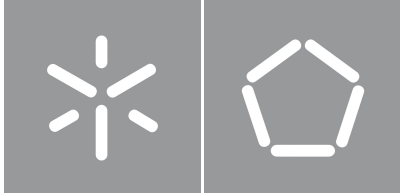


Universidade do Minho
Escola de Engenharia

Maria Beatriz Cardoso Gonçalves Barbosa e Moreira

Otimizações de Armazenamento Distribuído para Aprendizagem Profunda



Universidade do Minho
Escola de Engenharia

Maria Beatriz Cardoso Gonçalves Barbosa e Moreira

Otimizações de Armazenamento Distribuído para Aprendizagem Profunda

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação de
João Tiago Medeiros Paulo (Orientador)
Ricardo Gonçalves Macedo (Co-orientador)
Cláudia Vanessa Martins de Brito (Supervisora
no Centro de Investigação)

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho:



CC BY-NC-SA

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Agradecimentos

Esta dissertação vem concluir a minha jornada na Universidade do Minho e só foi possível graças ao apoio e suporte que recebi de várias pessoas, às quais estou muito grata.

Em primeiro lugar quero agradecer aos professores que me acompanharam neste projeto. Estou extremamente grata aos professores João Paulo e Ricardo Macedo pela oportunidade de trabalhar neste tema, por terem estado sempre disponíveis para me aconselhar e apresentar novas alternativas, bem como, para me encorajar e tranquilizar ao longo deste processo. Estendo ainda os agradecimentos à professora Cláudia Brito por ter partilhado comigo os seus conhecimentos na área de Inteligência Artificial e me ter aconselhado quanto aos modelos e estratégias que poderiam ser relevantes conhecer e explorar. Quero agradecer o tempo que todos os mencionados disponibilizaram para reunir regularmente comigo. Esses momentos contribuíram para me manter focada e me dar a confiança para necessária para ultrapassar os momentos mais complicados e incertezas que foram surgindo.

Um agradecimento muito especial aos meus pais que investiram na minha educação, que me ajudaram a seguir a carreira que escolhi, que me incentivaram a ser sempre diligente e dedicada, que acreditaram em mim mesmo nos momentos difíceis e tornaram tudo isto possível.

Quero ainda agradecer a todos os amigos que fiz na universidade pelo tempo que passamos juntos, projetos nos quais colaborámos quer em unidades curriculares, quer fora delas e por todas as memórias que fizemos e tornaram estes cinco anos tão inesquecíveis.

Por fim, quero agradecer ao **Texas Advanced Computing Center (TACC)** por disponibilizar a infraestrutura **HPC** (Frontera) na qual os testes desta dissertação foram executados e à Fundação para a Ciência e a Tecnologia, I.P., por ter financiado o desenvolvimento desta dissertação no âmbito dos projetos UIDP/50014/2020 e LA/P/0063/2020.

Declaração de Integridade

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, Braga, fevereiro 2024

Maria Beatriz Cardoso Gonçalves Barbosa e Moreira

Abstract

In today's world the utilization of **Deep Learning (DL)** is intrinsically integrated in the activity of several enterprises and industries. It allows us to extract knowledge from data, detect patterns and make predictions, increasing the competitiveness and quality of the services provided. However, the **DL** frameworks (e.g., *TensorFlow*, *PyTorch*, *Apache MxNet*) require not only considerable of computational power, but also efficient data storage, since they need to deal with large amounts of data. In particular, in each iteration of the **DL** model train different batches of the training dataset are accessed to be processed and incorporated in the model. The retrieval of this data can be a bottleneck to the performance of the system, since the datasets are getting increasingly bigger, reaching sizes in the order of **TBs**.

In the case of multi-node **DL** this becomes increasingly critical since there are many compute nodes training models, possibly with the same dataset, resulting in more requests directed to the shared file system competing with each other. If data could be stored nearer to the computational nodes and those nodes shared the data with one another, it would reduce the **I/O** pressure in the shared storage system and potentially reduce the time taken by these accesses and, consequently, the training time.

This thesis presents DistMonarch, a **DL** framework agnostic system that takes advantage of the storage system hierarchy by copying data to levels closer to each compute node and allows the nodes to share data with each other, in a transparent manner. Results show that using this system reduces accesses to the shared file system by up to 90% and training time of some models and configurations by up to 48%.

Keywords: **I/O** Optimization, Multi-Node Deep Learning

Resumo

No mundo atual a utilização de **Aprendizagem Profunda (AP)** está intrinsecamente integrada na atividade de diversas empresas e instituições. Esta permite extrair conhecimento de dados, detetar padrões e efetuar previsões, aumentando assim a competitividade e qualidade dos serviços prestados. No entanto, as ferramentas de **AP** (p. ex., *TensorFlow*, *PyTorch*, *Apache MxNet*) requerem não só um grande poder de computação, mas também de armazenamento, uma vez que necessitam de lidar com grandes quantidades de dados. Nomeadamente, em cada iteração do treino de um modelo de **AP** são acedidas diferentes amostras de dados do conjunto de dados de treino para serem processadas e incorporadas no modelo. A obtenção desses dados pode ser um gargalo no desempenho do sistema, uma vez que os conjuntos de dados são cada vez maiores, chegando a tamanhos na ordem dos **TBs**.

No caso de abordagens **AP** multi-nodo isto é ainda mais crítico, uma vez que temos vários nodos de computação a treinar possivelmente com o mesmo conjunto de dados, resultando em mais pedidos feitos ao sistema de ficheiros partilhado a concorrer entre si. Se os dados pudessem ser guardados mais perto dos nodos computacionais e os dados fossem partilhados entre eles isso permitiria reduzir a pressão de **E/S** no sistema de armazenamento partilhado e potencialmente reduzir os tempos dos acessos e, por conseguinte, do treino.

Esta tese apresenta o *DistMonarch*, um sistema agnóstico à ferramenta de **AP** que aproveita a hierarquia do sistema de armazenamento existente em infraestruturas de computação avançada, copiando dados para níveis mais próximos de cada nodo de computação e permite que os vários nodos partilhem esses dados entre si de forma transparente. Os resultados mostram que a utilização deste sistema permite reduzir os acessos ao sistema de ficheiros partilhado até 90% e tempo de treino de alguns modelos e configurações até 48%.

Palavras-chave: Otimização de **E/S**, Aprendizagem Profunda Multi-nodo

Conteúdo

1	Introdução	1
1.1	Problema	3
1.2	Objetivos e Contribuições	3
1.3	Resultados	5
1.4	Estrutura do Documento	5
2	Estado da arte	7
2.1	Enquadramento Teórico	7
2.1.1	Aprendizagem Máquina	7
2.1.2	Aprendizagem Profunda	8
2.1.3	Execução do Processo de Treino	11
2.1.4	Aprendizagem Profunda Distribuída	12
2.1.5	Aprendizagem Profunda Multimodelo	18
2.1.6	HPC em Aprendizagem Profunda	19
2.2	Trabalho Relacionado	20
2.2.1	Otimizações de Pré-processamento e Carregamento de Dados	20
2.2.2	Otimizações de Acesso a Dados: Caching	21
2.2.3	Otimizações de Obtenção de Dados: Aproveitamento da Estratificação do Sistema de Armazenamento	24
2.2.4	Resumo	27
3	Testes Preliminares	30
3.1	Tempo de Leitura do Sistema de Ficheiros Paralelo, do Armazenamento Local ou de Trocar Dados Entre Nodos	31
3.1.1	Configurações dos Testes	31
3.1.2	Resultados	31

3.2	Estudo dos Pedidos E/S	33
3.2.1	Configurações dos Testes	33
3.2.2	Resultados	36
3.3	Sumário	40
4	DistMonarch	41
4.1	Arquitetura da Solução	42
4.1.1	Cenários de Uso	44
4.2	Fluxo de Operações	45
4.2.1	Inicialização	45
4.2.2	Obtenção dos Dados	46
4.2.3	Obtenção Prévia de Dados	47
4.3	Implementação	48
4.3.1	Configurações	48
4.3.2	Aplicabilidade a Várias Ferramentas de AP	48
4.3.3	Gestor de Mapeamento	49
4.3.4	Permutação de Dados entre Trabalhadores	49
4.4	Resumo	49
5	Avaliação Experimental	51
5.1	Configurações	51
5.2	Resultados	53
5.2.1	Cargas de Trabalho Heterogéneas	54
5.2.2	Cargas de Trabalho Homogéneas	61
5.2.3	Teste Distribuído com <i>ParameterServerStrategy</i>	69
5.2.4	Teste de Longa Duração e Análise de Acurácia	74
5.3	Resumo	76
6	Conclusões e trabalho futuro	78
6.1	Perspetiva de Trabalho Futuro	79

Lista de Figuras

1	Funcionamento do treino de modelos de inteligência artificial	9
2	Tipos de ajuste de modelos a um conjunto de dados (imagens retiradas de [43]).	10
3	Exemplo de Shuffling	10
4	Divisão do tempo de execução	12
5	trabalhadores em distribuição de dados	14
6	trabalhadores em distribuição de modelo	14
7	Acesso a dados em modelos com paralelismo de dados	16
8	Acesso a dados em modelos com paralelismo de modelo	16
9	Exemplo de funcionamento da primeira fase do Ring All-reduce	16
10	Exemplo de funcionamento da segunda fase do Ring All-reduce	17
11	Exemplo de uma arquitetura de Servidor de Parâmetros	18
12	Exemplo de uma arquitetura de Servidor de Parâmetros do TensorFlow2	18
13	Exemplo de uma abordagem multimodelo.	19
14	Arquitetura do <i>Monarch</i> (retirado de [43])	26
15	Débito e latência médios de obtenção de dados para 256 KiBs , 512 KiBs e 1 MiBs de dados.	32
16	Débito e latência médios de obtenção de dados para os 50 MiBs , 100 MiBs e 119 MiBs de dados.	32
17	Índice dos ficheiros lidos ao longo de uma época de treino para carga de trabalho heterogénea.	36
18	Índice dos ficheiros lidos ao longo de uma época de treino para carga de trabalho homogénea.	37
19	Índice dos ficheiros lidos ao longo de uma época de treino com <i>MultiWorkerStrategy</i>	38
20	Índice dos ficheiros lidos ao longo de uma época de treino com <i>ParameterServerStrategy</i>	39

21	Arquitetura para a solução desenvolvida.	42
22	Localização do <i>DistMonarch</i> em multimodelo e <i>ring-allreduce</i>	44
23	Localização do <i>DistMonarch</i> em estratégias SP	45
24	Esquema do fluxo de obtenção de dados.	47
25	Exemplo de variáveis de ambiente a definir antes de correr o <i>DistMonarch</i>	48
26	Tempo médio por época para carga de trabalho heterogénea com 4 e 6 nodos a treinar modelos em simultâneo.	54
27	Tempo de execução para carga de trabalho heterogénea com 4 e 6 nodos a treinar modelos em simultâneo. Contemplando 5 épocas de <i>Alexnet</i> , <i>Lenet</i> e <i>Shufflenet</i> , 6 épocas de <i>Resnet18</i> , 4 épocas de <i>InceptionV3</i> e 2 épocas de <i>VGG19</i>	55
28	Número de leituras do Lustre de cada modelo no cenário de carga de trabalho heterogénea com 4 e 6 nodos ao longo de 5 épocas de <i>Alexnet</i> , <i>Lenet</i> e <i>Shufflenet</i> , 6 épocas de <i>Resnet18</i> , 4 épocas de <i>InceptionV3</i> e 2 épocas de <i>VGG19</i>	57
29	Variação da utilização de CPU ao longo de 3 épocas para cada modelo do teste de carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.	59
30	Tempo médio por época para carga de trabalho homogénea com 6 nodos a treinar modelos em simultâneo.	61
31	Tempo de execução de 10 épocas para carga de trabalho homogénea com 4 e 6 nodos a treinar modelos em simultâneo.	62
32	Número de leituras do Lustre para carga de trabalho homogénea com 4 nodos durante 1 e 3 épocas.	63
33	Número de leituras do Lustre para carga de trabalho homogénea com 6 nodos durante 1 e 3 épocas.	64
34	Variação da percentagem de utilização de GPU ao longo de 3 épocas para o modelo <i>Lenet</i> com o otimizador <i>Adam</i> e uma amostra de 256 KiBs	67
35	Tempo médio por época para <i>ParameterServerStrategy</i> com 4 trabalhadores.	69
36	Tempo de execução para <i>ParameterServerStrategy</i> com 4 trabalhadores.	69
37	Variação do tempo de cada época ao longo de 9 épocas.	71
38	Número de leituras do Lustre para <i>ParameterServerStrategy</i> com 4 nodos durante 1 e 3 épocas.	71
39	Variação da percentagem de utilização de CPU de <i>ParameterServerStrategy</i> com 4 trabalhadores ao longo de 3 épocas.	73

40	Número de épocas concluídas para cada modelos e tipo de execução em 40 horas. . .	75
----	---	----

Lista de Tabelas

1	Tabela de sistemas de otimizações de obtenção de dados e respetivos problemas.	29
2	Especificações do ambiente experimental	30
3	Operações intercetadas e respetivas parâmetros e valores de retorno registados	35
4	Tabela de percentagem de utilização de CPU para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.	58
5	Tabela de percentagem de utilização de CPU para carga de trabalho heterogénea com 6 nodos ao longo de 3 épocas.	59
6	Tabela de percentagem de utilização de GPU para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.	60
7	Tabela de percentagem de utilização de GPU para carga de trabalho heterogénea com 6 nodos.	60
8	Tabela de percentagem de utilização de memória de GPU para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.	60
9	Tabela de percentagem de utilização de memória de GPU para carga de trabalho heterogénea com 6 nodos.	61
10	Tabela de percentagem de utilização de CPU para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.	65
11	Tabela de percentagem de utilização de CPU para carga de trabalho heterogénea com 6 nodos ao longo de 3 épocas.	65
12	Tabela de percentagem de utilização de GPU para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.	65
13	Tabela de percentagem de utilização de GPU para carga de trabalho heterogénea com 6 nodos ao longo de 3 épocas.	66

14	Tabela de percentagem de utilização de memória de GPU para carga de trabalho heterogénea carga de trabalho heterogénea com 4 nodos.	68
15	Tabela de percentagem de utilização de memória de GPU para carga de trabalho heterogénea carga de trabalho heterogénea com 6 nodos.	68
16	Tabela com percentagem média de utilização de CPU ao longo de 3 épocas.	72
17	Tabela com percentagem média de utilização de GPU ao longo de 3 épocas de <i>ParameterServerStrategy</i>	73
18	Tabela com percentagem média de utilização da memória dos GPUs ao longo de 3 épocas de <i>ParameterServerStrategy</i>	74

Acrónimos

AM Aprendizagem Máquina.

AP Aprendizagem Profunda.

APD Aprendizagem Profunda Distribuída.

APD Aprendizagem Profunda Distribuída.

AVL Adelson-Velsky and Landis.

CF Caminho do Ficheiro.

CPU Central Process Unit.

DF Descritor de Ficheiro.

DFs Descritores de Ficheiros.

DL Deep Learning.

DLFS Deep Learning File System.

E/S Entrada/Saída.

FIFO First In First Out.

GBs GigaBytes.

GiBs GibiBytes.

GPU Graphical Processing Unit.

HPC High-Performance Computing (Computação Avançada).

HVAC High-Velocity AI Cache.

I/O Input/Output.

IA Inteligência Artificial.

IP Internet Protocol.

KiBs Kibibytes.

LDL Locality-aware Data Loading.

MiBs MebiBytes.

MPI Message Passing Interface.

POSIX Portable Operating System Interface.

RNA Rede Neural Artificial.

RNP Rede Neural Profunda.

RNPs Redes Neurais Profundas.

RPC Remote Procedure Call.

SGD Stochastic Gradient Descent.

SP Servidor de Parâmetros.

TACC Texas Advanced Computing Center.

TBs Terabytes.

TCP Transmission Control Protocol.

TF TensorFlow.

TiBs Tebibytes.

TLU Threshold Logic Unit.

TPU Tensor Processing Unit.

Capítulo 1

Introdução

Na última década a **Aprendizagem Máquina (AM)** (uma área da **Inteligência Artificial (IA)**) tem vindo a reunir o interesse da comunidade científica e tem-se integrado cada vez mais na atividade de diversas empresas. Alguns exemplos deste fenómeno são, por exemplo, no setor da saúde, em que a **AM** está a ser utilizada para reconhecer padrões nas doenças de modo a conseguir identificar o diagnóstico mais apropriado e respetivo tratamento [16, 34]. No setor de retalhos as empresas recorrem a modelos de **AM** para ajustar os artigos que são recomendados a cada cliente com base no que estes estariam mais interessados em comprar [14, 15]. Na indústria da moda, em particular, estes modelos são utilizados para criar imagens de realidade aumentada de modo a permitir que as pessoas possam visualizar como é que uma roupa lhes ficaria sem a terem de experimentar [11, 12, 13]. No setor financeiro a **AM** é utilizada para diversos fins desde deteção de potenciais transferências fraudulentas [5, 6, 7] a consultoria financeira [8, 9, 10]. Mesmo em atividades mais quotidianas como desbloquear o telemóvel ou descobrir como chegar a uma dada localização geográfica, muitas vezes temos modelos de **AM** envolvidos, para permitir o uso de funcionalidades de reconhecimento facial e nos aconselhar qual o melhor roteiro.

Normalmente estes algoritmos realizam inúmeras operações matemáticas (equações algébricas e matriciais). Como as **Graphical Processing Unit (GPU)**s e as **Tensor Processing Unit (TPU)**s são unidades de processamento mais especializadas, são capazes de processar cálculos matemáticos significativamente mais rápido do que **CPU**s. Deste modo, estão bastante adaptados às necessidades de computação de programas de **IA** (permitindo diminuir significativamente os seus tempos de execução).

Como tal, houve uma adoção generalizada destes equipamentos para o treino destes modelos. No entanto, enquanto que o treino do modelo é, genericamente, levado a cabo nesses equipamentos e o pré-processamento dos dados pode lá ser executado (com auxílio de ferramentas como o DALI [4]), a leitura de dados tem de ser feita em **CPU**. Assim, para tirar o melhor proveito possível do equipamento é preciso que o sistema de armazenamento consiga alimentar o modelo com dados à mesma velocidade que este faz a sua computação. Apesar disto, análises de desempenho destas aplicações [38, 61] demonstram

que o tempo gasto em **E/S** de dados acaba, diversas vezes, por representar um gargalo na execução (principalmente em contextos de computação de alto desempenho com sistema de ficheiros paralelo). Isto resulta numa diminuição da eficiência do processo de treino e num subaproveitamento de recursos computacionais.

Modelos de **AP** mais complexos têm uma maior capacidade de aprendizagem. Na prática isto significa que estes modelos são capazes de detetar mais padrões nos dados, o que significa que para conjuntos de dados de menores dimensões os algoritmos vão extrair padrões do ruído nos dados afetando as generalizações feitas, podendo resultar na diminuição da acurácia do modelo. Assim, modelos mais complexos necessitam de grandes quantidades de dados para diminuir o impacto de ruído nos dados e assim serem mais exatos. Graças ao aumento na capacidade dos equipamentos de armazenamento e ao enorme crescimento da Internet [45], conseguimos ter conjuntos de dados de centenas de **GiBs** ou até de **TiBs** (como *ImageNet* [44] ou o *OpenImages* [22]). Deste modo, é possível ter conjuntos de dados grandes o suficiente para alimentar modelos com elevada complexidade. Não obstante, quanto maior for o conjunto de dados a processar, mais vezes temos de ir ao sistema de armazenamento buscá-los e se este passo estiver a limitar a eficiência do processo de treino este aumento implica que o processamento fique bloqueado até obter os dados necessários. Consequentemente, os tempos de execução destes processos podem ser de horas ou até mesmo de dias.

De modo a diminuir os tempos de execução surgiram abordagens de treino distribuídas (**APD**), que procuram repartir os dados pelos recursos disponíveis. Nestes, é possível pôr diversos **GPUs** ou **TPUs** (independentemente de se pertencem ou não à mesma máquina) a processar partes diferentes do conjunto de dados. Deste modo, quer o tempo de espera pela obtenção de cada amostra de dados, quer o tempo de processamento das mesmas permanece o mesmo, mas é possível fazer com que ocorram paralelamente. Assim, ao tirarmos partido da escala horizontal de recursos, conseguimos diminuir os tempos de execução.

Outra metodologia que vem tirar partido da escala horizontal de recursos é a abordagem multimodelo. Esta é similar à abordagem anterior na medida em que ambas permitem a paralelização da computação. No entanto, enquanto que na abordagem distribuída todos os nodos estão a treinar um único modelo, nesta estão a ser treinados diversos modelos em simultâneo.

As infraestruturas **HPC** possuem diversos nodos, com várias *cores* e um sistema de armazenamento distribuído de dimensões consideráveis estando assim bastante adaptados às exigências destes programas e sendo, consequentemente, as estruturas mais comumente utilizadas para os executar. No entanto, o conjunto de dados de treino é normalmente colocado num sistema de ficheiros partilhado pelos

diversos utilizadores, pelo que os diversos pedidos dos vários nodos estão a competir entre si. Em ambas as abordagens multi-nodo que referimos temos diversos nodos a fazer pedidos da leitura simultâneos e consequentemente a competir pelos recursos de armazenamento. Para além de competirem entre si têm também de competir com pedidos de diversos outros programas e utilizadores, resultando num aumento dos tempos de resposta aos pedidos **E/S** e num atraso no treino dos modelos [42, 43, 46].

1.1 Problema

Em infraestruturas **HPC** é bastante comum cada nodo ter recursos de armazenamento local, mas estes têm dimensões significativamente mais limitadas do que os sistemas de ficheiros partilhados, pelo que muitas vezes não têm capacidade para guardar o conjunto de dados de treino. Por este motivo, os conjuntos de dados costumam ser guardados no sistema de ficheiros partilhado. No entanto, este é partilhado por todos os nodos do supercomputador, potencializando tempos de resposta maiores e sujeitos a maior variabilidade. Desta forma, é importante arranjar uma forma de melhorar o desempenho do sistema de armazenamento para **AP** multi-nodo de modo a reduzir a carga no sistema de ficheiros partilhado. Já existem ferramentas que fazem isso, tirando partido de recursos de armazenamento local (como o *Monarch* abordado na secção 2.2.3). Todavia, estas estão feitas a pensar em treino local, não estando adaptadas para o facto de que existem vários nodos a correr e todos eles têm o seu próprio disco local. A capacidade de armazenamento conjunta desses nodos é superior à sua individual, pelo que distribuir o conjunto de dados por estes permite que mais dados fiquem em armazenamento local, diminuindo consequentemente o número de acessos ao sistema de ficheiros partilhado.

Assim, surgem problemas como saber como distribuir o conjunto de dados, saber que dados é que cada nodo tem guardado em disco local, como cada nodo pode aproveitar os dados guardados pelos restantes de modo a diminuir ao máximo a necessidade de ir ao sistema de ficheiros partilhado e como o fazer sem necessitar da intervenção do utilizador. De momento ainda não existe nenhuma solução que permita fazer isto de forma transparente à ferramenta de **AP**, pelo que esta tese procura dar resposta a estes problemas.

1.2 Objetivos e Contribuições

O principal objetivo desta dissertação é melhorar o desempenho do treino de programas de **Aprendizagem Profunda** em abordagens multi-nodo (diminuindo o tempo de execução) e minimizar a interferência dos pedidos de **E/S** ao sistema de ficheiros partilhado que contém o conjunto de dados de treino.

De modo a melhorar o desempenho destas abordagens é necessário fazer com que as leituras deixem de representar um gargalo na execução ou que, pelo menos, o tempo consumido a esperar pela obtenção dos dados necessários para a iteração seguinte diminua.

Para isto, é necessário ter uma ferramenta ou biblioteca de *tiering* do sistema de armazenamento que permita que a **AP** multi-nodo aproveite o armazenamento local de cada nodo. Esta ferramenta deve ainda otimizar o armazenamento com base nos nodos e respetivo armazenamento local disponíveis.

Para atingir estes objetivos, esta dissertação traz quatro contribuições:

- Um estudo dos pedidos **E/S** de programas de **Aprendizagem Profunda (AP)** em abordagens multi-nodo, que procura identificar padrões que possam ser utilizados para otimizar o sistema de armazenamento. Este estudo permitiu verificar que enquanto que nas abordagens multimodelo testadas cada programa lê a totalidade dos dados por uma ordem aleatória independente da dos restantes, na estratégia *MultiWorkerMirroredStrategy* [19] os vários nodos dividem o conjunto de dados entre si, sendo que cada nodo só lê os dados que é responsável por processar. Já no *ParameterServerStrategy* [24] os vários nodos estavam a ler a totalidade do conjunto de dados em todas as épocas independentemente dos dados que processem. Foi ainda possível verificar que modelos como *Alexnet*, *Lenet*, *Resnet18* e *Shufflenet* têm o acesso a dados como fator limitante, sendo mais significativo para os 2 primeiros, enquanto que modelos como o *InceptionV3* e o *VGG19* são limitados pela utilização de **GPU**.
- Um estudo que analisa e compara o desempenho de fazer leituras do sistema de ficheiros partilhado, do armazenamento local de cada nodo e de partilhar dados guardados no armazenamento local entre nodos. Este veio validar a eficiência da troca de dados entre nodos para programas de **AP** uma vez que demonstrou que, para as dimensões normalmente utilizadas para as amostras de dados, trocar dados entre nodos é comparável com lê-los diretamente do sistema de armazenamento local, sendo assim mais eficiente do que ler do sistema de ficheiros partilhado.
- O desenho de um sistema de *tiering* adaptado a **AP** multi-nodo, o *DistMonarch*. Este procura permitir que ferramentas de **AP** tirem partido dos recursos de armazenamento local disponíveis nos vários nodos envolvidos de uma forma transparente. Para tal, o *DistMonarch* serve de intermediário entre os pedidos de dados das ferramentas de **AP** e o sistema de armazenamento, permitindo não só que dados sejam obtidos de armazenamento local do nodo como do armazenamento local de nodos que também estejam a correr uma instância do *DistMonarch*. Em específico, o sistema corre *threads* para puxar ficheiros do sistema de ficheiros partilhado para o armazenamento local

e para fazer a pré-obtenção de dados que estão disponíveis noutros nodos de modo a ter os dados em camadas de armazenamento mais rápidas e diminuir o impacto do tempo de partilha de dados entre nodos, respetivamente.

- Um protótipo desse sistema e respetiva avaliação, de modo a podermos confirmar se se verifica uma diminuição no número de pedidos feitos ao sistema de ficheiros partilhado e, idealmente, uma melhoria nos tempos de execução. A avaliação foi feita no supercomputador Frontera [68] do **TACC** [31] com diversos programas de **AP** multi-nodo que utilizam a ferramenta **TensorFlow**. Esta demonstrou que, para programas que tenham uma boa aleatoriedade na ordem de leitura de dados, a utilização do *DistMonarch* permite reduzir o número de pedidos feitos ao sistema de ficheiros paralelo até 90% caso o tamanho conjunto dos discos locais dos nodos seja suficiente para guardar a totalidade do conjunto de dados e 48% caso contrário. Para programas com reduzida aleatoriedade na ordem das leituras dos dados a redução passa a cerca de 19% e 12%, respetivamente. Foi ainda possível verificar que a ferramenta consegue reduzir o tempo de treino de vários modelos e abordagens, principalmente para modelos limitados pelos acessos ao sistema de armazenamento em estratégias que apresentem elevada aleatoriedade na ordem de acesso aos dados.

1.3 Resultados

O protótipo do *DistMonarch* mencionado anteriormente encontra-se disponível em <https://github.com/dsrhaslab/DistMonarch> para poder ser utilizado pela comunidade. Já a ferramenta de captura de chamadas ao sistema operativo mencionada no capítulo de Testes Preliminares 3.2.1, o *sdlprof*, está disponível em <https://github.com/dsrhaslab/sdlprof>. Esta ferramenta permite à comunidade colecionar registos das chamadas ao sistema que o programa com que é corrido faz, sendo possível selecionar as operações e respetivos argumentos que devem aparecer no ficheiro resultante.

1.4 Estrutura do Documento

Este documento está dividido em seis capítulos diferentes. O primeiro onde o problema a resolver é contextualizado e explicado, os objetivos para a solução final são definidos e onde definimos as contribuições que esta dissertação traz.

O capítulo 2 encontra-se dividido em duas secções principais. A primeira expõe o enquadramento

teórico que orientou o desenvolvimento deste projeto, explicando diversos conceitos relevantes para este trabalho. Já a segunda secção fala de diversos estudos e projetos relacionados com a melhoria dos tempos de execução de programas **AP** através da otimização da utilização de sistemas de armazenamento.

O capítulo 3 apresenta testes preliminares levados a cabo com o objetivo de identificar padrões de acesso a dados em programas **AP** que possam ser utilizados para otimizar o sistema de armazenamento e de validar a eficiência da partilha de dados entre nodos quando comparada com a leitura do sistema de ficheiros partilhados.

O capítulo 4 explica a abordagem seguida para resolver o problema, contemplando a arquitetura, fluxo de operações e detalhes de implementação do *DistMonarch*. O capítulo 5, por sua vez, expõe os resultados de testes experimentais desta.

Por fim, o capítulo 6 finda este documento expondo as conclusões alcançadas e oportunidades de trabalho futuro.

Capítulo 2

Estado da arte

Esta dissertação propõe uma solução de armazenamento para **Aprendizagem Profunda (AP)** em ambientes multi-nodo. Em vista disto, é importante compreender alguns conceitos relacionados com **AP**, bem como o seu funcionamento.

