Avaliação do Singularity como plataforma para executar aplicações em computação de alto desempenho

1st Deivid Francisco dos Santos
Escola Politécnica - PUCRS
deivid.santos@edu.pucrs.br

2nd Tiago Ferreto (orientador)
Escola Politécnica - PUCRS
tiago.ferreto@pucrs.br

***Resumo*—A portabilidade e reprodutibilidade têm se tornado cada vez mais importantes na computação. Visando atender estes requisitos, o mercado vem aderindo significativamente à utilização de contêineres no desenvolvimento de soluções. Além disso, diversas áreas de pesquisa também tem aderido ao uso de contêineres, como a computação de alto desempenho (HPC), que possui como foco o uso de *clusters* de máquinas para execução de aplicações com alta demanda computacional, que podem levar de minutos a dias até concluírem o seu processamento. Em um ambiente de HPC, cada aplicação precisa estar disponível em cada nó do *cluster*, assim como toda a *stack* de aplicações necessárias para que ela execute. Para facilitar o uso, a implementação e a portabilidade de aplicações em ambientes de HPC, tem se tornado popular o uso de sistemas de virtualização baseados em contêineres específicos para alto desempenho, como o *singularity*. Este artigo visa analisar o desempenho, e avaliar os pontos positivos e negativos de se utilizar contêineres como o *singularity* em um ambiente de HPC. A avaliação foi realizada em um *cluster* de instâncias EC2 do provedor de nuvem AWS, comparando a execução de diversas aplicações paralelas, baseadas no NAS Parallel Benchmarks, diretamente sobre as instâncias e utilizando o *singularity*. O objetivo desta avaliação é entender os ganhos e problemas no uso do *singularity*, para futuramente disponibilizá-lo para os usuários do Laboratório de Alto Desempenho da PUCRS.**

I. Introdução

A computação de alto desempenho (HPC) [1] é uma área extremamente importante da pesquisa, ajudando em previsões e cálculos complexos e longos de diversas outras áreas. É uma solução de *cluster* onde o poder computacional de várias máquinas é agregado para execução de forma paralela de programas que podem durar longos períodos. Para executar essas aplicações nos HPCs é necessária a configuração completa do ambiente com toda a *stack* de programas necessários para que ela execute. Às vezes, a configuração não é simples ou requer um trabalho repetitivo que deve ser feito em cada nó do *cluster* onde será executada. E também caso seja necessário executar em *clusters* diferentes, a mesma configuração deve ser feita novamente no novo *cluster*. Para tentar amenizar esses problemas, uma das opções possíveis é utilizar conteinerização [2], que vem crescendo e sendo cada vez mais usada, devido principalmente à melhoria da eficiência e produtividade dos desenvolvedores, simplificando e agilizando bastante a reprodutibilidade de aplicações.

Conteinerização é uma forma de virtualização [3] de Sistemas Operacionais que está se tornando o padrão de mercado por trazer esses benefícios no desenvolvimento de *software*. Nele, as aplicações são executadas em ambientes isolados que compartilham alguns recursos do sistema operacional onde estão sendo executadas, assim trazendo um consumo otimizado de recursos. Esse modelo de implementação enfatiza a portabilidade, onde por meio de arquivos e comandos simples é possível replicar toda a execução de uma aplicação e tornar o desenvolvimento muito mais ágil.

Existem diversas soluções de contêineres disponíveis atualmente. O *singularity* [4] é uma das mais utilizadas com foco na área de HPC. Este artigo avalia o *singularity* como uma plataforma para execuções de alto desempenho em HPC. A avaliação é feita através de validações utilizando programas de testes conhecidos em conjunto com o MPI [5], de forma nativa e com o *singularity* em ambiente de nuvem, entendendo suas vantagens e desvantagens nesses diversos cenários para no fim disponibilizá-lo para os usuários dentro do Laboratório de Alto Desempenho (LAD) da PUCRS.

II. Fundamentação teórica

A. Infraestrutura de alto desempenho

A computação de alto desempenho [6] é a área da computação que trabalha com cargas de processamento e rede muito grandes. HPC é uma solução onde o poder computacional de várias máquinas conectadas através da rede é agregado para trabalharem com o mesmo objetivo. Eles executam longos programas de forma paralela utilizando *frameworks* que fazem o rastreamento de recursos disponíveis para distribuir as tarefas de forma efetiva no gerenciamento do processamento entre as máquinas. Essa solução atualmente pode utilizar também a computação em nuvem com vários provedores disponíveis.

B. Virtualização

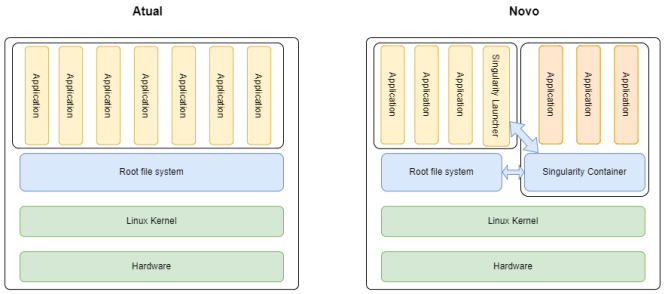
Existem diversos tipos de virtualização [7]. O mais conhecido é o ambiente utilizando máquinas virtuais (VMs). Nesse modelo, é possível utilizar um sistema operacional isolado dentro de outro sistema operacional. Na VM os recursos do computador *host* são alocados e isso torna possível trabalhar inteiramente dentro da máquina virtual como se

fosse outro computador. Isso traz muitas vantagens no desenvolvimento de *softwares* e principalmente em testes onde é possível criar e recriar com facilidade ambientes isolados e completos. As máquinas virtuais utilizam o HyperVisor. Essa é a principal camada das máquinas virtuais que será responsável por virtualizar os componentes de hardware utilizados pela VM em questão.

