UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Weslley da Costa Silva

Implementação de um Cluster para Aplicações de HPC utilizando Docker e Infiniband

Florianópolis
2023

Weslley da Costa Silva

Implementação de um Cluster para Aplicações de HPC utilizando Docker e Infiniband

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Orientador: Guilherme Arthur Gerônimo, Dr.
Coorientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Silva, Weslley da Costa
Implementação de um Cluster para Aplicações de HPC utilizando Docker e Infiniband / Weslley da Costa Silva ; orientador, Guilherme Arthur Gerônimo, coorientador, Odorico Machado Mendizabal, 2023.
52 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciências da Computação. 2. Docker. 3. HPC. 4. Infiniband. 5. SWARM. I. Gerônimo, Guilherme Arthur. II. Mendizabal, Odorico Machado. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Weslley da Costa Silva

Implementação de um Cluster para Aplicações de HPC utilizando Docker e Infiniband

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 25 de Julho de 2023.

Profª. Lúcia Helena Martins Pacheco, Dr.
Coordenador do Curso

Banca Examinadora:

Guilherme Arthur Gerônimo, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Odorico Machado Mendizabal, Dr.
Coorientador
Universidade Federal de Santa Catarina

Profª. Patrícia Della Méa Plentz, Dra.
Avaliador
Universidade Federal de Santa Catarina

Prof. Roberto Willrich, Dr.
Avaliador
Universidade Federal de Santa Catarina

Este trabalho é dedicado ao Google, mas aos meus pais e amigos também

AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão à minha família por nunca desistir de mim, por me apoiar e por estar presente em cada etapa desta jornada. Agradeço especialmente aos meus pais, Walter e Ivonete, por seu amor incondicional, paciência e por sempre me ouvirem, aconselharem e acalmarem nos momentos mais desafiadores. Seu apoio inabalável foi fundamental para que eu pudesse alcançar meus objetivos.

Quero dedicar um agradecimento especial ao meu amigo e quase irmão Cleiton. Mesmo me chamando de louco por trabalhar, estudar e ainda ter tempo para projetos paralelos, ele sempre esteve lá para me ajudar. Sua contribuição inestimável com meus artigos e até mesmo com a apresentação deste trabalho é algo que jamais esquecerei. Sua amizade e irmandade foram fundamentais para meu crescimento acadêmico e pessoal.

Expresso minha gratidão ao meu orientador, por sua orientação, conhecimento e sabedoria compartilhados ao longo deste trabalho.

Agradeço também ao meu coorientador Prof. Odorico, que durante nossos encontros me auxiliou, ensinou e foi muito mais do que um professor para mim. Sua dedicação e paixão pelo ensino deixaram uma marca em mim e sou grato por ter tido a oportunidade de aprender com ele.

Por fim, agradeço aos meus amigos, tanto aqueles que estiveram ao meu lado durante todo o percurso acadêmico, como também aqueles que estiveram distantes fisicamente, mas sempre presentes com palavras de encorajamento e apoio. Sua amizade e companheirismo foram um suporte vital ao longo dessa jornada, e sou grato por cada momento compartilhado, risadas e apoio mútuo.

A todos que mencionei e a todos os amigos que fizeram parte desta caminhada, meu mais sincero agradecimento. Suas presenças em minha vida foram e sempre serão inestimáveis.

Gostaria por fim, de expressar meu profundo agradecimento à SeTIC (Superintendência de Governança Eletrônica e Tecnologia da Informação e Comunicação) por todo o suporte e colaboração fornecidos ao longo deste trabalho. Agradeço especialmente por terem cedido o espaço e os equipamentos necessários para a realização dos testes e experimentos. Sem o apoio e recursos disponibilizados pela SeTIC, este trabalho não seria possível.

Weslley, Relaxa...
(zé)

RESUMO

Diante dos desafios da Computação de Alto Desempenho, surgem questões agravantes ao tentar provê-la na forma de autosserviço, como (i) a diversidade de softwares a serem suportados; (ii) a constante necessidade de atualização (bibliotecas, softwares, sistema operacional, etc.) e (iii) a execução em hardwares heterogêneos. Sanar estes pontos utilizando apenas os métodos tradicionais de gerenciamento de *clusters* ou de configuração, se demonstrou inviável dada a flexibilidade exigida pelo cenário. Assim, o objetivo deste projeto é o de desenvolver uma solução que (i) simplifique a manutenção dos clusters, (ii) possibilite ao usuário controlar o ambiente de execução do software e (iii) suporte uma diversidade de softwares evitando conflitos. Este trabalho aborda o contexto, requisitos, modelagem e a implementação do serviço na Superintendência de Governança Eletrônica e Tecnologia da Informação e Comunicação (SeTIC), criando um ambiente constituindo por 7 clusters que, juntos, somam mais de 40 nós, 7TB de RAM, 1.200 núcleos e 40.000 núcleos de processamento gráfico.

Palavras-chave: Docker. Infiniband. HPC. SLURM. SWARM.

RESUMO ESTENDIDO

Introdução

O capítulo aborda a importância da High Performance Computing (HPC) para aplicações científicas e de engenharia, destacando que muitos grupos de pesquisa optam por investir em infraestrutura própria para garantir maior controle e privacidade sobre os recursos computacionais. No entanto, a aquisição e manutenção dessa infraestrutura pode ser um desafio, principalmente para grupos de pesquisa com recursos financeiros limitados. Como alternativa, surgiu a ideia de clusters compartilhados entre grupos de pesquisa, mas isso trouxe novos desafios, como conflitos de dependências entre aplicações e problemas de usabilidade.

Com a evolução dos contêineres de software, tornou-se possível executar diversas aplicações com requisitos distintos em um único ambiente, utilizando contêineres isolados. Essa modernização resultou em estudos que exploram o uso de contêineres em ambientes de HPC, demonstrando a viabilidade do uso de ferramentas como Docker Swarm e Kubernetes em ambientes de alto processamento de dados e alta disponibilidade. Alguns desses estudos são citados, evidenciando o potencial dessas tecnologias para o gerenciamento eficiente de recursos e aplicativos em ambientes de HPC.

Objetivos

Este trabalho tem como objetivo geral configurar e implementar um ambiente baseado em contêineres de software para execução de aplicações em cluster de alto desempenho (HPC). A contribuição pretendida é simplificar a administração de clusters, permitindo o provisionamento fácil de novos servidores e possibilitando o uso de uma variedade maior de softwares, superando problemas de incompatibilidade entre bibliotecas. Para atingir esse objetivo, os objetivos específicos são: realizar um estudo sobre HPC e tecnologias utilizadas, explorar a virtualização e contêineres de software, configurar um cluster de alto desempenho usando contêineres, estabelecer uma rede de baixa latência para o cluster, disponibilizar o ambiente para a comunidade acadêmica e avaliar o desempenho da rede antes e após as alterações realizadas.

Resultados e Discussão

Após a implementação de um cluster de HPC utilizando Docker, constatou-se que é possível compartilhar uma infraestrutura entre diferentes usuários. No entanto, a utilização de redes infiniband apresentou desafios que requerem estudos mais aprofundados para otimizar seu desempenho. Alternativas como o Singularity possibilitam um melhor aproveitamento da rede infiniband, mas não oferecem a mesma facilidade de uso do Docker ou do Swarm como orquestrador.

Considerações Finais

Como estudo de caso na UFSC, observou-se um crescimento no uso por parte dos grupos de pesquisa. Durante um período de 6 meses, o autosserviço desenvolvido atendeu a 6 grupos diferentes, fornecendo mais de 23 softwares distintos. Foram processadas mais de 1400 horas e submetidos mais de 420 jobs, mesmo com a limitação da velocidade da rede infiniband. Isso indica que o projeto do serviço está na direção correta.

Para os próximos passos, é necessário aprofundar a pesquisa sobre a integração da rede infiniband com contêineres Docker, buscando uma forma de conectar a interface infiniband do host diretamente aos contêineres com o menor impacto possível na performance.

Palavras-chave: Docker. Infiniband. HPC. SLURM. SWARM.

ABSTRACT

In the face of High Performance Computing challenges, there are aggravating issues when attempting to provide it in a self-service manner, such as (i) the diversity of software to be supported, (ii) the constant need for updates (libraries, software, operating system, etc.), and (iii) execution on heterogeneous hardware. Addressing these points using traditional cluster management or configuration methods proved to be unfeasible given the flexibility required by the scenario. Therefore, the objective of this project is to develop a solution that (i) simplifies cluster maintenance, (ii) enables users to control the software execution environment, and (iii) supports a diversity of software, avoiding conflicts. This work addresses the context, requirements, modeling, and implementation of the service at SETIC, creating an environment consisting of 7 clusters that, together, total over 40 nodes, 7TB of RAM, 1,200 cores, and 40,000 graphics processing cores.

Keywords: Docker. Infiniband. HPC. SLURM. SWARM.

LISTA DE ILUSTRAÇÕES

Figura 1 – Processamento concorrente de tarefas em um *cluster* 30
Figura 2 – Processamento paralelo de tarefas em um *cluster* 31
Figura 3 – Estrutura de uma *Subnet Infiniband* 33
Figura 4 – Comparação Sistema Tradicional x Sistema Virtualizado 34
Figura 5 – Tipos de Hipervisores: (5a) Hipervisor Tipo 1; (5b) Hipervisor Tipo 2; (5c) Virtualização baseada em Contêineres. Adaptado de (LAUREANO; MAZIERO, 2008). 34
Figura 6 – Tipos de Hipervisores: (6a) Hipervisor Tipo 1; (6b) Hipervisor Tipo 2; Adaptado de (LAUREANO; MAZIERO, 2008). 36
Figura 7 – Arquitetura Funcional do *Network File System* (NFS) 38

Figura 8 – Yu e Huang (2015) 40
Figura 9 – Comunicação SSH entre nós *Master* e *Workers* 40

Figura 10 – Diagrama de interação. 43
Figura 11 – *Cluster* HPC OCN 44
Figura 12 – Imagens do *Cluster* OCN 45
Figura 13 – Distribuição de Contêineres no *Cluster* 46
Figura 14 – Diagrama de interação completa 47
Figura 15 – Interface Portainer - Visualização do *Cluster* 48
Figura 16 – *Dashboard* para a visualização de Jobs 48
Figura 17 – Comparação de velocidade da rede infiniband 49

LISTA DE TABELAS

Tabela 1 – Trabalhos selecionados 39

Tabela 2 – Hardware do Cluster OCN 44

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface. 31, 37
CPU	*Central Processing Unit*. 31, 32
GPU	*Graphics Processing Unit*. 31, 35, 36, 41
HPC	*High Performance Computing*. 25, 26, 29, 31, 32, 35, 36, 39, 41, 47, 49, 51
IA	Inteligência Artificial. 25
IBA	*InfiniBand Architecture*. 32
IBTA	*InfiniBand* ^{SM} *Trade Association*. 32
LXC	*Linux Container*. 35
MPI	*Message Passing Interface*. 30, 36, 40
NAT	*Network Address Translation*. 39
NFS	*Network File System*. 17, 38
RDMA	*Remote Direct Memory Access*. 32
RPC	*Remote Procedure Call*. 38
SeTIC	Superintendência de Governança Eletrônica e Tecnologia da Informação e Comunicação. 11
SLURM	*Simple Linux Utility for Resource Management*. 30, 31, 32
SQL	*Structured Query Language*. 47
SSH	*Secure Socket Shell*. 40
TCP	*Transmission Control Protocol*. 48
UDP	*User Datagram Protocol*. 49
UFSC	Universidade Federal de Santa Catarina. 41, 43, 51
VM	*Virtual Machine*. 34
VMM	*Virtual Machine Monitor*. 34

SUMÁRIO

1 INTRODUÇÃO 25
1.1 MOTIVAÇÃO E JUSTIFICATIVA 25
1.2 OBJETIVOS 26
1.2.1 Objetivo Geral 26
1.2.2 Objetivos Específicos 26
1.3 ORGANIZAÇÃO DO TRABALHO 27

2 CONCEITOS BÁSICOS 29
2.1 COMPUTAÇÃO DISTRIBUÍDA 29
2.1.1 *Clusters* e Processamento de Alto Desempenho 29
2.2 SLURM 30
2.3 *INFINIBAND* 32
2.4 VIRTUALIZAÇÃO 33
2.5 *CONTÊINERES* 35
2.6 ORQUESTRADORES DE CONTÊINERES 36
2.6.1 *Docker Swarm* 37
2.7 NFS 38

3 TRABALHOS RELACIONADOS 39

4 ABORDAGEM E DESENVOLVIMENTO 41
4.1 REQUISITOS DO SERVIÇO 41
4.1.1 Requisitos Funcionais 41
4.1.2 Requisitos Não Funcionais 41
4.2 DESIGN PROPOSTO 42
4.3 HARDWARE UTILIZADO 43
4.3.1 Servidores 44
4.3.2 Rede 44
4.4 IMPLEMENTAÇÃO 45
4.5 CENÁRIOS DE TESTES E RESULTADOS 48

5 CONSIDERAÇÕES FINAIS 51
5.1 PRÓXIMOS PASSOS 51

REFERÊNCIAS 53

APÊNDICES 57

APÊNDICE A – ARTIGO DO TCC 59

APÊNDICE B – GERADOR DE GRÁFICO PARA SAÍDA DO IPERF 67

APÊNDICE C – INFORMAÇÕES DE CONFIGURAÇÃO DO INFINIBAND 69

ANEXOS 71

ANEXO A – DOCKERFILE SLURM 73

ANEXO B – DOCKERFILE WORKFLOW READ 77

ANEXO C – DOCKERFILE WORKFLOW LOAD 79

ANEXO D – DOCKERFILE WORKFLOW STATUS 81

ANEXO E – DOCKERFILE IPERF 83

ANEXO F – DOCKER STACK TESTE IPERF 85

1 INTRODUÇÃO

A *High Performance Computing* (HPC) é uma área crucial para aplicações científicas e de engenharia, em que o poder computacional é fundamental para o sucesso dos projetos. Embora existam serviços de HPC em nuvem disponíveis para o público em geral fornecidos pela Oracle[1], AWS[2] e IBM[3], muitos grupos de pesquisa preferem investir em sua própria infraestrutura para garantir maior privacidade e controle sobre os recursos computacionais disponíveis.

No entanto, a aquisição e manutenção de uma infraestrutura própria pode ser um grande desafio, em especial para pequenos grupos de pesquisa com poucos recursos financeiros ou até mesmo sem recursos humanos suficientes de forma a conseguir gerenciar e manter todo o ambiente necessário. De forma a reduzir o alto custo inicial da estrutura, os custos envolvendo suporte e manutenção especializada, e a depreciação dos equipamentos, começou-se a buscar alternativas como *clusters* compartilhados entre grupos de pesquisa. Porém, com o uso compartilhado cria-se novas variáveis na equação como diferentes aplicações rodando em um mesmo ambiente, aplicações com requisitos e finalidades distintas, surgindo assim problemas oriundos deste compartilhamento de recursos como: conflitos de dependências entre aplicações, indisponibilidade de recursos, problemas de usabilidade, entre outros.

