ByteGAP: A Non-continuous Distributed Graph Computing System using Persistent Memory

Miaomiao Cheng[1,2], Jiujian Chen[1], Cheng Zhao[2,*], Cheng Chen[2], Yongmin Hu[2], Xiaoliang Cong[2], Liang Qin[2], Hexiang Lin[2], Rong Hua Li[1], Guoren Wang[1], Shuai Zhang[2] and Lei Zhang[2]

[1]Beijing Institute of Technology
[2]Douyin Vision Co., Ltd.

Abstract
Graph computing systems play a critical role in a variety of industrial applications. This study examines ByteDance's graph computing system workload, which challenges the conventional notion of a one-shot, lightweight graph computing task that can scale to trillions of edges. The workload includes both small and large-scale tasks separated by a 1000-second runtime threshold. The majority of the workload is dominated by small-scale tasks submitted arbitrarily, but with high time-sensitive requirements. Large-scale tasks make up the bulk of computing resources and occur periodically. Therefore, the graph computing system must be capable of pausing running tasks and prioritizing more critical ones. In this paper, we introduce ByteGAP, a non-continuous graph computing system that leverages PMEM's unique features, such as durability, byte-addressability, memory-like access, lower latency, and high capacity. The non-continuous approach uses checkpointing mechanisms to achieve effective fault detection and recovery. ByteGAP provides two key contributions: (1) lightweight distributed checkpointing based on PMEM, (2) efficient dual-mode PMEM management for optimizing PMEM read and write operations. Moreover, we present a comprehensive evaluation method that demonstrates the system's ability to handle the challenges associated with large-scale computing tasks. The findings lay the foundation for future research in distributed graph computing systems and advocate for a non-continuous approach to graph computing.

Keywords
graph, non-continuous graph processing, persistent memory

1. Introduction

Graph computing system is widely employed in industry[1, 2, 3], catering to a variety of application scenarios. In ByteDance, thousands of graph computing tasks run daily, playing a crucial role in various scenarios such as recommendations, fraud detection, and content auditing. To better understand the characteristics of real-world graph computing workloads, we collected data on tasks performed in ByteDance within a month. As illustrated in Figure 1, the task runtime distribution within ByteDance's graph computing system manifests distinct characteristics. It can be observed that graph computing tasks of different scales exhibit two types of characteristics. (1) Small-scale tasks, which can be completed within 1000 seconds, account for 90% of all tasks. This suggests that most tasks can be completed relatively quickly. (2) Large-scale tasks that take more than 1000 seconds to complete consume more than 90% of computing resources. This indicates that these tasks use the majority of computing resources.

ByteDance's workload reveals that besides the typical small-scale graph computing tasks,there are also large-scale tasks demanding significant resources. Large-scale graph computation tasks exhibit the following characteristics. (1) Extensive resource consumption and long computing duration. Large-scale graph computing tasks occupy 90% of system resources. In ByteDance,large-scale graph computations may take several hours. (2) Significant memory consumption. For large-scale graph computations in ByteDance, graph data can

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) —Workshop on Accelerating Analytics and Data Management Systems (ADMS'23), August 28 - September 1, 2023, Vancouver, Canada
*Corresponding author.
chengmiaomiao.123@bytedance.com (M. Cheng); jiujian@bit.edu.cn (J. Chen); zhaocheng.127@bytedance.com (C. Zhao); chencheng.sg@bytedance.com (C. Chen); huyongmin@bytedance.com (Y. Hu); congxiaoliang@bytedance.com (X. Cong); qinliang.touyi@bytedance.com (L. Qin); linhexiang@bytedance.com (H. Lin); rhli@bit.edu.cn (R. H. Li); wanggr@bit.edu.cn (G. Wang); zhangshuai.root@bytedance.com (S. Zhang); zhanglei.michael@bytedance.com (L. Zhang)


CEUR Workshop Proceedings (CEUR-WS.org)

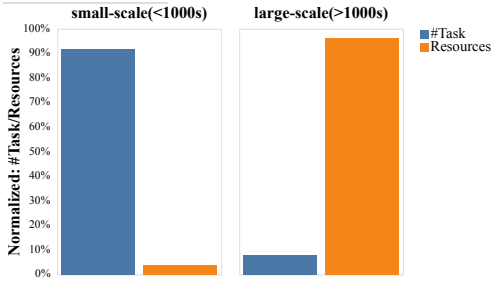


Figure 1: Percentage of the number of tasks and resources (running time multiplied by used core numbers) occupied for different task-scale in ByteDance graph computing workloads. Tasks are separated by a runtime threshold of 1000 seconds.

reach a trillion-scale, and at least 10TB of memory is required to store the pure graph data structure. These characteristics present challenges:

- Potential other task blockage due to lack of pause capability. Large-scale graph computing tasks cannot be paused, which may result in blocking other tasks when preemptive strategies are not in place [4, 5, 6].
- Lower tolerance for timeliness. Graph tasks must meet application timeliness, such as daily updated search tasks. For small-scale tasks, timeliness requirements can be ensured through a retry when encountering execution failures. However, large-scale tasks have higher retry costs, and when cluster failures occur during computation[7, 8],it may be impossible to satisfy the task's timeliness requirement.

Therefore, to address these challenges, graph computing systems must support checkpoint mechanisms and implement effective non-continuous graph computing techniques to ensure high availability within the graph computing cluster. The non-continuous graph computing system supports effective fault detection and recovery strategies and allows high priority tasks to interrupt low-priority. This approach offers significant advantages for large-scale graph computing systems.

