

UNIVERSIDADE ESTADUAL PAULISTA - UNESP

Instituto de Biociências, Letras e Ciências Exatas (IBILCE)- campus de São José do Rio Preto

JOÃO PEDRO RESENDE BARROSO

Hot Migration For Distributed Deep Learning Tasks in Kubernetes

São José do Rio Preto

2026

João Pedro Resende Barroso

Hot Migration For Distributed Deep Learning Tasks in Kubernetes

Dissertação, apresentada à Universidade Estadual Paulista (UNESP), Instituto de Biociências, Letras e Ciências Exatas (IBILCE), São José do Rio Preto, para obtenção do título de Mestre em Ciência da Computação.

Área de Concentração: Computação Aplicada

Orientador: Prof. Dr. Aleardo Manacero Jr.

São José do Rio Preto

2026

IMPACTO POTENCIAL DESTA PESQUISA

Esta pesquisa propõe um escalonador com migração ao vivo de contêineres para o treinamento distribuído de aprendizado profundo em Kubernetes, guiado pela taxa de crescimento das tarefas e pelo balanceamento de carga entre nós. Ao preservar o estado da computação, inclusive em GPUs, e acoplar a detecção especulativa de convergência à decisão de escalonamento, o trabalho avança o estado da arte e entrega um projeto construído sobre padrões abertos. Reunindo num único controlador conceitos eficientes de alocação e balanceamento de carga, ele aumenta a utilização efetiva dos clusters e reduz a ociosidade de recursos caros, o que diminui o custo e o consumo energético do treinamento de modelos. Com isso, viabiliza que grupos e instituições com orçamento limitado extraiam mais desempenho do mesmo hardware, ao mesmo tempo em que serve de material formativo em sistemas distribuídos e contêineres. A adesão a padrões abertos fortalece a competência nacional em computação de alto desempenho e nuvem e favorece a comparabilidade e o diálogo com a literatura internacional, ampliando o alcance do conhecimento gerado.

POTENTIAL IMPACT OF THIS RESEARCH

This research proposes a scheduler with live container migration for distributed deep learning training on Kubernetes, guided by the tasks' growth rate and by load balancing across nodes. By preserving computation state, including on GPUs, and coupling speculative convergence detection to the scheduling decision, the work advances the state of the art and delivers a design built on open standards. Bringing together, in a single controller, efficient concepts of allocation and load balancing, it raises the effective utilization of clusters and reduces the idle time of costly resources, thereby lowering the cost and energy consumption of model training. This enables budget-constrained groups and institutions to extract more performance from the same hardware, while also serving as teaching material on distributed systems and containers. Adherence to open standards strengthens national expertise in high-performance and cloud computing and fosters comparability and dialogue with the international literature, broadening the reach of the knowledge produced.

Dedico este trabalho ao meu sobrinho Miguel,
que continue sempre sendo a felicidade da família...

AGRADECIMENTOS

Agradecimentos primeiramente a meus pais, que fornecem apoio a todas fases da minha vida. Agradecimentos também ao professor Aleardo, meu orientador que está a tanto tempo me aturando, e a professora Renata, que sempre traz conselhos importantes aos meus projetos.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 88887.949404/2024-00.

“If only evil is remembered, you will never be happy in the sight of God.”
(Andrei Rublev, 1966)

RESUMO

O treinamento de redes neurais profundas consome recursos computacionais caros, como unidades de processamento gráfico (GPUs), sendo frequentemente necessário utilizar um sistema distribuído para a realização de tarefas de aprendizado em larga escala. Em sistemas baseados em contêineres, que utilizam *middlewares* como o *Kubernetes*, empregam-se, por padrão, escalonadores agnósticos, que não consideram fatores internos da aplicação, resultando na subutilização dos recursos do sistema. Este trabalho apresenta um escalonador especializado para tarefas de treinamento distribuído de modelos de aprendizado profundo em sistemas baseados em contêineres, empregando a migração ao vivo de contêineres como mecanismo central de balanceamento de carga. A estratégia proposta utiliza métricas relacionadas ao processo de treinamento, como acurácia e taxa de erro, para orientar o balanceamento das cargas de trabalho do sistema distribuído, migrando tarefas com menor taxa de evolução da acurácia para nós mais carregados, preservando nos nós com maior disponibilidade de recursos aquelas cuja acurácia apresenta maior taxa de evolução, reduzindo o tempo de conclusão do conjunto de tarefas. Para isso, emprega-se o *CRIU (Checkpoint/Restore in Userspace)*, permitindo a migração ao vivo de contêineres entre nós com preservação do estado de execução, sem perda de progresso e com *downtime* praticamente nulo. Os resultados experimentais mostram que a solução melhora a utilização dos recursos do sistema distribuído, evita as interrupções associadas à migração a frio e reduz o tempo de conclusão de grupos de tarefas executados simultaneamente em até 46% em comparação ao escalonador padrão do *Kubernetes*.

Palavras-Chave: aprendizado profundo; escalonamento de tarefas; migração de contêineres; computação em nuvem; sistemas distribuídos; *Kubernetes*.

ABSTRACT

Deep neural network training requires expensive computational resources, such as Graphics Processing Units (GPUs), making it frequently necessary to employ distributed systems for large-scale learning tasks. In container-based environments orchestrated by middlewares such as Kubernetes, the default schedulers are application-agnostic and do not consider application-specific characteristics, leading to underutilization of system resources. This work presents a specialized scheduler for distributed deep learning training in container-based environments, employing live container migration as the core load-balancing mechanism. The proposed strategy leverages training-related metrics, such as accuracy and error rate, to guide workload balancing across the distributed system by migrating tasks with lower accuracy improvement rates to more heavily loaded nodes while keeping tasks with higher accuracy improvement rates on nodes with greater resource availability, thereby reducing the overall completion time of the workload. To achieve this, the solution employs CRIU (*Checkpoint/Restore in Userspace*), enabling live migration of containers across nodes while preserving the execution state, without loss of progress and with nearly zero downtime. Experimental results show that the proposed solution improves resource utilization in the distributed system, avoids the interruptions associated with cold migration, and reduces the completion time of concurrently executed task groups by up to 46% compared with Kubernetes' default scheduler.

Keywords: deep learning; task scheduling; container migration; cloud computing; distributed systems; Kubernetes.

LISTA DE ILUSTRAÇÕES

Figura 1	Treinamento distribuído com paralelismo de dados e AllReduce.	24
Figura 2	Arquitetura do Kubernetes: plano de controle e nós de trabalho.	27
Figura 3	Fases da migração ao vivo de contêineres com CRIU.	31
Figura 4	Visão geral da arquitetura do <i>dlkubesched</i>	46
Figura 5	Utilização de CPU por nó ao longo da duração dos testes realizados no setup 1, utilizando migração ao vivo.	69
Figura 6	Utilização de CPU por nó ao longo da duração dos testes realizados no setup 2, utilizando migração ao vivo.	70
Figura 7	Utilização de CPU por nó ao longo da duração dos testes realizados no setup 3, utilizando migração ao vivo.	71
Figura 8	Utilização de CPU por nó ao longo da duração dos testes realizados no setup 4, utilizando migração ao vivo.	71
Figura 9	Utilização de CPU por nó ao longo da duração dos testes realizados no setup 5, utilizando migração ao vivo.	72

LISTA DE TABELAS

Tabela 1 – Comparativo de <i>middleware</i> entre paradigmas de sistemas distribuídos. . . .	21
Tabela 2 – Componentes do <i>dlkubesched</i> e suas responsabilidades.	46
Tabela 3 – Comparação entre os modos de migração.	55
Tabela 4 – Especificação dos nós do <i>cluster</i> experimental.	62
Tabela 5 – Componentes de <i>software</i> do ambiente experimental.	62
Tabela 6 – Especificação do <i>Setup</i> 1.	63
Tabela 7 – Especificação do <i>Setup</i> 2.	63
Tabela 8 – Especificação do <i>Setup</i> 3.	64
Tabela 9 – Especificação do <i>Setup</i> 4.	64
Tabela 10 – Especificação do <i>Setup</i> 5.	65
Tabela 11 – <i>Makespan</i> (média $\pm$ desvio-padrão, em segundos) e <i>speedup</i> por cenário. . .	67
Tabela 12 – Custo por migração por tarefa: a frio <i>versus</i> ao vivo (Experimento 4). . . .	68
Tabela 13 – Síntese de hipóteses, experimentos e resultados.	69

SUMÁRIO

1	INTRODUÇÃO	14
1.1	Relevância do problema	14
1.2	Objetivo	15
1.3	Organização do texto	15
2	INTRODUÇÃO TEÓRICA	17
2.1	Sistemas Distribuídos	17
2.1.1	Propriedades fundamentais	17
2.1.2	Evolução das plataformas de sistemas distribuídos	18
2.1.2.1	Clusters clássicos	18
2.1.2.2	Sistemas baseados em máquinas virtuais	18
2.1.2.3	Sistemas baseados em contêineres	19
2.2	Middleware	19
2.2.1	Middleware em clusters clássicos	19
2.2.2	Middleware em sistemas baseados em máquinas virtuais	20
2.2.3	Middleware em sistemas baseados em contêineres	20
2.3	Aprendizagem Profunda Distribuída	21
2.3.1	Redes neurais e o processo de treinamento	22
2.3.2	Demanda computacional e aceleradores de hardware	22
2.3.3	Estratégias de treinamento distribuído	23
2.3.3.1	Paralelismo de dados	23
2.3.3.2	Paralelismo de modelo	23
2.3.3.3	Paralelismo de pipeline	24
2.3.4	Principais arcabouços para treinamento distribuído	24
2.4	Kubernetes	25
2.4.1	Arquitetura do Kubernetes	25
2.4.1.1	Plano de controle	25
2.4.1.2	Plano de dados	26
2.4.2	Objetos fundamentais do Kubernetes	26
2.4.2.1	Pod	27
2.4.2.2	Job e IndexedJob	27
2.4.2.3	Modelo de recursos	27
2.4.3	O escalonador padrão do Kubernetes	28
2.4.4	Preempção no Kubernetes	28
2.5	Migração de contêineres	29

2.5.1	CRIU: Checkpoint/Restore In Userspace	30
2.5.2	Migração ao vivo de contêineres	30
2.5.3	Desafios na migração de contêineres para aprendizado profundo	31
2.6	Considerações finais	32
3	TRABALHOS RELACIONADOS	34
3.1	Migração ao vivo de Contêineres	34
3.1.1	CloudHopper	35
3.1.2	Sledge	35
3.1.3	MBDPC	36
3.1.4	Migração Segura e Leve	37
3.1.5	FastMig	37
3.1.6	CRIUgpu	38
3.1.7	KubeSPT	39
3.2	Escalonadores Especializados para Aprendizado de Máquina	40
3.2.1	Gandiva	40
3.2.2	Tiresias	40
3.2.3	Themis	41
3.2.4	SpeCon	41
3.2.5	SchedTune	42
3.2.6	Viody e Kistijantoro	42
3.3	Considerações Finais	43
4	MIGRAÇÃO AO VIVO PARA TAREFAS DE APRENDIZADO PROFUNDO	45
4.1	Visão Geral da Arquitetura	45
4.2	Recurso Personalizado <i>DeepLearningJob</i>	46
4.2.1	Estado e Métricas de Migração	48
4.3	Controlador	48
4.3.1	Reconciliação e Criação de Pods	49
4.3.2	Gerenciamento do Ciclo de Vida	49
4.3.3	Registro de Métricas de Migração	50
4.4	Escalonador	50
4.4.1	Filtragem de Nós	50
4.4.2	<i>Gang Scheduling</i> e Empacotamento	51
4.4.3	Vinculação Direcionada por Migração	52
4.5	Monitor de Crescimento	52
4.6	Migrador	54
4.6.1	Migração a Frio	54
4.6.2	migração ao vivo (<i>hot</i>)	54

4.7	Balanceamento de Carga por Migração	55
4.7.1	Definição de Nó Mais Utilizado	56
4.7.2	Rebalanceamento para o Nó Menos Utilizado	57
4.7.3	Interação entre Empacotamento e Balanceamento	58
4.8	Agente CRIU	58
4.9	Fluxo de Execução	59
4.10	Considerações Finais	60
5	VALIDAÇÃO DE RESULTADOS	61
5.1	Perguntas de Pesquisa e Hipóteses	61
5.2	Ambiente Experimental	62
5.3	Carga de Trabalho	62
5.4	Cenários de Implantação (<i>Setups</i>)	62
5.4.1	Setup 1 - Contenção máxima em nó único.	63
5.4.2	Setup 2 - Treinamento distribuído concentrado.	63
5.4.3	Setup 3 - Desbalanceamento moderado.	64
5.4.4	Setup 4 - Colocação equilibrada.	64
5.4.5	Setup 5 - Colocação equilibrada distribuída.	64
5.5	Métricas de Avaliação	65
5.6	Protocolo Experimental	66
5.7	Análise de <i>makespan</i>	66
5.7.1	Objetivo.	66
5.7.2	Configuração.	66
5.7.3	Resultados.	66
5.7.4	Análise.	66
5.8	Decomposição do <i>Downtime</i>	67
5.8.1	Objetivo.	67
5.8.2	Configuração.	67
5.8.3	Resultados.	67
5.8.4	Análise.	67
5.9	Experimento 3 - Utilização de Recursos	68
5.9.1	Objetivo.	68
5.9.2	Configuração.	68
5.9.3	Resultados.	68
5.9.4	Análise.	68
5.10	Discussão Geral	69
5.11	Considerações Finais	70
6	CONCLUSÃO	73
6.1	Conclusões Essenciais	73

6.2	Limitações	74
6.3	Trabalhos Futuros	74
6.4	Publicações	75
	REFERÊNCIAS	76

1 INTRODUÇÃO

O avanço do aprendizado profundo nas últimas décadas transformou de maneira profunda diversas áreas do conhecimento, desde o reconhecimento de imagens e o processamento de linguagem natural até a descoberta de fármacos e a condução autônoma de veículos (LeCun; Bengio; Hinton, 2015). Esse progresso, entretanto, é indissociável do crescimento exponencial dos modelos utilizados: arquiteturas contemporâneas chegam a centenas de bilhões de parâmetros e exigem semanas de processamento em milhares de unidades de processamento gráfico (GPUs) para convergirem (Brown et al., 2020). Diante dessa demanda, o treinamento distribuído deixou de ser uma otimização opcional e tornou-se uma necessidade para qualquer tarefa de aprendizado profundo de maior escala.

Nesse contexto, sistemas distribuídos baseados em contêineres emergiram como a infraestrutura predominante para a execução de cargas de trabalho de inteligência artificial em escala de produção (Lawson; Sica, 2026). O Kubernetes, orquestrador de contêineres de código aberto originalmente desenvolvido pela Google, tornou-se o padrão da indústria para o gerenciamento dessas cargas em ambientes de nuvem pública e privada (Mell; Grance, 2011). Sua arquitetura extensível permite que componentes personalizados, como escalonadores alternativos, sejam integrados ao sistema sem necessidade de modificações em sua estrutura central.

1.1 RELEVÂNCIA DO PROBLEMA

O escalonador padrão do Kubernetes foi projetado para cargas de trabalho genéricas de microsserviços, nos quais as tarefas são curtas, independentes entre si e tolerantes a reinicializações. Tarefas de aprendizado profundo distribuído apresentam características fundamentalmente distintas: são longas, exigem que todos os seus processos trabalhadores sejam iniciados simultaneamente (*gang scheduling*), dependem de recursos escassos, como GPUs, e acumulam estado de treinamento ao longo de horas ou dias de execução (Ben-Nun; Hoeftler, 2019).

Essa incompatibilidade entre o modelo de escalonamento genérico e as necessidades reais das cargas de aprendizado profundo gera dois problemas centrais. Primeiro, tarefas de alta prioridade submetidas a um *cluster* ocupado podem permanecer na fila por longos períodos, aguardando que trabalhos de menor importância liberem os recursos de GPU. Como o escalonador padrão não implementa técnicas preempção com o intuito de equilibrar a carga de trabalho de cada máquina, tarefas podem ficar inativas por um período elevado de tempo.

Além disso, mesmo em escalonadores de pesquisa que implementam alguma forma de preempção, a interrupção de uma tarefa de aprendizado profundo implica o descarte de todo o progresso de treinamento acumulado, pois o estado do modelo em memória é perdido quando o contêiner é encerrado. Mesmo com a presença de checkpoints incrementais durante o treinamento, o custo de reinicializações torna a preempção proibitivamente cara na prática, levando operadores

a evitá-la mesmo quando seria a decisão correta do ponto de vista do desempenho global do *cluster*.

A migração ao vivo (*hot migration*) de contêineres, viabilizada pelo mecanismo de *checkpoint/restore* de processos, oferece uma solução para esse segundo problema: é possível capturar o estado completo de um processo em execução, incluindo o estado do otimizador e os pesos da rede neural em memória, e restaurá-lo em outro nó do *cluster*, sem que o contêiner precise ser recomeçado do zero. Combinada com uma política de escalonamento baseada em preempção, essa capacidade abre caminho para um escalonador que equilibre o uso de recursos entre as diferentes máquinas de forma transparente e econômica, realocando seus recursos sem desperdiçar o progresso computacional já realizado.

Apesar do potencial desta combinação, a literatura carece de propostas que integrem preempção baseada em prioridades e migração ao vivo de contêineres em um escalonador Kubernetes voltado especificamente para cargas de aprendizado profundo distribuído.

1.2 OBJETIVO

O objetivo principal deste trabalho é projetar e avaliar um escalonador para o Kubernetes voltado a tarefas de aprendizado profundo distribuído, que utilize preempção baseada no balanceamento de carga e migração ao vivo de contêineres para reduzir o tempo de conclusão de um conjunto de tarefas e aumentar a utilização dos recursos do *cluster*.

Para atingir esse objetivo principal, os seguintes objetivos específicos foram estabelecidos:

- a) caracterizar os requisitos de escalonamento de tarefas de aprendizado profundo distribuído e identificar as limitações do escalonador padrão do Kubernetes frente a esses requisitos;
- b) propor uma política de escalonamento baseado na utilização de recursos de tarefas de aprendizado profundo, utilizando métricas como convergência para equilibrar as cargas de trabalho do sistema;
- c) integrar o mecanismo de migração ao vivo baseado em CRIU ao ciclo de vida de preempção do escalonador, preservando o estado de treinamento das tarefas interrompidas; e
- d) avaliar o escalonador proposto por meio de experimentos controlados, comparando métricas de *makespan*, tempo de espera e utilização de GPU com as do escalonador padrão do Kubernetes e com soluções da literatura.

1.3 ORGANIZAÇÃO DO TEXTO

Este texto está organizado em seis capítulos, além desta introdução. O Capítulo 2 apresenta a fundamentação teórica necessária para a compreensão do trabalho, abordando sistemas distribuídos baseados em contêineres, os fundamentos do aprendizado profundo distribuído, a arquitetura

do Kubernetes e os conceitos de preempção e migração de contêineres. O Capítulo 3 apresenta uma revisão dos trabalhos relacionados, discutindo escalonadores para tarefas de aprendizado profundo e abordagens de migração ao vivo em ambientes de contêineres. O Capítulo 4 apresenta a implementação do escalonador, mostrando as características e qualidades. O Capítulo 5 apresenta os resultados experimentais e a análise comparativa de desempenho. Por fim, o Capítulo 6 apresenta as conclusões do trabalho e as direções para trabalhos futuros.

2 INTRODUÇÃO TEÓRICA

Conforme abordado no Capítulo 1, este projeto tem como objetivo aprimorar o desempenho de tarefas de aprendizado profundo em sistemas distribuídos baseados em contêineres. Portanto, para um melhor entendimento dos desafios encontrados, é necessário conhecer esse tipo de sistema distribuído em questão e entender suas particularidades.

2.1 SISTEMAS DISTRIBUÍDOS

Um sistema distribuído é definido por Tanenbaum e Van Steen (Tanenbaum; Steen, 2007) como um conjunto de computadores independentes que se apresenta aos seus usuários como um sistema único e coerente. Essa definição encapsula duas propriedades essenciais e aparentemente contraditórias: do ponto de vista interno, o sistema é formado por múltiplos nós autônomos, cada um com sua própria memória e capacidade de processamento; do ponto de vista externo, o conjunto deve comportar-se de forma unificada, ocultando a distribuição física dos recursos.

As motivações para tal configuração são claras, entre elas temos a necessidade de processar volumes de dados que excedem a capacidade de uma única máquina; a busca por maior disponibilidade por meio da replicação de serviços; a redução de latência pela aproximação dos dados ao usuário; e a possibilidade de escalar horizontalmente a capacidade de processamento pela adição de novos nós, sem a necessidade de substituir o hardware existente.

2.1.1 Propriedades fundamentais

Para que um sistema distribuído cumpra seu propósito de forma eficaz, Tanenbaum e Van Steen identificam quatro propriedades fundamentais que devem ser perseguidas em seu projeto:

- a) **Transparência:** o sistema deve ocultar do usuário e das aplicações a complexidade inerente à sua natureza distribuída. Diferentes formas de transparência são identificadas na literatura, sendo as mais relevantes: transparência de acesso (recursos locais e remotos são acessados da mesma forma), de localização (o usuário desconhece onde o recurso está fisicamente), de replicação (múltiplas cópias de um recurso são gerenciadas de forma transparente) e de falha (o sistema continua operando mesmo na presença de falhas parciais);
- b) **Escalabilidade:** a capacidade do sistema de manter seu desempenho à medida que cresce em número de usuários, volume de dados ou área geográfica coberta. Um sistema é dito escalável quando pode crescer sem que seja necessário redesenhar sua arquitetura. A escalabilidade horizontal, que consiste na adição de novos nós, é a estratégia predominante em sistemas distribuídos modernos;

- c) **Tolerância a falhas:** a capacidade do sistema de continuar operando corretamente na presença de falhas em componentes individuais. Sistemas distribuídos alcançam tolerância a falhas por meio de replicação de dados, redundância de serviços e detecção e recuperação automática de nós defeituosos; e
- d) **Abertura:** a adoção de interfaces e protocolos padronizados que permitem a integração de componentes de diferentes origens e a extensão do sistema com novos serviços sem necessidade de modificar os existentes.

2.1.2 Evolução das plataformas de sistemas distribuídos

A forma como sistemas distribuídos são construídos e gerenciados evoluiu significativamente desde os primeiros *clusters* da década de 1980, em resposta ao desenvolvimento do hardware, das redes de interconexão e das demandas das aplicações.

