Excavating the Potential of Graph Workload on RDMA-based Far Memory Architecture

Jing Wang*, Chao Li*†, Taolei Wang*, Lu Zhang*, Pengyu Wang*, Junyi Mei*, Minyi Guo*†
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
†Shanghai Qi Zhi Institute, Shanghai, China
Email: {jing618, sjtuwtl, luzhang, wpybtw, meijunyi}@sjtu.edu.cn, {lichao, guo-my}@cs.sjtu.edu.cn

***Abstract*—Disaggregated architecture brings new opportunities to memory-consuming applications like graph processing. It allows one to outspread memory access pressure from local to far memory, providing an attractive alternative to disk-based processing. Although existing works on general-purpose far memory platforms show great potentials for application expansion, it is unclear how graph processing applications could benefit from disaggregated architecture, and how different optimization methods influence the overall performance.**

In this paper, we take the first step to analyze the impact of graph processing workload on disaggregated architecture by extending the GridGraph framework on top of the RDMA-based far memory system. We design Fargraph, a far memory coordination strategy for enhancing graph processing workload. Specifically, Fargraph reduces the overall data movement through a well-crafted, graph-aware data segment offloading mechanism. In addition, we use optimal data segment splitting and asynchronous data buffering to achieve graph iteration-friendly far memory access. We show that Fargraph achieves near-oracle performance for typical in-local-memory graph processing systems. Fargraph shows up to 8.3× speedup compared to Fastswap, the state-of-the-art, general-purpose far memory platform.

***Index Terms*—far memory, RDMA, graph processing**

I. Introduction

Today's various graph applications demand better memory performance at different graph scales [18], [32], [36], [39], [40]. In the past, most graph applications can be processed by a single-node system given the relatively small size of the graph in existing in-memory graph frameworks [29], [39], [40]. Distributed graph frameworks are required only for very large-scale data analytic problems due to the communication overhead [24], [35], [41]. Nevertheless, as shown in Figure 1-(a), many graph frameworks mainly focus on medium-sized graphs (from 1GB to several hundreds of GB) [17], [36], [42]. Although current out-of-core graph computing frameworks could handle medium-sized graphs with external storage, they suffer performance degradation due to the I/O bottleneck.

In addition, to process workload with various data input in the cloud, an important trend is to build disaggregated memory pools and enable *far memory* (i.e., remote main memory) accesses [5], [8], [20], [27]. In this case, memory-consuming programs like graph applications can easily scale out by oversubscribing memory if the local server has limited capacity. Meanwhile, with high-speed network protocols such as Remote Direct Memory Access (RDMA) [9] and Compute Express Link (CXL) [7], far memory access can achieve near-DRAM performance, as shown in Figure 1-(b). With appropriate memory management, better system utilization can be achieved. Such a far memory architecture has shown great promise in accommodating medium-sized graph processing applications. Consequently, it is expected to be a good complement to traditional single-node systems and distributed systems (detailed in Section II).

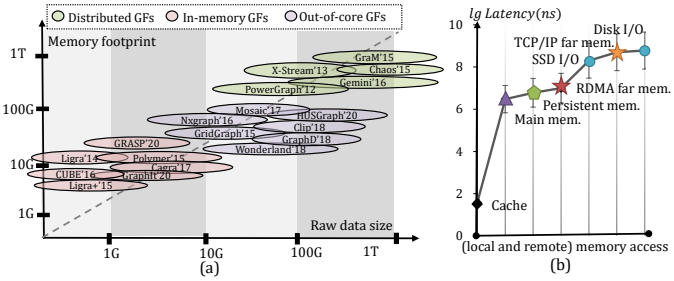

Fig. 1. (a) The case of memory usage expansion and the raw graph size of existing graph processing frameworks. (b) The data transfer duration of different memory access scenarios.

Nevertheless, simply performing memory offloading of graph processing on far memory architecture may not provide the best performance. A straightforward approach of far memory outspreading is to replace the original swap space with the far memory space [1], [12], [14], [16], without changing the strategy of page mapping and reordering in the original single-node graph framework. These works often limit local memory usage to trigger page faults and leverage high-speed network interfaces like RDMA to access far memory space. In other words, when the upper-layer application framework intends to access the far memory, it passively leaves all the pressure of deciding thrown-out parts to the OS kernel. Although this type of design is transparent to the application, it brings significant context switching overhead [1], [11], [12]. To reduce the above system overhead, recent studies attempt to build a user-level runtime to reduce kernel overhead [3], [25]. However, they are not aware of the workload characteristics and they may miss performance optimization opportunities when running graph workload on far memory (detailed in Section III).

The key opportunity of optimizing graph workload on far memory comes from two aspects: 1) the distinctive data segments and 2) the iterative execution model. First, a graph processing program features a group of data segments with

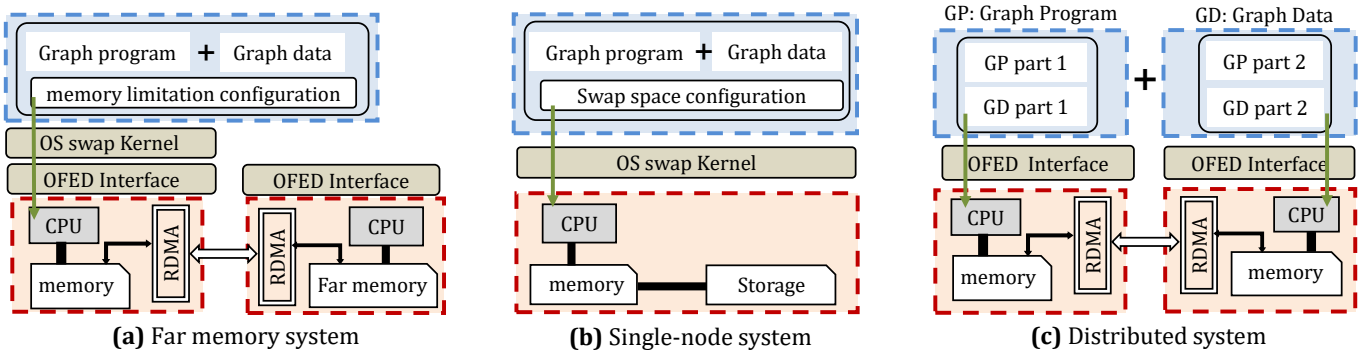

Fig. 2. Different architectures that can be leveraged for graph processing. (a) Far memory with modified swap mechanism of OS. (b) Single-node system with storage involved. (c) Distributed system communicated through RDMA.

distinctive characteristics [22], [29], [40]. On one hand, the size of different data segments may vary. Some data segments containing write-intensive vertices can be much smaller than others that maintain read-only edges. On the other hand, data segments in graph programs have different memory access patterns. Some data segments are requested frequently while some are accessed only once. Therefore, we need to determine the appropriate data segments that should be moved out (to far memory). Second, graph applications generally have many iterations that update vertex values by continuously searching graph data [18], [29], [42]. In each iteration, the program has to wait for the graph data to be fetched. Typically, the system fetches data by either propagating the current node value to neighbors through outgoing edges (push-based scheme) or gathering values from neighbors through incoming edges (pull-based scheme). In the existing far memory environment, the above data fetching operations may suffer frequent interruptions due to far memory access. It is desirable to minimize far memory overhead for iterative graph workloads.

