NPGraph: An Efficient Graph Computing Model in NUMA-Based Persistent Memory Systems

Baoke Li[1,2], Cong Cao[1], Fangfang Yuan[1], Yuling Yang[1,2], Majing Su[3], Yanbing Liu[1(✉)], and Jianhui Fu[4]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China
{libaoke,caocong,yangyuling,yuanfangfang,liuyanbing}@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100085, China
[3] The 6th Research Institute of China Electronic Corporations, Beijing, China
sumj@ncse.com.cn
[4] Shandong Institutes of Industrial Technology, Jinan, China

Abstract. The massive volume and the inherent imbalance of graphs are inevitable challenges for efficient graph computing, primarily due to the limited capacity of main memory (DRAM). Fortunately, a promising solution has emerged in the form of hybrid memory systems (HMS) which combine DRAM and persistent memory (PMEM) to enable data-centric graph computing. However, directly transitioning existing DRAM-based models to HMS can lead to inefficiency issues, especially when crossing Non-Uniform Memory Access (NUMA) nodes. In this paper, we present NPGraph, a novel approach that fully exploits the advantages of HMS for in-memory graph computing models. The main contributions of NPGraph lie in three aspects. Firstly, a dual-block graph representation strategy is devised to accelerate the process of subgraph construction. By utilizing data layering, it fully utilizes the storage architecture of HMS and optimizes the data access process. Secondly, an adaptive push-pull update strategy is proposed to optimize the message-updating process. With data-driven algorithms, it dynamically migrates subgraphs which are used in future iterations. Thirdly, the effectiveness of NPGraph is evaluated on five public graph data sets. Our model can improve the temporal locality and the spatial locality of graph computing concurrently. Extensive evaluation results show that NPGraph outperforms state-of-the-art graph computing models by 21.67%–32.03%.

Keywords: Graph computing · Adaptive updating strategy · Data-driven algorithms · Hybrid memory · NUMA

1 Introduction

In recent years, with the rapid development of artificial intelligence, graph computing has received wide attention. It plays significant roles in a spectrum of

H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 562, pp. 205–222, 2024.
https://doi.org/10.1007/978-3-031-54528-3_12

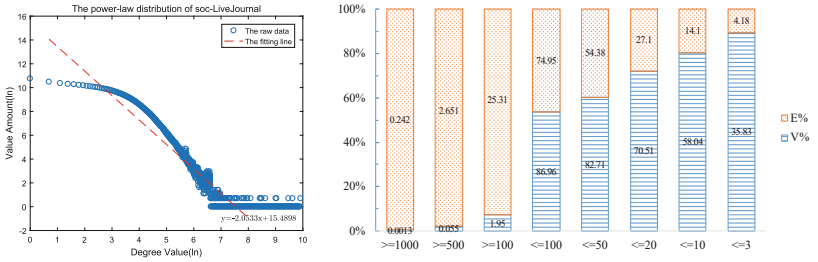

Fig. 1. The power-law analysis and degree distribution in soc-LiveJournal.

fields, ranging from relationship analysis and product recommendation to fraud detection. All these applications benefit from the advantages of graph computing: the flexibility of data modeling, the feasibility of pattern mining, and the visibility of association analysis.

However, with the rapid development of the internet and digital technology, the volume of graph data has grown exponentially which leads to an enormous memory footprint. For example, the foundation model GPT-3 has more than 175 billion parameters in deep learning networks. In addition to the enormous volume, graphs frequently display imbalanced distribution. In most cases, the degree distribution and vertex count satisfy the linear fitting relationship: $ln(Vertex) = ln(c) - rln(Degree)$. When r approaches 2, the graph displays a power-law characteristic. Figure 1 illustrates that in soc-LiveJournal, $r = 2.05$. In particular, 25% of the edges are connected to the top 2% of vertices. However, just 4% of the edges are connected to the bottom 36% of vertices. The inherent imbalanced characteristic can lead to load imbalance and parallel inefficiency.

To process the inherent challenges, many in-memory systems (e.g., Pregel [4], GraphLab [5] and HyVE [16]) provide large aggregated memory to avoid the overhead of disk I/O [2,19,28]. Some distributed in-memory systems like GraphX [6], Gemini [8,10] parallel process subgraphs to improve the scalability of the single-machine systems. As a kind of precious storage resource, DRAM is very expensive. Thus, its high price and low cost performance have seriously hampered the use of in-memory systems. NUMA-based systems such as Ligra [7] and GraphOne [9] are promising solutions for developing parallel graph computing on multi-core systems. Although the performance of these models can meet some basic demands, DRAM also must be large enough to accommodate all the graph data, which severely restricts the use of models for in-memory graph computation.

Recently, with the release of Intel Optane Persistent Memory, the research on persistent memory (PMEM) has made significant progress [11,20,21]. The new memory devices and traditional DRAM compose the hybrid memory systems (HMS) which are feasible solutions for large-scale data-centric applications [30]. Therefore, PMEM can be used in NUMA-based graph computing models to reduce the amount of DRAM and improve the scalability of in-memory graph computing systems.

However, the phenomenon of incommensurate scaling has always existed in computer systems. That is to say, there are definitely performance gaps between DRAM and PMEM. Compared to DRAM, PMEM shows low bandwidth and high read/write latency, as shown in Table 1. These performance gaps mean that the DRAM-based graph computing models might not be optimized in HMS [16, 17, 29]. As a result, properly using the benefits of NUMA-based graph computing models in HMS becomes critical.

Table 1. The features of PMEM compared to DRAM and SATA SSD.

