

UNIVERSIDADE DE BRASÍLIA

Faculdade do Gama

Programação para Sistemas Paralelos e Distribuídos

**Trabalho Final**

**Construindo aplicações de larga escala  
com frameworks de programação paralela/distribuída**

Antonio Aldisio 20/2028211

Fernando Miranda Calil 19/0106566

Lorrany Oliveira Souza 18/0113992

Brasília, DF

2023

## 1. Introdução

De acordo com Lima e Oliveira, em 1968 foi criado um importante autômato celular, conhecido como Jogo da Vida, pelo matemático John Conway. O Jogo da Vida visa projetar um conjunto de regras matemáticas simples que é capaz de formar padrões complexos de vida. O jogo faz a simulação das gerações sucessivas de uma sociedade de organismos vivos, podendo ser construído em um tabuleiro de mais de uma dimensão, entretanto, neste projeto é bidimensional e infinito em qualquer direção. Nas regras do jogo, o estado de uma determinada célula é definido de acordo com o estado atual das células vizinhas, neste caso cada célula possui 8 células vizinhas. As regras do jogo se encontra abaixo:

- Células vivas com menos de 2 vizinhas vivas morrem por abandono
- Células vivas com mais de 3 vizinhas vivas morrem de superpopulação
- Células mortas com exatamente 3 vizinhas vivas tornam-se vivas
- As demais células mantêm seu estado anterior.

Ainda segundo os autores Lima e Oliveira, antigamente o Jogo da Vida era estudado e desenvolvido de diversas formas, fazendo normalmente o uso de quadros negros, papel e lápis entre outros, menos computador. Até que, por volta dos anos 70, analistas interessados resolveram utilizar os mainframes da IBM para criar programas que fossem aptos a reproduzir computacionalmente o jogo. Entretanto, o algoritmo precisava de muita capacidade computacional, sendo necessária uma noite inteira para sua execução.

Com isso, o presente projeto visa o objetivo de construir uma aplicação para ela se comportar como uma aplicação de larga escala atendendo os requisitos de performance e de escala. Pensando em atingir o objetivo deste trabalho, a implementação do algoritmo do Jogo da Vida foi escolhido.

## 2. Metodologia

A palavra Método vem da palavra grega *méthodos*, formada por duas palavras *metá* que significa no meio de; através, entre, acrescida de *odós*, que significa “caminho”. Assim, podemos dizer que Método significa ao longo do

caminho, ou seja, “forma de proceder ao longo de um caminho” (TRUJILLO FERRARI, 1982, p. 19).

Segundo a prof. Liane, na ciência, a palavra método significa a maneira que o cientista escolhe para estudar e ampliar os conhecimentos sobre determinado objeto de estudo.

Com isso, a metodologia utilizada para a pesquisa foi a exploratória, pois visa explorar e conhecer melhor os requisitos do projeto. É classificada como abordagem quantitativa pois visa analisar qual tem a melhor performance. Como procedimento de coleta de dados foi utilizado a pesquisa bibliográfica.

### **3. Requisito de performance**

O requisito de performance é um elemento essencial no desenvolvimento de sistemas e aplicações, pois estabelece os critérios e expectativas relacionados ao desempenho do software. Ele engloba diversos aspectos, como tempo de resposta, capacidade, escalabilidade e eficiência de recursos. O tempo de resposta determina o período máximo aceitável para a aplicação responder a uma requisição, garantindo uma experiência ágil e responsiva ao usuário. A capacidade estabelece a quantidade máxima de usuários ou transações que o sistema deve suportar simultaneamente, enquanto a escalabilidade assegura a flexibilidade para lidar com o crescimento da demanda. Além disso, a eficiência de recursos visa otimizar o uso de elementos como CPU, memória e largura de banda, assegurando um funcionamento eficaz do software. Ao definir e cumprir esses requisitos, busca-se criar aplicações robustas, capazes de atender às expectativas dos usuários e às metas do negócio.