In this work, we ask this question: *how can graph processing applications gain their best performance on the emerging far memory architecture*? To answer this question, we take the first step to adapt a general graph processing framework GridGraph [42] to far memory environment. In recent years, far memory researches have been mainly focused on general-purpose design that can provide a better trade-off between performance and resource utilization. In contrast, we explore the benefits of an application-specific far memory platform. Our technique intends to unleash the full potential of far-memory-based graph processing from two primary perspectives: 1) smart data segment offloading and 2) efficient far memory interaction. We introduce the way to identify data segments that are most suitable to be placed on the far memory. We also reconfigure the RDMA system to fit the graph workload better. We demonstrate the necessity of jointly managing application programs and far memory systems with a software/hardware co-design approach to achieve the best performance.

Our contributions are listed as follows.

- We envision far memory as an attractive alternative to existing graph processing models. We analyze the key considerations of such a design.
- We propose Fargraph, a new system optimization strategy that combines graph-aware data segment offloading and iteration-friendly far memory interaction.
- We implement Fargraph based on the GridGraph framework and conduct a detailed case study. We demonstrate the potential of graph processing on far memory.

The rest of this paper is organized as follows. Section II compares different far memory architectures. Section III gives key observations to further motivate this work. Section IV presents Fargraph's graph-aware data offloading strategy and iteration-friendly far memory interaction. Section V further introduces the implementation of Fargraph. Section VI evaluates the performance of Fargraph. Section VII discusses related works, and finally, Section VIII concludes this paper.

II. Background

In this section, we introduce far memory and compare it with traditional systems in the context of graph processing.

A. Far Memory and Its Current Issues

Far memory architecture allows one to opportunistically borrow memory resources from a remote node. As shown in Figure 2-(a), this typically requires RDMA to accelerate memory access over the high-performance network. With proper far memory management, one can balance resource allocation and save local memory for more critical tasks.

Nevertheless, existing general-purpose far memory management schemes such as Infiniswap [12] and Fastswap [1] fail to fully unleash the potential of the far memory system. They cannot achieve full throughput due to a heavy reliance on the swapping mechanism of the OS when accessing far memory. The Linux Swap mechanism involves two parts: the front-end (i.e., Frontswap [21]) and the back-end module. By inserting an RDMA-based swap module into Frontswap, the back-end module is redirected from the disk to RDMA. Since the swap space of the front-end is indispensable for RDMA-based swap, the kernel-level context switch cannot be avoided, which significantly increases the latency of each far memory operation. In addition, for swap-based far memory, one still

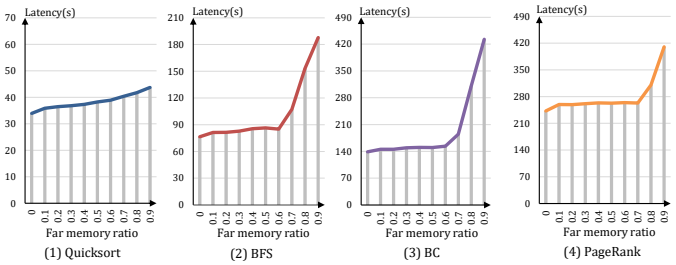

Fig. 3. The measured duration of computation-intensive program Quicksort and graph programs (BFS, Pagerank, BC) on growing far memory proportions.

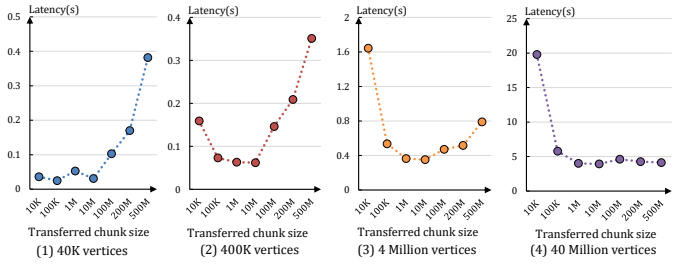

Fig. 4. The measured duration of transferring graph data with various transferring chunk size on RDMA. Sometimes it shows a "smile curve".

needs to create local swap space on a disk as the backup for the RDMA back-end [1], [12]. The far memory access procedure is often blocked due to the frequent disk access for RDMA failure backup.

B. A Comparison of Different Execution Models

Both single-node and distributed computing have been used for graph processing. As a complement to the two models, far memory provides a new option for scaling out memory with high performance. First, from the perspective of a single node, far memory provides a ready-to-use scheme for memory-intensive applications to oversubscribe memory, which outperforms disk-based I/O. Second, at the cluster level, far memory shows promise in further improving resource utilization especially for the SOTA shared-state schedulers like Omega [26], Apollo [2], etc. To sum up, in the graph processing domain, far memory allows one to achieve better performance per bit in a complex execution environment.

1) Far Memory vs. Single-Node Systems: The traditional single-node graph processing system, with all data loaded in local memory, often shows the best performance [22], [29], [39]. If the size of the graph is large, one can use disk storage as a memory extension [17], [24], [42], as shown in Figure 2-(b). However, this may come with considerable performance overhead even with a fast SSD. In contrast, far memory offers an attractive alternative to disk-based memory extension, especially in the cloud environment [1], [14]. Far memory access shows lower performance degradation compared with disk I/O. On the other hand, faced with fluctuating workload demand, far memory represents a more convenient way to oversubscribe memory on demand.

2) Far Memory vs. Distributed Systems: Distributed computing is often used to process large graphs, with CPU and memory on each node working together, as shown in Figure 2-(c). The key difference with far memory is that far memory focuses on data partition instead of task partition (which is common in the traditional distributed model). It is challenging to program directly due to the cost of both task and data partition. Traditional distributed model scales out the overall computing resource, while far memory systems aim to enhance the memory performance of each node. Far memory is well complementary to distributed systems since it can further enlarge the available memory capacity of each node.

III. Design Considerations

This section presents our key observations of running graph applications on far memory.

A. Duration Analysis

We analyze the impact of far memory usage on task duration in different scenarios. Specifically, we define *far memory ratio* α as the ratio of far memory usage to all the memory usage of the application. The proportion of local part is $(1-\alpha)$. We run a computation-intensive program Quicksort and memory-intensive graph programs include BFS, BC, and Pagerank on swap-based far memory platform Fastswap [1] as a motivating analysis. Figure 3 shows our measurements of task duration with different values of far memory ratio. We change the far memory ratio by linearly increasing local memory limitations.

Figure 3 shows that the duration of Fastswap [1] increases remarkably as far memory ratio grows. One reason is the kernel overhead. The duration increase of graph applications mainly comes from growing page faults when adding the far memory ratio. Since Fastswap is implemented at the kernel level, each page fault and page fetching from far memory involve kernel operations so that the system time increases quickly. If we design far memory access operations of graph applications at the user level, we will skip most of the kernel overhead and improve performance.

