GRAPHVINE: Exploiting Multicast for Scalable Graph Analytics

Leul Belayneh
University of Michigan, Ann Arbor
leulb@umich.edu

Valeria Bertacco
University of Michigan, Ann Arbor
valeria@umich.edu

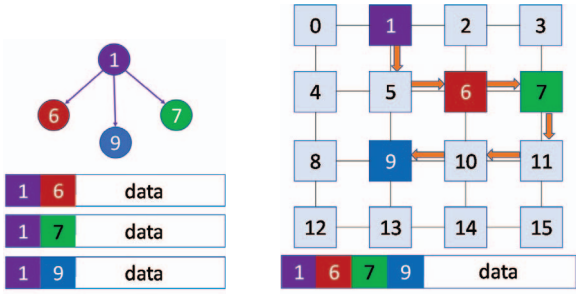
***Abstract*—The proliferation of graphs as a key data structure for big-data analytics has heightened the demand for efficient graph processing. To meet this demand, prior works have proposed processing in memory (PIM) solutions in 3D-stacked DRAMs, such as Hybrid Memory Cubes (HMCs). However, PIM-based architectures, despite considerable improvement over conventional architectures, continue to be hampered by the presence of high inter-cube communication traffic. In turn, this trait has limited the underlying processing elements from fully capitalizing on the memory bandwidth an HMC has to offer. In this paper, we show that it is possible to combine multiple messages emitted from a source node into a single multicast message, thus reducing the inter-cube communication without affecting the correctness of the execution. Hence, we propose to add `multicast` support at source and in-network routers to reduce vertex-update traffic. Our experimental evaluation shows that, by combining multiple messages emitted at the source, it is possible to achieve an average speedup of $2.4\times$ over a highly optimized PIM-based solution and reduce energy consumption by $3.4\times$, while incurring a modest power overhead of 6.8%.**

I. INTRODUCTION

With the growing prevalence of big data processing, graphs are a key data structure for data management and analysis, owing to their ability to represent relationships among different entities efficiently. Graphs are deployed in mapping a wide range of data, including social networks, road networks, citation networks, genome sequences, user ratings, etc.

The proliferation of graph-based applications has generated the need for a robust and efficient processing mechanism. However, sparsity in graph data structures, which translates to irregular memory accesses, continues to hamper the efficiency of graph processing. Indeed, this trait hinders cache efficiency, resulting in high memory access latency. Hence, numerous graph analytics works have focused on optimizing the memory subsystem [1], [2], [3], [4], [5]. Some of these works [1], [3] utilize on-chip scratchpads to accomodate irregular memory accesses. Others offload computation to the scratchpad unit [3]. In both cases, the memory bandwidth bottleneck, ultimately constrained by pin count per chip, limits the positive impact of these works [6].

With the advent of 3D-stacked DRAMs, such as HMCs, recent explorations utilize processing-in-memory architectures for graph analytics [6], [4], [7]. Tesseract [6] proposes a message-passing alternative, so as to eliminate potential cache coherency and synchronization overheads. However, despite considerable performance improvement, inter-cube communication, which entails communication between paritioned



(a) Messages generated for the sample graph

(b) path-based multicast packet flow in network

Figure 1: Example traversal of a multicast packet in a 4×4 2D mesh network with 16 HMCs. Source vertex 1 (mapped on cube 1) sends vertex-update message to vertex 6, 7, and 9 mapped to cube 6, 7, and 9, respectively.

vertices, has become the limiting factor. GraphP [7] applies a unique programming model to minimize the inter-cube communication bottleneck. The work uses a replication-based partitioning mechanism, which entails significant storage overhead and sacrifices programmability. MessageFusion [8] utilizes in-network processing to perform on-path computation; however, it is limited by the available on-chip buffer in the network.

In this work, we propose GRAPHVINE, which aims to address the inter-cube communication bottleneck; thus, unlocking significant performance improvement, while keeping programmability intact. GRAPHVINE can be applied to other PIM-based architectures as well, such as High Bandwidth Memories (HBMs). The key observation behind this proposition is that, in graph-based algorithms, multiple messages with the same payload are issued from a source vertex to its remotely located neighboring vertices. Most importantly, these vertex-update messages are released in close sequence, making them amenable to multicasting. Prior research has explored multicasting in the context of other applications, such as cache coherence and barrier synchronization [9], [10]. The path-based multicast technique, which we adopt here, has been identified for offering the best overall throughput. In a path-based technique, multicast packets are routed through all destinations, dropping-off messages at each destination. Figure 1 shows a path-based network traversal for a multicast packet generated from vertex 1 of a sample graph.

