Towards Disaggregated NDP Architectures for Large-scale Graph Analytics

Suyeon Lee*, Vishal Rao*, Ada Gavrilovska
Georgia Institute of Technology, Atlanta, USA
sylee0506@gatech.edu, vrao79@gatech.edu, ada@cc.gatech.edu

***Abstract*—The performance of large-scale graph analytics is limited by the capacity and performance of the memory subsystem on the platforms on which they execute. In this paper, we first discuss the limitations of existing approaches to scaling graph processing, and describe how they can be addressed via the use of disaggregated solutions with near-data processing (NDP) capabilities. Using observations from experimental analysis of the trade-offs for different types of graphs and analytics kernels, we identify the systems-level mechanisms that will be required by future graph analytics frameworks for disaggregated NDP architectures.**

***Index Terms*—graph analytics, disaggregated systems, near-data processing, computational memory**

I. Introduction

The unprecedented growth of interconnected data has made graph analytics a vital component in domains ranging from social-network analytics to bioinformatics [1]. Traditionally, large-scale graph analytics has relied on distributed graph processing systems on a cluster of homogeneous general-purpose severs [2]–[5]. However, distributed graph processing has high overheads due to the significant amount of data movement and entailed synchronization phases.

To address these costs, recent research has explored the use of near-memory accelerators [6]–[8]. Graph applications are often memory bound and exhibit low computational complexity, as they traverse vertex and edge data within every loop in nested iterations. Therefore, their performance largely depends on memory capabilities such as access latency and bandwidth. This makes them an excellent candidate to benefit from solutions based on near-memory acceleration, by allowing graph operations to be performed with lower latency and increased memory bandwidth (i.e., *memory-capacity-proportional* bandwidth [6]). However, when the graph size scales to trillions of vertices and edges, these solutions still suffer from high communication and synchronization overheads. In addition, since the compute and memory resources are tightly coupled in the distributed setup, memory-intensive large-scale graph processing leads to skewed cluster resource utilization.

New technologies for disaggregated memory systems offer opportunities for better balanced and more efficient systems for these workloads, where memory capacity demand is provided via large (shared) remote memory pools [9]–[11]. Figure 1(a) illustrates how graph processing works on top of such disaggregated architecture. Using the CSR graph representation, the hosts maintain the vertex list and vertex properties in local memory, while the much larger edge lists are kept in the remote memory pool. For every iteration of the graph processing, the host CPUs fetch edge data from the remote memory and process three functions locally to update the vertex properties: *Traverse*, *Apply*, and *Update*. Since the compute is only happening at the centralized local servers, the synchronization overhead is low. Additionally, disaggregation allows independent memory scaling for balanced resource utilization [12]. However, high data movement overheads persist, as a huge amount of edge data needs to be moved within every loop.

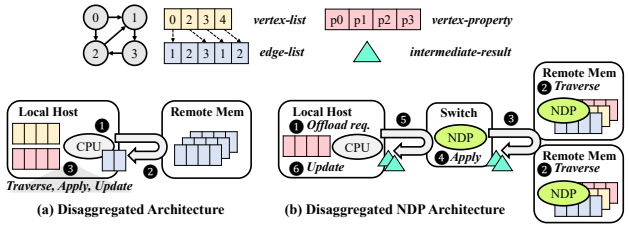


Fig. 1: Graph analytics on top of (a) disaggregated architectures vs. (b) disaggregated NDP architectures.

A more recent trend enabled by these disaggregation trends and standardization efforts such as CXL [18], is new classes of computational memory devices, which allow for processing functionality to be embedded near remote memory pools. Several such computational memory systems have recently been demonstrated [13], [14], [19]–[23]. These technologies present opportunities for high bandwidth and low latency communication between the memory units and computation, drastic data movement reductions, and better balanced system configurations, offering potential for significant improvements in end-to-end performance and system efficiency for graph workloads. Coupled with other components for in-network processing, via programmable switches and interfaces [24], [25], the current technology landscape presents opportunities for platforms with *near data processing* capabilities, where processing logic is applied near data, in memory, or in network. Such ***disaggregated NDP architectures*** present opportunities to address the communication overhead between local and remote memory and to effectively resolve the problem

This research was partially supported by the SRC/DARPA Center for Processing with Intelligent Storage and Memories (PRISM).
(* indicates equal contribution.)

