IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 7, 2019, accepted July 18, 2019, date of publication July 25, 2019, date of current version August 13, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2931058

NGraph: Parallel Graph Processing in Hybrid Memory Systems

WEI LIU, HAIKUN LIU, (Member, IEEE), XIAOFEI LIAO, (Member, IEEE), HAI JIN, (Fellow, IEEE), AND YU ZHANG, (Member, IEEE)

National Engineering Research Center for Big Data Technology, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
System Services Computing Technology and System Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
Cluster and Grid Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

Corresponding author: Haikun Liu (hkliu@hust.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1001603, and in part by the National Natural Science Foundation of China (NSFC) under Grant 61672251, Grant 61732010, and Grant 61825202.

ABSTRACT Big data applications like graph processing are highly imposed on memory capacity. Byte-addressable *non-volatile memory* (NVM) technologies can offer much larger memory capacity, lower cost per bit relative to traditional DRAM. They are expected to play a crucial role in mitigating I/O operations for big data processing. However, since the NVMs show higher access latency and lower bandwidth compared with DRAM, it is still challenging to fully exploit the advantages of both the DRAM and NVM for graph processing. In this paper, we propose *NGraph*, a new parallel graph processing framework specially designed for hybrid memory systems. According to different access patterns of graph data, NGraph exploits memory heterogeneity-aware data placement strategies to avoid random accesses and frequent updates to NVM. NGraph partitions graph by destination vertices and exploits a task decomposition scheme to avoid data contention between multicores. Meanwhile, the NGraph balances the execution time of parallel graph data processing on multicores through a work-stealing strategy. Moreover, the NGraph also proposes software-based data pre-fetching to improve cache hit rate, and supports huge page to reduce address translation overhead. We evaluate NGraph using a hybrid memory emulator. The experimental results show that NGraph can achieve up to 48.28% performance improvement for several typical benchmarks compared with the state-of-the-art systems Ligra and Polymer.

INDEX TERMS Graph processing, data placement, graph partitioning, DRAM/NVM hybrid memory.

I. INTRODUCTION

Social network, road network and biological network contain a lot of valuable information, and mining this information from massive data plays an important role in enterprises' decision making. Most of these data can be abstracted into graph, which attracts increasing research interest of industry and academia on the graph processing framework.

It is essential to process the graph-structure data with low cost and high efficiency. Because traditional DRAM technologies generally feature low memory density, high cost and power consumption, they cannot satisfy the increasing requirement of main memory for high-performance large graph processing. Although many distributed graph systems [1]–[4] or out-of-core [5]–[8] graph systems have been developed to use limited memory resource for large graph processing, these systems often suffer high memory cost (using multiple machines) or extremely low performance (high I/O cost due to disks).

Recently, emerging Non-Volatile Memory (NVM) [9] technologies such as PCM [10] and ReRAM [11] have the potential to radically change the landscape of memory systems. They generally show higher memory density, lower energy consumption and lower cost per GB than DRAM. The cost of NVM is about five times lower than that of DRAM [12], and will be lower with the development of technology. Although NVM shows lower bandwidth and higher access latency than that of DRAM, we find that a hybrid memory system has a potential to improve the performance of graph processing by up to 6 times compared to an out-of-core

The associate editor coordinating the review of this manuscript and approving it for publication was Walter Didimo.

 This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

system under the same constraint of memory cost. The hybrid memory system with a small size of DRAM and a relatively large amount of NVM can significantly reduce the total cost of graph processing system, and lead to higher performance per dollar (see Section II).

There are only a few preliminary studies [13]–[15] on graph processing in hybrid memory systems. Most work only represents an early example to showcase the benefit of using hybrid memories for graph processing [13], [14]. To use the hybrid memories effectively and efficiently, the primary challenging is to place different kinds of data on DRAM and NVM properly. The application performance may slow down if data is randomly placed on DRAM and NVM without considering the data access patterns. For example, when frequently-accessed (hot) data is stored in NVM, the accesses to high-latency NVM would slow down the execution of applications.

In this paper, we propose *NGraph*, a parallel graph processing framework specially designed for hybrid memory systems. NGraph exploits a memory heterogeneity-aware data placement strategy to avoid random accesses and frequent updates to NVM. We analyze the access patterns of different data structures in graph dataset, and place read-only and sequentially accessed graph data in NVM, while frequently accessed graph property and state data are placed in DRAM to mitigate performance degradation.

On the other hand, parallel graph processing in multicore systems usually leads to data contention, i.e., multiple threads may update the same data block concurrently. In order to guarantee data consistency and reduce the CPU stall time of concurrency control, it's necessary to mitigate the data contention between different processors. A general approach is to update a data block with an atomic operation, and other processors must wait for the data-exclusive thread to complete the operation. The atomic operation locks the entire cache line, and will cause CPU stalling for a majority of application execution time. In contrast, we partition graph by destination vertices and exploit a task decomposition scheme to avoid data contention among multiple cores, and thus avoiding the performance overhead of atomic data updates.

Moreover, NGraph also provides two optimization techniques to further improve the performance of graph processing in hybrid memory systems. We propose software-based data pre-fetching [16] by exploiting data locality to improve cache hit rate. Also, to mitigate the virtual-to-physical address translations for graph data with large memory footprint, NGraph supports hugepage [17] for memory allocation.

The main contributions of this paper are as follows:

- We study the opportunities and challenges of graph processing to exploit the NVM characteristics in hybrid memory systems.
- We propose NGraph, a parallel graph processing framework specifically designed for hybrid memories systems. It adopts a nvm-aware data placement strategy to mitigate the impact of relatively high NVM access latency on application performance. NGraph partitions graph by destination vertices and exploits a task decomposition scheme to avoid data contention between different processors. Moreover, NGraph leverages a work stealing strategy to minimize the time of parallel graph data processing on multicore systems.
- We also exploit data structure-aware software pre-fetching to improve cache hit ratio, and hugepage supporting to reduce address translation overhead. These two optimization techniques can further improve the performance of graph processing in hybrid memory systems.
- We implement NGraph and evaluate it with several typical graph applications. Experimental results show that NGraph can improve application performance by 10.79% to 48.28% compared to other state-of-the-art graph processing framework running in a hybrid memory system.

The rest of this paper is organized as follows. Section II introduces the NVM technology, and analyzes its opportunities and challenges in graph processing. Section III presents the design and implementation of NGraph. Section IV introduces two other optimization strategies in NGraph. The experimental results are described in Section V. Section VI presents the related work and we conclude this paper in Section VII.

II. BACKGROUND AND MOTIVATION

A. NVM TECHNOLOGIES

Non-Volatile Memories(NVM) such as MRAM [18], F-RAM [19], STT-MRAM [20], PCM [10], [21], [22] and RRAM [11] are characterized with byte-addressable, near-zero static energy consumption and high memory density. They have the potential to significantly expand the capacity of main memory and to reduce memory energy consumption. Despite the advantages of NVMs, they cannot substitute for DRAM currently because the relatively high read and write latency of NVMs may lead to significant performance degradation. Moreover, NVM's limited write endurance is also a constraint for long-term usage. As a result, a typical use of NVM is to combine it with a small size of DRAM to build a hybrid memory system. In this way, the main memory system can exploit the advantages of both NVM and DRAM and avoid their disadvantages [23]. It can both satisfy the requirement of large memory capacity for applications and guarantee the service level agreement (SLA). This paper aims to design a graph processing system on such hybrid memory system consisting of a small size of DRAM (GB) and a relatively large amount of NVM (TB). We compare the features of NVM with DRAM in Table 1. Here, the NVM represents Intel's newly released Optane DC Persistent Memory DIMMs [24].