- **Apache Spark**

O Spark é uma poderosa plataforma de processamento de dados de código aberto, projetada para realizar análise e processamento de grandes volumes de dados em escala distribuída. Baseado na computação em memória, o Spark oferece um ambiente de programação amigável, permitindo que desenvolvedores criem aplicativos complexos para processamento de

dados de forma eficiente e rápida. Sua arquitetura resiliente e tolerante a falhas, o RDD (Resilient Distributed Dataset), permite que os dados sejam distribuídos por vários nós de um cluster de computadores, garantindo assim a disponibilidade dos dados mesmo em casos de falhas de hardware. Além disso, o Spark suporta uma ampla gama de tarefas, como processamento de lotes, consultas SQL, análise de streaming e aprendizado de máquina, tornando-o uma escolha versátil para empresas e pesquisadores que precisam lidar com grandes conjuntos de dados e obter insights valiosos de maneira eficiente e escalável.

Pensando nisso, foi utilizado o Spark no algoritmo do jogo da vida, para analisarmos como seria a performance do código. Abaixo segue trechos do código:

Primeiro com o SparkContext, iremos inicializar o spark com o nome de GameOfLife. Dentro do segundo for colocamos o broadcast para transmitir o tabuleiro de entrada (tabulIn) para todos os nós trabalhadores do cluster Spark de forma eficiente. Então, criamos um RDD para paralelizar o processamento, e aplicamos a função UmaVida em cada linha do tabuleiro usando o rdd.foreach().

```
if __name__ == "__main__":
    sc = SparkContext(appName="GameOfLife")

    powmin = 2
    powmax = 4

    for pow in range(powmin, powmax + 1):
        tam = 1 << pow

        t0 = wall_time()
        tabulIn, tabulOut = InitTabul(tam)
        t1 = wall_time()

        iterations = 2 * (tam - 3)
        for _ in range(iterations):

            broadcasted_tabulIn = sc.broadcast(tabulIn)

            rdd = sc.parallelize(range(1, tam + 1))

            rdd.foreach(lambda i: UmaVida((broadcasted_tabulIn.value, tabulOut, tam, i)))

            tabulIn, tabulOut = tabulOut, tabulIn

        t2 = wall_time()
```

A variável global\_is\_correct é usada para reduzir os resultados de todas as partições para verificar se todas as partições geraram o resultado correto. Ao final o sc.getConf() recupera o endereço IP do driver Spark para ver se é

localhost. `global_is_correct` é a variável booleana que armazena o resultado da verificação se o tabuleiro final do jogo da vida está correto que foi explicada mais acima. Se o programa está sendo executado localmente e o tabuleiro final está correto, a mensagem "Ok, RESULTADO CORRETO" será impressa no console. Caso contrário, a mensagem "Nok, RESULTADO ERRADO" será impressa

```
is_correct = Correto(tabulIn, tam)
global_is_correct = sc.parallelize([is_correct]).reduce(lambda x, y: x and y)

if sc.getConf().get('spark.driver.host') == 'localhost':
    if global_is_correct:
        print("***Ok, RESULTADO CORRETO**")
    else:
        print("***Nok, RESULTADO ERRADO**")

t3 = wall_time()
print("-----RESULTADO-----")
print("tam=%d; tempos: init=%7.7f, comp=%7.7f, fim=%7.7f, tot=%7.7f" %
      (tam, t1 - t0, t2 - t1, t3 - t2, t3 - t0))
print("-----RESULTADO-----\n\n")

sc.stop()
```

- **OpenMP**

O OpenMP (Open Multi-Processing) é uma API (Interface de Programação de Aplicativos) de programação paralela que foi projetada para facilitar a criação de aplicações que aproveitam o poder de processadores multicore e multiprocessadores. Essa abordagem permite que os desenvolvedores introduzam paralelismo em seus programas de forma relativamente simples, adicionando diretivas especiais ao código-fonte. Com o OpenMP, é possível dividir tarefas em threads que podem ser executadas simultaneamente em diferentes núcleos de CPU, acelerando assim o processamento e melhorando o desempenho do software. Além disso, o OpenMP oferece suporte a diferentes níveis de paralelismo, permitindo que os programadores otimizem partes específicas do código que podem se beneficiar da execução paralela. Essa abordagem torna o OpenMP uma escolha popular para desenvolvedores que desejam tirar proveito da computação paralela em aplicações científicas, de modelagem, simulação e outras que envolvem tarefas

intensivas em CPU.