Por implementar todas essas camadas, as VMs têm desvantagens para o seu uso em aplicações finalizadas e em produção. O alto consumo de recursos e o tempo de inicialização são os principais fatores, pois um sistema operacional completo e com todos seus módulos precisa ser inicializado.

Outra opção de virtualização que soluciona ou ameniza alguns desses problemas é a utilização de contêineres. Assim como as VM's, os contêineres são ambientes virtuais que também garantem isolamento. Porém, a inicialização é muito mais rápida e o consumo de recursos mais otimizado, pois eles compartilham recursos e pacotes diretamente com o sistema operacional onde estão executando. Desta forma, não é necessário subir um novo sistema completo. Os contêineres são muito mais portáteis do que VM's, pois são bem mais leves e todos os itens necessários podem ficar em um único arquivo, bastando copiar e executar em outra máquina que tenha suporte. Existem algumas limitações ao se utilizar contêineres, como quais sistemas operacionais podem ser usados e onde podem ser executados.

Fig. 1. Modelo de contêineres



O modelo de como os contêineres funcionam, visto na Figura 1, demonstra que é criada uma camada acima do sistema operacional permitindo utilizar os recursos do *host* e podendo executar múltiplos contêineres ao mesmo tempo na mesma máquina. Atualmente, o modelo de contêineres já é utilizado em larga escala no mercado. As principais empresas no mundo fazem uso em produção por trazer agilidade no desenvolvimento.

Entre as diversas ferramentas de conteinerização existentes, a mais utilizada é o Docker [8]. A plataforma Docker disponibiliza uma ampla biblioteca de imagens de sistemas operacionais pré-configuradas, facilitando consideravelmente a implementação de contêineres com diversas ferramentas. Este processo pode ser realizado de maneira eficiente e rápida. Porém, considerando que o foco do trabalho é em ambientes de HPC, o Docker não foi desenvolvido para longos processamentos de dados e acesso a alguns recursos de hardware. Alguns problemas de isolamento como acesso a arquivos da máquina *host*, acesso aos ID's de usuário, uso de GPU's não são tratados diretamente pelo Docker.

Atualmente, existem algumas opções de conteinerização especificamente para uso em ambientes HPC na área científica, como o *singularity* [4], o Shifter [9], e o Charliecloud [10]. Todas as ferramentas com foco em HPC funcionam de maneira parecida, cada uma com suas vantagens. Pela necessidade de longos processamentos, elas não possuem o objetivo de rodar de maneira efêmera como contêineres comuns.

C. Singularity

O *singularity* [4] possui código aberto e foi desenvolvido na linguagem Golang pelo grupo de pesquisa do Lawrence Berkeley National Laboratory em 2015. Atualmente, está sob a responsabilidade da *singularity*-Ware, LLC e roda em sistemas operacionais Unix. É muito popular dentro da área de pesquisa, sendo possível utilizá-lo nos maiores laboratórios do mundo e até mesmo em ferramentas de computação em nuvem. O *singularity* foi pensado para o uso em ambientes de HPC, então ele traz alguns pontos que o diferencia de ferramentas comuns como Docker.

Uma aplicação que executa em HPC não deveria ter privilégios de administrador no cluster para evitar que um usuário não afete outros e também não consiga acessar o sistema de arquivos do *host*. O *singularity* não precisa de acesso *root* e executa a aplicação com o usuário que iniciou o contêiner. Também há a compatibilidade com as principais ferramentas e recursos de HPC, desde ferramentas para gerenciamento de recursos de computação paralela, como o Slurm [11] e o MPI [5], e também recursos de hardware como GPUs. O *singularity* não possui a necessidade de uma configuração personalizada de rede para que múltiplos contêineres se comuniquem. Por padrão, ele utiliza a rede da máquina *host*.

O *singularity* se destaca de outras soluções de HPC devido à sua compatibilidade completa com o Docker Hub [12], permitindo a utilização de todas as imagens disponíveis. Além disso, possui um formato próprio compatível com o Docker. O Docker Hub é um serviço público de registro de contêineres oferecido pela Docker. Funciona como uma biblioteca onde diversas imagens são armazenadas, é o maior repositório de imagens disponível, sendo que diversas empresas disponibilizam imagens oficiais. Nele é possível escolher qual imagem utilizar e importar no *singularity*, assim como é possível criar imagens personalizadas e disponibilizá-las publicamente.

