Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Fargraph+: Excavating the parallelism of graph processing workload on RDMA-based far memory system

Jing Wang [a], Chao Li [a,b,*], Yibo Liu [a], Taolei Wang [a], Junyi Mei [a], Lu Zhang [a], Pengyu Wang [a], Minyi Guo [a,b]

[a] *Shanghai Jiao Tong University, Shanghai, China*
[b] *Shanghai Qi Zhi Institute, Shanghai, China*

A R T I C L E   I N F O

Article history:
Received 30 August 2022
Received in revised form 7 December 2022
Accepted 24 February 2023
Available online 10 March 2023

Keywords:
Far memory
Graph processing
RDMA

A B S T R A C T

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 propose Fargraph+, a system with parallel graph data offloading and far memory coordination strategy for enhancing efficiency of graph processing workload on RDMA-based far memory architecture. 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 further configure efficient parallelism-oriented control to accelerate performance of multi-threading processing on graph iterations while improving memory efficiency of far memory access by utilizing RDMA queue features. We show that Fargraph+ achieves near-oracle performance for typical in-local-memory graph processing systems. Fargraph+ shows up to 11.2× speedup compared to Fastswap, the state-of-the-art, general-purpose far memory platform.

1. Introduction

Today's various graph applications demand better memory performance at different graph scales [22,51,53,44,41]. 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 [34,51,53]. Distributed graph frameworks are required only for very large-scale data analytic problems due to the communication overhead [43,55,29]. Nevertheless, as shown in Fig. 1-(a), many graph frameworks mainly focus on medium-sized graphs (from 1GB to several hundreds of GB) [54,21,44]. Although current out-of-core graph computing frameworks could handle medium-sized graphs with external storage like disk and SSD, they suffer significant performance degradation due to the I/O bottleneck.

In addition, to process workloads with various data inputs in the cloud, an important trend is to build disaggregated memory pools and enable *far memory* (i.e., remote main memory) accesses [32,24,9,6]. 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) [10] and Compute Express Link (CXL) [8], far memory access can achieve near-DRAM performance, as shown in Fig. 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 2).

* Corresponding author at: Shanghai Jiao Tong University, Shanghai, China.
E-mail addresses: jing618@sjtu.edu.cn (J. Wang), lichao@cs.sjtu.edu.cn (C. Li), liuyib@sjtu.edu.cn (Y. Liu), sjtuwtl@sjtu.edu.cn (T. Wang), meijunyi@sjtu.edu.cn (J. Mei), luzhang@sjtu.edu.cn (L. Zhang), wpybtw@sjtu.edu.cn (P. Wang), guo-my@cs.sjtu.edu.cn (M. Guo).

https://doi.org/10.1016/j.jpdc.2023.02.015
0743-7315/© 2023 Elsevier Inc. All rights reserved.

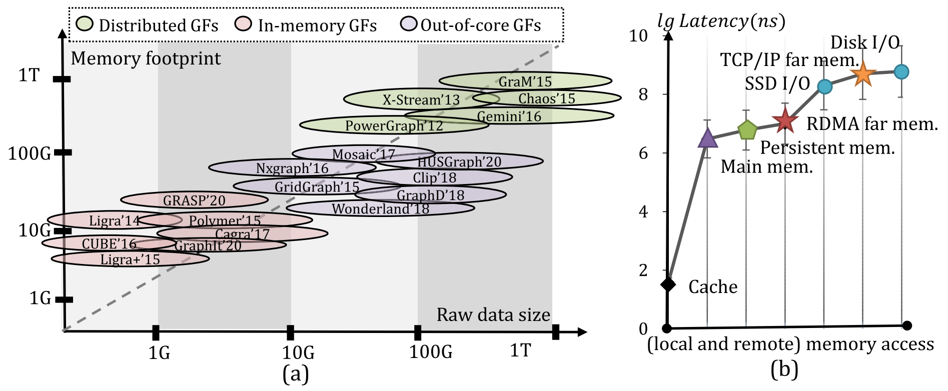


Fig. 1. (a) The case of memory usage expansion and the raw graph size of existing graph processing frameworks. (b) The data transfer duration of different memory access scenarios.

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 [13,20,2,18], 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 [13,2,12]. To reduce the above system overhead, recent studies attempt to build a user-level runtime to reduce kernel overhead [30,4]. However, they are not aware of the workload characteristics and they may miss performance optimization opportunities when running graph processing workload on far memory system (detailed in Section 3).

Furthermore, improving the parallelism level of graph processing on RDMA-based far memory environment needs careful consideration. First, one should choose a proper number of assigned threads on graph programs. Simply adding threads to graph programs may not provide better performance speedup, which may cause performance degradation and memory space underutilization due to graph data properties [26,47,46]. Second, one should have detailed parallelism configurations on RDMA-based far memory access. To adapt to multi-thread computation, one may start abundant RDMA queue pairs corresponding to the number of virtual CPU cores. Multi-threaded communication in MPI is limited by serial data fetching based on RDMA transmission patterns [14,35]. In this case, RDMA queues are not fully utilized so that bandwidth can be wasted [1].

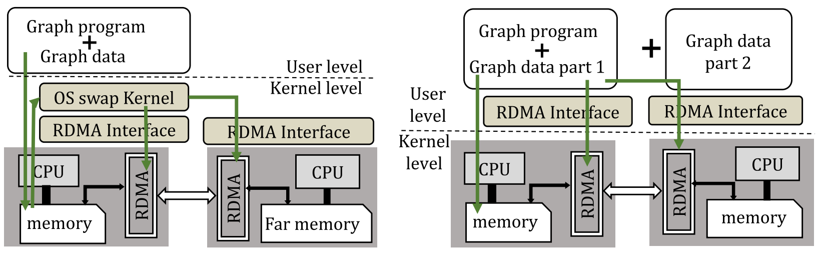
The key opportunity of optimizing graph workload on far memory comes from two aspects: 1) the distinctive data segments and 2) the iterative execution model. First, a graph processing program features a group of data segments with distinctive characteristics [34,26,51]. On one hand, the size of different 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 [22,34,54]. 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 [54] to far memory environment. In recent years, far memory research has 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 three primary perspectives: 1) smart data segment offloading 2) adaptive far memory interaction and 3) efficient parallelism optimization. We introduce the way to identify data segments that are most suitable to be placed on the far memory. We also reconfigure the parallel RDMA-based far memory access to fit the graph workload better. We further control the multi-threading design of graph applications on the basis of saving RDMA-related resources (CPU core, memory space, RDMA queues, etc.) to achieve higher efficiency. 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 and efficiency.

Our contributions are listed as follows.

- We envision RDMA-based far memory as an attractive alternative to existing graph processing models. We propose Fargraph+ based on Fargraph [38] and improve the parallelism performance and memory efficiency of graph processing on far memory.
- We design and optimize the parallelism of Fargraph+ based on parallel graph iteration control and optimized graph data structure for better performance of graph-aware data segment offloading and fetching.
- We further improve performance while reducing memory consumption of Fargraph+ by optimizing RDMA queue configuration for higher memory efficiency of iteration-friendly far memory interaction.
- We evaluate Fargraph+ in detail. We outperform the out-of-core graph framework GridGraph by up to 8.2× and the state-of-the-art far memory platform Fastswap by up to 11.2×. We also have up to 2.5x performance improvement compared with Fargraph.

The rest of this paper is organized as follows. Section 2 compares different far memory architectures. Section 3 gives key ob-



(a) Kernel-level Far memory system **(b)** User-level Far memory system

Fig. 2. Different architectures that can be leveraged for graph processing. (a) Kernel-level far memory with modified swap mechanism of OS. (b) User-level far memory communicated through RDMA.

servations to motivate this work. Section 4 presents the overview of Fargraph+. Our system mainly consists of three parts, including the front-end (graph-aware data segment offloading) in Section 5, the back-end (iteration-friendly far memory access) in Section 6 and the parallelism-oriented control in Section 7. Section 8 further introduces the implementation of Fargraph+. Section 9 introduces the evaluated experiments setup and Section 10 evaluates the performance results of Fargraph+. Section 11 discusses related works, and finally, Section 12 concludes this paper.

2. Background

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

2.1. Far memory and its current issues

Recently, there are two types of far memory architecture, fabric-based and NIC-based far memory architecture. Fabric-based far memory uses direct-connected far memory technologies, like NVLink [11] and CXL [8], and external memory devices can be accessed directly as alternative memory space of local memory [4,24]. They can achieve almost doubled bandwidth compared to RDMA-based far memory [6,28,9]. However, fabric-based far memory hardware device is often 10-100x more expensive due to the high cost of fabric-supported components. In addition, few open-source corresponding OS the direct-connected protocols, which makes it difficult to evaluate and design.