Com a modernização de tecnologias como a de contêineres de software, o ambiente torna-se flexível a ponto que seja possível a execução de diversas aplicações de requisitos variados, tudo em um único ambiente com diversos contêineres de aplicação isolados. Como consequência dessa modernização, surgiram estudos como: Yu e Huang (2015), Ermakov e Vasyukov (2017), Zhou et al. (2021) e Corrêa et al. (2016) que tratam da utilização de contêineres em ambientes de HPC, demonstrando a viabilidade do uso de ferramentas como Docker Swarm (MORAVCIK; KONTSEK, 2020) e Kubernetes (LUKSA, 2017) em ambientes de alto processamento de dados e alta disponibilidade.

1.1 MOTIVAÇÃO E JUSTIFICATIVA

Com o aumento expressivo do uso de computação em toda a nossa sociedade, ferramentas de Inteligência Artificial (IA) permeiam-nos diariamente através de carros quase autônomos, chats inteligentes, aplicações de reconhecimento de voz e imagem, além de sua participação em pesquisas científicas. Com a sociedade cada vez mais dependente de soluções que utilizam infraestruturas de HPC e redes de baixa latência, mais percebemos a falta de flexibilidade em nossas soluções que demandam alto poder de computação e comunicação entre processos, de forma a encontrarmos dificuldades no gerenciamento

[1] https://www.oracle.com/br/cloud/hpc/
[2] https://aws.amazon.com/pt/hpc/
[3] https://www.ibm.com/br-pt/high-performance-computing

de grandes *clusters* onde tenta-se dispor do máximo de aplicações compatíveis com o ambiente disponibilizado. Em sua maioria os conflitos começam com as bibliotecas compartilhadas, onde um software precisa de uma versão específica e mais antiga, quanto outro necessita de uma versão da mesma biblioteca porém mais atualizada e com isso começa a incompatibilidade de algumas aplicações com o mesmo ambiente de outras.

Diante de complicações que impedem um uso mais amplo da infraestrutura disponibilizada, começam a surgir estudos da utilização de contêineres de software (CONDE, 2020) para prover um ambiente isolado para cada aplicação, impedindo assim os conflitos anteriormente citados.

Retirando-se as complicações e apresentando uma infraestrutura que possa atender mais amplamente diferentes aplicações de variados grupos de pesquisa, o custo de aquisição dos equipamentos pode ser melhor divididos e a manutenção mais centralizada e com menor custo. Diante dessa possibilidade, a SETIC/UFSC vê como uma ótima oportunidade de prover HPC como serviço para a comunidade acadêmica, de forma a centralizar esforços em um único ponto e manter uma infraestrutura heterogênea, atualizada e com grande poder computacional.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Este trabalho tem como objetivo geral a configuração e implementação de um ambiente baseado em contêineres de software para a execução de aplicações em *cluster* para HPC. A contribuição pretendida por este trabalho é a de facilitar a administração de *clusters* para processamento de alto desempenho, de forma a simplificar o provisionamento de novos servidores ao ambiente e possibilitar a utilização por parte dos usuários a uma maior variedade de softwares que antes devido a problemas de incompatibilidade entre bibliotecas e suas diferentes versões não era possível.

1.2.2 Objetivos Específicos

Os objetivos específicos necessários para atingir o objetivo geral deste trabalho são:

- Realização de um estudo sobre HPC e tecnologias utilizadas;
- Realização de um estudo sobre virtualização e contêineres de software;
- Configuração de um *cluster* de alto desempenho com a utilização de contêineres de software;
- Disponibilização de uma rede de baixa latência para o *cluster*;
- Disponibilização para uso pela comunidade acadêmica;

- Avaliação dos ambientes de rede prévio e posterior.

1.3 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está dividido em 5 capítulos.

O Capítulo 1 é responsável por introduzir as motivações e justificativas que dão base à este trabalho, assim como também explicitar seus objetivos gerais e específicos. No Capítulo 2, são apresentados os principais conceitos que fundamentam as plataformas e ferramentas utilizadas no decorrer deste trabalho, tendo assim o objetivo de trazer toda a base para o entendimento das tecnologias que compreendem esse trabalho e sua implementação. Já no Capítulo 3, são apresentados alguns trabalhos dentro do escopo de pesquisa e comparação do presente trabalho. No Capítulo 4, são apresentados os requisitos funcionais e não funcionais, a estrutura da implementação apresentada, interações entre as ferramentas, códigos desenvolvidos e o funcionamento da implementação. Por fim, no Capítulo 5 serão expostas as conclusões obtidas após a implementação do serviço, próximos passos e trabalhos futuros.

2 CONCEITOS BÁSICOS

Nesse capítulo serão expostos os conceitos que fundamentam a temática a ser abordada nesse trabalho.

2.1 COMPUTAÇÃO DISTRIBUÍDA

A computação distribuída é uma seção da Ciência da Computação cujo alvo é a melhoria do desempenho de aplicações distribuídas e paralelas a partir do uso de infraestruturas computacionais complexas (DANTAS, 2005). Essas estruturas complexas, são coleções de computadores que trabalham em conjunto para resolver um determinado problema, dividindo-o em diversas partes menores e processando cada parte em computadores distintos que comunicam-se entre si e que do ponto de vista do usuário é um único sistema coerente (TANENBAUM; STEEN, 2006).

Como uma de suas características, a computação distribuída tem a capacidade de agregar computadores existentes em redes convencionais em um único sistema homogêneo ou heterogêneo e com grande capacidade de escalabilidade (COULOURIS; DOLLIMORE; KINDBERG, 2005), sendo assim possível a formação de diferentes configurações.

2.1.1 *Clusters* e Processamento de Alto Desempenho

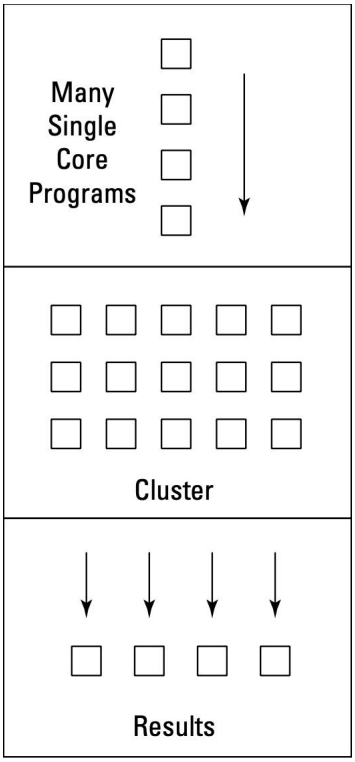
Um *cluster* de computadores caracteriza-se por um agrupamento de nós computacionais interconectados, executando aplicações coordenadas por um *software* gerenciador, que é responsável por delegar tarefas para cada nó do agrupamento e gerenciar toda a estrutura (TITTEL, 2012).

O uso desse tipo de estrutura computacional entrega uma porção de benefícios, dentre eles:

- Redução no tempo de solução, principalmente em simulações e modelos;
- Controle de Tarefas e priorização de trabalhos mais importantes, possibilitando um processamento mais ágil dos *Jobs*;
- Melhor aproveitamento de recursos computacionais;
- Melhora na confiabilidade, manutenção e disponibilidade do serviço mantido.

O processamento de alto desempenho - HPC, é uma classe de supercomputadores e *clusters* com o intuito de solucionar problemas avançados e de alta intensidade computacional (EADLINE, 2009). Esses computadores utilizam do poder computacional de cada nó combinado para aumentar a vazão de resolução de problemas computacionais, possibilitando assim executar diversas tarefas concorrentemente conforme pode ser visto na Figura 1, reduzindo o tempo final de computação exigida para uma solução.

Figura 1 – Processamento concorrente de tarefas em um *cluster*



Fonte: (EADLINE, 2009)

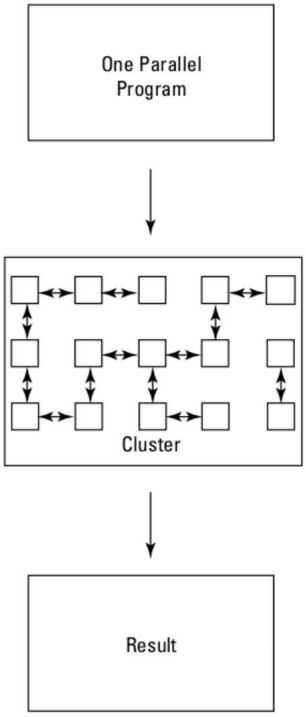
Além de ser possível utilizar esse poder computacional para o processamento concorrente, também é possível a utilização do mesmo para processar uma única tarefa paralelamente, dividindo em diversas sub-tarefas onde cada uma é destinada para um nó distinto, depois de terminada a sua computação junta-se os pedaços para se obter um único resultado, como é possível ver na Figura 2. A divisão de uma tarefa em sub-tarefas é feita no nível de *software*, isso quer dizer que a tarefa precisa ser modificada para que a mesma possibilite utilizar bibliotecas de processamento paralelo como a *Message Passing Interface* (MPI) para que seja possível esse processamento.

A partir do momento em que temos um alto nível de processamento de dados paralelos, se faz necessário um meio de comunicação com baixa latência e que consiga lidar com grandes fluxos de troca de informações entre tarefas e processos de modo que ainda se tenha confiabilidade e também velocidade na transmissão.

2.2 SLURM

O projeto *Simple Linux Utility for Resource Management* (SLURM) é um *software* de gerenciamento de recursos para computação de alto desempenho (YOO; JETTE; GRONDONA, 2003), desenvolvido pelo *Lawrence Livermore National Laboratory* (LLNL) nos Estados Unidos. O objetivo do SLURM é gerenciar a alocação de recursos computacionais de laboratório, incluindo a execução de trabalhos em *clusters* de computadores. O

Figura 2 – Processamento paralelo de tarefas em um *cluster*



Fonte: (EADLINE, 2009)

SLURM é amplamente utilizado em ambientes de HPC, como instituições de pesquisa e centros acadêmicos, e tem se tornado popular em todo o mundo, sendo considerado um dos melhores *softwares* de gerenciamento de recursos computacionais disponíveis atualmente.

O SLURM é projetado para receber *scripts* de trabalho em uma fila central e, em seguida, realocá-los para uma fila e agendá-los para executar de acordo com a disponibilidade de recursos e prioridade de execução. Isso permite que os usuários enviem trabalhos para execução e sejam automaticamente alocados de acordo com as políticas de gerenciamento de recursos definidas pelo administrador do sistema. O SLURM também possui recursos avançados de gerenciamento de filas, suporte a várias arquiteturas de hardware e uma API para desenvolvimento de ferramentas e integrações personalizadas.

Partição é um subconjunto lógico de recursos de um *cluster* gerenciado pelo SLURM, permitindo assim a divisão de recursos computacionais em grupos menores. As partições permitem que os administradores do sistema configurem políticas de gerenciamento de recursos, como limites de tempo de execução, número de tarefas simultâneas, quantidade de CPU, memória, GPU, etc. Isso possibilita uma gerência mais granularizada dos recursos, além de permitir a separação de recursos entre diferentes grupos de usuários ou para tarefas específicas. Por exemplo, pode-se criar uma partição para um grupo específico de usuários que precisa de acesso a um conjunto específico de recursos.

Job é uma unidade de trabalho submetida ao SLURM para ser executado em um *cluster*. Um *job* tem em sua estrutura uma ou mais tarefas, e ao ser submetido é acompa-

nhado de opções que especificam as necessidades de recursos a serem utilizados durante a sua execução, como quantidade de CPU, memória, tempo de execução, etc. O SLURM agendará o *job* para execução de acordo com a disponibilidade de recursos e as políticas de gerenciamento de recursos definidas para a partição em que o *job* foi submetido.

Job Step é uma parte dentro de um *job*, que é executada em um nó do cluster. Um *job* pode ter vários *job steps*, e os mesmos podem ser executados em sequência ou em paralelo por vários nós. Os *job steps* são úteis para dividir uma tarefa em partes menores que podem ser executadas em paralelo, acelerando a execução do *job*.

Políticas o SLURM permite aos administradores do sistema definir diversas políticas de alocação de recursos, de acordo com as necessidades e prioridades de cada usuário ou grupo de usuários. Algumas políticas comuns incluem limites de tempo de execução em trabalhos específicos, limites de recursos, políticas de prioridade, etc. As políticas podem ser configuradas para evitar o bloqueio de recursos computacionais por longos períodos, evitando que alguns *jobs* nunca sejam executados. Além disso, é possível criar políticas de uso justo, que dividem igualmente os recursos computacionais para todos os usuários, independente da quantidade de *jobs* submetidos por cada usuário.

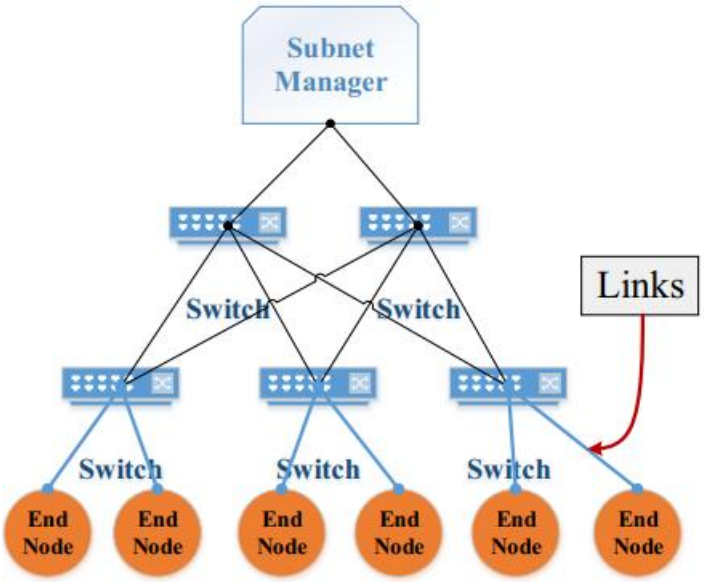
2.3 *INFINIBAND*

A *InfiniBand Architecture* (IBA) é uma arquitetura de rede de alta velocidade projetada de forma a conectar servidores e dispositivos de armazenamento em um data center (BUYYA; CORTES; JIN, 2002). Foi desenvolvido pela *InfiniBand* ^{SM} *Trade Association* (IBTA) para fornecer os níveis de confiabilidade, disponibilidade, desempenho e escalabilidade necessários para aplicações de HPC, como simulações, modelagem e análise de dados, e o que mais exigir largura de banda e baixa latência.

A principal motivação por trás do desenvolvimento do IBA foi o fato de o poder de processamento ter superado a capacidade de E/S de barramentos usados pela indústria. Assim, a IBA trabalha explicitamente com a E/S como comunicação, fornecendo comunicação de alta largura de banda com baixa sobrecarga entre dispositivos. Essa menor latência e melhoria no rendimento da transmissão ocorre devido o acesso direto a memória provido pelo *Remote Direct Memory Access* (RDMA) tanto no receptor quanto no transmissor.