Device	Operation	Bandwidth	Latency	Standby Power
DRAM	Read	14.5 GB/s	81 ns	Fresh Power
	Write	14.5 GB/s	86 ns	
PMEM	Read	7.45 GB/s	170 ns	Zero Power
	Write	2.25 GB/s	320 ns	
SATA SSD	Read	560 MB/s	10–100 µs	Zero Power
	Write	510 MB/s	10–100 µs	

In this paper, we propose NPGraph which is a novel and effective NUMA-based graph computing model specially developed in HMS. It utilizes a dual-block graph representation strategy to support the adaptive push-pull strategy. During the iteration process, it designs data-driven algorithms and a dynamic data migration strategy to optimize the overall performance. To the best of our knowledge, this is the first work to optimize the process of subgraph construction and the message updating model simultaneously. To summarize, the following are the primary contributions of our work:

- NPGraph adopts a lightweight compressed dual-block graph representation strategy to optimize the process of subgraph construction. Specifically, it separately stores the out-blocks and in-blocks of subgraphs in two NUMA nodes. By utilizing data layering in HMS, it restricts data access to specific out-blocks or in-blocks for different updating models.
- NPGraph provides an adaptive push-pull updating strategy to accommodate different data-driven algorithms. According to the activity of subgraphs, it adaptively selects the optimal updating strategy. Then, it dynamically migrates subgraphs which are used in future iterations.
- The effectiveness of NPGraph is thoroughly evaluated on 5 real-world graphs. It can improve the temporal locality and the spatial locality of graph computing at the same time. Extensive experimental results show that NPGraph outperforms state-of-the-art graph computing models by 21.67%–32.03%.

2 Background and Motivation

In this section, we first introduce the performance gaps between PMEM and traditional DRAM. Then, inspired by GraphChi [1], we obtain the optimization

process of subgraph construction. Finally, we present prominent features of the adaptive push-pull model and the data-driven algorithms. All of these motivate us to propose an efficient graph computing model: NPGraph.

2.1 Incommensurate Scaling in HMS

Most real-world graphs tend to show skewed distributions. As shown in Fig. 1, the degree distribution of vertices differs significantly. A little number of hub-vertices connect more than millions of edges. In order to simplify the processing of complicated graphs, many models(e.g., XPGraph [12], NVMGraph [13] and EPGraph [30]) separate the entire graph into numerous subgraphs and load these subgraphs into DRAM and PMEM.

However, the performance gaps between DRAM and PMEM in terms of bandwidth and latency are quite significant, as shown in Table 1. DRAM is more than 3 times faster than PMEM in random reading and writing. In contrast to that, PMEM has a latency that is more than 3 times that of DRAM. Significantly, between NUMA nodes, remote DRAM access is faster than local PMEM access [12]. Therefore, the efficiency of graph data access is concentrated during the iterative graph processing in HMS.

Evidently, the performance differences between DRAM and PMEM might cause the overall performance of typical in-memory systems to deteriorate. [30]. Thus, an effective storage strategy should thus be taken into account in HMS. Furthermore, graph algorithm execution time is related to the access efficiency of active vertices. However, the number of active vertices of different subgraphs in different iterations may vary greatly [30] which could result in the asymmetric convergence phenomena. It inspires us to design an efficient graph representation strategy and the data layering strategy in NUMA-based HMS.

2.2 Analysis of Subgraph Construction

Subgraph construction is an important process in graph computing. Such as GraphChi [1], it divides the vertices into disjoint intervals and breaks the edge list into tiny blocks. For a given vertex interval, it exploits the parallel sliding windows (PSW) to process all intervals and blocks. As shown in Fig. 2, during interval i, PSW loads the i-th in-block and traverses the i-th segments of all the other in-block to construct subgraph i. Obviously, the subgraph construction phase significantly degrades the overall performance.

If storing both in-blocks and out-blocks, there is no need to traverse all the in-blocks during the subgraph construction phase. That is to say, the construction subgraph i only needs the in-block i and the out-block i, which is undoubtedly efficient. Although this format increases storage space, it is common to trade space consumption for time efficiency, especially for PMEM.

In view of the fact that remote memory access is faster than local PMEM access when crossing NUMA nodes, it's wise to translate local PMEM data access to remote memory access. Therefore, separately storing the out-blocks and in-blocks of subgraphs between different NUMA nodes may be beneficial to subgraph construction.

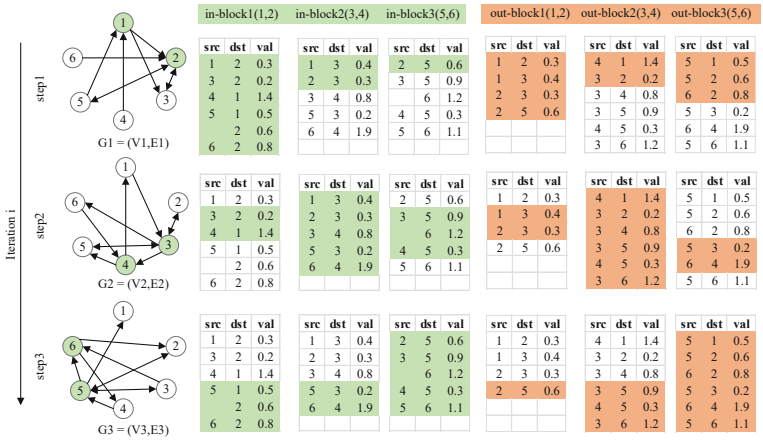

Fig. 2. Subgraph construction of PSW in GraphChi.

2.3 Push-Pull Model and Data-Driven Algorithms

The push and pull updating models are widely employed by large-scale graph processing systems, such as GraphLab [5], Ligra [7] and Gemini [8]. As shown in Fig. 3, in the push model, each vertex scatters (writes) the changes to its neighbors through its outgoing edges. Conversely, in the pull model, each vertex gathers (reads) information from its incoming neighbors and then updates its own value with the gathered information. Obviously, the number of active vertices and edges determines which scenarios the push and pull updating models are appropriate for. The push model, in particular, is appropriate for sparse active edge sets since it only traverses the outgoing edges of active vertices that have modified their values. The pull model, on the reverse hand, supports dense active edge sets since it eliminates data races during the message passing process.

However, many existing graph computing systems either use the push model or the pull model in HMS [9,12,16]. Furthermore, the push model for these systems is based on all vertices rather than the active vertices, as they put a higher priority on graph storage than graph computing efficiency.

