Universidade de Brasília (UnB)
Faculdade do Gama
Engenharia de Software

Arthur de Melo Garcia - 190024950
Eliás Yousef Santana Ali - 190027088
Erick Melo Vidal de Oliveira - 190027355
Paulo Vítor Silva Abi Acl - 190047968

Jogo da Vida Escalável

Brasília
2024

1 INTRODUÇÃO

O presente projeto tem como objetivo desenvolver uma aplicação baseada no conceito do Jogo da Vida, integrando tecnologias que otimizem tanto sua performance quanto sua escalabilidade. Para alcançar esse objetivo, utilizamos o Spark com foco na otimização do desempenho computacional, explorando técnicas de processamento distribuído, e comparamos seu desempenho com a aplicação desenvolvida em MPI/OpenMP. Além disso, empregamos o Kubernetes para gerenciar a elasticidade da aplicação, permitindo que ela se adapte dinamicamente às variações de carga conforme a quantidade de usuários. Dessa forma, buscamos criar um software escalável e robusto, capaz de ajustar seus recursos de maneira eficiente frente a diferentes níveis de demanda.

Para garantir uma abordagem colaborativa e ágil no desenvolvimento deste projeto, o grupo optou por realizar encontros síncronos utilizando a plataforma Discord. Durante essas reuniões, foram discutidas as principais etapas do trabalho e estabelecida a divisão de tarefas, considerando as habilidades e áreas de especialidade de cada integrante. Esse formato permitiu uma distribuição eficiente das responsabilidades, assegurando que cada membro pudesse contribuir de maneira mais eficaz ao projeto, atuando nas áreas em que possuía maior domínio técnico. Além disso, grande maioria do desenvolvimento realizado no trabalho foi de forma síncrona, o que possibilitou o compartilhamento de conhecimento entre as frentes que cada um possuía afinidade.

2 METODOLOGIA

Este tópico tem a finalidade de apresentar algumas estratégias, decisões, ferramentas e tecnologias utilizadas pelos integrantes do grupo, e que de alguma forma contribuíram para a realização do presente trabalho.

2.1 REQUISITOS

O levantamento de requisitos não foi feito de forma profissional, como se espera de um software de maior escala. O grupo simplesmente dedicou-se a leitura do documento disponibilizado pelo professor, que já foi bem satisfatório em termos de explicação e objetivos desejados, e posteriormente discutimos as ideias compreendidas do objetivo.

2.2 PLANEJAMENTO

Uma vez que entendido o trabalho, o grupo desenvolveu um plano que atendesse a carga horária e conhecimento de cada um dos membros. Chegamos a conclusão que a melhor alternativa foi utilizar a plataforma Discord como comunicação principal, onde realizamos os encontros síncronos para desenvolver o trabalho como um todo, e o WhatsApp para uma comunicação mais rápida, como horário disponível entre os membros, dúvidas, dificuldades e etc.

Além disso, seguindo os principais requisitos descritos no próprio trabalho, o grupo fez uma divisão de componentes a serem desenvolvidos. Isso ajudou a entender por onde começar nessas pequenas partes para então construir algo maior com base nessas menores frações. A divisão escolhida foi:

- Gerenciamento de cluster (Kubernetes): Para orquestrar a infraestrutura e gerenciar a escalabilidade dos contêineres.
- Análise e monitoramento (ElasticSearch/Kibana): Configurado para monitorar o comportamento da aplicação, coletando métricas como número de clientes e tempo de execução.
- Comunicação em tempo real (WebSockets): Esta parte recebe os dados de entrada dos usuários e envia para o engine que calcula os estágios do jogo da vida.
- Execução paralela (OpenMP/MPI): Para construção de uma engine que combina OpenMP para paralelismo em memória compartilhada e MPI para paralelismo distribuído
- Persistência e armazenamento (Spark): Utilizará Spark para processar o jogo de maneira distribuída, lidando com grandes volumes de dados gerados pela simulação.