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00202

TABLE I: Diverse characteristics of sample hardware with NDP capabilities.

Device Class	Examples	Device Capabilities	Target Functionality
Near-Memory Processing (PNM)	CXL-CMS [13] CXL-PNM [14]	High internal memory bandwidth (∼1.1 TB/s) Matrix/Vector computing units Support for FP operations	High memory bandwidth helps scale performance Simple vector computations that are memory-bandwidth bound FP-support increases range of supported workloads
Processing In-Memory (PIM)	UPMEM [15]	High aggregate memory bandwidth (∼1.7 TB/s) 1000s of in-order processing units (DPUs) Primitive support for FP operations	
In-Network Computing (INC)	SwitchML [16] SHARP [17]	Custom/Configurable ASICs ALUs with FP-support	Simple filter/aggregation operations FP-support increases range of supported workloads

of distribution-based systems. Figure 1(b) demonstrates its operations. For example, NDP units located close to memory, either on the remote memory or switch, allow the offload of memory-intensive graph functions such as Traverse and Apply. Disaggregated NDP architectures cover all the advantages of previous systems, including near-memory acceleration with higher bandwidth (from NDP), low synchronization overhead (from localizing shared data processing to fewer nodes), and balanced resource utilization (from disaggregation), but they also incur low communication overheads by reducing data that needs to be moved across the system interconnect.

However, although there is evidence of the benefits of NDP acceleration for graph analytics workloads, the current technology landscape introduces a number of different hardware design points. Realizing the NDP benefits for arbitrary graph workloads, is therefore not trivial. For instance, the processing capabilities impact the type of operations that can and should be offloaded. This can further be impacted by the details of the graph topology and how it is distributed (partitioned) across the system, and the consequent data movement and/or synchronization costs. Finally, graph workloads are known to exhibit irregular access patterns and dynamic behavior, presenting additional challenges to making efficient deployment decisions.

The goal of this work is to provide insights into the missing capabilities of future graph analytics frameworks, in order to realize the benefits that NDP technologies can provide for these workloads. We discuss the tradeoffs associated with existing architecture designs, and the opportunities to address their limitations via a disaggregated NDP solution. We then use data from experimental analysis of the tradeoffs for different types of graphs and analytics kernels on an emulated NDP system, to identify the systems-level mechanisms that will be required by future graph analytics frameworks for disaggregated NDP architectures.

II. Emerging NDP Hardware

Before proceeding with the discussion of the requirements of graph analytics frameworks, we first briefly summarize a few hardware designs, some of which are commercially available (Table I). The goal is to highlight some of the key features of this emerging hardware tier that will make them a good fit for disaggregated NDP architecture for graph analytics.

Two types of devices are a good fit for the NDP-enabled memory pools: *Processing Near-Memory* (PNM) and *Processing In-Memory* (PIM). These classes of devices provide high-bandwidth capabilities required to scale the traversal phase of the graph workload. CXL-PNM [14] and CXL-CMS [13] are PNM prototypes with good support for floating-point operations, making it viable to deploy more complex graph workloads such as pagerank and betweenness-centrality. UPMEM [15] is a commercially available PIM solution that offers very high aggregate bandwidth across the PIM cores. However, the limited support for complex integer operations (multiplies/divides) and floating point operations may restrict its usability for certain workloads.

In-network computing (INC) capabilities have been commercially available for a while now [26]–[28], and their utility has been demonstrated for a range of use cases [29]. SHARP [17] is an INC solution suitable for graph workloads. It uses the Mellanox SwitchIB-2 ASIC to perform the `MPI_AllReduce()` operation which can be used by graph runtimes to aggregate partial results from multiple sources. SwitchML [16] is an INC solution that uses the Intel Tofino [26] ASICs to reduce data-movement costs in distributed machine-learning applications by performing in-network aggregation. These ASICs can also be a good fit for aggregation operations in graph workloads.