Actually, the adaptive push-pull model works well in HMS. When running algorithms with sparse active edge sets, the push model enables selective data access that only traverses the active edges. It fully avoids the access of useless data. When running algorithms with dense active edge sets, the pull model sequentially accesses the edges of all vertices. It overcomes the challenges of random accesses and enables efficient parallelism.

In addition, based on work activation, graph algorithms can be classified into two categories: topology-driven and data-driven. For a topology-driven algorithm, all the vertices need to be processed in each iteration. In contrast, vertices in a data-driven algorithm are dynamically activated by their neighbors, meaning that user-defined functions determine whether a vertice is active or inactive. Data-driven algorithms enables developers to focus more on “hot vertices” in a

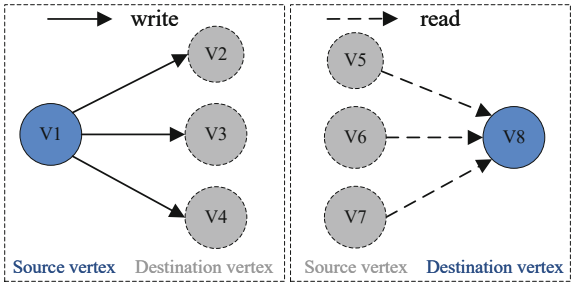

Fig. 3. The push and pull models.

graph that require more frequent updates. As a result, in many cases, data-driven algorithms can outperform topology-driven algorithms. To provide a comprehensive analysis of the data-driven push and pull models, let's take the PageRank algorithm as an example.

Algorithm 1 Push-based PageRank

Input: $G = (V, E)$, α, ϵ.
Output: PageRank PR.
1: Initialize $PR = (1-\alpha)\mathbf{e}$, $\mathbf{r} = \mathbf{0}$;
2: **for** (each $v \in V$) **do**
3: **for** (each $w \in S_v$) **do**
4: $r_v = r_v + 1/|D_w|$;
5: **end for**
6: $r_v = (1-\alpha) + \alpha \times r_v$;
7: $worklist.push(v)$;
8: **end for**
9: **while** (!$worklist.isEmpty()$) **do**
10: $v = worklist.pop()$;
11: $pr_v^{next} = pr_v + r_v$;
12: **for** (each $w \in D_v$) **do**
13: $r_w^{old} = r_w$;
14: $r_w = r_w + \alpha \times r_v/|D_v|$;
15: **if** $(r_w > \epsilon)\&(r_w^{old} < \epsilon)$ **then**
16: $worklist.push(w)$;
17: **end if**
18: **end for**
19: $r_w = 0$;
20: **end while**

Algorithm 2 Pull-based PageRank

Input: $G = (V, E)$, α, ϵ.
Output: PageRank PR.
1: Initialize $PR = (1-\alpha)\mathbf{e}$;
2: //All the vertices are active;
3: **for** (each $v \in V$) **do**
4: $worklist.push(v)$;
5: **end for**
6: **while** (!$worklist.isEmpty()$) **do**
7: $v = worklist.pop()$;
8: $pr_v^{next} = \alpha \times \Sigma_{w \in S_v} \frac{pr_w}{|D_w|} + (1-\alpha)$;
9: //update pr_v;
10: **if** $(|pr_v^{next} - pr_v| \geq \epsilon)$ **then**
11: $pr_v = pr_v^{next}$;
12: **for** (each $w \in D_v$) **do**
13: //Add destination vertices;
14: **if** $(|pr_v^{next} - pr_v| \geq \epsilon)$ **then**
15: $w \notin worklist$;
16: $worklist.push(w)$;
17: **end if**
18: **end for**
19: **end if**
20: **end while**

Algorithm 1 shows the push-based data-driven PageRank. Given a graph $G = (V, E)$, with vertex set V and edge set E, let α denote the damping ratio and ϵ denote the residual. Also, let us define S_v to be the set of incoming vertices of v and D_v to be the set of destination vertices of v. Initially, the worklist is set to be the entire vertex set V. An active vertex v updates its own value pr_v and only pushes r_v to destination vertices D_v. That is to say, the pr_v^{next} of vertex v is equivalent to the pr_v and its r_v, as seen in lines 9 through 20. Push-based algorithms invoke more frequent updates, which might be helpful to

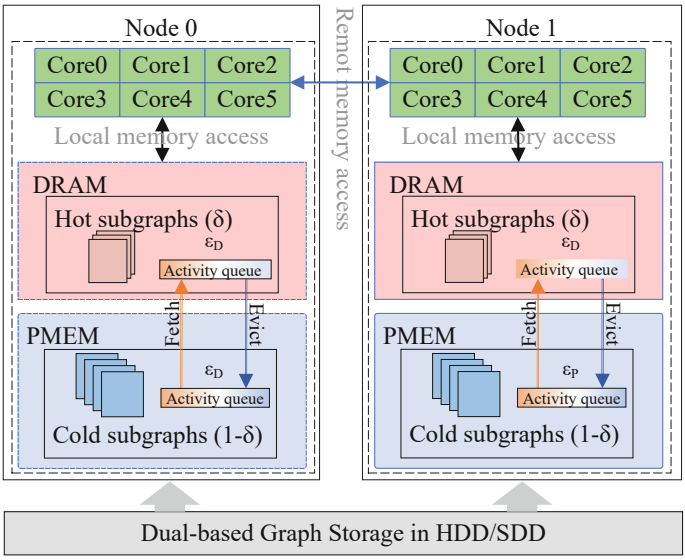

Fig. 4. The architecture of NPGraph.

achieve faster information propagation over the network. However, compared to pull-based algorithms, push-based algorithms can be more costly in the sense that they require more write operations.

Algorithm 2 shows the pull-based data-driven PageRank. The algorithm proceeds by picking a vertex v from the worklist, computing the v's PageRank value pr_v^{next}, and adding its destination vertices to the worklist. Indeed, when pr_v^{next} is updated, each of its outgoing vertex' residual r_v is added $\alpha \times r_v/|D_v|$, as shown between lines 6 and 20. Thus, it guarantees work efficiency by concentrating on a more active list of vertices.