NIC-based far memory architecture allows one to opportunistically borrow memory resources from a remote node. NIC-based far memory technology greatly benefits from the development of high-speed networks like RDMA, making memory disaggregation more practical. As shown in Fig. 2, 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 [13] and Fastswap [2] 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.

2.2. Kernel-level and user-level FM

We compare the kernel-level far memory system with OS swap kernel (the swap backend is RDMA) and the user-level far memory system with remote memory access on RDMA in Fig. 2. Replacing the swap backend directly with the RDMA kernel is a straightforward idea for far memory access. The Linux Swap mechanism involves two parts: the front-end (i.e., Frontswap [25]) and the back-end module. By inserting an RDMA-based swap module into Frontswap, the back-end module is redirected from the disk to

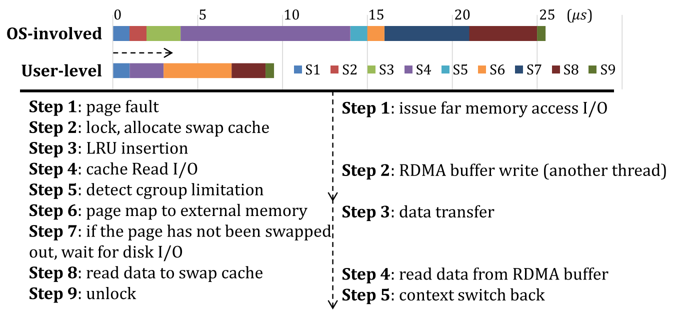


Fig. 3. Latency comparison of kernel-level far memory with modified swap mechanism of OS (on the left) and user-level far memory communicated through RDMA (on the right).

RDMA. Although kernel-level far memory is transparent to applications by passively swapping cold pages to RDMA far memory, it suffers significant overhead. 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 kernel-level swap-based far memory, one still needs to create local swap space on a disk as the backup for the RDMA back-end [2,13]. The far memory access procedure is often blocked due to the frequent disk access for RDMA failure backup.

Meanwhile, Fig. 3 shows the detailed execution delay of each step on kernel-level page access through the swap mechanism and user-level page fetching through RDMA protocol. User-level far memory has much lower remote memory access latency through RDMA operations than the kernel-level far memory with OS involved. User-level RDMA-based far memory earns performance from the skip of sequential page fault and the locked data swap for each data fetch. In addition, CPU-free RDMA allows data overlapping of local computation and far memory communication. Furthermore, RDMA-based far memory gives scalable memory fetching and updating with different sizes of data chunks when transferring. Thus, user-level RDMA-based far memory access can achieve higher performance and have more flexible programming methods compared with swap-based far memory systems.

2.3. 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, far memory provides a ready-to-use scheme for memory-intensive applications to oversubscribe memory, which outperforms disk-based I/O on a single node. Second, at the cluster level, far memory shows promise in further improving resource utilization, especially for the SOTA schedulers [31,3]. To sum up, far memory execution model allows one to achieve better performance per bit in a complex execution environment.

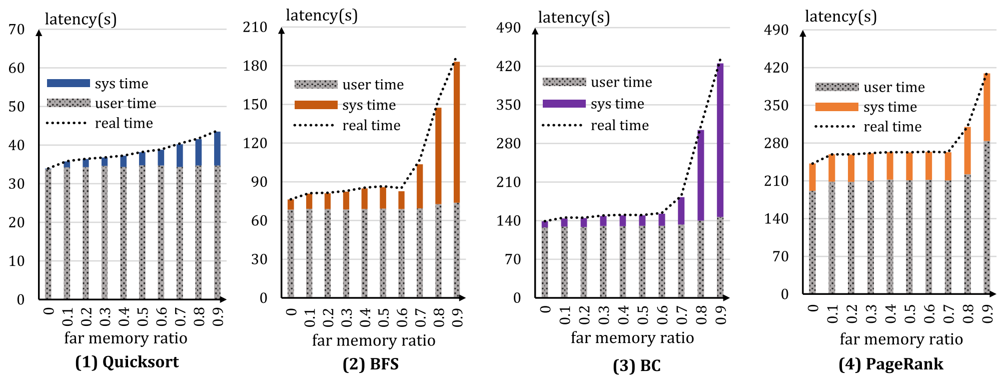


Fig. 4. The measured duration of computation-intensive program Quicksort and memory-intensive graph programs (BFS, PageRank, BC) on growing far memory proportions.

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 [34,26,53]. If the size of the graph is large, one can use disk storage as a memory extension [54,29,21], as shown in Fig. 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 [2,18]. 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.

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 Fig. 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 the existing distributed systems since it can further enlarge the available memory capacity of each node.

3. Design considerations

This section presents our key observations of running graph applications on far memory. This motivates us to design an optimized strategy of offloading graph data efficiently in RDMA-based far memory environment.

3.1. Graph-aware opportunities

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 including BFS, BC, and PageRank implemented in the graph processing framework GridGraph [54] on swap-based far memory platform Fastswap [2]. We use *Cgroup* to limit the local memory usage and leverage page swap to offload pages to RDMA-based far memory. Fig. 4 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.

Fig. 4 shows that the duration of applications on Fastswap [2] increases remarkably as far memory ratio grows. One reason for this is the kernel overhead, since the “sys time” representing the in-system latency grows significantly in Fig. 4. 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. Furthermore, swap-based far memory works like Fastswap [2] swap out the least accessed pages in the tail of LRU (least-recent-used) queues to far memory. However, as graph processing typically has irregular data access patterns, swap-based far-memory systems may swap out frequently accessed pages and cause back-and-forth data movement. If we design far memory access operations 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 Fig. 4. For general tasks like Quicksort (Fig. 4-(1)), we observe a continuous duration increase. Differently, we observe that graph programs have an obvious performance turning point in Fig. 4-(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 workloads have a distinguished set of hot pages with frequent memory access. When far memory allocation touches the hot pages (when far memory percentage exceeds the turning point), the system's performance becomes sensitive to the far memory ratio. The reason is that graph processing features much more frequent memory access on vertex-related pages than edge-related pages, as stated later in Fig. 8. This motivates us to carefully offload a large number of cold parts (edge-related data) to far memory.

In summary, it is desirable to design a memory offloading strategy of graph workload on far memory for better performance. *First*, user-level data offloading can reduce the overhead of page swap. *Second*, fine-grained partitioning of graph data and offloading memory-consuming and cold data to far memory can further improve workload performance and system efficiency.

3.2. RDMA efficiency issue

Although MLNX_OFED (OpenFabrics Enterprise Distribution) environment provides RDMA atomic operations, it is not easy for developers to configure RDMA settings without specific skills [13,2,12]. 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 Fig. 5 shows, we test the total duration when transferring different size of graphs (with 40 thousand to 40 million vertices and edges of 10

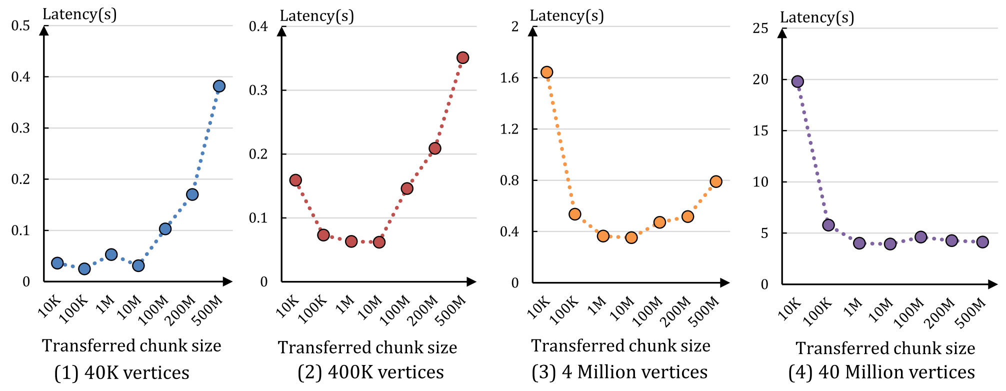


Fig. 5. The measured duration of transferring graph data with various transferring chunk size on RDMA. Sometimes it shows a "smile curve".