To maximize the multicast opportunity, we allow packets to

978-3-9819263-4-7/DATE20/©2020 EDAA

make turns in any direction while dropping-off messages and employ a deadlock recovery mechanism as in [11]. To enable easy drop-off of a message from a multicast packet, routers are augmented with a *Multicast Route Computation Unit* (*MRCU*) and a *comparator*. Note that these additional units operate in parallel with existing router pipeline stages, so no additional latency is incurred.

Contributions. We make the following contributions:

- We demonstrate the benefit of multicasting in graph processing and architect a multicast technique to minimize vertex-update messages flowing through PIM-based architecture.
- We develop GRAPHVINE, a PIM-based domain-specific architecture, supporting multicast for efficient graph analytics.
- We evaluate the performance and energy gains of GRAPHVINE and compare it with current state-of-the-art PIM-based solutions. Overall, GRAPHVINE improves the performance of a highly-optimized baseline by $2.4\times$, while achieving $3.4\times$ energy reduction, with a minimal hardware overhead of 10.2%. GRAPHVINE also provides a $1.4\times$ performance and energy savings as compared to Message-Fusion [8], a PIM-based graph acceleration solution.

To the best of our knowledge, this is the first work to show the benefit of multicasting in graph analytics.

II. BACKGROUND AND MOTIVATION

This section provides background on graph processing, PIM-based graph analytics, and multicast techniques.

A. Graph-based processing

Graphs provide a simple representation of relationships among a large number of entities. A Graph G = (V, E) comprises a set of vertices (V) interconnected through edges (E). For instance, in the PageRank algorithm, commonly used by search engines in response to a search query, vertices represent webpages, while edges correspond to hyperlinks.

Listing 1 illustrates the pseudocode for the PageRank algorithm. It comprises three phases: first an edge-processing phase, in which a source vertex fetches its rank and neighbor information from memory; second a reduce phase, in which source vertices compute their rank and send updates to neighboring vertices; and third an apply phase, in which the newly computed ranks from the current iteration are copied over to prepare for the next iteration. Note that the edge-processing phase is characterized by fairly regular memory accesses, while the reduce phase exhibits random memory accesses. Hence, prior works [1], [8] have recommended that vertex information (*vertexInfo*) be stored in a scratchpad memory, while neighbor data (*edgeInfo*) be streamlined from memory through a prefetcher.

Frequent random memory accesses in graph algorithms lead to inefficient cache usage and incur high memory access latency [1]. To circumvent this problem, some works recommend the use of scratchpad units [3], [1], while others leverage newly proposed memory systems such as HMC for higher and scalable memory bandwidth [6], [4].

```
1 // graph initialization
2
3 for(v in vertices) // edge processing phase
4   value = 0.85 * v.pagerank/v.out_degree
5   for(outgoing edge e(v, w)) // reduce phase
6      w.next_pagerank += value
7    end
8 end
9 for(vertex v in vertices) // apply phase
10   v.pagerank = v.next_pagerank
11 end
```

Listing 1: Pseudocode of PageRank highlighting the edge-processing, reduce, and apply phases.

B. Processing in memory (PIM)-based accelerator

The introduction of 3D stacked memories has enabled both PIM solutions and high density storage. For instance, in HMCs, up to 8 DRAM layers are interconnected through vertical electrical connections, known as through-silicon-vias (TSVs). TSVs offer lower memory access latency to the logic layer beneath 3D stacked memory. A single HMC is divided into 32 partitions, also known as vaults. The HMC enables the inclusion of a compute unit in the logic layer, accessing the DRAM layer on top of it through a vault controller, and providing a total of 512GB/s of internal bandwidth. Hence, numerous works [6], [4], [7], [8] have adopted HMC for graph analytics. However, the presence of high inter-cube communication has limited their performance. To reduce the inter-cube communication bottleneck, our work, GRAPHVINE, leverages multicasting for messages moving from a source vertex to neighboring vertices during the computation.

