Jogo da Vida: Paralelismo, Distribuição e Elasticidade

1. INTRODUÇÃO

Este relatório técnico explora a implementação do Jogo da Vida de John Conway, com ênfase em computação paralela e distribuída. O objetivo principal é investigar e comparar o desempenho de diferentes abordagens de execução: OpenMP/MPI e Apache Spark. Para tal, foi projetada uma arquitetura cliente-servidor robusta, utilizando contêineres Docker e preparada para orquestração com Kubernetes.

Este projeto integra-se a uma pesquisa mais ampla sobre "Construindo aplicações de larga escala com frameworks de programação paralela/distribuída". Buscamos proporcionar uma experiência prática, abordando desafios relacionados a performance e elasticidade em sistemas distribuídos. A intenção é otimizar a velocidade através do paralelismo e garantir que a aplicação se adapte de forma autônoma às variações de carga de trabalho.

O Jogo da Vida foi escolhido devido à sua simplicidade conceitual e facilidade de visualização, permitindo focar na engenharia de software para paralelismo e distribuição. Este documento detalha as etapas de desenvolvimento, os obstáculos superados e as análises comparativas que avaliam a eficácia de cada tecnologia empregada.

2. METODOLOGIA

Nosso processo de desenvolvimento foi estruturado em etapas sequenciais, integrando gradualmente conceitos e tecnologias de computação paralela e distribuída. A arquitetura central da aplicação permite que múltiplos clientes se conectem a um servidor de *sockets*. Este servidor atua como intermediário, comunicando-se com os "motores" de processamento (*engines*) de *backend*, responsáveis pelo cálculo das gerações do Jogo da Vida.

Inicialmente, planejamos integrar um banco de dados ElasticSearch para coletar métricas de performance. Embora essa integração não tenha sido totalmente finalizada devido a restrições de tempo e desafios técnicos imprevistos, a arquitetura permanece flexível para futuras adições.

A arquitetura foi concebida para atender tanto aos requisitos de performance, utilizando Apache Spark e OpenMP/MPI, quanto aos de elasticidade, com a premissa de orquestração via Kubernetes.

O servidor de *sockets*, desenvolvido em Python, emprega múltiplas *threads* para lidar com diversas conexões de rede simultaneamente. Essa capacidade é essencial para simular um ambiente de carga real e demonstrar a escalabilidade horizontal da camada de comunicação. A metodologia modular adotada facilitou a depuração e o teste independente de cada componente antes da integração final. Adicionalmente, a conteinerização com Docker foi um pilar desde o início, assegurando portabilidade, isolamento e uma implantação consistente, preparando o terreno para a orquestração com Kubernetes.

3. REQUISITO DE PERFORMANCE

O requisito de performance foi um dos pilares deste projeto, e o abordamos implementando, otimizando e comparando o Jogo da Vida com diversas abordagens de paralelismo e distribuição. Cada etapa foi pensada para extrair o máximo de desempenho de diferentes paradigmas computacionais.

3.1. ETAPA 1: JOGO DA VIDA SEQUENCIAL EM C

Começamos com a reconstrução de uma implementação básica do Jogo da Vida em C, no arquivo jogodavida.c. Essa versão sequencial foi nossa linha de base, essencial para validar a lógica do algoritmo e servir de referência para futuras comparações de desempenho. Ela nos permitiu, por exemplo, testar o comportamento de padrões como o "veleiro" para diferentes tamanhos de tabuleiro, confirmando que o algoritmo fundamental estava perfeito antes de introduzirmos as complexidades do paralelismo.

A função UmaVida itera sobre cada célula do tabuleiro, calculando o número de vizinhos vivos e aplicando as regras clássicas do Jogo da Vida para determinar o estado da célula na próxima geração. A função ind2d(i,j) é uma utilidade crucial para mapear coordenadas 2D para um índice 1D em um array linear.