A IBA trabalha a partir de uma pilha que tem inclusa a camada física, enlace, rede e transporte. A unidade básica de uma rede IBA é uma sub-rede, diante disso, uma sub-rede consiste em basicamente quatro componentes: host (*end node*), switches, *links* e gerenciamento da rede conforme pode ser visto na Figura 3.

Figura 3 – Estrutura de uma *Subnet Infiniband*



Fonte: Chung et al. (2016)

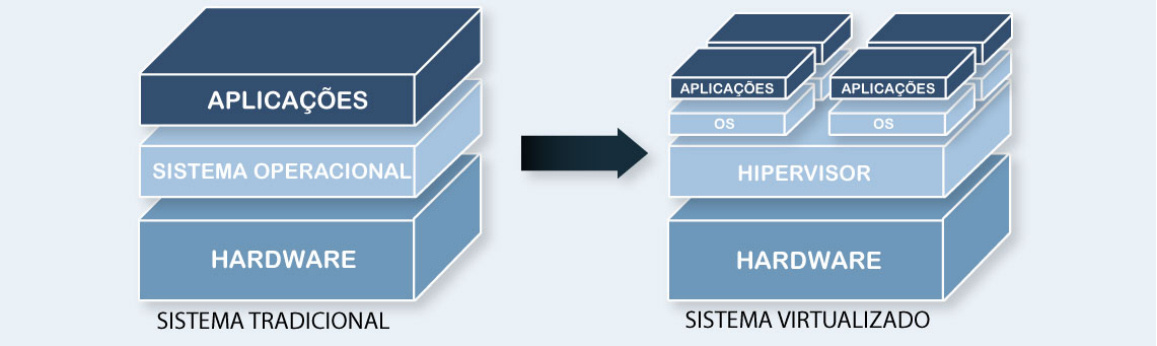
2.4 VIRTUALIZAÇÃO

Algum tempo atrás, entrávamos em *datacenters* e facilmente era encontrada a antiga filosofia de "um servidor por serviço", o que fazia sentido em diversas situações devido ao suporte e heterogeneidade dos clientes, além de sua segurança (CARISSIMI, 2008). Porém é de se imaginar que nem todo o potencial do *hardware* era devidamente aproveitado, já que por vezes algumas aplicações tinham uma carga de processamento muito baixa e assim não justificando por vezes o investimento. Nesse contexto, a virtualização surgiu como opção para reduzir esse problema.

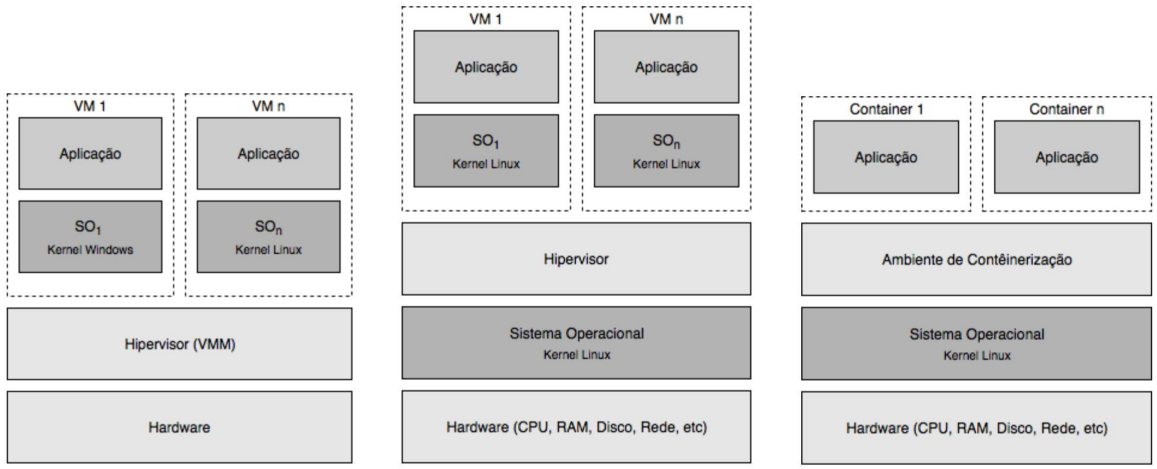
Um sistema computacional pode ser dividido em três camadas, sendo elas: *hardware*, responsável por executar as instruções recebidas do sistema operacional e de suas aplicações, o sistema operacional, responsável por fazer o controle de acesso ao *hardware* através das chamadas de sistemas, e por último as aplicações (LAUREANO; MAZIERO, 2008).

Diante dessas camadas descritas, percebe-se um hierarquização estrutural dos computadores formando-se interfaces para interação entre níveis, sendo possível serviços serem oferecidos às camadas diretamente ligadas. A partir desse ponto temos o surgimento da virtualização criando uma camada de *software* que serve como uma interface a fim de atuar entregando os serviços entre camadas. A Figura 4 apresenta a comparação entre um sistema tradicional e um sistema virtualizado, e é percebido que há uma camada de *software* chamada de *hipervisor*, servindo de interface entre o *hardware* e a camada de sistema operacional, possibilitando assim que as ações solicitadas por um sistema "A" possam ser executadas como ações de um sistema "B" (CARISSIMI, 2008).

Figura 4 – Comparação Sistema Tradicional x Sistema Virtualizado



Fonte: QNAP (2023)



(a) Hipervisor Tipo 1 (b) Hipervisor Tipo 2 (c) Conteinerização.

Figura 5 – Tipos de Hipervisores: (5a) Hipervisor Tipo 1; (5b) Hipervisor Tipo 2; (5c) Virtualização baseada em Contêineres. Adaptado de (LAUREANO; MAZIERO, 2008).

Diante da possibilidade de se utilizar uma camada de *software* como interface entre camadas, foi possível classificar e dividir a virtualização em dois principais tipos: virtualização a nível de *hardware* e a virtualização a nível de sistema operacional (Figura 5), onde sua classificação é dada pela localização da interface de *software* onde a mesma é aplicada.

A virtualização a nível de *hardware* tem como principal objetivo criar um ambiente onde seja possível emular a camada de *hardware* (Figura 5a). Esse tipo de virtualização tem a capacidade de interagir com a interface de *software* localizada entre a camada de *hardware* e o sistema operacional, possibilitando assim que que seja disponibilizado à máquina virtual o mesmo conjunto de instruções do processador físico, ou um outro diferente (CARISSIMI, 2008). O nome da camada de software responsável por esse tipo de virtualização se chama *Virtual Machine Monitor* (VMM), ou Hipervisor, Tipo 1. O principal trabalho deste hipervisor é fazer a tradução de todas as instruções do sistema operacional da *Virtual Machine* (VM) para o processador real. Como exemplos de hipervisores tipo 1 temos o Xen e o VMWare ESXi.

A virtualização a nível de sistema operacional, também conhecida como de VMM

Tipo 2 ou Hipervisor Tipo 2, acontece entre a camada do sistema operacional e a camada das aplicação permitindo que coexista espaços de usuário distintos no sistema operacional anfitrião (Figura 5b). São criadas partições lógicas independentes acima do sistema operacional, possibilitando a atuação de máquinas isoladas e a execução de aplicações, tudo isso a partir do núcleo compartilhado do sistema e cada espaço virtual tendo seus próprios recursos distintos, como rede e espaço de armazenamento (LAUREANO; MAZIERO, 2008). Como exemplos de hipervisor tipo 2 temos o Virtual Box e VMware Player (OLIVEIRA; CARISSIMI; TOSCANI, 2001).

Dentre as estratégias para criação de hipervisores temos: a virtualização total (ou completa), a paravirtualização e a conteinerização, que iremos no ater com mais detalhes a seguir.

2.5 *CONTÊINERES*

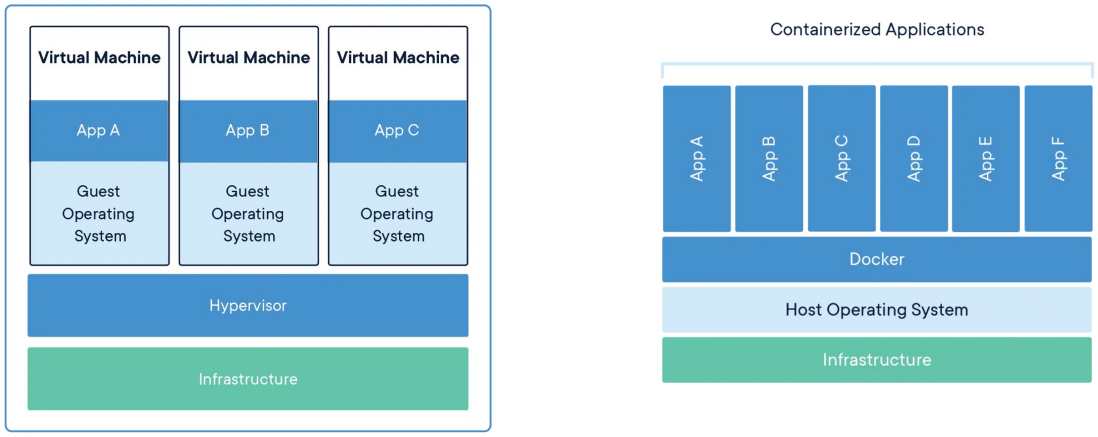
A conteinerização demonstrada na Figura 5c é uma estratégia caracterizada pela criação de diversas instâncias isoladas utilizando o *kernel* do sistema operacional hospedeiro. Criados a nível de sistema operacional, os ambientes virtuais isolados compartilham o mesmo *kernel* porém com usuários, sistema de arquivos *root*, interfaces de rede, processos e memórias próprios (FERREIRA et al., 2017).

Ao compararmos máquinas virtuais e *containers*, podemos ver similaridades no quesito isolamento e alocação de recursos computacionais, o que demonstra funcionalidade similar entre as duas arquiteturas porém tendo uma maior flexibilidade e eficiência em uma abordagem arquitetural distinta, conforme pode ser visto na Figura 5c.

Hoje temos uma diversidade de implementações de contêineres, cada qual com suas características, benefícios e problemas. Diante do paradigma de HPC, temos algumas implementações que já tem suporte nativo à interconexões de alto desempenho, utilização de aceleradores GPU e outras características que são de grande importância para um ambiente de softwares HPC. Abaixo são apresentadas implementações que são mais popularmente utilizadas.

- ***Docker***: é um projeto de código aberto construído tendo por base algumas tecnologias já conhecidas, como LXC Contêineres, virtualização de software e soluções baseadas em *hash* ou versionamento semelhante ao *git*. Atualmente é a plataforma padrão para implantação de microsserviços com ênfase em isolamento, ainda dispondo de uma plataforma de compartilhamento de imagens como o *Docker Hub*[1].
- ***Shifter***: uma das primeiras implementações de contêineres para HPC (GERHARDT et al., 2017). O *Shifter* permite a execução de contêineres sem privilégios e tem suporte à imagens *Docker* por meio de conversão para formato próprio do *Shifter*.

[1] https://hub.docker.com/



(a) Hipervisor Tipo 1 (b) Conteinerização.

Figura 6 – Tipos de Hipervisores: (6a) Hipervisor Tipo 1; (6b) Hipervisor Tipo 2; Adaptado de (LAUREANO; MAZIERO, 2008).

Também tem integração com o *Slurm* e suporte à vinculação do MPI do contêiner com o do host para aproveitar interconexões de alto desempenho.

- ***Singularity***: outro ambiente de execução de contêineres focado em HPC (KURTZER; SOCHAT; BAUER, 2017), permite a execução de contêineres sem privilégios mas para instalação ainda necessita de acesso administrativo, também tem suporte à conversão de imagens docker para o seu padrão. Com acesso direto à softwares otimizados pelo fabricante através de montagem no ambiente de execução, e suporte nativo a aceleradores GPU e interconexão de rede especializada.

2.6 ORQUESTRADORES DE CONTÊINERES

Orquestradores de contêineres são ferramentas que gerenciam o ciclo de vida dos contêineres em uma infraestrutura distribuída. Eles fazem a implantação, replicação de contêineres, monitoramento e o gerenciamento em um *cluster* ajudando a simplificar tarefas complexas, como distribuir contêineres em vários nós de um *cluster* mantendo o balanceamento da carga de trabalho, a descoberta de serviços, a alta disponibilidade de aplicações e a resiliência das aplicações.

- ***Docker Swarm***: O modo *Swarm*, também chamado de *Docker Swarm* (DOCKER, 2023a), é uma solução nativa do *Docker* para o gerenciamento e orquestração de *clusters* de contêineres *Docker* em larga escala. Por ser uma solução nativa do *Docker*, traz a simplicidade nos comandos e no processo de instalação e configuração do *cluster*.
- ***Kubernetes***: um orquestrador de contêineres de código aberto desenvolvido pelo Google (BURNS et al., 2016). Com recursos avançados de gerenciamento, automação, implantação, dimensionamento e operação de aplicativos em contêineres, o

Kubernetes se torna uma opção compatível com grandes ambientes onde a necessidade de escalabilidade das aplicações são bem atendidas.

- ***Apache Mesos***: uma plataforma de código aberto para o gerenciamento de *clusters* distribuídos e inclusive a orquestração de contêineres (FRAMPTON, 2018). Com pontos fortes em escalabilidade horizontal, tolerância a falhas e multiplexação de recursos, ele fornece uma camada de abstração que permite que os recursos do *cluster* possa ser compartilhado entre estruturas de aplicações, assim como o *Kubernetes* e o *Docker Swarm*

2.6.1 *Docker Swarm*

Desenvolvido em linguagem GO, a sua principal vantagem consiste em fazer parte do ambiente *Docker* (MORAVCIK; KONTSEK, 2020), fornecendo uma integração simples com a API do *Docker*. Todos os recursos disponíveis aos contêineres *Docker* podem ser utilizados no *Swarm* de forma a facilitar a gestão da infraestrutura por aqueles que já possuem conhecimentos do ambiente de contêineres *Docker*. Assim, não sendo preciso aprender conceitos de mecanismos de orquestração de outros fabricantes que podem ser casualmente diferentes do *Docker*. Abaixo é descrito um pouco da estrutura do *Swarm*.

Task|Tarefa: é uma unidade básica de trabalho atribuída aos nós do *cluster*. Representa a execução de um contêiner Docker em um nó específico, os comandos que definem esse contêiner, sua inicialização e funcionamento.

Service|Serviço: Define um conjunto de tarefas (tasks) de contêineres em um *cluster*. Normalmente composto por uma ou mais tarefas, cada uma representando a execução de um contêiner Docker em um nó do *cluster*.

