Game of Life: Implementação Serial

José Alan Teixeira[1], Nelson Cerqueira[2]

[1]Programa de Pós-graduação em Ciências da Computação

Universidade estadual de Feira de Santana (UEFS)
Feira de Santana – BA – Brasil

{joseasteixeira7, nelsonccnetoab}@gmail.com

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

3.1. 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.

3.2. 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.
load_grid_from_file: Ler o estado inicial da grade de um arquivo em texto.
Caso não seja encontrado o arquivo de texto com o estado inicial a função init_grid é chamada.

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

free_grid: Libera corretamente a memória alocada pela grade.

3.3. logic.c - Regras do Jogo da Vida

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

Funções:

3.4. 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

3.5. 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

A versão paralela foi desenvolvida a partir do código da versão serial através da utilização do OpenMP, as alterações foram realizadas em três ponto do código, que são: no arquivo logic.c o código foi paralelizado primeiro no momento de criação de uma grade temporária e depois no local mais importante do jogo, no qual a grade é dividida em quatro grupos de colunas e esses grupos são distribuídos de forma estática entre os threads, quando existir mais de um thread. O código também foi paralelizado no arquivo grid.c quando a grade é alocada dinamicamente com base nas dimensões informadas.

Nesse caso a arquitetura utilizada na paralelização foi a Phase Parallel Paradigm, na qual o computo de cada pacote de colunas acontece independentemente, e ao fim todos são sincretizados antes de iniciar outro computo.

5. Arquivos auxiliares

5.1. Arquivos de entrada

- Config.txt - fornrçe parâmetros de dimensões da grade, quantidade de interações, intervalo para captura da imagem e quantidade de threads que serão usados no caso da versão paralelizada.
- Estado_inicial – fornece um estado inicial para as células vivas e mortas, caso esse arquivo não seja disponibilizado o estado inicial é gerado de forma aleatória.

5.2. Arquivos de saída

- Tempos_parallel ou Tempos_serial – guardam os tempos gastos na execução na execução do código.
- Imagens portable bitmap – são geradas e armazenadas conforme o intervalo indicado.

5.3. Arquivo de execução

- Makefile – usado para automatizar a execução, caso esse arquivo seja usado não é necessário o arquivo de configuração em formato de texto, na versão serial é necessário definir quantas vezes o código será executado e quais as quantidades de interações em cada simulação, na versão paralelizada também é preciso definir para quantas threads os testes serão executados.

5.4. Como executar

Usando o compilador gcc:

- Caso use as configurações por meio do arquivo de texto deve ser executado o comando " gcc grid.c image.c logic.c main.c -o gol" para a versão serial ou "gcc -fopenmp grid.c image.c logic.c main.c -o gol" para a versão paralelizada, com isso será criado um arquivo executável chamado "gol", na sequência para executar

o arquivo executável usa-se o comando “./gol config.txt”, esse comado já informa o arquivo de texto com as configurações.

- Caso a execuções seja realizada através do Mekefile, basta usar o comando “make run”.

6. Medição de desempenho

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

- Cargas simuladas (número de colunas da grade): 100, 500, 1000, 5000 e 10000
- Dimensões fixas: 2000 X 2000
- Número de threads usados: 1, 2, 4 e 6.
- 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

Para todos os testes realizados a grade foi dividida em quatro blocos de coluna, ou seja, granularidade quatro.

A tabela 1, a seguir, mostra os tempos de execução para cada carga na versão serial do software; a tabela 2 mostra os tempos de execução em cada carga de trabalho do código paralelizado, e a tabela 3 mostra o cálculo do speedup.

