UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

Otimizando estruturas de grafos em memória persistente para arquiteturas NUMA

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Autor: Lucas Bastelli Spagnol
Orientador: Prof. Dr. Alexandro José Baldassin
Coorientador: Prof. Dr. Emílio de Camargo Francesquini

Rio Claro — SP
2025

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

Lucas Bastelli Spagnol

Otimizando estruturas de grafos em memória persistente para arquiteturas NUMA

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Alexandro José Baldassin
Coorientador: Prof. Dr. Emílio de Camargo Francesquini

Rio Claro — SP
2025

S733o

Spagnol, Lucas Bastelli
Otimizando estruturas de grafos em memória persistente para arquiteturas NUMA / Lucas Bastelli Spagnol. -- , 2025
64 f. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (UNESP), Instituto de Geociências e Ciências Exatas, Rio Claro,
Orientador: Alexandro José Baldassin
Coorientador: Emílio de Camargo Francesquini

1. Ciência da computação. 2. Teoria dos grafos. 3. Arquitetura de computador. 4. Computação paralela. 5. Sistemas de memória de computadores. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Dados fornecidos pelo autor(a).

Impacto potencial desta pesquisa

Esta pesquisa otimiza o processamento de grafos dinâmicos em memória persistente com técnicas conscientes de NUMA, acelerando análises em grande escala. Os resultados reduzem tempo de execução e consumo de energia em aplicações como redes sociais e sistemas de recomendação, contribuindo para infraestrutura computacional mais eficiente.

Potential impact of this research

This research optimizes dynamic graph processing on persistent memory using NUMA-aware techniques, speeding up large-scale analyses. The results reduce runtime and energy consumption in applications such as social networks and recommender systems, contributing to more efficient computing infrastructure.

UNIVERSIDADE ESTADUAL PAULISTA
“Júlio de Mesquita Filho”
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

Lucas Bastelli Spagnol

Otimizando estruturas de grafos em memória persistente para arquiteturas NUMA

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Comissão Examinadora

Prof. Dr. Alexandro José Baldassin
Universidade Estadual Paulista (UNESP)

Prof. Dr. Emílio de Camargo Francesquini
Universidade Federal do ABC (UFABC)

Prof. Dr. Orlando de Andrade Figueiredo
Universidade Estadual Paulista (UNESP)

Prof. Dr. Rodolfo Azevedo
Universidade Estadual de Campinas (UNICAMP)

Conceito: Aprovado.

Rio Claro (SP), 22 de agosto de 2025.

Agradecimentos

O presente trabalho foi realizado com apoio da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), Brasil. Processo nº 2023/04969-8. As opiniões, hipóteses e conclusões ou recomendações expressas neste material são de responsabilidade do(s) autor(es) e não necessariamente refletem a visão da FAPESP.

Resumo

Estruturas de grafos dinâmicos desempenham papel fundamental em aplicações que demandam processamento eficiente de grandes volumes de dados conectados, como redes sociais e sistemas de recomendação. Este trabalho apresenta uma adaptação do *framework* DGAP para ambientes NUMA, explorando o particionamento *round-robin* e a afinidade explícita de *threads* para otimizar o processamento de grafos dinâmicos em memória persistente. Os experimentos, conduzidos com dois conjuntos de dados reais do SNAP (Orkut e LiveJournal), avaliam o impacto das características topológicas dos grafos nas otimizações propostas e demonstram ganhos expressivos, com *speedup* de até 2,3$\times$ em algoritmos como *Connected Components*, além de evidenciarem limitações em grafos de baixa densidade e em algoritmos sensíveis à latência (como BFS), indicando a necessidade de estratégias adaptativas de balanceamento em ambientes NUMA.

Palavras-chave: grafos dinâmicos; memória persistente; NUMA; afinidade de *threads*; *speedup*.

Abstract

Dynamic graph structures play a key role in applications that require efficient processing of large volumes of connected data, such as social networks and recommender systems. This dissertation presents an adaptation of the DGAP framework to NUMA environments, exploring round-robin partitioning and explicit thread affinity to optimize the processing of dynamic graphs on persistent memory. Experiments conducted with two real-world datasets from SNAP (Orkut and LiveJournal) assess the impact of graph topological characteristics on the proposed optimizations and show expressive gains, with speedups of up to 2.3× in algorithms such as *Connected Components*, while also revealing limitations on low-density graphs and latency-sensitive algorithms (e.g., BFS), indicating the need for adaptive load-balancing strategies in NUMA environments.

Keywords: dynamic graphs; persistent memory; NUMA; thread affinity; speedup.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Computadores convencionalmente são divididos em armazenamento primário (memória DRAM) e secundário (HDs e SSDs). O armazenamento primário permite ao computador acessar rapidamente os dados, porém com pouco espaço de armazenamento e de forma volátil, ou seja, quando o computador é desligado, seja por falta de energia, erros, ou por comando do usuário, todos os dados armazenados nele são perdidos. Já no armazenamento secundário, é possível a retenção de dados mesmo após o desligamento do sistema mas, ao contrário da DRAM, esses dispositivos apresentam largura de banda pequena e tempos de resposta elevados quando comparados a ela, tornando-os inadequados para operações que exigem rapidez, sendo utilizados apenas para armazenamento de dados (HENNESSY; PATTERSON, 2017).

Os recentes avanços na indústria de semicondutores viabilizaram o desenvolvimento de tecnologias inovadoras para armazenamento persistente (BALDASSIN et al., 2021a). Essas tecnologias unem as características da DRAM e dos HDs e SSDs, como altas taxas de transferência de dados e granularidade de byte, juntamente com a grande capacidade de armazenamento e persistência dos dados. Isso possibilita a realização de operações na Memória Persistente (*Persistent Memory*, PM) de forma rápida e persistente, usando granularidade de byte ao invés de transferências em blocos.

Ao empregar esta nova tecnologia é necessário considerar vários aspectos presentes no sistema, os quais podem impactar significativamente o desempenho da aplicação. Um desses aspectos é o Acesso à Memória Não Uniforme (NUMA - *Non-Uniform Memory Access*) (DASHTI et al., 2013), (MEMARZIA; RAY; BHAVSAR, 2020). O NUMA possibilita que a memória, mesmo quando conectada ao barramento de um processador distinto, seja acessada pela CPU responsável pela execução do programa. Esta característica viabiliza a comunicação e a troca de dados entre todas as CPUs presentes em um servidor.

O NUMA oferece a vantagem de ampliar a capacidade de memória acessível por uma única CPU. No entanto, essa vantagem vem acompanhada de um aumento na latência ao acessar a memória vinculada a outro processador. Este fenômeno é bem conhecido quando se trabalha com processos na DRAM, mas também é observado com a Memória Persistente (LIU; CHEN; CHEN, 2023), (CHEN; QIU et al., 2022), (JAMIL et al., 2023), (JIA; JIANG;

XIONG, 2022).

Em virtude disso, torna-se imprescindível a implementação de estratégias que minimizem o impacto da latência no desempenho. Apesar de haver vários trabalhos no âmbito de memória volátil (DASHTI et al., 2013), técnicas que tiram proveito das especificidades da PM ainda não aparecem em muita quantidade na literatura.

As estruturas de dados (ED) são amplamente utilizadas para representar dados, servindo como base para muitos outros campos da ciência da computação (DROZDEK, 2012). Uma estrutura que está em ascensão quanto ao seu uso são os grafos, que estão sendo amplamente utilizados não só em bancos de dados de redes sociais, mas também em inteligências artificiais, aumentando significativamente sua aplicação (XIA et al., 2021). Assim, torna-se necessária a adaptação dessas estruturas para a memória persistente, buscando melhorar o desempenho das aplicações que as utilizam.

Neste contexto, surge o desafio de otimizar o processamento de grafos dinâmicos em Memória Persistente em ambientes NUMA. Embora estruturas como o DGAP (*Dynamic Graph Analysis on Persistent Memory*) (ISLAM; DAI, 2023) representem avanços para PMEM, sua implementação original não leva em conta os efeitos adversos típicos de arquiteturas multi-soquete, como a penalidade dos acessos remotos à memória. Isso resulta em gargalos significativos de desempenho, especialmente em servidores de grande porte, nos quais a afinidade entre dados e threads é determinante para a eficiência global.

Experimentos preliminares utilizando o algoritmo *PageRank* (PR), em diferentes cenários de alocação de threads, evidenciaram de forma clara o impacto negativo do efeito NUMA: observou-se uma perda de desempenho de até 17% quando todos os acessos ocorriam remotamente, em comparação ao cenário ideal com processamento local. Estes resultados reforçam a importância de técnicas que alinhem particionamento de dados e afinidade de threads para mitigar o impacto do NUMA em PMEM. A Figura 1.1 ilustra este fenômeno.

Diante desse cenário, este trabalho propõe e avalia extensões ao DGAP, com o objetivo de incorporar otimizações que permitam melhor particionamento do grafo entre os nós NUMA e associem o processamento de cada partição às *threads* locais correspondentes. Espera-se, por meio desta abordagem, não apenas mitigar o impacto do NUMA em *workloads* de análise de grafos, mas também oferecer um novo referencial de desempenho para aplicações em Memória Persistente.

Como principais contribuições, este estudo realiza uma análise quantitativa do impacto do NUMA sobre o DGAP, desenvolve e disponibiliza uma versão otimizada e avalia comparativamente diferentes estratégias de particionamento e afinidade, em conjunto com algoritmos clássicos de processamento de grafos e conjuntos de dados reais de larga escala. Por meio desse esforço, visa-se contribuir para a compreensão e o avanço de técnicas cientes da topologia de hardware no contexto de PM, promovendo maior eficiência no processamento de grafos em ambientes modernos.

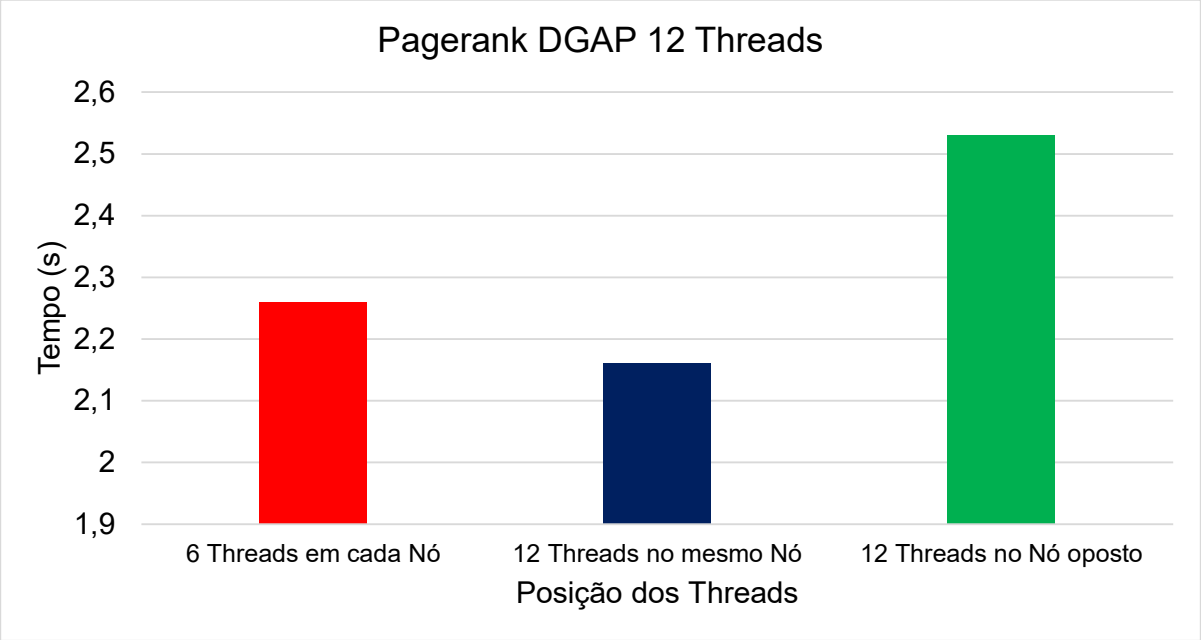

Figura 1.1: Comparação do efeito NUMA sobre o desempenho do DGAP.

1.1 Objetivos

O principal objetivo deste trabalho é adaptar o *framework* DGAP para ambientes com múltiplos nós NUMA, de modo a adaptá-lo para esses ambientes e, assim, otimizar o processamento de grafos dinâmicos em memória persistente. O foco reside na área de *graph analytics*, isto é, nas operações analíticas de leitura e consulta sobre grandes grafos. Dentre os objetivos específicos, destacam-se:

- Analisar o impacto das limitações de localidade e da comunicação entre nós em arquiteturas NUMA no desempenho do DGAP;
- Implementar estratégias de particionamento dos dados do grafo que maximizem o acesso local à memória;
- Avaliar, por meio de experimentos com conjuntos de dados reais de larga escala, os ganhos de desempenho obtidos com a adaptação proposta em relação ao *framework* original e a outras soluções do estado da arte;
- Discutir limitações e cenários onde a abordagem otimizada pode não trazer benefícios, indicando oportunidades para trabalhos futuros.

1.2 Contribuições

As principais contribuições deste trabalho são:

- O desenvolvimento de uma versão NUMA do DGAP, baseada em estratégias de particionamento Round-Robin e afinidade de *threads*, adaptada para ambientes com memória persistente;
- A avaliação sistemática do impacto dessas adaptações em diferentes algoritmos de análise de grafos (como *PageRank*, BFS, *Connected Components* e *Betweenness Centrality*), utilizando *datasets* reais de grande escala;
- A demonstração de ganhos de desempenho expressivos, atingindo aceleração de até 2,3x em comparação ao DGAP original em determinados cenários, evidenciando a importância de otimizações específicas para ambientes NUMA em aplicações de *graph analytics*;
- A disponibilização pública do código-fonte e do ambiente experimental, contribuindo para a reprodutibilidade e o avanço da pesquisa na área.

1.3 Organização do texto

Para facilitar a compreensão do desenvolvimento deste trabalho, a organização do texto é apresentada a seguir. O Capítulo 2 apresenta a fundamentação teórica sobre processamento de grafos e memória persistente em ambientes NUMA, servindo de base para o desenvolvimento desta pesquisa. O Capítulo 3 discute os trabalhos relacionados na área, contextualizando o cenário acadêmico. O Capítulo 4 detalha as adaptações propostas no DGAP para ambientes NUMA. No Capítulo 5, são descritos os experimentos realizados e a análise dos resultados obtidos. Por fim, o Capítulo 6 traz as conclusões do trabalho e sugestões para pesquisas futuras.

Capítulo 2

Fundamentos Básicos

Esta seção tem como objetivo elucidar os conceitos essenciais para o desenvolvimento desta pesquisa. Inicialmente, será apresentada a hierarquia de memória existente nos computadores (Seção 2.1), detalhando os dispositivos responsáveis por armazenar dados e o papel da memória persistente nesse contexto. Em seguida, a Seção 2.3 explicará o papel dos grafos e como a PM impacta essa estrutura. Por fim, será apresentado na 2.2 o conceito da tecnologia NUMA e seu impacto no acesso à memória.

2.1 Hierarquia de Memória

A memória em sistemas computacionais é tipicamente categorizada em dois tipos principais: armazenamento primário e secundário. O armazenamento primário, embora seja caracterizado por sua baixa latência, é volátil, o que significa que os dados armazenados são perdidos quando a energia é desligada. Por outro lado, o armazenamento secundário, apesar de apresentar uma latência maior, tem a vantagem de reter os dados mesmo após o sistema ser desligado. Esta distinção é fundamental para a arquitetura e o funcionamento eficiente dos sistemas computacionais modernos.

A Figura 2.1 ilustra a correlação entre a capacidade de armazenamento e a latência. Observa-se que existe uma relação direta entre essas duas variáveis: à medida que a capacidade de armazenamento aumenta, a latência também aumenta. Esta tendência pode ser atribuída ao fato de que as memórias com baixa latência tendem a ser mais caras por unidade de espaço, resultando em um custo mais elevado para maiores capacidades de armazenamento.

Como a memória rápida é cara e tem capacidade limitada, uma hierarquia de memória é organizada em vários níveis – cada um menor, mais rápido e mais caro por byte do que o próximo nível inferior, que está mais distante do processador. O objetivo é fornecer um sistema de memória com custo por byte quase tão baixo quanto o nível de memória mais barato e velocidade quase tão rápida quanto o nível mais rápido (HENNESSY; PATTERSON, 2017).

A Memória Persistente visa preencher a lacuna entre a DRAM, que é um armaze-

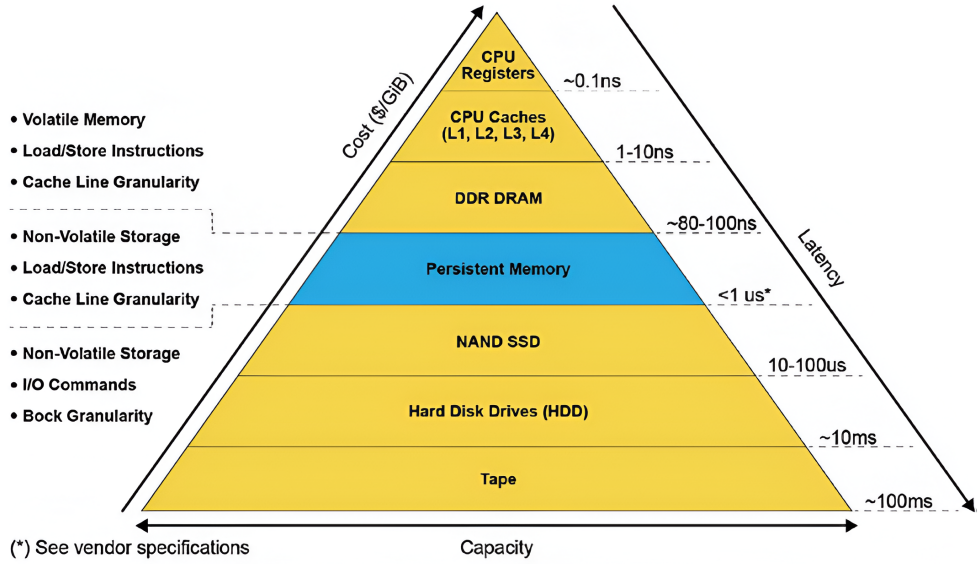

Figura 2.1: Pirâmide de Memórias relação Capacidade/Latência. Fonte: (SCARGALL, 2020)

namento primário e o SSD, armazenamento secundário. Ela combina as vantagens de ambos, oferecendo baixa latência e persistência de dados. A Memória Persistente tem múltiplas aplicações, podendo ser utilizada em conjunto com a DRAM ou até mesmo substituí-la em determinadas situações. Ela combina as vantagens da DRAM e do SSD, oferecendo baixa latência e persistência de dados. Uma explicação mais detalhada sobre este tópico pode ser encontrada na próxima seção.

