UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

SCIFLOW-BRIDGE: UMA FERRAMENTA PARA CRIAÇÃO E DISTRIBUIÇÃO DE IMAGENS DE CONTÊINERES PARA WORKFLOWS CIENTÍFICOS

TRABALHO DE GRADUAÇÃO

Bruno da Silva Alves

Santa Maria, RS, Brasil

2019

SCIFLOW-BRIDGE: UMA FERRAMENTA PARA CRIAÇÃO E DISTRIBUIÇÃO DE IMAGENS DE CONTÊINERES PARA WORKFLOWS CIENTÍFICOS

Bruno da Silva Alves

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientadora: Profª. Dr. Andrea Schwertner Charão

464
Santa Maria, RS, Brasil

2019

da Silva Alves, Bruno

Sciflow-Bridge: Uma Ferramenta para Criação e Distribuição de Imagens de Contêineres para Workflows Científicos / por Bruno da Silva Alves. – 2019.
65 f.: il.; 30 cm.

Orientadora: Andrea Schwertner Charão
Monografia (Graduação) - Universidade Federal de Santa Maria, Centro de Tecnologia, Curso de Ciência da Computação, RS, 2019.

1. Workflows Científicos. 2. Contêineres. 3. Sistemas Distribuído. I. Schwertner Charão, Andrea. II. Título.

E-mail: nomedoautor@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

SCIFLOW-BRIDGE: UMA FERRAMENTA PARA CRIAÇÃO E DISTRIBUIÇÃO DE IMAGENS DE CONTÊINERES PARA WORKFLOWS CIENTÍFICOS

elaborado por
Bruno da Silva Alves

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

COMISSÃO EXAMINADORA:

Andrea Schwertner Charão, Dr.
(Presidente/Orientadora)

João Vicente Ferreira Lima, Dr. (UFSM)

Adriano Petry, Dr. (INPE)

Santa Maria, 04 de Dezembro de 2019.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

SCIFLOW-BRIDGE: UMA FERRAMENTA PARA CRIAÇÃO E DISTRIBUIÇÃO DE IMAGENS DE CONTÊINERES PARA WORKFLOWS CIENTÍFICOS

AUTOR: BRUNO DA SILVA ALVES
ORIENTADORA: ANDREA SCHWERTNER CHARÃO
Local da Defesa e Data: Santa Maria, 04 de Dezembro de 2019.

Um Fluxo de Trabalho Científico define como um experimento é caracterizado, assim determinando em qual ordem tarefas devem ser executadas e sobre quais conjunto de dados as mesmas devem operar. Cada tarefa pode possuir diferentes dependências de ambiente, como bibliotecas, executáveis e dados de entrada. Os contêineres caracterizam uma virtualização leve e são capazes de encapsular várias camadas de software. Nesse contexto, o presente trabalho propõe-se a criar uma ferramenta que seja capaz de executar esses fluxos e gerenciar as dependências através da eficiente criação e distribuição de imagens de contêineres. Espera-se que ao se utilizar essa ferramenta, os pesquisadores consigam executar fluxos de maneira facilitada e consigam também usufruir dos recursos de processamento disponíveis nos laboratórios. Dois workflows da bioinformática foram executados com a ferramenta criada, os experimentos comprovaram que a ferramenta é capaz de executar workflows científicos, o uso da *cache* diminuiu o tempo de execução dos fluxos e que diferentes estratégias para o posicionamento dos contêineres também influenciam no tempo de execução. A utilização da ferramenta proporciona flexibilidade, escalabilidade e reprodutibilidade aos workflows científicos.

Palavras-chave: Workflows Científicos. Contêineres. Sistemas Distribuído.

ABSTRACT

Undergraduate Final Work
Curso de Ciência da Computação
Federal University of Santa Maria

SCIFLOW-BRIDGE: A TOOL FOR CREATION AND DISTRIBUTION OF CONTAINER IMAGES FOR SCIENTIFIC WORKFLOWS

AUTHOR: BRUNO DA SILVA ALVES
ADVISOR: ANDREA SCHWERTNER CHARÃO
Defense Place and Date: Santa Maria, December 04th, 2019.

A Scientific Workflow defines how an experiment is characterized, determining in which order tasks are performed and which dataset each task must operate on. Each task have different environment dependencies, such as libraries, executables, and input data. Containers feature light virtualization and are capable of encapsulating multiple layers of software. In this context, the present work proposes to create a tool that is capable of executing these workflows and managing dependencies through the efficient creation and distribution of container images. It is hoped that by using this tool, researchers will be able to execute workflows easily and will also be able to take advantage of the processing resources available in the laboratories. Two bioinformatics workflows were executed with the created tool, the experiments proved that the tool is capable of performing scientific workflows, the use of cache has decreased the execution time of the workflows and was observed that different strategies for the positioning of the containers also influences the runtime. The use of the tool gives scientific workflows flexibility, scalability and reproducibility.

Keywords: Scientific Workflows. Containers. Distributed Systems..

Dedico este trabalho à minha família e amigos.

AGRADECIMENTOS

Gostaria de agradecer e dedicar este trabalho à minha mãe Ivane e ao meu pai Claiton pelo apoio em todos os dias da minha vida e pelos ensinamentos e lições que me ajudaram durante esta trajetória e que formaram o meu caráter. Agradeço, também, ao meu irmão Clayton pelas conversas, lanches e caronas, mas também por ter me inspirado a escolher a área da tecnologia.

Agradeço à todos os professores que passaram pela minha formação, pelos ensinamentos e conselhos. Em especial, agradeço a minha orientadora Andrea Charão por me guiar na escrita deste trabalho e pelo entusiasmo no qual as novas ideias eram recebidas.

Agradeço aos meus amigos Carol, Felipe, Fernando, Gabriel, Greice, Luiza, Marcos, Mariana, Michel, Pedro e Vinícius e Yagor pela parceria e por estarem sempre presentes, principalmente nas noites de pizza, jogos e música, que eram acompanhadas de muitas risadas e, por vezes, conselhos. E Nathalia, obrigado por todos os abraços, carinhos e risadas que tornam os meus dias mais leves e alegres.

Agradeço à todos os amigos dos laboratórios por onde passei, obrigado pelas risadas, conversas não tão técnicas, inúmeras ideias de projetos engraçados e por todas as xícaras de café de proporções aleatórias.

“Ao infinito... e além!”
— BUZZ LIGHTYEAR

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

NASA	*National Aeronautics and Space Administration*
SO	Sistema Operacional

SUMÁRIO

1 INTRODUÇÃO

A comunidade científica ao redor do mundo realiza pesquisas com a finalidade de resolver problemas referentes a diversas áreas. Uma parte importante da pesquisa é a realização de experimentos, onde os cientistas podem comprovar ou refutar hipóteses previamente formuladas. Na realização desses experimentos, muitos cientistas utilizam ferramentas de *software* ou *hardware* que processam os dados coletados durante a pesquisa. Esse caminho que os dados devem percorrer e a determinação das tarefas que devem atuar sobre os mesmos definem um Workflow Científico.

A criação e definição desses fluxos não é uma tarefa trivial, pois em trabalhos científicos complexos muitos processos são definidos e várias dependências são estabelecidas. Nesse contexto, muitas ferramentas surgiram para auxiliar os cientistas na criação, manutenção, documentação e execução dos fluxos de trabalho científicos como o Projeto Kepler[1], Projeto Triana[2], Apache Taverna[3], SCIRun[4]. Essas ferramentas são conhecidas por Sistemas de Workflow Científico ou Engines de Workflow Científico. No entanto, esses sistemas não resolvem todos os desafios enfrentados pelos cientistas, principalmente no que diz respeito à definição de um ambiente sólido para a execução de tarefas. Para que as tarefas sejam executadas, é necessário que exista um ambiente com todos os *software* e bibliotecas utilizados.

Uma das maneiras para prover um ambiente configurado e pronto para as aplicações é a utilização da virtualização, que pode ser feita através do uso de máquinas virtuais ou, recentemente, através do uso de contêineres. Sabe-se que os contêineres são capazes de virtualizar um sistema operacional de forma leve (Kovacs, 2017), (F. J. P. Mulerickal; Paul; Sastri, 2015), assim os mesmos representam uma boa alternativa para esse contexto, onde várias tarefas podem ser executadas concomitantemente. Portanto, uma das alternativas que surgiram para atacar o problema da definição de ambientes foi a utilização de contêineres integrados aos Sistemas de Workflow. Entretanto, a integração entre contêineres e Workflows também apresenta alguns desafios, como a composição, distribuição e criação de contêineres. Nesses termos, o presente trabalho propõe-se a projetar e implementar uma solução para o problema de criação e distribuição de imagens de contêineres no contexto de Workflows Científicos.

[1] https://kepler-project.org/ - acessado em 01/05/2019.
[2] http://www.triana.co.uk - acessado em 01/05/2019.
[3] https://taverna.incubator.apache.org/ - acessado em 01/05/2019.
[4] http://www.sci.utah.edu/cibc-software/scirun.html - acessado em 01/05/2019.

1.1 Objetivos

1.1.1 Objetivo geral

Este trabalho tem o objetivo de criar uma ferramenta que gerencie as dependências dos fluxos de trabalho científicos através da eficiente criação e distribuição de imagens de contêineres para a execução de um conjunto de tarefas definidas por um workflow científico.

1.1.2 Objetivos Específicos

- Investigar tecnologias de contêiner disponíveis que podem ser utilizadas no contexto de Workflows Científicos.
- Comparar as Engines de Workflow Científico criadas, destacando pontos positivos e negativos.
- Investigar na literatura quais técnicas já foram utilizadas para a criação e distribuição de imagens no contexto abordado.
- Desenvolver uma ferramenta capaz de criar e distribuir imagens de contêineres para a execução de um Workflow Científico.
- Avaliar o desempenho da ferramenta desenvolvida.

2 FUNDAMENTOS E REVISÃO DA LITERATURA

Neste capítulo serão apresentados conceitos e definições sobre virtualização, abordando tópicos como as diferenças entre contêineres e máquinas virtuais, segurança de contêineres e serão detalhadas ferramentas capazes de virtualizar sistemas em nível de sistema operacional e em nível de hardware. Posteriormente, os workflows científicos são apresentados, onde uma análise sobre as estruturas básicas é feita, exemplos de workflows da bioinformática são apresentados e as *engines* para fluxos de trabalho são caracterizadas. Ao final deste capítulo, a integração entre contêineres e workflows científicos é discutida.

2.1 Tecnologias de Virtualização

"*Virtualização é um tipo de tecnologia que combina ou divide recursos computacionais a fim de disponibilizar um ou mais ambientes operacionais, e para isso, utiliza metodologias como particionamento ou agregação de hardware e software, simulação de máquina parcial ou completa, emulação, time-sharing e outros*"(NANDA; CHIUEH, 2005). Em geral, utiliza-se a virtualização quando se está a procura de um ambiente com as mesmas funcionalidades de um sistema operacional real, porém com um certo grau de isolamento e independência do hardware físico.

A utilização da virtualização pode ser aplicada a vários cenários, como por exemplo, para testes de aplicações em produção, onde se deseja verificar as funcionalidades e possíveis *bugs* de um *software* em desenvolvimento. Neste cenário, é importante que exista um ambiente sólido e isolado do resto do sistema, com as dependências instaladas. Um dos maiores casos de uso de sistemas virtualizados são as plataformas de computação em nuvem, na qual diversas aplicações são executadas em um servidor (ou em vários), cada uma com perfis diferentes de uso de recursos. Assim, a virtualização tornou-se uma das ferramentas básicas para a computação em nuvem (XAVIER et al., 2013).

Dentre os tipos de virtualização existentes, a virtualização de hardware e a virtualização a nível de Sistema Operacional (S.O) vêm se destacando, principalmente, na área de computação em nuvem. Enquanto a virtualização de hardware caracteriza um tipo de virtualização pesada e capaz de virtualizar um sistema operacional por completo, a virtualização à nível de S.O representa uma alternativa leve e capaz de isolar a execução de vários processos em um ambiente chamado de contêiner.

2.1.1 Contêineres

A primeira ideia que impulsionou o desenvolvimento de contêineres ocorreu no ano de 2000, com o surgimento do jail no FreeBSD. Os chamados jails eram ambientes seguros que poderiam ser compartilhados com vários usuários, possibilitando o particionamento do sistema em vários subsistemas (REDHAT, 2019 - acesso em 02/09/2019). Dessa forma, começou a surgir interesse em disponibilizar vários ambientes em um único Sistema Operacional e algumas funcionalidades do Kernel do Linux começaram a surgir. A criação do Control Groups (CGroups) fez com que os recursos como CPU, memória, disco e rede pudessem ser alocados por um grupo de processos e essa alocação poderia, ainda, ser limitada pelo Kernel. Outra funcionalidade que surgiu foi o Namespace, através do qual um processo enxerga um recurso (ou parte dele) como algo privado e de uso exclusivo, criando uma camada de abstração entre o que é visto pelo processo e o recurso real disponível.

A primeira solução concreta para uso de contêineres foi o projeto LXC (Linux Containers) que reuniu algumas das funcionalidades do Kernel e disponibilizou aos usuários uma interface para criar e gerenciar aplicações e sistemas com contêineres. O LXC utiliza as seguintes funcionalidades do Kernel: CGroups, namespaces, perfis Apparmor e SELinux, Seccomp policies, CHroots. A Tabela 2.1 detalha cada funcionalidade mencionada. Em 2008, surgiu o Docker, que inicialmente utilizava os contêineres do LXC, mas que mais tarde passou a usar uma solução própria de contêineres. A criação de uma biblioteca de imagens de contêineres foi um dos principais avanços que o Docker trouxe, pois permitiu que a comunidade criasse e disponibilizasse imagens em um repositório chamado Docker Hub[5]. A presença dessa biblioteca disseminou o uso de contêineres, pois dessa maneira era mais fácil e rápido obter imagens de contêineres prontas para o uso.

Em resumo, os contêineres são exemplos de virtualização em nível de Sistema Operacional, que utilizam algumas funcionalidades do Kernel para criar um ambiente virtual, seguro e isolado do sistema hospedeiro. Ainda, os contêineres caracterizam um tipo de virtualização leve, pois não utilizam uma camada de software que virtualiza o hardware (como um Hypervisor), assim conseguindo níveis de desempenho de CPU, memória e disco semelhantes aos níveis nativos (XAVIER et al., 2013)(Felter et al., 2015), quando configurado corretamente.

[5] https://hub.docker.com/search?q=&type=image - acessado em 02/09/2019.

Tabela 2.1 – Funcionalidades do Kernel

Nome	**Funcionalidade do Kernel**
Namespaces	Abstrai um recurso real (CPU, memória, disco, etc).
CGroups	Aloca recursos para um processo (ou grupo de processos).
Perfis Apparmor	Controlam as permissões de aplicações no acesso a arquivos.
Políticas Seccomp	Controlam quais chamadas de sistema podem ser feitas por um processo.
CHroots	Operação que altera a localização do diretório raiz de um processo.

2.1.2 Contêineres versus Máquinas Virtuais

A virtualização é utilizada em centros de processamento de dados e ambientes de computação em nuvem com a finalidade de desacoplar as aplicações do *hardware* na qual os serviços são executados(NANDA; CHIUEH, 2005). Nesses meios, a virtualização é empregada, tipicamente, através de máquinas virtuais. Assim, devido às semelhanças entre contêineres e máquinas virtuais, alguns pesquisadores têm se dedicado a investigar o impacto em desempenho que cada tipo de virtualização pode gerar.

A virtualização de hardware utilizada por máquinas virtuais é capaz de virtualizar um Sistema Operacional por completo. Para tal, ela necessita de uma camada de software chamada de Hypervisor, que por sua vez é responsável por criar e controlar as VMs (Máquinas Virtuais), assim como executar os sistemas convidados. São exemplos de tecnologias que utilizam virtualização de hardware: KVM, Xen, VMWare ESX, Oracle Virtual Box.

Em contraste às VMs, os contêineres são implementados através de uma técnica de virtualização leve, detalhado na seção 2.1.1. Atualmente, o Docker é um dos sistemas mais difundidos entre as soluções disponíveis para criação de contêineres. Porém, existem outros softwares, como: LXC, BSD, Windows Containers. Outro ponto importante é que as VMs e os contêineres precisam, em muitas situações, cooperar e atuar em conjunto para o provisionamento de serviços. Dessa forma, surgiram aplicações para controle e gerenciamento dessas tecnologias. Para o gerenciamento de VMs, existe o OpenStack, CloudStack, VMware vCloud e para contêineres existe o Kubernetes e o Docker Swarm.