Um dos principais objetivos do *singularity* é lidar com a reprodutibilidade e portabilidade, que são pontos muito importantes dentro da área de pesquisa. Isso é possível graças ao formato de imagem disponível para o *singularity* que possui a definição completa do contêiner e toda a sua configuração, sendo que com alguns comandos é possível executá-lo por completo. Também possui suporte aos principais recursos usados em HPC, sendo pensada para ter controle do ambiente onde é executada.

D. MPI

O MPI [5] é um padrão utilizado dentro da computação paralela e de alto desempenho, pois define um modelo para troca de mensagens entre processos. Com ele, é possível criar e executar aplicações capazes de paralelizar a execução entre diversos nós usando a rede, sem a necessidade de memória compartilhada entre eles.

III. Trabalhos relacionados

Existem diversos trabalhos similares ao sendo proposto neste artigo. Alguns deles são:

1) O artigo [4] é o artigo base que descreve a ferramenta *singularity*. Nele são demonstrados os objetivos, vantagens de se utilizar conteinerização com o *singularity* em aplicações de alto desempenho e também exemplos de casos de uso. Também é mostrado como usá-lo com seus principais comandos. O artigo traz detalhes da ferramenta, porém não aborda com mais detalhes o desempenho e também não realiza testes comparativos.
2) Em [13], os autores apresentam testes de desempenho utilizando o *singularity* aplicado no *cluster* Comet do centro de supercomputadores de San Diego. Nesse artigo foram utilizados três diferentes conjuntos de testes para realizar experimentos de latência, de taxa de transferência e tempo de execução, sendo eles o Intel MPI's Benchmarks (IMB), Ohio State University's Benchmarks, e o Jones ModelDB Mode. Os resultados foram extraídos por uma aplicação Python a partir da média de 10 execuções. Os resultados indicam que praticamente não há perda de desempenho ao comparar os resultados. Já, os ganhos de reprodutibilidade e portabilidade são significativos. Este artigo é importante por já trazer dados e testes de desempenho aplicados, mostrando algumas das vantagens da utilização dessas tecnologias.
3) O artigo [14] traz alguns outros pontos em relação a conteinerização em ambientes de HPC. O artigo faz comparações com outras ferramentas citadas anteriormente como o Docker, o Shifter e o Charliecloud e demonstrações da aplicação de contêineres em sistemas em computação em nuvem como a AWS. Da mesma forma que o artigo anterior, ele também utiliza o conjunto de aplicações do Intel MPI's Benchmarks (IMB) com foco em latência e banda utilizando o *singularity* no *cluster* Cray XC-series supercomputer e Docker em maquinas EC2 na AWS. Nele é possível notar algumas diferenças de desempenho, sendo possível notar que o *singularity* lida melhor com recursos de HPC quando comparado ao Docker.
4) Em [15], os autores abordam conteinerização em HPC de uma forma geral. Ele mostra o uso de diversas tecnologias, mas mostrando quais os desafios e quais problemas existem atualmente e possíveis soluções ao utilizar contêineres em aplicações de alto desempenho.

IV. Proposta

Esse artigo propõe avaliar e testar o desempenho e reprodutibilidade do *singularity* para viabilizar sua implementação no LAD da PUCRS. Atualmente, a configuração somente pode ser realizada forma manual. Cada vez que um novo programa é executado no LAD é necessário instalar toda a *stack* de programas faltantes no *cluster*.

Para disponibilizá-lo no LAD é importante realizar uma avaliação do *singularity* como plataforma para execuções de alto desempenho. Avaliando as vantagens potenciais do uso do *singularity*, bem como as possíveis perdas de desempenho quando comparado a uma execução nativa. Também analisar a reprodutibilidade de aplicações, entendendo o quão difícil é criar e configurar o contêiner, a otimização do tempo de desenvolvimento, também verificando como é seu uso em ambientes de nuvem como a Amazon Web Services, cuja relevância tem aumentado significativamente no contexto de computação de alto desempenho (HPC) [16].

O trabalho visa avaliar se é possível utilizar contêineres no LAD com ganhos de otimização de tempo de desenvolvimento e perda mínima de desempenho na computação. Os principais tópicos da avaliação são os seguintes:

- Desempenho: Realizar testes de desempenho comparando o tempo médio de execução em testes rodando aplicações com e sem o *singularity* diretamente em um *cluster* na AWS. Esses resultados demonstram se existe algum cenário onde ocorre perda significativa de desempenho ao utilizar o *singularity*.
- Dificuldade de instalação e execução: Buscar entender se existem dificuldades ao instalar e executar o *singularity* em diferentes ambientes, verificando se há ganhos na reprodutibilidade.
- Limitações de programas: Análise se qualquer tipo de aplicação que necessite de HPC pode ser executada no *singularity*, por exemplo através de testes com aplicações que necessitam de *GPU* ou outras necessidades específicas.

V. Metodologia

Para realizar testes de desempenho, foram realizadas execuções em um *cluster* AWS utilizando OpenMPI, comparando os tempos médios das execuções. Existem diversas aplicações de código aberto disponíveis para realização de testes de desempenho em HPC que utilizam paralelismo. O objetivo é executar essas aplicações diretamente no *cluster* de forma nativa e também utilizando contêineres com o *singularity*.