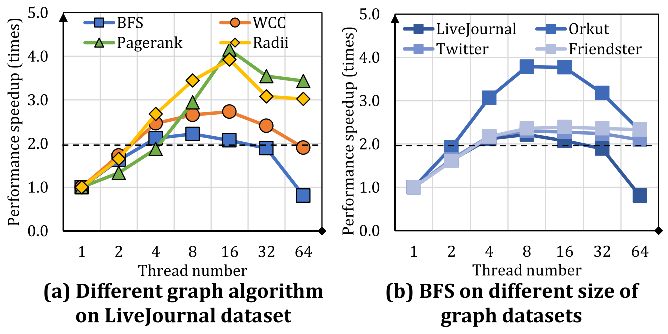


Fig. 6. Performance of graph applications varies on different graph algorithms and datasets at different parallelism level.

times of vertex number on RMAT format [5,16]). Note that RMAT is the abbreviation of Recursive Matrix that naturally generates power-law degree distributions among graph vertices and RMAT graphs are often stored in edge list format. The results show that neither small (about 10K) nor large (500M) chunk size is the optimal 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. Graph programs have a lot of iterations and the data to be fetched in the next iteration has dependencies on the previous iteration. Since the data transfer can be completely asynchronous between each RDMA channel, it often causes trouble when the newly arrived data overwrite the existing valid data during one-sided RDMA operations. Carefully configuring data fetching and buffering during each iteration can help us cache the right data while avoiding communication delay.

In summary, it is essential to set RDMA-based far memory access in the right way to fit the iterative graph execution model. *First*, configuring proper chunk size according to graph data is tricky to optimize data communication latency. *Second*, careful data buffering is important for correct data fetching in graph workloads.

3.3. Parallelism opportunities

Existing big data frameworks often improve the parallelism level of the applications by adding more threads. For example, one may use ready-to-use OpenMP, OpenMPI, Cilk, and Cilk+ to enable parallel computing automatically. We run classical graph algorithms of GridGraph [54] on real-world graphs with all data in memory, referring to Table 2 and 3. The overall runtime when assigning different numbers of threads is collected and the performance speedup at different parallelism levels is shown in Fig. 6.

Graph applications behave a little differently from the other easy-to-parallel applications. First, more threads may bring performance degradation. For example, when assigned 64 threads, the BFS acts worse than any other condition, as shown in Fig. 6-(a). Second, Graph programs often maintain an optimal range of thread numbers, and allocating thread numbers out of the optimal range may cause resource inefficiency. For example, the optimal range of BFS on Orkut dataset is 2 to 8, since 1 thread is less than 2× speedup (the dotted line in Fig. 6) and 16 threads can not bring performance benefit. Multi-threaded graph traversal increases inter-thread communication, which in turn undermines the benefits of parallelism in graph application. To avoid negative effects as well as save computing resources, we should choose the optimal range of thread numbers, which varies across different graph algorithms and datasets.

More importantly, the performance issue of parallelism can be worse when involving the RDMA environment. Although RDMA communication has been supported by the MPI standard [37] for several years, it is still difficult to access RDMA-based far memory efficiently [14,35]. First, multi-threaded communication in MPI is limited by serial data fetching based on RDMA transmission patterns. For each thread, RDMA will build queues corresponding to each virtual CPU core to prepare for data communication. However, in this case, the content in each queue is isolated so that the local process must wait for all the requested data to arrive. To solve this problem, one can configure RDMA to share received data by shared receive queue (SRQ) in multi-queue environments. Second, to adapt to multi-thread computation, one often starts abundant RDMA queue pairs which consume non-negligible memory space [1]. By default, the system will build send queues, receive queues, complete queues to hold events, and data content channels to transfer data for each thread. Events can be discovered to drive the next operation and content corresponding to each event will be transferred.

In summary, there are opportunities for adapting graph processing to parallel far memory access through RDMA. *First*, choosing the optimal parallelism level can improve workload performance with high resource efficiency. *Second*, an innovative configuration of resource sharing and allocation of RDMA queues can reduce abundant resource usage while supporting parallel far memory access.

4. Overview

The above analysis shows that smartly offloading graph workloads 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

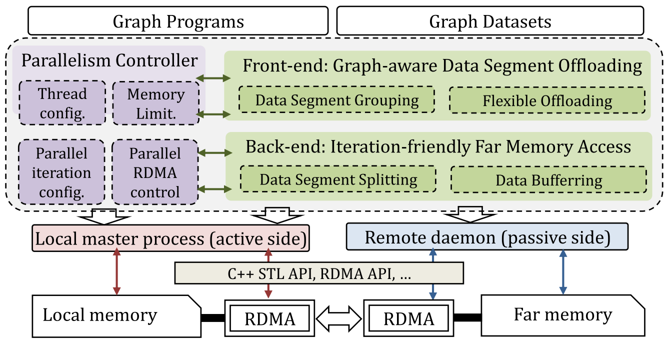

Graph Programs
Graph Datasets
Parallelism Controller
Thread config.
Memory Limit.
Parallel iteration config.
Parallel RDMA control
Front-end: Graph-aware Data Segment Offloading
Data Segment Grouping
Flexible Offloading
Back-end: Iteration-friendly Far Memory Access
Data Segment Splitting
Data Bufferring
Local master process (active side)
Remote daemon (passive side)
C++ STL API, RDMA API, ...
Local memory
RDMA
RDMA
Far memory

Fig. 7. An overview of Fargraph+ platform.

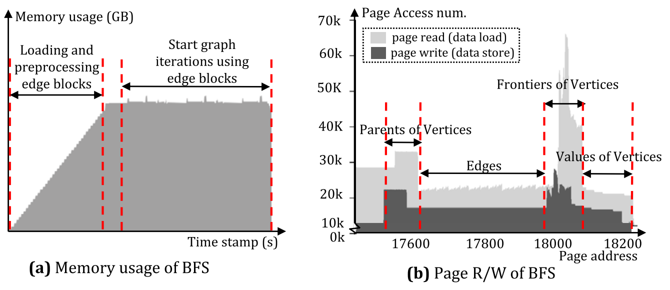


Fig. 8. We test the access frequency of each page in BFS. We show some distinguished data segments in the graph processing program.

far memory architecture in parallel efficiently. Fig. 7 provides an overview of Fargraph+ which mainly consists of three parts.

(1). Front-end. The graph-aware *Front-end* is the working set partition strategy that decides how to offload graph data segments by analyzing the memory access patterns of the graph. Fargraph+ front-end identifies and separates the working sets into two parts previously at the granularity of data segment. The front-end keeps hot data working sets in the local memory region and uses interfaces to fetch the other data working sets to remote regions with relative keys.

(2). Back-end. The RDMA-based *Back-end* designs the directive-Like far memory interaction interfaces to build up efficient iteration-friendly far memory access coordination on RDMA. Fargraph+ back-end implements the main optimization with the user-level RDMA operation. The back-end buffers the transmissions to support the one-side access and implement the overlap of data fetching and processing to reduce total latency.

(3). Controller. The parallelism-oriented *Controller* performs parallelism control of the graph data fetching and processing of the entire program, giving proper directions to the *Front-end* and *Back-end* according to the outsider performance pressure and resource limitation. Fargraph+ controller selects proper thread numbers and memory usage limitations for the current applications according to the type of graph algorithms and datasets. The controller configures shared-memory parallel programming in each iteration of the original graph programs. It also configures the parallel RDMA mechanism with queue resource sharing to achieve higher performance and efficiency of far memory access.

All the designs are implemented in the two distinct procedures. 1) the master process that resides on the active side (the client) and 2) the daemon process on the passive side (the far memory server). Our key strategies are mainly implemented on the active side, with assistance from the passive side. In Section 8 we introduce our implementation.

5. The front-end: 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.

5.1. Graph data segment grouping

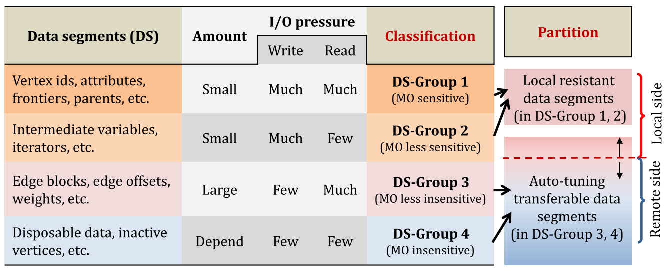
We investigate the data segments of stacks, queues, as well as graph-specific data items (e.g., the array of vertices) in the graph program. Data segment consists of a set of pages. In Fig. 8, we show the page access statistics when running BFS on a graph in LiveJournal format. We perform page address 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 at 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.

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. Fig. 9 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 T. If the page is accessed at time T, 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.

5.2. 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 memory server may cause nontrivial performance degradation.



