GraphQ: Scalable PIM-Based Graph Processing

Youwei Zhuo
Chao Wang
youweizh@usc.edu
wang484@usc.edu
University of Southern California

Mingxing Zhang
james0zan@gmail.com
Tsinghua University

Rui Wang
wangrui@buaa.edu.cn
Beihang University

Dimin Niu
dimin.niu@gmail.com
Alibaba Inc

Yanzhi Wang
yanz.wang@northeastern.edu
Northeastern University

Xuehai Qian
xuehai.qian@usc.edu
University of Southern California

ABSTRACT

Processing-In-Memory (PIM) architectures based on recent technology advances (e.g., Hybrid Memory Cube) demonstrate great potential for graph processing. However, existing solutions did not address the key challenge of graph processing—*irregular* data movements.

This paper proposes GraphQ, an improved PIM-based graph processing architecture over recent architecture Tesseract, that fundamentally *eliminates irregular data movements*. GraphQ is inspired by ideas from distributed graph processing and irregular applications to enable *static and structured* communication with *runtime and architecture co-design*. Specifically, GraphQ realizes: 1) batched and overlapped inter-cube communication by reordering vertex processing order; 2) streamlined inter-cube communication by using heterogeneous cores for different access types. Moreover, to tackle the discrepancy between inter-cube and inter-node bandwidth, we propose a hybrid execution model that performs additional local computation during the inter-node communication. This model is general enough and applicable to *asynchronous iterative* algorithms that can tolerate bounded stale values. Putting all together, GraphQ simultaneously maximizes intra-cube, inter-cube, and inter-node communication throughput. In a zSim-based simulator with five real-world graphs and four algorithms, GraphQ achieves on average 3.3× and maximum 13.9× speedup, 81% energy saving compared with Tesseract. We show that increasing memory size in PIM also proportionally increases compute capability: a 4-node GraphQ achieves 98.34× speedup compared with a single node with the same memory size and conventional memory hierarchy.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; • **Hardware → 3D integrated circuits**; **Emerging architectures**.

MICRO '52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358256

KEYWORDS

graph analytics, data movement, memory systems, 3D-stacked memory, processing-in-memory, near-data processing

ACM Reference Format:
Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *MICRO '52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture, October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358256

1 INTRODUCTION

Graphs capture relationships between data items, such as interactions or dependencies. Graph analytics have emerged as an important way to understand the relationships between heterogeneous types of data, allowing data analysts to draw valuable insights from patterns in the data for a wide range of applications, including machine learning tasks [69], natural language processing [4, 20, 70], anomaly detection [52, 62, 68], clustering [55, 58], recommendation [17, 21, 38, 44], social influence analysis [9, 63, 67], bioinformatics [3, 16, 29].

To improve programmability, graph-oriented programming model, e.g., vertex program [26, 40, 43] can easily express graph algorithms by allowing programmers to "think as the vertex". Programmers can express algorithms in vertex and edge function based on neighbor vertices. A graph can be represented as an adjacency matrix, where each element represents an edge between a source and a destination. Typically, the adjacency matrix is *sparse* and stored in compressed representation. One example is Compressed Sparse Row (CSR), which has three arrays: *1) vertex array* stores vertices sequentially with each entry pointing to the start of the vertex's outgoing edge list; *2) edge array* stores the edges of each vertex sequentially; and *3) compute array* keeps the updates of the destination vertex when each edge is processed. The accesses to vertex and edge array are mostly sequential, but accesses to compute array are random. Moreover, graph algorithms require high memory bandwidth since they perform a small amount of computation on randomly accessed data.

In essence, the two problems are both irregular data movements in conventional memory hierarchy. Processing-In-Memory (PIM) can reduce data movement between memory and computation by placing computing logic inside memory dies. Though once believed to be impractical, PIM recently becomes an attractive architecture due to the emerging 3D stacked memory technology, such as

Hybrid Memory Cube (HMC) [12] and High Bandwidth Memory (HBM) [31]. In general, the architecture is composed of multiple memory cubes connected by external links (e.g., SerDes links in HMC with 120GB/s per link). Within each cube, multiple DRAM dies are stacked with Through Silicon Via (TSV) and provide higher internal memory bandwidth up to 320GB/s. At the bottom of the dies, computation logic (e.g., simple cores) can be embedded. Performing computation at in-memory compute logic can reduce data movements in memory hierarchy. More importantly, PIM provides "memory-capacity-proportional" bandwidth and scalability.

Tesseract [1] is a PIM-based graph processing architecture that supports vertex programming model with architectural primitives to enable inter-cube communication. GraphP [72] is another architecture that co-designs the programming model and architecture to reduce inter-cube communication. The two schemes only reduce data movement but did not change its irregular nature, which has two implications.

The first problem is the high overhead in handling small messages in inter-cube communication with unpredictable arrival time. This communication is determined by graph partition, i.e., the graph data is partitioned into memory cubes. For an edge, if source and destination vertices are assigned to different cubes, inter-cube communication is required to update the corresponding remote compute array entry. Such communication is unpredictable since it depends on the processing order of destination vertices. In Tesseract, remote cubes handles inter-cube message with interrupt and executes a function to perform the update. Due to the unpredictable message arrival time, it incurs performance overhead since the receiver cube's local execution is interrupted. In addition, it will lead to load imbalance since some cube can receive more messages. It is ideal if each cube knows *when* the messages will arrive from *which* source.

Second, processing cores in a cube perform both sequential accesses to vertex and edge array and random accesses to compute array. This access pattern destructs cache locality with interference. Moreover, locality is further affected by performing remote compute array update requests during inter-cube communication.

Third, all prior architectures are based on a single PIM node. To support multi-node, while vertex programming model requires no change, the runtime system can be transparently extended to map remote destination vertex updates to either inter-cube or inter-node communication, the key challenge is the substantial lower bandwidth of inter-node communication (6GB/s) compared to inter-cube (120GB/s) and intra-cube (360GB/s) communication.

To address the challenges, this paper proposes *GraphQ*, an improved PIM-based graph processing architecture over the recent architecture Tesseract that *eliminates irregular data movements*. GraphQ is inspired by ideas from distributed graph processing and irregular applications to enable *static and structured* communication with *runtime and architecture co-design*. Specifically, GraphQ realizes: 1) batched and overlapped inter-cube communication by reordering vertex processing order; 2) streamlined inter-cube communication by using heterogeneous cores for different access types to eliminate the interference. Moreover, GraphQ is the first PIM-based graph processing architecture that supports *multi-node*. To tackle the discrepancy between inter-cube and inter-node communication bandwidth, we propose a hybrid execution model that is a midpoint between synchronous and asynchronous execution, performing additional local iterations during the long inter-node communication. This model is general enough and can be applied to asynchronous iterative algorithms tolerate bounded stale values [66]. Putting all together, GraphQ *simultaneously maximizes* intra-cube, inter-cube, and inter-node communication throughput. No code modification is required since the runtime system *transparently* realizes the ideas with minor architecture supports.

GraphH [13] is another PIM graph processing architecture that uses interconnection reconfiguration and relies on the host processor to control the switch status of all connections. Perhaps it is most closely related since it also reduces the irregularity of inter-cube data movement. Unlike GraphQ, GraphH only improves inter-cube communication and is purely implemented in hardware and cannot enjoy the flexibility of our co-designed approach.

We evaluate GraphQ with a zSim-based simulator using five real-world graphs and four algorithms, the results show that GraphQ achieves on average 3.3× and maximum 13.9× speedup, 81% energy saving compared with Tesseract. Comparing with GraphP, GraphQ achieves more speedup to Tesseract. With the hybrid model, a 4-node GraphQ achieves an average speedup of 2.98× compared with single-node GraphQ, and 98.34× speedup compared with a single node with the same memory size using conventional memory hierarchy.

2 BACKGROUND AND MOTIVATIONS

2.1 Basics of Graph Processing

Table 1: Vertex Programming APIs

Function	Input	Output
`processEdge`	source vertex value	partial update
`reduce`	reduced/partial update	reduced update
`apply`	reduced update/old value	new value

```
1  for (v ← Graph.vertices) {
2      for (e ← outEdges(v)) {
3          res = processEdge(e, v.value, ...)
4          u ← comp[e.dest]
5          u.temp = reduce(u.temp, res)
6      }
7  }
8  for (v ← Graph.vertices) {
9      v.value, v.active = apply(comp[v].temp, v.value)
10 }
```

Figure 1: Vertex Programming Model

A graph G is defined as an ordered pair (V, E), where V is a set of vertices connected by E, a set of edges. To ease the development of graph algorithms, several domain-specific programming models based on "think like a vertex" principle are proposed, such as vertex program [40], gather-apply-scatter program [19], amorphous data-parallel program [51] and some other frameworks [57]. Among them, vertex program is supported by many software and hardware accelerated graph processing frameworks, including Tesseract [1], GraphLab [39], and Graphicionado [22]. Figure 1 lists the semantics of three APIs of vertex program. Figure 1 shows a general graph application expressed with these primitives.