2.1 Enquadramento Teórico

2.1.1 Aprendizagem Máquina

Aprendizagem Máquina (AM) é uma área da **Inteligência Artificial (IA)** que usa algoritmos que permitem a computadores identificar padrões em dados de modo a fazer previsões. Todos estes algoritmos têm uma fase de treino em que o modelo vai analisando várias amostras de dados e deduzindo regras (“aprendizagem”). Isto permite que, com base em segmentos de dados, se consiga inferir outros e, assim, prever potenciais consequências e extrair conclusões. Existem 4 tipos principais de algoritmos de **AM**:

- **Aprendizagem Supervisionada:** O algoritmo é treinado com um conjunto de dados devidamente rotulado e com essa informação procura ser capaz de prever os rótulos corretos para outros dados.
- **Aprendizagem Não Supervisionada:** O algoritmo é treinado com um conjunto de dados sem rótulos. Com base na aglomeração dos dados, segundo os seus atributos e semelhanças, procura ser capaz de os agrupar (formar clusters). Normalmente necessita de uma fase de análise do contexto para permitir inferir o significado de cada grupo. Por vezes, é utilizado antes da Aprendizagem Supervisionada, para encontrar correlações dentro do conjunto de dados sem recorrer aos rótulos.

- **Aprendizagem Auto-supervisionada:** Algoritmo similar à Aprendizagem Supervisionada, mas baseia-se em algoritmos heurísticos em vez de nos rótulos disponibilizados por seres humanos.
- **Aprendizagem por reforço:** O algoritmo baseia-se num sistema de “recompensa/punição”, sem interferência do programador, em que o algoritmo recebe um reforço positivo (“recompensa”) ou negativo (“punição”), dependendo da qualidade das inferências que faz.

Todos estes algoritmos recorrem a ciclos de treino para iterarem pelos dados de treino. Em cada iteração uma amostra de dados é obtida e utilizada para treinar o modelo. Uma **época** é um conjunto de iterações que percorre a totalidade do conjunto de dados de treino.

2.1.2 Aprendizagem Profunda

Aprendizagem Profunda (AP) é um campo da **AM** em que os modelos são construídos com base em camadas de neurónios interconectados. Este tipo de modelos é denominado **Rede Neural Artificial (RNA)** e tem 3 tipos de camadas de neurónios:

- **Camadas de entrada:** camada inicial, responsável por preencher as variáveis de entrada com os dados que recebe.
- **Camadas ocultas:** camada(s) entre a de entrada e a de saída. Recebem os seus valores de entrada de neurónios de camadas de entrada ou de outras camadas ocultas e são responsáveis pela computação. **RNAs** com mais que uma destas camadas são denominados **Redes Neurais Profundas (RNP)**.
- **Camadas de saída:** camada final da rede. Recebe os seus valores de entrada de neurónios de camadas ocultas e com base nestes produz o *output* e transformam-no em variáveis de saída.

A lógica do ciclo de treino, ilustrado na Figura 1, funciona como explicado para **AM**, sendo que o processamento pode ser dividido em três partes diferentes: propagação, cálculo do erro, retropropagação.

Propagação

Fase em que a amostra de dados é passada desde a camada de entrada até à camada de saída e em que o modelo faz a computação da previsão do valor de saída. Normalmente, os neurónios funcionam como uma **Threshold Logic Unit (TLU)**, ou seja, as conexões que estabelecem uns com os outros têm pesos e é com base neles que computam o valor da soma ponderada dos valores de *input*. O valor obtido é depois passado a uma função de ativação que a transforma no valor de saída do neurónio.

Retropropagação

Após a obtenção do valor previsto pelo modelo, este recorre a uma função de perda para determinar o quão longe este valor estava do esperado (*loss score*). Este valor é depois propagado de forma ponderada (recorrendo aos pesos das conexões) pelos neurónios do modelo desde a camada de saída até à camada de entrada. O valor de erro obtido em cada unidade é denominado gradiente de erro.

Atualização dos Pesos do Modelo

Os pesos das conexões são atualizados, com base nos gradientes de erro calculados, de modo a tentar diminuir o valor do *loss score* e que, por conseguinte, as estimativas produzidas pelo modelo se aproximem ao máximo dos valores esperados.

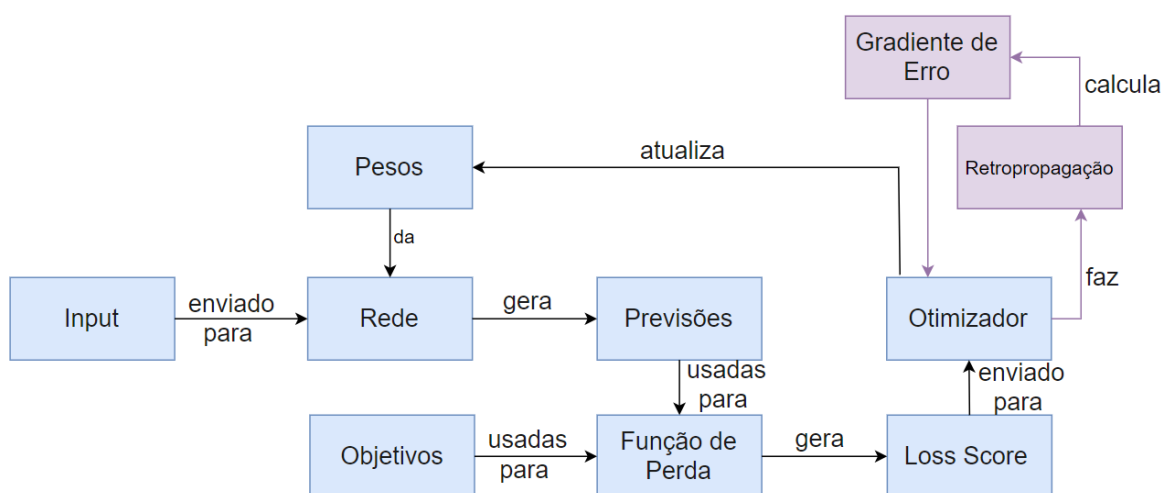


Figura 1: Funcionamento do treino de modelos de inteligência artificial

Sobre e Sub-ajuste

Um problema que ocorre com alguma frequência com algoritmos de **AM** é sobre-ajustar o modelo (*overfitting*) [Figura 2a] ao conjunto de dados de treino. Quão mais complexo for o modelo, maior a sua capacidade de aprendizagem e se os conjuntos de dados tiverem ruído ou forem demasiado pequenos para atenuar o impacto dos casos excecionais é possível que os modelos infiram regras ou detetem padrões errados. Esta especificação excessiva acaba por se revelar contraproducente, uma vez que as previsões do modelo para dados novos não estariam ajustadas, diminuindo a utilidade do mesmo. Contudo, também não queremos sub-ajustar o modelo (*underfitting*) [Figura 2c] ao conjunto de dados de treino, uma vez que queremos que as regras inferidas pelo modelo sejam suficientes para fazer previsões.

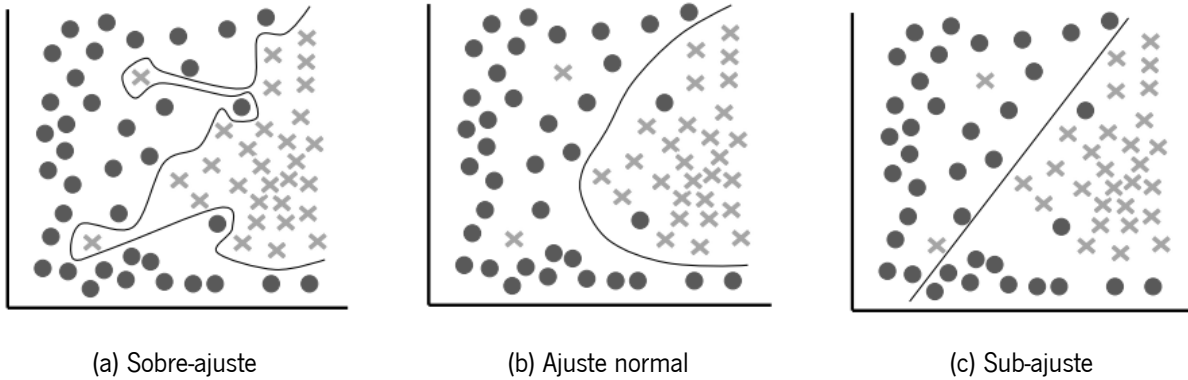


Figura 2: Tipos de ajuste de modelos a um conjunto de dados (imagens retiradas de [43]).

Existem vários métodos para diminuir o impacto do barulho do conjunto de dados de treino no modelo final.

Shuffling *Shuffling* é o processo de trocar a ordem em que os dados são apresentados ao modelo [Figura 3], mantendo sempre a associação entre os rótulos e os dados em si. Isto previne que o modelo infira regras com base na ordem em que vê os dados, garantindo que todas as amostras são estatisticamente independentes e têm a mesma distribuição de probabilidades.

Isto deve ser feito no início de cada época e pode ser levado a cabo de duas formas distintas:

1. Globalmente – baralha a ordem em que as amostras aparecem no conjunto de dados de treino antes do treino da época e depois faz a recolha pela ordem em que os dados aparecem. Normalmente isto é feito baralhando os nomes dos ficheiros a ler.
2. Por recolha – seleciona aleatoriamente a amostra de dados que deve ser recolhida.



Figura 3: Exemplo de Shuffling

Técnicas de Regularização

- Camadas de abandono (*Dropout Layers*) – Ao definir a estrutura do modelo pode-se inserir camadas de abandono. Estas servem para, a cada iteração, anular a contribuição de alguns dos neurónios

(impedindo que eles passem o seu output para a camada seguinte). Como este processo é feito a cada iteração os neurónios anulados vão variando, resultando num aumento da aleatoriedade inserida e, consequentemente, diminuindo a probabilidade de o modelo ficar sobre-ajustado.

- Regularização L1 e L2 [63] – Regularização L1 permite reduzir o volume de ruído no modelo gerado, passando a 0 os pesos de recursos que sejam muito pequenos, enquanto que a regularização L2 contribui para estabilizar os pesos de recursos com alta correlação entre si.
- Paragem Antecipada (*Early stopping*) [74] – otimização que procura parar o treino do modelo antes que este comece a ficar sobre-ajustado e sem comprometer a acurácia do modelo. Os tipos principais são (1) parar o treino quando se completa um determinado número de épocas, (2) parar quando a função de perdas começa a dar valores muito reduzidos ou (3) utilizando um conjunto de validação e parando o treino quando os erros na validação começam a aumentar.
- Aumento de Dados (*Data Augmentation*) [65] – é uma técnica que permite gerar novos dados de treino com base nos existentes no conjunto de treino, de modo a aumentar o conjunto de dados. Com mais dados a serem analisados é possível atenuar o impacto que o ruído pode ter no modelo.

Métricas

As métricas utilizadas para avaliar estes modelos estão normalmente relacionadas com a capacidade de previsão. Por essa lógica, métricas como a acurácia, a raiz quadrada média do erro, o erro absoluto, precisão, especificação e outras que permitem avaliar as previsões feitas pelos modelos são frequentemente utilizadas.

Ferramentas e Bibliotecas para AP

Existem diversas ferramentas e bibliotecas que têm como objetivo facilitar a produção, configuração e customização de modelos de AP, como, por exemplo, o Keras [17], o TensorFlow [35], o PyTorch [64], o Apache MXNet [39] ou o CAFFE [51]. Nesta tese tencionamos recorrer ao TF, uma vez que é uma das mais populares e recorrentemente utilizadas.

2.1.3 Execução do Processo de Treino

O tempo de execução de treino de modelos de AP pode ser dividido em três principais componentes:

- tempo de **E/S** de dados - tempo despendido entre ir ao sistema de armazenamento recolher conjuntos de dados para processar e entregar esses dados à rede neural. Deste modo, estamos a considerar o tempo de leitura dos dados e potenciais operações de decodificação e pré-processamento dos dados [Figura 4]. Este é dependente do valor de hiperparâmetros como o número de épocas e o tamanho das amostras, uma vez que estes fazem com que o número de vezes que acedemos ao sistema de armazenamento para ler dados e o tamanho lido varie. Pode ainda variar dependendo se a computação é levada a cabo em **CPU** ou **GPU**, uma vez que a leitura dos dados é quase sempre feita em **CPU**, pelo que se o treino em si for em **GPU** o tempo de transferência de contexto dos dados também é considerado nesta componente.

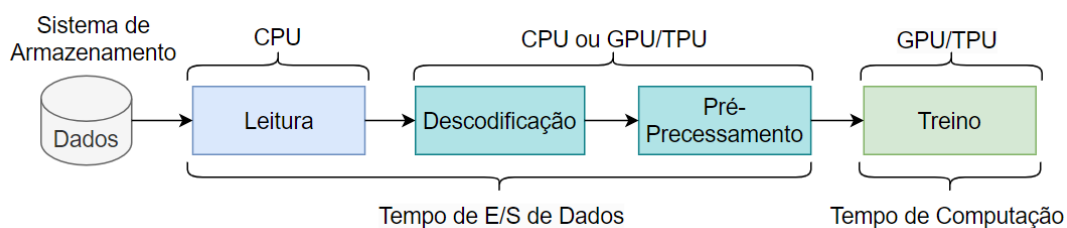


Figura 4: Divisão do tempo de execução

- tempo de computação – tempo que todas as iterações das várias épocas demoraram (soma dos vários tempos de propagação, cálculo do erro e retropropagação) e da atualização dos parâmetros do modelo. Este também é dependente do valor de vários hiperparâmetros, como o número de épocas, o tamanho das amostras, a complexidade e quantidade de camadas ocultas, entre outros.
- tempo de comunicação – existe principalmente em treino de **APD** e refere-se ao tempo gasto em trocas de informação entre os vários trabalhadores. Está principalmente relacionado com a largura de banda da rede, com a estratégia e política de sincronização seguidas e com a quantidade de trabalhadores (e potencialmente servidores de parâmetros) que estamos a considerar.

2.1.4 Aprendizagem Profunda Distribuída

Aprendizagem Profunda Distribuída (APD) é um tipo de **AP** em que são usados algoritmos multinodo. Estes algoritmos permitem distribuir o processo de treino por vários trabalhadores, aumentando o desempenho e acurácia, escalar mais eficientemente amostras de dados maiores e/ou modelos maiores e mais complexos. Existem duas vertentes de distribuição: distribuição de dados e distribuição de modelo.

Distribuição de Dados

Consiste em distribuir as amostras de dados pelos vários trabalhadores, sendo que em cada iteração a amostra de dados é dividida em mini-amostras que são distribuídas pelos trabalhadores. Isto permite acelerar o processo de treino, uma vez que as mini-amostras são processadas em paralelo em vez de sequencialmente e permite aumentar o tamanho da amostra, uma vez que cada trabalhador só vai processar uma fração desses dados.

No fim de cada iteração os parâmetros do modelo continuam a ter de ser atualizados. No entanto, se este processo for executado da mesma forma que antes de os dados serem distribuídos, os parâmetros do modelo de cada trabalhador só vão ter em consideração os dados processados no próprio, pelo que as contribuições obtidas pelos restantes trabalhadores não estariam a ser tidas em consideração. De modo a contornar esta situação é importante que tenhamos políticas de sincronismo, que ditem como é que a sincronização dos parâmetros deve ocorrer. Existem dois principais tipos de políticas de sincronismo [47, 50]:

- **síncronas** – a atualização dos parâmetros do modelo ocorre apenas quando todos os trabalhadores terminam o cálculo dos gradientes. Esta metodologia implica que os trabalhadores fiquem à espera uns dos outros no fim de cada iteração.
- **assíncronas** – a atualização dos parâmetros do modelo ocorre assim que um trabalhador termina o cálculo dos gradientes (gradientes locais). Permite que os trabalhadores iniciem as próximas iterações sem que todos tenham terminado a atual.

Estas políticas estão incorporadas nas estratégias disponibilizados pelas diversas ferramentas. Por exemplo, no caso do **TensorFlow (TF)**, estratégias como *MirroredStrategy* [18] e *MultiWorkerMirroredStrategy* [19] seguem uma abordagem síncrona, recorrendo a processos “*Ring All-Reduce*” para fazer a sincronização, sendo que se distinguem na medida em que enquanto a primeira apenas permite distribuição entre diversos **GPUs**, a segunda estratégia permite que os dados sejam distribuídos entre vários **GPUs** e máquinas diferentes. Já a estratégia *ParameterServerStrategy* [24, 25] recorre a uma abordagem assíncrona e permite distribuição dos dados por várias máquinas e **GPUs**.

Apesar de poder ser feita com diversas configurações e abordagens, a distribuição dos dados permite que o processo de treino convirja mais rapidamente.

Não obstante, como cada um dos trabalhadores tem de fazer a iteração com os dados que obteve, o modelo tem de caber na totalidade em cada um deles. Isto pode revelar-se uma limitação, uma vez que

modelos grandes são normalmente preferíveis, visto que tipicamente permitem uma análise e compreensão mais eficiente e pormenorizada.

Para além disso, temos uma maior sobrecarga da rede, resultante das mensagens trocadas para sincronização dos modelos.

Distribuição de dados também potencializa a existência de momentos de inatividade que resultam numa diminuição do desempenho. Estes momentos devem-se à necessidade de esperar por informações e costumam corresponder a momentos de atualização dos pesos do modelo, uma vez que é quando os trabalhadores estão dependentes de mensagens de terceiros (i.e., mensagens de outros trabalhadores ou do **SP** dependendo da estratégia que estiver a ser seguida).

Quando estamos a trabalhar com políticas de sincronização de modelo assíncronas é ainda possível a ocorrência de momentos de bloqueio. Estes acontecem quando alguns trabalhadores estão a utilizar pesos desatualizados para o cálculo dos gradientes (i.e., o **SP** calcula os novos pesos do modelo num momento N , mas recebe gradientes locais ocorridos no momento t , sendo $N > t > 0$) [50]. Fazer a atualização dos pesos com gradientes envelhecidos induz perda de acurácia e dissipação de recursos computacionais.

Distribuição de Modelo

Como ilustrado nas Figuras 5 e 6, enquanto que na Distribuição de Dados a mesma configuração é replicada diversas vezes e cada réplica é alimentada com diferentes blocos de dados cujo processamento é feito paralelamente, na Distribuição de Modelo o modelo é separado em diversos grupos que são distribuídos pelos trabalhadores disponíveis.

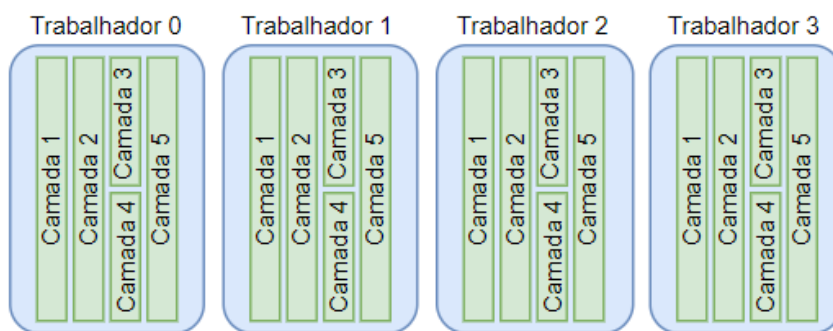


Figura 5: trabalhadores em distribuição de dados

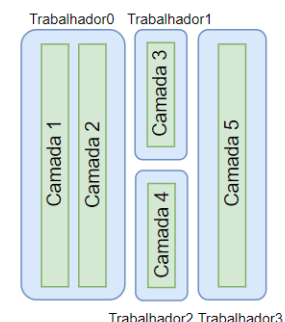


Figura 6: trabalhadores em distribuição de modelo

Como o modelo passa a estar disperso pelos vários trabalhadores deixamos de ter a limitação de que cada um deles tenha capacidade de armazenar em memória a totalidade do modelo. No entanto, como

existe dependência entre as diversas camadas, se o modelo for apenas distribuído pelos trabalhadores sem mais alterações, apenas teríamos um trabalhador ativo de cada vez.

Uma forma de contornar este problema é a implementação de um sistema de *pipeline* dentro de cada iteração. Este divide o bloco de dados (mini-conjunto) em mini-blocos (micro-conjunto) permitindo que possam existir períodos em que os vários trabalhadores estão ativos em simultâneo. Contudo:

- continuam a existir períodos em que trabalhadores estão parados, principalmente devido à necessidade de esperar pela retropropagação.
- a utilização de uma *pipeline* implica bastantes alterações ao modelo, uma vez que implica a reescrita do fluxo normal e que o *output* de uma camada tem de ser exatamente o *input* que a camada seguinte está preparada para receber.
- esta abordagem não permite ter controlos de fluxo ao nível dos estados da *pipeline*.

Uma abordagem similar, mas que vem resolver em grande parte os problemas anteriores, é a implementação de *pipelines* intercaladas. Estas já se encontram implementadas em ferramentas como o *Varuna* [36] e o *SageMaker* [2, 53] e baseiam-se na mesma ideia das *pipelines* anteriores mas procuram priorizar a execução da retropropagação de modo a evitar as bolhas de inatividades características das *pipelines* normais.

Acesso aos Dados

Quando estamos a trabalhar com paralelismo de dados cada trabalhador tem de recolher os dados que lhe competem para cada iteração, como se pode ver na Figura 7. Pelo que cada conjunto de dados só precisa de ser obtido por um trabalhador e por iteração. Contudo, se estivermos a usar um modelo com paralelismo de modelo, os diversos trabalhadores estão a trabalhar com o mesmo conjunto de dados, pelo que cada um vai precisar de os obter, como representado na Figura 8. Isto representa uma maior pressão dos pedidos de **E/S** no sistema de armazenamento.

Por outro lado, enquanto que com paralelismo de modelo em cada época cada trabalhador processa todos os dados do conjunto de treino, com o paralelismo de dados em cada época cada trabalhador só processa um sub-conjunto desses dados. Deste modo, com paralelismo de dados é mais complicado prever que amostras de dados é que cada nodo deve ler antes de estas serem pedidas.

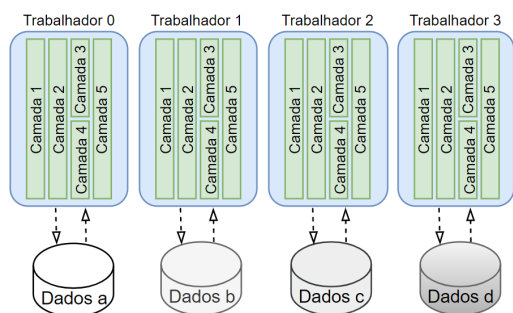


Figura 7: Acesso a dados em modelos com paralelismo de dados

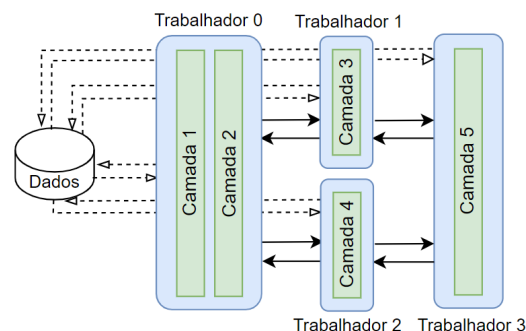


Figura 8: Acesso a dados em modelos com paralelismo de modelo

Ring All-reduce

Em execuções que sigam políticas de consistência síncronas, depois de todos os gradientes locais estarem calculados temos tantos gradientes quanto trabalhadores, mas para a atualização dos pesos do modelo recorreremos apenas a um. Para isso é necessário juntar os vários gradientes locais num único.

Uma das abordagens mais comuns que se pode seguir para este processo é o "Ring All-reduce". Esta tem 2 fases diferentes [66], uma primeira de "redução-dispersão" em que cada um dos gradientes locais é dividido em tantas partes quanto trabalhadores existentes e cada trabalhador envia um desses fragmentos a outro trabalhador. Ao receber um pedaço, cada trabalhador vai somá-lo ao seu pedaço de gradiente correspondente, como ilustrado na Figura 9. Esta fase termina quando todos os trabalhadores tiverem uma fração dos gradientes locais de todos os trabalhadores. Assim, implica menos uma iteração do que o número de trabalhadores envolvidos.

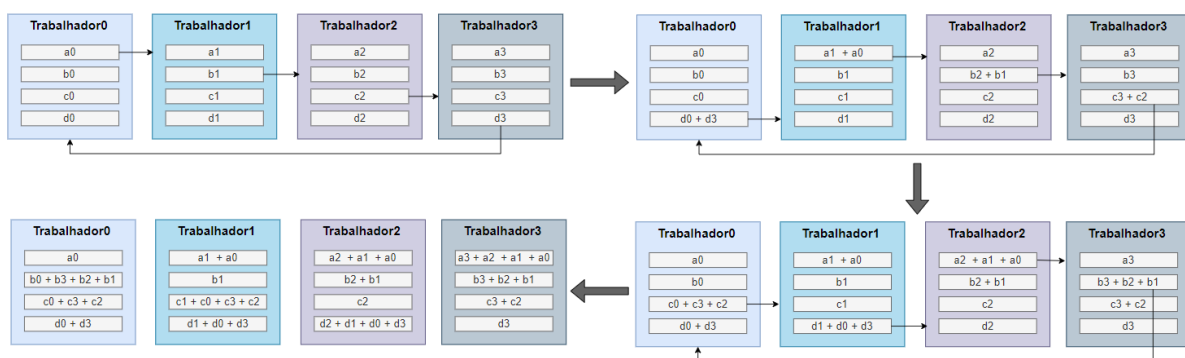


Figura 9: Exemplo de funcionamento da primeira fase do Ring All-reduce

A segunda, por sua vez, é uma fase de agregação [20], em que cada trabalhador envia, ao mesmo trabalhador que na fase anterior, o último pedaço que atualizou (uma vez que esse é o que tem a contribuição de todos os trabalhadores). Aquando da receção dessa porção cada trabalhador substitui a sua

pela que recebeu. Este processo repete-se até que todos os fragmentos tenham sido atualizados, como exemplificado na Figura 10.

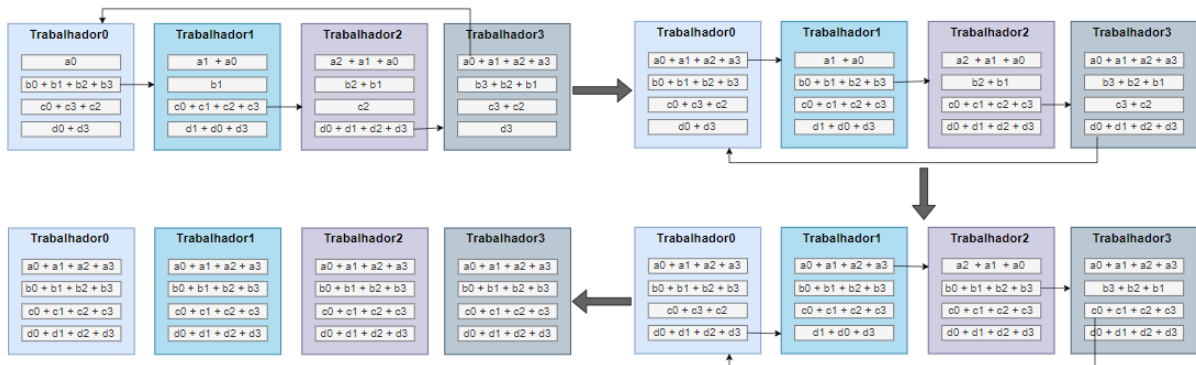


Figura 10: Exemplo de funcionamento da segunda fase do Ring All-reduce

Deste modo, quando este processo está completo, os gradientes locais dos vários trabalhadores são equivalentes, pelo que cada trabalhador já os pode utilizar para atualizar os pesos do seu modelo.

Arquitetura de Servidor de Parâmetros

Servidor de Parâmetros é uma arquitetura de distribuição de dados assíncrona em que cada nodo pode ter um de dois papéis: trabalhador ou **Servidor de Parâmetros (SP)** [Figura 11].

Os trabalhadores são responsáveis pela computação do modelo. Quando terminam de calcular o gradiente local enviam-no aos **SPs**. Estes têm parâmetros do modelo armazenados e ao receber um gradiente local de um trabalhador atualizam-nos e enviam os novos parâmetros para o trabalhador que mandou o gradiente, para que este possa iniciar a iteração seguinte.

A implementação desta arquitetura no **TF** (denominada *ParameterServerStrategy*) obriga a que um dos trabalhadores funcione como coordenador do programa. Este faz o papel de chefe e é responsável por distribuir o conjunto de dados pelos trabalhadores enviando-lhes pedidos [Figura 12]. Os trabalhadores e os **SPs** simplesmente correm um servidor padrão do **TensorFlow** que executa passivamente os pedidos que recebe.

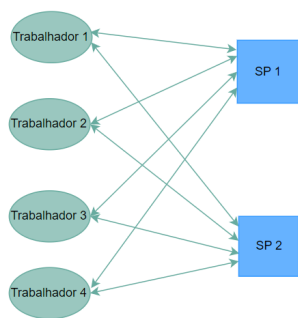


Figura 11: Exemplo de uma arquitetura de Servidor de Parâmetros

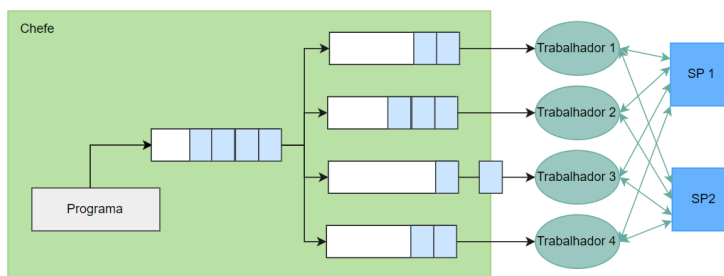


Figura 12: Exemplo de uma arquitetura de Servidor de Parâmetros do TensorFlow2

Como podemos ter vários trabalhadores e **SP**s esta arquitetura é tolerante a faltas de componentes deste tipo. Para além disso, a existência do chefe permite balanceamentos da carga de cada trabalhador e escalonamento dinâmico.

