

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME REZENDE ALLES

**Análise da utilização de tecnologias de
contêineres para aplicações de alto
desempenho**

Monografia apresentada como requisito parcial
para a obtenção do grau de Bacharel em Ciência
da Computação

Orientador: Prof. Dr. Alexandre Carissimi

Porto Alegre
2018

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Rui Vicente Oppermann

Vice-Reitora: Prof^a. Jane Fraga Tutikian

Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretora do Instituto de Informática: Prof^a. Carla Maria Dal Sasso Freitas

Coordenador do Curso de Ciência de Computação: Prof. Raul Fernando Weber

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço, primeiramente, aos meus pais e irmão, que deram todo o apoio, oportunidades e motivações necessários para a realização deste e tantos outros trabalhos ao longo da graduação. Obrigado pela compreensão nos momentos de ausência, pelos inúmeros conselhos e ideias trocadas e, principalmente, pelos momentos que passamos juntos.

À minha namorada, pelos conselhos, conhecimentos e experiências. Obrigado por compartilhar comigo todos esses anos de graduação e ser um exemplo de dedicação.

Ao meu professor e orientador, Alexandre Carissimi, pela paciência e dedicação que foram indispensáveis para a realização deste trabalho.

Agradeço também a todos os professores do Instituto de Informática da UFRGS, que contribuíram para a minha formação.

RESUMO

A utilização de técnicas de virtualização baseadas em contêineres tem se tornado uma prática extremamente popular no desenvolvimento e execução de aplicações. Tecnologias de contêineres relativamente novas, como Docker, já são onipresentes em provedores de computação em nuvem e *datacenters* de grandes empresas. Fatores como a abstração do ambiente de execução, isolamento de aplicações em contextos predefinidos e o baixo impacto no desempenho geral contribuem para a adoção de contêineres como plataforma de execução. No contexto de computação de alto desempenho, a virtualização através de contêineres traz benefícios adicionais, como a melhoria na reprodutibilidade de experimentos e a possibilidade de um usuário não *root* configurar o ambiente de execução que será reproduzido em um *cluster*.

Com base nisso, este trabalho tem por objetivo apresentar uma comparação e análise da viabilidade da utilização de duas tecnologias de contêineres diferentes - Docker e Singularity - em ambientes de computação paralela e distribuída. Além de critérios objetivos, relacionados ao desempenho dessas tecnologias, o trabalho apresenta também uma análise subjetiva, que envolve questões como facilidade de uso, implicações de segurança e modelo de virtualização empregado por cada tecnologia.

Palavras-chave: Contêineres. Virtualização. Grid. Cloud. IaaS. Computação paralela. Docker. Singularity. Benchmarks.

Analysis of the utilization of container technologies for high performance computing applications

ABSTRACT

The utilization of virtualization techniques based on containers has become an extremely popular habit in application development and execution. Container technologies that are relatively new, such as Docker, are already ubiquitous in cloud computing providers and big companies' datacenters. Factors such as the abstraction of the execution environment, isolation of application into predefined contexts and the low impact in overall performance contribute for the adoption of containers as a platform of execution. In the context of high performance computing, virtualization through containers brings additional benefits, such as improved reproducibility of experiments and the possibility of a non root user to configure the execution environment that will be reproduced in a cluster.

On these grounds, this work's objective is to present a comparison and viability analysis of the utilization of two different container technologies - Docker and Singularity - in parallel and distributed computing environments. Apart from objective criteria, related to the performance of these technologies, the work also presents a subjective analysis, covering aspects such as ease of use, security implications and virtualization model applied by each technology.

Keywords: Containers. Virtualization. Grid. Cloud. IaaS. Parallel computing. Docker. Singularity. Benchmarks.

LISTA DE FIGURAS

Figura 2.1	Comparação entre tipos de serviços oferecidos por provedores de computação em nuvem	18
Figura 2.2	Comparação entre organizações de hipervisores	21
Figura 3.1	Exemplo de isolamento de aplicações em contêineres	25
Figura 3.2	Isolamento de contêineres Docker, através de <i>namespaces</i>	28
Figura 3.3	Estrutura de um <i>Dockerfile</i>	29
Figura 3.4	Organização de múltiplos <i>Docker Engines</i> em um <i>Docker Swarm</i>	32
Figura 3.5	Arquitetura de um <i>cluster</i> com <i>Docker Swarm</i> e 5 máquinas físicas	33
Figura 3.6	Isolamento de contêineres Singularity, com <i>namespaces</i> compartilhados entre contêiner e SO nativo	34
Figura 3.7	Exemplo de um <i>Singularity recipe file</i>	36
Figura 4.1	Distribuição da Grid5000	41
Figura 4.2	Ambientes de execução utilizados nos experimentos	45
Figura 4.3	Comparação de desempenho entre soluções no <i>benchmark</i> EP	48
Figura 4.4	Comparação de desempenho entre soluções no <i>benchmark</i> Ondes3D	50
Figura 4.5	Comparação de desempenho entre soluções no <i>benchmark</i> Ping-Pong	51

LISTA DE TABELAS

Tabela 3.1	Comparação entre plataformas Docker e Singularity	40
Tabela 4.1	Variações de fatores nos experimentos EP e Ondes3D	46
Tabela 4.2	Tempo de execução na simulação do terremoto SISHUAN	49

LISTA DE ABREVIATURAS E SIGLAS

API	<i>Application Programming Interface</i>
BRGM	<i>Bureau de Recherches Géologiques et Minières</i>
CUDA	<i>Compute Unified Device Architecture</i>
EP	<i>Embarrassingly Parallel</i>
GB	<i>Gigabytes</i>
GNU	<i>GNU is Not UNIX</i>
GPPD	<i>Grupo de Processamento Paralelo e Distribuído</i>
GPU	<i>Graphics Processing Unit</i>
IaaS	<i>Infrastructure as a Service</i>
IP	<i>Internet Protocol</i>
IPC	<i>Inter Process Communication</i>
JVM	<i>Java Virtual Machine</i>
KB	<i>Kilobytes</i>
LXC	<i>Linux Containers</i>
MB	<i>Megabytes</i>
MPI	<i>Message Passing Interface</i>
MPICH	<i>Message Passing Interface over Chameleon</i>
NAS	<i>NASA Advanced Supercomputing</i>
NASA	<i>National Aeronautics and Space Administration</i>
PaaS	<i>Platform as a Service</i>
PID	<i>Process Identifier</i>
PVM	<i>Parallel Virtual Machine</i>
RAM	<i>Random Access Memory</i>
REST	<i>Representational State Transfer</i>

SaaS *Software as a Service*

SLURM *Simple Linux Utility for Resource Management*

SO *Sistema Operacional*

SSH *Secure Shell*

TBB *Threading Building Blocks*

SUMÁRIO

1 INTRODUÇÃO	12
1.1 Objetivos do trabalho	13
1.2 Organização do trabalho	13
2 COMPUTAÇÃO PARALELA E DISTRIBUÍDA	14
2.1 Processamento paralelo	14
2.1.1 Plataformas de processamento paralelo	14
2.1.2 Paradigmas de programação	15
2.2 Clusters e Grids	16
2.3 Computação em nuvem	17
2.4 Virtualização	19
2.4.1 Virtualização em nível de hardware	20
2.4.1.1 Hipervisor de tipo 1	20
2.4.1.2 Hipervisor de tipo 2	21
2.4.1.3 Virtualização completa e Paravirtualização	22
2.4.2 Virtualização em nível de sistema operacional	23
2.4.3 Virtualização em nível de linguagens de programação.....	23
2.5 Resumo	24
3 TECNOLOGIAS DE CONTÊINERES	25
3.1 Contêineres	25
3.2 Docker	27
3.2.1 Virtualização	28
3.2.2 Imagens	28
3.2.3 Gerenciamento de contêineres	30
3.2.4 Execução de contêineres	31
3.2.5 Docker em ambientes distribuídos	31
3.2.6 Arquitetura de um <i>cluster</i> MPI com Docker	32
3.3 Singularity	33
3.3.1 Virtualização	34
3.3.2 Imagens	35
3.3.3 Execução	36
3.3.4 Desenvolvimento com Singularity	37
3.3.5 Arquitetura de um <i>cluster</i> MPI com Singularity	37
3.4 Comparação entre soluções	38
3.4.1 Imagem de contêiner	38
3.4.2 Reprodutibilidade.....	38
3.4.3 Instalação em ambientes compartilhados.....	39
3.4.4 Compatibilidade e portabilidade	39
3.4.5 Tabela comparativa	40
3.5 Resumo	40
4 AVALIAÇÃO DE DESEMPENHO	41
4.1 Grid 5000	41
4.2 Benchmarks	42
4.2.1 EP NAS Parallel Benchmark	43
4.2.2 Ondes3D	43
4.2.3 Ping-Pong.....	44
4.2.4 Versões de software utilizadas	44
4.3 Metodologia	45

4.4 Resultados.....	47
4.4.1 NAS EP	47
4.4.2 Ondes3D	49
4.4.3 Ping-Pong.....	51
4.5 Resumo.....	52
5 CONCLUSÕES	54
5.1 Dificuldades encontradas	55
5.2 Trabalhos Futuros.....	56
REFERÊNCIAS.....	57

1 INTRODUÇÃO

A utilização de tecnologias de virtualização para ambientes de desenvolvimento e execução de software vem passando por uma grande ascensão. As vantagens que um ambiente virtualizado oferece, como a facilidade na migração de aplicações entre diferentes conjuntos de hardware, são bastante visadas na indústria de software. Outro fator que contribui para a popularidade da virtualização é a sua utilização, em larga escala, em provedores de computação em nuvem.

No caso da computação em nuvem, a virtualização oferece ainda a vantagem de isolar completamente os sistemas operacionais, fazendo com que múltiplos servidores virtuais possam ser executados em uma única máquina física. Essa abordagem possibilita uma redução nos custos de operação e manutenção - justificados pela facilitação na gestão da infraestrutura, menor número de máquinas físicas e economia de energia - e um melhor aproveitamento do hardware existente. Com a redução de custos, a virtualização permite que serviços de computação em nuvem sejam oferecidos a usuários finais por um preço baixo. Esse fator contribui significativamente para a popularidade de serviços de computação em nuvem.

Por outro lado, os benefícios trazidos pela virtualização não são muito utilizados na execução de aplicações de alto desempenho, como simulações e experimentos científicos. O fator que mais contribui para isso é o sobrecusto envolvido na utilização de máquinas virtuais tradicionais, o que acaba prejudicando o desempenho das aplicações. Outro ponto negativo, inerente ao uso de máquinas virtuais, é a dificuldade em resolver problemas de compatibilidade com hardware específico (comumente utilizado em computação de alto desempenho) como placas aceleradoras e adaptadores de rede de alta velocidade.

Como uma alternativa ao uso de máquinas virtuais, algumas abordagens de virtualização em nível de sistema operacional têm se tornado cada vez mais populares. No caso de sistemas operacionais GNU/Linux, esse tipo de virtualização é chamado de contêineres. O principal diferencial na utilização de contêineres, quando comparados às máquinas virtuais, é o baixo sobrecusto observado na execução de aplicações virtualizadas. Com base nisso, a utilização de contêineres passa a ser uma possível alternativa para trazer os benefícios da virtualização para ambientes de computação de alto desempenho.

1.1 Objetivos do trabalho

O objetivo deste trabalho é estudar a viabilidade da execução de aplicações de alto desempenho utilizando métodos de virtualização em nível de sistema operacional (contêineres). Para isso, serão comparadas duas soluções de contêineres, Docker e Singularity, quanto à sua viabilidade na execução de aplicações paralelas.

Os dois principais critérios para este estudo são o tempo de execução de aplicações e a facilidade de instalação, configuração e utilização das ferramentas de virtualização. Para fins de comparação, a execução em um ambiente nativo (não virtualizado) também será considerada.

1.2 Organização do trabalho

Este trabalho é composto por 4 capítulos, além desta introdução. No capítulo 2, são apresentados alguns conceitos básicos de computação paralela e distribuída, que são fundamentais para a contextualização do trabalho. O capítulo 3 discute as tecnologias de contêineres - Docker e Singularity - que serão comparadas uma à outra. O capítulo 4 fornece a descrição dos experimentos realizados e a apresentação dos resultados. Por fim, as conclusões são apresentadas no capítulo 5.

2 COMPUTAÇÃO PARALELA E DISTRIBUÍDA

O objetivo deste capítulo é apresentar conceitos básicos de computação paralela e distribuída, bem como tecnologias utilizadas nesse ramo. Serão apresentados dois dos paradigmas de programação paralela mais populares e as plataformas correspondentes a eles. Será fornecida também uma breve introdução a *clusters* de computadores, *grids* e computação em nuvem. Por fim serão apresentados conceitos de virtualização, fundamentais para a compreensão do funcionamento de serviços de computação em nuvem.

2.1 Processamento paralelo

Um computador paralelo é definido por um conjunto de processadores que são capazes de trabalhar de maneira colaborativa para resolver um problema computacional (FOSTER, 1995). Processamento paralelo, portanto, consiste em utilizar os processadores disponibilizados pelo hardware para aumentar o número total de instruções executadas em um dado intervalo de tempo. Para isso, é necessário que o programador esteja ciente dos recursos disponíveis, e aplique esse conhecimento para ditar como uma aplicação deve separar os conjuntos de instruções para que eles sejam executados por diferentes processadores. A isso, dá-se o nome de programação paralela.

2.1.1 Plataformas de processamento paralelo

A programação paralela consiste na divisão de um problema em partes menores que podem ser resolvidos em paralelo, usando múltiplas unidades de execução. A forma como as unidades de execução são organizadas, e a variedade de processadores disponíveis, dão origem às diferentes plataformas de processamento paralelo. Dentre essas, destacam-se as máquinas *multicore*, as máquinas híbridas e os *clusters* de computadores.

Máquinas *multicore* são computadores que possuem múltiplos *cores* (ou núcleos) de processamento em um ou mais processadores. A memória nessas máquinas é compartilhada, e as instruções são distribuídas entre os *cores* do(s) processador(es).

Máquinas híbridas são computadores - potencialmente *multicore* - que contam com algum outro tipo de unidade de execução capaz de executar instruções. A variação mais comum de máquina híbrida é a combinação de um (ou mais) processadores com uma

(ou mais) GPUs. No modelo de máquinas híbridas a memória pode ser compartilhada ou não, e os problemas são distribuídos entre as diversas unidades de execução disponíveis.

Um *cluster* de computadores, também chamado de agregado, é um conjunto de máquinas (*multicore* e/ou híbridas) interconectadas por uma rede. Cada máquina em um *cluster* é chamada de nodo, ou nó. A memória em um único nodo continua sendo compartilhada entre seus processadores ou *cores*. Entretanto, ao conectar vários nodos a uma mesma rede, o conceito de memória “global” se perde. Por conta disso, a troca de informações entre os nodos de um *cluster* deve ser feita via troca de mensagens ou através do emprego de um sistema baseado em um modelo de memória distribuída.

2.1.2 Paradigmas de programação

Para que uma aplicação paralela possa se beneficiar dos recursos oferecidos pela plataforma de hardware, é necessário que ela seja programada para esse fim. Existem, portanto, paradigmas de programação paralela que são comumente aplicados no desenvolvimento dessas aplicações. Esses paradigmas são fortemente dependentes das características da plataforma e são divididos em dois grupos.