A. Configuração do cluster na AWS

Para execução dos testes foi utilizado um *cluster* com 8 instâncias EC2 da AWS. As instâncias são do tipo m5.large, que são de propósito geral. Esse tipo de EC2 na AWS possui 2 vCPUs com 8GB de memória DDR4, processador Intel Xeon Scalable de até 3,1 GHz e largura de banda de até 10Gbps. Em todas as instâncias foi utilizado o Ubuntu Server 22.04 com as instalações do *singularity* 3.11.11 e OpenMPI 4.1.2.

As instâncias são configuradas dentro de uma mesma VPC e mesmo security group, onde uma é definida como *master* e possui acesso SSH à todas as outras para ser possível executar o OpenMPI.

Fig. 2. Instâncias criadas na AWS

Name	Instance ID	Instance state	Instance type
MasterUbuntuM5Large	i-06ddecdbfb39aa92b	Running	m5.large
Slave1UbuntuM5Large	i-0c447efbbfef9542a	Running	m5.large
Slave2UbuntuM5Large	i-0b1006ba91ab0cc58	Running	m5.large
Slave3UbuntuM5Large	i-0f9ca4d355674bdff	Running	m5.large
Slave4UbuntuM5Large	i-09b434f95e14d6db1	Running	m5.large
Slave5UbuntuM5Large	i-099403b25abdafd5d	Running	m5.large
Slave6UbuntuM5Large	i-032cbed812ddcc0c1	Running	m5.large
Slave7UbuntuM5Large	i-0c9fe09dba370f724	Running	m5.large

B. Benchmarks para avaliação de desempenho

O NPB (NASA *Advanced Supercomputing Parallel Benchmarks*) [17] foi escolhido para avaliação do *singularity*. Ele consiste em um conjunto de programas criados para realização de testes de desempenho com tipos diferentes de paralelismo projetado para simular diferentes casos de uso de HPC. Cada *benchmark* avalia um tipo de métrica, como tempo de execução, desempenho de comunicação, uso de memória e desempenho de processamento paralelo.

Os *benchmarks* usados para os testes desse artigo são:

- *Integer Sort* (IS): Esse *benchmark* faz a ordenação de números inteiros. Ele enfatiza a velocidade de computação de números inteiros e comunicação constante entre processos para definir a ordem dos números, assim como velocidade de acesso à memória.
- *Embarrassingly Parallel* (EP): Esse *benchmark* gera números aleatórios em cada uma das unidades de processamento separadamente e assim necessita de comunicação mínima entre os processos. No fim, avalia a capacidade de processamento sem interferência de outras partes da computação, como rede ou comunicação.
- *Conjugate Gradient* (CG): Esse *benchmark* avalia a capacidade do sistema de lidar com equações lineares esparsas. Ela realiza cálculos complexos e avalia a eficiência do sistema com matrizes esparsas.
- *MultiGrid* (MG): Este *benchmark* avalia a capacidade de um sistema de resolver equações diferenciais usando um método *multigrid*. Ele também enfatiza e avalia a eficiência na comunicação entre processos.
- *Fourier Transform* (FT): Esse *benchmark* avalia a eficiência de um sistema de computação paralela ao calcular a Transformada de Fourier Discreta. É um teste que também enfatiza de maneira significativa a comunicação entre processos.
- *Lower-Upper symmetric Gauss-Seidel* (LU): Esse *benchmark* avalia a capacidade do sistema de lidar com equações lineares usando o método *Lower-Upper symmetric Gauss-Seidel*. Realiza cálculos complexos e também a comunicação intensiva entre processos.

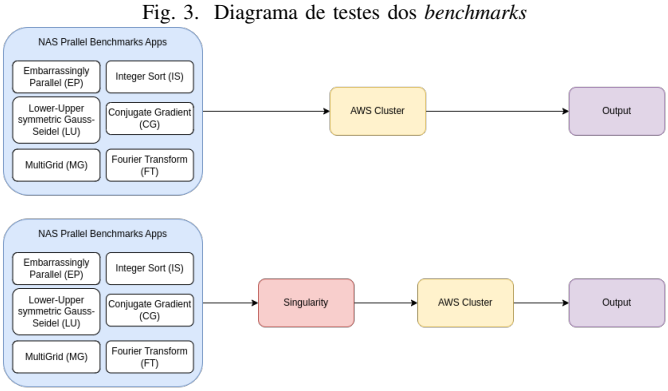
Cada *benchmark* do NPB possui diversas classes [18] que definem o tamanho da estrutura principal utilizada durante a execução. No caso dos *benchmarks* citados anteriormente, as classes disponíveis são:

- S e W: desenvolvida para testes rápidos.
- A, B e C: desenvolvida para testes em cenários padrão. O tamanho da estrutura cresce aproximadamente 4x entre cada classe.
- D, E e F: desenvolvida para testes longos e para rodar nos maiores laboratórios de supercomputadores do mundo. O tamanho da estrutura cresce aproximadamente 16x entre cada classe.

Para a realização dos testes foram escolhidos os *benchmarks* citados anteriormente com as classes S, W, A, B e C. As classes maiores não foram utilizadas, pois elas requerem um processamento maior do que o disponível atualmente no *cluster* criado na AWS.