Trecho da função UmaVida (original):

```
void UmaVida (int* tabulIn, int* tabulout, int tam) {
    int i, j, vizviv;
    for (i=1; i<=tam; i++) {
        for (j=1; j<=tam; j++) {
            vizviv = tabulIn [ind2d(i-1,j-1)] + tabulIn[ind2d(i-1,j)] +
                     tabulIn [ind2d(i-1,j+1)] + tabulIn [ind2d(i,j-1)] +
                     tabulIn[ind2d(i,j+1)] + tabulIn [ind2d(i+1,j-1)] +
```

```
                    tabulIn [ind2d(i+1,j)] + tabulIn [ind2d(i+1,j+1)];
            if (tabulIn[ind2d(i,j)] && vizviv < 2)
                tabulout[ind2d(i,j)] = 0;
            else if (tabulIn [ind2d(i,j)] && vizviv > 3)
                tabulout[ind2d(i,j)] = 0;
            else if (!tabulIn [ind2d(i,j)] && vizviv == 3)
                tabulOut [ind2d(i,j)] = 1;
            else
                tabulout[ind2d(i,j)] = tabulIn[ind2d(i,j)];
        }
    }
}
```

Dificuldades e Soluções: Nenhuma.

3.2. ETAPA 2: OTIMIZAÇÃO COM OPENMP

Nesta etapa, o código C foi adaptado para explorar o paralelismo por *threads*, utilizando a API OpenMP. A estratégia adotada foi paralelizar o laço externo (for (i=1; i<=tam; i++)) da função UmaVida, distribuindo as iterações entre as *threads* disponíveis. Essa abordagem é eficaz porque o cálculo de uma linha é amplamente independente das outras (com acessos de leitura ao tabulIn garantindo ausência de *race conditions*).

A diretiva #pragma omp parallel for instrui o compilador a distribuir as iterações. A cláusula private (j, vizviv) é crucial para garantir que j e vizviv sejam privadas a cada *thread*, evitando condições de corrida.

Trecho adaptado com OpenMP:

```
void UmaVida (int* tabulIn, int* tabulout, int tam) {
    int i, j, vizviv;
    #pragma omp parallel for private (j, vizviv)
    for (i=1; i<=tam; i++) {
        for (j=1; j<=tam; j++) {
            vizviv = tabulIn [ind2d(i-1,j-1)] +
                     tabulIn [ind2d(i-1,j)] +
                     tabulIn [ind2d(i-1,j+1)] + tabulIn
                     [ind2d(i,j-1)] +
                     tabulIn [ind2d(i,j+1)] + tabulIn
                     [ind2d(i+1,j-1)] +
                     tabulIn [ind2d(i+1,j)]+tabulIn
                     [ind2d(i+1,j+1)];
            if (tabulIn [ind2d(i,j)] && vizviv < 2)
                tabulout [ind2d(i,j)] = 0;
            else if (tabulIn [ind2d(i,j)] && vizviv > 3)
                tabulout [ind2d(i,j)] = 0;
            else if (!tabulIn [ind2d(i,j)] && vizviv == 3)
```

```
                tabulout [ind2d(i,j)] = 1;
            else
                tabulout [ind2d(i,j)] = tabulIn [ind2d(i,j)];
        }
    }
}
```

Dificuldades e Soluções: A principal dificuldade foi a incompatibilidade do compilador Clang padrão com as diretivas OpenMP. A solução foi instalar a *toolchain* LLVM completa com suporte a OpenMP via Homebrew, permitindo a compilação e execução paralela.

3.3. ETAPA 3: MODELO HÍBRIDO COM MPI + OPENMP

A etapa mais avançada em paralelismo de baixo nível envolveu a implementação de uma versão híbrida que combinasse Message Passing Interface (MPI) para paralelismo de memória distribuída e OpenMP para paralelismo de memória compartilhada. Esta abordagem visa aproveitar as vantagens de ambos os modelos.

O tabuleiro foi logicamente particionado entre os processos MPI, com cada processo responsável por um subconjunto de linhas. Para garantir o cálculo correto nas bordas, foi implementada uma troca de "linhas fantasmas". A comunicação foi realizada utilizando MPI_Send e MPI_Recv, garantindo a consistência dos dados nas bordas antes de cada iteração.

Trecho da lógica de troca de *ghost rows* (simplificado):

```
if (rank == 0) {
    MPI_Send(&tabulIn [ind2d(tam, 1)], tam, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
    MPI_Recv(&tabulIn [ind2d(tam + 1, 1)], tam, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if (rank == num_procs - 1) {
    MPI_Recv(&tabulIn [ind2d(0, 1)], tam, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&tabulIn [ind2d(1, 1)], tam, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);
} else {
    MPI_Send(&tabulIn [ind2d(tam, 1)], tam, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
    MPI_Recv(&tabulIn [ind2d(0, 1)], tam, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&tabulIn [ind2d(tam + 1, 1)], tam, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&tabulIn [ind2d(1, 1)], tam, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);
}
```

Dificuldades e Soluções: Uma dificuldadefoi o gerenciamento do escopo de variáveis e a inicialização de sub-tabuleiros para cada processo MPI. Foi necessário ajustar a assinatura da função InitTabul para receber parâmetros como rank e num_procs, permitindo a inicialização adequada dos dados locais.

