Otimizando Estruturas de Grafos em Memória Persistente para Arquiteturas NUMA

Lucas Spagnol[1], **Bruno Honorio**[1], **Otávio Scarparo Souza**[1], **Alexandro Baldassin**[1], **Emilio Francesquini**[2]

[1]Universidade Estadual Paulista (UNESP) – Rio Claro, SP

{lucas.b.spagnol, bruno.honorio, otavio.scarparo-souza, alexrc}@unesp.br

[2]Universidade Federal do ABC (UFABC) – Santo André, SP

e.francesquini@ufabc.edu.br

Abstract. *Estruturas de grafos dinâmicos desempenham papel fundamental em aplicações que demandam processamento eficiente de grandes volumes de dados conectados entre si, como redes sociais e sistemas de recomendação. Este trabalho apresenta uma implementação de grafos dinâmicos em memória persistente para ambientes NUMA, explorando o particionamento round-robin e a afinidade explícita de threads para otimizar o processamento. Os experimentos foram conduzidos com dois conjuntos de dados reais de grande escala extraídos do repositório SNAP (Orkut e LiveJournal), permitindo avaliar o impacto das características topológicas dos grafos nas otimizações propostas. Os resultados experimentais demonstraram ganhos expressivos de desempenho, com speedup de até 2,3× em algoritmos como Connected Components(CC), quando comparado à versão original. Por outro lado, evidenciaram-se limitações da abordagem em grafos de baixa densidade e em algoritmos sensíveis à latência de acesso remoto, como BFS, ressaltando a importância de considerar a topologia do grafo na escolha de estratégias de particionamento.*

1. Introdução

O armazenamento em computadores é tradicionalmente dividido entre memória primária (DRAM) e secundária (HDs e SSDs). A DRAM é rápida, mas volátil e com capacidade limitada, enquanto o armazenamento secundário preserva dados, porém com maior latência e menor largura de banda [Hennessy and Patterson 2017]. Avanços recentes em semicondutores viabilizaram a Memória Persistente (*Persistent Memory* – PM), que combina a velocidade e granularidade de byte da DRAM com a persistência e capacidade de HDs/SSDs [Baldassin et al. 2021]. O uso de PM exige considerar aspectos como o Acesso à Memória Não Uniforme (NUMA), que permite acesso a memórias ligadas a outros processadores, ampliando a capacidade disponível, mas com aumento de latência [Dashti et al. 2013, Memarzia et al. 2019]. Esse efeito, conhecido na DRAM, também impacta a PM [Liu et al. 2023, Chen et al. 2022, Jamil et al. 2023, Jia et al. 2022], tornando essenciais estratégias para mitigar perdas de desempenho. Contudo, há poucas técnicas voltadas especificamente à PM.

O processamento eficiente de grafos tornou-se um pilar para aplicações de larga escala, especialmente em domínios como redes sociais e inteligência artificial [Drozdek 2012, Xia et al. 2021]. Embora os desafios de localidade de dados em arquiteturas NUMA com memória volátil sejam bem documentados, a interseção desses

desafios com Memória Persistente (PM) permanece significativamente menos explorada. Essa lacuna é evidente em soluções de ponta como o DGAP [Islam and Dai 2023], que, apesar de seus avanços, não foram projetadas para mitigar as penalidades de acesso remoto inerentes a servidores multi-soquete. A relevância deste problema é corroborada por nossos experimentos com o algoritmo *PageRank*, que revelaram perdas de desempenho de até 17% em cenários de acesso totalmente remoto, reforçando a necessidade crítica de alinhar o particionamento de dados com a afinidade de *threads*.

Diante desse cenário, o presente artigo ataca essa lacuna ao apresentar uma análise quantitativa do impacto da NUMA e introduzir uma versão otimizada do DGAP. Para validar a proposta, nossa avaliação experimental explora distintas estratégias de particionamento e afinidade em algoritmos de grafos representativos, utilizando grafos do mundo real. O objetivo final é avançar o estado da arte em técnicas de processamento de grafos cientes da topologia de memória no domínio de PM, visando obter maior eficiência computacional. Nossos resultados experimentais demonstram que a estratégia proposta alcança ganhos de desempenho expressivos, com um *speedup* de até 2,3$\times$ em algoritmos como o de *Connected Components*. A análise, no entanto, expõe limitações da abordagem em grafos de baixa densidade e em algoritmos sensíveis à latência, como o *Breadth-First Search* (BFS). Essa descoberta ressalta uma contribuição central do trabalho: a eficácia de uma estratégia de otimização depende intrinsecamente da topologia do grafo. Portanto, nossas conclusões não apenas indicam a necessidade de desenvolver estratégias de balanceamento adaptativas, mas também abrem novas frentes de investigação para otimizações de desempenho em ambientes com múltiplos nós NUMA.