Graph workloads show different duration trends in the context of far memory, as shown in Figure 3. For general tasks like Quicksort (Figure 3-(1)), we observe a continuous duration increase. Differently, we observe that graph programs have an obvious performance turning point in Figure 3-(2),(3),(4). The relationship between far memory ratio and workload duration is quite similar in the three graph programs. Graph programs exhibit a duration curve that stays relatively flat when the far memory ratio is less than 0.6. The duration increases rapidly if the far memory usage is larger than the turning point. For example, the turning point is 0.6 for BFS, 0.6 for BC, and 0.7 for Pagerank in our experiment. The main reason for the turning point is that graph workload has a distinguished set of hot pages with frequent memory access. When far memory allocation touches the hot pages, the performance becomes sensitive to the far memory ratio.

In summary, running graph program on far memory is non-trivial and memory offloading ratio can greatly affect workload

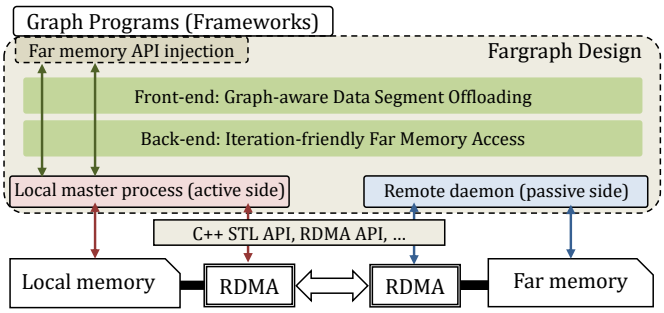

Fig. 5. An overview of Fargraph platform

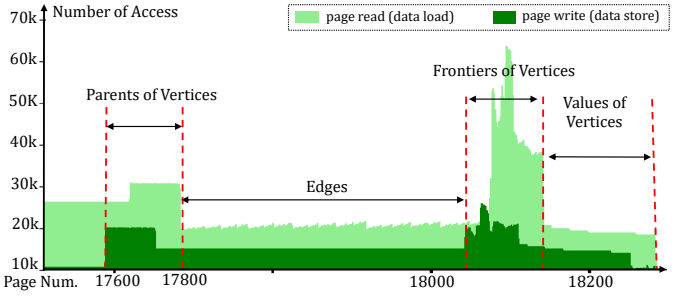

Fig. 6. We test the access frequency of each page in BFS. We show some distinguished data segments in the graph processing program.

performance. It is desirable to increase the awareness of graph workload. Fortunately, graph data are often preprocessed from raw edge list to CSR, CSC,or grid structure in today's computing framework [29], [42]. The preprocessing procedure makes it easier to capture graph properties and to offload graph data in a more efficient and fine-grained manner.

B. Efficiency Issue

Although RDMA has MLNX_OFED (OpenFabrics Enterprise Distribution) environment and provides RDMA atomic operations, it is not easy for developers to configure RDMA settings without specific skills [1], [11], [12]. For example, we often transfer data with chunks of aligned size through network frames on RDMA. It is important to choose a proper chunk size due to the total performance impact. As Figure 4 shows, we test the total duration when transferring different size of graphs (with 40 thousand to 40 million vertices and edges of 10 times of vertex number on RMAT format [4]). The results show that neither small (about 10K) or large (500M) chunk size are proper chunk size to achieve the best behavior.

Importantly, graph applications spend most of the time in data fetching and value updating. In each iteration, the graph algorithm fetches data from both local and far memory and processes them in the local region. Since the data transfer can be completely asynchronous between each RDMA channel, it often causes troubles that the newly arrived data may overwrite valid data during one-sided RDMA operation. Therefore, data fetching and buffering are critical during each iteration. Carefully configuring data transfer can help us cache the right data while avoiding communication delay.

In summary, it is also tricky to setup RDMA in the right way when running graph workload. There are several tuning knobs of far memory access when configuring the network. It is important to choose the right communication configuration that is friendly to the iterative graph execution model.

IV. Fargraph Design

The above analysis shows that smartly offloading graph workload to the remote memory space is critical. In addition, efficiently fetching data using RDMA with proper configurations is necessary to achieve better performance.

We propose Fargraph, an optimization strategy that allows graph programs to run on far memory architecture efficiently. Specifically, we base our design on GridGraph, a representative graph processing framework. Figure 5 provides an overview of Fargraph. It mainly consists of two parts: the front-end *graph-aware data segment offloading* strategy and the back-end *iteration-friendly far memory interaction* optimization. The front-end design of Fargraph mainly analyzes the memory access patterns of graph programs and makes decisions of data offloading to far memory. The back-end works cooperatively to build up efficient far memory access coordination on RDMA for the entire program.

There are two distinct procedures: 1) the master process that resides on the active side (initiator client) and 2) the daemon process on the passive side (responder server). Our key strategies are mainly implemented on the active side, with the assistance from the passive side. In Section V we introduce our implementation.

A. Graph-Aware Data Segment Offloading

At the front-end, we first analyze the data segments of the graph program and classify them into several data segment groups (DS-Group). We then determine data segments that are preferable to be transferred to the remote side in advance for each particular DS-Group.

1) Graph Data Segment Grouping: We investigate the data segments of stacks, queues, as well as graph-specific data items (e.g., array of vertices) in the graph program. Data segment consists of a set of pages. In Figure 6, we show the page access statistics when running BFS on a graph with rMat format. We perform page address tracing and we show the allocated memory area corresponding to Parents of Vertices, Frontiers of Vertices, Edge lists, etc. As we can see, edge-related data segments are often accessed in limited times compared to vertex-related data segments, such as vertex values, parents, frontiers, etc. Meanwhile, the number of traversed edges (memory read) can be several orders of magnitude greater than the number of vertices. Frequent value updating (memory write) always happens on a small group of vertices. It is undesirable if those frequently-accessed data segments are offloaded to the remote side (what the existing works often do). The above analysis shows that one can achieve better far memory offloading effectiveness at the data segment level by analyzing the graph workload.

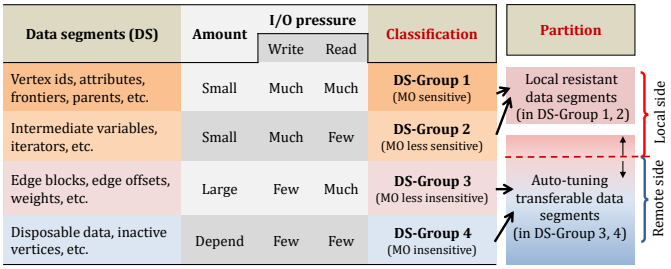

Fig. 7. Description and classification of data segments. We show the data partition method based on transferable data segments on the right part.

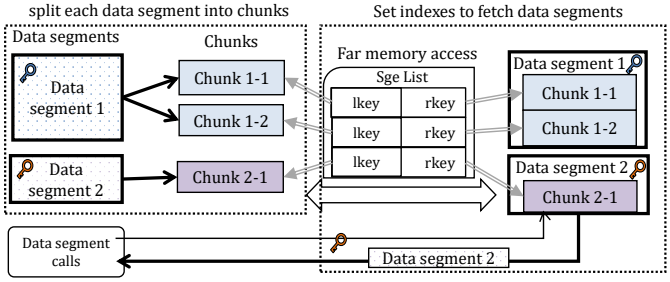

Fig. 8. We split data segment into chunks. We use indexes to facilitate data segment fetching on far memory.