Manager Nodes|Nós Mestre: Consiste um ou mais nós responsáveis por gerenciar o *cluster* e coordenar as atividades dos demais nós do *cluster*. É obrigatório ao menos um nó do tipo *manager* que será o responsável por controlar o estado do *cluster* e garantir que as *tasks* do serviços configurados estejam em execução pelos respectivos nós *workers*. Um nó *manager* também é constituído pelos seguintes itens:

- **API**: responsável por receber comandos dos usuários e criar novos serviços baseados em parâmetros definidos em comandos recebidos.
- ***Orchestrator*|Orquestrador**: Recebe as definições dos serviços e cria as tarefas
- ***Allocator*|Alocador**: Associa endereços IP
- ***Scheduler*|Escalonador**: Escalona tarefas e as designa ao nós *workers*
- ***Dispatcher*|Traduzir**: Controla os nós *workers*

Worker Nodes|Nós Trabalhadores: recebem as tarefas a serem executadas diretamente do nó *manager* e depois as executam. Também encaminham o *status* das atuais tarefas em execução no respectivo nó e também seu próprio *status*, assim

o manager recebe essas informações e consegue decidir para quem encaminhar as próximas tarefas a serem executadas.

2.7 NFS

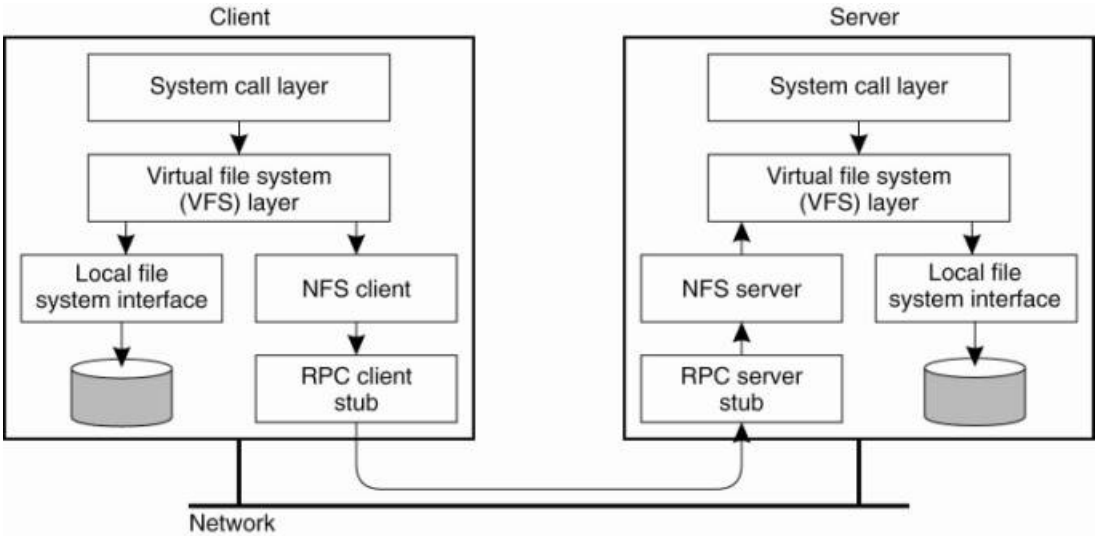
O NFS é um sistema de arquivos distribuído baseado em *Remote Procedure Call* (RPC), conforme a Figura 7, e projetado de forma a ser independente de máquina, sistema operacional, arquitetura de rede e protocolo de transporte, o NFS funciona como um compartilhamento de arquivos permitindo que computadores em rede acessem e compartilhem arquivos de forma transparente como se estivessem armazenados localmente (PAWLOWSKI et al., 2000). Diante de um cenário distribuído, como um *cluster*, ter o armazenamento de dados em um serviço NFS possibilita o acesso de qualquer computador pertencente ao *cluster* independentemente do sistema operacional utilizado.

Centralização: Com NFS é possível centralizar o armazenamento em um único servidor, simplificando desde o gerenciamento dos dados até a política de backup.

Escalabilidade: Sua estrutura é projetada para ser escalável, lidando com um grande número de clientes e compartilhamentos simultâneos.

Compatibilidade: Ampla gama de sistemas operacionais suportados, permitindo comunicação e compartilhamento de dados entre diferentes sistemas e arquiteturas.

Figura 7 – Arquitetura Funcional do NFS



Fonte: Marchese (2006)

3 TRABALHOS RELACIONADOS

Como suporte para a realização do presente trabalho, foi realizada uma pesquisa em bases acadêmicas como *ACM digital library*[1], *IEEEXplore*[2] e ferramentas de busca consolidadas, como o *Google Scholar*[3] contendo os termos *"HPC'*,*"cluster"*,*"docker"* e *"containers"*. Desta pesquisa, foram selecionados os trabalhos apresentados na Tabela 1. Na seção a seguir, são comentados alguns dos resultados relevantes destas obras para o desenvolvimento deste trabalho.

Tabela 1 – Trabalhos selecionados

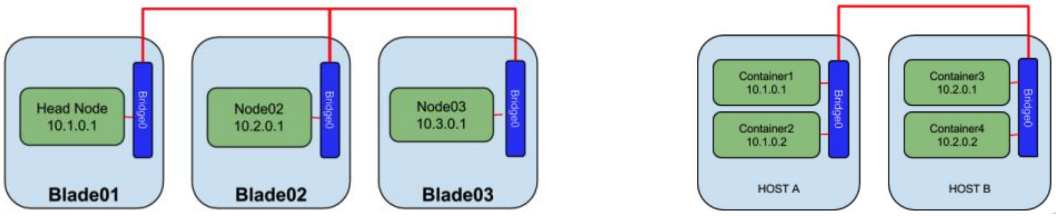
Título	**Autoria**
Containers in HPC: a survey	Tesser e Borin (2023)
Containers in HPC: Is it worth it?	Conde (2020)
Building a Virtual HPC Cluster with Auto Scaling by the Docker	Yu e Huang (2015)
Distributed MPI cluster with Docker Swarm mode	Nguyen e Bein (2017)

Fonte: o autor.

No trabalho de Tesser e Borin (2023), nos é apresentado uma visão geral sobre a utilização de contêineres em ambientes de HPC. Os autores reforçam a importância em montar ambientes com suporte a uma variedade de casos de uso e softwares utilizados por seus usuários sem sobrecarregar os usuários e os administradores do serviço. Também são apresentados pontos positivos como a facilidade na reprodutibilidade dos experimentos já que a utilização de contêineres de software possibilita criar um mesmo ambiente isolado diversas vezes, a ponto do processo de experimentação ser igual em todos os testes, e como um dos pontos negativos apresentados, temos os problemas relacionados ao paradigma de dados compartilhados entre processos das aplicações paralelas que vai de encontro ao isolamento que é proposto pela estrutura de contêineres. São apresentadas ferramentas próximas ao Docker, porém voltadas às particularidades dos ambientes de HPC como Shifter (GERHARDT et al., 2017), Singularity (KURTZER; SOCHAT; BAUER, 2017) e Charliecloud (PRIEDHORSKY; RANDLES, 2017).

Diante de um ambiente experimental simples, o trabalho de Yu e Huang (2015) é apresentada uma estrutura onde não é utilizada nenhuma ferramenta de orquestração de contêineres. Sua proposta inclui uma arquitetura composta de uma interface bridge0, como pode ser vista na Figura 8a e 8b, interface a qual é responsável pela comunicação entre os contêineres localizados em diferentes *hosts* sem a necessidade de *Network Address Translation* (NAT), já que a *bridge* padrão do sistema, a Docker0, é limitada a comunicação interna (máquina local). Através da adoção de contêineres de software como forma de montar um ambiente escalável, foi confirmada a viabilidade utilizando contêineres Docker.

[1] https://dl.acm.org/
[2] https://ieeexplore.ieee.org/
[3] https://scholar.google.com



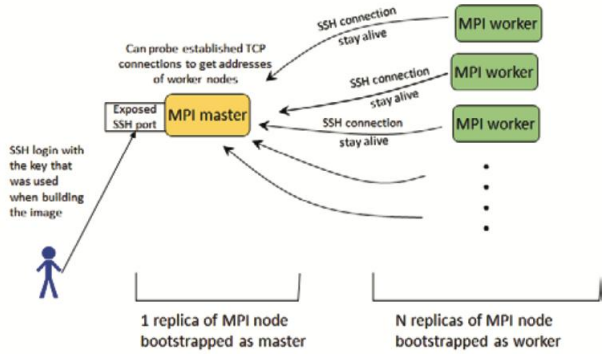
(a) Arquitetura do sistema (b) Comunicação entre contêineres em nós diferentes

Figura 8 – Yu e Huang (2015)

Como proposta de estudos futuros foram elencados questões performáticas relacionadas à interconexão entre contêineres e *hosts* e uma ferramenta de orquestração de contêineres como Docker Swarm.

No trabalho de Nguyen e Bein (2017) vemos a implementação de um *cluster* MPI através de contêineres utilizando *Docker Swarm* como orquestrador do ambiente. A implementação MPI utilizada foi a MPICH (BAYSER; CERQUEIRA, 2017) compilada em um contêiner com a imagem base do linux Alpine tendo cerca de 150 MB, tendo as principais ferramentas de compilação e a configuração necessária para o funcionamento do *Secure Socket Shell* (SSH) conforme Figura 9. Como conclusão os autores apresentam uma solução para a construção de um cluster MPI baseado na estrutura do Docker e utilizando o Swarm como orquestrador dos contêineres com o MPICH. Com a utilização do Swarm, eles também apontam que não é necessário um serviço de descoberta de serviços como o Consul (RITI; FLYNN, 2021) pois o próprio Swarm tem uma solução pronta para essa descoberta.

Figura 9 – Comunicação SSH entre nós *Master* e *Workers*



Fonte: Nguyen e Bein (2017)

4 ABORDAGEM E DESENVOLVIMENTO

Neste capítulo serão expostos os requisitos funcionais, não funcionais e a implementação do serviço.

4.1 REQUISITOS DO SERVIÇO

De forma a se iniciar um serviço de qualidade para a comunidade acadêmica, foi realizado um levantamento de requisitos funcionais e não-funcionais para o correto mapeamento de esforços e demandas do modelo de HPC como serviço proposto dentro do ambiente institucional da Universidade Federal de Santa Catarina (UFSC).

4.1.1 Requisitos Funcionais

Políticas: Diante da variedade de perfis de uso do serviço, percebeu-se a necessidade de políticas de priorização de uso. O serviço de HPC é amplamente utilizado em contextos acadêmicos, de pesquisa e projetos, dessa forma cada contexto deve ter uma prioridade no uso do serviço. Essas políticas de priorização devem estar ligadas aos grupos de uso em si e não diretamente aos usuários, e ainda deve-se poder inferir tal priorização à grupos de ativos e seus elementos diretamente.

Simplicidade: De forma a manter simples a manutenção e expansão dos *clusters* utilizados para o serviço de HPC, definiu-se alguns pontos com relação ao sistema operacional: (i) apenas pacotes básicos devem ser instalados e (ii) apenas a plataforma de execução escolhida deve ser instalada. Diante dessas definições, podemos garantir que o sistema será de fácil manutenção já que não será necessário conhecimento específicos em bibliotecas de HPC e nenhum outro software de monitoração ou escalonamento de filas. Desta forma, a inserção de novos *hosts* no *cluster* pode se dar de maneira simplificada.

Hardware: Com a crescente demanda por treinamento de redes neurais e a constante utilização de softwares que exigem comunicação de baixa latência entre processos, dois pré-requisitos essencialmente necessários à solução foram incorporados como a necessidade de suporte a processamento em GPUs e intercomunicação de baixa latência através de redes InfiniBand (BUYYA; CORTES; JIN, 2002) de até 40Gbps.

4.1.2 Requisitos Não Funcionais

Multi-Clusters: Devido a heterogeneidade da infraestrutura hoje existente, tanto no quesito de infraestrutura computacional quanto na geolocalização dos ativos, o suporte a agrupamento e divisão de recursos computacionais deve ser parte da solução. Dado a necessidade de submissão de *jobs* para grupos de ativos que contenham de-

terminadas especificações (e.g. Processadores especiais, intercomunicação de alto desempenho etc) e/ou também por requisitos não técnicos, como ativos do laboratório X, presentes no campus Y etc.

Interface Web: De forma a atender uma variedade de usuários, é preciso que a utilização do serviço seja a mais simples possível. Assim escolhemos por disponibilizar uma interface web acessível para a submissão do *jobs*, não sendo necessária a instalação de nenhum software por parte do usuário para a utilização do serviço. De modo a simplificar a submissão, o usuário deve entrar com o mínimo necessário de informações: (i) O *software* a ser executado, (ii) o comando e argumentos a serem utilizados, e (iii) o local (grupo de ativos) onde processar o *job*. A entrada e saída de dados também é feita por meio de uma interface web, onde é possível fazer o *upload* e *download* dos arquivos.

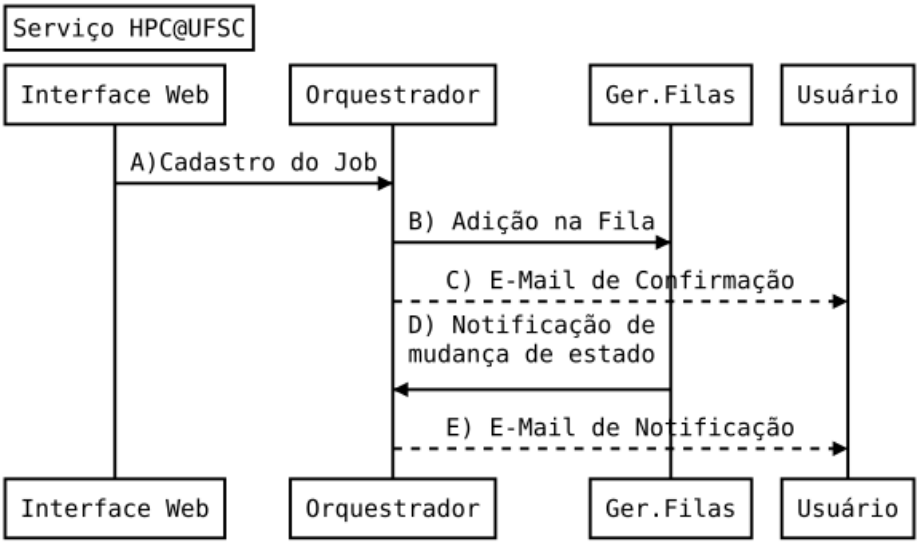
Ambiente: Considerando a pluralidade de *softwares* que serão executar no serviço, é de suma importância transferir a responsabilidade sobre o ambiente em que o *software* irá ser executado para o demandante. Como resultado dessa transferência, percebe-se dois grandes impactos: (i) desoneração da equipe de T.I. da responsabilidade de fazer a manutenção do ambiente no qual o *software* é executado (e.g. requerimentos de software, bibliotecas específicas e etc). (ii) usuários do serviço com um maior controle sob o ambiente utilizado, permitindo a solução dos ínfimos pré-requisitos necessários, como versões de bibliotecas e personalização do código fonte utilizado, a ponto de permitir a expansão do número de softwares suportados e oferecidos pelo serviço.