2. Fundamentação Teórica

Esta seção apresenta os conceitos fundamentais para este trabalho. Primeiramente, aborda-se a hierarquia de memória, com ênfase na Memória Persistente (PM) e seus desafios inerentes de consistência e durabilidade. Em seguida, discute-se a arquitetura NUMA, detalhando como o acesso não uniforme à memória exacerba os gargalos de desempenho em sistemas com PM. Por fim, são apresentadas as estruturas de dados e representações de grafos que constituem a base de nossa implementação.

2.1. Hierarquia de Memória e Memória Persistente

A hierarquia de memória convencional busca um equilíbrio entre custo, capacidade e latência, organizando-se em memórias primárias (e.g., DRAM volátil) e secundárias (e.g., SSDs persistentes). A Memória Persistente (PM) emerge como uma camada intermediária, unindo a baixa latência e o endereçamento a byte da DRAM com a não volatilidade do armazenamento secundário. Contudo, a integração da PM em sistemas de alto desempenho introduz alguns desafios que incluem:

1. **Consistência:** Garantir a consistência dos dados diante de falhas é um desafio fundamental, pois operações de escrita interrompidas podem corromper as estruturas salvas na memória. Tipicamente a solução envolve mecanismos de transação, como os oferecidos pela biblioteca PMDK (*Persistent Memory Development Kit*) [Scargall 2020], que asseguram a atomicidade por meio de primitivas de *commit* e *rollback*.
2. **Latência de Escrita e Durabilidade:** A latência de escrita da PM é superior à da DRAM e suas células de memória têm uma durabilidade reduzida. Portanto, a otimização do desempenho e da durabilidade requer a minimização de escritas, usualmente por meio de técnicas como processamento em lote e *logging*.

2.2. Arquitetura NUMA e seu Impacto

Em sistemas multi-soquete, a arquitetura NUMA (*Non-Uniform Memory Access*) organiza a memória em nós locais, cada um diretamente associado a um processador. O acesso de um processador a um nó de memória não local (remoto) incorre em latências significativamente maiores. Esse gargalo, conhecido como *efeito NUMA*, é potencializado em sistemas com PM, que já possuem uma latência base superior à da DRAM. Atentar-se a esse efeito e tentar atenuá-lo é crucial para o desenvolvimento de sistemas com um bom desempenho. Este processo normalmente envolve envolve diversas estratégias como o uso de alocadores de memória cientes de NUMA (*NUMA-aware*), políticas de intercalação de memória (*interleaving*) e de definição de afinidade de *threads* para maximizar o acesso a dados locais [Li et al. 2022].

2.3. Grafos e suas Representações

Grafos, formalmente definidos como um par $G = (V, E)$, são estruturas de dados fundamentais para modelar relações em diversas áreas [West 2001]. Sua adaptação para a PM é um campo de pesquisa ativo [Xia et al. 2021]. A escolha da representação de um grafo impõe um compromisso (*trade-off*) entre uso de memória, eficiência de acesso e flexibilidade para modificações. As principais representações incluem:

- **Lista de Arestas:** Estrutura simples que facilita a inserção de arestas, mas é ineficiente para consultar a vizinhança de um vértice.
- **Matriz de Adjacência:** Permite a verificação de arestas em tempo constante ($O(1)$), mas possui alto custo de memória ($O(V^2)$) para grafos esparsos e é inflexível a modificações estruturais.
- **Lista de Adjacência:** Representação balanceada, eficiente para consultar vizinhos, mas pode incorrer em custos de realocação de memória e ser lenta para buscas de arestas específicas.
- **Compressed Sparse Row (CSR):** Extremamente compacta e eficiente para a análise de grafos estáticos, mas inadequada para grafos dinâmicos, pois qualquer inserção exige a reconstrução parcial ou total da estrutura.

Dentre essas opções, a representação *Compressed Sparse Row* (CSR) se destaca pela sua pegada de memória reduzida, sendo especialmente relevante para o processamento de grafos estáticos em larga escala. Por essa razão, detalhamos sua estrutura a seguir. O formato CSR armazena grafos utilizando dois vetores principais [Kelly 2020]:

1. **Vetor de *Offsets*:** Um vetor de tamanho $|V| + 1$, onde a i-ésima posição armazena o índice de início do bloco de vizinhos do vértice i no vetor de destinos.
2. **Vetor de Arestas:** Armazena de forma contígua os identificadores dos vértices de destino de todas as arestas, agrupados pelo vértice de origem.

