AkôFlow: um *Middleware* para Execução de *Workflows* Científicos em Múltiplos Ambientes Conteinerizados*

Wesley Ferreira[1], Liliane Kunstmann[2], Aline Paes[1], Marcos Bedo[1], Daniel de Oliveira[1]

[1]Instituto de Computação – Universidade Federal do Fluminense (UFF)

wesleyferreira@id.uff.br, {alinepaes, marcosbedo, danielcmo}@ic.uff.br

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

lneves@cos.ufrj.br

Abstract. *Various workflows produce a large volume of data and require parallelism techniques and distributed environments to reduce execution time. Workflow Systems run these workflows, which support efficient execution but focus on specific environments. Container technology has emerged as a solution for applications to run in heterogeneous environments through OS virtualization. Although there are container management and orchestration solutions, e.g., Kubernetes, they do not focus on scientific workflows. In this paper, we propose* `AkôFlow`*, a middleware for parallel execution of scientific workflows in containerized environments.* `AkôFlow` *allows scientists to explore the parallel execution of activities with support for provenance capture. We evaluated* `AkôFlow` *with an astronomy workflow, and the results were promising.*

Resumo. *Diversos workflows produzem um grande volume de dados e requerem técnicas de paralelismo e ambientes distribuídos para reduzir o tempo de execução. Esses workflows são executados por Sistemas de Workflow, que apoiam a execução eficiente, mas focam em ambientes específicos. A tecnologia de contêineres surgiu como solução para que uma aplicação execute em ambientes heterogêneos por meio da virtualização do SO. Embora existam soluções de gerenciamento e orquestração de contêineres, e.g., Kubernetes, elas não focam em workflows científicos. Neste artigo, propomos o* `AkôFlow`*, um middleware para execução paralela de workflows científicos em ambientes conteinerizados. O* `AkôFlow` *permite ao cientista explorar a execução paralela de atividades, com apoio à captura de proveniência. Avaliamos o* `AkôFlow` *com um workflow da astronomia e os resultados foram promissores.*

1. Introdução

Os *workflows* são abstrações usadas para modelar processos complexos, geralmente representados por Grafos Acíclicos Dirigidos (DAGs). Nessas representações, os vértices correspondem a atividades (associadas a programas) e as arestas indicam dependências de dados entre as atividades [de Oliveira et al. 2019]. A execução de uma atividade, denominada *ativação* [Ogasawara et al. 2011] a partir deste ponto, consome um subconjunto específico de dados e parâmetros. Embora seja possível implementar *workflows* de diversas

*O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001. Os autores gostariam de agradecer também a CNPq e FAPERJ. Os experimentos foram realizados com o suporte do programa *Google Cloud Research Credits* com o número GCP19980904O.

formas, é comum utilizar Sistemas de *Workflows*, que permitem aos usuários especificar, executar e monitorar *workflows* em diferentes infraestruturas computacionais.

Muitos *workflows* são intensivos em processamento e produção de dados, exigindo a aplicação de técnicas de paralelismo aliadas ao uso de ambientes de processamento de alto desempenho (HPC), onde múltiplos processadores estão disponíveis para uso, de forma que resultados sejam obtidos em tempo hábil. Diversos sistemas de *workflows* já oferecem mecanismos para a execução de *workflows* nesses ambientes, como o Pegasus [Deelman et al. 2021], o SciCumulus [de Oliveira et al. 2010], o SAMbA-RaP [Guedes et al. 2020], o Chiron [Ogasawara et al. 2013] e o Parsl [Babuji et al. 2019]. Embora representem um avanço significativo, esses sistemas são projetados para infraestruturas de computação específicas. Por exemplo, o Pegasus requer o uso do escalonador HTCondor [Shah et al. 2014], que pode não estar presente em muitas infraestruturas. Já o SciCumulus foi projetado para uso em ambientes de nuvem [de Oliveira et al. 2012]. Além disso, *workflows* utilizam múltiplas bibliotecas, e essa miríade de soluções e suas dependências cria um ecossistema de *software* complexo. Comumente, as instalações de HPC enfrentam dificuldades para oferecer apoio a diversos *softwares* e bibliotecas [Kunstmann et al. 2022].

Uma das maneiras de diminuir o impacto da configuração da infraestrutura na execução de um *workflow* é por meio do uso de contêineres [Struhár et al. 2020]. A conteinerização pode ser definida como uma virtualização em nível de sistema operacional, onde há compartilhamento de *kernel*. Um contêiner empacota e isola aplicações, suas dependências, bibliotecas de sistema, configurações e outros binários, além dos arquivos de configuração necessários para seu funcionamento, permitindo que um mesmo contêiner seja migrado e executado entre infraestruturas computacionais heterogêneas (*e.g.*, *clusters* e nuvens) com o mínimo de esforço por parte do usuário.

Diversas soluções para conteinerização surgiram na última década, observando demandas específicas de HPC como o Singularity [Kurtzer et al. 2017] e plataformas de orquestração de contêineres que automatizam a implantação, o dimensionamento e a gerência de aplicações em contêineres, como o Kubernetes[1]. Apesar de representarem um avanço, tais soluções não foram projetadas para trabalhar com *workflows* científicos. Por exemplo, o orquestrador do Kubernetes, por padrão, considera um *pod*[2] por vez, sem levar em conta a dependência entre *pods* pertencentes à mesma atividade do *workflow* [Carrión 2023]. Além disso, essas soluções ainda apresentam limitações no que diz respeito à captura de dados de proveniência [Freire et al. 2008, de Oliveira et al. 2015] e ao escalonamento de contêineres de acordo com critérios definidos pelos usuários (*e.g.*, redução de tempo de execução ou redução de custo financeiro).