Em um trabalho passado já tínhamos utilizado esse engine no algoritmo do Jogo da Vida, pensando em saber como ficaria a performance. Segue abaixo partes de trechos utilizando o OpenMP:

O OpenMP foi utilizado na função UmaVida que é a principal responsável por aplicar as regras do jogo da vida.

```
void UmaVida(int* tabulIn, int* tabulOut, int tam) {
    int i, j, vizviv;

    #pragma omp parallel for private(i, j, vizviv) shared(tabulIn, tabulOut)
    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)];
        } /* fim-for */
    } /* fim-for */
} /* fim-UmaVida */
```

O #pragma omp parallel for foi utilizado para paralelizar o loop interno na função UmaVida, onde o cálculo das novas células vivas é realizado. As iterações do loop são independentes, pois cada célula é atualizada com base apenas nos valores de suas células vizinhas. Consequentemente, é possível paralelizar com segurança essas iterações e distribuir o trabalho entre os threads disponíveis para acelerar a execução do código.

- **MPI**

MPI (Message Passing Interface) é uma especificação de biblioteca de programação amplamente utilizada para o desenvolvimento de aplicações paralelas e distribuídas. Essa interface fornece um conjunto de rotinas e funções que permitem a comunicação e sincronização eficiente entre

processos ou threads em um ambiente de computação distribuída. O MPI é especialmente popular em ambientes de computação de alto desempenho, onde é necessário aproveitar ao máximo os recursos disponíveis, distribuindo tarefas entre diferentes nós de processamento para acelerar a execução de cálculos complexos. Com o MPI, os desenvolvedores podem criar programas que exploram o potencial de clusters de computadores, supercomputadores e ambientes de computação paralela, possibilitando a resolução de problemas complexos em áreas como simulação, modelagem, análise de dados e pesquisa científica em geral.

Em um trabalho passado já tínhamos utilizado esse engine no algoritmo do Jogo da Vida, pensando em saber como ficaria a performance. Segue abaixo partes de trechos utilizando o MPI:

Diferentemente do OpenMP, o MPI foi utilizado na função main. Nesse trecho do código está sendo feita a inicialização do ambiente MPI, e pegando com a `MPI_Comm_size` a quantidade total de processos e com a `MPI_Comm_rank` a identificação do processo atual.

```
int main(int argc, char** argv) {
    int pow, rank = 0, size = 0;
    int i, tam, *tabulIn, *tabulOut;
    char msg[9];
    double t0, t1, t2, t3;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Dentro do primeiro for utilizamos o `MPI_Barrier` e o `MPI_Bcast`. O `MPI_Barrier` é utilizado para sincronização entre os processos, garantindo que cada etapa importante da execução seja concluída por todos os processos antes de prosseguir. Já o `MPI_Bcast` é utilizada para transmitir dados de um processo (nesse caso, o processo 0) para todos os outros processos do comunicador (`MPI_COMM_WORLD`), no código, foi usado para enviar o tabuleiro inicial (`tabulIn`) para todos os processos, garantindo que todos comecem com o mesmo estado inicial do tabuleiro.

```

for (pow = POWMIN; pow <= POWMAX; pow++) {
    tam = 1 << pow;

    t0 = wall_time();

    tabulIn = (int*) malloc ((tam+2)*(tam+2)*sizeof(int)*tam);
    tabulOut = (int*) malloc ((tam+2)*(tam+2)*sizeof(int)*tam);

    InitTabul(tabulIn, tabulOut, tam);
    t1 = wall_time();

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Bcast(tabulIn, (tam+2)*(tam+2), MPI_INT, 0, MPI_COMM_WORLD);

    for (i = 0; i < 2*(tam-3); i++) {
        UmaVida(tabulIn, tabulOut, tam);
        UmaVida(tabulOut, tabulIn, tam);
    }
}

```

Na parte mais embaixo foi utilizado o MPI\_Gather que é uma função utilizada para coletar dados de todos os processos (nesse caso, tabulIn) e os agrupa no processo raiz (processo 0). No código, é utilizado para reunir os tabuleiros de cada processo após a evolução do jogo em paralelo.

