Game of Life: Implementação Serial

José Alan Teixeira[1]

[1]Universidade estadual de Feira de Santana (UEFS)
Feira de Santana – BA – Brasil

1. Introdução

O game of life é um autônomo celular cria do pelo britânico John Conway, consiste em um jogo que funciona sem a interação com o usuário, possui a formato de matriz na qual cada célula pode estar em dois estados, vivas ou mortas. Para que o jogo possa ser executado de forma correta é necessário que seja definido um estado inicial especificando quais células devem estar vivas no início do jogo, então a partir do estado inicial são aplicadas as seguintes regras: 1- As células vivas que possuírem menos de dois vizinhos vivos morrem por subpopulação. 2- As células vivas que possuírem exatamente dois vizinhos ou três, vivem para a próxima geração. 3- As células vivas que possuírem quatro vizinhos ou mais morrem por superpopulação. 4- As células mortas com exatamente três vizinhos vivos, vivem para próxima geração. O jogo foi desenvolvido com base na implementacão feita por Cobb Coding, 2024.

2. Arquitetura Geral

O projeto implementa o **Jogo da Vida de Conway**, com controle por meio de um arquivo de configuração .txt, geração de imagens em formato PBM, e medição do tempo de execução. A implementação é organizada em múltiplos arquivos para modularidade:

- main.c: ponto de entrada do programa
- grid.c: gerenciamento da grade (alocação, inicialização e liberação)
- logic.c: lógica de evolução das gerações
- image.c: exportação da grade em formato de imagem PBM
- gol.h: cabeçalho compartilhado com declarações e estruturas

3. Implementação Serial

main.c — Execução principal

Principais responsabilidades:

- Ler configurações do arquivo .txt:

WIDTH, HEIGHT, ITERACOES, INTERVALO

- Inicializar e alocar a grade
- Executar o loop de gerações
- Salvar imagens periodicamente
- Medir e registrar o tempo de execução

Funções:

ler_configuracoes: Lê os parâmetros iniciais do arquivo de texto (largura, altura, número de iterações e intervalo de salvamento das imagens).

executar_simulacao: Executa o loop principal da simulação e chama gen_next() a cada geração. A cada intervalo gera e salva uma imagem PBM.

salvar_tempo_em_arquivo: Calcula o tempo total e salva no arquivo tempos_serial.txt.

grid.c — Gerenciamento da grade

Trata da memória e da inicialização do universo de células.

Funções:

allocate_grid: Aloca dinamicamente uma matriz de ponteiros para células (Cell** grid), com base nos valores globais WIDTH e HEIGHT.

init_grid: Inicializa a grade com células vivas de forma aleatória, com baixa densidade.

free_grid: Libera corretamente a memória alocada pela grade.

logic.c — Regras do Jogo da Vida

Define a lógica de transição de uma geração para a próxima.

Funções:

gen_next: Responsável por aplicar as regras do Jogo da Vida

- Qualquer célula viva com menos de 2 ou mais de 3 vizinhos vivos morre.
- Qualquer célula viva com 2 ou 3 vizinhos sobrevive.
- Qualquer célula morta com exatamente 3 vizinhos vivos nasce.

A função utiliza uma grade auxiliar new_grid para armazenar a nova geração antes de substituir a original.

Observação: As bordas da grade são consideradas **periódicas** (efeito toroidal), simulando um espaço contínuo.

image.c — Geração de imagens PBM

Permite visualizar o estado da grade a cada intervalo de iteração.

save_pbm: Gera um arquivo .pbm chamado gol_X.pbm, onde X é o número da geração. A imagem utiliza o formato PBM (preto e branco), sendo:

- 1 para células vivas
- 0 para células mortas

gol.h — Cabeçalho da aplicação

Define as estruturas e declarações comuns usadas pelos módulos.