Large-scale graph computing workloads require a large amount of storage space, with DRAM delivering exceptional performance and SSDs providing persistent checkpoint storage and extending DRAM. In addition, PMEM boasts unique characteristics, such as durability, byte-addressability, memory-like access, lower latency, and high capacity[9, 10], rendering it a better choice for constructing checkpoint-centric graph computing systems. However, deploying large-scale graph computation on PMEM presents a challenge. To ensure data consistency, graph computation requests write operations follow a certain order [11, 12]. However, keeping the writes to PMEM connected via memory bus in a certain order requires a special set of CPU instructions such as CLFLUSHOPT or CLWB followed by SFENSE, which has been proven to be extremely expensive[13, 14, 15].

In this paper, we propose a non-continuous distributed graph computing system based on PMEM, ByteGAP. To handle ByteDance-style graph computing workloads, we suggest a non-continuous strategy that preserves checkpoints for graph topology structures and vertex states during the iterative computing process. To increase efficiency, we further specialize in the design of PMEM checkpoints. To efficiently store topology structure and vertex states from various graph computing algorithms, we design a dual-mode PMEM management strategy that supports both fixed and variable data allocation. In summary, the main contributions of this paper are as follows.

- Lightweight distributed checkpointing. ByteGAP supports efficient, lightweight distributed checkpoints based on PMEM to facilitate effective non-continuous large-scale graph computation. Tailored to graph computation by considering both graph topology structure and vertex state data. We have developed a checkpoint saving mechanism using PMEM to improve system reliability and performance.
- Efficient dual-mode PMEM management. To accommodate multiple data patterns in the system, ByteGAP integrates a carefully-designed dual-mode persistent memory manager to handle and optimize PMEM read and write operations. To achieve high data access performance, the persistent memory manager employs an innovative dual-model PMEM allocator design, integrating both pool-based and log-based PMEM allocators to support fixed-size data and non-fixed-size data storage, respectively.

The rest of this paper is organized as follows. §2 describes the overall system architecture. We discuss the PMEM-based checkpoint-centric system design in §3. Experimental evaluation is reported in §4, and §5 reviews the related works. At last, we conclude our work in §6.

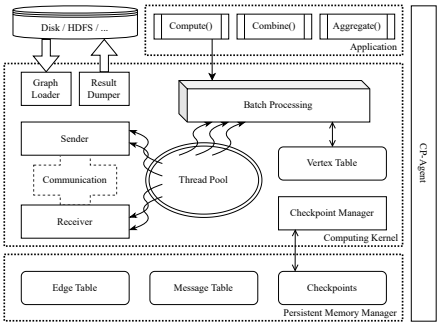


Figure 2: The system architecture of ByteGAP

2. System Architecture

We depict the architecture of ByteGAP in Figure 2, omitting some trivial connections between components and data for readability. The system can be seen as consisting of four parts (i.e., Application, CP-Agent, Computing Kernel and Persistent Memory Manager). Each node in the ByteGAP cluster will deploy only one process to perform graph computation while fully occupying all CPU resources.

The global runtime environment is managed by the CP-Agent. We build agents to spawn, rendezvous, and monitor workers across all nodes. They play a crucial role in achieving the checkpointing for our system. Beyond the system layer, user-defined graph algorithms are written as Application by specifying and injecting exposed interfaces. The Compute() method serves as the kernel for vertex-centric graph processing. It is defined by the users to implement the specific computation logic over each vertex along with its neighbors, following the classic 'think like a vertex' programming paradigm, based on various distributed graph algorithms. In addition, there are two optional functions that users can customize. The Combine() method is used to combine these messages targeting the same destination vertex locally, reducing communication overhead across nodes. The Aggregate() method periodically aggregates global intermediate results/statistics across nodes during computation, acting as a signal for global algorithm behaviors or system controls.

Computing Kernel is the core part of ByteGAP. It executes applications based on the BSP model, using user-defined methods in iterations. We further divide the worker into several components according to their functions. The Sender and the Receiver drives the tasks between computation and communication, including combining messages and constructing the Message Table in the Persistent Memory Manager. Batch Processing represents a collection of computing operations that we will discuss in §3.1. It invokes the computing method in multiple threads, generating messages and updating the value of vertices. Moreover, all threads used in processing and communication are managed by a Thread Pool. To perform a graph algorithm computation, workers first use the Graph Loader to load the graph from the Hadoop Distributed File System (HDFS) or local file system, where users can self-define the input data format of vertices and edges in the files. Loaded data will be partitioned among distributed workers and finally saved on DRAM for Vertex Table and persistent memory for Edge Table respectively, with contiguous storage to obtain the performance benefits of sequential read/write. The algorithm is conducted in rounds of supersteps, where every worker processes the computation in batches and shuffles messages to others in each superstep. When the last superstep is finished, the Result Dumper will write the results back to the storage devices in a user-defined format and terminate the program.

The underlying part is the Persistent Memory Manager (§3.3), which handles access to PMEM devices and persistent data management (including data storage, indexing, memory allocation, and garbage collection). Benefiting from the byte-addressable and good read/write performance of PMEM, we leverage it to store massive data in both Edge/Message Tables and CheckPoints. For the consideration of checkpointing, the Checkpoint Manager (§3.2) will generate and manage checkpoints happened at certain supersteps. When entering a failure recovery process, workers will use these checkpoints to restore themselves to continue distributed computing. We heavily engineered our system to be adapted for persistent memory, to achieve the best system performance with checkpointing.