3.4. ETAPA 4: ENGINE APACHE SPARK

Nesta etapa, a implementação do Jogo da Vida foi portada para o ambiente Apache Spark, utilizando a API PySpark. O objetivo foi explorar um paradigma de computação distribuída de alto nível, voltado para processamento de grandes volumes de dados e tolerância a falhas. O tabuleiro foi modelado como RDDs (Resilient Distributed Datasets), a abstração fundamental de dados do Spark.

As operações para calcular a próxima geração foram realizadas através de flatMap, groupByKey e map. A iteratividade do Jogo da Vida se beneficia do cache de RDDs (.cache()), que mantém os dados em memória através das iterações, reduzindo a necessidade de recomputação.

É importante notar que esta implementação não funcionou completamente em um ambiente de *cluster* distribuído real, sendo testada e validada apenas em modo local.

Trecho conceitual em PySpark:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("GameOfLifeSpark").getOrCreate()
initial_board_rdd = spark.sparkContext.parallelize([
    ((1, 1), 0),
    ((1, 2), 1),
    ((1, 3), 0),
    ((2, 1), 0),
    ((2, 2), 0),
    ((2, 3), 1),
    ((3, 1), 1),
    ((3, 2), 1),
    ((3, 3), 1)
])
```

Dificuldades e Soluções: As principais dificuldades foram problemas com o gerenciamento de pacotes Python (pip) e a lentidão percebida com grandes volumes de dados em uma máquina local. Para o problema do pip, a solução envolveu o uso de ambientes virtuais e a correta instalação das dependências. Para a performance local, foi reconhecido que o ambiente local serve principalmente para validação funcional, e a performance do Spark seria plenamente observável em um *cluster* real.

4. REQUISITO DE ELASTICIDADE

O requisito de elasticidade, a capacidade de um sistema se adaptar a cargas de trabalho variáveis através do escalonamento de recursos, foi abordado primariamente através da containerização da aplicação com Docker e da implementação de um servidor de *sockets* em Python. Embora a orquestração completa com Kubernetes fosse o objetivo final, a validação de seus componentes fundamentais foi realizada.

A containerização com Docker oferece portabilidade e isolamento, empacotando a aplicação e suas dependências em uma unidade executável leve. Isso simplifica a implantação e garante consistência em diferentes ambientes. O servidor *socket* Python, projetado como o ponto de entrada da aplicação, foi desenvolvido para gerenciar múltiplas conexões de clientes simultaneamente.

O projeto previa a integração direta do *backend* de processamento (C/MPI/OpenMP ou Spark) com o *socket server* dentro dos contêineres Docker, e posteriormente, a adoção de Kubernetes para a orquestração, auto-escalonamento e balanceamento de carga dessas instâncias de servidor.

Entretanto, a integração completa dos *backends* de processamento intensivo (C/MPI e Spark) com o *socket server* dentro dos contêineres Docker não foi totalmente funcional devido a desafios técnicos.

Trecho do servidor simulado (server_final.py):

```
import socket
import threading
HOST = '0.0.0.0'
PORT = 65432
def handle_client(conn, addr):
    print(f"Conectado por {addr}")
    with conn:
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = f"Simulando resposta para: {data.decode()}"
            conn.sendall (response.encode())
    print(f"Conexão com {addr} encerrada")
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print(f"Servidor escutando em {HOST}: {PORT}")
    while True:
        conn, addr = s.accept()
```

