Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Plataformas Virtuais para o Ensino de Computação Paralela: Desafios e Desempenho Usando Contêineres

Marco Adriano Tette Schaefer

São Carlos (SP)

Plataformas Virtuais para o Ensino de Computação Paralela: Desafios e Desempenho Usando Contêineres

Marco Adriano Tette Schaefer

Orientador: Paulo Sérgio Lopes de Souza

Monografia referente ao projeto de conclusão de curso dentro do escopo da disciplina SSC0670 do Departamento de Sistemas de Computação do Instituto de Ciências Matemáticas e de Computação – ICMC-USP para obtenção do título de Bacharel Engenheiro de Computação.

Área de concentração: Sistemas Distribuídos e Programação Concorrente.

USP - São Carlos
25 de novembro de 2019

Agradecimentos

Esta seção é destinada a agradecer a todos aqueles que, de alguma forma, contribuíram para que este trabalho fosse possível.

Primeiramente, gostaria de agradecer aos meus pais pelo apoio oferecido ao longo da graduação. Sem esse apoio, não seria possível que eu participasse deste curso de graduação, tampouco seria possível a realização deste trabalho.

Em seguida, gostaria de dedicar um agradecimento aos meus amigos e familiares, pelos momentos em que convivemos juntos ao longo desta jornada. É gratificante sempre poder contar com uma mão amiga em momentos de felicidade e momentos de dificuldade.

Também gostaria de dedicar um agradecimento ao meu orientador, o professor Paulo Sérgio, e ao doutorando Naylor Garcia. Agradeço muito pelo esforço dedicado à revisão e orientação deste trabalho, pelo empenho e reuniões que realizamos, bem como pela organização e construção do artigo desenvolvido ao longo do projeto.

Resumo

Na prática do ensino de computação paralela, um dos maiores desafios atualmente é conseguir disponibilizar equipamentos, infraestrutura e ambiente para a execução de experimentos e aprendizados na área. É custoso dispôr de um *cluster* com diversas máquinas para a execução de códigos paralelos. Além disso, é oneroso preparar um ambiente facilmente replicável para a execução de códigos que façam uso de *frameworks*. Nesse contexto, este trabalho disserta a respeito da implementação de um sistema facilmente replicável, que permite executar códigos paralelos em um *cluster* formado por contêineres.

Sumário

Listas

Lista de Tabelas

Lista de Figuras

1 Introdução

1.1 Contextualização e motivação

Sistemas computacionais atuais baseiam-se em paralelismo e multiprocessamento por conta das limitações da famosa Lei de Moore [1]. Essa lei profetizou que o número de transistores em um *chip* dobraria a cada 18 meses, pelo mesmo custo. Por muito tempo, observou-se esse efeito nos processadores presentes no mercado, porém, já há alguns anos, percebe-se que não é viável continuar aumentando apenas parâmetros como o número de transistores e a velocidade de *clock* do *chip*. Como alternativa, sistemas atuais baseiam-se fortemente em paralelismo.

Atualmente no mercado é comum encontrar dispositivos com mais de um núcleo de processamento. Dentre esses dispositivos, pode-se destacar máquinas *dektop*, *notebooks*, e até mesmo dispositivos móveis como aparelhos celulares e *tablets*. Possuir mais de um núcleo de processamento possibilita ao dispositivo processar simultaneamente instruções diferentes, podendo executar mais tarefas em menos tempo, aumentando assim seu poder de processamento sem necessariamente ter de aumentar seu número de transistores por núcleo, ou diminuir o tempo de *clock* do chip.

A programação paralela está presente em diversos sistemas que fazem parte do dia-a-dia de sociedades com fácil acesso à internet. Por exemplo, sistemas computacionais que analisam grandes volumes de dados e requerem grande desempenho baseiam-se bastante em programação paralela, como, por exemplo, sistemas para previsão do tempo, ou sistemas de inteligência artificial e aprendizado de máquina. Em arquiteturas que requerem alta disponibilidade, como, por exemplo, uma API *web* com grande número de acessos, é uma prática comum dispôr de várias instâncias da mesma aplicação, que, paralelamente, conseguem suprir às requisições de todos os usuários do sistema.

Os cursos de computação oferecem várias disciplinas de programação sequencial, em diferentes contextos, mas isso não ocorre na mesma proporção para a programação paralela. Dentre os possíveis fatores para isso, pode-se destacar o alto custo associado à compra e à manutenção de *hardware* dedicado para o ensino de programação paralela como, por exemplo, um *cluster* composto por diversos computadores interligados. Pode-se citar também, a alta complexidade de configuração de um ambiente propício para a execução de códigos no contexto de programação paralela. De fato, a programação paralela não é trivial, pois, além dos pontos já citados, pode empregar diferentes modelos de programação como *PThreads* [2], *OpenMP* [3], *MPI* [4], *CUDA* [5] entre outros.

O ensino de programação paralela geralmente requer um alto gasto com *hardware*. Geralmente, é necessário dispôr de um *cluster* de diversas máquinas interligadas, que estejam devidamente configuradas, e que sejam aptas a executar os códigos baseados na ferramenta de programação paralela escolhida. Além disso, deve-se levar em conta os gastos com a manutenção do mesmo.

Um outro fator de complexidade é que, mesmo para *softwares* paralelos básicos, o aluno necessita pensar de uma maneira diferente quando comparado à programação sequencial, e seu *software* precisa refletir isso. Em um ambiente no qual o processamento passa a ser distribuído, a sincronização dos resultados passa a ser um problema frequente e não trivial.

No contexto dessa nova maneira de se projetar *software*, o aluno deve ter conhecimento em um novo conceito: A comunicação. Processamento paralelo envolve muitas vezes comunicação e sincronização entre as partes processantes. Isso pode ser feito, por exemplo, utilizando-se passagem de mensagens, ou por meio de uma memória compartilhada entre os processos participantes.

Caso os cursos de computação pudessem utilizar recursos já disponíveis em seus laboratórios de ensino, desobrigando o conhecimento de detalhes da plataforma, e facilitando o uso durante as aulas, o ensino da programação paralela seria popularizado, aumentando a oferta de futuros profissionais melhor capacitados. A aptidão no contexto de programação paralela, poderá permitir ao profissional desenvolver aplicações mais eficientes, que melhor aproveitem os multiprocessadores do dispositivo, e que, por consequência, tenham mais qualidade.

1.2 Objetivos

O grande, e principal, objetivo deste trabalho é contribuir para a popularização do ensino da programação paralela, provendo camadas de abstração que escondam complexidades de configuração do sistema e que sejam de baixo custo de implementação e utilização.

Para alcançar o grande objetivo, este trabalho deseja construir e preparar contêineres para o ensino de programação paralela. Esses contêineres deverão ser de simples configuração, e poderão ser utilizados por alunos e professores para a prática do ensino de programação. Nos contêineres utilizados pelo professor, haverá um serviço que expõe uma API para o recebimento de códigos que podem fazer uso das ferramentas de programação paralela *OpenMP* [3] e *MPI* [4]. No contêiner utilizado pelos alunos, haverá um programa capaz de enviar um código para execução remota, informando parâmetros adicionais de execução como, por exemplo, o número de processos participantes da execução, e o número de *hosts* envolvidos.

1.3 Organização do trabalho

Este trabalho está organizado em 6 capítulos. No primeiro capítulo, encontra-se apenas esta introdução ao trabalho realizado, destacando contexto, motivações e objetivos pretendidos. No segundo capítulo, são apresentadas as listas figuras e tabelas deste trabalho. O terceiro capítulo contém a introdução deste trabalho, apresentando ao leitor contextualização a respeito do tema, a motivação para a realização do projeto, bem como os objetivos propostos e a organização deste documento. No quarto capítulo encontra-se a revisão bibliográfica, na qual apresenta-se para o leitor os conceitos de programação paralela e virtualização, conceitos estes que serão os pontos chave do trabalho desenvolvido. No quinto capítulo é apresentado o desenvolvimento do trabalho, discorrendo sobre o projeto em si, detalhes das atividades realizadas, detalhes do sistema desenvolvido, avaliação entre diferentes tecnologias para o compartilhamento de arquivos e diretórios, bem como os resultados obtidos nos projetos implementados e nos experimentos propostos. No sexto e último capítulo são apresentadas as conclusões sobre o trabalho, bem como listadas as contribuições realizadas, o relacionamento entre o curso e o projeto, além de considerações sobre o curso de graduação e possíveis trabalhos futuros. Ao final do documento, são apresentados apêndices referenciados ao longo do texto.

2 Revisão bibliográfica

2.1 Programação Paralela

A programação paralela viabiliza a utilização de diversos processos e cálculos simultâneos. Da mesma forma que a programação paralela abre novas possibilidades por conta de permitir execuções simultâneas, também introduz novos desafios por conta de sua complexidade adicional em comparação à programação sequencial, além de introduzir também novas ferramentas [6].

Quando comparada à programação sequencial, uma das principais diferenças que a programação paralela introduz é a necessidade de comunicação entre diferentes processos, estejam eles sendo executados na mesma máquina ou não. Por meio dessa comunicação, diferentes processos podem trocar informações entre si [7].

A troca de informações entre diferentes processos pode ocorrer, por exemplo, por meio de uma memória compartilhada. Nesse cenário, os processos participantes da execução têm acesso à mesma memória e, com isso, podem escrever seus resultados diretamente nela para que sejam acessados por outro processo. É necessário ter cautela para que não ocorram inconsistências nos resultados, podendo ser necessário, por exemplo, o uso de um *mutex* [8] para não permitir o acesso simultâneo de mais de um processo a um determinado dado na memória, quando este for compartilhado para escrita e leitura entre diferentes processos ou *threads*.

A comunicação entre diferentes processos também pode ocorrer por meio de troca de mensagens. Nesse cenário, os processos participantes não escrevem na memória utilizada pelos demais, e realizam sua comunicação por meio de instruções de passagem de mensagens. Portanto, em determinados trechos do código, haverá instruções que enviam ou que esperam o recebimento de dados de demais processos.