In many cases, the benefit of filtering active vertices outweighs the overhead of residual computations. For graph algorithms, the execution order of active vertices is crucial [30]. For example, in push-based PageRank, whenever a vertex v has a new residual r_v, and then its pr_v is updated. The overall residual r_v is reduced by $(1-\alpha) \times r_v$. This suggests that if the large residual vertices are processed first, the algorithm might converge faster.

3 Model Design

As shown in Fig. 4, NPGraph is a novel and effective graph computing model specially developed in NUMA-based HMS. The specific implementation of NPGraph is introduced in this section.

3.1 Overview of NPGraph

NPGraph aims to improve the efficiency of large-scale graph processing in HMS. It adopts a lightweight compressed dual-block representation to organize graphs.

What's more, it provides an adaptive push-pull updating strategy across NUMA nodes to accommodate different data-driven algorithms.

Dual-Block Graph Representation. Like GraphChi [1], NPGraph divides graph vertices and corresponding edges into tiny intervals and blocks. But it adopts a dual-block graph representation strategy in forward and backward manner, respectively. Although the storage space has increased, it can support push-pull hybrid updating models effectively. It fully utilizes the random data access function of DRAM and the large capacity feature of PMEM in HMS. By translating local PMEM data access to remote memory access, it improves the data access efficiency to a certain extent.

Adaptive Push-Pull Strategy. As mentioned above, the forward manner of in-memory graph structures is suitable for the push updating model, and the backward manner is instrumental in the pull updating model. NPGraph adopts a hybrid push-pull strategy to dynamically adjust the updating model. Inspired by EPGraph [30], it fetches and evicts blocks in HMS. What's more, it proposes a dynamic data migration strategy between DRAM and PMEM. To a certain extent, it improves the temporal locality and the spatial locality of graph computing in HMS.

3.2 Dual-Block Graph Representation

Like some existing systems (e.g., GraphChi [1], Ligra [7] and Gemini [8]), NPGraph partitions graph $G = (V, E)$ into P subgraphs: $G_1 \sim G_p$. That is, the vertex set V and edge set E are split into P disjoint intervals and blocks: $V_1 \sim V_p$ and $E_1 \sim E_p$. Each subgraph $G_i = (V_i, E_i)$ is associated with both the forward and backward manners, as shown in Fig. 5. The two manners store the out-edges and in-edges of vertex sets in each subgraph, respectively.

In-memory graph structure data includes graph structure data, attribute data and status data. Specifically, graph structure data includes vertices and corresponding out-edges or in-edges. Graph attribute data is the value of vertices. Graph status data indicates which vertices have been accessed or the values have been updated. During the implementation of dual-block graph representation, *Row* and *Col* represent the vertices and connected edges, as shown in Fig. 5. In order to reduce data storage space, graph structure data are generally based on CSR compression format. In addition, to facilitate the execution of graph algorithms, the in-memory status data and attribute data have to be built. They are separately marked as S_{curr}, S_{next}, D_{curr} and D_{next}.

As shown in Fig. 5(a), the graph G with six vertices and thirteen edges is partitioned into $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$. $V_0 = \{0, 1, 2\}$ and $V_1 = \{3, 4, 5\}$. Its forward manner and backward manner are organized as Fig. 5(b) and (c). Execution flows start with active vertices to travel and update their outgoing neighbors or themselves. In the forward manner, there are two execution flows indicated by orange and blue arrows, separately. Specifically, during one iterative processing, due to $S_{curr}[0]$=1 and $S_{curr}[5]$=1, v_0 and v_5 are active vertices. The vertex v_0 has out-edges $E_0 = Col[Row[0], Row[1]) = \{(0, 1), (0, 2)\}$. Another

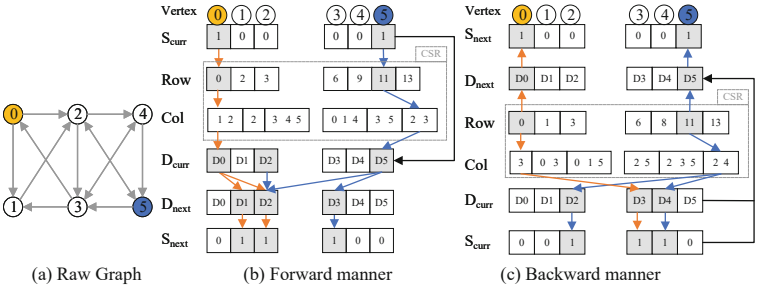

Fig. 5. Forward and backward manners of in-memory graph structures.

vertex v_5 has out-edges $E_5 = Col[Row[5], sizeof(Row)) = \{(5,2),(5,3)\}$. In the push model, the corresponding $D_{next}[1,2,3]$ and $S_{next}[1,2,3]$ of v_1, v_2 and v_3 are updated during the subsequent execution flows.

However, in the backward manner, execution flows need to update vertices v_0 and v_5 by gathering the information of their in-edge neighbors. v_0 needs to gather information from v_3. According to $D_{curr}[3]$ and $S_{curr}[3]$, v_3 updates its $D_{next}[0]$ and $S_{next}[0]$. Similarly, to v_5, it needs to gather information from v_2 and v_4, then update its $D_{next}[5]$ and $S_{next}[5]$.

Inspired by the incommensurate scaling of HMS, we analyze the data access rate of DRAM in HMS [30]. According to the execution flow of the in-memory graph structures mentioned in Fig. 5, graph status data and attribute data (S_{curr}, S_{next}, D_{curr} and D_{next}) should be placed in DRAM. Based on sparsity and density, graph structure data (Row and Col) are placed in DRAM and PMEM selectively. Based on the dual-block representation and layered placement, NPGraph may get better performance in HMS. In addition, due to the fact that the remote memory access is faster than local PMEM access, NPGraph converts local PMEM data access to remote memory access when crossing NUMA nodes. During the programming process, two NUMA nodes respectively store the forward and backward manners graph data. We will evaluate the efficiency of NPGraph in Sect. 4.