Após essa divisão, os integrantes partiram para o desenvolvimento e estruturação dos componentes, buscando se reunirem em duplas para evitar erros e retrabalho, além de minimizar as possibilidades de algum membro encontrar-se travado em alguma etapa do trabalho. No final do desenvolvimento dos componentes, nos reunimos em algumas chamadas síncronas para lidar com a integração de todas as partes.

3 REQUISITOS DE PERFORMANCE

A paralelização do sistema possui vários objetivos, entre elas a melhoria de desempenho da aplicação, melhor utilização de recursos disponíveis, escalabilidade e etc. Mesmo que em um escopo reduzido como o do trabalho apresentado, foi de extrema importância o desenvolvimento dessa etapa, para que os integrantes compreendessem a utilização e finalidade das tecnologias na prática.

3.1 Desenvolvimento com MPI e OpenMP:

O OpenMP e o MPI são duas abordagens complementares de paralelismo usadas para melhorar o desempenho de aplicações em sistemas de computação. O OpenMP é uma API projetada para sistemas com memória compartilhada, como computadores multicores, onde as tarefas podem ser divididas entre várias threads que compartilham o mesmo espaço de memória, tornando a paralelização simples e eficiente por meio de diretivas no código. Por outro lado, o MPI (Message Passing Interface) é ideal para sistemas distribuídos, onde cada processo possui sua própria memória e a comunicação ocorre via troca de mensagens entre processos localizados em diferentes nós de um cluster. Quando utilizados juntos, OpenMP e MPI oferecem um modelo de paralelismo híbrido, onde o OpenMP gerencia a execução paralela dentro de cada nó, enquanto o MPI facilita a comunicação entre os nós, permitindo que aplicações escalem eficientemente em clusters de alta performance e aproveitem o melhor de ambos os mundos: memória compartilhada e distribuída.

3.1.1 O que foi feito

No início, o programa recebe parâmetros para definir o tamanho do grid baseado em potências de dois. Cada processo então inicializa seu segmento do grid, configurando estados iniciais de células com algumas células vivas para começar a simulação. A função *UmaVida()* é usada para calcular as novas gerações baseadas nas regras do jogo. O OpenMP ajuda a acelerar esses cálculos ao paralelizar o processamento dentro de cada linha do grid.

Para garantir que os resultados sejam consistentes e precisos, a função *TrocaBordas(),* um exemplo claro do uso do MPI para sincronizar estados entre processos, é utilizada para garantir que os processos compartilhem corretamente o

estado das células que residem em suas fronteiras compartilhadas. Isso é essencial para manter a integridade do modelo de simulação em um ambiente distribuído.

3.1.2. Dificuldades

A principal dificuldade foi garantir a sincronização entre os processos MPI e a distribuição adequada do trabalho entre eles, de modo que cada processo manipule sua porção do tabuleiro de maneira eficiente, evitando sobrecarga de comunicação. Além disso, foi necessário gerenciar o uso correto de OpenMP para paralelizar eficientemente as operações dentro de cada processo sem criar sobrecarga extra.

3.1.3 Solução

Para resolver o problema de sincronização entre processos, o código utiliza o MPI_Comm_rank e MPI_Comm_size para garantir que cada processo calcule apenas sua parte do tabuleiro, de acordo com seu rank e o número total de processos (size). Para melhorar a comunicação, as partes do tabuleiro de borda são sincronizadas entre os processos usando mensagens MPI, garantindo consistência entre as fronteiras das partes. Já o uso de OpenMP permite que, dentro de cada processo, o cálculo das células seja feito de forma paralela, utilizando a diretiva #pragma omp parallel for, o que reduz o tempo de execução ao distribuir o cálculo de linhas do tabuleiro entre múltiplos threads. A combinação de MPI para comunicação entre processos e OpenMP para processamento paralelo dentro de cada processo proporciona uma solução eficiente para o Jogo da Vida em um ambiente de memória distribuída e compartilhada.

3.2 Desenvolvimento com Spark:

O Apache Spark é uma ferramenta de código aberto voltada para o processamento distribuído de grandes volumes de dados. Ele se destaca por sua alta performance, suportando operações em memória e permitindo o paralelismo eficiente de tarefas. Uma de suas características mais poderosas é a capacidade de distribuir operações em um cluster de nós de processamento, permitindo o processamento paralelo de grandes conjuntos de dados, o que é ideal para tarefas intensivas, como simulações computacionais.