2.1.2.1 Clusters clássicos

Os primeiros sistemas distribuídos de propósito geral, popularizados na década de 1990, eram formados por conjuntos homogêneos de máquinas conectadas por redes de alta velocidade e controlados por um *middleware* de gerenciamento de recursos. O modelo predominante era o processamento em lote (*batch processing*), no qual as tarefas são submetidas a uma fila, e o gerenciador de recursos as distribui pelos nós disponíveis conforme suas necessidades de CPU e memória. Exemplos representativos dessa classe incluem os *clusters* Beowulf e as grades computacionais (*grids*) como o BOINC e o HTCondor.

Nessa arquitetura, cada nó executa diretamente sobre seu sistema operacional, sem nenhuma camada de virtualização. O acoplamento entre software e hardware é, portanto, forte: uma aplicação desenvolvida para um determinado ambiente pode não funcionar em outro sem modificações.

2.1.2.2 Sistemas baseados em máquinas virtuais

Com a maturação das tecnologias de virtualização, em especial com a popularização do hipervisor Xen (Barham et al., 2003) e, posteriormente, do KVM (*Kernel-based Virtual Machine*), tornou-se possível abstrair o hardware físico em máquinas virtuais (VMs), cada uma executando seu próprio sistema operacional completo. Essa abordagem permitiu consolidar múltiplas cargas de trabalho em um mesmo servidor físico com isolamento robusto entre elas.

A virtualização introduziu uma separação fundamental entre hardware e software: uma VM pode ser criada, migrada e destruída sem qualquer dependência do hardware subjacente. Essa flexibilidade foi o alicerce tecnológico sobre o qual a computação em nuvem foi construída (Mell; Grance, 2011), permitindo que provedores como Amazon Web Services, Microsoft Azure e Google Cloud oferecessem recursos computacionais como serviço sob demanda.

A principal desvantagem das VMs reside em seu custo de inicialização, da ordem de minutos, e no overhead imposto pelo sistema operacional virtual, que duplica o consumo de recursos de memória e CPU.

2.1.2.3 Sistemas baseados em contêineres

Contêineres surgiram como uma alternativa mais leve à virtualização completa. Em vez de virtualizar o hardware, um contêiner isola uma aplicação no nível do sistema operacional, utilizando recursos nativos do *kernel* Linux: *namespaces*, que limitam a visibilidade do processo ao seu próprio ambiente (rede, sistema de arquivos, árvore de processos), e *cgroups* (*control groups*), que impõem limites ao consumo de recursos como CPU, memória e I/O (Burns et al., 2022).

A popularização do Docker a partir de 2013 padronizou o empacotamento e a distribuição de aplicações em imagens de contêiner, tornando os contêineres portáteis entre qualquer sistema que execute o *runtime* compatível. Em relação às VMs, os contêineres apresentam inicialização em milissegundos, menor consumo de memória e maior densidade de implantação em um mesmo servidor físico.

Em sistemas distribuídos de larga escala, a gestão manual de centenas ou milhares de contêineres torna-se inviável, motivando o surgimento de orquestradores como o Kubernetes, discutido posteriormente.

2.2 MIDDLEWARE

O termo *middleware* designa a camada de software posicionada entre o sistema operacional e as aplicações distribuídas, responsável por abstrair as heterogeneidades do hardware e da rede e por oferecer serviços comuns, como comunicação, sincronização, descoberta de serviços e gerenciamento de recursos, de forma transparente às aplicações (Bernstein, 1996). Sem essa camada, cada aplicação distribuída precisaria reimplementar individualmente toda a lógica de comunicação entre nós, tratamento de falhas e alocação de recursos.

A composição do *middleware* varia significativamente conforme o paradigma de virtualização adotado pela plataforma distribuída. As três classes de sistemas apresentadas na seção anterior, *clusters* clássicos, sistemas baseados em máquinas virtuais e sistemas baseados em contêineres, apresentam pilhas de *middleware* distintas, cada uma refletindo o nível de abstração e as características de isolamento de seu modelo de execução.

2.2.1 Middleware em clusters clássicos

Em *clusters* clássicos, em que cada nó executa diretamente sobre o sistema operacional sem virtualização, o *middleware* é composto principalmente por:

- a) **Gerenciadores de recursos e filas de trabalho:** sistemas como SLURM (*Simple Linux Utility for Resource Management*), PBS (*Portable Batch System*) e LSF (*Load Sharing Facility*) são responsáveis por receber trabalhos submetidos pelos usuários, gerenciar uma fila de espera e alocar nós conforme a disponibilidade de recursos. O modelo de execução é tipicamente em lote (*batch*), com prioridades estáticas;
- b) **Bibliotecas de passagem de mensagens:** o MPI (*Message Passing Interface*) é o padrão dominante para comunicação entre processos em *clusters* de alto desempenho. Implementações como OpenMPI e MPICH fornecem operações de comunicação ponto a ponto e coletivas (como *broadcast* e *AllReduce*) que abstraem a rede física subjacente; e
- c) **Sistemas de arquivos paralelos:** soluções como Lustre e GPFS (*General Parallel File System*) permitem que todos os nós do *cluster* acessem um mesmo sistema de arquivos compartilhado com alto desempenho, essencial para cargas que precisam ler e escrever grandes volumes de dados de forma concorrente.

2.2.2 Middleware em sistemas baseados em máquinas virtuais

Em sistemas baseados em máquinas virtuais, uma camada adicional de *middleware*, o *hypervisor*, é introduzida abaixo do sistema operacional de cada nó. Sobre essa base, o *middleware* de gerenciamento de VMs oferece:

- a) **Hipervisores:** software como Xen, KVM e VMware ESXi gerenciam a criação, o escalonamento de CPU e o isolamento de memória das VMs sobre o hardware físico. São classificados em Tipo 1 (*bare-metal*, executam diretamente sobre o hardware) e Tipo 2 (executam sobre um sistema operacional hospedeiro);
- b) **Plataformas de gerenciamento de nuvem:** soluções como OpenStack e VMware vCenter orquestram o ciclo de vida de VMs em *clusters* de servidores, expondo APIs para criação, migração e destruição de instâncias; e
- c) **Redes virtuais:** tecnologias como Open vSwitch (OVS) e SDN (*Software-Defined Networking*) implementam a rede entre VMs de forma programável, permitindo o isolamento de tráfego e a reconfiguração dinâmica da topologia de rede sem intervenção no hardware físico.

2.2.3 Middleware em sistemas baseados em contêineres

Em sistemas baseados em contêineres, o *middleware* abandona a abstração de máquinas completas em favor de uma abstração de serviços e cargas de trabalho. Os principais componentes são:

- a) **Orquestradores de contêineres:** o Kubernetes é o orquestrador mais utilizado, responsável por escalonar, monitorar e gerenciar o ciclo de vida de contêineres em *clusters* de servidores. Alternativas incluem Docker Swarm e Apache Mesos;
- b) **Runtimes de contêineres:** *containerd* e *CRI-O* são responsáveis pela execução efetiva dos contêineres no nível do sistema operacional, comunicando-se com o orquestrador por meio da interface CRI (*Container Runtime Interface*); e
- c) **Malhas de serviço** (*service meshes*): ferramentas como Istio e Linkerd gerenciam a comunicação entre microserviços, oferecendo funcionalidades como balanceamento de carga, criptografia mútua (mTLS), rastreamento distribuído e controle de tráfego de forma transparente às aplicações.

A Tabela 1 sintetiza as principais diferenças entre as pilhas de *middleware* dos três paradigmas discutidos.

Tabela 1 – Comparativo de *middleware* entre paradigmas de sistemas distribuídos.

Aspecto	Cluster clássico	Baseado em VMs	Baseado em contêineres
Unidade de execução	Processo	Máquina virtual	Contêiner
Isolamento	Nível de SO	SO completo	<i>Namespace/cgroup</i>
Tempo de inicialização	Minutos	Minutos	Milissegundos
Escalonador	SLURM / PBS	OpenStack / vCenter	Kubernetes
Comunicação	MPI / OpenMPI	Filas de mensagens	<i>Service mesh</i> / gRPC
Armazenamento	Lustre / GPFS	Ceph / SAN	CSI plugins
Virtualização de rede	Não	OVS / SDN	CNI plugins
Migração de estado	Checkpoint manual	vMotion / live migration	CRIU

Fonte: Elaborada pelo autor.

Atualmente, em serviços de nuvem que oferecem SaaS (*Software como Serviço*), PaaS (*Plataforma como Serviço*) e CaaS (*Contêineres como Serviço*) (Mell; Grance, 2011), sistemas baseados em contêineres são o modelo predominante, pela combinação de baixo *overhead*, alta densidade de implantação e capacidade de escalonamento elástico que essa arquitetura proporciona.

2.3 APRENDIZAGEM PROFUNDA DISTRIBUÍDA

O aprendizado profundo (*deep learning*) é uma subárea do aprendizado de máquina que utiliza redes neurais artificiais com múltiplas camadas de processamento para extrair representações hierárquicas de dados brutos (LeCun; Bengio; Hinton, 2015). Diferentemente de abordagens

tradicionais, em que características relevantes são definidas manualmente por especialistas, modelos de aprendizado profundo aprendem essas representações diretamente a partir dos dados durante o processo de treinamento.

A capacidade de modelar relações complexas em dados de alta dimensionalidade tornou o aprendizado profundo o estado da arte em tarefas como reconhecimento de imagens, processamento de linguagem natural, síntese de voz e condução autônoma (Goodfellow; Bengio; Courville, 2016). Essa versatilidade, entretanto, vem acompanhada de um custo computacional elevado, exigindo infraestruturas distribuídas para o treinamento de modelos em escala.

2.3.1 Redes neurais e o processo de treinamento

Uma rede neural artificial é composta por camadas de nós interconectados, chamados neurônios, organizadas em uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída. Cada conexão entre neurônios possui um peso, que representa a intensidade da influência de um neurônio sobre outro. O conjunto de todos os pesos define os parâmetros do modelo.

O treinamento de uma rede neural consiste em ajustar iterativamente esses parâmetros de forma a minimizar uma função de perda $\mathcal{L}$, que mede a discrepância entre as saídas produzidas pelo modelo e as respostas esperadas. Esse ajuste é realizado pelo algoritmo de gradiente descendente estocástico (*Stochastic Gradient Descent* – SGD) e suas variantes, como Adam (Kingma; Ba, 2015) e RMSProp.

A cada iteração de treinamento, denominada *step*, são executadas três etapas fundamentais:

- a) **Propagação direta** (*forward pass*): o lote de dados de entrada (*mini-batch*) é processado camada a camada até produzir uma predição na saída da rede;
- b) **Cálculo da perda**: a função de perda é avaliada comparando a predição com o valor esperado;
- c) **Retropropagação** (*backward pass*): os gradientes da função de perda em relação a cada parâmetro são calculados por meio da regra da cadeia e os pesos são atualizados na direção oposta ao gradiente.

Um conjunto completo de iterações sobre todo o conjunto de dados de treinamento é denominado *época*. Modelos de grande porte podem exigir centenas de épocas e bilhões de parâmetros para convergir, o que torna o treinamento inviável em um único dispositivo dentro de um tempo razoável.

2.3.2 Demanda computacional e aceleradores de hardware

A operação dominante no treinamento de redes neurais profundas é a multiplicação de matrizes, presente tanto na propagação direta quanto na retropropagação. Essa característica

torna o treinamento altamente paralelizável e favorece o uso de unidades de processamento gráfico (GPUs), que possuem milhares de núcleos de cálculo operando em paralelo, em contraste com as dezenas de núcleos presentes em CPUs modernas.

Modelos contemporâneos, como os grandes modelos de linguagem (*Large Language Models* – LLMs) da família GPT (Brown et al., 2020) e PaLM (Chowdhery et al., 2023), chegam a centenas de bilhões de parâmetros. O treinamento desses modelos pode demandar milhares de GPUs operando simultaneamente por semanas, evidenciando a necessidade de sistemas distribuídos eficientes para a execução dessas cargas de trabalho.

Além da capacidade de processamento, a memória da GPU constitui um gargalo crítico. Um modelo com N parâmetros em precisão de 32 bits (*float32*) ocupa $4N$ bytes apenas para armazenar os pesos. Somando os gradientes e os estados do otimizador, o consumo de memória pode ultrapassar facilmente a capacidade de uma única GPU, impondo a necessidade de distribuição do modelo entre múltiplos dispositivos.

2.3.3 Estratégias de treinamento distribuído

O treinamento distribuído de redes neurais profundas pode ser organizado segundo diferentes estratégias de paralelização, cada uma com características próprias quanto à divisão do trabalho, ao volume de comunicação entre dispositivos e à complexidade de implementação (Ben-Nun; Hoefler, 2019).

2.3.3.1 Paralelismo de dados

No paralelismo de dados, o conjunto de treinamento é particionado entre os trabalhadores (*workers*), cada um dos quais mantém uma réplica completa do modelo. A cada iteração, cada trabalhador processa um subconjunto distinto dos dados e calcula gradientes locais. Em seguida, os gradientes de todos os trabalhadores são agregados, geralmente pela média, e aplicados simultaneamente a todas as réplicas, mantendo-as idênticas.

É a estratégia mais amplamente adotada por sua relativa simplicidade de implementação e boa escalabilidade horizontal. Sua limitação reside no fato de que o modelo completo deve caber na memória de cada dispositivo, o que impede seu uso para modelos que excedam essa capacidade.

2.3.3.2 Paralelismo de modelo

No paralelismo de modelo, as camadas ou grupos de camadas da rede são distribuídas entre diferentes dispositivos. Cada dispositivo é responsável por um subconjunto das operações do modelo, e as ativações são transferidas sequencialmente entre eles durante a propagação direta e inversa.

Essa abordagem é especialmente útil quando um modelo é grande demais para caber em um único dispositivo. A principal desvantagem é a baixa utilização dos dispositivos: enquanto

um deles processa sua fração do modelo, os demais ficam ociosos aguardando as ativações da camada anterior.

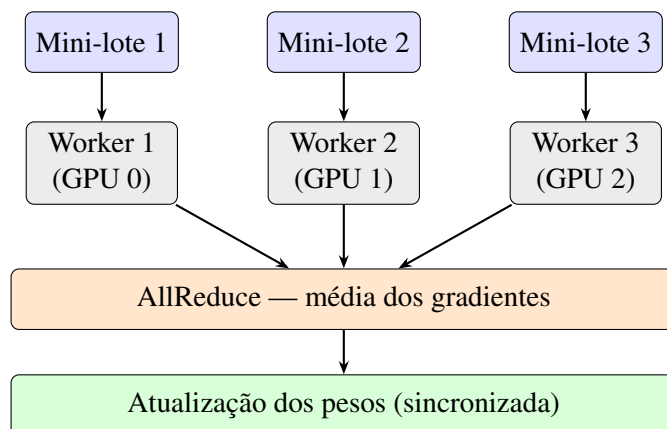
2.3.3.3 Paralelismo de pipeline

O paralelismo de *pipeline* é uma evolução do paralelismo de modelo que busca reduzir a ociosidade dos dispositivos. O lote de dados é subdividido em microlotes (*microbatches*), que são injetados no *pipeline* de forma intercalada. Enquanto um dispositivo processa um microlote na sua fração do modelo, o dispositivo anterior já inicia o processamento do microlote seguinte, aumentando a sobreposição entre computação e comunicação (Narayanan et al., 2021).

Na prática, sistemas de grande escala combinam as três estratégias, paralelismo de dados, de modelo e de *pipeline*, em uma abordagem denominada paralelismo 3D, capaz de utilizar eficientemente milhares de aceleradores simultaneamente.

A Figura 1 ilustra o fluxo de treinamento no paradigma de paralelismo de dados, evidenciando a etapa de agregação de gradientes por meio da operação *AllReduce*.

Figura 1 – Treinamento distribuído com paralelismo de dados e AllReduce.



Fonte: Elaborada pelo autor.

2.3.4 Principais arcabouços para treinamento distribuído

Diversos arcabouços (*frameworks*) foram desenvolvidos para facilitar a implementação de treinamento distribuído, abstraindo a complexidade das operações coletivas e do gerenciamento de dispositivos:

- PyTorch Distributed** (Li et al., 2020): integrado ao PyTorch, fornece primitivas de comunicação coletiva e a interface *DistributedDataParallel* (DDP) para paralelismo de dados com sincronização eficiente de gradientes;
- Horovod** (Sergeev; Balso, 2018): desenvolvido pela Uber, oferece uma camada de abstração sobre MPI e NCCL para paralelismo de dados, sendo compatível com TensorFlow, PyTorch e Keras;

- c) **DeepSpeed** (Rajbhandari et al., 2020): biblioteca da Microsoft que implementa otimizações avançadas de memória, como o ZeRO (*Zero Redundancy Optimizer*), permitindo o treinamento de modelos com centenas de bilhões de parâmetros; e
- d) **Megatron-LM** (Shoeybi et al., 2019): desenvolvido pela NVIDIA, especializado em paralelismo de modelo e de *pipeline* para modelos de linguagem de grande escala.

Esses arcabouços são tipicamente implantados sobre infraestruturas de contêineres orquestradas pelo Kubernetes, que é responsável por alocar os recursos de GPU, gerenciar o ciclo de vida dos trabalhadores e garantir que todos os nós de um trabalho distribuído sejam inicializados em conjunto, requisito fundamental para o correto funcionamento das operações coletivas de comunicação.

2.4 KUBERNETES

O Kubernetes é um sistema de orquestração de contêineres de código aberto, originalmente desenvolvido pela Google e doado à *Cloud Native Computing Foundation* (CNCF) em 2014 (Burns et al., 2016). Seu objetivo central é automatizar a implantação, o escalonamento e o gerenciamento de aplicações containerizadas em *clusters* de máquinas físicas ou virtuais, abstraindo a complexidade da infraestrutura subjacente.

A adoção do Kubernetes como plataforma para execução de cargas de trabalho de aprendizado de máquina cresceu de forma significativa nos últimos anos, impulsionada pela sua capacidade de gerenciar recursos heterogêneos, incluindo GPUs e outros aceleradores, e pela disponibilidade de extensões voltadas especificamente para esse domínio (Burns et al., 2022).

2.4.1 Arquitetura do Kubernetes

O Kubernetes adota uma arquitetura cliente-servidor dividida em dois planos funcionais: o *plano de controle* (*control plane*), responsável pelas decisões globais sobre o *cluster*, e o *plano de dados* (*data plane*), composto pelos nós de trabalho nos quais as cargas de trabalho efetivamente são executadas.

2.4.1.1 Plano de controle

O plano de controle é formado pelos seguintes componentes principais:

- a) **API Server** (*kube-apiserver*): é o ponto central de comunicação do *cluster*. Todos os demais componentes, incluindo operadores externos e ferramentas de linha de comando, interagem com o Kubernetes exclusivamente por meio de sua API REST. O API Server valida e persiste os objetos de estado no *etcd*;
- b) **etcd**: banco de dados distribuído do tipo chave-valor, utilizado como repositório de estado de todo o *cluster*. Armazena a configuração de todos os objetos Kubernetes, como *Pods*,

Deployments e *ConfigMaps*. Por ser o único componente com persistência de estado, sua disponibilidade é crítica para o funcionamento do *cluster*;

- c) **Escalonador** (*kube-scheduler*): observa os *Pods* recém-criados que ainda não foram atribuídos a nenhum nó e seleciona o nó mais adequado para cada um deles, com base em critérios de recursos disponíveis, restrições de afinidade e políticas de escalonamento. O escalonador padrão do Kubernetes é extensível por meio de um *framework* de *plugins* denominado *Scheduling Framework* (Kubernetes Community, 2019);
- d) **Gerenciador de controladores** (*kube-controller-manager*): executa um conjunto de controladores que observam continuamente o estado do *cluster* e atuam para aproximá-lo do estado desejado. Exemplos incluem o controlador de réplicas, que garante o número correto de cópias de um *Pod*, e o controlador de *Jobs*, que gerencia a execução de tarefas em lote.

2.4.1.2 Plano de dados

Cada nó de trabalho executa os seguintes componentes:

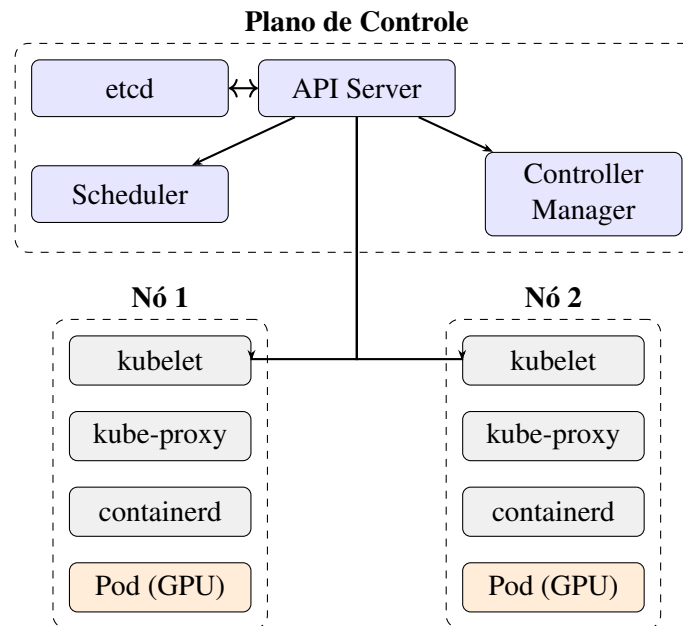
- a) **Kubelet**: agente principal do nó. Recebe as especificações de *Pods* atribuídos ao nó pelo escalonador e instrui o *runtime* de contêineres a iniciá-los, monitorá-los e encerrá-los. Reporta periodicamente o estado dos *Pods* e a capacidade de recursos do nó ao API Server;
- b) **Kube-proxy**: componente responsável pela implementação das regras de rede que permitem a comunicação entre *Pods* e a exposição de serviços, gerenciando regras de *iptables* ou *eBPF* no sistema operacional do nó; e
- c) **Container Runtime**: software responsável pela execução efetiva dos contêineres. O Kubernetes se comunica com o *runtime* por meio da interface padronizada CRI (*Container Runtime Interface*). As implementações mais utilizadas são *containerd* e *CRI-O*.