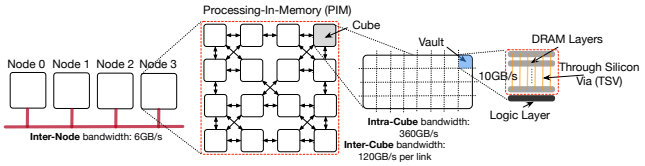

Figure 2: Processing-In-Memory Architecture

During processing, each vertex in vertex array is visited and all its outgoing edges in edge array are processed, involving three steps. 1) *process*: for each outgoing edge `e` of vertex `v`, function `processEdge` computes the contribution of source vertex `v` through edge `e` to the destination vertex `u` (accessed in line 4). 2) *reduce*: from the perspective of `u`, a new update returned by `processEdge` (`res`) is combined using a `reduce` function with the existing value of `u` in compute array, i.e., `u.temp`, incurring a random access. 3) *apply*: after the whole graph is processed in an iteration, the new value of each vertex in compute array is applied to vertex array with the `apply` function. In an iterative graph algorithm, the procedure repeats multiple iterations until certain convergence condition has been reached.

We can summarize two characteristics of graph processing: random accesses to compute array (line 4); the high ratio of memory accesses to computation (`processEdge` is typically simple). As a result, graph algorithms incur random accesses and require high memory bandwidth.

2.2 Processing-In-Memory

Processing-In-Memory (PIM) architecture reduces data movements by performing computations close to where the data are stored. 3D memory technologies (e.g., Hybrid Memory Cubes (HMC) [12] and High Bandwidth Memory (HBM) [31]) make PIM feasible by integrating memory dies and compute logic in the same package, achieving high memory bandwidth and low latency.

Similar to Tesseract and GraphP, this paper considers a general PIM architecture (shown in Figure 2) that captures key features of specific PIM implementations. The architecture is composed of multiple *cubes* connected by external links (e.g., SerDes links in HMC with 120GB/s per link). Within each cube, multiple DRAM dies are stacked with Through Silicon Via (TSV) and provide higher internal memory bandwidth up to 320GB/s. At the bottom of the dies, computational logics (e.g., simple cores) could be embedded. In Tesseract [1], a small single-issue in-order core is placed at the logic die of each vault. It is feasible because the area of 32 ARM Cortex-A5 processors including an FPU (0.68 mm^2 for each core [5]) corresponds to only 9.6% of the area of an 8 Gb DRAM die area (e.g., 226 mm^2 [56]). GraphQ assumes the same setting. With 16 cubes, the whole system delivers 5 TB/s memory bandwidth, considerably larger than conventional memory systems. Moreover, the memory bandwidth grows proportionally with capacity in a scalable manner.

2.3 PIM-Based Graph Processing

Tesseract [1] is a PIM-based graph processing accelerator with 16 cubes. Tesseract provides low-level primitives to support vertex program model. For each vertex, the program iterates over all its edges/neighbors and executes a `put` function for each of them. The signature of this `put` function is `put(id, void* func, void* arg, size_t arg_size, void* prefethch_addr)`. It executes a function call `func` with argument `arg` on the *id*-th cube, therefore it could be either 1) a remote call if the destination vertex resides on a different cube from source vertex; or otherwise 2) a local function call. In the end, a `barrier` ensures that all operations in one iteration are performed before the next iteration.

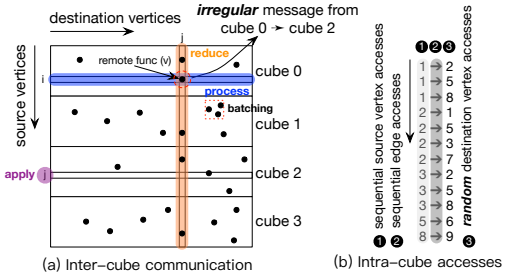

Figure 3: Tesseract Communication and Access Pattern

Figure 3 (a) shows the **inter-cube** communication in an adjacency matrix view, where the rows and columns correspond to the source and destination vertex of edges, and each dot is an edge. All vertices are partitioned among four cubes—each cube is assigned to a set of rows. The circled dot represents an edge from a vertex in cube 0 to a vertex in cube 2: $(v_i \rightarrow v_j)$, which corresponds to an inter-cube message from cube 0 to cube 2. Thus, each edge across cubes incurs such a message and the destination cube is determined by the graph structure (e.g., the destination of the edge). These *small* and *irregular* inter-cube messages are generated during execution to *unpredictable* destination cube at *any time*.

On the receiver side, the core is interrupted to execute the remote function, incurring overhead due to context switch. Tesseract uses batching to mitigate interrupt overhead by buffering the received remote function calls in a queue and executing multiple functions together at certain point later. It can be seen as the square in Figure 3 (a): the functions corresponding to the edges inside the square are executed in batch by a core in remote cube. Due to the large number of inter-cube messages, batches generated are too small to offset the performance impact of interrupt overhead.

Moreover, irregular communication between cubes may incur imbalanced load and hardware utilization. Due to graph-dependent communication pattern, when messages are sent to the same cube from different senders, its message queues may become full and put backpressure on the network to prevent senders from generating more messages. In this case, cores in the receiver cube will be overwhelmed by handling remote function call requests without making progress in processing its local data.

Finally, the dynamic communication pattern leads to excessive energy consumption of inter-cube links. To save energy, each inter-cube link can be set to a low-power state (e.g., the Power-Down mode in HMC [23]). However, this optimization is not applicable to the scenario when the message can be sent at *any* time.

Figure 3 shows the problem with **intra-cube** data movement. If the destination of an edge is in the local cube, a local apply is performed, which incurs random accesses and causes locality interference. Specifically, accesses to vertex array (❶) and edge array (❷) are sequential reads. However, the accesses to compute array for the destination vertices are random (❸). Besides, remote function call also incur random accesses.

GraphH [13] shares some similarity with GraphQ on reducing irregularity of inter-cube communication, we defer the comparison to Section 3.4 after a thorough explanation of ideas of GraphQ.

2.4 Lessons Learned and Design Principles

We believe that an efficient PIM-based architecture should ideally satisfy three requirements. First, inter-cube communication should be *predictable*, i.e., each cube should know exactly *when* the message will arrive from *which* source cube. This would largely eliminate the interrupt overhead in the current designs. There will be still overhead on the receiving cube side to execute the remote function, but these operations will not interfere with the current processing on that cube since they happen at known times. Second, inter-cube data movement should be handled by *heterogeneous* cores in a decoupled manner due to the different access patterns. It is critical to reduce interference given that data from different array share the same cache. Third, the multi-node PIM-based graph processing architecture must efficiently handle the large *discrepancy* of inter-node, inter-cube, and intra-cube bandwidth. The bottom line is that the design should achieve speedup over the conventional memory hierarchy with the same memory size when the graph data are distributed into the PIMs of all nodes.

3 GRAPHQ ARCHITECTURE

We propose GraphQ, the first multi-node PIM-based graph processing architecture built on the recent work Tesseract. Our solution is inspired by techniques in distributed graph processing and irregular applications, but they have never been applied and investigated in the context of PIM. To mitigate the interrupt handling overhead, GraphQ uses predictable inter-cube communication, which is supported by simple reordering of edge processing order according to graph partition in cubes. To enable efficient intra-cube data movement, we divide the cores in a cube into two heterogeneous groups. To hide long inter-node communication latency, we propose a hybrid execution model to perform additional local computations during inter-node communication.

3.1 Predictable Inter-Cube Communication

Batched Communication We propose an execution model that supports *predictable and batched communication*, which is enabled by two key ideas. Shown in Figure 4, the *reduce* step is performed in the source cube. For each edge, instead of sending the function and parameters to a remote cube, the source cube locally reduces the values for each destination vertex. In the matrix view, the reduced value is generated in the cube of the source vertices of edges in the same column. Second, we generate all messages for the *same remote cube* together, so that they can be naturally batched. We partition the whole matrix into *blocks*, each of which contains the edges that will contribute to the batched message between a pair of cubes. For example, the third block in the first row will generate a batched message from cube 0 to cube 2.

In GraphQ, *apply* step in each cube is performed by reducing $(N-1)$ batched messages from other cubes, where N is the number of cubes. In our example, $N = 4$, the cube 2 will reduce the three batched messages in different colors and then update its local vertices with new values.