Estruturas: State e Cell;

Declarações globais: **grid; extern int WIDTH, HEIGHT;

Funções declaradas:

- allocate_grid, free_grid, init_grid
- gen_next
- save_pbm

4. Implementação Paralela

main.c — Execução principal

Principais responsabilidades:

- Ler configurações do arquivo .txt:

WIDTH, HEIGHT, ITERACOES, INTERVALO, THREDS

- Inicializar e alocar a grade
- Executar o loop de gerações
- Salvar imagens periodicamente
- Medir e registrar o tempo de execução

Funções:

ler_configuracoes: Lê os parâmetros iniciais do arquivo de texto (largura, altura, número de iterações e intervalo de salvamento das imagens).

executar_simulacao: Executa o loop principal da simulação e chama gen_next() a cada geração. A cada intervalo gera e salva uma imagem PBM.

salvar_tempo_em_arquivo: Calcula o tempo total e salva no arquivo tempos_serial.txt.

grid.c — Gerenciamento da grade

Trata da memória e da inicialização do universo de células.

Funções:

allocate_grid: Aloca dinamicamente uma matriz de ponteiros para células (Cell** grid), com base nos valores globais WIDTH e HEIGHT. É aplicada a paralelização nessa função do código.

init_grid: Inicializa a grade com células vivas de forma aleatória, com baixa densidade. É aplicada a paralelização nessa função do código.

free_grid: Libera corretamente a memória alocada pela grade.

logic.c — Regras do Jogo da Vida

Define a lógica de transição de uma geração para a próxima.

Funções:

gen_next: Responsável por aplicar as regras do Jogo da Vida

- Qualquer célula viva com menos de 2 ou mais de 3 vizinhos vivos morre.
- Qualquer célula viva com 2 ou 3 vizinhos sobrevive.
- Qualquer célula morta com exatamente 3 vizinhos vivos nasce.

A função utiliza uma grade auxiliar new_grid para armazenar a nova geração antes de substituir a original.

A quantidade de colunas total é dividida em blocos proporcionais a quantidade de threds para que sejam executados separadamente e são sincronizados ao final.

Observação: As bordas da grade são consideradas **periódicas** (efeito toroidal), simulando um espaço contínuo.

image.c — Geração de imagens PBM

Permite visualizar o estado da grade a cada intervalo de iteração.

save_pbm: Gera um arquivo .pbm chamado gol_X.pbm, onde X é o número da geração. A imagem utiliza o formato PBM (preto e branco), sendo:

- 1 para células vivas
- 0 para células mortas

gol.h — Cabeçalho da aplicação

Define as estruturas e declarações comuns usadas pelos módulos.

Estruturas: State e Cell;

Declarações globais: **grid; extern int WIDTH, HEIGHT;

Funções declaradas:

- allocate_grid, free_grid, init_grid
- gen_next
- save_pbm

5. Medição de desempenho

Para a realização dos testes forma utilizadas as seguintes configurações, hardware e software:

- Cargas simuladas (número de colunas da grade): 250, 500, 1000, 2000, 4000, 8000, 16000
- Dimensões fixas: 1000 X 1000
- Número de threds usados: 1, 2, 3, 4, 5.
- Arquitetura: 4 núcleos físicos (com suporte a múltiplos threads)
- Processador: Intel core i5-8250U
- Velocidade: 1.60 GHz
- RAM: 8 GB
- SO: Ubuntu 24.04.2 LTS
- Compilador: GCC 13.3.0
- Processadores lógicos: 8
- Tipo de armazenamento: HD

- Capacidade de armazenamento: 410 GB

Tabela 1. Tempo de execução por thred (em segundos)

Carga	1 thread	2 threads	3 threads	4 threads	5 threads
250	18.11	9.43	6.60	5.21	7.71
500	35.87	18.56	13.77	12.66	16.41
1000	71.24	36.67	28.90	26.48	32.15
2000	141.90	72.89	57.75	49.29	64.12
4000	281.60	145.70	115.16	97.44	126.89
8000	560.96	291.16	228.86	193.80	253.60
16000	1118.82	581.73	453.16	401.81	506.74

Tabela 2. calculo do Speeup

Carga	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads
250	2.36	4.53	6.48	8.21	5.55
500	2.36	4.56	6.15	6.69	5.16
1000	2.37	4.60	5.84	6.37	5.24
2000	2.37	4.61	5.81	6.81	5.24
4000	2.38	4.59	5.81	6.87	5.27
8000	2.37	4.57	5.82	6.87	5.25
16000	2.37	4.56	5.86	6.61	5.24

