GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition

^{1}Mingxing Zhang†§ ^{1}Youwei Zhuo‡ Chao Wang‡ Mingyu Gao* Yongwei Wu†
Kang Chen† Christos Kozyrakis* Xuehai Qian‡
†*Tsinghua University*^{2} ‡*University of Southern California* **Stanford University* §*Sangfor Technologies Inc.*

***Abstract*—Processing-In-Memory (PIM) is an effective technique that reduces data movements by integrating processing units within memory. The recent advance of "big data" and 3D stacking technology make PIM a practical and viable solution for the modern data processing workloads. It is exemplified by the recent research interests on PIM-based acceleration. Among them, TESSERACT is a PIM-enabled parallel graph processing architecture based on Micron's Hybrid Memory Cube (HMC), one of the most prominent 3D-stacked memory technologies. It implements a Pregel-like vertex-centric programming model, so that users could develop programs in the familiar interface while taking advantage of PIM. Despite the orders of magnitude speedup compared to DRAM-based systems, TESSERACT generates excessive cross-cube communications through SerDes links, whose bandwidth is much less than the aggregated local bandwidth of HMCs. Our investigation indicates that this is because of the restricted data organization required by the vertex programming model.**

In this paper, we argue that a PIM-based graph processing system should take *data organization as a first-order design consideration*. Following this principle, we propose GRAPHP, a novel HMC-based software/hardware co-designed graph processing system that drastically reduces communication and energy consumption compared to TESSERACT. GRAPHP features three key techniques. *1) "Source-cut" partitioning*, which fundamentally changes the cross-cube communication from one remote put per cross-cube edge to one update per replica. *2) "Two-phase Vertex Program"*, a programming model designed for the "source-cut" partitioning with two operations: GenUpdate and ApplyUpdate. *3) Hierarchical communication and overlapping*, which further improves performance with unique opportunities offered by the proposed partitioning and programming model. We evaluate GRAPHP using a cycle accurate simulator with 5 real-world graphs and 4 algorithms. The results show that it provides on average 1.7 speedup and 89% energy saving compared to TESSERACT.

I. INTRODUCTION

Processing-In-Memory (PIM) is an effective technique that reduces data movements by integrating processing units within memory. While conceptually appealing, early works [1], [2] only achieved limited success due to both technology restrictions and lack of appropriate applications. However, with the recent advance of "big data" and 3D stacking technology, both problems seem to become solvable.

On the application side, modern big data applications operate on massive datasets with significant data movements, posing great challenges to conventional computer architecture. Among them, graph analytics [3], [4] in particular received intensive research interests, because graphs naturally capture relationships between data items and allow data analysts to draw valuable insights from the patterns in the data for a wide range of applications. However, graph processing poses great challenges to memory systems. It is well-known for the *poor locality* because of the random accesses in traversing the neighborhood vertices, and *high memory bandwidth requirement*, because the computations on data accesses from memory are typically simple.

On the technology side, 3D integration [5] enables stacking logic and memory chips together through TSV-based interconnection, which provides high bandwidth with scalability and energy-efficiency. One of the most prominent 3D-stacked memory technologies is Micron's Hybrid Memory Cube (HMC) [6], which consists of a logic die stacked with several DRAM dies. With this technology, it is possible to build a system that consists of multiple HMCs, which can provide *1)* high capacity of main memory that is large enough for in-memory big data processing; and, more importantly, *2) memory-capacity-proportional* bandwidth, which is essential for applications with poor locality and high memory bandwidth requirement.

As a result, due to the advances in both application and technology, the research community and industry again became increasingly interested in applying PIM to various applications like machine learning [7], natural language processing [8], [9], [10], social influence analysis [11], [12], [13] and many others [14], [15]. Among these kinds of applications, PIM (e.g., HMC) is specially suitable for building efficient architecture for graph processing frameworks.

TESSERACT [16] is a PIM-enabled parallel graph processing architecture. It implements a Pregel-like vertex-centric programming model [3] on top of the HMC architecture, so that users could develop programs in the familiar interface while taking advantage of PIM. The results show that TESSERACT can be orders of magnitude faster than DRAM-

[1]M. Zhang and Y. Zhuo equally contributed to this work.

[2]M. Zhang, Y. Wu, and K. Chen are with the Department of Computer Science and Technology, Graduate School at Shenzhen, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; M. Zhang is also with Sangfor Technologies Inc.

based in-memory graph processing systems.

Despite the promising results, TESSERACT generates *excessive cross-cube communications* through SerDes links, whose bandwidth is much less than the aggregated local bandwidth of HMCs. Such cross-cube communications delay the executions in memory cubes, and eventually affect HMC's internal bandwidth utilization. In fact, the results in [16] confirms this observation: the bandwidth utilization of TESSERACT is usually less than 40%. Moreover, TESSERACT adopts the Dragonfly topology to connect HMCs [6], which provides higher connectivity and shorter diameter than the simpler topology like mesh. However, Dragonfly is still not fully symmetric, which means that the bandwidth of certain critical cross-cube links may sustain much higher throughput than the others, becoming bottlenecks that further hampering TESSERACT's performance.

Our investigation shows that this problem is due to a missing consideration, — *data organization*, and the *suboptimal order* in considering different aspects of the system. To develop an efficient graph processing system, a careful co-design of both the software and hardware components of the systems is needed. Typically, we need to consider the following four issues: *1)* programming model, which effects the user programmability and algorithm expressiveness; *2)* runtime system, which maps programs to architecture; *3)* data organization, which determines the communication pattern; and *4)* architecture, which determines the efficiency of execution; In TESSERACT, *data organization aspect is not treated as a primary concern* and is subsequently determined by the presumed programming model.

Specifically, TESSERACT follows the "vertex program" programming model that first proposed by Pregel [3], where a *vertex function* is defined for all vertices. This vertex program takes the vertex's value as parameter and updates the outgoing neighbors, — the destinations of all outgoing edges (potentially in different ways). If a vertex and all its outgoing neighbors are in the same cube, the vertex function is executed locally. Otherwise, the cross-cube messages are incurred to remotely perform the reduce function. Let the vertex be v and its k outgoing neighbors are $\{u_1, u_2, ..., u_k\}$, in TESSERACT, for any outgoing neighbor u_i that is in a different cube than v, a `put` message is sent from v's cube to u_i's cube, containing a reduce function and u_i's value as the parameter. This message asks u_i's cube to perform the reduce as a *remote function*. We see that, *determined by vertex program model, each cross-cube edge incurs a cross-cube message*, and hence the amount of cross-cube communications is proportional to the number of cross-cube edges.

In order to reduce this number, Junwhan *et al.* [16] have tried to use METIS [17] to obtain a better partitioning for TESSERACT, but the result is not that promising. Only very small performance improvements are achieved for 3 out of 5 benchmarks tested; and the METIS-generated partitioning even leads to *worse* performance for one of the rest two benchmarks. Moreover, the complexity of METIS prohibits its application in real-world large graphs.

To resolve the issue in the conventional design flow, we argue that a PIM-based graph processing system should take *data organization as a first-order design consideration*. This principle is important because: *1)* data organization affects cross-cube communication, workload balance, and synchronization overhead, which directly translate into the energy consumption; *2)* if the programming model is decided first, this fixed programming model may prohibit users from using the optimal data partitioning method; *3)* co-designing data organization and interconnection structure can enable extra opportunities and benefits such as broadcasting and overlapping. Therefore, we propose a different order of design consideration: one should first choose the proper data organization with less communication, then design the programming model based on it, finally, the architecture and runtime optimizations could be applied to further improve performance.