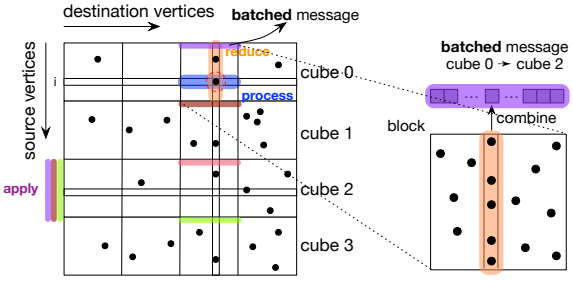

Figure 4: Batched Communication in GraphQ

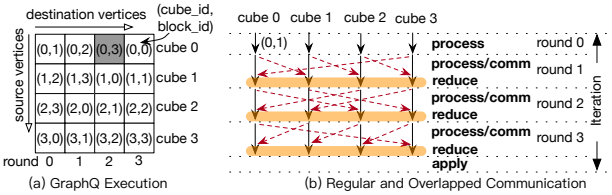

Figure 5: Overlapped Computation and Communication

Rounded Execution The batched communication enables new optimization to support the overlap of communication and computation with balanced execution. The insight is illustrated in Figure 5. We use the (cube_id, block_id) pair to indicate the source and destination of the batched messages. The order of batched messages is determined by the order of blocks in each cube (from left to right in the figure). For example, cube 1 should first process the block (1,2), which will generate a batched message from cube 1 to cube 2. We call this execution model as *rounded execution*, where each iteration is divided into M rounds, M is the number of cubes. The rounded execution is *synchronous*, which means that all cubes have to finish one round before entering the next. With four cubes, there are in total four rounds. The key insight is that, after one round, each cube will only generate *one* batched message for *one* remote cube. Following this principle, the destination cubes should be "interleaved": in the first round, cube 0 generates a message to cube 1, cube 1 generates a message to cube 2, etc. With the starting rounds in all cubes determined, it is easy to derive the others: each cube only need to process the increasing rounds after the first one. For the first $(M-1)$ rounds, the destinations of batched messages are organized in a *circulant* manner. The last round is the same for all cubes: they process the block that will generate only local updates and no inter-cube messages.

At the end of each round, a barrier ensures that all cubes receive the batched message. When all batched messages are received in all cubes, they perform *reduce*, which accumulate the updates from the source cube. After that, the batched message buffer can be reclaimed and is available to be used by the next round. Therefore, only *one* receive message buffer is needed for each cube. Since rounds are executed synchronously, the load imbalance among different cubes in the same round may increase execution time. We study this effect by comparing the sum of maximum cube computation in each round (for rounded execution) with the maximum of the sum of cube computation in all rounds (for no rounded execution). For graphs used in our evaluation, we report the results in Section 5.

In summary, rounded execution enables balanced execution with two properties: *a)* the batched messages from previous round can

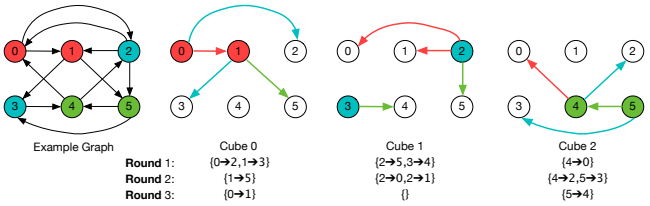

Figure 6: Example of GraphQ

be overlapped with the execution of the current round, so the messages can be sent in the *non-blocking* manner; and *b)* each cube only receives *one batched message in one round*, so only one receive buffer is needed. Note that the irregular inter-cube communication cannot achieve full overlapping, it is the fundamental difference between GraphQ and GraphP/Tesseract. In fact, GraphP tried the idea of overlapping but the results showed that benefits are little. Therefore, GraphP and GraphQ are orthogonal.

Figure 6 shows a concrete example based on a small graph. Suppose we have three cubes, the vertices partitioned into each cube are indicated with different colors. On the right, we show the vertices and edges assigned to each cube. The edges processed in different rounds are shown in the color of the destination cube. The edge sets processed in each cube/round are also shown below.

Preprocessing Preprocessing is common for all graph processing frameworks, including both Tesseract and GraphQ. There are two common preprocessing steps: (1) convert graph in text format, e.g., SNAP (Stanford Network Analysis Project) format [35] or Matrix Market format [48], to binary graph data structure, e.g., CSR (Compressed Sparse Row); (2) partition the graph among cubes. GraphQ requires an additional step to group edge block for each round together to enable batched communication. This incurs small overhead because it does not require an extra iteration over the input. Specifically, during graph partition, we can maintain an edge list for each remote cube, each edge is placed into the corresponding edge list based on its destination vertex. In the end, the edge lists are concatenated together to generate the new combined edge list for each cube.

For graphs used in the simulation, the total preprocessing time is within 3 seconds in a single-thread implementation. Table 2 compares preprocessing time of several large graphs in seconds between GraphQ and Tesseract, we can see that the overhead for both schemes are low, considering it is a an one-time overhead that can be amortized over executions. The difference between the two is the additional overhead due to edge reordering in GraphQ, e.g., for R-MAT-Scale27, GraphQ requires about additional 10s to preprocess the graph. The increases in all three large graphs are less than 10%. In general, the one-time preprocessing time and execution time of graph processing systems are reported separately [11, 19, 73].

Table 2: Preprocessing overhead for large graph datasets

Graphs	$\|V\|$	$\|E\|$	Tesseract	GraphQ
Twitter-2010 [28] (TW)	42M	1.5B	36.7	39.7
Friendster [34] (FR)	66M	1.8B	53.5	58.2
R-MAT-Scale27 [10] (R27)	134M	4.3B	130.2	140.2

3.2 Decoupled Intra-Cube Data Movements

We put the intra-cube architecture of GraphQ in the same context with the conventional multicore and Tesseract. Figure 8 (a) shows

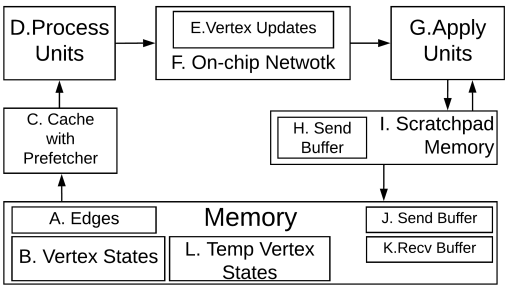

Figure 7: GraphQ Intra-Cube Architecture

a conventional multicore architecture, where a last-level shared cache is placed below private caches. While shared cache facilitates the inter-core communication, it causes a series of issues, including data races which require atomic operations/locks, and coherence protocol. Given the poor locality of graph processing applications, conventional cache hierarchy is not effective.

In comparison, Tesseract eliminates the shared cache and only use a small private cache for each core and a simple prefetcher, as shown in Figure 8 (b). Without the shared cache, Tesseract uses message passing for intra-cube communication, using the same mechanism as inter-cube communication. Specifically, each core has a message queue, a *global router* of each cube inspects the local message queues and sends messages to any core (local or remote) in the system. Without a shared cache, write accesses directly update memory. Since atomic operations and locks are much slower in memory, Tesseract avoids using them by assigning a disjoint set of vertices for each cube to update.

The intra-cube architecture of GraphQ is shown in Figure 8 (c), which has two key differences compared with Tesseract. First, inter-cube and intra-cube data movements are handled separately. For inter-cube communication, batch messages are generated in batch message buffer in memory and sent by routers in our runtime system (see Section 4.3). Intra-cube messages are handled by message queues and *local routers*, because the message source and destination are within the same cube. Second, sequential and random accesses are processed separately.

We divide the cores in the same cube into two groups: *Process Units (PUs)*, which execute `processEdge` function and generate update messages, involving sequential vertex and edge reads in vertex/edge array; and *Apply Units (AUs)*, which directly receive messages from PUs and perform `reduce` and `apply` function, involving random accesses to vertices in compute array. This organization eliminates locality interference. Moreover, we replace the private cache attached to each AU with scratchpad memory (SPM), serving as a buffer of vertex values with contiguous ID. Each AU randomly accesses on-chip SPM, which provides shorter latency and higher bandwidth than L1 cache. At the end of a round, the batched message is ready in SPM and can be written sequentially to batched message buffer in memory. While the functionalities of PUs and AUs are different, they are actually a *subset* of a more general core in Tesseract. For larger graphs when all destination vertices can not fit into SPMs, we will have sub-partitions (more details in Section 4.3.2). The ratio between PUs and AUs are determined empirically and we show the performance of different ratios in Section 5.5.