Since we still have not got the commercial NVM device during this work, our performance studies on NVM are

TABLE 1. The features of NVM compared to DRAM.

Parameter	DRAM	NVM
Capacity	GBs	TBs
Read Latency	75ns	150ns to 300ns
Write Latency	80ns	320ns to 640ns
Cost	4500$ (128GB)	695$ (128GB)
Endurance	10^{16}	10^6 to 10^8

primarily conducted on a hybrid memory emulator. There are mainly two ways to simulate NVM devices. One approach is to simulate the hybrid memory system by software (e.g. ZSim [25], Gem5 [26], NVMain [27], DRAMSim2 [28]). These simulators exhibit extremely low speed for simulating the execution of applications, and they are generally unfeasible to run applications with very large memory footprint. Another approach is to use hardware emulators (e.g. HMPE [29], QUARTZ [30], HME [31]). These hardware emulators use DRAM to emulate the performance characteristics of NVM. Applications run on these emulators are much close to the execution on a real hardware. They also support running applications with large memory footprint. As both HMPE [29] and QUARTZ [30] do not support the emulation of NVM write latency, we evaluate the performance of graph processing in hybrid memory systems using HME, which can accurately emulate all performance features of NVM with trivial software overhead.

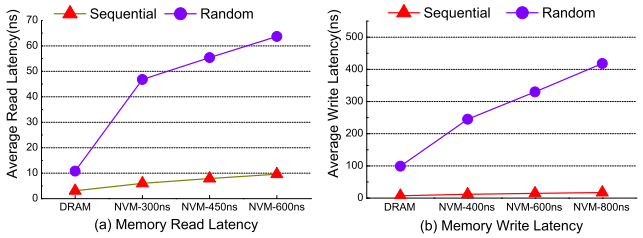

FIGURE 1. Average data read/write latency for different access patterns under various NVM latency settings.

Modern computer systems often leverage several techniques to hide memory access latency, such as hardware-based data pre-fetching, memory level parallelism (MLP) and cache. Program's performance degradation is not exactly proportional to the increased latency of NVM. We measure the read and write latency of different memory access patterns under various NVM latency configurations. As shown in Figure 1(a), we find that NVM random read latency can be up to 6.9 times higher than that of NVM sequential read. In Figure 1(b), NVM random write latency can be up to 20.7 times higher than that of NVM sequential write latency, while the performance gap between sequential write and random write for DRAM is much narrow. These observations indicate that the data access patterns should be particularly considered to optimize the data placement in hybrid memory systems.

B. OPPORTUNITIES AND CHALLENGES OF GRAPH PROCESSING IN HYBRID MEMORY SYSTEMS

Graph processing follows a scatter-gather programming model, and is often implemented through vertex-centric or edge-centric approach:

- Vertex-centric: it performs iterative computing over vertices (e.g., Ligra [32]). For each vertex, the value of local vertex is propagated to neighbor vertices in the scatter phase, and the gather phase obtains the values of neighbor vertices to update the local vertex value.
- Edge-centric: it performs iterative computing over edges to avoid random access to edges (e.g., X-stream [6]). For each edge, the scatter phase appends updates to a list named *Uout*. After the shuffle phase, these re-arranged updates are applied to vertices in the gather phase.

To demonstrate the benefit of graph processing in hybrid memory systems, the performance improvement of hybrid memory system over out-of-core systems(i.e. traditional DRAM memory + disk) is evaluated under a constraint of the same cost. The work [12] revealed that the cost of DRAM per GB is about five times of NVM. Accordingly, we assume that the cost of 16G DRAM is equal to 10G DRAM plus 30G NVM. We measure the performance of various graph algorithms under these two different memory configurations. The X-stream system and the twitter dataset [33] are used in our experimental studies.

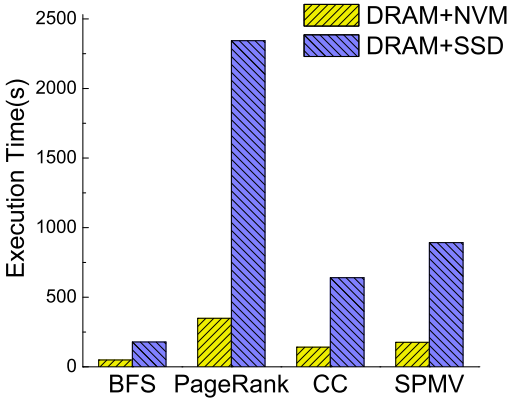

FIGURE 2. Performance gap between hybrid memory system and "DRAM+Disk".

Figure 2 shows the execution time of various graph algorithms in the two memory systems. We can find that hybrid memory system can achieve higher performance at the same cost. The performance of BFS, PageRank, CC and SPMV is improved by 2.47, 5.96, 3.52, and 4.06 times, respectively. For the system with 16G DRAM, graph data cannot be fully loaded into memory, leading to frequently data swapping between memory and disk. In contrast, the hybrid memory system provides a much larger memory space, and all graph data can be processed in memory without suffering the high latency of I/O operations.

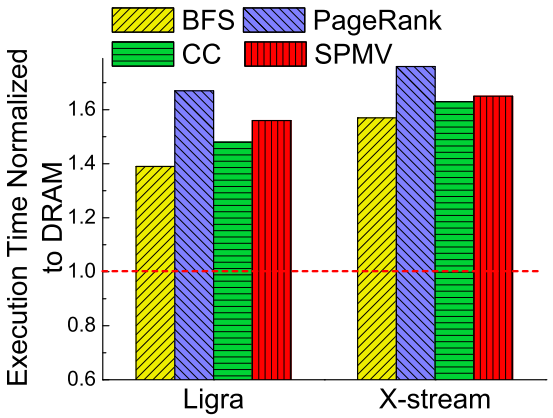

FIGURE 3. Performance gap between hybrid memory system and DRAM-only system.

As mentioned above, previous graph processing systems have not considered the memory heterogeneity, and data is randomly placed in two kinds of memories. Applications usually cannot achieve optimal performance without optimizing data placement. We further measure the performance gap between a full-DRAM memory system and a hybrid memory system when running different graph applications. In this experiment, we run Ligra and X-stream using the twitter dataset. Assume NVM read/write latencies are two and eight times as high as that of DRAM, respectively, and NVM bandwidth is only half of DRAM. Figure 3 shows that the performance of graph applications degrades by 39% to 73% in hybrid memory system. The experimental results suggest that there is vast room to optimize the performance of graph processing by redesigning the data placement strategy for hybrid memory systems.

III. DESIGN AND IMPLEMENTATION

NGraph is a NVM-aware parallel graph processing framework in hybrid memory systems that inherits Ligra's programming model. NGraph optimizes data placement based on access patterns of graph data structures and NVM characteristics, and thus reduces frequent random access to NVM. Moreover, by graph partitioning and task decomposition, NGraph eliminates data competition among multiple threads and balances load among processors.