Por outro lado, os sistemas de *workflow* oferecem mecanismos de captura de dados de proveniência e políticas de escalonamento que podem ser definidas pelo usuário, mas ainda se mostram limitados no que se refere à execução em ambientes conteinerizados. Por exemplo, o Pegasus consegue executar *workflows* em ambientes conteinerizados, desde que todas as atividades do *workflow* possam ser executadas em um único contêiner, o que raramente é uma realidade. Embora o usuário possa combinar múltiplos contêineres, essa tarefa pode ser tediosa e propensa a erros, se realizada por um ser humano.

Dessa forma, o objetivo deste artigo é introduzir o *middleware* `AkôFlow`, que visa

[1] `https://kubernetes.io/pt-br/`

[2] Um *pod* é um grupo de contêineres que compartilham um espaço de rede local.

permitir a execução eficiente de *workflows* científicos em ambientes conteinerizados. O AkôFlow é construído sobre a plataforma Kubernetes, mas gerencia as dependências entre múltiplos contêineres no ambiente seguindo as dependências de dados definidas na especificação do *workflow*. Assim, o AkôFlow permite que diferentes contêineres com diferentes capacidades de processamento e armazenamento sejam utilizados na execução do *workflow*, oferecendo maior flexibilidade para o escalonamento. Além disso, o AkôFlow captura os dados de proveniência da execução do *workflow* no ambiente conteinerizado. Finalmente, o AkôFlow possibilita que uma mesma especificação de *workflow* seja executada em diversas infraestruturas diferentes, desde que ofereçam suporte ao Kubernetes. O AkôFlow foi avaliado no ambiente de nuvem *Google Cloud Platform* com a execução do *workflow* Montage, e os resultados mostraram-se promissores.

O restante do artigo segue a seguinte organização. A Seção 2 aborda conceitos importantes para o AkôFlow como a contêinerização e define o modelo de aplicação e arquitetura considerados. Na Seção 3, apresentamos a arquitetura do AkôFlow. Na Seção 4, discutimos os resultados experimentais obtidos. A Seção 5 trata dos trabalhos relacionados, e, por fim, a Seção 6, conclui o artigo.

2. Referencial Teórico

Esta seção apresenta definições importantes para a compreensão deste artigo. Inicialmente, definimos o modelo de aplicação considerado pelo AkôFlow, *i.e.*, *workflows* científicos, e a arquitetura de destino, *i.e.*, ambientes conteinerizados. Em seguida, discutimos alguns princípios da conteinerização.

2.1. Modelo de Aplicação e Arquitetura

O AkôFlow considera um conjunto de aplicações que podem ser modelados como um *workflow* representado como um DAG. Embora existam múltiplos formalismos para *workflows*, neste artigo seguimos o formalismo definido por [Silva et al. 2018] e [Teylo et al. 2017]. Um *workflow* é representado como $G = (V, A, a, \omega)$, onde tanto os conjuntos N de ativações quanto o conjunto D de arquivos de dados são representados como vértices. Dessa forma, $V = N \cup D$. Adicionalmente, A representa o conjunto de arestas no grafo G, onde a relação de consumo/produção entre ativações e arquivos de dados é explicitada. Finalmente, $a_i \in a$ representa a quantidade de processamento associada à ativação $i \in N$, enquanto ω_{ij} é o custo associado à aresta (i, j), onde $i, j \in N$.

Em G, toda ativação é sempre precedida e sucedida por um dado, logo, podemos definir o predecessor de uma ativação $i \in N$ como $Pred(i) = j \in N \wedge \exists d \in D \,|(j, d) \in A \wedge (d, i) \in A$. Da mesma forma, o sucessor de uma ativação $i \in N$ pode ser definido como $Succ(i) = j \in N \wedge \exists d \in D|\ (i, d) \in A \wedge (d, j) \in A$. Como seguimos o modelo de aplicação proposto por [Teylo et al. 2017], uma ativação é sempre precedida ou sucedida por um arquivo de dados em G. A Figura 1(a) apresenta um exemplo onde o *workflow* possui duas ativações (i_1 e i_2, em vermelho). Enquanto i_1 consome três arquivos de dados estáticos (*i.e.*, entradas do *workflow*), ele produz dois arquivos de dados dinâmicos (*i.e.*, gerados pela execução do *workflow*). Então, os arquivos de dados d_4 e d_5 são consumidos pela ativação i_2, que produz o arquivo de dados d_6.

O modelo de arquitetura define as principais características do ambiente alvo em que o AkôFlow atuará, *i.e.*, o ambiente conteinerizado. Podemos definir C_o como o conjunto de contêineres que podem ser instanciados para executar ativações $i \in N$. Cada contêiner

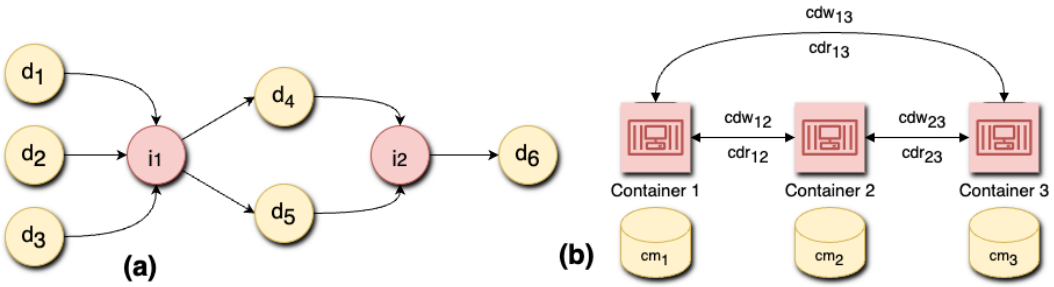

Figura 1. (a) Modelo de aplicação (b) Modelo de arquitetura do `AkôFlow`

j possui capacidade de armazenamento local cm_j e uma capacidade de processamento cp_j. Vale notar que cp_j é definido a partir da quantidade de vCPUs de um contêiner j. Os dados podem ser armazenados tanto em um volume acoplado ao contêiner $j \in C_o$ quanto em um volume desacoplado de contêineres $j \in B$. Vale notar que apenas os contêineres podem executar ativações do *workflow*. Um contêiner $j \in C_o$ é sempre implantado em uma máquina hospedeira $k \in M$, onde o conjunto M pode contar máquinas virtuais e máquinas físicas. Além disso, podemos definir um custo de comunicação de escrita cdw_i e de leitura cdr_i em cada *link* do ambiente, *i.e.*, em cada ligação entre dois contêineres. Finalmente, o tempo de execução de uma ativação $i \in N$ em um contêiner $j \in C_o$ é dado por $t_{ij} = a_i \times cp_j$. A Figura 1(b) apresenta um ambiente com três contêineres e suas características de processamento, armazenamento e os custos de escrita e leitura entre cada par de contêineres.