C. Multicast techniques

Multicasting merges two or more packets originating from the same source and containing identical payloads. Multicasting can reduce network congestion by reducing the number of packets in flight in the network. There are two common multicast techniques: tree-based and path-based.

In tree-based multicasting [12], packets split at designated branch points (intermediate routers) before reaching their final destination. Thus, this technique is best suited for latency-sensitive applications, since multicast packets will always take the optimal route from source to destinations.

In contrast, path-based multicasting [9] implements a sequential message drop-off from a multicast packet. Hence, ordering the destinations so as to minimize path length has a positive impact on performance. The technique best fits throughput oriented applications, including graph-based applications, which process significant amounts of data concurrently.

III. THE GRAPHVINE ARCHITECTURE

GRAPHVINE augments a highly-optimized PIM-based architecture similar to Tesseract [6]. Figure 2 shows the high-level architecture of GRAPHVINE. It comprises a set of customized processing elements, a next-line prefetcher, and a scratchpad unit, situated beneath each vault. Each processing element includes *edge processor*, *reduce*, and *apply* units,

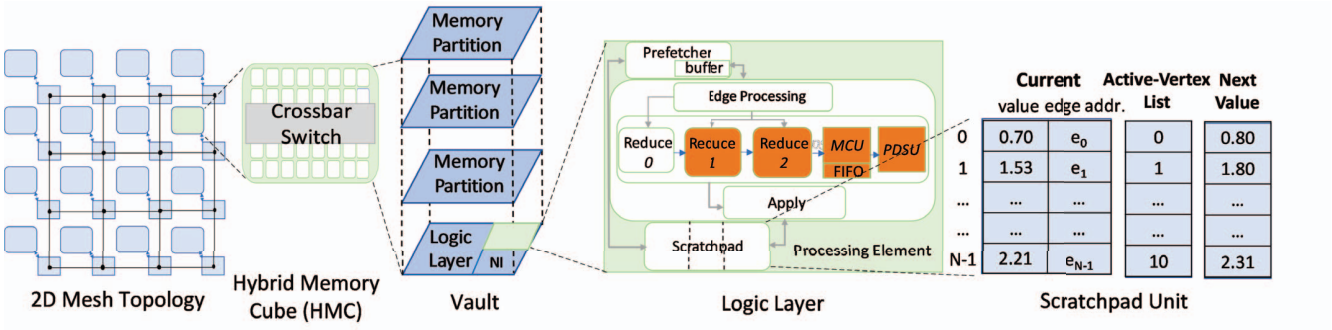


Figure 2: The GRAPHVINE architecture augments a PIM-based baseline with additional *Reduce* units, a *Message Combining Unit* (*MCU*) with an associated FIFO and a *Packet Destinations Sorter Unit* (*PDSU*) (shown in orange).

suited to execute specific phases in vertex-centric applications like the one in Listing 1. The scratchpad unit stores data related to vertices, so as to alleviate the cost of random accesses to memory. As shown in Figure 2, the scratchpad unit comprises a *current* scratchpad, a *next* scratchpad, and an *active-vertex* scratchpad. The *current* and *next* scratchpads store values associated with vertices for the current and next iteration. At the end of each iteration, *next* is copied to *current*. For instance, in the PageRank algorithm shown in Listing 1, the *pagerank* of a vertex is read from the *current* scratchpad, while the *next_pagerank* value is immediately stored in the *next* scratchpad. With reference to Figure 2, vertex 0 has a *pagerank* of 0.70 and its *next_pagerank* has been updated to 0.80 during the same iteration. Hence, the *current* scratchpad is accessed when reading *vertexInfo* of a given vertex, alongside the initial and final memory location of its *edgeInfo*. On the other hand, the *next* scratchpad is accessed to store updates to a vertex. Finally, the *active-vertex* scratchpad is employed only by active list-based algorithms to identify active vertices. In cases where a dataset exceeds a scratchpad's capacity, a slicing technique can be applied to segment the graph [1]. Each segment, which fits within a single scratchpad, is processed individually, ensuring optimized utilization.