2.1.5 Aprendizagem Profunda Multimodelo

Enquanto que para as abordagens multi-nodo anteriores os diversos nodos procuravam acelerar o treino e permitir treinar modelos de maiores dimensões, a aprendizagem profunda multimodelo consiste em ter diversos modelos de **AP** (locais ou distribuídos) a treinar ao mesmo tempo, e permitindo assim paralelizar o seu tempo de treino. Isto é consideravelmente útil e comum, principalmente em cenários como, por exemplo, quando estamos à procura dos hiperparâmetros ideais de um modelo (uma vez que este processo implica treinar o modelo diversas vezes com diversas combinações de hiperparâmetros para concluir quais apresentam melhores métricas).

Os vários modelos são executados em nodos diferentes funcionando como explicado anteriormente, mas podem estar a aceder ao mesmo conjunto de dados. Como ao longo de cada época de treino cada modelo pede a totalidade do conjunto de dados e os vários modelos estão a correr paralelamente temos diversos pedidos de dados a concorrer uns com os outros, podendo levar à contenção da execução do treino dos diversos modelos.

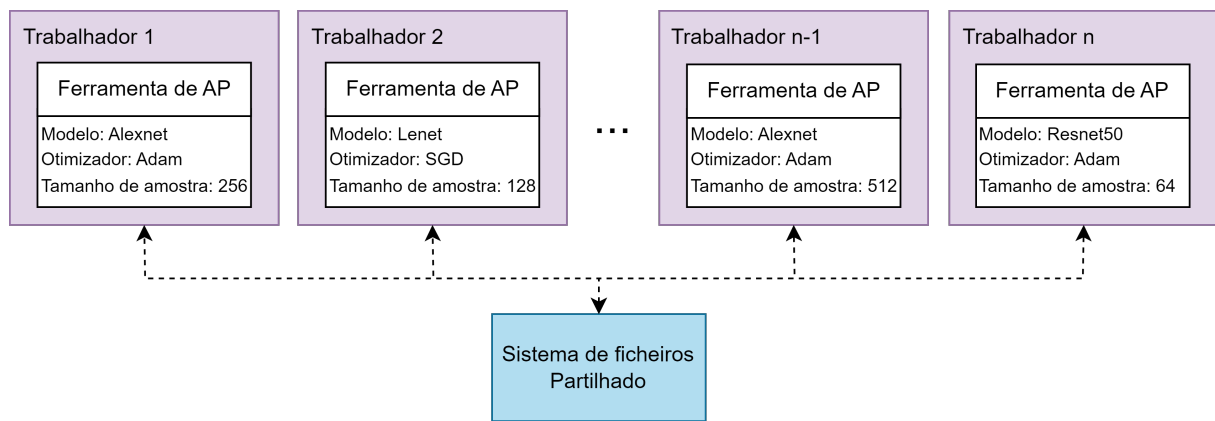


Figura 13: Exemplo de uma abordagem multimodelo.

2.1.6 HPC em Aprendizagem Profunda

Como referido anteriormente, os modelos de **AP** estão a ficar cada vez mais complexos e a consumir conjuntos de dados de maiores dimensões. Deste modo, estão a tornar-se mais exigentes quer ao nível de armazenamento requerido, quer ao nível da computação. No caso específico de modelos de **APD** temos diversos trabalhadores que têm de conseguir comunicar uns com os outros, pelo que o número de unidades de processamento disponíveis e a largura de banda da rede que as une também passam a ser fatores a ter em atenção. As infraestruturas **HPC** têm a capacidade de dividir tarefas e distribuí-las por diversos *cores* dentro de cada nodo computacional e de distribuir a computação entre vários nodos computacionais e contam com servidores potentes e recursos como **GPUs**, pelo que são bastante convenientes para a execução destes programas. O sistema de armazenamento destas infraestruturas é baseado num sistema de ficheiros partilhado entre todos os nodos. Este permite um acesso centralizado a dados, ou seja, permite que utilizadores acessem de forma simples e rápido a dados que podem estar persistidos em diversos dispositivos de armazenamento, sem que seja necessário ao utilizador ter conhecimento de qual está a utilizar ou das diferenças entre eles. Para além disso, alguns supercomputadores contam ainda com armazenamento local em cada nodo. No entanto, assim que um nodo deixa de estar alocado a um utilizador o conteúdo do disco local é removido e normalmente o conjunto de dados de treino não cabe no armazenamento local de cada nodo, pelo que é colocado no sistema de ficheiros partilhado. Isso implica que os pedidos de leitura dos programas de **AP** sejam feitos diretamente ao sistema de ficheiros, mas este não está otimizado para o tipo de carga de trabalho que estes produzem (as infraestruturas **HPC** necessitam de metadados para fazer a associação entre os recursos de armazenamento e os endereços que os utilizadores utilizam, mas estes não escalam para as leituras aleatórias de diversos ficheiros de pequenas dimensões características de programas de **AP** resultando em problemas de escalabilidade).

do sistema de armazenamento). Disto resulta um pior desempenho na resposta aos pedidos **E/S**, que neste caso corresponde a tempo em que a computação dos modelos está parada, afetando o tempo de execução destes programas.

2.2 Trabalho Relacionado

Diversos estudos e sistemas foram feitos com o objetivo de melhorar o desempenho de **IA**. Alguns, procuram aumentar ao máximo a taxa de utilização dos recursos disponíveis (**GPU**, **CPU**, etc.), recorrendo para isso à partilha de recursos e ao escalonamento dos mesmos [58, 59, 62, 70, 72, 76, 77].

Por outro lado, outros focam-se em diminuir ou remover o gargalo que operações de **E/S** do sistema de armazenamento podem originar no treino de programas **Aprendizagem Profunda (AP)** e/ou **APD**. Estes vão encontro ao objetivo desta tese, pelo que, em baixo são expostos e explicados os mais relevantes.

2.2.1 Otimizações de Pré-processamento e Carregamento de Dados

Algumas propostas vêm otimizar o processo de pré-processamento de dados feito pelas ferramentas de **IA**.

O PRISMA [60] propõe uma arquitetura de armazenamento definido por software que procura acelerar o processo de treino de programas de **AP**. Este sistema sabe a ordem em que os ficheiros vão ser pedidos pelo programa, uma vez que essa é definida no início de cada iteração. Isto permite que o plano de dados (que é a parte responsável pela implementação da lógica de **E/S**) vá buscar para memória os dados antes de eles serem pedidos. O plano de controlo (que é a parte do programa responsável por gerir a lógica de **E/S** com base em regras definidas pelo utilizador) vai ajustando este processo variando o número de *threads* que estão a ler do sistema de armazenamento e adaptando o tamanho do *buffer* onde as amostras de dados lidas são guardadas até serem pedidas.

O DALI [4] é um substituto dos carregadores de dados das ferramentas de **IA**. Este permite que o processamento dos dados (como decodificação, corte ou redimensionamento) passe a ser feito em **GPU**, diminuindo ou removendo o gargalo causado pelo **CPU**, de forma transparente. Utiliza ainda técnicas de otimização de carregamento de dados como execução paralela e obtenção prévia de amostras de dados.

O Svogor et al. [69] propõe a paralelização da obtenção de dados de uma amostra no Pytorch. O carregador de dados desta ferramenta de **AP** já faz a paralelização de amostras, no entanto cada amostra é carregada de forma sequencial. Para resolver esta situação, foi adicionada uma nova camada de concorrência na qual em vez de cada nodo ter um carregador de dados passa a ter múltiplos para

aceder paralelamente aos dados da amostra.

O `tf.data.service` [37] propõe o desacoplamento do pré-processamento da computação dos dados para **TF**, uma vez que o processamento dos dados pode causar um gargalo na execução, levando a desperdício de recursos computacionais. Esta separação permite que o pré-processamento dos dados seja distribuído por vários **CPU**s e que os recursos alocados a cada programa de **AM** possam ser ajustados de modo a evitar que haja recursos parados. Cada nodo lê dados do armazenamento remoto, aplica-lhes as transformações devidas e guarda-os depois num *buffer*, de modo a estarem prontos para serem processados. Estes são posteriormente enviados para o programa de **AP** quando receber um pedido **RPC**. Para além disso, cada nodo utiliza uma *Sliding Window Cache* para registar os dados que já processou. Assim, se vários programas de **AP** concorrentes estiverem a correr sobre o mesmo conjunto de dados, esta abordagem permite aproveitar o pré-processamento feito, em vez de o repetir para cada um deles.

TFRECORDS [30] e RecordIO [26] são formatos de dados que aglomeram o conjunto de dados em menos ficheiros, de maior dimensão do que os originais que normalmente são constituídos por diversos ficheiros de pequenas dimensões divididos por várias diretorias. Isto permite aumentar as leituras sequenciais e reduzir as operações de metadados, o que resulta num aumento da eficiência do processo de obtenção de dados. Apesar de algumas ferramentas de **IA** já importarem certos conjuntos de dados neste formato, quando tal não acontece é necessário, por um lado, converter o conjunto de dados para este formato antes de poder treinar e, por outro, adaptar o código do programa de **IA** para ter em consideração a nova organização dos dados. Adicionalmente, no caso dos TFRecords como passamos a ter diversas amostras diferentes no mesmo ficheiro e sem nenhuma indexação adicional, deixa de ser possível aceder a amostras de dados do mesmo ficheiro de forma aleatória.

2.2.2 Otimizações de Acesso a Dados: Caching

Outras propostas procuram otimizar o processo de obtenção de dados, recorrendo a *caching* dos mesmos. Neste contexto estamos a considerar *caching* o processo de guardar dados, seja em memória ou numa camada de armazenamento, de forma a facilitar o acesso a estes. Isto permite reduzir o número de leituras necessárias, tornando assim a resposta mais eficiente.

O CoordDL [61] propõe um sistema que recorre a uma *cache* em memória especializada (MinIO). Esta foi desenvolvida com base nos padrões de acesso a memória típico do treino de modelos de **AP** de modo a diminuir os períodos em que a computação está parada à espera de dados. Quando estamos a treinar um modelo localmente no decorrer de cada época todos os dados do conjunto de treino são lidos, tendo

por isso a mesma probabilidade de serem pedidos. Este sistema aproveita essa propriedade e ao longo da primeira época vai preenchendo a *cache* com os dados que são pedidos e uma vez que esta atinja a sua capacidade máxima permanece inalterada, não havendo assim remoção de dados da *cache*.

Já no cenário distribuído, os dados são repartidos pelos nodos em cada época deixando de haver garantias que todos eles vão tentar ler dados que estão em *cache*. Para contornar esta situação este sistema disponibiliza *caches* particionadas que permitem que os nodos acedam remotamente a ficheiros uns dos outros. Para tal, é necessária a existência de metadados que façam um mapeamento da distribuição dos dados pelas *caches* dos nodos. Adicionalmente, este sistema tem ainda a capacidade de procurar os hiper-parâmetros mais apropriados para o modelo a ser treinado, levando a cabo simulações com amostras de dados já processadas.

O DIESEL [71] propõe um sistema de armazenamento e *caching* que procura agregar e converter pequenos ficheiros de dados de treino (e respetivos metadados) em blocos (*chunks*) de dados de maiores dimensões e o *shuffle* passa a ser feito entre blocos. Isto permite leituras de dados mais eficiente (semelhante aos TFRecords do TF).

O DLFS [79] propõe a desagregação de armazenamento para melhorar a eficiência de RNP. Para isso, aloca uma coleção de recursos de armazenamento, locais e/ou remotos, e prepara o conjunto de dados que vai ser utilizado para o treino de modelos. Ao utilizar NVMe-oF [21] ao nível do utilizador, com base no SPDK (*Storage Performance Development Kit*) [29], este permite que várias máquinas diferentes possam aceder aos vários dispositivos NVMe. Para saber que dados é que cada dispositivo tem, o sistema mantém um mapeamento, no formato de uma árvore AVL balanceada, dos mesmos em memória.

O Plumber [56] propõe uma ferramenta que recorre a modelação analítica para encontrar gargalos de execução na pipeline de entrada de programas AP. Esta inicialmente começa por avaliar o desempenho e utilização de recursos (como disco, CPU, etc.). Com as informações obtidas, cria um gráfico que descreve o comportamento do programa e faz otimizações, como introdução de *caching*, *prefetching* e/ou paralelismo, sem a intervenção do utilizador. É ainda possível adicionar outras otimizações às anteriormente referidas.

O One Access [52] propõe uma camada unificada de acesso a dados, ou seja, este artigo procura juntar o processo de carregamento (e idealmente de pré-processamento) de dados de vários programas de AP num único sistema. Para isto este sistema fica entre os programas de AP e o sistema de armazenamento e atua em duas fases. A primeira recorre a *reservoir sampling*, obtendo dados aleatórios (fazendo, no entanto, acessos sequenciais) e fazendo o seu pré-processamento. Na segunda são geradas amostras com base nesses dados que são depois passadas aos programas de AP. Isto permite evitar fazer leituras

verdadeiramente aleatórias e ter de repetir leituras e pré-processamento de dados para vários programas.

O **LDL** [73] propõe uma solução que recorre a *caching* distribuído para acelerar o carregamento de dados em programas de treino distribuído em grande escala. No algoritmo **Locality-aware Data Loading (LDL)** cada trabalhador povoa a sua *cache* local e não há substituição dos dados da *cache*, pelo que uma vez cheia não é mais alterada. Como em treino distribuído o carregamento da amostra de dados é distribuído pelos vários trabalhadores, cada trabalhador vai verificar quais dos dados da amostra pedida é que possui na sua *cache* local, no entanto, isto pode desbalancear o processo. Para combater isso é adicionada uma fase de balanceamento, na qual é definido que dados devem ser carregados por cada nodo. Cada nodo começa por verificar se esses dados estão na sua *cache* local, em caso negativo, pode pedir esses dados a outro nodo, caso esteja na *cache* global, ou então pode ler os dados do sistema de armazenamento. É ainda de salientar que no processo de distribuir os dados da amostra pelos vários trabalhadores a ordem deles dentro da amostra não é a originalmente pedida pelo programa de **APD**.

Otimizações de acesso a dados: Substituição de dados

Alguns dos sistemas que implementam *caching* para programas de **AP** exploram uma abordagem de substituição dos dados pedidos pela ferramenta de **IA** por dados que já tenham sido previamente obtidos e guardados na *cache*. Isto permite que os dados estejam prontos a ser devolvidos, diminuindo assim o tempo que o programa tem de esperar por eles. No entanto, alterar a ordem em que os dados são usados no treino acaba por afetar a acurácia do modelo resultante.

O Quiver [57] propõe a utilização de uma *cache* distribuída para as amostras de dados consumidas por programas de **AP**. Esta permite partilhar dados entre diversos programas e/ou utilizadores de forma segura (procurando que não haja vazamento indevido de dados e baseando a indexação da *cache* nos conteúdos das entradas), sendo que a prioridade de alocação de *cache* dos vários programas é definida dinamicamente. Para além disso, descarta da *cache* dados que já tenham sido utilizados por todos os programas e faz substituição de amostra de dados devolvendo amostras selecionadas aleatoriamente entre as guardadas em *cache*. Por um lado, essa substituição evita que o programa tenha de ficar à espera de que esses dados sejam lidos de disco, quando não estão em memória, mas por outro diminui o impacto positivo do baralhamento dos dados (podendo dificultar a convergência e aumentar a probabilidade de sobre-ajuste do modelo resultante).

O DeepIO [78] propõe um sistema desenhado para o Tensorflow que procura diminuir o número de operações de leitura aleatórias e de pequena dimensão executadas por programas de **AP**. Para tal, este guarda os dados em memória e responde aos pedidos de leitura com estes. No caso de o conjunto de

dados não caber em memória, este sistema utiliza uma lógica de EOO (*Entropy-aware Opportunistic Ordering*), que permite escolher que amostra de dados é passada ao programa de **AP**, escolhendo elementos que estejam em memória.

O Shade [55] propõe um sistema de *caching*, para programas de **APD**, implementado no *PyTorch* que introduz o conceito de classificação de dados e de geração de amostras com base na sua prioridade. A classificação de dados consiste numa avaliação da importância relativa dos dados, tendo em consideração a relevância destes para a acurácia do modelo. Esta é utilizada para decidir que dados manter em *cache*, evitando fazer remoções aleatórias. Estas classificações vão sendo atualizadas ao longo do processo de treino, pelo que estes são avaliados em diversas amostras e mini-amostras, potencializando assim uma classificação mais correta. A política de criação de amostras baseada na “prioridade” dos dados baseia-se nessas mesmas classificações e garante que dados mais importantes para a acurácia do modelo são acedidos várias vezes. Combinadas estas melhoram significativamente a probabilidade de os dados pedidos estarem na *cache*, diminuindo assim o tempo de resposta aos pedidos de obtenção de dados e permitindo assim diminuir os tempos de treino do modelo.

O iCache [40] propõe um sistema de *caching*, implementado no *PyTorch*, que se baseia na importância dos dados para o modelo resultante para gerir os dados guardados. Este sistema conta com duas regiões distintas, *H-cache* e *L-cache*, em que a primeira guarda os dados com elevada importância e a segunda os de menor. Os dados da *H-cache* são mantidos atualizados com base a sua importância, sendo que se surgirem dados com maior importância, os dados com menor classificação guardados são substituídos. Quando um programa de **IA** pede dados de reduzida importância que não estão na *L-cache* estes são substituídos por dados que lá estejam, diminuindo assim o número de leituras feitas do sistema de armazenamento e permitindo respostas mais rápidas aos pedidos de obtenção de dados. Adicionalmente, este sistema permite gerir os pedidos de dados de vários programas de **IA**, avaliando a relação custo-eficácia de guardar dados para cada um desses programas e calcula a importância dos dados pedidos por estes com base na importância destes para os programas que considera beneficiarem de *caching* dos dados.

2.2.3 Otimizações de Obtenção de Dados: Aproveitamento da Estratificação do Sistema de Armazenamento

Por fim, existe a alternativa de otimizar o processo de obtenção de dados recorrendo a *tiering*, ou seja, com base no aproveitamento da hierarquia do sistema de armazenamento. Estes procuram tirar proveito da existência de diversas camadas (como memória, disco local e sistema de ficheiros partilhado) e com

diferentes características (como capacidade e velocidade de acesso), guardando alguns dos dados em camadas superiores, capazes de responder mais rapidamente aos pedidos de dados.

O Monarch [43] propõem um sistema que tira partido das várias camadas do sistema de armazenamento de infraestruturas **HPC** para melhorar os tempos de execução e remover pressão dos pedidos de **E/S** do sistema de ficheiros partilhado. Este recorre à variável de ambiente **LD_PRELOAD** para interceptar e reimplementar as chamadas **POSIX** feitas pelos programas de **AP**. Isto permite que seja agnóstico à ferramenta de **AP** e transparente (não implicando alterações ao código de **AP**). Ao longo da primeira época os dados vão sendo copiados (em segundo plano) da última camada (como o sistema de ficheiros partilhados) para as restantes camadas (como o disco local do nodo). Como em cada época todos os dados vão ser processados exatamente uma vez, a probabilidade de ser lida é exatamente a mesma para todas elas. Por este motivo os ficheiros começam por ser colocados na primeira camada de armazenamento (a mais rápida) e quando atinge o limite da capacidade desta camada, começa a povoar a camada imediatamente abaixo, não havendo remoções nem substituições. É ainda de salientar que quando é pedida uma amostra de um ficheiro de grandes dimensões, a amostra é enviada na resposta, mas o *Monarch* fica a carregar o resto do ficheiro para o nível adequado de armazenamento, como se pode verificar no fluxo da Figura 14. Nas épocas subsequentes as leituras são feitas da primeira camada da hierarquia de armazenamento que tem os dados pedidos, recorrendo a metadados para fazer o mapeamento entre os caminhos que o programa **AP** usa para pedir dados e a posição para o qual o ficheiro foi copiado. Assim, este consegue evitar ir sempre à última camada e, consequentemente, diminuir o tempo de resposta à ferramenta de **AP**.

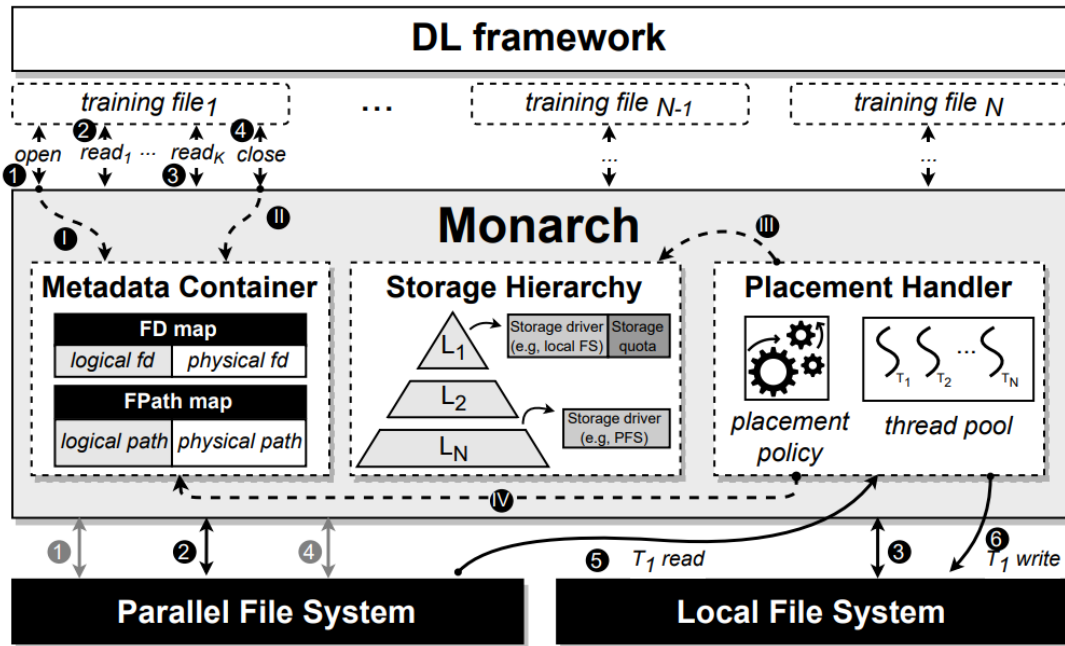


Figura 14: Arquitetura do *Monarch* (retirado de [43])

O **HVAC** [54] propõe uma solução de *caching* distribuída que aproveita o armazenamento local de cada nodo e/ou nodos próximos. Ao correr um programa de **IA** com esta solução, é alocada em cada nodo uma instância do servidor **HVAC** (usando o armazenamento local). Este vai ter uma *thread* dedicada a gerir uma fila (**FIFO**) que monitoriza e faz a gestão das operações de **E/S** geradas pelo programa de **AP**. Estas operações são executadas pelo **HVAC daemon** que as retira da fila e guarda os dados no armazenamento local do nodo. O **HVAC** particiona o conjunto de dados entre os vários nodos e funciona como *cache* de leitura, não permitindo escritas. Isto elimina impacto de desempenho significativo na utilização de metadados (em processos como verificações, *locking*, etc.). Não obstante, caso o espaço combinado dos vários nodos não seja suficiente, é possível ativar a substituição de dados da *cache* quer de forma aleatória, quer seguindo políticas específicas.

O NoPFS [38] (NoPFS) propõe um sistema que com base no valor utilizado para inserir aleatoriedade consegue prever a ordem pela qual as amostras de dados vão ser pedidas. Deste modo é possível carregar os ficheiros para memória antes de serem pedidos. Para além disso, concluíram que para cada trabalhador há amostras de dados que têm uma maior probabilidade de serem pedidas. Deste modo, sabendo quais são é possível colocar essas amostras em camadas da hierarquia de armazenamento mais rápidas, tirando assim melhor proveito dos recursos disponíveis e diminuindo o tempo de leitura dos dados.

O FanStore [75] propõe um sistema de ficheiros partilhado efêmero que apenas existe durante o

tempo de execução do programa de **AP**. Para tal, conta com uma ou mais *threads* que tratam dos pedidos de dados do programa **APD**, sendo que estas *threads* manipulam os metadados (i.e., o mapeamento dos ficheiros de treino que é transmitido entre nodos) guardados localmente para conseguir obter os dados necessários, sendo que estes podem estar armazenados localmente ou em outros nodos. No segundo caso, os dados são obtidos, recorrendo a **MPI** para permitir acesso remoto aos ficheiros onde estes se encontram. No entanto, este sistema necessita de um passo extra antes de iniciar o treino em que o utilizador passa a lista de todos os ficheiros de treino, para que estes os possa processar e reorganizar em partições mapeadas nos metadados.

O Serizawa and Tatebe [67] propõe uma classe de carregamento de dados para a ferramenta de **AP** Chainer DL [3] que recorre a *pipelines* para esconder o tempo gasto a copiar o conjunto de dados para disco local, sobrepondo-o com o de leitura dos dados. Para isto, recorre a 3 fases, na primeira é gerada uma lista de índices que define os dados que vão constituir as amostras. Na segunda essa lista é partilhada entre os vários processos para que estes possam obter estes dados e colocá-los no armazenamento local (paralelizando a cópia destes). Na terceira fase os dados são lidos do armazenamento local. Estas fases funcionam de forma assíncrona e sobrepõem-se. No entanto, não considera o caso de o disco local não ter capacidade para armazenar o conjunto de dados.

2.2.4 Resumo

Os programas de **AP** implicam elevada capacidade computacional e de armazenamento, beneficiando por isso de ser executados em infraestruturas **HPC**. No entanto, devido às dimensões dos conjuntos de dados, estes são normalmente colocados no sistema de ficheiros partilhado que é comum a todos os utilizadores. Como este não está preparado nem para padrões de acesso aleatórios, nem para receber um elevado número de pedidos, de relativamente pequenas dimensões, a serem feitos em simultâneo, pode resultar numa maior pressão **E/S** e, consequentemente, em maiores tempos de resposta às leituras, aumentando o tempo de execução dos programas que delas dependem.

De facto, diversos estudos vêm mostrar que o acesso a dados funciona como fator limitante do desempenho de programas **AP** [41, 42, 46, 61]. Com isto em mente, as ferramentas de **AP** já disponibilizam mecanismos para otimizar a forma como o carregamento dos dados é feito, no entanto o gargalo no desempenho manteve-se. Assim, foram desenvolvidas diversas otimizações que procuram diminuir o impacto do armazenamento na execução de programas **AP**, como:

- partilhar e agendar os recursos disponíveis (**GPU**, **CPU**, etc.) [58, 59, 62, 70, 72, 76, 77].

- alterar o formato em que os ficheiros são persistidos no sistema de armazenamento, de forma a potenciar leituras contíguas e de maiores dimensões [26, 30].
- passar os dados lidos para o GPU antes de terem sido pré-processados [4].
- otimizar a forma como os dados são carregados, recorrendo a *pipelines* de leitura que procuram obter as amostras de dados antes de a aplicação as pedir [37, 60, 69].
- implementar ferramentas que utilizem os recursos existentes para otimizar o acesso a dados.

Esta dissertação procura aproveitar os recursos de armazenamento disponíveis para reduzir a pressão dos pedidos de E/S do sistema de armazenamento e acelerar o treino alterando para isso a forma como as leituras são feitas. Assim os sistemas abordados nos quatros primeiros pontos não só não competem com o sistema desenvolvido, como podem ser utilizados em combinação com ele.

O sistema desenvolvido enquadra-se no quinto tipo de otimizações. Esta engloba 3 tipos de sistemas com otimizações diferentes. Alguns destes focam-se em desenvolver mecanismos de *caching* de dados para AP [56, 61, 67, 71, 73, 79]. No entanto, ao contrário desta dissertação (i) vários destes sistemas são feitos para funcionar para ferramentas de AP específicas e implicam alterações significativas no código do programa de AP [61, 67, 71, 73, 79], (ii) alguns deles são incompatíveis com as otimizações de carregamento de dados e/ou com *buffering* parcial do conjunto de dados [67], (iii) alguns deles implicam recursos adicionais para tratamento dos dados e utilização de metadados [71], (iv) alguns destes sistemas não estão adaptados a abordagens multi-nodo [56, 71]. Alguns destes sistemas recorrem a mecanismos de substituição de dados [57, 78]. Tal como o sistema desenvolvido nesta dissertação, os sistemas baseados em substituição de dados estão normalmente adaptados a abordagens multi-nodo, no entanto, são intrusivos no código da ferramenta de AP e por alterarem a ordem em que os dados são passados aos programas de IA podem afetar negativamente a acurácia dos modelos resultantes.

Por último alguns sistemas procuram aproveitar as camadas do sistema de armazenamento para melhorar os tempos de obtenção das amostras de dados [38, 43, 54, 75]. Esta dissertação procura seguir esta abordagem, no entanto, algumas destas ferramentas são intrusivas e incompatíveis com otimizações modulares (como a obtenção pró-ativa de dados feita pelas ferramentas de AP ou por ferramentas externas [4]) [38], outras dependem de fases complexas de preparação de dados e metadados [75], enquanto que outras não estão adaptadas a abordagens multi-nodo [43] ou fazem a distribuição dos dados pelos nodos com base em sistemas de *hashing* em vez de terem em consideração se o nodo que fica com eles no disco local os utiliza e coloca todos os pedidos de E/S de dados dos programas numa fila FIFO [54].