2.1.1 Memória Persistente

A memória persistente é uma tecnologia que visa a velocidade, latência e endereçamento por byte (como memórias DRAM), mas com a possibilidade de manter os dados após o desligamento da máquina, característica de armazenamentos não voláteis, como SSDs e HDs (BALDASSIN et al., 2021b). Essa característica traz benefícios, como por exemplo, o aumento na velocidade de acesso a dados persistentes, já que esses dispositivos estão conectados diretamente no barramento do processador, portanto mais rápido que HDs e SSDs (WEILAND et al., 2019). Por outro lado, como uma falha pode acontecer a qualquer momento, é preciso mais atenção na escrita de código que acessa esses dispositivos: se a energia é interrompida no momento da atualização de uma estrutura de dados, na reinicialização inconsistências podem estar presentes. Considere, por exemplo, a operação de transferência bancária, onde o valor é debitado de um cliente e creditado a outro. Se há uma falha depois do débito, mas antes do crédito, o sistema ficará em um estado inconsistente.

Por ser uma memória com aspectos distintos das habituais, é necessário acessá-la de forma diferente.

Considere, por exemplo, uma lista encadeada que contém os valores 1, 3, 7 e 9, conforme ilustrado na Figura 2.2. Suponha que desejamos inserir o valor 5 nessa lista. Du-

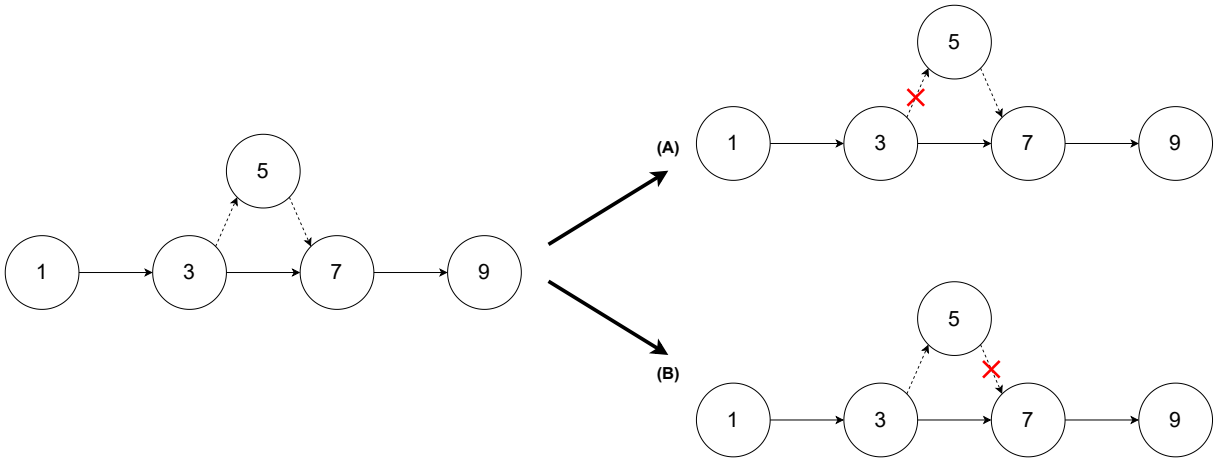

Figura 2.2: Caso de falha em inserção.

rante esse processo de inserção, se alguma falha ocorresse (por exemplo, queda de energia), poderíamos enfrentar duas situações problemáticas. A primeira situação (A) é a possibilidade do valor ser alocado na memória, mas não ser efetivamente inserido na lista. Isso resultaria em um vazamento de memória, onde há um desperdício de espaço na memória. A segunda situação (B) ocorre quando o nó contendo o valor 3 é atualizado para apontar para o novo nó 5, mas o ponteiro do novo nó 5 não é ajustado para apontar para o nó com o valor 7. Isso resultaria na perda de todos os dados subsequentes na lista, uma vez que o encadeamento dos nós é interrompido. Essas situações destacam a importância de garantir a integridade dos dados durante as operações. As abordagens atuais utilizam o conceito de *transação*, originado dos bancos de dados, para viabilizar a alteração das estruturas de dados. Atualmente, existem pelo menos duas bibliotecas que fornecem esse tipo de abstração. A primeira, conhecida como Atlas (CHAKRABARTI; BOEHM; BHANDARI, 2014), foi desenvolvida pela HP, mas hoje não é mais atualizada com frequência. A biblioteca mais conhecida é a PMDK (*Persistent Memory Development Kit*), da Intel (SCARGALL, 2020). Uma transação garante que todas as alterações realizadas dentro dela sejam ou efetivadas (*committed*) e consideradas persistentes, ou então abortadas (*aborted*) e revertidas (*rollback*); nesse caso, é como se nada tivesse acontecido. Dessa forma, se ocorrer uma queda de energia ou falha do sistema enquanto uma transação está sendo executada, todas as alterações realizadas até aquele ponto serão revertidas.

A Figura 2.3 mostra as camadas típicas fornecidas pela PMDK. A primeira camada, abstraída na figura como um sistema de arquivos, é fornecida pelo sistema operacional (SO). Essa interface tem sido padronizada pela SNIA (*Storage Networking Industry Association*) e é conhecida por DAX (*Direct Access*) (PAPPAS, 2018). DAX basicamente garante que o acesso ao PM possa ser feito diretamente na granularidade de bytes. Sistemas como Windows e Linux já fornecem esse suporte atualmente. A segunda camada, `libpmem`, fornece abstrações para as primitivas disponibilizadas pelo SO e hardware, como por exemplo instruções de descarga (*flush*) de cache. As funcionalidades da terceira camada permitem a criação de transações e alocação de memória persistente. Já as camadas 4 e 5 não fazem parte estritamente da PMDK, embora muitas vezes sejam apresentadas de forma conjunta porque são úteis no desenvolvimento

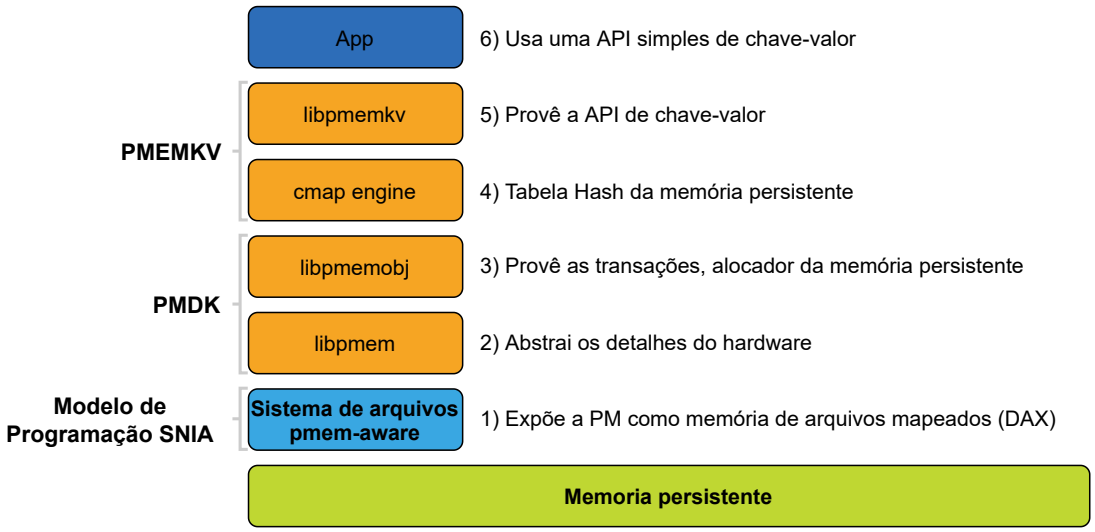

Figura 2.3: Camadas de software presentes na PMDK. Fonte: (SCARGALL, 2020)

de aplicações que empreguem armazenamento de estruturas do tipo chave-valor. Finalmente, a aplicação (sexto nível) é desenvolvida a partir de uma ou mais camadas fornecidas pela PMDK.

Com o progresso das memórias persistentes que possuem endereçamento a *byte*, torna-se imprescindível adaptar as estruturas de dados para essas memórias, como os grafos, amplamente utilizados em diversas áreas, incluindo bancos de dados, para que possam ser utilizados de maneira eficaz. Diferentemente das estruturas empregadas na memória DRAM, que são eliminadas em caso de falha na máquina, as estruturas adaptadas para as memórias persistentes devem ser modificadas para serem resistentes a falhas (CHEN; HU et al., 2024).

Outro aspecto crítico ao utilizar memória persistente refere-se ao custo e à durabilidade das operações de escrita. Diferentemente da DRAM, a PM apresenta latências significativamente mais elevadas para gravação e está sujeita ao desgaste físico de suas células, o que limita sua vida útil. Por esse motivo, trabalhos focados na PM, como os *frameworks* DGAP (ISLAM; DAI, 2023) e XPGraph (WANG et al., 2022), enfatizam a necessidade de reduzir ao máximo o número de escritas diretas nesse tipo de memória. Para isso, empregam técnicas como *logs*, *buffers* intermediários e agrupamento de operações em lote, buscando minimizar a frequência e o volume das gravações realizadas. Essas estratégias são fundamentais não apenas para otimizar o desempenho, mas também para assegurar a escalabilidade e a confiabilidade do sistema ao longo do tempo.

2.2 NUMA

A arquitetura de memória NUMA (*Non-Uniform Memory Access*) é um tipo de organização em que a memória do sistema é dividida em vários módulos ou nós independentes (DASHTI et al., 2013). Cada um desses nós pode ser acessado por múltiplos processadores ou núcleos. Nos

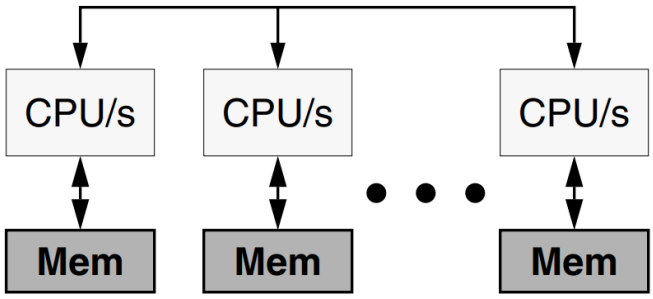

Figura 2.4: Exemplo de uma arquitetura NUMA, retirado de (JACOB; NG; WANG, 2008).

sistemas NUMA, os módulos de memória são conectados aos processadores ou núcleos através de uma interconexão de alta velocidade. Esta interconexão facilita uma comunicação rápida entre os processadores e módulos de memória, mas também introduz algumas sobrecargas de desempenho devido à necessidade de transferência de dados entre nós (CHEN; QIU et al., 2022). Em particular, uma solicitação para qualquer memória não local (ou seja, qualquer memória não diretamente conectada à CPU que faz a solicitação) provavelmente apresentará uma latência mais alta do que uma solicitação para uma memória local (JACOB; NG; WANG, 2008).

Na arquitetura NUMA, o acesso a dados entre soquetes sofre degradação do desempenho devido ao acesso lento à memória remota e à largura de banda limitada da interconexão da CPU (JIA; JIANG; XIONG, 2022), fenômeno conhecido como *efeito NUMA*. O efeito NUMA refere-se às diferenças nas latências de memória entre a memória local e remota de um nó da CPU. Essas latências irregulares de memória são causadas principalmente pela comunicação entre nós da CPU, sobrecarga do protocolo de coerência de cache e acessos remotos à memória (JAMIL et al., 2023). Quando se trata da PM, os efeitos do NUMA são ainda mais visíveis, pois sua latência é maior do que a da DRAM (LI et al., 2022). Devido a isso, existem vários métodos estabelecidos que visam minimizar a queda de desempenho causada por essa latência mais alta.

A Figura 2.4 apresenta um exemplo de conexões NUMA, no qual há vários CPUs em uma máquina, cada um com sua respectiva memória, interconectados entre si.

1. **Alocadores de Memória** – Os alocadores de memória dinâmica são ferramentas essenciais para rastrear e gerenciar a memória dinâmica ao longo da vida útil de uma aplicação. Frequentemente, o impacto no desempenho dos alocadores de memória é negligenciado, priorizando-se a exploração de maneiras de otimizar os algoritmos da aplicação. No entanto, estudos demonstraram que o desempenho pode ser significativamente aprimorado ao se utilizar alocadores alternativos aos convencionais, como o malloc (MEMARZIA; RAY; BHAVSAR, 2020). Portanto, é crucial avaliar qual alocador proporcionará maior estabilidade ao escalar o problema. Alocadores como `tbbmalloc` e `jemalloc` têm demonstrado escalabilidade e eficiência notáveis, resultando em reduções substanciais

no tempo de execução em diversas cargas de trabalho e máquinas (MEMARZIA; RAY; BHAVSAR, 2020).

2. **Distribuição na Memória** – As páginas de memória nem sempre são acessadas pelas mesmas *threads* que as alocaram. As políticas de posicionamento de memória são empregadas para controlar a localização das páginas de memória. Diferentes estratégias de posicionamento de memória, como *First Touch* e *Interleaved*, podem ser utilizadas para otimizar os padrões de acesso à memória em sistemas NUMA (MEMARZIA; RAY; BHAVSAR, 2020). O balanceamento de carga, que é o método de distribuição equitativa do tráfego de rede em um conjunto de recursos que suporta um aplicativo, é fundamental em um sistema NUMA. Através de funções como `numa_alloc_onnode`, permitem ao programador distribuir a alocação entre os nós de forma balanceada.

3. **Configuração do Sistema Operacional** – O ajuste de configurações do sistema operacional, como o AutoNUMA (THE LINUX KERNEL DEVELOPERS, 2023b) e as Transparent Hugepages (THE LINUX KERNEL DEVELOPERS, 2023c), pode aprimorar o desempenho de cargas de trabalho de análise de dados. De fato, a otimização dessas configurações pode levar a reduções significativas na latência de consultas em sistemas de banco de dados como `MonetDB` e `PostgreSQL`. Um estudo de caso com a carga de trabalho TPC-H no `MonetDB`, por exemplo, demonstrou que a latência pode ser reduzida em até 20% ao substituir o alocador de memória e em até 43% por meio de ajustes no sistema operacional (MEMARZIA; RAY; BHAVSAR, 2020). Contudo, tais otimizações perdem a eficácia quando se utiliza memória persistente no modo *App Direct* com formato DAX (THE LINUX KERNEL DEVELOPERS, 2023a), pois o acesso direto a esse tipo de memória contorna o cache de página do sistema operacional.

4. **Prevenção do Acesso a Nós Distintos** – As estratégias de posicionamento e agendamento de *threads* são um fator determinante para o desempenho em arquiteturas NUMA. Nesses sistemas, a definição de uma política de alocação eficiente é um passo essencial para otimizar a performance, uma vez que reduz, ou até mesmo evita, o acesso a nós de memória remotos. Essa prática previne a degradação de desempenho inerente à maior latência da interconexão entre os nós (MEMARZIA; RAY; BHAVSAR, 2020).

2.3 Grafos

As estruturas de dados são fundamentais para os sistemas computacionais, pois possibilitam o armazenamento e a manipulação eficaz de informações em diversos níveis da hierarquia de memória em várias aplicações. Com o surgimento da memória persistente, torna-se essencial adaptar ou desenvolver estruturas de dados capazes de explorar não apenas o desempenho, mas também a durabilidade oferecida por esse novo paradigma de armazenamento.

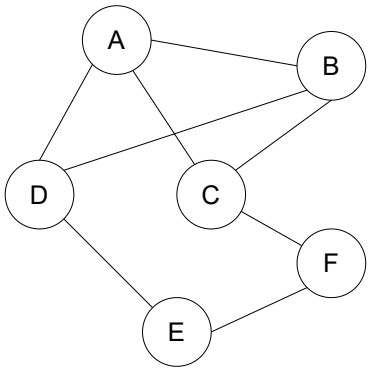

Figura 2.5: Exemplo de grafo.

Dentre as diversas estruturas, os grafos ocupam posição de destaque devido à sua capacidade de modelar relações complexas entre entidades. Formalmente, um *grafo* é definido como um par ordenado $G = (V, E)$, onde V é um conjunto finito de vértices (ou nós) e $E \subseteq V \times V$ é um conjunto de arestas, que representam conexões entre pares de vértices (WEST, 2001). Os grafos podem ser classificados, de acordo com a natureza de suas arestas, em *direcionados* e *não direcionados*. Em um grafo direcionado, cada aresta possui um sentido definido, partindo de um vértice de origem para um de destino, isto é, a aresta é um par ordenado (u, v). Já em um grafo não direcionado, as arestas não possuem orientação, representando apenas a existência de uma ligação entre dois vértices e podem ser descritas como subconjuntos não ordenados $\{u, v\}$.

A distinção entre grafos direcionados e não direcionados é relevante pois influencia diretamente tanto na escolha dos algoritmos de processamento quanto na estrutura dos dados empregada. Diversas aplicações reais, como redes de comunicação, mapas rodoviários, bancos de dados e, principalmente, redes sociais, podem ser representadas por grafos, sendo o tipo de grafo escolhido dependente da natureza das relações a serem modeladas.

O estudo dos grafos, desde suas origens no século XVIII com Euler, tornou-se uma das áreas mais sofisticadas da matemática e ciência da computação (DROZDEK, 2012). Seu uso disseminou-se para além da teoria, alcançando aplicações práticas em análise de redes sociais, rotas de transporte, sistemas de recomendação, bancos de dados e inteligência artificial. Grafos destacam-se ainda por sua habilidade em representar eficientemente relações complexas e por sua capacidade de manipular grandes volumes de dados, mantendo desempenho e flexibilidade. Por esses motivos, sua adaptação para Memória Persistente tem se tornado tema recorrente de pesquisa (ISLAM; DAI, 2023), (WANG et al., 2022).

A Figura 2.5 ilustra um exemplo simples de grafo e suas conexões, demonstrando a estrutura fundamental dessa representação.

2.3.1 Representações de Grafos

A escolha da representação de um grafo é guiada por um compromisso entre diversos fatores, como o consumo de memória, a eficiência em operações de busca e a flexibilidade para adicionar ou remover nós. Essa decisão reflete o objetivo principal da aplicação: priorizar o

Tabela 2.1: Exemplo de representação de um grafo em *Edge List*.

Edge List
(A, B)
(A, C)
(A, D)
(B, C)
(B, D)
(C, F)
(D, E)
(E, F)

desempenho computacional ou a otimização do uso de espaço. Existem diversas maneiras de representar grafos em memória, sendo as mais comuns as matrizes de adjacência e as listas de adjacência. Cada representação possui características distintas, que acarretam vantagens e desvantagens (WHEATMAN; XU, 2018).

Lista de Arestas

A Lista de Arestas (*Edge List*) é a forma mais simples de representar grafos na memória. Consiste em uma lista sequencial de arestas, sendo eficiente para adição de arestas, pois novas arestas são inseridas ao final da lista, evitando a necessidade de mover dados ou reorganizá-los. No entanto, é uma estrutura lenta para acessos a vértices, pois requer a varredura de toda a lista de arestas (ISLAM; DAI, 2023).

A Tabela 2.1 apresenta a representação, em *Edge List*, do grafo ilustrado na Figura 2.5. Neste exemplo, a lista está ordenada, isto é, em ordem crescente de acordo com o ID do vértice inicial.