Em relação ao desempenho de VMs, (YOUNGE et al., 2011) chegou à conclusão de que entre KVM, Xen e VirtualBox, a melhor escolha para aplicações que exigem alto desempenho é o KVM, pois foi o sistema que apresentou os melhores resultados entre os *benchmarks* testados (CPU, memória, disco, rede). Na comparação entre os dois tipos de virtualização, (Felter et al., 2015) verificou as diferenças entre o KVM e o Docker, concluindo que ambas não causam

sobrecargas significativas em aplicações que utilizam CPU e memória, porém causavam grande sobrecarga em aplicações que realizavam muitas operações de entrada e saída, como acesso a disco constante. Para tais aplicações, o Docker possui a opção de utilizar "volumes" o que diminui significativamente a sobrecarga, se comparado aos níveis causados pelo KVM.

Ainda, (XAVIER et al., 2013) mostrou que o Docker possui níveis de desempenho de CPU, memória, disco e rede comparáveis aos níveis de um sistema nativo. Por outro lado, a utilização do Xen, para execução dos mesmos experimentos, apresentou um *overhead* superior em todos os cenários propostos, influenciando em mais de 30% na taxa de transferência da memória. Apesar de todos os benefícios da utilização de contêineres evidenciados, existem pontos que as VMs ainda são melhores, como na segurança e no isolamento entre ambiente virtualizado e o sistema hospedeiro (Combe; Martin; Di Pietro, 2016).

2.1.3 Segurança de Contêineres

Na virtualização em nível de S.O, o Kernel é responsável por executar tanto as chamadas do sistema hospedeiro quanto dos contêineres executando os sistemas convidados. Quando comparados às VMs, os contêineres possuem uma camada de isolamento menor entre as aplicações que executam no ambiente virtualizado e o sistema nativo. Quando uma máquina virtual maliciosa tenta utilizar um determinado recurso de maneira indevida, seja por sobrecarga ou por acesso não permitido, o Hypervisor pode detectar tal ação e tomar atitudes. Já um contêiner malicioso pode sobrecarregar o Kernel com chamadas e impossibilitar que as rotinas do sistema nativo sejam executadas.

Por tais motivos, é importante configurar adequadamente os recursos aos quais os contêineres possuem acesso. Em (Combe; Martin; Di Pietro, 2016) é ressaltado que os contêineres possuem problemas em relação ao isolamento e segurança, porém a maioria desses problemas são diminuídos quando os mesmos são utilizados da forma como foram projetados, ou seja, como microsserviços. Os contêineres devem executar tarefas simples e únicas e rotinas administrativas devem ser responsabilidade do sistema hospedeiro. Portanto, configurar um contêiner como um sistema completo (algo como uma VM executando vários serviços) pode abrir brechas de segurança, uma vez que as configurações padrões de segurança e isolamento do Docker não estão preparadas para esse tipo de uso.

2.1.4 Ferramentas para Virtualização de Sistemas

Conforme citado anteriormente, existem diversas implementações de contêineres e dentre as mais famosas e utilizadas estão o Docker, Linux Containers (LXC) e OpenVZ (Open Virtuozzo). Todas essas ferramentas são *open-sources* e representam aplicações que utilizam virtualização em nível de Sistema Operacional. O Docker e o LXC utilizam estratégias parecidas, fornecendo uma API simples para que usuários consigam instanciar e gerenciar contêineres. Umas das diferenças é que o LXC permite que contêineres sejam vistos com um substituto as máquinas virtuais, já os contêineres do Docker encapsulam microsserviços que fazem parte de um grande serviço. Em contraste, o OpenVZ pode criar vários contêineres isolados em que cada um roda como servidor *stand-alone* conhecidos como VPSs (*Virtual Private Servers*).

Quando se quer executar partes de um Workflow Científico dentro de um contêiner, é necessário que exista uma maneira de gerenciar todas as tarefas que devem ser criadas, possibilitando que as mesmas sejam inicializadas, monitoradas e terminadas em diferentes hospedeiros. A maioria dessas funcionalidades citadas são implementadas por ferramentas chamadas de orquestradores. Os orquestradores são responsáveis por gerenciar todo o ciclo de vida dos contêineres. Dessa maneira, o suporte a softwares que permitam orquestração é um dos pontos que devem ser avaliados quando se deseja escolher uma tecnologia de contêiner para integrar a workflows científicos.

Algumas ferramentas utilizadas para orquestração de contêineres são o Docker Swarm, Google Kubernetes e Mesos. Todas as ferramentas possuem suporte aos contêineres do Docker. Pode-se dizer, inclusive, que as mesmas foram projetadas para funcionar com o Docker. Quanto ao LXC, existem discussões em fóruns que argumentam que deveriam existir orquestradores para a plataforma, pois simplificariam processos e permitiriam que o LXC se tornasse tão popular quanto o Docker. Os orquestradores citados não suportam o OpenVZ, porém essa tecnologia possui uma solução própria, com uma infraestrutura capaz de gerenciar o ciclo de vida dos contêineres.

Outro ponto importante que deve ser destacado é o suporte dos desenvolvedores para as ferramentas de contêineres. Conforme mais usuários da ferramenta surgem, a mesma é colocada a prova em diferentes cenários, assim novos *bugs* são encontrados e mais importante: novas funcionalidades são discutidas. Dessa forma, um suporte ativo dos desenvolvedores garante que *bugs* sejam corrigidos e que as reivindicações da comunidade sejam atendidas. Avaliando-

se os repositórios dos códigos[6] de cada ferramenta citada foi constatado que os desenvolvedores estão ativos e as últimas submissões de códigos são recentes.

Além dos próprios desenvolvedores gerarem documentações para suas ferramentas, a comunidade de usuários também pode criar seus próprios tutorias e soluções para diferentes contextos, assim contribuindo para que novos projetos surjam ou, simplesmente, para que dúvidas sobre todos os aspectos do software sejam sanadas. A fim de medir a popularidade de cada ferramenta, foi feita uma pesquisa sobre o número de perguntas feitas pela comunidade no *website* StackOverflow[7], um dos sites mais consagrados onde usuários podem discutir diversos tópicos sobre temas que envolvem a computação. Outra métrica utilizada foi a pesquisa sobre o número de repositórios criados que utilizam alguma dessas ferramentas presentes nas plataformas GitHub[8], BitBucket[9] e GitLab[10], sendo essas plataformas populares onde novos projetos de *software* podem ser criados e compartilhados. A tabela 2.2 mostra os números obtidos durante a pesquisa.

Tabela 2.2 – Popularidade das Ferramentas de Contêineres

Pesquisa	**Docker**	**LXC**	**OpenVZ**
Repositórios no GitHub	415.415	1.181	427
Repositórios no BitBucket	8.864	42	11
Projetos no GitLab	Mais de 100	Mais de 100	7
Perguntas em que o nome da ferramenta aparecia no título	500	287	76
Perguntas com no mínimo uma resposta	500	211	63
Percentual de perguntas respondidas	100%	50%	36%

Workflows Científicos possuem dependências diversas de bibliotecas, códigos e ferramentas. Logo, é comum que seja necessário instalar vários pacotes antes da execução do workflow. Os gerenciadores de pacotes presentes nos sistemas Unix facilitam essa tarefa, pois os mesmos automatizam o processo de instalação, atualização e remoção de pacotes. Porém, essa facilidade pode limitar a execução de um workflow ao uso de uma certa distribuição, pois pacotes podem estar indisponíveis para certas versões. Assim, para que o processo de configuração seja facilitado, é importante que as ferramentas de conteinerização possuam suporte a diver-

[6] Repositório do Docker: https://github.com/docker/docker-ce
Repositório do LXC: https://github.com/lxc/lxc
Repositório do OpenVZ: https://github.com/OpenVZ
[7] https://stackoverflow.com/ - acessado em 04/11/2019.
[8] https://github.com/ - acessado em 04/11/2019.
[9] https://bitbucket.org/ - acessado em 04/11/2019.
[10] https://gitlab.com/ - acessado em 04/11/2019.

sas distribuições e imagens. O Docker possui o Docker Hub [11], uma biblioteca de imagens de contêineres, onde a comunidade pode compartilhar imagens que já encapsulam uma camada de software. No Docker Hub também são disponibilizadas imagens com suportes para diferentes arquiteturas e sistemas operacionais. Para o LXC, existe um servidor [12] que disponibiliza várias imagens que podem ser criadas a partir de modelos produzidos pela comunidade, mas essas imagens são disponibilizadas sem nenhuma garantia de segurança. Para o OpenVZ existe um repositório oficial [13] com algumas imagens, porém as últimas modificações foram feitas em 2016.

A tabela 2.3 resume os principais pontos avaliados para a escolha de uma ferramenta de contêiner. Quando é feita uma análise geral dos pontos colocados em questão, percebe-se que o Docker é uma ferramenta consolidada que apresenta uma série de vantagens quando comparada ao LXC e ao OpenVZ. Consequentemente, o Docker foi escolhido como a ferramenta responsável por virtualizar os sistemas dentro da solução proposta pelo Sciflow-Bridge.

Tabela 2.3 – Comparação entre as Tecnologias de Contêineres

Aspecto avaliado	**Docker**	**LXC**	**OpenVZ**
A. Suporte ativo dos desenvolvedores?	Sim	Sim	Sim
B. Suportado por algum orquestrador?	Sim	Não	Não
C. Número de imagens disponibilizadas.	Mais de 2.000.000	288	163

2.2 Workflows Científicos

Os avanços na tecnologia e a popularização dos computadores trouxeram agilidade e facilidade aos cientistas de todas as áreas de pesquisa. A NASA, por exemplo, já utilizava computadores em quatro dos seus primeiros programas espaciais: Gemini, Apollo, Skylab, e Shuttle. Os computadores executavam várias tarefas no programa espacial como controle de parâmetros e cálculos de reentrada, mas uma das tarefas mais importantes eram as rotinas de simulação. O Gemini era responsável por calcular três níveis de simulações: sistema de validação de equações, simulação man-in-the-loop que ajudava a definir requisitos, procedimentos e exibições e por fim simulações que caracterizavam o desempenho do software (TOMAYKO, 1988). O uso dos computadores nos programas espaciais contribuiu para o avanço na ciência e a definição dos fluxos de simulações economizaram tempo e impulsionaram a pesquisa.

[11] Docker Hub: https://hub.docker.com/search?q=&type=image
[12] Servidor LXC: https://us.images.linuxcontainers.org/
[13] Repositório OpenVZ: https://download.openvz.org/template/precreated/

Em trabalhos científicos onde análises e simulações são feitas, existe o desafio de projetar e executar as várias tarefas que devem atuar sobre os dados. Há um grande interesse em automatizar esse processo, pois o mesmo pode tomar uma quantia de tempo significativa na elaboração dos projetos. Dessa forma, a definição do conjunto de tarefas e a ordem que as mesmas devem executar formam um fluxo de trabalho (workflow). De maneira geral, um workflow descreve como processos devem ser coordenados para atingirem um objetivo comum (BARKER; HEMERT, 2008), e essa definição pode ajudar no processo de automatização de experimentos.

A utilização de workflows para descrever uma sequência de atividades teve origem no ramo de negócios. Nesse modelo, a execução é orientada ao controle com padrões e eventos definidos. Já no modelo científico, a execução de workflows é orientada ao fluxo de dados, uma vez que a obtenção e tratamento de dados são partes importantes de uma pesquisa. Assim, um workflow científico tenta capturar uma série de passos analíticos, os quais descrevem o processo de experimentos computacionais (BARKER; HEMERT, 2008).

As diferenças entre esses dois tipos de workflows são evidenciadas quando uma análise sobre as dependências é feita. Na indústria, as dependências entre os processos são apenas de controle, por outro lado, no meio acadêmico as dependências são mais abrangentes e se referem tanto a controle quanto a dados(Yildiz; Guabtni; Ngu, 2009). Na condução de experimentos, tarefas dependem de dados que precisam ser pré-processados, agrupados e particionados por outras tarefas. Desse modo, a descrição dessas dependências é algo importante e que deve ser considerado na construção de workflows científicos.

Experimentos são feitos com o intuito de comprovar ou refutar hipóteses. Nesses termos, as tarefas e processos que compõem esses experimentos podem ser modificados e adaptados à medida que a pesquisa evolui. Portanto, é importante que as ferramentas utilizadas para determinação de um workflow científico permitam que o mesmo possa ser modificado ao longo do tempo, contribuindo para o andamento da pesquisa e proporcionando que novas descobertas sejam feitas.

2.2.1 Estruturas básicas de um Workflow Científico

O trabalho de (Bharathi et al., 2008) focou em caracterizar as estruturas básicas presentes em um workflow científico, onde foram definidas cinco estruturas básicas: processo, *pipeline*, distribuição de dados, agregação de dados e redistribuição de dados. A figura 2.1 representa graficamente cada estrutura citada. O **processo** é a estrutura mais simples que recebe uma

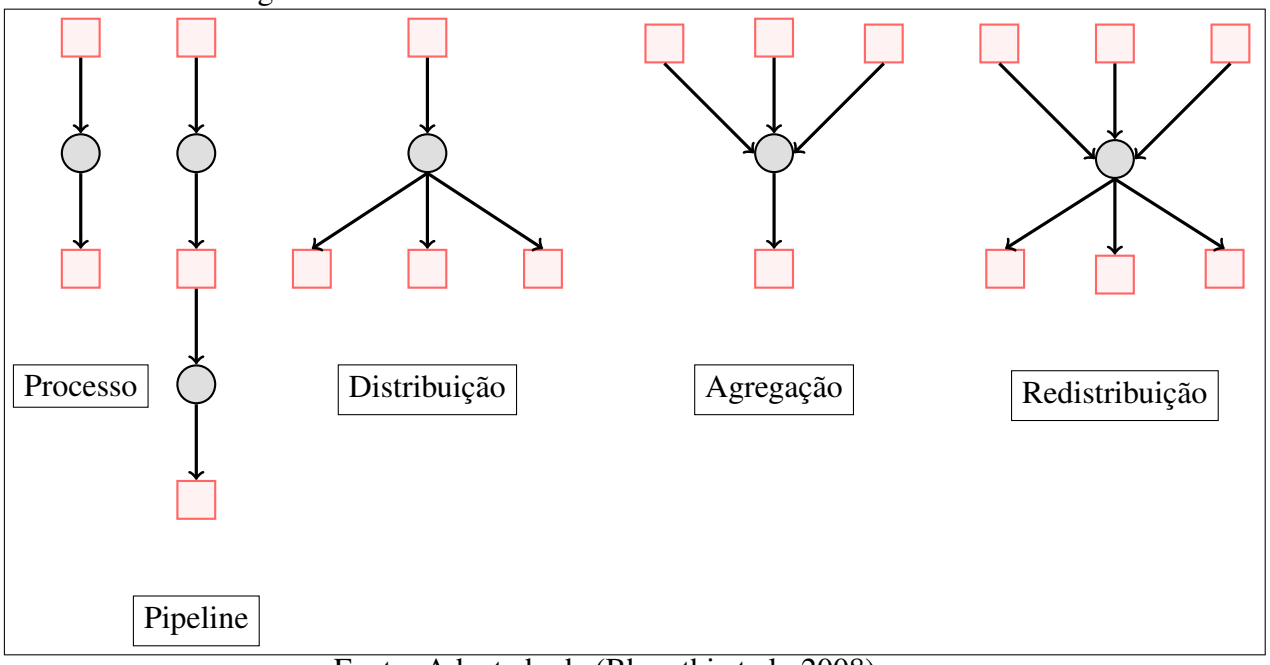
entrada, realiza um processamento e gera uma saída. O ***pipeline*** consiste em dois ou mais processos encadeados de forma sequencial, de forma que um processo atua sobre os dados gerados por outro processo. Na **distribuição dos dados**, uma tarefa recebe uma entrada e produz várias saídas. Esse tipo de estrutura é utilizada quando se tem uma tarefa que produz saídas que são consumidas por várias outras tarefas ou quando se quer dividir uma entrada para várias tarefas. Já a **agregação de dados** pode ser vista como o processo contrário da distribuição, onde várias entradas são processadas e somente uma saída é gerada. A agregação de dados é comum nos estágios finais dos fluxos de trabalhos, onde se deseja coletar e reunir todas as saídas geradas anteriormente. Por último, a redistribuição de dados é uma estrutura que recebe várias entradas e gera várias saídas, pode ser vista como um ponto de sincronização do sistema.