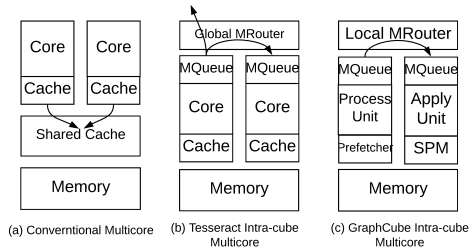

Figure 8: Intra-Cube Architecture Comparison

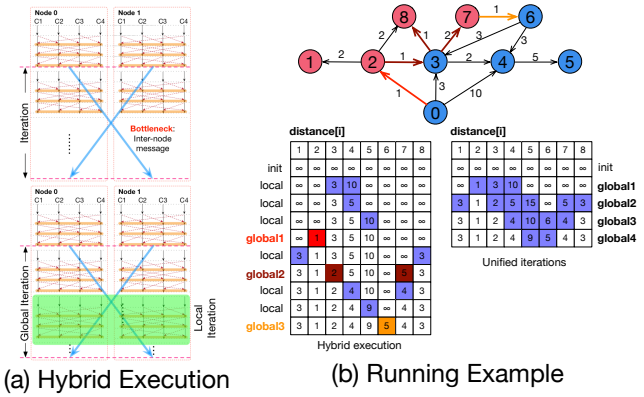

Figure 9: Hybrid Execution Model

3.3 Tolerating Inter-Node Latency

The key challenge of multi-node PIM system is the long inter-node communication latency. Due to the large gap shown in Figure 2, the conventional communication and computation overlapping cannot fully hide such long latency. Figure 9 (a) shows the scenario when the idea of batched and overlapped communication in Section 3.1 is applied to inter-node communication. Unfortunately, the execution of next iteration finishes long before receiving the batched message from a remote node, during this time, each node is idle. According to our experiments, inter-node communication takes 82% to 91% of total execution time, which implies significant time wasted waiting for remote node messages. Note that this issue may get worse in Tesseract because a cube could send and receive both inter-cube and inter-node message at any time, the large latency difference will result in more imbalanced execution.

To solve this problem, we propose a simple but effective bandwidth-aware hybrid execution model, that performs potentially useful computation during idle time. The idea is shown in Figure 9 (a). When each node finishes the execution of an iteration and has to wait for the remote batched message, they can run more iterations based on local subgraph. In this way, the PIM of each node can make use of the idle wait cycles to perform local computation. Specifically, we call the normal iteration as *global iteration*, which is composed of multiple *local iterations*. In a global iteration, the first local iteration is performed after the most recent remote updates are received, the other local iterations are performed based on local subgraph only. In other words, each node runs several local iterations "asynchronously" within the cube before a global synchronization among node at the end of the global iteration. The mechanism proposed in Section 3.1 is still applicable to the local iteration, which is composed of several rounds. During inter-node communication, the loads for local iterations in different nodes can be different, but since it is opportunistic and all overlapped with much longer inter-node communication, such imbalance is not an issue. This model matches the hierarchical bandwidth in multi-node PIM by overlapping the longer inter-node communication with more local computation. Essentially, this design presents a *mid-point* between synchronous and asynchronous execution model—inside a global iteration, the nodes execute asynchronously.

In general, the hybrid execution model is applicable for *asynchronous iterative algorithms* as defined in ASPIRE [66] that can tolerate stale values. The computations that are left behind can be viewed to generate stale values. As long as the system can ensure *bounded staleness*, the results are correct, and there is no need to recovery re-computation. ASPIRE ensures such property by maintaining the staleness information with each vertex. In our hybrid model, since the global synchronization will eventually happen, the staleness is also bounded. All the four algorithms we evaluated (BFS, WCC, PageRank and SSSP) belong to this algorithm category. We experimentally confirmed that they all produce the correct results. Note that the property is proved in [66]. The intuition of the correctness is that, the remote intermediate results (e.g., shortest path in other subgraphs) will eventually be propagated to the local node and "correct" the potentially staled local results. Similar idea was applied in out-of-core system [2], where the subgraph loaded into memory (the other parts are in disk) are processed with multiple iterations. The purpose is to tolerate longer disk access latency. We uniquely apply the idea in multi-node PIM architecture.

Figure 9 (b) shows a running example of Single Source Shortest Path (SSSP) with Bellman-Ford algorithm [7]. We compare hybrid execution model with unified iterations. The graph is partitioned into two nodes, indicated in different colors. On the bottom left, we show the change of distance vector using hybrid execution. We can see that the whole execution incurs three global iterations, they are marked by different colors. The edges in the graph that result in inter-node communication are also marked with the corresponding colors. We can see that during the first global iteration, three local iterations are executed. During the second one, only one local iterations is executed—this case is the same as unified iteration. During the third global iteration, two local iterations are executed. On the bottom right, we show the change of distance vector with unified iterations. We can see that in total four global iterations are needed—one iteration more than the hybrid execution. Note that it is a small example showing the insights, the iteration reduction is only one. In real graphs, as we show in the evaluation, the benefit is significant.

The hybrid execution model is general and widely applicable, because many algorithms executed in the graph parallel execution model are asynchronous and iterative. The parallel version of a certain algorithm is typically different from the optimal sequential version. For example, Dijkstra algorithm for SSSP [15] is sequential and difficult to parallelize. For this reason, the graph processing framework normally uses the non-optimal (i.e., may lead to redundant work) but more relaxed iterative algorithms (e.g., Bellman-Ford algorithm [7] for SSSP) which are more amendable for parallel execution. The hybrid execution is applicable to all such relaxed iterative algorithms used in parallel graph processing. It is possible to support sequential optimal algorithms (e.g., Dijkstra) in a type

of architecture that can execute logically sequential tasks in parallel with speculative execution, e.g., Ordered Parallelism [24, 45]. Since PIM does not provide the architectural supports required for speculative tasks and most parallel graph processing frameworks have not yet considered that option, we leave it as future work.

3.4 Novelty and Discussion

GraphQ is inspired by the ideas in distributed graph processing and irregular applications. It does not weaken the contribution of this paper because: 1) we are the first to investigate the benefits in the context of PIM with detailed architecture model; and 2) some subtle but important differences exist.

The communication and computation overlapping are well-known optimizations in high performance computing [14, 54]. Our architectural model captures key aspects such as the reduction of communication energy and each core's interrupt overhead. They are not considered in distributed graph processing (e.g., Gemini [73]). The idea of local reduction near sources is explored in PowerGraph [19]. However, it is only one technique (among the three) that enables the predictable and batched communication in GraphQ.

The more subtle distinction is batching. Grappa [47] and ASPIRE [66] are two closely related recent work. They are both latency-tolerant distributed shared memory (DSM) system and use batching to reduce communication overhead. The key difference between GraphQ is *message aggregation*. In Grappa and ASPIRE, the message aggregation is *dynamic and unstructured*, which means that if the sender accumulates several messages for the same destination, a batched message can be formed and sent to the receiver. It is dynamic because the aggregation is determined by the runtime processing order of vertices and the graph. Due to the unstructured batching, the communication is still irregular. In comparison, the message aggregation in GraphQ is *static and structured*, which means that the messages to the same destination are forced to be generated and sent. As a result, the communication becomes regular. In fact, the batching mechanism in Tesseract is exactly dynamic and unstructured: during one iteration, a cube can send *multiple* batched messages to the same remote cube. In GraphQ, a cube can only send *one* batched message to the same remote cube. Tesseract is more similar to Grappa and ASPIRE. Moreover, due to regular communication, GraphQ enables link power down.

Both Grappa and ASPIRE require an outstanding thread or core to manage communication. In particular, ASPIRE needs to maintain the per-vertex information on staleness. While it is possible to have such support in a DSM with a x86 core working on a full-fledged NIC, it is infeasible to implement in PIM.

Next, we discuss two closely related schemes. GraphH [13] also uses the idea of rounded execution and batching. It is implemented by reconfiguring interconnection network and relies on the host processor to control the switch status of all connections. Instead, GraphQ only requires lightweight hardware primitives. We believe that our approach is more flexible and incurs much less hardware modifications. Nevertheless, besides rounded execution that enhance the efficiency of inter-cube communication. GraphP [72] uses replicas to reduce inter-cube communication, converting one message per inter-cube edge to one message per replica synchronization. However, such replica synchronization still incurs irregular communication, making it suffer from the similar drawbacks as Tesseract. The irregular communication in GraphP also limits its capability to overlap communication with computation, which explains the reported minor improvement (Figure 10 and 11 in [72]).

In summary, the key takeaway in GraphQ is that, the static and structured communication pattern is critical to achieve good throughput and performance on PIM-based architecture.