A Figura 2 apresenta uma visão geral da arquitetura do Kubernetes, destacando os componentes do plano de controle e do plano de dados e os principais fluxos de comunicação entre eles.

2.4.2 Objetos fundamentais do Kubernetes

O Kubernetes representa o estado desejado do *cluster* por meio de objetos declarativos armazenados no *etcd*. Os objetos mais relevantes para a execução de cargas de aprendizado profundo são descritos a seguir.

Figura 2 – Arquitetura do Kubernetes: plano de controle e nós de trabalho.



Fonte: Elaborada pelo autor.

2.4.2.1 Pod

O *Pod* é a menor unidade implantável do Kubernetes. Consiste em um ou mais contêineres que compartilham o mesmo namespace de rede e sistema de arquivos temporário. Em cargas de aprendizado profundo distribuído, cada trabalhador (*worker*) do processo de treinamento é tipicamente representado por um *Pod* distinto, cada um requisitando uma ou mais GPUs.

2.4.2.2 Job e IndexedJob

O objeto *Job* gerencia a execução de tarefas em lote, garantindo que um número especificado de *Pods* seja executado até a conclusão. O *IndexedJob* é uma variação em que cada *Pod* recebe um índice numérico único, utilizado pelos arcabouços de treinamento distribuído para identificar o papel de cada trabalhador, como mestre e trabalhadores no paradigma de servidor de parâmetros.

2.4.2.3 Modelo de recursos

O Kubernetes gerencia recursos computacionais por meio de dois conceitos complementares associados a cada contêiner:

- Requisição (*request*):** quantidade mínima de recurso garantida ao contêiner. O escalonador utiliza as requisições para decidir em qual nó alocar o *Pod*, garantindo que o nó possua capacidade suficiente; e
- Limite (*limit*):** quantidade máxima de recurso que o contêiner pode consumir. Para CPU, o excesso é limitado pelo *kernel*; para memória, o excesso provoca a finalização do contêiner

por *Out Of Memory* (OOM).

Recursos de GPU são expostos ao Kubernetes por meio de *Device Plugins*, que registram a capacidade do acelerador em cada nó. Diferentemente de CPU e memória, GPUs são tratadas como recursos inteiros e indivisíveis: uma requisição de GPU deve ser satisfeita por uma unidade física completa, o que torna a disputa por esse recurso particularmente sensível às decisões do escalonador. Uma forma de contornar esse problema, utiliza-se o conceito de *time-slice*, o qual faz com que uma GPU possa ser utilizada por vários processos por uma parcela de tempo, o que pode ser útil em alguns casos.

2.4.3 O escalonador padrão do Kubernetes

O *kube-scheduler* opera em dois estágios principais para cada *Pod* não atribuído (Kubernetes Community, 2019):

- a) **Filtragem** (*filtering*): percorre todos os nós do *cluster* e elimina aqueles que não satisfazem as restrições do *Pod*, capacidade de recursos insuficiente, restrições de afinidade não atendidas, *taints* incompatíveis, entre outros. O resultado é um conjunto de nós candidatos; e
- b) **Pontuação** (*scoring*): atribui uma pontuação a cada nó candidato com base em um conjunto de funções de prioridade, como balanceamento de carga e proximidade dos dados. O *Pod* é atribuído ao nó de maior pontuação.

O *Scheduling Framework* permite que *plugins* personalizados sejam inseridos em pontos de extensão bem definidos desse ciclo, como *PreFilter*, *Filter*, *Score*, *Reserve* e *PostBind*, sem a necessidade de modificar o código-fonte do escalonador principal. Essa arquitetura é a base sobre a qual o escalonador proposto neste trabalho é construído.

Uma limitação relevante do escalonador padrão para cargas de aprendizado profundo é a ausência de *gang scheduling*: o escalonador trata cada *Pod* de forma independente, podendo alocar parte dos trabalhadores de um trabalho distribuído enquanto os demais aguardam na fila. Essa situação, denominada *head-of-line blocking* parcial, mantém GPUs alocadas e ociosas por um trabalhador que não pode iniciar o treinamento enquanto seus pares não são escalonados (Gu et al., 2019).

2.4.4 Preempção no Kubernetes

O Kubernetes implementa um mecanismo nativo de preempção baseado em *PriorityClass*, um objeto que associa um valor numérico de prioridade a um conjunto de *Pods*. Quando um *Pod* de alta prioridade não encontra nós com capacidade suficiente, o escalonador pode nomear candidatos a preempção: *Pods* de menor prioridade em execução cujos recursos, somados,

seriam suficientes para acomodar o *Pod* pendente. Os *Pods* selecionados são encerrados e o nó é liberado.

Esse mecanismo, contudo, apresenta limitações críticas para cargas de aprendizado profundo:

- a) a preempção provoca o encerramento imediato dos contêineres selecionados, no caso de tarefas de aprendizado de máquina, descartando o estado de treinamento acumulado em memória; e
- b) não há suporte nativo para a preservação e transferência do estado dos processos interrompidos, tornando necessário que o trabalho preemptado reinicie do zero ou a partir de um *checkpoint* previamente salvo em disco, quando disponível.

Essas limitações motivam a proposta deste trabalho: integrar a preempção baseada em prioridades com a migração ao vivo de contêineres, de modo que o estado de treinamento seja preservado e restaurado automaticamente, reduzindo o custo efetivo da preempção para as tarefas interrompidas.

2.5 MIGRAÇÃO DE CONTÊINERES

A migração de processos entre nós de um sistema distribuído é um mecanismo estabelecido na literatura de sistemas operacionais e computação em grade, com aplicações que vão do balanceamento de carga à tolerância a falhas (Milojicic et al., 2000). Em sistemas baseados em contêineres, a migração consiste em capturar o estado completo de um contêiner em execução em um nó de origem e restaurá-lo em um nó de destino, de forma que o processo retome sua execução a partir do ponto em que foi interrompido, sem perda do estado computacional acumulado.

Dois paradigmas de migração são distinguidos na literatura conforme o impacto na disponibilidade do processo migrado:

- a) **Migração a frio** (*cold migration*): o contêiner é encerrado no nó de origem, sua configuração é transferida para o nó de destino, no qual ele é reiniciado. O processo fica indisponível durante toda a transferência. É a abordagem mais simples, porém implica uma interrupção (*downtime*) proporcional ao volume de informação a ser recuperada; e
- b) **migração ao vivo** (*hot migration* ou *live migration*): o processo continua executando no nó de origem enquanto seu estado é transferido de forma iterativa para o nó de destino. Somente ao final, quando o estado residual é pequeno o suficiente, o processo é brevemente suspenso para a transferência final, resultando em um *downtime* mínimo. Essa abordagem é amplamente utilizada em ambientes de máquinas virtuais (Clark et al., 2005) e constitui o objeto central desta seção.

2.5.1 CRIU: Checkpoint/Restore In Userspace

O CRIU (*Checkpoint/Restore In Userspace*) é uma ferramenta de código aberto para o sistema operacional Linux que implementa a captura e restauração do estado de processos em execução inteiramente no espaço de usuário, sem a necessidade de modificações no *kernel* (CRIU Project, 2012). É o mecanismo subjacente mais utilizado para a implementação de migração ao vivo de processos entre sistemas. Nos últimos anos, o desenvolvimento da aplicação, com foco em sistemas baseados em contêineres, fez com que se tornasse a solução padrão para migração ao vivo nesse contexto.

A operação de *checkpoint* de contêiner realizada pelo CRIU compreende a captura dos seguintes elementos do estado do processo:

- a) **Memória:** o conteúdo de todas as regiões de memória mapeadas pelo processo, incluindo a *stack*, o *heap* e regiões de memória compartilhada. As páginas são escritas em arquivos;
- b) **Descritores de arquivo:** o estado de todos os arquivos abertos pelo processo, incluindo arquivos regulares, *pipes*, *sockets* e outros descritores. O CRIU preserva os deslocamentos de leitura/escrita e, no caso de *sockets* de rede, captura o estado das conexões TCP estabelecidas utilizando extensões do *kernel*;
- c) **Estado dos registradores e contexto de execução:** o conteúdo dos registradores da CPU para cada *thread* do processo, o que permite que a execução seja retomada exatamente na instrução em que foi suspensa; e
- d) **Namespaces e *cgroups*:** as configurações de isolamento do contêiner, incluindo namespaces de rede, PID, montagem e usuário, bem como os limites de recursos impostos pelos *cgroups*.

A operação de restauração reconstrói todos esses elementos no nó de destino, recriando o ambiente de execução do contêiner e retomando os processos a partir do estado capturado. Do ponto de vista da aplicação em execução, a migração é transparente: o processo não é capaz de distinguir se foi migrado ou se executou continuamente no mesmo nó.

2.5.2 Migração ao vivo de contêineres

A migração ao vivo de contêineres combina o mecanismo de *checkpoint/restauração* do CRIU com uma estratégia de transferência iterativa de memória, análoga à técnica de *pre-copy* utilizada na migração ao vivo de máquinas virtuais (Clark et al., 2005). O processo ocorre nas seguintes etapas:

- a) **Transferência iterativa das páginas de memória (*pre-copy*):** o contêiner continua executando no nó de origem enquanto suas páginas de memória são copiadas progressivamente

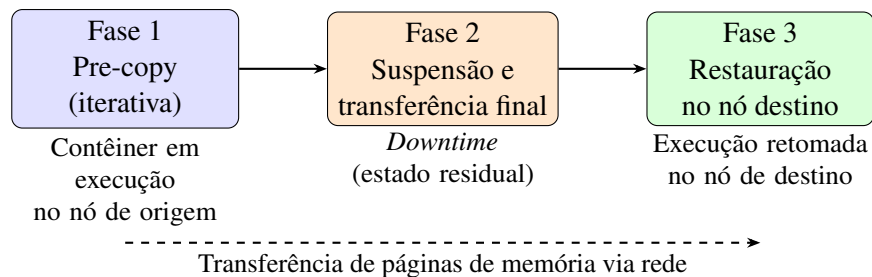
para o nó de destino. Páginas modificadas durante a transferência (*dirty pages*) são rastreadas e retransferidas nas iterações seguintes, até que o conjunto de páginas modificadas residuais seja suficientemente pequeno;

- b) **Suspensão e transferência final:** o contêiner é brevemente suspenso no nó de origem. O estado residual, páginas ainda não transferidas, descritores de arquivo e contexto de execução, é transferido para o nó de destino; e
- c) **Restauração e retomada:** o contêiner é restaurado no nó de destino e retoma a execução. O nó de origem libera os recursos previamente alocados.

A duração do *downtime* na etapa de suspensão depende fundamentalmente do volume de estado residual não transferido ao final da fase de *pre-copy*. Em aplicações com alta taxa de modificação de memória, como é o caso durante a retropropagação em redes neurais profundas, o conjunto de páginas modificadas por iteração pode ser grande, exigindo um número maior de rodadas de *pre-copy* ou aceitando um *downtime* maior na transferência final.

A Figura 3 ilustra as três fases do protocolo de migração ao vivo, destacando o momento de suspensão do contêiner de origem e a retomada no nó de destino.

Figura 3 – Fases da migração ao vivo de contêineres com CRIU.



Fonte: Elaborada pelo autor.

2.5.3 Desafios na migração de contêineres para aprendizado profundo

A aplicação de migração ao vivo de contêineres executando cargas de aprendizado profundo distribuído introduz desafios específicos que vão além dos cenários tradicionais de migração de serviços *stateless*:

- a) **Volume de estado em memória:** modelos de aprendizado profundo modernos armazenam em memória os pesos do modelo, os gradientes e os estados do otimizador. Em modelos grandes, esse conjunto pode facilmente superar dezenas de gigabytes por trabalhador, tornando a fase de *pre-copy* longa e a largura de banda da rede de interconexão um gargalo crítico;

- b) **Estado de aceleradores de hardware:** o CRIU opera sobre processos do espaço de usuário e captura o estado da memória principal (RAM). O estado dos aceleradores de hardware, como o conteúdo da memória da GPU, não é capturado diretamente pelo CRIU e requer mecanismos complementares. Bibliotecas como o NVIDIA CUDA oferecem APIs de captura de contexto que podem ser integradas ao fluxo de migração, mas essa integração aumenta a complexidade da implementação (NVIDIA Corporation, 2023);
- c) **Coordenação em treinamento distribuído:** em um trabalho de treinamento com múltiplos trabalhadores realizando operações coletivas síncronas, a migração de um único trabalhador implica a suspensão coordenada de todos os demais, pois operações como *AllReduce* são bloqueantes. A migração deve, portanto, ser tratada como uma operação de grupo, o que aumenta a complexidade do protocolo de migração; e
- d) **Reconexão de canais de comunicação:** após a restauração, os trabalhadores migrados precisam restabelecer os canais de comunicação com os demais nós do grupo de treinamento. A reconfiguração do grupo de processos distribuídos, por exemplo, por meio do *rendezvous* do PyTorch Distributed, deve ser automatizada pelo escalonador para que a retomada do treinamento seja transparente à aplicação.

Esses desafios evidenciam que a migração ao vivo de contêineres para aprendizado profundo distribuído requer uma integração cuidadosa entre o escalonador, o mecanismo de migração e os arcabouços de treinamento. A proposta apresentada neste trabalho aborda esses pontos por meio de um protocolo de migração coordenada que é descrito em detalhe no Capítulo 4.

2.6 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a fundamentação teórica necessária para a compreensão do trabalho proposto. Foram abordados quatro temas interconectados que constituem o alicerce conceitual do escalonador desenvolvido.

Inicialmente, foram discutidos os fundamentos dos sistemas distribuídos, incluindo suas propriedades essenciais de transparência, escalabilidade e tolerância a falhas, e a evolução das plataformas de execução distribuída desde os *clusters* clássicos até os sistemas baseados em contêineres. Em seguida, foi apresentada a camada de *middleware* característica de cada paradigma, sintetizada na Tabela 1, que evidencia o deslocamento progressivo do nível de abstração: de processos sobre sistema operacional, passando por máquinas virtuais completas, até serviços e contêineres gerenciados por orquestradores.

A seção de aprendizado profundo distribuído contextualizou a demanda computacional das cargas de trabalho de interesse, modelos com bilhões de parâmetros, treinados em *clusters* de GPUs por meio de estratégias de paralelismo de dados, de modelo e de *pipeline*, e identificou os arcabouços que implementam essas estratégias sobre infraestruturas de contêineres.

A seção sobre Kubernetes detalhou a arquitetura do orquestrador, seu modelo de recursos, o ciclo de escalonamento do *kube-scheduler* e os pontos de extensão do *Scheduling Framework*, que são a interface de integração do escalonador proposto neste trabalho. Foram também identificadas as limitações da preempção nativa para cargas de aprendizado profundo, mostrando a necessidade de uma alternativa apropriada.

Por fim, a seção de migração de contêineres apresentou o mecanismo CRIU como base para a migração ao vivo, descreveu as fases do protocolo de *pre-copy* e identificou os desafios específicos introduzidos pelo grande volume de estado em memória das cargas de aprendizado profundo distribuído, em particular a necessidade de coordenar a suspensão de múltiplos trabalhadores e de reconectar o grupo de processos após a restauração.

3 TRABALHOS RELACIONADOS

Neste capítulo são discutidos os trabalhos da literatura que abordam os dois eixos temáticos centrais desta dissertação: a migração ao vivo de contêineres em ambientes de orquestração e o escalonamento especializado de cargas de aprendizado de máquina em *clusters* de GPU. O objetivo é posicionar a proposta deste trabalho em relação ao estado da arte, identificando as lacunas que motivam a contribuição aqui apresentada.

3.1 MIGRAÇÃO AO VIVO DE CONTÊINERES

O problema da migração de processos entre nós de sistemas distribuídos é tratado na literatura desde a década de 1980. Milojevic et al. (2000) apresentam uma revisão abrangente dos trabalhos publicados até o início dos anos 2000, organizando as abordagens segundo os critérios de transparência, desempenho e suporte do sistema operacional. Os autores identificam que a principal barreira para a adoção de migração em produção é o alto custo de suspensão dos processos durante a transferência de estado, problema que a técnica de *pre-copy* viria a amenizar.

A migração ao vivo com *pre-copy* foi consolidada no contexto de máquinas virtuais pelo trabalho seminal de Clark et al. (2005), que descreveu sua implementação no hipervisor Xen. A ideia central, que é a transferência de páginas de memória iterativamente enquanto a VM permanece em execução e realizar a suspensão apenas para a transferência do estado residual, reduziu o *downtime* de migração de dezenas de segundos para dezenas de milissegundos em cargas típicas de servidor *web*. Esse resultado estabeleceu a migração ao vivo como mecanismo viável para ambientes de produção e passou a ser amplamente adotado por plataformas de nuvem como o VMware vMotion e o KVM.

A transição do paradigma de máquinas virtuais para contêineres deslocou o foco para mecanismos de *checkpoint/restauração* no nível do processo. O CRIU (*Checkpoint/Restore In Userspace*) (CRIU Project, 2012) emergiu como a implementação de referência para o sistema operacional Linux, operando inteiramente no espaço de usuário por meio das chamadas de sistema *ptrace* e *proc*. Ao contrário dos hipervisores, que gerenciam o estado de uma máquina virtual completa, o CRIU captura seletivamente o estado de um conjunto de processos, como explicado no capítulo 2.

A integração do CRIU ao ecossistema de contêineres ocorreu de forma gradual. O suporte experimental ao comando *docker checkpoint* foi introduzido no Docker a partir da versão 1.13, e o *containerd* passou a expor operações de *checkpoint* por meio de sua API interna, base sobre a qual o Kubernetes pode orquestrar o salvamento do estado de *Pods*. Contudo, a integração entre o Kubernetes e o CRIU permanece sem suporte oficial estável até a redação deste trabalho, sendo tratada por meio de extensões experimentais e operadores de terceiros.

3.1.1 CloudHopper

Benjaponpitak, Karakate & Sripanidkulchai (2020) propõem o CloudHopper, o primeiro sistema a demonstrar migração ao vivo automatizada de contêineres interdependentes entre diferentes provedores de nuvem pública (Amazon Web Services, Google Cloud Platform e Microsoft Azure). O trabalho é relevante por ser o primeiro a validar a combinação de CRIU com a técnica de *pre-dump* e *pre-copy* em um ambiente de migração multi-nó sobre a área geográfica (*wide area*), com conectividade de rede real e cargas de trabalho realistas.

O protocolo de migração adotado pelo CloudHopper é composto por cinco etapas:

1. *pre-dump*, em que o CRIU captura uma imagem inicial da memória do contêiner sem congelá-lo, permitindo que o serviço continue respondendo;
2. *pre-copy*, em que a imagem do *pre-dump* e o armazenamento persistente são transferidos para o nó de destino via *rsync* através de uma VPN;
3. *checkpoint*, em que o contêiner é suspenso e as alterações de memória realizadas durante o *pre-copy* são capturadas;
4. transferência das mudanças incrementais; e
5. restauração no destino.

Durante as etapas 3 e 4, em que o contêiner de origem está suspenso, um *proxy* HAProxy retém as conexões de clientes ativos e as redireciona para o destino após a restauração, ocultando a interrupção dos usuários finais.

A avaliação do sistema migra um par de contêineres WordPress e MySQL entre os três provedores, variando a carga de clientes concorrentes. O *downtime* observado varia entre 7,5 segundos (sem carga) e 24 segundos (400 conexões concorrentes), sendo dominado pelo volume de dados do *checkpoint*, que pode superar 1 GB para o contêiner MySQL com muitas escritas. A vazão de transferência com escalonamento paralelo de múltiplos contêineres atinge 793 Mbps entre AWS e GCP.

O CloudHopper, contudo, é restrito a cargas de trabalho *web* sem aceleradores de hardware, e o mecanismo de retenção de conexões é específico para o HAProxy. Não há suporte a Kubernetes, a recursos de GPU ou a múltiplos processos de treinamento que exijam coordenação de grupo.

3.1.2 Sledge

Xu et al. (2020) identificam que a migração ao vivo de contêineres Docker exige a transferência de três componentes independentes: a imagem do contêiner, o estado de execução em tempo real e o contexto de gerenciamento do daemon Docker. Soluções anteriores, como o *Live Migration Manager* (LMM), tratavam esses três componentes de forma sequencial e dependiam

de um registro centralizado de imagens para a transferência da camada de sistema de arquivos, o que introduzia gargalos significativos de rede e armazenamento.

O Sledge aborda cada componente com uma técnica dedicada. Para a imagem, é proposto o *Lightweight Container Registry* (LCR), um protocolo de distribuição ponto a ponto (*peer-to-peer*) que elimina a dependência de um registro centralizado: o nó de destino recupera as camadas da imagem diretamente do nó de origem por meio de um servidor HTTP temporário, reduzindo a latência de transferência ao evitar o desvio pelo registro. Para o estado de execução, o Sledge utiliza o protocolo de migração incremental do CRIU em modalidade *pre-copy*, transferindo páginas de memória suja de forma iterativa enquanto o contêiner ainda executa. Para o contexto de gerenciamento, é proposto o *Dynamic Context Loading* (DCL), um mecanismo que carrega o contexto administrativo do Docker, dados de configuração, redes e volumes associados ao contêiner, diretamente na memória do daemon em execução no nó de destino, sem necessidade de reiniciá-lo.

Os experimentos conduzidos pelos autores demonstram que o Sledge reduz em 57% o tempo total de migração, 55% o tempo de transferência de imagem e 70% o *downtime* em relação ao LMM. A separação explícita dos três componentes de migração representa uma contribuição metodológica relevante, pois permite otimizar cada etapa de forma independente. O Sledge não aborda, porém, contêineres com estado de GPU nem considera o escalonamento coordenado de múltiplos contêineres interdependentes.

3.1.3 MBDPC

Lu & Jiang (2022) propõem o MBDPC (*Migration Based on Dirty Page prediction and Compression*), uma otimização do protocolo de *pre-copy* que combina predição de páginas sujas e compressão incremental para reduzir o volume de dados transferidos durante a fase iterativa da migração.

A técnica de predição é baseada em um modelo de *Random Forest* que analisa o histórico de memórias sujas (*dirty pages*) na memória, capturado pelos *bitmaps* de páginas sujas produzidos pelo CRIU em iterações anteriores, e classifica páginas como frequentemente sujas (*hot*) ou raramente sujas (*cold*). Páginas classificadas como *hot* em rodadas anteriores e que permanecem sujas na rodada atual são omitidas das transferências intermediárias: o sistema assume que elas continuarão sendo modificadas até a iteração final (*stop-and-copy*), de modo que enviá-las prematuramente geraria retransmissões desnecessárias. Após a predição, as páginas selecionadas são comprimidas com o algoritmo LZ4 antes de cada transferência incremental, reduzindo adicionalmente a carga sobre o enlace de rede.

