`ProvDeploy`: Explorando Alternativas de Conteinerização com Proveniência para Aplicações Científicas com PAD

Liliane Kunstmann[1], **Débora Pina**[1], **Lyncoln S. de Oliveira**[1], **Daniel de Oliveira**[2], **Marta Mattoso**[1]

[1]PESC/COPPE – Universidade Federal do Rio de Janeiro (COPPE/UFRJ)

[2]Instituto de Computação – Universidade Federal Fluminense (IC/UFF)

{lneves, dbpina, oliveiral, marta}@cos.ufrj.br, danielcmo@ic.uff.br

Resumo. *As aplicações científicas demandam ambientes de Processamento de Alto Desempenho (PAD). Essas aplicações possuem diversos componentes advindos de bibliotecas e diferentes ambientes, tornando a pilha de software a ser gerenciada no momento da implantação e execução nada trivial. Essa complexidade aumenta caso o usuário necessite acoplar serviços de captura de dados de proveniência à sua aplicação. Este artigo apresenta o* `ProvDeploy` *para auxiliar o usuário na configuração de contêineres para sua aplicação com captura de proveniência. O* `ProvDeploy` *foi avaliado com uma aplicação intensiva em dados da área de Bioinformática, explorando alternativas de conteinerização em dois ambientes de PAD.*

1. Introdução

Diversas aplicações, sejam elas científicas ou comerciais, fazem processamento intensivo de dados e demandam ambientes de Processamento de Alto Desempenho (PAD) [Elia et al. 2021]. Soluções computacionais vêm sendo propostas para apoiar o desenvolvimento de aplicações capazes de processar e extrair informação útil desses dados. É comum que aplicações em larga escala façam uso de bibliotecas para processamento paralelo (*e.g.*, Dask[1]), reuso de código específico da aplicação, ou ainda plataformas de processamento de *streams* (*e.g.*, Kafka). Essa miríade de soluções acabou por criar um ecossistema de *software* complexo, e as instalações de PAD não conseguem mais acompanhar esse crescimento acelerado de *softwares* e bibliotecas [Yuan and Wildish 2020]. Como resultado, os desenvolvedores precisam realizar um trabalho extra (e por muitas vezes complexo) para implantar suas aplicações, *e.g.*, instalar múltiplos *softwares* a partir do código-fonte, alterar variáveis de ambiente, instalar dependências e alterar arquivos de sistema, o que pode ser uma tarefa propensa a erros (por conta de incompatibilidades de versões, por exemplo).

Esta complexidade demanda o uso de sistemas auxiliares para fazer a depuração, perfilagem de código e a captura de dados de proveniência [Freire et al. 2008] para auxiliar na análise, monitoramento e reprodução das execuções. Os dados de proveniência podem ajudar a análise de dados intermediários durante a execução [Ocaña et al. 2015], a tolerância a falhas [Guedes et al. 2020], e principalmente a reprodução das execuções com PAD [Harrell et al. 2022]. Entretanto, além de adicionar mais uma camada na já complexa pilha de *software* que o usuário tem que manipular, um serviço de captura de proveniência possui sua própria pilha de *software*.

O SciPhy [de Oliveira et al. 2012] é um *workflow* da área de Bioinformática, que recebe como entrada um grande conjunto de sequências de DNA, RNA e aminoácidos,

[1]https://www.dask.org/

e produz como resultado uma árvore filogenética (*i.e.*, árvore que representa as relações evolutivas entre várias espécies). O SciPhy possui vários componentes que processam os dados em paralelo, necessitando de PAD. A versão dessa aplicação desenvolvida em Python é usada ao longo deste artigo. O SciPhy possui dependências de *software* de diferentes bibliotecas (*e.g.*, alinhamento múltiplo de sequência), que não são simples de serem instaladas. Além disso, o SciPhy não captura dados de proveniência de forma nativa, o que faz com que o *workflow* tenha que invocar uma biblioteca externa (*i.e.*, DfAnalyzer [Silva et al. 2020]).

Uma das maneiras de resolver problemas ligados à implantação de aplicações em ambientes de PAD, em especial as científicas, é por meio de contêineres [Struhár et al. 2020]. A conteinerização pode ser descrita como uma virtualização em nível de sistema operacional, onde há compartilhamento de *kernel*. Apesar da popularidade no uso de contêineres, seu uso em PAD não é tão simples [Liu and Guitart 2022]. Ainda cabe ao usuário/desenvolvedor configurar como os contêineres serão criados (*i.e.*, quais aplicações, bibliotecas e arcabouços estarão em cada contêiner), o que pode ser complexo dependendo da aplicação em questão. Adicionalmente, por conta das vulnerabilidades que contêineres podem introduzir nas instalações de PAD (*e.g.*, *clusters* e supercomputadores), a sua adoção e uso são restritos a poucas tecnologias (*e.g.*, Singularity), ao invés do clássico Docker. Contêineres são aplicações *stateless*, logo, ao fim de uma execução os dados são perdidos e o usuário deve cuidar da persistência de dados (*i.e.*, salvar os dados em volumes ou diretórios).