```
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()
```

Dificuldades e Soluções: A principal dificuldade na etapa de elasticidade e containerização foi a integração direta dos *backends* de performance (especialmente o em C/MPI) com o servidor de *sockets* Python dentro de uma única imagem Docker (python:3.11-slim). Problemas como a ausência de bibliotecas cruciais (e.g., libopenmpi-dev) e conflitos de portas dificultaram a compilação e execução.

Para contornar esses impedimentos, a solução foi desenvolver um servidor simulado (server_final.py). Este servidor, em vez de chamar o *backend* C/MPI, respondia com uma *string* simulada. Isso foi crucial para validar a arquitetura cliente-servidor e testar a comunicação via *sockets*, demonstrando a capacidade do servidor de lidar com múltiplas conexões simultâneas. Este passo foi fundamental para prosseguir com a validação funcional da arquitetura de elasticidade proposta.

A arquitetura final prevê que as instâncias da aplicação servidora sejam configuradas como contêineres leves e efêmeros, gerenciados pelo Kubernetes. Isso garantiria a orquestração, o escalonamento automático, a recuperação de falhas e o balanceamento de carga, elementos essenciais para uma aplicação verdadeiramente elástica e robusta.

5. ANÁLISE DE RESULTADOS E COMPARAÇÃO

Para avaliar o requisito de performance, comparamos os tempos de execução das diferentes implementações do Jogo da Vida. Devido à impossibilidade de integrar um banco de dados ElasticSearch para coleta e análise automatizada de métricas, os tempos de execução foram coletados manualmente durante os testes locais. É importante ressaltar que esses valores são aproximados e foram obtidos em um ambiente controlado, o que fornece uma base sólida para a comparação relativa entre as abordagens.

Cenário de Teste: Todos os testes foram realizados com um tabuleiro de tamanho 2^{10} x 2^{10} células (1024x1024 células) e simulando 2 * (tam-3) gerações, correspondendo a aproximadamente 2042 gerações. A infraestrutura utilizada foi planejada para minimizar variáveis externas, com os testes calibrados em uma máquina com configuração consistente para evitar conclusões influenciadas por diferentes ambientes de execução. A máquina utilizada possuía um processador multi-core e memória suficiente para as operações.

Implementação	Tempo de Execução (aproximado)	Observações
Jogo da Vida Sequencial em C	~35.0 segundos	Linha de base de performance, com alta complexidade de tempo para tabuleiros grandes em modelo unithread.
C com OpenMP	~10.0 segundos	Redução significativa do tempo de execução (~3.5x mais rápido) com paralelismo por *threads* (ex: 4 *threads*). Benefício direto da exploração de múltiplos núcleos em uma única máquina, aproveitando a memória compartilhada.

Implementação	Tempo de Execução (aproximado)	Observações
MPI + OpenMP (Híbrido)	~5.0 segundos	Maior ganho de performance (~2x mais rápido que OpenMP, ~7x mais rápido que sequencial). Obtido com múltiplos processos MPI e *threads* OpenMP por processo (ex: 4 processos MPI, 2 *threads* OpenMP por processo). A combinação de comunicação entre processos (MPI) e paralelismo intra-processo (OpenMP) se complementam bem, minimizando o overhead.
Apache Spark (PySpark Local)	~20.0 segundos	Maior sobrecarga de inicialização e gerenciamento de RDDs em ambiente local. A performance, embora razoável, não superou as implementações em C de baixo nível em um único nó. O foco é na escalabilidade horizontal e tolerância a falhas, não necessariamente na performance bruta.

Discussão:

- **OpenMP:** A introdução do OpenMP resultou em uma melhoria substancial de performance, demonstrando a eficácia do paralelismo de memória compartilhada em um único nó. A redução do tempo de execução foi proporcional ao número de *threads* disponíveis e à natureza do problema, permitindo paralelização quase linear. O overhead é mínimo, ideal para otimizações em uma máquina.
- **MPI + OpenMP (Híbrido):** O modelo híbrido apresentou a melhor performance. A divisão de trabalho entre processos (MPI) distribuiu a carga entre diferentes espaços de memória, enquanto o paralelismo interno com *threads* (OpenMP) otimizou o uso dos núcleos. A troca de "*ghost rows*" entre processos MPI foi crucial para manter a consistência dos dados nas bordas, e sua implementação eficiente minimizou o overhead de comunicação.
- **Apache Spark:** Embora o Spark não tenha superado as implementações em C (MPI/OpenMP) em ambiente de máquina única, sua arquitetura é projetada para escalabilidade horizontal em *clusters* distribuídos. A sobrecarga de inicialização do ambiente Spark e operações de RDDs introduzem um overhead perceptível em ambientes pequenos. Contudo, este overhead é compensado pela capacidade do Spark de processar volumes massivos de dados em *clusters*, oferecendo tolerância a falhas e um modelo de programação de alto nível. A elasticidade é um ponto forte inegável do Spark, permitindo adicionar ou remover nós dinamicamente.

