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 Hadoop e o Spark com foco na otimização do desempenho computacional, explorando técnicas de processamento distribuído. 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.

2 METODOLOGIA

Primeiramente, o grupo se dedicou a dividir a aplicação em diferentes componentes. Isso ajudou a entender por onde começar nessas pequenas partes para então construir algo maior com base nessas menores frações. Sendo assim, essa foi a divisão escolhida:

- Gerenciamento de cluster (Kubernetes): Para orquestrar a infraestrutura e gerenciar a escalabilidade dos contêineres.
- Análise e monitoramento (ElasticSearch/Kibana): Para coletar logs e monitorar a performance da aplicação.
- Comunicação em tempo real (WebSockets): Para enviar atualizações de estado aos usuários em tempo real.

- Execução paralela (OpenMP/MPI): Para melhorar a performance computacional do Jogo da Vida.
- Persistência e armazenamento (Hadoop/Spark): Para lidar com grandes volumes de dados gerados pela simulação, distribuindo o processamento e otimizando a execução.

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 uma única chamada para lidar com a integração de todas as partes.

3 REQUISITOS DE PERFORMANCE

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.

- **O que foi feito**: O tabuleiro do Jogo da Vida é dividido entre os processos MPI, de forma que cada processo calcula sua respectiva parte do tabuleiro. Em cada parte, as células são processadas em paralelo usando OpenMP, aproveitando múltiplos threads por meio da diretiva #pragma omp parallel for.

- **Computação com OpenMP**: Cada processo MPI utilizou OpenMP para paralelizar o cálculo das células dentro de sua fatia do tabuleiro. A diretiva #pragma omp parallel for foi usada para distribuir as iterações do loop entre múltiplas threads, melhorando o desempenho dentro de cada processo.
- **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.
- **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 Hadoop e Spark:

O Hadoop e o Spark são duas tecnologias amplamente usadas para processamento distribuído de grandes volumes de dados, mas com abordagens distintas. O Hadoop é baseado no modelo de programação MapReduce, onde os dados são processados em disco de forma distribuída em clusters, oferecendo alta tolerância a falhas e escalabilidade horizontal, mas com latência maior devido ao uso intensivo de leitura e gravação em disco. Já o Spark se destaca pelo seu processamento em memória (in-memory), tornando-o muito mais rápido para operações iterativas ou em tempo real, ao manipular dados através de estruturas

chamadas RDDs (Resilient Distributed Datasets), mantendo também a escalabilidade em clusters. Enquanto o Hadoop é mais adequado para tarefas de processamento em batch e cenários onde a resiliência a falhas é crítica, o Spark é preferido para cenários de processamento rápido e análise em tempo real, proporcionando maior eficiência em muitas aplicações.

- **O que foi feito:** A aplicação paraleliza o cálculo do Jogo da Vida utilizando o Spark para processar diferentes tamanhos de tabuleiro em paralelo. O cálculo para cada potência é distribuído entre os nós do cluster Spark; A aplicação utiliza asyncio para gerenciar múltiplos clientes conectados ao servidor simultaneamente. Cada cliente pode enviar uma requisição com dois números, e a aplicação usa o Spark para processar essa faixa de potências de forma distribuída; O sistema também integra a aplicação com o Elasticsearch para armazenar e monitorar o desempenho de cada execução, incluindo o tempo total de processamento e o status do cálculo.
- **Dificuldades:** Um dos maiores desafios foi garantir que o cálculo do Jogo da Vida fosse distribuído corretamente entre os nós do Spark.
- **Solução:** Para garantir que o cálculo do Jogo da Vida seja distribuído corretamente entre os nós do Spark, é fundamental dividir o tabuleiro em partes menores e distribuí-las como partições para cada executor no cluster. Utilizar RDDs (Resilient Distributed Datasets) para representar o tabuleiro permite que cada parte do grid seja processada paralelamente em diferentes nós. A função de evolução do Jogo da Vida pode ser aplicada a cada partição usando a transformação mapPartitions, e a sincronização dos resultados pode ser realizada com ações como reduceByKey para combinar as partes processadas. Configurar corretamente os recursos do cluster e monitorar o desempenho através da Spark UI ajuda a garantir que a carga de trabalho seja equilibrada e o processamento seja eficiente.

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:

- **Elasticsearch** para armazenamento e monitoramento de dados, utilizado para armazenar informações sobre o desempenho das execuções.
- **OpenMP-app** para executar o serviço OpenMP, responsável por processar as tarefas paralelas da aplicação.
- **Kibana** para visualização de dados coletados pelo Elasticsearch, permitindo a análise e monitoramento das execuções.
- **TCP Server** para gerenciar as conexões e comunicações de rede necessárias para a aplicação.