A captura de dados de proveniência é normalmente uma atividade intensiva em dados, podendo interferir no desempenho da execução da aplicação no contêiner, dependendo de como seja configurada. Essa configuração dos contêineres se torna ainda mais crítica quando os dados de proveniência são utilizados para monitorar e ajustar a execução da aplicação principal. Logo, dependendo de como o usuário configurar os contêineres (*e.g.*, definir quantos e em qual contêiner estarão os componentes da aplicação principal e os serviços de proveniência), muitos problemas podem surgir, como o aumento de comunicação entre contêineres e do tempo de transferência e criação de imagens, e conflitos de versão de *software* entre a pilha da aplicação principal e a pilha de *software* para captura de proveniência. Além disso, alguns componentes da aplicação já vêm conteinerizados e há uma ordem para iniciar cada componente antes que a aplicação principal execute.

Propomos nesse artigo o arcabouço `ProvDeploy` para auxiliar a implantação e consequente execução de aplicações com apoio de dados de proveniência em ambientes de PAD. O `ProvDeploy` considera toda a pilha de *software* necessária tanto pela aplicação principal quanto pelo serviço de proveniência no momento da implantação. No entanto, a escolha da configuração dos contêineres ainda precisa ser definida pelo usuário. O mais comum e mais simples é colocar toda a pilha de software em um só contêiner, no entanto, explorar diferentes componentes na aplicação se torna complexo. No outro extremo estaria a definição de um contêiner para cada componente externo da aplicação. Nesse caso, a sobrecarga de ativação dos diversos contêineres pode comprometer o desempenho da aplicação. Não foram encontradas na literatura avaliações de configurações de contêineres considerando o uso intensivo de dados em ambientes PAD. Apresentamos o uso do `ProvDeploy` em diferentes configurações com o SciPhy. Os resultados dessa avaliação indicam que a sobrecarga do uso de vários contêineres não é o fator determinante na configuração e sim os aspectos de usabilidade. O restante do artigo apresenta na Seção 2 o referencial teórico e os trabalhos relacionados. A Seção 3 apresenta o arcabouço

`ProvDeploy`, a Seção 4 mostra a avaliação de diferentes cenários de configuração com o SciPhy, e, finalmente, a Seção 5 conclui o artigo.

2. Referencial Teórico e Trabalhos Relacionados

2.1. Princípios de Conteinerização

Apesar de o termo "contêiner" inicialmente remeter a um local de armazenamento, um contêiner de *software* [Struhár et al. 2020] é mais que apenas um repositório de armazenamento, já que ele apoia o empacotamento da pilha de *software* para que ela se torne executável em múltiplos ambientes computacionais. A conteinerização permite mover a execução de um *software* de um ambiente para outro de forma simplificada, desde que os ambientes ofereçam o suporte necessário para conteinerização. As diferentes tecnologias de conteinerização, apesar de serem compatíveis em sua maioria (por meio do *runc*), se diferenciam em suas *engines*, *e.g.,* Docker, Singularity e CRI-O. A *engine* é um componente responsável pela criação de imagens de contêiner a partir de uma descrição fornecida pelo usuário. Uma imagem de contêiner é um pacote de *software* executável que contém código-fonte, bibliotecas, arquivos, dados, *etc*. A descrição fornecida pelo usuário deve conter qual a pilha de *software* a ser movida, suas dependências e dados necessários para sua execução. Com base nessa descrição, a *engine* empacota esses elementos e usa técnicas de virtualização para compor uma imagem da pilha de *software* para ela poder ser descompactada e executada em seu destino. Essa imagem pode ser usada para criar outras imagens ou para executar processos isolados (conteinerizados).

As imagens de contêiner podem ser compartilhadas entre diferentes usuários e publicadas em servidores públicos de contêiner como DockerHub (https://hub.docker.com/). Esses servidores são apontados como uma das razões da popularização do uso de contêineres, uma vez que eles disponibilizam imagens que podem ser publicadas e utilizadas por empresas e usuários. Entretanto, por compartilharem o *kernel*, os contêineres podem apresentar riscos à segurança do ambiente no qual executam. Ainda, por não serem modificáveis, as imagens de contêiner com o tempo podem se tornar vulneráveis a ataques cibernéticos. Existem soluções de contêiner que são voltadas especificamente para ambientes de PAD. Dentre as soluções para PAD, podemos citar Shifter, CharlieCloud e Singularity que atendem a necessidade de customização de ambientes proporcionando maior liberdade ao usuário dentro do ambiente PAD. Essas soluções visam à mobilidade de ambientes de execução e mitigam problemas causados pelo compartilhamento de *kernel*. No entanto, ainda há problemas relacionados aos uso de contêineres em PAD, como as permissões nas diferentes instalações de PAD, familiaridade com a tecnologia e decisão de melhores formas de implantação, compatibilidade de arquiteturas e configuração dos contêineres para executar determinada aplicação. Quanto à implantação, é comum que usuários de PAD criem uma única imagem que satisfaça toda a pilha de *software* de sua aplicação, o que se torna de difícil manutenção. Por outro lado, manter os componentes todos em contêineres separados, ajuda no reuso de imagens, mas demanda que o usuário realize algum tipo de gerência desses contêineres durante a execução da aplicação, como envio de chamadas ao serviço de captura de dados de proveniência. Assim, a configuração de contêineres se torna uma questão importante.