In this work, we classify data segments into a few well-crafted groups i.e., *DS-Groups*, based on graph properties. Empirically, we divide data segments into four groups according to memory access behaviors. Figure 7 shows our classification methodology. ① The DS-Group 1 consists of *memory offloading (short as MO) sensitive* data segments in which data is often read and written in a highly frequent manner, such as Vertex ids, attributes, frontiers, parents(a subset of vertices). ② The DS-Group 2 consists of *MO less sensitive* data segments such as intermediate data variables. They are often written or generated temporarily during computing but do not need to be read from memory. ③ The DS-Group 3 contains *MO less insensitive* data segments with pages read many times and few rewritten, like edge blocks. The read-only feature, if used properly, is well suited for the RDMA environment. ④ The DS-Group 4 (cold segments) is *MO insensitive* data segments, staying untouched for the majority of the time.

There are two ways to classify DS-groups. The first is offline analysis. One can calculate the average page access frequency of each data segment and treat the above-average data segments as the *MO-sensitive* data segments and the below-average ones as *MO-insensitive* data segments. The other way is to track the pages in each data segment online periodically, which is more accurate but time-consuming. One can set a time threshold H. If the page is accessed at time H, it will be labeled *Read/Write-much*; otherwise, it will be labeled *Read/Write-few*. Then we classify data segments into DS-groups according to the labels of these pages. In this work, We use offline profiling to identify the characteristics of data segments (DSs) and classified DSs to guide directive placement for simplicity and practicality.

2) Flexible Data Segments Offloading: The DS Group provides a way for memory offloading. For example, one can keep MO-sensitive data segments (DS-Groups 1 and 2) locally and move all the MO-insensitive data segments (DS-Groups 3 and 4) to remote memory. However, as mentioned earlier, it is likely that the MO-insensitive data segments such as edge blocks are the majority among all the data segments. In this case, restricting local memory usage and moving a huge amount of data segments of DS-Groups 3 and 4 to a remote node may cause nontrivial performance degradation.

To cope with the above issue, we further define *resistant data segment set* and *transferable data segment set (TDSS)*, which refers to a refined set of selected data segments in related DS-Groups that are preferable to be transferred to far memory. The data segments in TDSS are transferable data segments with transferable labels. The idea is enlightened by the concept of *Writable Working Set* [6] which defines a set of pages that are critical to program live migration. We keep data segments in DS-Group 1 and 2 as local resistant data segments that always stay in the local memory. We further select data segments from MO-insensitive groups (DS-Groups 3 and 4)to the remote side, as indicated in Figure 7.

We adopt a more flexible data partition approach based on TDSS. Our design allows one to keep part of the TDSS in local memory. We set priorities for the transferable data segments to decide the preferable offloading order of them according to the ratio of local and remote data. In this way, we can achieve a flexible trade-off between local memory saving and far memory performance in practice. Specifically, we give high priority to the read-only data segments in the transferable data segments. The insight behind this is that most data segments in TDSS of a graph processing workload are read-only (e.g., edge blocks). Fetching read-only data with a one-sided read allows us to minimize data transmission overhead. We can evict the data fetched from far memory as soon as the data is released to save local memory space.

B. Iteration-Friendly Far Memory Access

In the following, we further discuss how to improve far memory access efficiency given the above data segment offloading strategy based on RDMA during each graph iteration. The current RDMA one-sided read mechanism allows one to directly fetch data from remote memory without waiting for system handshaking. However, an appropriate configuration is essential for maximizing the benefits of graph processing on far memory. In our back-end, we break the transferred data into chunks with one-sided access within each iteration, and we overlap the computation and communication in different iterations to reduce the total duration.

1) Data Segment Splitting: Appropriate data transfer is critical. RDMA-based far memory supports memory fetching and updating with different sizes of data chunks. Each chunk is viewed as the basic unit of one-sided read/write. Since the size of each data segment is somehow different, we transfer data based on a finer-grained unit: data chunk. As the left part

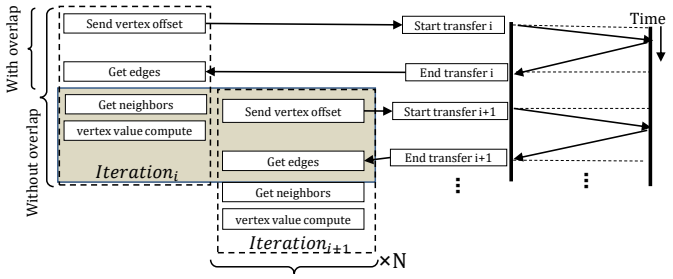

Fig. 9. The iteration-friendly far memory access with overlapping.

of Figure 8 shows, we transparently split each data segment into multiple chunks when writing to the far memory, and we merge these chunks into the data segments when fetching back. If the data segment size is smaller than the size of a chunk, it will not be divided.

We set indexes for remote data segments, as the right part of Figure 8 shows. It reduces the traversal cost of finding the corresponding data segments in far memory. For example, we use the vertex IDs of each grid block as the indexes of edges that are stored in the far memory. We can also use bitmap offsets to guide fast neighbor access accordingly. To ensure secure and isolated memory access, we send the local memory protection key (called lkey) with the indexes together and then fetch the remote key (rkey) and the corresponding data back.

2) Data Segment Buffering: Our second optimization overlaps data computation and communication. We configure the system in such a way that it starts to request the data for the next iteration when the current iteration is still underway as Figure 9 shows. We design buffers that support iteration pipeline overlap by hiding communication when fetching data segments, shown in Figure 10. For grid or shard-like graph processing, it is easy to get the offsets of the next blocks earlier than the next loading. It is feasible since the start and end of the data transmission are two separate control events with RDMA. Note that sometimes an one-sided read can be very fast, to the extent that it may overwrite the last data block. If there is no buffer, the data block supplied for the current iteration will be overwritten by the new operation before being processed. Thus, we also design send/receive buffer pairs for the local master process and the remote daemon process.

If we want to hide the communication overhead completely, the data transfer time needs to be shorter than the execution time for each iteration. In fact, for most graph iterations, data transmission is often much time-consuming (e.g., 2x) than the processing of the obtained data blocks from far memory (as shown in our experiment). As a result, the communication latency cannot be fully hided and the time for each iteration is extended. Therefore, workloads with shorter transfer time and longer execution time often have better speedup than the others. In our experiment, the graph workloads have smaller frontier size (i.e., shorter transfer time) in their early-stage iterations and therefore we observer notable performance improvement at the beginning of their execution.