		Local	Multi-Nodo	Intrusivo e/ou específico	Altera a ordem de consumo de dados	Incompatível com algumas das otimizações de carregamento de dados	Necessita de recursos adicionais
Caching	CoordDL [61]	✓	✓	✗			
	DIESEL [71]	✓		✗			✗
	DLFS [79]	✓	✓	✗			
	Plumber [56]	✓		✗		✗	
	One Access [52]		✓				✗
	LDL [73]		✓	✗	✗		
	Quiver [57]		✓	✗	✗		
	DeepIO [78]		✓	✗	✗		
	Shade [55]		✓	✗	✗		
	iCache [40]	✓	✓	✗	✗		
Tiering	Monarch [43]	✓					
	NoPFS [38]	✓	✓	✗		✗	
	HVAC [54]	✓	✓				
	FanStore [75]	✓	✓				✗
	Serizawa and Tatebe [67]	✓		✗		✗	

Tabela 1: Tabela de sistemas de otimizações de obtenção de dados e respetivos problemas.

Com base nisto, concluímos que é relevante desenvolver uma ferramenta que tire partido do sistema de armazenamento por camadas para melhorar o desempenho de programas de **AP** em abordagens multi-nodo. Esta deve ser agnóstica à ferramenta de **AP** e não deve implicar alterações no código desses programas. Deve ainda ter um impacto mínimo nas métricas de avaliação dos modelos resultantes e permitir a utilização de otimizações modulares.

Como referido acima o sistema *Monarch* [43] trata de diversos dos objetivos definidos para este sistema (secção 1.2), como melhorar o desempenho de treino de programas de **Aprendizagem Profunda** (diminuindo o tempo de execução) e minimizar a interferência dos pedidos **E/S** ao sistema de ficheiros partilhado. No entanto, esta ferramenta foi desenvolvida a pensar em programas **AP** locais em que o treino é feito em apenas um nodo (treino local), enquanto que no contexto desta dissertação o foco está em **AP** multi-nodo. Ao ter vários nodos a treinar, todos eles vão contar com um disco local que pode ser aproveitado para guardar partes do conjunto de dados. No entanto, o *Monarch* não tem isso em consideração, não tirando, por conseguinte, o melhor proveito do espaço disponível. Deste modo, o sistema a desenvolver vai-se basear neste, mas adaptando-se às peculiaridades de abordagens multi-nodo.

Capítulo 3

Testes Preliminares

De modo adaptar o sistema a desenvolver ao funcionamento de programas de **APD** é necessário primeiro entender melhor como é que os pedidos de **E/S** são feitos e aferir a eficiência da troca de dados entre trabalhadores.

Estes testes foram executados numa infraestrutura HPC, uma vez que, como explicado no capítulo anterior, estas infraestruturas estão bastante adaptadas às necessidades computacionais de programas de **AP**. O supercomputador utilizado foi o Frontera [68] do **Texas Advanced Computing Center (TACC)** [31] em que utilizamos nodos com GPU (nodos RTX), com as especificações da Tabela 2.

Recurso	Especificações
CPUs	2x 16-core Intel Xeon E5-2620 v4 (“Broadwell”)
GPUs	4x Nvidia Quadro RTX 5000 16GB GDDR6
RAM	128 GiB DDR4
Disco Local	partição de 144GiB num SSD de 240GiB
Sistema de ficheiros Partilhado	Lustre
Sistema Operativo	CentOS 7.9
Kernel	versão 3.10
Sistema de Ficheiros	XFS
GCC	versão 9.1.0
CMAKE	versão 3.24.2
Python	versão 3.8.2
Tensorflow	versão 2.10.1
CUDA	versão 11.3
Cudnn	versão 8.2.1
NCCL	versão 2.9.9

Tabela 2: Especificações do ambiente experimental

3.1 Tempo de Leitura do Sistema de Ficheiros Paralelo, do Armazenamento Local ou de Trocar Dados Entre Nodos

Considerando que a cada época vários trabalhadores leem ficheiros diferentes e seguindo uma abordagem como a do *Monarch* [43], em que puxamos para o disco local de cada trabalhador os dados lidos por ele na primeira época (enquanto houver espaço para eles), seria possível os trabalhadores trocarem dados entre si, reduzindo ainda mais o número de leituras ao sistema de ficheiros paralelo necessárias. No entanto, é relevante prever como é que a substituição de leituras ao sistema de ficheiros paralelo por troca de dados com um outro trabalhador que tem esses dados guardados em disco local o afetaria. Para este fim testamos o débito e tempo de execução de ambas as operações.

3.1.1 Configurações dos Testes

Para este teste executamos 4 cenários diferentes de obtenção dos dados do conjunto de dados *Imagenet*.

- Cenário 1: os dados estão guardados no sistema de ficheiros paralelo (Lustre [48]) e lidos de lá;
- Cenário 2: os dados estão guardados no armazenamento local e lidos de lá;
- Cenário 3: os dados estão guardados no armazenamento local de um nodo e outro nodo liga-se via **RPC**, usando *infiniband*, para lhe pedir dados;
- Cenário 4: procura isolar o impacto da troca de dados pela rede, evitando a leitura de disco local. Assim, é similar ao Cenário 3, mas em vez de o nodo que tem os dados os ler do disco local simplesmente devolve uma *string* com o tamanho correto que foi previamente gerada e que está guardada em memória.

Cada um destes cenários foi testado com amostras de dados de 6 tamanhos distintos: 256 **KiBs**, 512 **KiBs**, 1 **MiBs**, 50 **MiBs**, 100 **MiBs** e o tamanho do maior TFRecord do Imagenet, ou seja, aproximadamente 119 **MiBs**.

3.1.2 Resultados

Para avaliar as várias combinações de configurações anteriormente apresentadas corremos três testes. Em todos os testes procuramos descobrir o débito (em Mebibytes por segundo) e o tempo de execução (em segundos) de cada operação de obtenção de dados, de modo a podermos comparar o desempenho das várias abordagens para os diversos tamanhos.

Os gráficos da Figura 15 apresentam a latência e débito médios das operações de obtenção de dados para amostras de 256 KiBs, 512 KiBs e 1 MiBs nos quatro cenários. Comparando as colunas dos cenários 1 e 2 é possível concluir que leituras do armazenamento local apresentam menor latência que as feitas ao sistema de ficheiros paralelo. Por outro lado, analisando as colunas dos cenários 2 e 3 verificamos que a troca de dados entre nodos é semelhante a ler diretamente do disco local e, consequentemente, mais rápido do que ler do sistema de armazenamento paralelo.

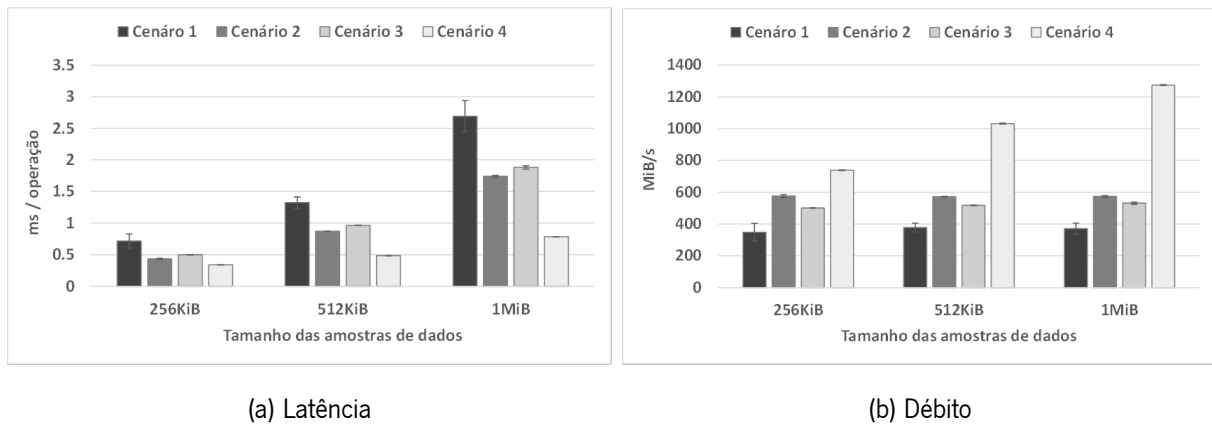


Figura 15: Débito e latência médios de obtenção de dados para 256 KiBs, 512 KiBs e 1 MiBs de dados.

Os gráficos da Figura 16 apresentam a latência e débito médios das operações de obtenção de dados para amostras de 50 MiBs, 100 MiBs e aproximadamente 119 MiBs nos quatro cenários. Comparando as colunas dos dois primeiros cenários verifica-se que ler do armazenamento local continua a ser mais eficiente. No entanto, para amostras com estas dimensões a troca de dados entre nodos (cenário 3) passa a ser comparável a ler do Lustre.

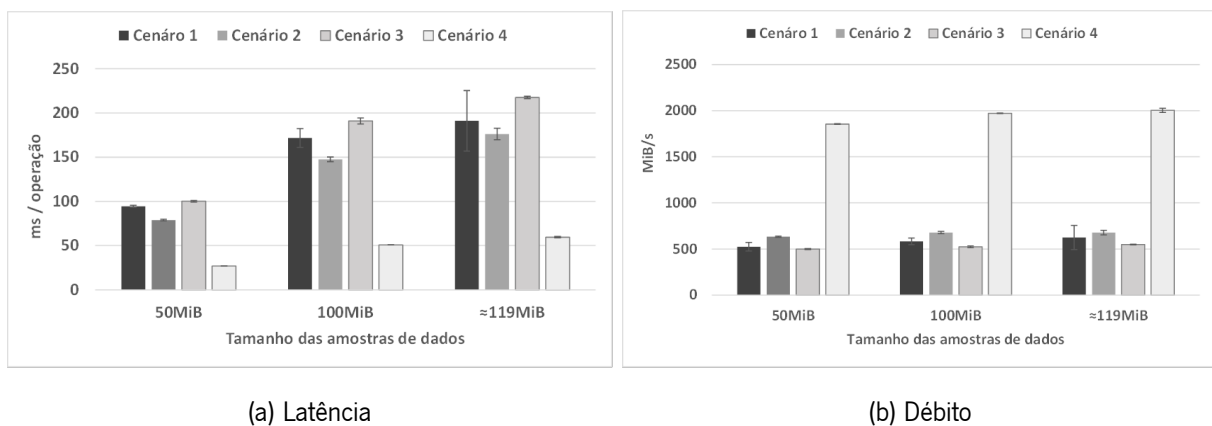


Figura 16: Débito e latência médios de obtenção de dados para os 50 MiBs, 100 MiBs e 119 MiBs de dados.

Recorrendo agora ao cenário 4, de ambas as figuras, é possível confirmar que a diferença de latência no cenário 3 entre as duas figuras resulta do impacto do envio dos dados pela rede, que se torna mais significativo à medida que aumentamos o tamanho das amostras de dados. Deste modo, quanto maior a dimensão do conteúdo a enviar, maior é o peso que esta operação tem na rede, resultando assim num aumento do tempo de partilha de dados entre os nodos e na redução do débito do acesso aos dados. Assim, como as amostras de dados utilizadas em **AP** normalmente se encontram dentro dos valores abrangidos na Figura 15, ao pedir dados a outros nodos que os tenham no disco local, não só diminuámos o número de acessos ao Lustre, como conseguimos obter esses mesmos dados num menor período de tempo, tornando assim o processo de obtenção de dados mais eficiente.

3.2 Estudo dos Pedidos E/S

De modo a compreender melhor como é que ferramentas de **AP**, em abordagens multi-nodo, efetuam os pedidos de **E/S** ao sistema de armazenamento, precisamos de analisar como é que estes são feitos ao longo do treino. Este estudo apresenta diversos testes, em que treinamos vários modelos extraindo diversas métricas que nos permitem inferir como é que esses acessos estão a ser feitos e quais são os fatores limitadores das várias estratégias e modelos.

3.2.1 Configurações dos Testes

Nestes testes procuramos analisar o padrão de acesso a dados para duas abordagens multi-nodo diferentes: vários treinos de modelos locais simultâneos e treino distribuído.

Para a versão de vários treinos de modelos locais testamos duas abordagens diferentes de cargas de trabalho: uma com carga de trabalho heterogénea correndo múltiplos modelos diferentes e uma com carga de trabalho homogénea em que corremos vários programas diferentes a treinar o mesmo modelo com hiperparâmetros diferentes. Para os testes de carga de trabalho heterogénea corremos os modelos *AlexNet*, *LeNet*, *ResNet18*, *Shufflenet*, *Inception-v3* e *VGG19* simultaneamente, mas em nodos computacionais diferentes. Os dois primeiros foram escolhidos por serem modelos cujo treino é limitado pelo gargalo causado pela obtenção dos dados [43], enquanto que os restantes foram selecionados por serem modelos mais recentes e bastante utilizados. Para todos eles o tamanho de amostra de dados foi definido como 512 **KiBs** e o otimizador utilizado foi Adam com a taxa de aprendizagem a 0,1. Nos testes de carga de trabalho homogénea corremos 6 programas de **AP** com o modelo *Lenet*, variando o tamanho das amostras de dados (que assumiu os valores de 256, 512 e 1024 **KiBs**) e o otimizador (que variou

entre o Adam e o **SGD**).

Já para os testes com abordagens distribuídas utilizamos as estratégias `ParameterServerStrategy` e `MultiWorkerStrategy` do **TensorFlow** (portanto, uma arquitetura assíncrona e uma síncrona), com os modelos *AlexNet* e *Lenet*, *Adam* como otimizador e amostras de 256 **KiBs**.

Quanto aos dados utilizados para o treino procurámos conjuntos de dados que estejam a ser utilizados pela comunidade e que tenham um tamanho significativo, de modo a não caber na memória dos nodos (que para estes testes foi limitada a 68 **GiBs**), pelo que recorremos ao *Imagenet* (900.000 imagens de treino, aproximadamente 100 **GBs**, convertidas em 1024 TFRecords) e ao *OpenImages* (1.743.042 imagens de treino, aproximadamente 500 **GBs**, convertidas em 5294 TFRecords).

No contexto deste estudo considerámos que seria relevante obter as seguintes métricas:

- percentagem de utilização do **GPU**
- percentagem de utilização do **CPU**
- percentagem de utilização da memória
- largura de banda da rede
- largura de banda do disco
- tempos de execução
- chamadas de sistema

Recorrendo ao *Remora* [27], uma ferramenta de monitorização de utilização de recursos, conseguimos obter ou inferir as cinco primeiras métricas referidas e a própria ferramenta de **AP** dá-nos o penúltimo ponto. No entanto, não tínhamos uma forma direta de obter o registo das chamadas de sistema com os seus argumentos e valores de retorno de uma forma direta, pelo que foi necessário desenvolver uma ferramenta que nos permitisse ter estas informações.

Ferramenta de Captura das Chamadas ao Sistema Operativo

Para desenvolver a referida ferramenta precisávamos de conseguir intercetar as chamadas ao sistema operativo feitas pelas ferramentas de **AP**. Isto foi possível recorrendo à variável de ambiente `LD_PRELOAD`, uma vez que guardando nela o endereço de um objeto ou biblioteca o carregador dinâmico do Linux carrega-a antes das restantes bibliotecas. Assim, as funções que lá estão implementadas vão substituir as que estão nas bibliotecas originais, permitindo redefinir o seu comportamento. Deste modo conseguimos que a ferramenta desenvolvida intercete as chamadas de várias funções relevantes da biblioteca GNU C (ou libc) [32]. O comportamento destas operações é ligeiramente alterado, para que para além de executar a operação desejada faça um registo da mesma num ficheiro específico. Cada

registo contém o momento em que o registo foi feito (em microssegundos), o identificador do processo, o nome da operação e informações relevantes que a caracterizam. Na tabela 3 está uma síntese das operações que estão a ser intercetadas, até ao momento, juntamente com as informações recolhidas para cada uma.

Operação	Informações recolhidas
<i>Open, Open64, Openat</i>	• caminho para o ficheiro a ler (<i>path</i>) • identificador do ficheiro a abrir (valor de retorno)
<i>Close</i>	• identificador do ficheiro a fechar (<i>fd</i>)
<i>Mmap</i>	• endereço para o mapeamento (<i>addr</i>) • número de bytes com que o mapeamento é iniciado (<i>length</i>) • tipo de proteção desejada para o mapeamento (<i>prot</i>) • tipo de visibilidade do mapeamento para outros processos (<i>flags</i>) • identificador do ficheiro a mapear (<i>fd</i>) • posição a partir da qual o mapeamento deve ocorrer (<i>offset</i>)
<i>MunMap</i>	• endereço a partir de onde a remoção do mapeamento deve ocorrer (<i>addr</i>) • número de bytes a remover (<i>length</i>) • valor a informar se a operação foi bem-sucedida (valor de retorno)
<i>Read</i>	• identificador do ficheiro a ler (<i>fd</i>) • número de bytes a ler do ficheiro (<i>count</i>) • número de bytes efetivamente lidos (valor de retorno)
<i>Write</i>	• identificador do ficheiro onde escrever (<i>fd</i>) • número de bytes a escrever (<i>count</i>) • número de bytes efetivamente escritos (valor de retorno)
<i>Pread, Pread64</i>	• identificador do ficheiro a ler (<i>fd</i>) • número de bytes a ler (<i>count</i>) • posição do ficheiro onde a leitura deve começar (<i>offset</i>) • número de bytes efetivamente lidos (valor de retorno)
<i>Pwrite, Pwrite64</i>	• identificador do ficheiro onde escrever (<i>fd</i>) • número de bytes a escrever (<i>count</i>) • posição do ficheiro onde a escrita deve começar (<i>offset</i>) • número de bytes efetivamente escritos (valor de retorno)
<i>Getxattr, Lgetxattr</i>	• caminho associado ao atributo a obter (<i>path</i>) • identificador do atributo a obter (<i>name</i>) • valor do atributo (<i>value</i>) • tamanho do valor do atributo (valor de retorno)
<i>Fgetxattr</i>	• identificador associado ao atributo a obter (<i>fd</i>) • identificador do atributo a obter (<i>name</i>) • valor do atributo (<i>value</i>) • tamanho do valor do atributo (valor de retorno)

Tabela 3: Operações intercetadas e respetivos parâmetros e valores de retorno registados

Esta ferramenta pode fazer estes registos num de dois formatos de ficheiro, dependendo do que estiver definido no seu ficheiro de configuração.

1. ficheiro TXT com os diversos registos expostos num formato que facilita leitura dos mesmos por seres humanos.
2. ficheiro JSON, em que cada registo é feito no formato de ficheiro JSON independente, que facilita o processamento dos dados.

3.2.2 Resultados

Ao analisar os ficheiros obtidos foi possível perceber que, tal como esperávamos, a abordagem e estratégia seguida afeta os padrões de obtenção de dados.

Carga de Trabalhos Heterogénea

Nos testes de carga de trabalho heterogénea, tal como previsto, em cada época, cada um dos nodos leu a totalidade do conjunto de dados. No entanto, como resultado da disparidade dos tempos de uma época entre modelos diferentes, verificou-se que as leituras dos dados estavam temporalmente distribuídas de formas muito distintas. No gráfico da Figura 17 pode-se ver como os 6 modelos utilizados distribuíram as suas leituras ao longo de uma época.

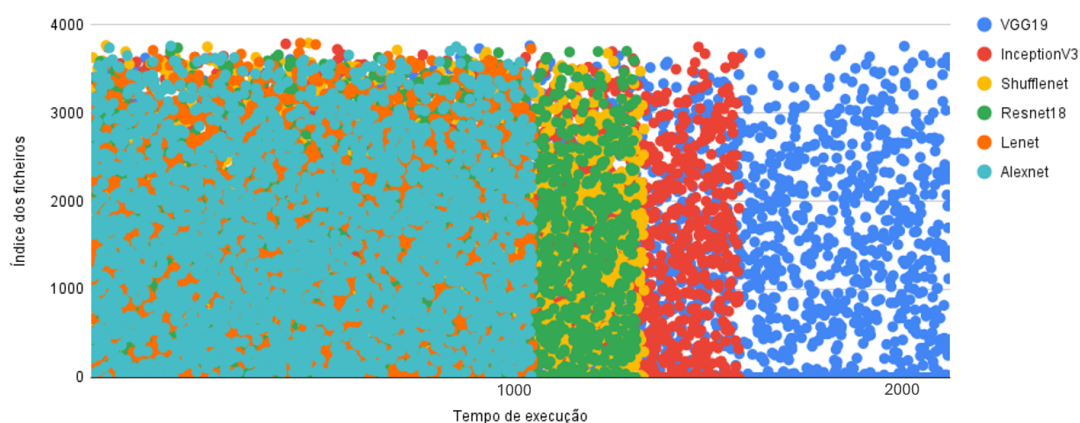


Figura 17: Índice dos ficheiros lidos ao longo de uma época de treino para carga de trabalho heterogénea.

Foi ainda possível verificar que a ordem em que os ficheiros foram lidos nas várias épocas era aleatória (*shuffling*).

Já a análise dos recursos utilizados pelos vários modelos leva à conclusão de que os vários modelos têm gargalos de execução diferentes.

Comparando os tempos de treino quando o conjunto de dados está no disco local e no Lustre é possível concluir que o *Lenet*, o *Alexnet*, o *Resnet18* e o *Shufflenet* têm o acesso a dados como gargalo, uma vez que estes diminuem quando o acesso a dados é feito do disco local e que nenhum dos outros recursos está a ter percentagens de utilização significativas. Já o *InceptionV3* e o *VGG19* são limitados pelo GPU apresentando percentagens de utilização superiores a 90%.

Carga de Trabalhos Homogénea

Nos testes de carga de trabalho homogénea, tal como nos anteriores, em cada época cada nodo leu a totalidade do conjunto de dados e a ordem das leituras entre épocas e execuções variava. Contudo, neste cenário, as épocas tiveram durações mais similares, pelo que a distribuição dos pedidos de leitura foi mais uniforme e temporalmente concentrada do que no cenário anterior, como se pode verificar no gráfico da Figura 18 que apresenta os índices dos ficheiros do conjunto de dados *OpenImages* lidos pelos vários programas ao longo de uma época de treino.

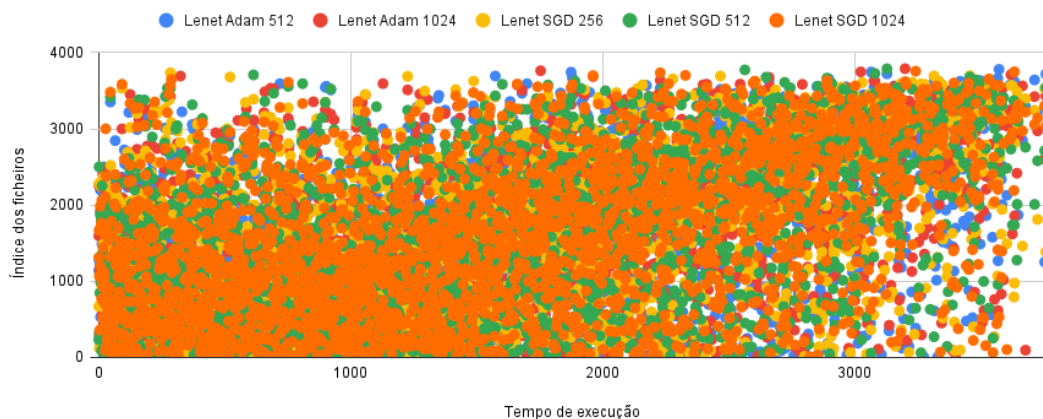


Figura 18: Índice dos ficheiros lidos ao longo de uma época de treino para carga de trabalho homogénea.

Como os vários ficheiros são todos lidos por todos os programas a abordagem que seguimos nesta dissertação vai permitir reduzir acessos, diminuindo assim a pressão de E/S causada no sistema de ficheiros partilhado e deverá permitir reduzir o tempo de treino. No entanto, como todas as leituras dos vários nodos são feitas ao longo de períodos de tempo mais curtos, é possível que a utilização do sistema desenvolvido nesta dissertação possa não ter um impacto tão significativo como no cenário anterior.

MultiWorkerStrategy

Com o *MultiWorkerStrategy* o conjunto de dados é distribuído pelos trabalhadores de forma uniforme, ou seja, o conjunto de dados é distribuído por todos eles de modo que processam o mesmo número de

amostras de dados (diferentes entre si) e correm durante o mesmo período de tempo, como se pode ver no gráfico da Figura 19, que exemplifica a distribuição dos ficheiros de dados durante uma época de treino (neste caso com 2 trabalhadores e a treinar com *Imagenet*). Isto vai de encontro ao que esperávamos ver, uma vez que esta é uma abordagem de distribuição de dados síncrona. Em cada época as várias amostras de dados são processadas por uma ordem aleatória, mas os trabalhadores nunca processam amostras diferentes.

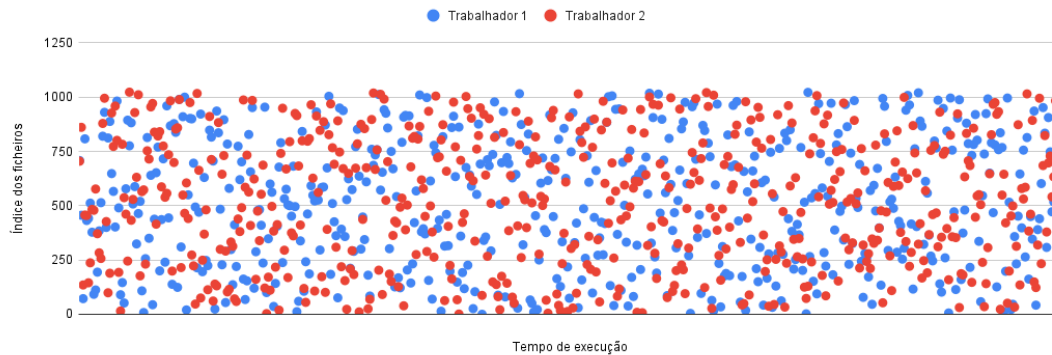


Figura 19: Índice dos ficheiros lidos ao longo de uma época de treino com *MultiWorkerStrategy*

A análise dos ficheiros abertos e lidos por cada trabalhador ao longo de várias épocas levou-nos a concluir que a atribuição de ficheiros de dados e trabalhadores feita inicialmente se mantinha para as restantes. Isto implica que os ficheiros usados por um trabalhador para treinar nas várias épocas são sempre os mesmos, sendo apenas servidos por ordens diferentes.

Comparando o desempenho dos testes em que os dados estavam guardados no Lustre e no disco local verificamos que não houve variações significativas, pelo que o acesso aos dados não é o fator limitante. Analisando a utilização das unidades de processamento (**CPU**, **GPU**) foi possível verificar que nenhuma delas estava a limitar a execução, uma vez que: i) a utilização média de **GPU** em cada trabalhador ronda os 40% para *Alexnet* e 60% para *Lenet*. ii) a utilização média de **CPU** em cada trabalhador ronda os 60% e 65% quando treinamos o modelo com *Imagenet* e 20% e 30% quando o conjunto de dados é o *OpenImages*.

ParameterServerStrategy

Já com o *ParameterServerStrategy*, ao contrário do que era esperado, todos os trabalhadores abrem e leem os ficheiros do conjunto de dados na totalidade (podendo haver algumas variações na ordem em que o fazem, mas não muito significativas), mas só processam os que lhe foram atribuídos pelo chefe. Acreditamos que isto resulta do *prefetch* de dados e do facto de, nesta estratégia, os trabalhadores não

saberem que dados vão utilizar a seguir para treinar o modelo até acabarem de treinar uma amostra e serem informados pelo trabalhador chefe.

O gráfico da Figura 20 procura exemplificar a distribuição dos ficheiros de dados lidos por dois trabalhadores durante uma época de treino (com *Imagenet*) e permite verificar que ambos os trabalhadores estão a ler a totalidade do conjunto de dados de forma cíclica. Podemos ainda verificar que as linhas do gráfico se sobrepõem latamente, havendo pequenas variações na ordem em que ficheiros com índices próximos são lidos.



Figura 20: Índice dos ficheiros lidos ao longo de uma época de treino com *ParameterServerStrategy*

Comparando o desempenho dos testes em que os dados estavam guardados no Lustre e no disco local verificamos que não houve variações significativas, pelo que o disco local não é o fator limitante.

Analisando a utilização das unidades de processamento (**CPU**, **GPU**) foi possível verificar que nenhuma delas estava a limitar a execução, uma vez que: i) a utilização média de **GPU** nos trabalhadores é de cerca de 30% para *Alexnet* e *Lenet*. ii) a utilização média de **CPU** nos trabalhadores é de cerca de 60% quando treinamos com o conjunto de dados *OpenImages*.

Assim, tal como no cenário anterior, o armazenamento não aparenta estar a limitar a execução e o algoritmo não está a aproveitar todos os recursos computacionais disponíveis (**CPU**, **GPU**), pelo que o sistema desenvolvido nesta dissertação só deverá reduzir a pressão de **E/S** causada no sistema de ficheiros partilhado, não afetando significativamente o tempo de treino do modelo.

3.3 Sumário

A primeira avaliação experimental permitiu-nos confirmar que, para os tamanhos normalmente utilizados para as amostras de dados de programas de **AP**, pedir dados a outros nodos que os têm guardados no armazenamento local é comparável a ler diretamente do disco local. Assim, tirar partido dos dados guardados em outros nodos em vez de os ler do sistema de ficheiros partilhado deverá diminuir o número de acessos a este último, diminuindo assim a pressão **E/S** do sistema de armazenamento paralelo. Consequentemente, é previsível que esta alteração permita ainda reduzir os períodos que **CPUs** e **GPUs** passam parados à espera de dados em programas de **AP** limitados por acessos ao sistema de armazenamento e potencialmente diminuir os tempos de execução e respetiva variabilidade, melhorando assim o desempenho destes programas.