3.3 Adaptive Push-Pull Strategy

Due to the unequal distribution of graph data [5], the phenomenon of asymmetric convergence is common in graph computing. That is the reason sparse subgraphs usually converge rapidly and dense subgraphs converge gradually [15]. As mentioned in Sect. 2, sparse and dense subgraphs should be handled by push and pull models individually. In addition, graph algorithms' execution time is proportional to the memory access of active edges and vertices [30]. As a result, the run-time data migration strategy in HMS should be taken into account.

Algorithm 3 shows the computation procedure of the adaptive push-pull update strategy during one iteration. As mentioned in Algorithm 1 and 2, we

Algorithm 3. Adaptive Push-Pull Update Strategy

```
Initialize parameter: sum, ε, D_v, S_v, k;
/*Choose the optimal update model*/
for (each v ∈ V_i) do
    sum[i] = sum[i] + d_v;
    ε[i] = ε[i] + S_curr[v] × d_v;
end for
ε[i] = ε[i]/sum[i];
model = selectModel(ε[i], θ);
if (model = Push) then
    PushModel(V_i, E_i);//Algorithm 4;
else
    wardPullModel(V_i, E_i);//Algorithm 5;
end if
/ * DataMigrate * /
for (i = 1 : P) do
    if (G_i ∈ PMEM) then
        Descending ε_PMEM[i];
    else
        Ascending ε_DRAM[i];
    end if
end for
for (i = 1 : k) do
    /*use memcpy();*/
    MigrateData(G_DRAM[i], G_PMEM[i]);
end for
```

also maintain two copies of vertex values for each subgraph G_i: the source vertex set S_v and the destination vertex set D_v. S_v stores the vertex values of the previous iteration, serving as the source vertices. Similarly, D_v stores the vertex values of the current iteration, serving as the destination vertices. $selectModel()$ is a threshold function which is based on $\varepsilon[i]$ selecting θ. When $\varepsilon[i]$ is less than θ, it returns $Push$. Contrary, it returns $Pull$. For each subgraph, NPGraph adaptively selects the push or pull model to accommodate different graph computing tasks. The selection between the push and pull models is based on the number of active vertices and the data access performance prediction method.

Algorithm 4 shows the execution of the data-driven push model. It utilizes the corresponding out-index of the forward manner to process the out-edges of a vertex interval V_i. As soon as vertex v is in $worklist()$, it traverses the out-going edges and pushes the updates to their out-neighbors D_v with a update function defined by the researchers. In this process, it reads vertex values from D_{curr} and writes updates to D_{next}. If its destination vertices are activated, they are added to the *worklist* and will be scheduled in the next iteration, such as the push-based PageRank in Algorithm 1. Algorithm 5 shows the execution of the data-driven pull model. Similarly, it processes the in-edges of a vertex interval V_i by the corresponding in-index of the backward manner.

Algorithm 4 Push Model

```
1: /* for each active vertex v;*/
2: for (v ∈ worklist & V_i) do
3:    for (each edge e ∈ E_i) do
4:       D_v = e.dst;
5:    end for
6:    /*run user defined algorithm;*/
7:    if (Function(v, D_v) then
8:       worklist.add(D_v)
9:    end if
10: end for
```

Algorithm 5 Pull Model

```
1: for (v ∈ V_i) do
2:    for (each edge e ∈ E_i) do
3:       S_v = e.src;
4:    end for
5:    if (S_currv == 1) then
6:       if (Function(v, S_v) then
7:          worklist.add(S_v)
8:       end if
9:    end if
10: end for
```

As mentioned in Sect. 2.2, $S_{curr}[v]$, D_v and S_v represent the activities of different subgraphs. So, based on the relative activities, NPGraph executes evicting and fetching operations in HMS. Lines 14 to 23 of Algorithm 3 present the procedure of the dynamic data migration strategy. During the iterative process of graph algorithms like PageRank and WCC, the migration strategy calculates the relative activity ε of each subgraph in DRAM and PMEM separately. $\varepsilon_{PMEM}[i]$ represents the subgraph with the maximum ε in PMEM, and $\varepsilon_{DRAM}[i]$ represents the minimum ε in DRAM. Making use of the *memcpy*() function, it fetches and evicts $G_{PMEM}[i]$ and $G_{DRAM}[i]$, simultaneously.

Naturally, there is a certain of time cost for data migration [30]. But, based on the difference of reading and writing performance between DRAM and PMEM, reasonable ε and k can improve the data access efficiency. Due to the adaptive push-pull updating strategy and the dynamic data migration strategy, NPGraph maximizes the rate of cache hit in DRAM. Binding the process to a given processor and turning off the automatic NUMA balancing has benefits for improving graph computing performance between different NUMA nodes. We will evaluate the efficiency of the dynamic migration mechanism in Sect. 4.

4 Evaluation

In this section, we first introduce the evaluation environment, graph data sets and graph algorithms. Secondly, we conduct experiments to assess the efficiency of the dual-block graph representation strategy. Thirdly, we evaluate the influence of the adaptive push-pull updating strategy. At last, we evaluate the effectiveness of NPGraph by comparing it with state-of-the-art systems.

4.1 Experiment Setup

All experiments are conducted on 2 NUMA nodes. Each node includes an Intel Xeon Gold 5218R CPU which is equipped with twenty cores, forty threads. L2 and L3 cache are 32 KB and 27.5 MB respectively. To emulate the experimental setup with limited DRAM resources, each NUMA node is equiped with 16 GB of DRAM and 256 GB of Optane DC persistent memory modules. The experimental

Table 2. Five public graph data sets.

Dataset	Vertices	Edges	Type
Facebook [22]	4,039	88,234	Social Graphs
Soc-LiveJournal [23]	4,847,571	68,993,773	Social Graphs
Twitter-2010 [24]	61,578,416	246,313,664	Social Graphs
Friendster [25]	65,608,366	1,806,067,135	Game Graphs
Yahoo Web [26]	1,413,511,424	5,654,045,696	Web Graphs

platform runs on Ubuntu 18.04 LTS system and works for large-scale graph processing.

