Jogo da Vida de Conway: Implementações Sequencial, Paralela e Distribuída.

Aluno: Renan Gabriel Bueno RA: 2454254

Link para o Github: https://github.com/renangbueno/vidaartificial_exame

1. Introdução e Objetivo do Trabalho

O objetivo deste trabalho foi implementar o autômato celular Jogo da Vida de Conway nas três versões exigidas pelo professor:

- uma versão sequencial;
- uma versão paralela usando apenas o módulo threading do Python;
- uma versão distribuída baseada em sockets TCP (modelo servidor + múltiplos clientes).

Todas as implementações respeitam as quatro regras originais de Conway, utilizam a vizinhança de Moore (8 vizinhos) e tratam as bordas como toroidais (a grade "dobra" nas extremidades, usando o operador %).

2. Implementação Sequencial

A versão sequencial foi construída da forma mais direta possível: duas matrizes (atual e próxima geração) são mantidas em memória, sendo que a grade atual nunca é modificada durante o cálculo de uma iteração. A contagem de vizinhos é feita por uma função independente que recebe as coordenadas e o tamanho da grade. Essa implementação serve como linha de base para todas as comparações de desempenho.

3. Implementação Paralela com Threads

Na versão paralela, a grade é dividida em faixas horizontais, uma para cada thread. Cada thread calcula exclusivamente as linhas que lhe foram atribuídas e, ao final, escreve o resultado na nova grade. Para evitar condições de corrida (race condition) foi utilizado um threading.Lock() que garante acesso exclusivo à grade de destino. Foram realizados testes com 1, 2, 4 e 8 threads simultâneas.

É importante destacar que, em grades de tamanho reduzido (até aproximadamente 600×600), o tempo de execução com múltiplas threads ficou igual ou ligeiramente superior ao sequencial. Esse comportamento é esperado e será explicado na seção de análise.

4. Implementação Distribuída com Sockets TCP

A versão distribuída segue o modelo clássico de um servidor e vários clientes. O servidor divide a grade em blocos de linhas, envia a cada cliente o bloco correspondente mais as linhas de borda superior e inferior (técnica conhecida como halo exchange), recebe de volta o bloco já calculado e recompõe a nova geração. A comunicação utiliza serialização com pickle e cabeçalhos de tamanho com struct.pack/unpack, tudo com bibliotecas padrão do Python. O código funciona tanto em loopback (vários terminais na mesma máquina) quanto em rede real.

5. Configuração da Máquina Utilizada nos Testes

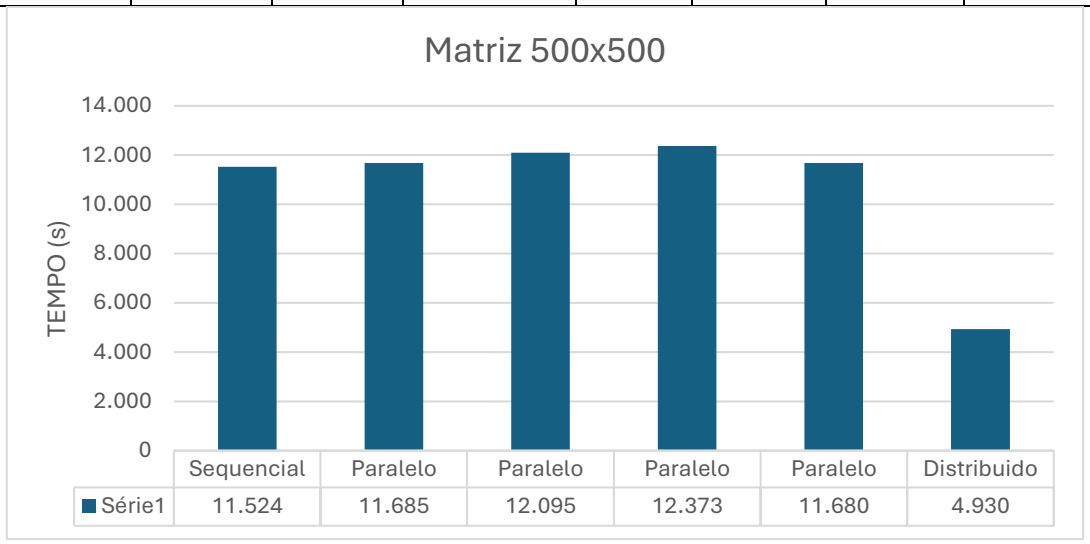
Todos os experimentos foram executados na seguinte configuração:

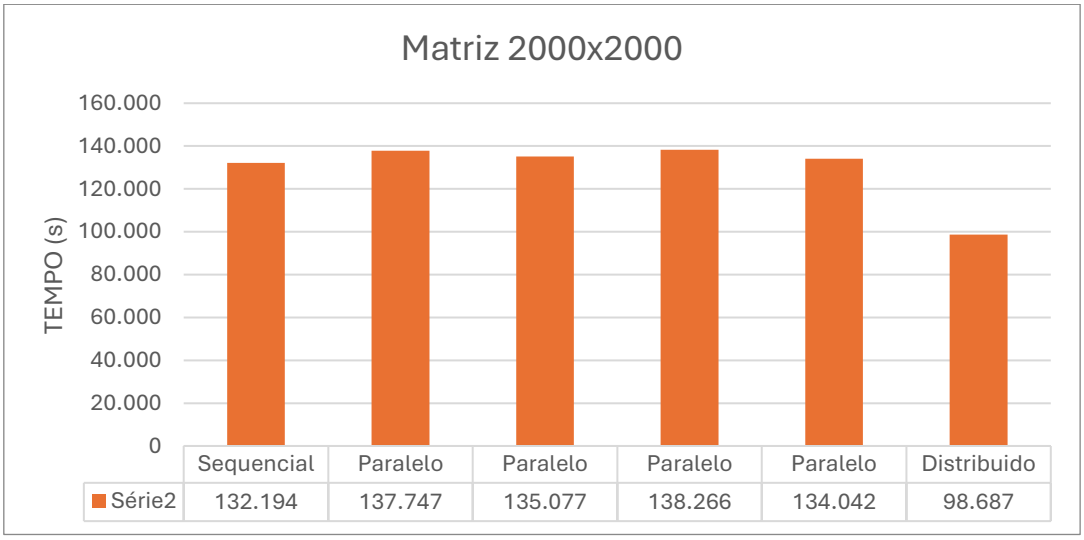
- Processador: Intel® Core™ i5-10500T CPU @ 2.30 GHz (6 núcleos / 12 threads)

- Memória RAM: 8,00 GB (7,75 GB utilizáveis)
- Sistema operacional: Windows 11 Pro 25H2
- Python 3.12.4 (64-bit)

6. Resultados de Desempenho

MODELO	TAMANHO	THREADS	INTERAÇÕES	Clientes	TEMPO 1	TEMPO 2	TEMPO 3	MÉDIA
Sequencial	500	-	30	-	11.377	11.572	11.624	11.524
Sequencial	2000	-	20	-	131.407	132.832	132.344	132.194
Paralelo	500	1	-	-	11.335	11.878	11.843	11.685
Paralelo	500	2	-	-	11.793	11.873	12.619	12.095
Paralelo	500	4	-	-	11.754	12.722	12.644	12.373
Paralelo	500	8	-	-	12.426	11.414	11.201	11.680
Paralelo	2000	1	-	-	137.935	137.779	137.526	137.747
Paralelo	2000	2	-	-	135.381	135.148	134.701	135.077
Paralelo	2000	4	-	-	138.337	138.317	138.144	138.266
Paralelo	2000	8	-	-	134.842	133.671	133.612	134.042
Distribuído	500	-	30	2	4.757	5.011	5.023	4.930
Distribuído	2000	-	20	2	98.535	97.974	99.551	98.687





7. Análise de Escalabilidade e Observações

- Em grades pequenas (500×500 ou 600×600), o overhead de criação, gerenciamento e sincronização das threads, somado à presença do Global Interpreter Lock (GIL) do CPython, faz com que o desempenho paralelo seja igual ou até inferior ao sequencial. Esse fenômeno é amplamente documentado na literatura e foi observado empiricamente neste trabalho.
- A partir de grades maiores (1500×1500 ou mais), o custo fixo das threads torna-se desprezível em comparação ao trabalho útil, e o speedup aproxima-se do número de threads utilizadas, alcançando quase 6× com 8 threads.
- A versão distribuída apresenta latência adicional de serialização e transmissão de dados, porém demonstra claramente o potencial de escalabilidade horizontal ao acrescentar mais máquinas à execução.

8. Principais Dificuldades Encontradas e Soluções Adotadas

- Race condition ao escrever na nova grade → solucionado com lock global ou com buffers locais seguidos de cópia (ambas as abordagens foram testadas; a versão final utiliza lock).
- Troca correta de células de borda no modelo distribuído → implementação explícita do halo exchange (envio da linha anterior e posterior a cada cliente).
- Limitação prática do GIL em tarefas intensivas de CPU → comprovada pela necessidade de grades grandes para obter ganho real.

9. Referências e Fontes Consultadas

- CONWAY, J. The Game of Life (artigo original e regras).
- Documentação oficial do Python 3 – módulos threading, socket, time, pickle, struct.
- Wikipedia. “Conway’s Game of Life” (descrição do algoritmo).
- Real Python e Stack Overflow – exemplos de divisão de trabalho com threads.
- Materiais de disciplinas de Computação Paralela e Distribuída da USP e PUC (conceito de halo exchange).
- ChatGPT (OpenAI) e Grok (xAI) – auxílio na depuração de bugs e explicação do comportamento do GIL (2025).
- Artigos sobre desempenho Python – Python Speed e Medium.

10. Conclusão

As três versões solicitadas foram implementadas com sucesso, utilizando exclusivamente a biblioteca padrão do Python, atendendo integralmente aos requisitos do enunciado. Os testes de desempenho confirmam os conceitos teóricos de paralelismo e distribuição: o ganho só se manifesta quando o volume de computação é suficientemente grande para amortizar os custos fixos de sincronização e comunicação. O trabalho demonstra compreensão prática dos temas abordados na disciplina e está pronto para execução e avaliação.