4.2 DESIGN PROPOSTO

Ao tentar prover Computação de Alto Desempenho como um serviço, agravam-se vários desafios dos já conhecidos. Resolver esses desafios a partir de métodos tradicionais de gerenciamento de *clusters* (e.g. OpenHPC, MaaS etc) ou de configuração (e.g. *Ansible*, *Chef* etc) se demonstrou inviável dada a flexibilidade exigida pelo cenário. De forma que após o levantamento de requisitos se fez necessário a montagem de um diagrama de interação (Figura 10) entre as camadas do serviço, a fim de que fosse possível identificar a melhor maneira de entregar o melhor fluxo para a solução dos desafios.

A Usuário entra com os dados básicos para submissão do *job*, como nome da imagem do contêiner e comando a ser executado;
B Orquestrador/Workflow adiciona o *job* na fila do Gerenciador de Filas;
C Usuário é notificado da entrada (ou falha) da submissão;
D Mudanças de status do *job* são registradas pelo Gerenciador de Filas (e.g. evolução na fila, ínicio, fim, falha, cancelamento etc);
E Orquestrador notifica o usuário sobre a evolução do seu *job*.

Figura 10 – Diagrama de interação.



Fonte: O autor.

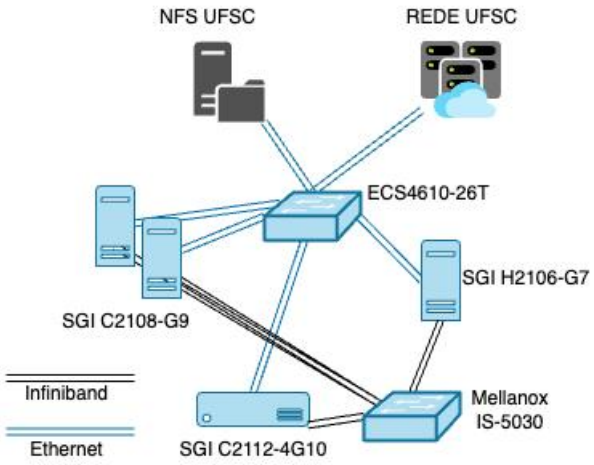
De forma a se obter um ambiente simples com o mínimo de *softwares* instalados e de forma a permitir o isolamento de ambientes e também a possibilidade do usuário do serviço poder personalizar seu espaço, código e bibliotecas utilizadas, os contêineres era a melhor plataforma base para se manter os requisitos básicos. Com isso, foi escolhido o Docker como *engine* de contêineres e o Swarm como o *software* orquestrador de contêineres a fim de se obter um certo nível de automação da estrutura.

4.3 HARDWARE UTILIZADO

O ambiente e o *hardware* utilizados para esse projeto, foram cedidos pela Superintendência de Governança Eletrônica e Tecnologia da Informação e Comunicação (SeTIC) da UFSC com a intenção de ser entregue o projeto como um auto-serviço para a comunidade acadêmica.

A Figura 11 representa o *cluster* Chamado de OCN, onde o mesmo é composto por 7 nós de computação interligados por uma rede *ethernet* de 1Gb/s de velocidade e uma rede *Infiniband* com interfaces que chegam até 40Gb/s. O *cluster* foi criado em uma plataforma de contêineres Docker (DOCKER, 2023b) orquestrados com Swarm (DOCKER, 2023a) e administrado por uma interface web chamada Portainer (PORTAINER, 2023). Como forma de manter centralizado os dados que irão ser utilizados pelo *cluster*, há um servidor NFS (PAWLOWSKI et al., 2000) da UFSC para atender a demanda compartilhada.

Figura 11 – *Cluster* HPC OCN



Fonte: O autor

4.3.1 Servidores

Esse projeto conta com 3 modelos de servidores distintos conforme a Tabela 2 apresenta, onde duas unidades são SGI C2108-G9, uma unidade SGI H2106-G7 e um chassi SGI C2112-4G10 que é composto por 4 unidades computacionais. Totalizando 7 servidores instalados com Ubuntu em sua versão 22.04, 208 cores de processamento e 654GB de memória ram DDR3.

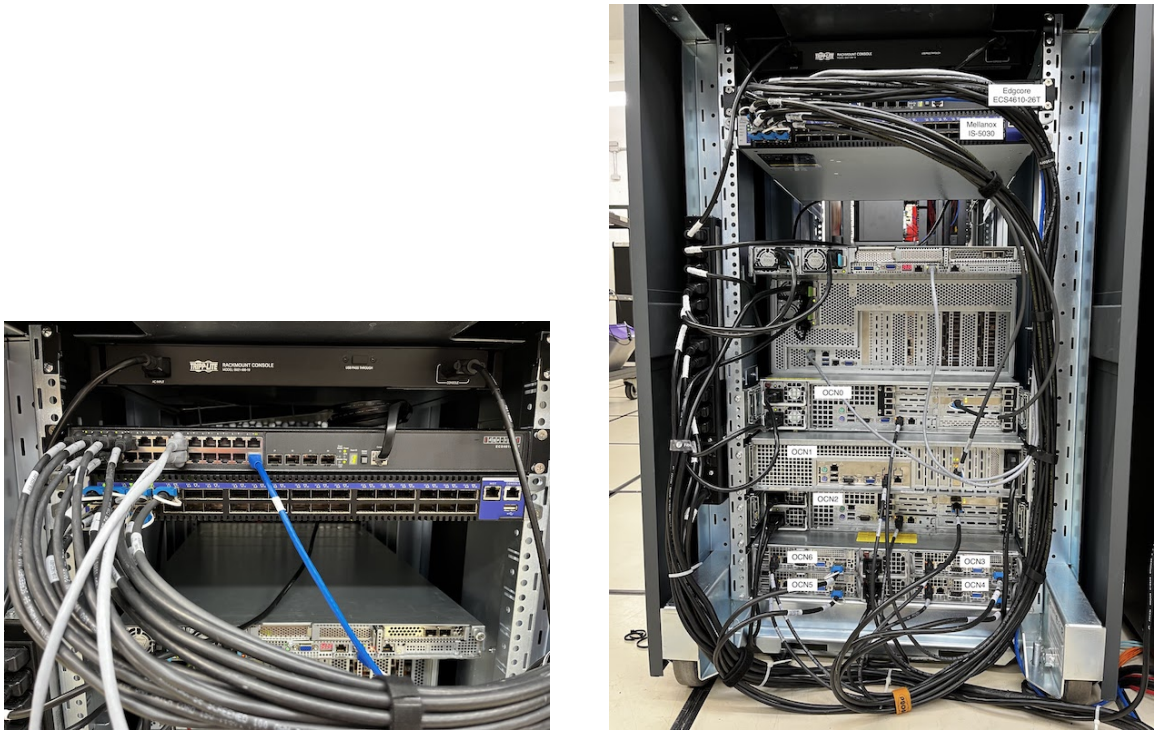
Tabela 2 – Hardware do Cluster OCN

Modelo	2x SGI C2108-G9	SGI H2106-G7	SGI C2112-4G10 4-bay
CPU	2x AMD Opteron 6344	2x AMD Opteron 6272	2xAMD Opteron 6376
Memória	64GB	270GB	64GB
Rede	2x1GbE	2x1GbE	2x1GbE
Infiniband	Mellanox MT25408A0	Mellanox MT25408A0	Mellanox MT25408A0

4.3.2 Rede

Existem duas redes distintas no *cluster*, sendo a primeira *ethernet* que é entregue por um cabo *uplink* na porta 24 do switch Edgecore ECS4610-26T que alimenta todos os nós do *cluster* com uma faixa pública de endereçamento. A segunda é uma rede privada apenas do *cluster*, onde os nós são interligados através de uma rede Infiniband concentrada no switch Mellanox IS-5030 com capacidade 40Gb/s por link. Nas Figuras 12a e 12b pode-se visualizar essas conexões dos switches com os *hosts*.

Figura 12 – Imagens do *Cluster* OCN



(a) Switches (b) Visão geral

Fonte: O autor

4.4 IMPLEMENTAÇÃO

Nos servidores do *cluster* foi instalado a versão 20.10.20 do Docker Engine, e configurado um cluster Swarm com 1 nó *Manager* e 6 nós *Workers*. Como os servidores tem duas redes configuradas (a ethernet e a infiniband), é importante informar ao Swarm qual rede ele deve utilizar para a comunicação entre os nós. Como forma de simplificar o *deploy* do serviço inicialmente, não foram utilizadas regras de boas práticas do Docker com relação ao número de *Managers* e *Workers* para o *cluster* ser tolerante à falhas. Na questão de comunicação entre os nós do cluster, foi inicializada a comunicação pela rede infiniband.

Com o intuito de manter simples a submissão de trabalhos para a fila, utilizamos um formulário feito a partir do *Google Forms* onde o usuário entra com as principais informações como nome da imagem do contêiner, o comando a ser executado e qual a partição do *cluster* deve executar o trabalho.

Foi desenvolvido um serviço orquestrador que faz a ponte entre o formulário de envio e a fila de trabalho do Slurm. O mesmo se chama *workflow* e é dividido em três contêineres e seus respectivos códigos disponíveis nos Anexos B, C e D.

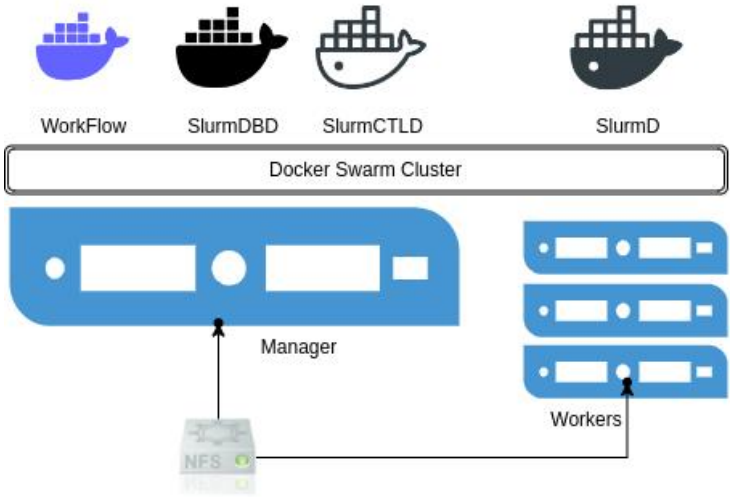
Read: sua função é ficar atento às submissões no formulário e inserir as informações de trabalhos submetidos por usuários autenticados e não autenticados nas respectivas

tabelas *new_jobs* e *rejected_jobs*.

Load: provê o serviço de busca de novos trabalhos na tabela *new_jobs* e submete para a fila do slurm do *cluster* solicitado. Ao ser submetido para a fila do slurm, o trabalho é inserido na tabela *current_jobs*.

Status: busca por trabalhos em produção na tabela *current_jobs* e verifica seu status e posição na fila, caso tenha tido alguma alteração encaminha um alerta ao usuário por e-mail. Caso encontre algum trabalho na tabela *rejected_jobs* ele notifica o usuário sobre o mesmo não ter sido submetido na fila do slurm.

Figura 13 – Distribuição de Contêineres no *Cluster*



Fonte: O autor.

Na Figura 13 é ilustrado a distribuição dos contêineres no *cluster*. Há 3 contêineres do Slurm com uma imagem[1] personalizada para funcionar os 3 serviços na mesma imagem docker, há também 3 contêineres que juntos formam o *workflow*[2].

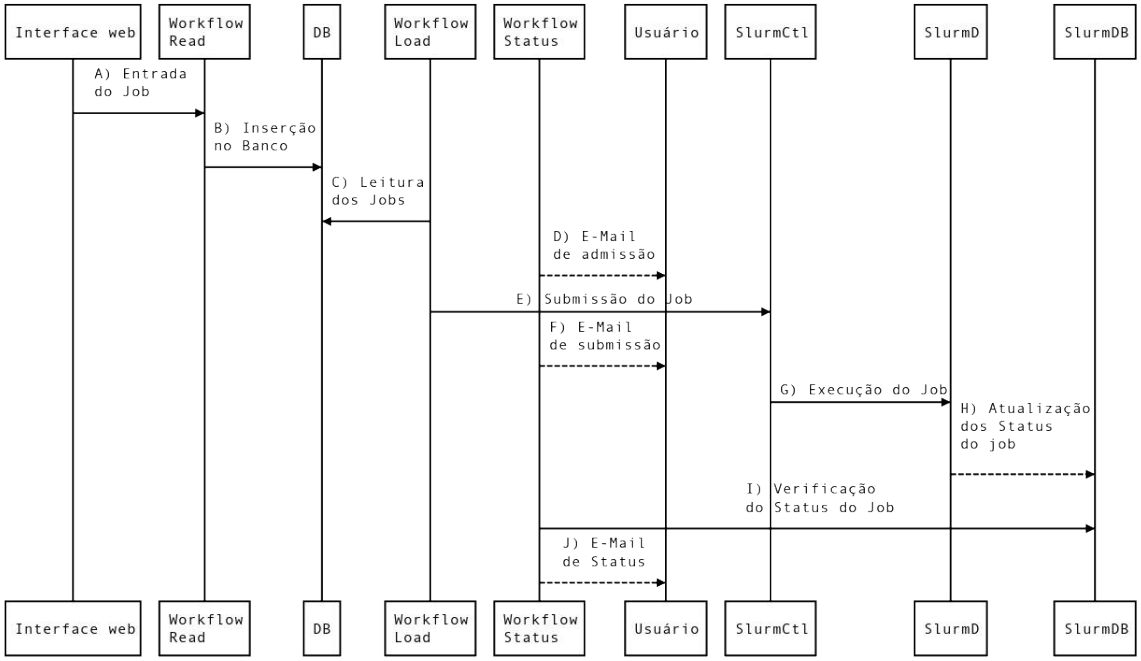
- **SlurmDBD:** instância do Daemon Slurm Database;
- **SlurmCTLD:** instância do Daemon Controlador do Slurm;
- **SlurmD:** instância do Daemon de execução do Slurm;
- **Read:** instância que busca trabalhos enviados no formulário;
- **Load:** instância que carrega os trabalhos para a fila do *cluster* ou rejeita-os; e
- **Status:** instância que informa o usuário sobre possíveis atualizações de status do trabalho.

Para um entendimento mais profundo do fluxo de trabalho, expandimos o diagrama de interação anterior de forma que os contêineres e seus serviços fossem melhor representados como na Figura 14.