III. Alternatives for Scaling Large-scale Graph Analytics

As the scale of graph analytics applications and graph data continues to rise, it is hard for a single general-purpose server to accommodate the scale of modern graphs. Large-scale graphs with billions of vertices and trillions of edges need to be partitioned and distributed across multiple machines, which leads to overheads and inefficiencies at scale. In this section, we outline the main system architectures currently used to tackle challenges related to large-scale graph processing using different approaches. The systems we explore commonly deploy graph workloads iteratively, wherein each iteration consists of a *traversal phase* and an *update phase*. Operations in these phases are applied on the graph frontier, i.e., the set of vertices that need to be processed in the current iteration. The traversal phase involves traversing the large edge list of the graph and performing simple computations to generate updates. The update phase involves applying these

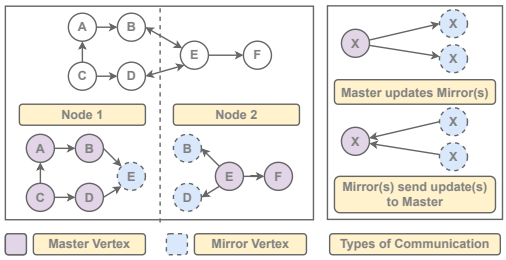


Fig. 2: Typical graph analytics on top of distributed architectures supporting graph partitioning and synchronized state across partitions.

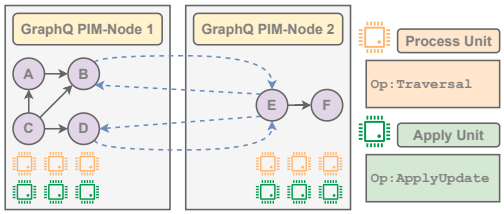


Fig. 3: Typical graph analytics on top of distributed NDP (PIM) architectures leveraging heterogeneous processing resources and near-memory acceleration for different graph operations.

updates to the structure of vertex properties and can comprise more complex computations.

A. Distributed Processing Architecture

Approaches based on distributed systems [2]–[5] enable large-scale graph processing by partitioning the graph structure across multiple servers. Gluon [2] is one such system that uses the concept of *masters* and *mirrors* to deal with graph partitioning as shown in Figure 2. The partitioning policy assigns each vertex to a host in the cluster. The master vertices on the host are the set of vertices that have been assigned to that host and they store the latest properties of the vertex. Mirror vertices are proxies of masters owned by other nodes, with their values updated by the connected master vertex. A master can have multiple mirrors and the number of mirrors depends on how the graph is partitioned. There are two types of communication patterns in this model as shown in Figure 2. During the traversal phase, the master propagates updates to its value to all its mirrors. During the apply phase, all the mirrors propagate vertex updates back to their master. A high number of mirrors results in high communication and synchronization overheads which degrades the overall performance.

Gluon uses partitioning policies and partitioning invariants to reduce the impact of these overheads, but performance can potentially degrade significantly when graphs are highly connected. In addition to the high communication and synchronization overheads, these systems have two main shortcomings. They fail to fully utilize the cluster compute capabilities leading to resource under-utilization [3], [30]. The traversal phase is bandwidth-intensive, which limits performance scalability as bandwidth does not increase with memory capacity in general-purpose servers.

B. Distributed Near-data Processing Architecture

Near-data processing (NDP) architectures [6]–[8], [31] for large-scale graph workloads are highly effective in handling the memory bandwidth limitation of general-purpose servers. For example, GraphQ [6] is an NDP solution that scales to multiple nodes. It uses specialized processing units for the different phases of the graph workload, given that each phase has different processing needs. As shown in Figure 3, GraphQ uses *Process Units* for the traversal phase and *Apply Units* for the update phase. By supporting near-memory acceleration with specialized hardware, these architectures provide *memory-capacity-proportional bandwidth* which enables the workload performance to scale with memory. They can also be designed to be more energy efficient than general-purpose servers [8].