Já com a segunda avaliação experimental foi possível compreender um pouco melhor os padrões de acesso a dados das várias estratégias e abordagens definidas e averiguar se este age como fator limitador da execução para cada um deles. Foi ainda verificada a utilização dos vários recursos disponíveis pelas várias combinações de modelos e estratégias. Assim, foi possível concluir que para todas as versões o sistema desenvolvido nesta tese deverá permitir reduzir o número de acessos ao sistema de ficheiros partilhado, diminuindo a pressão **E/S** causada neste. Adicionalmente, para as abordagens multi-nodo locais deveremos ainda conseguir diminuir os tempos de treino, especialmente de modelos como o *Alexnet*, o *Lenet*, o *Resnet18* e o *Shufflenet* para os quais o acesso ao sistema de armazenamento age como fator limitador.

Capítulo 4

DistMonarch

O *DistMonarch* é um sistema agnóstico à ferramenta de **IA**, que aproveita a hierarquia das camadas do sistema de armazenamento das infraestruturas **HPC** e tira partido de particularidades de **AP** multi-nodo. Este foi desenhado de modo a seguir os seguintes princípios base:

Desacoplamento. Ler dados do armazenamento local ou receber dados de outros nodos provou-se mais eficiente do que ler esses dados do sistema de ficheiros partilhado. Assim, de modo a possibilitar aplicar estas otimizações para as várias ferramentas de **IA** e sem interferir com otimizações que já estejam a ser feitas pelas próprias ferramentas, o *DistMonarch* é desacoplado da lógica interna das ferramentas de **IA**.

Transparência. O *DistMonarch* não implica nenhuma alteração ao código dos programas de **AP** nem à forma como as ferramentas de **IA** são utilizadas. Isto permite que seja facilmente utilizável em centros **HPC**.

Posicionamento de dados automático. A nossa solução faz a distribuição do conjunto de dados pelas várias camadas de armazenamento dos vários nodos de forma autónoma. O utilizador define a hierarquia entre as diversas camadas disponíveis num nodo num ficheiro de configurações e o programa trata de fazer a distribuição dos dados pelas várias camadas e nodos.

Otimizado para **AP.** O sistema foi desenhado para os padrões de **E/S** específicos de treino de modelos **AP**. Assim, tem em consideração que as leituras são feitas numa ordem aleatória, a totalidade dos dados vai ser lida em cada época e várias operações de leitura de pequenas dimensões vão ser feitas para ler dados dos mesmos ficheiros (quando utilizamos formatos como TFRecords [30]).

Redução de acessos ao sistema de ficheiros paralelo e do tempo de treino. O *DistMonarch* procura diminuir o número de operações **E/S** feitas ao sistema de ficheiro partilhado (reduzindo a pressão neste) e idealmente diminuir os tempos de treino de programas de **AP**.

4.1 Arquitetura da Solução

Como referimos anteriormente o *DistMonarch* baseia-se no *Monarch* [43]. Como tal mantivemos os seus elementos estruturais base, tendo adicionado novos elementos aos previamente existentes, como se pode verificar na Figura 21 (onde os novos componentes se encontram destacados com fundo laranja).

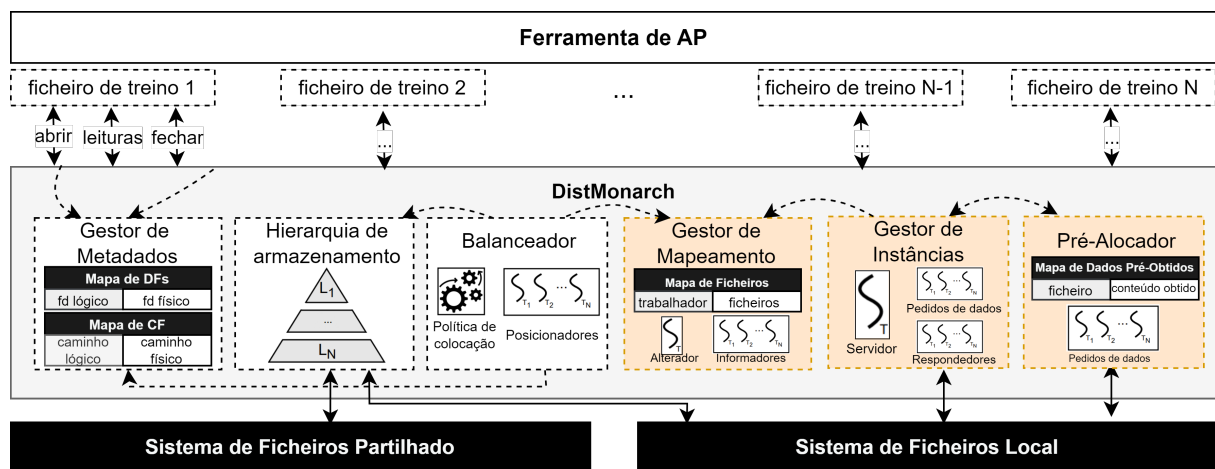


Figura 21: Arquitetura para a solução desenvolvida.

Enquanto que o *Monarch* se focava em guardar o máximo de dados possível nas camadas superiores do sistema de armazenamento, o nosso sistema tem em consideração:

- a existência de diversos trabalhadores diferentes a ler dados;
- o facto de que cada um deles tem acesso ao seu disco local;
- que o tempo de troca de dados entre trabalhadores, para os tamanhos normalmente utilizados por ferramentas de **AP** para ler amostras de dados, são similares aos de ler do próprio disco local.

Assim, aos três componentes do *Monarch* (Gestor de Metadados, Hierarquia de armazenamento e Balanceador) foram adicionados três novos componentes: o Gestor de Mapeamento, o Gestor de Instâncias e o Pré-Alocador.

- **Gestor de Metadados:** O *DistMonarch* copia ficheiros de dados da última camada de dados (p. ex., sistema de ficheiros paralelo) para as restantes de modo a conseguir dar resposta mais rapidamente aos pedidos da ferramenta de **AP**. No entanto, os pedidos de leitura vêm sempre

direcionados ao ficheiro original, pelo que é necessário utilizar metadados para fazer a tradução entre ambos. Assim, o Gestor de Metadados tem o mapa de **Descritores de Ficheiros (DFs)** que faz a associação entre o **DF** que vem nos pedidos de leitura (**DF** físico) e o que está associado ao ficheiro na camada mais apropriada (**DF** lógico) e o mapa de **CFs** que faz o mesmo, mas para os caminhos.

- **Hierarquia de Armazenamento:** Esta componente é responsável por organizar e fazer a gestão das camadas de armazenamento. Estas têm uma ordem associada que é definida com base nas configurações dadas pelos utilizadores. A última camada tem a totalidade do conjunto de dados, sendo apenas utilizada para leitura. As restantes camadas são inicializadas sem dados, podendo depois ser povoadas com dados pelo *DistMonarch*. A cada camada estão associados dados que permitem gerir o seu estado, como o seu caminho e a sua capacidade.
- **Balanceador:** O Balanceador é o componente responsável por fazer a colocação dos ficheiros de dados nas várias camadas da hierarquia de armazenamento. Assim, quando o sistema tem de ir buscar dados ao último nível de armazenamento, os dados são passados à ferramenta de **AP** e é criada uma *thread* de fundo para tentar puxar o ficheiro para camadas superiores. A política de colocação dita qual a camada na qual o ficheiro deve ser colocado, verificando qual é a primeira que tem capacidade para este.
- **Gestor de Mapeamento:** Para os trabalhadores poderem partilhar dados entre si é necessário que saibam que ficheiros de treino é que cada um tem em camadas superiores da hierarquia de armazenamento. O Gestor de Mapeamento é responsável por manter e atualizar essa informação. O mapa de ficheiros faz a associação entre os vários trabalhadores que estão a correr o *DistMonarch* e os ficheiros que cada qual já copiou para níveis mais apropriados do sistema de armazenamento.
- **Gestor de Instâncias:** O Gestor de Instâncias é o responsável por fazer a troca de dados entre instâncias do *DistMonarch*.
- **Pré-Alocador:** Nos testes preliminares concluímos que quando um programa de **AP** está a ler dados de ficheiros de grandes dimensões (como TFRrecords[30]) faz diversos pedidos de leitura de amostras de dados desse ficheiro até o ter lido na totalidade. Assim, quando uma instância do *DistMonarch* recebe o primeiro pedido de leitura de um ficheiro e outro nodo pode partilhar esses dados com ela, a amostra é partilhada entre as instâncias e devolvida à ferramenta de **AP**.

Em seguida, o Pré-Alocador começa a pedir o resto do ficheiro, por partes, e a guardá-lo no mapa de dados pré-obtidos. Deste modo, permite que, quando os subsequentes pedidos de leitura de dados desse ficheiro cheguem, estes já estejam disponíveis, possibilitando assim uma redução na latência da resposta.

4.1.1 Cenários de Uso

O *DistMonarch* fica entre o programa de **AP** e o sistema de armazenamento a interceder e responder, transparentemente, às operações que sigam a interface **POSIX** (pelo que, é compatível com qualquer ferramenta que a suporte ou exporte).

Adicionalmente, como esta ferramenta deve correr sobre programas **AP** multi-nodo, é relevante considerar que nem sempre faz sentido correr o *DistMonarch* em todos os nodos. Por exemplo, em estratégias *ring-Allreduce* ou quando temos vários modelos a treinar em simultâneo, todos os nodos leem dados e como tal faz sentido ter uma instância do *DistMonarch* em cada um para melhor gerir a obtenção e partilha dos dados e otimizar a quantidade de dados que podem estar guardados nos vários discos locais (Figura 22).

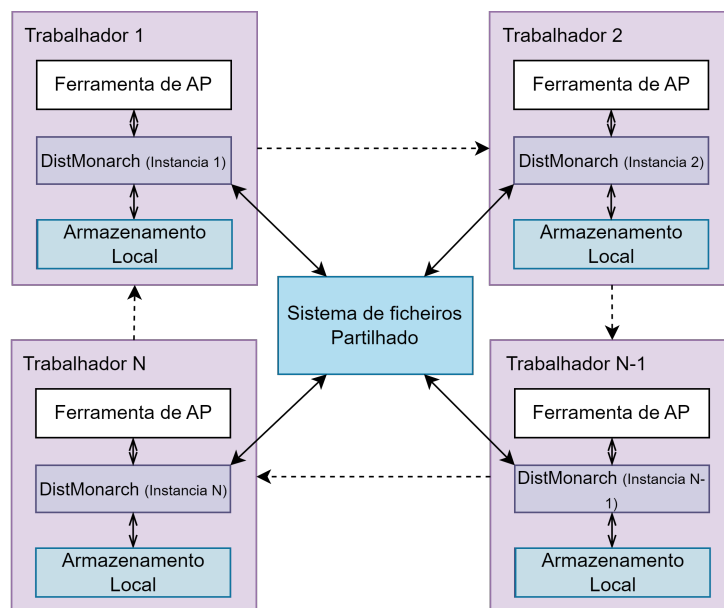


Figura 22: Localização do *DistMonarch* em multimodelo e *ring-allreduce*.

Já em estratégias que sigam uma arquitetura na qual nodos diferentes têm funções diferentes tal pode já não acontecer. Por exemplo, no caso de arquiteturas de servidor de parâmetros, há um nodo a servir de chefe, nodos a assumir o papel de servidores de parâmetros e nodos a funcionar como trabalhadores. No entanto, apenas os trabalhadores levam a cabo a fase de treino e como tal só eles é que fazem pedidos de leitura de dados ao sistema de armazenamento. Por este motivo, apenas nestes faz sentido ter instâncias

do sistema, ficando como demonstrado na Figura 23.

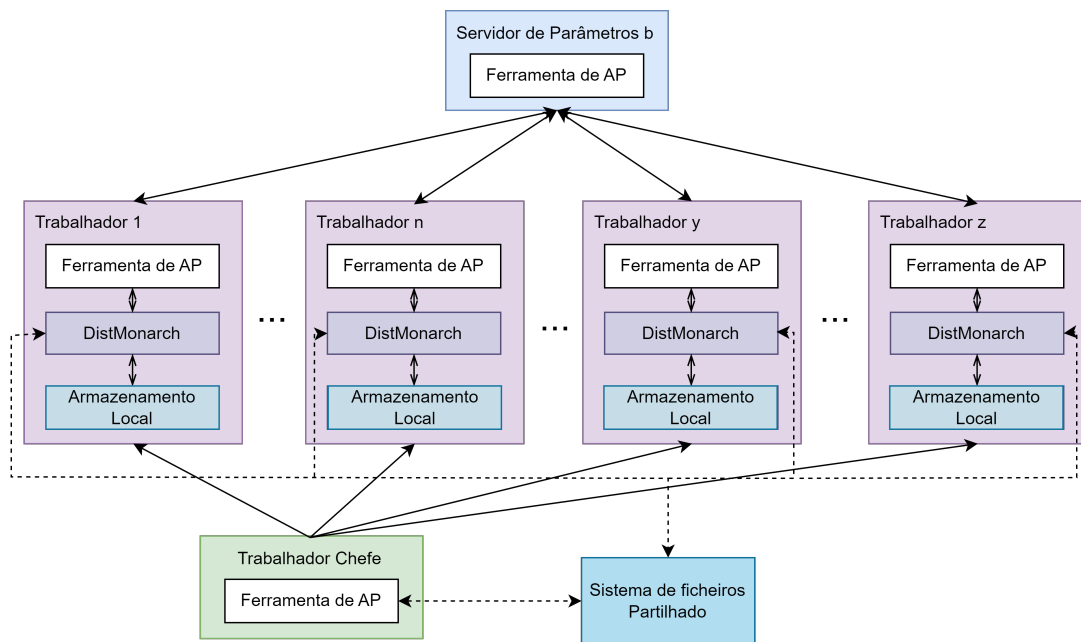


Figura 23: Localização do *DistMonarch* em estratégias SP.

4.2 Fluxo de Operações

O funcionamento do *DistMonarch* pode ser dividido em 3 vertentes diferentes:

- Inicialização: Passos necessários antes de se poder correr o sistema.
- Obtenção de dados: Lógica por trás da forma como o sistema obtém os dados que devolve ao programa de AP.
- Obtenção prévia de dados: Lógica envolvida na obtenção de dados antes de serem pedidos pelo programa de AP, de modo a diminuir a latência de resposta a esses pedidos.

Todos esses já existiam na versão local do *Monarch* [43], mas o fluxo de operações e passos de cada uma sofreram alterações, de modo a se adaptarem melhor a abordagens multi-nodo.

4.2.1 Inicialização

Tal como no *Monarch* original [43], antes de correr o sistema é necessário especificar no ficheiro de configurações as camadas de armazenamento a que este pode recorrer, por ordem de prioridade. Adicionalmente, antes de lançar cada instância do *DistMonarch* é ainda necessário especificar duas variáveis de ambiente:

- `WRKS_ADDRS` : endereços dos trabalhadores que vão correr o *DistMonarch*.
- `TASK_ID` : índice do trabalhador sob o qual aquela instância do *DistMonarch* está a correr.

Estas são necessárias para permitir que os diversos trabalhadores que estão a correr com o *DistMonarch* saibam como comunicar uns com os outros.

4.2.2 Obtenção dos Dados

Quando a ferramenta de **AP** faz um pedido de dados o *DistMonarch* interceta esse pedido e segue a lógica representada no esquema da Figura 24 no qual foi considerado uma hierarquia de armazenamento só com disco local e sistema de ficheiros partilhado.

Ao receber um pedido de obtenção de dados a instância do *DistMonarch* começa por verificar se esses dados se encontram disponíveis em níveis superiores da hierarquia do sistema de armazenamento. Se isto não se verificar, o *DistMonarch* confirma se os dados estão guardados no mapa de dados pré-obtidos. Se os dados se encontrarem lá, são utilizados para dar resposta ao pedido de obtenção de dados do programa de **AP**. Caso contrário verifica-se se algum dos outros nodos tem esses dados guardados em níveis superiores da hierarquia do sistema de armazenamento (como, por exemplo, o disco local), verificando se o ficheiro está associado a algum trabalhador no mapa de ficheiros. Caso algum nodo tenha esses dados, a instância do *DistMonarch* envia um pedido de permutação de dados a esse nodo e devolve os dados que recebe para dar resposta ao pedido do programa de **AP**. Já no cenário de nenhum nodo ter esses dados guardados num nível que não o último segue-se a mesma lógica que o *Monarch* [43] seguiria, ou seja, o ficheiro que contem esses dados é copiado para o nível de armazenamento mais apropriado (a primeira camada com espaço suficiente para guardas a totalidade do ficheiro). Caso nenhuma camada tenha espaço para a totalidade do ficheiro a localização do ficheiro mantém-se.

No caso em que o ficheiro é copiado para outro nível é ainda enviada uma mensagem aos restantes a informá-los da aquisição do ficheiro. Ao receber esta informação cada um deles atualiza o seu mapa de ficheiros, associando o ficheiro ao nodo que enviou a mensagem.

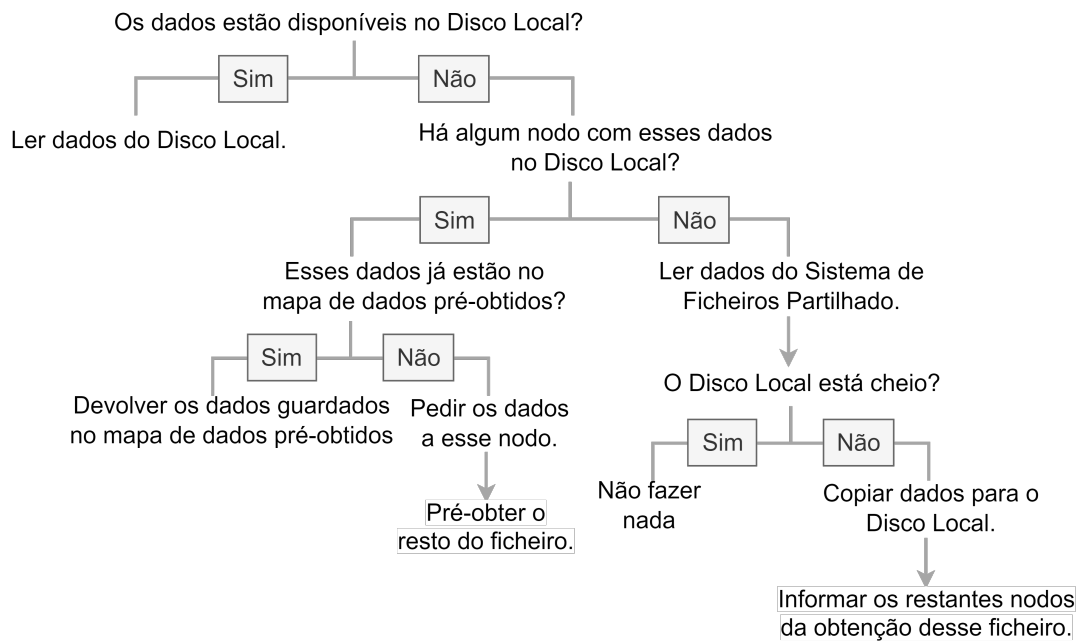


Figura 24: Esquema do fluxo de obtenção de dados.

4.2.3 Obtenção Prévia de Dados

Se os dados forem partilhados apenas quando o *DistMonarch* interceta o pedido de leitura da ferramenta de **AP**, esse pedido tem de esperar que a instância do *DistMonarch* verifique se alguma das outras instâncias tem aqueles dados guardados, peça os dados e só aí é que recebe uma resposta. De modo a diminuir esta espera o *DistMonarch* tira partido de que quando uma ferramenta de **AP** começa a ler um ficheiro lê-o até ao fim e procura fazer a pré-obtenção dos dados. Assim, quando um trabalhador pede uma pequena porção de dados de um ficheiro a outro trabalhador é criada uma entrada no mapa de dados pré-obtidos para o ficheiro em causa e lançada uma *thread* de fundo que vai pedido, de forma gradual, o resto do ficheiro. Os dados recebidos são guardados na entrada criada, sendo posteriormente eliminados quando o programa recebe um pedido para fechar o ficheiro.

Já no cenário em que os dados são lidos da última camada do sistema de armazenamento e uma das camadas superiores tem espaço suficiente para os guardar, o sistema tem o mesmo comportamento que o *Monarch* [43]. Isto é, ao ler os dados do ficheiro de treino, em plano de fundo começa a copiá-lo para o nível de armazenamento mais apropriado. Quando o ficheiro tiver sido todo copiado os metadados são atualizados, ou seja, o tamanho do ficheiro copiado é descontado ao espaço disponível na camada, e os mapas de **DFs** e **CFs** são informados da existência da cópia. Para tal é criada uma nova entrada em cada um deles, sendo que a entrada adicionada ao mapa de **DF** faz a associação entre o **DF** do ficheiro original e o **DF** do ficheiro que criamos e enquanto que a entrada adicionada ao mapa de **CF** relaciona

o caminho dos mesmos.

4.3 Implementação

O protótipo do *DistMonarch* foi implementado em C++ e adicionou aproximadamente quinhentas linhas de código ao do *Monarch*.

4.3.1 Configurações

O ficheiro YAML de configurações continua idêntico ao utilizado no *Monarch*, tendo informações como o caminho e a capacidade dos níveis de armazenamento disponíveis. O caminho para este ficheiro continua a ter de ser colocado na variável de ambiente `MONARCH_CONFIGS_PATH`.

O *DistMonarch* recorre à variável de ambiente `WRKS_ADDRS` para saber que nodos estão a correr instâncias do *DistMonarch* e qual o endereço **IP** que deve usar para se comunicar com elas. Esta deve ser a lista de todos os endereços dos nodos, que estão a correr com *DistMonarch*, separados por vírgulas, como demonstrado no exemplo da Figura 25.

Já a variável de ambiente `TASK_ID` deve ter o índice que o endereço do próprio nodo ocupa na lista de `WRKS_ADDRS`.

```
1 export WRKS_ADDRS="192.168.44.133,192.168.44.184,192.168.44.114"
2 export TASK_ID=0
```

Figura 25: Exemplo de variáveis de ambiente a definir antes de correr o *DistMonarch*.

4.3.2 Aplicabilidade a Várias Ferramentas de **AP**

O *DistMonarch* recorre ao `LD_PRELOAD` para intercetar as operações **POSIX** em tempo de execução feitas pelos programas sobre os quais está a correr. Isto permite substituir as operações da interface **POSIX** por novas funções com a mesma interface, mas implementações diferentes, alterando assim o comportamento resultante de forma transparente às ferramentas de **AP**. Deste modo, as operações que foram alteradas foram o *open* (para modificar o comportamento de abertura de ficheiros, passando a ter em consideração as cópias do ficheiro em outras camadas), o *pread*, o *mmap* (para adaptar as leituras, como descrito anteriormente) e o *close* (para alterar os ficheiros que o programa procura encerrar no cenário em que o abrimos e lemos de uma cópia e não do ficheiro base).

4.3.3 Gestor de Mapeamento

Cada instância do *DistMonarch* tem uma *thread* permanentemente a correr de fundo (o Alterador). Esta recebe as mensagens que cada nodo envia a comunicar que copiou um ficheiro para camadas superiores do sistema de armazenamento. Cada uma destas mensagens identifica o nodo que a enviou e o ficheiro que foi copiado. Assim, ao receber uma destas mensagens, é verificado se o mapa de ficheiros tem uma entrada para o nodo em questão. Se tiver então o nome do ficheiro é adicionado à lista de ficheiros dessa entrada, caso contrário, é adicionada uma a associação entre o nodo e o ficheiro.

4.3.4 Permutação de Dados entre Trabalhadores

Quando um nodo quer dados de um ficheiro que no mapa de ficheiros já está associado a outro nodo, envia uma mensagem **TCP** para a porta 50052 desse nodo com o nome do ficheiro de onde queremos dados, a posição onde começar a leitura e o tamanho a ler.

O pedido é recebido por uma *thread* servidor, do Gestor de Instâncias, que está sempre a correr de fundo. Esta, por sua vez cria uma nova *thread* (respondedor), por pedido, para obter e enviar os dados.

4.4 Resumo

O *DistMonarch* é uma ferramenta de armazenamento, baseada no *Monarch* [43], desenhada especificamente para programas **AP** multi-nodo que corram em infraestruturas **HPC**, onde podemos contar com um sistema de ficheiros paralelo partilhado entre todos os nodos e o armazenamento local de cada nodo. Este baseia-se na conclusão que ler dados do disco local e receber dados de outros nodos são operações mais eficientes do que ler do sistema de ficheiros partilhado. Assim, esta solução procura distribuir o conjunto de dados pelo armazenamento local dos vários nodos e permite que partilhem esses dados entre si. Isto possibilita a diminuição do número de ficheiros que têm de ser lidos do sistema de ficheiros partilhado, diminuindo assim a pressão **E/S** exercida no mesmo, e potencializando a melhoria do desempenho de treino de programas de **AP** em abordagens multi-nodo. Este permite ainda obter dados antes de os programas de **AP** os pedirem, uma vez que assim que dados de um ficheiro são pedidos o resto do ficheiro ou é puxado para o disco local do nodo ou é pedido a outro nodo e guardado numa *cache* em memória. No entanto, os ficheiros que são colocados em memória são removidos quando o programa de **AP** manda fechar o ficheiro.

O *DistMonarch* é compatível com qualquer ferramenta de **AP** que suporte ou exporte a interface **POSIX**. As

operações que seguem essa interface são interceptadas permitindo que o *DistMonarch* atue sem que seja necessário fazer qualquer tipo de alteração ao código dos programas de **AP** para o correr. Adicionalmente, como esta ferramenta apenas altera a localização dos dados e a forma como os obtemos, continua a ser compatível com otimizações e configurações feitas pela ferramenta de **AP**.

Capítulo 5

Avaliação Experimental

Para avaliar a ferramenta desenvolvida procuramos responder às seguintes perguntas:

- O *DistMonarch* consegue diminuir a pressão de **E/S** no sistema de ficheiros partilhado?
- O *DistMonarch* melhora o tempo de treino dos modelos?
- O *DistMonarch* impacta a acurácia dos modelos treinados?

Para tal procuramos comparar os resultados que obtidos com o *DistMonarch* com os obtidos ao correr sem nenhuma ferramenta de otimização de armazenamento e com o *Monarch* [43].

5.1 Configurações

Hardware e Software. Estes testes foram corridos nos mesmos nodos e com as mesmas versões especificadas para os testes preliminares. No entanto, nestes testes a memória RAM foi limitada a 68 **GiBs**, de modo a diminuir o impacto da *Page Cache* nos resultados obtidos.

Configurações do Monarch e do DistMonarch. O *Monarch* e o *DistMonarch* correram exatamente com as mesmas configurações, tendo na hierarquia de armazenamento duas camadas. A primeira corresponde ao disco local e a segunda corresponde à diretoria do Lustre onde o conjunto de dados está guardado. Adicionalmente, para poder testar o cenário em que o conjunto de dados não cabe na combinação dos discos locais dos vários nodos foram corridos testes em que o disco local foi limitado de 118 para 60 **GiBs**.

Framework, Estratégia e Pré-processamento. Para estes testes focamo-nos no **TensorFlow** [35], por ser uma das ferramentas de **AP** mais populares. Aplicamos as estratégias *MirroredStrategy* [18] e *ParameterServerStrategy* [24], utilizando os 4 **GPUs** disponíveis em cada nodo. No que diz respeito à

pipeline de entrada, a ordem de acesso aos ficheiros de dados era diferente em cada época (fazendo um *shuffle* da lista de ficheiros a ler) e utilizamos o *shuffle* do próprio **TF** para construir e utilizar um *shuffle buffer*, aumentando a aleatoriedade da ordem de acesso aos dados. Recorremos ainda a otimizações de **E/S** como o *Interleave* e o *Map* do **TF** que permitem paralelizar o carregamento e pré-processamento dos dados, respetivamente. O grau de paralelismo utilizado nestas foi definido de modo a melhor otimizar o processo, tendo sido utilizado o *tf.data.experimental.AUTOTUNE*.

Conjunto de Dados. Para estes testes procuramos utilizar conjuntos de dados que estejam a ser utilizados pela comunidade e de grandes dimensões, de modo a não caberem na combinação da memória e do disco local de cada nodo, para tal, recorremos ao *OpenImages V5* [23].

Este é composto por 1.743.042 imagens de treino, 125.436 imagens de teste e 41.620 imagens de validação, que correspondem a 517 **GiBs**, 37 **GiBs** e 13 **GiBs**, respetivamente. Cada um destes grupos foi convertido para *TfRecords* resultando em 5294, 379 e 133 ficheiros aproximadamente com 100 **MiBs** cada.

Modelos e Hiperparâmetros. De modo a simular situações de modelos com padrões e ritmos de acesso a dados variáveis e/ou similares temos dois tipos de testes com treino local.

O primeiro procura analisar o cenário em que temos diversos modelos com vários ritmos de obtenção de dados diferentes (cargas de trabalhos mais heterogéneas). Os modelos escolhidos para este teste foram *AlexNet*, *LeNet*, *ResNet18*, *Shufflenet*, *Inception-v3* e *VGG19*. Todos estes modelos foram treinados com amostras de 512 **GiBs**, com o otimizador *Adam* e uma taxa de aprendizagem de 0,1 durante uma hora e meia.