2.2.2 Infraestruturas para Execução de Workflows Científicos

Devido a grande quantidade de tarefas presentes em workflows científicos e a demanda por processamento de cada tarefa, esses fluxos são executados em Sistemas Distribuídos. "*Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente*"(TANENBAUM; STEEN, 2006). Dessa forma, as tarefas podem ser distribuídas nesses computadores para serem executadas em paralelo, quando a dependência de dados permite. Os vários computadores que fazem parte desse sistema também podem ser chamados de nós. Os workflows científicos podem ser executados em sistemas distribuídos como Clusters, Grids e Clouds(ZHENG; THAIN, 2015).

Segundo (TANENBAUM; STEEN, 2006), os termos clusters e grids podem ser caracterizados da seguinte forma: Cluster - é um conjunto de computadores semelhantes, que estão conectados por meio de uma rede local de alta velocidade e que cooperam para a execução de uma tarefa. Grid - é um sistema distribuído mais heterogêneo, onde os computadores presentes podem estar sob condições de rede distintas e diferir significativamente em termos de hardware. Já a Computação em Nuvem (Cloud Computing) pode ser vista como o provisionamento de um serviço de computação, onde existe uma infraestrutura de computadores por trás desse serviço, frequentemente pago com base na utilização dos recursos disponíveis (COULOURIS et al., 2011).

Figura 2.1 – Estruturas básicas de um Workflow Científico.



Fonte: Adaptado de (Bharathi et al., 2008).

2.2.3 Engines de Workflow Científico

Sistemas de Workflow científico oferecem um ambiente para que os fluxos sejam modelados e, posteriormente, executados sobre recursos computacionais distribuídos. Dentre os principais sistemas que surgiram, destacam-se o Kepler(Altintas et al., 2004), Taverna(OINN et al., 2004) e Triana(Majithia et al., 2004). Alguns sistemas de workflows possibilitam que os fluxos sejam definidos por meio de interfaces gráficas, e apesar disso, o nível de abstração pode ser muito baixo para cientistas que não possuem conhecimentos avançados em computação (BARKER; HEMERT, 2008). Nesses termos, os sistemas de workflows devem levar em consideração pontos como claridade dos fluxos, reusabilidade e flexibilidade para que os autores consigam executar o modelo e realizar alterações(MCPHILLIPS et al., 2009).

O Kepler é um sistema open-source, implementado com a linguagem Java e desenvolvido pela UC Berkeley em conjunto com o San Diego Supercomputer Center. O Kepler utiliza atores para a definição de fluxos de trabalhos. E eles são responsáveis por coletar um conjunto de dados de portas de entradas, processá-los e enviá-los para outros atores através de portas de saídas. Na elaboração de fluxos complexos, diversos atores são conectados para descrever as tarefas que devem ser executadas. O Kepler utiliza um formato XML para salvar os workflows gerados e permite, ainda, que novos atores sejam adicionados através do desenvolvimento de novas classes Java.

O Taverna também foi desenvolvido com a linguagem Java e o seu principal objetivo é satisfazer as necessidades dos profissionais da bioinformática que necessitam construir workflows científicos que executam sobre serviços web remotos (Curcin; Ghanem, 2008). O Taverna utiliza o conceito de processadores como unidades básicas que atuam sobre dados de entrada e geram saídas. A linguagem SCUFL(*Simple Conceptual Unified Flow Language*)(OINN et al., 2004) é utilizada para representar os fluxos de trabalho com grafos acíclicos diretamente conectados. E com essa definição feita, é possível executar as tarefas em recursos distribuídos através de serviços Web.

O Triana é um sistema desenvolvido pela universidade de Cardiff, *open-source* e desenvolvido com Java. Esse sistema consegue distribuir seções de um workflow para serem executados em máquinas remotas através de uma rede *peer-to-peer* (Hongbiao; Feng; Wanjun, 2010). O Triana utiliza componentes básicos chamados de unidade, cada unidade pode ser conectada a cabos de entrada e saída de dados por onde as dependências são enviadas e recebidas. Diversas são as linguagens de descrição de workflows suportadas pelo Triana, como WSFL(LEYMANN, 2001), DAG(Grafos Acíclicos Dirigidos), BPEL(WASSERMANN et al., 2007) e Petrinet(HACK, 1976), assim mostrando-se mais flexível do que os outros sistemas apresentados.

Na investigação de trabalhos relacionados, verificou-se que na integração de workflows a contêineres utilizou-se um sistema chamado de Makeflow(ALBRECHT et al., 2012). A seção 2.3 aborda os trabalhos que fizeram essa integração de forma mais detalhada. O Makeflow utiliza uma sintaxe para definição de workflows similar aos arquivos Makefiles, onde comandos que representam tarefas são definidos. Na determinação de cada tarefa deve ser informado quais os arquivos de entrada, necessários para a execução do comando e quais os arquivos de saída serão gerados pela tarefa. As tarefas do workflow definido com o Makeflow podem ser executadas em diferentes plataformas como serviços de *cloud* e sistemas *batch* como SLURM(YOO; JETTE; GRONDONA, 2003), Work Queue(ALBRECHT et al., 2012), HTCondor(Raman; Livny; Solomon, 1998) e outros. Ainda, o Makeflow diz ser capaz de gerenciar milhões de tarefas rodando em milhares de máquinas e dispõe de uma arquitetura capaz de se recuperar de falhas ao longo da execução de fluxos de trabalhos.

2.2.4 Exemplos de Workflows Científicos

Existem workflows científicos para várias áreas, a bioinformática, por exemplo, possui alguns exemplos de fluxos de trabalhos relevantes como o BLAST(DONKOR E. S., 1990), HECIL(CHOUDHURY; CHAKRABARTY; EMRICH, 2017) e o BWA(LI; DURBIN, 2009). O BLAST é uma aplicação que compara as semelhanças de uma proteína de referência ou uma sequência de DNA contra várias outras sequências através de um método heurístico, e retorna as correspondências semelhantes a amostra testada. Mais especificamente, o BLAST tenta localizar todas as palavras comuns de 3 letras entre a sequência de interesse e as demais sequências (DONKOR E. S., 2014). Esse processo de alinhamento de sequências, realizado pelo BLAST, é importante para identificar similaridades que podem ser um resultado de relações estruturais, funcionais ou evolucionárias. Já o HECIL atua no sequenciamento de genomas, mais especificamente na correção de erros de leituras longas sequenciamento de genomas. Uma abordagem de aprendizagem iterativa é utilizada para corrigir possíveis fragmentações geradas pelo sequenciamento de genomas.

2.3 Integração entre Contêineres e Workflows

Os trabalhos realizados a respeito do desempenho de contêineres despertaram nos pesquisadores a ideia de solucionar os problemas de dependências dos workflows científicos com o provisionamento de ambientes por meio da virtualização em nível de S.O. Em (SWEENEY; THAIN, 2018) é feito uma análise de como os contêineres devem ser integrados a Workflows Científicos de uma maneira eficiente. Eles destacam que os contêineres não devem ser utilizados como uma grande caixa onde todas as bibliotecas, dados das aplicações e executáveis devem ser depositados, pois essa estratégia, apesar de simples, é ineficiente já que uma grande quantidade de dados irrelevantes são transferidos para cada nó do sistema.

2.3.1 Por que utilizar contêineres com Workflows?

As várias tarefas executadas em um workflow científico possuem diversas dependências de bibliotecas, executáveis, dados de entrada, entre outros. Essa característica faz com que um workflow seja fortemente dependente do ambiente no qual é executado, as tarefas desse fluxo necessitam de várias camadas de dependências para sua correta execução. A virtualização oferecida pelos contêineres, permite que uma imagem de contêiner encapsule todas as depen-

dências necessárias para que as tarefas consigam ser executadas. Deste modo, os fluxos de trabalhos científicos passam a ficar atrelados somente às imagens de contêineres que utilizam, e não mais a todo um sistema composto por *hardware* e *software*. Tais imagens podem ser movidas de um servidor para o outro, garantindo uma maior flexibilidade. Dessa forma, um cientista interessado em rodar algum workflow pode baixar o contêiner e já executar as tarefas, sem se preocupar com a configuração, por vezes extensa, de todo um ambiente.

Além disso, as imagens de contêineres possuem um sistema de camadas, onde a cada modificação adicionada à imagem é posta sobre a camada anterior. Assim, novas bibliotecas podem ser testadas, e caso alguma venha a corromper o ambiente de execução, é fácil retornar ao estado anterior onde o ambiente estava estável. Portanto a integração de contêineres a Workflows científicos proporcionam um meio para virtualizar ambientes, dão uma maior flexibilidade e facilitam a reprodutibilidade.

2.3.2 Onde os contêineres se encaixam dentro do Workflow?

Tipicamente, as tarefas em um workflow científico são executadas por nós em um sistema distribuído. Esse sistema é composto por nós trabalhadores e nós gerentes. Os nós gerentes conhecem todas as tarefas que fazem parte do workflow e assim, as distribui entre os nós trabalhadores. O nó trabalhador recebe as tarefas do gerente, as executa e envia os resultados de volta ao gerente. Dessa forma, os contêineres podem ser posicionados de duas formas. Na primeira, cada nó trabalhador é na verdade um contêiner que executa as tarefas atribuídas pelo gerente. Na segunda, o nó trabalhador lança contêineres para cada tarefa recebida pelo gerente. Quando as tarefas são executadas dentro de contêineres, garante-se uma maior segurança, uma vez que cada tarefa executa isoladamente no seu próprio contêiner(SWEENEY; THAIN, 2018). Porém, instanciar um contêiner também requer recursos de processamento e dependendo da quantidade de tarefas exigidas pelo workflow, esse procedimento pode se tornar um gargalo no sistema.

2.3.3 Resumindo

Portanto, os workflows científicos apresentam diversas dependências e os contêineres se apresentam como uma virtualização leve capaz de encapsular tais dependências. Diversos trabalhos foram feitos à respeito da integração de contêineres e workflows científicos (ZHENG; THAIN, 2015), (SWEENEY; THAIN, 2018), (ALBRECHT et al., 2012) e com eles é possível perceber que essa integração garante uma maior flexibilidade para os fluxos. No entanto,

tal integração deve ser feita levando em consideração alguns aspectos como a disposição dos contêineres no sistema, o envio das imagens de contêineres entre os nós do sistema, a gerência dos contêineres, uso de *cache*, entre outros. No decorrer deste trabalho, tais aspectos serão abordados e estratégias serão definidas para que uma integração eficiente seja feita.

3 A FERRAMENTA SCIFLOW-BRIDGE

Este capítulo trata de diversos aspectos do planejamento e desenvolvimento da ferramenta Sciflow-Bridge. Primeiramente são definidos os pré-requisitos que a solução deve atender e posteriormente são relatadas questões de implementação da ferramenta.

3.1 Definição dos pré-requisitos

A execução de diversas tarefas em um ambiente distribuído, que é a essência da execução de um Workflow Científico, pode gerar diversos problemas referentes à configuração de ambientes, da rede e dos nós presentes no sistema. Deste modo, a execução de workflows abrange uma grande variedade de tópicos e a definição de uma série de pré-requisitos é importante, pois define um escopo no qual a ferramenta deve proporcionar soluções.

De forma geral, a ferramenta Sciflow-Bridge busca atender os seguintes pré-requisitos:

1. Analisar quais são as dependências de cada tarefa executada pelo workflow.
2. Gerenciar as dependências de entrada e saída de dados e as dependências implícitas como bibliotecas e executáveis.
3. Transferir as dependências entre os nós do sistema de forma eficiente.
4. Executar um workflow científico definido através de um modelo.
5. Definir uma estratégia para a execução de um workflow utilizando os benefícios oferecidos pelos contêineres.
6. Gerenciar as tarefas geradas pelo workflow científico.
7. Gerenciar os contêineres que fazem parte do sistema.
8. Utilizar padrões existentes para a definição de um workflow científico e para a definição das configurações do sistema.

3.2 Desenvolvimento da Ferramenta

A presente seção apresenta o que foi desenvolvido para atender a cada um dos pré-requisitos estabelecidos e também demonstra quais ferramentas de software foram utilizadas

para solucionar os problemas no gerenciamento de dependências em workflows científicos.

3.2.1 Análise de Dependências

A execução de workflows científicos é orientada a dados, onde cada tarefa atua sobre um conjunto de dados e produz um conjunto de saída. Para que cada tarefa consiga processar os dados de entrada são desenvolvidos *scripts* e bibliotecas que leem os dados e realizam operações de acordo com o objetivo de cada pesquisa. Assim, cada tarefa depende de uma série de arquivos, como arquivos de entrada, arquivos de configuração, bibliotecas e programas de terceiros, entre outros. E para que cada tarefa consiga executar corretamente, um ambiente contendo todas as dependências deve ser configurado. No entanto, a criação desses ambientes é uma tarefa custosa e que muitas vezes exige conhecimentos técnicos que nem todo pesquisador possui.

Existem, basicamente, dois tipos de dependências dentro dos fluxos de trabalho científicos. O primeiro tipo se refere ao conjunto de dados de entrada, que cada tarefa recebe e deve processar, e o conjunto de dados de saída gerados pela mesma. O segundo tipo, são as dependências implícitas de cada tarefa, ou seja, dependências que devem estar presente no sistema, mas que não fazem parte do workflow, como códigos fonte, arquivos de configuração e bibliotecas que devem ser carregadas durante a execução da tarefa. Por exemplo, uma tarefa básica que lê um arquivo com números inteiros e calcula a soma total, possui como dependências implícitas o código fonte do programa, o compilador da linguagem utilizada, as bibliotecas básicas da linguagem e arquivos de configuração do sistema.

Nesses termos, cada tarefa possui seu conjunto de dependências implícitas e de entrada/saída, e cada ambiente onde se deseja executar tal tarefa precisa conter este conjunto. O processo de analisar e definir quais são as dependências de cada tarefa é um passo importante para a definição dos ambientes que devem rodá-las. Esta etapa pode ser feita manualmente, porém é algo trabalhoso de ser feito e que exige um tempo significativo. Consequentemente, automatizar esse processo economiza tempo e recursos. Existem ferramentas para analisar a árvore de dependência em códigos fontes, como o CppDepend[14] para C e C++, Snakefood[15] e Modulefinder[16] para Python, Jdeps[17] para Java. Porém cada ferramenta é específica para cada

[14] CppDepend: https://www.cppdepend.com/
[15] SnakeFood: https://pypi.org/project/snakefood/
[16] Modulefinder: https://docs.python.org/3/library/modulefinder.html
[17] Jdeps: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html

linguagem, e em um workflow científico diversas são as linguagens utilizadas, logo é necessário prover uma solução mais abrangente.

Em resumo, uma tarefa é simplesmente a execução de um código ou um *script*, e analisar as dependências de uma tarefa se resume em detectar quais arquivos foram necessários para a execução da mesma. Nos sistemas Linux, uma das formas de verificar quais arquivos foram abertos e verificados durante a execução de um programa é através do uso do utilitário Strace. O Strace é utilizado para monitorar e interferir com as interações entre um processo e o Kernel do Linux, essas interações incluem chamadas de sistema, entregas de sinal e trocas no estado do processo (STRACE, 2019 (acesso em Setembro, 2019). O Strace utiliza uma funcionalidade do Kernel chamada de Ptrace (*Process Trace*) que provê um meio no qual um processo pode observar e controlar a execução de outro processo, e é geralmente utilizado para implementar depuradores de código (PTRACE, 2019 (acesso em Setembro, 2019).

Assim, foi criado um programa, escrito na linguagem Python, que utiliza o Strace para capturar as chamadas feitas pela tarefa e realizar a análise das dependências. Tal programa faz a filtragem das seguintes chamadas de sistema: execve – que executa um programa, openat – abre um arquivo relativo ao diretório informado, stat – retorna um conjunto de informações sobre algum arquivo, comumente utilizada para a verificar a presença de um determinado arquivo. O programa ainda verifica se o sinal retornado pelas chamadas indica um caso de sucesso. Após a etapa de filtragem, gera-se uma lista com todos os arquivos que foram abertos, verificados ou executados.

Ao final desse processo, uma lista com todos os arquivos no qual a aplicação depende é gerada. Essa lista facilita o processo de construção de uma imagem de contêiner capaz de executar tal aplicação. Uma vez que se conhece todas os arquivos utilizados, a procura por bibliotecas ou pacotes faltantes é facilitada. A figura 3.1 mostra a árvore de dependências gerada pelo analisador para uma das tarefas do workflow Blast. Com esta lista em mãos, o cientista ou a pessoa encarregada de criar a imagem de contêiner onde as tarefas irão executar, pode utilizar um utilitário do Linux como o apt-file, que lista todos os pacotes que possuem determinados arquivos. A figura 3.2 mostra os vários pacotes retornados pelo apt-file, quando um dos arquivos apontados pelo analisador é pesquisado.