Todos os testes foram conduzidos utilizando OpenMPI [19], dado que são casos de uso mais comuns e mais próximos aos utilizados no LAD da PUCRS. Ao utilizar o OpenMPI é possível testar o cenário onde os nós de processamento são distribuídos no *cluster*, não possuindo memória compartilhada. Para automatizar a execução dos testes foi criado um programa em Python que executa comandos com *shell* no Linux com o comando mpirun e executa todas as *benchmarks*. Para ter resultados mais precisos, o programa executa 5 vezes cada *benchmark* nativamente e com o *singularity*, utilizando os mesmos parâmetros. O próprio NPB expõe o resultado da execução, o principal dado utilizado é o tempo de execução exposto pelos *benchmarks*, em segundos. A média de tempo das 5 execuções é salva em um arquivo csv que é posteriormente utilizado para gerar os gráficos comparativos.

Fig. 3. Diagrama de testes dos *benchmarks*



Na Figura 3 é possível ver o exemplo do fluxo seguido para realização dos testes, onde foi executado os mesmos programas diretamente no *cluster* da AWS e depois novamente com os mesmos parâmetros, porém utilizando o *singularity*.

Assim como em todo o mercado, na pesquisa e na computação de alto desempenho há um significativo crescimento no uso de plataformas de computação em nuvem por trazerem muitas facilidades, como redução de custos e simplicidade no uso e configuração de máquinas. A AWS (Amazon Web Services) atualmente é a plataforma mais utilizada e

Fig. 4. IS *benchmark*

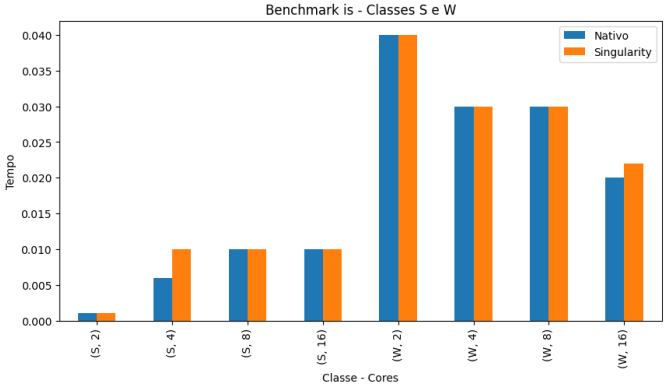
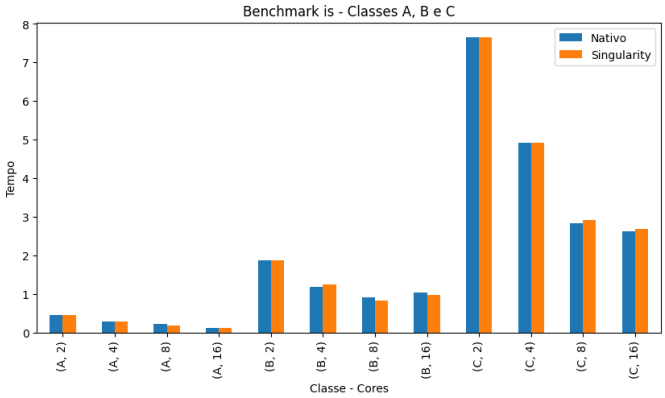


Fig. 5. IS *benchmark*



ela provê diversos serviços, incluindo *clusters* para HPC e instâncias EC2 que, por meio de conexões por redes de alta velocidade, conseguem trabalhar em conjunto. E agregado a isso conseguimos utilizar contêineres nesses ambientes que serão utilizados nos testes propostos.

VI. Resultados

A. Avaliação de Desempenho

Nos gráficos das Figuras 4 até 15 são mostrados os resultados dos testes comparativos entre a execução nativa e o *singularity*. Eles estão divididos em 2 gráficos por *benchmark*, onde o primeiro mostra as classes S e W e o outro mostrando as classes A, B e C. As barras representam o tempo médio em segundos das 5 execuções feitas.

No geral, é possível notar uma variação de tempo mais inconsistente nas classes menores, que executam muito rapidamente e ocorre uma diferença maior entre os tempos de execução, com cenários onde até mesmo a execução nativa durou um tempo maior.

Nas Figuras 4 e 5 é possível avaliar o primeiro cenário de testes. Com poucos dados, a classe S inclusive executa de maneira mais rápida ao utilizar somente dois núcleos de processamento, pois a troca de contexto acaba consumindo mais tempo do que executar o processamento em si. Ao utilizar o *singularity*, é possível notar uma diferença no tempo de execução, porém muito pequena, mesmo nas classes maiores onde o tempo de execução é muito maior e ocorrem muitas comunicações entre os processos, ou seja, na comunicação entre os contêineres instalados nos nós do cluster.

Nos gráficos das Figuras 6 e 7 é possível avaliar o cenário de testes onde o processamento dos dados é muito mais importante que a comunicação entre eles. Isso significa que ocorrem poucas comunicações entre os contêineres. Os gráficos demonstram que ao usar o *singularity*, existe um pequeno aumento de tempo médio de execução.