A. GRAPH DATA STRUCTURE AND NVM-AWARE DATA PLACEMENT

1) IN-MEMORY GRAPH DATA STRUCTURE

The graph data mainly consists of graph structure data, vertex property data, and runtime state data. The in-memory data structure of graph is shown in the left dashed box of Figure 4. Graph structure data includes edge and vertex structures, which are usually stored in two edge arrays and a vertex array, respectively. The in-edge array and out-edge array store the source and destination vertices, respectively. The vertex array stores vertex's metadata, including vertex ID, in-degree and out-degree, the addresses of in- and out-edges. Vertex property data (such as the parent of a vertex in

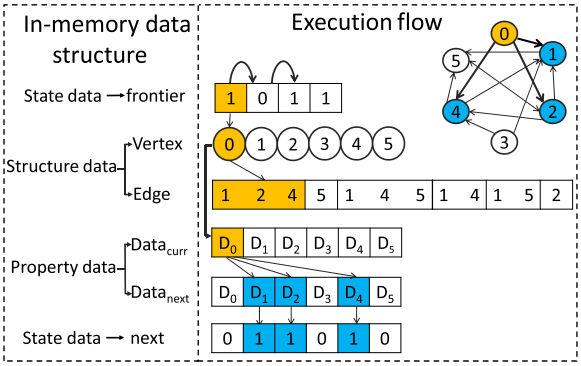

FIGURE 4. In-memory data structure and the execution flow of graph.

BFS, the rank value in PR) is stored in two arrays. $Data_{curr}$ maintains the property value from previous iteration, $Data_{next}$ stores the property value in current iteration. The run state data (a vertex is active or inactive) is also stores in two arrays, i.e., *Frontier* array and *Next* array. In traditional graph processing frameworks, these data are randomly allocated in hybrid memory system, which may cause some data to access NVM frequently, affecting the performance of program and reducing the service life of NVM.

2) GRAPH DATA ACCESS PATTERN AND NVM-AWARE DATA PLACEMENT

We design a data placement strategy based on the access patterns of different data structures. The basic principle is to place randomly accessed and frequently updated data in DRAM, and sequentially accessed and read-only data in NVM. Because the sequential access speed of NVM is about 6 times faster than its random access. In this way, we can make full use of the advantages of NVM and avoid the disadvantages of NVM's high write latency and limited write endurance.

In a static graph, the graph structure remains unchanged during the execution of programs, so the graph structure data is read-only. For a natural graph, the graph structure data is much larger than vertex property data and state data, usually accounting for about 71% to 83% of the total data size [34], [35]. The execution flow and access pattern of graph processing are shown in the right dashed box of Figure 4. Since graph structure data is sequentially stored in edge array and vertex array, its memory accesses are sequential, program has good special locality when accessing these data. Therefore, it is reasonable to place the sequentially accessed and read-only graph structure data in large-capacity NVM. During graph processing, vertex property data is usually updated along with edges. Due to the irregularity of graph, these updates often spread throughout the whole graph, resulting in random accessing to these data. Therefore, vertex property data is suitable for locating in DRAM. In addition, since runtime state data (such as the vertex state array) is small and frequently accessed and updated, and its memory is reallocated in each iteration, it should be also placed in DRAM.

We first obtain the size of available DRAM in the hybrid memory system. According to the data type of vertex property data and the size of input graph, we calculate the memory size used by vertex property data and then determine where it is placed. If it is smaller than available DRAM size, all property data is placed in DRAM. Otherwise, the vertices of the graph are sorted according to the degree of each vertex. The property data of vertices with large degrees is placed in DRAM, and the remaining data is placed in NVM.

3) ANALYSIS OF MEMORY ACCESS COSTS

We analyzed the proportion of accesses to the two kinds of memories in conventional graph processing frameworks and NGraph. To simplify the complexity of the analysis, we ignore the effect of cache on memory accesses, and assume that there are no isolated vertices in the graph, and DRAM accounts for $a\%$ of total capacity of hybrid memories. Previous frameworks do not distinguish the type of data and randomly place data in hybrid memory systems, so there would be approximate $a\%$ of DRAM access. In hybrid memory systems, the value of a is usually small.

NGraph places graph property data and state data in DRAM, which is usually applicable due to their small memory footprints. In the following, we discuss the memory accesses of active and inactive vertices in each iteration, respectively. As shown in Figure 4, for a active vertex v with degree d, the processing thread first scans the state array *frontier* (one time of DRAM access), and accesses the vertex structure array to get the address of the edge array (one time of NVM access), and then visits the edge structure array (d times of NVM accesses) to get neighbor vertices' ID. Meanwhile, the processing thread obtains the property data of vertex v from the data array $Data_{curr}$ (one time of DRAM access), and then updates the property data $Data_{next}$ of neighbor vertices (d times of DRAM accesses). Finally, it updates the status array *next* (d times of DRAM accesses). We can find that two-thirds of data accesses are located in DRAM for the active vertex. For inactive vertices, the processing thread only needs to visit the state array *frontier* once (one time of DRAM access).

From above analysis, we can find that our data placement strategy can guarantee more than 66.7% of memory references are located in relatively small DRAM. The memory references distributed on DRAM are much greater than previous graph processing frameworks in hybrid memory systems.

B. PARALLEL GRAPH PROCESSING

1) ANALYSIS OF PARALLEL GRAPH PROCESSING

Graph partitioning is widely used in distributed graph processing systems and multi-core systems. An intuitive partition scheme is to evenly distribute vertices to each core. For example, Ligra system uses CilkPlus to implement lightweight multi-core parallel processing. In essential, each core processes the same number of vertices. Because the degree of every vertex is not uniform, the number of edges processed by each core are usually not equal. As the computation complexity of graph processing is usually linear to the number of edges, this partition scheme usually causes load imbalance among multiple processors.

In contrast, the vertex-cut scheme [2] in distributed graph processing systems evenly partitions the edges, and processes the partitioned sub-graph by each processor. This scheme can achieve better CPU load balance. However, we find that multiple cores often need to update the property data of a vertex simultaneously during parallel graph processing. Atomic update is a general approach to guarantee the concurrency control and to avoid data competition, which in turn leads to a new load imbalance problem. On the one hand, when a thread is updating a vertex property data, other threads can't update the same vertex property data until the thread completes the processing of the data, causing idle waiting of other processors. On the other hand, atomic update is often very costly, because it locks the entire cache line in which the property data is stored. When other processors need to update other property data in the same cache line, they also need to wait for the completing of the previous update operation.

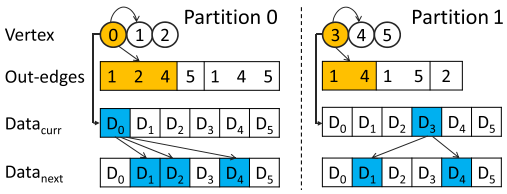

FIGURE 5. Graph partitioning and processing.

As shown in Figure 5, the edges of the graph in Figure 4 are evenly partitioned into two sub-graphs (i.e., two partitions) according to the source vertices. Thread 0 starts to process the vertices in sub-graph 0 from vertex 0, and thread 1 starts to process the vertices in sub-graph 1 from vertex 3. Each thread is bound to a processor and all threads compute in parallel. Thread 0 starts processing vertex 0, accesses the out-edge list of vertex 0, and updates the property data of vertex 1, 2 and 4 according to the property data of vertex 0. At the same time, thread 1 starts processing vertex 3, accesses the out-edge list of vertex 3, and updates the property data of vertex 1 and 4 according to the property data of vertex 3. As shown in Figure 6, for atomic updating of property data of vertex 1 concurrently by thread 0 and thread 1, the two threads can only process the same vertex alternately, and vertex 4 is updated in the same way by the two threads. In addition, if multiple threads update the same or adjacent vertices simultaneously, the duplicated data in other cores' private caches should be invalidated to guarantee cache consistency. For example, if vertex property data a and b are located in the same cache line, updating the data a in a core would lead to invalidating the same cache line in another core that would update the data b. As a result, frequent cache invalidation also causes serious performance degradation.