Figura 3.1 – Árvore de dependências gerada pelo analisador.

```
root@container: $tree dep
dep
|__ blastall
|__ etc
|   |-- ld.so.cache
|   |-- nsswitch.conf
|   \-- passwd
|__ lib
|   \-- x86_64-linux-gnu
|       |-- libc.so.6
|       |-- libm.so.6
|       |-- libnsl.so.1
|       |-- libnss_compat.so.2
|       |-- libnss_files.so.2
|       |-- libnss_nis.so.2
|       \-- libpthread.so.0
|__ nt
|   |-- nt.44.nhr
|   |-- nt.44.nin
|   |-- nt.44.nnd
|   |-- nt.44.nni
|   |-- nt.44.nsd
|   |-- nt.44.nsi
|   |-- nt.44.nsq
|   \-- nt.nal
\__ small.fasta.0

4 directories, 20 files
```

3.2.2 Criação de Ambientes Conteinerizados

Os contêineres são uma forma leve de virtualização e representam uma boa solução quando se quer provisionar um ambiente pronto e configurado para receber uma determinada aplicação. Inclusive, muitas ferramentas estão disponibilizando imagens de contêineres com suas soluções instaladas e prontas para o uso, assim diminuindo o tempo com que o usuário gastaria com a instalação. Esse é o caso de ferramentas como: Postgres, Nginx, Mysql, Java JRE Server. Assim, quando tarefas dentro de um workflow necessitam de um ambiente consolidado para sua execução, os contêineres apresentam-se como uma possível solução.

O presente trabalho propõe que, para a criação desse ambiente sólido, seja escolhida uma imagem de contêiner base, a qual pode ser escolhida de acordo com os pacotes utilizados

Figura 3.2 – Saída do Utilitário apt-file.

```
root@container:~$ apt-file find libc.so.6
libc6: /lib/x86_64-linux-gnu/libc.so.6
libc6-amd64-cross: /usr/x86_64-linux-gnu/lib/libc.so.6
libc6-amd64-i386-cross: /usr/i686-linux-gnu/lib64/libc.so.6
libc6-amd64-x32-cross: /usr/x86_64-linux-gnux32/lib64/libc.so.6
libc6-arm64-cross: /usr/aarch64-linux-gnu/lib/libc.so.6
libc6-armel-cross: /usr/arm-linux-gnueabi/lib/libc.so.6
libc6-armel-cross: /usr/arm-linux-gnueabihf/libsf/libc.so.6
libc6-armhf-cross: /usr/arm-linux-gnueabi/libhf/libc.so.6
libc6-armhf-cross: /usr/arm-linux-gnueabihf/lib/libc.so.6
libc6-hppa-cross: /usr/hppa-linux-gnu/lib/libc.so.6
libc6-i386: /lib32/libc.so.6
```

pelas tarefas. Esta imagem deverá estar presente em todos os nós em que se deseja executar as tarefas dos workflows. Assim, essas imagens devem conter também todas as dependências implícitas e, portanto, um técnico ou um pesquisador capacitado deve escolher uma imagem base e após isso instalar todas as ferramentas e bibliotecas utilizadas pelas tarefas presentes no fluxo de trabalho científico.

3.2.3 Transferência de Imagens de Contêineres

O Docker proporciona algumas maneiras para transferir imagens de contêineres pela rede. A primeira, é simplesmente salvar a imagem como um arquivo compactado no formato *tar* e transferi-la para os outros nós. Porém, é importante considerar os custos para exportação e importação da imagem, além é claro o custo do envio pela rede. Estes custos podem ser tornar altos, pois toda vez que se deseja enviar o arquivo *tar*, a imagem de contêiner por completo é exportada, enviada e importada. Outra maneira, é criar um Dockerfile, que pode ser visto como um arquivo que possui uma receita para a criação da imagem. Desta maneira o uso da rede é mínimo no envio de tal arquivo. Porém, ao receber o Dockerfile cada nó do sistema deve realizar o *download* da imagem base e em seguida instalar os pacotes utilizados na imagem e realizar possíveis configurações, o que pode gerar uma sobrecarga na rede quando muitos nós tentam realizar o *download* simultaneamente.

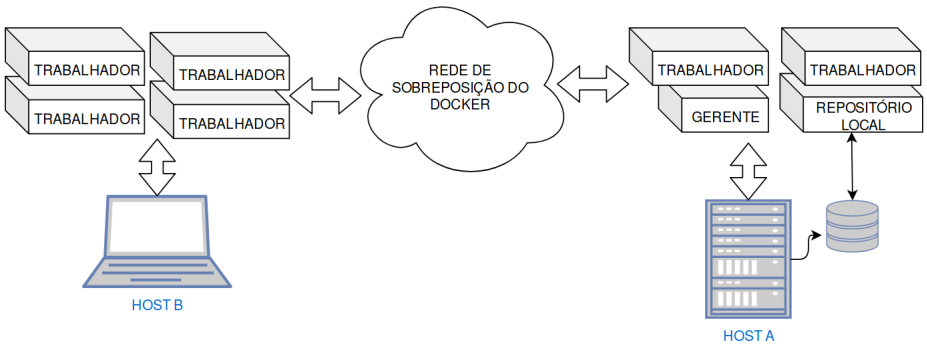
Dessa forma, transferir imagens de contêineres pela rede envolve um custo significativo e pode representar um gargalo no sistema. Assim, a solução proposta pelo presente trabalho é que junto ao nó gerente do sistema, exista um contêiner que atue como um repositório local

de imagens. Assim, um nó trabalhador antes de executar uma tarefa, deve verificar se possui a imagem de contêiner para tal. Se não a possuir, o nó trabalhador deve baixar a imagem do repositório local. Essa estratégia é interessante, pois permite que as imagens de contêineres sejam atualizadas nos nós trabalhadores de maneira eficiente. Uma vez que as imagens do Docker são organizadas por camadas, e ao baixar a imagem do repositório local, o nó trabalhador deve baixar somente as camadas que ainda não possui. Portanto, o gerenciamento das dependências implícitas é resolvido por meio das imagens de contêineres, um repositório local e envio de camadas de contêineres gerenciados pelo Docker. Além disso, o processo de execução de workflows é iterativo, uma vez que os cientistas vão adicionando ou removendo tarefas ao fluxo, á medida que o trabalho avança, assim é interessante prover esse meio pelo qual as imagens de contêineres podem ser atualizadas nos nós trabalhadores.

3.2.4 Conectando *hosts* com o Docker Swarm

A execução do workflow, no cenário proposto, pode ser visto como um grande serviço que possui várias tarefas cooperando para chegar a algum objetivo. Para que as mesmas consigam trocar informações e serem coordenadas, é imprescindível que exista uma forma de comunicação. Além disso, quando tarefas são executadas em um ambiente distribuído, é importante que exista um nó gerente capaz de gerenciar a execução das mesmas. Por tais motivos, a presença de um orquestrador tornou-se indispensável. O presente trabalho adotou o Docker Swarm como uma forma de gerenciar contêineres, conectar os contêineres presentes em cada *host*, garantir um estado mínimo de funcionamento do sistema e como uma forma de criar os serviços de workflow.

Figura 3.3 – Infraestrutura Proposta.



O cenário no qual o Docker Swarm foi configurado é composto por dois *hosts*, onde o primeiro atua como gerente, trabalhador e repositório de imagens de contêineres e o segundo

atua apenas como trabalhador. A figura 3.3 demonstra a disposição dos *hosts* e os paralelepípedos representam os contêineres que estão rodando sobre cada máquina. O Docker Swarm permite a criação de redes de sobreposição, que são criadas sobre a rede onde cada *host* está conectado e permite que contêineres se conectem a mesma e iniciem uma comunicação segura. Porém, durante a fase de configuração do Swarm, notou-se que os contêineres não conseguiam se comunicar quando os *hosts* estavam conectados a sub-redes diferentes, essa limitação do Docker Swarm foi confirmada em fóruns de discussão da ferramenta. A solução adotada foi criar interfaces virtuais em cada nó e conectá-los a mesma sub-rede, assim, foi possível conectar os contêineres através da rede de sobreposição do Docker Swarm.

3.2.5 Integração com uma Engine de Workflow

Conforme apresentado na seção 2.2.3, existem diversas implementações de *engines* de workflow científico. E a escolha da *engine* Makeflow se deu por várias razões, entre elas: capacidade de executar computações distribuídas de larga escala, tolerância a falhas, presença de um repositório com exemplos de workflows científicos[18], facilidade de uso e compatibilidade com um sistema para execução em vários *hosts*. O Makeflow possui uma maneira simples de descrever um workflow científico. Ele utiliza um arquivo com a sintaxe similar aos makefiles utilizados para compilação de códigos C e C++. Outra vantagem do Makeflow é que com ele é possível explicitar quais são as dependências de cada tarefa, dessa forma uma tarefa no workflow é descrita através dos seguintes parâmetros: arquivos de entrada, comando para executar a tarefa e arquivos de saída. A figura 3.4 mostra a estrutura de um arquivo makeflow que possui a descrição de duas tarefas, onde a tarefa 2 recebe como entrada o arquivo gerado pela tarefa 1. Além dos arquivos de entrada, saída e o comando para o processamento da tarefa, é possível adicionar ao Makeflow informações referentes aos recursos mínimos esperados para que um trabalhador execute a tarefa. Informações sobre espaço em disco, memória RAM e núcleos de processamento podem ser adicionadas às tarefas, dessa forma, somente os nós trabalhadores que possuem os recursos mínimos devem executar as tarefas.

Algo importante de ser destacado é que o Makeflow é uma ferramenta de um pacote de software conhecido como cctools (The Cooperative Computing Tools). Esse pacote possui um conjunto de ferramentas destinadas à computação distribuída e de larga escala em cluster, clouds

[18] Exemplos de Workflows: https://github.com/cooperative-computing-lab/makeflow-examples

Figura 3.4 – Estrutura de um arquivo makeflow.

```
saida_tarefa1: dependencia_de_entrada1
$./tarefa1 -in dependencia_de_entrada1 -out saida_tarefa1

saida_tarefa2: saida_tarefa1
$./tarefa2 -in saida_tarefa1 -out saida_tarefa2
```

e grids. No decorrer deste trabalho, outras ferramentas deste pacote também serão utilizadas, logo o próprio pacote tornou-se uma dependência para a execução de cada tarefa no fluxo. Assim, uma camada que contém as ferramentas do cctools compiladas e prontas para o uso foi adicionada a imagem de contêiner base. Assim, esta imagem de contêiner deve ser transferida aos nós trabalhadores do sistema.

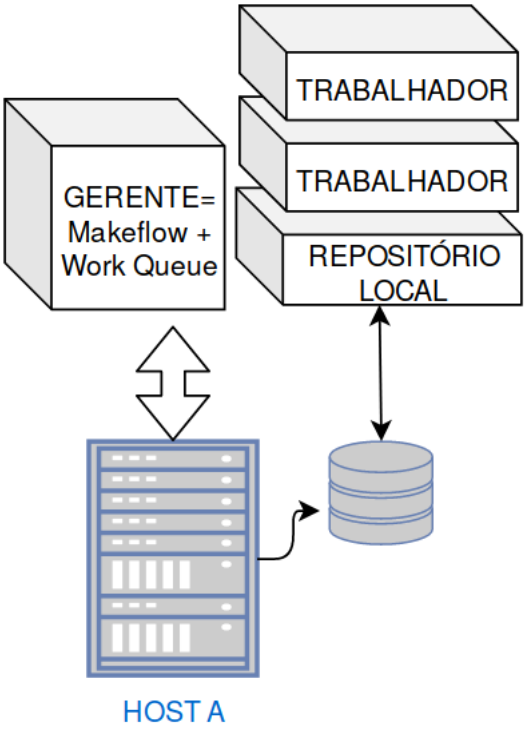
3.2.6 Transferência de Dependências

Uma vez que as imagens de contêineres foram transferidas para os nós trabalhadores, os contêineres foram iniciados e conectados, falta ainda mais uma etapa antes que as tarefas possam ser executadas. A etapa faltante refere-se ao envio das dependências de entrada, ou seja, o envio dos arquivos que as tarefas irão processar. Esta etapa é controlada por outra ferramenta do conjunto cctools, chamada de Work Queue(ALBRECHT et al., 2012). O Work Queue é responsável por ler as tarefas definidas pelo Makeflow, distribuir as tarefas entre os nós trabalhadores, enviar os arquivos de dependência e por fim recuperar os arquivos de saída da tarefa.

O Work Queue, ao ler as tarefas do Makeflow, define quais tarefas devem executar em qual ordem. A ordem de execução é importante, pois existem tarefas que dependem das saídas de outras tarefas. Os nós trabalhadores devem então se conectar ao servidor Work Queue e informar que estão aptos para processar tarefas. Quando um trabalhador é conectado, algumas informações são enviadas, como a quantidade de núcleos de processamento disponível, a quantidade de memória RAM disponível, o espaço em disco disponível, entre outros. Essas informações são relevantes no momento em que o Work Queue decide atribuir uma nova tarefa para um nó trabalhador. Dessa forma, o gerente só atribui tarefas para os nós que são capazes de executá-las, ou seja, possuem os recursos mínimos disponíveis.

Portanto, o *host* A, no cenário proposto, que é o *host* com maior capacidade de processamento, fica encarregado de executar o contêiner gerente, que irá atuar como o servidor Work

Figura 3.5 – Configuração do *Host* A.



Queue, conforme mostrado na figura 3.5.

3.2.7 Uma Pedra no Caminho

Quando o Work Queue envia as tarefas para os nós trabalhadores, os arquivos de entradas também são enviados. Cada trabalhador do Work Queue, ao iniciar sua execução, cria um diretório para receber os arquivos e executar neste diretório as tarefas recebidas. Desse modo, o nó trabalhador cria uma *sandbox* (caixa de areia) onde os arquivos são colocados e onde a tarefa executa-rá, a caixa atua como um ambiente controlado onde o trabalhador pode processar as tarefas, sem se preocupar em alterar arquivos importante de outras áreas do sistema de arquivos. Assim, quando o fluxo de trabalho chega ao fim e não existem mais tarefas para serem processadas, cada trabalhador apaga a sua caixa de areia, deixando o sistema de arquivos inalterado.

No entanto, essa abordagem é somente válida para quando os nós trabalhadores são executados diretamente sobre o sistema operacional da máquina hospedeira, onde alterações no sistema de arquivo podem travar a máquina por inteira e vários processos podem ser prejudicados. Quando contêineres são utilizados, os próprios contêineres atuam como uma grande caixa de areia, onde alterações podem ser feitas sem prejudicar o sistema de arquivo do hospedeiro.

Como já mencionado, a criação e processamento de workflows científicos é um processo iterativo, onde o fluxo inteiro deve ser executado várias vezes com pequenas alterações. Desta maneira, ao enviar um arquivo de um *host* para o outro, é interessante que esse arquivo seja guardado, para que em uma próxima execução do mesmo fluxo, não seja necessário gastar tempo de processamento para enviar novamente o mesmo arquivo, que não foi alterado. Esse processo, onde os arquivos são armazenados para que um novo envio não seja feito é conhecido como uso de *cache*.

O Work Queue permite que os arquivos sejam guardados em *cache*. A estratégia utilizada é a seguinte: o nó gerente envia um arquivo para o trabalhador, e então guarda a informação de que tal nó já recebeu o arquivo enviado. O nó trabalhador, por sua vez, armazena esse arquivo dentro do seu diretório de caixa de areia e o mantém até que seja desconectado pelo gerente. Assim, quando uma nova tarefa utiliza o mesmo arquivo anterior, o nó gerente tem conhecimento de que o arquivo já foi enviado, e não o envia novamente. Quando a estratégia de cache é utilizada, ao final do fluxo, o nó trabalhador apaga todo os arquivos da sua caixa de areia, menos aqueles marcados para serem guardados. O uso da cache dessa maneira é aplicável quando os nós trabalhadores são executados diretamente sobre o sistema operacional do hospedeiro, e são executadas de maneira contínua.

O problema está quando esta estratégia é aplicada sobre contêineres, que são unidades feitas para serem voláteis e que podem ser substituídos por outros contêineres quando uma execução falha. Assim, se esta estratégia fosse aplicada com o uso de contêineres, ao final da execução do fluxo, todos os contêineres seriam terminados e os arquivos marcados para ficarem na cache seriam perdidos. Assim, em uma nova rodada na execução do fluxo, todos os arquivos teriam que ser enviados novamente, mesmo que não alterados.

