesos, el último bloque de trabajo resulta de tamaño diferente. En este caso, se mitigó asignando al último hilo o proceso las filas restantes tras la división entera, garantizando que todo el tablero fuera cubierto aunque no de manera perfectamente equilibrada. Otra dificultad estuvo relacionada con el coste de impresión de la matriz durante cada generación, el cual generaba un cuello de botella considerable. Por esta razón, se optó por omitir la impresión visual del tablero durante las pruebas de rendimiento, limitando la salida a los resultados de tiempo total de ejecución.

En general, ambas implementaciones comparten algunos retos técnicos como lo es el relacionado al balance de carga, donde en el caso en que el número de filas no no sea divisible con respecto al número de hilos o procesos implementado, pues

uno de estos hilos o procesos tendrá más o menos trabajo que los otros; por ejemplo, si hay 100 filas en la matriz y se quiere usar 7 hilos o procesos, pues el último tendrá una cantidad distinta de trabajo. En la implementación como tal, se mitigó al redondear por debajo y asignarle el resto de filas al último hilo o proceso, haciendo que deba trabajar más que los otros. Otra dificultad está relacionada con el coste de impresión de la matriz para ver la evolución de sus estados por cada generación. Al ser costoso, se decidió omitir la impresión.

Finalmente, la sincronización de las unidades de trabajo se resolvió mediante mecanismos apropiados en cada caso, con barreras implícitas (join()) en los hilos y mediante la recolección de resultados del Pool en el caso de los procesos, asegurando que el estado del juego no se actualizara hasta que todas las tareas hubieran finalizado correctamente.

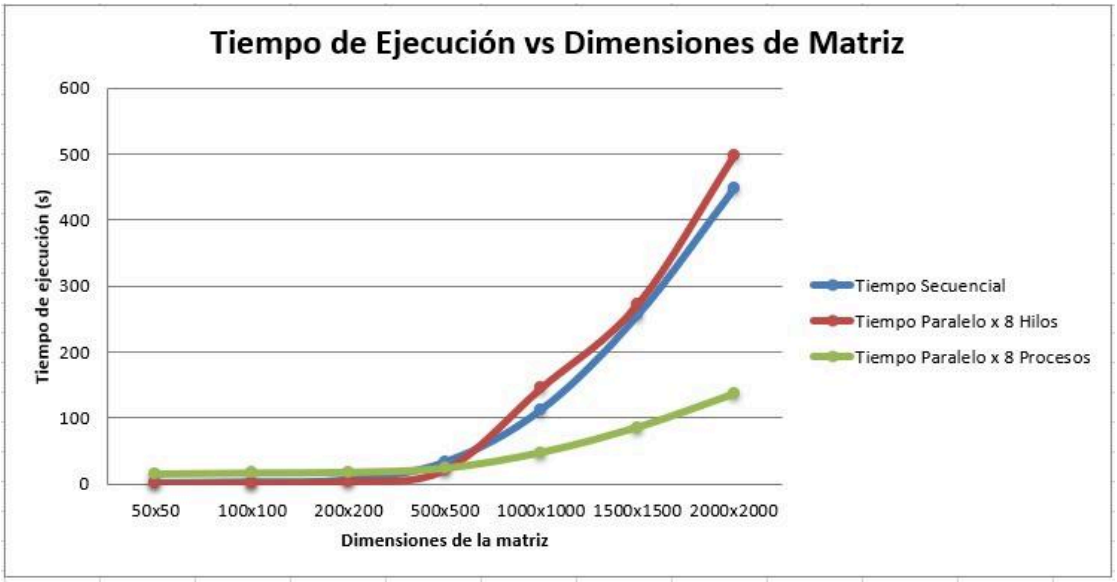
Análisis de Resultados

En esta etapa, se realizaron diversas pruebas utilizando tanto la estrategia secuencial como las paralelas, con el objetivo de analizar los beneficios de estas últimas. La siguiente tabla muestra los resultados obtenidos al ejecutar el Juego de la Vida con diferentes tamaños de matriz durante 50 generaciones. Se trabajó con tres versiones del algoritmo: el Tiempo Secuencial, que representa la ejecución tradicional sin paralelismo; el Tiempo Paralelo x 8 Hilos, implementado con Threading; y el Tiempo Paralelo x 8 Procesos, desarrollado con Multiprocessing. Los resultados permiten apreciar cómo cambia el tiempo de ejecución según el tipo de procesamiento y el tamaño de la matriz.