2.2. Princípios de Conteinerização

Um contêiner de *software* [Struhár et al. 2020] é uma ferramenta que possibilita o encapsulamento completo de uma pilha de *software*, tornando-a operacional em diversas infraestruturas computacionais. Através da conteinerização, torna-se viável a migração da execução de um *software* de um ambiente para outro de maneira transparente para o usuário, contanto que os ambientes envolvidos suportem a mesma tecnologia de conteinerização. Embora as diversas tecnologias de conteinerização sejam, em teoria, compatíveis entre si, elas se diferenciam significativamente em suas *engines* e objetivos específicos, *e.g.*, Docker e Singularity. A *engine*, o cerne operacional de uma abordagem de conteinerização, é responsável pela tarefa de criar imagens de contêiner a partir de um arquivo de descrição fornecido pelo usuário.

Cada imagem de contêiner compreende um conjunto de elementos essenciais para a execução de uma determinada aplicação, abrangendo desde o próprio executável da aplicação até bibliotecas, arquivos e dados. A *engine*, então, consolida e empacota todos esses componentes utilizando técnicas avançadas de virtualização, resultando em uma imagem da pilha de *software* pronta para ser executada em qualquer infraestrutura de destino.

Diversas plataformas de orquestração de contêineres se encontram disponíveis para uso, *e.g.*, *Docker Compose*, *Docker Swarm* e *Kubernetes*. Atualmente, o Kubernetes é a plataforma considerada padrão *de facto* na indústria dada a sua capacidade de integração com serviços de nuvem. Por outro lado, o *Docker Compose* é limitado à execução de múltiplos contêineres em uma única máquina hospedeira, o *Docker Swarm* possibilita a implantação em *clusters*, embora sem integração direta com serviços específicos de nuvem. Diferentemente dos anteriores, o Kubernetes permite a implantação em *clusters*, mas também a

utilização de recursos de nuvem diretamente associados aos contêineres.

O Kubernetes introduz o conceito de *Pod*, que é a menor unidade de implantação que pode ser gerenciada e escalonada no Kubernetes, funcionando como um *host* lógico que encapsula um ou mais contêineres. Desse modo, em um *Pod*, os contêineres compartilham o mesmo contexto de rede e recursos de armazenamento permitindo alocação de disco, e limites de memória e CPU para execução.

3. Abordagem Proposta: AkôFlow

O AkôFlow é um *middleware* desenvolvido para a execução de *workflows* científicos em ambientes conteinerizados. Conforme mencionado anteriormente, ele foi construído sobre o Kubernetes e utiliza contêineres Docker[3]. O AkôFlow interage com a API nativa do Kubernetes para gerenciar os recursos em um *cluster* de contêineres. Dessa forma, ele cria contêineres Docker para executar ativações a partir de uma imagem, diretamente em qualquer infraestrutura que suporte Kubernetes, *e.g.*, um supercomputador ou a nuvem.

3.1. Arquitetura do AkôFlow

A arquitetura do AkôFlow é apresentada na Figura 2 e é composta por cinco camadas: (i) Cliente, (ii) Servidor, (iii) *Proxy*, (iv) *Worker* e (v) Metadados e Proveniência. A camada *Cliente* é implementada como uma ferramenta de linha de comando que permite ao usuário enviar requisições de execução de *workflows* para o *Servidor* AkôFlow. A camada *Cliente* serializa os arquivos de definição dos *workflows* e os envia para o *Servidor* por meio do componente *HTTP Server*. Na versão atual do AkôFlow, a camada *Cliente* foi implementada utilizando apenas componentes nativos da linguagem *Go*[4]. Isso torna o binário que implementa a camada *Cliente* leve e portátil, podendo ser compilado para qualquer SO.

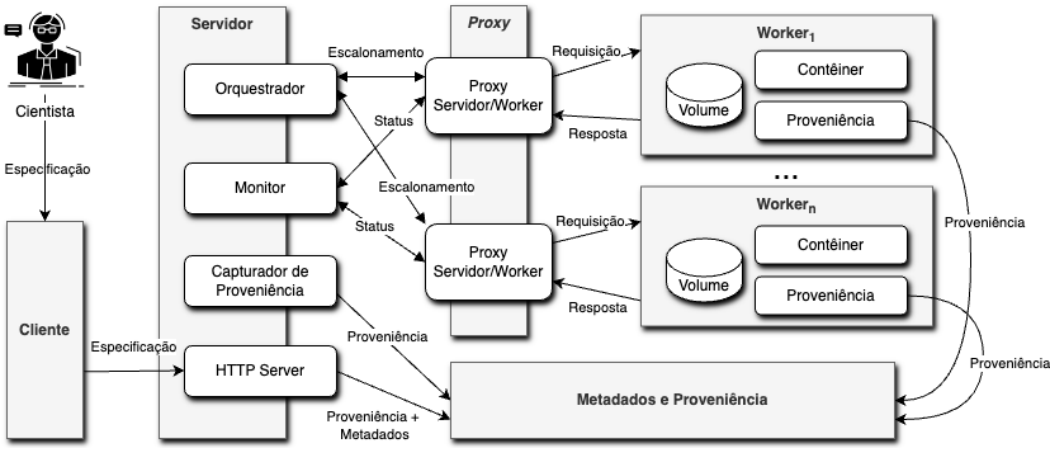

Figura 2. A Arquitetura do AkôFlow

Toda a especificação do *workflow* é feita em um arquivo no formato YAML (Figura 3), que é enviado do *Cliente* do AkôFlow para o *Servidor*. Nesse arquivo, o usuário define o nome do *workflow*, as imagens que serão utilizadas na execução e as ativações que devem ser executadas com suas respectivas dependências de dados. Além disso, o usuário pode limitar a quantidade de recursos para cada ativação, especificando CPU, memória e disco. O