A Figura 1 ilustra essa estrutura. O Vetor de *Offsets* indica o intervalo de índices no Vetor de Arestas para cada vértice. Por exemplo, os vizinhos do vértice B (v_1) começam no índice 3 (*offset*[1]) e terminam antes do índice 6 (*offset*[2]), correspondendo aos destinos A, C, D.

O formato CSR apresenta uma dualidade crítica no contexto deste trabalho. Por um lado, sua principal vantagem é ser compacto (pois armazena apenas as arestas existentes) e a localidade de dados (ao manter os vizinhos de um vértice em posições contíguas de memória). Essa característica torna sua persistência trivial: uma vez construída, a estrutura pode ser mapeada diretamente como um arquivo na memória, sem necessidade de

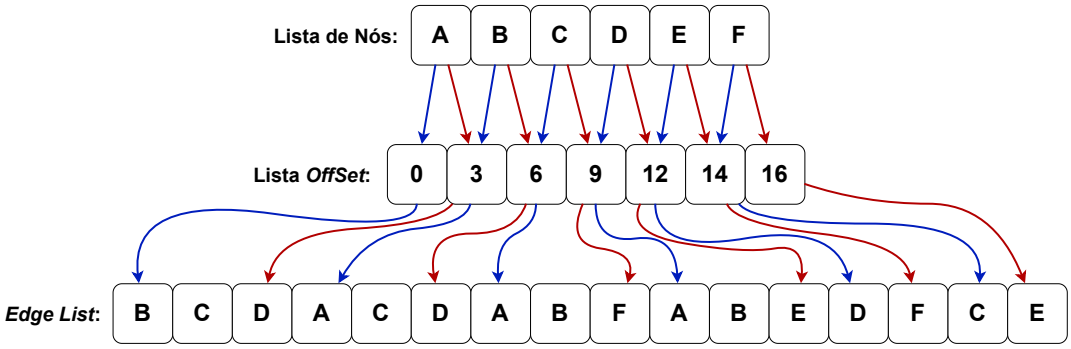


Figura 1. Exemplo de um grafo e sua representação em formato CSR.

análises extras [Kelly 2020]. Por outro lado, sua estrutura estática impõe uma limitação para grafos dinâmicos, pois a inserção de novos vértices ou arestas exige operações custosas de deslocamento de dados e atualização de *offsets* [Wheatman and Xu 2018].

3. Trabalhos Relacionados

A literatura pertinente a este trabalho converge em duas frentes principais: a minimização do efeito NUMA em Memória Persistente (PM) e a especialização de frameworks de processamento de grafos para este novo hardware.

3.1. Lidando com o efeito NUMA em Memória Persistente

Estudos recentes demonstram que o acesso remoto a nós NUMA degrada severamente o desempenho de escrita na PM, um gargalo que se acentua com o aumento de *threads* [Zhu et al. 2021, Chen et al. 2023]. Para minimizar este problema, as abordagens na literatura convergem para duas estratégias principais:

1. **Particionamento e Afinidade de Threads:** Consiste em particionar os dados entre os nós NUMA e fixar a execução das *threads* aos seus nós locais correspondentes, eliminando ou minimizando acessos remotos [Salam et al. 2022, Liu et al. 2023].
2. **Migração Dinâmica de Dados:** Envolve mover páginas de memória dinamicamente para o nó local da *thread* que as acessa com mais frequência, como implementado por frameworks como o Dragonfly [Han et al. 2021, Michailidis et al. 2022].

Contudo, a eficácia dessas estratégias depende de sua integração a estruturas de dados que, por sua vez, sejam otimizadas para as particularidades da PM.

3.2. Frameworks de Grafos para Memória Persistente

No domínio do processamento de grafos, os frameworks para PM evoluíram a partir de precursores projetados para DRAM. Os dois trabalhos que mais se aproximam de nossa proposta são o XPGraph e o DGAP. O XPGraph [Wang et al. 2022] é um *framework* voltado a arquiteturas NUMA que utiliza dois métodos distintos de particionamento de grafos para balancear a distribuição de vértices e arestas entre os nós: o particionamento Round-Robin e o particionamento in/out graph. Ele se destaca por otimizações que consideram a distribuição *power-law* e o *preferential attachment* dos grafos. No entanto, sua arquitetura é baseada em múltiplas estruturas de dados (como listas de arestas para inserções e lista de adjacência para análises), o que exige uma etapa de conversão entre estruturas, introduzindo sobrecarga (*overhead*) de processamento e movimentação de dados.