Following the above design principle, we propose GRAPHP, a novel HMC-based software/hardware co-designed graph processing system that drastically reduces communication and energy consumption compared to TESSERACT. GRAPHP features three key techniques.

• **"Source-cut" Partitioning**. This algorithm ensures that a vertex and all its incoming edges are assigned in the same cube. As a result, if an edge *(u, v)* is assigned to cube i, all the incoming edges of vertex v will also be assigned to cube i. But, at the same time, the source vertex u of this edge may be assigned to other cubes. In such case, for an edge with the source vertex in a remote cube, the local cube maintains a *replica* of the source, which will be synchronized with the *master* in remote cube in each iteration. This mechanism fundamentally changes the cross-cube communication from one remote `put` per cross-cube edge to **one *update* per replica**. We show that it generates *strictly less* communication compared to TESSERACT. Moreover, source-cut is a heuristic-based algorithm in which the assignment of each edge can be processed independently. As a result, the partitioning overhead is much less than METIS [17].

• **"Two-phase Vertex Program"**, a programming model designed for the "source-cut" partitioning with two operations: `GenUpdate`, which generates the vertex value update based on all (local) incoming edges; and `ApplyUpdate`, which applies the update to each vertex. The replica synchronization is handled transparently by the software framework. This model slightly trade-offs the expressiveness for less communication. However, the real-world applications (e.g., pagerank) typically do not need the flexibility provided by the general vertex program. We believe this model is sufficiently expressive, in the worst case, it can be augmented to express more general vertex function (see Section III-B).

• **Hierarchical Communication and Overlapping**. The replica synchronization requires that the updates from master to replicas are the *same*. This property enables the hierarchical communication which avoids sending the same messages when possible, thus, reduces the communication amount in certain bottleneck links between cubes. Moreover, two-phase vertex program model naturally leads to an overlapping mechanism, which can further hide the latency of cross-cube communication.

According to our evaluation results, GRAPHP effectively reduces the communication amount by 35% - 98% and reaches 1.7x average, 3.9x maximum speedup and reduces 89% average, 96% maximum energy cost compared to TESSERACT.

II. BACKGROUND AND MOTIVATION

A. Hybrid Memory Cube

Recently, 3D integration technology [5] is available to enable Process-In-Memory (PIM) [18], we focus on Hybrid Memory Cube (HMC) [6], which is one of the most promising implementations. Nevertheless, other alternatives, such as JEDEC's High Bandwidth Memory specification [19], typically share similar principle as HMC, thus the proposed techniques should also apply to them.

An HMC device (i.e., a cube) is a single chip stack that consists of several memory dies/layers and a single logic die/layer. Two kinds of bandwidth are defined: *1) Internal* bandwidth, which caps the maximum data transfer speed between memory dies and the logic dies of a same cube; and *2) External* bandwidth, which is provided by a cube to external devices (e.g., other cubes and the host processor).

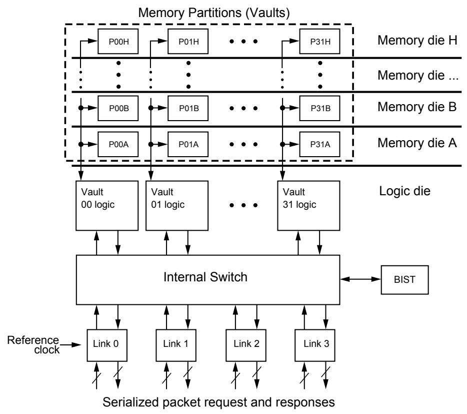


Figure 1. An Example Implementation of HMC.

Figure 1 depicts the architecture of a cube defined by Hybrid Memory Cube Specification 2.1 [6]. Each cube contains 32 vertical slices (called vaults), at most 4 multiple serial links as the off-chip interface, and a crossbar network that connects them. Each vault consists of a logic layer and several memory layers, which can provide up to 256 MB of memory space (i.e., 8 GB space per cube). These layers are connected through low-power Trough Silicon Via (TSV). Since each TSV can provide up to 10 GB/s of bandwidth, the maximum internal bandwidth of a cube is $32*10 = 320$ GB/s. In contrast, if the default configuration is used, each off-chip link will contain 16 input lanes and 16 output lanes for full duplex operation, which provide at most 480 GB/s external bandwidth (i.e., 120 GB/s per link).

Besides the capability of providing high density and bandwidth, HMC also makes it possible to integrate computation logics into its logical die/layer. In TESSERACT, a single-issue, in-order core and a prefetcher are placed in the logic die of each vault (i.e., 32 cores per cube). It is possible, because the area of 32 ARM Cortex-A5 processors including an FPU (0.68 mm2 for each core [20]) corresponds to only 9.6% of the area of an 8 Gb DRAM die area (e.g., 226 mm2 [21]). We use the same configuration in GRAPHP.

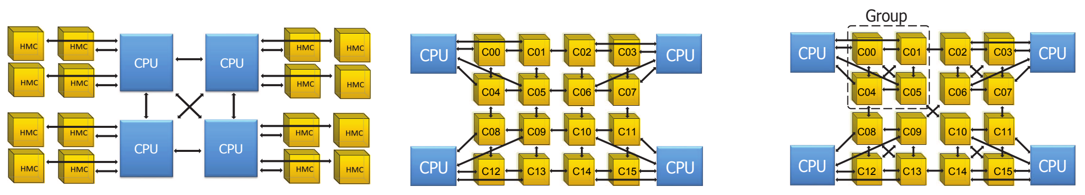
B. Interconnection

The key benefit that HMC can provide is *memory-capacity-proportional* bandwidth, which is achieved by using multiple HMCs. Typically, a system that contains N HMCs can provide $N * 8$ GB memory space and $N * 320$ GB/s aggregation internal bandwidth. However, this aggregated bandwidth depends on the interconnection network that connects these HMCs and host processors.

The straightforward design choice is "processor-centric network", which simply reuses the current NUMA architecture and replaces traditional DIMMs with HMCs. Figure 2 (a) presents a typical system that has four processor sockets. In this case, Intel QuickPath Interconnect (QPI) technology is used to built a fully-connected interconnection network among the processors, and each HMC is exclusively attached to a particular processor (i.e., there isn't a direct connection between HMCs). Although this network organization is simple and compatible with the current architecture, Kim *et al.* [22] concludes that this processor-centric organization does not fully utilize the additional opportunities offered by multiple HMCs.

Since the routing/switching capacity can be supported by HMC's logic die, it is possible to use more sophisticated topologies and connectivities that were infeasible with traditional DIMM-based DRAM modules. To take this opportunity, Kim *et al.* [22] proposes "memory-centric network", in which HMCs can directly connect to other HMCs and there is no direct connection between processors (i.e., all processor channels are connected to HMCs and not to any other processors). According to the evaluation, the throughput of a memory-centric network can exceed the throughput of a processor-centric network by up to $2.8\times$.