2.2. Serviços de Proveniência

Os dados de proveniência podem ser definidos como a descrição da origem de um elemento de dados, a atividade que o gerou e o agente desta geração [Moreau and Groth 2013].

Dessa forma, os dados de proveniência dizem respeito ao histórico de derivação de um conjunto de resultados, incluindo as atividades (ou processos) que levaram a tais resultados. As vantagens da captura de dados de proveniência são bem conhecidas, como aumento de qualidade dos dados, interpretação e análise dos resultados, além de sua reprodução [Moreau and Groth 2013]. A definição dos dados de proveniência a serem capturados requer a determinação dos pontos do *workflow* a serem capturados. Os dados de proveniência são capturados à medida que a aplicação é executada e persistidos em arquivos ou sistemas de banco de dados, aumentando a movimentação de dados durante a execução.

Além da captura, a consulta aos dados de proveniência pode ser necessária ainda durante a execução [Mattoso et al. 2015]. Esse tipo de consulta se mostra necessário para aplicações em PAD, pois permite o acompanhamento da execução por meio do caminho de derivação dos dados. Um fator fundamental para a captura de proveniência em aplicações com PAD é que a sobrecarga adicionada pelo serviço de proveniência seja baixa a ponto de ser desprezível. A DfAnalyzer [Silva et al. 2020] é um exemplo de ferramenta que realiza captura e gerência de dados de proveniência, possibilita o processamento de consultas em tempo de execução em ambientes PAD, e provê uma persistência de dados assíncrona, que não compromete o tempo de execução da aplicação principal. Além disso, a DfAnalyzer permite que o usuário especifique os dados de interesse a serem capturados.

2.3. Trabalhos Relacionados

Apesar da popularização do Docker e Kubernetes, essas soluções de conteinerização ainda encontram limitações em ambientes de PAD [Baliś et al. 2022, Zhou et al. 2021]. Existem abordagens que combinam o Docker e serviços de dados de proveniência para apoiar a reprodução e a execução de aplicações [Chirigati et al. 2016, Malik et al. 2018]. O Sciunits [Malik et al. 2018] é uma ferramenta para repetição e reuso de aplicações. O Reprozip [Chirigati et al. 2016] é uma ferramenta pioneira que disponibiliza um artefato reprodutível em diferentes formatos (*e.g.*, zip, Docker, vagrant). O Sciunits gera um objeto de pesquisa reutilizável (*i.e.*, *Research Object* [Bechhofer et al. 2010]) sendo uma imagem Docker criada a partir das chamadas de sistema ocorridas durante a execução. Esse objeto de pesquisa inclui os dados consumidos e gerados pela aplicação, documentação e dados de proveniência, de forma que possa ser utilizado para repetir a execução em diferentes ambientes. Ainda na direção de auxiliar a repetição e verificação de resultados, [Williams and Tosh 2021] propõem *HyperProvenance* uma arquitetura de alto nível para ambientes PAD heterogêneos e que permite a execução de aplicações baseadas em microsserviços e uso de contêineres. Apesar de ter seu foco em PAD, a *HyperProvenance* não auxilia o usuário na configuração e implantação de sua aplicação em um ambiente de PAD, mas sim em registrar dados que permitam a repetição e verificação de resultados.

O PROV-CRT [Ahmad et al. 2020] é uma abordagem cujo objetivo é capturar dados de proveniência da imagem durante sua criação para auditoria e repetição, porém, esta solução não realiza captura de dados durante a execução da imagem gerada. A captura de proveniência em contêineres não é uma tarefa simples. Uma das dificuldades da captura é diferenciar os processos conteinerizados dos processos pertencentes ao sistema operacional hospedeiro, por isso, há soluções que realizam essa separação adicionando sensibilidade ao contêiner [Hassan et al. 2018], *namespaces* [Pasquier et al. 2017] ou ambos [Chen et al. 2021]. Estratégias de integração de contêineres e sistemas de *workflow* são propostas em [Zheng and Thain 2015] que integra o Docker com o Makeflow e o Work Queue. Enquanto o Makeflow é uma ferramenta de linha de comando para a

execução de aplicações científicas intensivas em dados em vários sistemas de execução distribuídos, o Work Queue é um mecanismo de execução leve para sistemas distribuídos. Assim, o trabalho de [Zheng and Thain 2015] apresenta o apoio à criação do ambiente conteinerizado, mas sem considerar questões relacionadas à proveniência. Além das abordagens supracitadas, existem soluções comerciais que visam a auxiliar a execução com uso de contêineres como o Pachyderm (https://www.pachyderm.com/), o Kubeflow (https://www.kubeflow.org/) e o Polyaxon (https://polyaxon.com/). Todas essas soluções são baseadas em Kubernetes, o que as torna menos apropriadas para ambientes de PAD [Baliś et al. 2022, Zhou et al. 2021].