3.2.8 Contorne a Pedra: A Solução Proposta

Para garantir que o sistema de cache funcionasse no contexto de contêineres, foi necessário analisar o código fonte do Work Queue e do Makeflow, afim de determinar como esse processo era feito, e consequentemente propor mudanças. O sistema de cache do Work Queue é utilizado de modo que o servidor Work Queue (gerente) armazena todos os arquivos que estão sendo mantidos em cache pelos trabalhadores. O gerente armazena uma tabela *hash*, que associa um identificador do trabalhador (chave) aos arquivos que o mesmo possui em *cache* (valores). A tabela 3.1 é um exemplo simplificado de como essas informações são armazenadas.

Para garantir que arquivos com o mesmo nome sejam enviados novamente quando sofrerem mudanças, a tabela *hash* armazena o nome do arquivo mais a *hash* do seu conteúdo. A hash pode ser vista como uma função que recebe o conteúdo de cada arquivo e gera um valor de tamanho fixo, assim mudanças podem ser detectadas, mesmo que os arquivos continuem com os mesmos nomes.

Tabela 3.1 – Exemplo de Tabela Hash do Work Queue

Chave	**Valor**
Node-ca09	arquivo-entrada1-bb82, arquivo-de-entrada1-kj3n
Node-br10	arquivo-entrada13-ll08
Node-as36	arquivo-entrada13-ll08

Para contornar o problema da volatilidade dos contêineres, é possível configurar um Docker Volume, que funciona como uma espécie de disco virtual, o qual é armazenado sobre o sistema nativo do hospedeiro e que é persistente, mesmo que o contêiner seja desligado, os dados armazenados nesses volumes continuam acessíveis. A vantagem de um Volume Docker é que vários contêineres podem acessar o mesmo volume, compartilhando dados. Assim, os contêineres podem acessar o mesmo volume e armazenar os arquivos que devem ser mantidos em *cache*, para que em novas execuções do mesmo fluxo.

O principal problema dessa abordagem é que o trabalhador Work Queue, quando iniciado recebe um identificador único. Desse modo, mesmo que os arquivos estejam presentes na cache, localizada no volume do Docker, o nó gerente continuará enviando os arquivos, pois as chaves e valores da tabela Hash, armazenada pelo gerente, não coincidem. O nó trabalhador, nesse contexto, não tem conhecimento de que possui tal arquivo. Assim, foi necessário adicionar uma mensagem ao protocolo de comunicação do Work Queue para que os nós trabalhadores pudessem informar que já possuem arquivos em sua cache.

Portanto, com essa nova abordagem, o nó trabalhador, encapsulado por um contêiner, ao iniciar sua execução faz uma varredura dos arquivos presentes no volume Docker, os transfere para o seu diretório de caixa de areia, e informa ao nó gerente os arquivos que possui. O nó gerente, por sua vez, recebe esta informação e armazena na tabela hash os arquivos recebidos, com base no identificador do nó trabalhador recebido. Dessa forma, quando o fluxo for executado novamente, e todos os nós trabalhadores (contêineres) forem criados novamente, os arquivos presentes no volume do Docker não serão enviados novamente, garantindo um sistema de *cache*.

3.2.9 Gerência de Contêineres e Tarefas

O Docker permite a definição de aplicações que utilizam vários contêineres por meio de um arquivo no formato YAML chamado de Docker Compose. Através deste arquivo é possível determinar quais contêineres devem executar, quais imagens devem ser usadas, quais volumes devem ser montados. É permitido, ainda, a definição de políticas como o lugar (em qual *host*) em que cada contêiner deve executar e o número mínimo de instâncias de cada serviço. Com o Docker Compose, todo o ciclo de vida de uma aplicação pode ser gerenciado e são avaliados aspectos como a criação, remoção e reconstrução de serviços e verificação da situação de cada serviço. Dessa forma, esta funcionalidade do Docker foi utilizada para a definição dps serviços como os nós trabalhadores, nó gerente e repositório local de imagens de contêineres. Definiu-se políticas onde o nó gerente e o repositório local de imagens de contêineres devem executar nos *hosts* definidos como gerentes do Swarm.

O Work Queue, além de distribuir as tarefas criadas por meio do Makeflow também é responsável por gerenciar-las, ele que deve criar as tarefas e garantir que as mesmas sejam executadas em algum dos trabalhadores. Caso algum trabalhador falhe na execução de alguma tarefa, o Work Queue possui mecanismos que permitem a detecção da falha e posterior re-alocação da tarefa para outro trabalhador. Durante a definição de uma tarefa o pesquisador pode acrescentar informações como tempo máximo para execução, número de processadores necessários, quantidade de armazenamento mínimo para recebimento das dependências mais os arquivos de saída. Todas essas informações são úteis na hora que o Work Queue decide em qual trabalhador cada tarefa deve executar. De forma geral, o cenário proposto é que o Docker Compose fica responsável por gerenciar os contêineres presentes no sistema, analogamente o Work Queue fica responsável pelas tarefas.

3.2.10 Passos para a Execução de um Workflow Científico

A estratégia utilizada para integrar a execução de workflows científicos com uma ferramenta de virtualização a nivel de sistema, conhecida como contêiner, envolve os seguintes passos:

1. **Escolha de uma imagem base.** O primeiro passo para a integração é a escolha de uma imagem de contêiner para a execução do workflow científico. O Docker disponibiliza várias imagens por meio do Docker Hub e essa escolha pode variar em cada caso. O

conjunto de software utilizado pelas tarefas do workflow pode determinar qual sistema ou distribuição deve ser escolhido.

2. **Instalação de Dependências.** Após a escolha da imagem base, deve-se instalar todas as dependências utilizadas pelas tarefas do fluxo de trabalho. Nesse passo, o analisador de dependências pode ser utilizado para identificar quais as dependências de cada tarefa do fluxo e assim auxiliar no processo de instalação de bibliotecas e pacotes.

3. **Configuração do Swarm.** Deve ser configurado o Docker Swarm em cada *host* em que se deseja executar as tarefas e pelo menos um nó deve ser eleito como o gerente. O nó gerente possui maiores responsabilidades do que os nós trabalhadores, assim sendo vital para a execução de todo o fluxo de trabalho e deve portanto ser configurado em um *host* confiável e robusto.

4. **Criação dos contêineres trabalhadores.** Cada *host* deverá baixar a imagem de contêiner situada no repositório de imagens de contêineres local. Após o download da imagem, os contêineres trabalhadores devem se conectar ao contêiner gerente e esperar pela atribuição de tarefas.

   Após a transferência das dependências, cada *host* que atua como trabalhador deve criar um contêiner com a imagem base e transferir para os mesmos as dependências implícitas. Após esse passo, cada contêiner deve atuar como um trabalhador, esperando as tarefas distribuídas pelo nó mestre, presente no mesmo *host* gerente do Swarm.

5. **Execução do Workflow Científico.** O *host* gerente deve utilizar o Work Queue para distribuir as tarefas do workflow científico definido por meio do Makeflow.

6. **Coleta dos Resultados.** A medida que os trabalhadores terminam suas tarefas, o Work Queue é responsável por coletar os arquivos de saída de cada tarefa. Ao final de todas as tarefas, os resultados estarão presentes no *host* gerente.

4 EXPERIMENTOS E RESULTADOS

4.1 Workflows Científicos Escolhidos

Para a realização dos experimentos, foram escolhidos dois Workflows científicos do conjunto de workflows de exemplo disponibilizado por Nick Hazekamp e Douglas Thain[19]. O primeiro workflow escolhido foi o Blast. O Blast possui um conjunto de 15 tarefas, sendo 3 executadas localmente (no nó gerente) e 12 executadas de forma distribuída nos nós trabalhadores, as quais executam o mesmo programa (um binário chamado de *blastall*) para várias entradas distintas. O Blast possui como dependência de entrada, para todas as tarefas, uma pasta com arquivos que somam 565 MB. A figura 4.1 mostra o grafo que define o Blast, onde os círculos verdes representam as tarefas e os quadrados azuis são os dados de entrada. A tarefa inicial deve gerar os arquivos de entradas para as tarefas posteriores. As tarefas intermediárias devem processar os arquivos de entradas gerados. Por fim, o gerente executa 2 tarefas, onde uma deve reunir as saídas de erro das tarefas anteriores e a outra deve reunir as saídas bem sucedidas.

O segundo workflow escolhido foi o Hecil, que possui um formato de workflow singular. A figura 4.2 mostra o grafo do workflow Hecil simplificado, por questões de visualização são apresentadas menos tarefas. O Hecil possui tarefas mais diferenciadas do que o Blast, apresentando tarefas que utilizam *scripts* em Python, códigos em Perl e execução de binários. Desta forma, este workflow é um caso interessante para ser testado, pois com ele é possível verificar se a ferramenta proposta consegue gerenciar tais dependências. O Hecil possui um total de 112 tarefas que devem ser executas nos nós trabalhadores. Primeiramente, executa-se localmente 2 tarefas, onde são gerados arquivos de entradas para as tarefas posteriores. A parte superior do workflow executa um binário chamado *bwa*, responsável por mapear sequências em relação a um conjunto de genoma de referência[20]. A parte inferior executa um *script* Python que atua na correção de erros de leituras longas sequenciamento de genomas, o algoritmo é descrito em (CHOUDHURY; CHAKRABARTY; EMRICH, 2017).

[19] Repositório de Workflows do Makeflow: http://github.com/cooperative-computing-lab/makeflow-examples

[20] Burrows-Wheeler Aligner: http://bio-bwa.sourceforge.net/

Figura 4.1 – Workflow Blast.

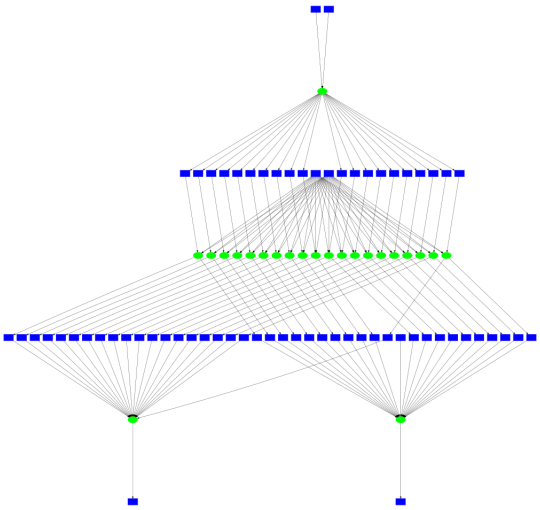
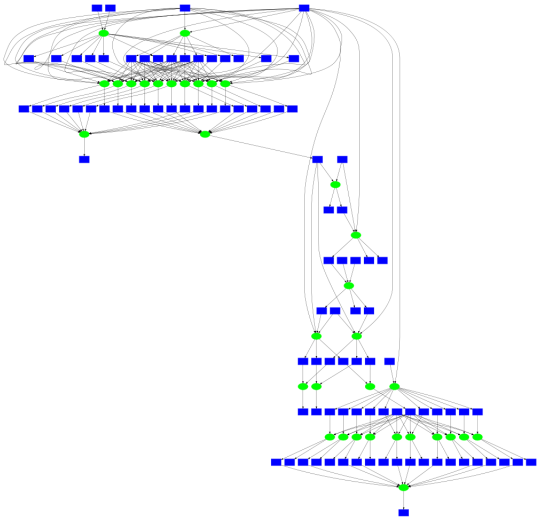


Figura 4.2 – Workflow Hecil.



4.2 Cenários Propostos

Na realização dos experimentos, a infraestrutura demonstrada na Figura 3.3 foi utilizada. O *host* A possui um processador Intel(R) Xeon(R) com 4 núcleos e 2 *threads* por núcleo, 12 GB de memória RAM. Já o *host* B possui um processador Intel(R) Core(TM) i5-3230M (Intel Core IvyBridge processor) com 2 núcleos e 2 *threads* por core, 8 GB de memória RAM. A condução dos experimentos descritos nessa seção visam, primeiramente, definir se é possível executar um workflow científico com a ferramenta proposta. Ainda, os cenários descritos a seguir utilizam 11 e 2 trabalhadores. Com 11 trabalhadores será testada a estratégia onde existe um contêiner para cada *thread* de cada host, assim no *host* A devem executar 7 contêineres trabalhadores mais

um contêiner que deve atuar como gerente; e no *host* B devem atuar 4 contêineres trabalhadores. Já nos cenários que utilizam 2 trabalhadores, será testada a estratégia onde cada contêiner possui acesso a todos os recursos de cada *host*, assim será utilizado 1 contêiner trabalhador com acesso a todos os *cores* do *host* B e 1 contêiner trabalhador com acesso a 7 *threads* do *host* A (1 *thread* é reservada para o contêiner gerente). Foram propostos 6 cenários para serem avaliados:

1. **Makeflow e Work Queue para 11 trabalhadores.** Neste cenário, os *hosts* A e B foram configurados para rodar em conjunto 11 contêineres trabalhadores. Os contêineres trabalhadores do *host* A possuíam acesso a 1 thread da máquina hospedeira. Foram instanciados 4 contêineres trabalhadores no *host* A. Os contêineres trabalhadores do *host* B possuíam acesso a 1 thread da máquina hospedeira. Foram instanciados 7 trabalhadores no *host* B. Ainda, no *host* B, foi instanciado um contêiner para executar o papel de gerente (servidor Work Queue) o outro contêiner que fazia o papel de repositório local de imagens de contêineres.

2. **Makeflow e Work Queue para 2 trabalhadores.** Neste cenário, os *hosts* A e B foram configurados para rodar somente um contêiner trabalhador. O contêiner trabalhador do *host* A, possuía acesso as 4 *threads* disponíveis na máquina hospedeira. O contêiner trabalhador do host B, possuía acesso a 7 *threads* da máquina hospedeira. Ainda, no *host* B, foi instanciado um contêiner para executar o papel de gerente (servidor Work Queue) o outro contêiner que fazia o papel de repositório local de imagens de contêineres.

3. **Makeflow e Work Queue modificado para 11 trabalhadores.** Neste cenário, os hosts A e B foram configurados para rodar em conjunto 11 contêineres trabalhadores. Os contêineres trabalhadores do *host* A possuíam acesso a 1 *thread* da máquina hospedeira. Foram instanciados 4 contêineres trabalhadores no *host* A. Os contêineres trabalhadores do *host* B possuíam acesso a 1 thread da máquina hospedeira. Foram instanciados 7 trabalhadores no *host* B. Ainda, no *host* B, foi instanciado um contêiner para executar o papel de gerente (servidor Work Queue) o outro contêiner que fazia o papel de repositório local de imagens de contêineres. A diferença deste cenário para o cenário 2, é aqui foram utilizados os códigos do Work Queue modificado para o uso de cache, assim em cada host configurou-se um volume Docker para receber os arquivos de dependências de entrada.

4. **Makeflow e Work Queue modificado para 2 trabalhadores.** Neste cenário, os *hosts* A e B foram configurados para rodar somente um contêiner trabalhador. O contêiner

trabalhador do *host* A, possuía acesso as 4 *threads* disponíveis na máquina hospedeira. O contêiner trabalhador do *host* B, possuía acesso a 7 *threads* da máquina hospedeira. Ainda, no *host* B, foi instanciado um contêiner para executar o papel de gerente (servidor Work Queue) o outro contêiner que fazia o papel de repositório local de imagens de contêineres. A diferença deste cenário para o cenário 1, é aqui foram utilizados os códigos do Work Queue modificado para o uso de cache, assim em cada host configurou-se um volume Docker para receber os arquivos de dependências de entrada.