As stated in Table 2, all graph data sets utilized in the studies are actual graphs. Social networks Facebook and soc-LiveJournal are chosen to evaluate NPGraph. Twitter-2010, Friendster, and Yahoo Web are all social networks, gaming sites, and web sites in their own right. They include trillions of edges and vertices. With dual-block format, the last 3 graphs are 1.56x, 1.94x, and 6.81x bigger than the available DRAM, respectively.

In the evaluations we performed, we utilized the representative PageRank and the traversal-based WCC. PageRank is set to run for 10 iterations in experiments. WCC continues to run until it converges. These algorithms give a thorough examination and have various calculation properties. Finally, NPGraph is compared with state-of-the-art systems: GraphOne [9] and XPGraph [12].

4.2 Effect of Adaptive Push-Pull Strategy

We make a full comparison of the forward-push model, the backward-pull model and the dual-adaptive push-pull model. We conduct single-threaded experiments with WCC and PageRank on the above five public graph data sets. The execution time of the above three update models are shown in Fig. 6. Obviously, the overall performance of the adaptive push-pull strategy outperforms push-based implementations. To be more precise, for WCC, the adaptive strategy performs 15.3%–27.6% better than the push model. For PageRank, the performance of adaptive strategy is 14.7%–28.9% higher than push model.

In this experiment, one surprising result is that the push-based implementation outperforms the pull-based model. In general, the read-mostly nature of the pull-based model is more cache-friendly. However, due to the six degrees of separation, there is a lot of random data access during graph computing iteratively. It is interesting to see that optimizing for cache behavior (pull-based model) may not necessarily be as successful as optimizing for transferring the most information rapidly (push-based model). That is to say, for updating propagation, the push-based model has more advantages than the pull-based model. The extra write operations in the push-based model are not just an alternate implementation of the vertex update but rather influence the scheduling of tasks. They convert states of vertices, allowing vertices to only be processed when they are profitable. This improved scheduling makes up for the increased write load.

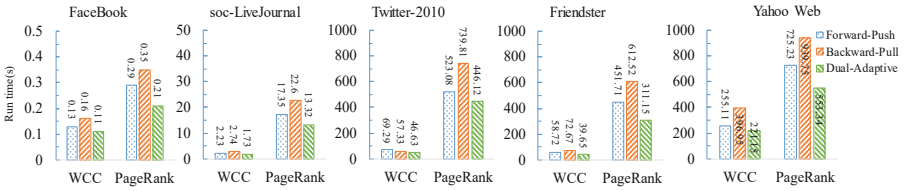

Fig. 6. Execution time of different update strategies.

4.3 Effect of Data Layering Strategy

To evaluate the efficacy of the data layering method in HMS, we conduct comparative experiments with a multi-threaded configuration (from 1 to 64 threads) on Friendster. As mentioned in Sect. 3.2, a graph G is partitioned into P subgraphs. By the offset of *Row* and the size of *Col*, it's easy to determine the sparsity or density of subgraphs. Therefore, it's a natural approach to load dense subgraphs into DRAM, preferentially. One situation is that all the structure data of subgraphs is loaded in PMEM. Under this condition, as shown in Fig. 7, PageRank and WCC are labeled as PR and WCC. As a comparison, depending on the DRAM size and the size of the data sets, we set the top 20% of subgraphs to be loaded into memory. In this case, PageRank and WCC are labeled as PR-L and WCC-L respectively.

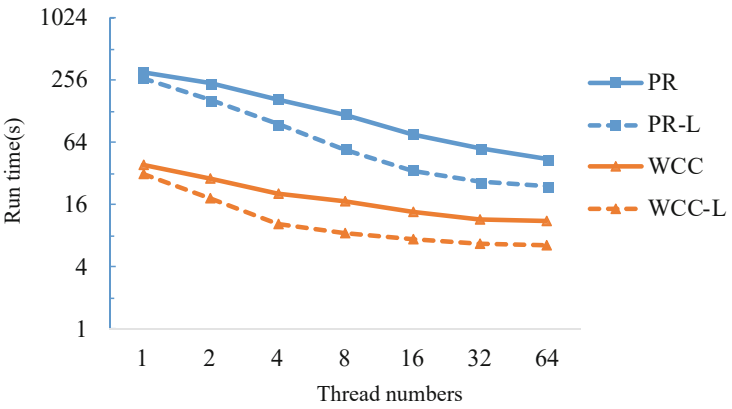

Fig. 7. Multi-thread execution time for PageRank and WCC on Friendster.

More importantly, we adopt an adaptive push-pull strategy to obtain optimal performance. Meanwhile, exchange primitives are used in code programs such as *atomic_compare_exchange*(). All the execution time decreases as the number of threads increases. As we can see in Fig. 7, the performance of PR-L and WCC-L are 1.51–1.83 times that of PR and WCC. Similarly, this trend demonstrates

that the graph data-layering strategy can take advantage of DRAM and PMEM in HMS. Therefore, the overall performance of NPGraph can be considerably improved by the data layering strategy.

We may make two inferences from the findings above: (1) Due to its sensitivity to heterogeneous memory, PageRank outperforms WCC in terms of performance. (2) We can observe that NPGraph has excellent scalability from the trend of the multi-threaded execution times.

4.4 Effect of Dynamic Data Migration Strategy

In order to evaluate the effectiveness of the data migration strategy, we run WCC and PageRank on the above 5 graphs. NPGraph is organized by the adaptive push-pull format, the layered format and data migration format respectively. Uniformly, they are marked as NPGraph-A, NPGraph-L and NPGraph-LM respectively. It is generally acknowledged that parallel computing can entirely employ graph computing systems. Thus, we evaluate their effectiveness with 64 threads.