Data segments (DS)	Amount	I/O pressure		Classification	Partition
		Write	Read		
Vertex ids, attributes, frontiers, parents, etc.	Small	Much	Much	DS-Group 1 (MO sensitive)	Local resistant data segments (in DS-Group 1, 2)
Intermediate variables, iterators, etc.	Small	Much	Few	DS-Group 2 (MO less sensitive)	
Edge blocks, edge offsets, weights, etc.	Large	Few	Much	DS-Group 3 (MO less insensitive)	Auto-tuning transferable data segments (in DS-Group 3, 4)
Disposable data, inactive vertices, etc.	Depend	Few	Few	DS-Group 4 (MO insensitive)	

Fig. 9. The description and classification of data segments. We show the data partition method based on transferable data segments on the right part.

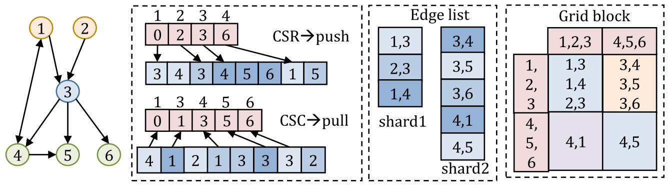


Fig. 10. The graph data structure of CSR, CSC, Grid Block and Edge List.

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*[7] which defines a set of pages that are critical to programs when processing 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 Fig. 9.

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.

5.3. Parallel data segments fetching

Proper graph data structure. Fetching graph data segments in the memory space in parallel can shorten the overall latency. However, random memory access in graph processing programs may block parallel processing. Fortunately, graph data are often preprocessed from raw edge lists to CSR, CSC, or grid block structure in today's computing framework [34,54]. Fig. 10 shows the general graph data structure of CSR, CSC, grid Block, and edge List. The preprocessing procedure makes it easier to capture graph properties and to offload graph data in a more efficient and fine-grained manner. Inspired by GridGraph [54], we can use ordered graph data so that more graph data can be accessed from registered physical contiguous memory. The ordered grid block data structures gain more benefits from parallel processing on far memory system compared with unsorted CSR or CSC data structures.

Multi-thread data block fetching. Also, there are some tricks when assigning threads to programs in our system. The overall

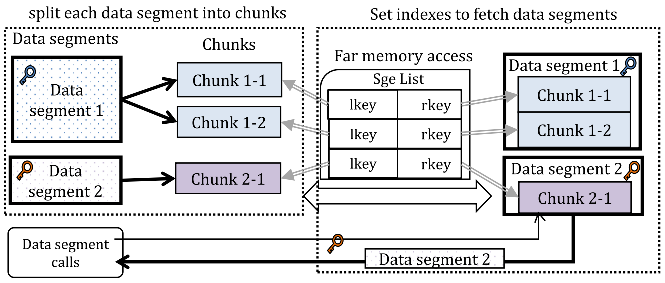


Fig. 11. We split data segment into chunks. We use indexes to facilitate data segment fetching on far memory.

performance is greatly influenced by the number of graph blocks in the graph grid. The programs often perform when the thread number is an integer multiple of the graph partition number. Specifically, Executing processes the same as the number of CPU cores in a machine and spreading a process to a different CPU can be a better practice compared to assigning all the threads on the same processor. In addition, when working with Symmetric Multi-Processing (SMP) machines with NUMA support, binding the processes to specific CPU cores may provide better utilization of the CPU resource thus providing better performance. Thus in our system, we use *taskset* utility to set processor affinity for each process to optimize overall latency.

6. The back-end: 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.

6.1. Data segment splitting

Appropriate data transfer is critical. RDMA-based far memory environment 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 of Fig. 11 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 Fig. 11 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.

6.2. Data segment buffering

Our second optimization overlaps data computation and communication. We configure the system in such a way that it starts

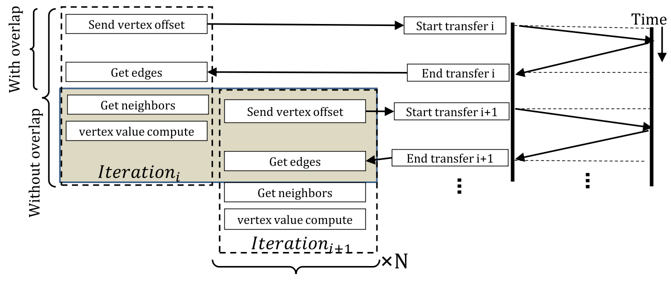


Fig. 12. The iteration-friendly far memory access with overlapping.

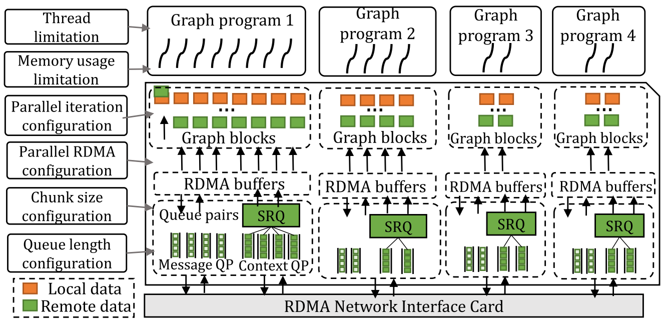


Fig. 13. An example of parallel control on Fargraph+.

to request the data for the next iteration when the current iteration is still underway as Fig. 12 shows. We design buffers that support iteration pipeline overlap by hiding communication when fetching data segments, shown in Fig. 14. 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 a 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 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 more 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 hidden 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 a smaller frontier size (i.e., shorter transfer time) in their early-stage iterations, and therefore we observe notable performance improvement at the beginning of their execution.

6.3. Parallel RDMA configuration

Appropriate RDMA configurations are essential for maximizing the benefits of parallel graph processing on far memory. It is thus important to set proper data channels for the transferred data and instructions. An optimized operation is to *separate control operations and data channel* so that we can parallel the transferring of control-related operations and data read/write operations. In RDMA protocols, the transferred data through RDMA includes *Small* messages and *Large* contexts. *Small* messages guarantee the promotion of event-based RDMA communication, which includes control operations (create/destroy/query/modify), data indexes, RDMA event states, read or write requests, etc. Oftentimes, they are very expensive because most of the time, they perform a context switch. Sometimes they allocate or free dynamic memory and sometimes they are involved in accessing the RDMA device. Sending these small messages in a special queue in *Message Queue Pairs (QP)*, as shown in Fig. 13) will provide better latency since it eliminates the need for the RDMA device to perform an extra read (over the PCIe bus) in order to read the message payload. As for *Large* contexts, i.e. offloaded data on the far memory, they often provide large data chunks that can feed to local programs. Thus, we send them in the Work Request of the *Context Queue Pairs (QP)* (as shown in Fig. 13) to gain faster far memory access.

In order to improve memory efficiency, we further compress the space by configuring RDMA queues on the basis of ensuring application performance. We carefully configure the queue length and queue number to reduce the memory resource usage of RDMA. Memory consumption of RDMA protocol mainly depends on the total memory size of data contexts in RDMA queues. To save memory space, it is better to reduce the size of the used queues to the minimum. In each thread, one should have at least one message queue pair (send queue and receive queue) and one data context pair. However, when configuring the Queue Pair(QP) in multi-thread conditions, the size of queues grows rapidly. Instead of having a separate receive queue and posting many receive requests for every queue pair, we set Shared Receive Queues (SRQ) to save the total number of outstanding receive requests and reduce the total consumed memory, as shown in Fig. 13. Second, setting message queues (which is stated above) can reduce local memory usage since small messages require smaller buffers. As graph workloads are memory-intensive, we rarely have performance down gradation due to cutting queues while keeping the RDMA device busy. We will further show the effectiveness of our design in our evaluation.

7. The parallelism-oriented control

Inspired by CongraPlus system which develops a NUMA-aware scheduler [27], we designed a multi-threaded controller with two phases: offline collection phase and online allocation phase. In the *offline data collection phase,* the commonly used graph datasets and graph algorithms are synthesized and run with a different number of threads to obtain the information important for scheduling. We run the synthesis of graph datasets and algorithms to collect run-time information. We build offline tables of execution time, threads, far memory ratio, dataset partition number, queue length, etc. In the *online resource allocation phase*, the controller configures the resource limitations according to the available remaining resource and the offline table to achieve optimal efficiency (best performance on the limited resource) for the program.

We give an example of the parallelism control procedure of Fargraph+, as shown in Fig. 13. We have implemented the optimized configuration of parallel iteration, parallel RDMA, chunk size and queue length in the program. We further allocate proper thread and far memory resource to the program according to the offline table. To perform better on NUMA architecture, we assign different tasks to different sockets to get balanced loads. By binding the execution of a specific task to a socket, the Linux first-touch mechanism can be utilized to avoid cross-node memory access.