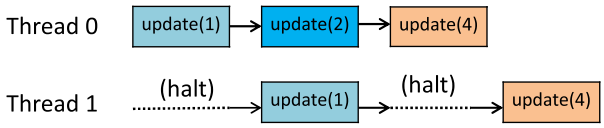

FIGURE 6. Parallel processing flows of two threads.

Algorithm 1 Graph Partitioning by Target Vertices

```
procedure Partition(G,numOfCore)
    average = m/numOfCore;
    for each v ∈ V do
        if |E_i| > average and i < numOfCore then
            i++;
        end if
        E_i = E_i ∪ inedge(v);
    end for
    return G_i = (V, E_i);
end procedure
```

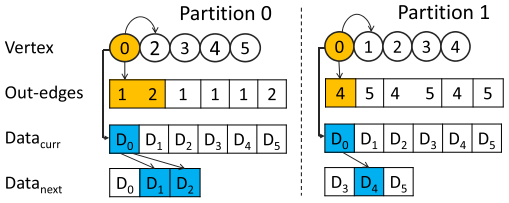

FIGURE 7. Graph partitioned by target vertices.

2) GRAPH PARTITIONING

In order to avoid CPU stalling caused by data competition, we partition the edges into different sub-graphs according to their target vertices, as described in Algorithm 1. As shown in Figure 7, the graph in Figure 4 is partitioned into two partitions. 6 edges with target vertices {0, 1, 2} (for example, $V_0 \rightarrow V_1$, $V_0 \rightarrow V_2$) are assigned to partition 0, and other 6 edges with target vertices {3, 4, 5} (for example, $V_0 \rightarrow V_4$, $V_1 \rightarrow V_5$) are assigned to partition 1. Each thread sequentially processes the vertices in a partition, and only updates the neighbor vertices' property data according to the edges whose target vertices are in the same partition. This scheme avoids updating the same vertex's property data by multiple threads, and can avoid the potential atomic updates due to data contention.

When the edges of a vertex are partitioned into different sub-graphs, each sub-graph contains replications of all vertex's structure data. Graph traversal must visit all vertices in each sub-graph, increasing memory consumption and many control messages. Fortunately, when graph is partitioned by destination vertices, each sub-graph has many vertices with a degree of zero [36]. We do not need to store these vertices with a degree of zero in a sub-graph. Figure 8 shows the percentage of vertices with zero degree when graphs are partitioned into different number of sub-graphs. Inspired by this observation, we only store the structure data of vertices that have edges in each sub-graph. In this way, each core only performs a portion of computations for partial vertices that have edges in each sub-graph, and thus reduces the memory consumption of vertices' structure data and the number of control messages.

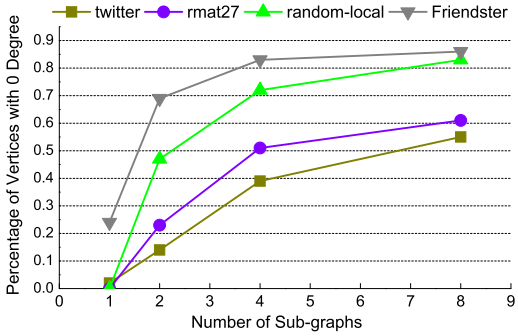

FIGURE 8. The percentage of vertices with zero degree when partitioned into different number of sub-graphs.

3) LOAD BALANCE

Since only the edges of active vertices should be processed, it often cannot achieve good load balance by partitioning edges evenly. We employ a work stealing scheme to further improve the load balance among multiple cores.

We partition the data in each sub-graph into sub-blocks, and each thread processes these sub-blocks for a sub-graph from the beginning to the end. When a thread finishes processing corresponding sub-graph, it checks whether there are any other sub-graphs that have not been completely processed yet. If so, the thread whose job has been finished will assist to process the unfinished sub-graph from its last data block. When all sub-graphs have been processed, the program can begin the next iteration.

C. PROGRAMMING MODEL

Assume a static graph $G = (V, E)$ is processed in NGraph, where V and E represent the vertex set and the edge set, respectively. The numbers of vertices and edges are $|V| = n$ and $|E| = m$, respectively. NGraph inherits Ligra's programming interfaces: *edg-map*() and *ver-map*(). Like Ligra, NGraph uses *Frontier* to represent the subset of vertices.

- edg-map (*G, Frontier, F, C*): *subvertex*
  For all edges whose source vertices are in the *Frontier* and target vertices satisfy the condition *C*, *edg-map* executes the user-defined function *F*, and returns a set of vertices when the return value is true, i.e.,
  $Out = \{u \in V \mid F(u, v) = true \wedge (u, v) \in E\}$.
- ver-map (*G, Frontier, F*): *subvertex*
  For vertices in *Frontier*, *ver-map* executes the user-defined function *F*, and returns a set of vertices when the return value is true, i.e., $Out = \{u \in V \mid F(u) = true\}$.

Algorithm 2 describes the pseudo-code of PageRank algorithm in NGraph. Let *G* represent the graph to be processed, $Data_{curr}$ and $Data_{next}$ represent the graph property data, and *Frontier* represent the active vertex set. The procedure *PR-edg-F* updates the target vertices' property values based on the ranks of their source vertices. The procedure *PR-ver-F* normalizes the rank value of vertex *v*, and judges whether

the absolute difference between the new rank and the old rank is greater than a given threshold. If so, v will be alive in next round. The procedure *PageRank* repeatedly executes *PR-edg-F* function and *PR-ver-F* function till the number of iterations exceeds a given value or the convergence condition is met.

Algorithm 2 PageRank in NGraph

$Data_{curr} = \{1/n, \ldots, 1/n\}$;
$Data_{next} = \{0, \ldots, 0\}$;

procedure PR-edg-F(s,d)
  $writeAdd(\&Data_{next}[d], Data_{curr}[s]/OutDegree(s))$;
end procedure

procedure PR-ver-F(v)
  $Data_{next}[v] = 0.85 \times Data_{next}[v] + 0.15/n$;
  if $Data_{next}[v] - Data_{curr}[v] > \epsilon$ **then**
    v *is alive*;
  end if
  $Data_{curr}[v] = 0$;
end procedure

procedure PageRank(G,maxite)
  $Frontier \leftarrow V$;
  $i = 0$;
  while $i < maxite$ *and* $Frontier \neq \phi$ **do**
    $Frontier = edg\text{-}map(G, Frontier, PR\text{-}edg\text{-}F, true)$;
    $Frontier = ver\text{-}map(G, Frontier, PR\text{-}ver\text{-}F)$;
    $Swap(Data_{curr}, Data_{next})$;
    $i++$;
  end while
end procedure

IV. OPTIMIZATIONS ON NGRAPH

We leverage two optimization techniques to further reduce NVM access latency and improve the performance of NGraph.

A. HUGEPAGE SUPPORTING

With the rapid growth of graph dataset and the correspondingly large capacity of main memory provided by NVM, the virtual-to-physical address translation has became a new performance bottleneck of graph processing systems. When more memory pages are used, there is a potential to cause more TLB misses. In general, a TLB miss leads to three additional memory accesses for page table walking, which requires hundreds of CPU cycles to get the physical address of a page. A high TLB miss rate often implies significant performance degradation.

To mitigate the address translation overhead, an effective approach is to use a larger size of memory page, such as huge pages or super pages. The Linux OS has already supported Direct Huge Pages (DHP) management with pre-allocation and transparent hugepage (THP) management using dynamic allocation. The former requires users to manually pre-allocate huge pages, and allocate memory using *mmap()* in program. This approach needs to change user's program codes. To simplify the programming and support legacy applications, NGraph uses THP to support huge pages without the modification of applications.