As shown in Table 3, NPGraph-M outperforms NPGraph-L which exceeds a pure NPGraph-A version. More specifically, the performance of NPGraph-M is 14.3%–34.6% higher than that of NPGraph-A. It also confirms the experimental results in Sect. 4.3. More than that, the performance of NPGraph-M is 14.1%–23.5% higher than that of NPGraph-L in WCC. Meanwhile, the performance of NPGraph-M is 12.9%–18.5% higher than that of NPGraph-L in PageRank.

By thoroughly analyzing the experimental results of Table 3, we can determine the effectiveness of the dynamic data migration strategy in HMS. Based on this strategy, the performance of NPGraph has been further improved. It further proves that combining data layering and dynamic data migration can improve the temporal and spatial locality of graph computing in HMS.

Table 3. Execution time of WCC and PageRank(in seconds)

Algorithm	Dataset	NPGraph-A	NPGraph-L	NPGraph-M
WCC	Facebook	0.021	0.018	**0.019**
	LiveJournal	0.216	0.176	**0.201**
	Twitter-2010	5.755	4.374	**3.409**
	Friendster	11.21	6.537	**5.613**
	Yahoo Web	31.43	22.41	**17.14**
PageRank	Facebook	0.026	0.017	**0.019**
	LiveJournal	0.716	0.531	**0.601**
	Twitter-2010	33.12	24.31	**21.16**
	Friendster	40.67	32.08	**26.13**
	Yahoo Web	102.29	83.11	**69.73**

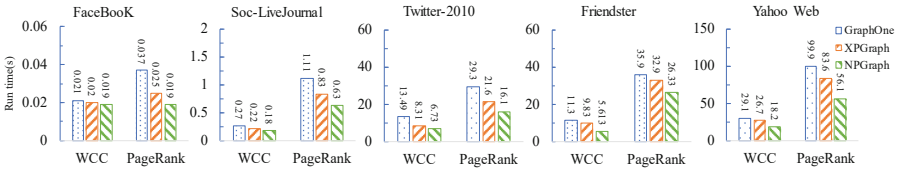

Fig. 8. Compare with state-of-the-art graph computing models.

4.5 Comparison with Other Systems

We compare NPGraph with state-of-the-art in-memory systems: GraphOne [9] and XPGraph [12]. They are all capable of parallel graph processing. To get the optimal performance, we provide two NUMA-nodes. Each node has 16 GB DRAM and 256 GB PMEM. To make it fair, GraphOne, XPGraph and NPGraph are executed in the same environment and the number of threads is set to 64. In Fig. 8, we display the execution time for each of the five different graphs.

It is obviously that NPGraph outperforms GraphOne and XPGraph in terms of the overall efficiency. Specifically, it outperforms GraphOne by 27.36% to 43.8% on Pagerank and WCC. Moreover, it outperforms XPGraph by 21.67% to 32.03% on Pagerank and WCC. The combination of the graph data layering approach with the dynamic data migration mechanism enhances the performance of NPGraph.

In order to store scale-free graphs, edge list and adjacency list, two of the most popular in-memory graph storage formats, are combined to create a hybrid storage format in GraphOne [9]. The vertex-centric random access pattern is used to navigate the data of graph structures. However, since it ignores the dynamic properties of graph computing, it must traverse all in-memory graph data, which is time-consuming. Furthermore, it generates a significant number of writing operations of intermediate results, resulting in a massive amount of I/O overhead. Based on NUMA, XPGraph [12] develops an XPLine-friendly graph access model with vertex-centric graph buffering. However, its main solution is large-scale evolving graphs. It overlooks the issue of load imbalance across numerous threads in each iteration. That is NPGraph's most powerful feature. Based on the adaptive push-pull strategy and the predictive data migration strategy, NPGraph brings lower memory access costs and better utilization of parallelism in the NUMA-based hybrid memory system.

5 Related Work

Current single-machine graph computing models enable clients to store, analyze and mine large-scale graph data sets [17,18,27,30]. These systems can be divided into traditional in-memory models and emerging hybrid memory models.

Traditional In-memory Models. To load the entire graph data, they normally use a the highest level machine, such as [3,7,14]. The underlying properties such as NUMA, memory locality, and multi-cores are utilized. Specifically,

GraphIt [3] has able to run on varying sizes of graphs, even though they have different structures. It separates computation and scheduling through its graph calculation programs. Ligra [7] is an easy-to-use platform for different graph algorithms that enhances single-machine parallel graph computing. At the same time, it takes advantage of the hybrid push-pull model. Thus, graph algorithms are simple to implement in Ligra. By utilizing NUMA characteristics, Polymer [14] creates a hierarchical barrier to increase parallelism. It suggests a differential data placement method to decrease random remote accesses, based on the observation that the bandwidth of sequential remote access is larger than that of random remote access. Additionally, it changes random remote access into sequential remote access by utilizing vertex replications.

Due to the huge capacity of DRAM, these above models avoid the overhead of disk I/O. However, their implementation is challenging when big graphs cannot fit in main memory. Thus, the hybrid memory system must be considered in single-machine graph computing models.

Emerging Hybrid-Memory Models. With the unprecedented development of persistent memory (PMEM), a few research on graph computing in persistent memory have been conducted. For example, Huang *et al.* [16] places all the edges in PMEM and all the vertices in DRAM. They make use of SRAM as a form of data buffer to decrease the random access to vertices. Based on the interval-block graph partition approach, it increases the effectiveness of data access. However, rather than emphasizing the overall performance of graph computing systems, it concentrates on lowering the amount of energy they consume. GraphOne [9] uses a hybrid storage format by combining the edge list and the adjacency list to process evolving graphs. Specifically, it first uses a circular edge log in DRAM to store the latest graph updates in the edge list format. Besides, it also uses many adjacency lists to store the older data, i.e., edges that are archived periodically from the edge log, thus supporting efficient graph queries. XPGraph [12] first introduces a PMEM-based graph store model. It proposes a PMEM-friendly graph access model to support high-performance dynamic graph stores for large-scale graphs. By using NUMA-friendly graph access, XPGraph manages the process of flushing graph data to PMEM. Based on the above systems, XPGraph realizes performance optimization in HMS.