Tabla 1. Tiempo de ejecución vs. Tamaño de la matriz

N° Generaciones	Dimensiones	Tiempo Secuencial	Tiempo Paralelo x 8 Hilos	Tiempo Paralelo x 8 Procesos
50	50x50	1.68	1.48 s	15.47 s
50	100x100	2.80	2.21 s	16.92 s
50	200x200	6.10	4.73 s	17.99 s
50	500x500	32.64	21.96 s	24.13 s
50	1000x1000	112.61 s	145.87 s	48.04 s
50	1500x1500	256.33 s	272.07 s	86.00 s
50	2000x2000	449.06 s	496.64 s	136.47 s

En la siguiente gráfica se muestra cómo cambia el tiempo de ejecución del Juego de la Vida a medida que aumenta el tamaño de la matriz. Se comparan las tres versiones del algoritmo: la secuencial, la paralela con 8 hilos y la paralela con 8 procesos. En la imagen se puede notar que, para matrices pequeñas, las diferencias entre las tres versiones son mínimas, ya que el costo de crear hilos o procesos compensa el trabajo realizado. Sin embargo, a medida que la matriz crece, la versión con Multiprocessing (8 procesos) muestra un mejor desempeño, manteniendo tiempos más bajos en comparación con las versiones secuencial y con hilos. Esto se debe a que los procesos aprovechan de forma más eficiente los núcleos del procesador, distribuyendo la carga de trabajo. En cambio, la versión con threading no muestra una mejora significativa frente a la secuencial, lo que es esperado en Python, ya que los hilos están limitados por el Global Interpreter Lock (GIL). De acuerdo a esto, la gráfica nos muestra cómo el uso del procesamiento en paralelo se vuelve más beneficioso cuando el tamaño del problema es grande.



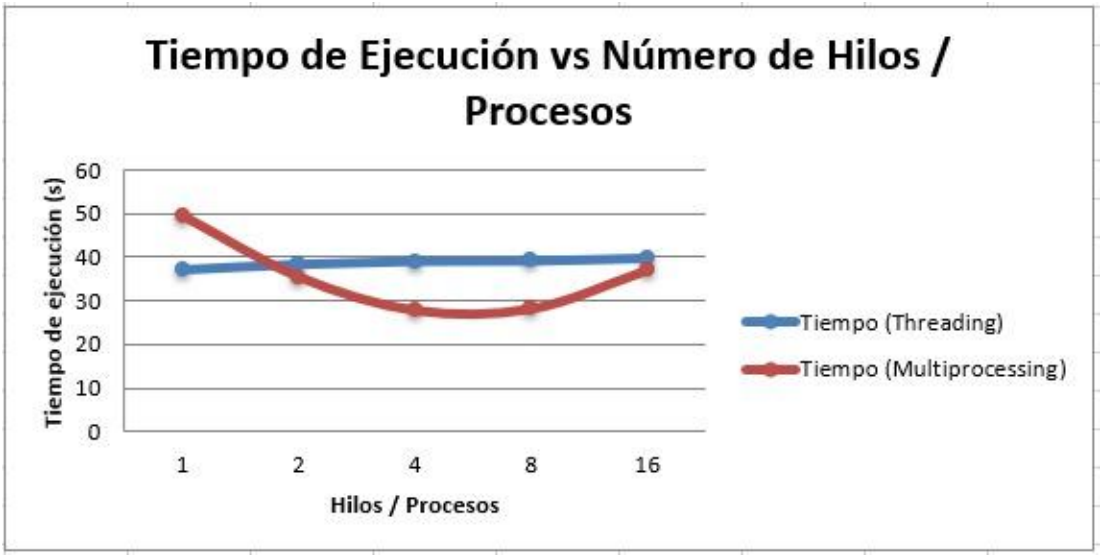
En la siguiente tabla se presentan los resultados obtenidos al ejecutar el Juego de la Vida con una matriz de 500x500 durante 50 generaciones, variando la cantidad de hilos o procesos utilizados. El objetivo de esta prueba fue analizar cómo influye el número de unidades de ejecución en el rendimiento del programa, tanto con Threading como con Multiprocessing. De esta manera se puede comparar el rendimiento del algoritmo según el número de hilos o procesos y determinar hasta qué punto el aumento de paralelismo mejora el desempeño.