3.2.1 O que foi feito

O código começa criando uma sessão Spark com seis núcleos disponíveis localmente, como definido pelo comando master("local[6]"). Em seguida, um intervalo de potências (usado para determinar o tamanho da matriz do jogo) é passado como argumento de linha de comando e distribuído entre as diferentes instâncias paralelizadas com o comando *sparkContext.parallelize()*. O Spark então distribui a execução da função *jogoVida* para cada valor de potência no intervalo logo então, a função *jogoVida* é responsável por inicializar o tabuleiro do jogo e executar as regras de evolução das células em cada iteração, além disso, após a execução de cada simulação, o tempo gasto é calculado e os dados são enviados para o Elasticsearch, uma ferramenta de análise e monitoramento em tempo real.

3.2.2 Dificuldades

Um dos desafios foi garantir que o paralelismo fosse feito de forma eficiente, garantindo que cada iteração do jogo fosse distribuída corretamente entre os núcleos disponíveis. O Spark facilita isso com a função parallelize, mas foi necessário ajustar o número de partições para evitar sobrecarga ou subutilização dos recursos.

3.2.3 Solução

O uso de *parallelize(potencias, len(potencias))* garantiu que cada simulação fosse processada em paralelo, distribuindo corretamente o trabalho entre os núcleos disponíveis.

4 REQUISITOS DE ELASTICIDADE

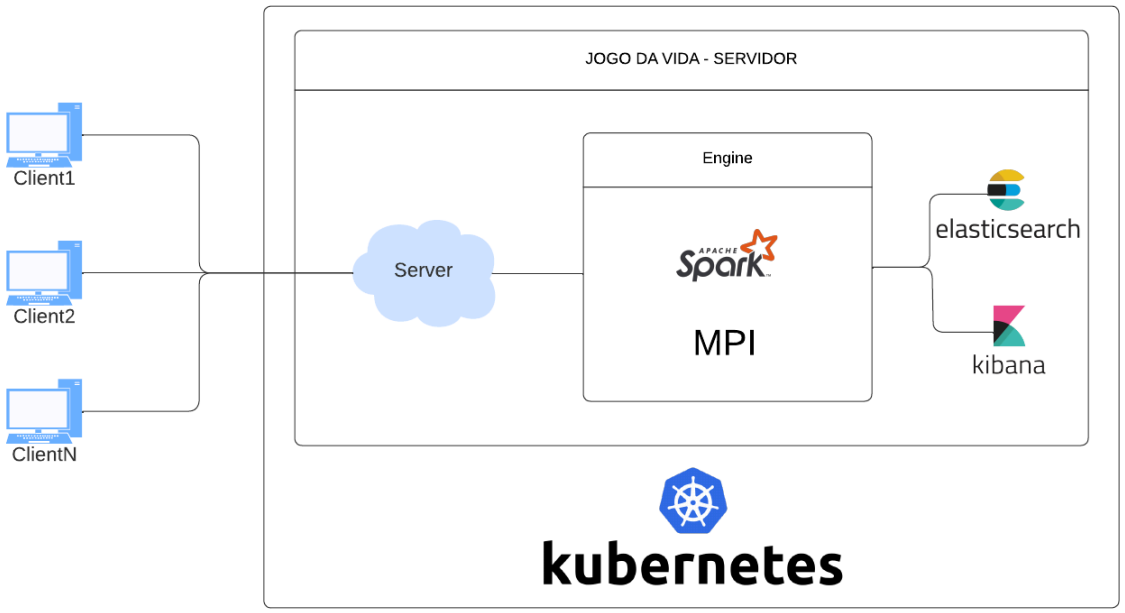
4.1 Kubernetes

Kubernetes é uma plataforma de orquestração de contêineres de código aberto, originalmente desenvolvida pelo Google, que automatiza o processo de implantação, escalabilidade e gerenciamento de aplicações conteinerizadas. O Kubernetes permite que desenvolvedores e operadores tratem o gerenciamento de aplicações em contêineres de forma eficiente, otimizando recursos de hardware e oferecendo a capacidade de escalar automaticamente quando a demanda aumenta.