NDP over distributed environments accelerates graph operations within individual nodes, however, the design does not fundamentally change inter-node data movement. Therefore, this architecture continues to exhibit high communication overhead. GraphQ uses a hybrid execution model that overlaps computations with inter-node communication to hide the communication overhead, however, they cannot eliminate it. For example, in cases where there are few computations to perform, the overlap can be minimal. These systems also suffer from resource under-utilization due to the tight coupling between memory and compute. As shown in Figure 4, different graphs and workloads have vastly varying needs as described in section III-C. Tight coupling between memory and compute does not provide the flexibility to adjust the resource allocations to match these needs.

C. Disaggregated Processing Architecture

Solutions based on disaggregated architectures [9]–[11] for graph processing aim to solve the resource flexibility issue seen in the previous systems based on distributed designs. A disaggregated architecture, shown in Figure 1(a), consists of separate pools of compute and memory nodes, where multiple host servers can be connected to a shared remote memory pool. Remote memory is a second-tier resource compared to the local host memory in terms of access latency. Therefore, prior works on top of disaggregated systems focus on data tiering and reducing the overhead of data movement between different tiers.

Resource disaggregation between memory and compute allows independent resource provisioning and scaling to meet the unique needs of different workloads. We observe that compute and memory requirements vary according to the graph kernel and graph data; thereby, requiring the independent scaling of compute and memory resources. Figure 4 shows the compute and memory requirements for four graph kernels (PR, CC, SSSP, BFS) on two real-world graphs (uk-2005, twitter7). The points in the orange box show how different kernels can

TABLE II: Comparison between previous works vs. disaggregated NDP solution to process large-scale graph analytics. Communication overhead is directly related to the amount of data movement. Synchronization overhead is related to the time delay while waiting for the partial results to be fetched over the network for a final update.

System Architecture	Near-Memory Acceleration	Communication Overhead	Synchronization Overhead	Resource Utilization
Distributed [2]–[5]	✗	High	High	Skewed
Distributed NDP/PIM [6]	✓	High	High	Skewed
Disaggregated [9]–[11]	✗	High	Low	Balanced
Disaggregated NDP [this work]	✓	Low	Low	Balanced

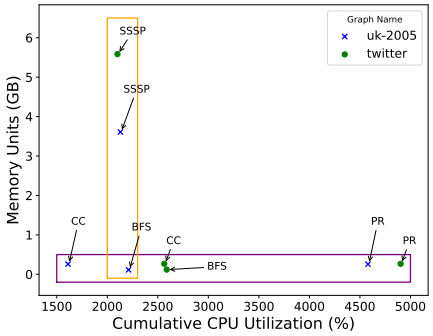


Fig. 4: Varying compute-memory resource requirements for different graph workloads. (i) Orange box shows how workloads can have similar compute needs but different memory needs. (ii) Purple box shows how workloads with similar memory needs can have different computing needs.

have similar computing needs while having different memory needs. The points in the purple box show how different kernels can have similar memory needs while having different computing needs. The results indicate that deploying graph workloads will necessitate an unbalanced usage between different resources. Since distributed systems with general servers have tightly coupled compute and memory, running large-scale graph analytics will lead to skewed resource utilization, wasting either compute or memory. Resource disaggregation provides a solution to the skewed resource utilization problem, which persists even for distributed systems with integrated NDP components, by decoupling the resource hierarchies of compute and memory. It enables systems to achieve balanced resource utilization with a lower total cost of ownership.

Another benefit of these approaches is lower synchronization overheads across the fewer compute nodes as they are not waiting for any processing to complete on the remote memory node(s) which only store data. During the deployment of the graph workload, synchronization is required when the vertex updates from the mirrors are applied to the masters during the compute phase. This happens on all the nodes in a distributed deployment, whereas it only happens on the compute nodes in a disaggregated deployment, resulting in lower synchronization overheads than the distributed deployments.

Existing solutions for graph processing over disaggregated systems [9]–[11] are based on the observation that vertex and edge data have different usage patterns: vertex data is accessed more frequently and the edge data is read-only. Therefore, they store edge data on a slower but larger second tier of memory while vertex data is primarily stored in fast host memory. During the traversal phase, the required subset of the edge data is fetched from disaggregated memory to be consecutively processed on the host. However, they still suffer from large communication overheads due to significant data movement, proportional to the amount of edge data that needs to be pulled from remote memory during every iteration. This is exacerbated for highly connected graphs, where all neighbor vertices and edges need to be fetched whenever an update has to be made. For example, FAM-Graph [9] shows that large graphs often have vertices with outgoing degrees in the order of hundreds of thousands of edges. Some works [10], [11] mitigate the data movement time by overlapping computation and communication, but the fundamental data movement costs are still not addressed.