```

t2 = wall_time();

MPI_Gather(rank == 0 ? MPI_IN_PLACE : tabulIn, (tam+2)*(tam+2), MPI_INT,
          tabulIn, (tam+2)*(tam+2), MPI_INT, 0, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

if (rank == 0) {
    if (Correto(tabulIn, tam))
        printf("**RESULTADO CORRETO*\n");
    else
        printf("**RESULTADO ERRADO*\n");

    t3 = wall_time();

    printf("tam=%d; tempos: init=%7.7f, comp=%7.7f, fim=%7.7f, tot=%7.7f \n",
          tam, t1 - t0, t2 - t1, t3 - t2, t3 - t0);
}

```

- **MPI + OpenMP**

Pensando em analisar se a performance misturando os dois seria melhor que utilizando só uma individual. Foi feito o algoritmo do Jogo da Vida



misturando as duas engines para analisar a sua performance. Segue trechos do código abaixo:

A parte do MPI ficou da mesma forma que a anterior.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

for (pow = powmin; pow <= powmax; pow++) {
    tam = 1 << pow;

    t0 = wall_time();

    tabulIn = (int*) malloc ((tam+2)*(tam+2)*sizeof(int)*tam);
    tabulOut = (int*) malloc ((tam+2)*(tam+2)*sizeof(int)*tam);

    InitTabul(tabulIn, tabulOut, tam);
    t1 = wall_time();

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Bcast(tabulIn, (tam+2)*(tam+2), MPI_INT, 0, MPI_COMM_WORLD);
```

Entretanto foi adicionada a função #pragma omp parallel for private.

```
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)];
        }
    }
}
```

#### 4. Requisito de elasticidade

O requisito de elasticidade é um componente essencial para garantir a escalabilidade dinâmica e adaptativa de sistemas e aplicações em ambientes

computacionais modernos. Ele se refere à capacidade do software de se ajustar automaticamente em resposta às mudanças na demanda e carga de trabalho, expandindo ou reduzindo seus recursos de forma flexível. Com a elasticidade, a aplicação pode dimensionar recursos, como servidores, capacidade de armazenamento ou poder de processamento, de acordo com as necessidades reais em tempo real. Isso permite que o sistema mantenha um desempenho ideal mesmo diante de picos de tráfego ou flutuações na demanda, garantindo uma experiência contínua e eficiente para os usuários finais. A elasticidade é especialmente valiosa em ambientes de computação em nuvem, onde os recursos podem ser provisionados e liberados de forma rápida e sob demanda, ajudando a otimizar os custos e a eficiência operacional do software.

- **Kubernetes**

O Kubernetes é uma plataforma de orquestração de contêineres de código aberto que revolucionou a forma como as aplicações são implementadas, escaladas e gerenciadas na era da computação em nuvem. Desenvolvido originalmente pelo Google, o Kubernetes oferece uma solução robusta e flexível para automatizar a distribuição e operação de aplicações em ambientes de infraestrutura distribuídos. Sua arquitetura altamente modular e suas poderosas funcionalidades permitem que os desenvolvedores e administradores de sistemas criem e gerenciem implantações complexas de contêineres de maneira eficiente, garantindo alta disponibilidade, escalabilidade horizontal e uma resiliência impressionante. Com o Kubernetes, as equipes de desenvolvimento podem se concentrar no desenvolvimento de software, enquanto a plataforma cuida da complexidade subjacente de executar e dimensionar as aplicações, tornando-se uma peça fundamental na jornada rumo à orquestração moderna de contêineres.

- **Aplicações e configurações realizadas**

Para aplicação dos requisitos de elasticidade no projeto, foi criado um Kubernetes por meio da cloud da DigitalOcean. Em que foi necessário criar um

documento dockerfile para cada versão do algoritmo do Jogo da Vida, e fazer a configuração do docker, em seguida configurado o kubernetes seguindo os passos abaixo:

```
#!/bin/bash
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl
curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg |
sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-archive-keyring.gpg

echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg]
https://apt.kubernetes.io/ kubernetes-xenial main" |
sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl

sudo rm /etc/containerd/config.toml
sudo systemctl restart containerd
# Master
kubeadm config images pull
sudo kubeadm init
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
kubectl create -f https://docs.projectcalico.org/manifests/calico.yaml
```

