Sistema de Ficheiros Distribuído para Treino Eficiente de Modelos de Aprendizagem Profunda

Henrique Pereira
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg57876@alunos.uminho.pt

Pedro Oliveira
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg55093@alunos.uminho.pt

Simão Antunes
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg57903@alunos.uminho.pt

Abstract

Este artigo apresenta o desenvolvimento de um sistema de ficheiros distribuído otimizado para workloads de treino de modelos de deep learning. A solução proposta utiliza a plataforma FUSE para garantir compatibilidade POSIX com aplicações como PyTorch e implementa replicação passiva para garantir a disponibilidade dos dados em caso de falhas. Como otimização de desempenho, foi desenvolvida uma cache LRU. A avaliação experimental foi conduzida utilizando o MLPerf™ Storage Benchmark Suite, simulando cargas de trabalho realistas de treino de modelos ResNet50 sobre aceleradores NVIDIA H100.

Keywords

Distributed File System, Deep Learning, FUSE, POSIX, Replication, Caching, Fault Tolerance, Storage Benchmarking, MLPerf

ACM Reference Format:
Henrique Pereira, Pedro Oliveira, and Simão Antunes. 2025. Sistema de Ficheiros Distribuído para Treino Eficiente de Modelos de Aprendizagem Profunda. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introdução

O grande investimento na área de deep learning tem levado a um crescimento exponencial no volume de dados processados e na complexidade dos modelos utilizados. Em particular, o treino de modelos de deep learning sobre aceleradores como GPUs de alto desempenho exige sistemas de armazenamento de dados capazes de fornecer uma elevada taxa de transferência, baixa latência e alta disponibilidade dos dados. Os sistemas de ficheiros tradicionais frequentemente não conseguem satisfazer estes requisitos (nomeadamente a alta disponibilidade), tornando-se um gargalo para a eficiência do processo de treino.

Este artigo aborda o desafio de desenhar e implementar um sistema de ficheiros distribuído otimizado para cargas de trabalho intensivas típicas do treino de modelos de deep learning. O sistema foi desenvolvido com base na plataforma FUSE, garantindo compatibilidade com a interface POSIX e, por conseguinte, com frameworks amplamente utilizadas como o PyTorch. Uma das principais dificuldades consiste em encontrar um equilíbrio entre desempenho e resiliência: garantir um acesso rápido aos dados sem comprometer a sua integridade e disponibilidade em caso de falhas.

Neste contexto, os desafios concretos de investigação explorados incluem:

- Como garantir compatibilidade POSIX num sistema distribuído baseado em FUSE, sem comprometer a escalabilidade?
- Como implementar tolerância a falhas eficaz, com replicação passiva de dados, minimizando o overhead de escrita?
- Que mecanismos de caching podem ser aplicados para reduzir a latência de acesso aos dados em workloads de treino?
- Como validar empiricamente a eficácia das otimizações introduzidas, utilizando benchmarks realistas?

As principais contribuições deste trabalho são:

- A conceção e implementação de um sistema de ficheiros distribuído com suporte POSIX, orientado a workloads de deep learning.
- Um mecanismo de replicação passiva para tolerância a falhas, assegurando a disponibilidade dos dados em cenários de falha parcial.
- O desenvolvimento de uma cache LRU local no cliente, reduzindo a latência de acesso a dados recentemente utilizados e o uso da rede.
- Uma avaliação experimental utilizando o benchmark MLPerf™ Storage, demonstrando os ganhos de desempenho com treino de modelos ResNet50 sobre aceleradores NVIDIA H100.

Os resultados obtidos evidenciam a viabilidade da abordagem proposta, mostrando melhorias significativas em termos de latência e throughput, sem comprometer a resiliência do sistema. Este artigo documenta o processo de conceção, implementação e avaliação da solução desenvolvida, discutindo as decisões técnicas e os trade-offs associados.

2 Desenho, Arquitetura e Algoritmos

Com o objetivo de fornecer uma solução eficiente e resiliente para o armazenamento de dados em workloads de treino de modelos de deep learning, foi concebido e implementado um sistema de ficheiros distribuído que combina compatibilidade POSIX com tolerância a falhas e otimizações de desempenho. Esta secção descreve o desenho da solução proposta, a sua arquitetura modular e os principais algoritmos envolvidos no tratamento de operações de leitura e escrita, assim como nos mecanismos de cache e replicação.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

2.1 Arquitetura

A arquitetura do sistema de ficheiros distribuído proposto baseia-se num modelo cliente-servidor com três componentes principais: Cliente FUSE, Servidor de Metadados (NameNode) e Servidores de Dados (DataNodes). Esta arquitetura inspira-se em sistemas como o HDFS, mas foi desenhada e otimizada especificamente para cargas de trabalho de treino de modelos de deep learning com requisitos elevados de desempenho e disponibilidade.