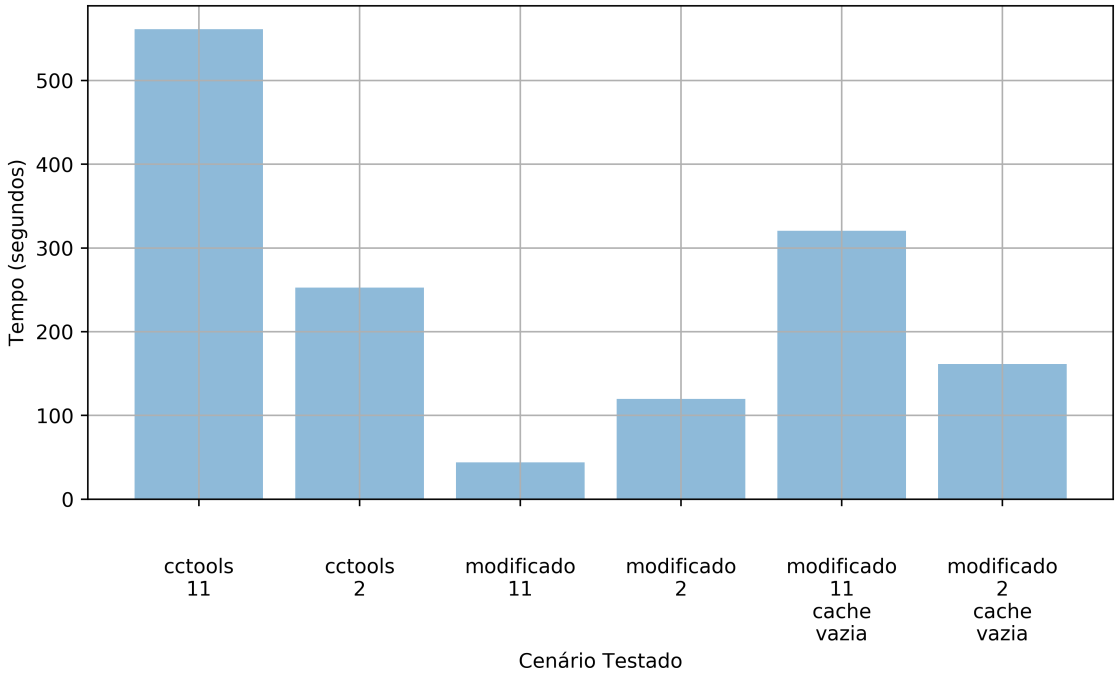
5. **Makeflow e Work Queue modificado para 11 trabalhadores com cache vazia.** Neste cenário, os *hosts* A e B foram configurados da mesma forma que o cenário 3, porém para cada execução, o volume do Docker não possuía nenhum dado previamente.

6. **Makeflow e Work Queue modificado para 2 trabalhadores com cache vazia.** Neste cenário, os *hosts* A e B foram configurados da mesma forma que o cenário 4, porém para cada execução, o volume do Docker não possuía nenhum dado previamente.

4.3 Avaliação do Desempenho do Workflow Blast

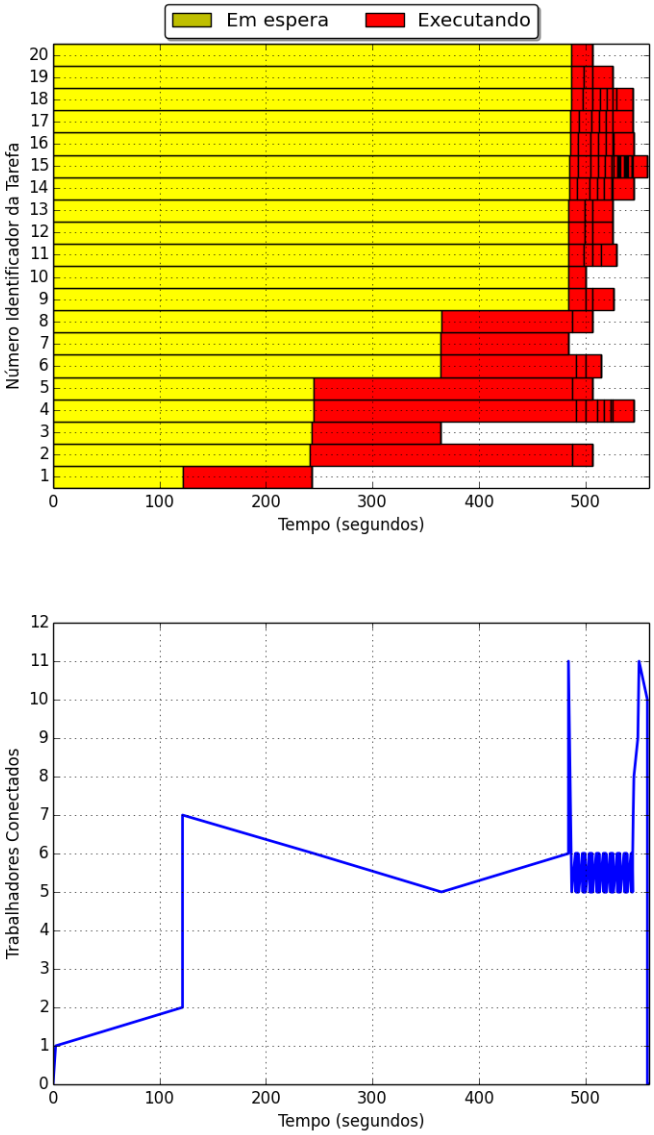
A figura 4.3 demonstra a média dos tempos de execução para cada cenário proposto, onde cada cenário foi executado 10 vezes. A legenda *cctools* refere-se as execuções onde foram utilizadas o Makeflow e o Work Queue sem modificações (cenários 1 e 2). Nota-se que o cenário que levou menos tempo, foi o cenário com 11 trabalhadores e com as modificações para uso da *cache*. A execução que levou mais tempo, foi o cenário onde utilizou-se 11 trabalhadores com o Makeflow e Work Queue sem modificações.

Figura 4.3 – Tempo de Execução Médio do Workflow Blast.



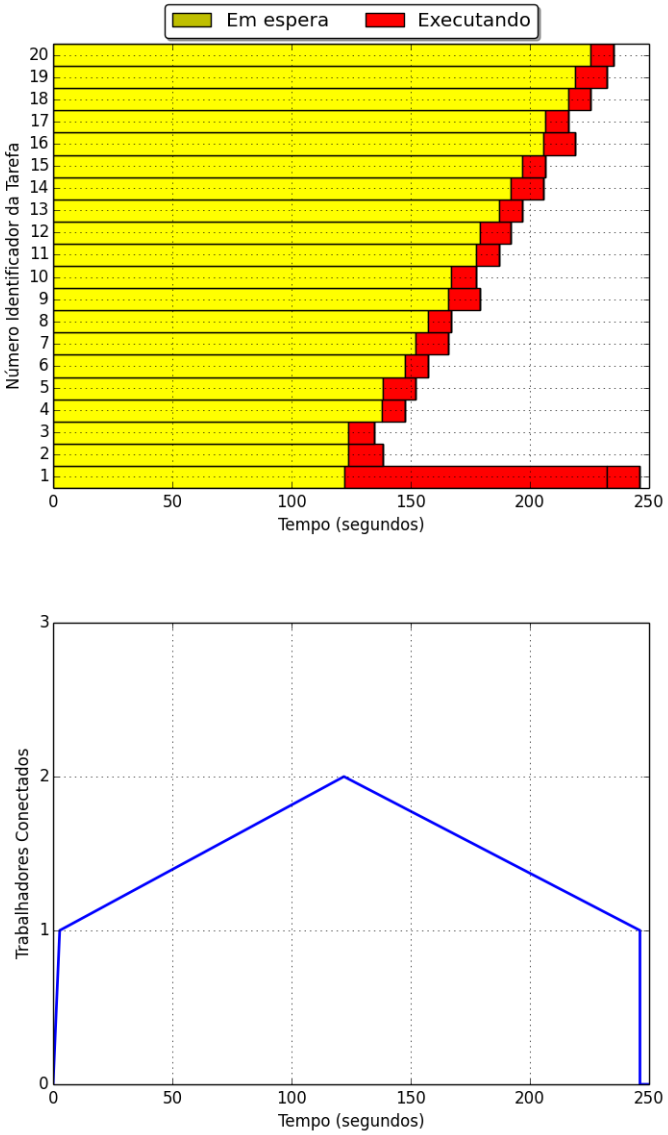
A figura 4.4 detalha o cenário que em que o maior tempo de execução foi obtido. Percebe-se que o tempo que cada tarefa fica em espera é bastante significativo, isso se dá, pois o gerente Work Queue deve mandar as dependências de entrada para cada trabalhador do sistema, assim multiplicando por 11 a quantidade de dados trafegados na rede. Nota-se também que o Work Queue possui dificuldades em gerenciar as conexões de mais trabalhadores ao mesmo tempo que envia as dependências. Percebe-se que próximo ao tempo de 500 segundos, várias tarefas foram executadas em paralelo, pode-se afirmar que nesse tempo, o Work Queue conseguiu enviar todas as dependências para cada trabalhador, e eles passaram e executá-las em paralelo.

Figura 4.4 – Execução do Blast para o Cenário 1.



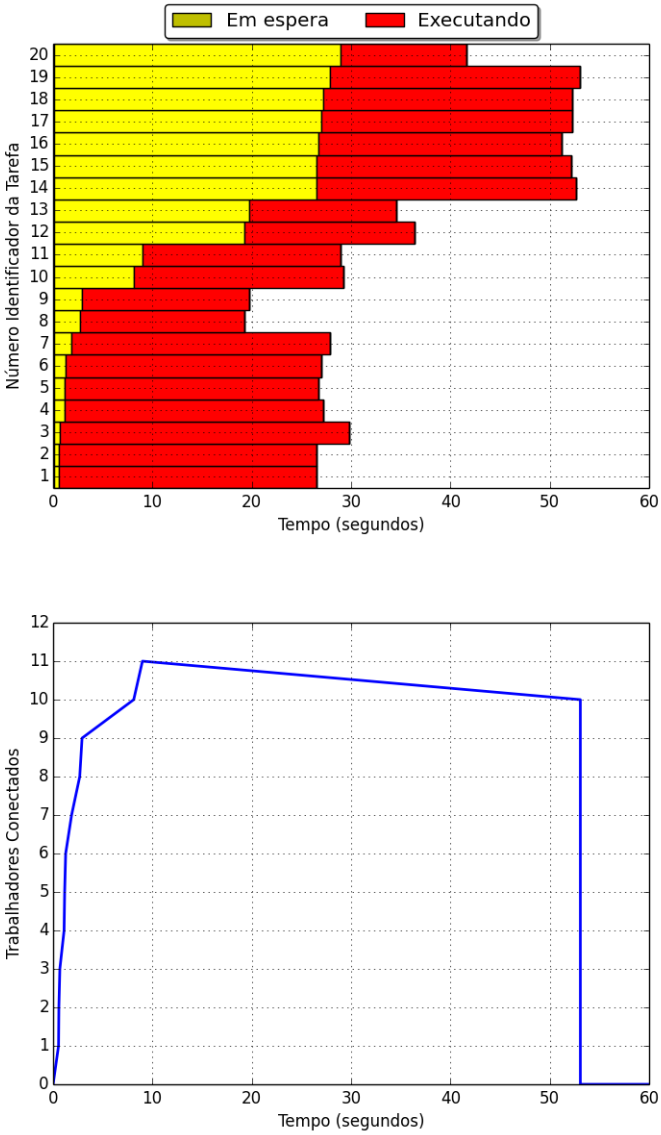
A figura 4.5 detalha o cenário 2, onde existe um menor número de trabalhadores, porém percebe-se que para esse workflow, o paralelismo em nível de nós conectados (figura 4.4) obteve tempos menores do que em relação ao paralelismo de *threads* (figura 4.5).

Figura 4.5 – Execução do Blast para o Cenário 2.



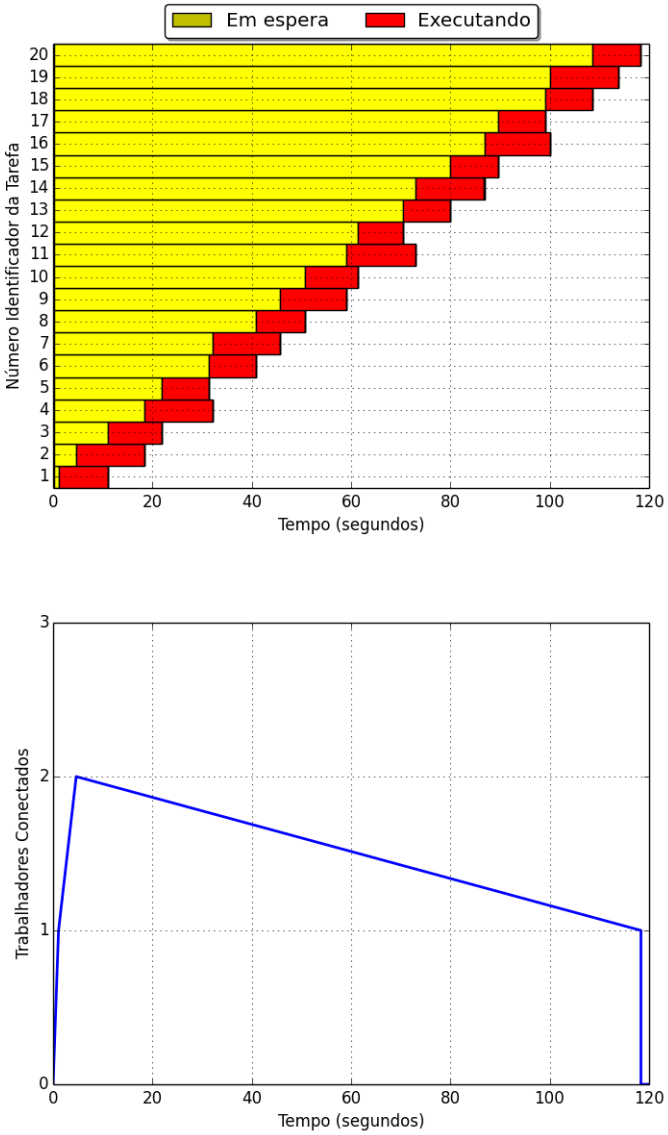
A figura 4.6 detalha o cenário 3, onde o menor tempo de execução foi obtido. Percebe-se que com o uso da *cache*, o tempo de espera para cada tarefa executar diminuiu bastante, uma vez que não existem dependências de entradas para serem enviadas. O nível de paralelismo de tarefas foi bem alto, quando comparado em relação aos outros cenários. Percebe-se, também, que o Work Queue gerenciou melhor as conexões dos trabalhadores, quando não precisou enviar dados de dependências pela rede.

Figura 4.6 – Execução do Blast para o Cenário 3.



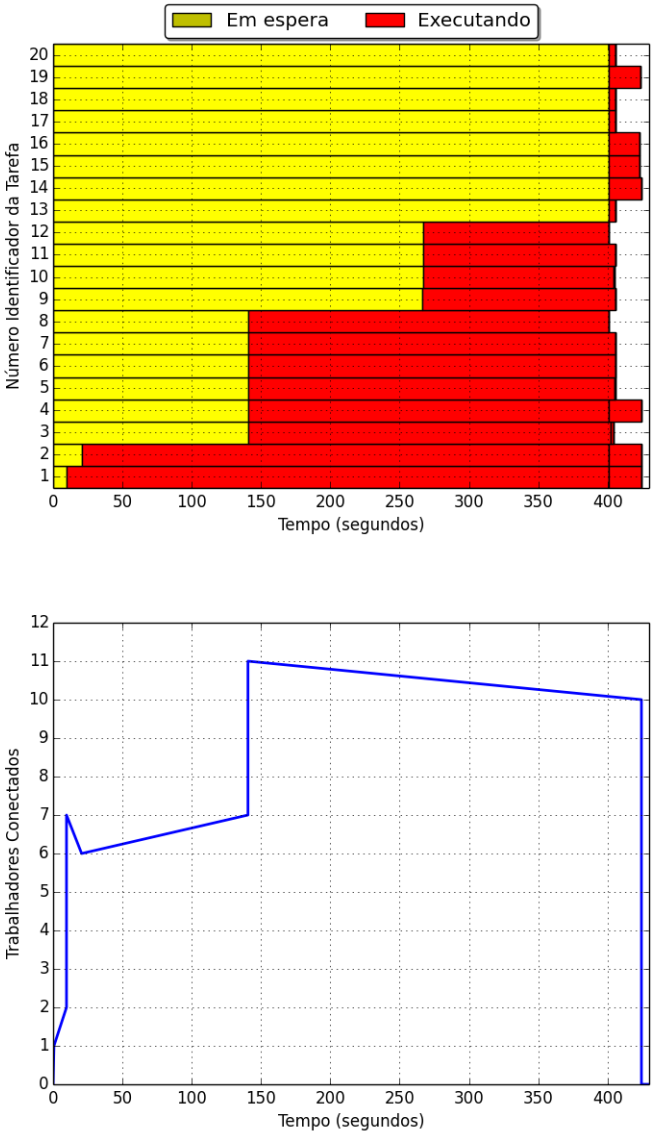
A figura 4.7 detalha o cenário 4, e nela é possível ver com mais clareza, que mesmo sem as dependências para serem enviadas, o Work Queue não fez o uso das *threads* disponíveis para o paralelismo das tarefas.

Figura 4.7 – Execução do Blast para o Cenário 4.