Moreover, Kim *et al.* [22] also evaluated various different kinds of topologies to interconnect HMCs. Two of the most prevalently used examples are presented in Figure 2 (b) and Figure 2 (c). Among different topologies, Dragonfly [23] is suggested as the favorable choice, because it *1)* has higher connectivity and shorter diameter than simple topology



(a) Processor-centric Network (b) Memory-centric Network: Mesh (c) Memory-centric Network: Dragonfly

Figure 2. Examples of HMCs' Interconnection.

like Mesh; *2)* achieves a similar performance as the best interconnection topology, named flattened butterfly [24], in their evaluation of 16 HMCs; and *3)* does not face the same scalability problem as flattened butterfly.

In this paper, we will use the memory-centric network and Dragonfly topology as suggested and used by previous works [22]. However, the techniques proposed are not tightly coupled with this particular architecture.

C. Bottleneck

Based on the HMC implementation discussed in Section II-A, the maximum external bandwidth of a cube ($480\ GB/s$) is actually larger than its internal bandwidth ($320\ GB/s$). However, due to the limitation on the number of pins, this external bandwidth does not scale with the number of HMCs. Thus, the aggregation internal bandwidth of a real system will largely surpass the available external bandwidth.

Take the Dragonfly topology shown in Figure 2 (c) as an example, it presents a typical HMC-based PIM system that contains 16 HMCs. As we can see, since at most 4 off-chip links are provided by a cube, it is impossible to achieve a full-connection between the cubes. To be realizable, Dragonfly splits the total 16 HMCs into 4 groups and only achieves the full connection within each group. In contrast, only one link is provided for each pair of groups. As a result, the bandwidth that caps cross-group communication is bounded by the bandwidth of a link, which is only $120\ GB/s$. As a comparison, the aggregation internal bandwidth of the entire PIM system is $16 * 320\ GB/s = 5.12\ TB/s$. It is why data organization is extremely important for a HMC-based PIM system and should be taken as the first-order consideration. In TESSERACT, the simple partitioning strategy leads to excessive cross-cube communications, which prohibits the applications from fully utilizing the aggregation internal bandwidth of HMC.

It is also notable that the load of different external links are *not* equal. For example, if we assume that the amount of communication is equal for each pair of two HMCs, each of the cross-group link in the Dragonfly topology will need to serve $4 * 4 = 16$ pairs of HMCs communication (e.g., 4 HMCs in each group). As a comparison, the link between HMC C0 and HMC C1 only serves the communication between *(1)* HMC C0 and C1 (1 pair) and *(2)* HMC C0 and the top-right HMC group (C2, C3, C6, C7) (4 pairs), which is less than 1/3 of the 16 pairs formerly calculated. This implies that the links across groups can easily become the bottleneck and should be particularly optimized.

Essentially, these bottleneck is rooted from the fact that only limited external links are provided by each HMC, which means that they cannot be simply avoided by using other topologies. As an illustration, Figure 2 (b) presents the Mesh topology. In this case, there are only four links between HMC group (C0$\sim$C7) and HMC group (C8$\sim$C15), so that each of them need to server $8 * 8/4 = 16$ pairs of HMCs' communication, which is the same as the bottleneck of the Dragonfly topology. Even worse, the number of these bottleneck links is 8 in Mesh and only 6 in Dragonfly.

D. PIM-Based Accelerator

The current 3D-stacking based PIM technologies offer great opportunities for graph analytics because: *1)* 3D-stacking provides high density, which opens up the possibility of in-memory graph processing; *2)* the memory-capacity-proportional bandwidth is ideal for graph processing applications that lack temporal locality but require high memory bandwidth; *3)* various programming abstractions have been proposed for graph processing to improve programmability, for PIM-based accelerators, they can be naturally used to hide architectural details.

```
 1 count = 0;
 2 do {
 3   ...
 4   list_for (v: graph.vertices) {
 5     value = 0.85 * v.pagerank / v.out_degree;
 6     list_for (w: v.successors) {
 7       arg = (w, value);
 8       put(w.id, function(w, value) {
 9           w.next_pagerank += value;
10         }, &arg, sizeof(arg), &w.next_pagerank);
11     }
12   }
13   barrier();
14   ...
15 } while (diff > e && ++count < max_iteration);
```

Figure 3. Pseudocode of PageRank in Tesseract.

TESSERACT is a 16-HMC system using Dragonfly interconnection in Figure 2 (c). It provides users with low level APIs which can conveniently be composed to a programming model that similar to Pregel's vertex program. Figure 3 shows the PageRank computation using TESSERACT's programming interface, where the main procedure is a simple two-level nested loop (i.e., ling 5 $\sim$ line 13). The outer loop iterates on all vertices in the graph. For each vertex, the program iterates on all its outgoing edges/neighbors in the inner loop 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* prefethc_addr)`. It executes a remote function call `func` with argument `arg` on the `id`-th HMC.

Specifically, for every vertex, the program first calculates the proper pagerank division based on the pagerank sent to the vertex and out degree, the result is stored in `value` (line 6). Then, a user-defined vertex function is called for every outgoing edge to add `value` to the corresponding destination vertex's pagerank for the next iteration (`w.next_pagerank`) (line 10). This function is executed asynchronously and cross-cube communication is incurred when the outgoing neighbor is in a different cube. Finally, a `barrier` is applied to ensure that all the updates performed by vertex functions in the current iteration have been completed. It is easy to see that this API is equivalent to Pregel's [3] vertex program, which assures the programmability of TESSERACT. For the cross-cube remote function calls, blocking will lead to unacceptable latency, therefore, TESSERACT implements them in a non-blocking manner. A cube could also combine several remote functions together to reduce the performance impact due to interrupts on receiver cores.

Nevertheless, the optimization techniques in TESSERACT are only used to hide cross-cube communication latency. None of them can *reduce* the amount of cross-cube communication. Essentially, it is due to the inefficiency of TESSERACT's simple graph partitioning, which is constrained by the vertex program model. Specifically, only edge-cut (i.e., the graph is partitioned in vertex granularity and a vertex can only be assigned to one cube) can be used. The results show that even the sophisticated METIS partitioner [17] cannot improve performance much (in one case, even make it worse). As another consequence, the bandwidth utilization of TESSERACT is usually less than 40%.

III. GRAPHP ARCHITECTURE

In this section, we describe GRAPHP, a software/hardware co-designed HMC-based architecture for graph processing. First, we propose a new graph partitioning algorithm that would drastically reduce cross-cube communication. Then, a programming model is designed to match the partitioning method. Finally, we discuss the optimization opportunities offered by our approach, optimized broadcast and overlapping, to further improve the performance.