Thread allocation. In the thread allocation module, we prefer to give more cores to tasks that are can achieve better overall latency in multi-threaded situations. Performance improvement is collected and calculated in the offline table. We measure the reduction of overall runtime when assigning one more core and assign the core to the task with the largest time reduction. The allocation will be repeated until a conflict arises or multi-core no longer brings performance gains. In this work, we reduce conflict drawbacks by stopping launching a task without minimum available resources in the system. If there is no free CPU core or the free local memory cannot meet the turning point local memory requirements of the current task, it will not be launched.

Far memory allocation. We adjust the far memory ratio of each arrived program according to the resource limitation of the current system. We will offload the insensitive graph data segments directly according to the memory resource limitation. As discussed in 3.1, there is a runtime turning point in the graph program as the far memory ratio increases. In order to make more efficient use of local memory, we set the far memory ratio of all the concurrently running graph programs to the ratio corresponding to the performance turning point. In this case, the local memory of each task can be squeezed as much as possible so that the server can maintain more graph programs to achieve higher task throughput and better overall memory efficiency.

8. Directive-like implementation

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

8.1. 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 flags to each data segment in a data segments list (DS_list) based on the given far memory ratio (detailed in Section 3.1). *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 regions, respectively. They are transferred along with the data. *Far_read_start()* starts a one-sided read of each data segment and implies the beginning time of data segment fetching. To cooperate with multi-thread control, the thread ID is passing as a parameter and communicates with each RDMA data queue provided in each thread. *Far_read_complete()* returns the rkey and index of the fetched data when the data transmission finishes. Our 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. Then, the later *Far_read_start()* are placed as soon as getting the next neighbors in each iteration, as shown in Fig. 12. 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 transfers. *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+.