Nesse contexto, a arquitetura paralela do *hardware* e os sistemas operacionais utilizados são de fundamental importância. As arquiteturas paralelas podem ser classificadas de acordo com a taxonomia de *Flynn* [9]. Essa taxonomia classifica as arquiteturas de acordo com a pluralidade de processadores e pluralidade de dados utilizados nas execuções.

Dentre as arquiteturas classificadas por *Flynn*, pode-se destacar as arquiteturas *MIMD* (*Multiple Instruction, Multiple Data*). *Hardwares* baseados nessas arquiteturas são capazes de processar múltiplas instruções com múltiplos dados simultaneamente. Sistemas baseados nessas arquiteturas podem ainda contar com memória distribuída, ou memória compartilhada, fator esse que pode influenciar na estratégia escolhida para a comunicação entre os processos (passagem de mensagens ou acesso à memória compartilhada).

Também pode-se destacar as arquiteturas *SIMD* (*Single instruction, multiple data*). *GPUs* (*Graphics Processing Unit*), por exemplo, são baseadas nesse tipo de arquitetura (embora apresentem características a mais que máquinas *SIMD* puras. São *hardwares* capazes de executar uma mesma instrução em um conjunto grande de dados. Esse tipo de arquitetura é muito útil quando necessita-se, por exemplo, realizar operações em grandes matrizes de dados.

Do ponto de vista de sistemas operacionais, é muito comum atualmente encontrar sistemas operacionais de rede baseados em *Linux*. Os sistemas operacionais utilizados devem implementar rotinas e funções de sistema que permitam a geração de processos e a comunicação/sincronização entre os diversos processos/*threads* participantes da execução.

Para a criação de códigos de programação paralela, pode-se empregar diferentes modelos de programação. Dentre os modelos disponíveis para a linguagem C, pode-se destacar, por exemplo, *PThreads*. *PThreads* é a implementação de *threads* especificada pelo padrão POSIX [10]. Essa implementação permite que o programador gerencie a criação e execução de diversas *threads* simultaneamente, realizando chamadas de sistema. Para sua utilização, basta usar um

sistema operacional que implemente o padrão *POSIX*, que possua o compilador da linguagem C (*GCC*) [11] e que possua a biblioteca *pthread*. No momento da compilação, basta informar para o compilador a *flag -lpthread*. O Trecho 1 contém um exemplo básico da criação e finalização de *threads* por meio das funções da biblioteca *PThreads*

Trecho 1: Uso básico da biblioteca *PThreads*

```
pthread_t threads[2];

int i;

for(i=0; i<2; i++) {
    pthread_create(&(threads[i]), NULL, thread_func, NULL);
}

for(i=0; i<2; i++) {
    pthread_join(threads[i], NULL);
}
```

Um outro modelo que merece destaque por sua grande utilização é o *OpenMP* (*Open Multi-Processing*) [3]. O *OpenMP* é uma API que implementa métodos para executar e gerenciar processos, permitindo a comunicação entre eles por meio de uma memória compartilhada. Sua utilização requer a instalação de bibliotecas e dependências específicas, bem como o uso da *flag -fopenmp* no momento da compilação. Sua utilização se dá por meio de diretivas de compilação do tipo *#pragma* informadas no código, especificando um trecho do código sequencial no qual se deseja paralelizar automaticamente pelo *OpenMP*. No Trecho 2 é possível observar um uso trivial de uma diretiva do *OpenMP*, onde a impressão da *string* será feita por diferentes *threads* (neste caso o número de *threads* será o mesmo que o número de núcleos do processador em uso).

Trecho 2: Exemplo de diretiva da biblioteca *OpenMP*

```
#pragma omp parallel
 {
     printf("Hello World!\n");
 }
```

Também é de grande importância o modelo de passagem de mensagens determinado pelo padrão *MPI* (*Message Passing Interface*)[4]. O padrão *MPI* possui diversas implementações como, por exemplo, *OpenMPI* [4], *MPICH* [12], *LAM_MPI* [13], entre outros. No padrão *MPI*, a passagem de mensagens pode ser feita em um único *host* com processos distintos, ou até mesmo em *hosts* distintos. Caso a passagem de mensagens seja realizada em um mesmo *host*, a configuração da plataforma para a execução é mais simplificada, bastando apenas instalar as dependências do *MPI*, e utilizar um compilador e um executor específicos para a implementação. Caso a passagem de mensagens vá ocorrer entre *hosts* distintos, a configuração torna-se mais complexa, sendo necessário configurar todo o acesso remoto (*SSH* [14]) entre os *hosts* envolvidos, além de também ser desejável (e usual) configurar acesso a um mesmo sistema de arquivos compartilhado, como, por exemplo, um diretório montado via *NFS* [15].

O Trecho 3 contém um simples uso do *MPI*. O código ilustrado apenas retorna uma mensagem para cada processo participante da execução. Esta monografia não detalhará as funções do *MPI* para a passagem de mensagens. Para mais informações sobre a construção de algoritmos com

o padrão MPI, consulte [4].

Trecho 3: Exemplo de uso do *MPI*

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d
        processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

O Trecho 4 exemplifica a compilação e execução de um arquivo que faz uso do *MPI*. É necessário o uso de *scripts* para a compilação e execução no MPI, respectivamente o *mpicc* e o *mpiexec* (ou *mpirun* neste último). Tais *scripts* encapsulam detalhes de compilação pelo *gcc* [11] e de execução dos processos em potencialmente diferentes *hosts*. Os *hosts* que participarão da execução podem ser listados por meio de um *hostfile* no *mpiexec*. Um *hostfile* é simplesmente um arquivo que, em cada linha, contém um endereço (ou um nome que possa ser resolvido) de um *host*.

Trecho 4: Exemplo de compilação e execução de um arquivo que faz uso do *MPI*

```
mpicc main.c -o main;
mpiexec -n 4 --hostfile ./hostfile main
```

A criação de programas no contexto da programação paralela requer conhecimentos não apenas de programação em si, mas também de vários detalhes da infraestrutura básica, envolvendo hardware e software. Considerando o ensino, essas aulas utilizam laboratórios compartilhados com outras disciplinas, o que exige bastante flexibilidade destes para otimizar o uso dos recursos disponíveis aos alunos na instituição de ensino. Nesse cenário, a virtualização é uma poderosa ferramenta capaz de proporcionar flexibilidade na criação de *hosts* virtuais, sem a necessidade

de obtenção de *hardware* adicional. A virtualização também favorece em muito a configuração dos sistemas que utilizarão a plataforma compartilhada, pois isola características específicas dos sistemas que serão utilizados.

A alta complexidade na criação e manutenção de um *cluster* composto por diversas máquinas físicas, além da dificuldade de se reutilizar o *hardware* para propósitos de disciplinas de outras áreas, bem como a alta dificuldade de se configurar toda essa arquitetura, são problemas que podem ser resolvidos ou atenuados por meio da virtualização.

2.2 Virtualização

A virtualização é o processo de se abstrair, por meio de *software*, componentes de *hardware* e sistema operacional. Caso o *hardware* hospedeiro possua suporte, pode-se virtualizar até mesmo um sistema operacional diferente do contido na máquina hospedeira, como se fosse, de fato, uma máquina física diferente.

Dessa forma pode-se, por exemplo, possuindo uma única máquina, criar diversas máquinas virtuais independentes, como se fossem, de fato, máquinas físicas diferentes. Esse processo ajuda a reduzir gastos monetários em compras de novas máquinas, e permite boa flexibilização, visto que, a qualquer momento, pode-se alterar as configurações das máquinas virtualizadas, bem como acrescentar ou diminuir o número de máquinas, sem ser necessário fazer alterações no *hardware* utilizado.

A virtualização pode impactar no desempenho das aplicações executadas, pois insere uma camada extra de software entre as tais aplicações e o hardware que de fato as executará. No contexto de programação paralela, a melhoria do desempenho é um fator essencial e motivador para da mesma. Dentre os pontos críticos de desempenho em sistemas virtualizados pode-se destacar, por exemplo, sistemas de arquivos remotos ou locais, como, por exemplo, o uso do *NFS* [15].

O *NFS*, ou *Network File System*, é um sistema para compartilhamento de arquivos e diretórios através da rede. Essa tecnologia permite que um diretório exposto seja montado na mesma máquina física local, ou em uma máquina física remota. Dentre suas peculiaridades, pode-se destacar que o *NFS* possui dois modos de operação: síncrono e assíncrono. No modo síncrono, há persistência dos dados em disco a cada alteração, ou seja, as alterações são imediatamente gravadas em disco. Já no modo assíncrono, os dados podem permanecer em memória durante algum tempo antes de serem gravados em disco. O modo assíncrono oferece melhor desempenho já que não precisa esperar o disco gravar as alterações; mas em caso de queda de energia por exemplo, há maior risco de perda de dados, já que dados que estavam apenas em memória serão perdidos no desligamento [16].

A virtualização não se faz presente apenas por meio de máquinas virtuais. Um outro recurso cada vez mais utilizado são os contêineres. Diferentemente de uma máquina virtual, um contêiner é uma abstração de *software*, utilizada para replicar um determinado ambiente em qualquer máquina capaz de executar tal tecnologia. É possível abstrair o sistema operacional em um ambiente controlado, sem ser necessário abstrair o *hardware* como ocorre em máquinas virtuais. Esse ambiente pode, teoricamente, ser replicado por qualquer máquina capaz de interpretar tal tecnologia, de forma que a configuração e o ambiente da máquina hospedeira, não interferem no ambiente criado pelo contêiner. Dessa forma, consegue-se eliminar interferências causadas por configurações específicas da máquina do usuário e, assim, replicar a mesma execução em máquinas físicas diferentes com sistemas operacionais e configurações diferentes. Essa característica é altamente utilizada pela indústria, tendo em vista que, por meio de contêineres, um desenvolvedor pode garantir que o mesmo ambiente que foi utilizado durante o desenvol-

vimento em sua máquina local pode ser replicado em um servidor remoto, eliminando assim, possíveis erros que poderiam ser causados por conta de se executar a aplicação em ambiente com características e configurações diferentes [17].

A utilização de contêineres corrobora com algumas das boas práticas descritas nos *12 Fatores* [18], que são um manifesto com 12 metodologias para serem seguidas no desenvolvimento de uma aplicação *web*. Dentre as boas práticas, pode-se destacar o décimo item da lista: "Paridade entre desenvolvimento e produção"[19]. O uso de contêineres possibilita a criação de ambientes de desenvolvimento e produção muito semelhantes.