Paradigma de memória compartilhada: descreve modelos de programação para máquinas *multicore* ou híbridas, que possuem algum tipo de memória global acessível a todas as unidades de processamento. Geralmente, a memória compartilhada é a própria memória do sistema (RAM), que é acessível a todos os *cores* de um processador. Outros exemplos de memória compartilhada são memórias *cache* e a memória de vídeo em uma GPU. Como todas as plataformas possuem suporte a algum tipo de memória compartilhada, esse paradigma é bastante abrangente quanto ao número de ferramentas de programação disponíveis. Alguns exemplos de ferramentas que permitem programação utilizando o paradigma de memória compartilhada são OpenMP (DAGUM; MENON, 1998), CUDA (NICKOLLS et al., 2008), OpenCL (STONE; GOHARA; SHI, 2010), TBB (PHEATT, 2008) e Pthreads (BUTENHOF, 1997).

Paradigma de troca de mensagens: descreve modelos de programação para plataformas que não possuem memória compartilhada, como *clusters* de computadores. Dessa forma, a maneira de se transmitir informações entre unidades de execução é através de troca de mensagens. Nesse paradigma, unidades de execução enviam mensagens umas às outras através de uma API de comunicação. Exemplos de ferramentas para programação nesse paradigma são MPI (FORUM, 1994) e PVM (GEIST et al., 1994).

2.2 Clusters e Grids

Conforme já mencionado, um *cluster* é formado por um conjunto de computadores interconectados por uma rede local. Os *clusters* são bastante populares em computação paralela, pois permitem agrupar um grande número de computadores para executarem, em conjunto, uma aplicação paralela. Por conta da organização geográfica próxima, e do número de computadores em um *cluster* (geralmente na ordem de grandeza de dezenas ou algumas centenas de nodos), é comum que os nodos sejam interligados por redes de alta velocidade e baixa latência.

Uma organização popular de *clusters* é a arquitetura Beowulf (BOUKERCHE; AL-SHAIKH; NOTARE, 2007), onde o conjunto de nodos compartilha um sistema de arquivos único, fornecido por um nodo da rede. Em um *cluster* com essa arquitetura, um dos nodos é chamado de *front-end* e é responsável pela interface de acesso do usuário com o *cluster*. Esse nodo fica conectado a duas redes: a rede externa, que pode ser tanto a Internet quanto uma rede local, através da qual o usuário se conecta ao *front-end*; e a rede interna, que conecta o *front-end* aos demais nodos, denominados nodos de cálculo. Estes últimos não possuem nenhum tipo de conexão com redes externas, e são utilizados apenas para executar as tarefas enviadas pelo usuário através do *front-end*.

Apesar da organização em *cluster* fornecer a vantagem de agrupar uma grande quantidade de processadores em um espaço relativamente pequeno, para aplicações de escala maior, o poder computacional oferecido por um único *cluster* pode não ser suficiente. Além disso, aumentar o poder computacional do *cluster*, adicionando mais nodos de cálculo, nem sempre é viável economicamente.

Uma opção para o problema de escala em um *cluster* é o uso de uma *grid* (grade, em português). A organização de computadores em *grid* consiste na união de diversos *clusters*, pertencentes a organizações diferentes, com o objetivo de fornecer poder computacional como um serviço público, similar à energia elétrica (o nome *grid* é derivado da nomenclatura dada a redes elétricas, *power grids*). Os *clusters* que compõem uma *grid* podem estar geograficamente distantes uns dos outros (espalhados ao redor de um país, por exemplo), e são geralmente conectados através da Internet (PARDESHI; PATIL; DHUMALE, 2013).

A implementação de uma *grid*, entretanto, também possui empecilhos práticos. É natural, por exemplo, que cada organização possua suas próprias políticas de segurança da informação e acesso aos recursos computacionais. Como existem diversos *clusters* (e

diversas organizações) envolvidos, essas políticas podem ser potencialmente divergentes. A conciliação dos interesses de todas as partes, de forma a fornecer um serviço uniforme para todos os usuários, nem sempre é uma tarefa trivial. Adicionalmente, um conjunto de hardware em larga escala, como é o caso em uma *grid*, possui altos custos de manutenção. Além da substituição de hardware em casos de falha, há a necessidade de manter os equipamentos atualizados para garantir alto desempenho e ter uma equipe dedicada para a administração da infraestrutura.

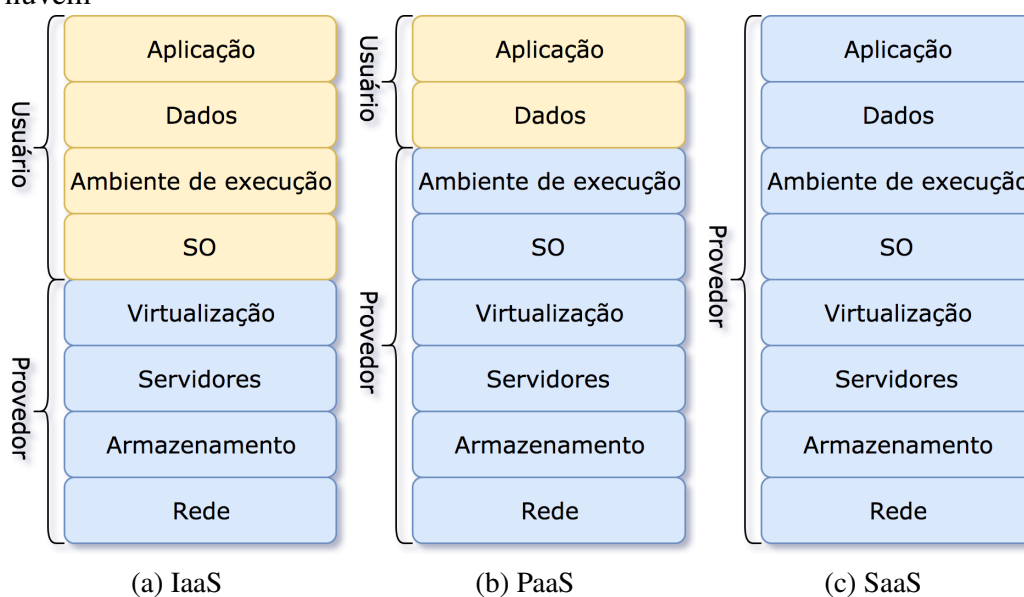
Uma outra alternativa para o problema de escala em *clusters* é a utilização de infraestruturas de computação em nuvem. Essa alternativa resolve o problema da gerência distribuída em uma *grid*, além de também oferecer poder computacional como um serviço para seus usuários.

2.3 Computação em nuvem

A computação em nuvem nada mais é do que um modelo de negócio que permite a usuários o acesso a recursos computacionais como processamento, armazenamento, comunicação de rede, serviços e aplicações através da Internet (MELL; GRANCE, 2011). Diferentemente de *grids*, um provedor de computação em nuvem conta com um modelo de gerência centralizado, onde uma única organização é dona de toda a infraestrutura, o que elimina a divergência entre diferentes políticas de segurança da informação. Essa infraestrutura pode ser utilizada sob demanda, o que faz com que o poder computacional em questão seja visto não mais como um produto, mas sim como um serviço. Essa abordagem, possibilitada por conta do uso de tecnologias de virtualização, permite um modelo de pagamento *pay-as-you-go*, onde o usuário é cobrado apenas pelos recursos utilizados. É desse modelo de negócio que surgem os conceitos típicos de computação em nuvem IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*) e SaaS (*Software as a Service*).

IaaS é um modelo que oferece infraestrutura de hardware para aplicações baseadas na computação em nuvem. Os serviços oferecidos por um provedor desse tipo são recursos físicos de rede, armazenamento e processamento (servidores), conforme ilustrado na Figura 2.1a. Através da camada de virtualização, é permitido ao usuário aumentar a quantidade de recursos contratados em situações de alta demanda, e diminuir a quantidade de recursos quando a demanda for baixa. A isso, dá-se o nome de elasticidade (ELASTICITY, 2017). Um exemplo de serviço oferecido como IaaS é o Elastic Compute Cloud

Figura 2.1: Comparação entre tipos de serviços oferecidos por provedores de computação em nuvem



Fonte: Adaptado de (HOSTINGADVICE, 2017).

(EC2, 2017), da Amazon, que permite a contratação de máquinas virtuais e realiza a cobrança por hora de utilização.

PaaS é o serviço que oferece plataformas para desenvolvimento, instalação e execução de aplicações através da computação em nuvem. Conforme ilustrado na Figura 2.1b, um provedor de serviços PaaS também fornece infraestrutura de hardware, mas em uma plataforma que já vem com sistema operacional, bibliotecas, ambientes de desenvolvimento e execução de aplicação, que permite aos usuários abstrair os detalhes de baixo nível. Exemplos de serviços que podem ser fornecidos no modelo PaaS são ferramentas de gerência de bancos de dados, servidores de *builds* automatizados e serviços de versionamento de código. Um exemplo de serviço oferecido como PaaS é o Relational Database Service (RDS, 2017), da Amazon, utilizado para instanciar e gerenciar bancos de dados relacionais.

SaaS é um modelo de distribuição de software baseado em serviços. Na Figura 2.1c, é possível perceber que o provedor é responsável por todos os aspectos da aplicação, desde a infraestrutura de hardware até a gerência de dados e manutenção do software. Nesse modelo, o usuário final acessa aplicações disponíveis pelos recursos da computação em nuvem através da Internet. Os exemplos mais populares de SaaS são as ferramentas da Google para email (Gmail), processamento de texto (Google Docs) e armazenamento em nuvem (Google Drive).

Ainda, os provedores de serviços de computação em nuvem podem ser divididos

em dois tipos básicos: público ou privado. Um provedor público permite que um usuário qualquer aloque recursos para suprir suas necessidades, e está disponível para o público em geral. Os recursos são pertencentes a uma infraestrutura de terceiros, como a Amazon, Microsoft ou Google, e o usuário paga apenas pelos recursos utilizados de acordo com a política de cada provedor. Exemplo de provedores públicos são a Amazon Web Services, Microsoft Azure e Google Cloud Platform.

Um provedor privado existe para suprir a necessidade de um único usuário ou organização. Esse modelo passa a ser relevante quando o usuário não pode fornecer os dados de suas aplicações a uma infraestrutura de terceiros. Exemplo desse caso de uso são organizações que lidam com dados médicos ou informações sigilosas. Assim, é necessário que a organização adquira e gerencie sua própria infraestrutura, evitando o vazamento de dados para uma organização externa através do controle de acesso de usuários.

Na realidade, ainda há dois outros tipos de nuvem: comunitária e híbrida. Uma nuvem comunitária refere-se ao caso onde diversas organizações com um objetivo específico compartilham a infraestrutura, e os custos são divididos entre os membros. Uma nuvem híbrida, por outro lado, é a união de duas ou mais nuvens de tipos quaisquer, conectadas de forma a permitir a troca de informações entre elas.

A gerência de uma infraestrutura de nuvem é feita através de uma ferramenta especializada. Apesar de os grandes provedores de nuvem pública (como Amazon Web Services e Google Cloud Platform) possuírem soluções proprietárias, existem também alternativas de código aberto com a mesma finalidade. Exemplo dessas são o OpenStack (OPENSTACK, 2017), CloudStack (CLOUDSTACK, 2017) e Eucalyptus (EUCALYPTUS, 2017).

Para fornecer os serviços de computação em nuvem, tanto pública quanto privada, de forma escalável, e minimizando o custo de aquisição de infraestrutura, a tecnologia central é a virtualização, que será explicada a seguir.

2.4 Virtualização

A virtualização tem por objetivo emular recursos, geralmente de hardware, através de uma camada de software, tornando possível a execução de múltiplos ambientes completos (sistemas operacionais com memória, processador, armazenamento, etc) aparentemente isolados em um único hardware físico. Essa característica é muito importante em provedores de computação em nuvem, pois permite que diversos usuários tenham a per-

cepção de controlar um recurso isolado (como um servidor) apesar de estarem dividindo a mesma máquina física. Ao alocar diversos usuários em uma única máquina física, a virtualização permite o uso mais eficiente do hardware disponível, reduzindo gastos com a aquisição de novas máquinas, energia elétrica, manutenção, controle de temperatura, entre outros.

Para atingir esse objetivo, tecnologias de virtualização abstraem as características físicas de um recurso computacional de forma a tornar o acesso aos componentes de hardware mais uniforme do ponto de vista de sistemas, aplicações e usuários (CARISSIMI, 2009). Existem 3 níveis básicos de virtualização: em nível de hardware, em nível de sistema operacional e em nível de linguagem de programação.

2.4.1 Virtualização em nível de hardware

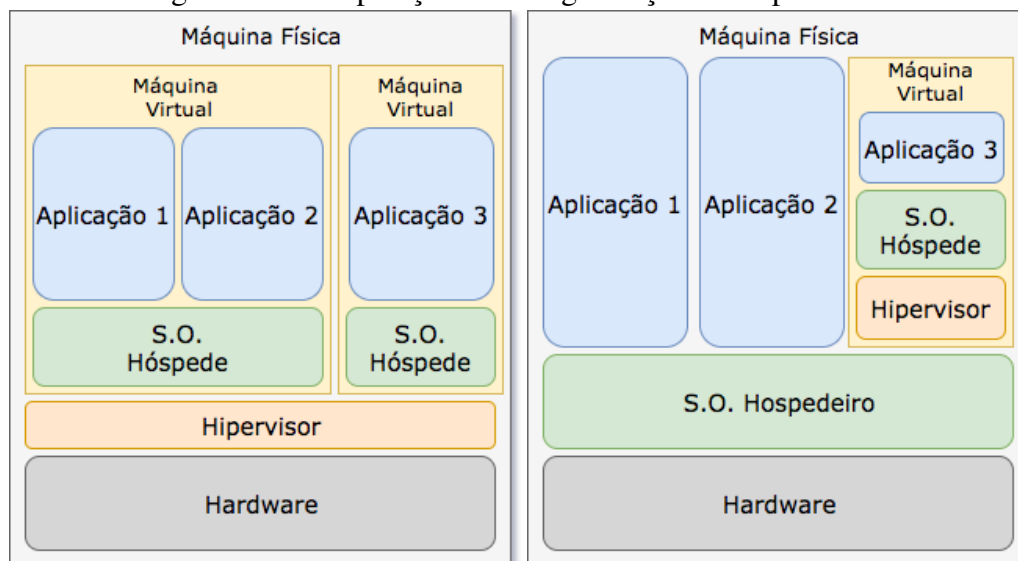
A principal característica da virtualização em nível de hardware é que ela fornece uma abstração completa do sistema físico onde sistemas operacionais executam. Para isso, é necessário adicionar uma camada de software ao sistema operacional, capaz de detectar e gerenciar os recursos de hardware disponíveis. Essa camada de software é chamada de Monitor de Máquina Virtual, ou hipervisor (*hypervisor*). O hipervisor é o responsável por providenciar meios de acesso ao hardware físico para os sistemas virtualizados, de forma a garantir o isolamento entre eles.

Os hipervisores são divididos em dois tipos distintos, e são diferenciados pela maneira como são implementados. Eles são chamados hipervisores de tipo 1 e tipo 2, e estão ilustrados na Figura 2.2.