Principais Componentes:

- Nó (Node): É uma máquina (física ou virtual) que executa as aplicações e está sob o gerenciamento do cluster Kubernetes. Cada nó pode rodar múltiplos contêineres.
- Pod: A menor unidade de implantação no Kubernetes. Um pod pode conter um ou mais contêineres que compartilham o mesmo ambiente de execução, incluindo armazenamento e rede.
- Controlador de Replicação: Garante que um número específico de réplicas de um pod esteja rodando a qualquer momento.
- Service: Define uma política para expor os pods ao tráfego externo (ou para comunicação interna dentro do cluster).
- Horizontal Pod Autoscaler (HPA): Um recurso que ajusta automaticamente o número de réplicas de pods com base no uso de CPU ou outras métricas configuradas.
- Namespace: Usado para isolar recursos dentro do cluster, facilitando o gerenciamento de múltiplas aplicações ou usuários.

O que foi feito: A aplicação foi configurada no Kubernetes com vários deployments e serviços para gerenciar diferentes componentes do sistema. Foram criados os seguintes deployments seguindo a arquitetura da imagem abaixo:



- **MPI-engine** para executar o serviço MPI/OpenMP, responsável por processar as tarefas paralelas da aplicação.

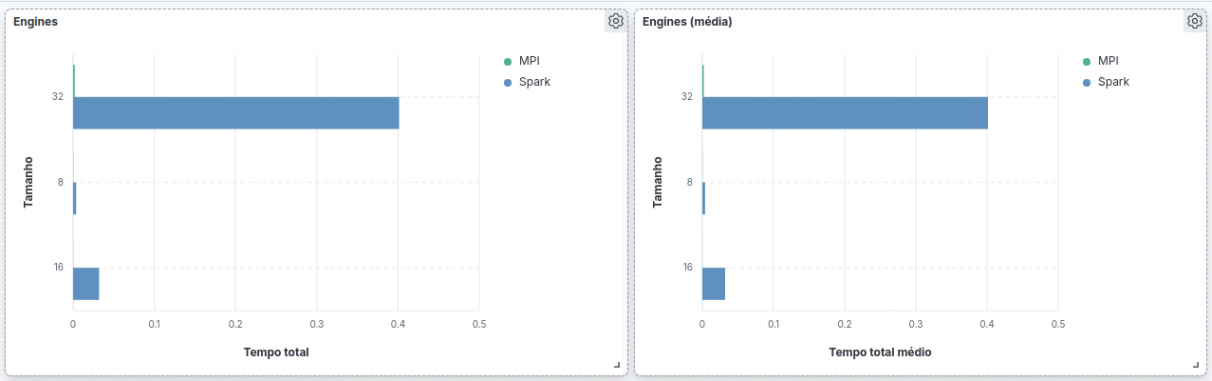
- **Spark-engine** para realizar o processamento distribuído das tarefas, aproveitando o paralelismo e a escalabilidade oferecidos pelo Spark.
- **TCP Server** para gerenciar as conexões e comunicações de rede necessárias para a aplicação.
- **Elasticsearch** para armazenamento e monitoramento de dados, utilizado para armazenar informações sobre o desempenho das execuções.
- **Kibana** para visualização de dados coletados pelo Elasticsearch, permitindo a análise e monitoramento das execuções..

Dificuldades: a principal dificuldade é que nenhum dos integrantes tinha experiência com a tecnologia, então essa definitivamente foi a parte mais demorada e trabalhosa do projeto desenvolvido. Dentro das dificuldades enfrentadas durante o trabalho, uma das maiores foi garantir a integração e comunicação eficaz entre os diferentes serviços do cluster Kubernetes, e sendo ainda mais específico, assegurar que o Spark e o Mpi pudesse se comunicar corretamente tanto com o server tcp quanto com o Elasticsearch.

Solução: os integrantes do grupo precisaram se inteirar acerca da tecnologia para concretizar sua utilização no projeto proposto, e decidiram fazer através de conteúdos avulsos na internet, principalmente no YouTube. Sobre a integração e comunicação entre os serviços do cluster, utilizamos o conceito de service discovery do kubernetes, permitindo que todos os serviços pudessem se localizar e comunicar entre si.