3. System Design

We first introduce the computing kernel of ByteGAP in §3.1, which is the essential part of the system's execution. Then, we will discuss how to achieve fast recovery with checkpoints in §3.2, and discuss how to store and manage data in persistent memory in §3.3.

3.1. Computing Kernel

In this section, we will introduce the basic data layout and computing module of ByteGAP.

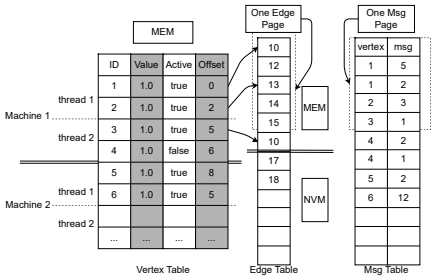


Figure 3: The data layout of ByteGAP

Page-based data layout. CSR is a widely used format for storing sparse graphs and matrices in graph systems [11, 16, 17]. We employ CSR to store graph data for the benefit of cache prefetching and efficient data access. Figure 3 depicts the data layout format of ByteGAP. All data associated with vertices is stored as a table in memory for efficient updating and fast access. Typically, each vertex takes one row in the array, storing the vertex ID, the vertex value, the active state, and an offset to the start position of its adjacency-list in the Edge Page. Each active state and vertex value is modifiable at run-time for algorithm processing. To adapt the CSR format on PMEM, we organize the adjacency lists of all vertices into a global linked page-list (i.e., Edge Table), where each page has a fixed length to store a fixed number of edges. Then, the offset of $vertex[i]$ and $vertex[i+1]$ can clearly indicate the start and end position (i.e., page number and offset on the page) of the adjacency-list of $vertex[i]$. Accordingly, to fetch the neighbors of one given vertex, we can iterate the pages on the Edge Table by sequentially reading on PMEM, which performs performance (i.e., latency and throughput) comparable to DRAM.

In addition, we also need to iterate over messages belonging to each vertex during the computation. Each message includes a source vertex ID and message data. Similar to the Edge Table, we also arrange all message data into a global list of linked pages on PMEM, called the Message Table.

Message passing. While computing, vertex can send messages to any other vertices to be processed in the next superstep. Message passing between vertices may appear across different workers. The two components of ByteGAP, Sender and Receiver, are used to manage this process between workers. Alongside the compute dispatcher, Sender grabs messages from compute threads and sends them out once the message count reaches a batch size. Meanwhile, Receiver accepts these streaming messages and saves them to the Message Table. It is expected that communication costs will overlap with computation in such pipelines.

To reduce the total amount of messages processed in each superstep, we take message properties-based approaches in different scenarios. When multiple messages sent to a vertex can be combined, a user-defined Combiner is adopted in both Sender and Receiver. In one sending batch, messages will be combined at the same destination. And after being received, messages will also be merged into one final message during generating the Message Table. Therefore, the indices of batches of messages can be easily calculated by the offsets in the Message Table for computing dispatchers in the next superstep.

We also use mirror vertices to reduce communication costs when each vertex sends the same message to its neighbors. A Remote Message Table is built alongside the resulting Message Table. When a vertex needs to send an identical message to multiple neighbors on a node, there will be a mirror vertex in the Remote Message Table on the target node. Message passing only occurs between the original vertex and its mirrors. Neighbors then fetch messages from the mirrors to fill the Message Table for the next superstep. For ease of programming, mirror vertices are transparent to algorithms and improvements are optional to reduce communication costs. We also take optimization for vertices with huge degrees. They will have mirror vertices in each node and the messages passing to them are first gathered locally after sending them out. To prevent concurrent write conflicts on such mirrors, we collect messages within each thread and combine them upon completion of each superstep. Based on the skewed distribution of graph vertices, we dynamically select the threshold for degrees, trading mirror memory usage for reduced global communication costs.

3.2. Lightweight Distributed Checkpointing

To facilitate the recovery of our computing workers, we construct checkpoints at periodic number of supersteps (called *CP*) that need to do checkpoint. For ease of access, Checkpoint Manager saves checkpoints as key-value pairs in the PMEM store for fast retrieving after process restart. The keys are generated based on both the identifiers of each system component and the current system superstep number, where the values are the specific states or computing data that need to be stored inside checkpoints.

Leader-follower checkpointing. Checkpointing occurs at the beginning of each *CP*, then each worker will enter into a specific routine to construct the checkpoint individually. These checkpoints exist on

the local persistent memory of each node, where we discuss its format in 3.3. Note that to ensure the integrity of the newly generated checkpoint before safely removing the previous checkpoint, all nodes should set up a global barrier before the atomic checkpoint swap. In addition, we assign a node to be the worker leader, for using it to generate the system's unique Last Checkpoint ID (LCI). LCI is a basic concept for the checkpoint mechanism, which indicates what the last global *CP* is. We only allow the leader to generate and store the LCI, and broadcast it to other followers during the recovery, to avoid inconsistent distribution, considering if one worker fails at checkpointing while the others successfully save the new checkpoint. Therefore, at the end of the checkpointing routine, each follower will send a completed notification to the leader after its checkpoints have persisted. And only after receiving all completed messages, the leader can update the LCI and acknowledge it to followers by notifying them that the previous checkpoint is out of date. When CP-Agents relaunch all workers, the leader will load the LCI from its persistent layer and broadcast it to all followers to restore the checkpointed data.