[1] https://codigos.ufsc.br/setic-hpc/slurm
[2] https://codigos.ufsc.br/setic-hpc/workflow2-0

A Interface salva os dados básico do Job mais as credenciais de acesso, conta do grupo e partição de execução.
B Informações são capturadas do formulário e são inseridas no Banco de Dados.
C Jobs novos são puxados do Banco de Dados;
D Após a autenticação, validação e sanitização dos dados, o usuário é notificado da admissão do Job;
E Job é incluído no SLURM via linha de comando;
F Notificação de sucesso ou falha é enviada.
G Quando há recurso disponível, o Job é executado. Caso contrario fica aguardando na fila.
H O SlurmDB registra o status dos Jobs na fila e durante a execução.
I Mudanças dos status dos jobs são verificadas para acionar gatilhos.
J Notificação ao usuário sobre a evolução do seu Job é enviada (e.g. evolução na fila, inicio, fim, falha cancelamento etc);

Figura 14 – Diagrama de interação completa



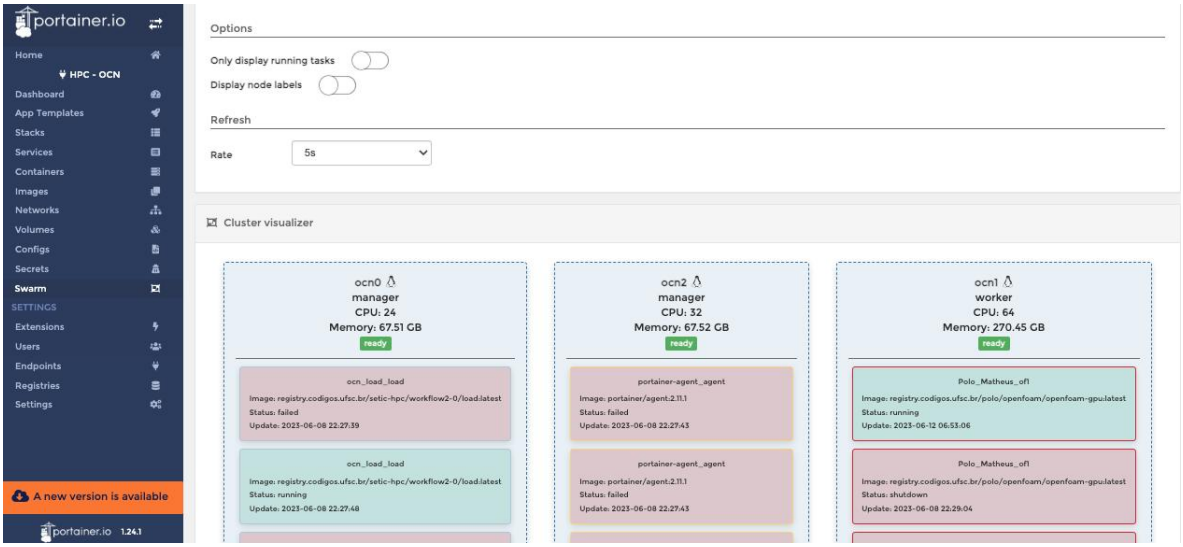
Fonte: O autor

De forma a termos um controle administrativos mais amigável do *cluster*, foi configurado uma instância do Portainer[3] que roda no nó *manager* e seu agente que roda em todos os nós do *cluster* de forma a permitir a coleta de dados do nó para a interface informando status e outras informações importantes para o bom funcionamento do *cluster*. Na Figura 15 pode ser visto o status do *cluster* com informações sobre o número de nós e onde cada contêiner está rodando.

Como forma de mostrar publicamente estatísticas e informações sobre a fila de trabalho do serviço de HPC, foi encontrada a ferramenta Metabase[4] que permite a criação de *dashboards* personalizáveis a partir de sintaxe *Structured Query Language* (SQL) sendo

[3] https://portainer.io
[4] https://www.metabase.com

Figura 15 – Interface Portainer - Visualização do *Cluster*



Fonte: O autor

possível a montagem de gráficos e etc. Diante dessa ferramenta, foi criado o *dashboard*[5] de acompanhamento conforme a Figura 16.

Figura 16 – *Dashboard* para a visualização de Jobs

Fonte: O autor

4.5 CENÁRIOS DE TESTES E RESULTADOS

O principal cenário de teste consistiu em avaliar a utilização da rede de baixa latência no quesito de ganho de performance para o *cluster*. Foi então utilizada a ferramenta IPERF (DINIZ; JUNIOR, 2014) que gera e faz a medição de tráfego tanto *Transmission*

[5] https://dashboards.setic.ufsc.br/public/dashboard/2eb0b36c-40f2-4b88-a031-19cdd62cc3f2

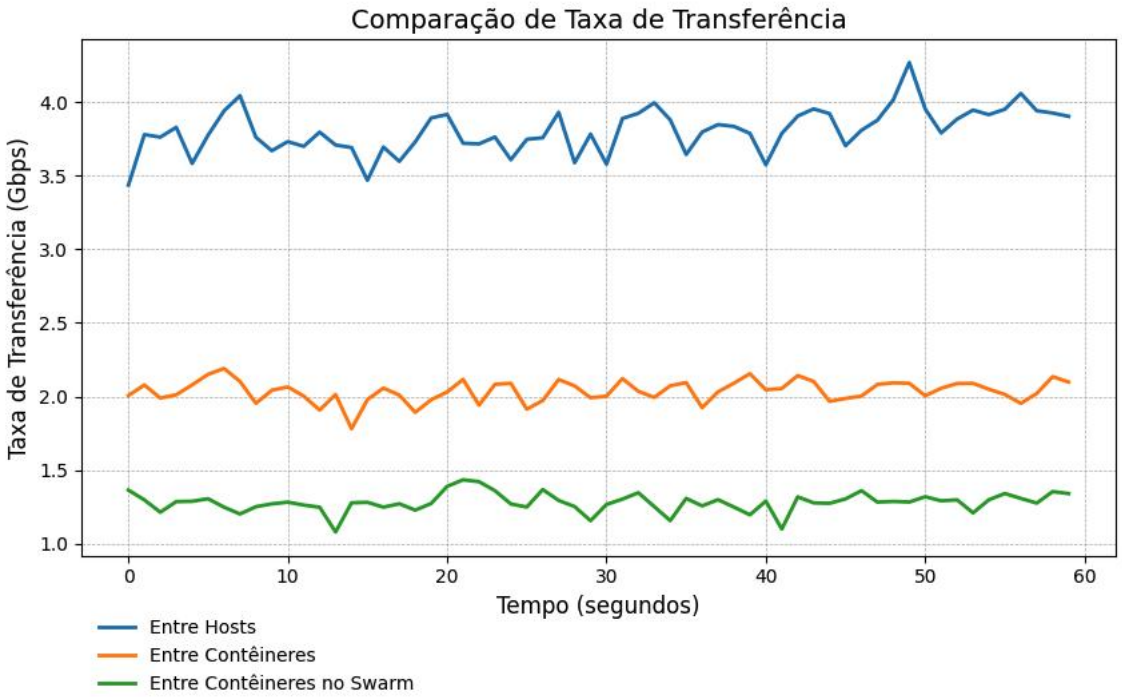
Control Protocol (TCP) quanto *User Datagram Protocol* (UDP) para se obter estatísticas de uso.

Foram considerados três cenários, onde:

- **Conexão entre hosts físicos:** intuito de representar um cluster HPC clássico, utilizando uma conexão de baixa latência como infiniband.
- **Conexão entre contêineres:** representando a comunicação entre dois contêineres em um mesmo *host* físico sem a camada de orquestração do swarm.
- **Conexão entre contêineres no Swarm:** o cenário implementado por este trabalho, onde existe a comunicação entre contêineres que se encontram em *hosts* diferentes e se comunicam por uma rede infiniband.

Por questões de limitação de hardware do cluster, não foi possível utilizar em sua totalidade os links infiniband de 40Gbps, por conta disso os links funcionaram apenas a 4Gbps de velocidade conforme configurações apontadas no Apêndice C. De forma que 4Gbps ainda é superior à velocidade das interfaces de rede dos servidores (1Gbps), prosseguimos com a utilização desse hardware para os testes do trabalho.

Figura 17 – Comparação de velocidade da rede infiniband



Fonte: O autor

Podemos verificar na Figura 17 as velocidades resultantes dos três cenários. Tais valores foram obtidos através do IPERF na sua versão 3 rodando nos hosts e também através de imagens[6] de contêineres criados a partir do *docker file* disponível no Anexo E

[6] https://hub.docker.com/r/networkstatic/iperf3/

e no repositório do *github*[7].

Conforme demonstrado, a velocidade da rede infiniband varia conforme o cenário, tendo o pior rendimento o cenário do trabalho. Esse baixo rendimento se dá pelas camadas utilizadas pelo *swarm* para que seja criada uma rede entre os contêineres em diferentes hosts físicos como se estivessem em uma única rede. Outro ponto também observado é a falta da disponibilização da interface infinband diretamente ao contêiner, evitando as camadas impostas pelo orquestrador.

[7] https://github.com/nerdalert/iperf3

5 CONSIDERAÇÕES FINAIS

A implementação de *clusters* de HPC com Docker é possível e sim, facilita o compartilhamento de uma infraestrutura com diversos tipos de usuários, porém ainda não é perfeita com a utilização de redes infiniband. De acordo com os resultados obtidos na fase de experimentação, temos uma problema à ser estudado mais profundamente para que possamos utilizar da rede de baixa latência de forma mais otimizada. Outras soluções como Singularity (KURTZER; SOCHAT; BAUER, 2017), já possibilitam essa melhor utilização de rede infiniband, porém não entrega a facilidade de utilização que o Docker ou até mesmo o Swarm como orquestrador pode entregar.

Como case interno da UFSC, já houve crescimento no uso por parte dos grupos de pesquisa. No período compreendido de setembro de 2022 à fevereiro de 2023 (6 meses) o autosserviço desenvolvido atendeu a pelo menos 6 grupos de pequisa, provendo mais de 23 softwares distintos, totalizando mais de 1400 horas de processamento e mais de 420 *jobs* submetidos mesmo com o cenário de baixa velocidade na rede infiniband, apontando que o projeto do serviço está na direção correta.

5.1 PRÓXIMOS PASSOS

De forma a seguir os estudos da utilização de redes infiniband com contêineres docker, é necessário uma busca mais profunda por uma forma de entregar a interface infiniband do host diretamente ao contêiner com a menor perda possível para que haja um real ganho na performance de comunicação entre contêineres em diferentes nós do *cluster*.

Outras possibilidades ainda são (i) a funcionalidade de pré-requisitos entre os trabalhos submetidos (e.g. fluxo de trabalho) e (ii) a evolução da interface web disponibilizada ao usuários para a submissão dos trabalhos.

REFERÊNCIAS

BAYSER, M. de; CERQUEIRA, R. Integrating mpi with docker for hpc. In: **2017 IEEE International Conference on Cloud Engineering (IC2E)**. [S.l.: s.n.], 2017. p. 259–265.

BURNS, B. et al. Borg, omega, and kubernetes. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 59, n. 5, p. 50–57, apr 2016. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/2890784.

BUYYA, R.; CORTES, T.; JIN, H. An introduction to the infiniband architecture. In: _____. **High Performance Mass Storage and Parallel I/O: Technologies and Applications**. [s.n.], 2002. p. 616–632. Disponível em: https://doi.org/10.1109/9780470544839.ch4.

CARISSIMI, A. Virtualização: da teoria a soluções. **Minicursos do Simpósio Brasileiro de Redes de Computadores–SBRC**, v. 2008, p. 173–207, 2008.

CHUNG, M. T. et al. Provision of docker and infiniband in high performance computing. In: **2016 International Conference on Advanced Computing and Applications (ACOMP)**. [S.l.: s.n.], 2016. p. 127–134.

CONDE, K. P. **Containers in HPC: Is it worth it?** Tese (Doutorado) — UPC, Facultat d'Informàtica de Barcelona, Departament d'Arquitectura de Computadors, Jun 2020. Disponível em: http://hdl.handle.net/2117/335295.

CORRÊA, J. N. G. F. et al. Lifter-disponibilização de aplicações via containers de software em um cluster de alto desempenho. Florianópolis, SC, 2016.

COULOURIS, G. F.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems: concepts and design**. [S.l.]: pearson education, 2005.

DANTAS, M. A. **Computação distribuída de alto desempenho: redes, clusters e grids computacionais**. [S.l.]: Axcel books, 2005.

DINIZ, P. H.; JUNIOR, N. A. Ferramenta iperf: geração e medição de tráfego tcp e udp. **Notas Técnicas**, v. 4, n. 2, 2014.

DOCKER. **Swarm mode overview**. 2023. https://docs.docker.com/engine/swarm/. Acessado: 01-04-2023.

DOCKER. **What Docker?** 2023. https://www.docker.com/what-docker/. Acessado: 01-04-2023.

EADLINE, D. **High performance computing for dummies**. [S.l.]: Wiley Publishing, Inc., 2009.

ERMAKOV, A.; VASYUKOV, A. Testing docker performance for HPC applications. **CoRR**, abs/1704.05592, 2017. Disponível em: http://arxiv.org/abs/1704.05592.

FERREIRA, U. J. S. F. et al. Análise de tecnologias de virtualização e hardware de baixo custo para infraestutura de nuvem de pequeno porte. Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, 2017.

FRAMPTON, M. Apache mesos. In: _____. **Complete Guide to Open Source Big Data Stack**. Berkeley, CA: Apress, 2018. p. 97–137. ISBN 978-1-4842-2149-5. Disponível em: https://doi.org/10.1007/978-1-4842-2149-5_4.

GERHARDT, L. et al. Shifter: Containers for hpc. **Journal of Physics: Conference Series**, IOP Publishing, v. 898, n. 8, p. 082021, oct 2017. Disponível em: https://dx.doi.org/10.1088/1742-6596/898/8/082021.

KURTZER, G. M.; SOCHAT, V.; BAUER, M. W. Singularity: Scientific containers for mobility of compute. **PLOS ONE**, Public Library of Science, v. 12, n. 5, p. 1–20, 05 2017. Disponível em: https://doi.org/10.1371/journal.pone.0177459.

LAUREANO, M. A. P.; MAZIERO, C. A. Virtualização: Conceitos e aplicações em segurança. **Sociedade Brasileira de Computação**, 2008.

LUKSA, M. **Kubernetes in action**. [S.l.]: Simon and Schuster, 2017.

MARCHESE, F. T. **Distributed File Systems**. 2006. http://csis.pace.edu/ marchese/CS865/Lectures/Chap11/Chapter11.htm. Acessado: 01-04-2023.

MORAVCIK, M.; KONTSEK, M. Overview of docker container orchestration tools. In: **2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)**. [S.l.: s.n.], 2020. p. 475–480.

NGUYEN, N.; BEIN, D. Distributed mpi cluster with docker swarm mode. In: **2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)**. [S.l.: s.n.], 2017. p. 1–7.

OLIVEIRA, R. S. d.; CARISSIMI, A. d. S.; TOSCANI, S. S. Sistemas operacionais. **Revista de informática teórica e aplicada. Porto Alegre. Vol. 8, n. 3 (dez. 2001), p. 7-39**, 2001.

PAWLOWSKI, B. et al. The nfs version 4 protocol. In: CITESEER. **In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000**. [S.l.], 2000.

PORTAINER. **Container Management**. 2023. https://portainer.io. Acessado: 16-03-2023.

PRIEDHORSKY, R.; RANDLES, T. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: Association for Computing Machinery, 2017. (SC '17). ISBN 9781450351140. Disponível em: https://doi.org/10.1145/3126908.3126925.