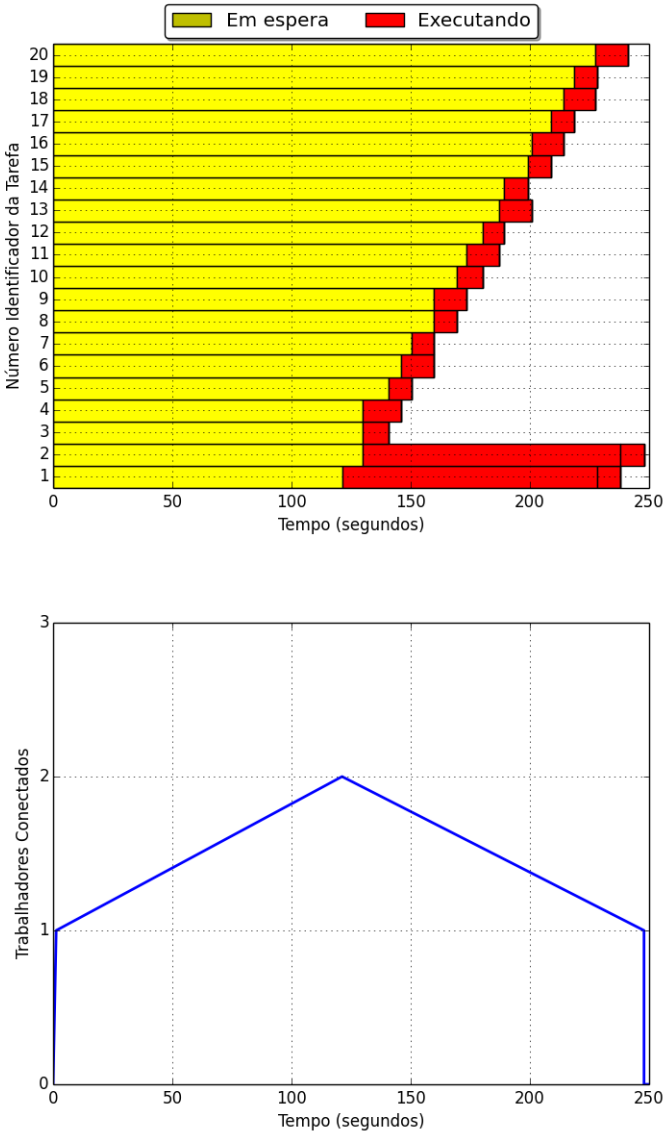
A figura 4.8 mostra que mesmo em um cenário onde a cache se encontra vazia, o uso do volume Docker diminui o tempo para o envio das dependências, pois nesse cenário os dados chegam a todos os contêineres trabalhadores de cada host de uma só vez, pois os mesmos compartilham os dados do volume no qual estão conectados.

Figura 4.8 – Execução do Blast para o Cenário 5.



A figura 4.9 mostra que o cenário 6 é mais eficiente do que o cenário 2 para o envio das dependências, mas o tempo ainda é prejudicado pelo baixo nível de paralelismo alcançado.

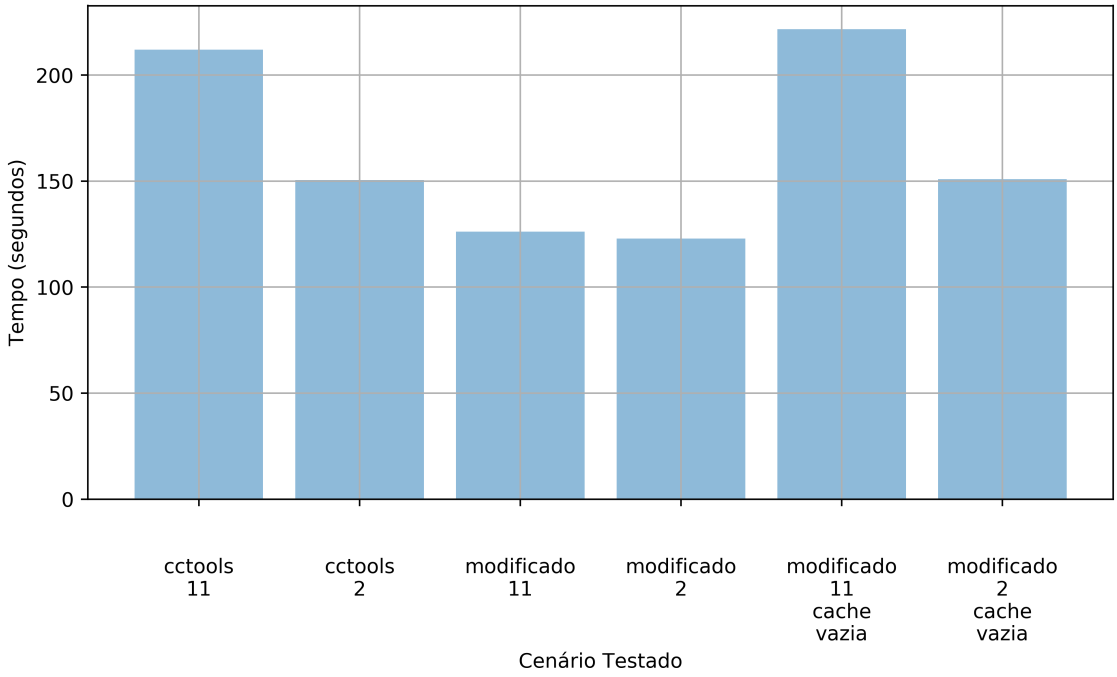
Figura 4.9 – Execução do Blast para o Cenário 6.



4.4 Avaliação do Desempenho do Workflow Hecil

A figura 4.10 demonstra a média dos tempos de execução para cada cenário proposto, onde cada cenário foi executado 10 vezes. A primeira coluna reflete a média dos tempos de execução do workflow diretamente sobre o sistema hospedeiro (*baremetal*), onde 1 trabalhador executou sobre cada *host* e o *host* A executou o gerente Work Queue. A legenda *cctools* refere-se as execuções onde foram utilizadas o Makeflow e o Work Queue sem modificações (cenários 1 e 2). Nota-se que o cenário que levou menos tempo, foi o cenário com 2 trabalhadores e com as modificações para uso da *cache*. A execução que levou mais tempo, foi o cenário onde utilizou-se 11 trabalhadores com, onde a cache se encontrava vazia.

Figura 4.10 – Tempo de Execução Médio do Workflow Hecil.



A figura 4.11 mostra que várias tarefas foram executadas ao mesmo tempo, mas quando comparado a figura 4.12, os tempos das tarefas executando foram maiores. Este é um caso diferente do Blast, pois as tarefas desse workflow são beneficiadas quando existem menos trabalhadores, pois conseguem executar em paralelo nas threads disponíveis.

Figura 4.11 – Execução do Hecil para o Cenário 1.

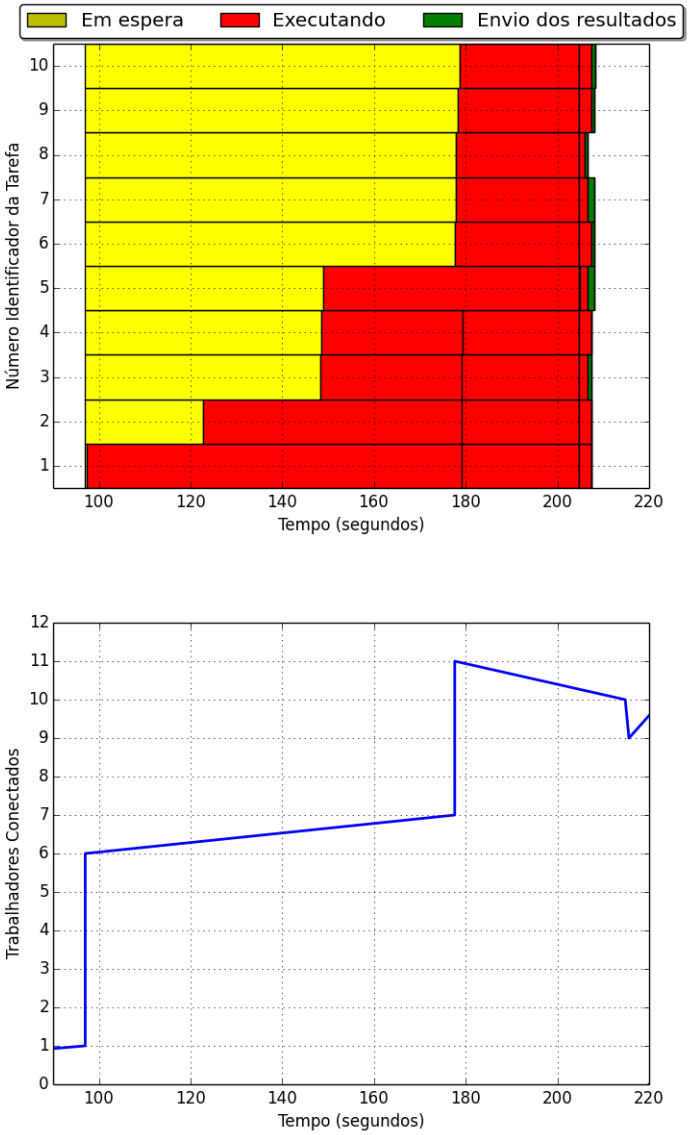
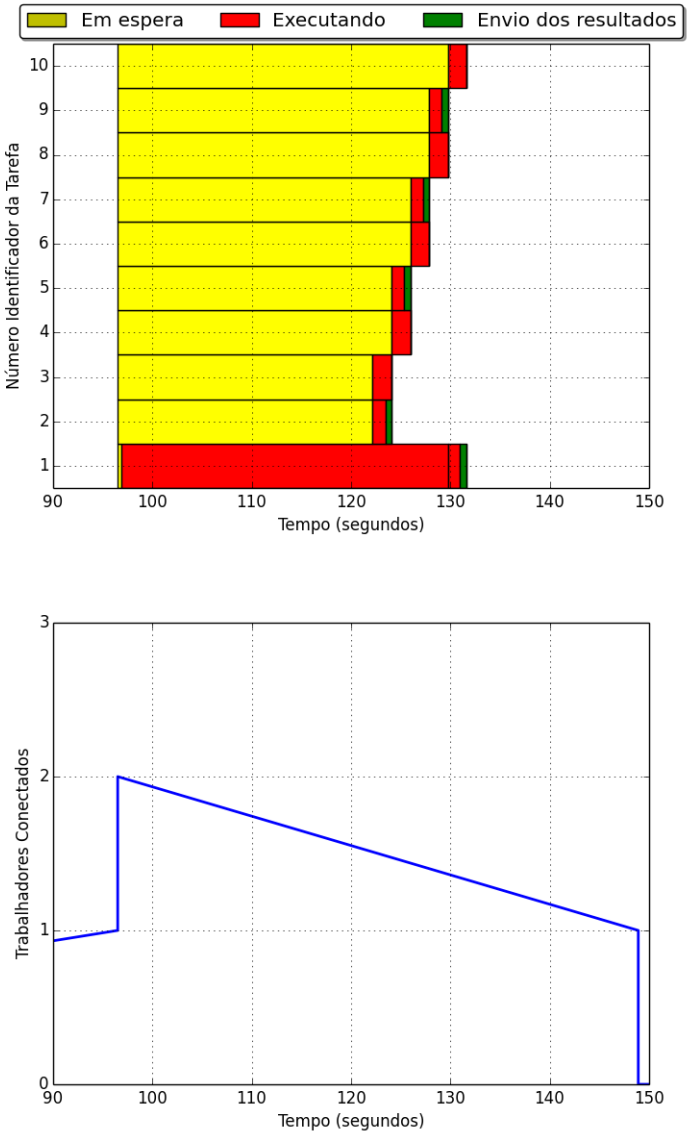


Figura 4.12 – Execução do Hecil para o Cenário 2.



As figuras 4.13 e 4.14 mostram a execução do workflow Hecil com o uso da cache. Nota-se que tempo de execução é bem inferior se comparado aos outros cenários, pois as dependências de entradas não precisam ser transferidas entre o gerente e os trabalhadores. Porém, é possível perceber que o cenário com menos trabalhadores, executa cada tarefa de forma mais rápida, pelo paralelismo em nível de thread.

Figura 4.13 – Execução do Hecil para o Cenário 3.

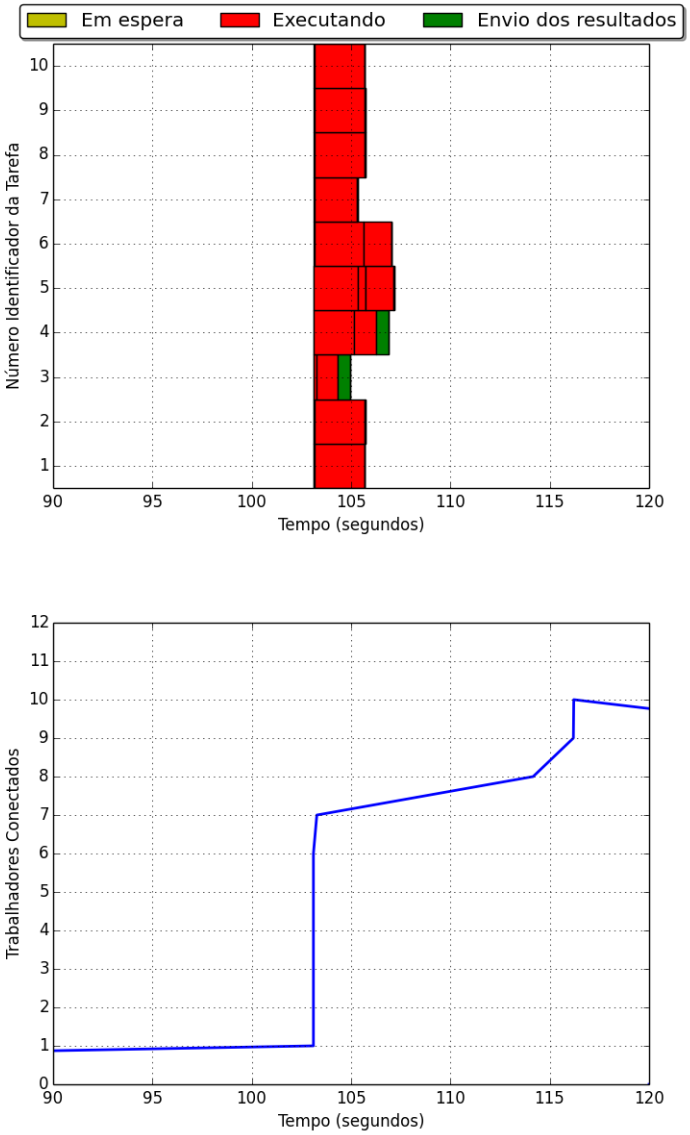
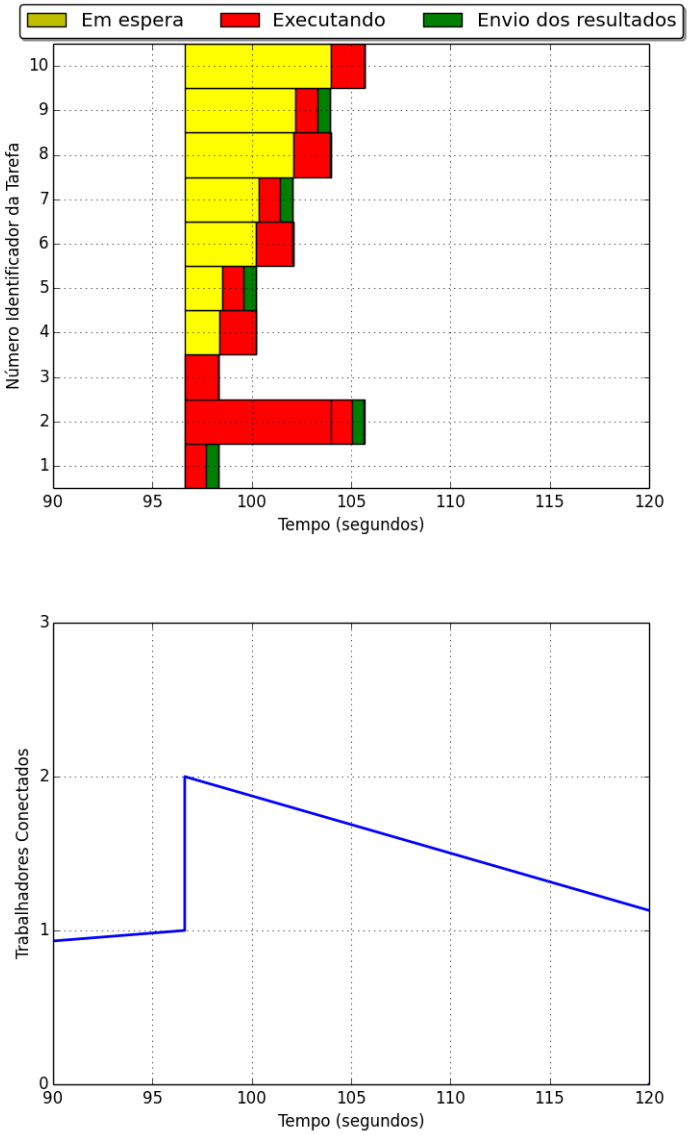


Figura 4.14 – Execução do Hecil para o Cenário 4.