Os autores avaliam o MBDPC em cargas sintéticas de alta taxa de sujamento e relatam reduções de até 61,8% no tempo total de migração e de 65% no volume de dados transmitidos em relação ao protocolo de *pre-copy* padrão do CRIU. O trabalho é limitado a contêineres de única instância (sem distribuição multi-nó) e não considera o estado de aceleradores de hardware; a predição de páginas sujas constitui, contudo, uma técnica de potencial aplicação ao cenário de

migração ao vivo em *clusters* com GPU, nos quais o volume de memória transferível pode ser substancial.

3.1.4 Migração Segura e Leve

Singh et al. (2024) propõem uma técnica de migração de contêineres que combina leveza (*lightweight*) e segurança como requisitos de projeto conjuntos. A motivação é que abordagens existentes, como o *pre-copy* completo, transferem um volume excessivo de páginas de memória na fase de *pre-dump*, incluindo páginas inativas que não serão utilizadas pelo processo no período entre o *checkpoint* e a retomada da execução.

Para reduzir esse volume, os autores propõem uma política de *pre-dump* baseada em probabilidade: páginas são incluídas na transferência inicial apenas se sua probabilidade de acesso, calculada a partir de atributos como *dirty bit*, indicador de uso recente e tipo de mapeamento de memória, supera um limiar configurável. O processo é dividido em três fases algorítmicas: identificação de páginas ativas por análise de atributos, cálculo da taxa de atualização de cada página com base no histórico de *bitmaps* e finalização do *checkpoint* com envio seletivo. Para a segurança, a transferência é realizada sobre canal SSH/SFTP com autenticação por par de chaves RSA-2048 e verificação de integridade via SHA-256 em cada arquivo de *checkpoint*.

Os experimentos relatam uma redução de 26,48% no tamanho do *pre-dump* e de 31,04% no volume de dados transferidos durante as iterações de *pre-copy* em relação ao protocolo padrão. A proposta é avaliada em um cenário de nuvem pública com máquinas virtuais e não aborda migração em ambientes Kubernetes nem contêineres com estado de GPU. A política de transferência seletiva de páginas constitui, porém, um princípio complementar às técnicas de predição discutidas no MBDPC (Lu; Jiang, 2022).

3.1.5 FastMig

Manatura et al. (2024) propõem o FastMig, um sistema de migração ao vivo de serviços containerizados construído sobre o FastFreeze, um utilitário de *checkpoint*/restauração baseado em CRIU, especialmente projetado para contêineres. A principal contribuição do FastMig é a introdução de duas operações adicionais ao fluxo tradicional de migração: o *pós-checkpointing* e a *pré-restauração*.

A operação de *pós-checkpointing* desacopla o encerramento do processo de serviço do encerramento do contêiner, permitindo que instruções de finalização graciosa, como escrever logs, liberar recursos e persistir estado, sejam executadas após o *checkpoint* e antes do contêiner ser removido da origem. Isso é possível porque o FastMig introduz uma camada de gerenciamento de serviço (*FastFreeze Daemon*) que age como processo *init* do contêiner, decoplando o ciclo de vida do serviço do ciclo de vida do contêiner.

A operação de *pré-restauração*, por sua vez, implementa a técnica de *warm restoration*: o contêiner é iniciado no nó de destino em modo de espera (*standby*) antes mesmo da transferência

dos arquivos de *checkpoint*, eliminando o tempo de inicialização do contêiner do caminho crítico da migração. Quando os arquivos chegam, a restauração acontece sobre um contêiner já aquecido, o que reduz significativamente o *downtime* percebido. Os experimentos mostram uma redução de 30 vezes no tempo de migração em comparação com o FastFreeze original para serviços com múltiplos processos, e uma redução de aproximadamente 2 segundos no tempo de migração por cada porta exposta no contêiner ao utilizar o *warm restoration*.

O FastMig também inclui um mecanismo de tolerância a falhas configurável que analisa o código de saída do serviço e decide automaticamente como reiniciá-lo, a partir do *checkpoint*, do zero ou em modo de espera, sem necessidade de recriar o contêiner. Adicionalmente, o sistema expõe uma interface HTTP para que orquestradores externos possam solicitar operações de migração sem acesso direto ao CLI do contêiner, facilitando a integração com plataformas como o Kubernetes.

O FastMig, contudo, utiliza o método *stop-and-copy* para a transferência de memória, sem a otimização de *pre-copy* iterativo, e não oferece suporte a GPUs. O foco do trabalho é em serviços CPU-bound de propósito geral, sem abordar os requisitos de coordenação de grupo inerentes ao treinamento distribuído.

Um desafio específico diz respeito à captura do estado de aceleradores de hardware. O CRIU opera sobre memória principal; a memória da GPU não é capturada pelo mecanismo padrão. Abordagens baseadas em *device proxy*, como o Cricket (NVIDIA Corporation, 2023), interceptam, registram e reproduzem as chamadas de API do dispositivo (CUDA, ROCm) para reconstruir o estado da GPU durante a restauração. Contudo, essa abordagem introduz *overhead* de desempenho em estado estável proporcional ao número de chamadas interceptadas, e exige que as aplicações sejam compiladas com bibliotecas dinamicamente vinculadas, o que é incompatível com diversas distribuições de *frameworks* como o PyTorch.

3.1.6 CRIUgpu

Stoyanov et al. (2025) propõem o CRIUgpu, publicado em 2025, que aborda o problema da captura de estado de GPU sem recorrer à interceptação de API. O CRIUgpu estende o CRIU por meio de *plugins* carregáveis que interagem diretamente com os *drivers* de GPU no nível do *kernel*. Um *plugin* CUDA utiliza a ferramenta *cuda-checkpoint* para bloquear (*lock*), capturar e restaurar o estado de contextos CUDA de aplicações NVIDIA, enquanto um *plugin* AMD realiza operações análogas via chamadas *ioctl* sobre o KFD (*Kernel Fusion Driver*) para aplicações ROCm. O resultado é um *snapshot* unificado CPU-GPU sem necessidade de modificações nas aplicações, sem recompilação e sem *overhead* de desempenho em estado estável.

A operação de *checkpoint* com o *plugin* CUDA ocorre em três etapas coordenadas

1. o *hook* *PAUSE_DEVICES* é invocado antes do congelamento dos processos CPU, colocando todas as tarefas CUDA em estado bloqueado e interrompendo trabalho pendente na GPU;

2. o *hook CHECKPOINT_DEVICES* captura o estado de memória da GPU para a memória do *host*; e
3. o CRIU realiza o *checkpoint* convencional dos processos CPU, agora incluindo o estado de GPU serializado nos arquivos de imagem. Na restauração, a sequência é invertida: o CRIU restaura os processos CPU, e o *hook RESUME_DEVICES_LATE* reinicia a execução na GPU a partir do estado capturado.

A avaliação do CRIUgpu sobre modelos de linguagem de grande porte evidencia a viabilidade da abordagem: o *checkpoint* de um LLaMA 3.1 com 8B parâmetros em uma GPU H100 requer aproximadamente 77 segundos, e a restauração aproximadamente 39 segundos. O estado da GPU representa mais de 97% do tamanho total do *snapshot*, demonstrando a dominância da memória do acelerador no volume de dados transferidos. O trabalho demonstra ainda que os tempos de *checkpoint* e restauração escalam linearmente com o número de GPUs para cargas com paralelismo de dados em nó único.

Em resumo, a migração ao vivo de contêineres com suporte a GPU avançou significativamente com o CRIUgpu (Stoyanov et al., 2025), mas sua aplicação a cargas de aprendizado profundo distribuído em Kubernetes ainda carece de soluções que, coordenem a suspensão simultânea de múltiplos trabalhadores em diferentes nós e integrem o protocolo de migração ao ciclo de vida do escalonador Kubernetes.

3.1.7 KubeSPT

Zhang et al. (2025) propõem o KubeSPT (*Stateful Pod Teleportation*), um sistema de migração ao vivo de *Pods* com estado projetado nativamente para o Kubernetes. O trabalho parte da observação de que o Kubernetes não oferece suporte nativo à migração ao vivo de *Pods*: o mecanismo padrão de movimentação, evicção e re-escalonamento, destrói o estado do contêiner e exige reinicialização completa da aplicação.

O KubeSPT aborda duas classes de problemas técnicos. O primeiro é a sincronização de estado de rede: em Kubernetes, cada *pod* possui um *namespace* de rede próprio, gerenciado pelo CNI, que difere do *namespace* de rede do contêiner subjacente. Para sincronizar esses dois espaços de nome durante a migração, o KubeSPT introduz o componente T-Proxy, que intercepta e redireciona o tráfego entre o *namespace* do *pod* e o do contêiner, mantendo as conexões ativas durante e após o *checkpoint*. O segundo problema é a restauração eficiente de memória: ao retomar o contêiner no nó de destino, o KubeSPT aplica a técnica de *Hot Data + Lazy-Restore*, que restaura imediatamente as páginas de memória quentes, aquelas acessadas com maior frequência antes do *checkpoint* e adia a restauração das demais para o momento em que são efetivamente acessadas pelo processo, utilizando o mecanismo *userfaultfd* do kernel Linux.

Os experimentos do KubeSPT demonstram uma redução de 86 a 93% no *downtime* em relação à abordagem de evicção e recriação de *Pods*. O sistema representa a proposta mais recente e

mais próxima do cenário desta dissertação em termos de integração com o Kubernetes; porém, não aborda contêineres com aceleradores de GPU, não considera treinamento distribuído com múltiplos trabalhadores e não integra o mecanismo de migração ao ciclo de vida do escalonador.

3.2 ESCALONADORES ESPECIALIZADOS PARA APRENDIZADO DE MÁQUINA

A inadequação do escalonador genérico do Kubernetes para cargas de aprendizado de máquina motivou uma linha de pesquisa dedicada a escalonadores especializados. Os trabalhos mais influentes da área são apresentados a seguir.

3.2.1 Gandiva

Xiao et al. (2018) propõem o Gandiva, um escalonador para *clusters* de GPU que introduz duas contribuições principais: o compartilhamento temporal (*time-sharing*) de GPUs entre tarefas e a introspecção do comportamento de treinamento para guiar decisões de escalonamento.

O compartilhamento temporal é viabilizado pela capacidade de *suspend/resume* do CUDA, que permite salvar e restaurar o contexto de execução de uma tarefa na GPU sem encerrar o processo. Com isso, o Gandiva consegue multiplexar múltiplas tarefas em uma mesma GPU, aumentando a utilização do recurso e reduzindo o tempo de resposta de tarefas interativas, como buscas de hiperparâmetros, que de outra forma precisariam aguardar na fila até que uma GPU inteira ficasse disponível.

A introspecção é realizada por um perfilador embutido no escalonador, que monitora o progresso das tarefas em termos de iterações por segundo e eficiência de uso da GPU. Tarefas com baixa eficiência são candidatas a migração para outros nós do *cluster*, o que o Gandiva implementa por meio de *checkpoint* em disco seguido de restauração no nó de destino. Essa abordagem é equivalente à migração a frio: a tarefa é encerrada, seu estado é salvo em armazenamento compartilhado e reiniciada no novo nó. Não há migração ao vivo; o progresso computacional acumulado desde o último *checkpoint* periódico em disco é perdido.

3.2.2 Tiresias

Gu et al. (2019) propõem o Tiresias, um gerenciador de *cluster* de GPU para treinamento distribuído que aborda dois problemas distintos: a política de escalonamento e o *gang scheduling*.

Para a política de escalonamento, o Tiresias adapta o algoritmo *Least Attained Service* (LAS), variante do *Shortest Job First* para duração de tarefas desconhecida, ao contexto de *clusters* de GPU. Em vez de priorizar tarefas com menor tempo de CPU acumulado, o Tiresias prioriza tarefas que consumiram menos recursos de GPU até o momento, favorecendo tarefas recém-submetidas e reduzindo o tempo de conclusão médio no *cluster*. Uma extensão bidimensional do algoritmo, denominada 2DAS, leva em conta tanto o número de GPUs quanto o tempo de execução ao calcular o serviço acumulado, lidando melhor com tarefas de diferentes escalas.

Para o *gang scheduling*, o Tiresias introduz a noção de *skew*: quando um conjunto de *Pods* de um trabalho distribuído não pode ser escalonado integralmente por falta de recursos, o Tiresias adia o início do trabalho em vez de alocar apenas uma parte dos trabalhadores e manter GPUs ociosas aguardando os demais.

A preempção no Tiresias segue o modelo padrão de encerramento de contêineres: quando uma tarefa de maior prioridade (menor serviço acumulado) não encontra recursos disponíveis, tarefas de menor prioridade são encerradas e devem reiniciar do zero ou a partir de um *checkpoint* salvo periodicamente em disco. O custo dessa perda de progresso é reconhecido pelos autores como uma limitação, mas não é endereçado pela proposta.

3.2.3 Themis

Mahajan et al. (2020) abordam o problema do escalonamento de *clusters* de GPU sob a perspectiva da justiça (*fairness*). Os autores identificam que métricas tradicionais de justiça, como *max-min fairness*, são inadequadas para cargas de aprendizado de máquina porque não levam em conta a heterogeneidade dos benefícios que diferentes alocações de GPU proporcionam a diferentes tarefas.

O Themis propõe uma nova métrica denominada *finish-time fairness* (*rho-fairness*): uma alocação é considerada justa se o tempo de conclusão de cada tarefa, na presença de outras tarefas concorrentes, não excede em mais de um fator ρ o tempo que levaria se a tarefa executasse sozinha em sua fração justa de recursos. Para encontrar alocações que satisfaçam essa métrica, o Themis formula o problema como um leilão de dois estágios: as tarefas fazem lances com base em seu ganho marginal por GPU adicional, e o escalonador aloca recursos priorizando as tarefas com maior ganho relativo.

A proposta do Themis é focada em garantias de justiça entre múltiplos usuários concorrentes, não em reduzir o tempo de conclusão de tarefas individuais de alta prioridade. A preempção, quando necessária, é igualmente implementada por encerramento de contêineres e *checkpoint* em disco, sem mecanismo de migração ao vivo.

3.2.4 SpeCon

Mao et al. (2022) propõem o SpeCon (*Speculative Container Scheduling*), um escalonador de contêineres para tarefas de aprendizado profundo em Kubernetes que explora o comportamento convergente do treinamento para liberar recursos de forma preditiva.

A premissa do SpeCon é que modelos de aprendizado profundo percorrem fases distintas ao longo do treinamento: na fase inicial, a função de avaliação (*loss* ou acurácia) evolui rapidamente, caracterizando contêineres em progresso (*Progressing Containers*, PC); em seguida, a taxa de melhoria diminui e o modelo entra em platô (*Watching Containers*, WC); por fim, a convergência é atingida e as iterações subsequentes não produzem ganho relevante (*Converged Containers*, CC). O SpeCon classifica dinamicamente cada tarefa em uma dessas três categorias por meio de

um *Container Monitor* implantado como *DaemonSet* em cada nó do *cluster*. Tarefas em estado CC são candidatas a migração ou encerramento especulativo, liberando recursos para tarefas em estado PC.

A arquitetura do SpeCon compreende três componentes: o *Container Monitor*, responsável pela coleta de métricas de progresso; o *SpeCon Scheduler*, que substitui o escalonador padrão do Kubernetes e toma decisões de alocação com base na classificação de estado; e o *SpeCon Messenger*, que coordena a comunicação entre os demais componentes. Os experimentos reportados pelos autores indicam redução de 41,5% no tempo de conclusão de tarefas individuais e de 24,7% no *makespan* total do *cluster*.

3.2.5 SchedTune

Albahar et al. (2022) propõe o SchedTune, um escalonador heterogêneo de GPU para tarefas de aprendizado profundo que utiliza modelos preditivos para otimizar a alocação de recursos em *clusters* com GPUs de diferentes gerações e capacidades.

O problema central abordado pelo SchedTune é que escalonadores agnósticos à heterogeneidade de GPU frequentemente alocam tarefas em dispositivos incompatíveis com suas demandas de memória, resultando em erros de memória insuficiente (*out-of-memory*, OOM) ou em subutilização de aceleradores de alta capacidade. Para contornar isso, o SchedTune treina um modelo de *Random Forest Regressor* que prediz, a partir de características do modelo e do *dataset*, a demanda de memória de GPU e o tempo estimado de conclusão da tarefa (*Job Completion Time*, JCT) para cada tipo de GPU disponível no *cluster*. O escalonador utiliza essas previsões para atribuir cada tarefa ao acelerador que minimiza o JCT sem exceder a capacidade de memória disponível.

Os experimentos relatados pelos autores demonstram aumento de 81% na utilização de memória de GPU, eliminação completa de erros de OOM e redução de 17,5% no *makespan* em relação a escalonadores heurísticos. O SchedTune é projetado para tarefas de nó único: não considera treinamento distribuído com múltiplos trabalhadores e não implementa mecanismo de migração ou preempção com preservação de estado.

3.2.6 Viody e Kistijantoro

Viody & Kistijantoro (2022) propõe uma arquitetura de escalonamento para treinamento distribuído de aprendizado profundo em Kubernetes que integra migração de contêineres como mecanismo de rebalanceamento de carga. O trabalho é o mais diretamente relacionado à proposta desta dissertação, por ser o único da literatura a combinar explicitamente os dois eixos temáticos deste capítulo: migração de contêineres e escalonamento especializado para aprendizado profundo em Kubernetes.

A arquitetura proposta estende o operador DRAGON (Lin; Yeh; Chou, 2019), voltado à execução de tarefas distribuídas com TensorFlow usando a estratégia de servidor de parâmetros

assíncrono. Sobre essa base, os autores adicionam três componentes: suporte a tarefas de treinamento preemptíveis, marcadas com um atributo de prioridade em suas definições de recurso personalizado (CRD); um *Container Monitor DaemonSet* que coleta métricas de progresso de treinamento em cada nó; e uma política de migração que, ao detectar tarefas convergidas ou com progresso lento, decide migrar seus contêineres para nós com melhor disponibilidade de recursos ou com maior afinidade de dados, visando a rebalancear a carga do *cluster* sem interromper definitivamente o treinamento.

Os experimentos reportados indicam redução de 55,7% no tempo total de execução em relação ao DRAGON sem migração, o qual já apresentava redução de 60% no tempo total comparado ao escalonador padrão do Kubernetes. O trabalho demonstra que a combinação de monitoramento de progresso e migração de contêineres é viável em ambiente Kubernetes; porém, apresenta limitações importantes: a estratégia de servidor de parâmetros assíncrono não é compatível com a comunicação coletiva síncrona baseada em AllReduce, padrão nas implementações modernas de treinamento distribuído com PyTorch e NCCL, pois a migração não preserva o estado de GPU via *checkpoint* de driver. Além disso, o método de migração se limita ao método clássico, sem migração ao vivo. Essas lacunas são exatamente o que a proposta desta dissertação busca endereçar.

3.3 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os trabalhos relacionados nos dois eixos temáticos centrais desta dissertação.

Na área de migração ao vivo de contêineres, os trabalhos de Milojevic et al. (2000) e Clark et al. (2005) estabeleceram os fundamentos teóricos da migração de processos e da técnica de *pre-copy*, respectivamente, enquanto o CRIU (CRIU Project, 2012) consolidou a implementação de referência para sistemas Linux. O Sledge (Xu et al., 2020) estruturou a migração de contêineres Docker em três componentes independentes, imagem (LCR), estado de execução (CRIU incremental) e contexto de gerenciamento (DCL), e demonstrou que otimizar cada componente separadamente reduz o tempo total de migração em 57%. O MBDPC (Lu; Jiang, 2022) introduziu previsão de páginas sujas via *Random Forest* combinada com compressão LZ4 incremental, reduzindo o volume de dados transferidos durante o *pre-copy* em até 65%. Singh et al. (2024) complementaram essa direção com uma política de *pre-dump* seletivo baseada em probabilidade de acesso, reduzindo o tamanho do *checkpoint* inicial em 26%, e adicionaram controles de segurança via SSH e SHA-256. O CloudHopper (Benjaponpitak; Karakate; Sripanidkulchai, 2020) demonstrou a viabilidade de migrar contêineres interdependentes entre provedores de nuvem distintos. O FastMig (Manatura et al., 2024) avançou ao introduzir a técnica de *warm restoration*, reduzindo o *downtime* em até 30 vezes em relação ao FastFreeze original. O KubeSPT (Zhang et al., 2025) integrou a migração ao ciclo de vida nativo do Kubernetes por meio do T-Proxy e da restauração *hot-data + lazy-restore*, eliminando a necessidade de recriar *pods*.

Por fim, o CRIUgpu (Stoyanov et al., 2025) resolveu o problema da captura de estado de GPU sem interceptações por APIs, através de *plugins* de *driver*, viabilizando *snapshots* unificados de CPU-GPU. A lacuna remanescente nesse eixo é a ausência de suporte a migração multi-nó coordenada com comunicação coletiva NCCL e com preservação de estado de GPU integrada ao ciclo de preempção de escalonadores Kubernetes.

Na área de escalonamento especializado para aprendizado de máquina, Gandiva (Xiao et al., 2018), Tiresias (Gu et al., 2019) e Themis (Mahajan et al., 2020) representam trabalhos influentes, abordando respectivamente o compartilhamento temporal de GPU, o escalonamento por serviço acumulado e a justiça de tempo de conclusão. O SpeCon (Mao et al., 2022) introduziu o escalonamento especulativo baseado na fase de convergência do treinamento, classificando tarefas como PC, WC ou CC, e demonstrou que a migração de contêineres convergidos libera recursos de forma eficiente, a custos de não preservar o estado do processo. O SchedTune (Albahar et al., 2022) demonstrou que modelos preditivos de memória e JCT permitem alocação heterogênea de GPU sem erros de OOM, ainda que restrito a tarefas de nó único. Viody & Kistijantoro (2022) deram o passo mais próximo da proposta desta dissertação ao integrar migração de contêineres ao escalonamento distribuído em Kubernetes, porém restrito à estratégia assíncrona de servidor de parâmetros e sem preservação de estado de GPU. Nenhum dos trabalhos de escalonamento combina preempção baseada em prioridades com migração ao vivo síncrona para treinamento distribuído com AllReduce e NCCL.