Matrizes de Adjacência

Para um grafo com N nós, utiliza-se uma matriz de dimensão N x N, na qual a posição (x, y) corresponde à aresta do nó `x` para o nó `y`. O valor 1 nesta posição simboliza a existência da aresta, enquanto o valor 0 indica sua inexistência.

Esse método, contudo, pode levar a um alto consumo de memória, visto que mapeia todas as conexões possíveis, tanto as existentes quanto as inexistentes. Portanto, sua aplicação é mais eficiente em grafos muito densos, nos quais a maioria das arestas de fato existe, minimizando o espaço não utilizado.

As duas principais desvantagens dessa representação são o seu elevado consumo de memória e a necessidade de reconstruir toda a estrutura a cada adição de um novo nó. Contudo, esses pontos negativos são contrabalançados pela alta eficiência em operações de busca (WHEATMAN; XU, 2018).

O grafo da Figura 2.5 é utilizado para exemplificar a representação por matriz de

Tabela 2.2: Representação do Grafo em Matriz de Adjacência

	A	**B**	**C**	**D**	**E**	**F**
A	0	1	1	1	0	0
B	1	0	1	1	0	0
C	1	1	0	0	0	1
D	1	1	0	0	1	0
E	0	0	0	1	0	1
F	0	0	1	0	1	0

Tabela 2.3: Exemplo de representação de um grafo em lista de adjacência.

Nó	**Lista de Adjacentes**
A	$\rightarrow$ [B $\rightarrow$ C $\rightarrow$ D]
B	$\rightarrow$ [A $\rightarrow$ C $\rightarrow$ D]
C	$\rightarrow$ [A $\rightarrow$ B $\rightarrow$ F]
D	$\rightarrow$ [A $\rightarrow$ B $\rightarrow$ E]
E	$\rightarrow$ [D $\rightarrow$ F]
F	$\rightarrow$ [C $\rightarrow$ E]

adjacência, cuja estrutura resultante é exibida na Tabela 2.2. Nela, a conexão entre os nós A e B, por exemplo, é representada na célula que corresponde à interseção da linha A com a coluna B. Como essa aresta de fato existe, o valor nesta posição é 1.

Lista de Adjacência

A lista de adjacência é outra representação comumente utilizada para grafos. Nesse formato, cada nó é associado a uma lista — geralmente, uma lista encadeada — que contém todos os seus vizinhos diretos. A principal vantagem dessa estrutura está na elevada eficiência para consultas de vizinhança. Em contrapartida, apresenta algumas desvantagens tanto na busca quanto na inserção: a busca por uma aresta pode exigir percorrer uma lista extensa, o que demanda tempo, e, no caso da inserção, a desvantagem reside no custo de atualização, pois a adição de novas arestas pode exigir realocação de memória, tornando o processo computacionalmente mais custoso (WHEATMAN; XU, 2018; WANG et al., 2022).

A Figura 2.5 é novamente utilizada como exemplo, desta vez para ilustrar a representação em lista de adjacência. A Tabela 2.3 exibe essa estrutura, na qual cada vértice está associado a uma lista encadeada de seus vizinhos, de modo que cada elo da lista simboliza uma aresta.

Compressed Sparse Row

O formato *Compressed Sparse Row* (CSR) é uma técnica eficiente para armazenar matrizes esparsas e, por extensão, grafos (KELLY, 2020). Ele compacta a representação do grafo em

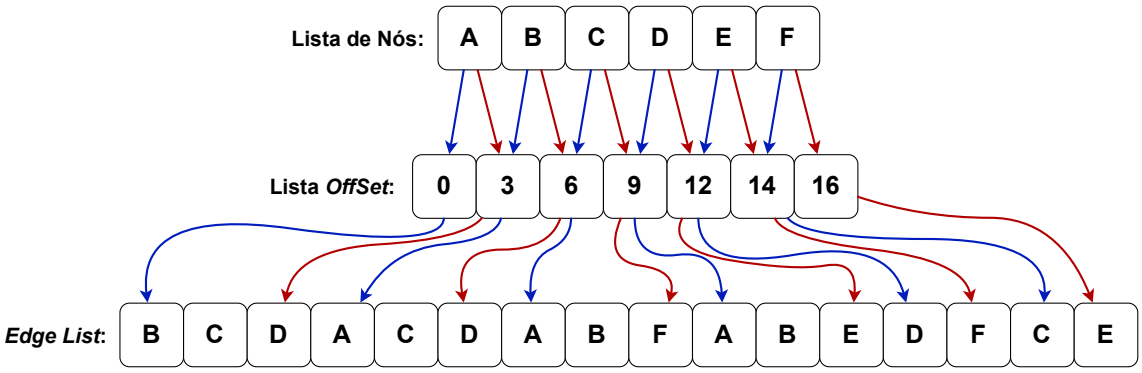

Figura 2.6: Exemplo CSR.

dois vetores principais, que descrevem as arestas na memória:

1. **Lista de *Offsets*:** Um vetor que indica onde começa o bloco de arestas de cada nó de origem na lista de arestas. Para um grafo de N nós, este vetor possui N+1 elementos. O i-ésimo elemento aponta para o início das arestas do nó `i` e o (i+1)-ésimo elemento aponta para o seu fim. Ou seja, para o quinto vértice, por exemplo, o quinto elemento na Lista de *Offsets* indica a posição na lista de arestas onde começam as suas arestas e o sexto elemento indica a posição onde começam as arestas do sexto vértice, definindo assim o fim das arestas do quinto.

2. **Lista de Arestas:** Um vetor que armazena os identificadores dos nós de destino de todas as arestas do grafo. As arestas são listadas sequencialmente, agrupadas pelo nó de origem (primeiro todas as arestas do nó 0, depois do nó 1 e assim por diante).

A Figura 2.6 ilustra a representação em CSR do grafo da Figura 2.5. Nessa estrutura, a Lista *OffSet* indica o índice inicial da lista de vizinhos de cada nó, que estão armazenados no *Edge List*. O papel das setas é demonstrar essa relação: as azuis apontam para o início da lista de um nó, enquanto as vermelhas indicam o seu fim. Para exemplificar, considere o vértice B. Sua lista de vizinhos começa no índice 3 do *Edge List*, conforme indicado pela seta azul que parte do seu respectivo *offset*. O fim dessa lista é determinado pelo *offset* do vértice seguinte, C, que aponta para a posição 6. Portanto, a lista de vizinhos de B compreende os índices de 3 a 5 (ou seja, $6 - 1$).

A principal vantagem do CSR é sua extrema compacidade, pois armazena apenas as entradas não nulas da matriz de adjacência de um grafo. Com isso, praticamente elimina toda a sobrecarga de espaço, como os ponteiros "próximo" presentes em cada nó de uma lista de adjacência, ou o desperdício causado pelas entradas zero em uma matriz de adjacência. Persistir a representação CSR em arquivos mapeados em memória torna-se um processo simples e eficiente. Após a conversão inicial para o formato CSR, não são necessárias análises ou serializações adicionais: o arquivo já se encontra no formato compacto em memória, pronto para as análises subsequentes, que acessam os dados diretamente por meio de instruções de carregamento, após o mapeamento do arquivo em memória (KELLY, 2020).

Já o principal problema do CSR é a necessidade de reconstruir todo o grafo ao inserir um novo vértice, o que torna o processo de inserção muito lento. Ao inserir uma aresta, é preciso deslocar todos os valores subsequentes uma posição à frente e atualizar a Lista de *Offsets*. Dessa forma, a operação de inserção representa uma limitação do CSR (WHEATMAN; XU, 2018).

Capítulo 3

Trabalhos Relacionados

Esta seção apresenta e discute os principais trabalhos relacionados à mitigação do efeito NUMA sobre Memória Persistente, bem como à otimização de estruturas de dados e grafos nesse contexto. A seleção priorizou estudos que contribuem para a compreensão do problema e das soluções existentes, situando a presente pesquisa em um panorama mais amplo.

3.1 Desafios em Memória Persistente e NUMA

Diversos trabalhos têm como premissa a adaptação de estruturas de dados para Memória Persistente. Zhang et al. (ZHANG et al., 2018) adaptam uma *hash table* visando reduzir o número de escritas na PM, enquanto Lavinsky et al. (LAVINSKY; ZHANG, 2022) implementam uma estrutura de árvore para o mesmo fim. Contudo, a maioria dessas propostas não considera o impacto do NUMA, aspecto que pode alterar significativamente o desempenho dos sistemas.

Considerando esse cenário, outros autores voltam-se especificamente à avaliação e mitigação dos efeitos do NUMA em PM. Por exemplo, Zhu et al. (ZHU et al., 2021) e Chen et al. (CHEN; CHE et al., 2023) estudam como o acesso remoto a nós NUMA afeta o desempenho: em experimentos utilizando dois processadores Intel Xeon e seis módulos de 128GB de Intel Optane, foi evidenciado que o acesso remoto reduz substancialmente a taxa de gravação, porém com baixo impacto sobre leituras. Essa redução tende a ser mais acentuada com o aumento do número de *threads*.

Ainda nesse contexto, Chen et al. (CHEN; HU et al., 2024) fornecem uma análise detalhada sobre o desempenho e a otimização de tabelas *hash* em PM, investigando o impacto da largura de banda, instruções de CPU e configurações NUMA. O estudo aponta a necessidade de metodologias específicas para minimizar a degradação do desempenho associada ao NUMA.

A partir dessa constatação, surgem diferentes estratégias para lidar com esses desafios impostos pelo NUMA em ambientes com Memória Persistente. Um exemplo é apresentado por Salam et al. (SALAM et al., 2022), que sugerem uma abordagem focada na redução dos acessos remotos, ao permitir que cada *thread* opere de forma independente e com memória

alocada localmente. Essa proposta visa eliminar conflitos e dependências entre as *threads*, otimizando a eficiência do sistema.

Complementando essa tendência de minimizar acessos remotos, Han et al. (HAN et al., 2021) exploram os principais desafios na integração entre NUMA e PM. Os autores, ao identificarem gargalos de performance, propõem a solução *Dragonfly*, baseada na migração dinâmica de dados para o nó onde a tarefa está sendo executada, reduzindo o tempo de acesso e aumentando o desempenho.

O conceito de migração de dados, também tratado como balanceamento de carga, é recorrente na literatura, sendo uma alternativa bastante explorada para distribuir tarefas e operações entre os diferentes nós NUMA. Nessa direção, Michailidis et al. (MICHAILIDIS; SWANSON; ZHAO, 2022) introduzem um mecanismo de realocação automática, no qual o *kernel* periodicamente transfere páginas de memória para o nó que executa o processo, mitigando os impactos dos acessos remotos.

Além das abordagens de migração, outra vertente importante considera a limitação dos acessos cruzados entre processadores e nós, priorizando sempre que possível a execução sobre a memória local. Nesse sentido, Liu et al. (LIU; CHEN; CHEN, 2023) propõem a partição de tabelas com base na afinidade NUMA, assegurando que transações ocorram prioritariamente próximo aos dados. Seguindo uma ideia semelhante, Jia et al. (JIA; JIANG; XIONG, 2022) estruturam *IO thread pools* por soquete, de modo a manter o processamento o mais local possível e, assim, minimizar latências e gargalos provocados por acessos remotos à PM.

Ampliando ainda mais as alternativas para reduzir operações de acesso remoto, Zou et al. (ZOU et al., 2022) sugerem a manutenção de uma subtabela *hash* dedicada a cada nó NUMA. Dessa maneira, os dados são sempre armazenados e processados no nó correspondente, permitindo que todas as operações ocorram de forma local, sem a necessidade de sincronização ou comunicação entre diferentes nós.

3.2 *Frameworks* de Grafos

No contexto de grafos, destacam-se o GraphOne e o CSR (KUMAR; HUANG, 2020; FIRMLI et al., 2021), ambos desenvolvidos para manter grafos na DRAM. Esses trabalhos são relevantes porque propõem métodos inovadores para o armazenamento eficiente de grafos, conforme detalhado na Seção 2.3.1, sendo que o GraphOne utiliza a Lista de Adjacência (KUMAR; HUANG, 2020) e o CSR adota o próprio método *Compressed Sparse Row*.

O GraphOne tem como objetivo unir a rápida inserção à análise eficiente de dados na memória. Sistemas anteriores normalmente priorizavam uma dessas características, sem oferecer uma solução integrada. O GraphOne, além disso, permite a inserção dinâmica de arestas, possibilitando adicionar novas conexões após a criação do grafo, por meio da combinação das estruturas *Edge List* e Lista de Adjacência (KUMAR; HUANG, 2020). Para isso, introduz o conceito de *dual versioning* e a abstração *GraphView*, que facilita o acesso concorrente e

consistente ao grafo em diferentes níveis de atualização.

O CSR, por sua vez, visa compactar os índices de coluna das entradas não nulas em um *array* denso, proporcionando um armazenamento mais compacto e contínuo na memória do que a lista de adjacência, reduzindo *overheads* e acessos aleatórios (FIRMLI et al., 2021). Essa técnica é especialmente eficaz para grafos esparsos e estáticos, maximizando a performance de leitura e favorecendo operações em *memory-mapped files* (mmap), o que a torna adequada para PM. No entanto, sua principal limitação está na pouca flexibilidade para inserção dinâmica de arestas.

Com a popularização da Memória Persistente, surgiram *frameworks* para grafos adaptados a esse novo cenário, como o DGAP e o XPGraph (ISLAM; DAI, 2023; WANG et al., 2022), ambos baseados nessas estruturas clássicas. O XPGraph se diferencia por unir diversas estratégias: ao utilizar particionamento Round-Robin dos vértices e arestas e técnicas de balanceamento de carga, armazena os dados do grafo segmentados em múltiplos nós NUMA, combinando, assim, operações locais e balanceamento dinâmico. Diferentemente do XPGraph e do próprio GraphOne, o DGAP faz uso apenas da estrutura CSR adaptada e inovada para a PM, baseando-se no formato *Package Compressed Sparse Row* (PCSR), que reserva espaços entre as arestas de cada nó para facilitar inserções e remoções dinâmicas.

No DGAP, as arestas do grafo são armazenadas em um array de arestas (*edge array*), componente central da estrutura CSR. Nessa representação, todas as arestas são organizadas sequencialmente em memória, enquanto um vetor auxiliar indica, para cada vértice, o índice inicial de sua respectiva lista de vizinhos dentro do *array* principal. Para permitir inserções dinâmicas sem a necessidade de reconstruir toda a estrutura, o DGAP divide o *array* de arestas em seções, cada uma reservando espaço adicional para futuras inserções. Além disso, o DGAP traz inovações como a estrutura *per-section edge log*, que permite agrupar modificações em lotes antes de consolidá-las no array de arestas, reduzindo o número de alterações individuais e otimizando o desempenho (ISLAM; DAI, 2023). O funcionamento detalhado do DGAP será apresentado na Seção 4.2.

Assim, observa-se que as principais estratégias para mitigar os desafios impostos por NUMA e PM são o balanceamento de carga e a alocação local de *threads*, propostas já exploradas em DRAM, com adaptações relevantes para tecnologias emergentes. O balanceamento de carga envolve a migração de dados ou *threads* para evitar sobrecarga em um único nó, enquanto a alocação local busca otimizar o acesso à memória, aproveitando a topologia do hardware. Abordagens como Co-localização de Páginas e o *Thread clustering* complementam essas técnicas, ajustando a migração conforme o perfil de acesso e o custo das operações (DASHTI et al., 2013; LEPERS; QUEMA; FEDOROVA, 2015).

O XPGraph se destaca como o estudo mais completo na área de grafos para Memória Persistente, apresentando suporte robusto para ambientes NUMA e servindo, inclusive, como inspiração para o desenvolvimento deste trabalho. Entre as otimizações propostas pelo XPGraph para ambientes NUMA (WANG et al., 2022), destaca-se a divisão do grafo entre os

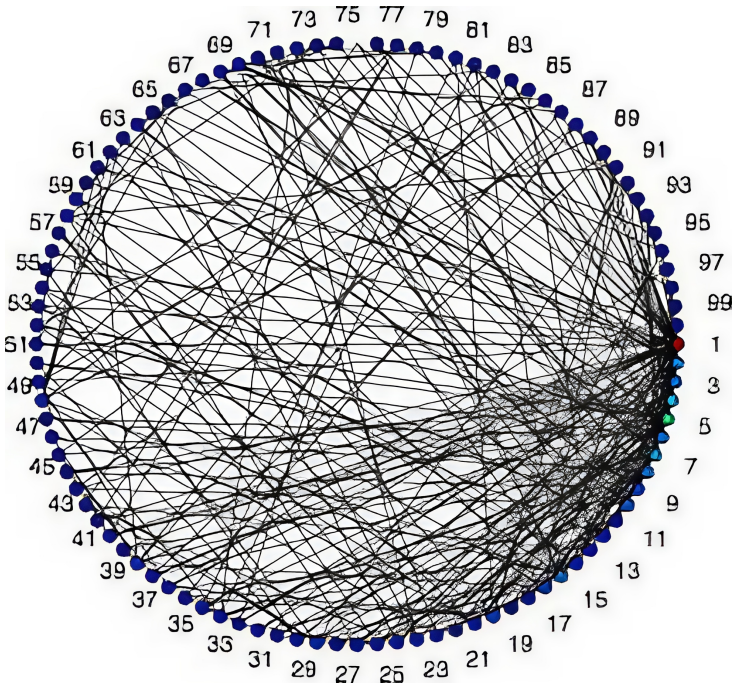

Figura 3.1: Ligações entre vértices mais antigos (KARYOTIS; KHOUZANI, 2016).

nós utilizando um esquema Round-Robin baseado em *hash*, no qual cada vértice V é alocado no nó N conforme a expressão $V\%2 = N$. Dessa forma, o vértice 0 é alocado no Nó 0, o vértice 1 no Nó 1, o vértice 2 retorna ao Nó 0 e assim sucessivamente.

É importante ressaltar, contudo, que somente a estratégia de divisão não garante desempenho otimizado. Para que os benefícios sejam efetivamente alcançados, é fundamental implementar o *bind* das *threads* aos respectivos nós, adotando uma lógica que evite acessos remotos entre os nós. Dessa maneira, cada *thread* é fixada ao nó correspondente aos dados que manipula, minimizando a latência de acesso à memória persistente e otimizando o desempenho do sistema.

Esse método possibilita um balanceamento mais eficiente das arestas entre os nós, quando comparado a abordagens mais simples, como a divisão "ao meio"do grafo, distribuindo de forma equilibrada a quantidade de arestas entre os dois nós NUMA (KRAUSE et al., 2017). Tal eficiência decorre de dois fatores principais.

O primeiro fator é a chamada *power-law*, que caracteriza o fato de poucos vértices apresentarem graus elevados, isto é, conectarem-se a diversos outros vértices, enquanto a maioria possui poucas ligações (WANG et al., 2022). O XPGraph, ao avaliar quatro *datasets*, observou que vértices com até duas ligações representam 40% do total, enquanto aqueles com quatro a sete ligações correspondem a 20%, sendo poucos os que apresentam mais de 64 ligações.