D. THIS WORK: *Disaggregated NDP Architecture*

Table II summarizes the limitations of the previous system architectures to process large-scale graph analytics. We point out that none of the existing works that deploy graph workloads on distributed, distributed NDP, and disaggregated systems fully satisfy every benefit of each architecture. In addition, all systems have a high volume of data movement, where the communication time is becoming a major bottleneck of end-to-end performance. In this paper, we consider disaggregated architectures with NDP support. As shown in Table II, when used for graph analytics, such systems can provide for both low communication and synchronization overhead, as well as memory bandwidth expansion and balanced resource utilization.

Several features of the disaggregated NDP architecture shown in Figure 1(b) can reduce data movement overheads compared to the distributed and disaggregated architectures explained in the previous sections. Data transmitted from the memory nodes to the compute nodes is reduced because the number of per-vertex updates is generally smaller than the edge lists that are fetched from the memory nodes in the disaggregated architectures. The in-network compute units aggregate partial updates to further reduce the data-movement overheads that are otherwise incurred while transmitting updates from the mirrors to the masters in the distributed architectures. When deploying a computationally heavy graph

workload using the distributed NDP architecture, a graph may be partitioned among more nodes than necessary to account for the computing needs of the workload. The disaggregated architecture decouples the compute and memory to avoid resource over-provisioning. This allows the graph to be partitioned on the necessary number of nodes, further curtailing data movement costs.

At the same time, this design retains the benefits of acceleration via low-latency and high-bandwidth data access, data processing on NDP devices, and the disaggregation benefits of lower synchronization overheads and balanced resource utilization.

In summary, disaggregated NDP architectures combine the benefits of the existing approaches, while addressing their limitations, and present a promising solution for platforms for large-scale graph analytics.

IV. Missing Systems Functionality for Disaggregated NDP Architectures

Given the promise of disaggregated NDP architectures for graph analytics, it is important to understand the system-level functionality required to effectively deploy and execute graph workloads on such systems. In this section, using empirical evidence obtained from evaluating different graph application deployments over an emulated NDP system, we make observations regarding the missing systems support for disaggregated NDP graph frameworks. Concretely, we present evidence of the need for programming and runtime mechanisms and abstractions to control which graph operations to offload near-data and when to switch to near-data processing, and to allow for such decisions to be made dynamically, considering scale, workload, and platform characteristics.

Our analysis is based on a common model for graph analytics used in related work. Graphs are represented using the CSR format, which consists of two data structures – the list of vertices (and their properties) and the list of edges, as shown in Figure 1. The latter can be orders of magnitude larger than the former [9]. At scale, these structures need to be distributed across multiple machines. Disaggregation can help alleviate the overheads of distribution explained in Section III. Two mechanisms help reduce the overheads of distribution in the disaggregated deployment – NDP offload and in-network aggregation.

The preliminary results presented in this section are obtained from a prototype of the disaggregated model shown in Figure 1(b). The prototype is designed using the Galois [32] system and uses various data structures to keep track of the graph partitions, message buffers, and partial updates while simulating the distributed system. The prototype splits the traversal and update phases and calculates the amount of data moved between these phases during every iteration. Separate buffers are used to store the vertex updates generated during the traversal phase on each memory node. These buffers are used to calculate the amount of data moved to propagate the vertex updates to the compute nodes. The updates are aggregated and applied to the vertex properties during the compute phase, after which the new frontier is generated. The frontier size is used to calculate the amount of data moved to propagate the updated vertex properties to the memory nodes. The evaluation is performed on an Intel Skylake machine with 2 Intel Xeon Gold 6142 CPUs and 384 GB of DRAM, and using several graph datasets: the Twitter7 Graph (41M Nodes, 1.4B Edges), UK-2005 Graph (39M Nodes, 936M Edges), and com-LiveJournal Graph (3M Nodes, 69M Edges) [33].