A implementação de contêineres mais utilizada é a oferecida pelo *Docker* [20]. *Docker* é um *software* capaz de criar e gerenciar contêineres em uma máquina hospedeira. Por meio de um *Dockerfile*, o usuário pode especificar detalhadamente o ambiente que será construído em seu contêiner, desde o sistema operacional que será utilizado, até as aplicações e dependências que estarão instaladas, bem como até mesmo configurações de usuário e arquivos contidos no contêiner. Além disso, é possível utilizar imagens de contêiner públicas disponibilizadas em um *Docker registry*, e até mesmo compor novas imagens de contêiner utilizando imagens já existentes. O Trecho 5 contém um exemplo básico da utilização de *Docker* para executar um contêiner contendo *Nginx*, um popular servidor *web*, e mapeando a porta 80 do contêiner para a porta 80 do *host* e, assim, conseguir expor externamente um serviço contido no contêiner.

Trecho 5: Exemplo básico de utilização do *Docker*

```
1 docker run -p 80:80 nginx
```

Observando o exemplo contido no Trecho 5 pode-se notar a simplicidade do processo. Por meio do *Docker*, não é necessário instalar as dependências exigidas pelo *Nginx*, tampouco gastar esforço em configurações do servidor. Com um único comando, pode-se utilizar um servidor *web* e expor seus serviços externamente.

Devido à sua alta flexibilidade, contêineres são utilizados largamente em aplicações portadas para a *cloud*. Provedores como a *AWS* (*Amazon Web Services*) [21] e a *Google Cloud* [22] por exemplo, oferecem diversos serviços para a utilização de contêineres, como registros e orquestradores.

No contexto do ensino de programação paralela, contêineres se fazem promissores devido à sua alta flexibilidade, além de seu baixo custo de criação de manutenção quando comparado, por exemplo, à utilização de um *cluster* composto por diversas máquinas físicas. Um ambiente mais flexível e de menor custo, pode contribuir para a disseminação do ensino de programação paralela.

3 Desenvolvimento do trabalho

3.1 Considerações Iniciais

Este capítulo tem a finalidade de descrever o trabalho desenvolvido. Discorre-se a respeito dos objetivos específicos propostos no projeto, bem como sobre atividades que foram desenvolvidas. São apresentados detalhes de implementação e decisões técnicas feitas ao longo do desenvolvimento. O capítulo também apresenta os experimentos comparando as tecnologias *NFS* e *Docker Volumes*, no que diz respeito ao compartilhamento de arquivos e diretórios entre contêineres. Ao final do capítulo, são apresentados os resultados obtidos com o trabalho desenvolvido.

3.2 Projeto

O projeto desenvolvido neste trabalho é um sistema para ajudar a disseminar o ensino da programação paralela, diminuindo os custos de criação e manutenção, diminuindo a dificuldade de configuração, e diminuindo o tempo necessário para sua utilização.

O sistema tem como objetivo permitir a execução de códigos que fazem uso das ferramentas *OpenMP* e *MPI* de forma remota, em uma arquitetura que contém diversos *hosts* para participar do processo de execução.

Dessa forma, o sistema foi dividido em *back-end* e *front-end*. O *back-end* é o componente do sistema que deverá ser executado pela máquina utilizada pelo professor, e será responsável por receber os códigos enviados pelos alunos, executá-los de acordo com os parâmetros de execução recebidos, e devolver uma resposta para o aluno que enviou o código. O *back-end* foi projetado para ser flexível, de forma que seja simples mudar as configurações do sistema, podendo, por exemplo, adicionar ou remover *hosts* virtuais de forma simples, sem ser necessário possuir conhecimentos avançados para isso. O *back-end* contém também, um serviço que expõe uma API para realizar o recebimento e execução dos códigos enviados pelos alunos.

O *front-end* é o componente que deverá ser executado na(s) máquina(s) utilizada(s) pelo(s) aluno(s). O *front-end* contém uma aplicação capaz de receber um arquivo desenvolvido em C e que utilize as ferramentas citadas anteriormente, e enviá-lo para execução remota no *back-end*. A aplicação também é capaz de receber parâmetros de execução como o número de *hosts* envolvidos na execução e o número de processos que serão utilizados e informá-los ao *back-end* via requisição *HTTP*.

Como desempenho é um fator importante em programação paralela, e o compartilhamento de diretórios e arquivos é um ponto crítico, o projeto também envolve o estudo e comparação do compartilhamento de diretórios e arquivos entre as tecnologias *NFS* [15] e *Docker Volumes* [23]. Tem-se como objetivo compará-las sob os pontos de vista de desempenho e complexidade de configuração.

A arquitetura geral do sistema é apresentada na Figura 1. Pode-se observar que a arquitetura permite que diversos alunos utilizem o sistema simultaneamente.

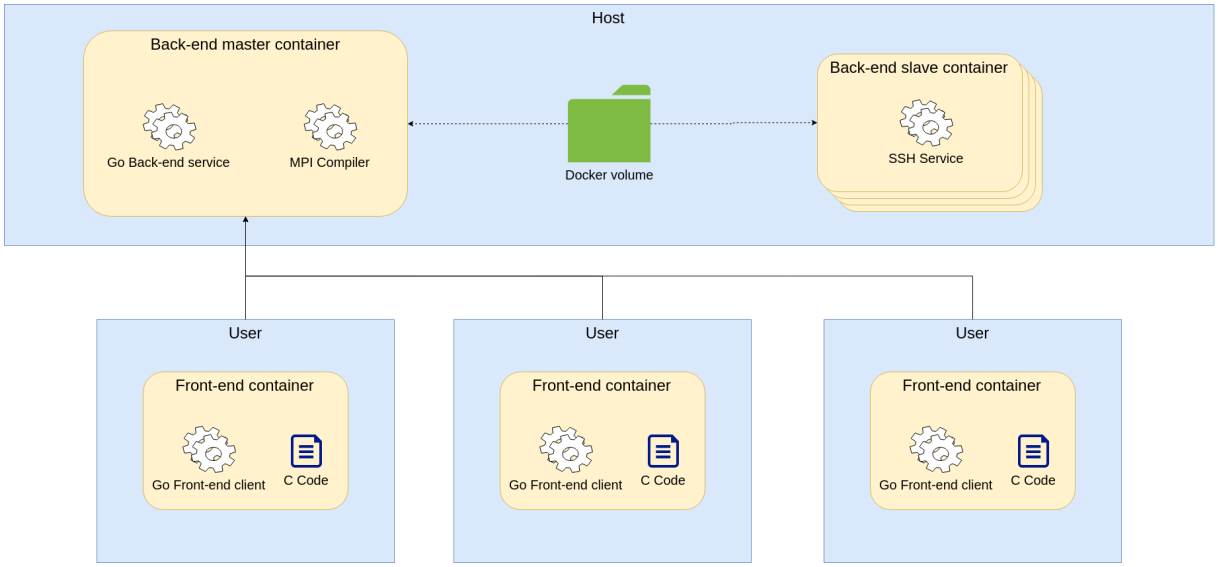


Figura 1: Representação da arquitetura do trabalho.

3.3 Descrição das atividades realizadas

Para a construção do projeto, inicialmente foi realizado um estudo a respeito das dependências necessárias para a criação de um ambiente com suporte à execução de códigos que façam uso das ferramentas *OpenMP* e *MPI*. O resultado desse estudo foi aplicado na criação das imagens *Docker* que descrevem os contêineres que compõem o sistema.

Idealizou-se a arquitetura ilustrada na Figura 1. A arquitetura foi proposta visando praticidade e, principalmente, flexibilidade. Foi projetada de maneira a suportar o uso simultâneo de diversos alunos, tanto para execuções locais nos próprios contêineres de *front-end* quanto remotamente utilizando os contêineres que compõem o *back-end*. Além disso, pode-se facilmente alterar a quantidade de contêineres escravos, podendo acrescentar ou diminuir o número de contêineres, permitindo, por exemplo, que os alunos utilizem um número maior de *hosts* nas execuções dos códigos enviados.

Também foi realizada uma avaliação comparativa entre as tecnologias *NFS* e *Docker Volumes* no que diz respeito ao compartilhamento de volumes entre contêineres dentro de um mesmo *host* físico. A comparação levou em conta aspectos relativos à complexidade de configuração e utilização, porém, o fator mais profundamente avaliado foi o desempenho. Foram realizados testes de *benchmark* entre as tecnologias, incluindo dois modos diferentes de operação do *NFS*, e com isso mensurou-se o desempenho de ambas as tecnologias em diferentes cenários. Os resultados serão apresentados nas próximas seções.

3.3.1 Detalhes do Sistema Desenvolvido

Inicialmente, foi construída uma imagem de contêiner com suporte à execução dos códigos que fazem uso das ferramentas *OpenMP* e *MPI*. Como sistema operacional base, foi escolhido o sistema *Debian*, em sua versão 8. Utilizou-se uma imagem com uma versão mais enxuta desse sistema operacional, denominada *debian:8-slim*, que está disponível publicamente no *Docker Hub* [24].

Após a escolha da imagem base do contêiner dos *hosts* do *back-end*, foi necessário determinar as dependências necessárias para a execução dos códigos. O *Dockerfile* descreve a instalação dos pacotes de cliente e servidor *SSH* [14], bem como a instalação das dependências específicas do

MPI que, nas distribuições baseadas em *Debian*, levam o nome de *openmpi-bin* e *mpi-default-dev*.

Como o contêiner fará uso de *SSH*, foram gerados de antemão arquivos para a configuração do serviço *SSH*. Gerou-se um par de chaves RSA, uma chave pública e uma chave privada, para realizar a autenticação via *SSH* e assim habilitar o acesso remoto entre os contêineres que compõem o *back-end* sem a necessidade de uma senha. Para o *MPI* poder utilizar múltiplos *hosts* nas execuções, é necessário que o *host* que iniciou a execução do programa tenha acesso remoto irrestrito aos demais *hosts* que irão participar da execução. O Trecho 6 ilustra um exemplo de geração de um par de chaves RSA. Ao longo da execução do comando, pode-se informar o caminho para a geração das chaves pública e privada.