O segundo fator é o *Preferential Attachment*, que indica que vértices mais antigos em um *dataset* têm maior probabilidade de receber novas conexões à medida que novos vértices são inseridos (KARYOTIS; KHOUZANI, 2016). A Figura 3.1 ilustra esse comportamento, já que, em conjuntos de dados reais, é comum cada vértice receber um identificador sequencial

que reflete sua ordem de inserção. Assim, o primeiro vértice adicionado tem ID 0, o segundo ID 1 e assim sucessivamente.

Dessa forma, uma simples divisão ao meio — isto é, em um grafo com N vértices, alocando os primeiros $N/2$ vértices em um nó e os demais no segundo — resultaria em um desequilíbrio significativo na distribuição das arestas, concentrando a maior parte delas na primeira metade do grafo.

Além disso, o XPGraph incorpora otimizações voltadas para a inserção dinâmica de vértices, como o *vertex-centric graph buffering*, que consiste em um *buffer* implementado na DRAM para cada vértice. Esse *buffer* acumula múltiplas inserções até atingir sua capacidade máxima e, então, realiza todas as inserções de uma única vez na memória persistente, reduzindo, assim, o número de operações de escrita. Considerando a distribuição *power-law*, o XPGraph atribui *buffers* de tamanhos distintos para cada vértice, evitando, desse modo, a alocação desnecessária de espaço na DRAM.

3.3 Comparação DGAP e XPGraph

Apesar dos avanços apresentados pelo XPGraph ao abordar a persistência e o desempenho em ambientes de Memória Persistente, o DGAP propõe uma arquitetura baseada em uma única estrutura centralizada, o CSR mutável, que elimina a necessidade de conversões e sincronizações entre múltiplas representações do grafo, como ocorre no XPGraph.

No XPGraph, mesmo com otimizações para memória persistente, há a duplicação de estruturas: o *framework* mantém tanto uma lista de arestas quanto uma lista de adjacências em sua arquitetura (WANG et al., 2022). Essa duplicidade existe porque o XPGraph utiliza a lista de arestas para garantir durabilidade e inserção eficiente dos dados, enquanto a lista de adjacências é empregada para análise rápida. Para que a estrutura de análise reflita os dados mais recentes, torna-se necessária a sincronização e migração periódica dos dados entre essas duas estruturas, processo conhecido como *conversão entre estruturas*.

Em outras palavras, a conversão entre estruturas refere-se à etapa na qual o conteúdo da lista de arestas (utilizada para gravação eficiente e durabilidade) é transferido e consolidado na lista de adjacências (otimizada para consultas analíticas). Esse mecanismo, embora traga vantagens ao separar responsabilidades, introduz *overheads* consideráveis, pois envolve movimentação de grandes volumes de dados entre as listas e requer etapas adicionais de conversão e sincronização, especialmente sob cargas de trabalho dinâmicas e em ambientes NUMA, onde a movimentação de dados entre nós pode penalizar ainda mais o desempenho (ISLAM; DAI, 2023).

O DGAP foi originalmente concebido sem suporte específico para arquiteturas NUMA, focando apenas na otimização do desempenho sobre Memória Persistente. Nesse contexto, a principal contribuição deste trabalho consiste justamente em tornar o DGAP ciente da topologia NUMA (NUMA-aware), aliando os benefícios comprovados de sua estrutura simplifi-

cada à eficiência no acesso à memória em servidores modernos. Essa proposta e as otimizações desenvolvidas para ambientes NUMA serão discutidas em detalhes ao longo deste documento.

A Tabela 3.1 apresenta um resumo comparativo das principais características dos *frameworks* discutidos.

Tabela 3.1: Comparação entre frameworks para processamento de grafos dinâmicos em PMEM

Framework	**Estrutura**	**NUMA**	**PMEM**	**Conversão entre Estruturas**
DGAP	CSR mutável	Não	Sim	Não
XPGraph	Edge Log + Lista Adj.	Sim	Sim	Sim
GraphOne	Edge Log + Lista Adj.	Não	Não	Sim
CSR tradicional	CSR	Não	Não	Não
Nossa proposta	CSR mutável (DGAP) + NUMA	**Sim**	Sim	Não

Capítulo 4

Otimização do DGAP para Arquiteturas NUMA

Neste capítulo, apresenta-se uma otimização do *framework* DGAP para ambientes NUMA. Primeiro, descrevem-se os principais componentes e estruturas de dados do DGAP, destacando as suas limitações em cenários de memória não uniforme. Em seguida, detalham-se as adaptações implementadas para maximizar a localidade de memória, evitando acessos remotos. Por fim, discutem-se os desafios enfrentados durante o desenvolvimento da adaptação.

4.1 Desafios e Contexto

A análise de grafos em larga escala sobre Memória Persistente (PM) e arquiteturas NUMA apresenta desafios fundamentais relacionados à escalabilidade, ao custo de acesso e à consistência dos dados (WANG et al., 2022). Enquanto a adoção de PM traz benefícios de capacidade e persistência, ela impõe limitações próprias, como a amplificação de leituras e escritas devido à granularidade de acesso (XPLine no caso da Optane DC) e uma sensibilidade muito maior a acessos remotos (*cross-NUMA*) do que na DRAM. Experimentos prévios mostraram que a simples migração de sistemas otimizados para DRAM, como o GraphOne, para PM acarreta quedas drásticas de desempenho devido ao aumento no custo das operações de escrita randômica e ao efeito dos acessos remotos entre nós (WANG et al., 2022). Por exemplo, em sistemas com PM, pequenas atualizações (como escrita de IDs de vértice de 4 *bytes*) podem induzir operações de leitura-modificação-escrita de 256 *bytes*, penalizando severamente o desempenho. Este fenômeno é conhecido como *write amplification*.

Além disso, *workloads* de *graph analytics* geralmente envolvem predominância de operações de leitura (busca/traversal), mas ainda assim precisam suportar inserções e remoções eficientes, especialmente em grafos dinâmicos. Por isso, é fundamental que o projeto das estruturas de dados busque mitigar o custo de acessos aleatórios, maximize a localidade e promova balanceamento de carga entre nós, tanto do ponto de vista de dados quanto de *threads*.

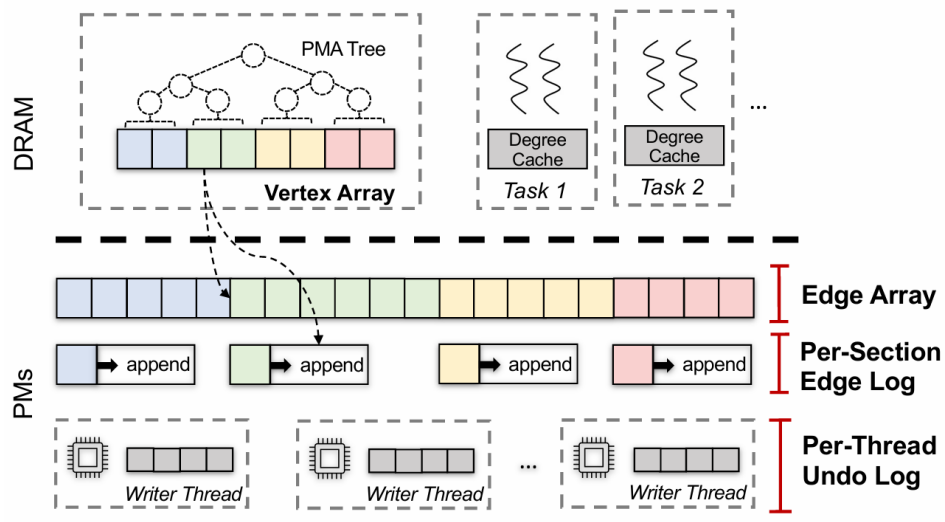

Figura 4.1: Estrutura do DGAP. Imagem retirada de (ISLAM; DAI, 2023).

O *framework* XPGraph surge como uma resposta a esses desafios, propondo três pilares: bufferização por vértice, gerenciamento hierárquico de *buffers* com ajuste dinâmico e acesso NUMA-friendly por meio de distribuição controlada dos dados e *threads* (WANG et al., 2022). Tais decisões arquiteturais influenciaram as adaptações propostas nesta dissertação, que buscam aliar simplicidade estrutural e desempenho em ambientes PM-NUMA.

4.2 DGAP

O DGAP foi concebido para proporcionar eficiência tanto na análise quanto na atualização de grafos dinâmicos empregando Memória Persistente. Seu projeto fundamenta-se na adoção de uma estrutura central baseada em CSR, complementada por mecanismos especializados que visam maximizar a performance e a consistência em ambientes persistentes.

A adoção do formato CSR (*Compressed Sparse Row*) se justifica por ser uma das formas mais compactas e eficientes de armazenar grafos esparsos para *workloads* centrados em leitura, pois elimina *overheads* de ponteiros e permite varredura sequencial dos vizinhos de cada vértice, maximizando o aproveitamento de *cache* e minimizando acessos aleatórios (KELLY, 2020). O CSR, porém, não é trivial de manipular para inserções e deleções dinâmicas; por isso, o DGAP complementa a estrutura básica com o uso do *Packed Memory Array* (PMA), que permite manter a compactação do CSR mesmo com atualizações dinâmicas.

A seguir, detalham-se os principais componentes estruturais do DGAP, cuja compreensão é imprescindível para o entendimento das adaptações e otimizações introduzidas na otimização. A Figura 4.1 ilustra a estrutura, destacando as seções na *Edge Array* por cores.

- ***Vertex Array*:** O DGAP armazena todos os vértices de maneira consecutiva em um *array* situado na DRAM. Cada elemento desse vetor, com tamanho de 16 bytes, contém informações essenciais: o grau atual do vértice, o índice de início no *edge array* e, quando

pertinente, um ponteiro para o *edge log* em caso de inserções pendentes. Esse componente permanece integralmente na DRAM a fim de evitar atualizações frequentes na PM, mitigando o impacto da elevada latência de escrita característica desse tipo de memória (ISLAM; DAI, 2023);

- ***Edge Array*:** As arestas do grafo são mantidas em um *array* implementado segundo o conceito de *Packed Memory Array*, residindo diretamente na Memória Persistente. Cada aresta armazena exclusivamente o identificador do vértice de destino, aproveitando o fato de que todas as arestas de uma mesma lista compartilham o vértice de origem. Adicionalmente, o DGAP insere, no início da lista de arestas de cada vértice, um elemento especial denominado *pivot*, um valor negativo representando o identificador do vértice de origem, que possibilita a reconstrução eficiente da *Vertex Array* após falhas, visto que esta estrutura reside em DRAM e não é persistida (ISLAM; DAI, 2023). Para monitorar a densidade de cada seção, o DGAP emprega uma árvore binária baseada em PMA, onde caso inserções ou remoções alterem a densidade de uma seção para fora do intervalo desejado, são iniciadas operações de rebalanceamento, com o intuito de redistribuir lacunas entre seções adjacentes (ISLAM; DAI, 2023);

- ***Per-Section Edge Log*:** Considerando as limitações de desempenho resultantes de operações de escrita em pequenas regiões na Memória Persistente, o DGAP adota um *log* de arestas por seção. Esse componente funciona como um *buffer* localizado na Memória Persistente para cada seção do *edge array*, armazenando temporariamente inserções de arestas para posterior realização em lote. Tal abordagem reduz consideravelmente a amplificação de escrita, promovendo melhor aproveitamento do *buffer* interno da memória (ISLAM; DAI, 2023);

- **Per-Thread Undo Log:** A fim de garantir a consistência transacional das operações de balanceamento e rebalanço, o DGAP implementa um *undo log* persistente por *thread*. Antes de operações potencialmente destrutivas, os dados afetados são previamente copiados para o *undo log* da respectiva *thread*, permitindo, em caso de falha, a recuperação de um estado consistente sem os custos típicos de transações tradicionais, como as oferecidas pela PMDK (ISLAM; DAI, 2023).

O projeto dessas estruturas visa equilibrar a necessidade de alta performance em operações dinâmicas de grafos, incluindo inserção, remoção, consultas e análise, com os requisitos de durabilidade e *crash-consistency* impostos pela Memória Persistente. Enquanto o *edge array* principal permanece na PM, metadados voláteis e de atualização frequente, como o vetor de vértices e mecanismos de concorrência, são mantidos na DRAM, em consonância com as recomendações mais recentes da literatura. A escolha pelo CSR confere maior eficiência espacial e melhor performance de acesso sequencial do que listas de adjacência ou matrizes de

adjacência, especialmente para grafos esparsos, sendo ideal para *workloads* de análise (KELLY, 2020).

4.3 Adicionando processamento NUMA no DGAP

Com o objetivo de explorar o paralelismo proporcionado por arquiteturas NUMA e otimizar a distribuição dos dados para um ambiente NUMA, foram conduzidas diversas adaptações e investigações no DGAP, com foco principal as operações de análise sobre o grafo (analytics), não abrangendo mecanismos de alteração dinâmica, como inserções e remoções de arestas ou vértices, que serão detalhadas nesta seção de maneira integrada.

Antes de adotar estratégias manuais para a distribuição de dados, avaliou-se a possibilidade de os mecanismos automáticos de gerenciamento NUMA, como o `AutoNUMA`, otimizarem a localidade de memória de maneira transparente. O `AutoNUMA` acompanha os acessos de cada processo e transfere as páginas de memória entre os nós NUMA de forma dinâmica, com o objetivo de aproximar os dados das CPUs que mais os utilizam (THE LINUX KERNEL DEVELOPERS, 2023b).

Contudo, essa estratégia não é eficaz quando a memória persistente é empregada no modo de acesso direto (DAX). O `AutoNUMA` foi desenvolvido principalmente para administrar páginas anônimas, como *heap* e pilha, que não estão diretamente relacionadas a arquivos. No modo DAX, a memória persistente é representada como um arquivo, com cada endereço virtual vinculado diretamente ao endereço físico no dispositivo PMEM. Se o sistema operacional migrar essas páginas entre nós NUMA, o mapeamento direto é quebrado, o que viola o princípio do acesso direto e afeta tanto a consistência quanto a persistência dos dados (THE LINUX KERNEL DEVELOPERS, 2023a). Outro problema é que o `AutoNUMA` não trabalha com a consistência dos dados em casos de falhas. Por exemplo, se houver uma falha de energia durante a migração de dados de um nodo NUMA persistente para outro, não haveria como se recuperar dessa falha quando o sistema fosse restaurado.

A ineficácia das ferramentas automáticas impôs, portanto, a necessidade de abordar o balanceamento de carga e a localidade de dados explicitamente no nível da aplicação. Nesse sentido, a estratégia inicial consistiu em uma divisão simples do grafo, alocando-se os primeiros $N/2$ vértices em um nó e os restantes no segundo, tanto no *Edge Array* (na Memória Persistente) quanto no *Vertex Array* (na DRAM). Essa estratégia visava evitar acessos remotos e, assim, explorar o potencial da arquitetura NUMA da máquina utilizada.

Tal abordagem, apesar de sua simplicidade, mostrou-se insuficiente para grafos reais, principalmente em função das características de distribuição de grau, geralmente do tipo *power-law*. Como apontado na literatura (WANG et al., 2022), grafos extraídos de aplicações reais frequentemente exibem uma concentração de grande número de arestas em poucos vértices. Uma distribuição sequencial dos vértices tende a alocar os vértices de grau mais alto em um único nó, ocasionando severo desbalanceamento de carga e aumento do tempo de resposta

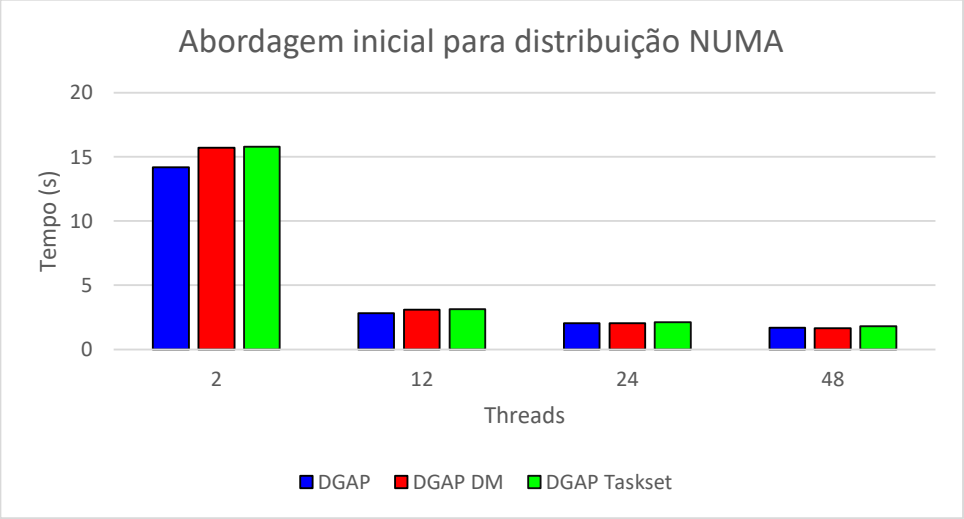

Figura 4.2: Comparação de desempenho entre DGAP, DGAP DM e DGAP Taskset nos testes iniciais com *PageRank*.

para operações de análise.

A implementação dessa abordagem exigiu, além da alocação dos novos *arrays*, a adaptação de múltiplas funções do *framework* DGAP, tais como a função de inserção e funções responsáveis pela distribuição das arestas com base no grau dos vértices de origem, evitando a concentração de densidade em regiões específicas do *array*. Este processo revelou-se o mais complexo, devido à necessidade de reorganização da PMA e da adequação das funções de manipulação das estruturas de dados, além da revisão de funções utilizadas pelos *benchmarks*, como as funções que recuperam o grau de um vértice, entre outras, cujo não alinhamento poderia comprometer a correta execução do código.

Concluída essa etapa, foram realizados testes de validação e desempenho, comparando o grafo adaptado com o DGAP original, tanto em termos de correção estrutural quanto de resultados produzidos pelos *benchmarks*, como o *score* do *PageRank*. Os testes iniciais, contudo, indicaram que não houve ganhos de desempenho em relação ao DGAP original quando este era executado em ambiente NUMA com distribuição forçada das *threads* (via comando `taskset`). Ademais, em comparação ao DGAP sem o uso de `taskset`, observou-se desempenho consideravelmente inferior.

A Figura 4.2 ilustra a comparação de desempenho no *PageRank* entre o DGAP modificado (DGAP DM), o DGAP original com distribuição forçada de *threads* (DGAP Taskset) e o DGAP original, evidenciando os desafios enfrentados nessa etapa inicial.

Esse resultado motivou uma análise mais aprofundada das causas do desequilíbrio de desempenho. Uma das primeiras hipóteses avaliadas foi a distribuição inadequada dos *threads*, que, no modo automático, eram alocados de forma subótima, resultando em frequentes acessos *cross-node*. Para mitigar esse efeito, adaptou-se os *benchmarks* para realizar *bind* explícito dos *threads* aos respectivos nós, medida que, inesperadamente, agravou o desempenho.