O Capítulo 4 apresenta a arquitetura do escalonador proposto, a política de preempção e o protocolo de migração ao vivo integrado ao ciclo de vida do Kubernetes.

4 MIGRAÇÃO AO VIVO PARA TAREFAS DE APRENDIZADO PROFUNDO

Nos capítulos anteriores se mostrou evidente a importância do desenvolvimento de soluções melhores para escalonamento de tarefas de aprendizado profundo em ambientes baseado em contêineres. Neste capítulo está exposta a solução abordada deste trabalho, descrevendo a arquitetura do escalonador e seu funcionamento, mostrando como o escalonador age para efetuar um melhor equilíbrio de carga utilizando migração ao vivo de contêineres.

4.1 VISÃO GERAL DA ARQUITETURA

Baseando-se no desenvolvimento do projeto proposto por (Viody; Kistijantoro, 2022), este trabalho integra conceitos de escalonamento como gang-scheduling e locality-awareness, que visam uma alocação de tarefas mais eficiente; reduzindo o *overhead* de comunicação entre os contêineres, junto com a técnica migração ao vivo que proporciona uma melhor utilização de recursos durante a execução dessas tarefas, fazendo com que terminem em um menor tempo.

O escalonador proposto, chamado *dlkubeshed*, é implementado na linguagem Go, e através de sua API, estende o Kubernetes com a capacidade de migrar tarefas de aprendizado profundo distribuído entre nós do *cluster* de forma reativa ao progresso do treinamento. Ao contrário de um escalonador genérico, que decide apenas o posicionamento inicial dos *pods*, o *dlkubeshed* acompanha continuamente o estado de convergência de cada tarefa e, quando identifica o momento oportuno, aciona uma migração para equilibrar a utilização dos recursos *cluster*.

O sistema é organizado em dois processos implantados no *cluster* Kubernetes: o *processo escalonador* e o *agente CRIU*.

O **processo escalonador** é implantado como um *Deployment* no plano de controle do *cluster* e agrega quatro componentes internos, executados como *goroutines* concorrentes: o *Controlador* (*pkg/controller*), o *Escalonador* (*pkg/scheduler*), o *Monitor de Convergência* (*pkg/monitor*) e o *Migrador* (*pkg/migrator*). Esses quatro componentes compartilham um único cliente da API do Kubernetes e cooperam por meio de *callbacks* definidos no início da aplicação, sem acoplamento direto entre pacotes. Essa organização segue o padrão de *dependency injection*: a função principal instancia cada componente e registra os *callbacks* que ligam a detecção de convergência (Monitor) ao acionamento da migração (Migrador) e à atualização de estado (Controlador), de modo que nenhum pacote importa outro diretamente.

O **agente CRIU** é implantado como um *DaemonSet* em todos os nós de trabalho do *cluster* e expõe uma API HTTP simples na porta 9100 do hospedeiro. Sua única responsabilidade é receber uma requisição de *checkpoint* do Migrador, acionar a API de *checkpoint* de contêiner do Kubernetes via *proxy* do nó e copiar o arquivo produzido pelo kubelet para o nó destino, no qual os *pods* serão reiniciados.

A Figura 4 ilustra a relação entre os componentes. O eixo central de coordenação é a

API do Kubernetes: todos os componentes lêem e escrevem estado exclusivamente por meio dela, o que torna o sistema tolerante à reinicialização, já que nenhum estado crítico é mantido em memória entre ciclos de controle. Se o processo escalonador for reiniciado, ele reconstrói sua visão do mundo relistando os objetos *DeepLearningJob* e *pods* pela API, e retoma a operação a partir do estado persistido no *status* de cada recurso. Essa propriedade, herdada do modelo declarativo do Kubernetes, é fundamental para a robustez do sistema em um ambiente sujeito a falhas de nós.

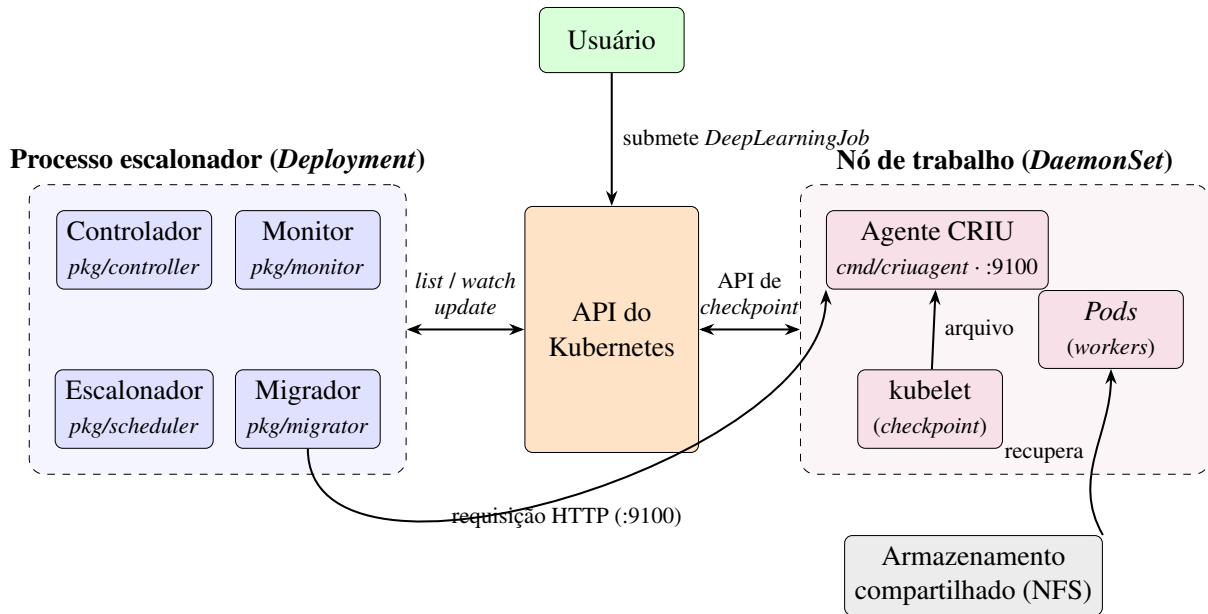


Figura 4 – Visão geral da arquitetura do *dlkubeshed*.

Fonte: Elaborado pelo autor.

A Tabela 2 resume a responsabilidade de cada componente e o pacote Go correspondente na implementação.

Tabela 2 – Componentes do *dlkubeshed* e suas responsabilidades.

Componente	Pacote	Responsabilidade principal
Controlador	<i>pkg/controller</i>	Materializar o DLJ em <i>pods</i> e gerir o ciclo de vida
Escalonador	<i>pkg/scheduler</i>	Posicionar <i>pods</i> com <i>gang scheduling</i>
Monitor	<i>pkg/monitor</i>	Detectar convergência e disparar a migração
Migrador	<i>pkg/migrator</i>	Executar a migração (a frio, ao vivo)
Agente CRIU	<i>cmd/criuagent</i>	Acionar o <i>checkpoint</i> de contêiner no nó

Fonte: Elaborado pelo autor.

4.2 RECURSO PERSONALIZADO *DEEPLARNINGJOB*

A interface primária do sistema com o usuário é o recurso personalizado (*Custom Resource*) *DeepLearningJob* (abreviado *dlj*), o qual é instalado no *cluster* antes de qualquer operação

e valida automaticamente as especificações submetidas pelos usuários contra o esquema pre-estabelecido. A adoção de um recurso personalizado, em vez de anotações em objetos nativos, permite que o usuário expresse a intenção de treinamento distribuído de forma declarativa e concisa, delegando ao Controlador a criação dos objetos Kubernetes de baixo nível.

A especificação de um *DeepLearningJob* agrupa os seguintes campos principais:

- **workers**: define o número de réplicas de trabalhadores (*replicas*), a imagem do contêiner e os recursos solicitados (CPU, memória, GPUs por réplica).
- **master** (opcional): especifica uma réplica de coordenação separada, utilizada na estratégia de servidor de parâmetros ou como nó de *rendezvous* para o PyTorch DDP.
- **placement**: controla a política de distribuição das tarefas de uma mesma tarefa pelos nós, assumindo os valores *distributed* (padrão, empacotamento por melhor ajuste) ou *colocated* (fixação de toda a tarefa em um único nó). A Seção 4.4 detalha o efeito de cada valor.
- **distributed**: configura o protocolo de comunicação coletiva (*nccl* ou *gloo*) e a porta de *rendezvous*.
- **checkpoint**: habilita o armazenamento periódico de *checkpoints* em nível de aplicação em um volume NFS e, opcionalmente, ativa a migração ao vivo via CRIU com as opções *leaveRunning* e *tcpEstablished*.
- **convergence**: define a métrica de monitoramento (*loss*, *accuracy* ou *gradientNorm*), o limiar de convergência, o tamanho da janela de suavização e a ação a executar ao detectar convergência (*migrate*, *scaleDown* ou *notify*).
- **migration**: configura o modo de migração (*none*, *cold*, *hot* ou *criugpu*), o seletor de nó alvo, o período de carência e o número máximo de migrações permitidas.

A Listagem 4.1 apresenta, de forma resumida, a definição em Go da estrutura de especificação, ilustrando o mapeamento direto entre os campos do CRD e os tipos internos do sistema.

```

1 type DeepLearningJobSpec struct {
2     Workers WorkerSpec // replicas, imagem, recursos, grupos
3     Distributed *DistributedSpec // backend coletivo e porta
4     Checkpoint CheckpointSpec // NFS + opcoes de CRIU
5     Convergence ConvergenceSpec // metrica, limiar, janela, ação
6     Migration *MigrationPolicy // modo, max. migracoes
7     Placement string // "distributed" | "colocated"
8 }

```

Listing 4.1 – Estrutura da especificação do *DeepLearningJob* (*pkg/apis/v1alpha1/types.go*).

4.2.1 Estado e Métricas de Migração

O *status* do recurso, atualizado exclusivamente pelo Controlador, registra a fase atual da tarefa (*Pending*, *Running*, *Converging*, *Migrating*, *Succeeded* ou *Failed*), os instantes de início e fim do treinamento, o número acumulado de migrações e o histórico completo de eventos de migração. Cada evento de migração é representado pela estrutura *MigrationEvent*, cujos campos foram projetados especificamente para viabilizar a avaliação experimental comparativa entre os modos de migração descrita no Capítulo 5.

A Listagem 4.2 apresenta a estrutura *MigrationEvent*. Além do *downtime* em milissegundos, cada evento registra: o modo de migração empregado; a duração e o tamanho do *checkpoint* (relevantes para os modo *hot*); a iteração de treinamento no instante da migração; o número de iterações perdidas (diferença entre a iteração corrente e a do último *checkpoint* em nível de aplicação, sempre zero para os modos ao vivo, que restauram o estado exato do processo); e o nó de destino escolhido pela política de balanceamento de carga.

```

1 type MigrationEvent struct {
2     StartTime metav1.Time // inicio da migracao
3     EndTime *metav1.Time // pod de volta em Running
4     DowntimeMs int64 // indisponibilidade (ms)
5     Mode string // cold | hot | criugpu
6     CheckpointDurationMs int64 // tempo de checkpoint (ms)
7     CheckpointBytes int64 // tamanho do snapshot
8     IterationAtMigration int64 // iteracao no disparo
9     IterationsLost int64 // iteracoes descartadas
10    TargetNode string // no escolhido para migração
11 }

```

Listing 4.2 – Estrutura de um evento de migração (*pkg/apis/v1alpha1/types.go*).

A separação entre o instante de início (*StartTime*), gravado no momento em que a migração é disparada, e o de término (*EndTime*), gravado quando o *pod* migrado retorna ao estado *Running*, permite computar o *downtime* de cada evento de forma direta. O campo *totalMigrationDowntimeMs* do *status* acumula a soma dos *downtimes* ao longo de toda a vida da tarefa, constituindo, junto ao tempo total de treinamento, a métrica primária de sobrecarga do mecanismo de migração.

4.3 CONTROLADOR

O Controlador é o componente responsável por materializar um *DeepLearningJob* em recursos Kubernetes concretos e por manter a consistência entre o estado desejado, expresso na especificação do CRD, e o estado observado, refletido nos *pods* e serviços em execução. Ele adota o padrão de projeto de *controlador de reconciliação*, padrão no ecossistema Kubernetes: em vez de executar uma sequência imperativa de ações, o Controlador compara continuamente o estado desejado com o observado e emite as operações necessárias para convergir um ao outro.

4.3.1 Reconciliação e Criação de Pods

Ao detectar a criação de um novo *DeepLearningJob* via um *informer* dinâmico, o Controlador executa a função *reconcile*, que lista os *Pods* existentes com o rótulo *dlj-name=<nome>* e cria os que estão faltando. Para cada trabalhador *i* de 0 a *replicas - 1*, é criado um *pod* com nome *<dlj>-worker-<i>*. Se o campo *master* estiver preenchido, um *pod* adicional com nome *<dlj>-master* e um *Service* de *rendezvous* são criados.

Cada *pod* é construído com os seguintes atributos que permitem ao restante do sistema identificá-lo e operá-lo:

- Rótulos *dl-workload=true* e *dlj-name=<nome>*, usados pelo Monitor e pelo Migrador para listar os *Pods* de interesse.
- Anotações *dl.scheduler/convergence-metric* e *dl.scheduler/convergence-threshold*, propagadas da especificação do DLJ para que o Monitor possa avaliar o critério de convergência sem consultar o CRD a cada iteração.
- Campo *spec.schedulerName: dlkubeshed*, que instrui o Kubernetes a não escalonar esse *pod* com o escalonador padrão, reservando-o para o componente Escalonador desta proposta.
- Variáveis de ambiente *DLJ_NAME* e *MIGRATION_MODE*, que permitem ao *script* de treinamento adaptar seu comportamento (por exemplo, restaurar de um *checkpoint* CRIU se *MIGRATION_MODE=hot* e um arquivo de imagem estiver presente no diretório *CHECKPOINT_DIR*).
- Rótulo *dlj-group=0* em todos os *Pods* quando *placement* é *colocated*, forçando o Escalonador a co-localizar toda a tarefa em um único nó (Seção 4.4).

Volumes NFS são montados automaticamente quando os campos *data.storageUri* e *checkpoint.storageUri* estiverem preenchidos, que serão utilizados para a manipulação dataset escolhido e armazenamento temporário dos checkpoints de cada contêiner.

4.3.2 Gerenciamento do Ciclo de Vida

O Controlador mantém um segundo *informer* sobre *Pods* com o rótulo *dl-workload=true* e reage a três tipos de transição:

- **Pod no estado *Running*:** se for a primeira transição da tarefa, registra o instante de início do treinamento e atualiza a fase para *Running*. Se a tarefa estava em fase *Migrating* e havia um evento de migração em andamento, o Controlador finaliza esse evento (Seção 4.3.3).
- **Pod no estado *Succeeded*:** quando todos os *Pods* da tarefa atingem essa fase, o Controlador atualiza a fase do DLJ para *Succeeded* e registra o instante de término. Como a

conclusão de uma tarefa libera os nós que ela ocupava, essa transição também dispara um rebalanceamento de carga (Seção 4.7): as tarefas em progresso reavaliam se um dos nós coincidem melhor com os requerimentos de recursos da tarefa.

- **Pod é deletado:** o Controlador chama *reconcile* novamente, recriando os *pods* faltantes. Esse mecanismo implementa a auto-restauração após uma migração: o Migrador finaliza os *pods* e o Controlador os recria; o Escalonador então os posiciona em novos nós.

4.3.3 Registro de Métricas de Migração

O ciclo de vida de um evento de migração é dividido em três momentos, refletindo a natureza assíncrona do processo. O padrão adotado, denominado *migração em andamento* (*InProgressMigration*), armazena o evento parcial no *status* do DLJ no instante do disparo e o completa quando o *pod* migrado retorna à execução.

1. **Início (*BeginMigration*).** Antes de qualquer evicção, o Controlador marca a fase do DLJ como *Migrating*, incrementa o contador de migrações, calcula o nó de destino (Seção 4.7) e grava o evento parcial em *status.inProgressMigration* com o instante de início, o modo e o nó alvo. Realizar essa gravação *antes* da evicção é essencial para evitar uma condição de corrida: caso contrário, os *pods* poderiam ser recriados antes de o nó alvo estar registrado, impedindo o Escalonador de aplicar o direcionamento.
2. **Complemento (*CompleteMigrationMetrics*).** Após o retorno do Migrador, o Controlador aplica um *merge patch* sobre o evento em andamento, adicionando as métricas coletadas durante o *checkpoint* (duração, tamanho em *bytes*, iteração no disparo e iterações perdidas), preservando os campos gravados na etapa anterior.
3. **Finalização (*handlePodRunning*).** Quando o *pod* migrado atinge *Running*, o Controlador calcula o *downtime* como a diferença entre o instante atual e o *StartTime* do evento em andamento, define o *EndTime*, anexa o evento completo ao histórico *migrationEvents*, acumula o *downtime* em *totalMigrationDowntimeMs* e retorna a fase para *Running*.

4.4 ESCALONADOR

O componente *Escalonador* implementa um módulo de escalonamento especializado que observa *pods* pendentes cujo campo *spec.schedulerName* é igual a *dlkubesched* e que ainda não foram vinculados a um nó (*spec.nodeName* vazio). Para cada *pod*, o Escalonador decide o nó de destino e emite um objeto *Binding* via API, instruindo o kubelet a iniciar o *pod* naquele nó.

4.4.1 Filtragem de Nós

O componente *NodeFilter* implementa uma etapa de filtragem que exclui nós que não estejam no estado *Ready* e nós sem GPUs disponíveis quando o *pod* solicita o recurso *nvidia.com/gpu*.

Para o cálculo dos recursos disponíveis de cada nó, subtrai-se da capacidade alocável do nó a soma das requisições de todos os *Pods* ativos nele posicionados, considerando CPU, memória e GPU. Os *Pods* nas fases *Succeeded* ou *Failed* são ignorados nesse cálculo, pois não consomem recursos.

4.4.2 Gang Scheduling e Empacotamento

Para *Pods* pertencentes a um *DeepLearningJob* (identificados pelo rótulo *dlj-name*), o Escalonador aplica a política de *gang scheduling*: todos os trabalhadores de uma mesma tarefa são escalonados simultaneamente ou nenhum deles é escalonado. É importante distinguir essa propriedade da política de *posicionamento*. O *gang scheduling* garante apenas a atomicidade da alocação, a verificação de que todos os trabalhadores cabem simultaneamente antes de vincular qualquer um deles, evitando o bloqueio mútuo de recursos que ocorreria se um subconjunto de trabalhadores fosse iniciado e mantivesse GPUs ocupadas aguardando os demais. Onde cada trabalhador é posicionado é uma decisão independente, ditada pela política de pontuação.

A política de posicionamento adotada é o *empacotamento por recurso mais alocado* (*MostAllocated* ou *bin packing*): entre os nós que ainda comportam o *Pod*, é escolhido aquele com *menor* folga de CPU e memória. Como os *Pods* de uma mesma tarefa são atribuídos sequencialmente e a disponibilidade do nó escolhido é decrementada a cada passo, os trabalhadores de uma tarefa tendem a se acumular no mesmo nó, transbordando para outro nó apenas quando o primeiro se esgota. Esse comportamento minimiza o tráfego de comunicação coletiva entre nós; a operação de *all-reduce* de gradientes, que domina o custo de rede do treinamento distribuído, ocorre localmente sempre que os trabalhadores estão co-localizados. A Listagem 4.3 apresenta o núcleo do algoritmo.

```

1 for _, pod := range pods {
2     req := podRequest(&pod)
3     bestNode := ""
4     bestScore := 0.0
5     for name, res := range avail {
6         if !fits(req, res) {
7             continue
8         }
9         // folga de CPU+memoria; menor = mais cheio.
10        // Escolhe o no mais cheio que ainda comporta o pod.
11        score := float64(res.cpu)/1000.0 +
12            float64(res.memory)/(1024*1024*1024)
13        if bestNode == "" || score < bestScore {
14            bestScore = score
15            bestNode = name
16        }
17    }
18    if bestNode == "" {

```

```

19     return nil, nil // gang nao cabe: adia tudo
20 }
21 assignments[pod.Name] = bestNode
22 avail[bestNode].cpu -= req.cpu
23 avail[bestNode].memory -= req.memory
24 avail[bestNode].gpu -= req.gpu
25 }

```

Listing 4.3 – Empacotamento de nós em *GangFit* (*pkg/scheduler/filter.go*).

4.4.3 Vinculação Direcionada por Migração

Após uma migração, o Controlador recria os *Pods* com a anotação *dl.scheduler/target-node*, contendo o nó menos utilizado escolhido pela política de balanceamento. Ao escalonar um *gang* cujos *Pods* portam essa anotação, o Escalonador tenta fixar toda a tarefa no nó indicado, desde que ele ainda comporte o conjunto completo; caso contrário, recorre ao empacotamento padrão.

4.5 MONITOR DE CRESCIMENTO

A decisão de *quando* migrar uma tarefa é delegada a um componente dedicado, o Monitor de Crescimento, cuja função é identificar as tarefas cujo treinamento entrou em regime de retornos decrescentes e que, por isso, se beneficiam menos dos recursos que ocupam. O método adotado deriva diretamente da estratégia especulativa do SpeCon (Mao et al., 2022) e de sua aplicação ao contexto de migração de contêineres por Viody & Kistijantoro (2022): em vez de observar o valor absoluto da função de perda, o monitor classifica cada tarefa segundo a *taxa de crescimento* de sua acurácia, migrando aquelas cujo crescimento se tornou lento para liberar recursos às tarefas ainda em progresso acelerado.

Seguindo Viody & Kistijantoro (2022), define-se para cada tarefa uma pontuação de progresso G dado pela taxa de variação de sua função de avaliação E , porém, no trabalho apresentado, a acurácia é medida sobre um subconjunto do conjunto de teste ao final de cada época pelo processo de *rank 0*.