3. Abordagem Proposta: `ProvDeploy`

O `ProvDeploy` é um arcabouço que visa a promover a implantação de aplicações científicas executadas em ambientes de PAD integrada a serviços de proveniência. Por meio dos dados de proveniência, os usuários podem realizar o acompanhamento de suas aplicações em tempo real, ajudando a depuração e as mudanças de parâmetros durante a execução (caso a aplicação permita adaptações). O `ProvDeploy` recebe como entrada (i) os conjuntos de dados a serem processados, (ii) a especificação da aplicação principal e (iii) um catálogo que contém os endereços das imagens de contêineres disponíveis que podem ser implantadas. A partir das entradas, o `ProvDeploy` produz como saída um objeto de pesquisa (*i.e.*, *research object*) que contém todos os dados, metadados, bibliotecas e dependências usados na execução da aplicação em questão. As imagens de entrada podem ser encontradas em diferentes servidores de imagens públicas, *e.g.*, DockerHub, Binder, NVIDIA NGC, *etc*. A ideia por trás do uso de imagens públicas é reduzir o tempo de implantação de contêineres e ajudar o usuário a explorar imagens alternativas.

Com base nas dependências encontradas na especificação da aplicação principal, o `ProvDeploy` é capaz de iniciar contêineres a partir de várias imagens existentes, incluindo imagens de contêiner com o serviço de proveniência escolhido. É importante ressaltar que o `ProvDeploy` permite ao usuário escolher entre vários serviços de proveniência disponíveis, mas apenas um deles pode ser definido como padrão. O serviço de proveniência escolhido pode exigir a implantação de contêineres com sistemas de gerência de banco de dados (SGBDs), *e.g.*, MonetDB, PostgreSQL, *etc*. O `ProvDeploy` então define a melhor estratégia de configuração dos contêineres. Ao final da execução da aplicação no ambiente de PAD, o `ProvDeploy` "une" todos os contêineres usados em uma única imagem, criando assim um objeto de pesquisa compartilhável. Observe que o objeto de pesquisa gerado não reflete as características do ambiente de PAD no qual a aplicação foi executada. Ele contém apenas a pilha de *software* necessária para executar a aplicação principal, o serviço de proveniência escolhido e seu banco de dados de proveniência. Também vale a pena mencionar que o `ProvDeploy` não foi projetado para substituir orquestradores de contêineres (como Kubernetes), de forma que pode ser usado acoplado ao Kubernetes, uma vez que o `ProvDeploy` define quais contêineres farão parte do *cluster* de Kubernetes. O código-fonte do `ProvDeploy` pode ser obtido em `https://bitbucket.org/lilianeKunstmann/provdeploy/`.

A arquitetura do `ProvDeploy` é apresentada na Figura 1, tendo cinco componentes principais: (i) *Catálogo*, (ii) *Inicializador*, (iii) *Prov-Parser*, (iv) *Deployer* e (v) *Wrapper*. O *Catálogo* usa um banco de dados interno para armazenar os metadados associados às imagens dos contêineres que podem ser utilizadas pelo `ProvDeploy`. Ele contém o

endereço para acessar cada imagem, sua descrição, seu arquivo de definição (*e.g.*, *Dockerfile*, *recipes*), como executá-las e seus requisitos (*e.g.,* criação de volume, portas públicas). O *Catálogo* contém informações sobre os serviços de proveniência disponíveis que podem ser usados em conjunto com o `ProvDeploy` (*e.g.*, DfAnalyzer, noWorkflow) e imagens de contêineres de SGBDs (*e.g.*, MonetDB, PostgreSQL, mySQL). Este componente também representa informações acerca da versão do arquivo de configuração de contêiner.

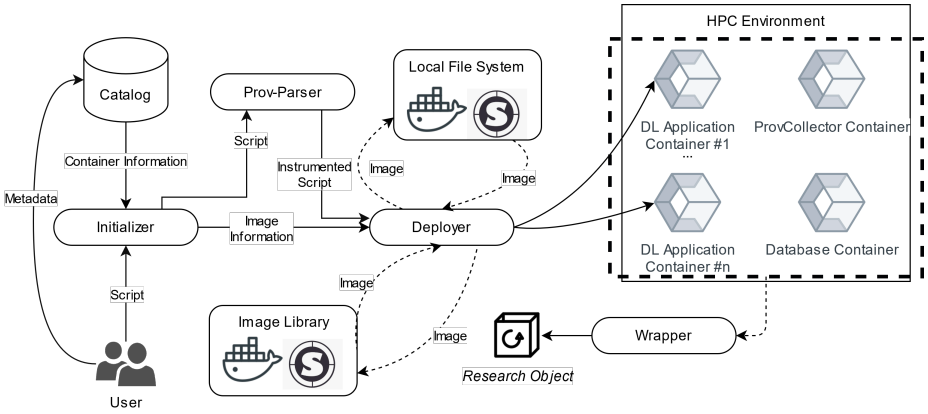

Figura 1. Arquitetura do `ProvDeploy`

O *Inicializador* consulta o *Catálogo* e envia as referências de imagens e detalhes de execução para o *Deployer*. O *Inicializador* verifica se o código da aplicação principal já se encontra instrumentado para trabalhar com o serviço de proveniência escolhido. Se o código estiver instrumentado para capturar dados de proveniência, o *Deployer* é invocado. A instrumentação pode ser definida como a inserção de chamadas ao serviço de proveniência no código da aplicação. Se o código não estiver instrumentado, o *Prov-Parser* é invocado e instrumenta o código da aplicação, identificando os pontos onde os dados de proveniência podem ser capturados. Essa instrumentação muda conforme o serviço de proveniência. O *Prov-Parser* identifica funções no código da aplicação e associa cada função a uma transformação de dados. Se o código não apresentar funções explícitas (*e.g.*, um *def <<Nome da Função>>():* em Python), o *Prov-Parser* assume uma única função para toda a aplicação. Vale ressaltar que a instrumentação manual, apesar de mais trabalhosa, consegue configurar o serviço de proveniência para funcionar na granularidade desejada. Em sua versão atual, o `ProvDeploy` define a DfAnalyzer como o serviço de proveniência padrão. Para armazenar os dados capturados, a DfAnalyzer usa o MonetDB como SGBD e o FastBit para a indexação de dados. Finalmente, o *Deployer* é responsável por configurar o ambiente onde a aplicação do usuário e o serviço de proveniência são executados. O *Deployer* avalia diferentes estratégias de configuração de conteinerização.