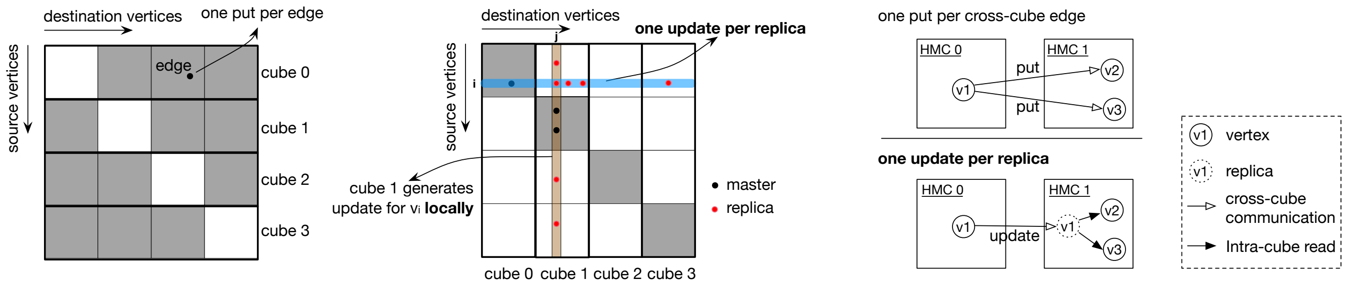
A. Source-Cut Partitioning

Let us start with a detailed understanding of the graph partition in TESSERACT through a matrix view. Consider Figure 4 (a), A graph can be considered as a matrix, where the rows and columns are corresponding to the source and destination vertices. In TESSERACT, a graph is partitioned among cubes, — each cube is assigned with a set of vertices (i.e., vertex-centric partition), corresponding to a set of rows. The edges are the non-zero elements in the matrix, denoted as black dots. With the graph partitioned, the matrix could be cut into grids, each of which contains edges from vertices in cube i to cube j. It is similar to the concept in GridGraph [25]. With N cubes, the whole matrix is divided into N^2 grids. The grids on the diagonal contain the local edges, whose source and destination vertex are in the same cube. As discussed earlier, each non-local edge incurs a cross-cube communication in TESSERACT. They are essentially the edges in the grey grids. Assume that edges distribute in the graph uniformly, the amount of cross-cube communication in one iteration is $O(N(N-1)\frac{|E|}{N^2}) = O(\frac{(N-1)}{N}|E|)$. We can see it is roughly the number of edges in the graph.

Next, we propose *source-cut*, in which a graph is partitioned such that, when a vertex (e.g., v_j) is assigned to a cube (e.g., cube 1), *all the incoming edges of v_j are also assigned to the same cube*. The idea is shown in Figure 4 (b). Different from TESSERACT, the matrix is cut *vertically*, — each cube is assigned with a set of columns, not rows. To perform the essential operations in graph algorithm, — propagating the value of the source vertex through an edge to the destination, a *replica* (denoted as red ●) is generated if a cube only holds the edge and its destination vertex. The *masters* (denoted as black ●) are the vertices in a cube that serve as the destination. With this data organization, the column of v_j corresponds to v_j's all incoming edges and neighbors, therefore, v_j's update can be computed *locally*. The sources of edges in a column can be masters (black ●) or replicas (●). Similar to earlier discussion, after the matrix is divided into grids, the ones on the diagonal represent the edges in a cube where both their source and destination vertex are masters.

The communication in source-cut is caused by *replica synchronization*, in which the value of master vertex is used to update the replicas in all other cubes. In the matrix view, it means that each master vertex in the diagonal grids updates its replicas in other cubes in the same row. In Figure 4 (b), consider the master vertex v_i in cube 0. In replica synchronization, cube 0 needs to send v_i's value to both cube 1 and cube 3, but not cube 2. Because cube 2 does not have any edge from v_i. Note that *only one* message is sent from cube 0 to cube 1, even if there are *three edges* from v_i to different vertices in cube 1. This is the key property why source-cut generates *strictly less communication* compared to vertex-centric partition: in the same case, it will incur three messages from cube 1 to cube 2 (refer to Figure 4 (a)). This property informally proves that: with the *same* master-to-cube assignment, source-cut always generates less or equal amount of communication compared to vertex-centric partition.

In essence, source-cut generates **one update per replica** while the graph partition for vertex program would incur **one put per cross-cube edge**. It is illustrated in Figure 4 (c) in a graph view. Then, we can calculate the commu-



(a) Graph Partition for Vertex Program (b) Source–Cut in Matrix View (c) Source–Cut in Graph View

Figure 4. Graph Partitioning for Vertex Program (a) and Source-Cut (b)(c).

nication amount of source-cut. We define the *replication factor*, λ, which includes *both* master and replicas. Then, the communication amount due to replica synchronization is $O(N(\lambda-1)\frac{|V|}{N}) = O((\lambda-1)|V|)$. This is an estimation as it assumes that each cube contains similar number of vertices. The maximum value of λ is $(N-1)$, therefore, the maximum communication cost is $O((N-1)|V|)$. Comparing it with the earlier calculated vertex-centric partition estimation $O(N(N-1)\frac{|E|}{N^2}) = O(\frac{(N-1)}{N}|E|)$, we see that from the equations the communication amount of source-cut is *not* strictly less than vertex-centric partition. We show that there is no contradiction as follows.

For source-cut, to reach the maximum communication $O((N-1)|V|)$, at least $N^2(N-1)$ edges are needed. In particular, they are all in non-diagonal grids. For example, assume $N=4, |V|=16$, and each cube contains 4 vertices, we need at least 48 edges in the white grids in Figure 4 (b). Specifically, each grid contains 4 edges: from the 4 vertices corresponding to each row to any master vertex in the column. In this way, 3 replica synchronizations are needed for each row, with 16 rows, in total 48 cross-cube communications. It is easy to see that, in vertex-centric partition, the same amount of communication is incurred as well, because the source and destination of each of the 48 edge are not in the same cube. However, if we put $|E| = 48, N = 4$ to $O(\frac{(N-1)}{N}|E|)$, we would get only 36 communications. It is because the equation assumes that the edges are uniformly distributed in all cubes, which is not true in this case. Overall, this is an example that for certain graph, source-cut can incur *at most the same amount* of communication as vertex-centric partition.

The implementation of source-cut is much simpler than the complex algorithm used in METIS [17]. One simple implementation of source-cut is to define a hash function `hash(v)` and assign an edge (u,v) to `hash(v)`$\%N$, where N is the number of HMCs (i.e., 16 in our system). Note that, although source-cut ensures that all the incoming edges of a vertex are assigned to the same HMC, it does not provide any guarantee on the outgoing edges. As a result, if an edge (u,v) is assigned to HMC i, all the edges with form $(*,v)$ will also be assigned to HMC i, but some or even all of the edges with form $(u,\ *)$ are not assigned to HMC i. In that case, we need to set up a replica of vertex u in HMC i to store the newest value of vertex u.

B. Two-Phase Vertex Program

Based on source-cut, we propose a new programming model named "Two-Phase Vertex Program", in which the unit of data processing is the incoming edges in source-cut. As discussed in Section I, the programming model and data organization can interact with each other, therefore, a co-design is required. Our "Two-phase Vertex Program" splits a vertex program into two phases: *1)* **Generate** phase, where all the incoming edges of a vertex and their corresponding sources are read and used to generate an update for their shared destination vertex; and *2)* **Apply** phase, where the update is applied to the corresponding vertex's every replica. Our new model is designed for source-cut. First, since each vertex and all its incoming edges are in the same cube, the Generate phase could be performed locally. Second, the communication only happens before the Apply phase, which provides "one update per replica", instead of "one put per cross-cube edge" in vertex-centric partition in TESSERACT.

Figure 5 shows a PageRank implementation that *1)* uses the same set of APIs as TESSERACT; *2)* is equivalent to the implementation described in Figure 3; but *3)* is programmed in "Two-phase Vertex Program" model. As we can see, the first loop iterates on all the replicas to calculate the proper share given to each edge (by dividing new pagerank with the outgoing degree of the corresponding vertex). In the next two-level nested loop, the outer loop iterates on every vertex. Then, for every vertex v, the program first iterates all its incoming edges to calculate the new pagerank and then broadcasts this new value to all its replicas. Due to source-cut partition, all the computations during incoming-edge iterations occur locally and hence do not incur any communication.