Lightweight checkpoints. After loading the graph, topologies, i.e. edges and partition information, are involved in every computation and message sending. They will not change during the computation of many algorithms, so that only one checkpoint for them is required before the first superstep of computing. And at each superstep, as described in §3.1, the computing dispatchers need to use the updated states and messages received from local vertices from the last superstep, which are required to be saved in checkpoint at each *CP*. We adopt a lightweight checkpointing approach that is dedicated to reducing the extra overhead based on whether the data is mutable or not and which memory they mainly use. While edges are usually immutable and persist only once at the beginning, vertex-related data, i.e., Vertex Table and Message Table, take up the bulk of the workload when checkpointing.

For vertices stored in DRAM, we serialize them down to the PMEM layer in parallel to make full use of PMEM bandwidth at minimum cost. The message checkpoint is thoughtfully considered in other works [18, 19] as its size can be proportional to the number of edges. However, we have already placed messages in the PMEM as Message Table before storing checkpoints. We only need to treat them as checkpoints and keep them available until the next checkpoint is generated.

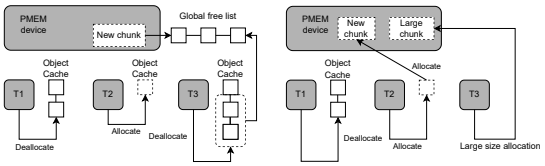
Specifically, we use three buffers in Message Table to store received messages. One buffer is used for regular supersteps, it will be overwritten by new incoming messages iteratively at each superstep. The other two buffers work interchangeably for checkpointing rounds, in which case it can ensure that the latest checkpointed messages are always available even as we write the next checkpoint. We only need to store an index to track which buffer holds the latest messages at each *CP*. Therefore, the overall checkpointing overhead is lightweight due to the spread of data in our system. And during recovery, we only need to restore the indices of the key-value pairs in persistent memory, as we will describe below.

Checkpointing with agent. Inspired by Torch Distributed Elastic [20], which enables PyTorch to run in a fault-tolerant manner, we achieve our CP-Agent to manage distributed computing workers in ByteGAP. To assemble the essential environment for communication and monitor the running status of each computing process. It is commonly assumed that as cluster size increases, the chance of the entire system failing due to the failure of a single worker increases as well. Therefore, how to solve retrieval in a distributed graph computing system becomes a primary concern in our design. The basic idea is to re-spawn working processes with the help of CP-Agent, where CP-Agent ensures that the entire program can be fully executed or terminated normally. If some processes encounter fatal errors during the computation, CP-Agent will charge the recovery of process rank, communication environment, and worker monitoring.

3.3. Dual-Mode Persistent Memory Management

We use the Persistent Memory Manager (PMM) to manage ByteGAP persistent memory space for high-performance PMEM access. PMM mainly supports data storage for Edge Table, Message Table, and checkpoints. We conduct *Page* as the minimum allocation unit for PMEM with a fixed length. A series of pages can form the Edge Table and the Message Table as a linked page-list. To facilitate optimal data access and allocation, we set the page size to be the OS physical page size. While checkpointing data can be arbitrary in length, PMM also needs to support dynamic memory allocation. Further, as we mentioned in §3.2, PMM should organize stored elements as key-value pairs, to better categorize different checkpointed components for their on- and off-load.

Dual-mode allocator. Based on the above reasons, we adopt both pool-based and log-based allocators for fixed and non-fixed data storage, respectively,



(a) Pool-based allocator (b) Log-based allocator

Figure 4: PMEM allocators

to achieve efficient data management (i.e., allocation, deletion, garbage collection) on PMEM. To meet fixed size memory allocation requirements, the pool-based allocator implements a shared memory management strategy as follows. As illustrated in figure4a, when a thread attempts to allocate an object, it first searches through its own private object cache to get a free object. If no free object is left, it will fetch free objects from a global free-list to refill its own local object cache. Only if there are not enough free objects in the global free-list, the thread will allocate a new chunk from the PMEM device, and using this new chunk to generate free objects to refill the object cache. When a thread wants to deallocate an object, the thread first sets the tombstone bit of the deleted object and then adds the object to the private object cache. If the thread finds too many free objects (e.g., twice the bulk load size) in the object cache, it will move some free objects from its private free-list to the global free-list. Obviously, such a strategy can provide a pool-based allocator with excellent data locality and high performance for data allocation. While, for the log-based allocator depicted in figure4b, it manages the memory space in chunks, but each chunk may have a different size. The log-based allocator charges the allocation of objects with variable lengths, hence it needs to migrate data from existing chunks to a newly allocated chunk during data defragmentation to improve data locality. Specifically, when the allocator finds a highly fragmented chunk, it first scans the chunk to locate all valid records of the chunk, and then copies those valid records to a new chunk for seeking tighter data arrangement. Next, the allocator maps the memory access of the migration record to the new memory address, and finally reclaims the memory space of the old chunk.

Persistent data lifetime. For persistent data stored in PMM belonging to different system components, we should use different types to manage their lifetime at runtime. For example, when a new checkpoint is generated, the old checkpoint does not need to be maintained. While deleting an outdated checkpoint is not an atomic operation, a failure may occur during asynchronous deletion and CP-Agent will restart all workers after that. Then, in the progress of PMM's recovery through PMEM devices scanning, we can directly drop those useless records and remain the useful records based on their lifetime type. Therefore, we classify all data managed by the PMM into three types of lifetime: Keep, Delete, Exclusively Keep.