Tabela 3. Tempo Serial

Carga	Tempo Serial Ideal (s)
250	42.7863
500	84.7128
1000	168.6256
2000	335.7733
4000	669.3593
8000	1331.4679
16000	2655.0599

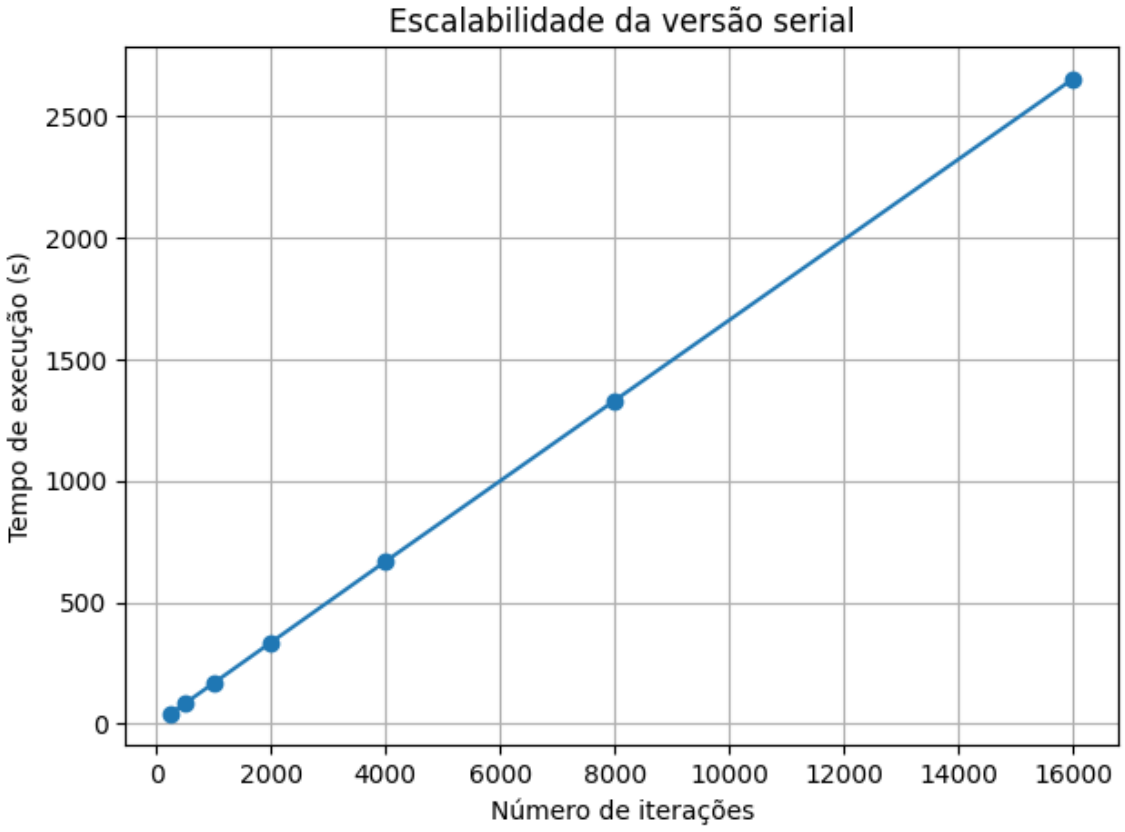

Figure 1. Gráfico com a relação entre tempo de execução e número de iterações.