Também foi configurado o Grafana que é uma plataforma de análise e visualização de dados de código aberto, altamente popular entre engenheiros de sistemas e analistas de dados. Com sua interface intuitiva e amigável, o Grafana permite que os usuários conectem diversas fontes de dados, como bancos de dados, sistemas de monitoramento e métricas, além de serviços em nuvem, para criar painéis personalizados e informativos. Com recursos de gráficos interativos, tabelas dinâmicas e alertas configuráveis, o Grafana possibilita a análise profunda de informações críticas em tempo real. Sua flexibilidade e extensibilidade também são notáveis, graças ao suporte a diversos plugins e integrações com outras ferramentas de monitoramento e análise de dados. Em resumo, o Grafana é uma solução poderosa e versátil que capacita organizações a transformar dados complexos em insights valiosos para tomar decisões informadas e estratégicas.

Além do disso foi configurado o Prometheus também que é um sistema de monitoramento e alerta de código aberto, amplamente utilizado na




comunidade de infraestrutura de TI e DevOps. Com a sua abordagem de coleta de métricas baseada em HTTP, o Prometheus permite monitorar uma ampla gama de alvos, como servidores, aplicativos e dispositivos de rede, de forma altamente escalável e eficiente. Sua arquitetura é fundamentada em um modelo de coleta de métricas com etiquetas (labels), proporcionando uma flexibilidade incomparável na organização e consulta dos dados. Além disso, o Prometheus possui uma linguagem de consulta poderosa, chamada PromQL, que possibilita a análise e visualização dos dados coletados de maneira ágil e personalizada. Com recursos avançados de alertas, é possível configurar notificações proativas com base em limiares predefinidos, garantindo uma resposta rápida a eventos críticos. O Prometheus tem se destacado como uma ferramenta essencial no ecossistema de monitoramento moderno, tornando-se uma escolha preferida para equipes que buscam obter uma visão abrangente e confiável da saúde de seus sistemas em tempo real.

Foi implementado também o Kafka para estabelecer os parâmetros do jogo da vida. Kafka é uma plataforma de streaming de dados distribuída e de código aberto, desenvolvida para atender aos desafios de lidar com grandes volumes de dados em tempo real. Criado pelo LinkedIn e posteriormente doado à Apache Software Foundation, o Kafka tornou-se uma peça fundamental na arquitetura de muitas empresas modernas. Com sua natureza distribuída, escalável e tolerante a falhas, o Kafka permite que os dados sejam transmitidos de forma eficiente e confiável entre aplicativos e sistemas em tempo real ou em lote. Através de um mecanismo de publish/subscribe, os produtores enviam mensagens para tópicos, enquanto os consumidores se inscrevem nesses tópicos para receber e processar os dados. Além disso, o Kafka armazena as mensagens em log de forma durável, garantindo a capacidade de reprocessar e reproduzir eventos passados. Sua arquitetura flexível e modular possibilita a integração com diversas ferramentas e ecossistemas, tornando-o uma escolha popular para casos de uso como transmissão de eventos, análise de dados em tempo real, agregação de logs e replicação de dados em ambientes distribuídos. Em resumo, o Kafka se tornou uma tecnologia essencial para lidar com o processamento de dados em larga escala, permitindo que as organizações construam sistemas robustos e resilientes em face dos desafios do mundo moderno de dados em tempo real.

Foi feita também a configuração do Kibana que é uma poderosa e popular plataforma de visualização e exploração de dados projetada para trabalhar em conjunto com o Elasticsearch, que é um mecanismo de busca e análise de dados em tempo real. Desenvolvido pela Elastic, o Kibana permite que os usuários interajam com os dados indexados no Elasticsearch de forma intuitiva e eficiente, transformando-os em gráficos, tabelas, mapas e diversos tipos de visualizações personalizadas. Através de uma interface amigável e baseada na web, o Kibana facilita a análise e interpretação de grandes volumes de dados, tornando-se uma ferramenta essencial para equipes de negócios, analistas de dados e desenvolvedores. Além disso, com suas funcionalidades avançadas de pesquisa e filtragem, o Kibana possibilita a obtenção de insights valiosos, a detecção de tendências e padrões, e a criação de painéis interativos para monitoramento e tomada de decisões em tempo real. Com seu foco em usabilidade e visualização, o Kibana se destaca como uma solução completa para transformar dados brutos em informações valiosas, contribuindo para uma melhor compreensão dos dados e otimização de processos em diversas áreas, como análise de negócios, segurança da informação e monitoramento de infraestruturas tecnológicas.