Aprofundando-se a investigação, identificou-se um grave desbalanceamento na dis-

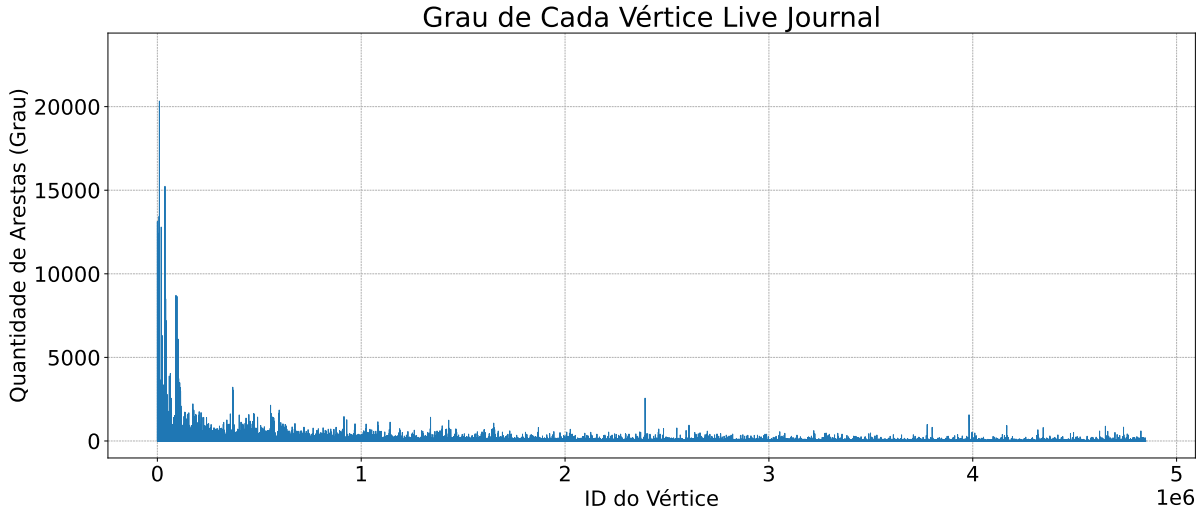

Figura 4.3: Arestas por Vértice Live Journal.

tribuição das arestas entre os nós, embora a quantidade de vértices permanecesse equilibrada (1.572.864 em cada nó). No nó 0, foram contabilizadas 160.276.767 arestas, ao passo que, no nó 1, esse número foi de apenas 77.239.127, evidenciando um desbalanceamento que impactava diretamente o processamento, principalmente na presença de *binds* fixos, já que um nó passava a ser responsável por uma carga computacional muito superior, sem a redistribuição automática proporcionada pelo `OpenMP`.

Esse fenômeno decorre das propriedades dos grafos tratados, que apresentam distribuição *power-law* e efeito de *Preferential Attachment* (vide Capítulo 3), resultando em poucos vértices iniciais com alta concentração de conexões (WANG et al., 2022). Os gráficos apresentados nas Figuras 4.3 e 4.4 ilustram esse comportamento para os conjuntos de dados Live Journal e Orkut, respectivamente, demonstrando a preponderância de arestas em uma das extremidades da ordenação dos vértices.

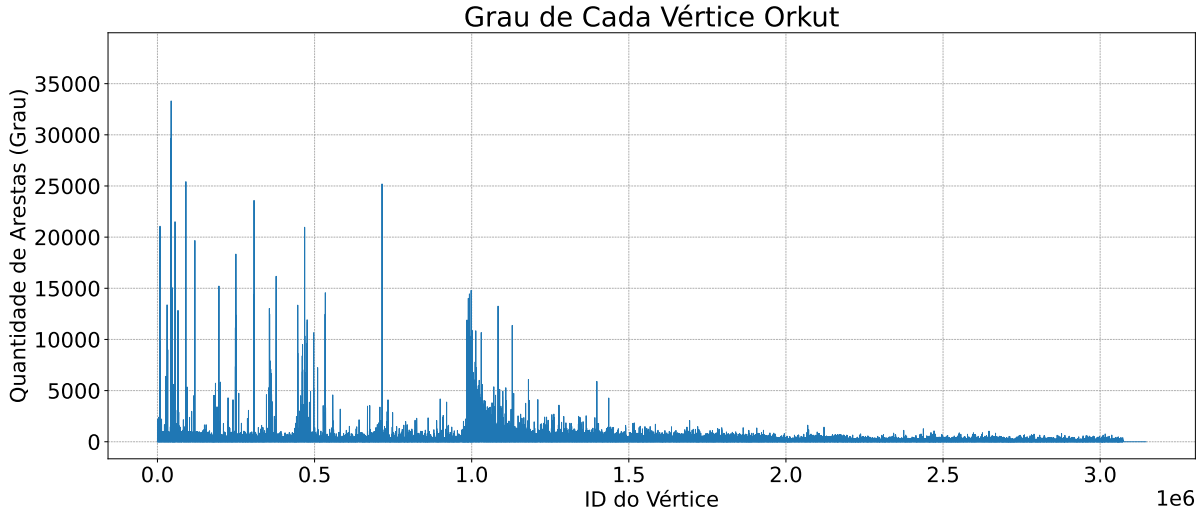

Figura 4.4: Arestas por Vértice Orkut.

Diante desse contexto, optou-se por adotar uma estratégia alternativa de distribui-

ção, inspirada no *framework* XPGraph, na qual o nó de destino de cada vértice é determinado pelo resultado da operação $V\%N$, sendo V o identificador do vértice e N o número de nós NUMA (WANG et al., 2022). Essa abordagem caracteriza-se como uma distribuição Round-Robin via *hash*, amplamente utilizada na literatura para balanceamento de carga em ambientes paralelos. O método proporciona uma distribuição mais uniforme do grafo entre os nós, minimizando eventuais desequilíbrios, excetuando casos atípicos em que, por exemplo, vértices de identificadores pares concentrem desproporcionalmente mais arestas que os ímpares.

A Figura 4.5 exemplifica como funciona a estratégia de distribuição Round-Robin dos vértices entre os nós NUMA. No exemplo apresentado, cada vértice é atribuído a um nó por meio da operação $V\%2$, onde V é o identificador do vértice. Dessa forma, todos os vértices pares ficam no Node 0 e os ímpares no Node 1. Esse tipo de distribuição garante um balanceamento mais uniforme dos vértices entre os nós, evitando a sobrecarga em apenas um lado do sistema. Além disso, é uma abordagem simples, mas eficiente para dividir grafos de grande porte, especialmente quando a distribuição do grau dos vértices é bastante heterogênea, como ocorre em aplicações reais.

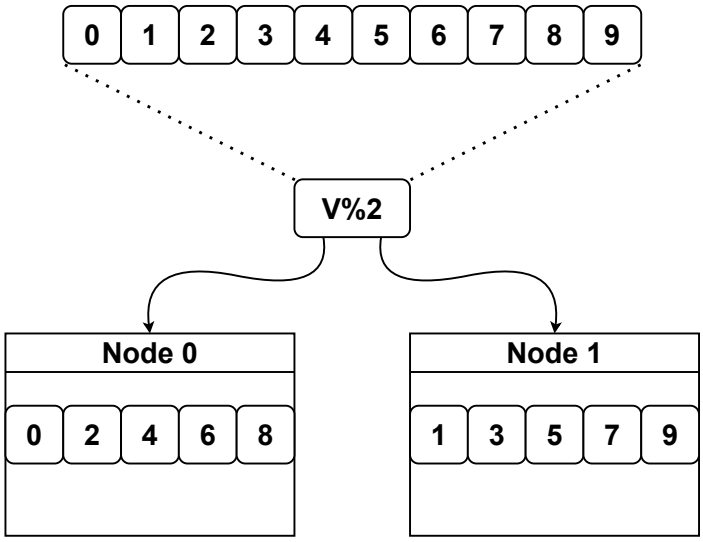

Figura 4.5: Exemplo de distribuição dos vértices do grafo entre dois nós NUMA utilizando o método Round-Robin.

No XPGraph, a estratégia de distribuição via Round-Robin é complementada por mecanismos de bufferização dinâmica (*subgraphs*, *in-out graphs*) e pelo ajuste do tamanho dos *buffers* conforme a frequência e grau dos vértices (WANG et al., 2022). Esses mecanismos têm o objetivo de otimizar tanto o balanceamento de carga quanto o acesso local em ambientes NUMA-PMEM, atenuando o impacto dos acessos cruzados e maximizando a eficiência de leitura e escrita persistente.

O processo de adaptação das estruturas do DGAP a essa nova política de distribuição mostrou-se mais ágil em virtude da experiência adquirida nas etapas anteriores. Ao término das modificações, medições apontaram um balanceamento quase ideal: o nó 0 passou a arma-

zenar 118.729.840 arestas (49,98%) e o nó 1, 118.786.054 (50,02%), demonstrando a eficácia da abordagem.

Apesar da melhoria na distribuição, persistiram desafios relacionados à execução dos *benchmarks*. Devido ao entrelaçamento dos vértices entre os nós, o *OpenMP* não conseguia mais distribuir o trabalho de forma eficiente sem um mecanismo adicional de vinculação das *threads* aos respectivos nós, a fim de evitar acessos remotos (*cross-node*). Para tanto, foram adaptados os principais algoritmos (*PageRank*, *Betweenness Centrality*(BC), entre outros) para realizar explicitamente esse vínculo, adotando-se uma estratégia similar à da distribuição dos vértices: cada *thread* é vinculada ao nó resultante de $T\%N$, onde T representa o identificador da *thread* e N, a quantidade de nós NUMA. Dessa forma, cada *thread* inicia o processamento a partir do vértice correspondente ao seu identificador, avançando a cada iteração em incrementos do número total de *threads*.

No XPGraph, o *thread binding* é também fundamental para a minimização de latência em PM. O *framework* adota a política de alinhar o processamento de cada *thread* à localidade dos dados, evitando o tráfego inter-nó sempre que possível (WANG et al., 2022). Na adaptação do DGAP, verificou-se que a ausência de *binding* adequado resultava em desempenho errático, enquanto o vínculo explícito das *threads* aos nós NUMA permitiu ganhos consideráveis de estabilidade e eficiência.

Todas as otimizações concentram-se na forma como os vértices são particionados entre os nós NUMA (através do Round-Robin via *hash*) e na fixação das *threads* aos nós correspondentes, maximizando a localidade de memória. Em síntese, a distribuição Round-Robin dos vértices, aliada ao vínculo explícito das *threads* aos nós NUMA, revelou-se a estratégia mais eficaz para garantir o balanceamento das cargas de trabalho, mitigando os impactos negativos observados nas abordagens anteriores.

4.4 Comparação com XPGraph

A adoção da distribuição Round-Robin na adaptação do DGAP, inspirada no XPGraph, apresenta distinções relevantes tanto no aspecto estrutural quanto operacional. Enquanto o XPGraph recorre a uma infraestrutura mais complexa, com múltiplas representações (subgraphs, in-out graphs), *buffers* hierárquicos dinâmicos e mecanismos específicos para mitigar acessos remotos em PM (WANG et al., 2022), a adaptação aqui proposta busca manter a simplicidade arquitetural herdada do DGAP, priorizando a eficiência do acesso sequencial proporcionado pelo CSR/PMA, complementado por *logs* e *undo logs* persistentes.

No XPGraph, a escolha entre *subgraphs* (distribuição Round-Robin) e *in-out graphs* (separação entre listas de adjacência de entrada e saída) é motivada pela necessidade de atender tanto *workloads* de análise quanto cenários de atualização intensiva. O gerenciamento do tamanho dos *buffers* é adaptativo, conforme o grau dos vértices e o *framework* emprega técnicas para evitar que vértices de alto grau monopolizem o tráfego de um nó específico, reduzindo assim a

probabilidade de congestionamento em PMEM.

No DGAP, a adoção da distribuição Round-Robin dos vértices mostrou-se suficiente para garantir bom balanceamento de carga e minimização de acessos *cross-node*, sobretudo em *workloads* predominantemente de busca (*graph analytics*). A estratégia apresenta robustez mesmo diante de grafos com distribuição de grau enviesada, contanto que a função *hash* proporcione dispersão adequada dos vértices entre os nós NUMA. Uma diferença notável é que a adaptação do DGAP não exige nenhuma ordenação prévia dos vértices, nem requer conhecimento do grau ou frequência de acesso, ao contrário de abordagens que se beneficiam de ordenação por grau para otimizar a localidade (KELLY, 2020).

Por outro lado, enquanto a simplicidade da adaptação no DGAP favorece facilidade de implementação e menor *overhead* de gerenciamento, o XPGraph pode apresentar vantagens em cenários de alta taxa de atualização de arestas, graças ao seu mecanismo dinâmico de *buffers* e *logs* intermediários, embora à custa de maior complexidade de código e maior consumo de memória volátil.

Do ponto de vista prático, a escolha da estratégia mais adequada depende não apenas do perfil do grafo, mas principalmente da natureza do *workload*. Para *workloads* de *graph analytics* típicos, nos quais predominam operações de busca e análise de conectividade, a adaptação feita no DGAP oferece robustez e performance alinhadas à simplicidade do modelo CSR/PMA. Em contrapartida, para cenários que exigem alta taxa de inserções, exclusões e atualizações dinâmicas, XPGraph pode apresentar vantagens, a despeito do custo adicional em gerenciamento e sincronização.

É importante destacar que o desempenho oriúndo da adaptação realizada no DGAP não se limita à simplicidade de implementação. O uso do formato CSR, especialmente em sua versão adaptada com PMA e *logs* persistentes, oferece vantagens significativas em termos de eficiência espacial, acesso sequencial e escalabilidade em operações de análise de grafos. Enquanto o XPGraph adota múltiplas estruturas de dados e *buffers* intermediários para lidar com *workloads* altamente dinâmicos, essa abordagem acarreta maior fragmentação dos dados e um consumo de memória consideravelmente superior.

Para *workloads* típicos de *graph analytics*, nos quais predominam operações de busca e leitura massiva sobre grafos de grande escala, a estrutura CSR utilizada pelo DGAP proporciona não apenas desempenho superior, mas também maior facilidade de paralelização e melhor aproveitamento de *cache*, fatores essenciais em ambientes NUMA com PM. Dessa forma, a adaptação do DGAP se mostra uma solução mais adequada para a maioria das aplicações práticas, conciliando desempenho, robustez e escalabilidade, sem abrir mão da flexibilidade para operações dinâmicas quando necessário.

4.5 Aspectos de Implementação

A adaptação proposta neste trabalho foi realizada a partir do código-fonte original do DGAP, implementado em C++. As modificações foram incorporadas diretamente no código, mantendo uma base única de código para as versões original e otimizada. A seleção da versão desejada (original ou NUMA-aware) é feita por meio de *flags* de compilação, o que facilita o gerenciamento e a reprodutibilidade dos experimentos.

Para as adaptações de suporte à arquitetura NUMA, foram empregadas rotinas específicas para particionamento dos vértices e gerenciamento de afinidade de *threads*, integradas ao código em C++ com o auxílio da biblioteca PMDK para acesso eficiente à memória persistente. Adicionalmente, *scripts* de automação e coleta de dados foram desenvolvidos para padronizar a execução dos experimentos e a análise dos resultados, feitos em *python*. Todo o código desenvolvido para este trabalho está disponível publicamente no repositório GitHub.

Capítulo 5

Resultados

Neste capítulo são apresentados os resultados experimentais obtidos com a adaptação do DGAP para arquiteturas NUMA, conforme detalhado no Capítulo 4. Inicialmente, descrevem-se os *workloads* utilizados na avaliação, seguidos pela caracterização dos conjuntos de dados empregados, das abordagens de processamento de grafos selecionadas para comparação e do ambiente experimental adotado. Na sequência, são discutidos os resultados experimentais, destacando os ganhos de desempenho proporcionados pela adaptação do DGAP para ambientes NUMA.

5.1 Benchmarks

Para garantir uma comparação justa e uniforme entre os diferentes *frameworks* de grafos, foram utilizados quatro *benchmarks* padronizados, já integrados ao DGAP. Esses *benchmarks* são amplamente reconhecidos na literatura e cobrem aspectos distintos do processamento de grafos, permitindo avaliar tanto desempenho computacional quanto escalabilidade. A seguir, são descritos de forma sucinta os algoritmos adotados e o que cada um avalia:

- **PageRank**: Algoritmo clássico de ranqueamento, originalmente desenvolvido para ordenar páginas na web (BRIN; PAGE, 1998). Sua lógica atribui um peso a cada vértice do grafo de acordo com o número e a qualidade dos vértices que apontam para ele. A cada iteração, o algoritmo recalcula a importância relativa dos nós, convergindo para uma distribuição estável de "relevância". O *PageRank* é particularmente relevante para avaliar o desempenho em operações iterativas de larga escala, típicas em análise de grafos (BRIN; PAGE, 1998; HAGBERG; SCHULT; SWART, 2008);

- **Betweenness Centrality**: Mede a importância de um vértice com base na quantidade de caminhos mínimos que passam por ele. Quanto mais vezes um vértice aparece como intermediário nos caminhos mais curtos entre outros pares de vértices, maior seu valor de centralidade. Esse *benchmark* é exigente do ponto de vista computacional e serve para testar o desempenho do *framework* em operações de cálculo global e acesso intensivo a diferentes partes do grafo (FREEMAN, 1977);

- **Breadth-First Search (BFS)**: Algoritmo fundamental para busca em grafos, explora todos os vizinhos de cada vértice em "largura", expandindo camada por camada. O BFS é muito utilizado para medir o tempo de resposta do *framework* em operações de travessia e descoberta de conectividade, sendo sensível à eficiência de acesso à memória e à estrutura de dados adotada (CORMEN et al., 2009);

- **Connected Components (CC)**: Algoritmo que identifica subconjuntos de vértices interconectados entre si, mas desconectados do restante do grafo. Em grafos não direcionados, cada componente representa um "bloco" isolado. Este *benchmark* avalia a capacidade do *framework* de percorrer grandes volumes de dados e segmentar o grafo de acordo com sua estrutura de conectividade (WEST, 2001).

5.2 Conjuntos de Dados

Os experimentos descritos neste trabalho empregaram dois grafos reais extraídos do repositório SNAP (LESKOVEC; KREVL, 2014): Orkut e LiveJournal. Ambos são amplamente adotados como *benchmarks* em estudos de processamento de grafos devido à sua grande escala e à representatividade de redes sociais do mundo real (ISLAM; DAI, 2023).

A Tabela 5.1 apresenta as principais características topológicas de cada conjunto de dados, incluindo o número total de vértices, arestas e o grau médio aproximado de cada grafo. Esses parâmetros são fundamentais para a análise dos resultados, uma vez que influenciam diretamente o impacto das otimizações NUMA e das estratégias de particionamento avaliadas.

Tabela 5.1: Principais propriedades dos conjuntos de dados utilizados

Conjunto de Dados	Vértices	Arestas	Grau médio
Orkut	3 072 626	234 370 166	76
LiveJournal	4 847 570	85 702 474	18

O grafo Orkut é caracterizado por alta densidade, apresentando um grau médio aproximado de 76, ou seja, cada vértice possui, em média, 76 conexões. Por outro lado, o LiveJournal possui uma estrutura significativamente menos densa, com grau médio de 18. Essa diferença estrutural influencia diretamente o comportamento dos algoritmos de processamento paralelos e o impacto das otimizações NUMA, conforme discutido na análise dos resultados.