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:  #pragma omp parallel num_threads(parallelism)
5:  for each DS_i in transferable_DS_list do
6:      Far_write_start(trans_flag, DS_i, index, lkey, threadID, ...);
7:  end for
8:  Far_write_complete(DS_indexes, rkey, threadID, ...);
9:  ...continue... //waiting for data segments calls
10: //start read far DS_Current in another process;
11: Far_read_start(DS_Current, index, rkey, threadID, ...);
12: while (in each processing loop) do
13:     ...continue... //original data process
14:     #pragma omp parallel num_threads(parallelism)
15:     while calling DS_Current do
16:         if DS_Current is prepared then
17:             Far_read_complete(DS_Current,index,threadID, ...);
18:         end if
19:     end while
20:     // start receive the next DS;
21:     Far_read_start(DS_Next, index, rkey, threadID, ...)
22:     ...continue...//original data process
23:     if DS_Current finishes occupying then
24:         Free DS_Current in local RAM;
25:     end if
26: end while
```

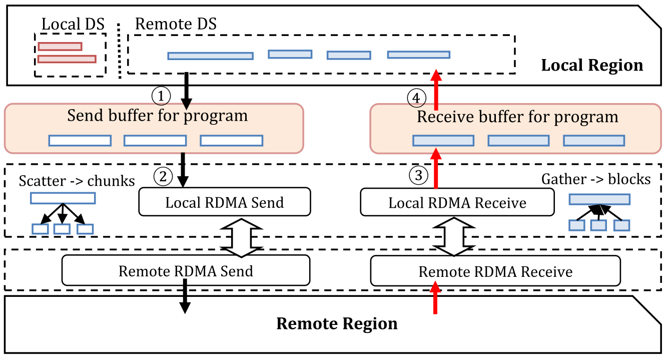


Fig. 14. The detailed workflow of data segment pre-transferring and far memory coordination on Fargraph+.

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 graph data into the main memory before preprocessing, and we can replace the buffer copy operations with our APIs. One can extend our design to support new runtimes and protocols of specific devices like NVLink [11] and CXL fabrics [8]. We leave the extension of supporting more applications and more far memory backends to future work.

8.2. Fargraph+ workflow

The workflow of Fargraph+ is shown in Fig. 14.

(1). Pre-processing. We start by configuring the thread number of graph programs in the front-end with customed parallelism. RDMA queues in the back-end. To start far memory access initialization, we create RDMA queues and 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

Table 1
Evaluated System Management Strategies.

Schemes	Applications	Mem. Limit.	Par. Limit.	Exe. Environment
Original	GridGraph	Yes	No	Disk
Fastswap	GridGraph	Yes	No	kernel-level FM
Fargraph	Modified GridGraph	No	No	user-level FM
Fargraph+	Modified GridGraph	Yes	Yes	user-level FM
Oracle	Modified GridGraph	No	No	Local memory

Table 2
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
RMAT-1T(RM1)	100M	1T	76.2GB	202.2GB
RMAT-5T(RM5)	100M	5T	152.4GB	354.4GB

Table 3
Evaluated Graph Processing Algorithms.

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

authorization. We also build RDMA queues (including shared 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 asynchronously. We then pre-transfer the decided data segments to far memory on the passive server with RDMA write (①, ② in Fig. 14) and obtain their far memory keys.

(2). Far memory coordination. Fig. 14 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()*. Note that we fetch graph data in parallel graph iterations with multiple threads, we use shared receive queues for fetching remote data segments thread each thread. We fetch corresponding edge blocks (i.e. DSs) in order and start the next until all the current concurrent data arrive. 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 a buffer to pre-fetch the transferred data asynchronously (③ in Fig. 14). We also use *srq_post_receive* to continuously receive data read from far memory through the shared receive queue 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 Fig. 14).

9. Experimental setup

We introduce the experimental setup design which supports the design choices.

9.1. 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. We use *OpenMP* interfaces to parallel our graph programs. We also use *pthread* to manage the threads allocation and communication in the local process. Each node is provisioned with two 16-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 5.6.0 of the OFED kernel, and it uses RoCE (RDMA over Converged Ethernet) protocol.

9.2. Evaluated system strategies

We consider the following strategies as Table 1 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 kernel-level far memory platform which outperforms many previous works [13,20]. It is an RDMA-based far memory platform with swap kernel and local disk involved [2]. We consider it as a key baseline strategy in this work. 3) *Fargraph*. This scheme uses all the optimizations that we propose without parallel configuration and resource optimization. 4) *Fargraph+*. This scheme uses all the optimizations that we propose. 5) *Oracle*. This is the ideal design case of far memory, which keeps all the data in the local main memory (best performance).

9.3. Evaluated graph workloads

We evaluate 6 graph datasets together with 4 representative graph algorithms in our experiment. The datasets contain 4 real-world graphs including LiveJournal (LJ), Orkut (OR), Twitter7 (TW), Friendster (FR), and 2 generated large graphs including RMAT graph with 100 trillion edges (RM1) and RMAT graph with 500 trillion edges (RM5) generated by PaRMAT tool [16]. Note that the memory footprint of RM1 and RM5 exceeds the size of local memory in our evaluation. More details are given in Table 2. The evaluated graph algorithms are shown in Table 3. 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 [54] 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+). Note that the performance comparison between disk-based and RDMA-based works is almost one order of magnitude, and one can refer to the speedup of far memory over disk I/O in previous works [13,20,2].

10. Results

This section presents detailed experiment results that further support our design choices and demonstrate the efficiency of Fargraph+. We compare our design with the classic single-node graph processing framework GridGraph [54] and the state-of-the-art RDMA-based far memory engine Fastswap [2]. We also give the performance breakdown to show the optimization on Fargraph+.

10.1. Overall performance

We first present the overall optimization effectiveness of Fargraph+ across 24 workloads. Fig. 15 compares Fargraph+ with all the other evaluated schemes.

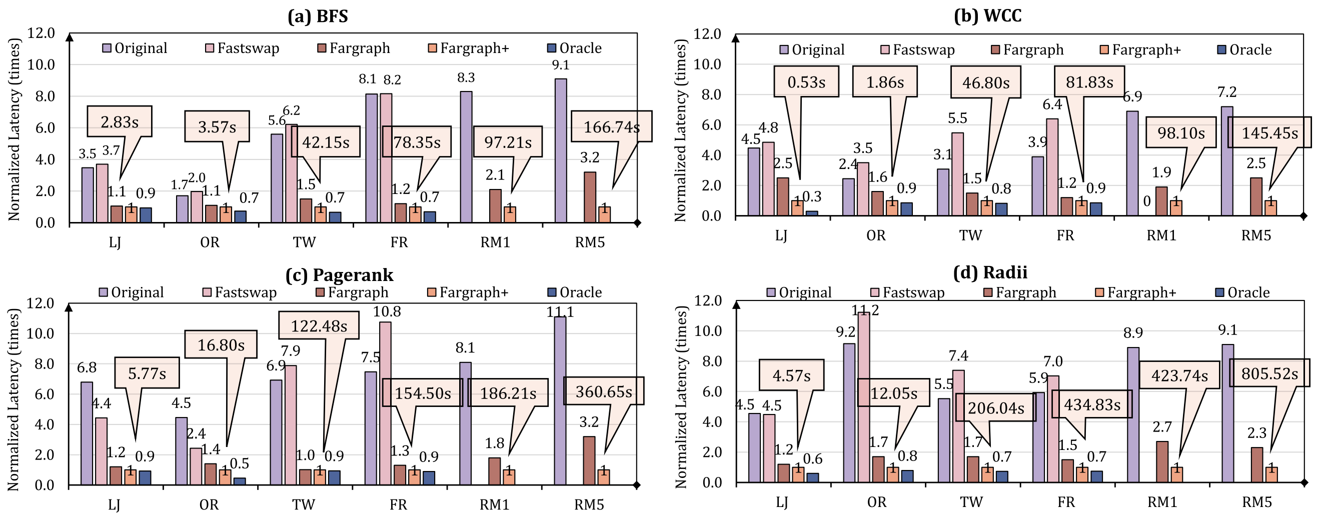


Fig. 15. The total performance comparison of baselines by testing 24 graph workloads on Fargraph+. We normalized the Latency of each baseline by dividing the latency of our system. The values of the bars refer to the speedup of our work compared with all the baselines.

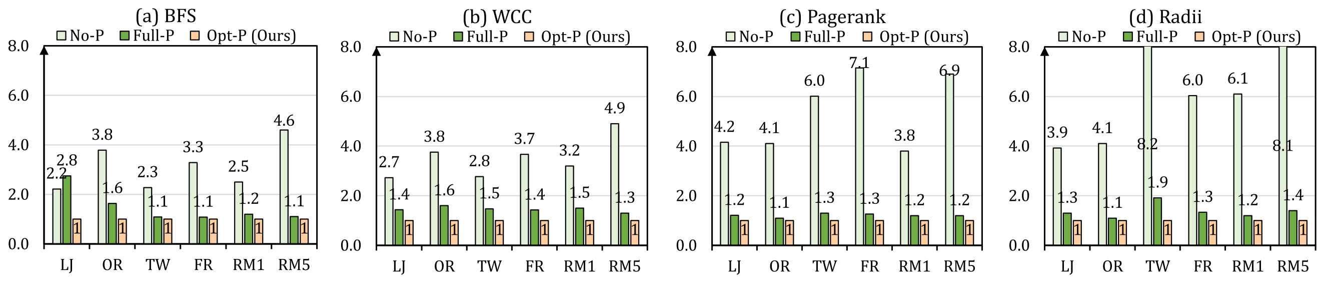


Fig. 16. We show the performance comparison of 24 graph workloads with multiple threads on Fargraph+. No-P represents single-thread allocation with no parallelism. Full-P refers to starting as many threads as possible (32 threads in our evaluation) on each CPU core with full parallelism. Opt-P is our method with optimal thread allocation that can achieve the best performance.

For many datasets, computation-centric algorithms like PageRank and Radii show relatively higher performance improvement compared to 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 parallelism potential can be more obvious when processing computation-centric algorithms on larger graphs such as PageRank and Radii on RM1 and RM5, which has more chance to hide latency in computing 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. On larger graph datasets RM1 and RM5 which can not be fit into local memory, we compare the baselines including Original, Fargraph and Fargraph+. Fastswap has limited available far memory size and cannot handle them. In the results, we observe remarkable performance speedup than baselines and the parallel potential on larger datasets. We show significant performance speedups, such as 9.1× of BFS and 11.1× of PageRank on RM5. The experimental results show that greater benefits can be gained from parallelism on larger datasets. For example, Fargraph+ shows more speedup than Fargraph on larger datasets such as 3.2× of PageRank on RM5 and 2.7× of Radii on RM1.

Overall, the results show that Fargraph+ is more efficient and is closer to an oracle design compared with Fastswap. We can achieve up to 9.2× better performance compared to *Original*, and up to 11.2× performance compared to *Fastswap*. Compared with Fargraph, we also achieve up to 3.2x speedup due to the proper design of parallel data fetching and optimized RDMA queue configuration, especially on larger datasets. 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.

10.2. Performance of parallelism control

We test the parallelism control performance of the 24 workloads on Fargraph+ system by offloading all edge data to far memory. We assign them with different threads to show the speedup of parallel design on Fargraph+. We test programs with 1 to 64 threads separately and analyze the results. We list the overall latency of three conditions in Fig. 16, including 1 thread with no parallelism (short as No-P), thread number equal to CPU cores with Full parallelism (short as Full-P), and the optimized thread number that can achieve the best performance in Fargraph+ system (short as Opt-P). To better present the parallelism speedup, we normalized the overall latency with the least latency (i.e. Opt-P). We show a different range of performance speedup over the sub-optimal conditions. Overall, our parallelism control can have the best performance and higher efficiency by saving CPU and memory resources.

Graph algorithms show different latency trends on various levels of parallelism. In our evaluation, computation-centric algo-

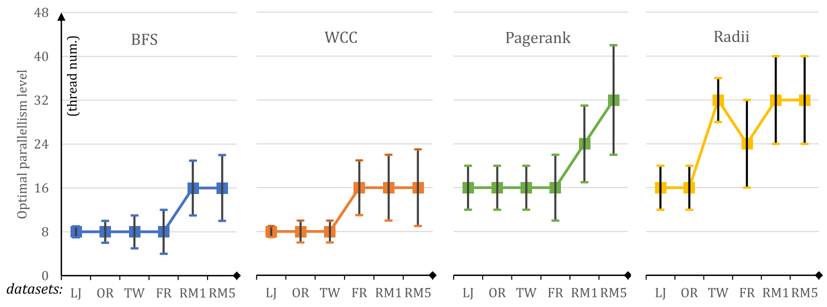


Fig. 17. The optimal parallelism ranges and the median values of graph workloads.

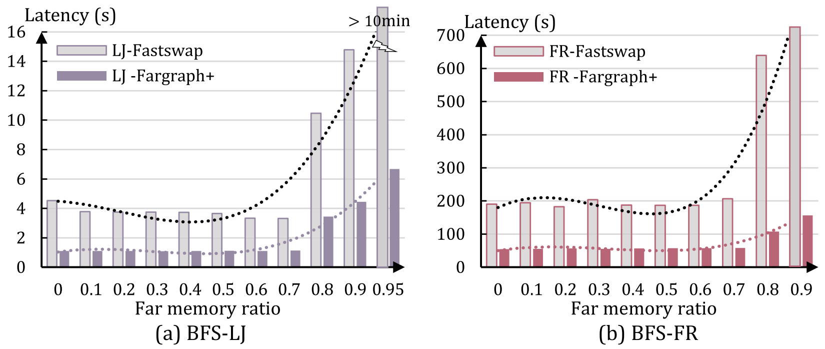


Fig. 18. The duration of BFS on dataset LJ and FR on far memory platform Fargraph+ and Fastswap under rising far memory ratios.