Trecho 6: Exemplo de geração de um par de chaves RSA

```
1 ssh-keygen -t rsa -b 4096 -C "user@mail.com"
```

Além do par de chaves pública e privada, o serviço *SSH* permite a criação de um arquivo denominado *authorized_keys*. Nesse arquivo, escreve-se a chave pública de todos os *hosts* autorizados a acessar o *host* via *SSH*. Esse arquivo é o que garante que um contêiner possa ter seu acesso autorizado a um outro contêiner, algo que é vital para a execução dos códigos que fazem uso do *MPI*.

Adicionalmente, também é utilizado um arquivo denominado *known_hosts*. O cliente *SSH* do *host* utiliza esse arquivo para identificar todos os *hosts* já conhecidos e que, portanto, podem ser acessados sem que seja necessária uma confirmação por parte do usuário. Como o *back-end* foi projetado para funcionar sem que seja necessária autorização do usuário para que os contêineres possam se acessar via *SSH*, a presença desse arquivo é fundamental.

Para que todos esses arquivos relativos ao *SSH* possam ser utilizados, o serviço de *SSH* impõe que eles tenham acesso restrito a certos níveis. No caso, o diretório que contém esses arquivos (por padrão, dentro da *home* do usuário é criado um diretório denominado *.ssh*) deve ser permissionado no modo 700. Esse modo permite que o dono do diretório tenha total controle sobre ele, enquanto que os demais usuários não possuem nenhuma permissão sobre o mesmo. No caso dos arquivos contidos no diretório, é necessário que os mesmos sejam permissionados no modo 600. Esse modo permite que o dono dos arquivos possa lê-los e escrever neles, enquanto que os demais usuários não possuem autorização para nenhuma ação sobre eles. O Trecho 7 ilustra um exemplo de comando usado para alterar as permissões sobre um diretório ou arquivo.

Trecho 7: Exemplo de comando para alterar as permissões de um arquivo

```
1 chmod 600 .ssh/authorized_keys
```

Após a preparação de todas as chaves de acesso e arquivos de configuração, é necessário iniciar o serviço de *SSH* para que o *host* possa ser acessado remotamente. O Trecho 8 ilustra o comando utilizado para iniciar o servidor *SSH* em distribuições *Linux* baseadas em *Debian*.

Trecho 8: Comando utilizado para iniciar o servidor *SSH*

```
1 service ssh start
```

A imagem base, que foi utilizada no início deste trabalho para compor as demais imagens que serão apresentadas, está contida no Trecho 16.

Após a construção de uma imagem base de contêiner capaz de executar códigos *OpenMP* e *MPI*, e capaz de acessar e ser acessada remotamente por outros *hosts*, foi projetado um serviço

que expõe uma API para o recebimento de código. Esse serviço tem como função expor uma rota *HTTP* para o recebimento de código e parâmetros de execução. Em seguida, o serviço coordena a execução desse código utilizando o número de *hosts* e processos informados nos parâmetros de execução, e, por fim, retorna a resposta obtida ao remetente.

Esse serviço foi desenvolvido na linguagem Go [25], conhecida por ser uma linguagem de alto desempenho.

O serviço expõe uma rota *HTTP* no *endpoint /job*. Nessa rota, ele escuta requisições *HTTP* com o verbo *POST*, e espera como parâmetros o arquivo *C* informado em um parâmetro *file*, o número de *hosts* utilizados na execução do código informado em um parâmetro *hosts*, e o número de processos que serão utilizados na execução informado pelo parâmetro *processes*. O Trecho 9 ilustra um exemplo de *script* que pode ser utilizado para enviar uma requisição com os parâmetros citados usando a ferramenta *curl*, comumente presente em distribuições Linux.

Trecho 9: Exemplo de script que pode ser usado para enviar um código para execução.

```
curl -X POST \
http://localhost:8000/job \
-F hosts=5 \
-F processes=10 \
-F file=@/path/to/file.c
```

Após recebidos os parâmetros, o serviço irá preparar a execução do código recebido. A primeira etapa é a geração de um arquivo *hostfile*, contendo uma lista dos *hosts* que participarão da execução. O serviço contém um pequeno arquivo de configuração que lista todos os *hosts* conhecidos. Ao receber o valor *n* no parâmetro *hosts* pela requisição *HTTP*, são escritos no arquivo *hostfile* os *n* primeiros *hosts* lidos do arquivo de configuração, limitado até o número máximo de *hosts* conhecidos.

Após preparado o arquivo com a lista de *hosts*, o serviço realiza a compilação do código recebido. O código é compilado utilizando o *mpicc* [26], programa responsável por compilar códigos que fazem uso do *MPI*. Dentre as *flags* de compilação, é informada também a *flag* que habilita o uso da biblioteca do *OpenMP*. Dessa forma, é possível utilizar ambas as ferramentas e manter sempre o mesmo comando para compilação do código. O Trecho 10 ilustra o comando utilizado para compilar códigos que podem fazer uso tanto do *OpenMP* quanto do *MPI*.

Trecho 10: Comando usado para compilar os códigos recebidos.

```
mpicc -fopenmp main.c -o main
```

Após a compilação do código recebido, é realizada a execução com os parâmetros informados. O código é executando utilizando o *mpiexec* [27], programa responsável por executar códigos que façam uso do *MPI*. Dentre as *flags* de execução, são informados o número de processos participantes, bem como o caminho para o *hostfile* gerado anteriormente. O Trecho 11 ilustra o comando utilizado para executar os códigos recebidos, informando os parâmetros de execução.

Trecho 11: Comando usado para executar os códigos recebidos.

```
mpiexec -n 4 --hostfile ./hostfile main
```

Ao final da execução, a resposta é devolvida ao usuário como resposta da própria requisição *HTTP*.

Após a construção do serviço responsável pelo recebimento e execução dos códigos recebi-

dos, adaptou-se o *Dockerfile* apresentado no Trecho 16 para realizar também a compilação do serviço e incluí-lo no contêiner final. Para tal, utilizou-se um processo conhecido como *multi-stage build* [28]. Nesse processo, utilizou-se no início do *Dockerfile* uma outra imagem base, que contém o ferramental necessário para compilar o serviço. Compila-se, então, a aplicação em uma primeira imagem, e porta-se o binário obtido para a segunda imagem, que irá compor o contêiner final que será utilizado. O *Dockerfile* final da imagem utilizada pelo contêiner mestre do *back-end* encontra-se no Trecho 17.

3.3.2 Avaliação de Desempenho do NFS e Docker Volumes

Como pode ser observado na Figura 1, que ilustra a arquitetura geral do sistema, o *back-end* é composto por um contêiner mestre, que contém o serviço de recebimento e execução de código, e vários contêineres escravos que auxiliam na execução dos códigos recebidos. Todos esses contêineres precisam ter acesso a um mesmo diretório no qual são compartilhados os binários gerados após a compilação dos códigos. Tradicionalmente, implementações de *clusters* para a execução de código *MPI* comumente utilizam o *NFS* [29] para realizar o compartilhamento de diretórios. Por outro lado, quando se deseja compartilhar um mesmo volume entre diferentes contêineres, é comum a utilização de *Docker Volumes*. Então, foi realizado um estudo comparativo entre *NFS* e *Docker Volumes*, levando-se em conta desempenho e dificuldade de configuração.

Inicialmente, comparou-se a dificuldade de configuração de ambas as tecnologias. No caso do *NFS*, para que os contêineres possam montar um diretório externo, ou até mesmo permitir que um diretório deles seja acessado por outro *host*, é necessário que a máquina hospedeira esteja executando um serviço do *NFS*. Considerando o contexto de contêineres, em que deseja-se criar ambientes que possam ser facilmente replicáveis em outras máquinas, o fato de ser necessário que a máquina hospedeira esteja executando um serviço específico é uma grande desvantagem. Além disso, os contêineres que farão uso do *NFS* precisam conter as dependências necessárias para tal. O Trecho 12 mostra exemplos de comandos que podem ser utilizados para iniciar um servidor *NFS*, e um comando que pode ser utilizado para montar um volume remoto pelo *NFS*.

Trecho 12: Exemplos de comandos utilizados para iniciar um servidor *NFS*, e para montar um volume externo.

```
1 sudo systemctl start nfs-server.service;
2 sudo mount -v -t nfs 172.17.0.2:/ /mnt/nfs;
```

Por outro lado, a utilização de *Docker volumes* em contêineres tem uma configuração muito mais simples. Diferentemente do *NFS*, não é necessário que a máquina hospedeira execute um serviço a parte, tampouco é necessário instalar dependências específicas no contêiner. Iniciando um contêiner pela linha de comando do *Docker*, basta informar uma *flag* adicional, contendo o caminho do diretório da máquina hospedeira, e o caminho no qual o diretório será mapeado para dentro do contêiner. O Trecho 13 ilustra um exemplo de comando que pode ser utilizado para iniciar um contêiner, mapeando um volume da máquina hospedeira para dentro do contêiner.

Trecho 13: Exemplo de comando utilizado para iniciar um contêiner, mapeando um volume.

```
1 docker run -v /host/path:/container/path image_name
```

Em seguida, elaborou-se um experimento para comparar o desempenho entre *NFS* e *Docker Volumes*. Foi utilizada uma máquina virtual com o sistema operacional *Ubuntu Server* 18.04.3 LTS. Para diminuir os efeitos de *caching* e aumentar a taxa de acessos a disco, foram provisionado

apenas 512MB de memória RAM para a máquina virtual [30]. A máquina hospedeira do experimento possui um processador *Intel Core i7 -3537U*, 8GB de memória RAM, um disco rígido *HDD*, e um *SSD* [30].

A ferramenta de *benchmarking* utilizada foi o SysBench [31]. Escolheu-se essa ferramenta com base em um levantamento de trabalhos relacionados, no qual notou-se que o *Sysbench* foi a ferramenta mais utilizada por estes [30]. A Tabela 1 apresenta o levantamento obtido.