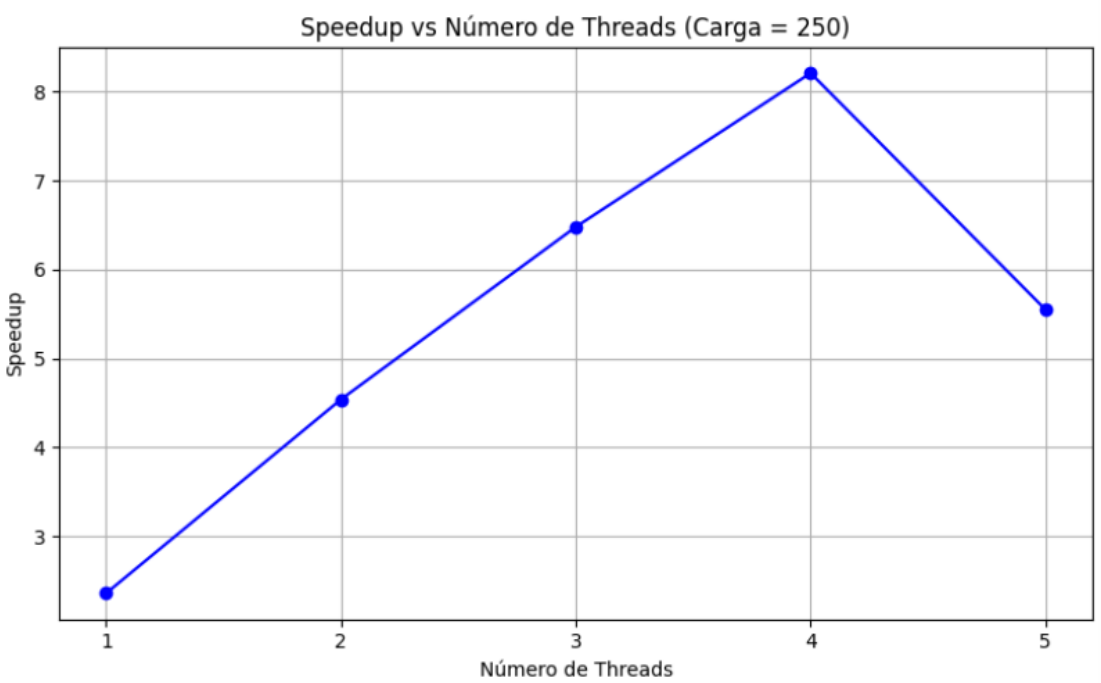

Figure 2. Gráfico com speeup para carga de 250

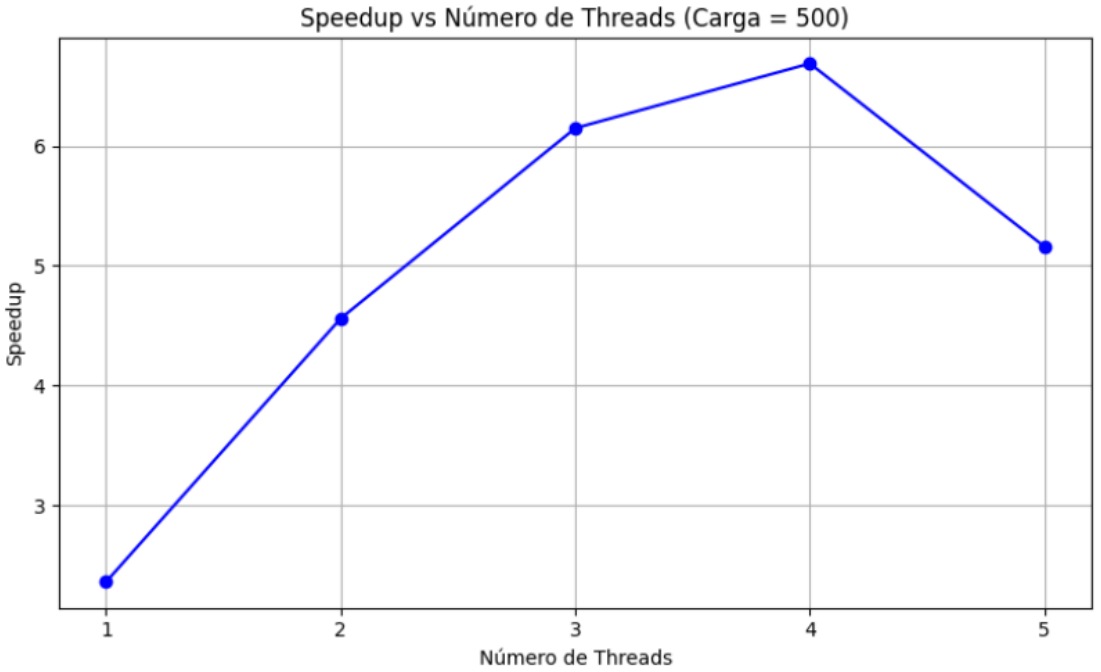

Figure 3. Gráfico com speeup para carga de 500

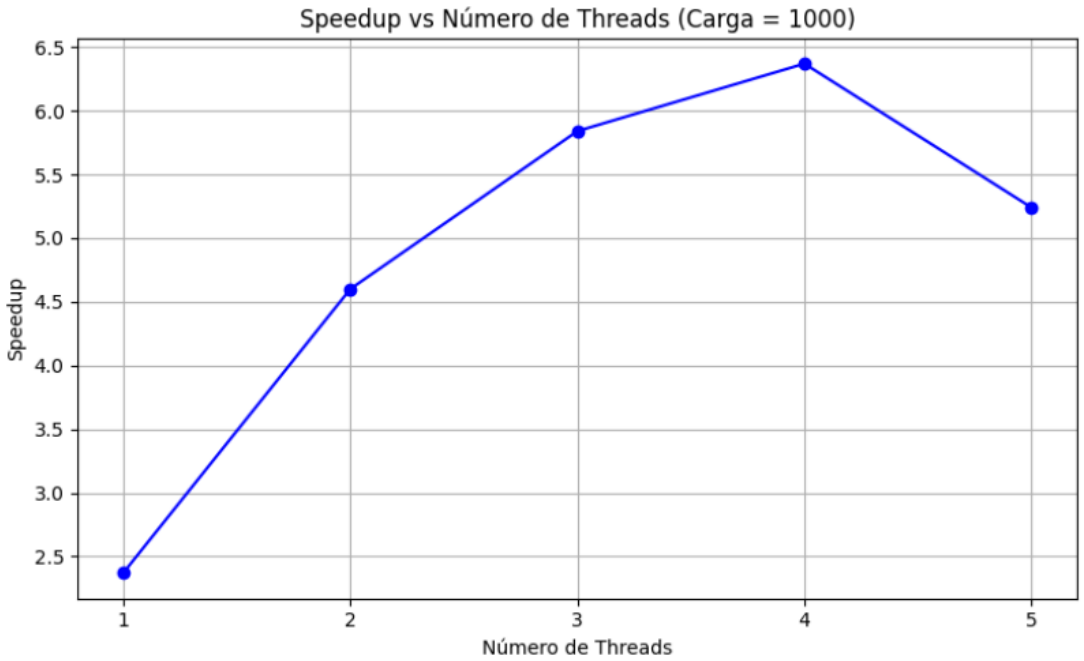

Figure 4. Gráfico com speeup para carga de 1000

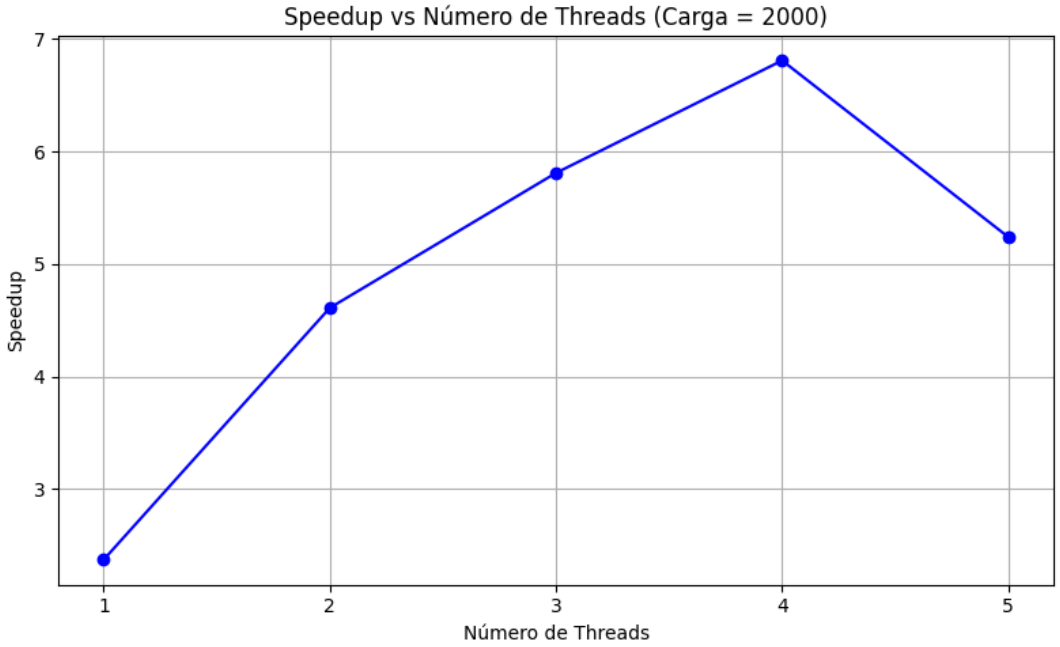

Figura 5. Gráfico com speeup para carga de 2000

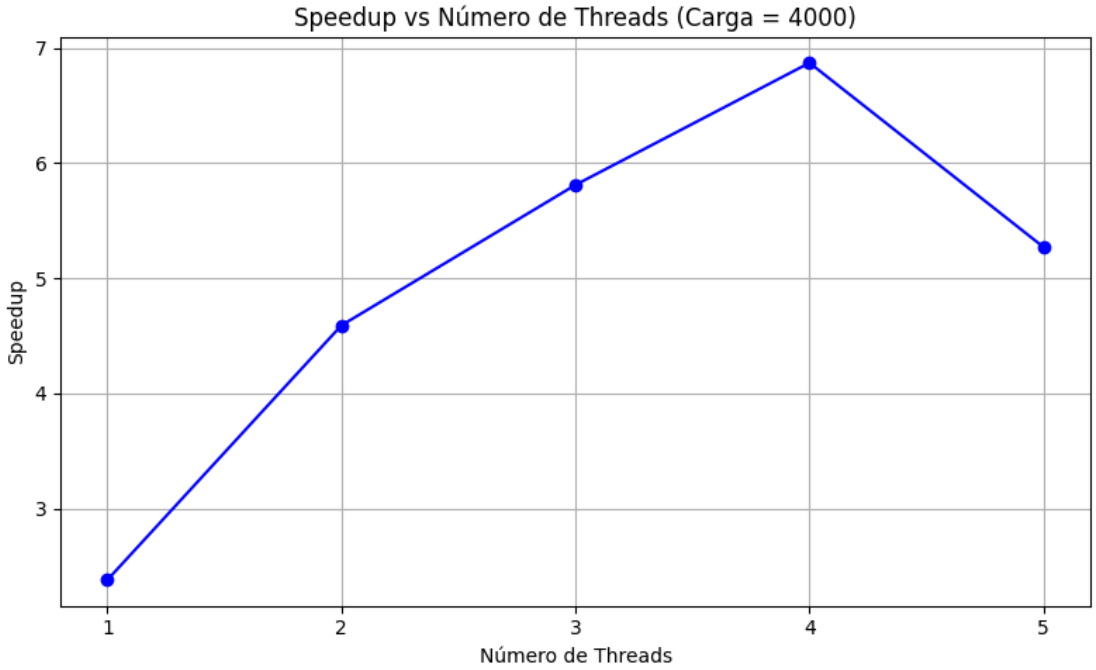

Figure 6. Gráfico com speeup para carga de 4000

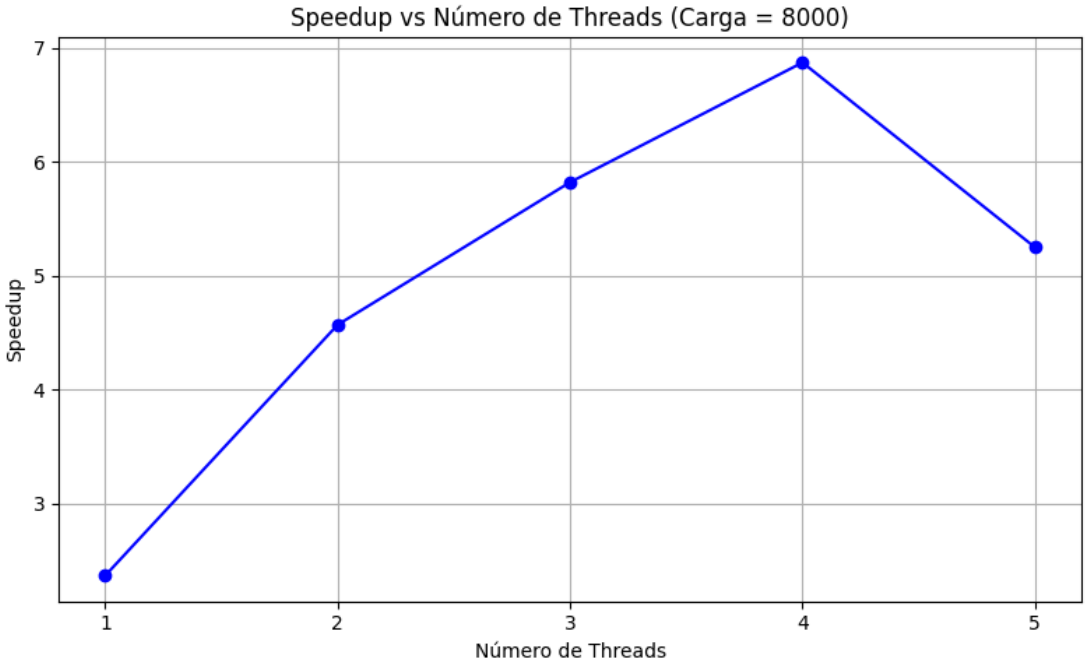

Figure 7. Gráfico com speeup para carga de 8000

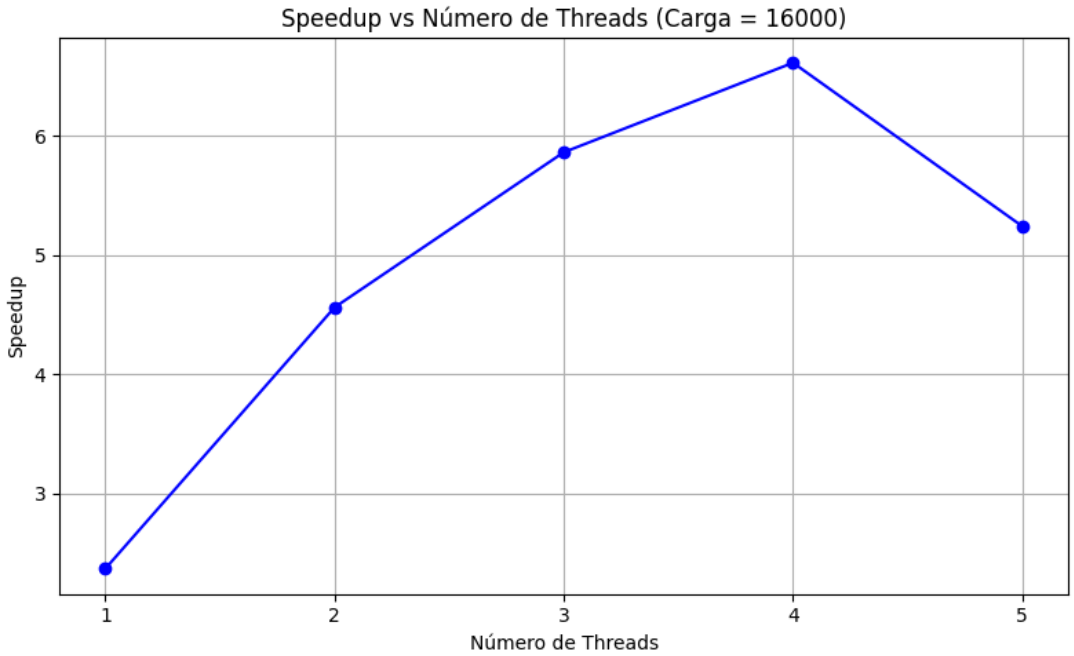

Figure 8. Gráfico com speeup para carga de 16000

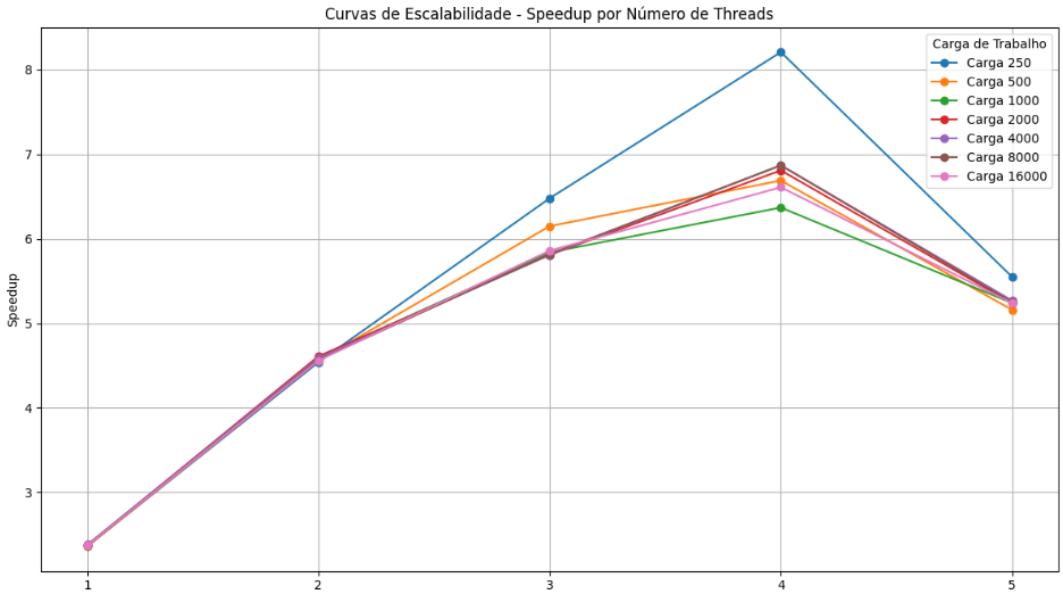