4 GRAPHQ IMPLEMENTATION

GraphQ is implemented by runtime and architecture co-design: the *architecture* provides the communication and synchronization primitives; and *runtime system* orchestrates the execution of the user-defined functions with batched inter-cube communication and decoupled intra-cube data movements using architectural primitives. The proposed techniques are *transparently* implemented, thus require no code modification. Similar to other graph processing frameworks, GraphQ is *fully compatible* with the widely-used vertex programming model. Programmers only need to define three functions: `processEdge`, `reduce`, `apply`.

4.1 Architectural Primitives

4.1.1 Inter-Cube: Batched Communication.

Communication Primitives GraphQ architecture provides three inter-cube primitives: `initBatch`, `sendBatch`, `recvBatch`.

To initiate communication, each cube registers its send and receive buffer through `initBatch`. The local router allocates two shadow send and receive buffer in memory, and initializes status flags for each buffer. These flags keep track of buffer availability.

The `sendBatch` is, in essence, buffered non-blocking asynchronous send. This primitive offloads the send operation to router after the send buffer in memory is copied to its corresponding shadow buffer, so that the computation can proceed and send buffer can be reused. Before starting writing to remote buffer, the router checks the flag of remote shadow receive buffer to make sure that it can be overwritten.

The `recvBatch` is blocking synchronous and does not require any parameters. In our ordered batched communication, the source cube id of messages can be inferred from round ID. The status of receive buffer indicates whether new messages have arrived. If so, the messages are copied from shadow receive buffer to receive buffer in memory and flags are reset. The `reduce` function can be executed only after receiving updates from remote cubes, except the last round.

Inter-cube Link In GraphQ, with respect to one intercube routing algorithm, we know at static time that some links are idle in the entire round. Moreover, at runtime, when the communication time is overlapped by computation, we can set the links to idle or sleep mode when the communication has completed. In HMC, this is can be achieved by setting the "Power-Down" signal according to HMC specification 2.1 [12]. We also take into consideration the link state transition time (150 µs). The benefit of this optimization is more prominent when the graph size is larger, because the transition time will become negligible.

4.1.2 Intra-Cube: Specialized Message Passing.

Compute Units GraphQ uses single-issue in-order cores in the logic dies to meet the thermal and area requirements. We leverage heterogeneous cores to fully utilize the high memory bandwidth.

Process Unit (PU) is responsible for *sequentially* reading vertices and edges from memory and performing operations (`processEdge`). A simple stride prefetcher is employed to match the high memory bandwidth and hide latency. The output of PUs are update messages to be sent to AUs through the on-chip network.

Apply Unit (AU) receives update messages from PUs, performs `reduce`, and writes (`apply`) destination vertices with random accesses. In essence, AUs prepare the batched messages to be sent to a remote cube at the end of a round. Instead of using a prefetcher, we replace the private cache of each core with a programmer controlled scratchpad memory (SPM) as data cache. On one hand, SPM is faster than L1 cache. On the other hand, SPM allows the software to explicitly allocate space in SPM, and vertex data will not be evicted due to cache replacements.

The PUs and AUs in a cube form a data movement *pipeline*: PUs continuously perform data processing and send update messages to AUs through on-chip interconnect; AUs randomly fill SPM with the reduced updates. In the end of a round, the prepared batched message is written sequentially from SPM to send buffer in memory.

On-chip Interconnect In order to support the pipelined data movements from PU to AU, GraphQ provides primitives: `Send`, `Recv`, `Sync`; and architecture supports: local router, message queues, interconnection between cores for intra-cube communication. The separation of inter-cube and intra-cube communication leads to simpler interconnect and router design and lower pressure to hardware resources. In our implementation, each PU has a send queue and each AU has a receive queue. The local router is responsible for moving data from send queues of PUs to receive queues of AUs.

`Send` is executed asynchronously. For a PU, sending a message simply means moving data from register to its send queue and continuing the execution without waiting for any return value. The message transfer will be handled by the local router. `Recv` fetches one message from queue. When a new message arrives, AU uses `Recv` to move it from receive queue to register. `Sync` is a signal emitted by PUs to notify all AUs in the same cube, indicating that it has reached the end of a round.

Although `Send` is asynchronous, it can block a PU when its send queue is full. This is a common problem for both Tesseract and GraphQ. However, because GraphQ's on-chip network only handles intra-cube communication, in practice we can choose proper queues capacity (less than that in Tesseract) to buffer the incoming send messages.

To avoid the overhead of context switch, we implement the primitives by extending the instruction set of PU and AU. In addition, reading from or writing to message queue takes one cycle, without stalling processor pipeline. It is reasonable for the small intra-cube messages of graph applications, in which the essential operation is updating vertex values. Typically, the vertex value has basic data type less than 64 bits, such as `int` or `float`. It is true for all applications evaluated in the paper. With 64-bits vertex ID, the message size is only 128 bit.

4.2 Parameter Consideration

In the implementation, we need to decide certain key parameters. The first is the number of PUs and AUs. The total number of cores is limited by the size of logic die per cube. The number of PUs should fully exploit memory bandwidth. For example, in HMC, the available internal bandwidth is 320 GB/s. Suppose the PUs run at 1 GHz and a prefetcher of 64B works perfectly with sequential pattern, 6 PUs are enough. In contrast, AUs should be able to consume the stream of messages from PUs and clear the receive queue in time. We tune these parameters so that the execution is rarely blocked by hardware resources limitation. In GraphQ, the number of PUs and AUs are both 8, and the queue size is 16.

The second issue is the size of scratchpad memory. In Tesseract, the compute unit is ARM Cortex A5, with configurable L1 data cache from 4KB up to 64KB. We expect our program-controlled SPM has capacity of the same order and thus use 64KB SPM per AU. In total, we have 8*64 KB SPMs per cube, which can hold 128K vertex values of 4 Bytes. Since in one round, only 1/16 of total vertices can be destination, graphs with less than 2M (16*128K) vertices can be completely held in SPMs. If the input graph size increases, we can further divide the vertices into more blocks as we will see later in Section 4.3.2.

4.3 GraphQ Runtime System

GraphQ allows programmers to specify graph applications using vertex programming interface; and the mechanisms for regular communication and data movements are supported transparently by runtime system. Due to space limit, we do not include inter-node supports, but it is similar to inter-cube. The number of local iterations in a global iteration can be specified as a parameter. For intra-cube execution, we will first explain the case when the memory usage of destination vertices does not exceed the SPM capacity, and then discuss the solution to scale to larger graphs at the end of Section 4.3.2.

4.3.1 Inter-Cube Communication. Figure 10 shows the pseudocode for one iteration concurrently executed in each cube based on vertex programming API and batched communication primitives. The current cube ID is stored in `myId`, ranging from 0 to (`cubeNum`-1), `cubeNum` is the total number of cubes. The source and destination of cube ID of inter-cube communication in a round is `fromId` and `toId`, respectively. To enable batched communication, three local buffers are used for each cube. They are `sendBuffer`, which buffers the updates to be sent to remote cubes in each round; `recvBuffer`, which buffers the updates received from other cubes by the end of each round; and `tempBuffer`, which stores the partially reduced values.

In each round, graph blocks corresponding to destinations in cube `toId` are streamed in (line 7 and 8), update messages are generated (line 9) and reduced in `sendBuffer` (line 10). At the end of the round, i.e., computation has finished, each cube performs inter-cube communication (line 13 and 14). During the period of rounds, updates from remote cube in the previous round are received with `recvBuf` with `recvBatch` primitive (line 20) and reduced in `tempBuffer` (line 22). Updates in `sendBuffer` are transferred in batch using `sendBatch` primitive (line 14).

In the first round, we will not call `recvBatch` (if-statement at line 19), because no messages are expected to arrive in round 0: the first batched message will be sent in the end of round 0, which will be received by the end of round 1, so the overlapped computation and communication starts from round 1. In the last round, `sendBatch`

```
1  sendBuf = local Array[DataType]
2  recvBuf = local Array[DataType]
3  tempBuf = local Array[DataType] //partially reduced
       values
4  InitBatch(sendBuf, recvBuf)
5  for (roundId = 0; roundId < cubeNum; roundId++) {
6      toId = (myId + roundId + 1) % cubeNum
7      for (v <- GraphBlock.vertices) {
8          for (u <- outNbrs(v)) { //overlapped with comm
9              res = processEdge(u.value, v.value, ...)
10             sendBuf(u) = reduce(sendBuf(u), res)
11         }
12     }
13     if (roundId != (cubeNum-1)) { //end of each round
14         SendBatch(toId)           //except last round
15     } else { //end of last round
16         for (v <- Partition.vertices)
17             tempBuf(v) = reduce(tempBuf(v),
                   sendBuf(v)) //local reduce
18     }
19     if (roundId != 0) {
20         RecvBatch() //each round except first
21         for (v <- Partition.vertices)
22             tempBuf(v) = reduce(tempBuf(v),
                   recvBuf(v)) //per-round reduce
23     }
24 }
25 for (v <- Partition.vertices) //final apply
26     v.value, v.active = apply(tempBuf(v), v.value)
```

Figure 10: Ordered Batch Inter-cube Communication Code

```
1  for (v <- Partition.vertices) {
2      for (e <- GraphBlock.outEdges(v)) {
3          res = processEdge(e, v.value, ...)
4          Send(res)
5      }
6  }
7  Sync()
```

Figure 11: Process Unit Code

is omitted (if-statement at line 13), because in round (cubeNum-1) the generated updates should be reduced locally.