B. DATA PRE-FETCHING

Data pre-fetching is an effective way to reduce memory access latency. A recent work [37] has shown that data pre-fetching according to data structure, access patterns and reuse distance can bring about 19%-102% performance improvement for graph processing. We design a data pre-fetching scheme that combines OS-supported hardware pre-fetching with software-based pre-fetching to reduce memory access latency in NGraph.

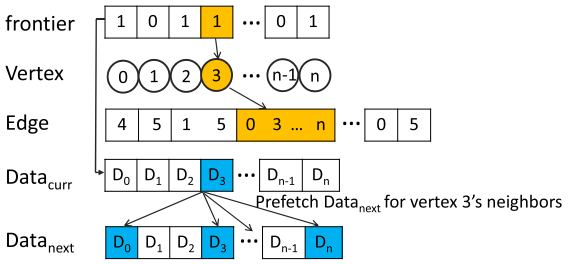

FIGURE 9. Graph data-aware software pre-fetching.

For the graph structure data in NVM, because it is often read-only and sequentially accessed, the existing hardware pre-fetcher is effective and efficient. However, for randomly accessed graph property data, the proposed graph partitioning strategy transforms the data access pattern from inter-core accesses to intra-core accesses. This avoids the cache interference due to concurrency control and does not affect the data pre-fetching, facilitating us to use software pre-fetching to improve cache hit rate. First, in order to support dynamic computation, we use a bitmap to record active vertices. Note that pre-fetching the property data of neighbor vertices for adjacent vertex may be useless, because it may be not active. For example, in Figure 9, when the work thread processes vertex 0, it is useless to pre-fetch the property data of vertex 1's neighbor vertices. Second, to pre-fetch the property data directly for next active vertex, we need to get next active vertex first and then pre-fetch the property data of its neighbor vertices. Simple software pre-fetching usually leads to high software overhead. To make a tradeoff between the cost and benefit of data pre-fetching, we only pre-fetch the vertex property data when a vertex's degree is greater than a given threshold d. As shown in Figure 9, when vertex 3 is processed, we pre-fetch the property data of its neighbor vertices since its degree is larger than d. We study the impact of different thresholds on application performance, and find that the application performance is significantly improved when d increases from 1 to 20, and becomes roughly stable

TABLE 2. System configuration.

CPU	20 cores, Intel(R) Xeon(R) CPU E5-2650 v3 @2.30GHz
L1 Cache	private 32 KB, 8-ways, 64 B cache block
L2 Cache	private 256 KB, 8-ways, 64 B cache block
L3 Cache	shared 25.6 MB, 20-ways, 64 B cache block
DRAM	10 GB capacity, 64 GB/s Bandwidth, 150 ns read latency, 200 ns write latency
NVM	256 GB capacity, 32 GB/s Bandwidth, 300 ns read latency, 800 ns write latency

when d ranges from 21 to 100, and gradually decreases when d becomes larger than 100. In our experiment, we empirically set d as 20.

V. EVALUATION

Based on the graph processing framework Ligra, NGraph is implemented in C++ codes using *pthreads* library. We use the hardware simulator HME to build up a hybrid memory system based on DRAM. Our experiments are conducted on the emulated hybrid memory system. The detailed configuration of the testbed is shown in Table 2. We set the available DRAM on node 0 to be 10G at the booting time. For the characteristics of NVM, we have referred to the previous work [13].

A. WORKLOADS AND DATASETS

Four typical graph workloads are used to test the performance of NGraph on different datasets:

PageRank: it is an iterative algorithm to rank web pages in Google's search engine. The web pages and the links are modeled as graph's vertices and edges. PageRank calculates the ranks of vertices by each vertex updates the ranks of their neighbor vertices along with the out-edge according to their own ranks iteratively, till the specified number of iterations is reached or the convergence condition is met.

SPMV: it is a general-purpose algorithm in scientific computing, such as solving sparse linear equations and eigenvalues. SPMV multiplies a sparse matrix with a dense vector. The sparse matrix can be stored in weighted graph format, the values of the vectors can be stored in each vertex.

BFS: it is a graph traversal algorithm and a building blocks for many graph algorithms. BFS starts from a root vertex, it visits all the neighbor vertices of the root vertex first, and then accesses its neighbor vertices iteratively till all the vertices are accessed.

CC: it is an iterative algorithm that finds all connected sub-graphs and the maximum connected sub-graph in a graph. If vertex i and j are in different connected sub-graphs, and there is no path from i to j in the graph, the two vertices i and j in each sub-graph are not reachable.

We adopt two synthetic graphs and two real-world graphs as the input data, as shown in table 3.

- Rmat27 uses RMAT generator [38] provided by Graph500 [39] to synthesize a graph of scale 27 with 134.2 million vertices and 2.14 billion edges.
- Random-local [32] is a synthetic random graph with 200 million vertices and 2 billion edges.
- Twitter is a graph of social network from the real world with 14.7 million vertices and 1.47 billion edges.
- Friendster [40] is a social network re-launched as a game website with 65.6 million vertices and 1.81 billion edges.

TABLE 3. Synthetic and real-world graphs.

Datasets	Vertices	Edges	Size
twitter	1.7M	1.4B	12.5G
Rmat27	134.2M	2.14B	21.4G
Random-local	200M	2B	19.5G
Friendster	65.6M	1.81B	22.7G

B. SYSTEM PERFORMANCE

We measure the execution time of various algorithms under different datasets in NGraph using 8 threads in hybrid memory, and compared the results with two state-of-the-art single-machine graph computing frameworks—Ligra and Polymer. For the fairness of comparison, the hugepage optimization strategy and OS-supported hardware pre-fetcher are also enabled for other frameworks.

TABLE 4. The application execution time (seconds) for different datasets and graph processing frameworks.

Applications	Graph Dataset	NGraph	Ligra	Polymer
PageRank	twitter	301.46	432.45	552.93
	Rmat27	344.86	515.87	666.78
	Random-local	848.18	1308.52	1552.87
	Friendster	616.48	913.17	1101.64
BFS	twitter	17.48	19.63	20.55
	Rmat27	23.95	26.85	27.01
	Random-local	42.79	48.67	49.78
	Friendster	31.60	36.27	36.36
CC	twitter	25.70	30.88	33.53
	Rmat27	31.16	38.54	39.63
	Random-local	55.23	69.39	72.05
	Friendster	38.98	46.12	48.79
SPMV	twitter	142.56	179.25	223.06
	Rmat27	184.68	242.23	277.09
	Random-local	378.72	487.66	556.86
	Friendster	302.58	398.50	436.56

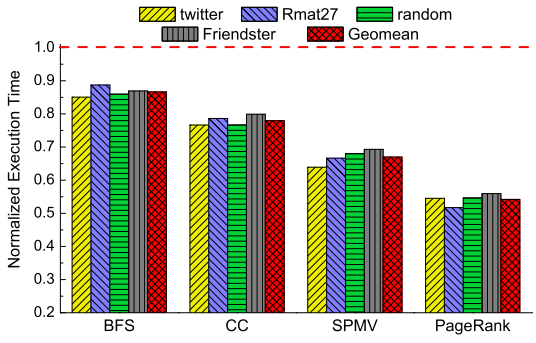

FIGURE 10. The execution time of algorithms in NGraph, all relative to Polymer (NGraph/Polymer).