Algorithm 1 Program Adjustment with Fargraph Interfaces

```
1:  Add_transferable_flag(DS_list, far_ratio, ...);
2:  Build_connection(IP,port,memory_region_size, ...);
3:  //send all TDSS to far memory when preparation
4:  for each DS_i in transferable_DS_list  do
5:     Far_write_start(transfer_flag, DS_i, index, lkey, ...);
6:  end for
7:  Far_write_complete(DS_indexes, rkey, ...);
8:  ...continue... //waiting for data segments calls
9:  //start read far DS_Current in another process;
10: Far_read_start(DS_Current, index, rkey, ...);
11: while (in each processing loop) do
12:    ...continue... //original data process
13:    while calling DS_Current  do
14:       if  DS_Current is prepared then
15:          Far_read_complete(DS_Current,index, rkey, ...);
16:       end if
17:    end while
18:    // start receive the next DS;
19:    Far_read_start(DS_Next, index, rkey, ...)
20:    ...continue.../original data process
21:    if DS_Current finishes occupying then
22:       Free DS_Current in local RAM;
23:    end if
24: end while
```

V. Directive-Like Implementation

We implement Fargraph with directive-like instructions, which slightly modifies the original graph framework. All the far memory operations through RDMA are encapsulated into concise function calls with necessary parameters. We insert our far memory access interfaces into the original graph frameworks to manage the selected data segments.

A. Interfaces Design

Our far memory access interfaces are described as follows. We mainly provide six interfaces for Fargraph. *Add_transferable_flag* makes data segment offloading decisions for the whole program. It adds transferable flag to each data segment in a data segments list (DS_list) based on the given far memory ratio (detailed in Section III-A). *Build_connection()* starts the connection by checking the IP address and the transmission port. It registers the memory regions on the local node with the given memory_region_size. *Far_write_start()* triggers the memory registration on the passive side and then starts writing data to far memory. *Far_write_complete()* returns once this round of sending data is accomplished. It obtains the indexes of data segments on the far memory. The *lkey* and *rkey* represent the protection key for the local and remote memory region, respectively. They are transferred along with the data. *Far_read_start()* starts one-sided read of each DS and implies the beginning time of DS fetching. *Far_read_complete()* returns the rkey and index of the fetched data when the data transmission finishes. Our

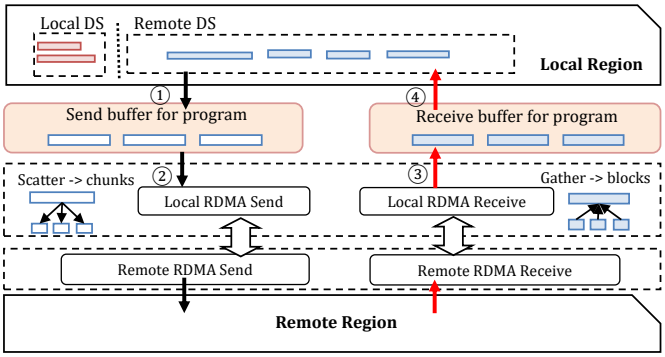

Fig. 10. The detailed workflow of data segment pre-transferring and far memory coordination on Fargraph.

chunk splitting and merging operations are embedded in the far memory read and write functions.

Interfaces Insertion: The key point of the modification is the location of the inserting interface. The pseudocode in Algorithm 1 demonstrates all the interface locations in the original program. The *Far_write_start()* and *Far_write_complete()* of each data segment are in the preprocessing stage. The first *Far_read_start()* is placed right before the beginning of all the iterations. The later *Far_read_start()* are placed as soon as getting the next neighbors in each iteration, as shown in Figure 9. For example, we choose to start transferring the next data segment (DS_Next) once the current data segment (DS_Current) is freed. This allows one to overlap the processing of the current data segment while the next data segment transferring. *Far_read_complete()* of each data segment is in the place where the original data segments are called. This ensures the correction of the transferred data segments. After insertion, the program can use far memory automatically with all the optimization of Fargraph.

The above modification can further apply to both out-of-core and in-memory graph frameworks. Out-of-core frameworks process part of the data from storage like disks, while in-memory frameworks process all the raw data in memory. For out-of-core frameworks, one can load and stream batches of edge blocks from far memory when processing large graphs by replacing disk access (disk I/O) or local memory access (Buffer I/O) with our APIs. For in-memory frameworks, we load the entire data into the main memory before preprocessing, and we replace buffer copy with our APIs.

Note that one can extend our design to support new runtimes and protocols of specific devices like NVLink [10] and CXL fabrics [7]. We leave the extension of supporting more applications and more far memory backend to future work.

B. Fargraph Workflow

The general workflow of Fargraph is shown in Figure 10.

(1) Pre-processing. It starts from far memory access initialization. We create event channels to receive key notifications such as address-resolved, route-resolved, and port-binding, etc. Afterward, the system needs to register memory regions and put them into RDMA's Protect Domains (PD) for memory authorization. We also build RDMA queues (including send, receive, and complete queues) on both the active and passive sides. We decide the transferable data segments and add labels and indexes to them. We then pre-transfer the decided data segments to far memory on the passive server with RDMA write (①, ② in Figure 10) and obtain their far memory keys.

(2) Far memory coordination. Figure 10 shows the general procedure of Fargraph in each iteration. There are two parts of far memory coordination. i) DS-based data fetching. We start to fetch the next DS once the frontier data of the current iteration is ready. We transfer the indexes of the required DSs to far memory as a parameter of function *Far_read_start()*. We then fetch corresponding edge blocks (i.e. DSs) in order. ii) Chunk-based RDMA transfer. When the local region requests data, we use RDMA one-sided read to fetch them. We divide the original data segments into multiple chunks using RDMA *SGE_LIST*. We devise an buffer to pre-fetch the transferred data asynchronously (③ in Figure 10). We also use *post_receive* to continuously receive data read from far memory and write the data into buffers. Meanwhile, we directly copy the received data from the buffer to the local region if the program requests the data (④ in Figure 10).

TABLE I
EVALUATED SYSTEM MANAGEMENT STRATEGIES

Schemes	Application Framework	Exe. Environment
Original	GridGraph with mem. limitations	Disk
Fastswap	GridGraph without mem. limitations	Fastswap (far memory)
Fargraph	GridGraph without mem. limitations	Fargraph (far memory)
Oracle	GridGraph without mem. limitations	Local main memory

VI. CASE STUDY

This section presents detailed experiment results that further support our design choices and demonstrate the efficiency of Fargraph. We give a case study of running GridGraph [42] with and without Fargraph optimization. We also compare our design with the state-of-the-art RDMA-based far memory engine Fastswap [1].

A. Experimental Setup

1) Hardware Environment: We build our far memory platform based on two servers: a client node and a memory node. On the client node, we use Cgroup2 to limit the local memory usage of each process if we need to trigger far memory access. Each node is provisioned with two 20-core Xeon CPUs, 128 GB of memory, and a Dual-Port Mellanox ConnectX-5 RDMA NIC supporting up to 70∼90Gb/s Ethernet. The RDMA driver is version 4.3.0 of the OFED kernel, and it uses RoCE (RDMA over Converged Ethernet) protocol.