To support multicast, GRAPHVINE augments both processing elements and in-network routers, as illustrated in Figure 2 and 3. The *Message Combining Unit* (*MCU*) combines unicast messages emerging from a processing element and generates a multicast packet. Once the multicast packet is injected into network, the router, with the support of the *Multicast Route Computation Unit* (*MRCU*), provides support in distributing it to all its destinations. Finally, upon reaching its destination, a multicast packet may update multiple entries of a scratchpad. Thus, a multi-ported scratchpad unit is used to parallelize these updates.

A. Message Combining Unit (MCU)

The *MCU* aggregates identical outgoing unicast messages from the *reduce* unit to generate a multicast packet prior to injection into the interconnect. The *MCU* utilizes a small-sized first-in-first-out (FIFO) buffer to accumulate the outgoing messages. A portion of these messages, which are emanating from the same source, will be combined into a multicast

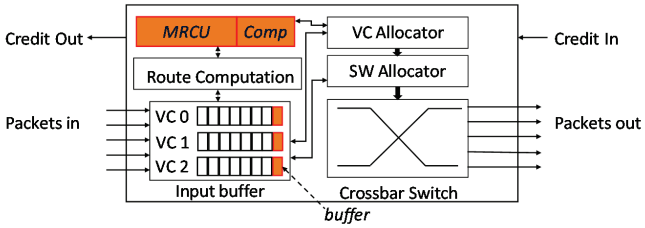


Figure 3: GRAPHVINE Router microarchitecture, including the *Multicast Route Computation Unit* (*MRCU*) and *comparator* (shown in orange).

packet. To increase opportunities, the FIFO buffer size should be at least equal to the maximum number of destinations a multicast packet can contain. In our implementation, multicast packets have one additional flit for each group of three destinations. For instance, our evaluation considers multicast packets up to maximum of 13 destinations, which corresponds to four additional flits and a 13-entry FIFO buffer.

B. Packet Destinations Sorter Unit (PDSU)

As described in Section II, our *Packet Destinations Sorter Unit* (*PDSU*) sorts packet destinations before injection into the network. The *PDSU* implements the low-distance algorithm [9]: the packet is first routed to the destination closest to the source node. Once messages for that destination are dropped, each subsequent destination is selected as the closest destination to the most recent drop-off. Note that this routing algorithm could potentially lead to a deadlock. To prevent that, we implemented an escape virtual channel to enable deadlock recovery, as recommended in [11]. Thus our multicast packets take minimal-length routes, even if they entail turns that would be forbidden in deadlock-free routing. In designing the *PDSU* we leveraged pipelining to hide its computation latency from impacting overall performance.

C. Multicast Route Computation Unit (MRCU)

To support the distribution of the multicast packets generated at the processing elements, we employ specialized hardware support at each router through *MRCU*s. *MRCU*s enable the drop-off of messages from a multicast packet at intermediate destinations. As the multicast packet approaches each of its destinations (specifically, one hop before), the

MRCU computes and updates its look-ahead route information based on the following destination. To enable this specialized packet management, we introduced light-weight hardware blocks. First, each virtual channel on the router is augmented with an entry *buffer*, which assists with message drop-off by storing the multicast payload. Note that a packet may contain multiple destinations for a same router. In such case, the *MRCU* must correctly traverse the destination list until it reaches one for the next node to be reached. GRAPHVINE uses a *comparator*, colocated with the *MRCU*, to achieve this goal. Note that both the *MRCU* and the *comparator* operate concurrently with the baseline router pipeline; thus, packets do not incur additional delay in transfer.

D. Multi-ported Next Scratchpad

Since a multicast packet can contain multiple destination vertices residing within a same vault, utilizing multiple *reduce* units enables concurrent processing. GRAPHVINE uses three reduce units per vault to simultaneously process flits destined to multiple vertices, hence avoiding serialization of vertex updates. To accomodate multiple simultaneous updates to the *next* data structure, we utilized a multi-ported scratchpad unit. It is possible to equip the system with more or fewer concurrent reduce units and corresponding read/write ports for the scratchpad, depending on the desired performance-power tradeoff. An analysis in this design space is deferred for future work. Finally, for further energy savings, GRAPHVINE adopts workload-aware power gating of scratchpad units [8].

IV. EXPERIMENTAL EVALUATION

In this work, GRAPHVINE is compared against two solutions: Tesseract [6], a baseline PIM-based architecture and MessageFusion [8], a state-of-the-art PIM-based solution for graph processing.