While it is possible to express the operations of "Two-phase Vertex Program" with TESSERACT's API, it is tedious and the new model requires a number of internal data structures that TESSERACT does not provide (e.g., the replica list). Therefore, we propose our own APIs as the higher level abstraction to enhance programmability. As shown in Figure 6, users of GRAPHP only need to write two functions, `GenUpdate` and `ApplyUpdate`, and all the other chores, e.g., replica synchronization will be handled by our system.

```
list_for (r: graph.all_replicas) {
  r.pagerank = r.next_pagerank;
  r.value = 0.85 * r.pagerank / r.out_degree;
}
list_for (v: graph.vertics) {
  update = 0;
  list_for (e: v.incoming_edges) {
    update += e.source.value;
  }
  list_for (r: v.replicas) {
    put(r.id, function(r, arg) {
         r.next_pagerank = arg
      }, &update, sizeof(update), &e.
           next_pagerank)
  }
}
barrier();
```

Figure 5. PageRank in Two-Phase Vertex Program.

Specifically, the input of GenUpdate function is the incoming edges of a specific vertex and the output is the corresponding update. In contrast, the input of ApplyUpdate function is the vertex property and the update generated in this iteration. It does not have output. In each iteration, GenUpdate function will be executed on every *vertex* once and ApplyUpdate will be executed on every *replica* once. One should note that both GenUpdate and ApplyUpdate can be executed locally. The replica synchronization (i.e., the broadcast of update to replicas) is transparently handled by our software framework. In other words, the communication pattern of our system is *fixed*. As we will see later in Section III-C and Section III-D, this higher-level abstraction not only ensures programmability but also provides the flexibility to apply additional optimizations. Due to the fixed communication pattern, it is possible to further optimize the architecture to reduce cross-cube communication on the bottleneck links.

```
GenUpdate(incoming_edges) {
  update = 0;
  list_for (e: incoming_edges) {
    update += e.source.value;
  }
  return update;
}

ApplyUpdate(v, update) {
  v.pagerank = update;
  v.value = 0.85 * v.pagerank / v.out_degree;
}
```

Figure 6. Two-Phase Vertex Program.

To illustrate the effectiveness of source-cut and "Two-Phase Vertex Program" model. Table I compares the amount of cross-cube communication on three real-world graphs. For every graph, we have tried three partitionings: *1) Random*, which randomly assigns a vertex to an HMC; *2) METIS*, which takes advantage of the advanced partitioning application METIS [17]; *3) Source-cut*, which randomly assigns a vertex and all its incoming edges to a cube. The first two are vertex-centric partitions, which can be used in TESSERACT. We report both the average and the maximum amount of cross-cube communication for every case. We see that, when *Random* is used, the skewness among all the $16 * 16 = 256$ pairs of cross-cube communication is not large. In contrast, although the advanced partitioner *METIS* can largely reduce the average amount of cross-cube communication, it usually leads to excessive skewness (i.e., a large difference between maximum and average communication). As a result, the maximum amount of cross-cube communication produced by *METIS* is sometimes much higher than *Random*. This observation explains the reason why in TESSERACT's evaluation METIS does not improve the performance as expected. Moreover, the cost of using METIS is huge: it not only takes long time but also consumes large amount of memory. As we note, the results of partitioning Twitter with *METIS* are not given in the table. This is because the METIS program failed for out of memory even when we use a machine with 1 TB memory.

For *Source-cut*, we assume that the argument size needed for the remote function call is the same as the data size of update generated by GenUpdate. From the results, we see that *Source-cut* incurs only 18.8% to 39.9% of communication compared with *Random*. Compared with *METIS*, *Source-cut* incurs 55.9% communication on Orkut, but it increases the communication on Livejournal graph by 54.4%. Note that it means *source-cut* must have a different vertex-to-cube assignment than *METIS*, because otherwise source-cut can be proven to generate less cross-cube communication (see Section III-A). However, *Source-cut* has much smaller maximum cross-cube communication: 68.4% and 92.6% reduction compared to *Random* and *METIS* on average, respectively. This leads to more balanced execution. More importantly, the partitioning cost of *METIS* is much higher than *Source-cut*.

Expressiveness of Two-Phase Vertex Program. Before proposing further architecture optimizations, we compare the expressiveness of the general vertex program and the proposed Two-Phase Vertex Program. In Figure 7, consider three vertices: $\{v_1, v_2\} \in$ HMC 0, $\{v_3\} \in$ HMC 1; and two edges: (v_1, v_3) and (v_2, v_3). In Two-Phase Vertex Program, there are replicas of v_1 and v_2 in HMC 1, v_3's GenUpdate can generate the update based on all v_3's incoming edges/neighbors. The restriction in Two-phase Vertex Program model is that the GenUpdate has to perform the same operation (e.g., defined as f$(v_1, v_2, ...)$ for all incoming edges/neighbors. In contrast, the general vertex program semantically allows performing different operations for each edge. For example, f1(v_1) and f2(v_2) and then v_3 could reduce the two results and apply. However, real-world applications (e.g., pagerank) do not need such flexibility. In fact, the extra flexibility may do more harm than good, — it may lead to many duplications (e.g., same remote function is sent for all outgoing neighbors) that is hard to be automatically removed. In contrast, Two-Phase Vertex Program inherently avoids these duplications.

We believe our model is sufficiently expressive. Moreover, it is possible to express the general vertex program with certain changes to the proposed model. Specifically, the GenUpdate function can concatenate the list of incoming edges/neighbors, then in ApplyUpdate function,

	Random		METIS		Source-cut	
	Average	Maximum	Average	Maximum	Average	Maximum
Orkut	457,754	470,959	187,532	843,270	107,206	109,706
LiveJournal	269,506	289,519	79,224	352,341	107,789	115,594
Twitter	5735801	6374486	failed	failed	1079390	1105802

Table I
COMPARISON OF CROSS-CUBE COMMUNICATION.

different functions could be applied to different incoming edges/neighbors. This change could perform the same computations of general vertex program in Figure 7. The cost is more complex function parameters and more memory space. Overall, we believe our Two-Phase Vertex Program provides an efficient mechanism to remove the redundant information that is not required in most applications. We argue that our approach is a sweet spot that balance the trade-off between generalizability and communication/performance.

As discussed in Section II-B, since each HMC provide only 4 links, it is impossible to achieve a full connection between 16 HMCs. Due to the interconnection topology, certain links cross groups serve much more cross-cube communication and could become the bottlenecks. Specifically, the bottleneck links of Dragonfly are the links between every pair of HMC groups. Each of them needs to serve 16 pairs of HMCs communication. As an illustration, Figure 9 presents: *1)* the average load of inner-group links (e.g., link between C0 and C1); and *2)* the average load of cross-group links (e.g., link between C5 and C10) for different graphs in GRAPHP. We see that, although source-cut can significantly reduce the communication load, it does not change the fact that cross-group links serve much more communication than inner-group links. As a result, these cross-group links usually become the bottleneck and may potentially hinder the performance and energy consumption.

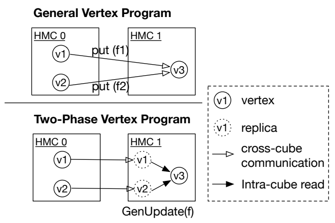


Figure 7. Expressiveness of Two-Phase Vertex Program.