5.3 Sistemas Comparados

Para avaliar a eficácia da otimização proposta, foram selecionados para comparação os seguintes sistemas e variantes de processamento de grafos dinâmicos sobre memória persistente:

CSR: Versão do formato *Compressed Sparse Row* adaptada para operar diretamente sobre memória persistente. A implementação foi modificada para que suas estruturas de dados residam na PM. Por ser uma estrutura estática, não possui suporte nativo a atualizações dinâmicas. É utilizada como linha de base para aferir o desempenho de análise de grafos, uma vez que sua organização de memória compacto oferece uma performance ótima para essa tarefa, permitindo quantificar a sobrecarga de sistemas que suportam dinamismo;

GraphOne: *Framework* originalmente projetado para análise eficiente de grafos em memória volátil, com garantia de durabilidade via armazenamento externo não volátil (ISLAM; DAI, 2023). Para adaptação à memória persistente, o GraphOne foi modificado para armazenar sua fase durável diretamente na PM, realizando a transferência dos dados da DRAM para a memória persistente após cada lote de inserções. Essa versão adaptada permite analisar o impacto de migrar sistemas originalmente baseados em DRAM para ambientes de memória persistente. Ressalta-se que, devido à sua estrutura baseada em listas de adjacências e *logs* de arestas, a eficiência do GraphOne pode ser limitada por operações de sincronização e movimentação de dados entre as estruturas, sobretudo em cenários altamente dinâmicos (ISLAM; DAI, 2023);

DGAP: *Framework* desenvolvido especificamente para operar sobre memória persistente, baseado em uma versão mutável do CSR, incorporando *logs* por seção e mecanismos eficientes de recuperação após falhas. O DGAP foi projetado para combinar desempenho em análise e eficiência em atualizações dinâmicas, minimizando problemas clássicos de *write amplification* e latência de persistência (ISLAM; DAI, 2023). Base para as adaptações deste trabalho;

DGAP DM: Primeira adaptação do DGAP, onde a divisão era feita dividindo a primeira metade de vértices em um nó e a segunda em outro. Será utilizada para comparar como a forma de dividir os vértices impacta o desempenho na parte de análise do grafo;

DGAP RR: Versão modificada do DGAP proposta neste trabalho, na qual os vértices são particionados entre os nós NUMA utilizando a estratégia Round-Robin baseada em *hash*, com afinidade explícita das *threads* aos nós de memória correspondentes. Não há alterações nas estruturas de dados internas, restringindo-se as modificações ao particionamento e ao gerenciamento de afinidade, visando maximizar a localidade de acesso à memória;

XPGraph: *Framework* de última geração projetado para memória persistente, fundamentado em *log* circular de arestas e listas de adjacências mantidos diretamente na PMEM (WANG et al., 2022). XPGraph foi desenvolvido para superar limitações de *frameworks* anteriores (como o próprio GraphOne) em ambientes de memória persistente, balanceando desempenho de atualização e análise. Nesta avaliação, foi empregada a versão original, sem otimizações NUMA, devido à indisponibilidade dos *benchmarks* adaptados para NUMA.

5.4 Ambiente Experimental

Os experimentos deste trabalho foram realizados em um servidor equipado com dois processadores Intel Xeon Gold 5317 (*CPU0* e *CPU1*), 256 GB de memória DRAM e oito módulos de memória persistente Intel Optane DC série 200, cada um com 128 GB, totalizando 1 TB. Os módulos de memória persistente foram configurados no modo *App Direct*, com suporte ao modo DAX (*Direct Access*) ativado. Tanto a memória persistente quanto a DRAM convencional estão conectadas diretamente aos barramentos dos processadores, sendo quatro módulos de memória persistente alocados a cada processador. Na configuração do sistema (*BIOS*), esses módulos foram agrupados em uma única unidade lógica de memória, resultando em dois nós de memória persistente apresentados ao sistema operacional, denominados $PM0$ e $PM1$.

Cada experimento foi repetido dez vezes. Para cada conjunto de amostras, calculamos o intervalo de confiança de 95 % pelo método *bootstrap*, utilizando 5 000 reamostragens (EFRON; TIBSHIRANI, 1994). A variação observada foi inferior a 3 % em todos os casos; para não sobrecarregar a leitura, as barras de erro não são exibidas nos gráficos.

Em todas as abordagens, exceto no DGAP Round-Robin, as execuções foram realizadas sem *bind* explícito das *threads*, permitindo que o sistema operacional gerenciasse o escalonamento entre os nós NUMA. Essa escolha visa reproduzir o comportamento típico de aplicações não NUMA-aware em ambientes reais. No caso do DGAP Round-Robin, por sua vez, a afinidade das *threads* aos nós NUMA é definida de forma automática pela própria implementação, assegurando que o posicionamento das *threads* seja controlado e não dependa das políticas do sistema operacional.

Todos os gráficos apresentados reportam o *speed-up* (razão de desempenho) em relação ao *baseline* **CSR com 1 *thread***. O uso dessa configuração como referência se justifica pelo fato de o CSR ser amplamente reconhecido na literatura como uma das implementações mais eficientes para grafos estáticos em DRAM, representando, portanto, um limite superior de desempenho para esse tipo de processamento (ISLAM; DAI, 2023; WANG et al., 2022). Assim, o *speed-up* S de uma abordagem A é calculado por:

$$S = \frac{T_{\mathrm{CSR,\,1T}}}{T_A}$$

em que $T_{\mathrm{CSR,\,1T}}$ é o tempo médio da implementação CSR em execução com 1 *thread* e T_A é o tempo médio da abordagem A com o respectivo número de *threads* considerado no experimento. Dessa forma, valores $S > 1$ indicam aceleração em relação ao *baseline*, enquanto $S < 1$ indicam degradação de desempenho. Tal escolha de referência também está alinhada com práticas adotadas em trabalhos anteriores, permitindo comparações diretas e contextualizadas dos resultados obtidos.

5.5 Resultados Obtidos

Os experimentos deste trabalho foram conduzidos utilizando dois conjuntos de dados extraídos do repositório SNAP (LESKOVEC; KREVL, 2014): Orkut e LiveJournal. Esses grafos representam redes sociais reais e são amplamente empregados como *benchmarks* em estudos de processamento de grafos devido à sua grande escala e relevância para aplicações práticas (ISLAM; DAI, 2023).

Como comentado anteriormente, os grafos Orkut e LiveJournal possuem diferenças quanto à densidade, variando número de vértices e arestas. Essas diferenças topológicas têm impacto direto no comportamento dos algoritmos paralelos e na eficácia das otimizações NUMA, influenciando a relação custo-benefício do particionamento e do gerenciamento de afinidade de *threads*.

Para facilitar a compreensão e destacar os efeitos dessas características estruturais nos resultados, a análise será apresentada separadamente para cada conjunto de dados, permitindo observar como o desempenho das abordagens varia de acordo com o perfil do grafo.

Durante o processo de avaliação comparativa, foi observado que a implementação NUMA do XPGraph apresentou variabilidade nos resultados do *PageRank* e *Connected Components* entre execuções sucessivas sob condições idênticas, o que dificultou sua utilização como referência confiável de desempenho nesta análise. Como parte da verificação, utilizou-se a biblioteca Python `networkx` (HAGBERG; SCHULT; SWART, 2008), cujos resultados para o *PageRank* coincidiram com aqueles obtidos pelo DGAP, sugerindo que a inconsistência observada estava limitada à versão avaliada do XPGraph.

Apesar dos esforços dedicados à utilização da versão NUMA do XPGraph, não foi possível obter resultados estáveis para comparação. Assim, optou-se por empregar, nesta avaliação, a versão do XPGraph utilizada no artigo do DGAP (ISLAM; DAI, 2023), que não contempla as otimizações NUMA.

5.5.1 Resultados Orkut

Os resultados experimentais evidenciam ganhos expressivos proporcionados pela estratégia de divisão Round-Robin em ambientes NUMA. Em cenários nos quais o efeito NUMA é pouco pronunciado — por exemplo, ao comparar com o DGAP executando em apenas duas *threads* — o desempenho da adaptação Round-Robin manteve-se equivalente ao da versão original. Nessa configuração, o DGAP concentra todo o processamento em um único nó NUMA, enquanto a versão baseada em Round-Robin distribui automaticamente os dados entre diferentes nós de memória.

A Figura 5.1 apresenta o desempenho de todos os *frameworks* no *benchmark PageRank*, utilizando 2, 12, 24 e 48 *threads*. Observa-se que, com 2 *threads*, a versão modificada obteve desempenho equivalente ao DGAP e superior ao XPGraph. Entretanto, em configurações em que o efeito NUMA se torna mais relevante, os resultados favorecem significativamente

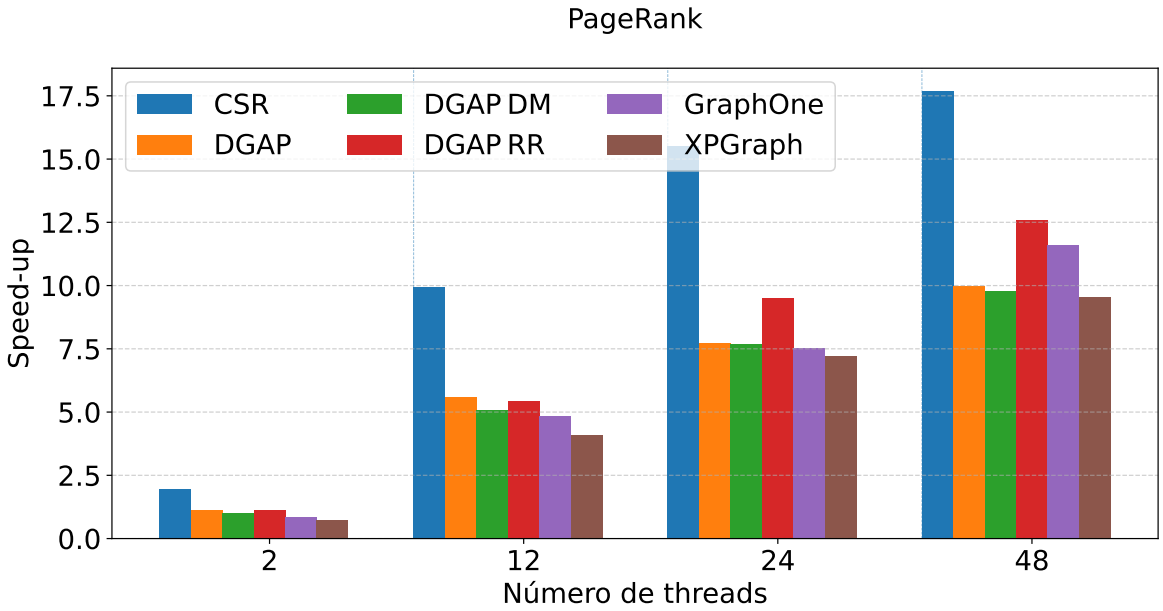

Figura 5.1: Resultados do *PageRank* no *dataset* Orkut.

a abordagem baseada em Round-Robin. No cenário com 48 *threads* (24 por nó NUMA), a solução proposta atingiu um *speedup* de aproximadamente 1,25× em relação ao DGAP original e de 1,32× em relação ao XPGraph.

Além disso, observa-se um ganho de desempenho considerável na execução com 24 *threads*. Esse resultado decorre do fato de que, na configuração original, o DGAP utiliza os *hyperthreads* da Intel, os quais não oferecem o mesmo desempenho dos núcleos físicos do processador, enquanto a versão Round-Robin explora todos os núcleos físicos disponíveis. Essa diferença resultou em um *speedup* de aproximadamente 1,23×.

Para o *Connected Components*, já com duas *threads* a adaptação Round-Robin atingiu *speedup* próximo de 1,29× em relação ao DGAP. Quando escalada para o ambiente com 48 *threads*, a aceleração foi ainda mais expressiva, chegando a 2,2× em comparação com a versão original, conforme ilustrado na Figura 5.2.

Esse resultado pode ser atribuído a dois fatores principais. Primeiro, a estratégia de particionamento Round-Robin, aliada à afinidade explícita das *threads* aos nós de memória, permite que o DGAP RR utilize de forma mais eficiente todos os canais de acesso à memória disponíveis no sistema NUMA, maximizando a largura de banda agregada e reduzindo gargalos de acesso. Segundo, o algoritmo de *Connected Components* favorece padrões de leitura sequencial e apresenta alto grau de paralelismo, o que possibilita que múltiplas *threads* percorram diferentes regiões do grafo de maneira independente e balanceada. Como o CSR tradicional costuma centralizar suas estruturas em um único nó de memória, acaba enfrentando maior contenção e penalidades de acesso remoto em cenários altamente paralelos. Dessa forma, a abordagem do DGAP RR não apenas reduz a latência média dos acessos, mas também distribui a carga computacional de maneira mais homogênea, superando até mesmo a implementação CSR

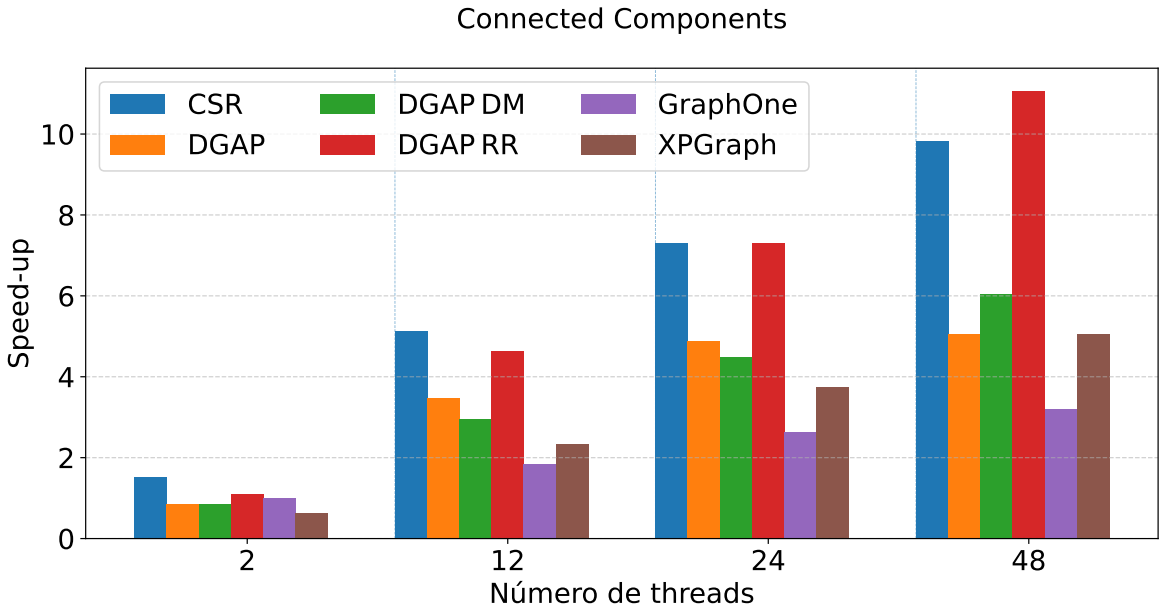

Figura 5.2: Resultados do *Connected Components* no *dataset* Orkut.

em configurações com elevado número de *threads*, como observado nos experimentos com 48 *threads*.

Em 48 *threads* o DGAP DM também apresentou uma leve melhora em comparação ao DGAP original, devido à distribuição entre os nós NUMA.

No caso do *benchmark* BFS, observa-se que os *buffers* em DRAM presentes no XPGraph garantem melhor desempenho, fazendo com que esse *framework* obtenha os melhores resultados para essa tarefa, visto que ele não é limitado pela latência de acesso à PM. O DGAP Round-Robin apresentou um *speedup* de 2x em relação à versão original. A Figura 5.3 apresenta esses resultados.

Para o *benchmark* BC, a adaptação também apresentou ganhos substanciais em ambientes NUMA, alcançando um *speedup* de aproximadamente 1,24× em relação ao DGAP original. Destaca-se, ainda, que a solução proposta superou até mesmo o CSR tradicional, atingindo cerca de 1,16× de aceleração. Os resultados estão ilustrados na Figura 5.4.

Observa-se que, em grafos de alta densidade, o DGAP RR consegue superar o CSR tradicional na maioria das aplicações quando se utiliza 48 *threads*. Esse desempenho superior decorre principalmente do melhor balanceamento de carga proporcionado pelo particionamento entre os nós NUMA, aliado à afinidade explícita das *threads*, o que permite explorar mais eficientemente toda a largura de banda de memória disponível. Por outro lado, em configurações nas quais o efeito de acesso remoto ainda não se manifesta de forma significativa, como nas execuções com 24 *threads* ou menos, o CSR permanece como a solução mais robusta, beneficiando-se de sua estrutura compacta e do acesso local à memória.

A Tabela 5.2 apresenta os tempos médios de execução (em segundos) para cada *benchmark*, considerando diferentes *frameworks* e números de *threads*, permitindo uma análise

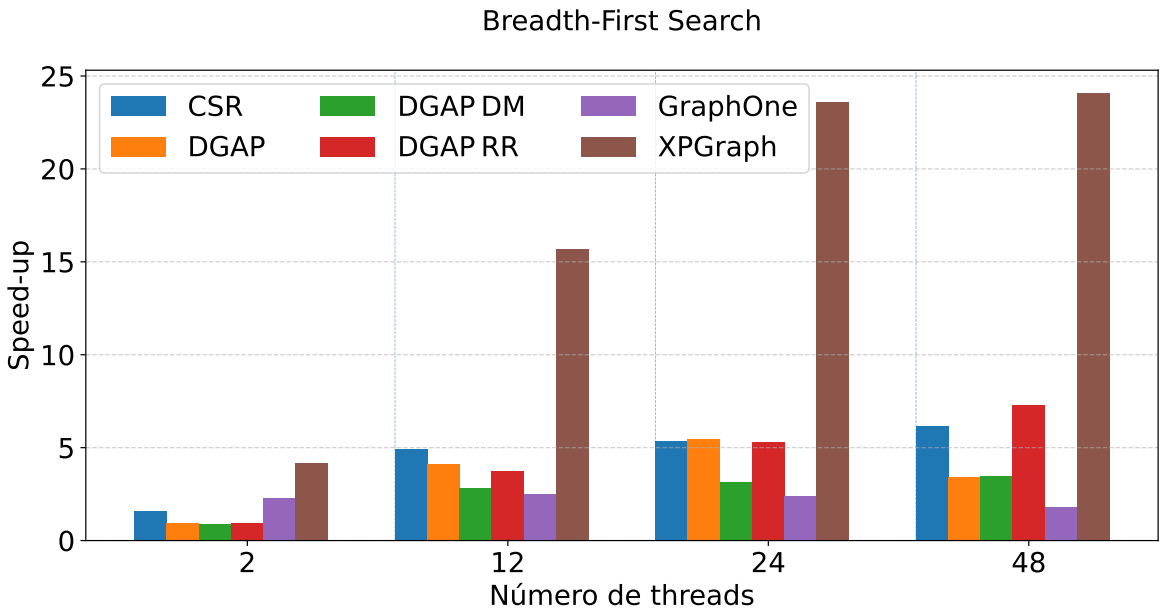

Figura 5.3: Resultados do *Breadth-First Search* no *dataset* Orkut.

detalhada do comportamento absoluto de cada abordagem no *dataset* Orkut. Observa-se, a partir dos valores absolutos, que o XPGraph obtém tempos de execução significativamente inferiores no *benchmark* BFS, destacando-se das demais abordagens para esse tipo de travessia. Por outro lado, em algoritmos como *Connected Components* e *Betweenness Centrality*, o DGAP RR se sobressai, alcançando os menores tempos entre as variantes avaliadas. Esses resultados reforçam a análise apresentada nos gráficos de *speedup*, evidenciando o impacto do padrão de acesso à memória sobre o desempenho relativo de cada *framework*. Já o DGAP original e o DGAP DM apresentam tempos de execução semelhantes na maioria dos *benchmarks* com 48 *threads*. No entanto, em 2 *threads*, o DGAP DM demonstra desempenho inferior em relação à versão original.