Já o segundo foca-se no cenário em que temos modelos com acessos a dados mais similares (cargas de trabalhos mais homogéneas). Neste cenário, vários programas de **AP** treinam o mesmo modelo, mas com hiperparâmetros diferentes. O modelo escolhido foi o *LeNet*, uma vez que, por ser um modelo cuja execução é limitada pelo armazenamento, é expectável que crie uma maior pressão de **E/S**. Tal como no cenário anterior o tempo de execução foi fixado em uma hora e meia e a taxa de aprendizagem em 0,1. Quanto a parâmetros como o tamanho das amostras e o otimizador a utilizar foram diferentes dependendo da instância do modelo que estivessemos a treinar, variando entre 256, 512 e 1024 **KiBs** e *Adam* [1] e **SGD** [28], respetivamente.

Já o cenário com treino distribuído foi corrido para o modelo *LeNet*, com amostras de 1024 **KiBs**, otimizador *Adam* e 0,1 de *learning rate* durante 1h e 30min.

Metodologia. Foram medidos o tempo de execução, tempo médio por época e a média de utilização dos recursos (como **CPU** e **GPU**) recorrendo ao Remora [27], ao `nvidia-smi` [33] e aos valores devolvidos pela ferramenta de **AP**.

5.2 Resultados

Para verificar os pontos referidos no início deste capítulo vamos analisar o *DistMonarch* com base no tempo de treino dos modelos, no número de pedidos de leitura do Lustre e na utilização de recursos, para os tipos de teste explicados acima (cargas de trabalho heterogéneas, cargas de trabalho homogéneas e *ParameterServerStrategy*). Já o impacto na acurácia é validado com base em teste de cargas de trabalho heterogéneas executados por 40 horas.

Nos vários testes comparamos 5 tipos de execuções:

- **Base** - o programa de **AP** em que os dados estão todos no Lustre.
- **Monarch** - o programa de **AP**, com o conjunto de dados guardado no Lustre, a correr com o *Monarch* e disco local com 110 **GiBs** de capacidade.
- **DistMonarch** - o programa de **AP**, com o conjunto de dados guardado no Lustre, a correr com o *DistMonarch* e disco local com 110 **GiBs** de capacidade.
- **Monarch limitado** - o programa de **AP**, com o conjunto de dados guardado no Lustre, a correr com o *Monarch* e disco local limitado a 60 **GiBs** de capacidade.
- **DistMonarch limitado** - o programa de **AP**, com o conjunto de dados guardado no Lustre, a correr com o *DistMonarch* e disco local limitado a 60 **GiBs** de capacidade.

As duas versões para cada uma das ferramentas procuram exemplificar o impacto da percentagem do conjunto de dados que cabe no disco local de cada nodo para cada uma das ferramentas.

5.2.1 Cargas de Trabalho Heterogêneas

Tempos de Treino dos modelos

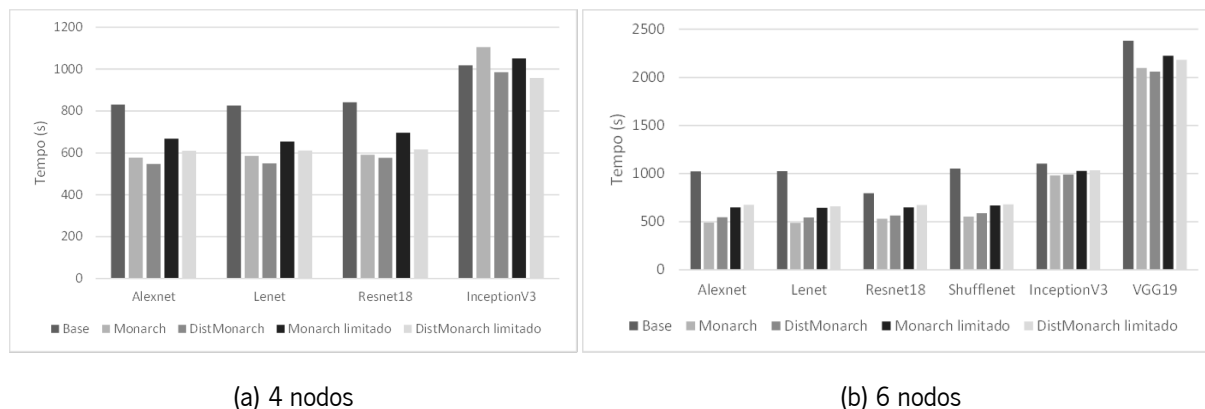


Figura 26: Tempo médio por época para carga de trabalho heterogênea com 4 e 6 nodos a treinar modelos em simultâneo.

Na Figura 26 estão representados os tempos, em média, de cada época que cada um dos modelos demora quando temos 4 e 6 nodos (26a e 26b, respetivamente) a treinar modelos diferentes em simultâneo. Como se pode verificar, no gráfico da Figura 26a, no cenário com 4 nodos para os modelos *Alexnet*, *Lenet* e *Resnet18* a utilização do *Monarch* ou do *DistMonarch* resulta numa diminuição no tempo médio de cada época comparativamente à versão base de aproximadamente 30% e 33%, respetivamente. Isto vai de encontro ao que esperávamos uma vez que tínhamos verificado previamente que estes modelos eram limitados pelo acesso aos dados e ambos estes sistemas tiram partido do disco local ler dados, potencializando assim respostas mais rápidas aos pedidos de leitura. Já entre o *Monarch* e o *DistMonarch* é possível verificar que o *DistMonarch* permitiu obter tempos menores (redução de 5% para o *Alexnet*, de 6% para o *Lenet* e de 2% para o *Resnet18*, comparativamente com o *Monarch*). Isto deve-se ao facto de o *DistMonarch* poder guardar dados diferentes no disco local dos vários nodos e assim ter uma maior percentagem do conjunto de dados que não tem de ser lida do Lustre. Entre as versões normais e as em que o disco local estava limitado, de cada uma das ferramentas, a versão não limitada apresenta tempos 6-15% menores. Isto resulta do facto de termos uma percentagem superior do conjunto de dados armazenada nos discos locais.

Por outro lado, para o *InceptionV3* os 5 tipos de execução apresentam valores muito próximos uns dos outros. Isto advém do facto de que, como vimos na secção 3, este modelo não ser limitado pelo acesso a dados.

No cenário com 6 nodos (representado na Figura 26b), por sua vez, verificamos que as execuções com *Monarch* e *DistMonarch* continuam a ter tempos médios por época menores do que a base. No entanto, ao contrário do cenário de 4 nodos, os modelos *Alexnet*, *Lenet*, *Resnet18* e *Shufflenet* apresentam tempos médios por época menores com *Monarch* (redução de 10% para os 2 primeiros e 6% para os outros 2 comparativamente ao *DistMonarch*). Já para o *InceptionV3* ambas as ferramentas apresentam tempos similares, enquanto que para o *VGG19*, o *DistMonarch* apresenta tempos 2% menores do que os do *Monarch*. Assim, o *Monarch* apresenta tempos médios de época menores que o *DistMonarch*. Isto pode ser justificado por termos mais nodos a comunicar uns com os outros, resultando em maior pressão na rede e aumentando a utilização dos *locks* dos mapas do *DistMonarch*. Mais uma vez a limitação do tamanho do disco resultou num aumento do tempo de cada época de 14-20% para os 4 primeiros modelos e 5-6% para os outros 2.

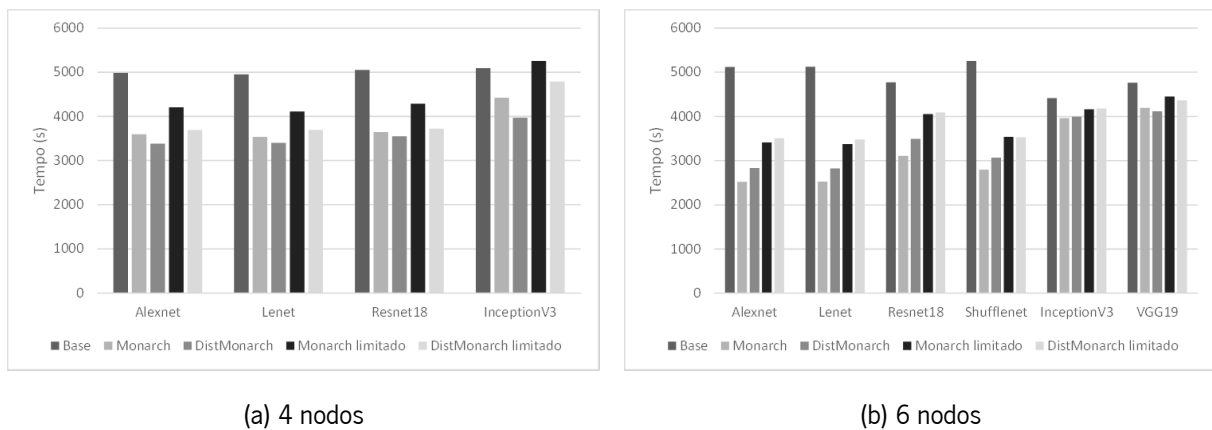


Figura 27: Tempo de execução para carga de trabalho heterogénea com 4 e 6 nodos a treinar modelos em simultâneo. Contemplando 5 épocas de *Alexnet*, *Lenet* e *Shufflenet*, 6 épocas de *Resnet18*, 4 épocas de *InceptionV3* e 2 épocas de *VGG19*.

A Figura 27, por sua vez, permite comparar o tempo que cada um dos tipos de execução levou a concluir o mesmo número de épocas (neste caso, o número de épocas concluído pela versão base, ou seja, 5 épocas para *Alexnet*, *Lenet* e *Shufflenet*, 6 épocas para *Resnet18*, 4 épocas para *InceptionV3* e 2 épocas para *VGG19*).

O gráfico da Figura 27a representa os tempos obtidos para o cenário em que temos 4 nodos a treinar com o mesmo conjunto de dados em simultâneo. Podemos verificar que a utilização do *Monarch* ou do *DistMonarch* permite poupar aproximadamente 23-24 minutos e 25-27 minutos e respetivamente comparativamente com a versão base para o disco de 110 GiBs e aproximadamente 13-14 minutos e 21-22 minutos para o disco limitado, para o *Alexnet*, *Lenet* e *Resnet18*. Já para o *InceptionV3* o *Monarch*

e o *DistMonarch* permitem poupar 11,2 e 18,8 minutos, respetivamente, sem limitação do tamanho do disco local.

Estes testes permitiram ainda verificar que, comparativamente à versão com *Monarch*, a utilização do *DistMonarch* permitiu reduzir 2-4 minutos nos primeiros 3 modelos e aproximadamente 8 minutos no *InceptionV3* para as versões não limitadas e 7-9 minutos para todos os modelos para as versões com o disco limitado.

Já o gráfico de Figura 27b representa o cenário de 6 nodos. Mais uma vez o *Monarch* e o *DistMonarch* permitiram obter resultados melhores que a execução base, sendo que, sem limitação do disco local, permitiram poupar aproximadamente 43 e 38 minutos para *Alexnet* e *Lenet*, 27 e 21 minutos para *Resnet18*, 40 e 36 minutos para *Shufflenet*, 7 minutos (para ambas as ferramentas) para *InceptionV3* e 9 e 11 minutos para *VGG19*, respetivamente. Assim, para este cenário, o *Monarch* permitiu poupar 4-6 minutos de treino comparativamente ao *DistMonarch* para os primeiros 4 modelos, enquanto que o *DistMonarch* permitiu poupar 2 minutos comparativamente ao *Monarch* para o *VGG19*. Tal como referido anteriormente, o *Monarch* apresenta tempos menores para a maioria dos modelos, isto pode ser justificado pelo facto de que as instâncias do *DistMonarch* trocam dados entre si ao longo de todas as épocas. Como comparativamente ao cenário anterior este tem mais nodos a pressão na rede é superior, podendo resultar em maiores tempos de partilha de dados. Adicionalmente, como os primeiros 4 modelos são limitados pelos acessos ao sistema de armazenamento essa diferença tem um impacto maior nestes do que nos últimos dois. No que diz respeito à eficiência das ferramentas quando o disco local está limitado a diferença entre elas é igual ou inferior a um minuto, não sendo significativa.

Pedidos ao Lustre

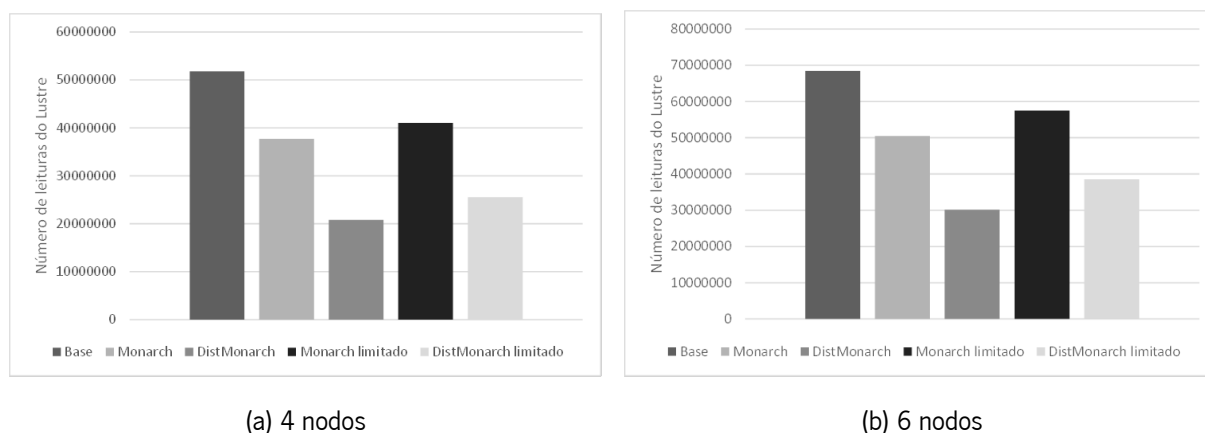


Figura 28: Número de leituras do Lustre de cada modelo no cenário de carga de trabalho heterogénea com 4 e 6 nodos ao longo de 5 épocas de *Alexnet*, *Lenet* e *Shufflenet*, 6 épocas de *Resnet18*, 4 épocas de *InceptionV3* e 2 épocas de *VGG19*.

O gráfico da Figura 28 apresenta o número de leituras feitas ao Lustre por cada uma das ferramentas durante o cenário de carga de trabalho heterogénea para 4 e 6 nodos (gráficos 28a e 28b respetivamente) ao longo de 5 épocas para *Alexnet*, *Lenet* e *Shufflenet*, 6 épocas para *Resnet18*, 4 épocas para *InceptionV3* e 2 épocas para *VGG19*.

Analisando os gráficos da figura é possível verificar que em ambos os cenários o número de leituras feitas ao Lustre com as ferramentas de *tiering* é significativamente menor do que na versão base.

De facto, com 4 nodos, comparativamente à versão base o *Monarch* permitiu poupar 17,9M (26,2%) acessos ao Lustre e a sua versão limitada 10,9M (15,9%), enquanto que o *DistMonarch* poupou 25,3M (36,9%) acessos na versão em que os discos locais não estavam limitados e 18,8M (27,5%) quando estes estão limitados. Comparando estes valores podemos ainda verificar que ambas as versões do *DistMonarch* permitem uma redução mais significativa do número de acessos ao Lustre do que o *Monarch*. Isto resulta do conjunto de dados de treino (517 GiBs) não caber no disco local de cada nodo (110 GiBs ou 60 GiBs na versão limitada), sendo que o *Monarch* tem sempre de ler do Lustre os dados que não ficaram no disco local dos nodos. Já o *DistMonarch* tira partido dos dados guardados no disco local dos vários nodos (110*4 GiBs ou 60*4 GiBs na versão limitada) permitindo reduzir o número de acessos ao Lustre comparativamente aos feitos pelo *Monarch*.

No cenário com 6 nodos o número de acessos feitos pela versão base e por ambas as versões do *Monarch* não sofreram grande variação, o que vai de encontro ao que esperávamos uma vez que o tamanho do

conjunto de dados e dos discos locais usados se mantiveram. Já as versões que correm o *DistMonarch* passaram a poupar 31,0M (59,8%) de acesso e 26,2M (50,6%) para a versão limitada, uma vez como cada nodo tem o seu disco local ao passarmos de 4 para 6 nodos passamos a poder contar com mais espaço de armazenamento e assim maior capacidade de armazenar dados.

Utilização de Recursos

- **CPU**

	InceptionV3	Resnet18	Lenet	Alexnet
Base	77,00	78.89	68.13	55,23
Monarch	70.94	70.47	63.43	57.60
DistMonarch	66.86	68.97	61.58	58.54
Monarch limitado	74,79	76,83	65,44	62,44
DistMonarch limitado	77,63	75.28	66.40	61.03

Tabela 4: Tabela de percentagem de utilização de **CPU** para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.

A tabela 4 apresenta a percentagem média de utilização de **CPU** quando temos 4 nodos a treinar modelos por 3 épocas. Nesta podemos verificar que a versão não limitada do *Monarch* e do *DistMonarch* têm percentagens médias de utilização menores do que a versão base e as versões limitadas para os vários modelos. Esta diminuição é resultante de na primeira época estas ferramentas fazerem cópias dos dados para o disco local, aumentando assim a pressão no sistema de armazenamento e diminuindo a utilização de **CPU** nesta, como se pode verificar nos gráficos da Figura 29. No entanto, nas épocas seguintes a percentagem de utilização aumenta, ficando superior à da versão base. Para 3 épocas este aumento não foi o suficiente para cobrir o impacto da primeira época, no entanto, é previsível que correndo mais épocas estes valores se aproximem. As versões limitadas fazem menos cópias uma vez que têm discos locais com menor capacidade, de modo que o impacto na percentagem de utilização é reduzido.

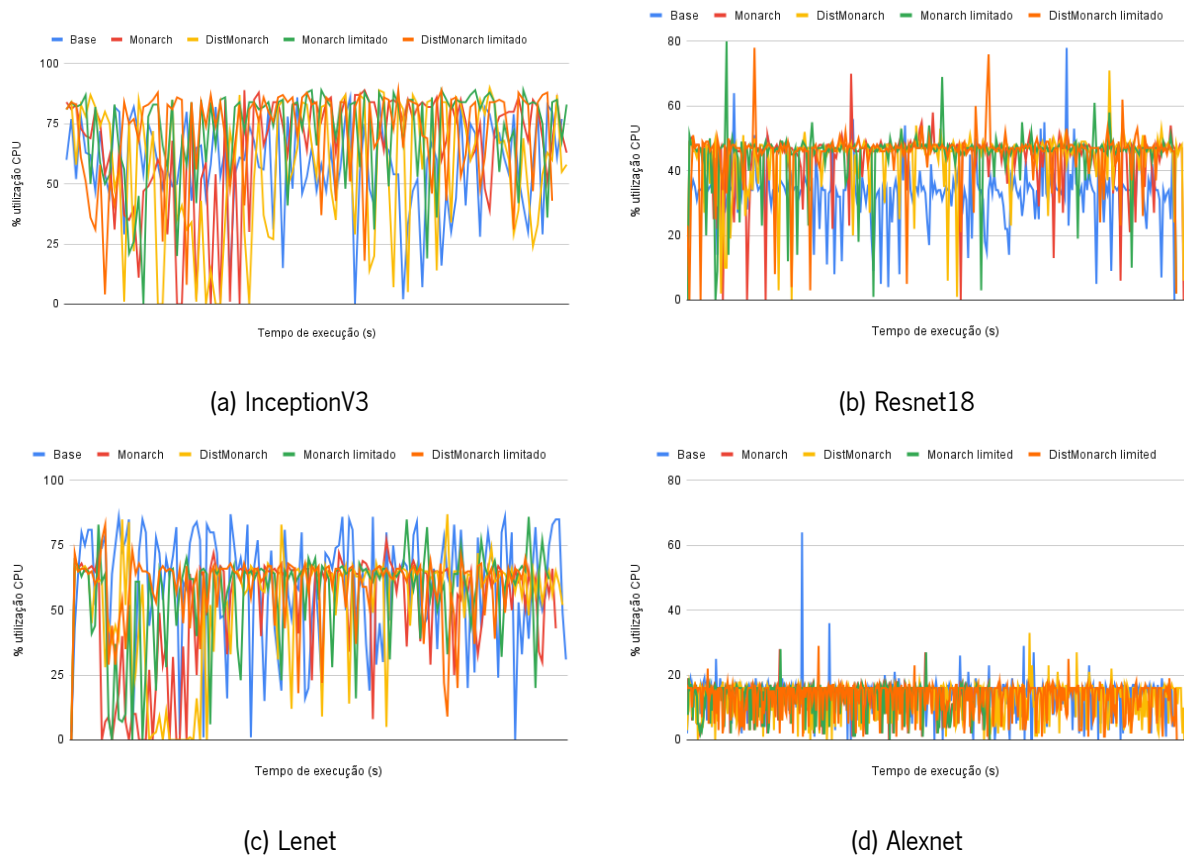


Figura 29: Variação da utilização de **CPU** ao longo de 3 épocas para cada modelo do teste de carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.

	VGG19	InceptionV3	Shufflenet	Resnet18	Lenet	Alexnet
Base	64,92	59,92	55,58	31,09	31,87	13,81
Monarch	70,39	68,86	57,76	39,37	40,36	12,49
DistMonarch	75,37	58,57	58,94	41,88	41,52	12,99
Monarch limitado	77,20	73,51	63,58	44,86	56,05	13,18
DistMonarch limitado	79,39	79,11	65,31	45,97	62,01	13,15

Tabela 5: Tabela de percentagem de utilização de **CPU** para carga de trabalho heterogénea com 6 nodos ao longo de 3 épocas.

Já a Tabela 5 apresenta a percentagem média de utilização de **CPU** quando temos 6 nodos a treinar modelos por 3 épocas. Como temos mais processos a tentar aceder ao mesmo conjunto de dados, a contenção causada pelo acesso a dados torna-se mais significativo e o efeito das ferramentas *Monarch* e *DistMonarch* já se torna mais evidente do que no cenário anterior. Deste modo, a média de utilização de **CPU** dos testes que utilizam ferramentas já ultrapassam a da versão base e a diferença entre os dois tamanhos de disco são menores do que no cenário anterior.

Globalmente não houve uma diferença muito significativa na percentagem de utilização do **CPU** para nenhum dos cenários.

- **GPU**

	InceptionV3	Resnet18	Lenet	Alexnet
Base	80,63	83,86	29,13	28,37
Monarch	79,96	76,23	29,47	28,55
DistMonarch	83,68	84,80	35,18	34,54
Monarch limitado	79,26	83,20	33,40	32,77
DistMonarch limitado	80,74	83,81	33,47	32,85

Tabela 6: Tabela de percentagem de utilização de **GPU** para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.

	VGG19	InceptionV3	Shufflenet	Resnet18	Lenet	Alexnet
Base	96,18	91,58	77,62	74,11	25,78	25,11
Monarch	97,36	86,61	72,00	76,37	30,38	28,33
DistMonarch	97,19	88,43	73,14	74,91	28,57	28,19
Monarch limitado	97,91	91,67	76,09	80,94	32,49	32,02
DistMonarch limitado	98,55	92,24	84,89	83,90	34,25	33,14

Tabela 7: Tabela de percentagem de utilização de **GPU** para carga de trabalho heterogénea com 6 nodos.

As tabelas 6 e 7 apresentam as percentagens médias de utilização de **GPU** para cada um dos modelos treinados quando temos 4 e 6 nodos, respetivamente, ao longo de 3 épocas. Estas percentagens seguem aproximadamente a mesma lógica identificada para a utilização de **CPU**, a primeira época tem uma percentagem de utilização inferior que a versão base, no entanto, o aumento conseguido é ligeiramente mais significativo, pelo que a percentagem de utilização de **GPU** das versões em que o disco local dos nodos não está limitado já apresenta percentagens de utilização maioritariamente superiores à versão base e às versões limitadas, mesmo no cenário com 4 nodos.

	InceptionV3	Resnet18	Lenet	Alexnet
Base	64,86	61,43	15,98	13,06
Monarch	64,18	55,23	16,12	10,88
DistMonarch	66,94	61,29	19,04	10,89
Monarch limitado	63,70	60,65	18,18	12,41
DistMonarch limitado	64,46	61,11	18,31	12,52

Tabela 8: Tabela de percentagem de utilização de memória de **GPU** para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.

	VGG19	InceptionV3	Shufflenet	Resnet18	Lenet	Alexnet
Base	70,50	53,93	58,72	53,46	14,22	9,64
Monarch	68,73	49,77	62,61	55,73	16,62	10,87
DistMonarch	68,89	52,02	58,72	54,13	15,71	10,70
Monarch limitado	69,81	52,83	61,22	59,10	17,66	12,16
DistMonarch limitado	70,07	53,90	68,47	60,76	18,67	12,57

Tabela 9: Tabela de percentagem de utilização de memória de **GPU** para carga de trabalho heterogénea com 6 nodos.

Já as tabelas 8 e 9 apresentam as percentagens médias de ocupação da memória do **GPU** para cada um dos modelos treinados quando temos 4 e 6 nodos, respetivamente, ao longo de 3 épocas. Como se pode perceber analisando-as, a utilização dos sistemas de *tiering* permitiu aumentar, na maioria dos casos, o consumo de memória do **GPU**, sendo que o *DistMonarch* apresenta os valores mais elevados de entre todas as versões.

No entanto, tal como para a utilização de **CPU**, não há variações muito significativas nem na percentagem média de utilização de **GPU**, nem da sua memória entre os vários tipos de execução.

5.2.2 Cargas de Trabalho Homogéneas

Tempos de Treino dos Modelos

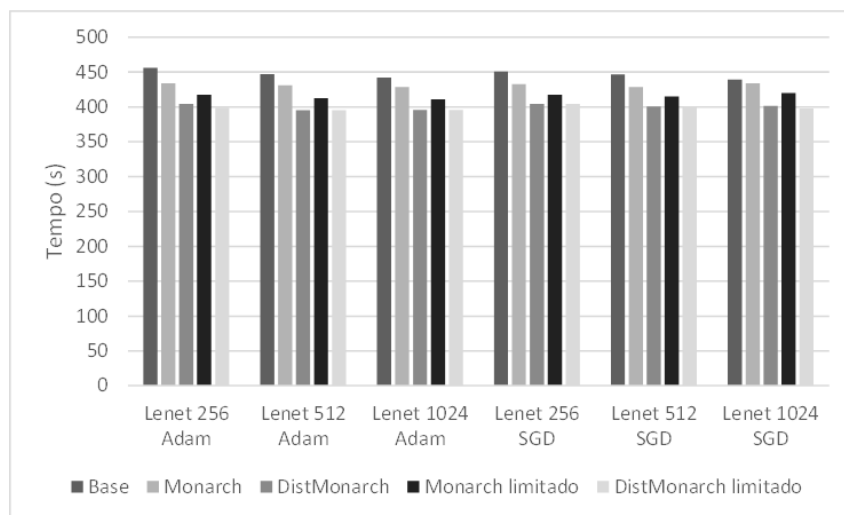


Figura 30: Tempo médio por época para carga de trabalho homogénea com 6 nodos a treinar modelos em simultâneo.

Na Figura 30 está representado o tempo que em média cada época demora quando temos 6 modelos *Lenet* com configurações diferentes a treinar em simultâneo.

Neste podemos verificar que a versão base é a que apresenta tempo médio por época maiores, sendo que o *Monarch* e o *DistMonarch* permitem poupar 1% a 4% e 8% a 11% do tempo da versão base, respetivamente. Já as versões limitadas apresentam uma melhoria ligeiramente mais significativa, passando a aproximadamente 4% a 8% e 9% a 13%, respetivamente. Esta diferença entre as duas versões de cada ferramenta é provavelmente resultante da primeira época de treino que normalmente é mais demorada para as versões não limitadas, porque as ferramentas estão a fazer cópias dos dados para o disco local dos nodos e as versões limitadas fazem menos cópias por terem menos espaço de disco disponível. No entanto, é previsível que quanto mais épocas forem corridas menor seja o impacto desta primeira época no tempo de treino uma vez que as versões em que os discos locais não estavam limitados apresentam tempos menores para as restantes épocas.

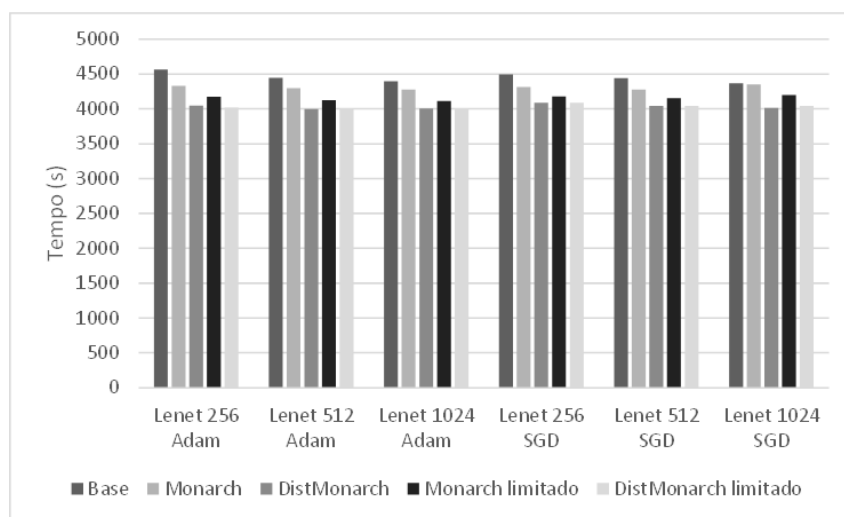


Figura 31: Tempo de execução de 10 épocas para carga de trabalho homogénea com 4 e 6 nodos a treinar modelos em simultâneo.