Tabla 2. Tiempo de ejecución vs. Número de hilos / procesos

N° Generaciones	Dimensiones	Hilos / Procesos	Tiempo (Threading)	Tiempo (Multiprocessing)
50	500x500	1	37.01 s	49.18 s
50	500x500	2	38.33 s	35.29 s
50	500x500	4	39.06 s	27.69 s
50	500x500	8	39.15 s	28.08 s
50	500x500	16	39.79 s	36.88 s

La siguiente gráfica muestra cómo cambia el tiempo de ejecución del algoritmo al aumentar el número de hilos o procesos. Se observa que con un solo hilo o proceso el tiempo es alto, pero al incrementar hasta 4 procesos el rendimiento mejora notablemente, reduciendo el tiempo de ejecución. Sin embargo, a partir de ese punto (más de 4 procesos), el tiempo vuelve a aumentar ligeramente, lo que indica que agregar más hilos o procesos ya no ofrece beneficios significativos e incluso puede generar sobrecarga. En general, el multiprocessing muestra mejor

desempeño que el threading, especialmente en el punto óptimo donde alcanza el menor tiempo de ejecución.



En general, los resultados muestran que el multiprocessing mejora notablemente el rendimiento del Juego de la Vida frente a la versión secuencial, sobre todo con matrices grandes, ya que aprovecha mejor los núcleos del procesador. En cambio, el threading no ofrece mejoras significativas. Además, se observó que el mejor desempeño se obtiene con alrededor de 4 procesos, ya que usar más genera sobrecarga sin reducir el tiempo de ejecución.

Conclusiones

El problema del Juego de la Vida que se aborda en esta actividad es un claro ejemplo de paralelismo de datos. Aquí, una misma operación, que consiste en evaluar el estado de una célula en función de sus vecinos, se aplica de manera independiente a cada elemento de la matriz. Desde cierta perspectiva, esto podría considerarse un tipo de paralelismo SIMD (Single Instruction, Multiple Data). Sin embargo, en la práctica, el rendimiento final no solo depende de la estructura del problema, sino también de las herramientas y la arquitectura de ejecución que se utilicen.

En la implementación con hilos, se intentó aprovechar la concurrencia utilizando la librería Threading, que es fácil de usar y permite que todos los hilos compartan el mismo espacio de memoria. Sin embargo, dado que se trata de un problema que requiere mucho cómputo, esta estrategia no resultó en beneficios reales, debido al Global Interpreter Lock (GIL) de Python. Como resultado, los hilos terminan alternándose para usar el procesador, lo que genera una ejecución que parece paralela, pero en realidad es secuencial. De hecho, los resultados obtenidos indicaron que esta versión fue un poco más lenta que la secuencial, debido al overhead adicional de crear y coordinar los hilos.

En contraste, la versión que utiliza procesos logró aprovechar realmente los recursos del sistema al ejecutar diferentes procesos en núcleos independientes del procesador. Cada proceso cuenta con su propio intérprete y no está limitado por el GIL, lo que permite un verdadero paralelismo. Gracias a esto, se notó una reducción considerable en los tiempos de ejecución, especialmente al trabajar con matrices más grandes. Sin embargo, esta estrategia no está libre de costos: dado que cada proceso tiene su propio espacio de memoria, es necesario copiar o serializar los datos antes de enviarlos a cada proceso, lo que introduce un pequeño retraso inicial. Aun así, este costo adicional se compensa ampliamente cuando el volumen de trabajo es grande, ya que los núcleos pueden operar de manera simultánea y equilibrada.

En promedio, se encontró que la estrategia con hilos tuvo un speedup de 1.113 con respecto a la secuencial, indicando que en promedio es un 11%, aproximadamente, más rápido. Por otra parte, la estrategia con procesos obtuvo un speedup de 1.511, es decir que fue 51%, aproximadamente más rápido con relación a los resultados del secuencial.

Repositorio

Para ingresar al repositorio donde se encuentran los scripts de las estrategias trabajadas, haga click en el siguiente enlace:

Enlace al repositorio.