rithms like PageRank and Radii show relatively higher performance improvement on higher parallelism compared to the traversal-centric algorithms, such as BFS and WCC. For example, the best speedup of BFS and WCC is 4.9x and 4.6x on RM5 dataset in Fig. 16-(a) and (b), while the best speedup of PageRank and Radii is 7.1x and 8.2x on FR and TW in Fig. 16-(c) and (d), respectively. Furthermore, WCC, PageRank, and Radii have a larger thread number of best performance. This infers that computation-centric algorithms can have more data parallelism opportunities when adding parallel design into the system.

Graph programs on large graphs can benefit more from parallelism design on larger datasets. Larger datasets like TW and RM5 show up to 8.2x and 8.1x performance speedup when using more threads, as shown in Fig. 16-(d). We present the optimal thread numbers range of each graph workload that can achieve over 2 times of speedup without taking up additional threads, as shown in Fig. 17. The results show that larger datasets prefer more threads. For example, WCC achieves the best performance with 8 threads on LJ, OR, TW, and 16 threads on FR. The reason is that larger datasets have more iterations of data fetching, so they can have higher parallelism when data is fetched from far memory in parallel. In addition, we show that computation-centric algorithms can benefit more from larger parallelism due to hiding more transfer latency in each iteration. For instance, PageRank and Radii show better performance with at least 16 threads on each dataset. Overall, our parallelism design shows obvious scalability on graph applications and is friendly to large datasets.

10.3. Performance of the front-end

Efficiency of Data Offloading Design. We start by evaluating the performance of Fargraph+'s front-end optimization, namely the graph-aware data segment offloading. We show that workload awareness allows Fargraph+ to achieve better performance. In Fig. 18 we compare Fastswap and Fargraph+ on BFS under different far memory ratios (the ratio of far memory usage and total memory demand).

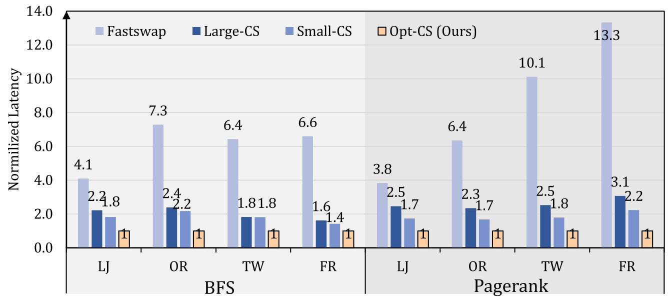


Fig. 19. The normalized performance of eight workloads (BFS and PageRank on LJ, OR, TW, FR datasets) with different chunk sizes, including large size Chunks (Large-CS) Small size chunks (Small-CS) and optimal chunk size (Opt-CS) which get best performance.

As shown in Figs. 18-(a) and (b), 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 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 could 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.

Performance Impact of Data Segment Splitting. Since Fargraph+ relies on data segment splitting (detailed in Section 6) to improve far memory efficiency, determining the appropriate chunk size is critical. In Fig. 19, 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

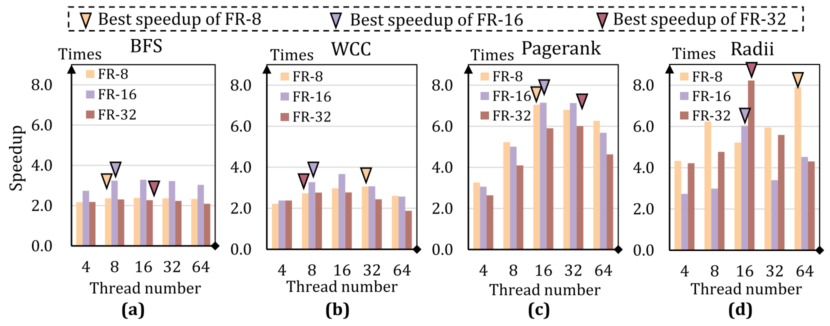


Fig. 20. The duration of graph algorithms on dataset FR with different partitions.

Table 4
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	Absolute value	↓ 0.48	↓ 0.6	↓ 12.38	↓ 23.38
reduction	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	Absolute value	↓ 0.52	↓ 3.28	↓ 29.49	↓ 32.78
reduction	Relative value	7%	12%	19%	14%

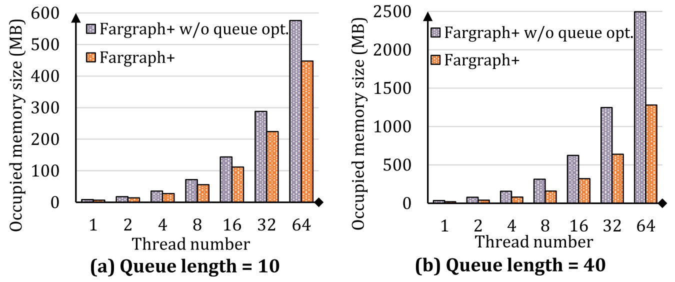


Fig. 21. The memory size comparison of original and optimized RDMA queue configuration.

reason for the smile-like curve is that the best far memory chunk size is not only decided by graph iterations but also relevant to 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.

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.

Furthermore, graph partitions affect the overall performance, as Fig. 20 shows. We test the latency of each workload on FR dataset with 8, 16 and 32 partitions over the best chunk size. The largest partitions may not have more benefits from higher parallelism. We flag the best speedup of applications on each partition number and we can configure the best thread number when using different data partitions. In most cases, BFS and PageRank graph programs with larger partitions gain more performance speedup on larger parallelism, while WCC and Radii graph programs with more partitions perform better when limiting the parallelism.

10.4. Performance of the back-end

Performance Impact of Data Segment Buffering. We 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 4, 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 Table 4, 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.

Efficiency of RDMA Queue Optimization. We further evaluate the memory consumption reduction of the optimized RDMA queue configuration by comparing Fargragh+ and Fargraph+ without queue optimization. The results are shown in Figs. 21-(a) and (b). The longer the queue length, the more events and buffered context spaces are left in local memory for data transferring. Thus, we measure the memory capacity savings brought by the optimization of queue configuration in the cases of multiple threads with different queue lengths. Figs. 21-(a) and (b) shows the results of memory occupation when queue length is 10 and 40 on different parallelism level.

Overall, the results show that Fargraph+ can compress up to 50% space compared with Fargraph+ on the basis of ensuring application performance. One can have more memory saving when using more threads. The reason is that we use shared receive queues to share events from each thread. Memory space saving is more obvious when using a longer queue length, saving up to 23% in the queue length of 10 and up to 50% in the queue length of 40 compared to the results in Figs. 21-(a) and (b). This is due to the design of message and context queue pair reconfiguration and we can save memory space corresponding to message events.

11. Related work

Disaggregated Memory Architectures. *Composable Disaggregated Infrastructure* (CDI) [23,19] 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 [32] introduces modular system implementation for hardware disaggregation. String-finger [24] builds a large memory pool with thousands of memory nodes and tens of CPUs. There are several works [20,2,36] focusing on extending their local main 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 far memory platforms.

System Support for Graph Processing. In general, there are three types of graph processing frameworks. 1) *In-memory* graph frameworks, such as Ligra [34], Cagra [51], GraphIt [53], and etc. In memory frameworks process graphs after all the source data are loaded into main memory [39]. 2) *Out-of-core* frameworks, such as GridGraph [54], Mosaic [21], HUSGraph [44] process large graphs with limited main memory and a large-capacity disk. Works with out-of-core execution patterns [41,33] load each graph block into the memory and process them streamingly. 3) *Distributed* frameworks, such as GraM [43], Gemini [55], and Chaos [29], 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 [48,42,15,50] and graph-based ML/AI applications [40].

Parallel Graph Processing Management. Parallel graph processing frameworks have different types of parallelism management methods when serving requests in a multi-user environment. Congra [26] proposes a scheduling method of multiple graph queries by collecting the memory bandwidth consumption characteristics on the optimal CPU cores. Kim et al. [17] devised methods for enabling efficient processing of multiple graph queries using MapReduce. Xue et al. [47,46] supports concurrent graph processing queries and proposes a graph structure sharing mechanism to avoid memory storage waste. Cgraph [52] designs a correlations-aware execution model, together with a core subgraph-based scheduling algorithm, to efficiently share the graph structure data in cache/memory and their accesses. GC-graph [45] shares the I/O access and processing of graph data among the CGP jobs and adaptively schedules the loading of graph data, which efficiently overcomes the I/O challenges in prior works. Uni-address [1] demonstrated a new thread management scheme which enables us to implement RDMA-based work stealing and reduces virtual memory usage of thread migration. Fargraph+ explores multi-threading effects in far memory environment and improves the efficiency of parallel graph processing.

RDMA-based Far Memory Acceleration. With kernel-bypass and fast-messaging features, the 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 [13] 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 [20]. 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 [43] processes graphs with distributed computing, using RDMA to pass messages. FAM-graph [49] offloads all graph edges to the remote disaggregated memory to efficiently tier data between local and remote memory. Different from existing works, Fargraph+ manages transferable data segments for graph workloads and optimizes graph processing with tailored RDMA control.

12. Conclusion and discussion

In this paper, we explore graph processing on emerging far memory architecture. We show that there are several challenges and opportunities in deploying graph workloads on far memory. We propose Fargraph+, an optimization strategy that allows one to run graph applications on far memory in parallel efficiently. The key novelty of Fargraph+ is three-folded, including the smart graph-aware data segment offloading, the adaptive far memory interaction and the efficient parallelism optimization. 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 9.2× and 11.2× 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 on disaggregated architecture.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Chao Li reports financial support was provided by National Natural Science Foundation of China (No. 61832006 and No. 61972247), and by Alibaba Innovative Research Program (No. ATS54DHZ1210007).

Data availability

Data will be made available on request.