O DGAP [Islam and Dai 2023], por outro lado, utiliza uma única estrutura de dados central: uma versão mutável do CSR, que permite inserções dinâmicas sem a necessidade de conversões. Essa abordagem simplifica a arquitetura e elimina o *overhead* de sincronização. Contudo, o DGAP, em sua concepção original, não possui suporte para arquiteturas NUMA.

Este trabalho se posiciona exatamente nesta lacuna, propondo estender a arquitetura eficiente e sem conversão do DGAP com otimizações para ambientes NUMA, unindo a simplicidade estrutural do DGAP com as estratégias de desempenho necessárias para hardware moderno, como as utilizadas pelo XPGraph.

4. Otimização do DGAP para Arquiteturas NUMA

Este capítulo apresenta a concepção e implementação de uma versão do framework DGAP otimizada para arquiteturas NUMA. Partindo de um design que não considera a topologia de memória, detalhamos as modificações realizadas para maximizar a localidade de dados e processamento. O foco da otimização foi atenuar o alto custo dos acessos remotos (*cross-NUMA*), um gargalo de desempenho crítico em Memória Persistente (PM), visando acelerar primordialmente as operações de análise de grafos (*graph analytics*).

4.1. Arquitetura Original do DGAP

O framework DGAP foi concebido para conciliar a eficiência de leitura do formato CSR com a capacidade de atualização dinâmica em PM. Sua arquitetura se baseia em uma versão mutável do CSR, aprimorada com o conceito de PMA (Packed Memory Array) [Islam and Dai 2023], que permite inserções e remoções de arestas sem reconstruir todo o grafo.

Para isso, sua arquitetura emprega uma estratégia de memória híbrida preza tanto pelo desempenho quanto pela durabilidade. O *Vertex Array* armazena metadados de acesso frequente como o grau dos vértices e ponteiros para suas listas de arestas e é mantido inteiramente na DRAM para reduzir o impacto da alta latência de escrita da PM. Já o *Edge Array*, que contém a estrutura principal do grafo, reside na PM, onde a consistência e a eficiência das atualizações são garantidas por dois mecanismos de *log* especializados. O primeiro, *Per-Section Edge Log*, funciona como um *buffer* para agrupar inserções de arestas em lote e reduzir a amplificação de escrita (*write amplification*). O segundo, *Per-Thread Undo Log*, oferece um mecanismo de consistência leve para as operações internas de rebalanceamento, assegurando a recuperação em caso de falhas sem o custo de um sistema de transações completo como o da PMDK.

Em síntese, o DGAP original oferece um design robusto para grafos dinâmicos persistentes, mas que não leva em conta a distribuição de memória em nós NUMA.

4.2. Estratégia de Otimização e Desafios

A otimização do DGAP foi guiada pela premissa de que o particionamento manual de dados e a afinidade de processamento superariam os mecanismos automáticos do sistema operacional (como o `AutoNUMA`), que se mostraram ineficazes para workloads em PM. O processo evoluiu em duas etapas:

1. **Abordagem Inicial (Divisão Ingênua):** A primeira tentativa consistiu em uma divisão ingênua do grafo, alocando a primeira metade dos vértices ao nó 0 e a segunda ao nó 1. Essa abordagem resultou em desempenho inferior ao original. A análise revelou um severo desbalanceamento de carga: devido à natureza

power-law dos grafos, a grande maioria das arestas — que representam o trabalho computacional — concentrou-se em um único nó, sobrecarregando-o e mantendo o outro ocioso.

2. **Solução adotada (Distribuição round-robin e afinidade de threads):** Inspirada no XPGraph, a estratégia bem-sucedida foi implementar uma política de distribuição *Round-Robin*, 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. Essa técnica garantiu um excelente balanceamento tanto dos vértices quanto das arestas entre os nós.
   Contudo, a distribuição de dados por si só foi insuficiente. O passo final e crucial foi implementar a afinidade de *threads* (*thread binding*), vinculando cada *thread* ao nó NUMA correspondente ($T \pmod N$). Essa combinação de particionamento de dados e afinidade de processamento garantiu que a maioria das operações ocorresse localmente, minimizando o tráfego entre nós e maximizando o desempenho.