Os gráficos das figuras 8 e 9 mostram resultados mais consistentes, onde a diferença de tempo é muito pequena, sendo que em alguns casos ao utilizar o *singularity*, o tempo foi até menor. Isso se deve principalmente ao ambiente de teste

Fig. 6. EP *benchmark*

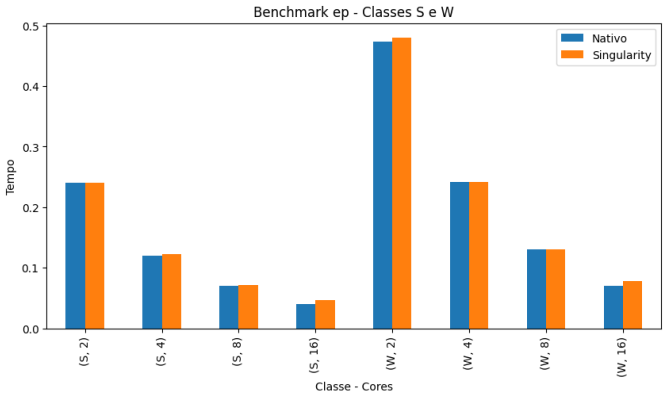


Fig. 7. EP *benchmark*

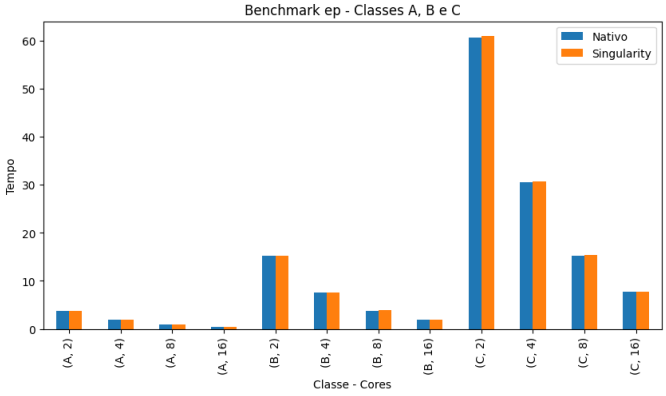


Fig. 8. CG *benchmark*

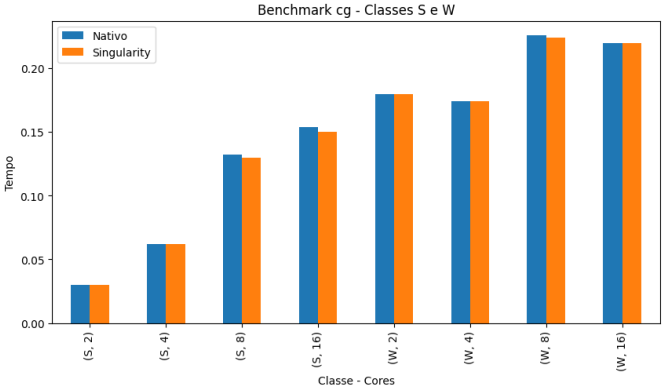
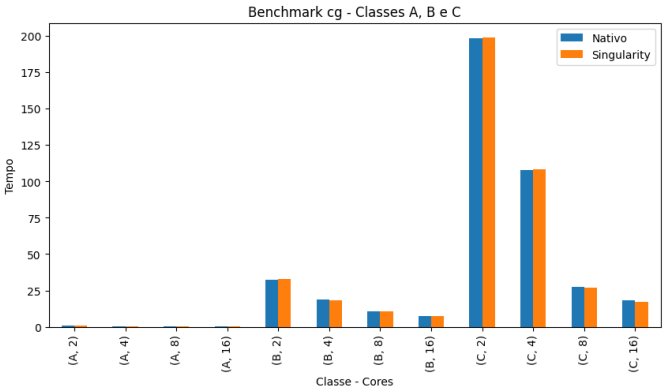


Fig. 9. CG *benchmark*



possuir variações que não são possíveis de serem controladas. A diferença de tempo entre os dois modos se torna irrelevante.

Os gráficos das Figuras 10 e 11 mostram resultados parecidos com os vistos nos testes do CG, porém com maior consistência. Nos principais testes, que são os de maior escala, o tempo de execução foi exatamente o mesmo na maioria dos cenários.