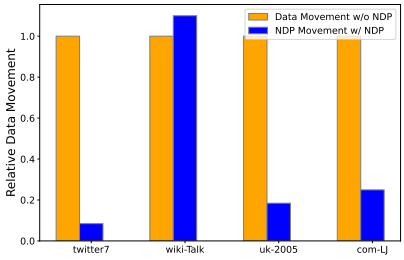


Fig. 5: Impact of offloading graph traversals on the data movement costs in disaggregated architectures with NDP for the PageRank workload.

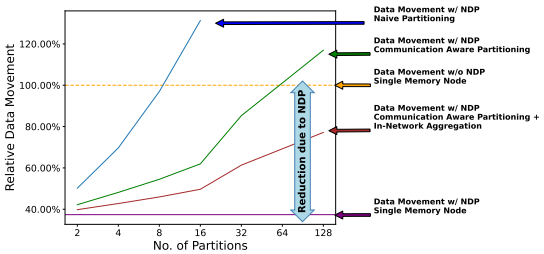
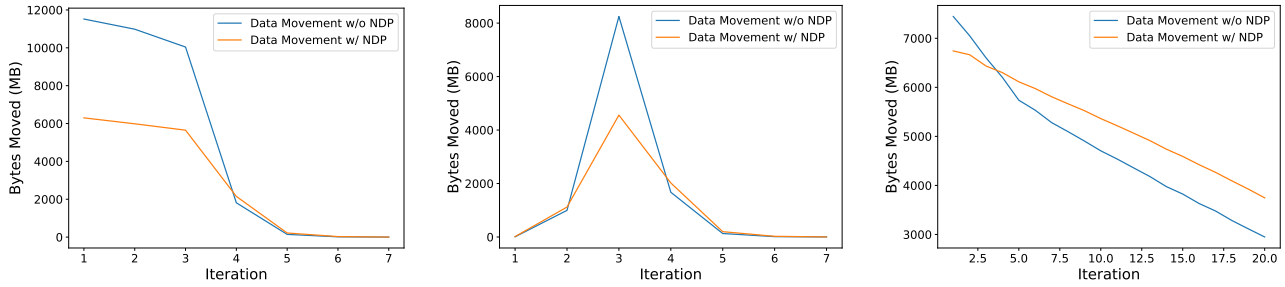


Fig. 6: Impact of partitioning and using in-network aggregation on data movement costs.
Graph: com-LiveJournal, Algorithm: PageRank

A. Determining the NDP Offload Capabilities

The traversal phase of the graph workload is memory-intensive and bandwidth-bound and the edge list is solely accessed during this phase. Thus, the edge list is partitioned across the nodes in the memory pool as shown in Figure 1. If passive memory pools without any processing capabilities were to be used to store the edge lists, the system would incur large data movement overheads. To process each vertex, the compute nodes would need to retrieve the neighbors. By offloading the traversal operations, the compute nodes have to retrieve one update per vertex mirror, incurring much lower data movement costs, as shown in Figure 5.

However, simply providing a programming API to specify the different types of operations (i.e., traverse vs. apply) is not sufficient, since deciding which operations to offload is not a trivial decision. For instance, Figure 5 shows that data movement costs can also increase when traversals are offloaded, as seen in the case of the *wiki-Talk* graph. This is because data-movement is dependent upon the workload and the graph



(a) Config - Graph: Twitter7 (32 Partitions), Algorithm: Connected Components

(b) Config - Graph: Twitter7 (32 Partitions), Algorithm: Single Source Shortest Path

(c) Config - Graph: UK-2005 (80 Partitions), Algorithm: Page Rank

Fig. 7: Comparison of data movement trends per iteration with and without NDP.

topology. The topology of the *wiki-Talk* graph results in a large number of intermediate updates which are 16 bytes in size for the PageRank workload. These factors make fetching the edge data, which requires 8 bytes per edge, the better alternative. In summary, the benefits of offloading operations are non-uniform and may depend on various factors such as the graph workload and the graph itself. The runtime deploying graph workloads on disaggregated NDP architectures must consider all these factors and this motivates the need to explicitly control different aspects of the offload mechanism.