4.3. Otimizações

A principal contribuição de nossa abordagem, em comparação ao XPGraph, reside na simplicidade arquitetural. Enquanto o XPGraph utiliza uma infraestrutura complexa com múltiplos *buffers* e conversões entre estruturas de dados, nossa adaptação preserva a estrutura de dados única e centralizada do DGAP (CSR mutável).

Isso resulta em menor *overhead* de gerenciamento e menor consumo de memória. Embora o XPGraph possa ter vantagens em cenários com altíssimas taxas de atualização, nossa abordagem oferece um equilíbrio superior entre desempenho, simplicidade e eficiência de recursos para *workloads* de análise de grafos, que são predominantes em muitas aplicações do mundo real.

A Figura 2 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.

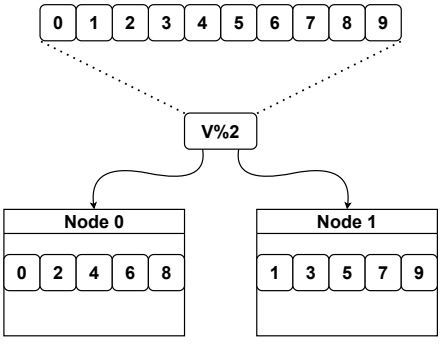


Figura 2. Exemplo de distribuição dos vértices do grafo entre dois nós NUMA utilizando o método Round-Robin.

As modificações propostas foram implementadas em C++ tendo como base o

código original do DGAP. A seleção entre a versão original e a otimizada para NUMA é controlada por *flags* de compilação. Todo o código-fonte está publicamente disponível em um repositório Github[1], com a finalidade de promover a transparência e a reprodutibilidade da pesquisa.

5. Resultados Experimentais

Esta seção avalia o desempenho da nossa proposta de otimização NUMA-aware, denominada DGAP RR (*Round-Robin*). A avaliação compara nossa abordagem com o DGAP original, a tentativa de divisão ingênua (DGAP DM), o XPGraph, GraphOne e uma linha de base em CSR estático. Foram utilizados quatro *benchmarks* padrão sobre dois datasets: Orkut (denso) e LiveJournal (esparso). Os experimentos foram conduzidos em um servidor de dois nós NUMA com 1 TB de Memória Persistente.

5.1. Sistemas Comparados

Para avaliar a eficácia da otimização proposta, o desempenho da nossa abordagem foi comparado com os seguintes sistemas e variantes:

CSR Versão estática do *Compressed Sparse Row* adaptada para PM. Serve como linha de base (*baseline*) para o desempenho máximo em análise de grafos, quantificando o *overhead* de sistemas dinâmicos;

GraphOne *Framework* baseado em DRAM e adaptado para PM [Islam and Dai 2023]. É utilizado para avaliar o impacto da migração de sistemas tradicionais para ambientes de memória persistente;

DGAP A implementação original do *framework* DGAP [Islam and Dai 2023], que não possui otimizações para NUMA. Serve como a base principal para comparação com as nossas modificações;

DGAP DM Nossa primeira tentativa de otimização, com uma divisão ingênua do grafo (metade/metade). É incluída para demonstrar o impacto negativo de uma estratégia de particionamento que não considera a estrutura do grafo;

DGAP RR A solução proposta neste trabalho. É a versão do DGAP modificada com particionamento *Round-Robin* dos vértices e afinidade explícita de *threads* para maximizar a localidade em ambientes NUMA;

XPGraph O competidor de estado da arte para grafos dinâmicos em PM [Wang et al. 2022].Nesta avaliação, foi empregada a versão original, sem otimizações NUMA, devido à indisponibilidade dos *benchmarks* adaptados para NUMA;

5.2. Ambiente Experimental

Em todas as abordagens, 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*. Assim, o *speed-up* S de uma abordagem A é