2) Evaluated System Strategies: We consider the following strategies as Table I shows. 1) *Original*. This scheme adopts the conventional out-of-core processing model of GridGraph on a single server. It leverages the disk to process medium-sized graphs. 2) *Fastswap*. It is a state-of-the-art, open-source

TABLE II
EVALUATED GRAPH DATASETS

Dataset	$\|V\|$	$\|E\|$	Edge Size	Mem. Footprint
Live Journal (LJ)	4,848 K	69 M	1.1 GB	2.4 GB
Orkut (OR)	3,072 K	117 M	1.8 GB	3.9 GB
Twitter7 (TW)	17 M	477 M	26.3 GB	47.7 GB
Friendster (FR)	65 M	1806 M	32.7 GB	60.4 GB

TABLE III
EVALUATED GRAPH PROCESSING ALGORITHMS

Algorithms	Description	Memory Access Feature
BFS	breadth-first search	random I/O
WCC	weak connected components	random I/O
PageRank	web page ranking	random I/O and sequential I/O
Radii	graph radii estimation	random I/O and sequential I/O

far memory platform which outperforms many previous works [12], [16]. It is an RDMA-based far memory platform with swap kernel and local disk involved [1]. We consider it as a key baseline strategy in this work. 3) *Fargraph*. This scheme uses all the optimizations that we propose. 4) *Oracle*. This is the ideal design case of far memory, which keeps all the data in the local main memory (best performance).

3) Evaluated Graph Workloads: We evaluate 4 graph datasets together with 4 representative graph algorithms in our experiment. The datasets contain 4 real-world graphs: LiveJournal (LJ), Orkut (OR), Twitter7 (TW), and Friendster (FR). More details are given in Table II. The evaluated graph algorithms are shown in Table III. Specifically, BFS and WCC are traversal-centric algorithms, while PageRank and Radii are computation-centric with heavy value computation in each iteration. We run 20 iterations for PageRank and find connected components in unweighted graphs in WCC.

We perform graph processing on the GridGraph [42] framework. GridGraph represents one of the state-of-the-art graph frameworks and it is popular for its powerful grid-based data structure. Another reason for choosing GridGraph is that it provides both buffer I/O version (in the memory) and direct I/O version (in the storage); this feature allows us to evaluate both kernel-level far memory (required by Fastswap) and user-level far memory (required by Fargraph).

B. Efficiency of the Front-end Design

We start by evaluating the performance of Fargraph's front-end optimization, namely the graph-aware data segment offloading. We show that increased workload awareness allows Fargraph to achieve better performance. In Figure 11 and Figure 12 we compare Fastswap and Fargraph on BFS under different far memory ratios (the ratio between far memory usage and total memory demand).

As shown in Figures 11 and 12, Fargraph shows lower task duration compared to Fastswap, especially when the far memory ratio is large. We observe that the duration of Fastswap rapidly increases if the far memory ratio is larger

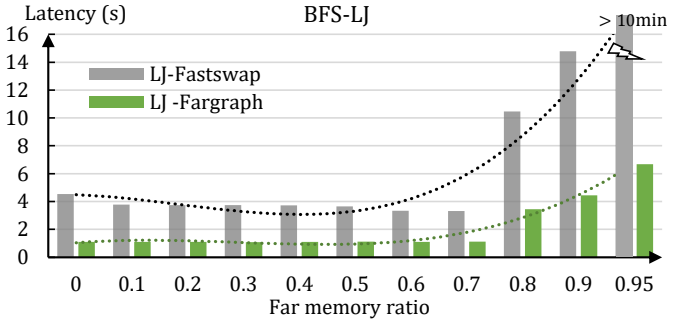

Fig. 11. The duration of BFS on dataset LJ on far memory platform Fargraph and Fastswap under rising far memory ratios.

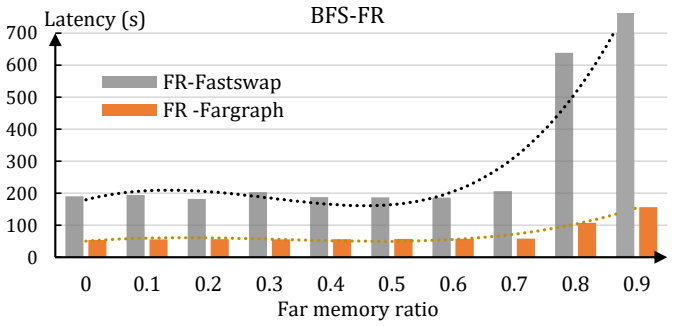

Fig. 12. The duration of BFS on dataset FR on far memory platform Fargraph and Fastswap under rising far memory ratios.

than 0.8. This is because the system starts to move MO-sensitive data segments to far memory. When the far memory ratio reaches 0.95, Fastswap cloud be too slow to meet user expectations and it cannot finish even after 10 minutes of execution on BFS-LJ. In contrast, Fargraph still maintains acceptable performance. The reason is that Fargraph uses a tailored data segment partition strategy and it can make the best use of far memory to process a larger amount of graph data.

C. Efficiency of the Back-end Design

1) Performance Impact of Data Segment Splitting: Since Fargraph relies on data segment splitting (detailed in Section IV-B) to improve far memory efficiency, determining the appropriate chunk size is critical. In Figure 13, we plot the smile-like duration curves of 4 workloads (BFS and PageRank on LJ and OR). The results are normalized to the duration under 4K chunk size. In particular, the duration under 4K chunk size is higher than the duration under 32K and 256K chunk size. This indicates that the 4K-page-based far memory access design (e.g., Fastswap) is not efficient enough. The reason for the smile-like curve is that the best far memory chunk size is determined by the smaller one between RDMA bandwidth and PCIe bandwidth. If RDMA transmission bandwidth (i.e., the frame size) cannot fill the PCIe channel, a larger chunk size means better performance. In contrast, if the RDMA bandwidth is too large, the total bandwidth can be limited by the PCIe channel.

TABLE IV
THE TOTAL PEFORMANCE COMPARISON OF 16 GRAPH WORKLOADS WITH 80% FAR MEMORY

Results (seconds)	BFS				WCC				PageRank				Radii			
	LJ	OR	TW	FR	LJ	OR	TW	FR	LJ	OR	TW	FR	LJ	OR	TW	FR
Oracle	2.63	2.61	27.88	54.33	0.16	1.59	38.5	69.6	5.33	7.73	115.24	137.76	2.74	9.44	150.36	320.45
Original	9.84	6.08	235.91	637.17	2.36	4.55	144.17	318.8	39.20	74.80	848.00	1153.6	20.75	110.3	1139.0	2578.0
Fastswap	10.46	7.03	262.24	639.0	2.56	6.52	256.4	523.1	25.53	40.80	966.03	1662.0	20.45	135.24	1524.6	3054.8
Fargraph	**2.97**	**3.93**	**63.23**	**94.02**	**1.32**	**2.98**	**70.2**	**98.2**	**6.92**	**23.52**	**123.70**	**200.85**	**5.48**	**20.49**	**350.26**	**652.24**
Sp(Original)	**3.3x**	**1.5x**	**3.7x**	**6.7x**	**1.8x**	**1.5x**	**2.1x**	**3.2x**	**5.7x**	**3.2x**	**6.9x**	**5.7x**	**3.8x**	**5.4x**	**3.3x**	**4.0x**
Sp(Fastswap)	**3.5x**	**1.8x**	**4.1x**	**6.8x**	**1.9x**	**2.2x**	**3.7x**	**5.3x**	**3.7x**	**1.7x**	**7.8x**	**8.3x**	**3.7x**	**6.6x**	**4.4x**	**4.7x**

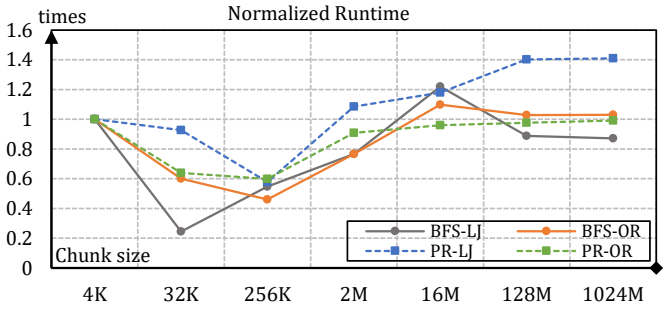

Fig. 13. The normalized performance of four workloads (BFS and Pagerank on dataset LJ and OR) with different chunk sizes.