C. Hierarchical Communication

In order to mitigate the imbalance, we develop the hierarchical communication mechanism to remove the redundancies in simple point-to-point communications. Please note that it is an additional opportunity enabled by source-cut and Two-phase Vertex Program, where it is guaranteed that the update sent to replicas of the same vertex must be the *same*. As a result, it is possible to sent only one copy of the update for every HMC group, in contrast of one copy per HMC.

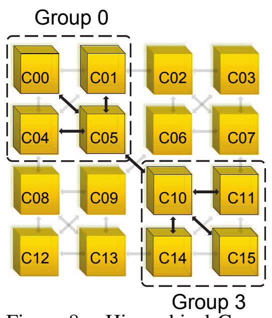


Figure 8. Hierarchical Communication

Specifically, for every pair of HMC groups in the Dragonfly topology, we will build a broadcasting tree as illustration by Figure 8. For a group, one HMC is selected as the broker (e.g., HMC C5 and HMC C10 for group 0 and group 3 in the example) which is responsible for *1)* gathering the needed update from its local group; *2)* sending them to the broker of the other group; and *3)* scattering the update inside a group as needed. As a simple example, suppose hash(v_1)%16 $= 0$, the master of v_1 is at HMC C0, and C10, C11, and C14 contain v_1's replicas. In the conventional point-to-point, the links between C5 and C10 need transfer the duplicated updates three times. With hierarchical communication, only one copy of the update value is transferred with our broadcasting tree.

D. Overlapped Execution

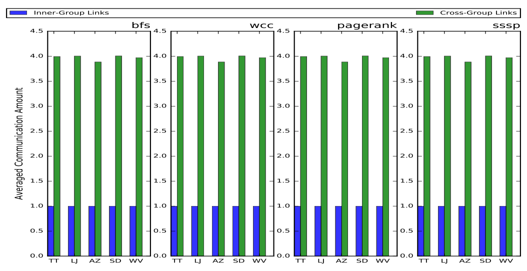


Figure 9. Averaged Communication. Inner Group Links v.s. Cross Group Links

In an architecture with multiple HMCs, it is important to hide the remote access latency of a cross-cube communication. To support this, TESSERACT uses non-blocking communication, so that a local core can continue its execution without waiting for the results. Moreover, since the execution of non-blocking remote calls can be delayed, it is possible to batch several such functions together so that the core in the receiver cube is only interrupted once. Otherwise, the performance will be significantly impacted by the frequent context switches.

In GRAPHP, one option is to perform replica synchronization at the end of each iteration, but it is also possible to overlap communications due to replica updates with the current execution. The insights are the same as TESSERACT: when a vertex finished the execution of GenUpdate, the cube contains the master could start performing the broadcast of the update immediately. The updates could be saved in the local message queues and get processed in ApplyUpdate.

The replica updates could be stored in the receiver cube's message queue. When GenUpdate of all master vertices are finished, a cube could process the earlier received replica updates in batch. Therefore, only the executions due to updates sent toward the end of an iteration may appear in the critical path of execution.

While conceptually simple, one caveat needs to be considered to support the overlapped execution in GraphP. Referring to Figure 4 (b), each cube has to follow the *"column-major order"* so that the complete updates can be generated continuously. It is because the whole column has to be accessed to generate the update for a master vertex. If *"row-major order"* is followed, we only generate the "partial updates" for all master vertices before all edges are processed. It means that the cube cannot send any updates until the end of iteration, thus, the overlapped execution does not apply.

On the other side, the choices of different orders may affect performance due to locality. The *"column-major order"* optimizes the write locality, since all edges in the same column incurs writes to the same vertex. However, it may incur some non-sequential reads of the source vertices. The *"row-major order"* optimizes the read locality, since the edges from the same vertex are processed together. It is interesting to investigate the interactions between locality and execution overlapping. Our evaluation results in Section V-A provide more insights to this trade-off.

To further improve the effectiveness of overlapping, sorting the vertices by their incoming degree may be beneficial. This means that the GenUpdate of the vertex is executed with the smallest incoming degree first. This simple optimization can make best use of the bandwidth. Due to text limit, we will not go detail on the communication/execution overlapping technique, and we leave it as an interesting alternative that can be explored in the design space of GraphP.

IV. Evaluation Methodology

Simulation Configuration We simulate the HMC architecture by building an HMC simulator called hmc-zSim. Specifically, we integrate an HMC interconnect component into zSim [26], a fast and scalable simulator designed for x86-64 multicores. We configure the simulator to have 16 HMC Cubes with each cube containing 32 in order cores. Each core is configured to equip with a 64K, 4-way Set-associative L1-D cache and 32K, 4-way Set-Associative L1-I cache. We use 64 Byte cache line and 1000 MHz as simulation frequency. The results are validated with [27].

Datasets Table II shows the graph dataset we use to evaluate GraphP. For each graph, we also show the replication factor λ. These data sets are retrieved from the Stanford Large Network Dataset Collection (SNAP) [28] and are representative of modern medium to large graphs, including social networks from Twitter, Slashdot, and LiveJournal as well as networks collected from other types of user activities, like voting and co-purchasing.

Applications We implement four widely used applications based on the two-phase vertex programming model, including BFS(Breadth First Search), PageRank(An algorithm to rank website pages), SSSP(Single Source Shortest Path) and WCC(Weakly Connected Component).

Performance Evaluation Methodology In order to evaluate the performance of GraphP, we run all the applications against the graph dataset on top of hmc-zsim. For each run, we get the cycle numbers of each core and the largest cycle number among all cores is deemed as the cycle spent for the entire run.

Energy Evaluation Methodology The energy consumption of the HMC interconnect is estimated as two parts. a) The dynamic consumption, which is proportional to the number of flit transfer events that happen between each pair of routers. 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 hmc-zsim to count the number of transfer events and turn to the router power/area modeling tool ORION 3.0 [33] to estimate unit energy cost, i.e., the power, of the routers and associating links in the interconnect. We also refer to McPAT [34] and [35] to double check the power results to ensure its correctness and reliability.

V. Evaluation

A. Performance

As a first part of our evaluation, we first present the overall performance of our system and compare it with the Tesseract-like system. Figure 10 and Figure 11 show our results, which includes the speedup of source-cut algorithm with and without the use of innner-group broadcasting (i.e, hierarchical communication) and overlapping optimization compared to the baseline.