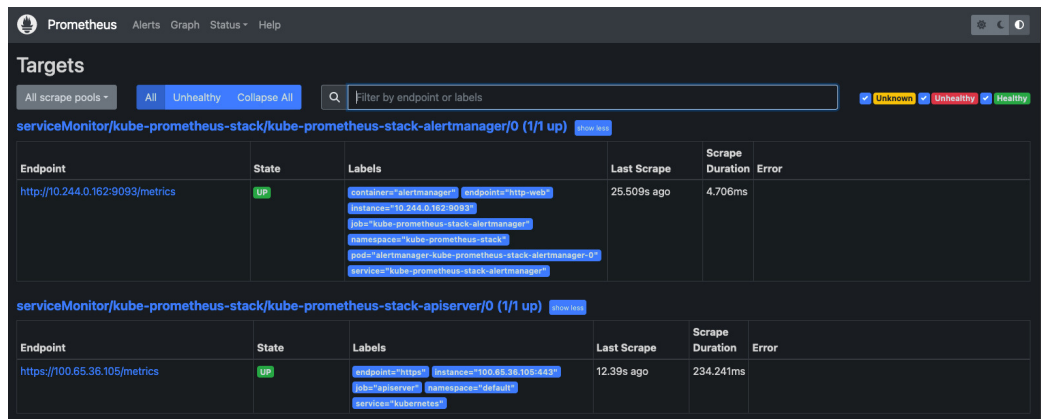
## 5. Análise dos resultados

Para implementação do kubernetes na nuvem do Digital Ocean, foram criados três nós, conforme figura abaixo:

Nodes										
Name	Labels	CPU Ready requests (cores)	CPU limits (cores)	CPU capacity (cores)	Memory requests (bytes)	Memory limits (bytes)	Memory capacity (bytes)	Pods	Create	
 kube-f9ruq	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/instance-type: s-2vcpu-4gb beta.kubernetes.io/os: linux <a href="#">Show all</a>	True 1.10 (58.00%)	502.00m (26.42%)	1.90	2.14Gi (70.32%)	1.77Gi (58.43%)	3.04Gi	13 (11.82%)	a day ago	⋮
 kube-f9ruy	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/instance-type: s-2vcpu-4gb beta.kubernetes.io/os: linux <a href="#">Show all</a>	True 1.10 (58.00%)	1.40 (73.79%)	1.90	1.48Gi (48.65%)	1.77Gi (58.43%)	3.04Gi	19 (17.27%)	a day ago	⋮
 kube-f9ruf	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/instance-type: s-2vcpu-4gb beta.kubernetes.io/os: linux <a href="#">Show all</a>	True 802.00m (42.21%)	302.00m (15.89%)	1.90	3.00Gi (98.75%)	2.73Gi (89.74%)	3.04Gi	10 (9.09%)	a day ago	⋮

Totalizando 42 pods, em que estão distribuídos o Kafka, ElasticSearch, Kibana, Grafana, Prometheus, aplicação desenvolvida, Nginx Ingress e Alert Manager.

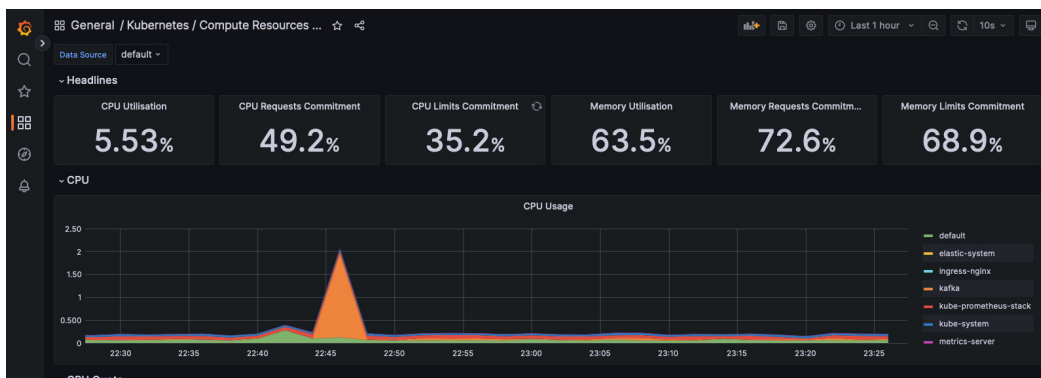
O Prometheus coleta os dados da aplicação do kubernetes, conforme imagem abaixo, por exemplo:



The screenshot shows the Prometheus Targets page with two service monitors. The first monitor is for the alertmanager, and the second is for the apiserver. Both are in an 'UP' state.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.244.0.162:9093/metrics	UP	<code>container="alertmanager"</code> <code>endpoints="http://web"</code> <code>instances="10.244.0.162:9093"</code> <code>job="kube-prometheus-stack-alertmanager"</code> <code>namespace="kube-prometheus-stack"</code> <code>pod="alertmanager-kube-prometheus-stack-alertmanager-0"</code> <code>services="kube-prometheus-stack-alertmanager"</code>	25.509s ago	4.706ms	
https://100.65.36.105/metrics	UP	<code>endpoint="https"</code> <code>instance="100.65.36.105:443"</code> <code>job="apiserver"</code> <code>namespace="default"</code> <code>services="kubernetes"</code>	12.39s ago	234.241ms	

Com isso o Grafana recolhe as informações e transforma em um dashboard para uma melhor visualização dos dados críticos, conforme imagem abaixo, por exemplo:



O Kafka foi utilizado para conseguir setar os parâmetros para o jogo da vida, na hora de rodar no kubernetes. Segue abaixo um trecho do código:

```

def call_c_program(mensagem):
    powmin, powmax, codeSelector = mensagem.split()
    powmin = powmin.decode('utf-8')
    powmax = powmax.decode('utf-8')
    codeSelector = codeSelector.decode('utf-8')

    if (codeSelector == 'mpi'):
        os.system(f"OMP_NUM_THREADS=4 mpirun -np 4 ./teste {powmin} {powmax}")
    else:
        os.system(f"python3 jogodavida.py {powmin} {powmax}")
        read_file_and_send_to_es(codeSelector)

def consume_kafka_topic(topic):
    consumer = Consumer({
        'bootstrap.servers': '10.245.179.235:9092',
        'group.id': 'foo',
        'auto.offset.reset': 'earliest',
        'session.timeout.ms': 6000,
    })

    try:
        consumer.subscribe([topic])
        while True:
            msg = consumer.poll(1.0)

            if msg is None:
                continue

```

Em relação a análise da performance do código, levando em conta o uso de cada engine individual (MPI e OpenMP), o que apresentou melhor performance foi inconclusivo, devido a alteração dos tempos conforme a mudança de horário. As figuras abaixo apresentam os resultados obtidos em um horário noturno.

#### - OpenMP

```

**RESULTADO CORRETO**
tam=8; tempos: init=0.0000072, comp=0.0004809, fim=0.0000432, tot=0.0005312
**RESULTADO CORRETO**
tam=16; tempos: init=0.0000122, comp=0.0005910, fim=0.0000110, tot=0.0006142
**RESULTADO CORRETO**
tam=32; tempos: init=0.0000198, comp=0.0025001, fim=0.0000041, tot=0.0025239
**RESULTADO CORRETO**
tam=64; tempos: init=0.0000148, comp=0.0029690, fim=0.0000091, tot=0.0029929
**RESULTADO CORRETO**
tam=128; tempos: init=0.0000658, comp=0.0221212, fim=0.0000339, tot=0.0222208
**RESULTADO CORRETO**
tam=256; tempos: init=0.0003059, comp=0.1776402, fim=0.0001268, tot=0.1780729
**RESULTADO CORRETO**
tam=512; tempos: init=0.0011899, comp=1.4168899, fim=0.0004990, tot=1.4185789
**RESULTADO CORRETO**
tam=1024; tempos: init=0.0046749, comp=11.5849011, fim=0.0019889, tot=11.5915649