[3] https://www.docker.com/
[4] https://go.dev/

AkôFlow é capaz de executar qualquer imagem previamente construída no padrão do Docker. Na camada do *Servidor*, assim que o *HTTP Server* recebe a especificação do *workflow* do *Cliente*, ele desserializa os *workflows*, cria os registros no banco de metadados e prepara os *workflows* para orquestração pelo *Orquestrador*. As imagens definidas no YAML são instanciadas em contêineres para cada ativação, que, consequentemente, na plataforma Kubernetes, se torna um *Pod* com a respectiva configuração de memória, CPU e disco.

O componente *Orquestrador* recebe então uma lista de ativações e estabelece a ordem de instanciação dos contêineres nos *Workers*. A ordem de instanciação depende da política de escalonamento escolhida pelo usuário. Na versão atual do AkôFlow, o escalonamento é guloso, *i.e.*, assim que um recurso se torna ocioso, uma nova ativação pode ser executada. No entanto, o orquestrador do AkôFlow pode ser estendido para considerar novos algoritmos de escalonamento. Apesar de adotar um escalonamento guloso, o *Orquestrador* pode seguir dois modelos de execução: *First-Data-First* (FDF) e *First-Activity-First* (FAF), conforme definido por [Ogasawara et al. 2011]. No modelo FDF (Figura 4(a)), assim que uma ativação é executada, a ativação que consumirá os dados recém-produzidos já está pronta para execução, seguindo um modelo de *pipeline*. Já no modelo FAF (Figura 4(b)), todas as ativações associadas a uma determinada atividade (*i.e.*, um programa) devem ser finalizadas para que as ativações da próxima atividade se iniciem.

```
name: wf-montage-050d
spec:
  image: ovvesley/akoflow-wf-montage:050d
  namespace: akoflow
  storageClassName: standard-rwo
  storageSize: 2Gi
  mountPath: /data
  activities:
    - name: mprojectid0000001
      run: mProject -X
poss2ukstu_blue_001_001.fits
pposs2ukstu_blue_001_001.fits region-
oversized.hdr
      memoryLimit: 256Mi
      cpuLimit: 500m

    - name: mprojectid0000002
      run: mProject -X
poss2ukstu_blue_001_002.fits
pposs2ukstu_blue_001_002.fits region-
oversized.hdr
      memoryLimit: 256Mi
      cpuLimit: 500m
```

Figura 3. Exemplo de arquivo YAML.

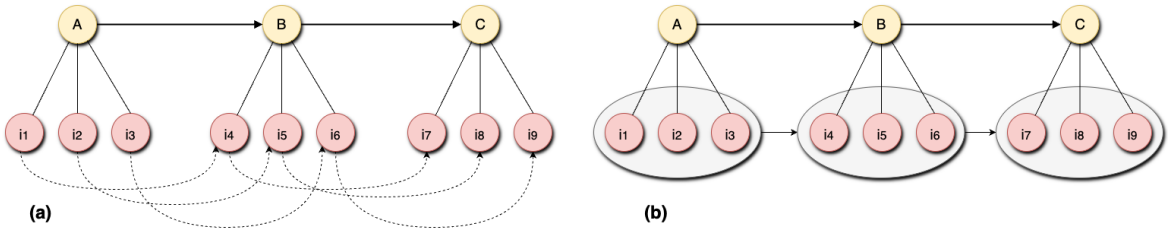

Figura 4. (a) *First-Data-First* (b) *First-Activity-First*

Uma vez que o escalonamento é definido pelo *Orquestrador*, ele inicia o despacho das ativações para execução. As ativações são processadas na camada dos *Workers*. No AkôFlow, é criado um *Proxy* entre o *Servidor* e cada *Worker* para facilitar a troca de mensagens. Na versão atual, cada componente opera em uma *Goroutine*[5] separada, comunicando-se por meio de *Channels*, que são mecanismos de comunicação entre *Goroutines*, usados para enviar e receber mensagens de forma segura e sincronizada em ambientes concorrentes. Em uma mesma execução de um *workflow*, múltiplos *Workers* podem existir. Cada um deles é associado a um ou mais contêineres que executam de fato as ativações. Dessa forma, o *Orquestrador* despacha um contêiner acoplado a um volume para ser executado no *Worker*. No *Worker*, os dados de proveniência também são coletados pelo *Coletor de Proveniência* e armazenados no banco de dados de proveniência (detalhado na SubSeção

[5] *Green threads* executadas em um único processo. Elas são leves e gerenciadas pelo *runtime* da linguagem *Go*, sendo mais eficientes do que as *threads* do sistema operacional.

3.2). A coleta dos dados de proveniência é realizada usando o *Metrics API*, um serviço nativo do Kubernetes que fornece um conjunto básico de métricas de consumo de recursos. É importante ressaltar que é o *Worker* que cria os recursos necessários no Kubernetes, como os discos e *namespaces* de trabalho. O *Worker* é o único componente que acessa a API do Kubernetes para gerenciar tais recursos. Uma vez que um *workflow* está em execução, o componente *Monitor* entra em ação. O *Monitor* é responsável por acompanhar cada ativação, determinando seu estado atual. Ele monitora métricas de CPU, memória e disco das ativações em execução, além de coletar *logs* de erro e registrar eventuais falhas que possam ocorrer. O código-fonte do `AkôFlow` se encontra disponível no GitHub `https://github.com/UFFeScience/akoflow`.