2.4.1.1 Hipervisor de tipo 1

A Figura 2.2a mostra um hipervisor de tipo 1, também chamado de nativo. Esse tipo de hipervisor é uma camada de software que fica imediatamente acima do hardware físico de uma máquina. O objetivo principal desse tipo de hipervisor é, portanto, gerenciar e compartilhar os recursos físicos de uma máquina entre todos os sistemas que executam acima dele, dando a impressão de que todos os sistemas virtualizados executam em uma máquina não compartilhada (CARISSIMI, 2009). Como um hipervisor desse tipo tem acesso direto a diversos aspectos do hardware como a gerência de processador e memória, o sobrecurso na execução de máquinas virtuais é consideravelmente menor (quando

Figura 2.2: Comparação entre organizações de hipervisores



(a) Organização de um hipervisor de tipo 1 (b) Organização de um hipervisor de tipo 2

Fonte: Autor.

comparado ao hipervisor de tipo 2). Entretanto, os sistemas operacionais precisam ser projetados com o conhecimento da existência do hipervisor (pois é com ele que o sistema operacional interage). Isso faz com que a compatibilidade com hipervisores de tipo 1 seja limitada a alguns sistemas operacionais. Exemplos de hipervisores do tipo 1 são o Microsoft Hyper-V (HYPER-V, 2017), o Xen (XEN, 2017) e o VMware ESXi (VMWARE, 2017).

2.4.1.2 Hipervisor de tipo 2

Na Figura 2.2b é possível ver a organização de um hipervisor de tipo 2. Esses hipervisores, também chamados de sistemas hóspedes, são caracterizados por executar sobre um sistema operacional hospedeiro. Dessa forma, o hipervisor nada mais é do que um processo no sistema operacional nativo, que interage com o hardware através das chamadas de sistema disponibilizadas pelo sistema operacional. Esse tipo de hipervisor apresenta diversas camadas de abstração para o hardware, de forma que o sistema hospede não precisa ter conhecimento da existência do hospedeiro. Por conta disso, a compatibilidade com os sistemas operacionais existentes atualmente é bastante alta. Como um hipervisor do tipo 2 precisa interagir diretamente com o sistema operacional hospedeiro para executar instruções, há um sobrecusto adicional na execução de máquinas virtuais. Um exemplo de hipervisor de tipo 2 é o VirtualBox (VIRTUALBOX, 2017).

2.4.1.3 Virtualização completa e Paravirtualização

Um dos empecilhos para a implementação de máquinas virtuais é garantir que o compartilhamento de recursos físicos não permita que dois ambientes virtualizados interfiram um no outro. Do ponto de vista do processador, isso significa definir como instruções privilegiadas serão executadas por um sistema hóspede, já que essas instruções, por questões de segurança, são restritas ao sistema hospedeiro. Em (POPEK; GOLDBERG, 1974 apud CARISSIMI, 2009), os autores concluíram que qualquer instrução que afete o comportamento do sistema deve ser interceptada pelo hipervisor e tratada adequadamente. Existem duas estratégias para a identificação e tratamento dessas instruções: virtualização completa e paravirtualização.

A técnica de virtualização completa consiste na tradução binária de instruções emitidas pelo sistema hóspede. Nessa abordagem, todas as instruções executadas pelo sistema hóspede são interceptadas e testadas pelo hipervisor, de forma a decidir se cada instrução pode ser executada diretamente no processador (no caso de instruções não privilegiadas) ou se é necessário tratamento adicional por parte do hipervisor (no caso de instruções privilegiadas ou instruções não privilegiadas e sensíveis). Essa técnica não requer nenhum tipo de modificação no sistema operacional hóspede, e o mesmo não precisa ter conhecimento da existência de um hipervisor. Entretanto, como é necessário testar toda e qualquer instrução emitida pela máquina virtual, essa abordagem causa um sobre-custo alto na execução de aplicações virtualizadas.

Como tentativa de reduzir o sobre-custo da virtualização completa, existe também a estratégia da paravirtualização. Esse tipo de virtualização considera que sistemas operacionais hóspedes têm conhecimento da existência do hipervisor e comunicam-se diretamente com ele. Assim, chamadas ao *kernel* do sistema operacional hóspede que executam instruções privilegiadas no processador são substituídas por uma chamada para o hipervisor, que deverá interceptar essa instrução para fazer o devido tratamento. Essa abordagem apresenta um sobre-custo consideravelmente menor quando comparada à virtualização completa, tendo em vista que o hipervisor só precisará interceptar instruções sensíveis, ou que executem em modo privilegiado.

Apesar de apresentar impacto menor no desempenho, a paravirtualização requer modificações no *kernel* do sistema operacional hóspede. Essa necessidade de modificação faz com que essa técnica não seja uma solução compatível com todos os sistemas operacionais. Uma outra abordagem, que também busca diminuir o sobre-custo da virtualização, é a utilização de extensões que dão suporte de hardware para a virtualização (LEE, 2014).

Essas extensões são implementadas nas microarquiteturas de processadores Intel e AMD, e têm por objetivo resolver os problemas encontrados na virtualização completa, como a execução de instruções privilegiadas pelo sistema operacional hóspede.

2.4.2 Virtualização em nível de sistema operacional

A virtualização em nível de sistema operacional tem como objetivo principal o isolamento de recursos do sistema para as aplicações. Os recursos que podem ser virtualizados são definidos pela implementação do *kernel* do sistema operacional, mas geralmente envolvem itens como árvores de processo, usuários, interfaces de rede e sistemas de arquivos. Dessa forma, uma aplicação que executa em um ambiente com esse tipo de virtualização tem a impressão de estar executando em uma máquina dedicada, e não tem conhecimento dos recursos compartilhados. É importante notar que, nesse tipo de virtualização, todas as aplicações compartilham o mesmo *kernel* de sistema operacional. Essa característica traz vantagens e desvantagens: por um lado, não é possível que diferentes *kernels* existam e executem ao mesmo tempo. Por outro, como há um único sistema em execução, a figura do hipervisor passa a ser desnecessária, fazendo com que o impacto desse tipo de virtualização no desempenho de uma aplicação seja mínimo.

Diversos sistemas operacionais possuem ferramentas que dão suporte à virtualização em nível de sistema operacional. Alguns exemplos são as *jails* do FreeBSD (FREEBSD, 2017) e *zones* em sistemas Solaris (SOLARIS, 2017). Em sistemas que executam o *kernel* Linux, esse tipo de virtualização é chamado de contêineres, e serão discutidas no capítulo 3.

2.4.3 Virtualização em nível de linguagens de programação

A virtualização em nível de linguagens de programação existe como forma de melhorar a portabilidade de aplicações em diferentes ambientes de execução. Ao introduzir uma máquina virtual para a execução de aplicações, elimina-se a necessidade de o programador compilar a sua aplicação para diferentes ambientes e arquiteturas. Do ponto de vista do desenvolvedor, é necessário apenas compilar a aplicação para que ela possa ser executada na máquina virtual. Naturalmente, esse tipo de virtualização requer a existência de uma máquina virtual compatível com a arquitetura de hardware na qual se deseja

executar a aplicação.

Um dos exemplos mais comuns de virtualização em nível de linguagem de programação é a *Java Virtual Machine* (JVM), usada para a execução de programas escritos na linguagem de alto nível Java. A JVM é capaz de interpretar seu próprio conjunto de códigos binários (*bytecode*), e aplicações Java são compiladas para esses códigos. Assim, a JVM interpreta os *bytecodes* da aplicação e executa as instruções equivalentes no hardware real (CARISSIMI, 2009). Apesar de melhorar consideravelmente o problema de portabilidade de aplicações, a virtualização em nível de linguagens de programação introduz uma nova camada de software no ambiente de execução. A inserção de uma máquina virtual para executar aplicações gera, portanto, um impacto negativo no desempenho. Essa é uma das principais desvantagens desse tipo de virtualização.

2.5 Resumo

Neste capítulo, foram apresentados alguns conceitos básicos de computação paralela, de computação em nuvem e de virtualização. A computação paralela é um recurso fundamental para a solução de problemas computacionais complexos, que muitas vezes não podem ser resolvidos em tempo hábil utilizando uma abordagem sequencial. Existem diversas plataformas que permitem processamento paralelo, entre elas: máquinas *multicore*, máquinas híbridas e *clusters* de computadores. Máquinas *multicore* e híbridas são geralmente programadas em um paradigma de memória compartilhada, enquanto *clusters* de computadores são geralmente programados utilizando trocas de mensagens. Há, ainda, outras plataformas mais complexas, como *grids*, que combinam o poder computacional de *clusters* de diversas organizações para alcançar um objetivo comum.

A computação em nuvem é um modelo de negócio que fornece poder computacional como serviço, de forma escalável e sob demanda, de forma que o usuário é cobrado de maneira proporcional aos recursos utilizados. Entre todas as tecnologias utilizadas em uma infraestrutura de nuvem, o maior alicerce é a virtualização, que permite a alocação e liberação de recursos conforme o necessário, promovendo um uso mais eficiente do hardware.

No próximo capítulo, serão apresentados detalhes de duas tecnologias de virtualização em nível de sistema operacional para ambientes Linux, chamadas Docker e Singularity. O desempenho dessas duas tecnologias será analisado no capítulo 4.

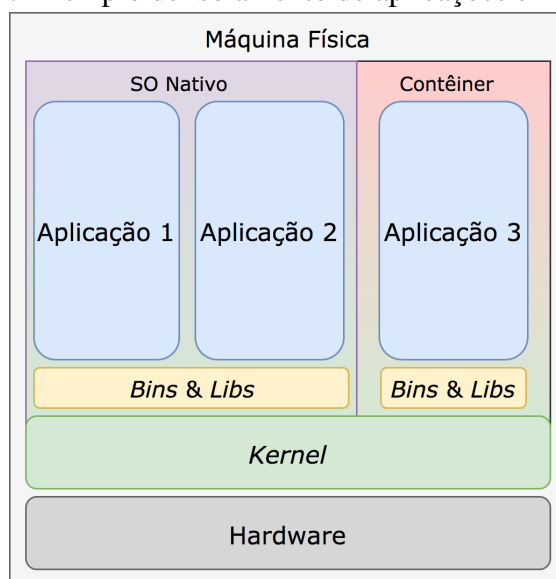
3 TECNOLOGIAS DE CONTÊINERES

Este capítulo tem por objetivo apresentar conceitos de virtualização em nível de sistema operacional para ambientes baseados em Linux. Além desses conceitos, serão apresentados detalhes de duas soluções de contêineres que foram usadas para a análise de desempenho deste trabalho: Docker e Singularity. Para cada uma das tecnologias, será apresentada uma solução que simula a arquitetura de um *cluster*, utilizando as ferramentas fornecidas pelas plataformas, para executar aplicações paralelas. Será apresentada também uma breve comparação entre essas duas tecnologias.

3.1 Contêineres

Contêineres são uma estratégia de virtualização em nível de sistema operacional para ambientes Linux. Essa técnica é baseada no isolamento de recursos específicos do *kernel* para que aplicações tenham a impressão de estarem executando em um ambiente isolado. Um esquema básico do funcionamento de contêineres é ilustrado na Figura 3.1. Um aspecto importante, mostrado nessa figura, é que o *kernel* do sistema operacional é compartilhado entre aplicações e contêineres. Entretanto, a forma de acesso ao *kernel* (do ponto de vista das aplicações) é diferente, pois o conjunto de aplicações (*Bins*) e bibliotecas (*Libs*) do sistema não é necessariamente o mesmo oferecido pelo sistema operacional nativo.

Figura 3.1: Exemplo de isolamento de aplicações em contêineres



Fonte: Adaptado de (DOCKER, 2017a)

Do ponto de vista da implementação, existem três ferramentas essenciais em um *kernel* Linux que permitem a criação e utilização de contêineres. São elas: *chroot*, *cgroups* e *namespaces*.

Chroot (*Change Root*) foi o primeiro passo para o isolamento de processos no mundo UNIX. Introduzido em 1979, o *chroot* é um comando utilizado para executar uma aplicação, ou *shell*, especificando um diretório *root* diferente do padrão. Dessa forma, a aplicação executada com *chroot* não terá acesso a qualquer diretório fora da árvore de diretórios especificada pelo comando (CHROOT, 2017), dando a impressão que o processo em questão está executando em um ambiente isolado. Posteriormente, a funcionalidade do *chroot* foi incorporada pelo recurso de *namespaces*. Por conta disso, as soluções de contêineres estudadas neste trabalho não fazem uso direto dessa ferramenta.

Cgroups (*Control Groups*) é uma funcionalidade do *kernel* utilizada para controlar recursos do sistema como tempo de processador, memória e recursos de rede. Através de *cgroups*, administradores de sistema podem controlar e consultar a utilização de recursos que são divididos entre processos de usuários (CGROUPS, 2017). Essa funcionalidade é importante para permitir que sejam estabelecidos limites de utilização dos recursos em contêineres, quando necessário.

Namespaces são abstrações que encapsulam recursos do sistema, de maneira a fornecer camadas de isolamento entre processos. Dessa forma, processos pertencentes ao *namespace* de um recurso têm a visão de que possuem uma instância isolada do mesmo. Mudanças nesse recurso são visíveis apenas aos processos que compartilham o mesmo *namespace*, mas são invisíveis a outros processos (NAMESPACES, 2017). Existem diversos *namespaces* no *kernel* do Linux, sendo alguns deles: *namespace* de rede, usado para que cada contêiner possa ter a sua própria pilha de rede; *namespace* de IPC (*inter process communication*), para que processos que utilizam esses recursos não interfiram uns nos outros; *namespace* de *mount*, usado para garantir o isolamento de sistemas de arquivos; *namespace* de PID (*Process Identifier*), usado para isolar a árvore de processos dentro de um contêiner; e *namespace* de usuário, que permite que um usuário seja *root* em um contêiner sem necessariamente ter acesso *root* fora dele.

As ferramentas acima são suficientes para a criação de ambientes virtualizados dentro de um sistema operacional GNU/Linux. Entretanto, a tarefa de gerenciar recursos do *kernel* através dessas ferramentas, manualmente, é bastante complexa e requer conhecimentos avançados sobre o funcionamento do sistema operacional. Essa complexidade fez com que esse tipo de virtualização fosse muito pouco utilizado até recentemente. A

popularização de contêineres se deu com o surgimento de ferramentas que simplificam a API disponibilizada pelo *kernel* Linux, de forma a tornar a criação de ambientes isolados uma tarefa mais simples para o usuário final.

Essa simplicidade vem, em grande parte, graças à abstração de um ambiente de execução em imagens de contêineres. Uma imagem, nesse caso, é a representação de um ambiente virtualizado em um formato que possa ser salvo em algum meio de armazenamento, para que esse mesmo ambiente possa ser reproduzido múltiplas vezes. Um contêiner, nesse sentido, nada mais é do que uma imagem em execução. A relação entre imagem e contêiner pode ser comparada à relação entre arquivo executável e processo: enquanto o primeiro é uma representação em disco, o segundo é definido pela execução dessa representação no processador.