No `ProvDeploy` são definidas cinco estratégias de configuração do ambiente, conforme apresentado na Figura 2: (i) *Imagem Única* - um contêiner para aplicação principal, serviço de proveniência e dados produzidos, (ii) *Modular Parcial (1)* - um contêiner para a aplicação e o serviço de proveniência e outro para dados produzidos, (iii) *Modular Parcial (2)* - um contêiner para a aplicação principal e os dados produzidos e outro para o serviço de proveniência, (iv) *Modular Parcial (3)* - um contêiner para a aplicação e outro para o serviço de proveniência e dados produzidos, e (v) *Modular Total* - um contêiner para a aplicação, um para os dados produzidos e outro para o serviço de proveniência . Em sua configuração padrão, o *Deployer* adota a estratégia *Modular Total*, porém o `ProvDeploy` pode explorar diferentes configurações e registrar a que apresentou melhor desempenho de

forma que a configuração possa ser usada em futuras execuções. Cada estratégia explorada apresenta vantagens e desvantagens, conforme discutido na Seção 4.

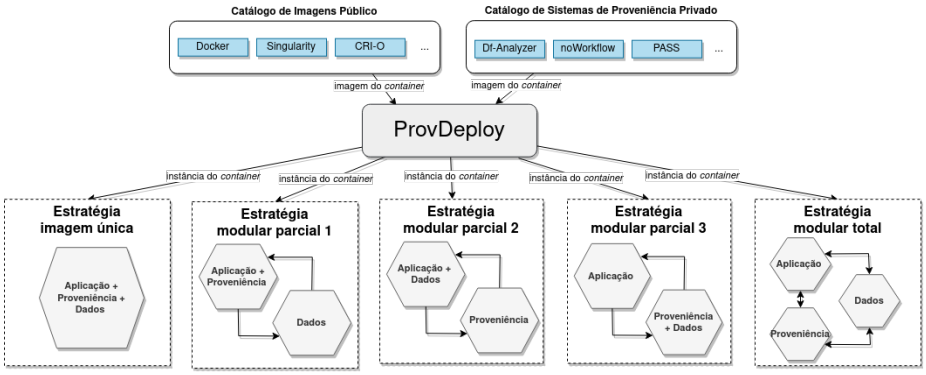

Figura 2. Estratégias de Configuração do Ambiente Conteinerizado

4. Avaliação Experimental

Esta seção apresenta a avaliação experimental do `ProvDeploy` com o SciPhy. O SciPhy é uma aplicação para gerar árvores filogenéticas com máxima verossimilhança. Ela foi projetada inicialmente para trabalhar com sequências de aminoácidos, mas pode ser usada para outros tipos de sequências biológicas. O SciPhy é um script composto por quatro atividades principais que são: (i) construção do alinhamento múltiplo (MSA); (ii) conversão de formato do alinhamento; (iii) pesquisa sobre o melhor modelo evolutivo a ser usado; e (iv) construção da árvore filogenética. Essas atividades executam, respectivamente, as seguintes bibliotecas e programas: programas de alinhamento múltiplo de sequências (permitindo ao usuário escolher entre o MAFFT, o Kalign, o ClustalW, o Muscle, ou o ProbCons), o ReadSeq, o ModelGenerator, e o RAxML. Apesar de conceitualmente o SciPhy ser computacionalmente simples, na prática ele pode ser demasiadamente complexo de ser executado devido ao processamento intensivo de dados consumidos e produzidos. Um experimento típico de filogenia pode analisar centenas ou milhares de arquivos contendo centenas ou milhares de sequências biológicas.

4.1. Configuração do Ambiente e do experimento

O SciPhy foi implantado com o `ProvDeploy` em dois ambientes: (i) o supercomputador Santos Dumont (SDumont), e (ii) o ambiente de nuvem AWS. Em ambos os casos o Singularity 3.8 foi o *software* de contêiner, por ser voltado a PAD. No SDumont foi utilizado um nó computacional com duas CPUs com processador Intel Xeon E5-2695v2 *Ivy Bridge* de 2,4GHZ, 24 núcleos (12 por CPU), 64GB DDR3 RAM e sistema operacional Linux RedHat 7.6, e para perfilagem a biblioteca sysstat 12. Na AWS foram utilizadas máquinas virtuais do tipo t3a.large com duas vCPU com processador AMD EPYC 7000 series turbo clock 2.5 GHz, 8 GiB, virtualização HVM, Linux/Unix, e para perfilagem a biblioteca sysstat 10.