QNAP. **O que é Virtualização?** 2023. https://www.qnapbrasil.com.br/blog/post/o-que-e-virtualizacao. Acessado: 01-02-2023.

RITI, P.; FLYNN, D. Consul hcl. In: _____. **Beginning HCL Programming: Using Hashicorp Language for Automation and Configuration**. Berkeley, CA: Apress, 2021. p. 107–127. ISBN 978-1-4842-6634-2. Disponível em: https://doi.org/10.1007/978-1-4842-6634-2_6.

TANENBAUM, A. S.; STEEN, M. v. **Distributed Systems: Principles and Paradigms (2nd Edition)**. USA: Prentice-Hall, Inc., 2006. ISBN 0132392275.

TESSER, R. K.; BORIN, E. Containers in hpc: a survey. **The Journal of Supercomputing**, v. 79, n. 5, p. 5759–5827, Mar 2023. ISSN 1573-0484. Disponível em: https://doi.org/10.1007/s11227-022-04848-y.

TITTEL, E. **Clusters For Dummies**. [S.l.]: John Wiley & Sons, 2012.

YOO, A. B.; JETTE, M. A.; GRONDONA, M. Slurm: Simple linux utility for resource management. In: FEITELSON, D.; RUDOLPH, L.; SCHWIEGELSHOHN, U. (Ed.). **Job Scheduling Strategies for Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 44–60. ISBN 978-3-540-39727-4.

YU, H.; HUANG, W. Building a virtual HPC cluster with auto scaling by the docker. **CoRR**, abs/1509.08231, 2015. Disponível em: http://arxiv.org/abs/1509.08231.

ZHOU, N. et al. Container orchestration on hpc systems through kubernetes. **Journal of Cloud Computing**, v. 10, n. 1, p. 16, Feb 2021. ISSN 2192-113X. Disponível em: https://doi.org/10.1186/s13677-021-00231-z.

Apêndices

APÊNDICE A – ARTIGO DO TCC

Implementação de um Cluster para Aplicações de HPC utilizando Docker e Infiniband

Weslley da C. Silva [1]

[1]Superintendência de Governança Eletrônica e T.I. e Comunicação – SeTIC
Universidade Federal de Santa Catarina – UFSC
Florianópolis – SC – Brasil

weslley.silva@ufsc.br

Resumo. *O custos elevados para a aquisição e manutenção de infraestrutura para HPC tem sido alvo de preocupação por parte dos pesquisadores, de forma a buscarem alternativas como a de clusters compartilhados entre grupos de pesquisa, utilização de virtualização e o uso de contêineres. Desta forma, diante da criação de novos desafios enfrentados pelas equipes, este trabalho apresenta uma proposta de um cluster HPC com o uso de contêineres Docker.*

1. Introdução

A *High Performance Computing* (HPC) é uma área crucial para aplicações científicas e de engenharia, em que o poder computacional é fundamental para o sucesso dos projetos. Embora existam serviços de HPC em nuvem disponíveis para o público em geral, muitos grupos de pesquisa acabam por preferir investir em infraestrutura própria para garantir uma maior privacidade e controle sobre os recursos computacionais disponíveis.

No entanto, a aquisição e manutenção de uma infraestrutura própria pode ser um grande desafio, tanto do lado financeiro quanto do lado de recursos humanos para gerenciar e manter toda a estrutura necessária. Alternativamente, *clusters* compartilhados entre grupos de pesquisas podem reduzir o alto custo de aquisição de hardware e a mão de obra para o gerenciamento dos sistemas, porém cria novos problemas dado o crescimento do seu uso como: compartilhamento de recursos, complexidade dos ambientes, problemas de usabilidade, indisponibilidade de recursos, conflitos de dependências e aplicações com diferentes requisitos.

Com a modernização de tecnologias como a de contêineres de software, tornou-se possível flexibilizar a execução de diversas aplicações em único nó, cada uma em um contêiner isolado. Como consequência dessa modernização, surgiram estudos como [Yu and Huang 2015], [Ermakov and Vasyukov 2017] e [Zhou et al. 2021] que tratam da utilização de contêineres em ambientes de HPC tornando-se possível manter e gerenciar diversos ambientes em uma mesma infraestrutura com isolamento de recursos e sem os conflitos de dependências que marcam os ambientes compartilhados.

Devidos os estudos citados anteriormente, foi percebido que é possível sanar diversas demandas de HPC da universidade através de uma solução de *cluster* compartilhado entre os grupos de pesquisa com a utilização de contêineres. Portanto esse trabalho propõem a construção de um *cluster* HPC por meio de contêineres *Docker* [Docker b], interconexão com *Infiniband* [Buyya et al. 2002] e o software SLURM [Yoo et al. 2003] para gerenciamento de filas de processamento.

2. Abordagem Adotada

Tendo sido feito o levantamento dos requisitos funcionais e não-funcionais da solução, foi idealizado um serviço simples e funcional. A Figura 1 apresenta a interação entre usuário e serviço, conforme os passos descritos a seguir.

A Usuário entra com os dados básicos para submissão do *job*, como nome da imagem do contêiner e comando a ser executado;
B Orquestrador/ adiciona o *job* na fila do Gerenciador de Filas;
C Usuário é notificado da entrada (ou falha) da submissão;
D Mudanças de status do *job* são registradas pelo Gerenciador de Filas (e.g. evolução na fila, ínicio, fim, falha, cancelamento etc);
E Orquestrador notifica o usuário sobre a evolução do seu *job*.

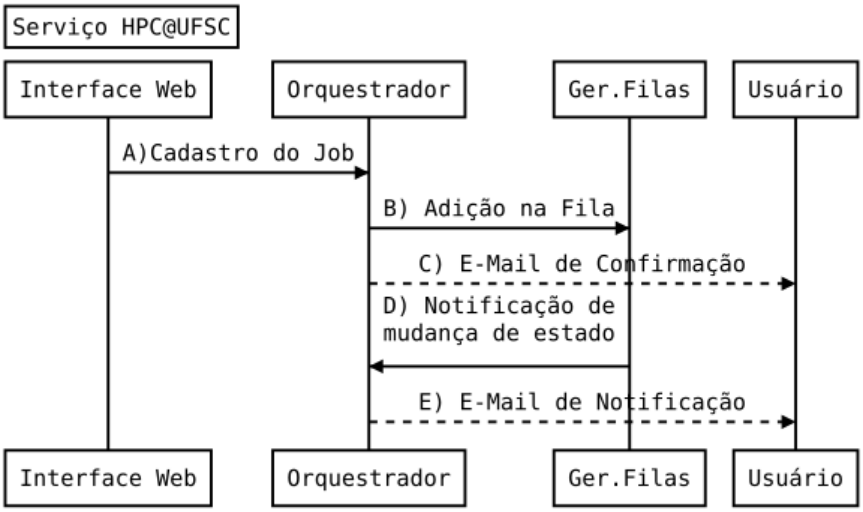


Figura 1. Diagrama de interação.

Para a redução de complexidade do projeto foi escolhido o orquestrador Docker Swarm [Docker a], já que o mesmo é uma ferramenta nativa da plataforma Docker para gerenciamento de *clusters* de contêineres em larga escala.

Tabela 1. Hardware do Cluster OCN

Modelo	2x SGI C2108-G9	SGI H2106-G7	SGI C2112-4G10 4-bay
CPU	2x AMD Opteron 6344	2x AMD Opteron 6272	2xAMD Opteron 6376
Memória	64GB	270GB	64GB
Rede	2x1GbE	2x1GbE	2x1GbE
Infiniband	Mellanox MT25408A0	Mellanox MT25408A0	Mellanox MT25408A0

Como hardware utilizado inicialmente no projeto, o *cluster* OCN compreende 4 servidores apresentados na Tabela 1, sendo que o modelo SGI C2112-4G10 tem 4 baias, ou seja, o mesmo comporta 4 nós com a configura descrita na Tabela 1. Ao todo soma-se 7 nodos, 248 cores de processamento e aproximadamente 675 GB de memória *RAM*. Para a interconexão entre os nós e *uplink*, é utilizado um *switch* Edgecore ECS4610-26T de 26

portas e para a interconexão entre os nodos do *cluster* através de *Infiniband* um Mellanox IS-5030.

Na camada de softwares, está sendo utilizando como S.O. o Ubuntu 20.04, o *docker engine* na versão v20.10.20 e o Portainer [Portainer] para gerenciamento das stacks.

3. Implementação

Inicialmente para manter simples a submissão de *jobs* para os usuários, foi decidido pela utilização de um *Google Forms* para o envio das tarefas a serem executadas no *cluster*. Os dados são processados por um *cluster Docker Swarm* que contém ao menos um *manager* e N *workers*.

No *cluster* há 4 tipos de contêineres: (i) uma instância do gerenciador do fluxo de trabalho (*workflow*/orquestrador), (ii) uma instância do *SlurmDBD*, (iii) uma instância do *SlurmCTLD* e (iv) N instâncias do *SlurmD*. De forma a simplificar a solução, foram concentrados em uma única imagem os serviços do SLURM (*SlurmDBD, SlurmCTLD e SlurmD*).

Para a orquestração dos serviços da solução, foi criado um arquivo *docker-compose.yml* que descreve a estrutura é o mesmo pode ser encontrado em: [SeTIC b] e [SeTIC a]. Também de forma a prover uma visualização da fila de processamento, foi desenvolvido utilizando a ferramenta Metabase [Metabase] um *dashboard* [HPC@UFSC] (Figura 2) com os dados do Banco do SLURM.

Figura 2. Dashboard para a visualização de Jobs

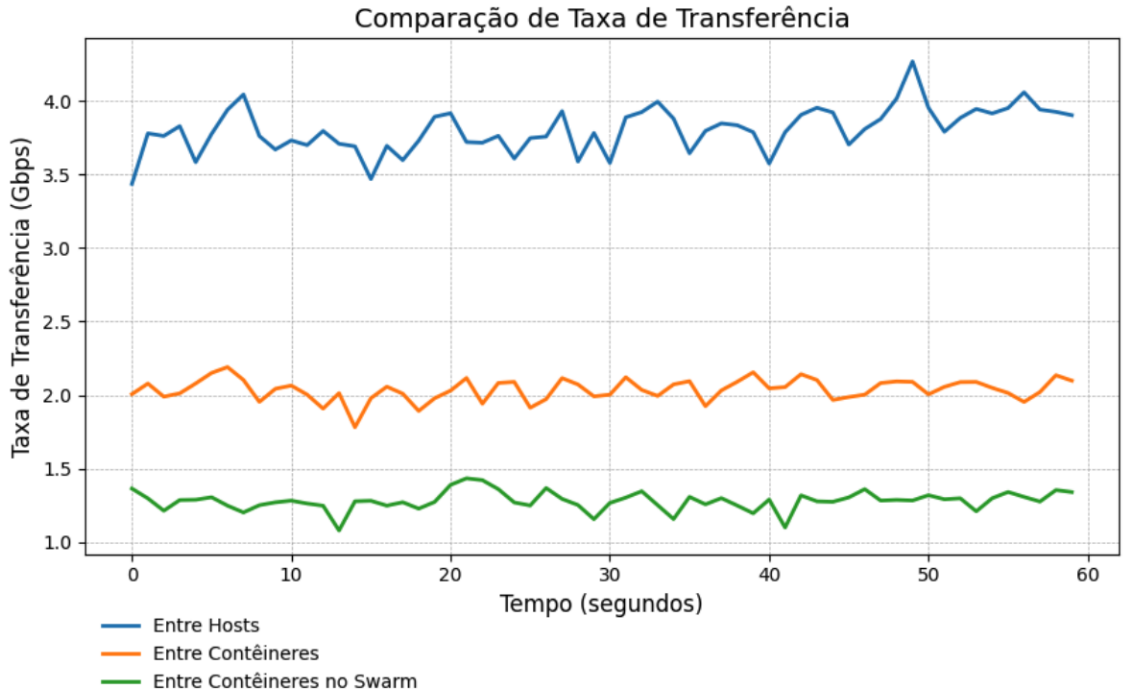
O principal cenário de teste consistiu em avaliar a utilização da rede de baixa latência no quesito de ganho de performance para o *cluster*. Foi então utilizada a ferramenta IPERF [Diniz and Junior 2014] que gera e faz a medição de tráfego tanto quanto para se obter estatísticas de uso.

Foram considerados três cenários, onde:

- **Conexão entre hosts físicos:** intuito de representar um cluster clássico, utilizando uma conexão de baixa latência como infiniband.
- **Conexão entre contêineres:** representando a comunicação entre dois contêineres em um mesmo *host* físico sem a camada de orquestração do swarm.
- **Conexão entre contêineres no Swarm:** o cenário implementado por este trabalho, onde existe a comunicação entre contêineres que se encontram em *hosts* diferentes e se comunicam por uma rede infiniband.

Por questões de limitação de hardware do cluster, não foi possível utilizar em sua totalidade os links infiniband de 40Gbps, por conta disso os links funcionaram apenas a 4Gbps de velocidade. De forma que 4Gbps ainda é superior à velocidade das interfaces de rede dos servidores (1Gbps), prosseguimos com a utilização desse hardware para os testes do trabalho.

Figura 3. Comparação de velocidade da rede infiniband



Podemos verificar na Figura 3 as velocidades resultantes dos três cenários. Tais valores foram obtidos através do IPERF na sua versão 3 rodando nos hosts e também através de imagens[1] de contêineres criados a partir do *docker file* disponível no repositório do *github*[2].

Conforme demonstrado, a velocidade da rede infiniband varia conforme o cenário, tendo o pior rendimento o cenário do trabalho. Esse baixo rendimento se dá pelas camadas utilizadas pelo *swarm* para que seja criada uma rede entre os contêineres em diferentes hosts físicos como se estivessem em uma única rede. Outro ponto também observado é

[1]https://hub.docker.com/r/networkstatic/iperf3/
[2]https://github.com/nerdalert/iperf3

a falta da disponibilização da interface infinband diretamente ao contêiner, evitando as camadas impostas pelo orquestrador.

4. Considerações Finais e Próximos passos

A implementação de *clusters* de com Docker é possível e sim, facilita o compartilhamento de uma infraestrutura com diversos tipos de usuários, porém ainda não é perfeita com a utilização de redes infiniband. De acordo com os resultados obtidos na fase de experimentação, temos uma problema à ser estudado mais profundamente para que possamos utilizar da rede de baixa latência de forma mais otimizada. Outras soluções como Singularity [Kurtzer et al. 2017], já possibilitam essa melhor utilização de rede infiniband, porém não entrega a facilidade de utilização que o Docker ou até mesmo o Swarm como orquestrador pode entregar.

Como case interno da UFSC, já houve crescimento no uso por parte dos grupos de pesquisa. No período compreendido de setembro de 2022 à fevereiro de 2023 (6 meses) o autosserviço desenvolvido atendeu a pelo menos 6 grupos de pequisa, provendo mais de 23 softwares distintos, totalizando mais de 1400 horas de processamento e mais de 420 *jobs* submetidos mesmo com o cenário de baixa velocidade na rede infiniband, apontando que o projeto do serviço está na direção correta.