A. Experimental Setup

To evaluate GRAPHVINE, we set up a custom cycle-accurate simulator. We use BookSim [13] to model a 4×4 2D mesh and a dragonfly interconnect. The setup includes a dimension-ordered and universal globally-adaptive load-balanced routing algorithm for 2D mesh and dragonfly, respectively. It uses a link width of 128 bits, and four-stage pipelined routers as shown in Figure 3. Each router has 3 VCs with 8 flit buffers per VC. Each node is an HMC cube, modeled with CasHMC [14], a cycle accurate HMC simulator. Each HMC cube is equipped with an 8GB of storage and contains 32 vaults. Each vault provides a bandwidth of 16GB/s. Underneath each vault, there are three scratchpads of size 128KB, 64KB, and 32KB for *current*, *next*, and *active-vertex* data, respectively. Finally, prefetch buffers are 1KB.

To model GRAPHVINE, we augmented the logic layer with a *PDSU* module for sorting the set of destinations of multicast packets. Based on our synthesis analysis, sorting more than 7 destinations in the *PDSU* requires more than two pipeline stages to hide latency, thus entailing a noticeable energy impact. We also added a FIFO buffer with as many entries as the number of destinations in a multicast packet. The *next* scratchpad unit is equipped with three ports to handle vertex-updates to multiple entries. We considered several design options for GRAPHVINE, including a range of the maximum number of destinations that can be contained in a single multicast packet: 4, 7, 10, or 13. To support multicasting, each router is augmented with a *MRCU* and a *comparator*. In addition, a one entry *buffer* of 8 bytes is added to each virtual channel.

Graphs	#Vertices	#Edges	Average Degree	Description
wiki-Vote (WV)	7.1K	103.7K	14.6	Votings in Wikipedia
Slashdot0811 (SD)	82.2K	1M	11.5	Friendships in Slashdot
Amazon0601 (AZ)	0.4M	3.4M	8.4	Amazon Products
Pokec (PK)	1.6M	30.6M	18.8	Friendships in Pokec
soc-LiveJournal1 (LJ)	4.8M	69.0M	14.2	Followers in LiveJournal
com-Orkut (OK)	3.1M	117.2M	38.1	Friendships in Orkut
roadNet-PA (rPA)	1M	1.5M	0.8	Roads in Pennsylvania
roadNet-TX (rTX)	1.4M	1.9M	1.3	Roads in Texas
roadNet-CA (rCA)	2M	2.8M	1.4	Roads in California
synthetic	5M	10-190M	2-38	Synthetic Datasets

TABLE I: Graph Datasets

B. Workloads

We evaluated GRAPHVINE on four algorithms using representative real world and synthetic datasets with a range of graph sizes and degrees. Average Teenager Follower (AT) calculates the number of teenager followers for each user (vertex). PageRank (PR) computes the importance of vertices in a graph based on their incoming degree. Breadth-First Search (BFS) traverses a graph from a source vertex across breadth. Connected Components (CC) finds the total number of subgraphs in a graph. The datasets considered are from SNAP [15] and are summarized in Table I. We generated the last dataset synthetically.

C. Overall Performance

We report the performance and energy savings of GRAPHVINE on a 4×4 2D mesh in the top plot of Figure 4, and compare it against MessageFusion [8], a recent solution that strives to boost the performance of graph-based applications on HMCs. The evaluation is carried out with a 7-cast solution since, as we will observe later, it is an optimal multicast width. As shown in the plot, the average performance of GRAPHVINE over Tesseract is $2.76\times$. This improvement comes from reducing inter-cube communication as well as concurrent processing of vertex-updates messages destined to a same node. In analyzing the performance of GRAPHVINE in a 4×4 dragonfly, we found that it offers an average speedup of $2.01\times$ over Tesseract. We wish to note how GRAPHVINE offers a significant performance improvement of $2.2\times$ even on road network datasets, which present a very low average degree. By comparison, MessageFusion can provide only a $1.8\times$ improvement on the same datasets. A notable improvement of GRAPHVINE over MessageFusion is also observed on active-list based algorithms (BFS and CC): since only a small portion of the vertices are active (exchanging messages) during each iteration of those algorithms, MessageFusion suffers from lack of coalescing opportunity. However, GRAPHVINE can