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 (No. ATS54DHZ1210007). We thank all the anonymous reviewers for their valuable feedback.

References

[1] S. Akiyama, K. Taura, Uni-address threads: scalable thread management for rdma-based work stealing, in: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2015, pp. 15–26.
[2] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M.K. Aguilera, A. Panda, S. Ratnasamy, S. Shenker, Can far memory improve job throughput?, in: European Conference on Computer Systems (EuroSys), 2020, pp. 1–16.
[3] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, 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.
[4] I. Calciu, M.T. Imran, I. Puddu, S. Kashyap, H. Al Maruf, O. Mutlu, A. Kolli, Rethinking software runtimes for disaggregated memory, in: Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021, pp. 79–92.
[5] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-mat: a recursive model for graph mining, in: SDM, 2004, pp. 442–446.
[6] E. Choukse, M.B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, 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.
[7] 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.
[8] C.E.L. Corporation, Compute express link, https://www.computeexpresslink.org/about-cxl, 2022.
[9] I. Corporation, Ibm power 9 cpu, https://www.ibm.com/it-infrastructure/power/power9, 2022.
[10] M. Corporation, Mellanox interconnect community, https://community.mellanox.com/s/, 2022.
[11] N. Corporation, Nvlink interconnect, http://www.nvidia.com/object/nvlink.html, 2022.

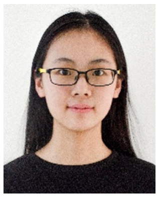
[12] A. Dragojević, D. Narayanan, M. Castro, O. Hodson, Farm: fast remote memory, in: USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2014, pp. 401–414.
[13] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, K.G. Shin, Efficient memory disaggregation with infiniswap, in: USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2017, pp. 649–667.
[14] N. Hjelm, M.G. Dosanjh, R.E. Grant, T. Groves, P. Bridges, D. Arnold, Improving mpi multi-threaded rma communication performance, in: Proceedings of the 47th International Conference on Parallel Processing (ICPP), 2018, pp. 1–11.
[15] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, 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, pp. 1–14.
[16] F. Khorasani, R. Gupta, L.N. Bhuyan, Parmat, a multi-threaded rmat graph generator, https://github.com/farkhor/PaRMAT, 2022.
[17] S.-H. Kim, K.-H. Lee, H. Choi, Y.-J. Lee, Parallel processing of multiple graph queries using mapreduce, in: The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA 2013), 2013, pp. 33–38.
[18] 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.
[19] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S.K. Reinhardt, T.F. Wenisch, Disaggregated memory for expansion and sharing in blade servers, in: International Symposium on Computer Architecture (ISCA), 2009, pp. 267–278.
[20] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, Y. Wu, Memory disaggregation: research problems and opportunities, in: IEEE International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 1664–1673.
[21] 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.
[22] J. Malicevic, B. Lepers, 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.
[23] D. Montgomery, The future of data infrastructure: cdi, https://www.datacenterknowledge.com/industry-perspectives/future-data-infrastructure, 2022.
[24] M. Ogleari, Y. Yu, C. Qian, E. Miller, 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.
[25] L. OS, Linux frontswap, https://www.kernel.org/doc/html/latest/vm/frontswap.html, 2022.
[26] P. Pan, 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.
[27] P. Pan, C. Li, M. Guo, Congraplus: towards efficient processing of concurrent graph queries on numa machines, IEEE Trans. Parallel Distrib. Syst. 30 (2019) 1990–2002.
[28] 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.
[29] A. Roy, L. Bindschaedler, J. Malicevic, W. Zwaenepoel, Chaos: scale-out graph processing from secondary storage, in: Symposium on Operating Systems Principles (SOSP), 2015, pp. 410–424.
[30] Z. Ruan, M. Schwarzkopf, M.K. Aguilera, A. Belay, Aifm: high-performance, application-integrated far memory, in: USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2020, pp. 315–332.
[31] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, J. Wilkes, Omega: flexible, scalable schedulers for large compute clusters, in: ACM European Conference on Computer Systems (EuroSys), 2013, pp. 351–364.
[32] Y. Shan, Y. Huang, Y. Chen, 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.
[33] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, M. Guo, Oversubscribing gpu unified virtual memory: implications and suggestions, in: ACM/SPEC International Conference on Performance Engineering (ICPE), 2022, pp. 67–75.
[34] J. Shun, 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, pp. 135–146.
[35] M. Si, A.J. Pena, P. Balaji, M. Takagi, Y. Ishikawa, Mt-mpi: multithreaded mpi for many-core environments, in: Proceedings of the 28th ACM International Conference on Supercomputing (SC), 2014, pp. 125–134.
[36] S.-Y. Tsai, Y. Shan, 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.
[37] K. University of Tennessee, A message-passing interface standard version 3.1, https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2022.
[38] J. Wang, C. Li, T. Wang, L. Zhang, P. Wang, J. Mei, M. Guo, Excavating the potential of graph workload on rdma-based far memory architecture, in: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2022, pp. 1029–1039.
[39] P. Wang, L. Zhang, C. Li, M. Guo, Excavating the potential of gpu for accelerating graph traversal, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 221–230.
[40] P. Wang, C. Li, J. Wang, T. Wang, L. Zhang, J. Leng, Q. Chen, 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, pp. 304–317.
[41] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, M. Guo, Grus: toward unified-memory-efficient high-performance graph processing on gpu, ACM Trans. Archit. Code Optim. (2021).
[42] X. Wang, C. Li, L. Zhang, X. Hou, Q. Chen, M. Guo, Exploring efficient microservice level parallelism, in: International Parallel and Distributed Processing Symposium (IPDPS), 2022, pp. 1–10.
[43] 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: ACM Symposium on Cloud Computing (SoCC), 2015, pp. 408–421.
[44] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, Y. Zhang, Hus-graph: i/o-efficient out-of-core graph processing with hybrid update strategy, in: International Conference on Parallel Processing (ICPP), 2018, pp. 1–10.
[45] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, Y. Zhang, P. Fang, Graphcp: an i/o-efficient concurrent graph processing framework, in: 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS), IEEE, 2021, pp. 1–10.
[46] J. Xue, Z. Yang, Z. Qu, S. Hou, Y. Dai, Seraph: an efficient, low-cost system for concurrent graph processing, in: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, 2014, pp. 227–238.
[47] J. Xue, Z. Yang, S. Hou, Y. Dai, Processing concurrent graph analytics with decoupled computation model, IEEE Trans. Comput. 66 (2016) 876–890.
[48] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, J. Xue, A locality-aware energy-efficient accelerator for graph mining applications, in: IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 895–907.
[49] D. Zahka, A. Gavrilovska, Fam-graph: graph analytics on disaggregated memory, in: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2022, pp. 81–92.
[50] L. Zhang, W. Feng, C. Li, X. Hou, P. Wang, J. Wang, M. Guo, Tapping into nfv environment for opportunistic serverless edge function deployment, in: IEEE Transactions on Computers (TC), 2021, pp. 1–10.
[51] 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.
[52] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, H. Liu, Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018, pp. 441–452.
[53] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, J. Shun, Optimizing ordered graph algorithms with graphit, in: The International Symposium on Code Generation and Optimization (CGO), 2020, pp. 158–170.
[54] X. Zhu, W. Han, 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.
[55] 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.

Jing Wang is a Ph.D. candidate at Shanghai Jiao Tong University, China. Her current research interests include computer architecture, disaggregated memory, and graph processing.

Chao Li is a professor with tenure in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His primary research area is system architecture design with an emphasis on energy-efficient, high-performance computers of large scale.

Yibo Liu is working toward the Master degree at Shanghai Jiao Tong University, China. His research interests include disaggregated memory and graph-structured analytics.

Taolei Wang is a Ph.D. student at Shanghai Jiao Tong University, China. His current research area includes computer architecture, disaggregated memory and cloud computing.



Junyi Mei is a Ph.D. student at Shanghai Jiao Tong University, China. Her current research interests include computer architecture and graph processing.

Lu Zhang got his Ph.D. from Shanghai Jiao Tong University, China. His research interests include edge computing, network function virtualization and serverless computing.

Pengyu Wang got his Ph.D. from Shanghai Jiao Tong University, China. His research interests include systems and architectures for graph processing and graph neural network.

Minyi Guo is an IEEE fellow and a chair professor in the Department of Computer Science and Engineering of Shanghai Jiao Tong University, China. He was the department head from 2009 to 2019. His research area includes parallel and distributed processing, compilers, cloud computing, pervasive computing, software engineering, embedded systems; green computing, etc.