- Cliente FUSE: Executa localmente na máquina do utilizador e interage com o sistema de ficheiros via a interface POSIX. Está responsável por gerir uma cache de blocos local e por intermediar os pedidos de leitura e escrita.
- Servidor de Metadados (NameNode): Mantém uma hashtable em memória com os metadados dos ficheiros — nomeadamente, a correspondência entre ficheiros e os DataNodes que os armazenam. Escreve também um log persistente para recuperação dos metadados em caso de falha.
- Servidores de Dados (DataNodes): Responsáveis por armazenar os ficheiros. Cada ficheiro é dividido em blocos de tamanho fixo (4kb) e transmitido sobre a rede entre o cliente e o servidor de dados, sendo cada bloco replicado de forma passiva para um ou mais DataNodes adicionais, garantindo assim tolerância a falhas.

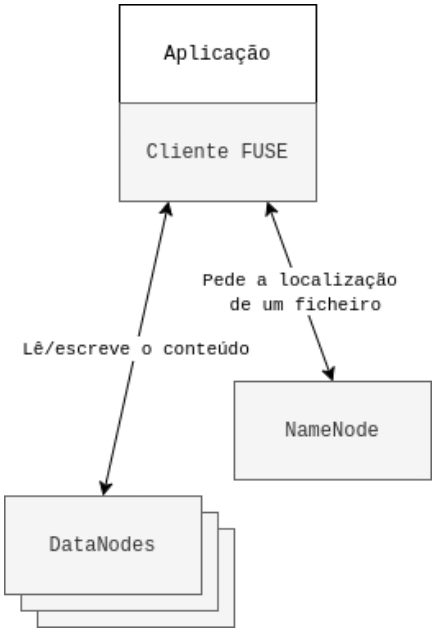

Figure 1: Componentes Principais da arquitetura

2.2 Implementação

O sistema foi implementado em C, utilizando a biblioteca FUSE para integração com o sistema de ficheiros. A comunicação entre clientes, NameNode e DataNodes é realizada com sockets TCP, usando threads dedicadas para aceitar conexões concorrentes. As estruturas de dados principais (como a cache LRU e a tabela de metadados) foram implementadas com base nas bibliotecas da GLib, garantindo eficiência e simplicidade de uso.

2.3 Comunicação

A comunicação entre os diferentes componentes do sistema (cliente, NameNode e DataNodes) é realizada através de mensagens serializadas sobre sockets TCP. Todas as mensagens seguem um formato comum com os seguintes campos principais:

Listing 1: Estrutura da mensagem (MSG) usada na comunicação entre componentes

```
typedef struct msg{
    int op;
    char path[PSIZE];
    char buffer[BSIZE];
    char servers[MAX_REPLICAS][PSIZE];
    int server_count;
    char primary_server[PSIZE];
    off_t offset;
    size_t size;
    ssize_t res;
    mode_t mode;
    struct stat st;
} MSG;
```

2.4 Fluxo de Operações

Escrita de Ficheiros - O processo de escrita é iniciado pelo cliente, seguindo os seguintes passos:

(1) **Pedido de Escrita**: O cliente envia ao NameNode um pedido *WRITE* para escrever um novo ficheiro.
(2) **Seleção de Servidores**: O NameNode escolhe um DataNode principal (primário) para armazenar o bloco, e um ou mais DataNodes secundários para manter réplicas. A escolha do servidor primário é baseada numa política simples de rotação para evitar escolher sempre o mesmo.
(3) **Resposta ao Cliente**: O NameNode devolve ao cliente os endereços dos servidores selecionados.
(4) **Transmissão de Dados**: O cliente envia o bloco de dados diretamente ao DataNode primário.
(5) Replicação Passiva: O DataNode primário, após armazenar o bloco, inicia a replicação assíncrona para os DataNodes secundários.
(6) **Atualização de Metadados**: O DataNode primário notifica o NameNode da conclusão, enviando os metadados (blocos e locais).
(7) **Persistência de Estado**: O NameNode atualiza a sua hashtable e escreve os metadados num log persistente para garantir resiliência.

Este processo garante alta disponibilidade dos dados e separação entre dados e metadados, melhorando a escalabilidade do sistema.

Leitura de Ficheiros - O fluxo de leitura segue uma abordagem otimizada com cache local:

(1) **Verificação na Cache Local**: O cliente verifica se o bloco solicitado está presente na sua cache LRU local. Se estiver, serve diretamente o pedido.