With PCI Express 3.0 (16 GB/s) and 9.6KB RDMA frame with dual-port on our two-CPU mainboard, the full-bandwidth chunk size is around $(16 \times 9.6 \times 2)$KB =307.2KB. Note that the optimal data chunk size varies due to different hardware resource configurations and different program behaviors. Our experiment results show that we can obtain the best performance at around 256KB for most of the workloads (Only BFS-LJ favors 32K chunk size) evaluated in this study. Therefore, we use 256KB in all of our experiments.

2) Performance Impact of Data Segment Buffering: We further evaluate the performance impact of data segment buffering which enables efficient iteration overlap. We show the results of two representative algorithms, namely, BFS and Pagerank. We measure the duration of the non-overlapped version (Fargraph w/o buffering) and the overlapped version (e.g., Fargraph). In Table V, we show the results of BFS and PageRank on 4 datasets. As we can see, data segment buffering brings task duration down by up to 19%.

In general, there is a striking difference between PageRank and BFS if we look at the duration reduction effect of data segment buffering. In the table we show the absolute and relative *duration reduction*. The relative duration reduction refers to the ratio between duration reduction and the original duration. It is evident that the reduced duration of Pagerank is larger than BFS. We also observe that the relative duration reduction of BFS is relatively stable while that of PageRank may increase significantly under larger graph datasets.

D. Overall Performance

We finally present the overall optimization effectiveness of Fargraph across 16 workloads. Table IV compares Fargraph with all the other evaluated schemes.

TABLE V
THE DURATION COMPARISON OF FARGRAPH DATA BUFFERING

	Duration (s)	BFS			
		LJ	OR	TW	FR
Schemes	Fargraph w/o buffering	3.45	4.53	75.61	107.40
	Fargraph buffering	2.97	3.93	63.23	94.02
Duration reduction	Absolute value	↓ 0.48	↓ 0.6	↓ 12.38	↓ 23.38
	Relative value	14%	13%	12%	13%
		PageRank			
Schemes	Fargraph w/o buffering	7.44	26.80	153.19	233.63
	Fargraph buffering	6.92	23.52	123.70	200.85
Duration reduction	Absolute value	↓ 0.52	↓ 3.28	↓ 29.49	↓ 32.78
	Relative value	7%	12%	19%	14%

For many datasets, computation-centric algorithms like Pagerank and Radii show relatively higher performance improvement compared to the traversal-centric algorithms, such as BFS and WCC. It is mainly because the data access patterns of BFS and WCC are more irregular than PageRank and Radii. Another reason is that the I/O overhead cannot be fully hidden by computation in graph iterations.

The results also demonstrate the attractive scalability of Fargraph. In most cases, Fargraph shows better performance improvement as the graph size grows. For example, BFS, WCC, and PageRank all yield an increasing speedup on datasets OR, TW, and FR. Radii has a different behavior mainly because the estimation of graph radius requires much more traversal time as the graph size grows.

Overall, The results show that Fargraph is more efficient and is closer to an oracle design compared with Fastswap. We can achieve 6.9× better performance compared to *Original*, and up to 8.3× performance compared to *Fastswap*. Note that our evaluation is conservative due to the use of a medium-sized dataset (instead of hundreds of GB). It is more challenging for Fargraph to make memory offloading decisions and hide communication latency with smaller datasets. Our design approach can be applied to many other graph frameworks and we expect it to show better performance on larger graphs.

E. Cost-Effectiveness of Memory Capacity

Finally, we estimate the cost-effectiveness of *RDMA-based* far memory and *NVLink-based* far memory, as shown in Figure 14. The NVLink-based far memory (i.e., direct-connected approach) represents an alternative to RDMA-based far memory (i.e., NIC-based approach). There is a growing interest in direct-connected far memory technologies [3], [20] on the next-generation I/O fabrics like NVLink and CXL. Recent works show that NVLink-based far memory can achieves almost doubled bandwidth than RDMA-based far memory

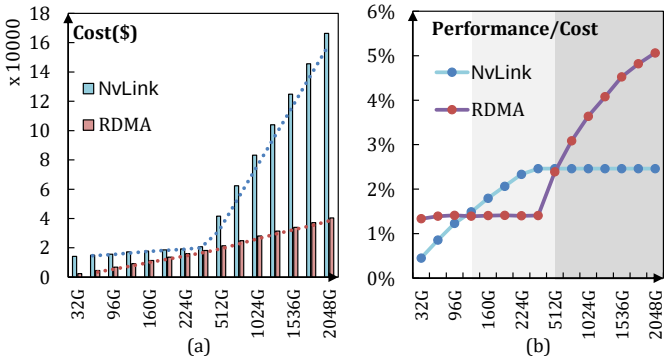

Fig. 14. Cost analysis for RDMA and NVLink based far memory systems.

[5]. However, it is often 10-100x more expensive due to the high cost of NVLink-compatible devices including boards, switches, and IBM POWER9 processors [8], [23]. We estimate the total cost based on the data-sheet of CPU, memory, mainboard, Heatsink, Mellanox Connect-X card, SSD, adapter, RAID switches, and NVLink supported devices.

In total, the cost of each NVLink-based machine is almost 10x more expensive than a RDMA-based machine. The trend in Figure 14-(a) shows that the cost of the direct-connected, NVLink-based far memory increases rapidly when the memory demand exceeds the memory capacity of a single node. This is because the CPU, memory and mainboard are much more expensive than the RDMA NIC and fabric. In addition, we also show the normalized performance-cost ratio in Figure 14-(b). We assume that the performance of NVLink-based machines are 2x of RDMA-based machines, which is reasonable according to prior work [27]. We find that the cost-effectiveness of RDMA-based design can be better when the requested extra memory capacity is in the range of 128-512G. This means RDMA-based far memory is suitable for the typical scale of far memory capacity. It can be more extensive for extremely large memory demand at this moment. In this case, we expect to use distributed computing other than far memory architecture.

VII. Related Work

Disaggregated Memory Architectures. *Composable Disaggregated Infrastructure*(CDI) [15], [19] gains considerable attention in recent years. It is proposed to break the fixed hardware components of monolithic servers into disaggregated, network-attached components. For example, LegoOS [27] introduces modular system implementation for hardware disaggregation. String-finger [20] builds a large memory pool with thousands of memory nodes and tens of CPUs. There are several works [1], [16], [30] focusing on extending their own memory to a special memory node with large DRAMs or NVMs in the rack. Differently, Fargraph provides an application-aware far memory optimization scheme, which is more efficient than general-purpose platforms.