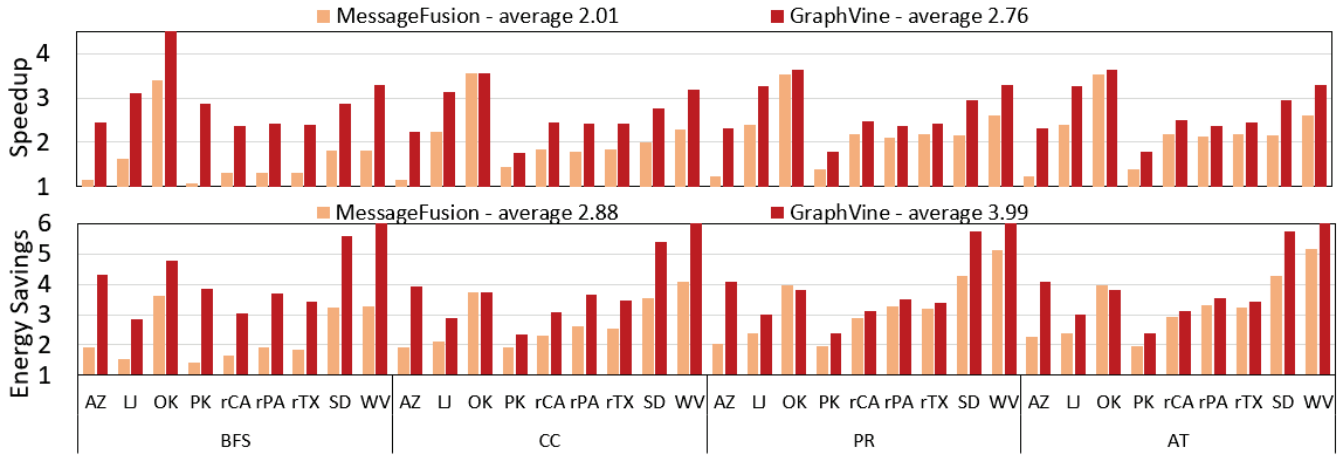


Figure 4: Performance and energy improvements of GRAPHVINE normalized to Tesseract on a 4×4 2D mesh.

still take advantage of messages emanating from those active vertices efficiently.

An evaluation of GRAPHVINE in the absence of a *Multi-ported Next Scratchpad* (see Section III) is shown in Figure 5, over a range of multicast widths. We observe that, the average performance of GRAPHVINE peaks at a width of 7 or 10. In general, the more the destinations in a multicast message, the less the traffic generated. However, beyond 10 destinations per message, the opportunity to create such deep multicast messages is reduced, and no longer makes up for the overhead incurred at the source. Note also that multicasting at a width of 7 entails significantly lower energy than a width of 10, thus, 7 is preferable as performance is equivalent. If we zoom in our analysis on low-degree datasets (road networks), we note that an optimal multicast width is 4, rather than 7 or 10, due the lower degree associated with most vertices, which offer fewer opportunities for multicasting.

D. Area, Energy/Power and Thermal Analysis

Our power and energy evaluation leverages workload execution time, and router and memory activities collected from the simulator. We used ORION3.0 [16] and Cacti [17] for modelling the interconnect and the scratchpad unit. We used [18], [19] to estimate the memory and logic layer energy consumption. Finally, we synthesized the GRAPHVINE units in processing elements and routers, along with the baseline processing element, by using IBM45nm SOI technology at 1GHz. The bottom plot of Figure 4 shows the normalized energy improvement of GRAPHVINE with respect to Tesseract. We note that GRAPHVINE achieves an average energy saving of $3.99\times$ ($1.38\times$) and $3.14\times$ ($1.53\times$) over Tesseract (MessageFusion) on a 4×4 2D mesh and dragonfly, respectively.

Overall, GRAPHVINE has a power overhead of 3.7% (6.8%) with its additional modules incurring a hardware overhead of 7% (10.2%) as compared to MessageFusion (Tesseract). The multi-ported scratchpad unit takes the lion's share in the overheads. Our thermal analysis shows that GRAPHVINE has a power density of $129.8\ mW/mm^2$, which lies below the thermal constraint of HMC [6].