(2) **Pedido de Localização**: Caso contrário, envia ao NameNode um pedido *READ* para saber quais servidores armazenam o bloco pretendido.
(3) **Seleção do Servidor**: O cliente escolhe um dos servidores disponíveis através de um esquema round-robin, distribuindo a carga uniformemente.
(4) **Pedido ao DataNode**: O cliente contacta o DataNode selecionado e obtém o bloco de dados.
(5) **Atualização da Cache**: O bloco é armazenado na cache LRU local para futuras leituras mais rápidas.

Este mecanismo de cache reduz significativamente a latência de leitura e o tráfego na rede, melhorando o desempenho para workloads com acesso repetido a ficheiros de treino.

2.5 Cache

Para otimizar o desempenho de leitura, foi desenvolvido um mecanismo de cache local no cliente, que guarda blocos de ficheiros recentemente acedidos. A implementação recorre a duas estruturas principais da GLib:

- Uma GHashTable para acesso rápido aos blocos armazenados, indexados por identificador (path + offset);
- Uma GQueue que mantém a ordem de acesso dos blocos, funcionando como uma lista de prioridades para a política Least Recently Used (LRU).

Quando um bloco é lido pela primeira vez, é armazenado na cache, e a sua referência é adicionada à cauda da GQueue. Se um bloco já existir, é movido para o fim da fila, representando o acesso recente.

A gestão da capacidade da cache é assegurada por uma thread de evicção dedicada. Esta thread corre em segundo plano e remove periodicamente os blocos mais antigos da cache sempre que o limite máximo configurado (em número de blocos) é atingido. O processo de remoção atualiza ambas as estruturas (GQueue e GHashTable) de forma sincronizada com GMutex para garantir consistência em ambientes multithreaded.

Este sistema de cache reduz significativamente o número de acessos remotos aos DataNodes, o que é particularmente eficaz para workloads repetitivas, típicas de treino de redes neuronais convolucionais (e.g., ResNet50).

2.6 Estratégia de Replicação

A replicação passiva foi implementada como forma de garantir tolerância a falhas sem introduzir bloqueios no caminho crítico da escrita. O processo ocorre em duas fases:

(1) Fase de Escrita Primária: O NameNode escolhe um DataNode primário para armazenar o bloco. O cliente envia os dados diretamente a este servidor.
(2) Fase de Replicação Assíncrona: Após armazenar localmente o bloco, o DataNode primário inicia a replicação para os restantes DataNodes definidos como réplicas (normalmente 1 ou 2). Esta replicação é feita de forma assíncrona, através de uma thread interna, que contacta cada réplica e transmite o conteúdo do bloco.

Este modelo de replicação permite alcançar alta disponibilidade sem impactar a latência de escrita percebida pelo cliente, uma vez que a confirmação da escrita é enviada ao cliente logo após a persistência no primário.

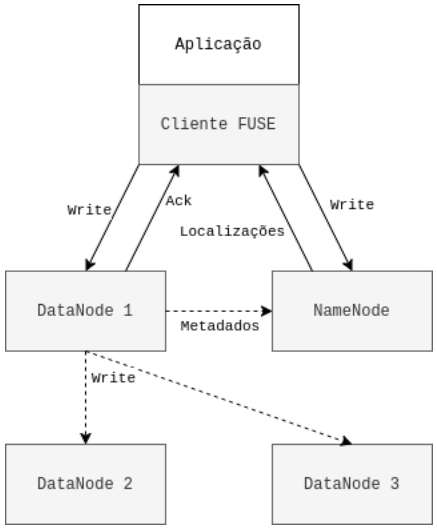

Figure 2: Fluxo de escrita e replicação

3 Avaliação Experimental

3.1 Objetivos da Avaliação

A avaliação experimental visa responder às seguintes questões principais:

- Qual é o impacto da cache local na performance do sistema em cargas de treino de modelos de deep learning?
- O sistema consegue escalar eficientemente para workloads realistas com aceleradores modernos (e.g., NVIDIA H100)?
- Como variam a taxa de transferência e a utilização do acelerador em função do tamanho da cache?

3.2 Metodologia

A avaliação foi conduzida utilizando a suíte MLPerf™ Storage Benchmark, com o cenário de treino da arquitetura ResNet50 sobre uma única GPU NVIDIA H100. Os testes foram realizados localmente com comunicação entre cliente, NameNode e DataNodes via TCP. Cada teste executou 5 épocas de treino sobre um dataset simulado com 2 ficheiros.