O GraphOne apresenta desempenho superior ao XPGraph em cenários com baixo número de *threads*, entretanto, à medida que o paralelismo aumenta, o XPGraph passa a se destacar, especialmente em configurações com 48 *threads*. Isso pode ser observado nos *benchmarks* BC e CC, nos quais o XPGraph apresenta desempenho inferior ao GraphOne com 2 *threads*, mas supera o concorrente quando se utiliza um maior número de *threads*.

5.5.2 Resultados LiveJournal

Os resultados obtidos com o LiveJournal evidenciam que o particionamento Round-Robin não é universalmente vantajoso. Esse grafo apresenta grau médio bem inferior ao Orkut ($\approx$18 vs.$\approx$76), de modo que cada vértice contém menos dados para serem processados a cada acesso. Em algoritmos sensíveis à latência, como o BFS, a distribuição dos dados entre nós NUMA aumenta a proporção de acessos remotos sem amortizar esse custo com trabalho útil suficiente,

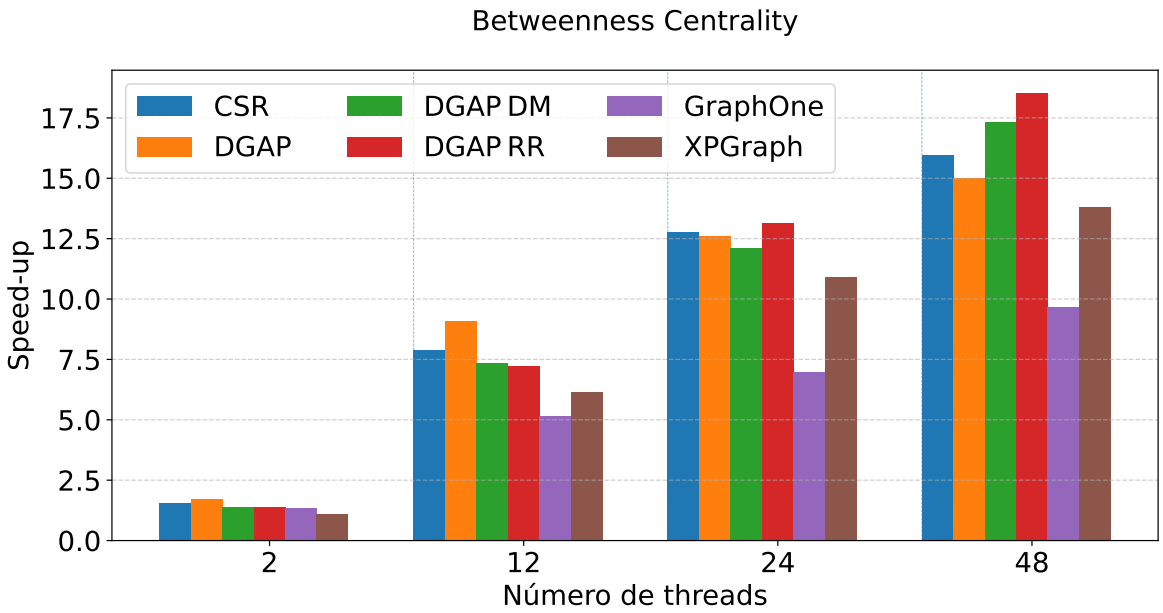

Figura 5.4: Resultados do *Betweenness Centrality* no *dataset* Orkut.

o que resulta em tempos de execução superiores ao DGAP original.

Além disso, o *benchmark* BFS, por acessar as arestas de vértices aleatórios a cada passo da busca, favorece implementações baseadas em lista de adjacências, como a do XPGraph, que conseguem acessar rapidamente os vizinhos de cada vértice mesmo em estruturas dispersas, minimizando penalidades de latência. Por outro lado, abordagens baseadas em CSR e PM, como o DGAP, sofrem maior impacto nesses acessos aleatórios, especialmente em cenários com baixo grau médio, agravando a latência de travessias e dificultando o escalonamento eficiente do BFS.

Em contraste, algoritmos com maior relação cálculo/acesso, como BC e CC, conseguem aproveitar melhor o paralelismo adicional proporcionado pela abordagem Round-Robin, compensando as penalidades de latência e resultando em aceleração significativa.

Adicionalmente, é importante ressaltar que algoritmos como *Connected Components* e *Betweenness Centrality* tendem a se beneficiar de padrões de acesso mais sequenciais e com maior localidade de memória, especialmente quando comparados a algoritmos de travessia como o BFS. Conforme caracterizado por Beamer et al. (BEAMER; LOW; AL., 2015), muitos algoritmos de análise de grafos apresentam taxas de acerto moderadas no cache de último nível e padrões de acesso mais regulares contribuem para melhor aproveitamento do sistema de memória, possibilitando um uso mais eficiente da largura de banda disponível. Dessa forma, otimizações que favorecem a localidade, como a distribuição balanceada e sequencial de vértices entre os nós NUMA, conseguem amortizar os custos de latência, favorecendo o escalonamento eficiente desses algoritmos. Como resultado, *benchmarks* como CC e BC exibem ganhos mais expressivos com a estratégia Round-Robin, ao contrário do observado em *workloads* com acesso mais aleatório, como o BFS.

Tabela 5.2: Tempos de execução (em segundos) para cada *benchmark*, considerando diferentes *frameworks* e número de *threads* para o *dataset* Orkut.

	CSR				DGAP				DGAP DM			
Benchmark	2T	12T	24T	48T	2T	12T	24T	48T	2T	12T	24T	48T
PageRank	8.10	1.59	1.02	0.89	14.22	2.84	2.04	1.59	15.73	3.12	2.06	1.62
BC	2.82	0.55	0.34	0.27	2.57	0.48	0.35	0.29	3.17	0.59	0.36	0.25
BFS	0.18	0.06	0.05	0.05	0.30	0.07	0.05	0.08	0.32	0.10	0.09	0.08
CC	1.15	0.34	0.24	0.18	2.06	0.50	0.36	0.35	2.06	0.59	0.39	0.29

	DGAP RR				GraphOne				XPGraph			
Benchmark	2T	12T	24T	48T	2T	12T	24T	48T	2T	12T	24T	48T
PageRank	14.22	2.91	1.67	1.26	18.40	3.26	2.11	1.37	21.96	3.87	2.20	1.66
BC	3.14	0.60	0.33	0.23	3.24	0.84	0.62	0.45	3.96	0.71	0.40	0.32
BFS	0.30	0.07	0.05	0.04	0.12	0.11	0.12	0.16	0.07	0.02	0.01	0.01
CC	1.60	0.38	0.24	0.16	1.75	0.95	0.67	0.55	2.82	0.75	0.47	0.34

Esses resultados confirmam observações prévias de que o impacto das otimizações NUMA depende fortemente da densidade e da distribuição de grau do grafo (ISLAM; DAI, 2023; WANG et al., 2022), como ilustrado nas Figuras 5.5, 5.6, 5.7 e 5.8. Destaca-se que a versão Round-Robin atingiu desempenho até 2,3 vezes superior ao DGAP tradicional no *benchmark* CC. Por outro lado, observou-se que todas as implementações enfrentaram dificuldades de escalabilidade no *benchmark* BFS, especialmente em 48 *threads*, com degradação de desempenho tanto na versão Round-Robin quanto no GraphOne.

A Tabela 5.3 apresenta os tempos médios de execução (em segundos) para cada *benchmark*, considerando diferentes *frameworks* e números de *threads*, permitindo uma análise detalhada do comportamento absoluto de cada abordagem no *dataset* LiveJournal. É possível observar que, no *benchmark* BFS, todos os *frameworks* apresentaram dificuldades em escalar o desempenho para 48 *threads*, havendo até casos em que o tempo de execução aumentou com o maior grau de paralelismo. Esse comportamento evidencia a natureza latência-sensível do BFS e o desafio de balancear o acesso à memória em ambientes NUMA para esse tipo de algoritmo.

Em síntese, os resultados experimentais evidenciam que a adaptação do DGAP para arquiteturas NUMA, por meio do particionamento Round-Robin e da afinidade explícita das *threads*, proporcionou ganhos expressivos de desempenho em diversos cenários, especialmente em grafos de maior densidade e em *benchmarks* caracterizados por forte demanda de largura de banda de memória, como *Connected Components* e *Betweenness Centrality*. Os experimentos também demonstraram que o impacto das otimizações NUMA não é homogêneo: a eficácia da abordagem proposta depende fortemente das características estruturais do grafo e do padrão de acesso de cada algoritmo, sendo que, em casos de menor densidade ou maior sensibilidade à latência, os ganhos podem ser limitados ou até revertidos.

Essas observações reforçam a importância de se considerar o perfil topológico dos dados e a natureza do processamento ao projetar estratégias de paralelização e particionamento

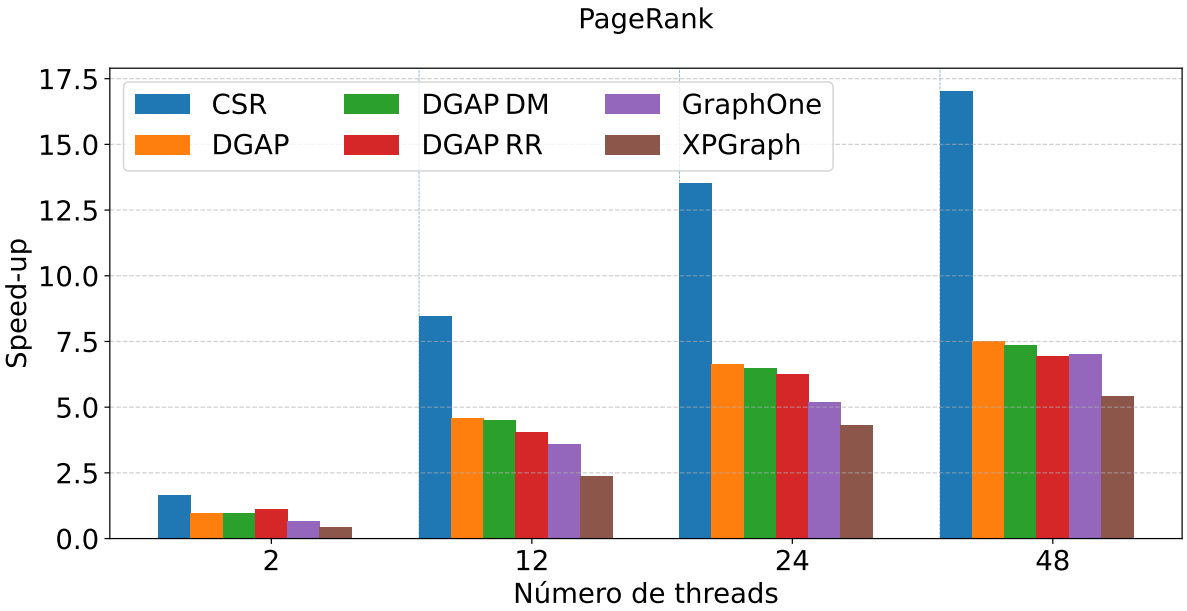

Figura 5.5: Resultados do *PageRank* no *dataset* LiveJournal.

em ambientes com múltiplos nós de memória. Os resultados obtidos não apenas validam a abordagem desenvolvida, mas também sugerem caminhos promissores para futuras investigações, especialmente no sentido de desenvolver mecanismos adaptativos que explorem, de forma dinâmica, a topologia e o comportamento dos grafos processados.

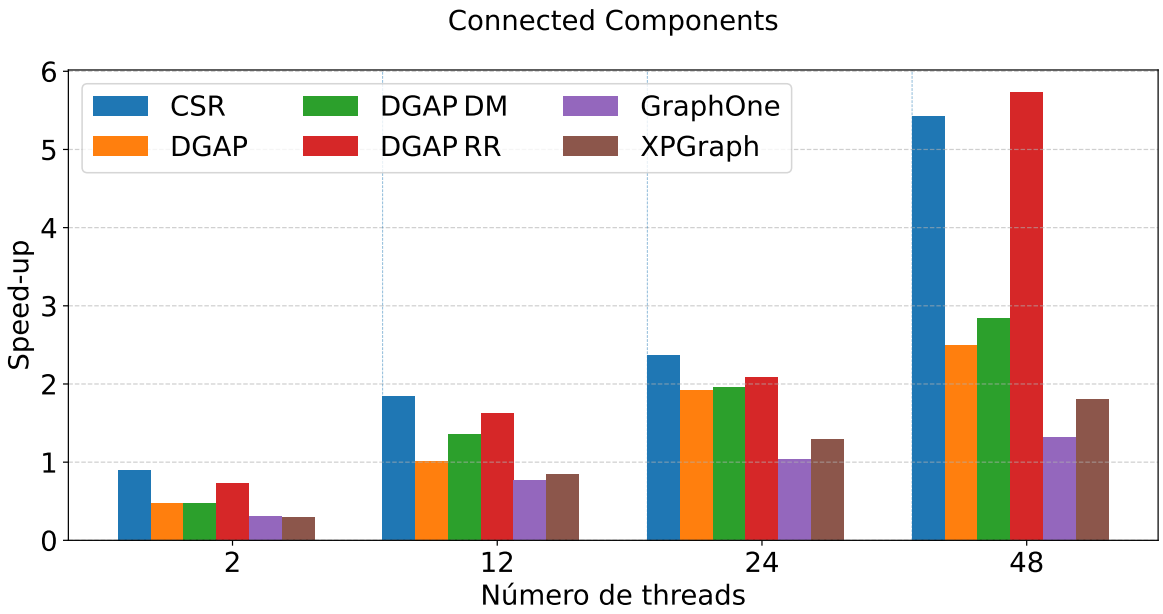

Figura 5.6: Resultados do *Connected Components* no *dataset* LiveJournal.

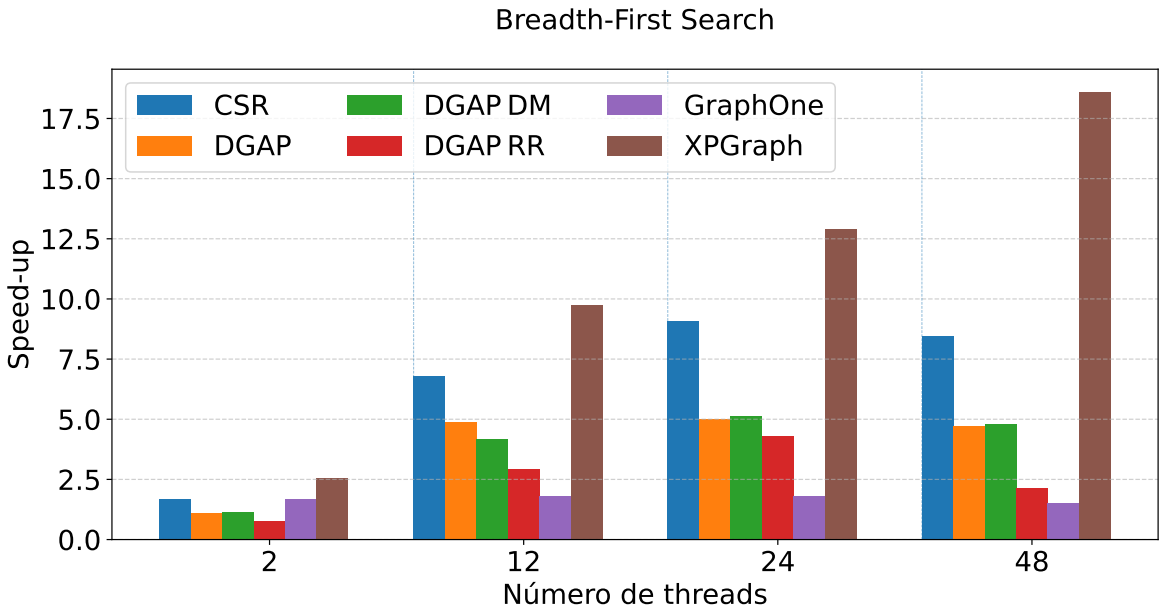

Figura 5.7: Resultados do *Breadth-First Search* no *dataset* LiveJournal.

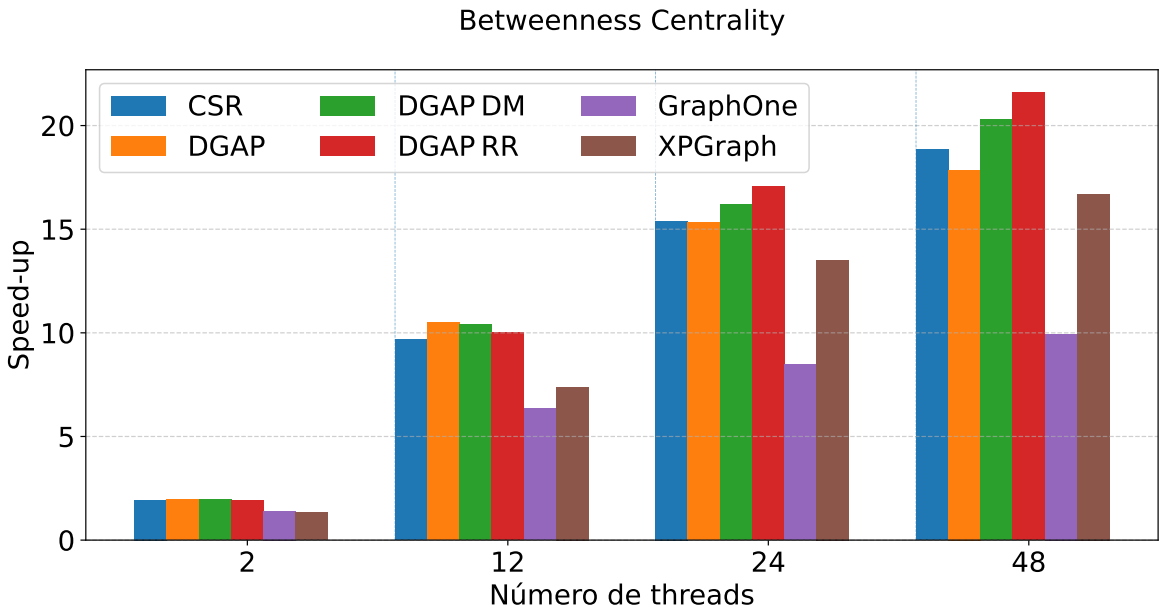

Figura 5.8: Resultados do *Betweenness Centrality* no *dataset* LiveJournal.