O experimento consiste em contêineres realizarem operações de escrita e leitura em um volume compartilhado. Ao todo, para compor os experimentos, foram realizadas 3100 medições, comparando a performance entre os *Docker Volumes*, *NFS* no modo síncrono e *NFS* no modo assíncrono [30]. Cada execução do experimento utilizou um *payload* de *2GB* e foi realizada durante 60 segundos [30]. Ao longo do experimento, foram avaliadas as performances em operações de escrita e leitura. Também avaliou-se o impacto da concorrência sobre a performance das operações, realizando o experimento tanto com um único contêiner quanto com 10 contêineres acessando simultaneamente o volume compartilhado [30].

Após o estudo sobre qual seria a tecnologia de compartilhamento de diretórios a ser utilizada, escolheu-se a utilização dos *Docker Volumes* neste trabalho. Para facilitar a configuração de toda arquitetura, utilizou-se o *Docker Compose* [32] para instanciar simultaneamente todos os contêineres do *back-end*, e desinstanciá-los quando desejado. O arquivo de descrição do *Docker Compose* é um arquivo no formato *YAML*, característico por ser bem legível e de fácil edição. Além disso, é possível informar neste arquivo de descrição os *Docker Volumes* que serão utilizados, facilitando ainda mais a configuração do ambiente. O arquivo de descrição do *Docker Compose* utilizado neste trabalho encontra-se descrito no Trecho 18.

Realizada a construção do *back-end*, foi projetada uma aplicação capaz de enviar códigos para execução remota, além de ser capaz de especificar alguns parâmetros de execução. Essa aplicação será um serviço que ficará contido no contêiner do *front-end* que será utilizado pelo aluno.

A aplicação também foi desenvolvida na linguagem *Go* [25], e sua utilização se dá pela linha de comando. É possível informar alguns parâmetros por *flags* no momento da execução. Essas *flags* serão enviadas como parâmetros na requisição *HTTP* para o *back-end*, e determinarão a quantidade de *hosts* e processos participantes da execução. A aplicação foi chamada de *rmtexec* (abreviação para *remote execution*). O Trecho 14 contém as intruções fornecidas pelo aplicação, bem como um exemplo de uso.

Trecho 14: Instruções e exemplo de utilização do *rmtexec*.

```
$ rmtexec -h
Usage of rmtexec:
  -file string
     Arquivo .c para execucao. Exemplo: --file main.c
  -hosts int
     Numero de hosts que sero usados na execucao. Exemplo:
         --hosts 3 (default 1)
  -processes int
     Numero de processos que sero usados na execucao. Exemplo:
         --processes 4 (default 1)

$ rmtexec --file main.c --hosts 2 --processes 4
```

Os parâmetros *hosts* e *processes*, caso não sejam informados, foram projetados para terem como padrão o valor 1. O único parâmetro obrigatório é o *file*, que especifica o caminho do

arquivo que será executado remotamente. Quando executada, então, a aplicação envia uma requisição *HTTP* com o método *POST* utilizando um *content-type* do tipo *multipart/form-data*. Cada um dos atributos, então, é enviado como um campo de um *form-data* na requisição. Ao final da execução, a resposta da requisição é a resposta produzida pelo código enviado. Essa resposta será exibida de volta para o aluno. No caso de erros de compilação ou execução, é retornado um erro para o usuário.

A única configuração necessária para o *rmtexec* é a rota que será utilizada para realizar a execução remota. Essa configuração será obtida ao carregar o arquivo localizado em */etc/rmtexec/config.yaml*. O diretório */etc* é comumente utilizado em ambientes *Linux* para o armazenamento de arquivos de configuração das aplicações do sistema. O Trecho 15 contém um exemplo de arquivo de configuração da aplicação *rmtexec*.

Trecho 15: Exemplo de configuração do *rmtexec*.

```
1 remote: http://172.17.0.1:8000/job
```

Finalizada a construção da aplicação *rmtexec*, foi projetada a imagem que descreve o contêiner do *front-end*, que será utilizado pelos alunos para enviarem seus códigos para execução remota. Esse contêiner por si só deve ser capaz de permitir execuções simples dentro de si (no caso, com um único *host*), além de ser capaz de compilar e portar o *rmtexec* para si. Novamente foi utilizada a técnica de *multi-stage build* [28] para gerar um contêiner construído em dois estágios. No primeiro estágio, é compilada a aplicação *rmtexec*. No segundo estágio, que tem como base a mesma imagem do *Debian* utilizada no *back-end*, são instaladas as dependências relativas ao *MPI*. Além disso, o binário gerado no primeiro estágio é copiado para o contêiner final, e alocado em um local contido no *PATH* do usuário. O *PATH*, em sistemas Linux, é o conjunto de diretórios nos quais o sistema irá procurar um binário para execução.

Como um dos objetivos deste projeto é que ele seja de simples configuração, também foi preparado um *Makefile* que abstrai os processos de construção e execução do contêiner. Na primeira execução, pode-se construir o contêiner do *front-end* a partir de um simples *make build*. Em seguida, pode-se executar o contêiner com um simples *make run*, e pode-se acessá-lo com um *make enter*. Dentro do contêiner, a aplicação *rmtexec* estará disponível para utilização, e, portanto, o aluno já poderá enviar seus códigos para execução remota.

3.4 Resultados obtidos

O sistema proposto e desenvolvido neste trabalho tem como principal objetivo, ajudar a popularizar e disseminar a prática do ensino de programação paralela. Dessa forma, desenvolveu-se uma arquitetura de um sistema simplificado para alunos e professores submeterem e executarem aplicações paralelas *MPI* e *OpenMP*. A arquitetura foi projetada para trazer flexibilidade de uso, fácil utilização e, principalmente, baixo custo de implementação e manutenção. Em situações tradicionais, a construção de um *cluster* de diversas máquinas com suporte para a execução de códigos com *MPI* e *OpenMP* é uma tarefa custosa e complexa. Por outro lado, a instanciação de um *cluster* com suporte à execução de código com as ferramentas citadas formado por contêineres em uma única máquina pode ser feito com um único comando. Tendo acesso ao código do sistema, um simples comando de *docker-compose up* é suficiente para instanciar um *cluster* de contêineres e já disponibilizá-lo para o uso. Além disso, desde que a máquina hospedeira tenha as dependências relativas ao *Docker* e ao *Docker-compose*, a arquitetura funcionará de maneira independente do sistema operacional da máquina hospedeira, aumentando assim, sua replicabilidade em diferentes máquinas com diferentes configurações. A Imagem 2 ilustra

ambos *back-end* e *front-end* em funcionamento, respectivamente dos lados esquerdo e direito.

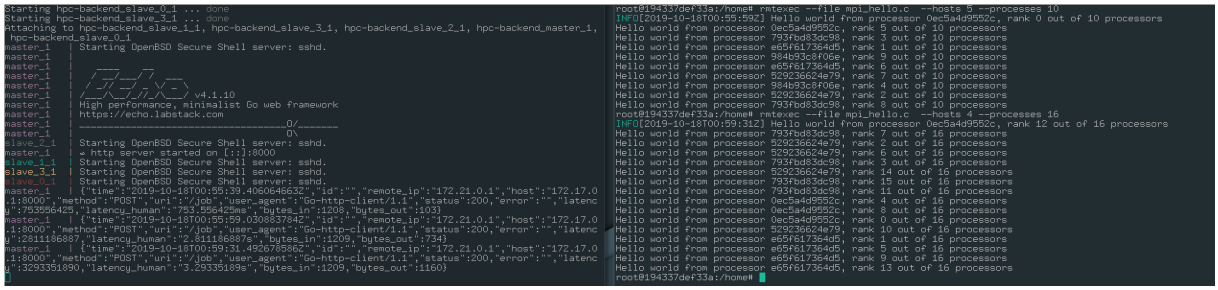
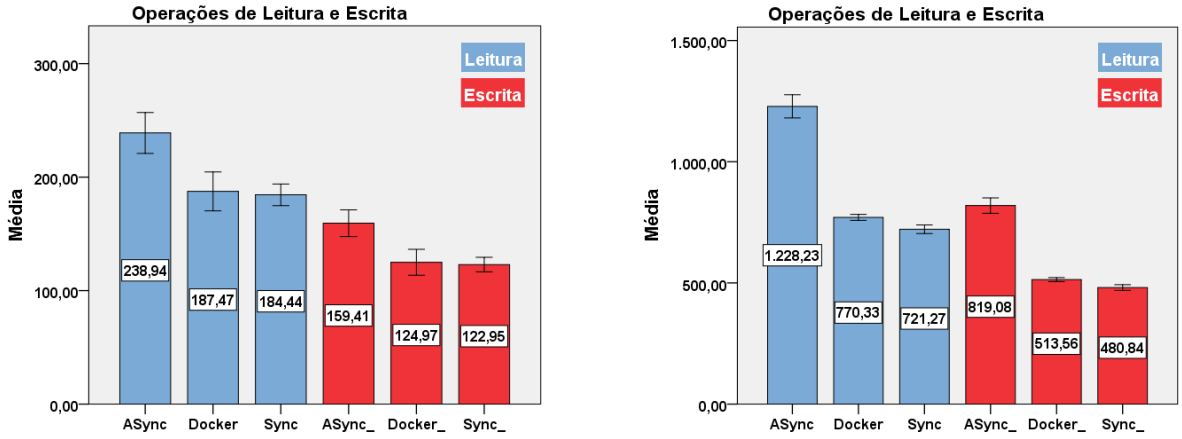


Figura 2: Exemplificação do *back-end* e *front-end* em funcionamento.

Além disso, foi desenvolvida também uma API de código aberto capaz de receber códigos enviados pelos alunos, gerenciar sua execução e devolver a resposta obtida na própria requisição. A API é exposta por um serviço executado sobre uma plataforma virtualizada, construída para ser capaz de executar códigos que fazem uso do *MPI*, de uma maneira muito mais simples do que pelas práticas convencionais.

Também desenvolveu-se um estudo a respeito da utilização dos sistemas *NFS* e *Docker Volumes* para o compartilhamento de diretórios e arquivos entre contêineres. Avaliou-se os passos necessários para sua utilização, porém, o principal resultado deu-se na comparação de desempenho [30].