[1] `https://github.com/otaviosso/na-graph`

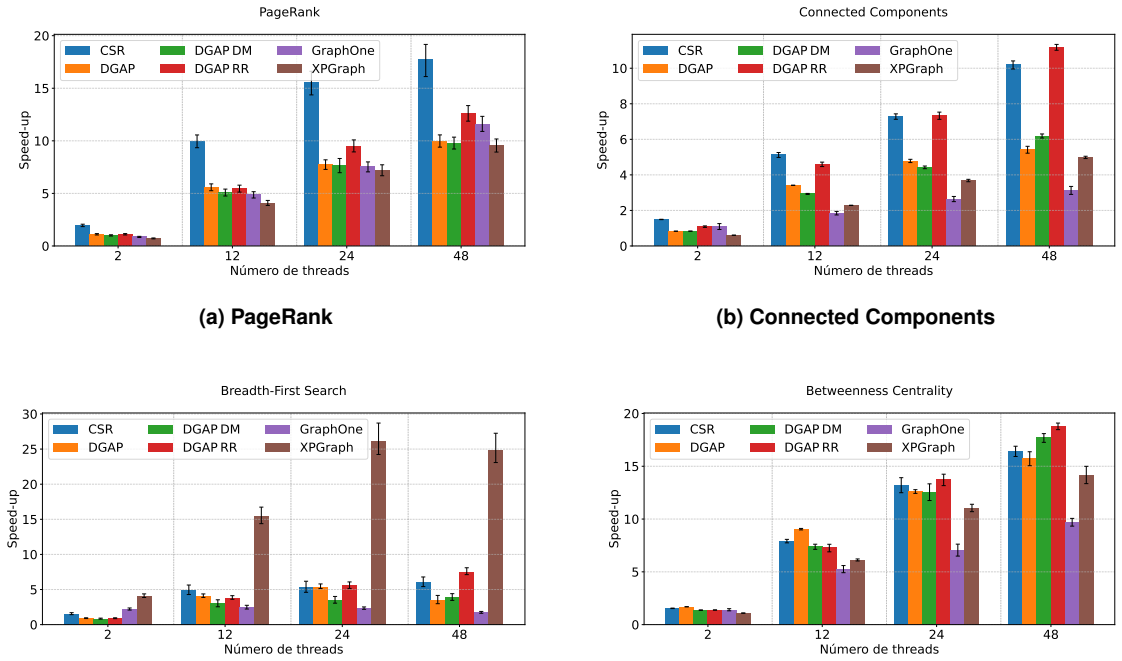
calculado por $S = \frac{T_{\mathrm{CSR,\,1T}}}{T_A}$, onde $T_{\mathrm{CSR,\,1T}}$ é o tempo médio da implementação CSR em execução com 1 *thread*, e T_A é o tempo médio da abordagem A com o respectivo número de *threads* considerado no experimento. Dessa forma, valores $S > 1$ indicam aceleração em 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.

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, com a primeira execução sendo o *warm-up*, assim, removida dos resultados. Para cada conjunto de amostras, calculamos o intervalo de confiança de 95 % pelo método *bootstrap*, utilizando 5 000 reamostragens [Efron and Tibshirani 1994].

5.3. Benchmarks Avaliados

- **PageRank**: Algoritmo clássico de ranqueamento, originalmente desenvolvido para ordenar páginas na web [Brin and 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 and Page 1998, Hagberg et al. 2008];
- **Betweenness Centrality (BC)**: 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**: 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**: 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].



(a) PageRank **(b) Connected Components**

(c) Breadth-First Search **(d) Betweenness Centrality**

Figura 3. Resultados de *speed-up* para os quatro *benchmarks* no *dataset* Orkut.

5.4. Análise no Grafo Denso (Orkut)

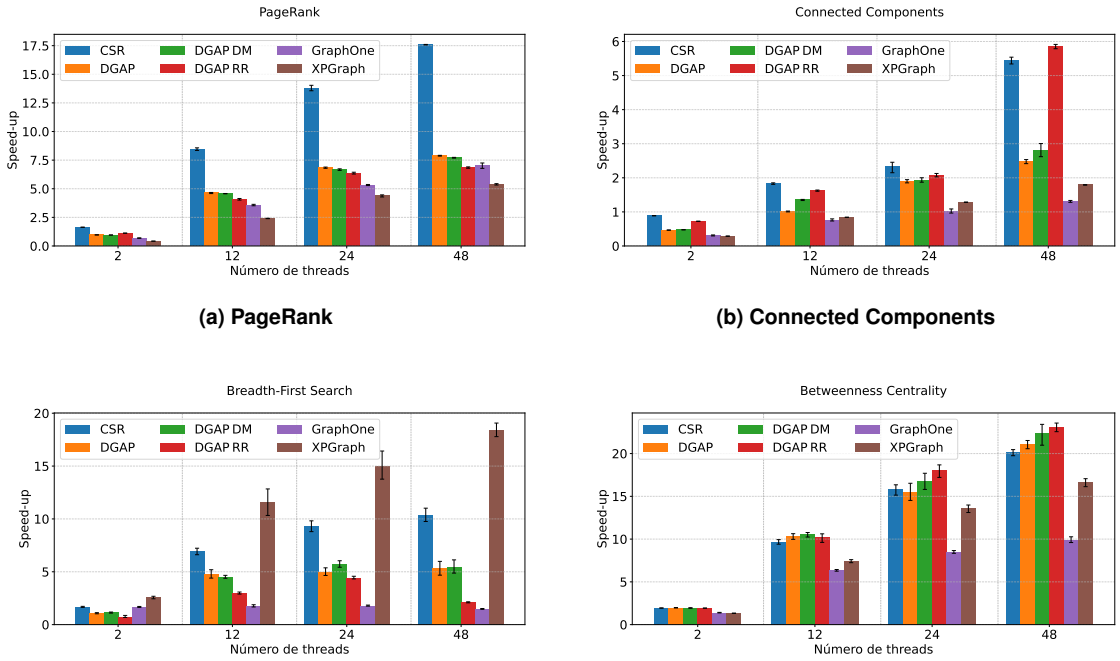
Para o *dataset* Orkut, caracterizado por sua alta densidade, nossa abordagem DGAP RR demonstrou ganhos de desempenho expressivos e consistentes, validando a eficácia da otimização para este tipo de grafo. A Figura 3 consolida os resultados de *speed-up* para os quatro *benchmarks*.