3.2. Modelo de Proveniência

Uma das funcionalidades que o `AkôFlow` oferece é a captura e armazenamento dos dados de proveniência das ativações executadas. A Figura 5 apresenta o modelo de proveniência do `AkôFlow`. O modelo consiste de quatro classes principais: (i) *Workflow*, (ii) *Activity*, (iii) *Metrics* e (iv) *Logs*. A classe *Workflow* contém os metadados de todos os *workflows* executados no `AkôFlow` como o seu *namespace*, o arquivo de definição e o estado atual. Cada *Workflow* possui múltiplas atividades associadas e seu dados são armazenados na classe *Activity*. Cada atividade está associada somente a um *workflow*, possui um *namespace*, um recurso no qual executa, além de seu estado atual.

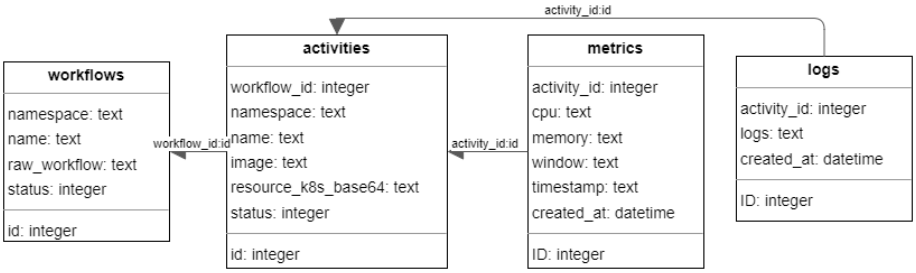

Figura 5. Modelo de Proveniência do `AkôFlow`

Cada atividade pode estar associada a múltiplas leituras de métricas (classe *Metrics*) de uso de recursos durante sua execução. Devido a uma limitação da API de coleta de métricas no Kubernetes, o intervalo disponível para coleta de métricas é de, no mínimo, 15 segundos. Esse intervalo mínimo de coleta se dá pelo uso do *Metrics Server*. Esse intervalo pode ser ajustado de acordo com as configurações do `AkôFlow`. Por padrão, o *AkôFlow* define uma janela de 15 segundos para recuperar os dados de CPU e de memória de cada ativação. Além das métricas de uso de recursos, a base de proveniência também contém os *logs* das atividades, *i.e.*, a saída padrão (*stdout*) das instruções executadas no contêiner. Em versões futuras do `AkôFlow`, planejamos coletar as métricas usando diretamente o *endpoint* `/metrics/resource` disponível no Kubelet, o processo principal do Kubernetes, permitindo assim disponibilizar os dados de proveniência em tempo real em um *dashboard*. Todos os dados de proveniência e metadados são armazenados usando o SQLite.

4. Avaliação Experimental

Nesta seção, avaliamos o `AkôFlow` executando um *workflow* da área de astronomia em um ambiente conteinerizado na nuvem *Google Cloud Platform* (GCP). Primeiramente,

apresentamos o *workflow* escolhido como estudo de caso (Subseção 4.1). Em seguida, detalhamos as configurações do ambiente e do experimento (Subseção 4.2). Finalmente, discutimos os resultados obtidos (Subseção 4.3).

4.1. Estudo de Caso: o *Workflow* Montage

O Montage [Sakellariou et al. 2009] (Figura 6) é um *workflow* que cria um mosaico a partir de múltiplas imagens do céu e é intensivo produção de dados. Ele é composto por sete atividades, cada uma associada a um programa do Montage *toolkit*[6]: (i) *mProject* (em amarelo) - projeta as imagens em uma determinada escala, (ii) *mDiffFit* (em azul) - calcula e ajusta as diferenças entre duas imagens astronômicas, (iii) *mConcatFit* (em vermelho) - combina múltiplas imagens em uma, (iv) *mBgModel* (em laranja) - modela e corrige diferenças no fundo em imagens com sobreposição, (v) *mBackground* (em verde) - remove o fundo das imagens, (vi) *mImgtbl* (em cinza claro) - extrai metadados para geração do mosaico, e (vii) *mAdd* (em cinza escuro) - cria o mosaico com a composição das imagens.

O Montage pode ser configurado para cobrir diferentes áreas do céu. A área de cobertura é definida em graus quadrados com imagens em três canais de cores (RGB). Dependendo da área de cobertura definida pelo usuário, mais ou menos ativações serão definidas no *workflow*. Nos experimentos apresentados nessa seção, executamos o Montage cobrindo 0,5 e 1,0 grau quadrado do céu. A quantidade de ativações por programa do *workflow* para cada uma das configurações de área coberta é apresentada na Tabela 1.

4.2. Configuração do Ambiente de Execução e do Experimento

Utilizamos o *Google Cloud Platform* (GCP) para criar máquinas virtuais nas quais os contêineres são instanciados durante a execução do *workflow*. A máquina virtual utilizada nos experimentos foi a `e2-highcpu-16`, que possui 16 vCPUs AMD Rome 2.25 GHz, 16GB RAM e 16 Gbps de taxa de transferência. Nessa máquina virtual foram instanciados os contêineres para execução do Montage.

Como o `AkôFlow` permite limitar a quantidade de recursos por contêiner que executa a ativação, a partir de experimentos preliminares definimos quatro diferentes cenários de distribuição de recursos: (i) C_1 - 0,5 vCPU para cada ativação, (ii) C_2 - 1,0 vCPU para cada ativação, (iii) C_3 - 2,0 vCPU para as ativações do *mProject* (computacionalmente intensiva) e 0,5 vCPU para as demais ativações e (iv) C_4 - 4,0 vCPU para as ativações do *mProject* e 0,5 vCPU para as demais ativações. Todas as execuções foram realizadas seguindo o modelo de execução *First-Data-First* (FDF), conforme discutido na Subseção 3.1.