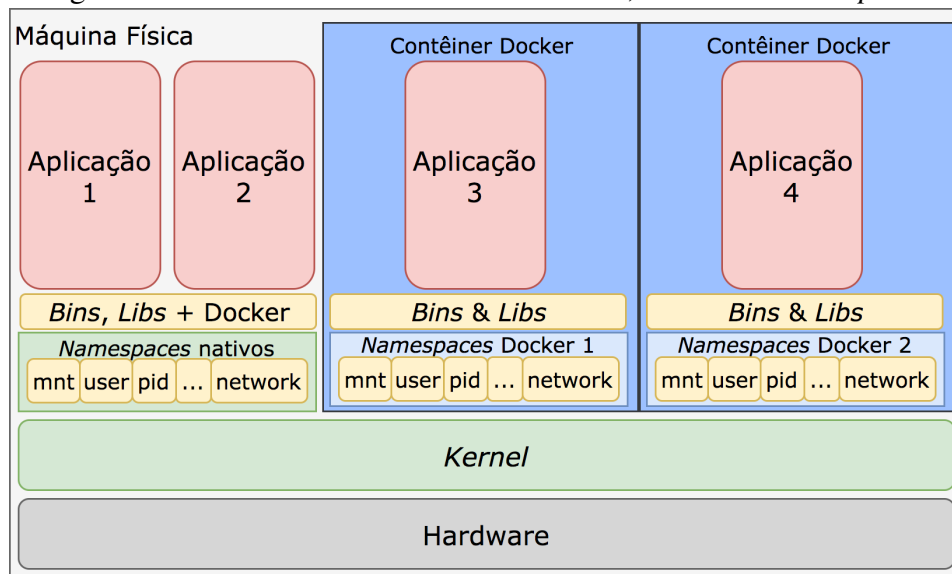
Ainda, com relação a imagens de contêineres, é importante ressaltar que a abstração do ambiente de execução não constitui um sistema operacional completo. Uma imagem de contêiner possui suas próprias bibliotecas e programas de sistema, mas não contém o *kernel* Linux. Esse último é fornecido pelo sistema operacional nativo, e é compartilhado entre os processos nativos e todos os contêineres em execução.

Nas seções seguintes, serão apresentadas as duas tecnologias de contêineres que serão utilizadas na análise experimental deste trabalho.

3.2 Docker

O Docker (DOCKER, 2017b) é uma plataforma de código aberto usada para o desenvolvimento, distribuição e execução de aplicações em ambientes com sistema operacional GNU/Linux. Através da instalação dessa plataforma no sistema operacional nativo, é possível criar contêineres que encapsulam as aplicações em ambientes completamente isolados, conforme mostra a Figura 3.2. O processo de criação de um contêiner, nesse caso, envolve a interação do usuário com a plataforma Docker, instalada no sistema operacional nativo. Essa plataforma, por sua vez, interage com o *kernel* do sistema para a criação dos *namespaces* que serão utilizados para o isolamento do contêiner.

Figura 3.2: Isolamento de contêineres Docker, através de *namespaces*.



Fonte: Autor.

3.2.1 Virtualização

O objetivo de um sistema como o Docker é possibilitar virtualização em ambientes que executam o *kernel* Linux, sem o sobrecusto adicional de uma máquina virtual. Isso é feito através da virtualização em nível de sistema operacional, discutida na seção 2.4.2. O estilo de virtualização do Docker, conforme explicitado na Figura 3.2, consiste em utilizar *namespaces* para isolar todos os serviços possíveis, fazendo com que o usuário de um contêiner Docker tenha a percepção de estar executando em uma máquina controlada por ele próprio, com privilégios de administrador e controle total sobre os recursos virtualizados.

3.2.2 Imagens

Uma imagem é, basicamente, uma cópia completa de um ambiente de execução (excluindo o *kernel* do sistema operacional). Uma imagem Docker armazena todo o conteúdo de um contêiner e é, portanto, a única fonte de informação necessária para inicializar uma aplicação virtualizada. Uma das funcionalidades responsáveis pela grande popularidade do Docker é a facilidade no processo de criação e compartilhamento de imagens. Elas podem ser construídas de duas formas: criando um ambiente de execução completamente novo ou referenciando uma imagem base (*base image*) já existente.

Para criar uma imagem completamente nova, é necessário compactar o sistema

rios têm acesso a imagens que variam desde distribuições Linux até aplicações completas como MySQL (MYSQL, 2017), RabbitMQ (RABBITMQ, 2017) e Nginx (NGINX, 2017).

Um último aspecto importante sobre imagens Docker é a estrutura física das mesmas. No Docker, uma imagem é representada como um conjunto de camadas (também chamadas de imagens intermediárias), onde cada camada representa uma mudança incremental pela qual o sistema passou para chegar no seu estado final. Mais especificamente, cada linha válida de um *Dockerfile* gera uma imagem intermediária, que incorpora a mudança da linha em questão na imagem final. Uma imagem Docker, portanto, só pode ser obtida se o usuário possuir todas as camadas que a formam. Por conta do nível de complexidade envolvido na organização de imagens, a gerência das mesmas é feita de maneira transparente pelo *Docker Engine*.

3.2.3 Gerenciamento de contêineres

Na máquina hospedeira, os contêineres são gerenciados por um *runtime* chamado *Docker Engine*. O *Docker Engine* é um servidor que atende a requisições através de uma API REST, e é responsável pela configuração, criação e interface dos contêineres com o mundo externo (tanto na máquina hospedeira quando na Internet). A comunicação com o *Docker Engine* é feita através de uma aplicação cliente (*Docker Client*), que pode conectar-se a um *Docker Engine* local (na mesma máquina) ou remoto (através da rede).

A comunicação entre *Docker Engine* e *Docker Client* é feita através de um *socket* UNIX, pertencente ao usuário *root*. Conseqüentemente, o processo do *Docker Engine* sempre executa como um processo do usuário *root*. Por conta dessa decisão de implementação, é necessário possuir acesso privilegiado a esse *socket* para interagir com o *Docker Engine*. Isso pode ser feito de duas maneiras: garantindo ao usuário acesso *root* ou adicionando o usuário no grupo *docker* na máquina hospedeira. Em ambientes não compartilhados, qualquer uma das alternativas é aceitável, pois o usuário em questão é o próprio administrador da máquina.

Por outro lado, em ambientes com múltiplos usuários, nenhuma das alternativas é suficiente para garantir a segurança dos usuários. Dar acesso *root* a algum usuário permite que ele tenha privilégios de administrador que podem ser indesejáveis dependendo da política de utilização da máquina. Da mesma forma, adicionar usuários não *root* ao grupo *docker* na máquina hospedeira também não é uma política de segurança válida,

pois existem ataques documentados, e bem conhecidos, que permitem o escalonamento de privilégios¹ para que um usuário obtenha acesso *root* a partir de um contêiner Docker (DOCKER. . . , 2017b). Essa característica faz do Docker uma solução não ideal para ambientes compartilhados, onde é desejável ter controle sobre quais usuários possuem privilégios de administrador.

3.2.4 Execução de contêineres

A execução de um contêiner é iniciada através do comando *docker run* a partir de um *shell* local ou de um comando remoto. Esse comando recebe como parâmetro uma imagem, parâmetros de configuração e o comando que será executado quando o contêiner for inicializado (caso ele não tenha sido especificado no *Dockerfile*). Ao executar o *docker run*, o *Docker Client* envia uma requisição para o *Docker Engine* com os parâmetros informados pelo usuário. O *Docker Engine*, por sua vez, interage com as APIs de *chroot*, *namespaces* e *cgroups* para criar um ambiente isolado de acordo com a configuração da imagem escolhida.

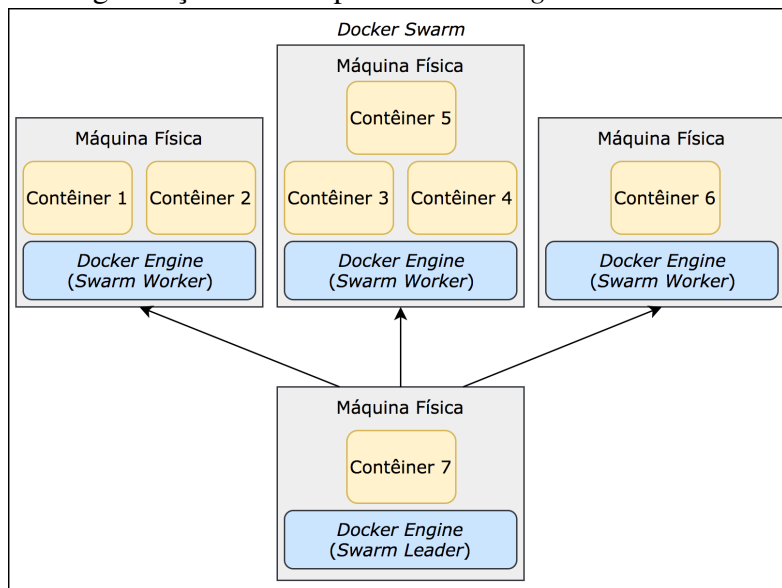
3.2.5 Docker em ambientes distribuídos

Conforme dito na seção 3.2.3, o *Docker Engine* é um servidor que fornece serviços através de uma API REST. Adicionalmente, um *Docker Client* pode se conectar tanto a um *Docker Engine* local quanto a um *Docker Engine* remoto. Além de permitir o desacoplamento entre cliente e servidor, a implementação do *Docker Engine* no formato de um servidor permite que múltiplos *Docker Engines* sejam configurados para trabalhar em conjunto, em um modelo distribuído. Para isso, existem diversas ferramentas que visam facilitar a gerência de contêineres em múltiplos *Docker Engines*. Uma dessas ferramentas, implementada nativamente no Docker, é o *Docker Swarm* (DOCKER. . . , 2017c), e sua arquitetura básica é ilustrada na Figura 3.4.

Em um *Docker Swarm*, um ou mais *Docker Engines* são classificados como líderes (*Swarm Leader*), e são os responsáveis pela criação de contêineres. Os demais, chamados de *Swarm Workers* na Figura 3.4, são dedicados apenas à execução de contêineres. A comunicação entre os *Docker Engines* envolvidos em um *Swarm* é feita através de uma

¹Escalonamento de privilégios ocorre quando um usuário consegue obter privilégios adicionais em um sistema.

Figura 3.4: Organização de múltiplos *Docker Engines* em um *Docker Swarm*.



Fonte: Autor.

rede de sobreposição (*overlay*) gerenciada pelo *Swarm Leader*, à qual todos os contêineres podem ser conectados (DOCKER..., 2017a).

Além da vantagem de aumentar o poder computacional disponível para a execução de contêineres, a distribuição dos mesmos em múltiplas máquinas físicas possibilita a melhora na disponibilidade dos serviços virtualizados: caso uma máquina apresente defeitos e tenha que ser desligada, o *Swarm Leader* detecta a perda da conexão com o *Docker Engine* e, automaticamente, inicializa novas instâncias dos contêineres afetados em máquinas físicas disponíveis.

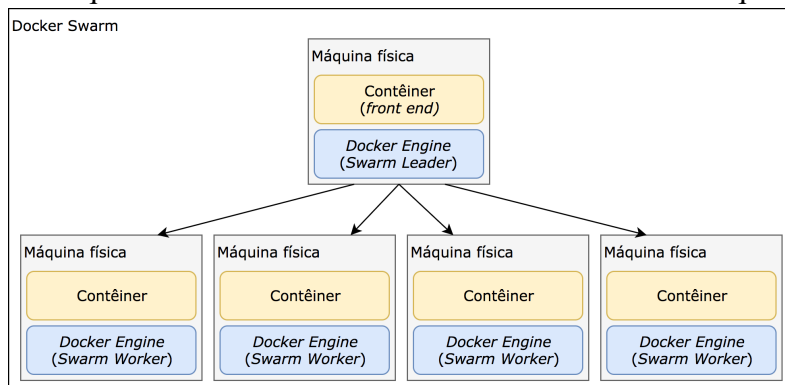
3.2.6 Arquitetura de um *cluster* MPI com Docker

Em (NGUYEN; BEIN, 2017), o autor propõe um *cluster* baseado em contêineres Docker com múltiplos nodos físicos. Essa proposta foi utilizada como base para a criação do *cluster* virtual com Docker, usado na análise experimental apresentada no capítulo 4.

Como o Docker virtualiza a camada de rede dos contêineres (utilizando o *namespace* de rede), os mesmos não são acessíveis (por padrão) fora da máquina física onde eles foram criados. Para que os contêineres possam ser distribuídos entre múltiplas máquinas físicas e a comunicação entre eles seja possível, foi utilizado o *Docker Swarm*. A organização do *cluster* de contêineres é feita da seguinte forma: o nodo que representa o *Swarm Leader* contém um único contêiner, que serve como *front end* do *cluster*. Como o

tamanho do *cluster* é variável (e deve ser especificado na inicialização), os demais contêineres são distribuídos igualmente entre os nodos conectados ao *cluster*. Todos os outros contêineres ficam com uma conexão SSH ativa para o *front end*. Dessa forma, é possível saber em tempo real o tamanho do *cluster* e o endereço IP de cada contêiner, através do comando *netstat* disponível no *shell* Linux.

Figura 3.5: Arquitetura de um *cluster* com *Docker Swarm* e 5 máquinas físicas.



Fonte: Autor.

A Figura 3.5 mostra a arquitetura do *cluster*. Na máquina registrada como *Swarm Leader* existe um único contêiner, e sua função é atuar como *front end* do *cluster*. Assim como em um *front end* real, todas as interações do usuário - como, por exemplo, a execução do comando *mpirun* para iniciar aplicações MPI - são feitas através desse contêiner. As demais máquinas físicas são usadas apenas como hospedeiras para contêineres de cálculo. Em casos onde é desejável possuir mais de um processo por máquina física, múltiplos contêineres podem ser instanciados nessa máquina.

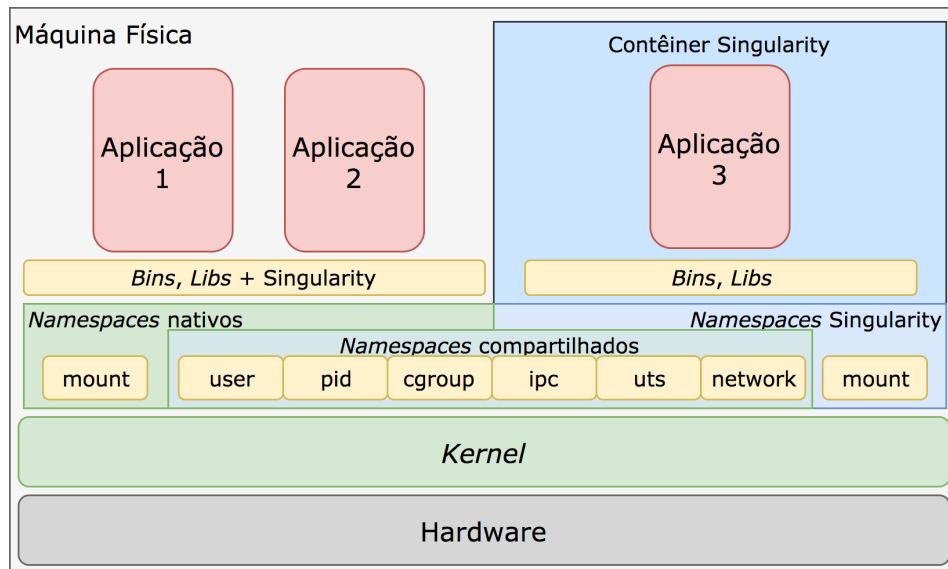
3.3 Singularity

Singularity é uma solução de virtualização baseada em contêineres, lançada no ano de 2016, cujo objetivo é trazer benefícios da virtualização, tais como reprodutibilidade e portabilidade, para ambientes de computação paralela (KURTZER, 2016). O desenvolvimento dessa plataforma foi guiado pelo desejo de executar aplicações em contêineres em um ambiente com recursos compartilhados (um *cluster* de computadores com múltiplos usuários, por exemplo) onde soluções já existentes, como o Docker, não atendem às políticas de segurança dos recursos, e portanto não podem ser instaladas.