B. Considering the Impact of Scale on NDP Offload

As the graphs increase in size, they need to be partitioned across multiple memory nodes. As the number of partitions increases, so does the volume of partial results generated by the offloaded computations. If the impact of distribution is not managed, the benefits of NDP will be overshadowed by the volume of the partial results, as shown in Figure 6. Carefully partitioning the graph to minimize the number of cross-edges can greatly reduce the number of partial results, but this is not a complete solution. The green plot in Figure 6 shows the data movement trend after the graph has been partitioned to minimize the number of partial updates using the METIS library [34]. Although the communication overheads are significantly reduced, the overheads of distribution still nullify the benefits of NDP offload at higher degrees of distribution. An effective graph runtime must incorporate mechanisms to assess and navigate this tradeoff.

C. Mechanism for In-Network Aggregation

The traversal operations executed on the sub-graphs (or graph partitions) on each memory node create partial updates that need to be aggregated in the compute phase of the graph workload. Near-data processing capabilities integrated in the network elements in cluster systems (e.g., like the switch shown in Figure 1(b)), can provide for in-transit/in-network aggregation capabilities that help reduce these overheads. The benefits of this are two-fold. First, data movement costs between the switch and the compute node are reduced by aggregating the updates. Second, the memory pressure on the compute node is reduced as it has to receive and buffer fewer updates. The aggregation operations are ideal for offloading as they require very basic compute units, usually required to perform simple min/max or arithmetic operations. The brown line in Figure 6 illustrates the potential benefit of in-network aggregation, which could reduce data movement by up to 0.65× when applied to aggregate all partial results in every iteration of the workload. This also restores the benefit of data movement reduction from NDP offload which was eliminated due to the scale of the graph distribution (as seen in the green and blue lines in the plot). The benefits of in-network aggregation are higher at higher degrees of partitioning because the volume of partial updates increases with the number of partitions. Note that the gains shown in this figure are hypothetical and there are other factors to consider such as the available buffer capacity of the switch. Regardless, they illustrate an opportunity that future graph frameworks should be able to exploit, in offering programming support to execute aggregations on in-network elements, and in providing the runtime mechanisms to understand the partitioning and the scale at which processing is performed to adequately configure where such operations should be deployed.

D. Need for Dynamic Decision Making

The decisions of which operations to offload, and where, are not static. Interestingly, they can vary even across iterations of the same graph application. Figure 7 shows the data movement trends per iteration during the execution of different graph workloads. The data movement reduction in the NDP offload case stems from transferring vertex updates vs. edge sets for each vertex in a frontier. Depending on the graph topology and size of vertex properties, this difference changes.

These figures show that offload is not always the better option. Graph frameworks can benefit from support to make these decisions dynamically, during every iteration, to allow the system to pick the best alternative and minimize data movement costs. Heuristics such as the frontier size, the number of cross-edges, and the degrees of the vertices in the frontier can be used to determine the better alternative in every iteration. These heuristics are a function of the graph partitioning scheme and the workload and can be used to reduce the data-movement costs.

V. Conclusion

The performance of large-scale graph analytics is dependent on fast access to large amounts of data, and is increasingly becoming limited by the performance of the underlying memory system and the associated data movement and synchronization overheads in distributed solutions. This paper considers the limitations of existing approaches for these workloads, and identifies an opportunity to address them via use of disaggregated systems with emerging computational memory and network devices. Such systems benefit from the *near-data processing* capabilities offered by these new types of devices, which allow for accelerated execution of certain classes of graph operations that process data "in-place" and drastically reduce communication overheads associated with data movement. At the same time, they also benefit from disaggregation to achieve reduced synchronization overheads and more balanced resource utilization. However, existing graph frameworks lack support to fully leverage the capabilities provided via these disaggregated systems with NDP. By analyzing the data movement associated with different deployment strategies of several graph analytics kernels and graphs, we identify the required functionality that needs to be provided by the programming and runtime support in future graph analytics frameworks.

References

[1] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, *et al.*, "The future is big graphs: a community view on graph processing systems," *Communications of the ACM*, vol. 64, no. 9, pp. 62–71, 2021.