E. Scalability and Sensitivity

We evaluated GRAPHVINE's scalability through performance comparison on a system four times larger, an 8×8 2D mesh network (2048 processing elements), and observed a comparable performance improvement of $2.38\times$ over Tesseract, indicating that our solution can sustain its performance boost across a range of underlying network sizes. We also analyzed the benefits of GRAPHVINE across a wide range of average degrees on synthetic datasets. As shown in Figure 6, GRAPHVINE's performance improves with increasing dataset's degree, as they provide more opportunity to generate wide multicast messages. More notably, a bigger gap between GRAPHVINE and MessageFusion is observed on the BFS algorithm. Despite the fact that BFS has relatively fewer vertex-update messages exchanged between active vertices, which in turn limits MessageFusion's performance, GRAPHVINE's multicasting maintains its strong improvement. Overall, GRAPHVINE has better scalability than MessageFusion in all cases except for some of the highest-degree datasets.

V. RELATED WORK

A. Graph analytics architectures

The recent rise of 3D stacked DRAMs, such as HMCs, has increased the memory bandwidth available to compute units. For example, GraphPIM[4] utilizes HMC to handle expensive atomic operation in graph analytics. Tesseract[6] uses a message passing model to avoid the inefficient and complex cache coherency and synchronization overheads in conventional architectures. GraphP [7] improves upon Tesseract[6] at the expense of programmability. Recently, MessageFusion[8] proposed messsage coalescing to minimize inter-cube communication. On the otherhand, GRAPHVINE improves on these works by using multicasting to reduce inter-cube communication bottlenecks. To the best of our knowledge, this is the first work to utilize multicasting for graph analytics.

B. Multicasting

The earliest works to employ multicast were developed in the off-chip networks domain [20], [21], [10]. Other studies

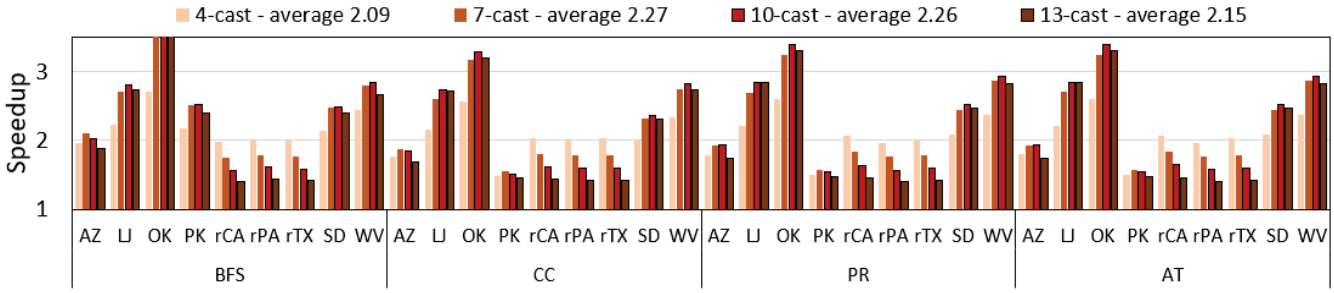


Figure 5: Performance improvement of GRAPHVINE with no *Multi-ported Next Scratchpad* normalized to Tesseract on a 4×4 2D mesh.

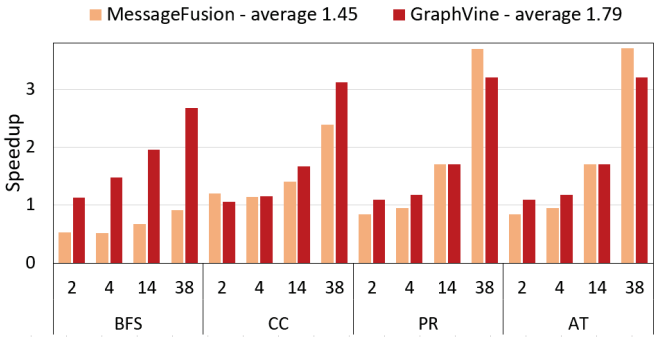


Figure 6: Performance improvement over synthetic datasets with a range of average degrees, normalized to Tesseract.

later adopted and developed various multicast techniques for on-chip networks as well [12], [22], [23]. Jerger, *et al.*, propose the need for multicasting and present Virtual Circuit Tree Multicast (VCTM), in which a lookup table is used to construct a multicast tree [12]. However, inefficient buffer utilization due to replicated flits and increased routing complexity makes the tree-based multicast unattractive. Lankes, *et al.* extend the hamilton path-based multicast routing algorithm and propose a low-distance algorithm, which has been adopted in this work [9]. Other works consider, multicasting for specific routing algorithms [23], 3D Networks-on-Chip [24], or irregular topologies [22]. However, since we use regular 2D mesh and dragonfly topologies, these techniques are not beneficial for our purposes.