Note that, while sendBuf (line 10) is nonblocking, the whole batched send communication might be blocked. It is because the sender needs to wait until the remote shadow receive buffer becomes available. As discussed in Section 4.1.1, this is ensured by the semantic of sendBatch primitive: the router will automatically check whether the shadow buffer on remote cube is available, if not, the data is kept in shadow send buffer and wait.

Finally, after finishing the rounded execution (in line 17) updates from local cube (i.e., in sendBuf) and remote cubes in this iteration have already been reduced (i.e., in tempBuf). The finally reduced updates are applied to vertex states (line 26), which will be used as new vertex states in the next iteration.

4.3.2 Intra-Cube Message Passing. In Figure 10, computation in each round (line 10 to 16 and line 18 to 23) is modified to leverage the pipelined heterogeneous compute units. The runtime operations for PUs and AUs are shown in Figure 11 and 12, respectively.

```
1  buffer = Arary[DataType]
2  countSync = 0
3  while (countSync < numPU) {
4      msg = Recv()
5      msg match {
6          case Sync() => countSync++
7          case Send(uData, uAddr) =>
8          buffer(uAddr) = reduce(buffer(uAddr), uData)
9      }
10 }
11 buffer.flush()
```

Figure 12: Apply Unit Code

The code in PU resembles the original program in Figure 10, except that the reduce function is replaced with send. The graph blocks in each PU is further divided and organized as a contiguous region in memory. Hence, all memory accesses in PU are sequential, which can benefit from the high intra-cube memory bandwidth. When all edges in its block have been processed, PU broadcasts sync message to all AUs in the same cube.

As shown in Figure 12, an AU allocates a buffer in SPM, as a copy of sendBuffer in Figure 10. AU uses a counter countSync to keep track of the number of PUs synced. The main body of AU implementation is a while-loop, which keeps peeking message queues with the Recv primitive. Once a new message arrives, the data is reduced in buffer residing in SPM. If the message is a Sync, countSync is increased. The loop exits when sync messages from all PUs have been received. Finally, each AU will flush the buffer in SPM to memory with sequential writes.

When the graph is larger, the destination vertices can not fit into the SPMs. GraphQ will divide the destination vertices to smaller sub-partitions and run the intra-cube execution for each sub-partition. Figure 13 shows the intra-cube execution of one round for one cube in matrix view of a graph. In the figure, we have 4 PUs, 4 AUs and 2 sub-partitions. Each sub-partition contains edges with half destination vertices subset. In the first run, only 3 edges will be processed by PUs, and they all have the same destination vertex assigned to AU3. After all the edges are applied, we synchronize and start the next run. In this way, we can run arbitrary large graphs while all the apply operations fall in SPMs.

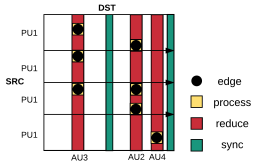

Figure 13: GraphQ Intra-Cube Execution

5 EVALUATION

5.1 Evaluation Methodology

We evaluate GraphQ based on zSim [53], a scalable x86-64 multicore simulator. We modified zSim according to HMC's memory and interconnection model, heterogeneous compute units, on-chip network and other hardware features. While zSim does not natively support HMC interconnection simulation, we insert a NOC layer between LLC and memory to simulate different intra-cube and inter-cube memory bandwidth. The results are validated against NDP [18]. For compute units, we use 256 single-issue in-order cores in Tesseract

Table 3: Graphs Datasets

Graphs	#Vertices	#Edges
ego-Twitter (TT) [42]	81K	2.4M
Soc-Slashdot0902 (SD) [36]	82K	0.95M
Amazon0302 (AZ) [32]	262K	1.2M
Wiki (WK) [8]	4.2M	101M
LiveJournal (LJ) [36]	4.8M	69M

and GraphP. Each core has 32KB L1 instruction cache and 64K L1 data cache. Cache line size is 64B and simulation frequency is 1000 MHz. In GraphQ, we also use the same number of cores-256 in total, in which 128 are PUs and 128 are AUs. PU has a 64B prefetcher with 4KB buffer and AU has a 64KB scratchpad memory. Each core has a 16-entry message queue and 32KB L1 instruction cache, with no L2 or shared cache. For memory configuration, we use 16 cubes (8 GB capacity, 512 banks). The cubes are connected with the Dragonfly topology [27]. The maximal internal data bandwidth of each cube is 320GB/s. We run four widely-used application benchmarks: Breadth-First Search (BFS), Weakly Connected Component (WCC), PageRank(PR) [50], Single Source Shortest (SSSP). [1] Figure 3 shows the graph datasets that we use in our experiment respectively. The dataset is similar as in GraphP, where we replace WikiVote [33] with a larger Wiki graph. [2]

The energy consumption of the inter-cube interconnect is estimated as two components: a) The dynamic consumption, which is proportional to the number of flit transfer events that happen among each pair of cubes; and b) the static consumption, which corresponds to the energy cost when the interconnect is plugged in power but in idle state (i.e., no transfer event happens). We use zSim to count the number of transfer events and use ORION 3.0 [25] to model the dynamic and static power of each router. We calculate the leakage energy of the whole interconnect from the flit transfers. We also validated Table 1 in [64] with McPAT [37].

5.2 Comparing with Tesseract

From Figure 14, GraphQ achieves 1.1x - 14x speedup across all four benchmarks with different graph inputs. Specifically, for WCC/PageRank with batching, the maximum speedups reach 6x/4x. When the intra-cube optimization is enabled, the further increase to 16x/6x. BFS and SSSP achieve less maximum speedup (2x-3x). The reason is that WCC and Pagerank are "all-active" benchmarks, i.e., all vertices in the graph are active in each iteration, while BFS and SSSP only enable partial vertices and edges. Thus WCC and Pagerank benefit more from batching communication. While there are other algorithms for WCC and PageRank that does not require all-active execution, we use the all-active in this paper (same in many other publications [22, 72–74]) to demonstrate the performance characteristics in various application settings.

Figure 15 shows the time breakdown. In GraphQ, computation and communication cannot fully overlap. The interruption adds extra overhead of about 10%, and synchronization taks 10% to 50% time. In GraphQ, two parts are almost invisible: communication

[1] For BFS, we do not use direction-switch optimization. For SSSP, We run Bellman-Ford algorithm for fixed number of iterations.

[2] We do not use road graphs because the dataset are relatively small to fit in one memory cube.

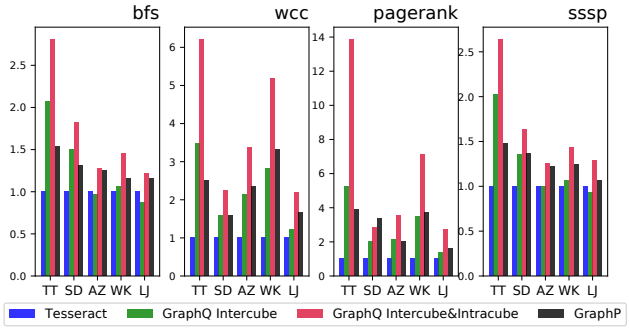

Figure 14: Performance

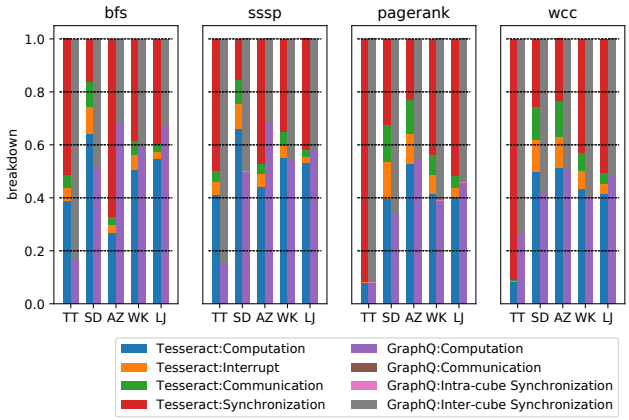

Figure 15: Execution time breakdown

is overlapped by computation; intra-cube synchronization (among compute units in the same cube) is low. The inter-cube synchronization percentage is high. In some cases it reaches 60% and is higher than Tesseract. However, considering that GraphQ total execution time is less, the absolute time wasted in synchronization is still less in GraphQ. The conclusion is that overall GraphQ regular batch communication is better than Tesseract irregular peer-to-peer communication. We admit that the current design has much room for improvement to reduce synchronization and achieves better load balance.