Analisando o gráfico da Figura 31, que apresenta o tempo que cada tipo de teste demorou a concluir 10 épocas, podemos constatar que o comportamento que verificamos com os tempos médios por época se mantém. Neste podemos verificar que, para 6 nodos, o *Monarch* poupou cerca de 3 minutos, passando a poupar 5 minutos quando o disco é limitado, enquanto que o *DistMonarch* poupou 8 minutos. para ambas as configurações.

Pedidos ao Lustre

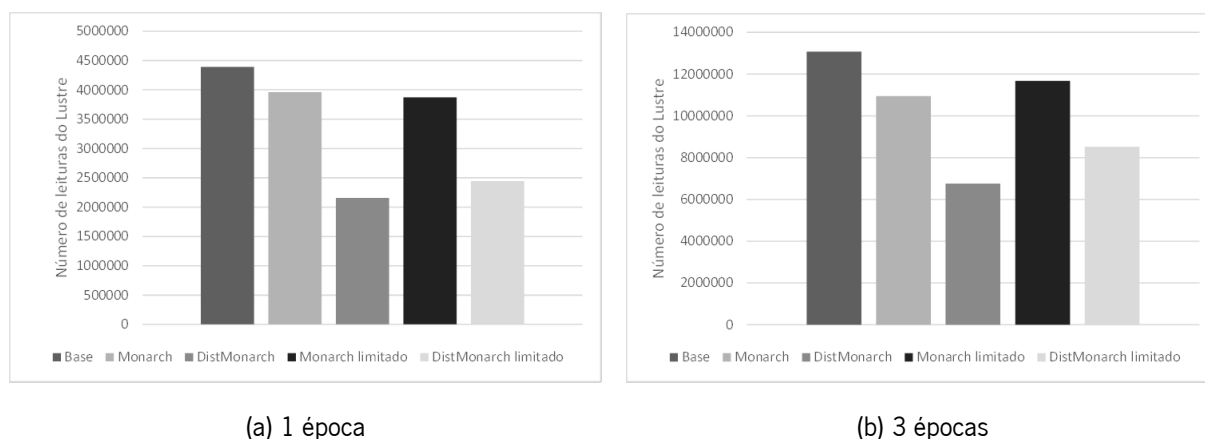


Figura 32: Número de leituras do Lustre para carga de trabalho homogénea com 4 nodos durante 1 e 3 épocas.

Os gráficos da Figura 32 apresentam o número de acessos ao Lustre no cenário de carga de trabalhos homogénea com 4 nodos, para cada um dos tipos de execução, durante 1 e 3 épocas (gráficos 32a e 32b, respetivamente). Como se pode verificar, ambas as ferramentas diminuem o número de acessos comparativamente à versão base, sendo que, ao longo de 4 épocas, o *Monarch* permite poupar mais de 2,1M pedidos (16,3%) e o *DistMonarch* mais de 6,3M (48%) quando os discos locais não estão limitados, passando a 1,4M (10,8%) e 4,6M (34,9%), respetivamente, para as versões com o disco local limitado. É ainda possível verificar que o *DistMonarch* permitiu poupar aproximadamente o dobro dos pedidos evitados pelo *Monarch* para a mesma capacidade do disco local dos nodos e que a versão em que o *DistMonarch* tem o tamanho do disco limitado a 60 GiBs faz menos leituras do disco local do que a versão não limitada do *Monarch*.

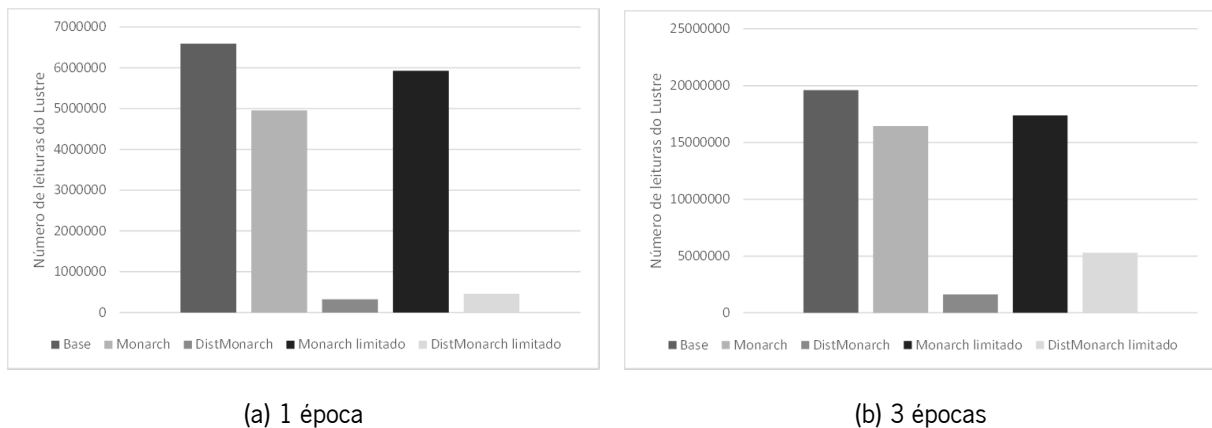


Figura 33: Número de leituras do Lustre para carga de trabalho homogénea com 6 nodos durante 1 e 3 épocas.

Já os gráficos da Figura 33 apresentam o número de acessos ao Lustre para ler dados de treino em cada um dos tipos de execução para 1 e 3 épocas (gráficos 32a e 32b respetivamente) para o cenário de carga de trabalhos homogénea com 6 nodos. Neste é possível verificar que o *Monarch* segue o mesmo padrão que no cenário com 4 nodos, tendo permitido poupar 3,2M (16,2%) e 2,2M (11,4%) acessos ao Lustre comparativamente à versão base, para as versões em que os discos têm 110 GiBs e 60 GiBs respetivamente.

Já o *DistMonarch* passa a ter uma diminuição muito mais significativa agora que conta com 6 nodos, tendo passado a poupar 18M (91,7%) de acessos ao Lustre e 14M (73%) para a versão em que o disco local dos nodos está limitado. Isto resulta do facto de que, neste cenário o conjunto de dados passa a caber no conjunto dos discos locais dos 6 nodos (quando não limitados), pelo que os acessos feitos ao Lustre são só os feitos na primeira época para copiar os dados para os discos locais e alguns que possam ser necessários para obter dados não tenham sido guardados no disco local como consequência da repetição de ficheiros em discos locais diferentes.

Utilização de Recursos

- CPU

	Lenet Adam 256	Lenet Adam 512	Lenet SGD 256	Lenet SGD 512
Base	58,67	59,20	55,62	58,93
Monarch	60,85	57,59	58,19	58,96
DistMonarch	53,92	56,11	65,09	63,07
Monarch limitado	56,14	53,36	52,78	53,38
DistMonarch limitado	62,15	61,19	54,90	53,45

Tabela 10: Tabela de percentagem de utilização de **CPU** para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.

	Lenet Adam 256	Lenet Adam 512	Lenet Adam 1024	Lenet SGD 256	Lenet SGD 512	Lenet SGD 1024
Base	52,43	54,39	57,19	54,12	55,43	52,37
Monarch	51,56	49,49	50,80	54,10	50,38	52,47
DistMonarch	50,18	47,29	50,97	48,28	48,27	48,75
Monarch limitado	60,49	53,71	53,92	53,42	49,79	54,84
DistMonarch limitado	52,98	56,48	66,36	54,88	55,43	52,58

Tabela 11: Tabela de percentagem de utilização de **CPU** para carga de trabalho heterogénea com 6 nodos ao longo de 3 épocas.

As tabelas 10 e 11 apresentam as percentagens médias de utilização de **CPU** para cada um dos modelos treinados nos testes com cargas de trabalhos homogéneas de 4 e 6 nodos, respetivamente. É possível verificar que para ambos os cenários as variações obtidas não são muito significativas, seguindo aproximadamente o mesmo padrão identificado no teste de cargas de trabalho heterogéneas.

- **GPU**

	Lenet Adam 256	Lenet Adam 512	Lenet SGD 256	Lenet SGD 512
Base	36,20	27,71	35,53	26,48
Monarch	33,62	26,12	33,53	25,93
DistMonarch	30,54	22,95	29,48	21,78
Monarch limitado	31,58	24,27	30,90	23,55
DistMonarch limitado	29,17	21,32	26,76	22,47

Tabela 12: Tabela de percentagem de utilização de **GPU** para carga de trabalho heterogénea com 4 nodos ao longo de 3 épocas.

	Lenet Adam 256	Lenet Adam 512	Lenet Adam 1024	Lenet SGD 256	Lenet SGD 512	Lenet SGD 1024
Base	32,43	25,39	21,02	31,82	24,77	21,28
Monarch	28,69	22,44	18,14	28,51	21,66	18,73
DistMonarch	29,60	18,55	16,87	24,12	20,87	19,96
Monarch limitado	31,92	24,39	20,36	30,90	23,65	19,47
DistMonarch limitado	27,36	21,95	21,82	25,29	21,57	16,23

Tabela 13: Tabela de percentagem de utilização de **GPU** para carga de trabalho heterogénea com 6 nodos ao longo de 3 épocas.

As percentagens médias de utilização de **GPU** de cada um dos modelos treinados nos testes com cargas de trabalhos homogéneas de 4 e 6 nodos estão expostas nas tabelas 12 e 13, respetivamente. Mais uma vez as variações verificadas não são muito significativas, mas mantém-se o padrão identificado no cenário de cargas heterogéneas em que as percentagens de utilização médias diminuem ligeiramente por causa da percentagem de utilização na primeira época ser significativamente mais baixa, como consequência da cópia de dados para os discos locais dos nodos.

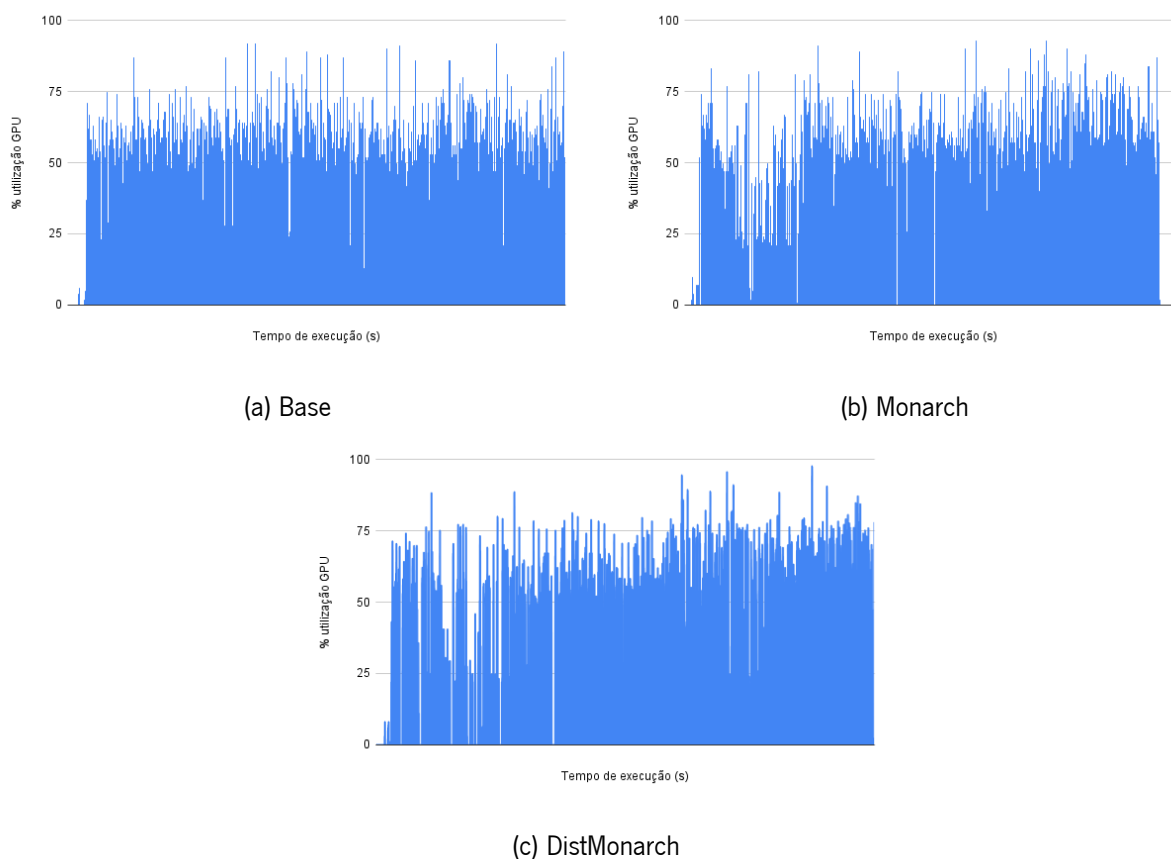


Figura 34: Variação da percentagem de utilização de **GPU** ao longo de 3 épocas para o modelo *Lenet* com o otimizador *Adam* e uma amostra de 256 **KiBs**.

Os gráficos da Figura 34 representam a variação percentagem de utilização de **GPU** ao longo da execução de 3 épocas para o modelo *Lenet* com o otimizador *Adam* e uma amostra de 256 **KiBs** para a versão base, com o *Monarch* e com o *DistMonarch* (34a, 34b e 34c, respetivamente). Como se pode verificar, quando corremos com as ferramentas, a primeira época de treino tem percentagens de utilização significativamente piores. No entanto, nas épocas seguintes atingem valores superiores, pelo que é previsível que correndo mais épocas o impacto desta época vá diminuindo.

	Lenet Adam 256	Lenet Adam 512	Lenet SGD 256	Lenet SGD 512
Base	16,64	15,23	16,12	14,62
Monarch	15,66	14,45	15,45	14,41
DistMonarch	14,25	12,63	13,56	11,37
Monarch limitado	14,63	13,44	14,12	13,05
DistMonarch limitado	13,61	11,83	26,76	12,10

Tabela 14: Tabela de percentagem de utilização de memória de **GPU** para carga de trabalho heterogénea carga de trabalho heterogénea com 4 nodos.

	Lenet Adam 256	Lenet Adam 512	Lenet Adam 1024	Lenet SGD 256	Lenet SGD 512	Lenet SGD 1024
Base	14,90	14,01	13,41	14,42	13,67	13,51
Monarch	13,37	12,49	11,58	13,28	12,03	11,85
DistMonarch	8,91	7,86	7,45	10,88	8,61	7,47
Monarch limitado	14,71	13,47	12,99	14,13	13,06	12,30
DistMonarch limitado	12,53	12,01	13,82	11,48	10,60	10,21

Tabela 15: Tabela de percentagem de utilização de memória de **GPU** para carga de trabalho heterogénea carga de trabalho heterogénea com 6 nodos.

Já as tabelas 14 e 15 apresentam as percentagens médias de ocupação da memória do **GPU** para cada um dos modelos treinados quando temos 4 e 6 nodos, respetivamente, ao longo de 3 épocas. Como se pode perceber comparando a variação destas percentagens com as variações de percentagem média de utilização de **GPU** estas seguem o mesmo padrão. Podemos ainda verificar que não há variações muito significativas na percentagem média de utilização da sua memória dos **GPU**s entre os vários tipos de execução, sendo esta na maioria dos casos inferior a 4%.

5.2.3 Teste Distribuído com ParameterServerStrategy

Tempos de Treino dos Modelos

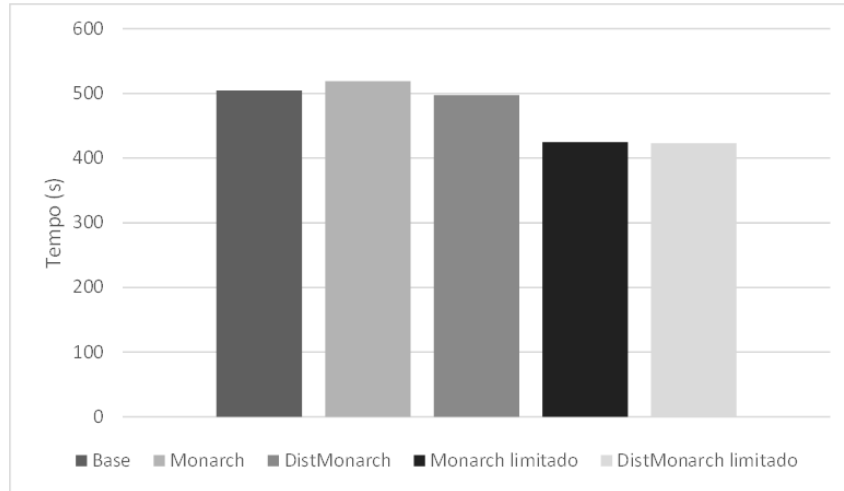


Figura 35: Tempo médio por época para *ParameterServerStrategy* com 4 trabalhadores.

Como se pode verificar no gráfico da Figura 35 a utilização das ferramentas *Monarch* e *DistMonarch* com o disco limitado permitiu diminuir os tempos médios de cada época em aproximadamente 16,7% e 17,1%, respetivamente, comparativamente com a versão base. Já quando os discos locais não estavam limitados, o *DistMonarch* permitiu reduzir o tempo em 2,7%, enquanto que o *Monarch* apresentou tempos 1,6% superiores.

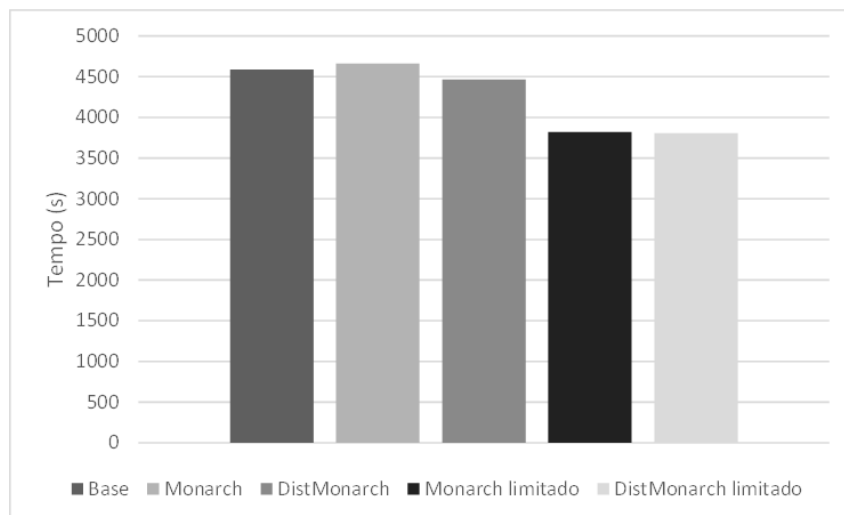


Figura 36: Tempo de execução para *ParameterServerStrategy* com 4 trabalhadores.

O gráfico da Figura 36 apresenta o tempo que cada uma das versões demora a concluir 9 épocas e

demonstra a mesma lógica. Com as versões limitadas do *Monarch* e do *DistMonarch* conseguimos poupar cerca de 12,6 minutos (16,7%) e 13 minutos (17,1%), respetivamente. O *DistMonarch* poupa aproximadamente 2 minutos (2,7%), enquanto que o *Monarch* aumenta o tempo de execução em um minuto (1,5%).

O facto de as versões em que o disco não está limitado apresentarem tempos de execução maiores deve-se ao facto de que como o disco local era maior durante a primeira época as ferramentas tentaram copiar mais dados, apresentando assim tempos maiores de execução nesta época, como se pode verificar na Figura 37.

Este processo implica um aumento da carga do sistema e, consequentemente, do tempo de execução da primeira época. Este é normalmente compensado pelo tempo poupado nas leituras das várias épocas, no entanto, como o tempo de acesso a dados não tem um grande impacto no tempo de execução do *ParameterServerStrategy* os ganhos em termos de tempo ao longo das restantes épocas acabam por o não cobrir. Adicionalmente, para este algoritmo, os dados a processar são distribuídos pelos vários nodos, pelo que nem todos os dados lidos por cada nodo são utilizados para treinar o modelo, não havendo garantias que os dados guardados no disco local vão ser úteis para todas as épocas, diminuindo assim a eficácia destas ferramentas.

Como as versões “limitadas” têm discos locais com menor capacidade de armazenamento menos ficheiros são copiados, pelo que o impacto da cópia de dados no tempo de execução é menor.

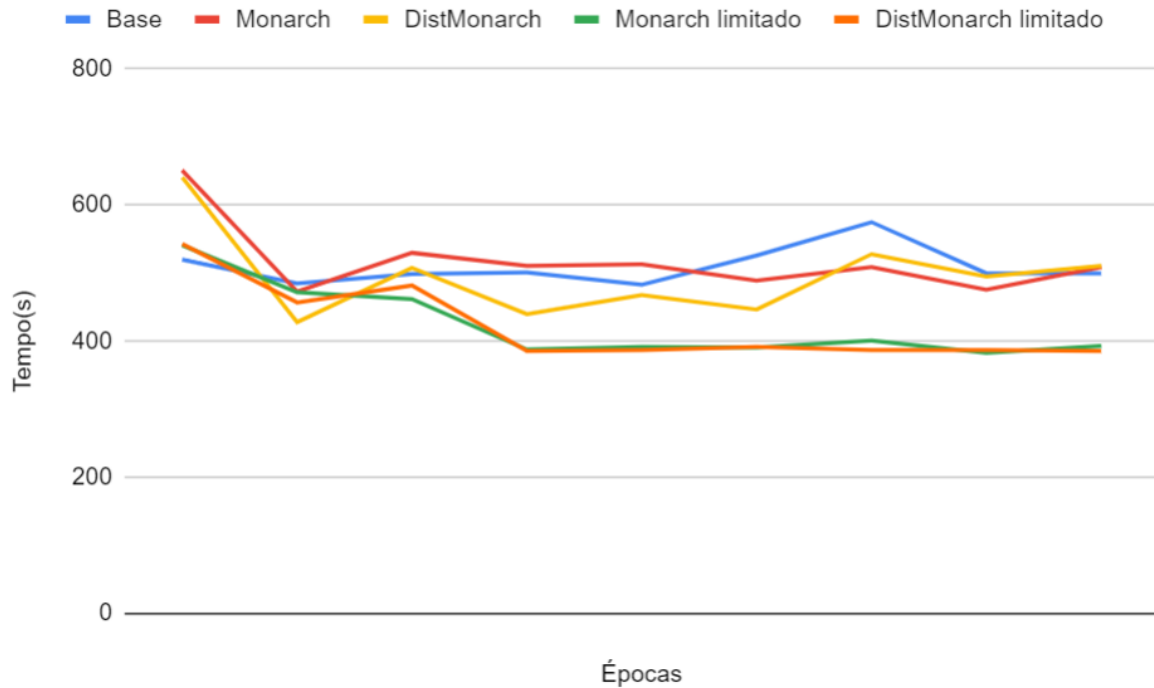


Figura 37: Variação do tempo de cada época ao longo de 9 épocas.

Pedidos ao Lustre

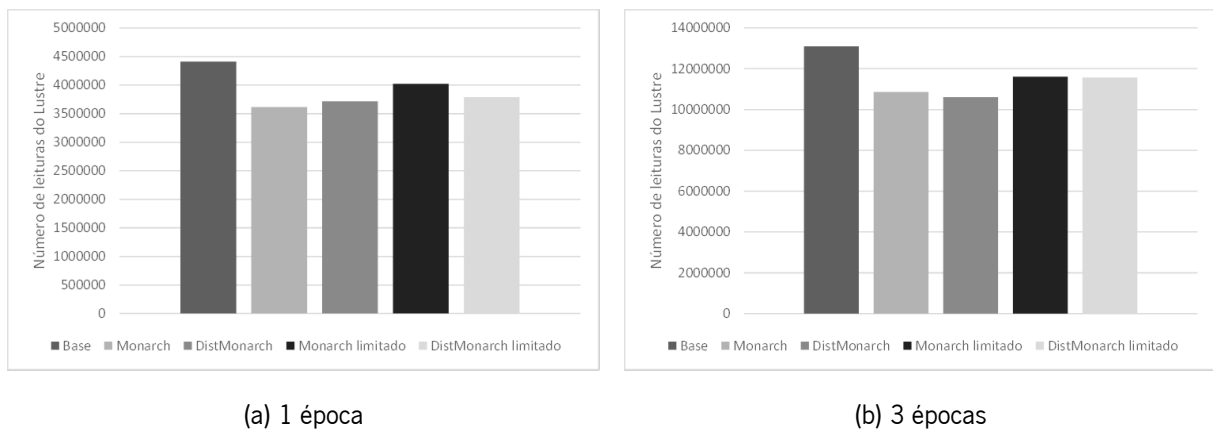


Figura 38: Número de leituras do Lustre para *ParameterServerStrategy* com 4 nodos durante 1 e 3 épocas.

Os gráficos da Figura 38 mostram o número de acessos ao Lustre feitos por cada uma das versões ao longo de 1 e 3 épocas. Como se pode verificar a utilização das ferramentas permite diminuir o número de acessos ao Lustre, sendo que ao longo de 3 épocas o *Monarch* e o *DistMonarch* permitiram poupar 2,2M (menos 17,0% dos acessos) e 2,5M (menos 18,97% dos acessos) acessos ao Lustre, respectiva-

mente. Tal como era previsível, as versões com o disco local limitado diminuíram menos o número de acessos, tendo poupado 1,5M (menos 11,4% dos acessos) e 1,5M (menos 11,7% dos acessos) acessos ao Lustre, respetivamente, comparativamente com a versão base. Para ambas as dimensões de disco local o *DistMonarch* permite uma maior redução da pressão E/S do sistema de ficheiros partilhado.

Utilização de Recursos

- **CPU**

	Chefe	Servidor de Parâmetros	Trabalhadores
Base	0,07	0,89	60,09
Monarch	0,07	0,37	58,04
DistMonarch	0,07	0,92	62,98
Monarch limitado	0,07	0,49	63,99
DistMonarch limitado	0,07	1,41	66,23

Tabela 16: Tabela com percentagem média de utilização de **CPU** ao longo de 3 épocas.

A tabela 16 apresenta a percentagem média de utilização de **CPU** para os três tipos de nodos para os cinco tipos de execução.

O chefe apresenta percentagens de utilização de **CPU** praticamente nulas para todas as versões, o que está de acordo com o que esperávamos, uma vez que este não é responsável nem por processar os dados, nem por fazer a computação dos gradientes. O mesmo ocorre para o servidor de parâmetros, que apesar de já ter percentagens de utilização superiores às do chefe nunca ultrapassam os 2% (correspondente ao cálculo dos gradientes).

Já os nodos trabalhadores aparentam ter uma percentagem de utilização de **CPU** superior e considerável. Comparando a percentagem média de utilização de **CPU** do *Monarch* com a da versão base podemos verificar que a ferramenta apresenta uma taxa menor de utilização. Isto pode ser justificado pelo impacto da primeira época referido anteriormente.

A Figura 39 demonstra como a percentagem de utilização varia ao longo de 3 épocas e permite concluir que, de facto, durante a primeira época as versões que utilizam o *Monarch* e o *DistMonarch* apresentam menor percentagem de utilização que a versão base. No entanto, nas restantes épocas apresentam percentagens de utilização comparáveis ou ligeiramente superiores e sujeitas a menor variabilidade.

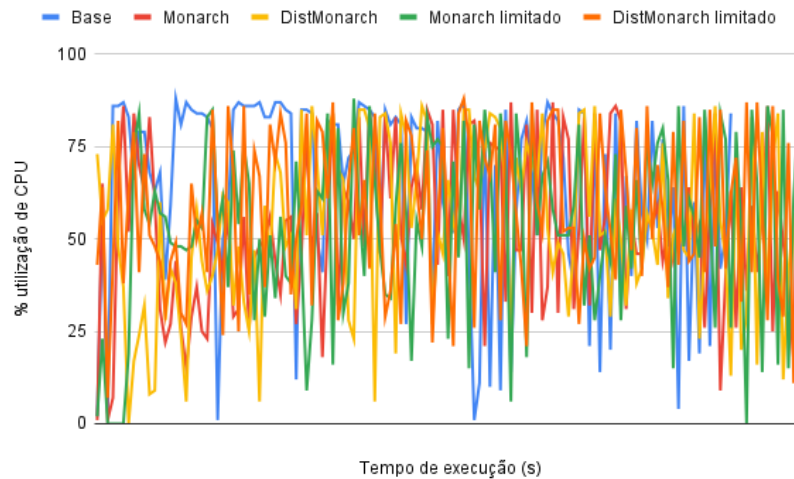


Figura 39: Variação da percentagem de utilização de **CPU** de *ParameterServerStrategy* com 4 trabalhadores ao longo de 3 épocas.

É ainda possível verificar que o *DistMonarch* apresenta taxas de utilização superiores e consequentemente cobrindo melhor o impacto da primeira época do que o *Monarch* e tendo, consequentemente, uma percentagem média de utilização ligeiramente superior. Adicionalmente, como ambas as versões limitadas copiam menos ficheiros para disco local o impacto na utilização de **CPU** na primeira época é menor do que nas versões não limitadas, pelo que foi mais facilmente compensado (o que explica o facto de as percentagens médias de utilização destas serem maiores do que as das versões não limitadas).

- **GPU**

	Chefe	Servidor de Parâmetros	Trabalhadores
Base	0,00	0,00	30,55
Monarch	0,02	0,00	23,31
DistMonarch	0,02	0,00	24,98
Monarch limitado	0,00	0,00	26,42
DistMonarch limitado	0,01	0,00	27,11

Tabela 17: Tabela com percentagem média de utilização de **GPU** ao longo de 3 épocas de *ParameterServerStrategy*.