- **Spark-app** para realizar o processamento distribuído das tarefas, aproveitando o paralelismo e a escalabilidade oferecidos pelo Spark.

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 Elasticsearch pudesse se comunicar corretamente com o Kibana e o OpenMP.

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

Primeiramente, tentamos rodar o código do jogo da vida localmente para verificar se houve alguma melhoria no desempenho. Estes foram os resultados obtidos:

Código padrão:

```
**RESULTADO CORRETO**
tam=8; tempos: init=0.0000138, comp=0.0000222, fim=0.0000188, tot=0.0000548
**RESULTADO CORRETO**
tam=16; tempos: init=0.0000050, comp=0.0002182, fim=0.0000041, tot=0.0002272
**RESULTADO CORRETO**
tam=32; tempos: init=0.0000079, comp=0.0021510, fim=0.0000150, tot=0.0021739
**RESULTADO CORRETO**
tam=64; tempos: init=0.0000451, comp=0.0163321, fim=0.0000319, tot=0.0164092
**RESULTADO CORRETO**
tam=128; tempos: init=0.0001299, comp=0.1487520, fim=0.0000560, tot=0.1489379
**RESULTADO CORRETO**
tam=256; tempos: init=0.0005040, comp=1.1695430, fim=0.0001838, tot=1.1702309
**RESULTADO CORRETO**
tam=512; tempos: init=0.0019040, comp=11.3651140, fim=0.0008821, tot=11.3679001
**RESULTADO CORRETO**
tam=1024; tempos: init=0.0093062, comp=107.7130208, fim=0.0035331, tot=107.7258601
```

Código com OMP/MPI:

```
**Ok, RESULTADO CORRETO**
tam=8; tempos: init=0.0000029, comp=0.0003331, fim=0.0000269, tot=0.0003629
**Ok, RESULTADO CORRETO**
tam=16; tempos: init=0.0000050, comp=0.0002449, fim=0.0000031, tot=0.0002530
**Ok, RESULTADO CORRETO**
tam=32; tempos: init=0.0000060, comp=0.0011141, fim=0.0000050, tot=0.0011251
**Ok, RESULTADO CORRETO**
tam=64; tempos: init=0.0000260, comp=0.0144899, fim=0.0003681, tot=0.0148840
**Ok, RESULTADO CORRETO**
tam=128; tempos: init=0.0001140, comp=0.2096231, fim=0.0001450, tot=0.2098820
**Ok, RESULTADO CORRETO**
tam=256; tempos: init=0.0004191, comp=1.1745338, fim=0.0002151, tot=1.1751680
**Ok, RESULTADO CORRETO**
tam=512; tempos: init=0.0020390, comp=13.1928880, fim=0.0009532, tot=13.1958802
**Ok, RESULTADO CORRETO**
tam=1024; tempos: init=0.0116329, comp=83.5254931, fim=0.0038409, tot=83.5409670
```

Com isso, conseguimos analisar que o tempo de execução com a versão paralelizada utilizando OMP e MPI em conjunto continuou praticamente o mesmo para os tamanhos menores da matriz, mas para uma matriz grande, de tamanho 1024, o tempo de resposta foi reduzido de 107s para 83s.

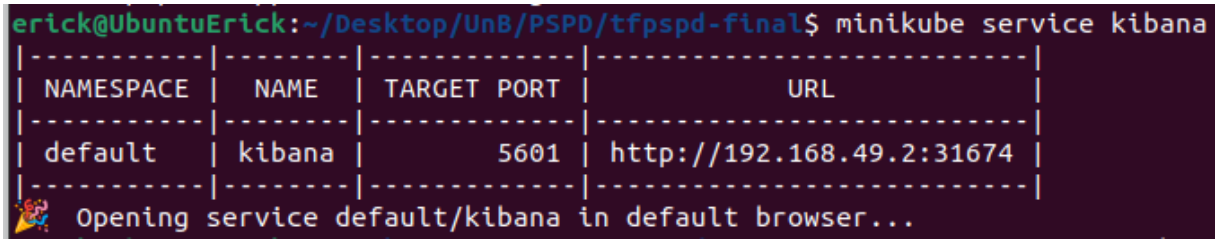
Em seguida, queríamos testar como ficaria o desempenho desta engine em relação a sua escalabilidade. Então criamos uma conexão via tcp para receber requisição de clientes da seguinte forma:

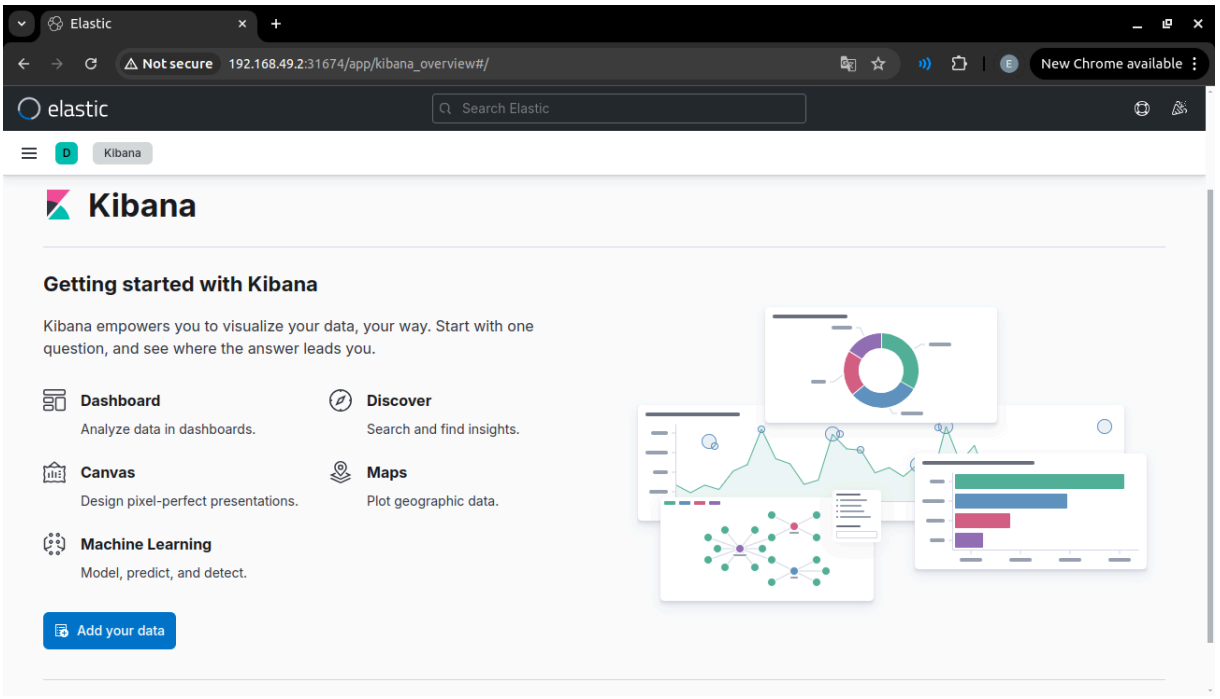
```
erick@UbuntuErick:~/Desktop/UnB/PSPD/tfpspd-final/socket$ python3 cliente.py
Digite a mensagem (ou 'sair' para sair): <3,5>
Digite a mensagem (ou 'sair' para sair):
```

```
erick@UbuntuErick:~/Desktop/UnB/PSPD/tfpspd-final/socket$ python3 cliente.py
Digite a mensagem (ou 'sair' para sair): <10,20>
Digite a mensagem (ou 'sair' para sair):
```

```
erick@UbuntuErick:~/Desktop/UnB/PSPD/tfpspd-final$ kubectl logs -f tcp-server-deployment-646b8f8769-lx5zt
Servidor iniciado em ('0.0.0.0', 8888)
Novo cliente conectado: ('10.244.0.1', 16406)
----------------------------MAPEANDO----------------------------
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_doc/1
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_doc
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_create/2
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_create/3
Mensagem recebida do cliente ('10.244.0.1', 16406): <3,5>
Novo cliente conectado: ('10.244.0.1', 9390)
----------------------------MAPEANDO----------------------------
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_doc/1
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_doc
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_create/2
Erro na requisição PUT: 400 Client Error: Bad Request for url: http://elasticsearch-service:9200/jogovida/_create/3
Mensagem recebida do cliente ('10.244.0.1', 9390): <10,20>
```

E também criamos o serviço kibana para ser utilizado como base de dados e exibir os resultados das execuções:





CONCLUSÃO

O desenvolvimento desta aplicação baseada no Jogo da Vida permitiu a integração de diferentes tecnologias buscando alcançar otimização de performance e elasticidade. A combinação de MPI e OpenMP proporcionou ganhos em termos de paralelismo, permitindo que a aplicação explorasse tanto memória compartilhada quanto distribuída.

Os desafios enfrentados, como a complexidade na comunicação entre os componentes do cluster Kubernetes e a falta de experiência prévia com algumas das tecnologias, dificultou a conclusão do projeto, não permitindo atingir os objetivos do trabalho.