O protocolo do experimento consistiu em executar a aplicação SciPhy com o serviço de captura de proveniência DfAnalyzer nos ambientes SDumont e AWS. O `ProvDeploy` foi configurado para explorar cada uma das estratégias de configuração do ambiente descritas na Tabela 1.

Tabela 1. Descrição das Estratégias de Configuração do Ambiente

Estratégia	Descrição	Objetivo
Estratégia imagem única	Contêiner único contendo todas as dependências para coleta de proveniência e execução do SciPhy	Menor quantidade de contêineres, não há comunicação entre contêineres e uma imagem é suficiente para realizar a execução completa.
Estratégia modular parcial	Dois contêineres com 3 variações: (1) um com a aplicação Sciphy e o serviço da proveniência e outro com o SGBD e os dados de proveniência; (2) um com a aplicação Sciphy e o SGBD e dados de proveniência e o segundo com o serviço da proveniência; (3) um com a aplicação Sciphy e o segundo com o serviço de proveniência, o SGBD e os dados de proveniência	(1) visa o compartilhamento de características (*e.g.*, bibliotecas); (2) atende a possibilidade de algum conflito entre características compartilhadas (*e.g.*, versão do Python); (3) atende a possibilidade e uma das aplicações já esteja previamente conteinerizada permitindo o reuso de imagens
Estratégia modular total	Três contêineres: o primeiro com a aplicação SciPhy, o segundo com o serviço de proveniência e o terceiro com o SGBD e os dados de proveniência.	Este caso representa uma modularização total contemplando a possibilidade de utilização de imagens presentes em registros públicos para execução da aplicação.

4.2. Discussão dos Resultados

As cinco estratégias apresentadas na Tabela 1 foram avaliadas segundo critérios de usabilidade e desempenho computacional. A estratégia modular parcial (MP) nas variações 1, 2 e 3 possuem a mesma avaliação qualitativa e não apresentaram diferenças significativas no desempenho, por isso, são apresentadas apenas como estratégia MP, sendo usado o desempenho da MP de variação (1). A Tabela 2 foi organizada com os critérios de: (i) facilidade de manutenção das imagens, (ii) facilidade de repetição da execução, (iii) facilidade de reaproveitamento e a (iv) possibilidade de compartilhamento de características. Em relação à facilidade de manutenção, quanto maior a pilha de *software* da imagem, maior a dificuldade de atualização principalmente considerando a possibilidade de conflitos e *softwares* legados. Os tempos de *pull* e *push* aumentam proporcionalmente ao tamanho das imagens bem como as possíveis vulnerabilidades, ainda, quando uma imagem é responsável pela execução de todas as tarefas, esta iniciará múltiplos processos pai, o que não é aconselhável, pois dificulta o controle de ferramentas externas (ex. Kubernetes) e facilita a geração de processos zumbis[2], logo consideramos a estratégia imagem única (IU) como difícil. Nesse quesito, a estratégia modular total (MT) é considerada fácil, pois pode ser executada utilizando apenas imagens públicas que podem ser minimizadas, com *pull* e *push* mais rápidos e há apenas um processo pai por imagem. As variações da estratégia MP são de dificuldade moderada por que ao menos uma das imagens pode ser baixada de repositórios públicos.

Em relação à facilidade de repetição, quanto mais imagens envolvidas em uma execução, maior a necessidade de gerenciar a interação entre as mesmas e maior a dependência das diferentes partes da aplicação. Nesse critério, apenas a estratégia IU pode ser conside-

[2]cloud.google.com/architecture/best-practices-for-building-containers

Tabela 2. Avaliação de usabilidade

Estratégia	Manutenção	Repetição	Reaproveitamento	Compartilhamento
Única - IU	Difícil	Fácil	Difícil	Sim
Parcial - MP	Moderado	Moderado	Moderado	Sim
Total - MT	Fácil	Moderado	Fácil	Não

rada fácil, uma vez que só gerencia um contêiner autossuficiente. Em relação ao reaproveitamento quanto à possibilidade de reutilizar uma determinada imagem para execução de diferentes tarefas, a estratégia IU por ser muito específica em sua pilha e muito grande acaba sendo de reaproveitamento difícil. O compartilhamento de características acontece quando um mesmo *software* ou biblioteca é utilizado para diferentes aplicações. No caso, SciPhy e DfAnalyzer podem compartilhar a instalação do Java nas estratégias IU e MP (1), mas não na MT. Observamos que a estratégia MT é a que mais favorece qualitativamente a execução via contêiner. Entretanto, observamos que as configurações podem atender diferentes necessidades, como casos em que uma parte da aplicação já esteja conteinerizada, por isso a avaliação do desempenho computacional das estratégias se faz necessária.