- Keep: Persistent data of this type should remain valid during the whole runtime of the job. For example, Edge Table is read only after it is generated, so it belongs to Keep type.
- Delete: Some data should be burned after reading and will not be saved to checkpoint, e.g. temporary Message Table data. These data are marked as Delete, which will not be recovered when the process restarts.
- Exclusively keep: All checkpoint-related data is indexed by the checkpoint ID, which refers to its associated checkpoint. This data belongs to the Exclusively Keep type, meaning it needs to be saved with the ICI index.

PMM supports data recovery when workers perform unexpected exits. Note that we reduce the recovery process overhead by using PMEM runtime storage. Unlike other systems [1, 21, 22, 19], which recovers the Message Table via recomputing or reloading from disk log, we directly store and fetch the Message Table in PMEM at runtime without any further operations.

4. Experiments

ByteGAP aims to provide scalable non-continuous support for graph computing. This section reports the results of our evaluation of ByteGAP in various environments.

4.1. Experimental Setup

Testbed. We evaluated ByteGAP on a cluster of up to 16 machines, each equipped with two Intel Xeon Platinum 8260 CPUs (48 cores in total) and 128GB of DRAM. Each node also contains 320GB of SSD and 512GB of Optane DC PMM, which will be configured in memory mode and app-direct mode depending on the situations in our evaluation.

Datasets. We adopt four real-world datasets for our evaluation. Table 1 shows the statistics of these four graphs: Twitter and Friendster from SNAP [23], and UK-2007 and UK-union from LWA [24]. Twitter and Friendster are the graphs that belong to the

Table 1
Dataset statistics

Dataset	$\|V\|$	$\|E\|$	Avg. *Degree*
Twitter	41,652,230	1,468,364,884	4.4
Friendster	65,608,366	1,806,067,135	16.0
UK-2007	105,896,555	3,738,733,648	35.3
UK-union	133,633,040	5,475,109,924	41.0

social networking domain, while UK-2007 and UK-union are both Internet graphs. The number of edges in each graph reaches the billion scale, and the degrees of Internet graphs are more than social graphs. In addition, we generated a weighted graph for each dataset by randomly assigning an integer from 0 to 100 to each edge.

Algorithms. We evaluated our system by four representative algorithms: PageRank, Connected Components (CC), Breadth First Search (BFS) and Shortest Single Source Path (SSSP). We ran these algorithms against other out-of-core systems for comparison. And since PageRank is implemented in all systems, we use it as a core test for several experiments.

4.2. Checkpointing Evaluation

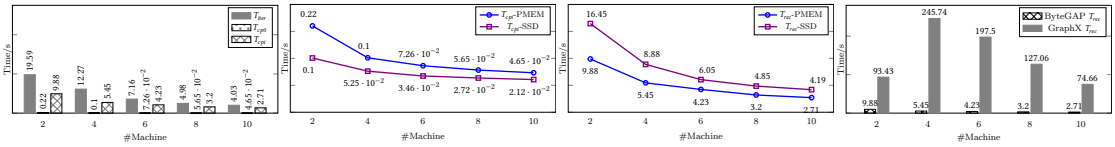
We also examined the pros and cons of checkpointing in ByteGAP. To evaluate the elapsed time on checkpointing and recovery, we ran PageRank on the Twitter dataset and configured the system to write checkpoints at each 10 superstep. In the next superstep after saving a checkpoint, one of computing kernels will kill itself to simulate a worker stopping. The PageRank algorithm ensures that each superstep has the same workload of computation and can generate the same number of messages. After the worker stops, CP-Agent will relaunch and guide the system to begin recovery. There are four important phases in failure and recovery. We represent the time metrics of this process as follows:

- T_{iter}: the time of each superstep without saving checkpoints, used to indicate the elapsed time of normal execution for comparison with checkpointing overhead. In our settings, this is the average time of ten supersteps.
- T_{cp0} and T_{cpi}: the time to write all needed checkpoints when the worker executes normally. We use T_{cp0} to denote time of the initial checkpointing before first superstep and use T_{cpi} as the time of saving checkpoints at the latest *CP* superstep.
- T_{rec}: the time of the recovery by checkpoints. As our agents will also kill survival workers to perform relaunching, these times represent the maximal time of recovery across all workers.

To fully evaluate performance, we recorded these time metrics for ByteGAP on 2 to 10 machines as shown in Figure 5a-figure 5c. There is one CP-Agent on each machine to keep PMM and Computing Kernel healthy, and each worker will only save checkpoints to and load back from its local storage. Figure 5a depicts the time metrics for the first two phases as described above. The time T_{cp0} of writing the initial checkpoint only takes up once in the whole running time. It is not as much scalable as T_{iter} because it includes time of saving the topology of graph. However, the additional cost for each *CP* superstep is always much smaller than normal execution. As the number of machines increases, the average execution time required for a superstep decreases proportionally. And T_{cpi} also decreases and stays at about one-third of T_{iter}.

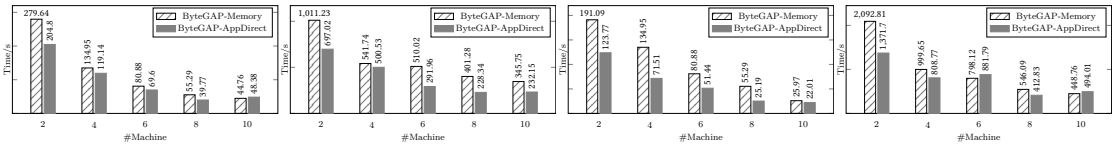
We ran ByteGAP to evaluate checkpoint performance. The PMM used SSD as the underlying storage, though our system supports this use case, it lacks optimization for PMEM to improve I/O efficiency. Figure 5b shows that checkpoints were faster on SSDs than PMEM. This is because the operating system has optimizations for SSD writes, including caching, which PMEM cannot currently use. We also ran ByteGAP in a testbed where the PMM uses a normal path on SSD as the underlying storage to inspect the time metrics about recovery. Although naturally supported in our system, such a use case lacks the specialized design for PMEM, which can improve I/O efficiency. Figure 5c illustrates the time of recovery with different machines. Each of the reported time decreases exponentially as the number of machines grows. Starting with a few machines, T_{rec} of running on PMEM is much less than running on SSDs. However, when there are more machines being involved, both of them perform close metrics. Despite hardware limitations, ByteGAP benefits from PMEM features and does not lose performance when using SSDs on multiple machines.