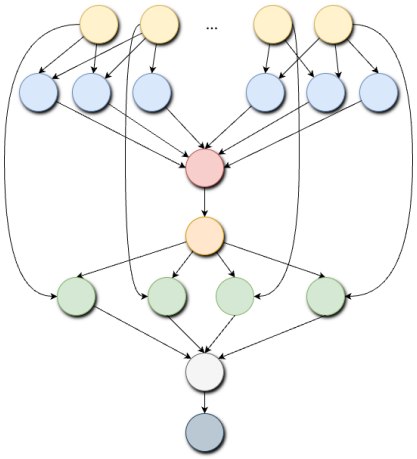

Figura 6. O *Workflow* Montage.

4.3. Discussão dos Resultados

A Figura 7 apresenta os tempos de execução, em segundos, para cada uma das configurações do Montage e para cada um dos cenários de alocação de recursos definidos (média de 5 execuções). Observa-se que não há vantagem aparente na alocação de

[6] `http://montage.ipac.caltech.edu/`

Tabela 1. Total de ativações por programa em cada configuração do Montage.

	Área de Cobertura do Céu	
Atividade	**0,5 grau**	**1,0 grau**
mProject	12	48
mDiffFit	18	360
mConcatFit	3	3
mBgmodel	3	3
mBackground	12	48
mImgtbl	3	3
mAdd	3	3
mViewer	4	4
Total	58	472

muitos recursos, especialmente mais de 1 vCPU por ativação no caso do *workflow* Montage. As aplicações que fazem parte do Montage *toolkit* e que são invocadas no *workflow* foram desenvolvidas no início dos anos 2000 e nenhuma delas foi projetada para executar em mais de um processador. Assim, percebe-se uma perda de desempenho nos cenários C_3 e C_4 para coberturas de 0,5 e 1,0 grau quadrado, uma vez que várias vCPUs permanecem desocupadas durante a execução. Por exemplo, no cenário C_4 são alocadas 4 vCPUs por ativação do *mProject*, mas somente uma é utilizada, resultando em três vCPUs ociosas para cada ativação do *mProject* executada. Em todas as configurações do Montage, o melhor desempenho foi alcançado no cenário C_2, onde foi alocada 1 vCPU para cada ativação do *workflow*. As Figuras 8, 9, 10 e 11 apresentam os planos de escalonamento do Montage para cada espaço de cobertura do céu e para cada cenário de alocação de recursos. É possível observar claramente o modelo de execução FDF em ação, onde ativações associadas a dois ou mais programas no fluxo são executadas concorrentemente.

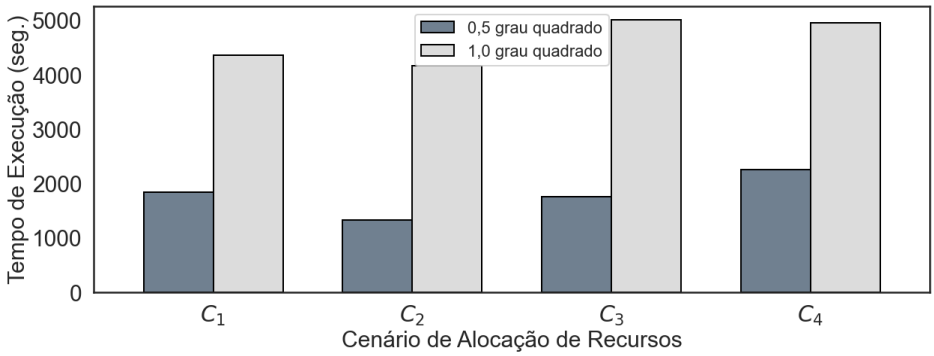

Figura 7. Tempo de execução em segundos do Montage usando o `AkôFlow`.

Como a execução dos experimentos foi realizada na GCP, há um custo financeiro associado tanto ao processamento quanto ao armazenamento de dados. A Tabela 2 apresenta os custos em R$ de processamento e armazenamento para cada uma das configurações do Montage e cenários de alocação de recursos. Pode-se perceber que o custo de processamento é aceitável em todos os casos, não excedendo R$ 2,89 no pior caso. Entretanto, o mesmo não se pode dizer do custo de armazenamento. O Montage, conforme discutido anteriormente, é um *workflow* intensivo na produção de dados. Dependendo do espaço de cobertura do céu, as imagens processadas aumentam consideravelmente de tamanho, devido

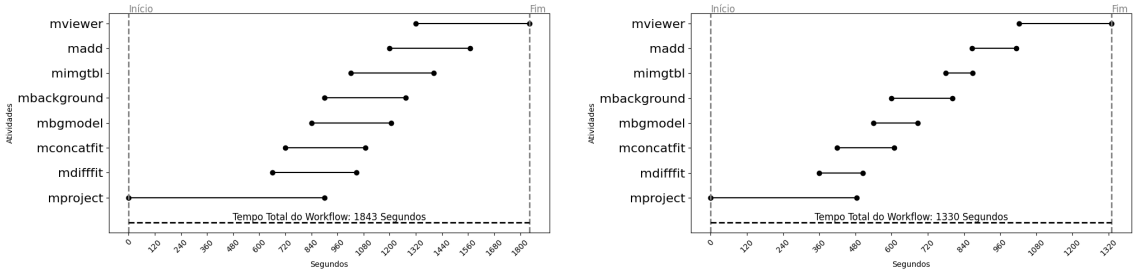

Figura 8. Escalonamentos realizados pelo `AkôFlow` para o Montage cobrindo 0,5 grau quadrado do céu e para os cenários C_1 (a) e C_2 (b).