Tabela 1. Tempo de execução por carga de trabalho código serial

Serial	
Carga	Tempo(s)
100	37,15
500	159,09
1000	305,13
5000	1.426,64
10000	2.827,55

Tabela 2. Tempo de execução por carga de trabalho código paralelizado

Parelelizado tempos(s)				
Carga	1 thread	2 threads	4 threads	6 threads
100	36,82	19,76	11,94	14,21
500	160,95	84,98	57,60	59,17
1000	310,35	163,64	111,09	118,18
5000	1529,04	781,14	522,56	532,20
10000	2892,41	1537,81	1032,50	1052,35

Tabela 3. Speedup

Speedup				
Carga	1 thread	2 threads	4 threads	6 threads
100	1,01	1,88	3,11	2,61
500	0,99	1,87	2,76	2,69
1000	0,98	1,86	2,75	2,58
5000	0,93	1,83	2,73	2,68
10000	0,98	1,84	2,74	2,69

Tabela 3. Tempo de execução por granularidade

Carga	4 blocos	12 blocos
100	11,94	16,36
500	57,60	59,92
1000	111,09	112,96
5000	522,56	530,50
10000	1032,50	1044,98

A tabela 4 compara os tempos de execução para as cargas aplicadas, variando a quantidade de blocos de colunas que a grade é dividida, a tabela mostra os tempos quando se utiliza a granularidade 4 e 12.

A seguir é apresentado o gráfico 1 mostrando o comportamento entre o tempo de execução em segundos e a carga de trabalho, do código serial. E o gráfico 2 mostra a relação entre speedup e a quantidade de threads para cada carga de trabalho.

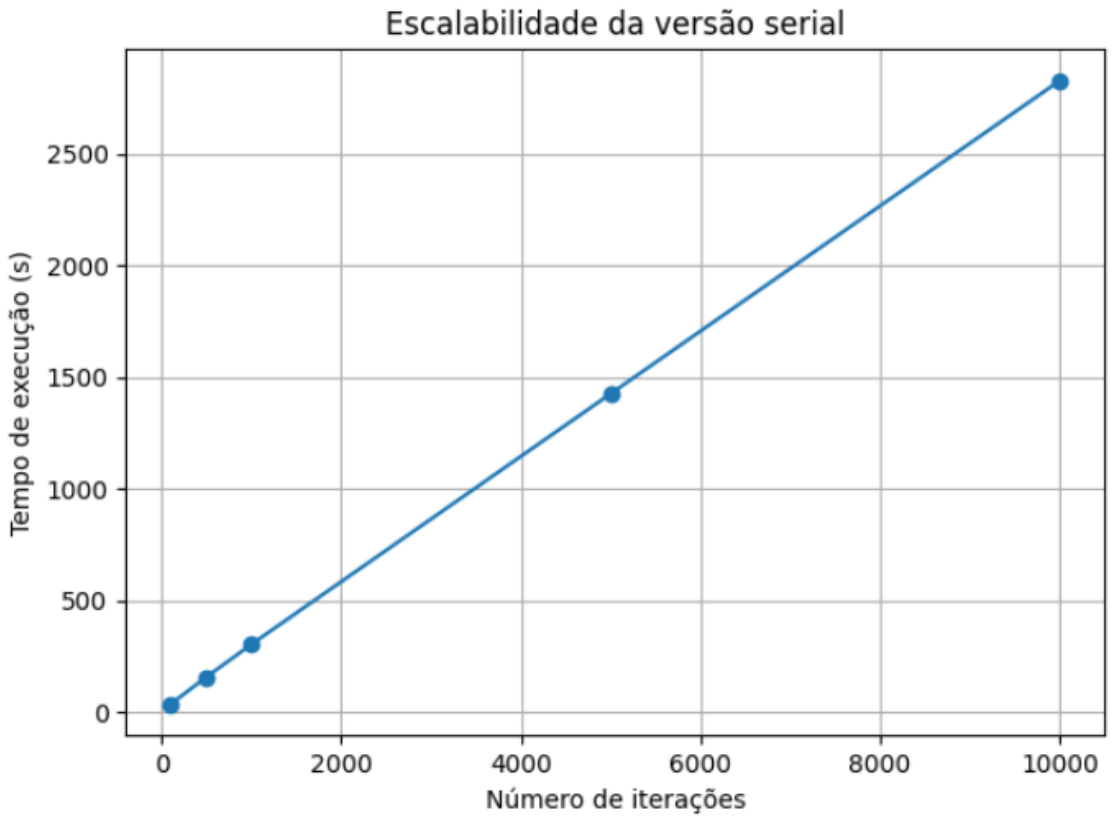

Figure 1. Gráfico com a relação entre tempo de execução e número de iterações.