Na Figura 3 são apresentados os resultados das execuções do experimento comparativo entre *NFS* e *Docker Volumes* para operações de leitura e escrita em um cenário sem concorrência, ou seja, com um único contêiner tendo acesso ao volume compartilhado. Pode-se observar que, nesse experimento, o *NFS* em modo assíncrono foi responsável pelos melhores resultados [30], tanto em operações de leitura, quanto em operações de escrita, mesmo em *hardwares* diferentes como, no caso, um disco *HDD* e um disco *SSD*. O *Docker Volume*, por sua vez, apresentou resultados levemente melhores do que os obtidos pelo *NFS* em modo síncrono. Dessa forma, caso não seja possível arcar com os riscos de confiabilidade oferecidos pelo *NFS* assíncrono, o *Docker Volume* seria uma melhor opção do ponto de vista de desempenho [30].



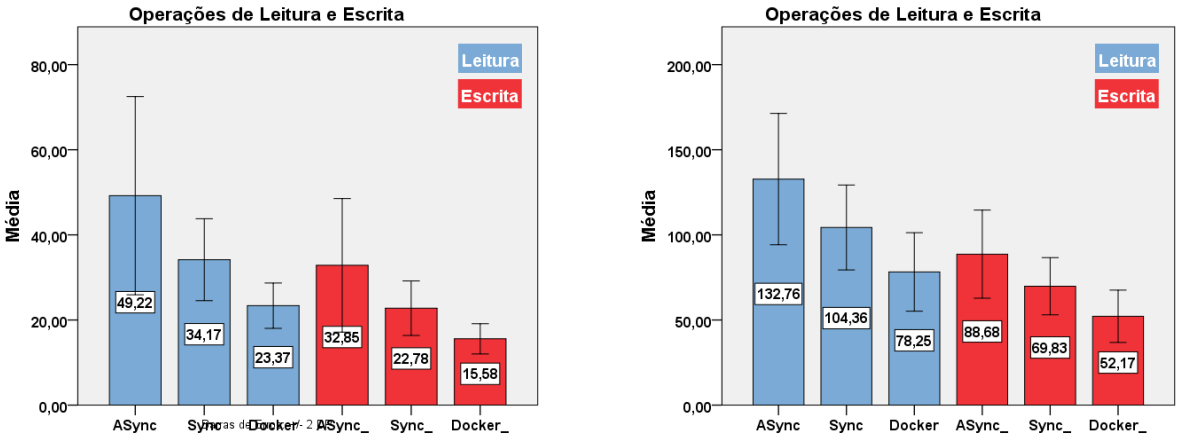
(a) Disco HDD

(b) Disco SSD

Figura 3: Quantidade de operações de leitura e escrita sem concorrência.

Em um cenário com 10 contêineres acessando concorrente o volume compartilhado, mais uma vez o *NFS* no modo assíncrono foi responsável pelos melhores resultados, apesar de uma

alta variação nos resultados das execuções. Nesse cenário, o *NFS* em modo síncrono também performou de maneira melhor do que os *Docker Volumes*. A Figura 4 apresenta os resultados obtidos nesta etapa do experimento [30].



(a) Disco HDD (b) Disco SSD

Figura 4: Quantidade de operações de leitura e escrita com 10 contêineres concorrentes.

Observou-se, portanto, que, se tratando de performance, o *NFS* em modo assíncrono proporciona os melhores resultados, porém, vale ressaltar que esse ganho de performance se dá em detrimento de confiabilidade, tendo em vista que o *NFS* nesse modo não persiste imediatamente as alterações em disco, mantendo-as temporariamente apenas em memória. Também é importante observar que um outro fator importante que deve ser levando em consideração no momento de escolher uma das tecnologias para o compartilhamento de volumes é a complexidade de configuração. Conforme discorrido na seção anterior, a configuração inicial para o uso do *NFS* é mais complexa do que a configuração inicial necessário para o uso de *Docker Volumes*, em um cenário no qual os *hosts* são contêineres.

Os resultados da análise de desempenho do *NFS* e *Docker Volumes* no compartilhamento de volumes em contêineres, em uma mesma máquina física, foram sintetizados em artigo, publicado no Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD 2019), ocorrido em Campo Grande/MS em outubro de 2019 [30].

3.5 Dificuldades e limitações

A maior dificuldade encontrada nas etapas iniciais do projeto foi com relação a utilização do *MPI* em um sistema formado por contêineres. Há escassa documentação disponível publicamente sobre o assunto. A maioria das implementações de arquiteturas para a utilização do *MPI* utiliza o *NFS* como sistema de compartilhamento de volumes e arquivos, enquanto que, se tratando de contêineres, o uso de *Docker Volumes* para o compartilhamento desses diretórios é uma prática mais comum. Como há pouca informação disponível a respeito do uso de *NFS* em contêineres ao invés do uso de *Docker Volumes*, foi realizado um estudo comparativo entre as tecnologias para auxiliar na escolha de qual ferramenta seria utilizada pelo sistema para compartilhar os volumes entre os contêineres que compõem o *back-end*.

Outra grande dificuldade foi a construção dos contêineres que compõem o *back-end*, de forma que o contêiner mestre tenha acesso irrestrito aos contêineres escravos via *SSH*. As configurações necessárias para habilitar acesso remoto via *SSH* por meio de chaves públicas e privadas sem

que seja necessária uma confirmação por parte do usuário são razoavelmente complexas, e, em muitos casos, falham silenciosamente em caso de erro, sem informar o problema ocorrido.

Uma das limitações do sistema proposto é que, sem o uso de um orquestrador, *Volumes Docker* só podem ser montados em contêineres na mesma máquina física, e não através da rede como é o caso do *NFS*. Para possibilitar o uso de um mesmo volume *Docker* através da rede em contêineres que estejam em máquinas físicas diferentes, é necessário o uso de algum orquestrador de contêineres, como *Kubernetes* [33] ou *Docker Swarm* [34].

3.6 Considerações finais

Este capítulo apresentou detalhes sobre o desenvolvimento do projeto e os resultados obtidos. Foram apresentadas as motivações e objetivos do projeto, a arquitetura e detalhes de implementação e funcionamento das diferentes partes do sistema proposto, bem como os resultados obtidos com a implementação. Além disso, discorreu-se sobre o estudo comparativo realizado sobre as tecnologias *NFS* e *Docker Volumes*, dissertando a respeito de desempenho e complexidade de configuração.

4 Conclusão

4.1 Contribuições

O trabalho desenvolvido apresenta uma solução de baixo custo de implementação, baixo custo de manutenção, e baixa complexidade de configuração para a prática do ensino de programação paralela com o uso das ferramentas *OpenMP* e *MPI*. Por conta dessas características, essa solução contribui para a disseminação do ensino de programação paralela, uma vez que recursos financeiros podem acabar deixando de ser um obstáculo.

A arquitetura desenvolvida é de simples utilização. Com um único comando, pode-se instanciar um *cluster* de contêineres para atuarem como *hosts* na execução de códigos paralelos, além de um serviço que expõe uma API para o recebimento de códigos externos, juntamente de parâmetros de execução. Todo o sistema desenvolvido é *open-source* e encontra-se disponível em repositórios públicos [35] [36], contribuindo assim, para disseminar o ensino prático de programação paralela.

4.2 Relacionamento entre o Curso e o Projeto

O projeto desenvolvido se relaciona com diversas disciplinas apresentadas ao longo do curso. O desenvolvimento desse projeto necessitou do conhecimento adquirido em diversas disciplinas do curso, como, por exemplo, Algoritmos e Estruturas de Dados, Redes de Computadores, Sistemas Operacionais, entre outras, podendo destacar, claro, Programação Concorrente, que caracteriza o núcleo deste trabalho.

Além de se relacionar diretamente com as disciplinas vistas ao longo do curso, o projeto utilizou diversas tecnologias e padrões presentes atualmente na indústria. As tecnologias utilizadas na construção do sistema realizado neste projeto são, em grande parte, largamente utilizadas no mercado. Por exemplo, é comum encontrar na indústria a utilização de *Docker* para a criação de contêineres, a utilização de *Go* para a criação de aplicações, entre outros. Portanto, o projeto relaciona-se bem com o curso por utilizar os conhecimentos apresentados em diversas disciplinas ministradas ao longo da formação, além de relacionar-se bem com a área de desenvolvimento por utilizar tecnologias presentes no mercado atualmente.

4.3 Considerações sobre o Curso de Graduação

O curso de Engenharia de Computação tem como escopo o ensino de disciplinas relacionadas tanto ao universo da computação, quanto ao universo da engenharia.

Tratando-se do escopo do ensino de computação, destaco os pontos que, em minha opinião, são positivos:

- O curso fornece uma boa base a respeito de lógica e algoritmos. O aluno é capaz de compreender os fundamentos ao longo do curso, e, caso possua interesse, tem total capacidade de aprender por si tópicos mais complexos no assunto.
- A dificuldade e exigência das disciplinas de computação evolui bem ao longo do curso.

Em minha opinião, os pontos negativos que destaco são:

- A escolha da linguagem inicial apresentada no curso, no caso, a linguagem C, é um ponto negativo. Acredito que, nas primeiras matérias de computação no curso, um dos objetivos

deve ser conseguir atrair o interesse dos alunos na área, e motivá-los a aprender mais sobre o assunto por conta própria. Nesse cenário, não considero a linguagem C uma boa escolha para isso, visto que é uma linguagem razoavelmente de baixo nível, na qual é necessária muita complexidade e implementação para a realização de tarefas ainda simples. Em contraste a isso, uma linguagem de mais alto nível como, por exemplo, *Python*, poderia acabar motivando melhor os alunos do curso, visto que é possível realizar tarefas mais complexas com menos esforço e menos dificuldade, fazendo com que os alunos vejam melhor os poderes que a computação traz para aqueles que a utilizam.

- Um outro ponto que considero negativo é a omissão de alguns temas importantes no currículo. Temas como desenvolvimento web, *clean code*, e até mesmo conteinerização não são devidamente apresentados aos alunos em disciplinas obrigatórias. Considero esses temas de extrema relevância, visto que grande parte dos alunos egressos acabando tendo contato direto com esses assuntos no mercado.