ANÁLISE DOS RESULTADOS

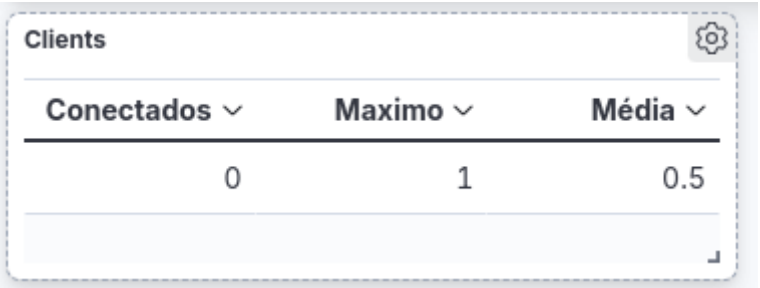
Tempo de Execução por Tamanho: No Kibana, o tempo de execução para diferentes tamanhos de entrada foi registrado e visualizado para cada engine (Spark, MPI, etc.). Um dos pontos mais significativos observados é que o código Spark apresenta tempos de execução consideravelmente maiores do que o código MPI, conforme mostra a imagem a seguir.



Isso é esperado devido às diferenças fundamentais entre as arquiteturas e abordagens de computação distribuída adotadas por cada engine:

- **MPI:** Projetado para alta eficiência em clusters de computação com comunicação direta entre processos. O MPI tem menos overhead de gerenciamento, o que o torna mais rápido para tarefas com comunicação intensa entre nós.
- **Spark:** Apesar de ser uma ferramenta poderosa para processar grandes volumes de dados, o Spark tem uma arquitetura mais pesada. Seu modelo de dados distribuídos (RDDs), a necessidade de gerenciamento de falhas e a serialização de dados em uma rede distribuída introduzem mais latência, o que justifica os tempos mais longos para a execução de tarefas quando comparado ao MPI, especialmente em cenários de menor volume de dados.

Conexões dos Clientes: O número de clientes conectados ao servidor também foi monitorado em tempo real e enviado para o Elasticsearch, permitindo visualizações em dashboards do Kibana conforme mostra a imagem abaixo.



Isso ajuda a identificar padrões de carga e volume de clientes ao longo do tempo. Monitorar essas métricas é fundamental para ajustar o escalonamento de recursos (scaling) no Kubernetes e garantir que a infraestrutura possa lidar com variações no número de clientes.

CONCLUSÃO

O desenvolvimento deste projeto proporcionou uma grande oportunidade para aplicar na prática conceitos de programação paralela e distribuída, além de explorar ferramentas robustas como MPI, OpenMP, Spark e Kubernetes. A combinação dessas ferramentas mostrou-se eficiente em diferentes contextos, com o MPI destacando-se em cenários com maior exigência de performance e comunicação direta entre processos, enquanto o Spark demonstrou ser uma solução mais flexível para o processamento de grandes volumes de dados, ainda que com maior overhead.

Dentre os maiores desafios enfrentados, destacamos a integração entre as diversas tecnologias e a curva de aprendizado acentuada, principalmente no uso do Kubernetes. Porém foi diante dessa dificuldade que nos possibilitou um aprendizado melhor e um maior domínio da ferramenta. A elasticidade e orquestração dos containers permitiram que o sistema se adaptasse a diferentes demandas, algo que não era tão simples de implementar manualmente.

Outro ponto importante foi a experiência de trabalhar de forma colaborativa e síncrona em boa parte do desenvolvimento, mesmo enfrentando algumas dificuldades de integração e comunicação entre os serviços, conseguimos, por meio de estudo autônomo e colaboração em grupo, superar os desafios e entregar uma solução funcional.

Em resumo, o projeto não apenas atendeu aos requisitos propostos, mas também ampliou nosso entendimento sobre como construir e gerenciar aplicações escaláveis, oferecendo uma base sólida para futuros trabalhos na área de computação distribuída e paralela.