Para reduzir a sensibilidade a oscilações de época para época, inerentes tanto à avaliação em subconjunto quanto ao ruído do gradiente estocástico, adota-se neste trabalho uma variante suavizada, na qual a pontuação corresponde à média do ganho de acurácia ao longo de uma janela de w épocas:

$$G(t) = \frac{E(t) - E(t - w)}{w}. \quad (1)$$

A classificação de cada tarefa depende de um limiar adaptativo α , obtido por regressão linear sobre o logaritmo do histórico da pontuação de progresso, o que equivale a modelar sua tendência como um decaimento exponencial:

$$\alpha(t) = a \cdot b^{-t}, \quad (2)$$

em que os coeficientes a e b são estimados pela regressão. A margem de tolerância β é derivada da qualidade do ajuste, medida pelo coeficiente de determinação R^2 :

$$\beta = 1 - R^2. \quad (3)$$

Com base em G , α e β , a tarefa percorre uma máquina de estados de três categorias: em progresso (*Progressing*), em observação (*Watching*) e convergida (*Converged*), de acordo com

$$\text{estado}(t) = \begin{cases} \text{em progresso,} & \text{se } G(t) > \alpha(t), \\ \text{em observação,} & \text{se } G(t) \leq \alpha(t), \\ \text{convergida,} & \text{se } G(t) (1 + \beta) < G(t - 1). \end{cases} \quad (4)$$

Uma tarefa em progresso melhora sua acurácia mais rapidamente que a tendência estimada pela Equação 2; ao cair abaixo dessa tendência, passa ao estado de observação; e, quando o escore decresce além da margem β em relação à medição anterior, é considerada convergida. As transições são monotônicas, uma tarefa nunca retorna do estado de observação para o de progresso, o que confere estabilidade à decisão e evita migrações espúrias provocadas por flutuações momentâneas da pontuação.

Uma tarefa é considerada de *crescimento lento* (*slow-growing*) quando alcança os estados de observação ou de convergência. Nesse instante, o processo de *rank 0*, o único que avalia o modelo e coordena o grupo de treinamento, registra a anotação `dl.scheduler/slow-growing=true` no próprio *pod*, juntamente com `dl.scheduler/eval-accuracy`, `dl.scheduler/progress-category` e `dl.scheduler/progress-score`. Esta última expõe o valor numérico da pontuação de progresso G da Equação 1 e é empregada, no rebalanceamento de carga (Seção 4.7), para ordenar as tarefas ativas e identificar a de progresso mais acelerado. O Monitor, ao detectar a anotação de crescimento lento, dispara duas ações: migra a tarefa de crescimento lento, deslocando-a para liberar recursos, e, em seguida, rebalanceia as tarefas de progresso para os nós com mais recursos disponíveis. A comunicação por anotações mantém o *script* de treinamento desacoplado do escalonador: toda a lógica de classificação executa junto ao treinamento, que dispõe naturalmente do histórico de avaliações, cabendo ao escalonador apenas a leitura de um valor booleano. A Listagem 4.4 resume a classificação realizada ao final de cada época.

```

1 # pontuação de progresso suavizado sobre a janela w
2 g = max(0.0, (acc - acc_hist[-1 - w]) / w)
3 # alpha(t) = a * b^-t por regressao log-linear; beta = 1 - R^2
4 alpha, beta = ajusta_regressao(hist_epocas, hist_g)
5 if estado == "em progresso" and g <= alpha:
6     estado = "em observacao"
7 if estado == "em observacao" and g_anterior is not None and g * (1 + beta

```

```

    ) < g_anterior:
8     estado = "convergida"
9     slow_growing = estado != "em progresso"

```

Listing 4.4 – Classificação de crescimento lento (*train_cpu_cifar10.py*).

4.6 MIGRADOR

O Migrador implementa duas estratégias de migração, selecionadas em tempo de execução pelo campo *migration.mode* do DLJ: migração a frio (*cold*), migração ao vivo sem GPU (*hot*). Cada uma das duas funções de migração retorna uma estrutura de métricas (*MigrationMetrics*) preenchida com os dados coletados durante a operação, que o Controlador posteriormente persiste no evento de migração.

4.6.1 Migração a Frio

Na migração a frio, o Migrador lista todos os *Pods* pertencentes ao DLJ e emite uma evicção (*Eviction*) para cada um com período de carência de 30 segundos. O kubelet aguarda o período de carência, terminando o processo do contêiner ordenadamente, e então o Controlador detecta a deleção dos *Pods* e os recria. Os novos *Pods*, ao serem iniciados, verificam a existência de um arquivo de *checkpoint* de treinamento periódico no volume NFS e restauram o andamento do treinamento a partir da última época gravada, sem qualquer interação com o CRIU. Esse modo é utilizado por (Viody; Kistijantoro, 2022) para o equilíbrio de carga de trabalho, sendo implementado aqui para comparação com o método de migração ao vivo.

4.6.2 migração ao vivo (*hot*)

Na migração ao vivo, o Migrador executa os seguintes passos em sequência:

1. Obtém o endereço IP interno do nó em que o *pod* alvo está sendo executado, consultando o objeto *Node* na API do Kubernetes.
2. Envia uma requisição HTTP *POST /checkpoint* ao agente CRIU do nó de origem, informando o *namespace*, o nome do *pod*, o nome do contêiner (*main*) e o caminho de destino da imagem CRIU no novo nó.
3. Executa-se a transferência desse *checkpoint* intermediário no nó destino, informado pelo escalonador *dlkubesched* via *rsync*.
4. Realizam-se outros *checkpoints* intermediários conforme a execução, transferindo-os para o novo nó.

5. Interrompe-se brevemente a aplicação, executa-se o *checkpoint* do restante do contêiner, ou seja, o que foi alterado desde os *checkpoints* anteriores, e realiza-se a transferência para o nó destino.

Os *pods* recriados pelo Controlador detectam, ao iniciar, a presença do arquivo *hot-<pod>.tar* em *CHECKPOINT_DIR* e chamam o comando de restauração do CRIU (*criu restore*) em vez de iniciar o processo de treinamento do zero. Como o CRIU restaura o estado exato do processo, o número de iterações perdidas nesse modo é **zero**.

Limitação de GPU no modo *hot*: o CRIU padrão não é capaz de capturar o estado da memória de dispositivos CUDA. Para tarefas com aceleradores de GPU, o *script* de treinamento deve transferir explicitamente os tensores do modelo da memória da GPU para a memória principal antes de o Monitor disparar a migração, por meio das APIs de serialização do PyTorch (*torch.save*).

A Tabela 3 compara as três estratégias de migração segundo as dimensões mais relevantes para a avaliação experimental.

Tabela 3 – Comparação entre os modos de migração.

Característica	<i>cold</i>	<i>hot</i>
Mecanismo de estado	<i>checkpoint app.</i>	CRIU (CPU)
Iterações perdidas	> 0	0
Estado de GPU	recalculado	manual
<i>Downtime</i> típico	alto	médio
Requer CRIU no nó	não	sim
Requer <i>cuda-checkpoint</i>	não	não

Fonte: Elaborado pelo autor.

4.7 BALANCEAMENTO DE CARGA POR MIGRAÇÃO

Enquanto o Monitor decide quando e qual tarefa migrar, com base na taxa de crescimento, o mecanismo de balanceamento de carga decide para onde a tarefa migra. Essa separação de responsabilidades é uma característica central do projeto: o gatilho da migração é a taxa de crescimento da tarefa, mas o destino é escolhido segundo a utilização instantânea dos nós, aproximando o comportamento do sistema ao de escalonadores voltados ao rebalanceamento de recursos.

O mecanismo distingue duas situações de migração, tratadas por caminhos de código independentes e com destinos opostos. A primeira, *migração por crescimento lento*, desloca uma tarefa que entrou em regime de retornos decrescentes para o *nó mais ocupado*, empacotando-a junto às demais e liberando os nós menos ocupados. A segunda, *rebalanceamento de carga*, é acionada sempre que recursos são liberados no *cluster*, seja porque uma tarefa concluiu o treinamento, seja porque uma tarefa de crescimento lento acabou de ser migrada para longe, e

desloca as tarefas, caso necessário, as de progresso mais acelerado, para os *nós menos utilizados*, aproveitando a folga recém-criada. As Seções seguintes detalham cada destino.

4.7.1 Definição de Nó Mais Utilizado

O nó de destino de uma migração é o *nó mais utilizado* do *cluster*. Essa escolha faz com que as tarefas que estejam convergindo mais lentamente migrem para nós mais ocupados, deixando as tarefas que estão em etapas anteriores em nós menos ocupados, fazendo com que o conjunto de tarefas terminem mais rapidamente.

A utilização de um nó é medida pela fração de recursos requisitados em relação à capacidade alocável, seguindo a formulação de pontuação de nós de trabalhos correlatos. A pontuação de utilização de um nó n é definida como

$$U(n) = \frac{1}{2} \cdot \frac{\text{CPU}_{\text{req}}(n)}{\text{CPU}_{\text{aloc}}(n)} + \frac{1}{2} \cdot \frac{\text{Mem}_{\text{req}}(n)}{\text{Mem}_{\text{aloc}}(n)}, \quad (5)$$

em que $\text{CPU}_{\text{req}}(n)$ e $\text{Mem}_{\text{req}}(n)$ são as somas das requisições de CPU e memória dos *Pods* ativos no nó, e $\text{CPU}_{\text{aloc}}(n)$ e $\text{Mem}_{\text{aloc}}(n)$ são as respectivas capacidades alocáveis. A pontuação $U(n)$ pertence ao intervalo $[0, 1]$, sendo zero para um nó ocioso e próxima de um para um nó saturado. O nó escolhido é aquele de maior $U(n)$ dentre os que ainda comportam a requisição do *pod* e são distintos do nó de origem.

É importante notar a distinção entre *menor utilização* e *maior folga absoluta*. As duas medidas coincidem apenas em *clusters* homogêneos; em *clusters* com nós de capacidades distintas, a Equação 5 normaliza a ocupação pela capacidade de cada nó, escolhendo corretamente o nó proporcionalmente mais livre. A Listagem 4.5 apresenta o cálculo.

```

1 func (f *NodeFilter) MostUsedNodeForPod(ctx context.Context,
2     pod *corev1.Pod, excludeNode string) (string, error) {
3     req := podRequest(pod)
4     best := ""
5     bestScore := -1.0 // U(n) >= 0; inicia abaixo
6     for i := range nodes.Items {
7         n := &nodes.Items[i]
8         if n.Name == excludeNode || !isNodeReady(n) {
9             continue
10        }
11        avail := availableOnNode(n, allPods.Items)
12        if !fits(req, &avail) {
13            continue
14        }
15        if s := usageScore(n, allPods.Items); s > bestScore {
16            bestScore = s
17            best = n.Name
18        }
19    }

```

```

20     return best, nil
21 }

```

Listing 4.5 – Seleção do nó mais utilizado (*pkg/scheduler/filter.go*).

4.7.2 Rebalanceamento para o Nó Menos Utilizado

O rebalanceamento de carga emprega o critério inverso ao da migração por crescimento lento: seu destino é o *nó menos utilizado* do *cluster*, isto é, aquele de menor pontuação $U(n)$ (Equação 5) dentre os que ainda comportam a requisição do *pod*. Ele é acionado em dois momentos em que capacidade é liberada: quando uma tarefa atinge a fase *Succeeded* (Seção 4.3.2) e quando uma tarefa de crescimento lento é migrada para longe, desocupando o nó em que estava. A Listagem 4.6 apresenta a seleção do destino, simétrica à do nó mais utilizado.

```

1 func (f *NodeFilter) LeastUsedNodeForPod(ctx context.Context,
2     pod *corev1.Pod, excludeNode string) (string, error) {
3     req := podRequest(pod)
4     best := ""
5     bestScore := 2.0 // U(n) <= 1; inicia acima
6     for i := range nodes.Items {
7         n := &nodes.Items[i]
8         if n.Name == excludeNode || !isNodeReady(n) {
9             continue
10        }
11        avail := availableOnNode(n, allPods.Items)
12        if !fits(req, &avail) {
13            continue
14        }
15        if s := usageScore(n, allPods.Items); s < bestScore {
16            bestScore = s
17            best = n.Name
18        }
19    }
20    return best, nil
21 }

```

Listing 4.6 – Seleção do nó menos utilizado (*pkg/scheduler/filter.go*).

Diferentemente da migração por crescimento lento, que desloca a própria tarefa que desacelerou, o rebalanceamento pode mover vários nós por evento, dependendo. Para n pois cada migração impõe um custo de *checkpoint* e de *downtime*, e mover todas as tarefas ativas a cada liberação de capacidade provocaria uma oscilação excessiva de posicionamento. As tarefas escolhidas são as de maior pontuação de progresso G , lida da anotação *dl.scheduler/progress-score* publicada pelo processo de *rank* 0 (Seção 4.5); descartam-se as tarefas de crescimento lento, tratadas pelo outro caminho, e a própria tarefa que disparou o evento, para que ela não seja reco-

locada no nó que acabou de liberar. Como apenas o processo de *rank 0* publica essas anotações, a busca percorre naturalmente um representante por tarefa; devolver esse *pod* é suficiente para conduzir a migração, pois o direcionamento é gravado no *status* do DLJ e aplicado a todo o grupo recriado (Seção 4.4.3).

4.7.3 Interação entre Empacotamento e Balanceamento

As políticas de empacotamento (posicionamento inicial) e balanceamento (destino da migração) empregam critérios opostos, sendo eles o nó mais cheio e o nó menos utilizado. O empacotamento concentra as tarefas em poucos nós no início, reduzindo o *downtime* de comunicação entre as tarefas; o balanceamento posteriormente redistribui a carga em dois sentidos complementares: empurra as tarefas de crescimento lento para o nó mais utilizado, agrupando-as junto às demais, e puxa a tarefa de progresso mais acelerado para o nó menos utilizado assim que capacidade é liberada, tipicamente um nó cuja tarefa já concluiu o treinamento. O resultado é um sistema que empacota agressivamente para maximizar a localidade de comunicação e, à medida que as tarefas desaceleram ou terminam, migra as remanescentes para aproveitar a capacidade liberada, mantendo alta a utilização do *cluster* sem sacrificar a localidade.

4.8 AGENTE CRIU

O agente CRIU é implantado como um *DaemonSet* no *namespace kube-system*, com tolerâncias que permitem sua execução em todos os nós do *cluster*. Ele expõe dois *endpoints* HTTP: */health* (sonda de vivacidade do Kubernetes) e */checkpoint* (utilizado pelo modo *hot*).

O fluxo de */checkpoint* é composto por três passos fundamentais:

1. Aciona a API de *checkpoint* de contêiner do Kubernetes via *proxy* do nó. Essa API instrui o kubelet a acionar o *runtime* de contêiner para executar um *checkpoint* intermediário do processo principal do contêiner, produzindo um arquivo tar em */var/lib/kubelet/checkpoints/* no nó.
2. Executa-se a transferência desse *checkpoint* intermediário no nó destino, informado pelo *dlkubesched* via *rsync*.
3. Realizam-se *checkpoints* intermediários adicionais durante a execução, os quais são transferidos para o novo nó.
4. Interrompe-se a aplicação, realiza-se o *checkpoint* do restante do contêiner, ou seja, o que foi alterado desde os *checkpoints* anteriores e realiza-se a transferência para o nó destino.

Os requisitos de infraestrutura por nó para os modos *hot* e *criugpu* são: Kubernetes 1.30 ou superior, containerd 1.7 ou superior com suporte a CRIU habilitado na configuração do *runtime*, e o binário *criu* instalado no sistema operacional hospedeiro.

4.9 FLUXO DE EXECUÇÃO

Para consolidar a descrição dos componentes individuais, esta seção apresenta o fluxo completo de execução de uma tarefa de aprendizado profundo distribuído no *dlkubeshed*, desde o envio pelo usuário até o término do treinamento.

(1) Submissão. O usuário aplica um manifesto *DeepLearningJob* ao *cluster* via *kubectl apply*. O servidor de API do Kubernetes valida o manifesto contra o esquema OpenAPI da CRD e o aloca no *etcd*.

(2) Criação de pods. O Controlador detecta o novo objeto via o *informer* dinâmico e executa *reconcile*: cria os *pods* de trabalhadores, o *pod* do mestre e os serviços de *rendezvous*. Os *pods* são criados com *spec.schedulerName: dlkubeshed* e ficam em estado *Pending*, aguardando o Escalonador.

(3) Gang scheduling. O Escalonador detecta os novos *pods* pendentes e identifica que pertencem a um DLJ pelo rótulo *dlj-name*. Adquire o *lock* de *gang* e executa o *GangScheduling*: se todos os trabalhadores couberem simultaneamente nos nós disponíveis, emite um *Binding* para cada um, empacotando-os no menor número de nós possível; caso contrário, adia o escalonamento até que recursos suficientes estejam disponíveis.

(4) Início do treinamento. O kubelet de cada nó inicia o contêiner. O *entrypoint* verifica a presença de um arquivo de imagem CRIU em *CHECKPOINT_DIR*: se presente, executa *criu restore* (modos *hot/criugpu*); se ausente, inicia o treinamento do zero ou retoma a partir do *checkpoint* periódico em nível de aplicação. O Controlador detecta as transições para *Running* e registra o instante de início do treinamento no *status* do DLJ.

(5) Treinamento e monitoramento. O *script* de treinamento escreve a variação da métrica de avaliação na anotação *dl.scheduler/loss-delta* do *pod* após cada época, além de atualizar o contador de iterações. O Monitor de Convergência percorre os *pods* a cada 30 segundos e compara o delta com o limiar do DLJ.

(6) Detecção de convergência e migração. Ao detectar que *loss-delta* entrou na janela de migração antecipada (*OnAlmostConverging*) ou atingiu o limiar (*OnConverging*), o Monitor dispara *triggerMigration*. A função consulta o modo de migração do DLJ, marca a fase como *Migrating*, calcula o nó mais utilizado e grava o evento parcial (*BeginMigration*); em seguida, aciona a estratégia de migração correspondente:

- No modo *cold*: o Migrador expulsa todos os *pods* do DLJ; o Controlador os recria com a anotação de nó alvo; o Escalonador os reposiciona no nó mais utilizado; o treinamento retoma do último *checkpoint* periódico.
- No modo *hot*: o Migrador chama o agente CRIU via */checkpoint* no nó de origem, salva o *snapshot* parcial do processo e começa enviar o arquivo para o nó destino, quando acaba a transferência do *checkpoint* intermediário, realiza-se o *checkpoint final* e envia-o para o

novo nó, encerrando os *Pods* de execução; os novos *Pods* restauram o processo via *criu restore*.

(7) Resumo e conclusão. Quando os *Pods* recriados atingem o estado *Running*, o Controlador calcula o *downtime* da migração, completa o evento com as métricas de *checkpoint* e o anexa ao histórico, retornando a fase para *Running*. Quando todos os trabalhadores atingem *Succeeded*, o Controlador registra o término e atualiza a fase para *Succeeded*.

4.10 CONSIDERAÇÕES FINAIS

Este capítulo descreveu a arquitetura do *dlkubeshed*, composto por dois binários, o processo escalonador e o agente CRIU que interagem exclusivamente via API do Kubernetes para orquestrar o ciclo de vida de tarefas de aprendizado profundo distribuído.

Os quatro componentes internos do processo escalonador, Controlador, Escalonador, Monitor de Convergência e Migrador, operam de forma desacoplada, compartilhando apenas o cliente da API do Kubernetes e trocando informações por meio de *callbacks* definidos na função principal. O Controlador materializa o DLJ em *Pods* e mantém o ciclo de vida, incluindo o registro em três etapas das métricas de migração; o Escalonador garante a regra de *gang scheduling* com posicionamento por empacotamento. O Monitor detecta convergência por anotações de *pod* e dispara a migração; o Migrador executa um dos dois modos (*cold* ou *hot*) conforme a política do DLJ; e o agente CRIU delega o *checkpoint* ao kubelet via API nativa do Kubernetes, com pré-validação da presença de *cuda-checkpoint* no modo *criugpu*.

Duas contribuições distinguem o sistema dos trabalhos anteriores. A primeira é o balanceamento de carga acoplado à convergência; o gatilho da migração é o estado de treinamento da tarefa, mas o destino é o nó mais utilizado do *cluster*, combinando a detecção de oportunidade com o rebalanceamento de recursos. A segunda é a quase ininterrupta execução das tarefas de aprendizado, que não precisa recuperar *checkpoints* de épocas anteriores para continuar a execução.

O Capítulo 5 descreve a metodologia de avaliação experimental e os resultados obtidos na comparação entre os dois métodos de migração e entre o *dlkubeshed* e um escalonador sem migração.

5 VALIDAÇÃO DE RESULTADOS

Este capítulo tem por objetivo avaliar empiricamente a eficácia do *dlkubesch*, verificando em que medida a migração de contêineres acionada pela detecção de crescimento lento reduz o tempo total de treinamento de cargas concorrentes sem prejudicar a qualidade dos modelos. A avaliação é organizada em torno de uma comparação central entre três configurações do sistema, **sem migração**, com **migração a frio** (*cold*) e com **migração ao vivo** (*hot*, baseada em CRIU).

A metodologia segue a linha adotada por Viody & Kistijantoro (2022), que quantifica o benefício da migração pela redução no tempo total de execução de um conjunto de tarefas concorrentes. Contudo, amplia-se aqui a análise para contemplar o *downtime* de cada migração e o número de iterações descartadas, dimensões que distinguem a migração a frio da migração ao vivo e que constituem a principal contribuição experimental deste trabalho.

5.1 PERGUNTAS DE PESQUISA E HIPÓTESES

A validação é guiada pelas seguintes perguntas de pesquisa (PP) e pelas hipóteses (H) correspondentes, cada uma associada a um ou mais experimentos das seções seguintes.

- **PP1 — Eficácia.** A migração reduz o tempo total de treinamento de cargas concorrentes em relação à ausência de migração?
 - **H1.** Sob contenção de recursos, tanto a migração a frio quanto a ao vivo reduzem o *makespan* do conjunto de tarefas, com vantagem para a migração ao vivo.
- **PP2 — Custo da preservação de estado.** Qual a diferença de custo entre migrar a frio e migrar diretamente?
 - **H2.** A migração ao vivo apresenta *downtime* menor e descarta zero iterações, ao custo de um *overhead* de *checkpoint* (tamanho e duração da imagem CRIU) maior que na migração a frio.
- **PP3 — Sensibilidade ao desbalanceamento.** O benefício da migração depende do grau de desequilíbrio inicial da carga?
 - **H3.** O ganho da migração cresce com o desbalanceamento inicial e tende a zero em cargas já equilibradas.
- **PP4 — Custo em regime favorável.** Qual o *overhead* da migração quando ela não é necessária?
 - **H4.** Em cargas sem contenção, a migração introduz uma pequena perda de desempenho, atribuível ao custo de avaliação e de *checkpoint*, sem benefício compensatório.