Tabela 5.3: Tempos de execução (em segundos) para cada *benchmark*, considerando diferentes *frameworks* e número de *threads* para o *dataset* LiveJournal.

	CSR				**DGAP**				**DGAP DM**			
Benchmark	2T	12T	24T	48T	2T	12T	24T	48T	2T	12T	24T	48T
PageRank	4.33	0.84	0.52	0.42	7.34	1.55	1.07	0.95	7.41	1.58	1.09	0.96
BC	2.26	0.45	0.28	0.23	2.23	0.42	0.29	0.25	2.25	0.42	0.27	0.22
BFS	0.19	0.05	0.03	0.04	0.28	0.06	0.06	0.07	0.28	0.08	0.06	0.07
CC	0.92	0.45	0.35	0.15	1.75	0.81	0.43	0.33	1.71	0.61	0.42	0.29

	DGAP RR				**GraphOne**				**XPGraph**			
Benchmark	2T	12T	24T	48T	2T	12T	24T	48T	2T	12T	24T	48T
PageRank	6.34	1.75	1.13	1.02	10.37	1.97	1.36	1.01	16.80	2.98	1.64	1.31
BC	2.27	0.44	0.26	0.20	3.13	0.69	0.52	0.44	3.29	0.59	0.32	0.26
BFS	0.40	0.11	0.07	0.15	0.19	0.17	0.17	0.21	0.12	0.03	0.02	0.02
CC	1.13	0.51	0.40	0.14	2.67	1.08	0.80	0.62	2.84	0.97	0.64	0.46

Capítulo 6

Conclusão

Neste trabalho, foi proposta e implementada uma adaptação do *framework* DGAP para ambientes NUMA, explorando o particionamento Round-Robin e a afinidade explícita de *threads*. Os resultados experimentais demonstraram que, para grafos com alta densidade ou carga computacional significativa por acesso, como o Orkut, a abordagem proposta proporcionou ganhos expressivos de desempenho, com aceleração superior a 2x em *benchmarks* como *Connected Components*.

Por outro lado, observou-se que tais otimizações não são universalmente vantajosas, sobretudo em grafos menos densos ou para algoritmos sensíveis à latência de acesso remoto, como BFS, onde o *overhead* do particionamento pode superar os benefícios esperados. Esses achados ressaltam a importância de considerar as características topológicas dos grafos e o perfil dos algoritmos na escolha de estratégias de paralelização e particionamento para ambientes NUMA.

Como principal contribuição, este trabalho evidencia a relevância do desenho de sistemas NUMA-aware para *frameworks* de processamento de grafos dinâmicos sobre memória persistente, ao mesmo tempo em que sinaliza limitações práticas e oportunidades para estratégias adaptativas.

6.1 Trabalhos Futuros

Embora os resultados obtidos demonstrem ganhos expressivos, o trabalho realizado abre novas perspectivas para aprimoramentos adicionais, visando adaptar o *framework* a uma gama ainda maior de cenários. Entre as possíveis extensões, destaca-se o desenvolvimento de estratégias híbridas de particionamento, que combinem a distribuição dos vértices com técnicas dinâmicas de balanceamento de carga, assegurando desempenho robusto mesmo em casos de distribuição irregular do grafo.

Além disso, a investigação de métodos mais avançados para o gerenciamento e migração de *threads* permitindo, por exemplo, que *threads* sejam realocadas entre diferentes

nós NUMA conforme a evolução do *workload*, representa um caminho promissor para ampliar ainda mais a eficiência e a escalabilidade da solução proposta.

Um aspecto importante a ser considerado em pesquisas futuras é a extensão das otimizações para cenários de grafos dinâmicos, o que inclui o suporte para operações de inserção e remoção de arestas e vértices, a fim de assegurar o balanceamento e o desempenho, mesmo com mudanças frequentes na estrutura do grafo. Por último, é importante ressaltar a necessidade de expandir a avaliação experimental para conjuntos de dados ainda mais amplos e variados, a fim de confirmar a solidez e a aplicabilidade das adaptações sugeridas em diferentes escalas e topologias.

Referências

BALDASSIN, Alexandro et al. Persistent Memory: A Survey of Programming Support and Implementations. **ACM Comput. Surv.**, v. 54, n. 7, jul. 2021. ISSN 0360-0300. DOI: 10.1145/3465402.

BALDASSIN, Alexandro et al. Persistent Memory: A Survey of Programming Support and Implementations. **ACM Comput. Surv.**, v. 54, n. 7, jul. 2021.

BEAMER, Scott; LOW, Yihan; AL., et. Locality exists in graph processing: Workload characterization on an ivy bridge server. In: IISWC. [S.l.: s.n.], 2015.

BRIN, Sergey; PAGE, Lawrence. The Anatomy of a Large-Scale Hypertextual Web Search Engine. **Computer Networks**, v. 30, p. 107–117, 1998. Disponível em: <https://snap.stanford.edu/class/cs224w-readings/Brin98Anatomy.pdf>.

CHAKRABARTI, Dhruva R.; BOEHM, Hans-J.; BHANDARI, Kumud. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In: PROCEEDINGS of the the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA). [S.l.: s.n.], out. 2014. P. 433–452.

CHEN, Yuetao; QIU, Keni et al. Smart scheduler: an adaptive NVM-aware thread scheduling approach on NUMA systems. **CCF Transactions on High Performance Computing**, v. 4, n. 4, p. 394–406, 2022. DOI: 10.1007/s42514-022-00110-2. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85139603932&doi=10.1007%2fs42514-022-00110-2&partnerID=40&md5=42b55bc839d6f4ec47fc6589639ec0c1>.

CHEN, Zhiwen; CHE, Wenkui et al. On the Performance Intricacies of Persistent Memory Aware Storage Engines. **IEEE Transactions on Knowledge and Data Engineering**, v. 35, n. 10, p. 10365–10382, 2023. DOI: 10.1109/TKDE.2023.3248643.

CHEN, Zhiwen; HU, Daokun et al. A quantitative evaluation of persistent memory hash indexes. **VLDB Journal**, v. 33, n. 2, p. 375–397, 2024. DOI: 10.1007/s00778-023-00812-1. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85170087225&doi=10.1007%2fs00778-023-00812-1&partnerID=40&md5=eadc66f77474d8d6c90d4b40ed8a94e2>.

CORMEN, Thomas H. et al. **Introduction to Algorithms**. 3rd. Cambridge, MA, USA: MIT press, 2009.

DASHTI, Mohammad et al. Traffic management: a holistic approach to memory placement on NUMA systems. In: PROCEEDINGS of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. Houston, Texas, USA: Association for Computing Machinery, 2013. (ASPLOS '13), p. 381–394. ISBN 9781450318709. DOI: 10.1145/2451116.2451157. Disponível em: <https://doi.org/10.1145/2451116.2451157>.

DROZDEK, A. **Data Structures and Algorithms in C++**. [S.l.]: Cengage Learning, 2012. ISBN 9781285415017. Disponível em: <https://books.google.com.br/books?id=PRgLAAAAQBAJ>.

EFRON, Bradley; TIBSHIRANI, Robert J. **An Introduction to the Bootstrap**. [S.l.]: Chapman & Hall/CRC, 1994.

FIRMLI, Soukaina et al. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. In_____. **24th International Conference on Principles of Distributed Systems (OPODIS 2020)**. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. v. 184. (Leibniz International Proceedings in Informatics (LIPIcs)), 17:1–17:16. ISBN 978-3-95977-176-4. DOI: 10.4230/LIPIcs.OPODIS.2020.17. Disponível em: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2020.17>.

FREEMAN, Linton C. A set of measures of centrality based on betweenness. **Sociometry**, American Sociological Association, v. 40, n. 1, p. 35–41, 1977.

HAGBERG, Aric A.; SCHULT, Daniel A.; SWART, Pieter J. Exploring network structure, dynamics, and function using NetworkX. In_____. **Proceedings of the 7th Python in Science Conference (SciPy2008)**. Pasadena, CA USA: [s.n.], 2008. P. 11–15.

HAN, Jungwook et al. Is Data Migration Evil in the NVM File System? In: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). [S.l.: s.n.], 2021. P. 26–31. DOI: 10.1109/ACSOS-C52956.2021.00024.

HENNESSY, John L.; PATTERSON, David A. **Computer Architecture: A Quantitative Approach**. 6th. [S.l.]: Morgan Kaufmann, nov. 2017.

ISLAM, Abdullah Al Raqibul; DAI, Dong. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In: PROC. of the SC'23. [S.l.: s.n.], 2023. DOI: 10.1145/3581784.3607106. Disponível em: <https://doi.org/10.1145/3581784.3607106>.

JACOB, Bruce; NG, Spencer W.; WANG, David T. CHAPTER 2 - Logical Organization. In _____. **Memory Systems**. Edição: Bruce Jacob, Spencer W. Ng e David T. Wang. San Francisco: Morgan Kaufmann, 2008. P. 79–115. ISBN 978-0-12-379751-3. DOI: https://doi.org/10.1016/B978-012379751-3.50004-7. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780123797513500047>.

JAMIL, Safdar et al. Scalable NUMA-aware persistent B+-tree for non-volatile memory devices. **Cluster Computing**, v. 26, n. 5, p. 2865–2881, 2023. DOI: 10.1007/s10586-022-03766-1. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85142164640&doi=10.1007%2fs10586-022-03766-1&partnerID=40&md5=60004205f1eff3a6b5b8204a2bf9ea0b>.

JIA, Wenqing; JIANG, Dejun; XIONG, Jin. NapFS: A High-Performance NUMA-Aware PM File System. In: 2022 IEEE 40th International Conference on Computer Design (ICCD). [S.l.: s.n.], 2022. P. 593–601. DOI: 10.1109/ICCD56317.2022.00093.

KARYOTIS, Vasileios; KHOUZANI, M.H.R. Chapter 5 - Malware-propagative Markov random fields. In _____. **Malware Diffusion Models for Wireless Complex Networks**. Edição: Vasileios Karyotis e M.H.R. Khouzani. Boston: Morgan Kaufmann, 2016. P. 107–138. ISBN 978-0-12-802714-1. DOI: https://doi.org/10.1016/B978-0-12-802714-1.00015-3. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780128027141000153>.

KELLY, Terence. Programming Workbench: Compressed Sparse Row Format for Representing Graphs. **;login: The USENIX Magazine**, USENIX Association, v. 45, n. 4, 2020. Disponível em: <https://www.usenix.org/publications/login/winter2020/kelly>.

KRAUSE, Alexander et al. Partitioning Strategy Selection for In-Memory Graph Pattern Matching on Multiprocessor Systems. In_____. **Euro-Par 2017: Parallel Processing**. Cham: Springer International Publishing, 2017. P. 149–163. ISBN 978-3-319-64203-1.

KUMAR, Pradeep; HUANG, H. Howie. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. **ACM Trans. Storage**, Association for Computing Machinery, New York, NY, USA, v. 15, n. 4, jan. 2020. ISSN 1553-3077. DOI: 10.1145/3364180. Disponível em: <https://doi.org/10.1145/3364180>.

LAVINSKY, Brandon; ZHANG, Xuechen. PM-Rtree: A Highly-Efficient Crash-Consistent R-tree for Persistent Memory. In: PROCEEDINGS of the 34th International Conference on Scientific and Statistical Database Management. Copenhagen, Denmark: Association for Computing Machinery, 2022. (SSDBM '22). ISBN 9781450396677. DOI: 10.1145/3538712.3538713. Disponível em: <https://doi.org/10.1145/3538712.3538713>.

LEPERS, Baptiste; QUEMA, Vivien; FEDOROVA, Alexandra. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15). Santa Clara, CA: USENIX Association, jul. 2015. P. 277–289. ISBN 978-1-931971-225. Disponível em: <https://www.usenix.org/conference/atc15/technical-session/presentation/lepers>.

LESKOVEC, Jure; KREVL, Andrej. **SNAP Datasets: Stanford Large Network Dataset Collection**. [S.l.: s.n.], 2014. http://snap.stanford.edu/data.

LI, Yuguo et al. A NUMA-aware Key-Value Store for Hybrid Memory Architecture. In: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). [S.l.: s.n.], 2022. P. 1–6. DOI: 10.1109/INFOCOMWKSHPS54753.2022.9798134.

LIU, Gang; CHEN, Leying; CHEN, Shimin. Zen+: a robust NUMA-aware OLTP engine optimized for non-volatile main memory. **VLDB Journal**, v. 32, n. 1, p. 123–148, 2023. DOI: 10.1007/s00778-022-00737-1. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85127641889&doi=10.1007%2fs00778-022-00737-1&partnerID=40&md5=c9ae87d8e4871af2bc8c775fbc04d0f3>.

MEMARZIA, Puya; RAY, Suprio; BHAVSAR, Virendra C. **Toward Efficient In-memory Data Analytics on NUMA Systems**. [S.l.: s.n.], 2020. arXiv: 1908.01860 [cs.DB].

MICHAILIDIS, Theodore; SWANSON, Steven; ZHAO, Jishen. PMShifter: enabling persistent memory fluidness in Linux. In: PROCEEDINGS of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems. Virtual Event, Singapore: Association for Computing Machinery, 2022. (APSys '22), p. 1–8. ISBN 9781450394413. DOI: 10.1145/3546591.3547523. Disponível em: <https://doi-org.ez87.periodicos.capes.gov.br/10.1145/3546591.3547523>.

PAPPAS, Jim. Why Persistent Memory Matters. How Did We Get Here, and What Lies Ahead? In: FLASH Memory Summit. [S.l.: s.n.], 2018.

SALAM, Abdul et al. Future-Based Persistent Spatial Data Structure for NVM-Based Manycore Machines. **IEEE Access**, v. 10, p. 114711–114724, 2022. DOI: 10.1109/ACCESS.2022.3216410.

SCARGALL, Steve. **Programming Persistent Memory - A Comprehensive Guide for Developers**. 1st. [S.l.]: Apress, 2020. ISBN 978-1-4842-4932-1.

THE LINUX KERNEL DEVELOPERS. **DAX - Direct Access for files**. [S.l.: s.n.], 2023. The Linux Kernel Documentation. Disponível em: https://www.kernel.org/doc/html/latest/filesystems/dax.html.

_____. **NUMA Balancing**. [S.l.: s.n.], 2023. The Linux Kernel Documentation. Disponível em: https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_balancing.html.

_____. **Transparent Huge Page Support**. [S.l.: s.n.], 2023. The Linux Kernel Documentation. Disponível em: https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html.

WANG, Rui et al. XPGraph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs. In: 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO). [S.l.: s.n.], 2022. P. 1308–1325. DOI: 10.1109/MICRO56248.2022.00091.

WEILAND, Michèle et al. An Early Evaluation of Intel's Optane DC Persistent Memory Module and Its Impact on High-Performance Scientific Applications. In: PROCEEDINGS of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). [S.l.: s.n.], nov. 2019. P. 1–19.

WEST, Douglas B. **Introduction to Graph Theory**. 2nd. [S.l.]: Prentice Hall, 2001.

WHEATMAN, Brian; XU, Helen. Packed Compressed Sparse Row: A Dynamic Graph Representation. In: 2018 IEEE High Performance extreme Computing Conference (HPEC). [S.l.: s.n.], 2018. P. 1–7. DOI: 10.1109/HPEC.2018.8547566.

XIA, Feng et al. Graph Learning: A Survey. **IEEE Transactions on Artificial Intelligence**, v. 2, n. 2, p. 109–127, 2021. DOI: 10.1109/TAI.2021.3076021.

ZHANG, Xiaoyi et al. A Write-efficient and Consistent Hashing Scheme for Non-Volatile Memory. In: PROCEEDINGS of the 47th International Conference on Parallel Processing. Eugene, OR, USA: Association for Computing Machinery, 2018. (ICPP '18). ISBN 9781450365109. DOI: 10.1145/3225058.3225109. Disponível em: <https://doi.org/10.1145/3225058.3225109>.

ZHU, Guangyu et al. An Empirical Evaluation of NVM-Aware File Systems on Intel Optane DC Persistent Memory Modules. **Electronics**, v. 10, n. 16, 2021. ISSN 2079-9292. DOI: 10.3390/electronics10161977. Disponível em: <https://www.mdpi.com/2079-9292/10/16/1977>.

ZOU, Xiaomin et al. SPHT: A scalable and high-performance hashing scheme for persistent memory. **Software - Practice and Experience**, v. 52, n. 7, p. 1679–1697, 2022. DOI: 10.1002/spe.3083. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85126741547&doi=10.1002%2fspe.3083&partnerID=40&md5=f22147ab3b9bb76aec5aef5fc43d83e0>.

DADOS CURRICULARES

IDENTIFICAÇÃO	
Nome	Lucas Bastelli Spagnol
Nome em citações bibliográficas	Spagnol, Lucas Bastelli; Spagnol, L. B.
Currículo Lattes	`https://lattes.cnpq.br/0642832066621314`
ORCID	`https://orcid.org/0000-0003-2201-0907`
FORMAÇÃO ACADÊMICA	
2019–2022	Bacharelado em Ciências da Computação — Bacharel. Universidade Estadual Paulista "Júlio de Mesquita Filho" (Unesp), Rio Claro.
2022–2025	Mestrado em Ciência da Computação — Mestre. Universidade Estadual Paulista "Júlio de Mesquita Filho" (Unesp), Rio Claro.
PRODUÇÃO BIBLIOGRÁFICA	
SPAGNOL, L. B. *Primeiras experiências com a programação de estruturas de dados persistentes*. In: ERAD-SP 2022. doi:10.5753/eradsp.2022.222240.	
SPAGNOL, L. B. *Programando para memória persistente: dificuldades, armadilhas e desempenho*. In: WSCAD 2022. doi:10.5753/wscad.2022.226384.	
SPAGNOL, L. B. *Investigando fatores de desempenho em dispositivos persistentes*. In: ERAD-SP 2024. doi:10.5753/eradsp.2024.239925.	
SPAGNOL, L. B. *Uma revisão sistemática sobre estruturas de dados em dispositivos persistentes contemporâneos*. In: SSCAD 2024. doi:10.5753/sscad.2024.244737.	
PARTICIPAÇÃO EM EVENTOS CIENTÍFICOS	
ERAD-SP 2022 — *Primeiras experiências com a programação de estruturas de dados persistentes*. doi:10.5753/eradsp.2022.222240.	
WSCAD 2022 — *Programando para memória persistente: dificuldades, armadilhas e desempenho*. doi:10.5753/wscad.2022.226384.	
ERAD-SP 2024 — *Investigando fatores de desempenho em dispositivos persistentes*. doi:10.5753/eradsp.2024.239925.	
SSCAD 2024 — *Uma revisão sistemática sobre estruturas de dados em dispositivos persistentes contemporâneos*. doi:10.5753/sscad.2024.244737.	