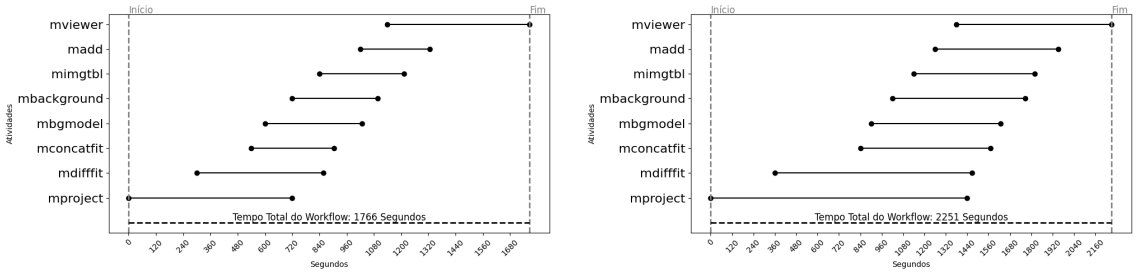

Figura 9. Escalonamentos realizados pelo `AkôFlow` para o Montage cobrindo 0,5 grau quadrado do céu e para os cenários C_3 (a) e C_4 (b).

à definição (em *pixels*) necessária para realizar a tarefa. Dessa forma, nos casos em que o Montage cobre 1,0 grau quadrado do céu, no pior cenário, o custo de processamento (R$ 2,55) representa apenas 0,84% do custo de armazenamento (R$ 302,08). Em *workflows* com diferentes comportamentos, ou seja, mais processamento e menor uso de disco, o custo total pode apresentar um comportamento bastante diferente. Inclusive, este é um dos trabalhos futuros: avaliar o `AkôFlow` com *workflows* intensivos em computação e não somente intensivos em produção de dados.

Tabela 2. Custo financeiro (em R$) para as diferentes configurações do Montage e cada cenário de alocação de recursos.

	Custo Financeiro (R$)	
Configuração/Cenário	**Processamento**	**Armazenamento**
0,5 grau - C_1	1,07	12,18
0,5 grau - C_2	0,77	12,18
0,5 grau - C_3	1,03	12,18
0,5 grau - C_4	1,31	12,18
1,0 grau - C_1	2,55	302,08
1,0 grau - C_2	2,43	302,08
1,0 grau - C_3	2,93	302,08
1,0 grau - C_4	2,89	302,08

5. Trabalhos Relacionados

Há uma abundância de trabalhos na literatura que propõem sistemas e *middlewares* para a execução de *workflows* científicos, como o Pegasus [Deelman et al. 2021] e o Parsl [Babuji et al. 2019]. Embora esses sistemas representem um avanço significativo, eles não

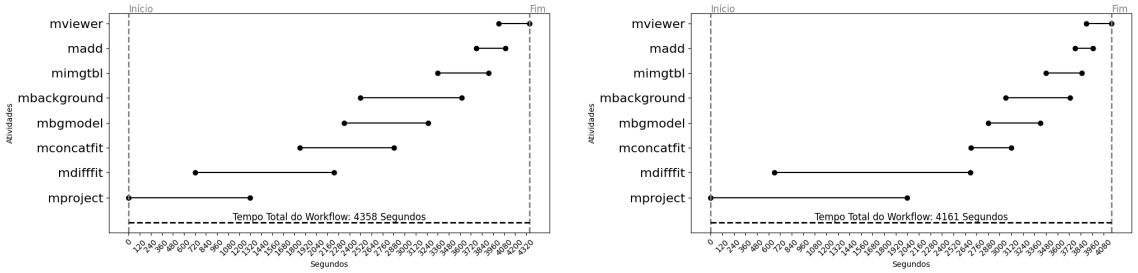

Figura 10. Escalonamentos realizados pelo `AkôFlow` para o Montage cobrindo 1,0 grau quadrado do céu e para os cenários C_1 (a) e C_2 (b).

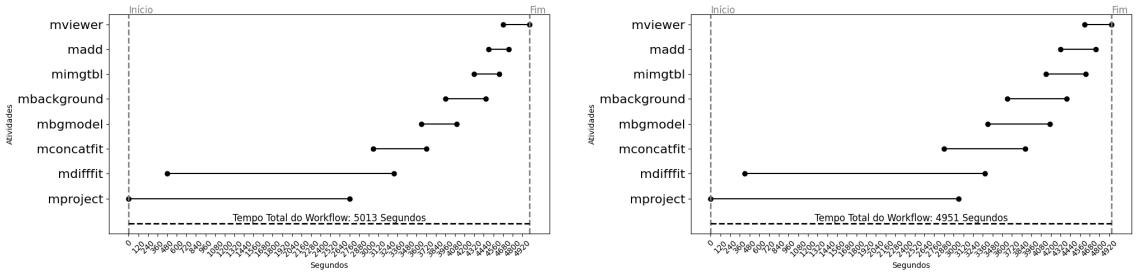

Figura 11. Escalonamentos realizados pelo `AkôFlow` para o Montage cobrindo 1,0 grau quadrado do céu e para os cenários C_3 (a) e C_4 (b).

são focados em ambientes conteinerizados. O Pegasus, por exemplo, até oferece suporte à execução de *workflows* com contêineres, mas de forma limitada, exigindo que o usuário encapsule todos os *softwares* e dependências em um único contêiner. Outras soluções são focadas na execução de *workflows* em ambientes conteinerizados, como em [Burkat et al. 2021], onde os autores avaliam a execução de *workflows* em ambientes conteinerizados, em especial acoplando o sistema de *workflow* Pegasus com serviços como o AWS Fargate, que permite a criação de contêineres elásticos. De fato, os contêineres elásticos permitem que a capacidade de processamento do contêiner seja ajustada em tempo de execução, o que é vantajoso. No entanto, a abordagem proposta por [Burkat et al. 2021] ainda enfrenta a limitação do Pegasus, que exige que todos os *softwares* necessários estejam encapsulados em um único contêiner.