[2] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, pp. 752–768, 2018.

[3] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A {Computation-Centric} distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 301–316, 2016.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "{PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pp. 17–30, 2012.

[5] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[6] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 712–725, 2019.

[7] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 544–557, IEEE, 2018.

[8] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[9] D. Zahka and A. Gavrilovska, "Fam-graph: Graph analytics on disaggregated memory," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 81–92, IEEE, 2022.

[10] W. Shaddix, M. Samani, M. Fariborz, S. Yoo, J. Lowe-Power, and V. Akella, "Tegra–scaling up terascale graph processing with disaggregated computing," *arXiv preprint arXiv:2404.03155*, 2024.

[11] J. Wang, C. Li, T. Wang, L. Zhang, P. Wang, J. Mei, and M. Guo, "Excavating the potential of graph workload on rdma-based far memory architecture," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1029–1039, IEEE, 2022.

[12] S. Lee, S. Ponnapalli, S. Singhal, M. K. Aguilera, K. Keeton, and V. Chidambaram, "Dinomo: An elastic, scalable, high-performance key-value store for disaggregated persistent memory," *Proc. VLDB Endow.*, vol. 15, p. 4023–4037, sep 2022.

[13] J. Sim, S. Ahn, T. Ahn, S. Lee, M. Rhee, J. Kim, K. Shin, D. Moon, E. Kim, and K. Park, "Computational cxl-memory solution for accelerating memory-intensive applications," *IEEE Computer Architecture Letters*, vol. 22, no. 1, pp. 5–8, 2023.

[14] S.-S. Park, K. Kim, J. So, J. Jung, J. Lee, K. Woo, N. Kim, Y. Lee, H. Kim, Y. Kwon, *et al.*, "An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 970–982, IEEE, 2024.

[15] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture," *arXiv preprint arXiv:2105.03814*, 2021.

[16] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling distributed machine learning with {In-Network} aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 785–808, 2021.

[17] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, *et al.*, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pp. 1–10, IEEE, 2016.

[18] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, (New York, NY, USA), p. 105–121, Association for Computing Machinery, 2023.

[19] W. Huangfu, K. T. Malladi, A. Chang, and Y. Xie, "Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 727–743, 2022.

[20] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi, and A. Fedorova, "A case study of Processing-in-Memory in off-the-Shelf systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 117–130, USENIX Association, 2021.

[21] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, (Boston, MA), pp. 585–600, USENIX Association, 2023.

[22] H. Ham, J. Hong, G. Park, Y. Shin, O. Woo, W. Yang, J. Bae, E. Park, H. Sung, E. Lim, and G. Kim, "Low-overhead general-purpose near-data processing in cxl memory expanders," 2024.

[23] J. Hermes, J. Minor, M. Wu, A. Patil, and E. V. Hensbergen, "Udon: A case for offloading to general purpose compute on cxl memory," 2024.

[24] F. Faghih, T. Ziegler, Z. István, and C. Binnig, "Smartnics in the cloud: The why, what and how of in-network processing for data-intensive applications," in *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, (New York, NY, USA), p. 556–560, Association for Computing Machinery, 2024.

[25] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Comput. Surv.*, vol. 54, may 2021.

[26] A. Agrawal and C. Kim, "Intel tofino2–a 12.9 tbps p4-programmable ethernet switch," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, pp. 1–32, IEEE Computer Society, 2020.

[27] "NVIDIA Quantum-X800 InfiniBand Platform — nvidia.com." https://www.nvidia.com/en-us/networking/products/infiniband/quantum-x800/.

[28] "Mellanox Spectrum-2 SN3700 32-Port 100GbE Open Ethernet Switch with Cumulus Linux - Part ID:

MSN3700-CS2RC - Colfax Direct — colfaxdirect.com." https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3766.
[29] S. Kianpisheh and T. Taleb, "A survey on in-network computing: Programmable data plane and technology specific applications," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 701–761, 2022.
[30] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *arXiv preprint arXiv:1904.07162*, 2019.
[31] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 105–117, 2015.
[32] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp. 456–471, 2013.
[33] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.
[34] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.