VI. CONCLUSION

PIM-based solutions are shown to be a promising approach to achieve high efficiency in graph analytics, since they tackle the domain's primary bottleneck, i.e., memory bandwidth. However, high inter-cube communication limits the PIM's processing elements from fully utilizing the abundant memory bandwidth. In this work, we propose GRAPHVINE, which equips both logic layers and in-network routers with `multicast` support to combat the inter-cube communication bottleneck. We evaluated the performance of GRAPHVINE, on representative workloads and achieved an average speedup of $2.4\times$ over a baseline PIM solution at a modest power overhead of 6.8%, and a $1.4\times$ speedup against a graph-application optimized PIM architecture.

Acknowledgements. This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] T. Ham *et al.* Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. MICRO*, 2016.
[2] M. Ozdal *et al.* Energy efficient architecture for graph analytics accelerators. In *Proc. ISCA*, 2016.
[3] A. Addisie *et al.* Heterogeneous memory subsystem for natural graph analytics. In *Proc. IISWC*, 2018.
[4] L. Nai *et al.* Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *Proc. HPCA*, 2017.
[5] A. Basak *et al.* Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proc. HPCA*, 2019.
[6] J. Ahn *et al.* A scalable processing-in-memory accelerator for parallel graph processing. In *Proc. ISCA*, 2015.
[7] M. Zhang *et al.* GraphP: Reducing communication for pim-based graph processing with efficient data partition. In *Proc. HPCA*, 2018.
[8] L. Belayneh *et al.* Messagefusion: On-path message coalescing for energy efficient and scalable graph analytics. In *Proc. ISLPED*, 2019.
[9] M. Daneshtalab *et al.* Low-distance path-based multicast routing algorithm for network-on-chips. In *IET Computers Digital Techniques*, 2009.
[10] M. Malumbres *et al.* An efficient implementation of tree-based multicast routing for distributed shared-memory multiprocessors. In *Proc. IPDPS*, 1996.
[11] A. Lankes *et al.* Comparison of deadlock recovery and avoidance mechanisms to approach message dependent deadlocks in on-chip networks. In *Proc. NOCS*, 2010.
[12] N. Jerger *et al.* Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Multicast Support. In *Proc. ISCA*, 2008.
[13] N. Jiang *et al.* A detailed and flexible cycle-accurate network-on-chip simulator. In *Proc. ISPASS*, 2013.
[14] D. Jeon *et al.* CasHMC: A cycle-accurate simulator for hybrid memory cube. *CAL*, 2017.
[15] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[16] A. Kahng *et al.* ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proc. DATE*, 2009.
[17] P. Shivakumar *et al.* Cacti 3.0: An integrated cache timing, power, and area model. 2001.
[18] S. Pugsley *et al.* NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. In *Proc. ISPASS*, 2014.
[19] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *Symposium on VLSI Technology Digest of Technical Papers*, 2012.
[20] X. Lin and L. Ni. Deadlock-free multicast wormhole routing in multicomputer networks. In *Proc. ISCA*, 1991.
[21] M. de Azevedo and D. Blough. Fault-tolerant clock synchronization of large multicomputers via multistep interactive convergence. In *Proc. ICDCS*, 1996.
[22] S. Rodrigo *et al.* Efficient unicast and multicast support for cmps. In *Proc. MICRO*, 2008.
[23] L. Wang *et al.* Recursive partitioning multicast: A bandwidth-efficient routing for networks-on-chip. In *Proc. NOCS*, 2009.
[24] M. Ebrahimi *et al.* Path-based partitioning methods for 3d networks-on-chip with minimal adaptive routing. In *IEEE Transactions on Computers*, 2014.