As figuras 4.15 e 4.16 mostram as execuções onde as caches se encontravam vazias. Nota-se que este cenários assemelham-se aos cenários 1 e 2. Devido ao tamanho das dependências de entrada desse workflow serem pequenos, o envio para mais de um nó trabalhador por meio do volume Docker compartilhado não fez tanta diferença. Um dos motivos para esses cenários alcançarem os maiores tempos é que as modificações adicionadas ao Work Queue geram mais mensagens entre os nós trabalhadores e o gerente. Assim, os cenários onde as caches estavam vazias alcançaram tempos maiores do que a implementação padrão do Work Queue (cenários 1 e 2) devido ao número de mensagens necessárias para notificar o gerente sobre os arquivos das caches de cada nó trabalhador.

Figura 4.15 – Execução do Hecil para o Cenário 5.

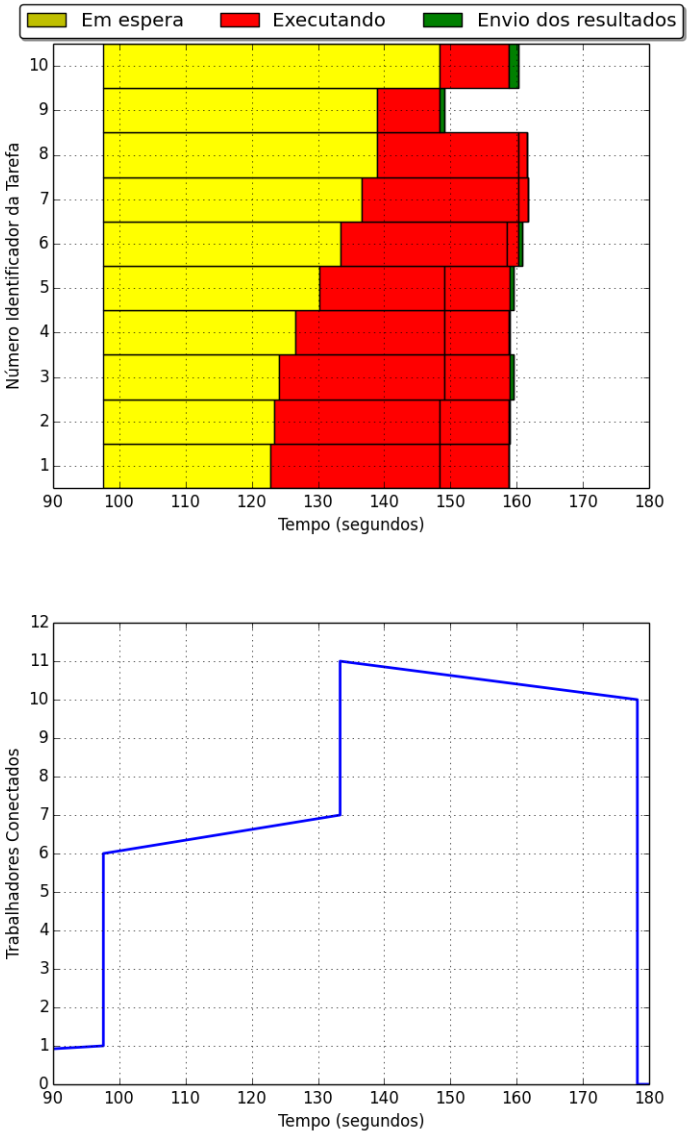
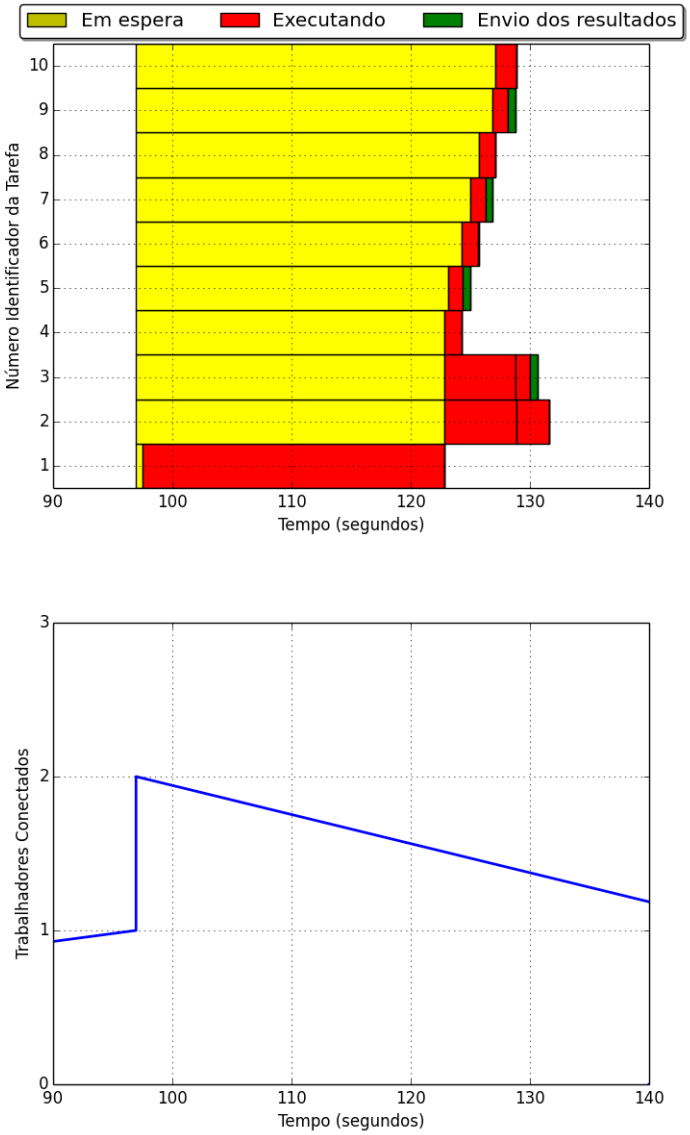


Figura 4.16 – Execução do Hecil para o Cenário 6.



5 CONCLUSÃO

A comunidade científica realiza pesquisas ao redor do mundo com a finalidade de resolver problemas científicos. Para a obtenção de resultados, muitas técnicas de análises são utilizadas na condução de experimentos. Na maior parte desses experimentos, determinadas tarefas são executadas sobre um conjunto de dados de entradas, produzindo um conjunto de dados de saída. A maneira como esse experimento é conduzido define um **fluxo de trabalho**. O Workflow científico determina em que ordem as tarefas devem ser executadas, sobre quais conjuntos de dados tais tarefas devem executar e quais as dependências de dados entre as mesmas.

Em um Workflow científico diversas são as tarefas que necessitam ser processadas, e cada uma delas possui dependências de bibliotecas, executáveis e pacotes. Desse modo, um dos problemas enfrentados pelos cientistas é na determinação de um ambiente sólido, em que todas as dependências são satisfeitas. Por tal motivo, o presente trabalhou teve como objetivo criar uma ferramenta capaz de gerenciar as dependências dos fluxos de trabalhos científicos, através da eficiente criação e distribuição de imagens de contêineres.

Assim para a execução deste trabalho, tecnologias de contêineres foram avaliadas e dentre as opções analisadas, utilizou-se os contêineres do Docker. Estes contêineres foram utilizados como um meio para fornecer um ambiente para que as tarefas executem sobre. Dentre as vantagens na utilização dessa abordagem, pode-se citar a flexibilidade, reprodutibilidade e escalabilidade. Flexibilidade, pois os contêineres garantem uma certa independência em relação ao hardware e ao sistema nativo do hospedeiro. A escalabilidade é garantida, na medida que mais contêineres podem ser instanciados em *hosts* para o processamento de workflows. E, por fim, reprodutibilidade, pois os contêineres facilitam que cientistas compartilhem as imagens de contêineres nas quais os fluxos foram executados.

Durante o andamento do trabalho, utilizou-se o sistema de workflow científico chamado de Makeflow. No entanto, foram encontradas dificuldades na adaptação deste sistema aos contêineres, uma vez que o escalonador de tarefas (Work Queue) foi projetado para funcionar com trabalhadores persistentes, abordagem essa totalmente diferente dos voláteis contêineres. A solução proposta para contornar esse problema foi alterar o protocolo de comunicação do Work Queue e utilizar os volumes do Docker para a criação de caches persistentes.

A solução proposta foi capaz de executar workflows e dentre os resultados obtidos, verificou-se que o uso da cache diminuía significativamente o tempo de execução dos work-

flows. Devido as características do desenvolvimento de workflows, como o modo iterativo, onde a cada execução do fluxo pequenas mudanças são feitas, o uso da cache acaba se aproveitando, o que leva a pensar que a solução proposta poderia ser aplicada a outros workflows.

Para trabalhos futuros fica a elaboração de um sistema adaptativo, onde a estratégia de instanciar um contêiner por host ou vários contêineres pode ser avaliado à medida que novas rodadas do workflow são executados. Também verificou-se que existem poucos workflows definidos com a linguagem do Makeflow, desse modo seria importante criar e disponibilizar outros workflows científicos presentes na literatura.

REFERÊNCIAS

ALBRECHT, M. et al. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In: ACM SIGMOD WORKSHOP ON SCALABLE WORKFLOW EXECUTION ENGINES AND TECHNOLOGIES, 1., New York, NY, USA. **Proceedings...** ACM, 2012. p.1:1–1:13. (SWEET '12).

Altintas, I. et al. Kepler: an extensible system for design and execution of scientific workflows. In: INTERNATIONAL CONFERENCE ON SCIENTIFIC AND STATISTICAL DATABASE MANAGEMENT, 2004., 16. **Proceedings...** [S.l.: s.n.], 2004. p.423–424.

BARKER, A.; HEMERT, J. van. Scientific Workflow: a survey and research directions. In: PARALLEL PROCESSING AND APPLIED MATHEMATICS, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2008. p.746–753.

Bharathi, S. et al. Characterization of scientific workflows. In: THIRD WORKSHOP ON WORKFLOWS IN SUPPORT OF LARGE-SCALE SCIENCE, 2008. **Anais...** [S.l.: s.n.], 2008. p.1–10.

CHOUDHURY, O.; CHAKRABARTY, A.; EMRICH, S. J. HECIL: a hybrid error correction algorithm for long reads with iterative learning. **bioRxiv**, [S.l.], 2017.

Combe, T.; Martin, A.; Di Pietro, R. To Docker or Not to Docker: a security perspective. **IEEE Cloud Computing**, [S.l.], v.3, n.5, p.54–62, Sep. 2016.

COULOURIS, G. et al. **Distributed Systems**: concepts and design. 5th.ed. USA: Addison-Wesley Publishing Company, 2011.

Curcin, V.; Ghanem, M. Scientific workflow systems - can one size fit all? In: CAIRO INTERNATIONAL BIOMEDICAL ENGINEERING CONFERENCE, 2008. **Anais...** [S.l.: s.n.], 2008. p.1–9.

DONKOR E. S., D. N. T. K. D. . A. T. K. Basic local alignment search tool. In: JOURNAL OF MOLECULAR BIOLOGY. **Anais...** [S.l.: s.n.], 1990. v.3, n.215, p.403–410.

DONKOR E. S., D. N. T. K. D. . A. T. K. Bioinformatics with basic local alignment search tool (BLAST) and fast alignment (FASTA). In: JOURNAL OF BIOINFORMATICS AND SEQUENCE ANALYSIS. **Anais...** [S.l.: s.n.], 2014. p.1–6.

F. J. P. Mulerickal and; Paul, B.; Sastri, Y. Evaluation of Docker containers based on hardware utilization. In: INTERNATIONAL CONFERENCE ON CONTROL COMMUNICATION COMPUTING INDIA (ICCC), 2015. **Anais...** [S.l.: s.n.], 2015. p.697–700.

Felter, W. et al. An updated performance comparison of virtual machines and Linux containers. In: IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2015. **Anais...** [S.l.: s.n.], 2015. p.171–172.

HACK, M. **PETRI NET LANGUAGE**. Cambridge, MA, USA: [s.n.], 1976.

Hongbiao, L.; Feng, L.; Wanjun, Y. The research of scientific workflow engine. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND SERVICE SCIENCES, 2010. **Anais...** [S.l.: s.n.], 2010. p.412–414.

Kovacs, A. Comparison of different Linux containers. In: INTERNATIONAL CONFERENCE ON TELECOMMUNICATIONS AND SIGNAL PROCESSING (TSP), 2017. **Anais...** [S.l.: s.n.], 2017. p.47–51.

LEYMANN, F. **Web Services Flow Language**. 1.ed. USA: IBM Software Group, 2001.

LI, H.; DURBIN, R. Fast and accurate short read alignment with Burrows–Wheeler transform. **Bioinformatics**, [S.l.], v.25, n.14, p.1754–1760, 05 2009.

Majithia, S. et al. Triana: a graphical web service composition and execution toolkit. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, 2004. **Proceedings...** [S.l.: s.n.], 2004. p.514–521.

MCPHILLIPS, T. et al. Scientific workflow design for mere mortals. **Future Generation Computer Systems**, [S.l.], v.25, n.5, p.541 – 551, 2009.

NANDA, S.; CHIUEH, T. cker. A Survey on Virtualization Technologies. In: . . **Anais...** [S.l.: s.n.], 2005.

OINN, T. et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. **Bioinformatics**, [S.l.], v.20, n.17, p.3045–3054, 06 2004.

OINN, T. et al. Delivering Web Service Coordination Capability to Users. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE ON ALTERNATE TRACK PAPERS &

POSTERS, 13., New York, NY, USA. **Proceedings...** ACM, 2004. p.438–439. (WWW Alt. '04).

PTRACE. **Process Trace**. [S.l.: s.n.], 2019 (acesso em Setembro, 2019). http://man7.org/linux/man-pages/man2/ptrace.2.html.

Raman, R.; Livny, M.; Solomon, M. Matchmaking: distributed resource management for high throughput computing. In: THE SEVENTH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING (CAT. NO.98TB100244). **Proceedings...** [S.l.: s.n.], 1998. p.140–146.

REDHAT. **O que é um container Linux?** [S.l.: s.n.], 2019 - acesso em 02/09/2019). https://www.redhat.com/pt-br/topics/containers/whats-a-linux-container.

STRACE. **Linux Syscall Traces**. [S.l.: s.n.], 2019 (acesso em Setembro, 2019). https://strace.io/.

SWEENEY, K. M. D.; THAIN, D. Efficient Integration of Containers into Scientific Workflows. In: WORKSHOP ON SCIENTIFIC CLOUD COMPUTING, 9., New York, NY, USA. **Proceedings...** ACM, 2018. p.7:1–7:6. (ScienceCloud'18).

TANENBAUM, A. S.; STEEN, M. v. **Distributed Systems**: principles and paradigms (2nd edition). Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

TOMAYKO, J. E. **Computers in Spaceflight**: the nasa experience. 1988.

WASSERMANN, B. et al. Sedna: a bpel-based environment for visual scientific workflow modelling. In: IN WORKFLOWS FOR ESCIENCE - SCIENTIFIC WORKFLOWS FOR GRIDS. **Anais...** Springer Verlag, 2007.

XAVIER, M. G. et al. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED, AND NETWORK-BASED PROCESSING, 2013. **Anais...** [S.l.: s.n.], 2013. p.233–240.

Yildiz, U.; Guabtni, A.; Ngu, A. H. H. Business versus Scientific Workflows: a comparative study. In: CONGRESS ON SERVICES - I, 2009. **Anais...** [S.l.: s.n.], 2009. p.340–343.

YOO, A. B.; JETTE, M. A.; GRONDONA, M. SLURM: simple linux utility for resource management. In: JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2003. p.44–60.

YOUNGE, A. J. et al. Analysis of Virtualization Technologies for High Performance Computing Environments. In: IEEE 4TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, Washington, DC, USA. **Anais...** IEEE Computer Society, 2011. p.9–16. (CLOUD '11).

ZHENG, C.; THAIN, D. Integrating Containers into Workflows: a case study using makeflow, work queue, and docker. In: INTERNATIONAL WORKSHOP ON VIRTUALIZATION TECHNOLOGIES IN DISTRIBUTED COMPUTING, 8., New York, NY, USA. **Proceedings...** ACM, 2015. p.31–38. (VTDC '15).