Figure 16 illustrates total bytes transferred by inter-cube routers, with results normalized to Tesseract. Compared with Tesseract, GraphQ reduces the communication amount by at least 70% in all experiments. First, Tesseract inter-cube global routers are responsible for handling both inter-cube and intra-cube communication, while intra-cube messages in GraphQ are sent through on-die network and not counted (Section 3.2). Second, the batch optimization in inter-cube communication combines some messages at the sender size (Section 3.1). Note that when running BFS and SSSP on workload AZ in GraphQ, the inter-cube transfer amount is negligible. This is because AZ has good locality and most updates are applied in the same cube without generating inter-cube messages.

5.3 Comparing with GraphP

We also implement GraphP and quantitatively compare it with GraphQ. Among the four common graph datasets, the overall performance of GraphQ is consistently better: GraphQ speedup is 3.2x on average and 13.9x at maximum, while GraphP is only 1.6x and

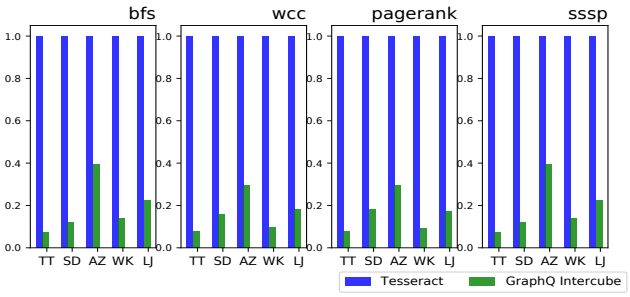

Figure 16: Communication

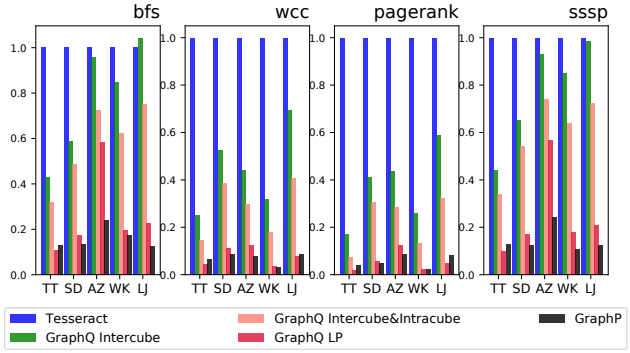

Figure 17: Energy Consumption

3.9x. In GraphP, overlapping is not effective: in several dataset applying the technique leads to slower result due to the dynamic execution. GraphQ enables efficient overlapping with structured batching. Moreover, GraphQ benefits from intra-cube optimization, which is considered in GraphP. The pipelined architecture leverages the high local memory bandwidth and accelerate the computation further by 56%.

5.4 Energy Consumption

Figure 17 shows the interconnect energy consumption of GraphQ compared with Tesseract. As we can see, batching contributes from 10% to 85% reduction and batching with PU/AU optimization contributes up to 90% reduction in energy cost in most of our experiment runs. The energy cost of the interconnect consists of both the static consumption and the dynamic consumption, which are determined by the execution time (performance) and communication amount, respectively. Figure 17 also demonstrates the energy cost with cubes' low-power option enabled. We see that GraphQ's capability of setting the low-power option further saves around 50% - 80% of energy across all benchmarks and thus drastically reduces energy cost by 81% on average and 98% in maximum.

5.5 Effect of PU/AU Ratio

Figure 18 shows different performance results under different PU/AU configuration, when running pagerank on enwiki graph. If there are only 2 to 4 PUs, the memory bandwidth is far from saturation and thus it spends 87% more cycles compared with the optimal case. If we replace most AUs with PUs, the system is bottlenecked by the AU operations, i.e., writing to SPM. In this setting, total cycles will increase by 93%. With poor configuration, the streamlined processing can even get worse performance.

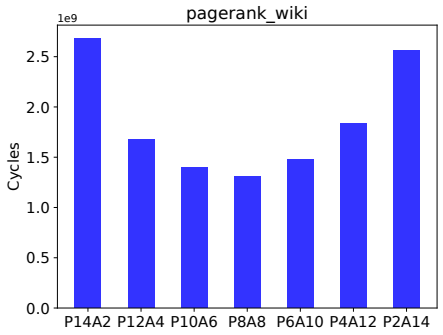

Figure 18: Performance w.r.t Different PU/AU Ratios

5.6 Multi-Node Performance

We evaluate a 4-node system where each machine has the same HMC memory setting as in the single-node performance evaluation. The inter-node, inter-cube and intra-cube bandwidth are 6GB/s, 120GB/s and 360GB/s, respectively. Each global iteration contains four local iterations. The results are shown in Figure 19 where speedup is normalized to GraphQ single node PIM performance. For many test cases, multi-node Tesseract speedup is less than 1, it is because the inter-node communication becomes the new bottleneck. Moreover, the problem of irregular communication in Tesseract becomes worse in a low bandwidth multi-node setting and significantly limits its scalability. Take PageRank as an example, single-node Tesseract is 3x to 14x slower than GraphQ, while the multi-node Tesseract can be 61x slower than GraphQ. The speedup of GraphQ is also less in multi-node setting, but due to regular batch communication and fast single-node design, multi-node GraphQ is consistently better than Tesseract.

GraphQ's hybrid execution alleviates this problem and makes PIM-based graph processing still efficient across different machines. In a 4-node PIM, the speedup is 2.98× compared with single-node GraphQ, which translates into a *98.34× speedup compared with a single node with conventional memory hierarchy of the same memory size.* This is because the 4-node system has more computing resources (more cores embedded in the cube) and memory bandwidth. Specifically considering hybrid execution model, it leads to average 39.3% (at most 2.57×) speedups than the executions without the optimization.

5.7 Larger Graphs

We can not run larger graphs due to simulation constraints. [3] However, for larger graphs, load imbalance among different cubes in the same round might lead to more inter-node synchronization and significant increase in total time.

To evaluate the issue for larger graphs, we use the similar methodology to investigate estimate load imbalance as in Section 3.1. If there is no inter-cube synchronization (no rounded execution in GraphQ), the total time will be determined by the slowest cube, i.e. the maximum of the sum of cube time in all rounds. If there is the additional synchronization in GraphQ, the time spent in each round will be determined by the slowest cube, i.e. maximum cube time in each round, and the total time is the sum of all rounds.

[3] While both [45] and GraphQ also uses zsim, the number of cores simulated in our study is much larger, so we can not run the same larger graphs.

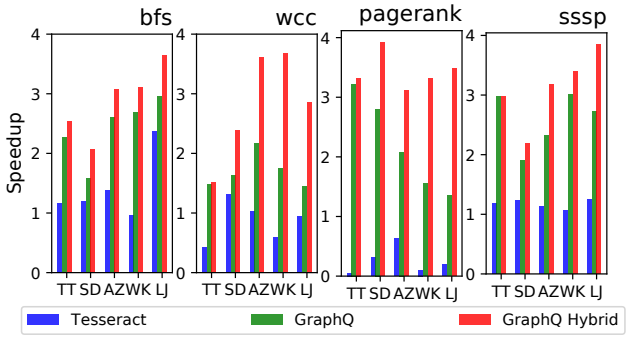

Figure 19: Multi-Node Performance

In graph processing, the execution time can be approximated by the number of traversed edges. To get the number of edges, we run three very large graphs (aforementioned in Table 2) in real distributed systems [73] with 16 nodes. The results show that load imbalance introduced by GraphQ synchronization is 33%, 27%, and 0.55% of total computation respectively. The average synchronization overhead is even smaller than the smaller graphs we use in Table 3.

Based on the above discussion, we believe that for larger graphs, the benefit of regular communication is more than the additional synchronization. Therefore, GraphQ is still better than Tesseract.

6 RELATED WORK

Tesseract [1] is the first PIM-based accelerator and is the baseline of this paper. Ozdal *et al.* [49] proposed an accelerator for asynchronous graph processing, which features efficient hardware scheduling and dependence tracking. Both GraphQ and Tesseract are designed for synchronous processing. GraphPIM [46] demonstrates the performance benefits for graph applications by adding the atomic operations to PIM. Graphicionado [22] is a high performance customized graph accelerator, based on specialized memory subsystem, instead of PIM. GraphP [72] proposes a graph partitioning method that reduces inter-cube communication, but it is based on single node and did not enable the regular data movements. [6] characterized the memory system performance of graph processing workloads and proposed a physically decoupled prefetcher that improves the performance of these workloads. Overlapping communication and computation [41, 54, 65], graph partition [30, 71], graph load balance [59], and graph load characterization [60, 61] is studied extensively in distributed computing setting. However, it is a new problem in PIM with multiple cubes and nodes.