Observa-se que em todos os cenários, o DGAP RR (em vermelho) escala bem com o aumento do número de *threads*. Os ganhos são particularmente notáveis em cenários de alto paralelismo (48 threads), onde o efeito NUMA é mais crítico. Conforme visto na Figura 3(b), o ganho mais expressivo ocorreu no Connected Components, onde o DGAP RR foi aproximadamente 2,19× mais rápido que o DGAP original, superando até a linha de base CSR. Resultados significativos também são vistos para o Betweenness Centrality (Figura 3(d)) e PageRank (Figura 3(a)). No BFS (Figura 3(c)), 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, já a nossa abordagem dobra o desempenho do DGAP original.

5.5. Análise no Grafo Esparso (LiveJournal)

Os resultados com o *dataset* LiveJournal (Figura 4) revelam uma nuance crucial: a eficácia da otimização NUMA é dependente do algoritmo e da estrutura do grafo. Para algoritmos com maior carga computacional por acesso, como *Connected Components* (Figura 4(b)) e *Betweenness Centrality* (Figura 4(d)), o DGAP RR continuou a ser a melhor abordagem, chegando a ser 2,3 vezes mais rápido que o DGAP original em CC.

Em contraste, para o BFS (Figura 4(c)), um algoritmo sensível à latência, a distribuição dos dados entre os nós foi prejudicial. Em um grafo esparso, o custo da latência para um acesso remoto não é amortizado pelo baixo volume de trabalho, resultando em degradação de desempenho. Isso evidencia um trade-off fundamental entre



(a) PageRank **(b) Connected Components**

(c) Breadth-First Search **(d) Betweenness Centrality**

Figura 4. Resultados de *speed-up* para os quatro benchmarks no *dataset* LiveJournal.

paralelismo e latência.

5.6. Conclusão dos Resultados

Em síntese, os resultados experimentais, visualmente consolidados nas Figuras 3 e 4, validam nossa abordagem. A otimização NUMA é altamente eficaz para grafos densos e para algoritmos sensíveis à largura de banda em grafos esparsos. Contudo, para algoritmos sensíveis à latência em grafos esparsos, a distribuição de dados pode ser contraproducente. Esta análise, apoiada pelos tempos de execução absolutos (omitidos aqui por brevidade), confirma a importância de considerar tanto a topologia do grafo quanto a natureza do algoritmo ao aplicar otimizações NUMA.

6. Conclusão

Este trabalho apresentou e avaliou uma otimização NUMA-aware para o *framework* de grafos em Memória Persistente DGAP. A principal contribuição foi demonstrar, por meio de uma avaliação experimental rigorosa, que a combinação de particionamento Round-Robin com afinidade de *threads* resulta em ganhos de desempenho expressivos — com acelerações superiores a 2x — para algoritmos sensíveis à largura de banda e grafos de alta densidade.

Como contribuição adicional desta pesquisa, fizemos a caracterização de um *trade-off* fundamental: a mesma estratégia que beneficia *workloads* com alta localidade pode degradar o desempenho em grafos esparsos ou algoritmos limitados pela latência de acessos remotos. Este resultado evidencia que a otimização de sistemas para PM em hardware NUMA não admite uma solução universal, exigindo um co-design que considere simultaneamente a topologia do hardware e as características da carga de trabalho.