Table 4 shows the execution time of different algorithms using the four dataset in different graph processing framework. To make the comparison more concise, Figure 10 shows the normalized execution time of algorithms in NGraph compared with Polymer. We can find that NGraph outperforms other two frameworks in all cases. NGraph can improve the performance of PageRank algorithm by up to 48.28% compared to Polymer. The geometry mean of performance improvement for the four algorithms are 13.34%, 22.05%, 33.03% and 45.79%, respectively. The traversal algorithms such as CC and BFS show less performance improvements, because relatively less memory accesses make them to be less sensitive to the hybrid memories.

The significant performance improvement of NGraph is mainly attributed to our nvm-aware data placement and graph parallel optimization schemes. Compared with Ligra, NGraph partitions a graph by destination vertices and exploits a task decomposition scheme to balance the load among multiple cores, and thus avoid data contention and CPU idle waiting. Because Polymer partitions edges evenly without considering the potential data contention and load balance within a node, it shows the worst performance among the three frameworks in most cases.

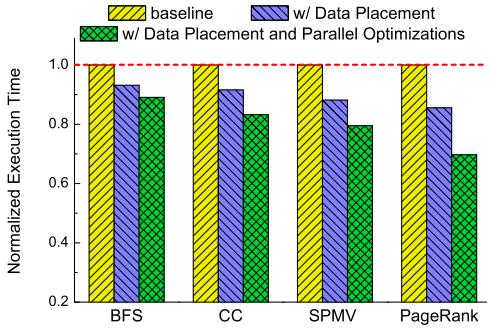

FIGURE 11. The impact of NGraph's data placement and parallel optimization techniques on performance improvement of applications.

In order to quantify the effects of NGraph's two major optimization techniques on performance improvement of graph processing in hybrid memory systems, we measure the execution time of NGraph with only NVM-aware data placement, and both with NVM-aware data placement and parallel optimization strategies (including graph partitioning and work stealing schemes). The baseline is Ligra without any optimization. All experiments use the twitter dataset. Figure 11 shows the normalized execution time. Our NVM-aware data placement scheme can improve the performance of four applications by 6.9% to 14.5%, and our parallel optimization strategies can further improve applications performance by 4.1% to 15.8%. For memory-intensive applications such as PageRank, our optimization techniques are more effective for performance improvement because it introduces much more memory accesses during the relatively longer execution time.

To verify whether NGraph can mitigate the side effect of high NVM access latency on the application performance,

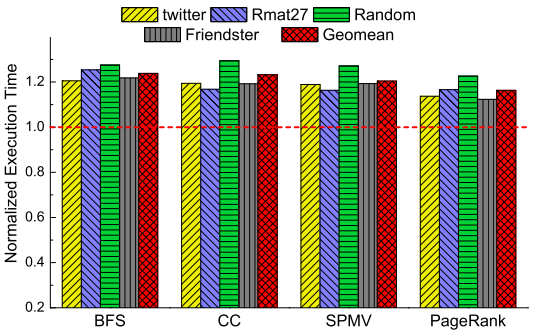

FIGURE 12. The execution time of applications running in a hybrid memory system and a DRAM-only memory.

we compare the execution time of NGraph running on a hybrid memory system and a DRAM-only system. Figure 12 shows the normalized experimental results, all relative to the DRAM-only system. We can find that NGraph results in 23.8%, 23.2%, 16.3% and 20.4% more execution time on average for BFS, CC, PageRank and SPMV, respectively. However, compared with Ligra running in the same hybrid memory system, NGraph can still improve application performance by 11.67%, 17.95%, 32.78% and 22.66%, respectively, as shown in Table 4. Thus, NGraph can significantly reduce the capacity of DRAM for large graph processing, while achieving DRAM-like performance (within 84%) by efficiently using a small size of DRAM.

C. BENEFIT OF USING HUGEPAGES

To demonstrate the benefit of using huge pages for large graph processing, we evaluate the execution time of NGraph with and without hugepage (2 MB) support for the twitter dataset, and also measure the impact of huge pages on TLB miss rate.

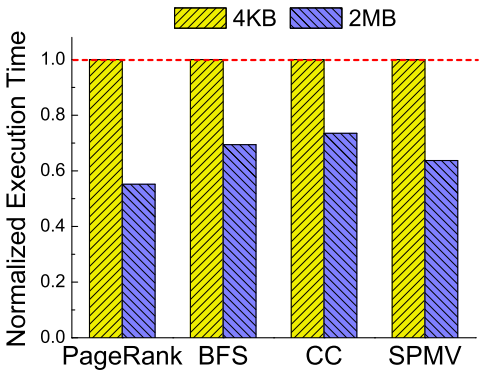

FIGURE 13. The normalized execution time using 2 MB and 4 KB memory pages.

As shown in Figure 13, NGraph can reduce application execution time by up to 44.8% using 2MB hugepage compared with 4KB pages. Also, huge pages can significantly reduce TLB misses. Table 5 shows that the TLB miss rate can be reduced by up to 63.38%. These results verify that hybrid memory systems using huge pages can significantly improve

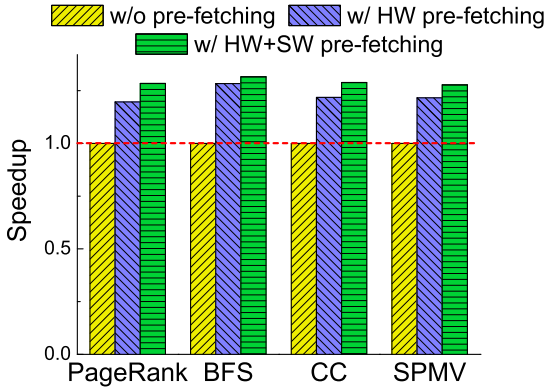

FIGURE 14. The performance speedup with and without data pre-fetching schemes.

TABLE 5. TLB miss rate of applications using 2 MB pages and 4 KB pages for the twitter dataset.

Applications	4 KB	2 MB	Reduction(%)
PageRank	32.84%	21.72%	33.86%
BFS	15.4%	5.65%	63.38%
CC	6.12%	3.43%	43.95%
SPMV	21.17%	13.47%	36.37%

the performance of processing graphs with large memory footprints.

D. BENEFIT OF DATA PRE-FETCHING

To demonstrate the effectiveness of data pre-fetching, we compare the execution time of NGraph without pre-fetching, with only hardware pre-fetching, and with both hardware pre-fetching and software pre-fetching. To further explain the benefit of data pre-fetching, we also measure LLC miss rates in different scenarios.

Figure 14 shows the performance speedup of different applications due to different pre-fetching schemes compared to the scenario without pre-fetching. Here, all experiments use the twitter dataset, and hugepage support and structure-aware data placement schemes are enabled. We can find that NGraph with hardware pre-fetching can lead to 19.67% to 28.94% performance improvement, and software pre-fetching for property data can further improve application performance by 3.24% to 8.74%. PageRank and SPMV benefit more from the software pre-fetching than BFS since they have more active vertices, which implies more memory accesses and larger room for performance improvement. Table 6 shows that hardware pre-fetching and software pre-fetching improve cache hit rate by up to 8.95% and 4.54%, respectively, and together they can improve cache hit rate by 8.57% to 10.98%.

E. OVERHEAD

Compared with other graph processing frameworks, NGraph mainly introduces two kinds of overheads: the additional time cost for graph partitioning and higher memory consumption. First, after loading the graph data into memory, NGraph needs to partition the graph by destination vertices and construct

TABLE 6. The LLC miss rates in different scenarios.