4.4 Trabalhos futuros

Existem diversas etapas que poderiam compor trabalhos futuros realizados sobre os resultados deste trabalho. Poderia-se, por exemplo, realizar a validação do sistema proposto com alunos em sala de aula, podendo dessa forma, obter *feedbacks* a respeito do uso, e com isso propor melhorias para serem acrescentas ao sistema proposto. Um outro trabalho que poderia ser realizado é a portabilidade do sistema proposto para a *cloud*, fazendo com que não seja mais necessário que o professor execute o *back-end* em sua máquina, e, dessa forma, seria necessário apenas que os alunos configurassem seus contêineres locais para enviarem os códigos para o endereço disponibilizado na *cloud*. Nesse mesmo trabalho, poderia-se utilizar um orquestrador de contêineres, como *Kubernetes*, para gerenciar os contêineres que compõem o *back-end* e, dessa forma, seria possível inclusive utilizar mais de um *host* físico para instanciar os contêineres do *back-end* e ainda assim utilizar o mesmo volume em todos os contêineres instanciados, independentemente de em qual máquina física eles se encontram. Um exemplo de arquitetura de sistema que faz o uso do *Kubernetes* é mostrado na Figura 5. Um outro trabalho possível, seria a construção de um sistema para o cadastro de professores e alunos, de forma a executar um *Saas* (*Software as a Service*) sobre o projeto proposto.

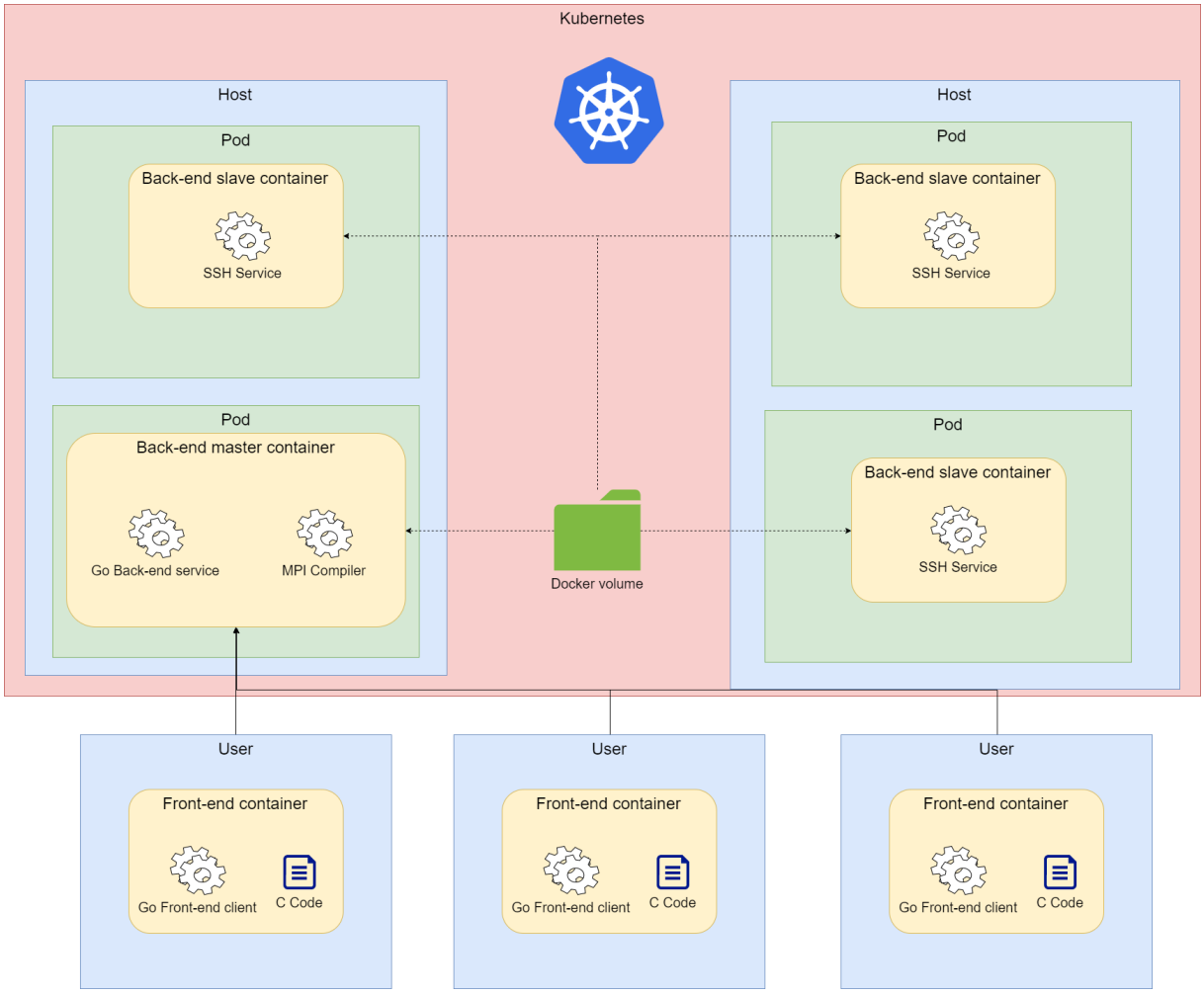


Figura 5: Exemplo de arquitetura utilizando o orquestrador *Kubernetes*.

Referências

[1] Moore's law - Disponível em: <https://www.kth.se/social/upload/507d1d3af276540519000002/Moore >Acesso em Out. 2019.

[2] PTHREADS - Linux Programmer's Manual - Disponível em: <http://man7.org/linux/man-pages/man7/pthreads.7.html >Acesso em Out. 2019.

[3] The OpenMP API specification for parallel programming - Disponível em: <https://www.openmp.org>Acesso em Mai. 2019.

[4] Open MPI: Open Source High Performance Computing - Disponível em: <https://www.open-mpi.org>Acesso em Mai. 2019.

[5] CUDA Toolkit Documentation - Disponível em: <https://docs.nvidia.com/cuda/ >Acesso em Out. 2019.

[6] Michael Quinn. *Parallel Programming*. McGraw-Hill Science/Engineering/Math, 2003.

[7] A.S TANENBAUM. *Sistemas Operacionais Modernos*. Pearson, 2010.

[8] Exclusão Mútua (mutex) - Disponível em: <https://edisciplinas.usp.br/pluginfile.php/3061851/mod$_r es$ $Mutex - v27.pdf > AcessoemOut$.2019.

[9] Computer Architecture | Flynn's taxonomy - Disponível em: <https://www.geeksforgeeks.org/computer-architecture-flynns-taxonomy/ >Acesso em Out. 2019.

[10] Posix Standard - Disponível em: <https://linuxhint.com/posix-standard/ >Acesso em Out. 2019.

[11] GCC, the GNU Compiler Collection - Disponível em: <https://gcc.gnu.org/ >Acesso em Out. 2019.

[12] MPICH | High-Performance Portable MPI - Disponível em: <https://www.mpich.org/ >Acesso em Mai. 2019.

[13] LAM / MPI Parallel Computing - Disponível em: <http://www.dcs.ed.ac.uk/home/trollius/www.osc.edu/lam.html >Acesso em Mai. 2019.

[14] ssh - Linux man page - Disponível em: <https://linux.die.net/man/1/ssh >Acesso em Out. 2019.

[15] NFS - Network File System - Disponível em: <http://web.mit.edu/rhel-doc/3/rhel-sag-pt$_b r - 3/ch - nfs.html > AcessoemOut$.2019.

[16] nfs - Linux man page - Disponível em: <https://linux.die.net/man/5/nfs >Acesso em Out. 2019.

[17] Docker and Reproducibility - Disponível em: <https://reproducible-analysis-workshop.readthedocs.io/en/latest/8.Intro-Docker.html >Acesso em Out. 2019.

[18] The Twelve Factor App - Disponível em: <https://12factor.net/pt$_br$/> $Acessoem Out$.2019.

[19] Dev/prod parity - Disponível em: <https://12factor.net/dev-prod-parity >Acesso em Out. 2019.

[20] Enterprise Container Platform for High-Velocity Innovation - Disponível em: <https://www.docker.com/>Acesso em Mai. 2019.

[21] Amazon Web Services (AWS) - Disponível em: <https://aws.amazon.com/ >Acesso em Out. 2019.

[22] Google Cloud: Cloud Computing Services - Disponível em: <https://cloud.google.com/ >Acesso em Out. 2019.

[23] Use volumes | Docker Documentation - Disponível em: <https://docs.docker.com/storage/volumes/>Acesso em Mai. 2019.

[24] Debian - Docker Hub - Disponível em: <https://hub.docker.com/$_/debian$ > $Acessoem Out$.2019.

[25] The Go Programming Language - Disponível em: <https://golang.org/>Acesso em Out. 2019.

[26] Compiles and links MPI programs written in C - Disponível em: <https://www.mpich.org/static/docs/v3.1.x/www1/mpicc.html >Acesso em Out. 2019.

[27] mpiexec - Disponível em: <https://www.mpich.org/static/docs/v3.1/www1/mpiexec.html >Acesso em Out. 2019.

[28] Use multi-stage builds - Disponível em: <https://docs.docker.com/develop/develop-images/multistage-build/ >Acesso em Out. 2019.

[29] Running an MPI Cluster within a LAN - Disponível em: <https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/ >Acesso em Out. 2019.

[30] N.G.; P.S.L. SOUZA M.A.T. Schaefer; N.G. Bachiega and S.M.S. Bruschi. Avaliação do docker volume e do nfs no compartilhamento de sistemas de arquivos em contêineres. In *Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD2019)*, volume 1, pages 1–8, October 2019.

[31] Sysbench - Scriptable database and system performance benchmark - Disponível em: <https://github.com/akopytov/sysbench >Acesso em Out. 2019.

[32] Overview of Docker Compose - Disponível em: <https://docs.docker.com/compose/ >Acesso em Out. 2019.

[33] Orquestração de contêiner pronto para produção - Disponível em: <https://kubernetes.io/pt/ >Acesso em Out. 2019.

[34] Swarm mode overview - Disponível em: <https://docs.docker.com/engine/swarm/ >Acesso em Out. 2019.