System Support for Graph Processing. In general, there are three types of graph processing frameworks. 1) *In-memory* graph frameworks, such as Ligra [29], Cagra [40], GraphIt [39], and etc. In memory frameworks process graphs after all the source data are loaded into the main memory [33]. 2) *Out-of-core* frameworks, such as GridGraph [42], Mosaic [17], HUSGraph [36] process large graphs with limited main memory and a large-capacity disk. Works with out-of-core execution patterns [28], [32] load each graph block into the memory and process them streamingly. 3) *Distributed* frameworks, such as GraM [35], Gemini [41], and Chaos [24], divide huge graphs into several parts and process them with Map-Reduce-style schemes. All of these works concentrate on the execution instead of data partition, especially in the context of remote memory access. Fargraph fills a critical void by enabling efficient graph processing on RDMA-based far memory. It can be extended to further support emerging applications like graph-structured cloud-native applications [13], [34], [37], [38] as well as graph-based ML/AI applications [31].

RDMA-based Far Memory Acceleration. With kernel-bypass and fast-messaging features, RDMA card has been widely used for speeding up remote memory access. For example, *general-purpose* far memory is drawing increasing attention in recent years. Infiniswap [12] proposes transparent remote memory paging based on RDMA. It is also feasible for a virtual machine to access not only its own isolated memory area, but also DRAM-based external memory and RDMA-based far memory [16]. Since graph computing has irregular memory access patterns, general-purpose far memory acceleration schemes cannot achieve the best performance. Consequently, designing *application-specific* far memory is also gaining popularity. For example, GraM [35] processes graphs with distributed computing, using RDMA to pass messages. Different from existing works, Fargraph manages transferable data segments for graph workloads and optimizes graph processing with tailored RDMA control.

VIII. Conclusion and Discussion

In this paper, we explore graph processing on emerging far memory architecture. We show that there are several challenges and opportunities of deploying graph workloads on far memory. We propose Fargraph, an optimization strategy that allows one to run graph applications on far memory efficiently. We implement Fargraph based on the GridGraph framework and conduct a case study to demonstrate its effectiveness. We show that Fargraph can achieve up to 6.9× and 8.3× speedup compared to conventional out-of-core graph processing framework and the state-of-the-art general-purpose far memory platform, respectively. We expect that our design will open a door for more efficient graph processing in the next-generation cloud with disaggregated architecture.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (No.61832006 and No.61972247), and by Alibaba Innovative Research Program. We thank all the anonymous reviewers for their valuable feedback. Corresponding author is Chao Li.

REFERENCES

[1] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *European Conference on Computer Systems (EuroSys)*, 2020, pp. 1–16.

[2] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 285–300.

[3] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. Al Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 79–92.

[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *SDM*, 2004.

[5] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 926–939.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen *et al.*, "Live migration of virtual machines," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005, pp. 273–286.

[7] C. E. L. Corporation, "Compute express link," https://www.computeexpresslink.org/about-cxl, accessed on June 1, 2021.

[8] I. Corporation, "Ibm power9 cpu," https://www.ibm.com/it-infrastructure/power/power9, accessed on June 1, 2021.

[9] M. Corporation, "Mellanox interconnect community," https://community.mellanox.com/s/, accessed on June 1, 2021.

[10] N. Corporation, "Nvlink interconnect," http://www.nvidia.com/object/nvlink.html, accessed on June 1, 2021.

[11] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[12] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 649–667.

[13] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, "Ant-man: Towards agile power management in the microservice era," in *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2020.

[14] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid *et al.*, "Software-defined far memory in warehouse-scale computers," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 317–330.

[15] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *International Symposium on Computer Architecture (ISCA)*, 2009, pp. 267–278.

[16] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, "Memory disaggregation: Research problems and opportunities," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1664–1673.

[17] S. Maass, C. Min, S. Kashyap *et al.*, "Mosaic: Processing a trillion-edge graph on a single machine," in *European Conference on Computer Systems (EuroSys)*, 2017, pp. 527–543.

[18] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *USENIX Annual Technical Conference (USENIX ATC)*, 2017, pp. 631–643.

[19] D. Montgomery, "The future of data infrastructure: Cdi," https://www.datacenterknowledge.com/industry-perspectives/future-data-infrastructure, accessed on June 1, 2021.

[20] M. Ogleari, Y. Yu, C. Qian, E. Miller, and J. Zhao, "String figure: A scalable and elastic memory network architecture," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 647–660.

[21] L. OS, "Linux frontswap," https://www.kernel.org/doc/html/latest/vm/frontswap.html, accessed on June 1, 2021.

[22] P. Pan and C. Li, "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines," in *International Conference on Computer Design (ICCD)*, 2017, pp. 217–224.

[23] C. Pinto, D. Syrivelis, M. Gazzetti *et al.*, "Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation," in *The IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 868–880.

[24] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 410–424.

[25] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "Aifm: High-performance, application-integrated far memory," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 315–332.

[26] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *ACM European Conference on Computer Systems (EuroSys)*, 2013.

[27] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed os for hardware resource disaggregation," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 69–87.

[28] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, and M. Guo, "Oversubscribing gpu unified virtual memory: Implications and suggestions," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2022.

[29] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2013.

[30] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores," in *USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 33–48.

[31] P. Wang, C. Li, J. Wang, T. Wang, L. Zhang, J. Leng, Q. Chen, and M. Guo, "Skywalker: Efficient alias-method-based graph sampling and random walk on gpus," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021.

[32] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward unified-memory-efficient high-performance graph processing on gpu," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2021.

[33] P. Wang, L. Zhang, C. Li, and M. Guo, "Excavating the potential of gpu for accelerating graph traversal," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 221–230.

[34] X. Wang, C. Li, L. Zhang, X. Hou, Q. Chen, and M. Guo, "Exploring efficient microservice level parallelism," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[35] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: Scaling graph computation to the trillions," in *ACM Symposium on Cloud Computing (SoCC)*, 2015, pp. 408–421.

[36] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, and Y. Zhang, "Hus-graph: I/o-efficient out-of-core graph processing with hybrid update strategy," in *International Conference on Parallel Processing (ICPP)*, 2018.

[37] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, "A locality-aware energy-efficient accelerator for graph mining applications," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 895–907.

[38] L. Zhang, W. Feng, C. Li, X. Hou, P. Wang, J. Wang, and M. Guo, "Tapping into nfv environment for opportunistic serverless edge function deployment," in *IEEE Transactions on Computers (TC)*, 2021.

[39] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with graphit," in *The International Symposium on Code Generation and Optimization (CGO)*, 2020, p. 158–170.

[40] Y. Zhang, V. Kiriansky, C. Mendis *et al.*, "Making caches work for graph analytics," in *IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302.

[41] X. Zhu, W. Chen *et al.*, "Gemini: A computation-centric distributed graph processing system," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 301–316.

[42] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *USENIX Annual Technical Conference (USENIX ATC)*, 2015, pp. 375–386.