```

#### - MPI

```

*RESULTADO CORRETO*
tam=8; tempos: init=0.0000160, comp=0.0001299, fim=0.0000749, tot=0.0002208
*RESULTADO CORRETO*
tam=16; tempos: init=0.0000119, comp=0.0005031, fim=0.0000660, tot=0.0005810
*RESULTADO CORRETO*
tam=32; tempos: init=0.0000391, comp=0.0040669, fim=0.0004270, tot=0.0045331
*RESULTADO CORRETO*
tam=64; tempos: init=0.0000610, comp=0.0191939, fim=0.0001762, tot=0.0194311
*RESULTADO CORRETO*
tam=128; tempos: init=0.0001020, comp=0.0877199, fim=0.0010931, tot=0.0889151
*RESULTADO CORRETO*
tam=256; tempos: init=0.0003040, comp=0.6924989, fim=0.0081141, tot=0.7009170
*RESULTADO CORRETO*
tam=512; tempos: init=0.0011821, comp=5.5893581, fim=0.0485759, tot=5.6391160
*RESULTADO CORRETO*
tam=1024; tempos: init=0.0047510, comp=46.5171821, fim=0.3277030, tot=46.8496361

```

Já se compararmos a performance entre o algoritmo que mistura o OpenMP com o MPI, em relação ao algoritmo que está utilizando o Spark. O do OpenMP+MPI demonstra uma melhor performance, conforme ilustrado abaixo:

- OpenMP + MPI

```

*RESULTADO CORRETO*
tam=8; tempos: init=0.0000050, comp=0.4657459, fim=0.0302219, tot=0.4959729
*RESULTADO CORRETO*
tam=16; tempos: init=0.0000081, comp=0.2762260, fim=0.6744199, tot=0.9506540
*RESULTADO CORRETO*
tam=32; tempos: init=0.0000210, comp=2.5332839, fim=0.0161819, tot=2.5494869
*RESULTADO CORRETO*
tam=64; tempos: init=0.0000300, comp=5.0677681, fim=0.0065670, tot=5.0743651
*RESULTADO CORRETO*
tam=128; tempos: init=0.0000911, comp=10.0013568, fim=0.0120990, tot=10.0135469
*RESULTADO CORRETO*
tam=256; tempos: init=0.0003290, comp=11.1436789, fim=0.0097809, tot=11.1537888
*RESULTADO CORRETO*
tam=512; tempos: init=0.0030000, comp=41.2540061, fim=0.1178930, tot=41.3748991

```

- Spark



```

-----RESULTADO-----
tam=2; tempos: init=0.0000069, comp=0.0000012, fim=1.4233835, tot=1.4233916
-----RESULTADO-----

-----RESULTADO-----
tam=4; tempos: init=0.0000050, comp=0.3049867, fim=0.1665599, tot=0.4715517
-----RESULTADO-----

-----RESULTADO-----
tam=8; tempos: init=0.0000072, comp=1.5053077, fim=0.1317985, tot=1.6371133
-----RESULTADO-----

-----RESULTADO-----
tam=16; tempos: init=0.0000057, comp=3.2093337, fim=0.1520126, tot=3.3613520
-----RESULTADO-----

```

## 6. Conclusão

Podemos concluir que para escolher entre o uso de MPI ou OpenMP, depende do tipo de problema que deseja-se resolver e também de fatores externos em relação ao cluster, como por exemplo, se está sendo muito acessado, ou se a internet está lenta entre outros. Pode-se concluir também que para o algoritmo do jogo da vida, o uso do OpenMP + MPI é mais eficiente do que apenas utilizar o Spark. Além disso, o uso de elasticidade é bem importante, com a implementação do Kubernetes para o orquestramento dos containers torna bem mais prático o Gerenciamento, e o uso do Prometheus juntamente com o grafana é importante para se ter uma análise de como está o sistema e se tem algo que está crítico.

## 7. Bibliografia

LIMA, Leandro Z.; OLIVEIRA, Pedro P. M. JOGO DA VIDA: CONCEITOS E APLICAÇÕES, 2014. Disponível em: <<https://bsi.uniriotec.br/wp-content/uploads/sites/31/2020/05/201406ZoucasMarques.pdf>>

ZANELLA, Liane C. H. Metodologia de Pesquisa. 2011. Disponível em:

<<https://www.atfcursosjuridicos.com.br/repositorio/material/3-leitura-extra-02.pdf>

>