A utilização de contêineres Singularity em uma máquina depende da instalação de um conjunto de aplicações no sistema operacional nativo. Essas aplicações são responsá-

veis pela interação com o *kernel* para a inicialização de contêineres e criação de imagens e *namespaces*. A Figura 3.6 mostra um esquema com os aspectos básicos de uma máquina executando contêineres Singularity.

Figura 3.6: Isolamento de contêineres Singularity, com *namespaces* compartilhados entre contêiner e SO nativo.



Fonte: Autor.

3.3.1 Virtualização

O sistema de contêineres Singularity tem por diferencial a criação de contêineres com um nível de virtualização menos invasivo do ponto de vista do isolamento. Como mostra a Figura 3.6, apenas os serviços essenciais do sistema operacional são virtualizados (no caso, apenas o *namespace* de *mount*), e o isolamento da maioria dos *namespaces* é opcional. Todos os *namespaces* que não são isolados pelo Singularity são compartilhados entre contêiner e sistema operacional nativo.

O não isolamento do *namespace* de usuário é particularmente importante para o contexto de uso do Singularity. Essa decisão faz com que o usuário que executa o contêiner seja o mesmo dentro e fora dele. Em outras palavras, um processo que executa em um contêiner sempre pertence ao usuário que criou o contêiner. Dessa forma, só é possível obter acesso *root* em um contêiner Singularity caso o usuário possua o privilégio na máquina hospedeira, e execute o contêiner como usuário *root*. Além disso, tentativas de escalonamento de privilégios dentro do contêiner são bloqueadas pelo próprio Singularity através de uma funcionalidade do *kernel* Linux chamada *no_new_privs*, que faz com que

não seja possível executar o comando *sudo* para obter privilégios, por exemplo.

Graças a essa característica, contêineres Singularity passam a ser vistos simplesmente como um processo do usuário que o executa. Dessa forma, o isolamento a partir de outros *namespaces* passa a não ser necessário pelas garantias de segurança do próprio sistema operacional hospedeiro. É possível, por exemplo, montar partes do sistema de arquivos diretamente no contêiner, pois o sistema hospedeiro garante que o usuário não terá acesso a arquivos aos quais ele não tem acesso na máquina hospedeira. De fato, alguns diretórios são automaticamente montados no sistema de arquivos do contêiner (como o diretório */tmp* e o diretório *home* do usuário), de forma a facilitar o compartilhamento de arquivos entre o sistema operacional nativo e contêiner.

Outro aspecto importante da visão do contêiner como um processo do usuário é a integração direta com recursos do sistema hospedeiro. No ambiente de um *cluster*, é comum que os usuários iniciem processos através de um escalonador e gerenciador de recursos, como o SLURM (JETTE; YOO; GRONDONA, 2002). Tratar o contêiner como um processo do usuário permite a utilização e a integração transparente desse contêiner com o gerenciador de recursos, uma vez que o último irá tratar o contêiner como um processo qualquer. Outros recursos que podem ser compartilhados entre sistema hospedeiro e contêiner são *drivers* de dispositivos, como placas aceleradoras ou interfaces de rede de alta velocidade, e escalonadores de aplicações paralelas como o *mpirun*, que é usado para executar aplicações MPI.

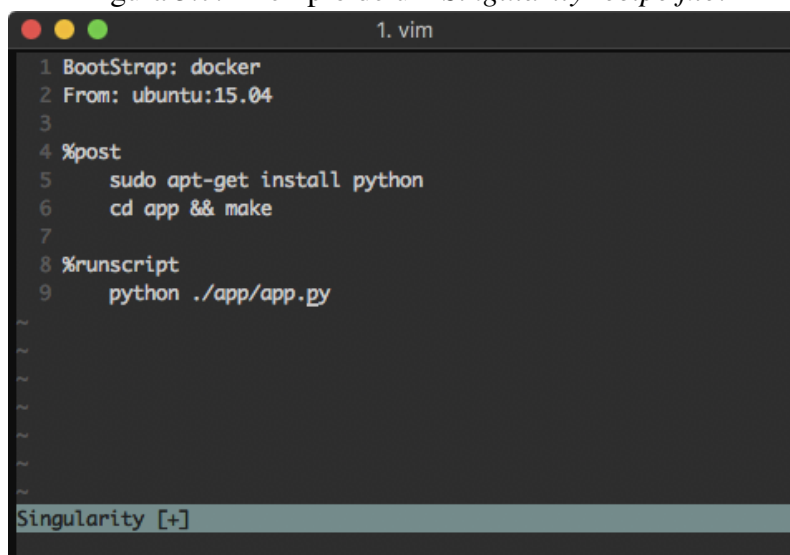
3.3.2 Imagens

O processo de criação de imagens no Singularity envolve a criação de uma imagem vazia e a instalação do ambiente de execução (bibliotecas, aplicações e dependências) nessa imagem. A instalação do ambiente de execução pode ser feita referenciando uma imagem já existente ou fornecendo um arquivo compactado com todos os arquivos necessários. Assim como no Docker, imagens Singularity não contêm o *kernel* do sistema operacional, pois o mesmo é compartilhado entre contêineres e o sistema operacional nativo.

Para modificar configurações e instalar aplicações no ambiente de uma imagem Singularity, é usado um arquivo de configuração chamado *Singularity recipe file*, exemplificado na Figura 3.7. De maneira similar ao que ocorre com um *Dockerfile*, é possível especificar uma imagem base, configurações do ambiente e instalar aplicações. No exem-

plô, as duas primeiras linhas do *recipe file* especificam qual a imagem base que será utilizada. O Singularity oferece compatibilidade com imagens Docker: na linha 1, onde é informada origem da imagem base, a palavra chave *docker* indica que o repositório referenciado é o *DockerHub*. A seção *%post* (linhas 5 e 6) é usada para especificar a instalação de pacotes adicionais e dependências para o ambiente de execução. Por fim, a seção *%runscript* (linha 9) define um *script* que será executado na inicialização do contêiner. Um último aspecto importante no *recipe file* é que, na linha 5, a instalação do pacote *python* é feita com o comando *sudo*. Como o Singularity não virtualiza o *namespace* de usuários, a criação da imagem precisa ser feita em um ambiente onde o usuário possua privilégios administrativos (ver seção 3.3.4).

Figura 3.7: Exemplo de um *Singularity recipe file*.



```
1 BootStrap: docker
2 From: ubuntu:15.04
3
4 %post
5     sudo apt-get install python
6     cd app && make
7
8 %runscript
9     python ./app/app.py
~
~
~
~
Singularity [+]
```

Fonte: Autor.

O resultado da criação de uma imagem é um arquivo compactado, contendo todo o sistema de arquivos do contêiner. O fato de a imagem ser um único arquivo autocontido facilita a distribuição da imagem entre diversos nodos, uma vez que o compartilhamento desse único arquivo é suficiente para transportar todo o ambiente de execução do contêiner.

3.3.3 Execução

Uma imagem Singularity, gerada a partir do comando *singularity build*, é um arquivo executável. Esse fato reforça a ideia de que um contêiner nada mais é do que um processo pertencente ao usuário que o executa. Ao executá-lo, o Singularity gera um

contêiner baseado na imagem em questão e executa, dentro do contêiner, o *script* de inicialização definido no *recipe file* durante a criação da imagem (caso ele exista). Outras formas de executar aplicações dentro de um contêiner Singularity são através dos comandos *singularity shell*, que fornece ao usuário um *shell* interativo, e *singularity exec*, que recebe um comando como parâmetro e executa o mesmo dentro do contêiner.

3.3.4 Desenvolvimento com Singularity

A solução de contêineres Singularity foi desenvolvida com foco na solução de problemas comumente encontrados por usuários de infraestruturas de computação paralela. Por conta disso, uma das premissas no fluxo de trabalho com Singularity é que existem dois ambientes distintos. O primeiro ambiente é usado para a criação da imagem. Nesse ambiente, é necessário que o usuário possua privilégios administrativos (*root*) para a criação e modificação do sistema de arquivos virtualizado, instalação de pacotes e dependências. O segundo ambiente é usado para a execução do contêiner. Esse ambiente pode ser controlado tanto pelo próprio usuário quando por terceiros, pois não são necessários privilégios administrativos para a execução de um contêiner Singularity.

Dessa forma, imagens Singularity podem ser criadas em um computador pessoal, controlado pelo usuário, e depois movidas para um ambiente com maior poder computacional (como um *cluster*) para a execução.

3.3.5 Arquitetura de um *cluster* MPI com Singularity

A utilização de contêineres Singularity em um ambiente de *cluster* é bastante simples. Como o contêiner, nesse caso, é visto como um processo qualquer do usuário que o executa, a virtualização de aplicações no *cluster* consiste simplesmente em substituir a execução da aplicação nativa pela execução do contêiner.

Por conta da transparência entre aplicações virtualizadas e o sistema operacional nativo, é possível reaproveitar todo o ferramental de software já existente no *cluster* para a execução de aplicações paralelas virtualizadas. No caso de aplicações MPI, por exemplo, a execução é realizada através do comando *mpirun* (executada no sistema operacional nativo), especificando o contêiner Singularity que deve ser executado. Como o próprio

contêiner é visto como o processo a ser executado, o *mpirun* fica encarregado de iniciar a execução do mesmo nos nós de cálculo.

3.4 Comparação entre soluções

Nas seções anteriores, foram apresentadas duas alternativas para a virtualização em nível de sistema operacional usando contêineres: Docker e Singularity. Esta seção faz uma breve comparação entre as duas ferramentas, do ponto de vista da utilização em um *cluster* de computação paralela.

3.4.1 Imagem de contêiner

Do ponto de vista da criação de uma imagem, ambas as plataformas oferecem soluções similares. É possível referenciar uma imagem já existente em algum repositório remoto, e personalizar a mesma para obter o ambiente de execução desejado.

Por outro lado, a representação e armazenamento de imagens difere entre uma plataforma e outra. A abordagem do Docker de dividir uma imagem em camadas gerenciadas pelo *Docker Engine* resulta em uma distribuição mais complexa da mesma entre os nodos de um *cluster*, pois é preciso importar a imagem desejada para o *Docker Engine* de cada nodo. Já no Singularity, a distribuição de uma imagem é idêntica à distribuição de qualquer outro arquivo. Como a imagem é gerenciada pelo próprio usuário, e é representada por um único arquivo no sistema de arquivos, é possível utilizar ferramentas conhecidas (como *scp* ou *rsync*) para a distribuição da mesma entre os nodos.

3.4.2 Reprodutibilidade

Através do uso de contêineres, é possível isolar o ambiente de execução em uma imagem autocontida e compartilhável. Assim, qualquer usuário com acesso a essa imagem (e que possa executar contêineres no sistema nativo) pode reproduzir a execução de uma aplicação no exato ambiente de software onde ela foi projetada. Por conta disso, a utilização de contêineres facilita a reprodutibilidade de experimentos científicos.

Entretanto, pelos motivos discutidos nas seções 3.4.1 e 3.4.3, apenas o Singularity se apresenta como uma solução viável em um *cluster* compartilhado. Outro ponto favo-

rável ao Singularity nesse sentido é que, como a imagem do contêiner é representada por um único arquivo, técnicas de criptografia podem ser utilizadas para garantir a integridade e autenticidade de imagens.

3.4.3 Instalação em ambientes compartilhados

Uma das motivações principais para o desenvolvimento do Singularity foi a dificuldade de instalação de uma solução de contêineres em ambientes compartilhados, como um *cluster* de computadores ou um centro de computação paralela. A resistência pela adoção do Docker como plataforma de contêineres nesses ambientes é explicada pelas implicações de segurança relacionadas ao *Docker Engine* executando como usuário *root*. Por conta disso, é comum encontrar administradores de *clusters* que não estejam dispostos a instalar o Docker em suas infraestruturas.

O Singularity, por outro lado, foi desenvolvido como uma solução para este problema. Assim, o mesmo implementa controles de acesso que permitem a sua integração em ambientes com múltiplos usuários, sem que existam implicações negativas para a administração do sistema e políticas de segurança. Em contraste com o Docker, o gerenciador de contêineres Singularity não executa como usuário *root*, os privilégios de usuários são mantidos idênticos ao executar um contêiner e não há a possibilidade de escalonamento de privilégios dentro de um contêiner. Por conta desses fatores, o Singularity é uma solução de contêineres aceita em diversas infraestruturas de computação paralela (SINGULARITY..., 2017a).

3.4.4 Compatibilidade e portabilidade

Conforme mencionado na seção 3.1, contêineres são um tipo de virtualização em nível de sistema operacional restrita a ambientes que executam o *kernel* Linux. Apesar disso, tanto Docker quanto Singularity fornecem instruções para instalar essas tecnologias em ambientes que executam sistemas operacionais Windows¹ e macOS. Em ambos os casos, a solução fornecida envolve a criação de uma máquina virtual (isto é, utilizando um hipervisor de tipo 1 ou tipo 2) que executa uma distribuição Linux. A partir daí, os

¹Existe também uma solução comercial de virtualização em nível de sistema operacional para Windows chamada *Docker Windows Containers*. Essa solução é baseada no *kernel* do Windows e, portanto, não é compatível com contêineres Linux estudados neste trabalho.

contêineres são criados com base no *kernel* Linux contido nessa distribuição.

Com relação a arquiteturas de computadores, o caso de uso típico dessas tecnologias é a criação de contêineres que são executados em arquiteturas *x86_64*. Essa é, portanto, a arquitetura oficialmente suportada. No caso do Docker, existem mecanismos desenvolvidos pelos usuários para instalar as ferramentas necessárias em arquiteturas *x86* (DOCKER-X86, 2017) e ARM (DOCKER-ARM, 2017). O Singularity, por outro lado, pode ser instalado nas arquiteturas mencionadas realizando o processo de instalação normalmente (ou seja, compilando o código fonte da aplicação). É importante ressaltar que a utilização de contêineres não garante a portabilidade de aplicações entre diferentes arquiteturas.

3.4.5 Tabela comparativa

A Tabela 3.1 apresenta um resumo com as principais características de cada uma das plataformas apresentadas neste capítulo.

Tabela 3.1: Comparação entre plataformas Docker e Singularity

	Docker	Singularity
Representação de imagem	Sequência de camadas	Arquivo único
Ambiente distribuído	Requer orquestrador externo (ex. <i>Docker Swarm</i>)	Suporta escalonadores de tarefas
Rede	Virtual (<i>overlay</i>)	Não virtualizada
Sistemas operacionais compatíveis	Linux, Windows, macOS	Linux, Windows, macOS
Privilégios <i>root</i> no contêiner	Sim	Não
Arquiteturas suportadas	<i>x86_64</i>	<i>x86_64</i> , <i>x86</i> e ARM

3.5 Resumo

Este capítulo apresentou duas plataformas de virtualização baseadas em contêineres - Docker e Singularity - que podem ser usadas para a execução de aplicações paralelas. Foram apresentados detalhes de cada uma das soluções tais como a criação de imagens, gerenciamento de contêineres e implicações de segurança, além de uma solução para a execução de contêineres em um *cluster* de computadores, para cada plataforma. No próximo capítulo, as plataformas apresentadas terão seu desempenho comparado à execução de aplicações paralelas diretamente no sistema operacional hospedeiro, através de uma infraestrutura física para a execução de aplicações paralelas.