A tabela 17 expõe a percentagem média de utilização de **GPU** de cada uma das três funções que um nodo pode desempenhar na arquitetura *ParameterServerStrategy* para cada tipo de execução. Podemos verificar que a utilização no nodo chefe e no servidor de parâmetros é praticamente nula

para todos os tipos de execução, o que vai de encontro ao que seria previsível, uma vez que, para as configurações e ferramentas utilizadas, apenas a computação dos dados é feita em **GPU** e isso é da responsabilidade dos trabalhadores.

No que diz respeito à percentagem média de utilização dos **GPUs** dos trabalhadores entre as várias versões não houve variações muito significativas. Tal como verificamos para o **CPU** a primeira época impacta negativamente a percentagem média de utilização dos **GPUs** das versões que correm os sistemas de *tiering*. No entanto, para este cenário, 3 épocas não foi o suficiente para nenhuma das versões compensar o seu impacto, pelo que as percentagens médias de utilização são menores para todos os testes que usam as ferramentas de *tiering*. Podemos, no entanto, verificar que as versões em que os discos locais dos nodos estão limitados apresentam percentagens médias de utilização superiores que as não limitadas (por terem feito menos cópias para disco local, afetando menos a percentagem da utilização de **GPU**).

	Chefe	Servidor de Parâmetros	Trabalhadores
Base	0,00	0,00	13,54
Monarch	0,01	0,00	10,71
DistMonarch	0,00	0,00	11,54
Monarch limitado	0,00	0,00	12,07
DistMonarch limitado	0,00	0,00	12,42

Tabela 18: Tabela com percentagem média de utilização da memória dos **GPUs** ao longo de 3 épocas de *ParameterServerStrategy*.

Já a tabela 18 apresenta a percentagem média de ocupação do **GPU** para os vários tipos de nodos do *ParameterServerStrategy* e permite-nos verificar que estas seguem o mesmo padrão que a percentagem média de utilização dos **GPUs** e que não há variações significativas entre as médias dos vários tipos de execução.

5.2.4 Teste de Longa Duração e Análise de Acurácia

De modo a avaliar o impacto do *DistMonarch* no desempenho do treino de modelos corremos ainda o cenário de cargas de trabalho heterogéneas com 6 nodos durante 40 horas. Estes foram corridos para os mesmos modelos e configurações, mas trocamos o conjunto de dados utilizado, passando a utilizar o *Imagenet*.

Tempos de Treino dos Modelos

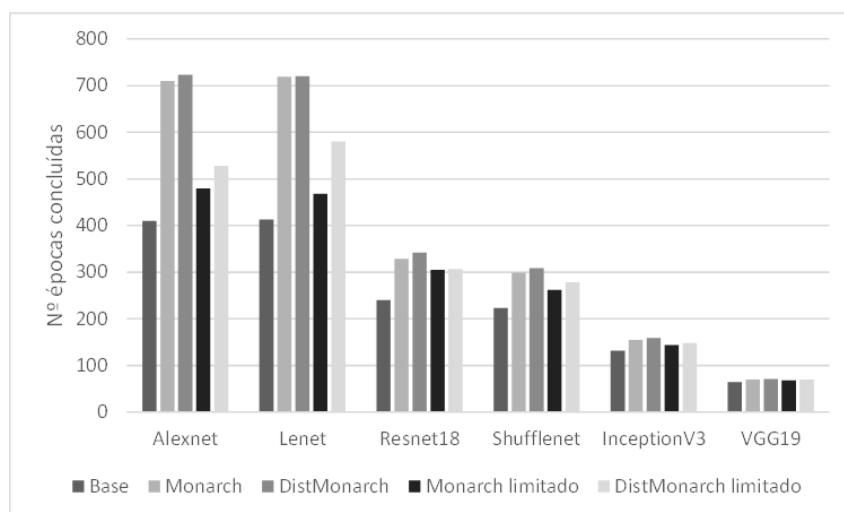


Figura 40: Número de épocas concluídas para cada modelos e tipo de execução em 40 horas.

O gráfico da Figura 40 representa o número de épocas de cada modelo que cada um dos tipos de execução foi capaz de concluir durante 40 horas. Como se pode verificar, apesar de a versão base ser a pior para todos os modelos, os em que o acesso a dados é um fator limitante apresentam uma variação mais significativa quando corridos com as ferramentas de *tiering*, sendo que o *Lenet* e o *Alexnet*, que tínhamos concluído que eram os mais limitados por este fator, apresentam a maior variação.

O *Monarch* permitiu correr mais 6 épocas do modelo *VGG19*, 23 épocas de *InceptionV3*, 76 épocas de *Shufflenet*, 89 épocas de *Resnet18*, 306 épocas de *Lenet* e 300 épocas de *Alexnet*, sendo que, para cada um desses modelos demorou menos 3,39 horas, 5.93 horas, 10.13 horas, 10.78 horas, 16.99 horas e 16.85 horas, respetivamente, a completar o mesmo número de épocas que a versão base.

Já o *DistMonarch* conseguiu completar mais épocas que o *Monarch* e que a versão base para todos os modelos, tendo corrido 7 épocas do modelo *VGG19*, 27 épocas de *InceptionV3*, 86 épocas de *Shufflenet*, 102 épocas de *Resnet18*, 307 épocas de *Lenet* e 313 épocas de *Alexnet*, sendo que, para cada um desses modelos demorou menos 3,89 horas, 6,78 horas, 39,95 horas, 11,91 horas, 17,02 horas e 17,27 horas, respetivamente, a completar o mesmo número de épocas que a versão base.

No que diz respeito às versões em que os discos locais foram limitados a 60 GiBs, tal como esperávamos, ambas as ferramentas concluíram menos épocas do que na sua versão não limitada e a diferença entre as duas ferramentas foi ainda mais significativa.

O *Monarch* com os discos locais limitados concluiu mais 4 épocas de *VGG19*, 12 épocas de *InceptionV3*,

39 épocas de *Shufflenet*, 65 épocas de *Resnet18*, 55 épocas de *Lenet* e 70 épocas de *Alexnet* do que a versão base. Isto significa que, para cada um desses modelos, conclui o mesmo número de épocas que a versão base necessitando de menos 2,32 horas, 3,32 horas, 5.94 horas, 8.51 horas, 4.69 horas e 5.82 horas, respetivamente.

Já o *DistMonarch* com os discos locais limitados completou mais 6 épocas de *VGG19*, 16 épocas de *InceptionV3*, 55 épocas de *Shufflenet*, 67 épocas de *Resnet18*, 167 épocas de *Lenet* e 118 épocas de *Alexnet* que a versão base, em que cada um desses modelos demorou menos 3,41 horas, 4,30 horas, 7,88 horas, 8,72 horas, 11,49 horas e 8,92 horas, respetivamente.

Acurácia dos Modelos

Para cada um dos modelos a curva de variação da acurácia ao longo das épocas para os vários tipos de execução são idênticas. Assim, para o mesmo número de épocas a acurácia alcançada em cada um dos modelos nas várias versões foi similar.

No entanto, como os vários tipos de execução completaram números diferentes de épocas em 40h a acurácia final dos vários modelos foi diferente. Assim, os modelos que não chegaram a convergir no tempo definido apresentam acurácias superiores nas versões que completaram mais épocas. Por exemplo, o *Resnet18* e o *InceptionV3* ficam com acurácia final de 12,99% e 40,54% para a versão base, 27,38% e 51,35% para o *Monarch*, 27,83% e 57,10% para o *DistMonarch*, 12,99% e 41,36% para o *Monarch* limitado e 25,66% e 49,85% para o *DistMonarch* limitado, respetivamente.

5.3 Resumo

Em todos os testes executados foi possível verificar que a utilização das ferramentas de *tiering* permitiu reduzir o número de acessos ao Lustre, sendo que o *DistMonarch* permitiu sempre reduções mais significativas do que o *Monarch*. Isto está de acordo com o que seria previsível uma vez que, enquanto que o *DistMonarch* procura distribuir os dados entre os discos locais dos vários nodos e fazer com que estes partilhem os dados entre si, o *Monarch* apenas tira proveito dos dados guardados no disco local do próprio nodo. A quantidade de leituras do Lustre evitadas com o *DistMonarch* depende da aleatoriedade da ordem na qual os dados são lidos pelos vários nodos, visto que se a ordem dos nodos for muito similar é previsível que os vários discos locais vão armazenar dados duplicados. Assim, **AP** multi-nodo com boa aleatoriedade (como os testes de carga de trabalhos heterogénea e homogénea) vai ter uma percentagem

de acessos salvos superior (podendo chegar até 90%) comparativamente a abordagens com aleatoriedade mais limitada (como o *ParameterServerStrategy*, para o qual ronda os 18%).

Ambas as ferramentas de *tiering* permitem reduzir o tempo de execução de modelos de **AP** cuja execução seja limitada pelo tempo de acesso a dados (como o *Alexnet*, *Lenet*, *Resnet18* e *Shufflenet*). Dependendo do modelo, abordagem e número de nodos que estiverem a ser considerados a ferramenta que apresenta maior impacto no tempo de execução varia. Na globalidade dos cenários o *DistMonarch* apresenta tempos menores do que o *Monarch*, sendo que esta lógica só se inverteu no cenário de cargas de trabalho heterogéneas com 6 nodos, provavelmente devido ao impacto dos *locks* dos mapas do *DistMonarch* e do peso da comunicação na rede.

Normalmente, quando utilizamos estes sistemas, a primeira época demora mais do que as restantes e apresenta percentagens médias de utilização de recursos computacionais menores, como consequência das cópias que estão a ser feitas. No entanto, nas restantes épocas já apresentam percentagens de utilização de ligeiramente superiores à versão base. Este aumento resulta do facto da obtenção de dados ser mais rápida e por consequente os **CPU**s e os **GPU**s passarem menos tempo parados à espera dos dados.

Podemos ainda verificar que nenhuma das ferramentas afeta a curva de variação da acurácia dos modelos. No entanto, quando as utilizamos concluímos o mesmo número de épocas em menos tempo, pelo que no mesmo período de tempo, se o modelo não convergir, somos capazes de atingir melhores acurácias. Como podemos verificar no cenários de 40 horas (com carga de trabalhos heterogénea) o *DistMonarch* conseguiu concluir mais épocas que a versão base e que o *Monarch* para a mesma capacidade de disco local. O que resultou em, depois de treinar durante o mesmo período de tempo, o modelo Resnet18 ter ficado com acurácia de 12,99% para a versão base, 27,38% para o *Monarch*, 27,83% para o *DistMonarch*, 12,99% para o *Monarch* limitado e 25,66% para o *DistMonarch* limitado. Isto revela-se especialmente útil na medida em que quando estamos a treinar em infraestruturas **HPC**, como o Frontera, há um tempo máximo imposto a cada programa durante o qual este pode executar.

Capítulo 6

Conclusões e trabalho futuro

Ao executar programas de **AP** pode haver gargalos na execução causados pelo acesso ao sistema de armazenamento [41, 42, 43, 46, 49]. Quando estes são executados em infraestruturas **HPC**, como os discos locais dos vários nodos não costumam ter tamanho suficiente para guardar conjuntos de dados de grandes dimensões na totalidade, estes são normalmente guardados no sistema de ficheiros partilhado. No entanto, este é partilhado pelos vários utilizadores aumentando assim ainda mais a concorrência pelo seu acesso e, por consequente, o impacto dos acessos a dados nestes programas. Este problema torna-se ainda mais visível quando temos mais do que um nodo a tentar aceder ao mesmo conjunto de dados, como quando estamos a treinar vários modelos diferentes em paralelo.

Já existem diversas abordagens para tentar solucionar este problema como a otimização do carregamento e pré-processamento de dados, desenvolvimento de formatos de ficheiros de dados que permitam leituras mais eficientes, *caching*, substituição de dados ou aproveitamento das várias camadas da hierarquia do sistema de armazenamento para reduzir o tempo de resposta aos pedidos de obtenção de dados. No entanto, estas têm algumas inconveniências associadas como, implicarem alterações no código dos programas **AP**, serem feitas para ferramentas de **AP** específicas [57, 61, 67, 71, 73, 78, 79] ou dependem de fases complexas de preparação de dados e metadados [75]. Adicionalmente, algumas alteram a ordem em que os dados são lidos afetando a acurácia do modelo treinado [57, 78] e várias não estão otimizadas para **AP** multi-nodo [56, 71].

A eficiência de obter dados partilhando-os entre nodos em vez de os ir buscar ao sistema de ficheiros partilhados e o padrão de acesso a dados de programas de **AP** multi-nodo foram averiguados com testes preliminares. Estes permitiram verificar que, para as dimensões normalmente utilizadas para treinar modelos **AP**, a partilha de dados entre nodos é comparável, em termos de desempenho, com lê-los das camadas superiores da hierarquia de armazenamento e compreender diversas características de vários programas de **AP**, como padrões de acesso a dados e fatores limitadores.

Com isto em mente, esta dissertação propôs o *DistMonarch*, um sistema de *tiering*, adaptado a **AP** multi-

nodo, que permite reduzir a carga no sistema de ficheiros partilhado e reduzir o tempo de execução de treino de modelos que tenham o acesso a dados como fator limitador. Para tal partiu-se do *Monarch* [43], um sistema aplicável a qualquer ferramenta **AP** que utilize **POSIX** e que já conseguia atingir os objetivos anteriores, de forma transparente, para o treino de um único modelo local. De modo a adaptá-lo a abordagens multi-nodo passamos a ter em consideração que os restantes nodos podem ter os dados em níveis de armazenamento mais eficientes. Assim, se um nodo não tiver os dados disponíveis em camadas superiores do sistema de armazenamento, verifica se há algum outro nodo que os tenha. Caso tal se verifique, os dados são partilhados entre os nodos.

De modo a validar o funcionamento e eficiência do *DistMonarch*, o protótipo desenvolvido foi testado com diversos modelos de **TF**. Os resultados obtidos demonstram que o *DistMonarch* permite reduzir a pressão **E/S** no sistema de ficheiros paralelo. Sendo que para programas que tenham uma boa aleatoriedade na ordem de consumo de dados consegue diminuir o número de pedidos feitos pelos programas até 90% caso o tamanho conjunto dos discos locais dos nodos seja suficiente para guardar a totalidade dos dados e 48% caso contrário. Já para programas com reduzida aleatoriedade na ordem das leituras dos dados conseguimos reduzir os acessos até cerca de 19% e 12%, para os dois cenários de tamanho conjunto dos discos locais dos nodos anteriores respetivamente. Foi ainda possível verificar que a utilização desta ferramenta permite reduzir o tempo de treino, principalmente para modelos limitados pelos acessos ao sistema de armazenamento em estratégias que apresentem elevada aleatoriedade na ordem de acesso aos dados.

6.1 Perspetiva de Trabalho Futuro

O *DistMonarch* está baseado em conceitos sólidos e alcançou globalmente os objetivos para o qual foi desenhado. No entanto, existem vários fatores que poderia ser interessante analisar e explorar de modo a melhorar a sua eficiência.

Gestão do Armazenamento Distribuído. Atualmente, antes de um ficheiro ser copiado para camadas superiores da hierarquia de memória é verificado se há algum outro nodo que já o tenha feito. No entanto, nada impede dois nodos de fazerem essa verificação em simultâneo e consequentemente um ficheiro ficar guardado por mais do que um nodo. Quando trabalhamos com conjuntos de dados de grandes dimensões e espaço de armazenamento limitado isto pode ser relevante, uma vez que essa repetição representa um desperdício de espaço de armazenamento e uma consequente redução na eficiência de resposta a pedidos de leitura (esse espaço podia ser aproveitado para outro ficheiro do conjunto

de dados de treino, evitando que este fosse obtido da última camada da hierarquia de armazenamento e diminuindo, consequentemente, o tempo de obtenção de dados do mesmo).

Outras configurações de Testes. Seria interessante validar o funcionamento e eficiência do sistema com outros conjuntos de dados para além do OpenImages [23]. Adicionalmente, seria ainda relevante testar o seu comportamento para um teste de maior escala em que fossem treinadas mais modelos do que os utilizados e diversas instâncias de cada um, fazendo variar os hiperparâmetros. Seria ainda relevante testar o sistema em ambientes com sistemas de ficheiros paralelos mais lentos e mais rápidos do que o utilizado.

Adicionalmente, como este sistema foi baseado no *Monarch* [43], algumas das propostas de trabalho futuro mencionadas por este mantêm-se válidas. Nomeadamente:

- **Automatizar algumas das configurações** que neste momento são definidas pelo utilizador, como o número de *threads* disponíveis para puxar dados para outras camadas de armazenamento e a posição que cada camada ocupa na hierarquia de armazenamento, facilitando assim a aplicação do sistema e otimizando o seu desempenho.
- Implementar uma **mecânica que agregasse ficheiros de dados** de forma transparente ao puxar os dados para as camadas superiores do sistema de armazenamento (evitando ter de fazer a transformação das imagens para TFRecords [30] *à priori*).

Bibliografia

- [1] Optimizer adam v2. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam.
- [2] Amazon SageMaker. <https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html>.
- [3] Chainer. <https://chainer.org/>.
- [4] NVIDIA Data Loading Library. <https://developer.nvidia.com/dali>.
- [5] Understanding AI Fraud Detection. <https://www.inscribe.ai/fraud-detection/ai-fraud-detection>, .
- [6] AI-Based Fraud Detection in Banking and Fintech: Use Cases and Benefits. <https://nexocode.com/blog/posts/ai-based-fraud-detection-in-banking-and-fintech-use-cases-and-benefits/>, .
- [7] AI Fraud Prevention. <https://finscience.com/en/blog/alternative-data/ai-fraud-prevention/>, .
- [8] How AI could elevate financial advisor performance. <https://capitalmarketsblog.accenture.com/how-ai-could-elevate-financial-advisor-performance>, .
- [9] The future role of AI in finance. <https://www.worldfinance.com/markets/the-future-role-of-ai-in-finance>, .
- [10] Artificial Intelligence (AI) in Finance. <https://brainpool.ai/ai-in-finance.html>, .
- [11] Augmented reality (AR) and virtual reality (VR) in fashion industry. <https://www.textiletoday.com.bd/augmented-reality-ar-virtual-reality-vr-fashion-industry/>, .

- [12] *Why AR clothing try-on is nearly here.* <https://www.voguebusiness.com/technology/why-ar-clothing-try-on-is-nearly-here>, .
- [13] *Augmented Reality in Fashion.* <https://rockpaperreality.com/insights/ar-use-cases/augmented-reality-in-fashion/>, .
- [14] *NVIDIA: Recommender System.* <https://developer.nvidia.com/blog/how-to-build-a-winning-recommendation-system-part-2-deep-learning-for-recommender-systems/>, .
- [15] *Machine Learning in Recommendation Systems.* <https://www.itransition.com/machine-learning/recommendation-systems>, .
- [16] *IQVIA: AI-DRIVEN PATIENT JOURNEY and Preventing Early Drug Discontinuation.* <https://www.iqvia.com/solutions/commercialization/brand-strategy-and-management/ai-driven-patient-journey> and https://www.iqvia.com/library/case-studies/preventing-early-drug-discontinuation?utm_source=google&utm_medium=cpc&utm_campaign=2022_CaseAIMLPharmCom_GBU_RWS_JM&utm_content=140880680061&utm_term=machine%20learning%20healthcare&gclid=EAIaIQobChMIv620-vSt_AIVCq53Ch0TKQL1EAAYASAAEgLbfvD_BwE, .
- [17] *Keras.* <https://keras.io/>.
- [18] *MirroredStrategy.* https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy.
- [19] *MultiWorkerMirroredStrategy.* https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/MultiWorkerMirroredStrategy.
- [20] *NVIDIA: NCCL: ACCELERATED MULTI-GPU COLLECTIVE COMMUNICATIONS.* <https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf>.
- [21] *Nvme-of.* <https://spdk.io/doc/nvmf.html>.
- [22] *Open Images Dataset.* <https://github.com/cvdfoundation/open-images-dataset>, .
- [23] *Open images dataset v5.* https://storage.googleapis.com/openimages/web/download_v5.html, .

- [24] *Treinamento do servidor de parâmetros com ParameterServerStrategy*. https://www.tensorflow.org/tutorials/distribute/parameter_server_training, .
- [25] *ParameterServerStrategy*. https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/ParameterServerStrategy, .
- [26] Recordio. <https://mxnet.apache.org/versions/1.8.0/api/python/docs/api/mxnet/recordio/index.html>.
- [27] *TACC/REMORA: REsource MOnitoring for Remote Applications*. <https://github.com/TACC/remora>.
- [28] Optimizer `sgd`. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/legacy/SGD.
- [29] Spdk. <https://spdk.io/>.
- [30] Tfrecord. https://www.tensorflow.org/tutorials/load_data/tfrecord.
- [31] Tacc. <https://www.tacc.utexas.edu/>.
- [32] *GNU C Library*. <https://www.gnu.org/software/libc/>.
- [33] Nvidia system management interface. <https://developer.nvidia.com/nvidia-system-management-interface>.
- [34] Deep learning for medical applications with unique data. 2022. URL <https://api.semanticscholar.org/CorpusID:246987867>.
- [35] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. *TensorFlow: A system for large-scale machine learning*. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016.
- [36] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. *Va-run: scalable, low-cost training of massive deep learning models*. *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2021. URL <https://api.semanticscholar.org/CorpusID:243847496>.

- [37] Andrew Audibert, Yangrui Chen, Dan Graur, Ana Klimovic, Jiri Simsa, and Chandramohan A. Thekath. A case for disaggregation of ml data processing. *ArXiv*, abs/2210.14826, 2022. URL <https://api.semanticscholar.org/CorpusID:253116849>.
- [38] Roman Böhringer, Nikoli Dryden, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021. URL <https://api.semanticscholar.org/CorpusID:231662511>.
- [39] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *ArXiv*, abs/1512.01274, 2015. URL <https://api.semanticscholar.org/CorpusID:1507815>.
- [40] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Xian-He Sun, and Gang Chen. icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training. *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 220–232, 2023. URL <https://api.semanticscholar.org/CorpusID:257721149>.
- [41] Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luís Santos, Pawel Herman, Sai B. Narasimhamurthy, and Erwin Laure. Characterizing deep-learning i/o workloads in tensorflow. *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 54–63, 2018. URL <https://api.semanticscholar.org/CorpusID:52939013>.
- [42] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam T. Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/o characterization and performance evaluation of beegfs for deep learning. *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, 2019. URL <https://api.semanticscholar.org/CorpusID:198963479>.
- [43] Marco Dantas, Diogo Leitão, Peter Cui, Ricardo Macedo, Xinlian Liu, Weijia Xu, and João Paulo. Accelerating deep learning training through transparent storage tiering. *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 21–30, 2022. URL <https://api.semanticscholar.org/CorpusID:250708716>.

- [44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. *ImageNet: A large-scale hierarchical image database*. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.
- [45] C. François. *Deep learning with Python*. Novembro de 2021.
- [46] Jingoo Han, Luna Xu, Muhammad M. Rafique, Ali Raza Butt, and Seung-Hwan Lim. A quantitative study of deep learning training on heterogeneous supercomputers. *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019. URL <https://api.semanticscholar.org/CorpusID:202547739>.
- [47] Vishakh Hegde and Sheema Usmani. *Parallel and Distributed Deep Learning*. 2020.
- [48] Andrew J. Hutton and Philipp Schwan. Lustre: Building a file system for 1,000-node clusters. 2003. URL <https://api.semanticscholar.org/CorpusID:59635805>.
- [49] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*, 2022. URL <https://api.semanticscholar.org/CorpusID:246904611>.
- [50] Danlin Jia, Manoj Pravakar Saha, Janki Bhimani, and Ningfang Mi. Performance and consistency analysis for distributed deep learning applications. *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2020. URL <https://api.semanticscholar.org/CorpusID:233197476>.
- [51] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM international conference on Multimedia (ACM MM)*, 2014. URL <https://api.semanticscholar.org/CorpusID:1799558>.
- [52] Aarati Kakaraparthi, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019. URL <https://api.semanticscholar.org/CorpusID:195470859>.
- [53] Can Bülent Karakuş, Rahul Huilgol, Fei Wu, Anirudh Subramanian, Cade Daniel, Derya Çavdar, Teng Xu, Haohan Chen, Arash Rahn timer, and Luis Carlos Quintela. Amazon sagemaker model

- parallelism: A general and flexible framework for large model training. *ArXiv*, abs/2111.05972, 2021. URL <https://api.semanticscholar.org/CorpusID:243985996>.
- [54] Awais Khan, Arnab Kumar Paul, Christopher Zimmer, Sarp H. Oral, Sajal Dash, Scott Atchley, and Feiyi Wang. Hvac: Removing i/o bottleneck for large-scale deep learning applications. *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 324–335, 2022. URL <https://api.semanticscholar.org/CorpusID:252998683>.
- [55] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab Kumar Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali Raza Butt. Shade: Enable fundamental cacheability for distributed deep learning training. In *USENIX Conference on File and Storage Technologies (FAST)*, 2023. URL <https://api.semanticscholar.org/CorpusID:257284156>.
- [56] Michael Kuchnik, Ana Klimovic, Jiri Simsa, George Amvrosiadis, and Virginia Smith. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *ArXiv*, abs/2111.04131, 2021. URL <https://api.semanticscholar.org/CorpusID:243848012>.
- [57] Abhishek Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *USENIX Conference on File and Storage Technologies*, 2020. URL <https://api.semanticscholar.org/CorpusID:211566705>.
- [58] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Z. Chen, Hao Zhang, Joseph E. Gonzalez, and Ioan Cristian Stoica. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *ArXiv*, abs/2302.11665, 2023. URL <https://api.semanticscholar.org/CorpusID:257102392>.
- [59] Jie Liu, Bogdan Nicolae, and Dong Li. Lobster: Load balance-aware i/o for distributed dnn training. *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*, 2022. URL <https://api.semanticscholar.org/CorpusID:255775634>.
- [60] Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito, Weijia Xu, Yusuke Tanimura, Jason H. Haga, and João Paulo. The case for storage optimization decoupling in deep learning frameworks. *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 649–656, 2021. URL <https://api.semanticscholar.org/CorpusID:238752526>.

- [61] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proceedings of the Very Large Data Bases (VLDB) Endowment*, 14:771–784, 03 2021. doi: 10.14778/3446095.3446100.
- [62] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond gpus for dnn scheduling on multi-tenant clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022. URL <https://api.semanticscholar.org/CorpusID:251950092>.
- [63] A. Y. Ng. *Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance*. 2004.
- [64] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, , and A. Lerer. *Automatic Differentiation in PyTorch*. Neural Information Processing Systems (NIPS) 2017 Workshop Autodiff, 2017.
- [65] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *ArXiv*, abs/1712.04621, 2017. URL <https://api.semanticscholar.org/CorpusID:12219403>.
- [66] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *ArXiv*, abs/1802.05799, 2018. URL <https://api.semanticscholar.org/CorpusID:3398835>.
- [67] Kazuhiro Serizawa and Osamu Tatebe. Accelerating machine learning i/o by overlapping data staging and mini-batch generations. *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)*, 2019. URL <https://api.semanticscholar.org/CorpusID:208334200>.
- [68] Dan Stanzione, John West, Richard Todd Evans, Tommy Minyard, Omar Ghattas, and Dhabaleswar K. Panda. Frontera: The evolution of leadership computing at the national science foundation. *Practice and Experience in Advanced Research Computing (PEARC)*, 2020. URL <https://api.semanticscholar.org/CorpusID:220666751>.
- [69] Ivan Svogor, Christian Eichenberger, Markus Spanring, Moritz Neun, and Michael Kopp. Profiling and improving the pytorch dataloader for high-latency storage: A technical report. *ArXiv*, abs/2211.04908, 2022. URL <https://api.semanticscholar.org/CorpusID:253420450>.

- [70] Sahil Tyagi and Prateek Sharma. Scavenger: A cloud service for optimizing cost and performance of ml training. *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 403–413, 2023. URL <https://api.semanticscholar.org/CorpusID:257495907>.
- [71] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. *Proceedings of the 49th International Conference on Parallel Processing (ICPP)*, 2020. URL <https://api.semanticscholar.org/CorpusID:221070171>.
- [72] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2022. URL <https://api.semanticscholar.org/CorpusID:238209483>.
- [73] Chih-Chieh Yang and Guojing Cong. Accelerating data loading in deep neural network training. *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 235–245, 2019. URL <https://api.semanticscholar.org/CorpusID:203641791>.
- [74] Y. Yao, L. Rosasco, and A. Caponnetto. *On Early Stopping in Gradient Descent Learning*. 2007.
- [75] Zhao Zhang, Lei Huang, Uri Manor, Linjing Fang, Gabriele Merlo, Craig Michoski, Jack Cazes, and Niall Gaffney. Fanstore: Enabling efficient and scalable i/o for distributed deep learning. *ArXiv*, abs/1809.10799, 2018. URL <https://api.semanticscholar.org/CorpusID:52891043>.
- [76] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. Silod: A co-design of caching and scheduling for deep learning clusters. *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*, 2023. URL <https://api.semanticscholar.org/CorpusID:258508799>.
- [77] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. *Proceedings of the Association for Computing Machinery's Special Interest Group on Data Communications (ACM SIGCOMM) 2022 Conference*, 2022. URL <https://api.semanticscholar.org/CorpusID:251496057>.

- [78] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam T. Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156, 2018. URL <https://api.semanticscholar.org/CorpusID:53283294>.
- [79] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam T. Moody, and Fahim Chowdhury. Efficient user-level storage disaggregation for deep learning. *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019. URL <https://api.semanticscholar.org/CorpusID:207960411>.

Este trabalho é financiado por fundos nacionais através da FCT – Fundação para a Ciência e a Tecnologia, I.P., no âmbito dos projetos UIDP/50014/2020 e LA/P/0063/2020.