6. CONCLUSÃO

O desenvolvimento deste projeto proporcionou uma oportunidade valiosa para explorar em profundidade diversas técnicas de paralelismo, comunicação entre processos e orquestração de aplicações em um cenário prático. O experimento permitiu não só validar o uso dessas técnicas em um problema real como o Jogo da Vida, mas também consolidar nosso entendimento sobre os *trade-offs* inerentes em termos de performance, complexidade de desenvolvimento e escalabilidade.

Em termos de performance bruta para o cenário de teste em um único nó, o modelo híbrido MPI + OpenMP se destacou, alcançando os melhores tempos de execução. Sua capacidade de combinar a distribuição de trabalho entre processos com o paralelismo de *threads* dentro de cada processo provou ser altamente eficaz para otimizar o uso de recursos computacionais. Em contraste, o Apache Spark, embora apresentasse um desempenho inferior em um ambiente local devido ao seu *overhead* de inicialização e gerenciamento de tarefas, demonstrou sua inerente escalabilidade e flexibilidade. O Spark é, sem dúvida, a escolha preferencial para lidar com grandes

volumes de dados em contextos distribuídos e para aplicações que exigem alta tolerância a falhas e elasticidade para se adaptar a cargas de trabalho variáveis.

A etapa de containerização da aplicação com Docker e a criação de um servidor simulado foram cruciais para validar a arquitetura cliente-servidor e para realizar testes funcionais, mesmo diante das limitações técnicas que impediram a integração completa do *backend* C/MPI ao *socket server* dentro do contêiner. Essa abordagem modular nos permitiu isolar e superar os desafios sequencialmente.

Este projeto reforça a importância de escolher a ferramenta ou *framework* de paralelismo/distribuição mais adequado para cada problema específico. A decisão deve levar em conta não apenas os requisitos de performance instantânea, mas também a necessidade de elasticidade, tolerância a falhas, e a facilidade de desenvolvimento e implantação em ambientes de produção. A containerização com Docker e a orquestração com Kubernetes são, sem dúvida, passos fundamentais e essenciais para a construção e implantação de aplicações robustas, escaláveis e resilientes em ambientes de produção modernos e complexos.

7. COMENTÁRIOS PESSOAIS

Murilo Perazzo: Participar deste projeto foi um desafio significativo, mas proporcionou um aprendizado imenso e aprofundado sobre os diversos paradigmas da computação paralela e distribuída. A complexidade do Jogo da Vida, combinada com a busca por otimização de performance, me obrigou a mergulhar de cabeça em cada tecnologia que usamos.

As etapas envolvendo OpenMP e MPI foram boas, embora eu já tivesse uma base sólida por conta da disciplina, colocar a mão na massa nesse trabalho foi excelente. Por outro lado, a experiência com Apache Spark abriu um novo universo de possibilidades para o processamento de dados em larga escala e me mostrou a complexidade inerente a frameworks de alto nível, percebendo como as abstrações podem simplificar o desenvolvimento, mas com um certo custo em controle e *overhead*.

Minha principal contribuição se concentrou intensamente na implementação e depuração dos *engines* de performance em C (OpenMP/MPI), garantindo que o algoritmo estivesse correto e eficiente. Além disso, dediquei-me à adaptação do projeto para o ambiente de contêineres Docker. As dificuldades com a compatibilidade da imagem python:3.11-slim, a ausência de bibliotecas de desenvolvimento de MPI, e a integração do *backend* C com o *socket server* Python dentro do contêiner foram os pontos mais críticos e exigiram muito de mim. Esses desafios demandaram soluções criativas e um grande esforço de depuração, culminando na estratégia bem-sucedida

do servidor simulado para conseguirmos prosseguir com os testes de arquitetura e validação funcional.

Minha principal dificuldade acabou sendo achar um grupo. Eu cheguei a me juntar com mais duas pessoas mas uma desistiu da matéria e a outra se juntou a outro grupo. Embora não pudesse fazer o trabalho sozinho, segundo as regras do trabalho, dessa vez foi necessário ou ficaria sem nota.

Em retrospectiva, sinto que o projeto foi um sucesso notável em termos de aquisição de conhecimento e experiência prática com tecnologias de ponta, altamente relevantes para a área de sistemas paralelos e distribuídos. Os aprendizados sobre otimização de performance, comunicação interprocessos, os *trade-offs* entre diferentes paradigmas e os desafios práticos de implantação em ambientes conteinerizados foram inestimáveis e, com certeza, serão de grande valia em minha trajetória profissional.