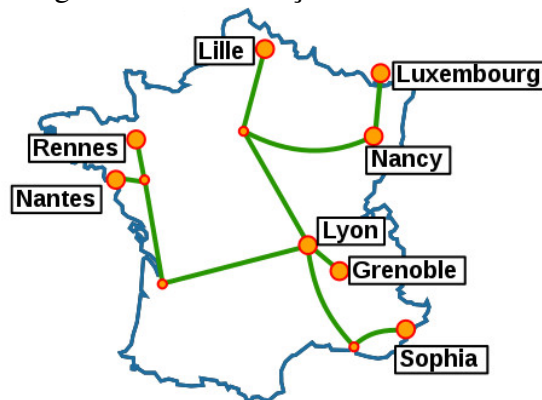
4 AVALIAÇÃO DE DESEMPENHO

O objetivo deste capítulo é avaliar o desempenho na execução de aplicações paralelas comparando as soluções baseadas em contêineres entre si e com aquela obtida por um sistema operacional nativo, sem virtualização. A plataforma de testes utilizada foi a Grid5000, que é introduzida a seguir. Também são descritos os *benchmarks* escolhidos e a metodologia para a execução dos experimentos. No final do capítulo, são apresentados os resultados obtidos e algumas conclusões parciais.

4.1 Grid 5000

A Grid5000 (GRID5000, 2017) é uma plataforma de testes de larga escala em uma arquitetura de *grid*. Essa plataforma é voltada para simulações e experimentos relacionados à ciência da computação. Seu foco principal é auxiliar em pesquisas nas áreas de computação em nuvem, computação de alto desempenho e *big data*. Localizada principalmente na França, essa plataforma é distribuída entre 8 cidades: Grenoble, Lille, Luxemburgo, Lyon, Nancy, Nantes, Rennes e Nice¹ (Figura 4.1). Em cada cidade há um centro de computação com um número variável de *clusters*, conectados à rede da Grid 5000.

Figura 4.1: Distribuição da Grid5000



Fonte: (GRID5000, 2017)

Os *clusters* que formam a Grid5000 fornecem uma grande variedade de hardware para os usuários, que podem reservar máquinas com características específicas para a execução de experimentos. Entre as famílias de hardware disponibilizados pela Grid 5000, estão (mas não limitam-se a): processadores AMD Opteron e Intel Xeon, de diversas

¹Em Nice, os *clusters* que participam da Grid5000 pertencem ao parque tecnológico *Sophia Antipolis*.

gerações; placas aceleradoras das séries Nvidia Tesla, Nvidia GTX, Nvidia Titan e Intel Xeon Phi; adaptadores de rede Ethernet, Myrinet e InfiniBand.

O acesso à Grid5000 é realizado através do estabelecimento de uma sessão remota de trabalho, via SSH, com um *gateway* de acesso. Ao conectar-se à Grid5000, o usuário é direcionado para esse *gateway* e, a partir daí, é preciso escolher a qual *site* o usuário deseja conectar. Cada uma das cidades que participam da Grid5000 corresponde a um *site*. Após a escolha, o usuário é direcionado para o *front-end* do *site* em questão.

A partir do *front-end*, é possível reservar recursos usando o escalonador padrão da Grid5000, chamado OAR (OAR, 2017). Ao enviar comandos para o escalonador, o usuário informa os parâmetros que especificam o hardware requisitado, o tempo de duração da reserva e o horário de início da reserva. Se a reserva for bem sucedida, o hardware estará disponível para o usuário no horário de início informado.

Por padrão, uma reserva fornece um usuário sem acesso *root* em uma distribuição Linux, geralmente, Debian. Esse cenário nem sempre é suficiente, pois pode ser necessário possuir acesso *root* na máquina para instalar pacotes específicos. Para solucionar esse problema, a Grid5000 oferece uma ferramenta para a instalação de imagens chamada Kadeploy3 (JEANVOINE; SARZYNIEC; NUSSBAUM, 2013), que permite ao usuário fazer uma instalação limpa de uma distribuição Linux, à qual o mesmo terá acesso *root* e poderá customizar de acordo com o necessário.

Para os testes realizados neste trabalho, foram utilizadas máquinas do *cluster Graphene*, na localidade de Nancy. Cada nodo desse *cluster* consiste de um processador Intel Xeon X3440, com 4 *cores*, 16GB de memória RAM e interface de rede Gigabit Ethernet. Durante os testes, foram utilizadas variações com 1, 5, 9 e 17 nodos desse mesmo *cluster*.

4.2 Benchmarks

Foram escolhidos 3 *benchmarks* diferentes para medir o desempenho das aplicações utilizando tecnologias de contêineres. São eles: *EP NAS Parallel Benchmark*, Ondes3D e Ping-Pong. Cada um desses *benchmarks* foi escolhido por conta de alguma característica específica que foi considerada relevante para fins da comparação de desempenho feita neste trabalho.

4.2.1 EP NAS Parallel Benchmark

O *NAS Parallel Benchmarks* (BAILEY et al., 1991) é um conjunto de *benchmarks* desenvolvido pela divisão de supercomputação da NASA. Os cálculos presentes nesse conjunto de *benchmarks* são baseados em aplicações relacionadas à dinâmica de fluidos, e têm como objetivo avaliar o desempenho de supercomputadores paralelos. Dentre os vários testes incluídos no *NAS Parallel Benchmark*, foi escolhido o *benchmark* EP.

O EP (*Embarrassingly Parallel*) é um *benchmark* que implementa um gerador de números aleatórios paralelo e que, como o nome indica, é uma aplicação com alto grau de paralelismo. Além disso, outra característica importante desse teste é o baixo nível de comunicação necessário entre processos ou *threads*. Ele foi escolhido para representar o caso ideal de uma aplicação paralela, com resultados esperados próximos de um *speedup* linear.

Pelo padrão dos *benchmarks* contidos no NAS, cada aplicação oferece suporte a diferentes tamanhos de problemas (chamados de classes). As classes disponíveis são, em ordem crescente: S (*small*) e W (*workstation*), que são projetadas para testes rápidos; A, B e C, cada uma com um problema 4 vezes maior do que a classe anterior; D, E e F, cada uma com um problema 16 vezes maior do que a classe anterior. Para este trabalho foi escolhida a classe B, por apresentar um tempo de execução compatível com a capacidade de processamento das máquinas utilizadas e com o tempo disponível para as execuções na Grid5000.

4.2.2 Ondes3D

O Ondes3D é uma aplicação usada para a simulação de propagação de ondas sísmicas, desenvolvido pelo Departamento de Pesquisas Geológicas e Minerais da França (*Bureau de Recherches Géologiques et Minières - BRGM*). O princípio de funcionamento dessa aplicação é a utilização de equações da física elastodinâmica para representar as ondas sísmicas, e a utilização do método de diferenças finitas para resolver essas equações. O Ondes3D apresenta características como desbalanceamento de carga e comunicação frequente entre processos. Essas duas características não são cobertas pelo *benchmark* EP.

A motivação principal para a escolha do Ondes3D como *benchmark* neste trabalho é a análise de desempenho de uma aplicação real, e não apenas de testes sintéticos. Além

disso, o Instituto de Informática da UFRGS tem uma parceria científica com o BRGM, e este trabalho contribui para reforçar essa interação.

Para a execução do Ondes3D, os parâmetros de configuração utilizados foram definidos pela simulação ESSAI (*teste*, em francês), estimados para executar em tempo hábil em um computador doméstico. Essa simulação, em pequena escala, foi escolhida para que fosse possível executar testes com um único processo ou *thread*, de forma a obter o tempo de execução sequencial. Além desse teste, foi feita uma execução da simulação de um terremoto real (SISHUAN) ocorrido na China em 2008, com magnitude 8.0 na escala Richter. Para essa simulação, foram utilizados 16 nodos e 64 processos nos mesmos *clusters* dos testes anteriores.

4.2.3 Ping-Pong

O *benchmark* Ping-Pong é bastante conhecido em computação distribuída. Ele é utilizado para medir a banda passante e a latência da comunicação entre dois processos. O funcionamento desse *benchmark* consiste em enviar mensagens entre um processo e outro, medindo o tempo decorrido entre o envio e o recebimento de uma mensagem. Neste trabalho, foi aplicada uma variação no Ping-Pong onde o tamanho das mensagens enviadas é dobrado a cada teste, até um tamanho máximo de mensagem de 1 MB.

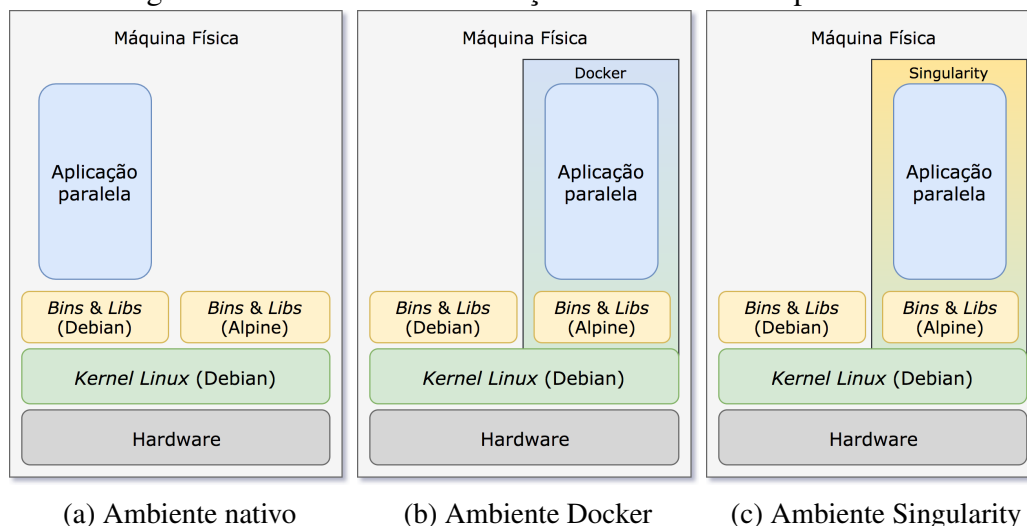
4.2.4 Versões de software utilizadas

Os testes em contêineres, tanto Docker quanto Singularity, foram executados a partir de uma imagem derivada do Alpine Linux (ALPINE, 2017). O Alpine Linux é uma distribuição baseada em ferramentas de sistema enxutas e altamente otimizadas, como a biblioteca padrão *musl libc* (MUSL, 2017) e o conjunto de ferramentas *busybox* (BUSYBOX, 2017). Adicionalmente, o Alpine Linux é bastante popular no contexto de contêineres por conta do tamanho extremamente reduzido, o que facilita a manutenção de um grande número de imagens e diminui o custo de armazenamento necessário para gerenciar ambientes com essa distribuição. Um contêiner executando uma distribuição do Alpine Linux requer aproximadamente 8MB de espaço em disco.

Os testes com o sistema nativo, sem nenhum tipo de virtualização, foram feitos usando a distribuição Linux Debian, na versão 8. A escolha do Debian 8 se deu

pela melhor compatibilidade dessa distribuição com as ferramentas disponibilizadas pela Grid5000. O *kernel* Linux é compartilhado entre contêineres e sistema operacional nativo, e, portanto, a versão é a mesma entre todas as execuções (3.16). A Figura 4.2 mostra os ambientes de execução utilizados nos testes.

Figura 4.2: Ambientes de execução utilizados nos experimentos



Para os *benchmarks* EP e Ondes3D, foram realizadas execuções usando versões MPI e OpenMP. O *benchmark* Ping-Pong foi executado apenas com MPI, pois o objetivo desse teste é avaliar o desempenho da comunicação em um ambiente de memória compartilhada.

A compilação de todos os *benchmarks* foi feita empregando o *GCC* (*GNU Compiler Collection*) na versão 5.3. A versão e *runtime* do OpenMP utilizados foram fornecidos pelo próprio compilador (OpenMP 4). Os *benchmarks* que envolvem MPI foram executados usando a implementação MPICH, na versão 3.2. Para todos os testes, foram utilizadas as configurações dos *makefiles* padrão, distribuídos com as aplicações. Dessa forma, não há otimizações do *GCC* para os *benchmarks* EP e Ping-Pong. No Ondes3D, são ativadas por padrão as otimizações de nível 2.

4.3 Metodologia

Para os *benchmarks* EP e Ondes3D, foram considerados três fatores de configuração diferentes: ambiente, número de processos ou *threads* e tipo de execução. As possibilidades de variação para cada uma das configurações são mostradas na Tabela 4.1.

O ambiente de execução é a base na qual a aplicação será executada. As possibi-

Tabela 4.1: Variações de fatores nos experimentos EP e Ondes3D

Fatores	Variáveis
Ambiente	Nativo, Docker, Singularity
Processos/threads	1, 4, 8, 16
Execução	MPI 4:1, MPI 1:1, OpenMP

lidades para esse fator são a utilização de contêineres Docker, contêineres Singularity e a execução nativa, sem uso de contêineres.

O número de processos, ou *threads*, refere-se a quantos processos (no caso de MPI) ou quantas *threads* (no caso de OpenMP) foram usados durante a execução do teste. As possibilidades para esse fator são 1, 4, 8 e 16.

O tipo de execução indica qual ferramenta e em que condições os testes foram executados. No caso do MPI, foram definidos dois tipos de execução distintos, chamados MPI 4:1 e MPI 1:1. No caso do MPI 4:1, são alocados no máximo 4 processos por nodo, de forma a não ultrapassar o limite de *cores* físicos da máquina. Dessa forma, as execuções com 1, 4, 8 e 16 processos foram executadas em 1, 1, 2 e 4 nodos, respectivamente. Na execução MPI 1:1 é alocado apenas um processo por nodo. Esse cenário foi incluído para explorar o pior caso de comunicação pela rede, uma vez que não existe comunicação intra-nodo nessa configuração. Assim, as execuções com 1, 4, 8 e 16 processos foram executadas em 1, 4, 8 e 16 nodos, respectivamente. Naturalmente, a execução OpenMP utiliza apenas um nodo. É importante lembrar que o processador do nodo escolhido possui 4 *cores* e que, portanto, é esperado as que execuções com 8 e 16 *threads* não apresentem melhoria no desempenho.

Foram geradas todas as combinações possíveis de configurações com base nas variações dos três fatores citados acima, e essas configurações serviram como base para a execução. Houve uma preocupação em obter resultados com validade estatística e, por conta disso, foi estipulado um nível de confiança de 99.74% (Z-TABLE, 2017). Para atingir esse nível de confiança, foram executadas 10 replicações de cada caso experimental. Adicionalmente, em casos raros onde *outliers* foram detectados, os mesmos foram removidos da amostra final.