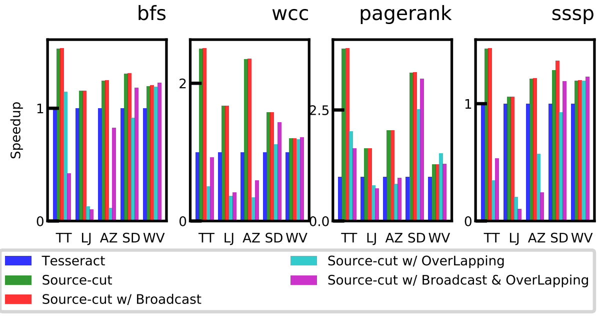


Figure 10. Performance of GraphP: 2DMesh

Our results show that GraphP (source-cut only) outperforms the Tesseract-like system by an 1.7x on average and 3.9x in maximum across all applications in our experiment. We note that the inner-group broadcast optimization

Graphs	#Vertices	#Edges	λ	Note
Wiki-Vote (WV)	7.1K	104K	2.96	Wikipedia who-votes-on-whom network [29]
ego-Twitter (TT)	81K	2.4M	3.79	Social circles from Twitter; Edges from all egonets combined. [30]
Soc-Slashdot0902 (SD)	82K	0.95M	4.40	Slashdot social network from February 2009 [31]
Amazon0302 (AZ)	262K	1.2M	2.68	Amazon product co-purchasing network from March 2 2003 [32]
LiveJournal (LJ)	4.8M	69M	4.18	LiveJournal online social network [31]

Table II
GRAPH DATASET

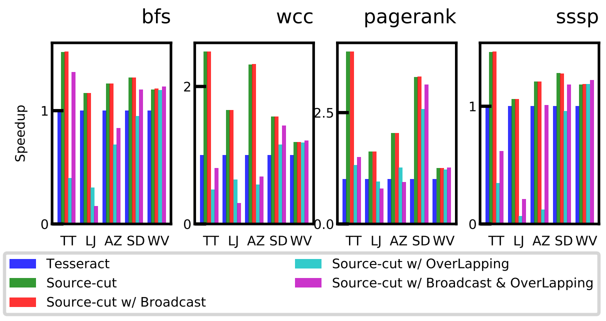


Figure 11. Performance of GRAPHP: Dragonfly

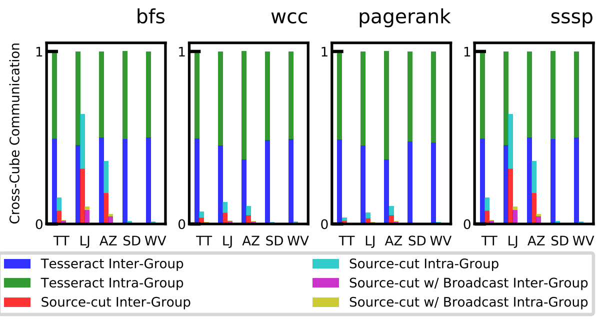


Figure 12. Cross-Cube Communication: 2DMesh

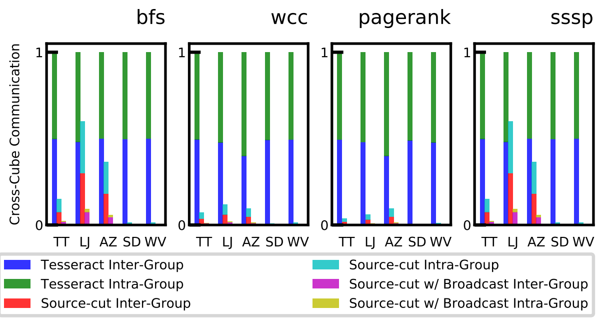


Figure 13. Cross-Cube Communication: Dragonfly

only contributes to marginal performance improvement compared with source-cut only. This is because communication is not bottleneck for GRAPHP and the reduced communication amount will not affect performance.

The results of overlapping are somewhat disappointing but interesting: in only one case (i.e., pagerank on WV), it achieves best performance, in a few cases (e.g., bfs/wcc/sssp on WV), it achieves similar (but a little worse) speedups compared to source-cut only (or with broadcast). Apparently, it is due to the trade-off we discussed in Section III-D, the different order of vertex/edge traversal leads to worse locality that eventually hurts performance. We also see that used with overlapping, broadcast always improves the performance, but still not as good as without overlapping.

B. Cross Cube Communication

Figure 12 and Figure 13 show cross-cube communication amount of GRAPHP normalized to the TESSERACT baseline. As we can see, source-cut only reduces 35% - 98% cross cube communications across all applications. With the broadcast optimization, cross cube communication is further reduced roughly 5/6 compared to the source-cut only setting, which causes more than 90% reduction compared to the TESSERACT baseline. We can also see that inter-group and inner-group communications each accounts for approximately half of total cross cube communications. However, given the fact that there are far fewer inter-group links compared to inner-cube links, the inter-group links are still over-loaded and become the potential bottleneck as reflected in Figure 9.

C. Bandwidth Utilization

As we have mentioned in Section III-C, the main reason of our speedup is the reduction on cross-cube communication, especially the reduction on bottleneck links. To further validate our argument, in this section, we present the data on measured bandwidth utilization. Specifically, we evaluate the utilization on *1)* aggregated internal bandwidth; *2)* normal inner-group cross-cube link; and *3)* corss-group bottleneck links. The first item is measured by calculating the division between the total amount of local HMC read/write and the total execution time. In contrast, the bandwidth on links is calculated by the amount of communication divided by time.

Figure 14 and Figure 15 show the result of our evaluation normalized to the TESSERACT baseline. As we can see from these figures, the aggregated internal bandwidth of each HMC of GRAPHP increases by from 1.1x to 46x. Especially, for graph inputs TT, LJ and AZ, the internal bandwidth utilization has increased by at least 5x. The inner-group bandwidth and cross-group bandwidth are both as expected: the source-cut algorithm enables an averaged of 81% as well as maximum of 98% reduction in inner-group link bandwidth. As for cross-group link bandwidth, the source-cut algorithm also reaches 80%-average and 98%-maximum utilization reduction. Inner-group broadcasting further reduces the inner-group and inter-group bandwidth utilization to a marginal amount compared to the baseline.

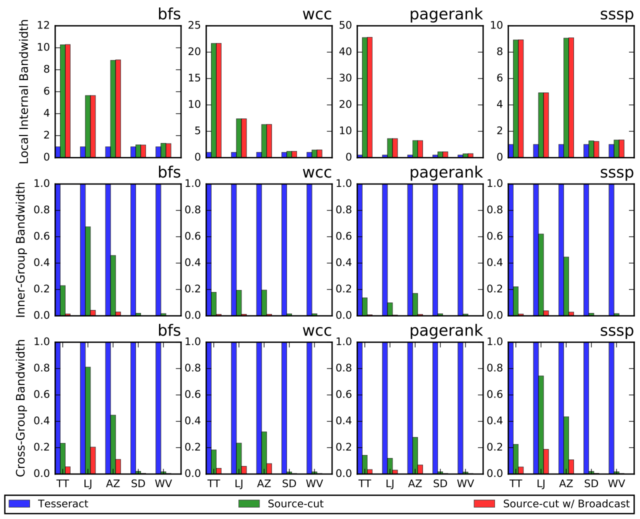


Figure 14. Bandwidth utilization: 2DMesh

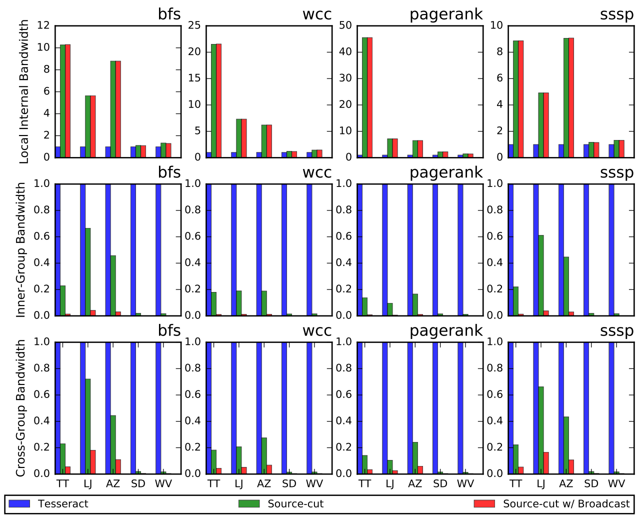


Figure 15. Bandwidth utilization: Dragonfly

D. *Scalability*

Figure 16 and 17 evaluate the scalability of GRAPHP by measuring the performance of 1/4/16-HMCs systems (i.e., 32/128/512 cores), normalized to the performance of the 1-HMC GRAPHP system. For all the applications, graph inputs TT, LJ, AZ exhibit 2x-6x improvement for 4-HMCs and 2.5x-30x improvement for 16-HMCs. However, SD and WV do not scale out as good as the other three graph inputs. The actual reason for such behavior is still unclear. We conjecture that it may be due to the these graphs' inherent structures themselves that hinder the computation from scale out. This behavior suggests that graphs may need to be classified properly so that each class of graphs can squeeze the HMC resources. We leave the detailed investigation of this issue as future work.