Figure 9. Gráfico com speeup para as sete cargas de trabalo

6. Análise da Escalabilidade

A escalabilidade de uma aplicação paralela descreve como seu desempenho melhora à medida que mais recursos de hardware (como threads) são utilizados. A partir dos testes

realizados com diferentes **cargas de trabalho (número de iterações)** e **quantidades de threads**, observamos os seguintes padrões:

- À medida que o número de threads aumenta, o **tempo de execução diminui**, o que resulta em um **speedup crescente** até certo ponto.
- Para 2 a 4 threads, o ganho de desempenho é significativo.
- A partir da quinta thread, os ganhos **diminuem ou até retrocedem**, indicando saturação ou sobrecarga.

Fatores que Limitam a Escalabilidade

Proporção de Cômputo e Comunicação

- O **cômputo** (aplicação das regras do Jogo da Vida) domina o tempo de execução para cargas maiores.
- No entanto, a **comunicação implícita entre threads** e a **sincronização de bordas** gera overhead, especialmente com mais threads.

Granularidade da Paralelização

- Se o número de colunas por thread for pequeno (poucas iterações ou grade pequena), o **overhead de gerenciamento dos threads supera os benefícios** da execução paralela.
- Isso é evidenciado pelo comportamento irregular com 5 threads — em que o tempo aumenta.

7. Conclusão