We also evaluated ByteGAP versus GraphX (Spark 3.0) on 2 to 10 PMEM-equipped servers to compare checkpoint performance. GraphX is a widely used graph processing system. For GraphX, we killed processes during computation to simulate task failure and recovery. T_{rec} is obtained by subtracting the time of normal running tasks from failed and recovered tasks. T_{iter} is also represented by the average time of the first 10 iterations. Since GraphX checkpointing time T_{cpi} involves synchronous copies



(a) Time of checkpoints (b) Time of checkpoints (c) Time of recovering worksers (d) ByteGAP vs GraphX for T_{rec}

Figure 5: Time metrics of checkpointing



(a) PageRank-Twitter (b) PageRank-UKunion (c) CC-Twitter (d) CC-UKunion

Figure 6: Results of scalability evaluation

and is difficult to measure, the overhead is also small. This experiment did not statistically analyze T_{cpi} but mainly focused on measuring the time of T_{rec}. Under 10 machines, we found that the average time required for a single iteration of GraphX is 27.1 seconds, while T_{rec} of GraphX takes 23.5 seconds. The average time required for a single iteration of ByteGAP is only 4.3 seconds, while T_{rec} takes only 2.7 seconds. ByteGAP achieved much shorter T_{iter} and T_{rec} than GraphX. For T_{rec} time comparison with 2 to 10 machines is shown in figure 5d, ByteGAP is much faster than GraphX for all numbers of machines. Because GraphX uses Spark's checkpoint function, only a portion of the computing resources are used after recovery in the beginning, so when there are too many computing nodes, the entire resources cannot be used immediately after recovery, which can lead to resource waste.

4.3. Scalability

We examine the scalability of ByteGAP in this section to evaluate the relationship between execution time and cluster size. Intuitively, processing performance can be improved by increasing the number of machines. Since persistent memory can be configured as Memory mode and App Direct mode, we evaluated scalability by the running time in both configurations with up to 10 machines. Specifically, we adopted two datasets, Twitter and UK-union, and used 2, 4, 6, 8, and 10 machines to run PageRank and CC algorithms with two modes respectively. Figure 6 reports the results of our evaluation. It takes different time when using Memory mode and App Direct mode as depicted. Therefore, in almost all experiments, ByteGAP using App Direct mode delivered a smaller execution time compared to using Memory mode with the same algorithm and the same number of machines. As in the case that computing connected components on the Twitter dataset in 8 machines, it takes 25.19s with App Direct mode and 55.29s with Memory mode. This indicates two reasons: (1) the data storage we designed is more suitable for I/O with direct access, and (2) using memory mode may result in memory swapping too frequently, reducing ByteGAP's performance.

Our experiments verify that increasing the number of machines can reduce ByteGAP execution time. The system benefits from more CPU resources and higher parallelism in both configurations as the number of machines increases, as more machines can provide more CPU resources. For example, when running PageRank on the Twitter dataset, using Memory mode takes 279.64s at 2 machines and only 55.29s at 8 machines, and using App Direct mode takes 204.8s at 2 machines and only 39.77s at 8 machines, depicting more machines needing to use less time. This means that we can achieve better performance of our system by adding more machines to improve parallelism. However, adding more machines to the cluster may involve new overhead, such as system consistency guarantees, network communication overhead across machines, and hardware access overhead, all of which tend to degrade system performance. Experiments show that the performance of using App Direct mode is worse than using Memory mode at 10 machines while computing PageRank on Twitter and CC on UK-union. It also indicates that performance degradation using App Direct mode occurs earlier than using Memory