Outro exemplo de abordagem com foco na execução de *workflows* em ambientes conteinerizados é o DEWE [Jiang et al. 2017]. O DEWE é uma *engine* de execução de *workflows* científicos em contêineres *serverless*, utilizando ambientes como o *AWS Lambda* e o *Google Cloud Functions*. Entretanto, o DEWE não foi projetado para executar *workflows* em ambientes conteinerizados que não seguem o paradigma *serverless*, o que constitui uma limitação significativa, principalmente por conta do custo associado a tais ambientes. Similarmente, [Zheng et al. 2017] apresentam um estudo que integra o sistema de *workflow* Makeflow com o escalonador Mesos, visando à execução eficiente de *workflows*. Apesar de representar um avanço, o acoplamento proposto pelos autores considera uma granularidade grossa do *workflow*, ou seja, um único contêiner é criado para múltiplas ativações do *workflow*, o que reduz a flexibilidade no escalonamento. Plataformas comerciais como

ARGO[7] e Kubeflow[8] permitem aos usuários desenvolver *workflows* em ambientes conteinerizados, em especial os *workflows* associados ao treinamento de modelos de Aprendizado de Máquina. Entretanto, essas plataformas não foram projetadas especificamente para *workflows* científicos, e não permitem extensões nos algoritmos de escalonamento, além de não capturarem dados de proveniência.

6. Conclusões

Workflows científicos de larga escala requerem o uso de técnicas de paralelismo combinadas com ambientes HPC para gerar resultados em tempo hábil. No entanto, esses *workflows* são normalmente projetados e modelados com foco em uma infraestrutura computacional específica, *e.g.*, *clusters* ou nuvens, o que torna a portabilidade entre infraestruturas heterogêneas uma tarefa complexa. Uma maneira de mitigar o impacto da configuração da infraestrutura e facilitar a portabilidade dos *workflows* entre diferentes ambientes é por meio do uso de conteinerização. Entretanto, os sistemas de *workflow* e as plataformas de orquestração de contêineres ainda são limitados no suporte à execução de *workflows* científicos em ambientes conteinerizados. Para abordar a execução paralela de *workflows* científicos em ambientes conteinerizados, propusemos o `AkôFlow`, um *middleware* leve para apoiar a execução de *workflows* em múltiplos contêineres. O `AkôFlow` foi avaliado por meio da execução do *workflow* Montage na GCP, e os resultados mostraram-se promissores. Embora esses sejam resultados para apenas um *workflow*, acreditamos que eles demonstram a viabilidade do `AkôFlow`. Experimentos com novos *workflows* precisam ser realizados para avaliar plenamente a abordagem e os dois modelos de execução discutidos neste artigo.

Referências

Babuji, Y. N. et al. (2019). Parsl: Pervasive parallel programming in python. In Weissman, J. B., Butt, A. R., and Smirni, E., editors, *HPDC'19*, pages 25–36. ACM.

Burkat, K., Pawlik, M., Balis, B., Malawski, M., Vahi, K., Rynge, M., da Silva, R. F., and Deelman, E. (2021). Serverless containers – rising viable approach to scientific workflows. In *eScience*, pages 40–49.

Carrión, C. (2023). Kubernetes scheduling: Taxonomy, ongoing issues and challenges. *ACM Comput. Surv.*, 55(7):138:1–138:37.

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

de Oliveira, D., Ogasawara, E. S., Baião, F. A., and Mattoso, M. (2010). Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. In *CLOUD'10*, pages 378–385.

de Oliveira, D., Silva, V., and Mattoso, M. (2015). How much domain data should be in provenance databases? In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*.

de Oliveira, D. C. M., Liu, J., and Pacitti, E. (2019). *Data-Intensive Workflow Management: For Clouds and Data-Intensive and Scalable Computing Environments*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

[7]`https://argoproj.github.io/workflows/`

[8]`https://www.kubeflow.org/`

Deelman, E., da Silva, R. F., Vahi, K., Rynge, M., Mayani, R., Tanaka, R., Whitcup, W. R., and Livny, M. (2021). The pegasus workflow management system: Translational computer science in practice. *J. Comput. Sci.*, 52:101200.

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., Martins, L. B., Falci, M. L. F., Silva, V., Ocaña, K. A., Mattoso, M., Bedo, M., and de Oliveira, D. (2020). Capturing and analyzing provenance from spark-based scientific workflows with samba-rap. *Future Generation Computer Systems*, 112:658 – 669.

Jiang, Q., Lee, Y. C., and Zomaya, A. Y. (2017). Serverless execution of scientific workflows. In *ICSOC 2017*, pages 706–721. Springer.

Kunstmann, L., Pina, D., Oliveira, L., Oliveira, D., and Mattoso, M. (2022). Provdeploy: Explorando alternativas de conteinerização com proveniência para aplicações científicas com pad. In *Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 49–60, Porto Alegre, RS, Brasil. SBC.

Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459.

Ogasawara, E. S., de Oliveira, D., Valduriez, P., Dias, J., Porto, F., and Mattoso, M. (2011). An algebraic approach for data-centric scientific workflows. *Proc. VLDB Endow.*, 4(12):1328–1339.

Ogasawara, E. S., Dias, J., Silva, V., Chirigati, F. S., de Oliveira, D., Porto, F., Valduriez, P., and Mattoso, M. (2013). Chiron: a parallel engine for algebraic scientific workflows. *Concurr. Comput. Pract. Exp.*, 25(16):2327–2341.

Sakellariou, R. et al. (2009). Mapping workflows on grid resources: Experiments with the montage workflow. In *ERCIM W. Group on Grids*, pages 119–132.

Shah, S. T., Lahaye, R. J. W. E., Kazmi, S. A. A., Chung, M. Y., and Hasan, S. F. (2014). Htcondor system for running extensive simulations related to D2D communication. In *ICTC*, pages 283–284. IEEE.

Silva, V., de Oliveira, D., Valduriez, P., and Mattoso, M. (2018). Dfanalyzer: runtime dataflow analysis of scientific applications using provenance. *Proceedings of the VLDB Endowment*, 11(12):2082–2085.

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.

Teylo, L., de Paula Junior, U., et al. (2017). A hybrid evolutionary algorithm for task scheduling and data assignment of data-intensive scientific workflows on clouds. *FGCS*, 76:1–17.

Zheng, C., Tovar, B., and Thain, D. (2017). Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In *CCGrid*, CCGrid '17, page 130–139. IEEE Press.