As conclusões deste estudo abrem diversas possibilidades para trabalhos futuros. A mais imediata é o desenvolvimento de políticas de particionamento e afinidade adaptati-

vas, capazes de se ajustar dinamicamente à estrutura do grafo e ao algoritmo em execução. Adicionalmente, a extensão da otimização para permitir cargas de trabalho dinâmicas (com inserções e remoções de arestas) e a validação da abordagem em uma gama mais ampla de *datasets* e arquiteturas de hardware representam caminhos promissores para consolidar e generalizar os resultados aqui apresentados.

Agradecimentos. Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos nº 2023/04969-8, 2023/04971-2, 2018/15519-5 e 2024/13771-0 pelo apoio a este trabalho.

Referências

Baldassin, A., Barreto, J., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7).

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30:107–117.

Chen, Y., Qiu, K., Chen, L., Jia, H., Zhang, Y., Xiao, L., and Liu, L. (2022). Smart scheduler: an adaptive nvm-aware thread scheduling approach on numa systems. *CCF Transactions on High Performance Computing*, 4(4):394 – 406.

Chen, Z., Che, W., Hu, D., He, X., Sun, J., and Chen, H. (2023). On the performance intricacies of persistent memory aware storage engines. *IEEE Transactions on Knowledge and Data Engineering*, 35(10):10365–10382.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT press, Cambridge, MA, USA, 3rd edition.

Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. (2013). Traffic management: a holistic approach to memory placement on numa systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 381–394.

Drozdek, A. (2012). *Data Structures and Algorithms in C++*. Cengage Learning.

Efron, B. and Tibshirani, R. J. (1994). *An Introduction to the Bootstrap*. Chapman & Hall/CRC.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41.

Hagberg, A. A., Schult, D. A., and Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15.

Han, J., Byun, H., Kwon, H., Park, S., and Kim, Y. (2021). Is data migration evil in the nvm file system? In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 26–31.

Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition.

Islam, A. A. R. and Dai, D. (2023). Dgap: Efficient dynamic graph analysis on persistent memory. In *Proc. of the SC'23*.

Jamil, S., Salam, A., Khan, A., Burgstaller, B., Park, S.-S., and Kim, Y. (2023). Scalable numa-aware persistent B+-tree for non-volatile memory devices. *Cluster Computing*, 26(5):2865 – 2881.

Jia, W., Jiang, D., and Xiong, J. (2022). Napfs: A high-performance numa-aware pm file system. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 593–601.

Kelly, T. (2020). Programming workbench: Compressed sparse row format for representing graphs. *;login: The USENIX Magazine*, 45(4).

Li, Y., Tan, S., Wang, Z., and Li, D. (2022). A numa-aware key-value store for hybrid memory architecture. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6.

Liu, G., Chen, L., and Chen, S. (2023). Zen+: a robust numa-aware oltp engine optimized for non-volatile main memory. *VLDB Journal*, 32(1):123 – 148.

Memarzia, P., Ray, S., and Bhavsar, V. C. (2019). Toward efficient in-memory data analytics on NUMA systems. *CoRR*, abs/1908.01860.

Michailidis, T., Swanson, S., and Zhao, J. (2022). PMShifter: enabling persistent memory fluidness in Linux. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 1–8.

Salam, A., Jamil, S., Jung, S., Park, S.-S., and Kim, Y. (2022). Future-based persistent spatial data structure for nvm-based manycore machines. *IEEE Access*, 10:114711–114724.

Scargall, S. (2020). *Programming Persistent Memory - A Comprehensive Guide for Developers*. Apress, 1st edition.

Wang, R., He, S., Zong, W., Li, Y., and Xu, Y. (2022). Xpgraph: Xpline-friendly persistent memory graph stores for large-scale evolving graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1308–1325.

West, D. B. (2001). *Introduction to Graph Theory*. Prentice Hall, 2nd edition.

Wheatman, B. and Xu, H. (2018). Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7.

Xia, F., Sun, K., Yu, S., Aziz, A., Wan, L., Pan, S., and Liu, H. (2021). Graph learning: A survey. *IEEE Transactions on Artificial Intelligence*, 2(2):109–127.

Zhu, G., Han, J., Lee, S., and Son, Y. (2021). An empirical evaluation of nvm-aware file systems on intel optane dc persistent memory modules. *Electronics*, 10(16).