5.2 AMBIENTE EXPERIMENTAL

Os experimentos foram executados em um *cluster* Kubernetes local, composto por três nós, cujas especificações constam da Tabela 4.

Tabela 4 – Especificação dos nós do *cluster* experimental.

Qtd.	CPU	Memória	GPU
3	Intel i7-3770	16 GB	Gráficos integrados

Fonte: Elaborado pelo autor.

O *software* de base e suas versões estão resumidos na Tabela 5. Todos os nós compartilham um sistema de arquivos em rede (NFS) que hospeda o *script* de treinamento, o *dataset* e o diretório de *checkpoints*, requisito para que ao utilizar migração a frio, as tarefas retomem a partir do último *checkpoint* realizado.

Tabela 5 – Componentes de *software* do ambiente experimental.

Componente	Versão / configuração
Distribuição Kubernetes	K8s v1.36
<i>Runtime</i> de contêiner	(com <i>feature gate ContainerCheckpoint</i>)
CRIU	4.1.1
PyTorch	2.11.0
<i>Backend</i> de comunicação	gloo (CPU)

Fonte: Elaborado pelo autor.

5.3 CARGA DE TRABALHO

As cargas de trabalho consistem em um cenário de competição entre tarefas concorrentes. Cada tarefa treina uma rede ResNet-18 sobre o conjunto CIFAR-10. A escolha desse modelo e dataset privilegia a reprodutibilidade e um tempo de época tratável em CPU; o modelo é suficientemente representativo para exercitar o custo de sincronização de gradientes entre nós, fator que, como discutido adiante, interage diretamente com as decisões de posicionamento e migração. O número de épocas é fixado em 30 para todas as execuções, e o gatilho de migração é a detecção de crescimento lento descrita na Seção 4.5.

5.4 CENÁRIOS DE IMPLANTAÇÃO (*SETUPS*)

Para exercitar o *dlkubesched* sob diferentes padrões de concorrência, contenção e distribuição inicial de carga, definem-se cinco cenários de implantação, que variam o número de tarefas, o número de réplicas por tarefa (grau de paralelismo do treinamento distribuído), as requisições e limites de CPU e memória e a colocação inicial das réplicas entre os nós. As Tabelas 6–10

detalham a especificação de cada cenário, tarefa a tarefa; os valores de memória são expressos em gibibytes e os de CPU em núcleos virtuais, e as requisições correspondem à reserva garantida a cada réplica, ao passo que os limites definem o teto de consumo.

5.4.1 Setup 1 - Contenção máxima em nó único.

Cinco tarefas de uma única réplica, homogêneas em recursos, são inicialmente empacotadas no mesmo nó (*node1*), como detalha a Tabela 6. O cenário maximiza a contenção de CPU e memória sobre um único nó, constituindo o caso mais favorável à migração: há forte desequilíbrio inicial e capacidade ociosa nos demais nós a ser reaproveitada.

Tabela 6 – Especificação do Setup 1.

Parâmetro	Job1	Job2	Job3	Job4	Job5
Réplicas	1	1	1	1	1
Req. memória (GB)	2	2	2	2	2
Lim. memória (GB)	4	4	4	4	4
Req. CPU (núcleos)	1	1	1	1	1
Lim. CPU (núcleos)	2	2	2	2	2
Nó escolhido	<i>node1</i>	<i>node1</i>	<i>node1</i>	<i>node1</i>	<i>node1</i>

Fonte: Elaborado pelo autor.

5.4.2 Setup 2 - Treinamento distribuído concentrado.

Três tarefas, cada uma com três réplicas (nove *pods* ao todo), com requisições de recurso reduzidas por réplica (0,75 núcleo e 1,5 GB), todas colocadas em *node1* (Tabela 7). Avalia o comportamento do sistema quando cada tarefa é, em si, um trabalho distribuído (*AllReduce* entre réplicas), medindo o efeito da migração sobre grupos de *pods* escalonados em conjunto.

Tabela 7 – Especificação do Setup 2.

Parâmetro	Job1	Job2	Job3
Réplicas	3	3	3
Req. memória (GB)	1,5	1,5	1,5
Lim. memória (GB)	4	4	4
Req. CPU (núcleos)	0,75	0,75	0,75
Lim. CPU (núcleos)	2	2	2
Nó escolhido	<i>node1</i>	<i>node1</i>	<i>node1</i>

Fonte: Elaborado pelo autor.

5.4.3 Setup 3 - Desbalanceamento moderado.

Três tarefas com números decrescentes de réplicas (3, 2 e 2) e colocação inicial parcialmente distribuída (Tabela 8). Representa um desequilíbrio intermediário entre a concentração total dos *setups* 1 e 2 e a distribuição equilibrada dos *setups* 4 e 5, permitindo observar ganhos de migração em um regime realista de carga assimétrica.

Tabela 8 – Especificação do Setup 3.

Parâmetro	Job1	Job2	Job3
Réplicas	3	2	2
Req. memória (GB)	2	2	2
Lim. memória (GB)	4	4	4
Req. CPU (núcleos)	1	1	1
Lim. CPU (núcleos)	2	2	2
Nó escolhido	<i>node1</i>	<i>node1</i>	<i>node2</i>

Fonte: Elaborado pelo autor.

5.4.4 Setup 4 - Colocação equilibrada.

Três tarefas de uma única réplica, distribuídas uma por nó (Tabela 9). Serve de *baseline* de carga balanceada: sem contenção inicial relevante, espera-se que a migração ofereça benefício marginal ou nulo, caracterizando o regime em que ela é contraindicada.

Tabela 9 – Especificação do Setup 4.

Parâmetro	Job1	Job2	Job3
Réplicas	1	1	1
Req. memória (GB)	2	2	2
Lim. memória (GB)	4	4	4
Req. CPU (núcleos)	1	1	1
Lim. CPU (núcleos)	2	2	2
Nó escolhido	<i>node1</i>	<i>node2</i>	<i>node3</i>

Fonte: Elaborado pelo autor.

5.4.5 Setup 5 - Colocação equilibrada distribuída.

Reproduz a especificação de recursos e a colocação inicial do Setup 4 (Tabela 10) mas de forma distribuída, servindo de cenário de controle para aferir a repetibilidade das medições em um contexto com vários contêineres por tarefa.

Cada um desses cenários é combinado com as configurações do sistema descritas na Seção 5.2, de modo que a colocação inicial (equilibrada ou desequilibrada) e o mecanismo de migração (ausente, a frio ou ao vivo) variem de forma controlada para uma melhor análise de resultados.

Tabela 10 – Especificação do *Setup 5*.

Parâmetro	Job1	Job2	Job3
Réplicas	3	3	3
Req. memória (GB)	2	2	2
Lim. memória (GB)	4	4	4
Req. CPU (núcleos)	1	1	1
Lim. CPU (núcleos)	2	2	2
Nó escolhido	<i>node1</i>	<i>node2</i>	<i>node3</i>

Fonte: Elaborado pelo autor.

5.5 MÉTRICAS DE AVALIAÇÃO

As métricas a seguir são extraídas do campo *status* do recurso *DeepLearningJob*, populado pelo Controlador ao longo da execução, e agregadas pelo utilitário de medição.

- Tempo total do lote (*makespan*). Principal métrica de eficácia, definido como o intervalo entre o início da primeira tarefa e o término da última:

$$M = \max_i (\text{fim}_i) - \min_i (\text{início}_i). \quad (1)$$

- Aceleração (*speedup*). Redução percentual do tempo em relação a uma configuração de referência T_{ref} (o *baseline* sem migração):

$$S = \frac{T_{\text{ref}} - T_{\text{método}}}{T_{\text{ref}}} \times 100\%. \quad (2)$$

- Tempo de indisponibilidade (*downtime*). Média, por migração, das janelas de indisponibilidade de cada operação, lida de *migrationEvents[].downtimeMs*.
- Iterações descartadas. Número de iterações de treinamento perdidas por migração. Para a migração a frio, corresponde à distância entre a iteração no instante da migração e a do último *checkpoint*:

$$\text{iter}_{\text{perdidas}} = \text{iter}_{\text{migração}} - \text{iter}_{\text{checkpoint}}. \quad (3)$$

Para a migração ao vivo, esse valor é nulo por construção, pois o CRIU restaura o estado exato do processo.

- Custo de *checkpoint*. Duração (*checkpointDurationMs*) e tamanho (*checkpointBytes*) da imagem gerada.

- Utilização de recursos. Fração de CPU e memória em uso por nó ao longo do tempo, amostrada periodicamente, usada para evidenciar o rebalanceamento promovido pela migração.

5.6 PROTOCOLO EXPERIMENTAL

Para garantir a validade estatística dos resultados, cada configuração de cada experimento é executada, no mínimo, três vezes, e os resultados são reportados como média acompanhada do desvio-padrão. As condições iniciais, a imagem de contêiner, versão do modelo, *dataset*, número de épocas e recursos requisitados, são idênticas entre as configurações comparadas em um mesmo experimento. Entre execuções, os *checkpoints* residuais são removidos do NFS, de modo que nenhuma execução reaproveite o estado de outra. O *speedup* é sempre calculado em relação ao escalonador padrão sem migração (Equação 2), o que isola o efeito da migração do efeito do escalonamento especializado.

5.7 ANÁLISE DE MAKESPAN

5.7.1 Objetivo.

Avaliar a hipótese H1, quantificando a redução de *makespan* proporcionada pela migração a frio e ao vivo em relação à ausência de migração, sob contenção de recursos.

5.7.2 Configuração.

As configurações apresentadas anteriormente são executadas no *cluster* três vezes por escalonador, utilizando como base de teste o escalonador sem migração.

5.7.3 Resultados.

A Tabela 11 apresenta o *makespan* médio (acompanhado do desvio-padrão) de cada cenário de implantação (Seção 5.4) sob as três configurações, sem migração, migração a frio e migração ao vivo, bem como o *speedup* de cada estratégia de migração em relação à execução sem migração.

5.7.4 Análise.

Os resultados confirmam H1 nos cenários com contenção de recursos: nos *setups* 1 a 3, ambas as estratégias de migração reduzem o *makespan* em relação ao teste base sem migração, e a migração ao vivo apresenta sistematicamente o menor tempo total, por não descartar iterações e por um *downtime* reduzido. O ganho é máximo no *Setup* 1 (contenção máxima em nó único), com *speedup* de 37,6% a frio e 46,7% ao vivo, e decresce à medida que a colocação inicial se torna menos desequilibrada (22,8% e 30,7% no *Setup* 3), antecipando o comportamento

Tabela 11 – *Makespan* (média $\pm$ desvio-padrão, em segundos) e *speedup* por cenário.

Cenário	<i>Makespan</i> (s)			<i>Speedup</i> (%)	
	Sem migração	A frio	Ao Vivo	Frio	Ao Vivo
<i>Setup 1</i>	23528,57 $\pm$ 1554	14682,14 $\pm$ 436	12547,14 $\pm$ 534	37,59	46,67
<i>Setup 2</i>	26304,45 $\pm$ 2340	16274,31 $\pm$ 472	15035,64 $\pm$ 486	38,16	42,83
<i>Setup 3</i>	20546,24 $\pm$ 758	15853,45 $\pm$ 240	14247,39 $\pm$ 455	22,84	30,65
<i>Setup 4</i>	10262,76 $\pm$ 241	10935,88 $\pm$ 186	10564,64 $\pm$ 256	−6,54	−2,94
<i>Setup 5</i>	9245,27 $\pm$ 105	9832,35 $\pm$ 176	9456,52 $\pm$ 120	−6,37	−2,28

Fonte: Elaborado pelo autor.

investigado no Experimento 2. Já nos *setups* 4 e 5, de colocação equilibrada, a migração produz *speedup* negativo (entre $-2,3\%$ e $-6,5\%$), evidenciando o *overhead* do mecanismo quando não há desequilíbrio a corrigir, resultado consistente com H4 e explorado nas próximas seções. Em todos os cenários, a diferença entre a migração ao vivo e a a frio quantifica o custo das iterações descartadas por esta última e a diferença de *downtime*. A comparação com o resultado de Viody & Kistijantoro (2022), que reportou redução de 55,7% em *cluster* homogêneo com servidor de parâmetros assíncrono, permite situar o ganho obtido em um cenário de comunicação coletiva síncrona e preservação de estado de processo.

5.8 DECOMPOSIÇÃO DO DOWNTIME

5.8.1 Objetivo.

Avaliar H2, contrastando o custo por migração das estratégias a frio e ao vivo.

5.8.2 Configuração.

A partir das execuções nas diferentes configurações, registram-se, para cada evento de migração, a duração e o tamanho do *checkpoint*, a janela de indisponibilidade e o número de iterações descartadas.

5.8.3 Resultados.

A Tabela 12 compara as duas estratégias por evento de migração.

5.8.4 Análise.

A hipótese H2 prevê *downtime* e iterações descartadas menores para a migração ao vivo, ao custo de um *overhead* de *checkpoint* inexistente na migração a frio. Esta é a principal distinção experimental entre as duas estratégias e o argumento central em favor da preservação de estado via CRIU: a migração a frio troca simplicidade por perda de trabalho, enquanto a migração ao vivo troca *overhead* de *checkpoint* por continuidade exata do treinamento. Além disso,

Tabela 12 – Custo por migração por tarefa: a frio *versus* ao vivo (Experimento 4).

Métrica por migração	A frio	Ao Vivo
<i>Downtime</i> (s)	5-7s	1-3s
Duração do <i>checkpoint</i> (s)	2	15
Tamanho do <i>checkpoint</i> (MB)	105	730
Iterações descartadas	370	0

Fonte: Elaborado pelo autor.

como observado na tabela, apesar do *checkpoint* da migração ao vivo ser 7 vezes maior que na migração a frio, o *downtime* permanece sendo menor do que utilizando migração a frio por causa da utilização de *checkpoints* incrementais.

É importante frisar que o *downtime* do método de migração a frio condiz somente com a inicialização do contêiner, não contando com o tempo que o processo de treinamento precisa para retomar do *checkpoint* salvo; diferentemente do *downtime* na migração ao vivo, que condiz com o tempo que o contêiner parou na máquina fonte e continuou na máquina destino.

5.9 EXPERIMENTO 3 - UTILIZAÇÃO DE RECURSOS

5.9.1 Objetivo.

Evidenciar visualmente o rebalanceamento promovido pela migração.

5.9.2 Configuração.

Durante as execuções, a utilização da CPU é amostrada periodicamente e plotada ao longo do tempo, para as configurações sem migração e com migração.

5.9.3 Resultados.

Os gráficos apresentados em Figura 5 até Figura 9 traçam a CPU requisitada (em núcleos) em cada um dos três nós ao longo do tempo, para os cinco cenários. Cada nó é representado por uma região translúcida de cor distinta, traçada da base até o número de núcleos que requisita; onde as regiões se sobrepõem, as cores se combinam, permanecendo cada uma visível pela respectiva borda: nos *setups* 1 a 3 a região do *node1* domina antes da migração e recua a uma faixa equilibrada, compartilhada com *node2* e *node3*, depois dela, ao passo que os *setups* 4 e 5, já equilibrados na colocação inicial, permanecem constantes, caso em que a migração é contraindicada.

5.9.4 Análise.

Com a migração, logo que uma das tarefas apresente um crescimento desacelerado, todos os nós irão realizar um processo de migração, fazendo com que sistemas desequilibrados sejam

Setup 1

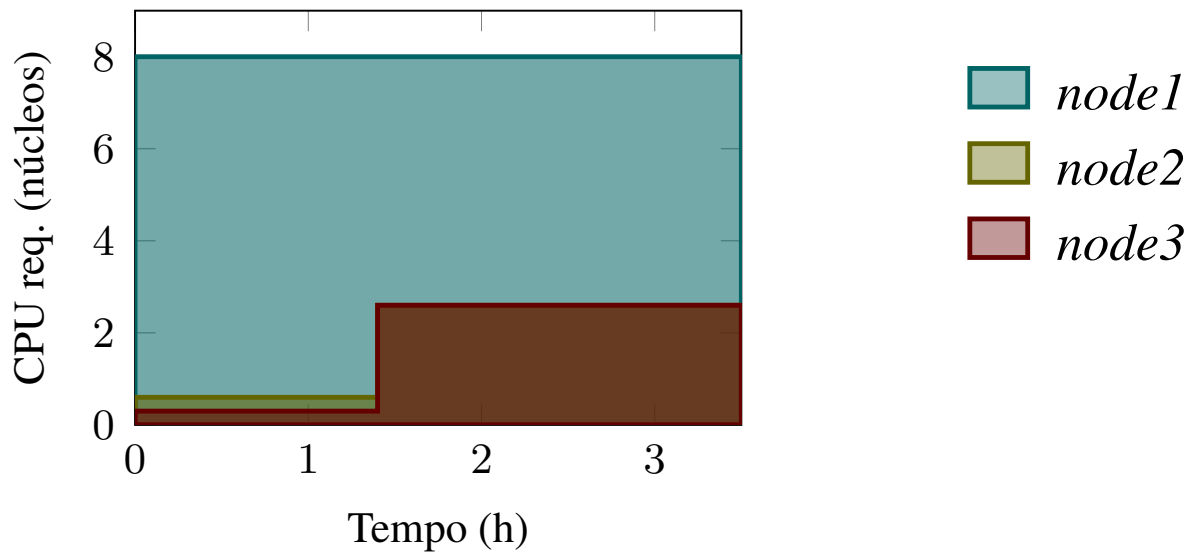


Figura 5 – Utilização de CPU por nó ao longo da duração dos testes realizados no setup 1, utilizando migração ao vivo.

Fonte: Elaborado pelo autor.

melhor utilizados ao passar das épocas. Nos casos das configurações já equilibradas, os nós não precisam sofrer migrações, fazendo com que o método seja ineficiente.

5.10 DISCUSSÃO GERAL

A Tabela 13 sintetiza a associação entre hipóteses, experimentos e o veredito esperado, servindo de guia para a interpretação conjunta dos resultados.

Tabela 13 – Síntese de hipóteses, experimentos e resultados.

Hipótese	Experimento(s)	Veredito
H1 (eficácia)	E1	O método reduz ainda mais o <i>makespan</i> da tarefas.
H2 (custo cold vs hot)	E2	O método reduz o <i>downtime</i> de migração.
H3 (desbalanceamento)	E2	O método funciona em configurações mais equilibradas.
H4 (<i>overhead</i>)	E3	O método sofre menos em configurações homogêneas.

Fonte: Elaborado pelo autor.

O método de migração ao vivo demonstrou ganhos de *speedup* de até 46% em comparação ao controlador sem nenhum tipo de migração, e até 17% de *speedup* em comparação ao método de migração a frio. No quesito custo, a migração ao vivo, embora tenha um *overhead* maior de *checkpoints* em comparação ao método de migração a frio, gera um *downtime* inferior. Além disso, devido ao funcionamento da migração ao vivo, em que o processo continua exatamente de onde parou, não ocorrem iterações perdidas, o que acelera ainda mais o progresso de aprendizado.

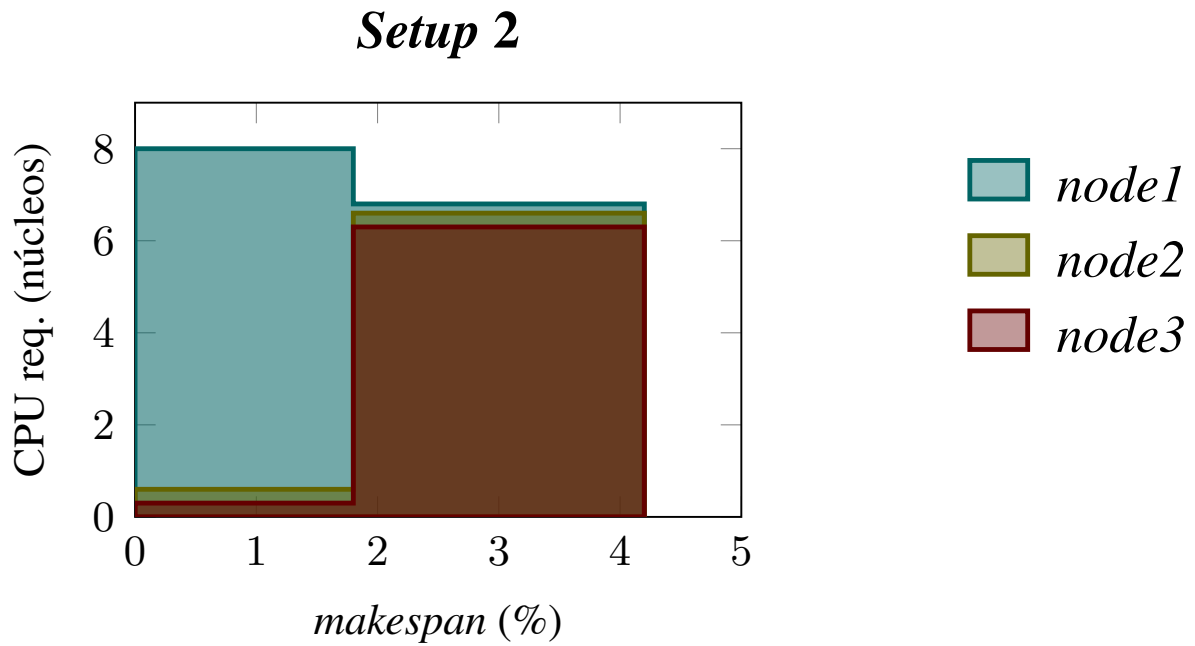


Figura 6 – Utilização de CPU por nó ao longo da duração dos testes realizados no setup 2, utilizando migração ao vivo.

Fonte: Elaborado pelo autor.

Os resultados da configuração pouco desequilibrada mostram que o método consegue ser efetivo em situações não ótimas, prevalecendo sobre o método de migração a frio. Contudo, os testes em *setups* homogêneos mostram que a migração ao vivo possui um *overhead* menor que a migração a frio, já que as migrações são mais eficientes, mesmo que redundantes.