O *benchmark* de rede (Ping-Pong) foi usado para avaliar a banda passante e latência média da rede em cada ambiente de execução (Docker, Singularity e nativo). Para isso, o *benchmark* foi configurado para enviar mensagens incrementalmente maiores, até um limite de tamanho de 1 MB. O tamanho inicial configurado para uma mensagem é de

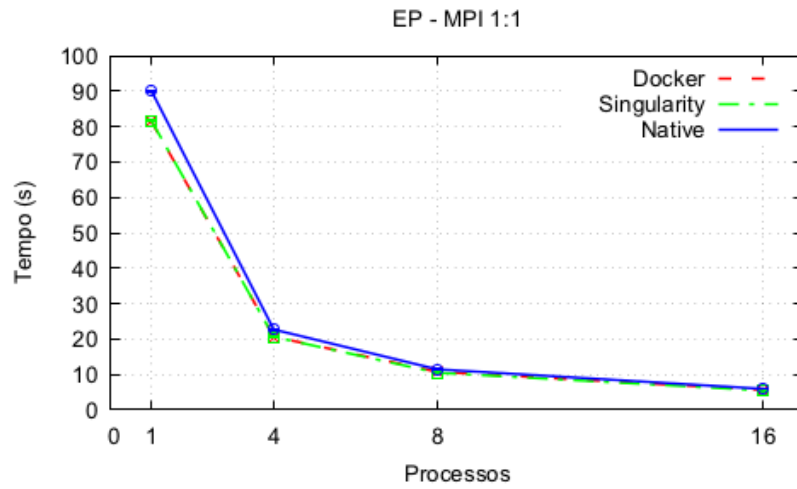
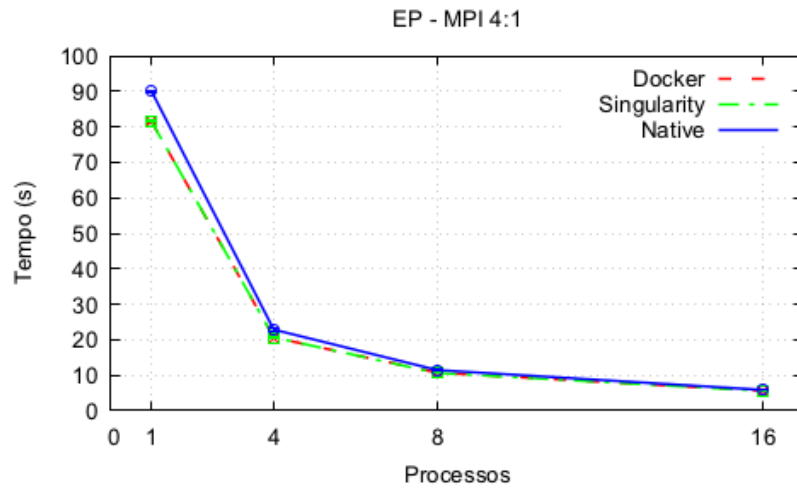
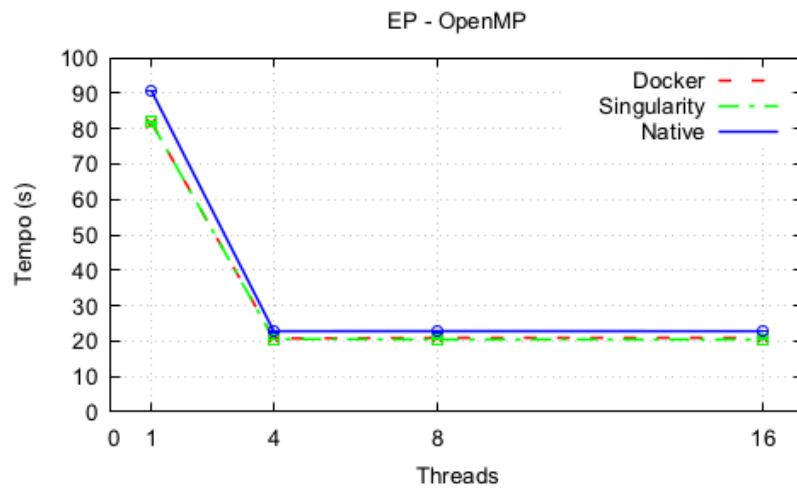
1 byte e as mensagens sucessivas são incrementadas em potências de 2. Após cada troca de mensagens, os processos são sincronizados com a utilização de uma barreira. Assim como nos *benchmarks* anteriores, o nível de confiança estipulado foi de 99.74% e cada caso de teste foi replicado 10 vezes.

4.4 Resultados

4.4.1 NAS EP

O conjunto de gráficos da Figura 4.3 mostra os resultados da execução dos testes para o *benchmark* EP. As execuções com Docker e Singularity apresentaram desempenho similar, o que indica que não há uma vantagem clara, em termos de desempenho, para a escolha de uma plataforma ao invés de outra para aplicações que exigem essencialmente cálculo. Comparando o desempenho de contêineres com as execuções no sistema operacional nativo, podemos observar que a execução nativa apresentou desempenho inferior às execuções com contêineres. Essa diferença de desempenho deve ser melhor investigada, mas está provavelmente relacionada ao uso de bibliotecas padrão (*libc*) diferentes nas distribuições Debian (execução nativa) e Alpine Linux (execução com contêineres). Adicionalmente, foi observado que o uso de contêineres não introduziu sobrecusto no desempenho da aplicação. Essa constatação é reforçada pelo fato da virtualização em nível de sistema operacional, com *cgroups* e *namespaces*, não adicionar nenhuma camada de software entre a aplicação e o sistema operacional. Dessa forma, o único sobrecusto existente para toda a aplicação é a inicialização do próprio contêiner. No caso do Singularity, o sobrecusto estimado fica na ordem de grandeza de milésimos de segundo (SINGULARITY... , 2017b).

Em uma das execuções MPI 4:1, com 16 processos do *benchmark* EP, foi detectado um *outlier* no ambiente de execução Docker. Não foram encontrados padrões para o aparecimento desse *outlier*, tendo em vista que ele ocorreu uma única vez. O tempo de execução foi de 31.377 segundos, o que excede em quase três vezes o desvio padrão da média das demais execuções, considerando uma amostra com esse elemento. O *outlier* foi removido da amostra final, por conta do tempo de execução extremamente anormal. Para tentar identificar a origem desse *outlier*, é necessário executar essa configuração de teste com mais replicações.

Figura 4.3: Comparação de desempenho entre soluções no *benchmark* EP(a) Tempo de execução para o *benchmark* EP MPI 1:1(b) Tempo de execução para o *benchmark* EP MPI 4:1(c) Tempo de execução para o *benchmark* EP OpenMP

4.4.2 Ondes3D

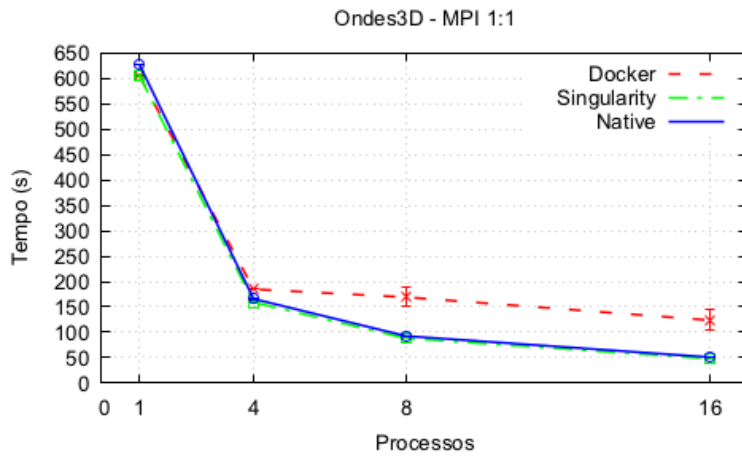
A Figura 4.4 mostra os resultados das execuções do *benchmark* Ondes3D. Nesse conjunto de testes, é possível observar que a tendência observada com o *benchmark* EP se mantém em parte. Comparando as execuções Singularity e nativa, a utilização de contêineres não gerou sobrecusto adicional. Entretanto, nos casos de MPI, o ambiente Docker apresentou degradação no desempenho. Essa degradação é mais evidente nos casos com 8 e 16 processos, e está provavelmente relacionada à quantidade de comunicação entre contêineres. A queda no desempenho da rede pode ser explicada pelo fato de os contêineres Docker estarem conectados a uma rede virtual, gerenciada pelo *Docker Swarm* (Seção 3.2.5), que pode ter introduzido sobrecusto adicional. Dois dados que reforçam essa suposição são a ausência da disparidade no desempenho quando há pouca ou nenhuma comunicação, como é o caso das execuções anteriores do *benchmark* EP e a execução do Ondes3D com OpenMP, onde o desempenho do Docker passou a ser comparável ao do Singularity.

Execução do Ondes3D em grande escala: também foi realizada uma execução do Ondes3D em uma simulação de um terremoto real. Para essa execução, foram necessários 16 nodos de cálculo, cada um com 4 processos MPI, totalizando 64 processos. Por conta do tempo de execução alto, foi realizada uma única execução para cada plataforma. O tempo de execução em cada uma das plataformas é mostrado na Tabela 4.2. Infelizmente, não foi possível realizar esse teste com contêineres Docker, pois o *front end* do *cluster* virtual não conseguiu identificar todos os 64 contêineres. Esse problema deve ser melhor investigado, mas a sua origem pode estar na forma como o *cluster* Docker utilizado identifica e monta a sua rede virtual (utilizando o comando *netstat*, Seção 3.2.6). A Tabela 4.2 mostra o tempo de execução ao executar a simulação nos ambientes nativo e Singularity.

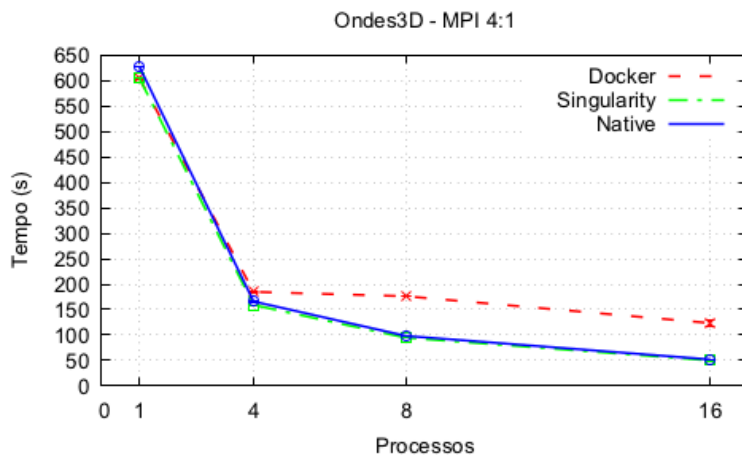
Tabela 4.2: Tempo de execução na simulação do terremoto SISHUAN

Ambiente	Tempo de execução aproximado (h:m:s)
Nativo	1:38:09
Singularity	1:35:47
Docker	-

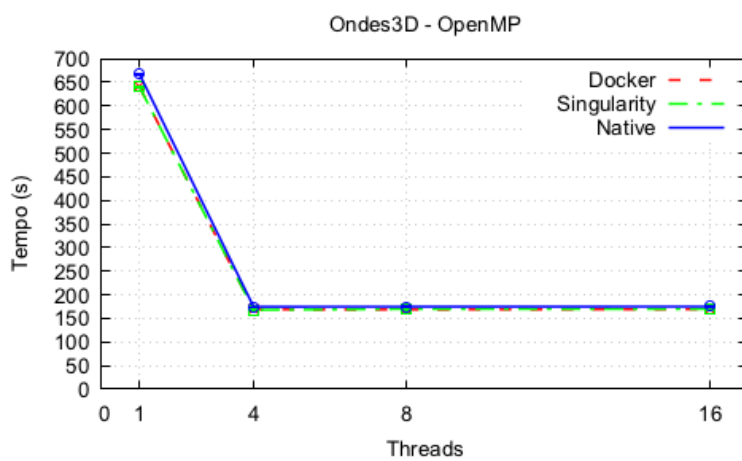
Os resultados apresentados na Tabela 4.2 reforçam a tendência observada nos testes anteriores, em que a utilização de contêineres (nesse caso, Singularity) não introduziu sobrecusto observável. Nesse teste, o tempo de execução com contêineres Singularity foi aproximadamente 2.58% menor do que a execução nativa equivalente.

Figura 4.4: Comparação de desempenho entre soluções no *benchmark* Ondes3D

(a) Tempo de execução Ondes3D MPI 1:1



(b) Tempo de execução Ondes3D MPI 4:1

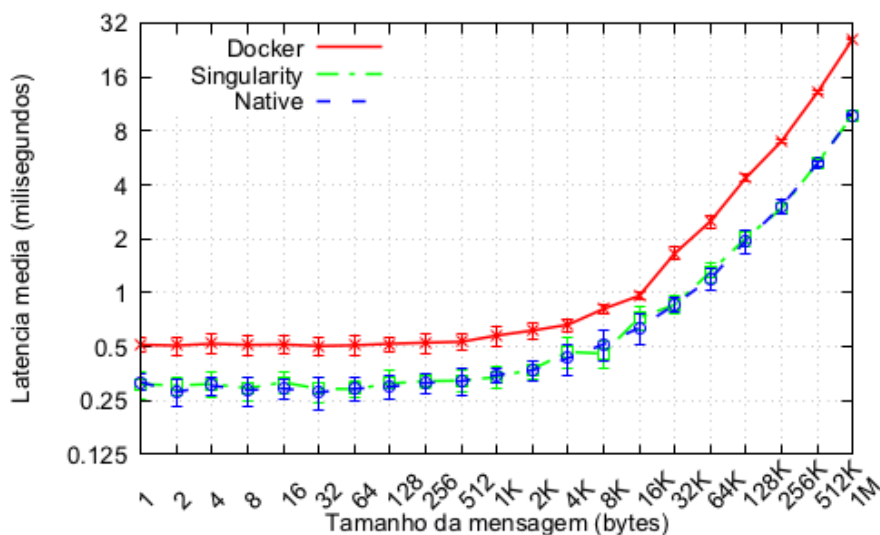


(c) Tempo de execução Ondes3D OpenMP

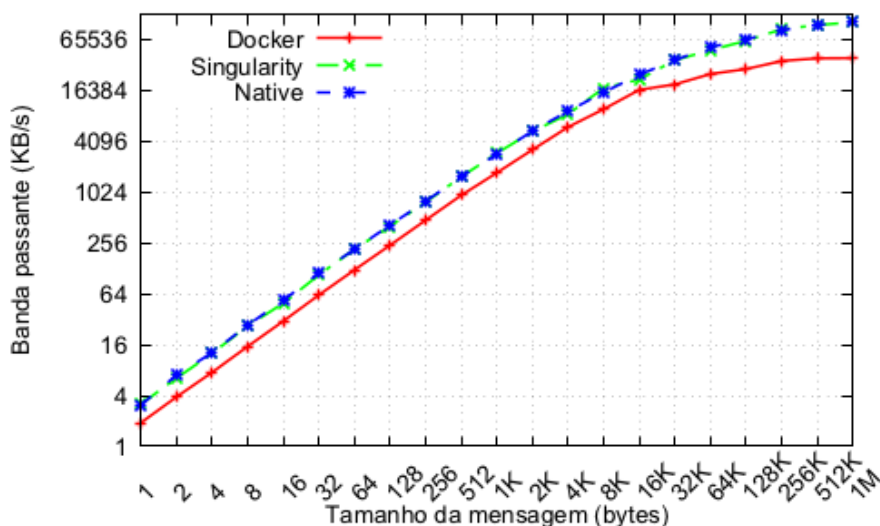
4.4.3 Ping-Pong

O *benchmark* Ping-Pong, cujos resultados são mostrados na Figura 4.5, foi escolhido para testar o desempenho de rede de cada uma das abordagens. Por conta da diferença entre a ordem de grandeza dos resultados apresentados nos gráficos, os mesmos encontram-se em escala logarítmica de base 2, em ambos os eixos. Analisando os gráficos, é possível observar que os resultados da execução reforçam que a queda no desempenho do Docker, ocorrido no experimento com Ondes3D, existe por conta do sobre-custo associado à comunicação pela rede. Os gráficos evidenciam perdas de desempenho que variam de 32% no melhor caso (com mensagens de tamanho 16 KB) a 166% no pior

Figura 4.5: Comparação de desempenho entre soluções no *benchmark* Ping-Pong



(a) Latência de rede em cada uma das soluções



(b) Banda passante em cada uma das soluções

caso (com mensagens de 1MB), aproximadamente, prejudicando o ambiente de execução com Docker. Nesse teste, é possível verificar que o desempenho da execução nativa e de contêineres Singularity são equivalentes.