De forma a seguir os estudos da utilização de redes infiniband com contêineres docker, é necessário uma busca mais profunda por uma forma de entregar a interface infiniband do host diretamente ao contêiner com a menor perda possível para que haja um real ganho na performance de comunicação entre contêineres em diferentes nós do *cluster*.

Agradecimentos

Agradecemos à Superintendência de Governança Eletrônica e Tecnologia da Informação e Comunicação (SeTIC) – UFSC, por proporcionar a infraestrutura necessária para que esse projeto pudesse ser realizado, e aos laboratórios que diariamente utilizam e testam o ambiente montado.

Referências

Buyya, R., Cortes, T., and Jin, H. (2002). *An Introduction to the InfiniBand Architecture*, pages 616–632.

Diniz, P. H. and Junior, N. A. (2014). Ferramenta iperf: geração e medição de tráfego tcp e udp. *Notas Técnicas*, 4(2).

Docker. Swarm mode overview. https://docs.docker.com/engine/swarm/. Acessado: 16-03-2023.

Docker. What docker? https://www.docker.com/what-docker/. Acessado: 16-03-2023.

Ermakov, A. and Vasyukov, A. (2017). Testing docker performance for HPC applications. *CoRR*, abs/1704.05592.

HPC@UFSC. Dashboard de filas. https://dashboards.setic.ufsc.br/public/dashboard/2eb0b36c-40f2-4b88-a031-19cdd62cc3f2. Acessado: 16-03-2023.

Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20.

Metabase. Projeto. https://www.metabase.com/. Acessado: 16-03-2023.

Portainer. Container management. https://portainer.io. Acessado: 16-03-2023.

SeTIC, D. Imagem slurm. https://codigos.ufsc.br/setic-hpc/slurm. Acessado: 16-03-2023.

SeTIC, D. Slurm workflow. https://codigos.ufsc.br/setic-hpc/workflow. Acessado: 16-03-2023.

Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In Feitelson, D., Rudolph, L., and Schwiegelshohn, U., editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg. Springer Berlin Heidelberg.

Yu, H. and Huang, W. (2015). Building a virtual HPC cluster with auto scaling by the docker. *CoRR*, abs/1509.08231.

Zhou, N., Georgiou, Y., Pospieszny, M., Zhong, L., Zhou, H., Niethammer, C., Pejak, B., Marko, O., and Hoppe, D. (2021). Container orchestration on hpc systems through kubernetes. *Journal of Cloud Computing*, 10(1):16.

APÊNDICE B – GERADOR DE GRÁFICO PARA SAÍDA DO IPERF

```python
import json
import matplotlib.pyplot as plt

def gerar_grafico_arquivos(arquivos,legendas,nome_imagem):

    fig, ax = plt.subplots(figsize=(10, 6))  # Define o tamanho da figura

    for i, arquivo in enumerate(arquivos):
        tempos = []
        taxas = []

        with open(arquivo, 'r') as arquivo_json:
            dados = json.load(arquivo_json)

        for item in dados['intervals']:
            tempos.append(item['sum']['start'])
            taxas.append(item['sum']['bits_per_second'] / 1e9)  # Converter para Gbps

        ax.plot(tempos, taxas, label=legendas[i], linewidth=2)

    ax.set_xlabel('Tempo (segundos)', fontsize=12)
    ax.set_ylabel('Taxa de Transferência (Gbps)', fontsize=12)
    ax.set_title('Comparação de Taxa de Transferência', fontsize=14)
    ax.grid(True, linestyle='--', linewidth=0.5)
    ax.legend(fontsize=10, frameon=False, bbox_to_anchor=(0, -0.5), loc='lower left') #
    ↪  Removendo a caixa da legenda, Movendo a caixa de legendas para o canto
    ↪  inferior esquerdo, abaixo do gráfico

    # Personalizando os ticks nos eixos x e y
    ax.tick_params(axis='x', which='both', bottom=True, top=False, labelbottom=True)
    ax.tick_params(axis='y', which='both', left=True, right=False, labelleft=True)

    plt.savefig(nome_imagem, dpi=300)  # Salva a imagem com o nome desejado e
    ↪  dpi definido

    plt.show()

# Exemplo de uso:
arquivos = ['iperf-out/ocn1.json', 'iperf-out/ocn1_docker_to_docker.json',
↪  'iperf-out/ocn1_swarm_to_swarm.json']
legendas = ['Entre Hosts','Entre Contêineres','Entre Contêineres no Swarm']

nome_imagem = 'grafico_comparacao.png'

gerar_grafico_arquivos(arquivos,legendas,nome_imagem)
```

APÊNDICE C – INFORMAÇÕES DE CONFIGURAÇÃO DO INFINIBAND

```
apt install ibverbs-utils infiniband-diags -y
```

```
root@ocn0:~\$ ibv_devices
    device                 node GUID
    ------              ----------------
    ibp5s0              0002c903002cd760
```

```
root@ocn0:~\$ ibv_devinfo ibp5s0

hca_id:     ibp5s0
        transport:                  InfiniBand (0)
        fw_ver:                     2.9.1000
        node_guid:                  0002:c903:002c:d760
        sys_image_guid:                 0002:c903:002c:d763
        vendor_id:                  0x02c9
        vendor_part_id:                 26428
        hw_ver:                     0xB0
        board_id:                   MT_0D90110009
        phys_port_cnt:                  1
                port:   1
                        state:              PORT_ACTIVE (4)
                        max_mtu:            4096 (5)
                        active_mtu:             4096 (5)
                        sm_lid:                 1
                        port_lid:           2
                        port_lmc:           0x00
                        link_layer:             InfiniBand

```

```
root@ocn0:~\$ ibstat ibp5s0
CA 'ibp5s0'
        CA type: MT26428
        Number of ports: 1
        Firmware version: 2.9.1000
        Hardware version: b0
        Node GUID: 0x0002c903002cd760
        System image GUID: 0x0002c903002cd763
        Port 1:
                State: Active
                Physical state: LinkUp
```

```
12                Rate: 40
13                Base lid: 2
14                LMC: 0
15                SM lid: 1
16                Capability mask: 0x02510868
17                Port GUID: 0x0002c903002cd761
18                Link layer: InfiniBand
```

Anexos

ANEXO A – DOCKERFILE SLURM

```
FROM centos:7

LABEL
↪  org.opencontainers.image.source="https://github.com/giovtorres/slurm-docker-cluster"
↪  \
    org.opencontainers.image.title="slurm-docker-cluster" \
    org.opencontainers.image.description="Slurm Docker cluster on CentOS 7" \
    org.label-schema.docker.cmd="docker-compose up -d" \
    maintainer="Giovanni Torres"

ARG SLURM_TAG=slurm-19-05-1-2
ARG GOSU_VERSION=1.11

RUN set -ex \
   && yum makecache fast \
   && yum -y update \
   && yum -y install epel-release \
   && yum -y install \
      wget \
      bzip2 \
      perl \
      gcc \
      gcc-c++\
      git \
      gnupg \
      make \
      munge \
      munge-devel \
      python-devel \
      python-pip \
      python34 \
      python34-devel \
      python34-pip \
      mariadb-server \
      mariadb-devel \
      psmisc \
      bash-completion \
      vim-enhanced \
      openssh-server \
   && yum clean all \
   && rm -rf /var/cache/yum

RUN ln -s /usr/bin/python3.4 /usr/bin/python3

RUN pip install Cython nose && pip3.4 install Cython nose

```

```
45  RUN set -ex \
46      && wget -O /usr/local/bin/gosu
        ↪  "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-
        ↪  amd64"
        ↪  \
47      && wget -O /usr/local/bin/gosu.asc
        ↪  "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-
        ↪  amd64.asc"
        ↪  \
48      && export GNUPGHOME="$(mktemp -d)" \
49      && gpg --batch --keyserver hkps://keys.openpgp.org --recv-keys
        ↪  B42F6819007F00F88E364FD4036A9C25BF357DD4 \
50      && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
51      && rm -rf "${GNUPGHOME}" /usr/local/bin/gosu.asc \
52      && chmod +x /usr/local/bin/gosu \
53      && gosu nobody true
54
55  RUN set -x \
56      && git clone https://github.com/SchedMD/slurm.git \
57      && pushd slurm \
58      && git checkout tags/$SLURM_TAG \
59      && ./configure --enable-debug --prefix=/usr --sysconfdir=/etc/slurm \
60         --with-mysql_config=/usr/bin  --libdir=/usr/lib64 \
61      && make install \
62      && install -D -m644 etc/cgroup.conf.example /etc/slurm/cgroup.conf.example \
63      && install -D -m644 etc/slurm.conf.example /etc/slurm/slurm.conf.example \
64      && install -D -m644 etc/slurmdbd.conf.example /etc/slurm/slurmdbd.conf.example \
65      && install -D -m644 contribs/slurm_completion_help/slurm_completion.sh
        ↪  /etc/profile.d/slurm_completion.sh \
66      && popd \
67      && rm -rf slurm \
68      && groupadd -r --gid=995 slurm \
69      && useradd -r -g slurm --uid=995 slurm \
70      && mkdir /etc/sysconfig/slurm \
71         /var/spool/slurmd \
72         /var/run/slurmd \
73         /var/run/slurmdbd \
74         /var/lib/slurmd \
75         /var/log/slurm \
76         /data \
77      && touch /var/lib/slurmd/node_state \
78         /var/lib/slurmd/front_end_state \
79         /var/lib/slurmd/job_state \
80         /var/lib/slurmd/resv_state \
81         /var/lib/slurmd/trigger_state \
82         /var/lib/slurmd/assoc_mgr_state \
83         /var/lib/slurmd/assoc_usage \
84         /var/lib/slurmd/qos_usage \
85         /var/lib/slurmd/fed_mgr_state \
```

```
86      && chown -R slurm:slurm /var/*/slurm* \
87      && /sbin/create-munge-key
88
89  #install docker
90  RUN curl -fsSL https://get.docker.com -o get-docker.sh && \
91      sh get-docker.sh
92
93  #set timezone
94  RUN cp /etc/localtime /root/old.timezone \
95      && rm /etc/localtime \
96      && ln -s /usr/share/zoneinfo/America/Sao_Paulo /etc/localtime
97
98  #config ssh
99  RUN ssh-keygen -t rsa -f /root/.ssh/id_rsa && \
100     cat /root/.ssh/id_rsa.pub >> /root/.ssh/authorized_keys && \
101     echo 'StrictHostKeyChecking no' >> /root/.ssh/config;
102
103 RUN (cd /lib/systemd/system/sysinit.target.wants/; for i in *; do [ $i == \
104     systemd-tmpfiles-setup.service ] || rm -f $i; done); \
105     rm -f /lib/systemd/system/multi-user.target.wants/*;\
106     rm -f /etc/systemd/system/*.wants/*;\
107     rm -f /lib/systemd/system/local-fs.target.wants/*; \
108     rm -f /lib/systemd/system/sockets.target.wants/*udev*; \
109     rm -f /lib/systemd/system/sockets.target.wants/*initctl*; \
110     rm -f /lib/systemd/system/basic.target.wants/*;\
111     rm -f /lib/systemd/system/anaconda.target.wants/*;
112
113 COPY ./confs/slurm.conf /etc/slurm/slurm.conf
114 COPY ./confs/slurmdbd.conf /etc/slurm/slurmdbd.conf
115 COPY docker-entrypoint.sh /usr/local/bin/docker-entrypoint.sh
116
117 ENTRYPOINT ["/usr/local/bin/docker-entrypoint.sh"]
118
119 VOLUME ["/scripts"]
120 CMD ["slurmdbd"]
121
122
```

ANEXO B – DOCKERFILE WORKFLOW READ

```
1  FROM python:3.9.15
2  ENV TZ=America/Sao_Paulo
3  USER root
4  RUN  apt update && \
5      apt install tzdata -y && \
6      apt install python3-pip --yes && \
7      pip3 install gspread && \
8      pip3 install --upgrade google-api-python-client oauth2client && \
9      pip3 install mysql-connector-python
10
11 COPY app /home/app
12 WORKDIR /home/app
13 ENTRYPOINT ["python3"]
14 CMD ["main.py"]
15
```

ANEXO C – DOCKERFILE WORKFLOW LOAD

```
1  FROM python:3.9.15
2  ENV TZ=America/Sao_Paulo
3  USER root
4  RUN  apt update && \
5      apt install tzdata -y && \
6      apt install python3-pip --yes && \
7      apt install curl --yes && \
8      pip3 install mysql-connector-python && \
9      curl -fsSL https://get.docker.com -o get-docker.sh && \
10     sh get-docker.sh
11 COPY app /home/app
12 WORKDIR /home/app
13 CMD ["main.py"]
14 ENTRYPOINT ["python3"]
15
```

ANEXO D – DOCKERFILE WORKFLOW STATUS

```
1  FROM python:3.9.15
2  ENV TZ=America/Sao_Paulo
3  USER root
4  RUN  apt update && \
5      apt install tzdata -y && \
6      apt install python3-pip --yes && \
7      pip3 install mysql-connector-python
8
9  COPY app /home/app
10 WORKDIR /home/app
11 CMD ["main.py"]
12 ENTRYPOINT ["python3"]
13
```

ANEXO E – DOCKERFILE IPERF

```
# iperf3 in a container
#
# Run as Server:
# docker run  -it --rm --name=iperf3-srv -p 5201:5201 networkstatic/iperf3 -s
#
# Run as Client (first get server IP address):
# docker inspect --format "{{ .NetworkSettings.IPAddress }}" iperf3-srv
# docker run  -it --rm networkstatic/iperf3 -c <SERVER_IP>
#
FROM debian:bullseye-slim
MAINTAINER Brent Salisbury <brent.salisbury@gmail.com>
# install binary and remove cache
RUN apt-get update \
    && apt-get install -y iperf3 \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Expose the default iperf3 server port
EXPOSE 5201

# entrypoint allows you to pass your arguments to the container at runtime
# very similar to a binary you would run. For example, in the following
# docker run -it <IMAGE> --help' is like running 'iperf3 --help'
ENTRYPOINT ["iperf3"]
```

ANEXO F – DOCKER STACK TESTE IPERF

```
version: '3'

services:
  iperf-server:
    image: networkstatic/iperf3
    container_name: "iperf3-s"
    hostname: "iperf-server"
    deploy: &hpc0
      placement:
        constraints:
          - "node.hostname==ocn0"
    ports:
      - "5201:5201/tcp"
    command: iperf3 -s

  iperf-client:
    image: networkstatic/iperf3
    container_name: "Iperf3-{{.Node.Hostname}}"
    hostname: "Iperf3-{{.Node.Hostname}}"
    deploy: &hpc0
      placement:
        constraints:
          - "node.hostname==ocn1"
    command: iperf3 -c iperf-server -J -P 4 -t 60
```