5.11 CONSIDERAÇÕES FINAIS

Este capítulo propôs um protocolo de validação organizado em torno de uma comparação central entre ausência de migração, migração a frio e migração ao vivo, complementada por experimentos que isolam o desbalanceamento inicial, o custo do mecanismo e a utilização de recursos. O conjunto de métricas, *makespan*, *speedup*, *downtime*, iterações descartadas e custo de *checkpoint*, fundamenta a verificação das hipóteses H1 a H4 e sustenta a tese de que a migração ao vivo acionada por crescimento lento reduz o tempo total de treinamento de cargas concorrentes, com vantagem da preservação de estado via CRIU sobre a migração a frio.

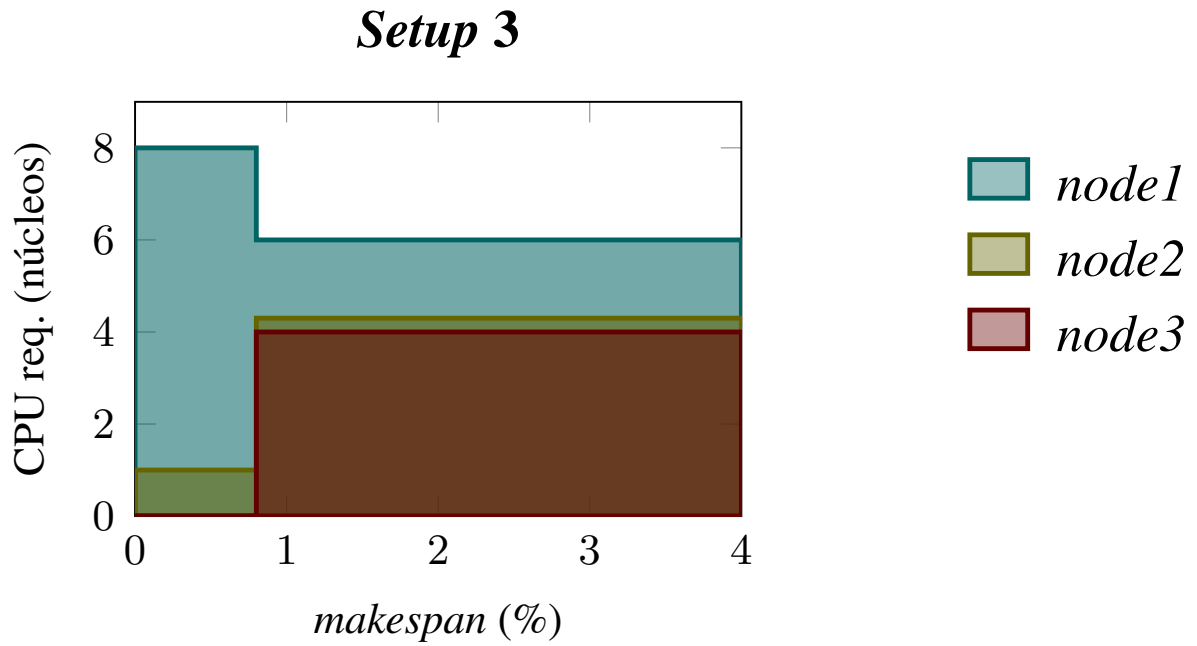


Figura 7 – Utilização de CPU por nó ao longo da duração dos testes realizados no setup 3, utilizando migração ao vivo.

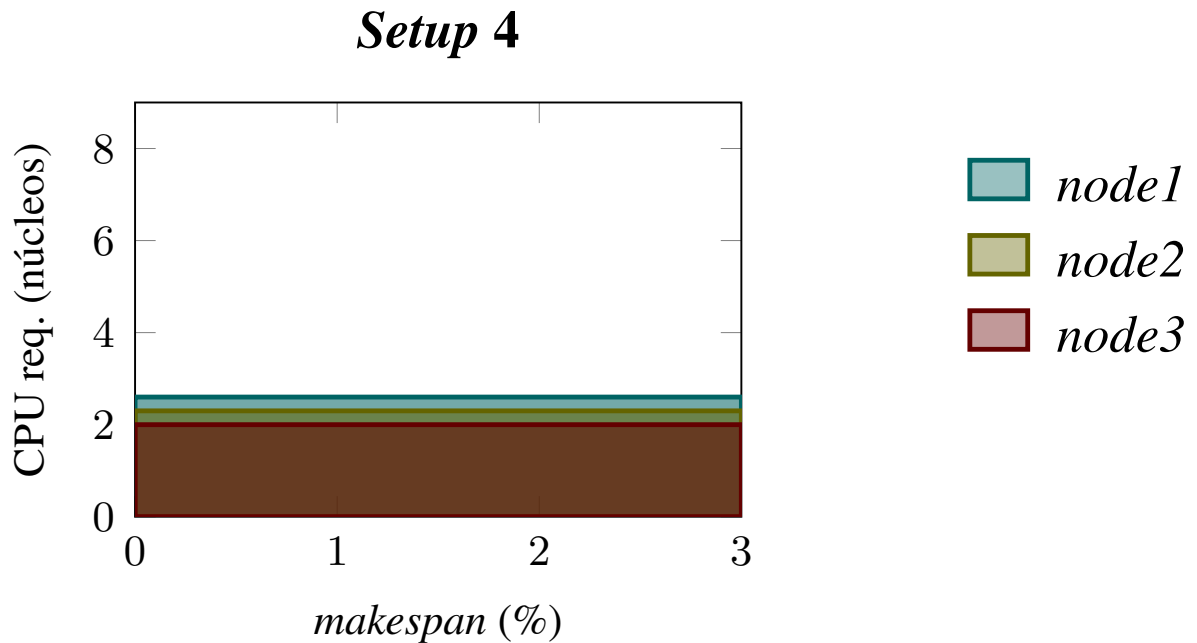


Figura 8 – Utilização de CPU por nó ao longo da duração dos testes realizados no setup 4, utilizando migração ao vivo.

Fonte: Elaborado pelo autor.

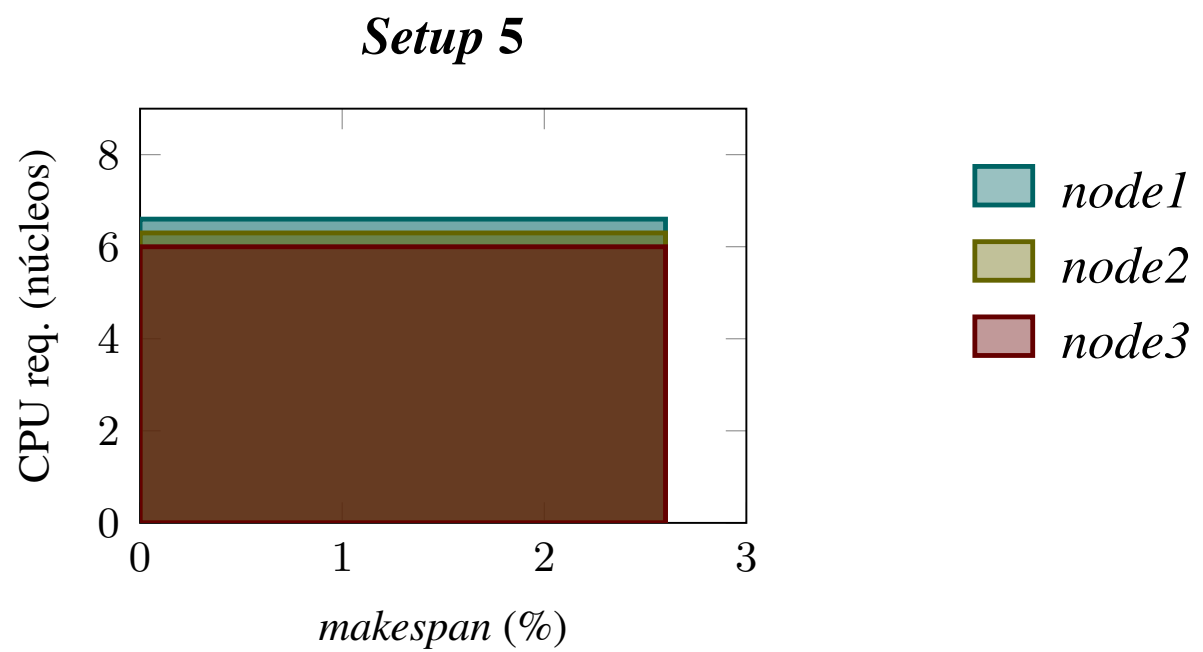


Figura 9 – Utilização de CPU por nó ao longo da duração dos testes realizados no setup 5, utilizando migração ao vivo.

Fonte: Elaborado pelo autor.

6 CONCLUSÃO

O crescimento das cargas de aprendizado profundo distribuído expôs uma incompatibilidade estrutural entre o modelo de escalonamento genérico do Kubernetes, concebido para tarefas curtas, independentes e tolerantes a reinicializações, e as necessidades reais dessas cargas, que são longas, exigem *gang scheduling*, disputam recursos escassos e acumulam estado de treinamento ao longo de horas de execução. Nesse contexto, a preempção convencional é proibitivamente cara, pois descarta todo o progresso computacional de uma tarefa interrompida. Este trabalho investigou o uso da migração ao vivo (*hot migration*) de contêineres, baseada em *checkpoint/restore* de processos, como mecanismo para tornar o rebalanceamento de carga transparente e econômico, preservando o estado de treinamento das tarefas realocadas.

Para tanto, foi projetado, implementado e avaliado o *dlkubesched*, um escalonador para Kubernetes voltado a tarefas de aprendizado profundo distribuído. O sistema é organizado em dois processos implantados no *cluster*, o processo escalonador, com quatro componentes desacoplados (Controlador, Escalonador, Monitor de Convergência e Migrador), e o agente CRIU, distribuído como *DaemonSet*, que cooperam exclusivamente por meio da API do Kubernetes. A interface com o usuário é o recurso personalizado *DeepLearningJob*, que expressa de forma declarativa a intenção de treinamento distribuído e delega ao Controlador a criação dos objetos nativos. Sobre essa base, o Migrador implementa dois modos de operação, a frio (*cold*) e ao vivo (*hot*, baseada em CRIU), acionados conforme a política declarada em cada tarefa.

6.1 CONCLUSÕES ESSENCIAIS

O objetivo principal, composto pela criação de um escalonador que utilizasse do método de migração ao vivo de contêineres para a aceleração de tarefas de aprendizado profundo distribuídas, foi cumprido, sendo evidenciado a eficiência do método em contraponto ao método padrão de escalonamento, chegando em casos de até 45% de *speedup*, utilizando modelos de aprendizado e *datasets* com complexidade e tamanho relevantes.

O projeto também demonstrou que o método de migração ao vivo pode ser utilizado com eficácia como alternativa ao método de checkpoint de modelo, apresentando tempo de *downtime* inferiores, por causa do método de *pre-copy*, que faz com que a execução das tarefas sejam praticamente ininterruptas, sem que haja a necessidade da pré-alocação de contêineres em vários nós, por exemplo.

Por fim, o projeto apresenta um *framework* atualizado para as versões do *Kubernetes* atuais, utilizando o estado da arte de *plugins*, como CRIU, para realizar a migração sem intervenção da aplicação. Por delegar a captura de estado ao kubelet e distribuir o agente de *checkpoint* como *DaemonSet* em todos os nós de trabalho, a solução independe de configurações específicas de *hardware* e pode ser implantada em *clusters* de diferentes portes e composições. Ao acoplar

a detecção de convergência ao rebalanceamento de recursos, o *dlkubesch* pode ser aplicado ao treinamento de uma ampla variedade de modelos, aproveitando de forma mais completa os recursos ociosos do sistema distribuído e reduzindo o tempo total de conclusão de cargas concorrentes.

6.2 LIMITAÇÕES

Os resultados devem ser interpretados à luz de algumas limitações do escopo experimental. Primeiro, a avaliação foi conduzida em um *cluster* homogêneo de baixa capacidade, com três nós e treinamento em CPU (*backend gloo*); embora o modo *criugpu* tenha sido implementado, a migração de estado de GPU não foi exercitada em *hardware* acelerado real, por falta de máquinas com mais de um acelerador. Segundo, a carga de trabalho restringiu-se a um único par modelo/conjunto de dados (ResNet-18 sobre CIFAR-10), escolhido por reprodutibilidade e tempo de época tratável, o que limita a generalização para modelos de maior escala e outros padrões de paralelismo. Por fim, o número de nós e de tarefas concorrentes é modesto frente a *clusters* de produção, de modo que os efeitos de escala permanecem em aberto.

6.3 TRABALHOS FUTUROS

As limitações apontadas delineiam direções naturais para a continuidade deste trabalho:

- 1) **Validação em *hardware* acelerado** - Com o desenvolvimento de *plugins* (Stoyanov et al., 2025), já é possível a utilização de migração ao vivo para modelos em GPU, sendo possível medir o impacto da migração sobre a localidade de comunicação e a topologia de interconexão (NVLink/InfiniBand).
- 2) **Escala e diversidade de cargas** - Estender a avaliação a *clusters* maiores, com mais nós e tarefas concorrentes, e a modelos e conjuntos de dados de maior porte (por exemplo, arquiteturas *transformer* e ImageNet), bem como a outros paradigmas de paralelismo, como o de modelo e o de *pipeline*.
- 3) **Heterogeneidade do *cluster*** - Investigar o comportamento do escalonador em ambientes heterogêneos, com nós de capacidades distintas e GPUs de gerações diferentes, situação em que a escolha do nó de destino da migração torna-se mais complexa.
- 4) **Aperfeiçoamento do gatilho de migração** - Explorar políticas de disparo mais sofisticadas, com autoajuste dos parâmetros do detector de crescimento lento e a incorporação de sinais adicionais além da convergência, como vazão, ocupação de fila e prioridade das tarefas, possivelmente por meio de técnicas preditivas para antecipar o ponto de retornos decrescentes.

- 5) **Preempção por prioridades e garantias de serviço** - Integrar de forma explícita a preempção baseada em prioridades e o atendimento a prazos (*deadlines*) e acordos de nível de serviço (SLAs), articulando o balanceamento de carga com políticas de justiça entre múltiplos usuários.
- 6) **Tolerância a falhas** - Reaproveitar a infraestrutura de *checkpoint/restore* não apenas para o balanceamento de carga, mas também como mecanismo de tolerância a falhas de nós, permitindo a retomada de tarefas de treinamento longas sem perda de progresso diante de falhas de *hardware*.

Em síntese, este trabalho demonstrou que a migração direta de contêineres, acionada pela convergência do treinamento e orientada ao rebalanceamento de recursos, é um mecanismo eficaz para reduzir o *makespan* de cargas de aprendizado profundo distribuído sob contenção, com custo residual quando desnecessária e sem prejuízo à qualidade do modelo. Ao acoplar a preservação de estado via CRIU ao ciclo de escalonamento do Kubernetes de forma transparente à aplicação, o *dlkubesch* oferece uma base sobre a qual as direções aqui apontadas podem ser exploradas rumo a um escalonamento mais eficiente de *clusters* de aprendizado profundo.

6.4 PUBLICAÇÕES

O desenvolvimento inicial deste projeto resultou na publicação de um artigo em congresso (Qualis A4), apresentado em 2025:

BARROSO, J. et al. A literature review on live migration in container-based distributed systems. In: Anais do XXVI Simpósio em Sistemas Computacionais de Alto Desempenho. Porto Alegre, RS, Brasil: SBC, 2025. p. 302–313.

Os resultados obtidos ao longo deste trabalho estão sendo compilados em um novo artigo, a ser submetido à publicação em uma revista científica a ser definida (provavelmente entre *Journal of Cloud Computing: Advances, Systems and Applications*, ou *Concurrency Computing: Practice and Experience* ou *Concurrent Engineering: Research and Applications*).

REFERÊNCIAS

- ALBAHAR, H. et al. SchedTune: A heterogeneity-aware GPU scheduler for deep learning. In: **Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)**. Taormina, Italy: IEEE, 2022. p. 695–705.
- BARHAM, P. et al. Xen and the art of virtualization. In: **Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP’03)**. New York, NY: ACM, 2003. p. 164–177.
- BEN-NUN, T.; HOEFLER, T. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. **ACM Computing Surveys**, ACM, v. 52, n. 4, p. 1–43, 2019.
- BENJAPONPITAK, T.; KARAKATE, M.; SRIPANIDKULCHAI, K. Enabling live migration of containerized applications across clouds. In: **Proceedings of the 2020 IEEE Conference on Computer Communications (INFOCOM)**. Virtual event: IEEE, 2020. p. 2529–2538.
- BERNSTEIN, P. A. Middleware: A model for distributed system services. **Communications of the ACM**, ACM, v. 39, n. 2, p. 86–98, 1996.
- BROWN, T. B. et al. Language models are few-shot learners. In: **Proceedings of the 34th International Conference on Neural Information Processing Systems**. Red Hook, NY, USA: Curran Associates Inc., 2020. (NIPS ’20). ISBN 9781713829546.
- BURNS, B. et al. **Kubernetes: Up and Running**. 3. ed. Sebastopol, CA: O’Reilly Media, 2022. ISBN 978-1-098-11020-8.
- BURNS, B. et al. Borg, omega, and kubernetes. **ACM Queue**, ACM, v. 14, n. 1, p. 70–93, 2016.
- CHOWDHERRY, A. et al. Palm: scaling language modeling with pathways. **J. Mach. Learn. Res.**, JMLR.org, v. 24, n. 1, jan. 2023. ISSN 1532-4435.
- CLARK, C. et al. Live migration of virtual machines. In: **Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI’05)**. Boston, MA: USENIX Association, 2005. p. 273–286.
- CRIU Project. **CRIU — Checkpoint/Restore In Userspace**. 2012. Open source project. Accessed: 2026-06-20. Disponível em: <https://criu.org>.
- GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. Cambridge, MA: MIT Press, 2016. Disponível em: <https://www.deeplearningbook.org>.
- GU, J. et al. Tiresias: A GPU cluster manager for distributed deep learning. In: **Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI’19)**. Boston, MA: USENIX Association, 2019. p. 485–500.
- KINGMA, D. P.; BA, J. **Adam: A Method for Stochastic Optimization**. 2015. Disponível em: <https://arxiv.org/abs/1412.6980>.
- Kubernetes Community. **Scheduling Framework**. 2019. Kubernetes Enhancement Proposal (KEP-624). Accessed: 2026-06-20. Disponível em: <https://github.com/kubernetes/enhancements/tree/master/keps/sig-scheduling/624-scheduling-framework>.

LAWSON, A.; SICA, J. **CNCF Annual Cloud Native Survey: The Infrastructure of AI's Future**. [S.l.], 2026. Accessed: 2026-08-07. Disponível em: <https://www.linuxfoundation.org/research/cncf-2025-annual-survey>.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, Nature Publishing Group, v. 521, n. 7553, p. 436–444, 2015.

LI, S. et al. PyTorch distributed: Experiences on accelerating data parallel training. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 13, n. 12, p. 3005–3018, 2020.

LIN, C.-Y.; YEH, T.-A.; CHOU, J. Dragon: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster. In: **Proceedings of the 9th International Conference on Cloud Computing and Services Science**. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2019. (CLOSER 2019), p. 569–577. ISBN 9789897583650. Disponível em: <https://doi.org/10.5220/0007707605690577>.

LU, Y.; JIANG, Y. A container pre-copy migration method based on dirty page prediction and compression. In: **Proceedings of the 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)**. Nanjing, China: IEEE, 2022. p. 704–711.

MAHAJAN, K. et al. Themis: Fair and efficient GPU cluster scheduling. In: **Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)**. Santa Clara, CA: USENIX Association, 2020. p. 289–304.

MANATURA, S. et al. Fastmig: Leveraging fastfreeze to establish robust service liquidity in cloud 2.0. In: **2024 IEEE 17th International Conference on Cloud Computing (CLOUD)**. Shenzhen, China: IEEE, 2024. p. 81–90.

MAO, Y. et al. Speculative container scheduling for deep learning applications in a kubernetes cluster. **IEEE Systems Journal**, IEEE, v. 16, n. 3, p. 3770–3781, 2022.

MELL, P.; GRANCE, T. **The NIST Definition of Cloud Computing**. 2011. (NIST Special Publication 800-145). Accessed: 2026-06-22. Disponível em: <https://doi.org/10.6028/NIST.SP.800-145>.

MILOJICIC, D. S. et al. Process migration. **ACM Computing Surveys**, ACM, v. 32, n. 3, p. 241–299, 2000.

NARAYANAN, D. et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)**. New York, NY: ACM, 2021.

NVIDIA Corporation. **CUDA C++ Programming Guide**. USA, 2023. Accessed: 2026-06-20. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide>.

RAJBHANDARI, S. et al. ZeRO: Memory optimizations toward training trillion parameter models. In: **SC20: International Conference for High Performance Computing, Networking, Storage and Analysis**. Virtual event: IEEE, 2020.

SERGEEV, A.; BALSIO, M. D. **Horovod: fast and easy distributed deep learning in TensorFlow**. 2018. Disponível em: <https://arxiv.org/abs/1802.05799>.

SHOEYBI, M. et al. **Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism**. 2019. Disponível em: <https://arxiv.org/abs/1909.08053>.

SINGH, G. et al. A secure and lightweight container migration technique in cloud computing. **Journal of King Saud University – Computer and Information Sciences**, Elsevier, v. 36, p. 101887, 2024.

STOYANOV, R. et al. **CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads**. 2025. Disponível em: <https://arxiv.org/abs/2502.16631>.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems: Principles and Paradigms**. 2. ed. Upper Saddle River, NJ: Prentice Hall, 2007. ISBN 978-0-13-239227-3.

VIODY, Y.; KISTIANTORO, A. I. Container migration for distributed deep learning training scheduling in kubernetes. In: **Proceedings of the 2022 9th International Conference on Advanced Informatics: Concept, Theory and Application (ICAICTA)**. Aichi, Japan: IEEE, 2022.

XIAO, W. et al. Gandiva: Introspective cluster scheduling for deep learning. In: **Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)**. Carlsbad, CA: USENIX Association, 2018. p. 595–610.

XU, B. et al. Sledge: Towards efficient live migration of docker containers. In: **Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing (CLOUD)**. Virtual event: IEEE, 2020. p. 321–328.

ZHANG, H. et al. KubeSPT: Stateful pod teleportation for service resilience with live migration. **IEEE Transactions on Services Computing**, IEEE, v. 18, n. 3, p. 1500–1514, 2025.