Applications	w/o pre-fetching	w/ HW pre-fetching	w/ HW+SW pre-fetching
PageRank	87.84%	81.82%	77.28%
BFS	62.36%	53.41%	51.38%
CC	58.53%	52.83%	49.95%
SPMV	65.24%	59.46%	56.59%

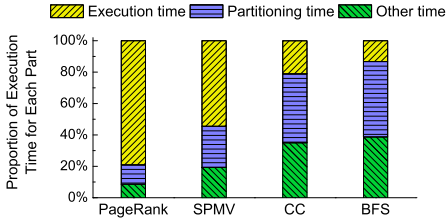

FIGURE 15. The proportion of graph partitioning time in the total execution time for different algorithms running the twitter dataset.

different sub-graphs for each core. Second, NGraph has replications of partial vertex structure data in sub-graphs, and thus increasing the memory consumption.

In order to analyze the time cost of graph partitioning, we break the total execution time of algorithms down into graph partitioning time, processing time and other time. Figure 15 shows the proportions of graph partitioning time for twitter dataset. We can find that graph partitioning only consumes a small proportion of total execution time for algorithms with a relatively long execution time (e.g., 12.35% for PageRank). However, for algorithms with short execution time, graph partitioning accounts for a large proportion (e.g. 48.01% for BFS). It should also be noted that once a graph is partitioned, the data can be used to run all algorithms, and thus can significantly reduce the execution time of these algorithms.

TABLE 7. The memory usage of NGraph (GB) and other systems.

graphs	NGraph	Ligra	Polymer
twitter	70.6	60.6	60.5
Rmat27	125.9	97.6	96.9
Random-local	131.8	89.6	89.2
Friendster	98.5	84.6	84.3

Table 7 compares the memory usage of NGraph and other systems when different graph datasets are partitiond into 8 sub-graphs. NGraph increases the memory consumption by about 16.5% to 47.1%. The increased memory consumption are mainly attributed to the replication of the vertex structure data. However, we can put the read-only structure data in large-capacity NVM, which is much cheaper than DRAM. We think the extra memory consumption is acceptable and beneficial for high-performance graph processing. In the future, we will explore compressed graph data structures to mitigate memory consumption of graph processing.

VI. RELATED WORK

There have been a few studies on graph processing in hybrid memory systems recently. The work presented by

Dulloor *et al.* [13] appears to be the first performance study on graph processing in NVM-based main memory. They use the hardware-based NVM emulator HMEP to analyze how different NVM bandwidth and latency settings affect the performance of different graph frameworks, and find that NVM causes performance degradation by 1.5 to 4 times compared to the DRAM-only system. Through a simple placement of data in the Graphmat framework, the performance approximates the DRAM-only system (only 20% slowdown). It also demonstrates that NVM can be used as main memory for graph processing when combining it with a small size of DRAM to build a hybrid memory system. Similar to [13], the work [14] evaluates the performance of graph processing system for different types of graph datasets in two cases: 1) NVM replaces DRAM as main memory and 2) DRAM is used as a buffer of NVM. Dulloor *et al.* propose X-mem [41], a data placement strategy for hybrid memory systems. It obtains the memory access characteristics of programs through offline analyzing, and then dynamically places and migrates data at runtime. They evaluate X-mem with a graph computing framework and demonstrate its effectiveness and high performance. Although these studies have presented the preliminary performance evaluation of graph processing in hybrid memory systems composed by DRAM and NVM, they have not yet explored the graph data access patterns and NVM features to make better data placement on DRAM and NVM.

HyVE [15] places vertices in DRAM and edges in NVM, and uses an on-chip SRAM as a buffer of vertex data to reduce the delay of random accessing to vertices. It adopts an interval-block based data partitioning [42] to improve data access locality. This work focuses on reducing the energy consumption of graph computing system rather than the performance. Our data placement strategy is inspired by HyVE [15], however, we further partition the vertex data into vertex property data and vertex structure data, and thus further reduce the consumption of DRAM and fully exploit the large capacity of NVM. Moreover, we also make some optimizations for graph processing in hybrid memory systems, such as software data pre-fetching and hugepage supporting.

Ligra [32] is a simple single-machine parallel graph computing framework in a shared-memory model. It provides two programming interfaces named *EdgeMap* and *VertexMap*, and a sparse and dense representation of vertex sets, as well as both push and pull execution modes. Under this framework, graph processing algorithms can be easily implemented. Our work is based on Ligra, and further enables several parallel optimizations for hybrid memory systems.

Polymer [43] is a NUMA-aware single-machine in-memory graph computing system. According to the fact that the bandwidth of sequential remote memory access is higher than that of local and remote random access, they proposed a differential data placement strategy to reduce remote memory accesses. Moreover, by using vertex replications, the remote random access is converted into a remote sequential access. We use a similar scheme to partition the graph, however, our goal is to avoid processor stalling caused by data contention and atomic operations.

Basak *et al.* [37] find that different types of data (such as graph structure data and property data) have different reuse distances. Traditional hardware pre-fetchers often ignore the types of graph data. Therefore, they design a graph data-aware hardware pre-fetcher called DROPLET, which physically decouples the pre-fetching of the graph property data and the structure data according to the reuse distance. Another work [44] finds that graph data structure and access patterns can be predicted in advance, and proposes a data-aware hardware pre-fetcher for graph processing. To avoid hardware modifications, we propose a data pre-fetching scheme combining the existing OS-supported hardware pre-fetching and data-aware software pre-fetching to reduce the memory access latency.

In distributed graph computing systems and multi-core based parallel graph processing systems, graphs should be partitioned into multiple sub-graphs. GraphGrind [36] analyzes the impact of graph partitioning on graph processing performance, and finds that graph partitioning causes extra graph traversal cost and load imbalance. By exploiting the compressed representation of graph data and adapting different graph partitioning methods to different graph applications, they can greatly improve the performance of graph processing. Our work is orthogonal to this proposal, we only store the vertices whose degree is not zero in sub-graphs to reduce the number of control instructions, and achieve better load balance through a work stealing strategy.

VII. CONCLUSION

This paper proposes NGraph, a parallel graph processing framework for hybrid memory systems composed of DRAM and NVM. NGraph improves the performance of graph processing by optimized hybrid memory management and CPU scheduling. First, NGraph reduces random data accesses and frequent updates on NVM based on different access patterns of graph data structures and NVM features. Second, NGraph avoids data competition among multi-core and eliminates atomic operations through graph partitioning and task decomposition, and thus reduce the time of CPU stalling. In addition, a CPU load balance scheme has also been proposed to minimize the time of parallel graph processing on multi-cores. Moreover, NGraph supports huge pages and data structure-aware software pre-fetching to further hide memory access latency. The experimental evaluation shows NGraph can achieve significant performance improvements over other state-of-the-art frameworks in hybrid memory systems. In the future, we will explore the proposed schemes in NGraph for distributed graph processing frameworks, and study dynamic placement and migration of graph data in a distributed shared hybrid memory pool.

REFERENCES

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2010, pp. 135–146.

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement.*, Hollywood, CA, USA, 2012, pp. 17–30.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[4] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: Unifying data-parallel and graph-parallel analytics," 2014, *arXiv:1402.2394*. [Online]. Available: https://arxiv.org/abs/1402.2394

[5] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement.*, Hollywood, CA, USA, 2012, pp. 31–46.

[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, New York, NY, USA, Nov. 2013, pp. 472–488.

[7] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He, "VENUS: Vertex-centric streamlined graph computation on a single PC," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Seoul, South Korea, Apr. 2015, pp. 1131–1142.

[8] H. Jin, P. Yao, and X. Liao, "Towards dataflow based graph processing," *Sci. China Inf. Sci.*, vol. 60, no. 12, Nov. 2017, Art. no. 126102.