7 CONCLUSION

This paper proposes GraphQ, a novel PIM-based graph processing architecture that eliminates irregular data movements. The key idea is to generate static and structured communication with runtime system and architecture co-design. Using a zSim-based simulator and five real-world graphs and four algorithms, the results show that GraphQ achieves on average 3.3× and maximum 13.9x speedup, 81% energy saving compared to Tesseract. Comparing to GraphP, GraphQ achieves more speedups to Tesseract. In addition, the 4-node GraphQ achieves 98.34× speedup compared to a single node with the same memory size using conventional memory hierarchy.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This research is supported by the National Science Foundation grants CCF-1657333, CCF-1717754, CNS-1717984, CCF-1750656, and CCF-1919289. It is also partially supported by the National Key R&D Program of China under grant 2017YFB0203201 and NSF China under grant 61732002.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 105–117.

[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*. 125–137.

[3] Tero Aittokallio and Benno Schwikowski. 2006. Graph-based methods for analysing networks in cell biology. *Briefings in bioinformatics* 7, 3 (2006), 243–255.

[4] Andrei Alexandrescu and Katrin Kirchhoff. 2007. Data-Driven Graph Construction for Semi-Supervised Graph-Based Learning in NLP.. In *HLT-NAACL*. 204–211.

[5] ARM. 2009. ARM Cortex-A5 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a5.php.

[6] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 373–386.

[7] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[8] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. ACM, 595–602.

[9] William M Campbell, Charlie K Dagli, and Clifford J Weinstein. 2013. Social network analysis with content and graphs. *Lincoln Laboratory Journal* 20, 1 (2013), 61–81.

[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 1.

[12] Hybrid Memory Cube Consortium et al. 2015. *Hybrid memory cube specification version 2.1.* Technical Report.

[13] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[14] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. 2005. Transformations to parallel codes for communication-computation overlap. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 58.

[15] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[16] Anton J Enright and Christos A Ouzounis. 2001. BioLayoutâĂŤan automatic graph layout algorithm for similarity visualization. *Bioinformatics* 17, 9 (2001), 853–854.

[17] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. 2007. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on knowledge and data engineering* 19, 3 (2007), 355–369.

[18] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 113–124.

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 17–30.

[20] Amit Goyal, Hal Daumé III, and Raul Guerra. 2012. Fast large-scale approximate graph construction for nlp. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics, 1069–1080.

[21] Ziyu Guan, Jiajun Bu, Qiaozhu Mei, Chun Chen, and Can Wang. 2009. Personalized tag recommendation using graph-based ranking on multi-type interrelated objects. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 540–547.

[22] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[23] Hybrid Memory Cube Consortium. 2015. *Hybrid Memory Cube Specification 2.1.*

[24] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 228–241. https://doi.org/10.1145/2830772.2830777

[25] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. 2012. ORION 2.0: A Power-Area Simulator for Interconnection Networks. *IEEE Trans. Very Large Scale Integr. Syst.* 20, 1 (Jan. 2012), 191–196. https://doi.org/10.1109/TVLSI.2010.2091686

[26] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2016. High-Level Programming Abstractions for Distributed Graph Processing. *CoRR* abs/1607.02646 (2016). http://arxiv.org/abs/1607.02646

[27] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 145–156.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. ACM, 591–600.

[29] Nicolas Le Novere, Michael Hucka, Huaiyu Mi, Stuart Moodie, Falk Schreiber, Anatoly Sorokin, Emek Demir, Katja Wegner, Mirit I Aladjem, Sarala M Wimalaratne, et al. 2009. The systems biology graphical notation. *Nature biotechnology* 27, 8 (2009), 735–741.

[30] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. 2015. Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 56, 12 pages. https://doi.org/10.1145/2807591.2807632

[31] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, et al. 2014. 25.2 A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, 432–433.

[32] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)* 1, 1 (2007), 5.

[33] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed Networks in Social Media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1361–1370. https://doi.org/10.1145/1753326.1753532

[34] Jure Leskovec and Andrej Krevl. 2014. friendster. https://snap.stanford.edu/data/com-Friendster.html

[35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[36] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[37] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 469–480.

[38] Shuchuan Lo and Chingching Lin. 2006. WMR–A Graph-Based Algorithm for Friend Recommendation. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*. IEEE Computer Society, 121–128.

[39] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[40] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[41] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. 2010. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 5–16. https://doi.org/10.1145/1810085.1810091

[42] Julian McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates Inc., USA, 539–547. http://dl.acm.org/citation.cfm?id=2999134.2999195

[43] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 25.

[44] Batul J Mirza, Benjamin J Keller, and Naren Ramakrishnan. 2003. Studying recommendation algorithms by graph analysis. *Journal of Intelligent Information Systems* 20, 2 (2003), 131–160.

[45] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*.

[46] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading

in Graph Computing Frameworks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 457–468.
[47] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2014. Grappa: A latency-tolerant runtime for large-scale irregular applications. In *International Workshop on Rack-Scale Computing (WRSC w/EuroSys)*.
[48] NIST (National Institute of Standards and Technology). 2000. Matrix Market. https://math.nist.gov/MatrixMarket/index.html.
[49] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 166–177.
[50] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).
[51] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, Vol. 46. ACM, 12–25.
[52] Meikang Qiu, Lei Zhang, Zhong Ming, Zhi Chen, Xiao Qin, and Laurence T Yang. 2013. Security-aware optimization for ubiquitous computing systems with SEAT graph approach. *J. Comput. System Sci.* 79, 5 (2013), 518–529.
[53] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963
[54] José Carlos Sancho, Kevin J Barker, Darren J Kerbyson, and Kei Davis. 2006. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In /. IEEE, 17.
[55] Satu Elisa Schaeffer. 2007. Survey: Graph Clustering. *Comput. Sci. Rev.* 1, 1 (Aug. 2007), 27–64. https://doi.org/10.1016/j.cosrev.2007.05.001
[56] Manjunath Shevgoor, Jung-Sik Kim, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Aniruddha N Udipi. 2013. Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 198–209.
[57] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 135–146.
[58] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. 2016. Parallel Local Graph Clustering. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1041–1052. https://doi.org/10.14778/2994509.2994522
[59] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John. 2016. Proxy-Guided Load Balancing of Graph Processing Workloads on Heterogeneous Clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*. 77–86. https://doi.org/10.1109/ICPP.2016.16
[60] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 154–168. https://doi.org/10.14778/3282495.3282501
[61] S. Song, X. Zheng, A. Gerstlauer, and L. K. John. 2016. Fine-grained power analysis of emerging graph processing workloads for cloud operations management. In *2016 IEEE International Conference on Big Data (Big Data)*. 2121–2126. https://doi.org/10.1109/BigData.2016.7840840
[62] AM Stankovic and MS Calovic. 1989. Graph oriented algorithm for the steady-state security enhancement in distribution networks. *IEEE Transactions on Power Delivery* 4, 1 (1989), 539–544.
[63] Lei Tang and Huan Liu. 2010. Graph mining applications to social network analysis. In *Managing and Mining Graph Data*. Springer, 487–513.
[64] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Sotware-Defined Cache Hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 652–665.
[65] Ta Quoc Viet and Tsutomu Yoshinaga. 2006. Improving linpack performance on SMP clusters with asynchronous MPI programming. *IPSJ Digital Courier* 2 (2006), 598–606.
[66] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. ASPIRE: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 861–878.
[67] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. 2011. Understanding graph sampling algorithms for social network analysis. In *2011 31st International Conference on Distributed Computing Systems Workshops*. IEEE, 123–128.
[68] Yong-Jie Wang, Ming Xian, Jin Liu, and Guo-yu Wang. 2007. Study of network security evaluation based on attack graph model. *JOURNAL-CHINA INSTITUTE OF COMMUNICATIONS* 28, 3 (2007), 29.
[69] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. Tux^2: Distributed Graph Computation for Machine Learning. In *The 14th USENIX Symposium on Networked Systems Design and Implementation*.
[70] Torsten Zesch and Iryna Gurevych. 2007. Analysis of the Wikipedia category graph for NLP applications. In *Proceedings of the TextGraphs-2 Workshop (NAACL-HLT 2007)*. 1–8.
[71] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *The 12th USENIX Symposium on Operating Systems Design and Implementation*.
[72] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 544–557.
[73] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System.. In *OSDI*. 301–316.
[74] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.