Fig. 10. MG *benchmark*

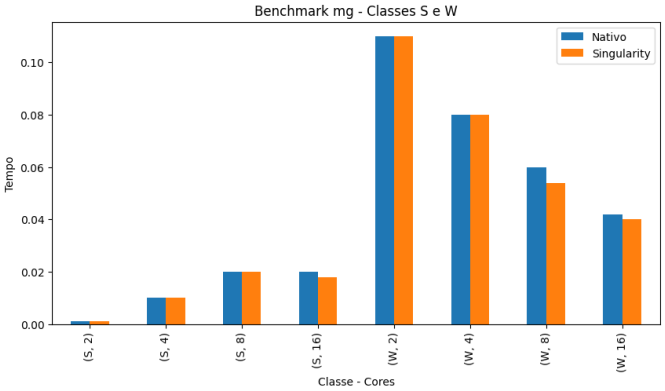


Fig. 11. MG *benchmark*

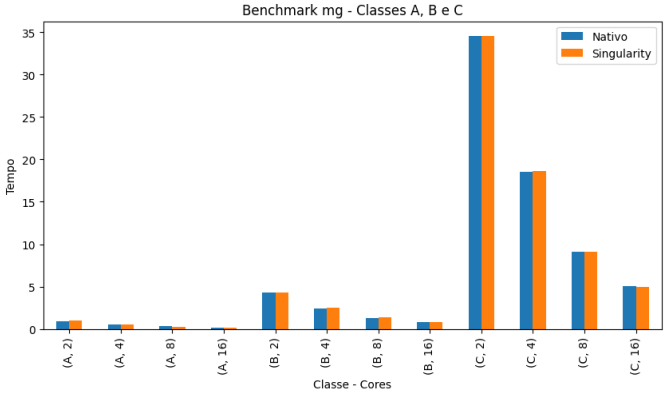
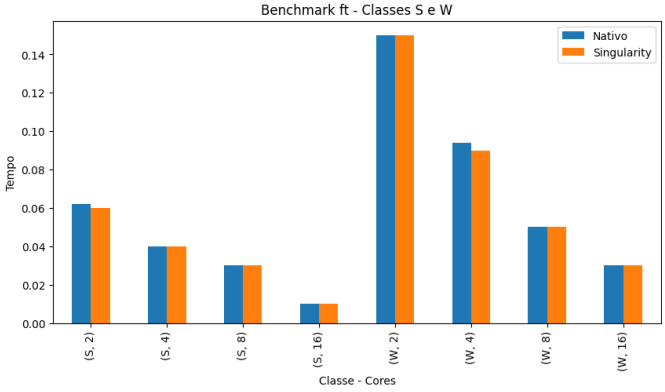


Fig. 12. FT *benchmark*



Os gráficos das Figuras 12, 13, 14 e 15 mostram que não há diferença de tempos de execução nativa e com o *singularity*. Nesses *benchmarks* alguns cenários a execução nativa foi mais rápida.

Fig. 13. FT *benchmark*

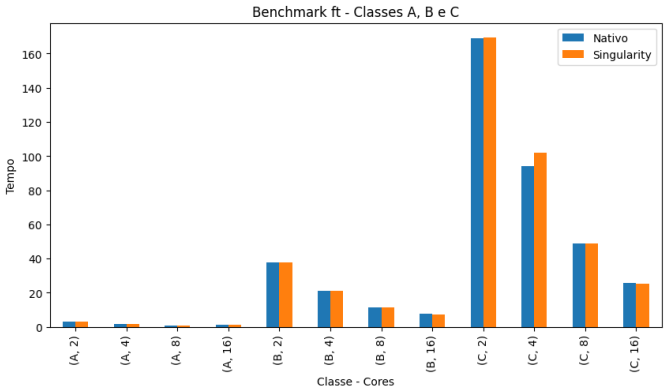


Fig. 14. LU *benchmark*

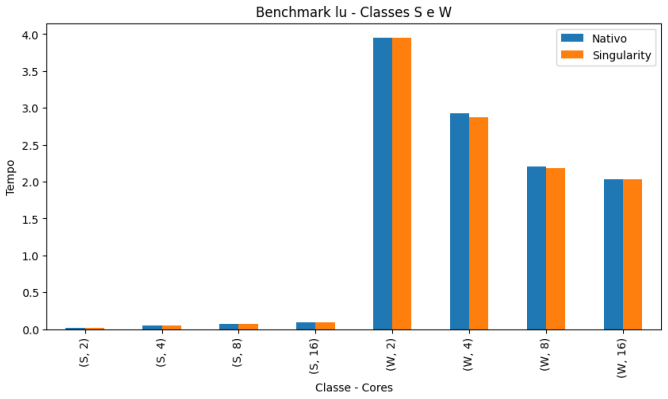
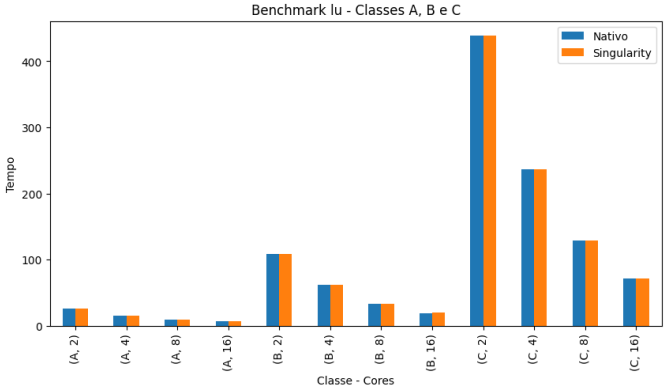


Fig. 15. LU *benchmark*



B. *Portabilidade e reprodutibilidade*

A portabilidade e a reprodutibilidade também são importantes ao se analisar esse tipo de ferramenta, pois esse é um dos principais objetivos ao se utilizar contêineres. Para configurar o contêiner com as características necessárias para rodar os programas do NPB é bastante simples.

Basta criar um arquivo texto seguindo o formato do *singularity* como visto na figura 16. Nas primeiras linhas do arquivo é mostrado que a fonte das imagens utilizadas será o docker e define a imagem base como o Ubuntu na versão 22.04 LTS. Nas linhas seguintes há a seção de *post*, que define o que será executado logo após a imagem base ser inicializada. Os comandos dessa seção estão instalando programas necessários para compilação do NPB e também compilando a *benchmark* IS com classe C. A seção *runscript* define o *script* que será executado quando o contêiner for iniciado. Com isso, é possível criar uma imagem do *singularity* com esse arquivo e executa-la em qualquer ambiente que possua o *singularity* instalado deixando fácil reproduzir execução em diferentes ambientes.