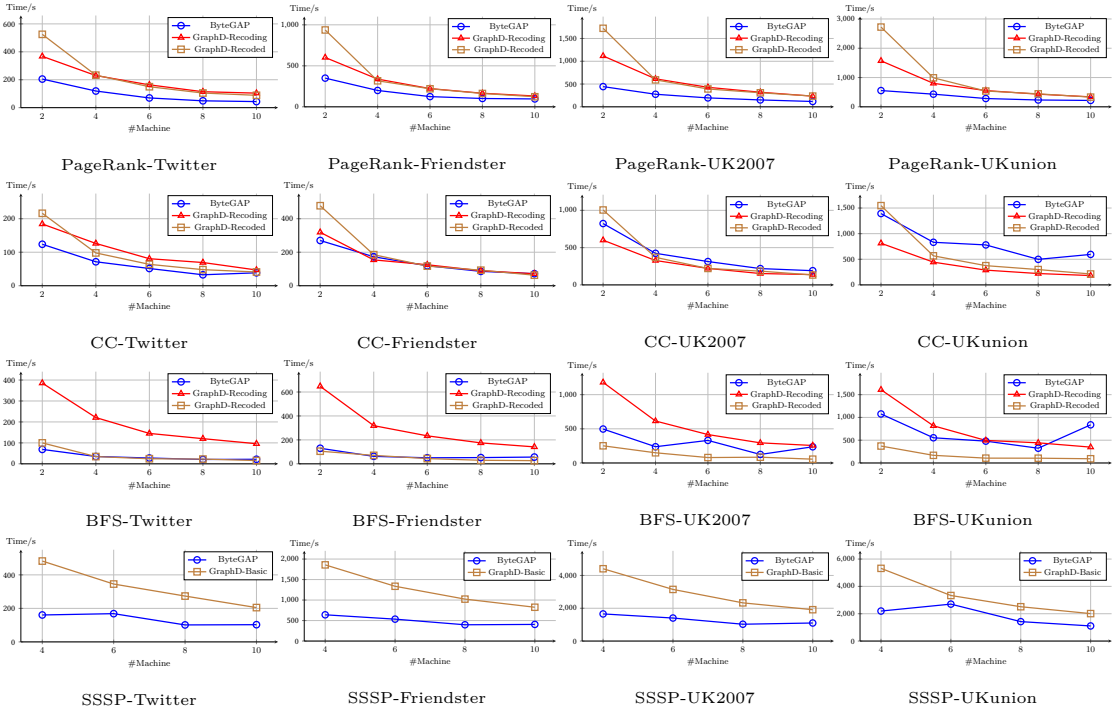
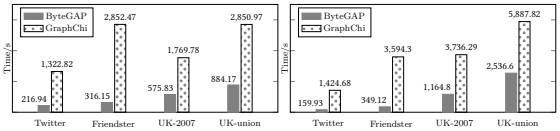


Figure 7: ByteGAP vs. GraphD on the cluster



(a) PageRank (b) CC

Figure 8: ByteGAP vs. GraphChi

mode. For example, running PageRank on Twitter dataset using App Direct mode takes 69.59s with 6 machines and 39.77s with 8 machines, but it takes 48.38s with 10 machines. However, this did not happen while using Memory mode. As a result, I/O performance remains a bottleneck in the storage layer of the distributed system with sufficient computing resources.

4.4. Comparison with Out-of-Core Systems

In this subsection, we compare the computing performance of ByteGAP with other out-of-core systems because they have similar use cases with external storage. GraphChi [17] and GraphD [25] are well-known disk-based systems running in a single machine and distributed environment, respectively. We have mounted the Optane PMM to an independent path so that it can be treated as a stand-alone filesystem by OS. All systems are configured to use this path and leave all DRAM as the main memory. While other systems accept the edgelist file format, input graphs for GraphChi are preprocessed and stored directly on PMEM. We compare the execution time of ByteGAP with the above systems by running PageRank, CC, BFS, and SSSP on all datasets individually.

We first report the execution time compared to GraphChi as shown in Figure 8. There is a different emphasis on the design of single-machine and distributed systems, such as the shared-memory or shared-nothing architecture and the synchronization mechanism between supersteps. We sequentially ran ByteGAP from single machine to multiple machines for comparison. To better measure the computing performance of the systems, we recorded the running time around the iteration of graph algorithms only for each system. We start with the time of computing PageRank and connected components in one machine by ByteGAP and GraphChi, as depicted in Figure 8a and Figure 8b. Consequently, ByteGAP outperforms GraphChi on all datasets. The Parallel Sliding Windows for accessing disks in

GraphChi is not suitable for PMEM, and GraphChi does not keep values of all vertices in memory like ByteGAP do. As a result, the execution time of ByteGAP is less than half that of the others for both algorithms.

We then report the time of ByteGAP and GraphD on 2 to 10 machines to examine execution in a distributed environment. Figure 7 lists the running time of four algorithms on all datasets. We chose the Recoded version of GraphD because it achieves better performance than the base version. Running time is denoted as GraphD-Recoded in each figure and we also count the required time of the GraphD-Recoding process that is needed beyond the graph loading. Because they do not provide the Recoded version of the SSSP algorithm, we fall back on running the basic version and only compare the running time on 4 to 10 machines because it is much slower. As depicted in the first three rows of Figure 7, ByteGAP ran faster than GraphD-Recoded on all datasets for PageRank and Twitter and Friendster datasets for CC. However, the disparity between them decreases as the number of machines increases. Also ByteGAP becomes a bit slower when running on the UK-2007 and UK-union datasets for CC. However, taking the overhead of the additional recoding process into account, it can be assumed that ByteGAP gets better performance than GraphD on all different numbers of machines. Moreover, it is significantly faster than the basic version of GraphD for computing SSSP as depicted in the last row of Figure 7. On the other hand, we observe that some spikes appear on the line chart of ByteGAP. The reason could be that the partitioning strategy we used (i.e., chunk-based partitioning) may lead to higher communication costs with the increase of machines involved, and ultimately affect overall execution time.

5. Related Works

Recently, many graph computing systems have been proposed following the vertex-centric programming model, which allows each vertex to access the received messages from the last superstep and send new messages to its neighbors after computing the updated value. Following Google's Pregel system [1], many Pregel-like open-source systems have been proposed, including Giraph [26], Pregel+ [27], GraphX [2], and so on. Like Pregel, these systems maintain the entire graph in the main memory during the computing procedure. GraM [28] is a new system architecture that exploits the benefits of multi-core and RDMA, where communication overhead is an essential concern. SHMEM-Graph [29] synthesizes several optimizations to address both computational and communication imbalances. Nevertheless, with an eye on the industry, only a few graph computing systems have been widely used in business.