4.5 Resumo

Neste capítulo foram apresentadas comparações de desempenho entre duas soluções de virtualização em nível de sistema operacional (Docker e Singularity) e uma solução nativa.

Do ponto de vista do desempenho em aplicações onde a computação predomina em relação à comunicação, a utilização de contêineres não introduziu sobrecusto que impactasse o tempo de execução. Esse resultado condiz com a teoria de virtualização por contêineres, pois a mesma apenas altera controles do *kernel* para alcançar o isolamento. Dessa forma, não é introduzida nenhuma camada de software entre a aplicação virtualizada e o *kernel* do sistema operacional, conforme mostra a Figura 3.1.

O desempenho no teste não sintético (usando a aplicação Ondes3D) demonstrou um ponto fraco para contêineres Docker, por conta do sobrecusto envolvido na utilização da rede virtualizada pelo *Docker Swarm*, necessária para o funcionamento da solução de *cluster* proposta para essa tecnologia.

Em linhas gerais, os resultados apresentados pelos experimentos indicam que a utilização de contêineres Singularity em aplicações paralelas e de alto desempenho é viável, pois não foi observada perda de desempenho ao introduzir essa tecnologia.

Além dos resultados concretos obtidos com os experimentos, os testes com contêineres tornaram explícita outra vantagem em utilizar virtualização em nível de sistema operacional. A execução com contêineres permite uma flexibilidade maior quanto à escolha de versões de software, bibliotecas e distribuição Linux. No contexto deste trabalho, essa característica foi mostrada na forma de contêineres executando bibliotecas do Alpine Linux em um ambiente cuja distribuição Linux nativa é o Debian. Esse aspecto é especialmente interessante em casos onde não é permitido ao usuário modificar a instalação do sistema operacional. Um exemplo prático desse caso, onde usuários de um *cluster* podem possuir privilégios limitados, são os *clusters* do Grupo de Processamento Paralelo e Distribuído (GPPD) da UFRGS. Além disso, mesmo em serviços de IaaS ou *grid* onde essa flexibilidade é permitida, a utilização de contêineres ainda é vantajosa por apresentar um tamanho de imagem e tempo de inicialização (*deployment*) consideravelmente menores

quando comparados à instalação de uma distribuição Linux completa.

A título de comparação, a imagem utilizada na Grid5000 para instalar a distribuição Debian usada nos testes (através da ferramenta Kadeploy3), ocupa cerca de 2 GB de espaço em disco e leva aproximadamente 10 minutos para que a instalação do ambiente seja concluída. As imagens de contêineres, tanto Docker quanto Singularity, com o ambiente de execução usado nos testes, ocupa aproximadamente 250MB e a inicialização do contêiner é quase instantânea.

5 CONCLUSÕES

A ubiquidade de sistemas de virtualização na indústria de software é evidente nos dias de hoje. A virtualização é uma abordagem extremamente popular tanto na utilização de máquinas virtuais - utilizadas em larga escala em provedores de computação em nuvem - quanto no uso de contêineres para o encapsulamento de aplicações.

Considerando a utilização de contêineres, o estado da arte para virtualização de serviços é a plataforma Docker. Extremamente popular por conta do isolamento quase completo entre aplicações, a facilidade na criação e compartilhamento de imagens e o sobrecurso computacional baixo, o Docker é empregado diariamente no desenvolvimento de software escalável e distribuído.

Um dos empecilhos para a utilização do Docker em ambientes de computação paralela, como *clusters* e *grids*, é o modelo de segurança e controle de acesso implementado nessa plataforma. As decisões de implementação não são compatíveis com ambientes compartilhados com múltiplos usuários. Esse fator faz com que a popularidade do Docker não seja reproduzida nesse meio, pois a instalação da plataforma é frequentemente negada por administradores de sistema.

Com o objetivo de trazer os benefícios da virtualização através de contêineres para a computação paralela, foi desenvolvido o Singularity. O Singularity resolve os principais empecilhos para a adoção de contêineres em computação de alto desempenho, sendo o principal deles o problema com segurança e controle de acesso em ambientes compartilhados. Outra promessa dessa tecnologia é uma integração mais transparente com *drivers* de dispositivos disponíveis no sistema operacional nativo, através de um modelo de virtualização mais flexível quanto aos aspectos do *kernel* que são virtualizados.

Em linhas gerais, a utilização de contêineres permite integrar diversos benefícios característicos de virtualização a ambientes de computação paralela. Dentre os benefícios observados, destacam-se o isolamento de aplicações, melhoras em questões de reprodutibilidade em função do ambiente de execução uniforme, portabilidade dos ambientes de execução e a possibilidade de o usuário definir qual distribuição Linux e versões de bibliotecas serão utilizadas no ambiente. Esse último ponto é especialmente interessante ao se considerar que, via de regra, administradores de um *cluster* de computação paralela não concedem privilégios administrativos para os usuários, que conseqüentemente não podem alterar facilmente configurações do sistema operacional nativo (como bibliotecas padrão) da máquina.

Através dos testes realizados neste trabalho, foi possível concluir que ambas as tecnologias estudadas são capazes de executar aplicações paralelas. Em aplicações caracterizadas por alto uso de processador, o desempenho de ambas tende a ser comparável. Quando há comunicação em rede, por outro lado, a solução utilizando contêineres Docker mostrou desempenho inferior, por conta da virtualização da pilha de rede que é feita por essa plataforma. Adicionalmente, foi observado que a utilização de tecnologias de contêineres não impactou o desempenho das aplicações testadas, quando comparado a uma execução nativa.

Considerando os resultados de todos os experimentos, é possível concluir que a utilização de contêineres Singularity como técnica de virtualização é viável em computação de alto desempenho. Essa conclusão é baseada em dois comportamentos essenciais, apresentados por essa plataforma. O primeiro deles é a compatibilidade com ambientes de *clusters*, por conta do modelo de virtualização e segurança da informação, que faz com que usuários possam executar aplicações em ambientes virtuais sem ter acesso privilegiado ao sistema operacional nativo. O segundo critério é a ausência de impacto no desempenho de aplicações, observado através dos resultados dos experimentos realizados neste trabalho.

5.1 Dificuldades encontradas

Durante a execução do trabalho, foram encontradas algumas dificuldades relacionadas com a instalação do software necessário nos ambientes nativo e virtualizado.

Com o objetivo de garantir um ambiente de execução uniforme entre as execuções nativa e com contêineres, foi feita uma tentativa de instalação da distribuição Alpine Linux em todas as plataformas. Apesar de a Grid5000 não restringir a instalação de sistemas operacionais, não foi possível instalar o Alpine Linux como sistema operacional nativo. Uma possível causa para esse problema é a incompatibilidade dessa distribuição, otimizada para executar em sistemas embarcados e com poucos recursos, com o ferramental de gestão da infraestrutura da Grid5000.

Para remediar o problema anterior, foi realizada a tentativa de migrar a distribuição do sistema operacional que executa nos contêineres para Debian ou CentOS, ambas suportadas pela Grid5000. Entretanto, nessas distribuições foi observada instabilidade na conexão entre contêineres Docker, gerenciadas pelo *Docker Swarm* e identificadas pelo comando *netstat* no *front end* do *cluster*. Essa instabilidade pode ter sido causada pelo

uso do *netstat*, que foi descoberto como obsoleto (*deprecated*) após a execução dos experimentos. Uma possível melhoria seria atualizar a arquitetura de *clusters* Docker para utilizar o comando *ss*, substituto do *netstat* em ambientes Linux.

5.2 Trabalhos Futuros

Com o objetivo de complementar o trabalho apresentado, algumas especulações do que pode ser feito no futuro são apresentadas nesta seção.

Um assunto que pode ser estudado com maior profundidade surgiu durante a tentativa de execução do *Ondes3D* em uma configuração de larga escala. Nesse caso, o Docker apresentou instabilidades na detecção de contêineres conectados à da rede virtual, gerenciada pelo *Docker Swarm*. Um possível trabalho, nesse caso, seria investigar a origem dessa instabilidade e buscar meios de resolvê-la. A suspeita inicial é que o problema esteja relacionado à forma como o *cluster* Docker identifica contêineres conectados ao *front-end*, através do comando *netstat*, e uma possível instabilidade desse comando ao lidar com conexões gerenciadas pelo *Docker Swarm*.

Outra oportunidade de melhoria na completude deste trabalho seria uniformizar o ambiente de execução proposto, de forma a utilizar a mesma distribuição Linux em todos os testes.

Por fim, um último ponto interessante para estudo seria a execução de testes utilizando máquinas híbridas, de forma a confirmar a compatibilidade e medir o desempenho dessas tecnologias de contêineres com placas aceleradoras, como GPUs.

REFERÊNCIAS

- ALPINE. 2017. Available from Internet: <<https://alpinelinux.org/about/>>.
- BAILEY, D. H. et al. **The NAS parallel benchmarks**. [S.l.], 1991.
- BOUKERCHE, A.; AL-SHAIKH, R. A.; NOTARE, M. S. M. A. Towards highly available and scalable high performance clusters. **J. Comput. Syst. Sci.**, Academic Press, Inc., Orlando, FL, USA, v. 73, n. 8, p. 1240–1251, dec. 2007. ISSN 0022-0000.
- BUSYBOX. 2017. Available from Internet: <<https://busybox.net/about.html>>.
- BUTENHOF, D. R. **Programming with POSIX threads**. [S.l.]: Addison-Wesley, 1997. ISBN 0201633922.
- CARISSIMI, A. da S. **Virtualização: Princípios básicos e aplicações**. 2009.
- CGROUPS. 2017. Available from Internet: <https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01>.
- CHROOT. 2017. Available from Internet: <https://wiki.archlinux.org/index.php/Change_root>.
- CLOUDSTACK. 2017. Available from Internet: <<https://cloudstack.apache.org/>>.
- DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. **Computational Science & Engineering, IEEE**, IEEE, v. 5, n. 1, p. 46–55, 1998.
- DOCKER. 2017. Available from Internet: <<https://www.docker.com/what-container>>.
- DOCKER. 2017. Available from Internet: <<https://www.docker.com/>>.
- DOCKER-ARM. 2017. Available from Internet: <<https://blog.hypriot.com/getting-started-with-docker-and-linux-on-the-raspberry-pi/>>.
- DOCKER Networking. 2017. Available from Internet: <<https://docs.docker.com/engine/userguide/networking/>>.
- DOCKER Privilege Escalation. 2017. Available from Internet: <<https://fosterelli.co/privilege-escalation-via-docker.html>>.
- DOCKER Swarm mode. 2017. Available from Internet: <<https://docs.docker.com/engine/swarm/>>.
- DOCKER-X86. 2017. Available from Internet: <<https://dzone.com/articles/docker-daemon-for-32-bit-architecture>>.
- DOCKERHUB. 2017. Available from Internet: <<https://hub.docker.com/>>.
- EC2. 2017. Available from Internet: <<https://aws.amazon.com/ec2/>>.
- ELASTICITY. 2017. Available from Internet: <<https://www.cloudassessments.com/blog/elasticity-cloud-computing/>>.

EUCALYPTUS. 2017. Available from Internet: <<https://docs.eucalyptus.com/eucalyptus/latest/>>.

FORUM, M. P. **MPI: A Message-Passing Interface Standard**. Knoxville, TN, USA, 1994.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949.

FREEBSD. 2017. Available from Internet: <<https://www.freebsd.org/>>.

GEIST, A. et al. **PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing**. Cambridge, MA, USA: MIT Press, 1994. ISBN 0-262-57108-0.

GRID5000. 2017. Available from Internet: <<https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home>>.

HOSTINGADVICE. 2017. Available from Internet: <<http://www.hostingadvice.com/how-to/iaas-vs-paas-vs-saas/>>.

HYPER-V. 2017. Available from Internet: <<https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/>>.

JEANVOINE, E.; SARZYNIEC, L.; NUSSBAUM, L. Kadeploy3: Efficient and Scalable Operating System Provisioning. **USENIX ;login:**, v. 38, n. 1, p. 38–44, feb. 2013.

JETTE, M. A.; YOO, A. B.; GRONDONA, M. Slurm: Simple linux utility for resource management. In: **In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003**. [S.l.]: Springer-Verlag, 2002. p. 44–60.

KURTZER, G. M. **Singularity 2.1.2 - Linux application and environment containers for science**. 2016. Available from Internet: <<https://doi.org/10.5281/zenodo.60736>>.

LEE, H. Virtualization basics: Understanding techniques and fundamentals. In: . [S.l.: s.n.], 2014.

MELL, P. M.; GRANCE, T. **SP 800-145. The NIST Definition of Cloud Computing**. Gaithersburg, MD, United States, 2011.

MUSL. 2017. Available from Internet: <<https://www.musl-libc.org/>>.

MYSQL. 2017. Available from Internet: <<https://www.mysql.com/>>.

NAMESPACES. 2017. Available from Internet: <<http://man7.org/linux/man-pages/man7/namespaces.7.html>>.

NGINX. 2017. Available from Internet: <<https://www.nginx.com/>>.

NGUYEN, N.; BEIN, D. Distributed mpi cluster with docker swarm mode. **IEEE 7th Annual Computing and Communication Workshop and Conference**, 2017.

NICKOLLS, J. et al. Scalable parallel programming with cuda. **Queue**, ACM, New York, NY, USA, v. 6, n. 2, p. 40–53, mar. 2008. ISSN 1542-7730.

OAR. 2017. Available from Internet: <<https://oar.imag.fr/>>.

OPENSTACK. 2017. Available from Internet: <<https://www.openstack.org/>>.

PARDESHI, S. N.; PATIL, C.; DHUMALE, S. Grid computing architecture and benefits. **Journal of Scientific and Research Publications**, v. 3, n. 8, aug. 2013. ISSN 2250-3153.

PHEATT, C. Intel®; threading building blocks. **J. Comput. Sci. Coll.**, Consortium for Computing Sciences in Colleges, USA, v. 23, n. 4, p. 298–298, abr. 2008. ISSN 1937-4771.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. **Commun. ACM**, ACM, New York, NY, USA, v. 17, n. 7, p. 412–421, jul. 1974. ISSN 0001-0782.

RABBITMQ. 2017. Available from Internet: <<https://www.rabbitmq.com/>>.

RDS. 2017. Available from Internet: <<https://aws.amazon.com/rds/>>.

SINGULARITY Clusters. 2017. Available from Internet: <<http://singularity.lbl.gov/citation-registration>>.

SINGULARITY Overhead. 2017. Available from Internet: <<http://singularity.lbl.gov/about#containers-are-image-based>>.

SOLARIS. 2017. Available from Internet: <<https://www.oracle.com/solaris/solaris11/index.html>>.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **IEEE Des. Test**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 3, p. 66–73, may 2010. ISSN 0740-7475.

VIRTUALBOX. 2017. Available from Internet: <<https://www.virtualbox.org/wiki/VirtualBox>>.

VMWARE. 2017. Available from Internet: <<https://www.vmware.com/products/esxi-and-esx.html>>.

XEN. 2017. Available from Internet: <<https://www.xenproject.org/>>.

Z-TABLE. 2017. Available from Internet: <<http://www.statisticshowto.com/tables/z-table/>>.