[9] D. Niu, Q. He, T. Cai, B. Chen, Y. Zhan, and J. Liang, "XPMFS: A new NVM file system for vehicle big data," *IEEE Access*, vol. 6, pp. 34863–34873, 2018.

[10] F. Pellizzer, A. Benvenuti, B. Gleixner, Y. Kim, B. Johnson, M. Magistretti, T. Marangon, A. Pirovano, R. Bez, and G. Atwood, "A 90 nm phase change memory technology for stand-alone non-volatile memory applications," in *Proc. Symp. VLSI Technol., Dig. Tech. Papers.*, Jun. 2006, pp. 122–123.

[11] D. Zhu, Y. Li, W. Shen, Z. Zhou, L. Liu, and X. Zhang, "Resistive random access memory and its applications in storage and nonvolatile logic," *J. Semicond.*, vol. 38, no. 7, Jul. 2017, Art. no. 071002.

[12] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," in *Proc. 12th USENIX Conf. File Storage Technol.*, Santa Clara, CA, USA, 2014, pp. 33–45.

[13] J. Malicevic, S. Dulloor, N. Sundaram, N. Satish, J. Jackson, and W. Zwaenepoel, "Exploiting NVM in large-scale graph analytics," in *Proc. 3rd Workshop Interact. NVM/FLASH Operating Syst. Workloads*, New York, NY, USA, Oct. 2015, pp. 2:1–2:9.

[14] M. Shantharam, K. Iwabuchi, P. Cicotti, L. Carrington, M. Gokhale, and R. Pearce, "Performance evaluation of scale-free graph algorithms in low latency non-volatile memory," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Jun. 2017, pp. 1021–1028.

[15] T. Huang, G. Dai, Y. Wang, and H. Yang, "HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 973–978.

[16] H. Guo, L. Huang, Y. Lü, J. Ma, C. Qain, S. Ma, and Z. Wang, "Accelerating BFS via data structure-aware prefetching on GPU," *IEEE Access*, vol. 6, pp. 60234–60248, 2018.

[17] X. Wang, T. Luo, J. Hu, Z. Wang, and Y. Luo, "Evaluating the impacts of hugepage on virtual machines," *Sci. China Inf. Sci.*, vol. 60, no. 1, Nov. 2016, Art. no. 012103.

[18] B. N. Engel, J. Akerman, B. Butcher, R. W. Dave, M. DeHerrera, M. Durlam, G. Grynkewich, J. Janesky, S. V. Pietambaram, N. D. Rizzo, and J. M. Slaughter, "A 4-Mb toggle MRAM based on a novel bit and switching method," *IEEE Trans. Magn.*, vol. 41, no. 1, pp. 132–136, Jan. 2005.

[19] D.-S. Yoon, J. S. Roha, S.-M. Lee, and H. K. Baik, "Alteration for a diffusion barrier design concept in future high-density dynamic and ferroelectric random access memory devices," *Progr. Mater. Sci.*, vol. 48, no. 4, pp. 275–371, 2003.

[20] C. J. Lin, S. H. Kang,Y. J. Wang, K. Lee, X. Zhu, W. C. Chen, X. Li, W. N. Hsu,Y. C. Kao, M. T. Liu, and Y. Lin, "45 nm low power CMOS logic compatible embedded STT MRAM utilizing a reverse-connection 1T/1MTJ cell," in *IEDM Tech. Dig.*, Baltimore, MD, USA, Dec. 2009, pp. 1–4. doi: 10.1109/IEDM.2009.5424368.

[21] K.-J. Lee, B. H. Cho, W. Y. Cho, S. Kang, B. G. Choi, H. R. Oh, C. S. Lee, H. J. Kim, J. M. Park, Q. Wang, and M. H. Park, "A 90 nm 1.8 v 512 mb diode-switch pram with 266 mb/s read throughput," *IEEE J. Solid-State Circuits*, vol. 28, no. 1, pp. 150–162, Jan. 2008.

[22] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," *Commun. ACM*, vol. 53, no. 7, pp. 99–106, Jul. 2010.

[23] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid dram/nvm memory architectures," in *Proc. Int. Conf. Supercomput.*, New York, NY, USA, Jun. 2017, pp. 26:1–26:10.

[24] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, and J. Zhao, "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv:1903.05714*. [Online]. Available: https://arxiv.org/abs/1903.05714

[25] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, Jun. 2013, pp. 475–486.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and R. Sen, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.

[27] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors," in *Proc. 1st Int. Qemu Users' Forum*, Mar. 2011, pp. 29–30.

[28] M. T. Yourst, "PTLsim: A cycle accurate full system x86–64 microarchitectural simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, San Jose, CA, USA, Apr. 2007, pp. 23–34.

[29] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, New York, NY, USA, Apr. 2014, pp. 15:1–15:15.

[30] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Proc. 16th Annu. Middleware Conf.*, New York, NY, USA, Nov. 2015, pp. 37–49.

[31] Z. Duan, H. Liu, X. Liao, and H. Jin, "HME: A lightweight emulator for hybrid memory," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1375–1380.

[32] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, Feb. 2013, pp. 135–146.

[33] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*, New York, NY, USA, Apr. 2010, pp. 591–600.

[34] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: An efficient, low-cost system for concurrent graph processing," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, New York, NY, USA, Jun. 2014, pp. 227–238.

[35] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu, "Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2018, pp. 441–452.

[36] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing load imbalance of graph partitioning," in *Proc. Int. Conf. Supercomput.*, New York, NY, USA, Jun. 2017, pp. 16:1–16:10.

[37] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2019, pp. 373–386.

[38] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.

[39] *Graph500*. Accessed: Nov. 2010. [Online]. Available: http://www.graph500.org

[40] *SNAP*. Accessed: Jun. 2014. [Online]. Available: http://snap.stanford.edu/data/index.html

[41] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proc. 11th Eur. Conf. Comput. Syst.*, New York, NY, USA, Apr. 2016, pp. 15:1–15:16.

[42] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, Santa Clara, CA, USA, 2015, pp. 375–386.

[43] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," *SIGPLAN Not.*, vol. 50, no. 8, pp. 183–193, Jan. 2015.

[44] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proc. Int. Conf. Supercomput.*, New York, NY, USA, Jun. 2016, pp. 39:1–39:11.

WEI LIU received the B.S. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2014, where he is currently pursuing the Ph.D. degree in computer science and technology. His research interests include in-memory computing and graph processing.

HAIKUN LIU received the Ph.D. degree from the Huazhong University of Science and Technology (HUST), China, where he is currently an Associate Professor with the School of Computer Science and Technology. His current research interests include in-memory computing, virtualization technologies, cloud computing, and distributed systems. He was a recipient of the Outstanding Doctoral Dissertation Award in Hubei, China.

XIAOFEI LIAO received the Ph.D. degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005, where he is currently a Professor with the School of Computer Science and Technology. His research interests include system virtualization, system software, and cloud computing.

HAI JIN received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 1994, where he is currently a Cheung Kung Scholars Chair Professor of computer science and engineering. In 1996, he received the German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany. He was with the University of Hong Kong, from 1998 to 2000, and a Visiting Scholar with the University of Southern California, from 1999 to 2000. He has coauthored 15 books and published more than 600 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a member of the ACM. He received the Excellent Youth Award from the National Science Foundation of China, in 2001. He is the Chief Scientist of ChinaGrid, the largest grid computing project in China, and National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security.

YU ZHANG received the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), in 2016, where he is currently a Postdoctoral Researcher with the School of Computer Science and Technology. His research interests include big data processing, cloud computing, and distributed systems. His current topic mainly focuses on application-driven big data processing and optimizations.