Foram testadas quatro configurações principais:

- Cache local de 1 MB
- Cache local de 10 MB
- Cache local de 100 MB
- Sem cache local (cache desativada no cliente)

Cada configuração foi avaliada com métricas como:

- Throughput de treino (amostras por segundo)
- Utilização do acelerador (%)
- Throughput de I/O (MB/s)
- Tempo por época

3.3 Resultados obtidos

Os resultados médios obtidos estão sintetizados na Tabela 1:

Table 1: Métricas de desempenho para diferentes tamanhos de cache

Configuração	Throughput (samples/s)	I/O (MB/s)	AU (%)
Sem cache	**501.04**	**31.31**	**35.11**
Cache 1 MB	381.38	23.83	26.71
Cache 10 MB	385.82	24.11	27.03
Cache 100 MB	370.05	23.13	25.94

3.4 Discussão dos Resultados

Surpreendentemente, a configuração sem cache local apresentou o melhor desempenho global, com um throughput de 501 amostras/segundo e uma utilização média do acelerador de 35,11%. Este comportamento pode ser explicado por diversos fatores:

- Evicção frequente na cache: Nas configurações com cache pequena (1 MB ou 10 MB), os blocos acedidos podem estar a ser continuamente descartados, o que introduz overhead adicional de gestão da cache (e sincronização com mutexes) sem ganho real em reutilização.
- Sobrecarga da política LRU: A gestão da cache com GQueue e GHashTable, embora eficiente em teoria, pode ter impacto negativo na latência em cenários de alta concorrência e acesso sequencial com pouca reutilização de blocos.
- Dataset demasiado pequeno: O benchmark alerta que o dataset cabe inteiramente em memória após a primeira época. Assim, com cache ativada, o sistema pode estar a competir com a cache do próprio sistema operativo ou com efeitos de file system caching, gerando interferência inesperada.
- Simetria das leituras: Sem cache, cada leitura provoca um pedido ao servidor de dados, o que mantém um fluxo contínuo de dados em rede. Esse padrão linear e previsível pode beneficiar de TCP congestion control, buffers otimizados e menor latência de lookup comparado com a verificação local na cache.

Em contraste, as configurações com cache (principalmente com 100 MB) mostraram menor throughput, possivelmente devido à sobreposição de caches (cliente + SO) e falta de reutilização de dados suficientemente frequente para justificar o uso local.

4 Trabalho Relacionado

Vários sistemas de ficheiros distribuídos foram propostos com o objetivo de fornecer alto desempenho e resiliência em ambientes de computação intensiva. O Hadoop Distributed File System (HDFS) é um dos exemplos mais conhecidos, com uma arquitetura centralizada semelhante à adotada neste trabalho (NameNode e DataNodes). No entanto, o HDFS não oferece suporte POSIX direto, limitando a sua utilização com frameworks modernas como PyTorch.

O CephFS representa uma solução POSIX-compliant mais moderna, baseada em replicação ativa e um modelo descentralizado de metadados. Embora bastante robusto, a sua complexidade torna-o difícil de adaptar e otimizar para workloads específicos de deep learning em clusters pequenos.

Em contexto académico, destacam-se também projetos como o TensorFlow File System (TFFS) e soluções de file caching aware for AI workloads, como discutido em artigos recentes de otimização de I/O em GPU clusters [1,2]. Estas soluções motivaram a inclusão de cache local e replicação no sistema aqui proposto, ainda que numa versão mais simplificada e focada no controlo total do código base.

5 Conclusão e Trabalho Futuro

Este artigo apresentou o desenvolvimento de um sistema de ficheiros distribuído orientado ao suporte eficiente de workloads de deep learning. A solução implementada oferece compatibilidade POSIX via FUSE, resiliência através de replicação passiva e uma cache local com política LRU. A avaliação experimental mostrou que, em determinadas configurações, a cache local pode não melhorar o desempenho, reforçando a importância de testes com workloads realistas.

Como trabalho futuro, identificam-se duas linhas de evolução promissoras:

- Data Striping entre DataNodes: A atual divisão de ficheiros em blocos é linear e cada bloco é armazenado integralmente num único servidor. A adoção de técnicas de data striping, com fragmentação e distribuição de blocos entre múltiplos servidores, poderá permitir maior paralelismo e throughput agregado, essencial para escalar o sistema a múltiplos aceleradores em paralelo.
- Distribuição dos Metadados: Atualmente, o NameNode representa um ponto único de falha e um potencial gargalo. Pretende-se explorar abordagens com múltiplos NameNodes distribuídos ou uma arquitetura baseada em consenso (ex: Raft, Paxos) para replicação consistente dos metadados, melhorando a escalabilidade e a tolerância a falhas.

Estas melhorias visam transformar o protótipo atual numa solução mais robusta e escalável, adequada a ambientes de produção distribuída em larga escala.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009