6. Conclusions

We present ByteGAP, a non-continuous distributed graph computing system using persistent memory. ByteGAP focuses on large-scale graph computing in industrial scenarios, which often demands a balance between system performance and resource cost. We propose a dual-mode allocator for managing data in persistent memory. Leveraging the persistent memory manager with a lightweight checkpointing method, ByteGAP supports massive data computation and fast recovery. Extensive experimental results on several large real-world graphs show that ByteGAP takes full advantage of persistent memory and achieves competitive performance compared to other out-of-core systems.

References

[1] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: SIGMOD, 2010, pp. 135–146.

[2] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, I. Stoica, Graphx: Graph processing in a distributed dataflow framework, in: OSDI, 2014, pp. 599–613.

[3] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, GRAPE: parallelizing sequential graph computations, Proc. VLDB Endow. 10 (2017) 1889–1892.

[4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th annual Symposium on Cloud Computing, 2013, pp. 1–16.

[5] Z. Rejiba, J. Chamanara, Custom scheduling in kubernetes: A survey on common problems and solution approaches 55 (2022). URL: https://doi.org/10.1145/3544788. doi:10.1145/3544788.

[6] T. Gonzalez, S. Sahni, Preemptive scheduling of uniform processor systems, Journal of the ACM (JACM) 25 (1978) 92–101.

[7] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, M. Annavaram, Check-n-run: a checkpointing system for training deep learning recommendation models, in: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 2022, pp. 929–943.

[8] T. D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM (JACM) 43 (1996) 225–267.

[9] L. Benson, L. Papke, T. Rabl, Perma-bench: Benchmarking persistent memory access, Proc. VLDB Endow. 15 (2022) 2463–2476. URL: https://www.vldb.org/pvldb/vol15/p2463-benson.pdf.

[10] S. Gugnani, A. Kashyap, X. Lu, Understanding the idiosyncrasies of real persistent memory, Proc. VLDB Endow. 14 (2020) 626–639. URL: http://www.vldb.org/pvldb/vol14/p626-gugnani.pdf. doi:10.14778/3436905.3436921.

[11] X. Zhu, W. Chen, W. Zheng, X. Ma, Gemini: A computation-centric distributed graph processing system, in: OSDI, 2016, pp. 301–316.

[12] J. Shun, G. E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in: A. Nicolau, X. Shen, S. P. Amarasinghe, R. W. Vuduc (Eds.), ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013, ACM, 2013, pp. 135–146. URL: https://doi.org/10.1145/2442516.2442530. doi:10.1145/2442516.2442530.

[13] C. Chen, J. Yang, M. Lu, T. Wang, Z. Zheng, Y. Chen, W. Dai, B. He, W.-F. Wong, G. Wu, Y. Zhao, A. Rudoff, Optimizing in-memory database engine for ai-powered on-line decision augmentation using persistent memory, Proc. VLDB Endow. 14 (2021) 799–812. URL: https://doi.org/10.14778/3446095.3446102. doi:10.14778/3446095.3446102.

[14] L. Benson, H. Makait, T. Rabl, Viper: An efficient hybrid pmem-dram key-value store, Proc. VLDB Endow. 14 (2021) 1544–1556. URL: http://www.vldb.org/pvldb/vol14/p1544-benson.pdf. doi:10.14778/3461535.3461543.

[15] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, J. Shu, Flatstore: An efficient log-structured key-value storage engine for persistent memory, in: J. R. Larus, L. Ceze, K. Strauss (Eds.), ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020, ACM, 2020, pp. 1077–1091. URL: https://doi.org/10.1145/3373376.3378515. doi:10.1145/3373376.3378515.

[16] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. M. Hellerstein, Graphlab: A new framework for parallel machine learning, CoRR abs/1408.2041 (2014).

[17] A. Kyrola, G. E. Blelloch, C. Guestrin, Graphchi: Large-scale graph computation on just a PC, in: OSDI, 2012, pp. 31–46.

[18] Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, B. M. Tudor, Fast failure recovery in distributed graph processing systems, Proc. VLDB Endow. 8 (2014) 437–448.

[19] D. Yan, J. Cheng, H. Chen, C. Long, P. V. Bangalore, Lightweight fault tolerance in pregel-like systems, in: ICPP, 2019, pp. 69:1–69:10.

[20] Torch distributed elastic, 2021. https://pytorch.org/docs/stable/distributed.elastic.html.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs, in: OSDI, 2012, pp. 17–30.

[22] S. Salihoglu, J. Widom, GPS: a graph processing system, in: SSDBM, 2013, pp. 22:1–22:12.

[23] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, 2014.

[24] P. Boldi, S. Vigna, The WebGraph framework I: Compression techniques, in: WWW, 2004, pp. 595–601.

[25] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, C. Zhang, Graphd: Distributed vertex-centric graph processing beyond the memory limit, IEEE Trans. Parallel Distributed Syst. 29 (2018) 99–114.

[26] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One trillion edges: Graph processing at facebook-scale, Proc. VLDB Endow. 8 (2015) 1804–1815.

[27] D. Yan, J. Cheng, Y. Lu, W. Ng, Effective techniques for message reduction and load balancing in distributed graph computation, CoRR abs/1503.00626 (2015).

[28] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, L. Zhou, Gram: scaling graph computation to the trillions, in: SoCC, 2015, pp. 408–421.

[29] H. Fu, M. G. Venkata, S. Salman, N. Imam, W. Yu, Shmemgraph: Efficient and balanced graph processing using one-sided communication, in: 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2018, pp. 513–522.