Após criar o arquivo com esses passos, somente é necessário gerar a imagem e copiá-la para os nós do *cluster* que possuem o *singularity* instalado, e também para qualquer *cluster* que necessite rodar a aplicação. Por fim, basta utilizar os comandos do *singularity* para executá-la. Isso torna muito fácil replicar a execução de aplicações no mesmo *cluster* ou em *clusters* diferentes.

Fig. 16. *Singularity* File

```
1  Bootstrap: docker
2  From: ubuntu:latest
3
4  %post
5      apt update
6      apt --assume-yes install wget build-essential openmpi-bin openmpi-common openmpi-doc libopenmpi-dev
7      wget -q https://www.nas.nasa.gov/assets/npb/NPB3.4.2.tar.gz
8      tar -xvzf NPB3.4.2.tar.gz
9      cd NPB3.4.2/NPB3.4-MPI
10     cp config/make.def.template config/make.def
11     cp config/suite.def.template config/suite.def
12     sed -i '/^FFLAGS/s/$/ -fallow-argument-mismatch/' config/make.def
13     echo "sp      S
14 sp      W
15 lu      A
16 is      C" > config/suite.def
17     make suite
18
19 %runscript
20     cd /NPB3.4.2/NPB3.4-MPI
21     ./bin/is.C.x
```

VII. Conclusão e Trabalhos Futuros

Os resultados dos testes demonstram que, apesar da pequena variação nos tempos nos cenários testados, o uso do *singularity* possui um desempenho muito próximo ao da execução nativa. O *singularity* pode incluir um pequeno *overhead* no desempenho, porém esse *overhead* é muito pequeno ou nenhum. É possível notar que mesmo em variados contextos, com cálculos complexos, matrizes esparsas, muito acesso à memória, alta comunicação entre processos ou até mesmo baixa comunicação entre processos, os resultados são parecidos. Em alguns casos foi possível notar que a execução nativa foi mais lenta que com o *singularity*, assim como também, mesmo usando a média de 5 execuções nos testes, ocorrerá uma pequena variação de tempo caso a execução seja feita novamente, isso é devido ao resultado natural da complexidade e da natureza estocástica dos sistemas computacionais.

Essa análise têm implicações para a computação de alto desempenho, sugerindo que o uso de contêineres pode oferecer benefícios de reprodutibilidade sem comprometer o desempenho das execuções. Com base nesses resultados, é possível concluir que a implementação do *singularity* no LAD da PUCRS é uma medida que trará benefícios significativos com uma nova opção de plataforma, tanto em termos de eficiência quanto em termos de facilidade de uso e reprodutibilidade.

Para trabalhos futuros é planejado realizar os mesmos testes porém utilizando outras ferramentas de contêiner para HPCs, como o Shifter e o Charliecloud. E também avaliá-las como plataformas para computação de alto desempenho. Assim como também executar no laboratório de alto desempenho da PUCRS.

Todos os códigos usados nesse artigo encontram-se no Github, através da URL:

https://github.com/deividfsantos/faculdade-8-semestre/tree/main/trabalho-conclusao-curso-2

Referências

[1] H. V. Bacellar, “Cluster: Computação de alto desempenho,” *Universidade de Campinas*, 2010.

[2] A. Mouat, *Using docker: developing and deploying software with containers*. " O'Reilly Media, Inc.", 2015.

[3] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[4] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[5] W. Gropp, W. D. Gropp, E. Lusk, A. Skjellum, and A. D. F. E. E. Lusk, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[6] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, "Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges," *ACM Comput. Surv.*, vol. 51, no. 1, jan 2018. [Online]. Available: https://doi.org/10.1145/3150224

[7] S. Giallorenzo, J. Mauro, M. G. Poulsen, and F. Siroky, "Virtualization costs: benchmarking containers and virtual machines against bare-metal," *SN Computer Science*, vol. 2, no. 5, p. 404, 2021.

[8] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[9] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for hpc," in *Journal of physics: Conference series*, vol. 898, no. 8. IOP Publishing, 2017, p. 082021.

[10] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2017, pp. 1–10.

[11] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper 9*. Springer, 2003, pp. 44–60.

[12] "Docker hub documentation," https://docs.docker.com/docker-hub/, acessado: 2023-06-03.

[13] E. Le and D. Paz, "Performance analysis of applications using singularity container on sdsc comet," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, ser. PEARC17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3093338.3106737

[14] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 74–81.

[15] R. S. Canon and A. Younge, "A case for portability and reproducibility of hpc containers," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2019, pp. 49–54.

[16] "The evolution of containers: Docker, kubernetes and the future," https://www.techtarget.com/searchitoperations/feature/Dive-into-the-decades-long-history-of-container-technology, acessado: 2023-06-01.

[17] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306

[18] "Problem sizes and parameters in nas parallel benchmarks," https://www.nas.nasa.gov/software/npb_problem_sizes.html, acessado: 2023-06-03.

[19] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 2004, pp. 97–104.