Avaliamos as estratégias IU, MP e MT com as métricas de tempo de execução e consumo de CPU. A Tabela 3 apresenta o tempo de execução médio (*i.e.*, $\overline{x}$) e o desvio padrão (*i.e.*, σ) em horas, para cinco execuções. Ao analisar a Tabela 3 podemos perceber que existe uma diferença grande entre os tempos de execução do SDumont e AWS, mas no caso do SDumont nota-se pouca variação entre os tempos de execução para cada estratégia. Esse resultado pode indicar que, em ambientes PAD, as diferentes estratégias não fazem grande diferença, havendo maior liberdade na escolha da estratégia de acordo com a necessidade de uso. No caso da execução na AWS, temos até 42 minutos de diferença entre as execuções. Considerando o total de 24h, esse valor pode não ser significativo, mas como o ambiente de nuvem cobra por segundo executado, essa diferença pode gerar impactos financeiros. A estratégia MP (1) foi a que apresentou o menor tempo de execução nos dois ambientes. No entanto, ao realizar o Teste-t de hipótese com valor de $alfa = 5\%$ em todos os casos $p - valor > alfa$, não rejeitando a hipótese nula, logo, não há diferença significativa entre as médias apresentadas.

Tabela 3. Tempos de Execução (h)

Estratégia	SDumont		AWS	
	x	σ	x	σ
Único - IU	3,673	0,026	25,046	0,697
Parcial - MP	3,666	0,001	24,418	0,475
Total - MT	3,690	0,015	24,717	0,713

O consumo de CPU das execuções das estratégias no SDumont e na AWS foi coletado por segundo e a média por minuto é apresentada dentro de um intervalo de 200 minutos nas Figuras 3, 4, 5, e 6. Os valores apresentados na Tabela 3 foram calculados sobre cinco execuções, mas os gráficos apresentam apenas três para facilitar a visualização. As estratégias IU, MP e MT apresentaram comportamento muito parecido. De modo geral, em todos os casos o uso de CPU é similar, ficando em 11% no SDumont e 30% na AWS.

5. Conclusão

O uso de contêineres de *software* como forma de facilitar a implantação de aplicações é uma realidade, tanto na academia quanto na indústria. Apesar de o conceito de contêiner representar um avanço, ainda existem desafios em seu uso em ambientes de PAD e

Figura 3. Consumo de CPU - Estratégias Única e Parcial - SDumont

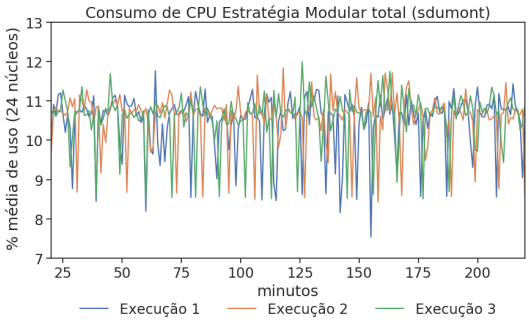

Figura 4. Consumo de CPU - Estratégia Total - SDumont

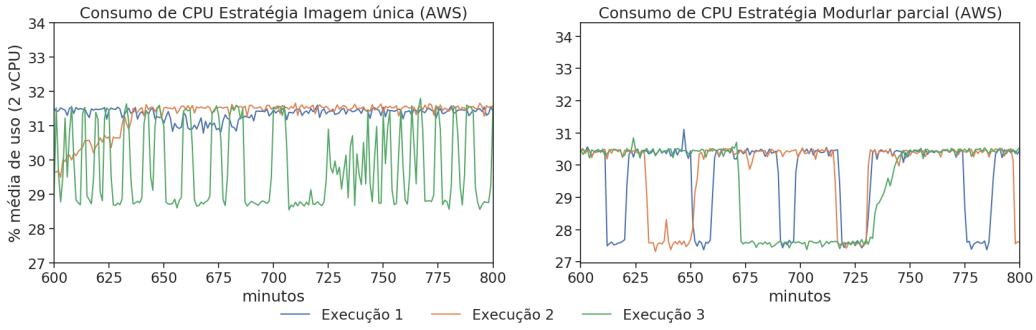

Figura 5. (rcial - AWS

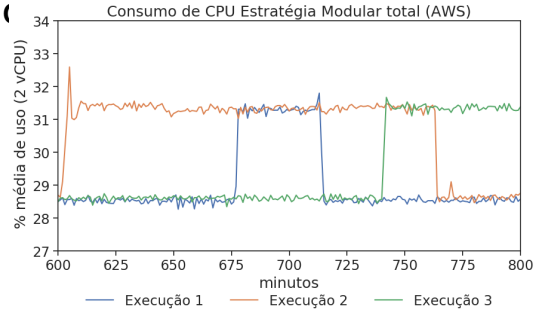

Figura 6. Consumo de CPU - Estratégia Total - AWS

quando executamos aplicações científicas que necessitam de serviços de reprodução, como a captura de dados de proveniência. O usuário tem a tarefa, muitas vezes não trivial, de definir uma configuração de contêineres para hospedar a aplicação principal, os dados consumidos e produzidos e o serviço de captura de proveniência. Se a configuração for mal realizada, impactos no tempo de execução e no custo financeiro poderão ocorrer. Assim, neste trabalho, apresentamos o `ProvDeploy`, um arcabouço que visa a auxiliar a execução de aplicações conteinerizadas. O `ProvDeploy` facilita a investigação de diferentes configurações de contêineres de forma a encontrar a estratégia mais adequada para uma determinada aplicação. O `ProvDeploy` foi avaliado por meio da execução da aplicação de Bioinformática SciPhy no SDumont e na AWS, e os resultados evidenciaram que o uso de

vários contêineres não implica custos adicionais significativos e possui a vantagem de ser flexível e modular. Como trabalhos futuros pretendemos incluir no `ProvDeploy` um módulo de recomendação, de avaliação da aplicação submetida pelo usuário para classificá-la em categorias e sugerir a melhor estratégia de execução baseada em execuções anteriores de aplicações com comportamentos semelhantes. Planejamos realizar uma avaliação sintética com mais instâncias e utilizar GPUs para verificar se o comportamento observado se mantém.