A partir da análise dos dados de desempenho obtidos com diferentes números de iterações e threads, podemos tirar as seguintes conclusões sobre o comportamento do algoritmo paralelizado:

A implementação com OpenMP foi capaz de **reduzir significativamente o tempo de execução** em comparação com a versão serial. Os **maiores ganhos de performance** ocorreram quando passamos de 1 para 2 e 4 threads — indicando que a maior parte do código é **altamente paralelizável**.

A partir de 5 threads, o desempenho começa a **estagnar ou regredir** em algumas cargas, revelando os efeitos do **overhead de paralelismo**, como: sincronização entre threads, desbalanceamento de carga.Isso mostra que há um **ponto ótimo de paralelismo**, além do qual os custos superam os benefícios.

Cargas maiores (mais iterações) mostram **maior eficiência paralela**. Isso ocorre porque, com mais trabalho a ser realizado, o **custo de comunicação e sincronização se dilui** em relação ao tempo de computação. Ou seja, o paralelismo é **mais vantajoso com cargas intensas**.

O algoritmo apresenta uma **escalabilidade razoável** até 4 threads, com **speedups entre ~4.5 e ~6.8x**, dependendo da carga. O comportamento observado segue a tendência esperada pela **Lei de Amdahl**, que impõe limites ao ganho conforme a fração não paralelizável aumenta.

References

Implementing Conway’s Game of Life in C, Cobb Codin. 2024