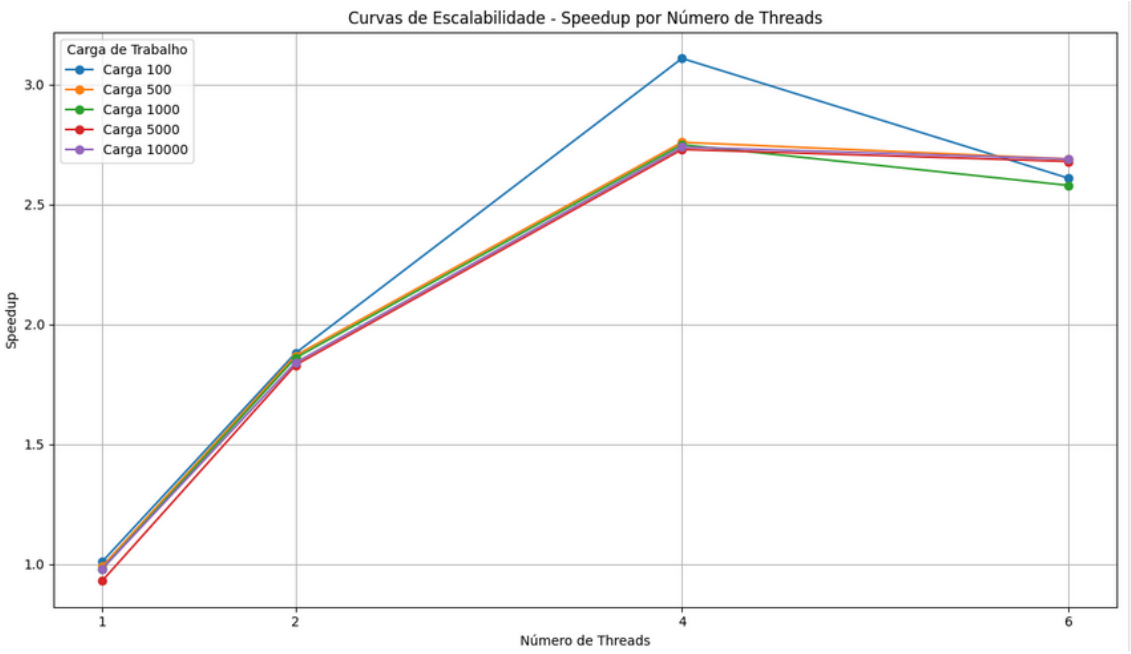

Figure 2. Gráfico com speedup para as três cargas de trabalho

7. Análise da Escalabilidade

A escalabilidade de uma aplicação paralela descreve como seu desempenho melhora à medida que mais recursos de hardware (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 e 4 threads, o ganho de desempenho é significativo.
- A partir da quinta thread, os ganhos diminuem, indicando saturação ou sobrecarga.

Fatores que Limitam a Escalabilidade

1. 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.

2. Granularidade da Paralelização

- Se o número de blocos de colunas (granularidade) for grande o desempenho comeca a diminuir, o overheadde gerenciamento dos threads supera os benefícios da execução paralela.
- Isso é evidenciado na tabela 4 que compara o tempo de execução usando 4 threads e variando a granularidade.

8. 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 se utilizou 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 quando se utiliza 4 threads, e com um número maio de threads 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.

Em resumo, os resultados evidenciam que a paralelização com OpenMP é eficaz, mas seu desempenho depende fortemente da quantidade de threads e da carga de trabalho. O algoritmo alcança seu melhor aproveitamento com até 4 threads, demonstrando que

existe um limite prático para o paralelismo eficiente, além do qual os ganhos se tornam marginais ou negativos. Esse comportamento reforça a importância de se considerar cuidadosamente tanto o grau de paralelização quanto o perfil da aplicação ao otimizar programas para execução concorrente.

References

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