[35] HPC Frontend with MPI support - Disponível em: <https://github.com/MarcoSchaefer/hpc-frontend >Acesso em Out. 2019.

[36] HPC Backend with MPI support - Disponível em: <https://github.com/MarcoSchaefer/hpc-backend >Acesso em Out. 2019.

[37] N. G. Bachiega, P. S. L. Souza, S. M. Bruschi, and S. d. R. S. de Souza. Container-based performance evaluation: A survey and challenges. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 398–403, April 2018.

[38] J. Che, C. Shi, Y. Yu, and W. Lin. A synthetical performance evaluation of openvz, xen and kvm. In *2010 IEEE Asia-Pacific Services Computing Conference*, pages 587–594, Dec 2010.

[39] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, Feb 2013.

[40] Anish Babu, Hareesh M. J., John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 Fourth International Conference on Advances in Computing and Communications*, pages 247–250, Aug 2014.

[41] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015.

[42] C. Ruiz, E. Jeanvoine, and L. Nussbaum. Performance evaluation of containers for hpc. In *In: Hunold S. et al. (eds) Euro-Par 2015: Parallel Processing Workshops. Euro-Par 2015*, pages 813–824, December 2015.

[43] T. Adufu, J. Choi, and Y. Kim. Is container-based technology a winner for high performance scientific applications? In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 507–510, Aug 2015.

[44] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, Sept 2015.

[45] D. Beserra, E. D. Moreno, P. T. Endo, J. Barreto, D. Sadok, and S. Fernandes. Performance analysis of lxc for hpc environments. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 358–363, July 2015.

[46] A. M. Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346, March 2015.

[47] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, March 2015.

[48] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 253–260, March 2015.

[49] M. T. Chung, N. Quang-Hung, M. T. Nguyen, and N. Thoai. Using docker in high performance computing applications. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57, July 2016.

[50] A. Jaikar, S.A.R. Shah, S. Bae, and S.Y. Noh. Performance evaluation of scientific workflow on openstack and openvz. *Social-Informatics and Telecommunications Engineering, LNICST*, 167:126–135, 2016.

[51] B. Ruan, H. Huang, D. Wu, and H. Jin. A performance study of containers in cloud environment. *In: Wang G., Han Y., Martínez Pérez G. (eds) Advances in Services Computing. APSCC 2016.*, 10065:343–356, 2016.

[52] S. Herbein, A. Dusia, A. Landwehr, S. McDaniel, J. Monsalve, Y. Yang, S.R. Seelam, and M. Taufer. Resource management for running hpc applications in container clouds. *Lecture Notes in Computer Science*, 9697:261–278, 2016.

[53] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose. Performance analysis of virtual machines and containers in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1204–1210, April 2016.

[54] R. S. V. Eiras, R. S. Couto, and M. G. Rubinstein. Performance evaluation of a virtualized http proxy in kvm and docker. In *2016 7th International Conference on the Network of the Future (NOF)*, pages 1–5, Nov 2016.

[55] Z. Kozhirbayev and R.O. Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175–182, 2017.

[56] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962, March 2017.

[57] I. Mavridis and H. Karatza. Performance and overhead study of containers running on top of virtual machines. In *2017 IEEE 19th Conference on Business Informatics (CBI)*, volume 02, pages 32–38, July 2017.

[58] A. Lingayat, R. R. Badre, and A. Kumar Gupta. Performance evaluation for deploying docker containers on baremetal and virtual machine. In *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*, pages 1019–1023, Oct 2018.

[59] H. Zeng, B. Wang, W. Deng, and W. Zhang. Measurement and evaluation for docker container networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 105–108, Oct 2017.

[60] N. Mizusawa, J. Kon, Y. Seki, J. Tao, and S. Yamaguchi. Performance improvement of file operations on overlayfs for containers. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 297–302, June 2018.

A

Dockerfile base

Trecho 16: *Dockerfile* utilizado como base para a execução dos códigos

```
FROM debian:8-slim

RUN apt-get update
RUN apt-get install -y openssh-client
RUN apt-get install -y openssh-server
RUN apt-get install -y openmpi-bin
RUN apt-get install -y mpi-default-dev

RUN mkdir root/.ssh
RUN chmod 700 /root/.ssh
COPY ./ssh_keys/id_rsa.pub /root/.ssh/authorized_keys
COPY ./ssh_keys/id_rsa.pub /root/.ssh/id_rsa.pub
COPY ./ssh_keys/id_rsa /root/.ssh/id_rsa
COPY ./ssh_keys/known_hosts /root/.ssh/known_hosts
RUN chmod 600 /root/.ssh/*

RUN apt-get install -y make

COPY ./ssh_keys/ssh_host_ecdsa_key /etc/ssh/ssh_host_rsa_key
COPY ./ssh_keys/ssh_host_ecdsa_key.pub /etc/ssh/ssh_host_rsa_key.pub
RUN chmod 600 /etc/ssh/ssh_host_rsa_key

RUN apt-get install -y sudo

CMD service ssh start && tail -f /dev/null
```

B

Dockerfile do contêiner mestre do *back-end*

Trecho 17: *Dockerfile* utilizado pelo contêiner mestre do *back-end*

```
FROM golang:1.12.7-stretch as builder

COPY ./ /api

WORKDIR /api

RUN go get -d
RUN go build -o main

FROM debian:8-slim

RUN apt-get update
RUN apt-get install -y openssh-client
RUN apt-get install -y openssh-server
RUN apt-get install -y openmpi-bin
RUN apt-get install -y mpi-default-dev

RUN mkdir root/.ssh
RUN chmod 700 /root/.ssh
COPY ./ssh_keys/id_rsa.pub /root/.ssh/authorized_keys
COPY ./ssh_keys/id_rsa.pub /root/.ssh/id_rsa.pub
COPY ./ssh_keys/id_rsa /root/.ssh/id_rsa
COPY ./ssh_keys/known_hosts /root/.ssh/known_hosts
RUN chmod 600 /root/.ssh/*

RUN apt-get install -y make

COPY ./ssh_keys/ssh_host_ecdsa_key /etc/ssh/ssh_host_rsa_key
COPY ./ssh_keys/ssh_host_ecdsa_key.pub /etc/ssh/ssh_host_rsa_key.pub
RUN chmod 600 /etc/ssh/ssh_host_rsa_key

RUN apt-get install -y sudo

COPY --from=builder /api/main /api/main
COPY --from=builder /api/config /api/config

WORKDIR /api

CMD service ssh start && ./main && tail -f /dev/null
```

C

Docker-compose do *back-end*.

Trecho 18: Arquivo *docker-compose.yaml*, responsável por instanciar todo o *back-end*.

```
FROM golang:1.12.7-stretch as builder

COPY ./ /api

WORKDIR /api

RUN go get -d
RUN go build -o main

FROM debian:8-slim

RUN apt-get update
RUN apt-get install -y openssh-client
RUN apt-get install -y openssh-server
RUN apt-get install -y openmpi-bin
RUN apt-get install -y mpi-default-dev

RUN mkdir root/.ssh
RUN chmod 700 /root/.ssh
COPY ./ssh_keys/id_rsa.pub /root/.ssh/authorized_keys
COPY ./ssh_keys/id_rsa.pub /root/.ssh/id_rsa.pub
COPY ./ssh_keys/id_rsa /root/.ssh/id_rsa
COPY ./ssh_keys/known_hosts /root/.ssh/known_hosts
RUN chmod 600 /root/.ssh/*

RUN apt-get install -y make

COPY ./ssh_keys/ssh_host_ecdsa_key /etc/ssh/ssh_host_rsa_key
COPY ./ssh_keys/ssh_host_ecdsa_key.pub /etc/ssh/ssh_host_rsa_key.pub
RUN chmod 600 /etc/ssh/ssh_host_rsa_key

RUN apt-get install -y sudo

COPY --from=builder /api/main /api/main
COPY --from=builder /api/config /api/config

WORKDIR /api

CMD service ssh start && ./main && tail -f /dev/null
```

D Trabalhos de *benchmarking* relacionados

A Tabela 1 apresenta as ferramentas de *benchmarking* utilizadas por trabalhos relacionados. Os resultados foram adaptados de [37].

Tabela 1: Trabalhos Relacionados

2[4]*Artigo	2[4]*Carga de Trabalho	Recursos					
		P	M	R	D	CV	O
[38]	SPECCPU, LINPACK, RAMSPEED, LMbench, IOzone, Bonnie++, NetIO, WebBench, SysBench e SPECJBB	X	X	X	X		[t]
[39]	LINPACK, STREAM, IO-zone, NetPIPE, NPB e IBS	X	X	X	X		
[40]	UnixBench						X
[41]	PXZ, LINPACK, STREAM, nuttcp, netperf, FIO, Redis e SysBench	X	X	X	X		X
[42]	NPB e TAU			X			X
[43]	autodock3		X				
[44]	CPU-intensive, Sysbench e netperf	X	X	X			X
[45]	HPL e NetPIPE	X		X			X
[46]	JMeter						X
[47]	Y-cruncher, NBENCH, Geekbench, noploop, Linpack, Bonnie++, Sysbench, IOzone, STREAM e netperf	X	X	X	X		
[48]	Swingbench e Sysbench	X	X		X		
[49]	HPL e Graph500		X				X
[50]	HTCondor	X	X	X			
[51]	SPEC CPU 2006, STREAM, FIO, netperf e HiBench	X	X	X	X		
[52]	LINPACK	X		X	X		
[53]	AIO Stress, Ram-speed, IOzone, Tbench, iperf, RuBBoS, ApacheBench, Blake2, 7-zip e OpenSSL benchmark		X	X	X		X
[45]	fs test				X		X
[54]	ApacheBench e Stress Tool			X			X
[55]	Y-cruncher, LINPACK, Geekbench, Bonnie++, Sysbench, STREAM, netperf e iperf	X	X	X	X		
[56]	Iperf, HardInfo, Bonnie++ e STREAM	X	X	X	X		
[57]	LINPACK, STREAM, IO-zone e netperf	X	X	X	X		
[58]	Ferramentas do Docker e Linux	X		X	X		
[59]	Ferramentas do Docker e Linux			X			
[60]	Ferramentas do Docker e Linux				X		[b]