However, they only focus on how to efficiently store graph data. As we all know, efficiently utilizing graph data is more important in graph computing. Inspired by the above works, our goal is to design an efficient graph computing model in the NUMA-based HMS. It is worth optimizing the process of subgraph construction and the adaptive push-pull model.

6 Conclusion

In this paper, we propose NPGraph which is an efficient graph computing model specially designed in NUMA-based hybrid memory systems. Simultaneously, it focuses on data storage optimization and model calculation optimization. With the graph data layering strategy, it utilizes the dual-block graph representation

strategy to support the adaptive push-pull strategy. To improve the data access efficiency, it simultaneously stores the forward and backward manners of graph data in two different NUMA nodes. What's more, during the adaptive updating process, it adopts data-driven algorithms and the dynamic data migration strategy to optimize the overall performance of graph computing.

Although it consumes twice the space to store the dual-block graph, it improves the temporal locality and the spatial locality. Moreover, it fully utilizes multi-threaded parallel technology to enhance overall performance between two NUMA nodes. Based on the above three portions, NPGraph is capable of calculating large-scale graphs effectively on a single machine. The experimental evaluation shows that NPGraph outperforms the other state-of-the-art models by 21.67% to 32.03%.

With a small amount of DRAM, NPGraph is designed as an efficient graph computing model in NUMA-based persistent memory systems. Although this model can automatically allocate graph data and adaptively update subgraphs, it needs to be extended. We plan to increase the model's scalability and multitasking capacity in the future.

References

1. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: OSDI'12, pp. 31–46 (2012)
2. Sun, P., Wen, Y., Ta, D., Xiao, X.: GraphMP: I/O-efficient big graph analytics on a single commodity machine. IEEE Trans. Big Data, 2908384 (2019, to be published). https://doi.org/10.1109/TBDATA
3. Zhang, Y., Yang, M., Baghdadi, R., et al.: Graphit: a high-performance graph DSL, In: Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, p. 121 (2018)
4. Malewicz, G., et al.: Pregel: a system for large-scale graph computing. In: SIGMOD'10, pp. 135–146 (2010)
5. Low, Y., et al.: Distributed GraphLab: a framework for machine learning in the cloud. Proc. VLDB Endow. **5**(8) (2012)
6. Gonzalez, J.E., Xin, R.S., et al.: GraphX: graph computing in a distributed dataflow system. In: OSDI'14, pp. 599–613 (2014)
7. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph computing system for shared memory. ACM SIGPLAN Not. **48**(8), 135–146 (2013). ACM
8. Zhou, S.: Gemini: graph estimation with matrix variate normal instances. Ann. Stat. **42**(2), 532–562 (2014)
9. Kumar, P., Huang, H.H.: Graphone: a data store for real-time analytics on evolving graphs. ACM Trans. Storage (TOS) **15**(4), 1–40 (2020)
10. Gonzalez, J.E., et al.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Usenix Conference on Operating Systems Design Implementation USENIX Association (2012)
11. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
12. Wang, R., et al.: XPGraph: XPline-friendly persistent memory graph stores for large-scale evolving graphs. In: 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE (2022)

13. Liu, W., Liu, H., Liao, X., Jin, H., Zhang, Y.: Straggler-aware parallel graph processing in hybrid memory systems. In: IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). Melbourne, Australia, 2021, pp. 217–226 (2021). https://doi.org/10.1109/CCGrid51090.2021.00031
14. Zhang, K., Chen, R., Chen, H.: Numa-aware graph-structured analytics. ACM SIGPLAN Not. **50**(8), 183–193 (2015)
15. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From think like a vertex to think like a graph. Proc. VLDB Endow. **7**(3), 193–204 (2013)
16. Huang, T., et al.: HyVE: hybrid vertex-edge memory hierarchy for energy-efficient graph processing. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE). IEEE (2018)
17. Gill, G., Dathathri, R., Hoang, L., et al.: Single machine graph analytics on massive datasets using Intel Optane DC persistent memory. Proc. VLDB Endow. **13**(8), 1304–1318 (2020)
18. Liu, H., Liu, R., Liao, X., Jin, H., He, B., Zhang, Y.: Object-level memory allocation and migration in hybrid memory systems. IEEE Trans. Comput. **69**(9), 1401–1413 (2020). https://doi.org/10.1109/TC.2020.2973134
19. Vora, K.: Lumos: dependency-driven disk-based graph processing. In: USENIX ATC, pp. 429–442 (2019)
20. Dang, Z., et al.: Nvalloc: rethinking heap metadata management in persistent memory allocators. In: ACM ASPLOS, pp. 115–127 (2022)
21. Wang, Q., Lu, Y., Li, J., Shu, J.: Nap: a black-box approach to NUMA-aware persistent memory indexes. In: USENIX OSDI, pp. 93–111 (2021)
22. http://snap.stanford.edu/data/ego-Facebook.html
23. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: membership, growth, and evolution. In: KDD'06, pp. 44–54 (2006)
24. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media. In: WWW'10, pp. 591–600 (2010)
25. Boldi, P., Vigna, S.: The webgraph system I: compression techniques. In: WWW'04, pp. 595–602 (2004)
26. http://developer.yahoo.com/blogs/616566076523839488/
27. Li, B., et al.: D^2Graph: an efficient and unified out-of-core graph computing model. In: 2021 IEEE ISPA, pp. 193–201 (2021). https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00038
28. Yu, J., et al.: DFOGraph: An I/O and Communication-Efficient System for Distributed Fully-out-of-Core Graph Processing (2021)
29. Liu, W., Liu, H., Liao, X., Jin, H., Zhang, Y: Straggler-aware parallel graph processing in hybrid memory systems. In: IEEE/ACM 21st International Symposium on Cluster. Cloud and Internet Computing (CCGrid) 2021, pp. 217–226 (2021). https://doi.org/10.1109/CCGrid51090.2021.00031
30. Li, B., et al.: EPGraph: an efficient graph computing model in persistent memory system. In: 2022 IEEE ISPA, pp. 9–17 (2022). https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom57177.2022.00009