Agradecimentos

Trabalho realizado com apoio do CNPq, FAPERJ e CAPES Código de financiamento 001.

Referências

Ahmad, R., Nakamura, Y., Manne, N. N., and Malik, T. (2020). Prov-crt: Provenance support for container runtimes. In *TaPP 2020*, pages 1–3.

Baliś, B., Broński, A., and Szarek, M. (2022). Auto-scaling of scientific workflows in kubernetes. In *ICCS*, pages 33–40. Springer.

Bechhofer, S., De Roure, D., Gamble, M., Goble, C., and Buchan, I. (2010). Research objects: Towards exchange and reuse of digital knowledge. *Nature Proc.*, pages 1–6.

Chen, X., Irshad, H., Chen, Y., Gehani, A., et al. (2021). Clarion: Sound and clear provenance tracking for microservice deployments. In *USENIX Security*, pages 3989–4006.

Chirigati, F., Rampin, R., Shasha, D. E., and Freire, J. (2016). Reprozip: Computational reproducibility with ease. In *SIGMOD*, pages 2085–2088. ACM.

de Oliveira, D., Ocaña, K. A., Baião, F., and Mattoso, M. (2012). A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds. *J. Grid Comput.*, 10(3):521–552.

Elia, D., Fiore, S., and Aloisio, G. (2021). Towards HPC and big data analytics convergence: Design and experimental evaluation of a HPDA framework for escience at scale. *IEEE Access*, 9:73307–73326.

Freire, J., Koop, D., Santos, E., and Silva, C. T. (2008). Provenance for computational tasks: A survey. *Computing in science & engineering*, 10(3):11–21.

Guedes, T., Jesus, L. A., Ocaña, K. A., Drummond, L., and de Oliveira, D. (2020). Provenance-based fault tolerance technique recommendation for cloud-based scientific workflows: a practical approach. *Cluster Comp.*, 23(1):123–148.

Harrell, S. L., Michael, S., and Maltzahn, C. (2022). Advancing adoption of reproducibility in HPC: A preface to the special section. *IEEE Trans. Par. Dist. Syst.*, 33(9):2011–2013.

Hassan, W. U., Aguse, L., Aguse, N., Bates, A., and Moyer, T. (2018). Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*, pages 1–15.

Liu, P. and Guitart, J. (2022). Performance characterization of containerization for HPC workloads on infiniband clusters: an empirical study. *Clust. Comput.*, 25(2):847–868.

Malik, T., Yuan, Z., Essawy, B. T., Castronova, A. M., Gan, T., Tarboton, D. G., Goodall, J. L., Peckham, S. D., Choi, E., and Bhatt, A. (2018). Sciunits: Reusable research objects. In *AGU Fall Meeting Abstracts*, volume 2018, pages IN34B–10.

Mattoso, M., Dias, J., Ocana, K. A., Ogasawara, E., Costa, F., Horta, F., Silva, V., and De Oliveira, D. (2015). Dynamic steering of hpc scientific workflows: A survey. *Future Generation Computer Systems*, 46:100–113.

Moreau, L. and Groth, P. (2013). Provenance: an introduction to prov. *Synthesis lectures on the semantic web: theory and technology*, 3(4):1–129. Morgan & Claypool Publishers.

Ocaña, K. A., Silva, V., de Oliveira, D., and Mattoso, M. (2015). Data analytics in bioinformatics: Data science in practice for genomics analysis workflows. In *IEEE e-Science*, pages 322–331. IEEE.

Pasquier, T., Han, X., Goldstein, M., Moyer, T., Eyers, D., Seltzer, M., and Bacon, J. (2017). Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 405–418, New York, NY, USA. ACM.

Silva, V., Campos, V., Guedes, T., Camata, J., de Oliveira, D., Coutinho, A. L., Valduriez, P., and Mattoso, M. (2020). Dfanalyzer: Runtime dataflow analysis tool for computational science and engineering applications. *SoftwareX*, 12:100592.

Struhár, V., Behnam, M., Ashjaei, M., and Papadopoulos, A. V. (2020). Real-time containers: A survey. In *Fog-IoT*, volume 80 of *OASIcs*, pages 7:1–7:9.

Williams, A. and Tosh, D. K. (2021). Scientific workflow provenance architecture for heterogeneous hpc environments. In *2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0921–0927.

Yuan, D. Y. and Wildish, T. (2020). Bioinformatics application with kubeflow for batch processing in clouds. In *HPDC*, pages 355–367. Springer.

Zheng, C. and Thain, D. (2015). Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *WVTDC*, pages 31–38.

Zhou, N., Georgiou, Y., Pospieszny, M., Zhong, L., Zhou, H., Niethammer, C., Pejak, B., Marko, O., and Hoppe, D. (2021). Container orchestration on hpc systems through kubernetes. *Journal of Cloud Computing*, 10(1):1–14.