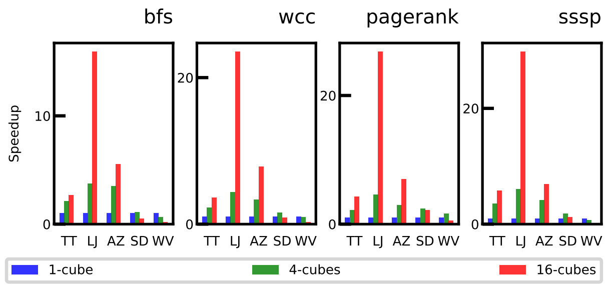


Figure 16. Scalability of GRAPHP - 2DMesh

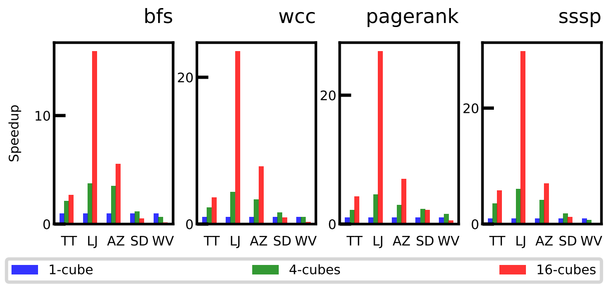


Figure 17. Scalability of GRAPHP - DragonFly

E. *Energy/Power Consumption Analysis*

Figure 18 and Figure 19 demonstrate the energy consumption of GRAPHP. Out of the four applications, source-cut only setting of GRAPHP reduces energy cost by 30% to 95% while source-cut with inner-group broadcasting further reduces energy cost for all the applications. For some graph inputs like TT, LJ and AZ, the broadcasting optimization accounts for more than 50% more energy reduction compared with the naive source-cut algorithm.

F. *Memory Overhead*

Since GRAPHP replicates only vertices not edges, it only increases the consumed memory from $|V| * (vertexsize) + |E| * (edgesize)$ to $\lambda * |V| * (vertexsize) + |E| * (edgesize)$. In a typical case that each vertex's size is 4 bytes and each edge's size is 8 bytes, our technique only leads to $(\lambda - 1) * 4 * |V|/(4|V| + 8|E|)$ additional memory. With this formula, the calculated overhead for datasets [WV, TT, SD, AZ, LJ] are only [6.47%, 4.63%, 14.1%, 16.5%, 10.7%]. The average overhead is only 10.4% and the largest is also smaller than 20%.

VI. RELATED WORK

Graph Processing Accelerators Mustafa *et al.* [36] proposed an accelerator for asynchronous graph processing, which features efficient hardware scheduling and dependence tracking. To use the system, programmers have to understand its architecture and modify existing code. Graphicionado [37] is a customized graph accelerator designed for high performance and energy efficiency, based on off-chip DRAM and on-chip eDRAM instead of PIM. Graphicionado

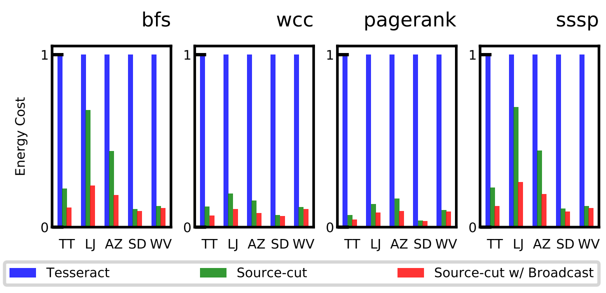


Figure 18. Energy Consumption of GRAPHP: 2DMesh

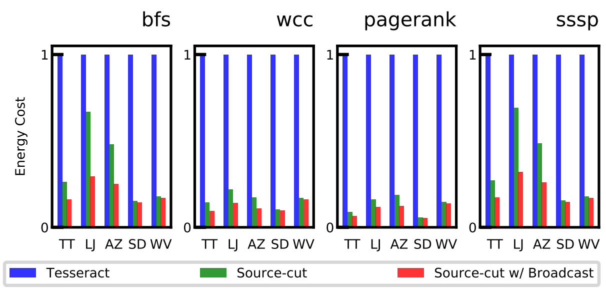


Figure 19. Energy Consumption of GRAPHP: Dragonfly

uses specialized memory subsystem for higher bandwidth. GraphPIM [38] demonstrates the performance benefits for graph applications by using PIM for the atomic operations. However, it does not focus on the inter-cube communications.

Large-Scale Graph Processing system There are many distributed graph processing systems proposed by researchers. Pregel [39] is the first distributed graph processing system and proposes a vertex-centric programming model, which is later inherited by many other graph processing systems including TESSERACT [25], [40], [41], [42], [43], [44], [45]. However, due to the problem of vertex-centric programming model (e.g., the enforcing of 1D partitioning), many new kinds of partitioning algorithm and the corresponding programming models (e.g., GAS proposed by PowerGraph [40] and hybrid-cut proposed by PowerLyra [46]). Certain parts of GRAPHP are inspired by these works, such as selective scheduling, removing the short sight of a vertex [47], but we adapt them into a PIM architecture and propose many more architecture-specific optimizations (e.g., broadcasting, overlapping).

Besides distributed graph processing, there are also many out-of-core graph processing systems that use disks. The key principle of such systems is to keep only a small portion of active graph data in memory and spill the remainder to disks. Although it is reported that these works can sometimes comparable with distributed systems that have hundreds of cores, it is also a well-known fact that all these works are bounded by the bandwidth of disks. As a result, all these works focus on how to enlarge the locality of disk I/O. In contrast, 3D stacking technologies provide high density, which opens up the possibility of in-memory big-data processing. Thus, the most significant problem is changed from increasing locality to reducing cross-cube communication in our work.

VII. CONCLUSION

This paper proposes GRAPHP, a novel HMC-based software/hardware co-designed graph processing system that drastically reduces communication and energy consumption compared to TESSERACT. GRAPHP features three key techniques. *1) "Source-cut" partitioning*, which fundamentally changes the cross-cube communication from one remote put per cross-cube edge to one update per replica. *2) "Two-phase Vertex Program"*, a programming model designed for the "source-cut" partitioning with two operations: GenUpdate and ApplyUpdate. *3) Hierarchical communication and overlapping*, which further improves performance with unique opportunities offered by the proposed partitioning and programming model. We evaluate GRAPHP using a cycle accurate simulator with 5 real-world graphs and 4 algorithms. The results show that it provides on average 1.7 speedup and 89% energy saving compared to TESSERACT.

ACKNOWLEDGMENT

This work is supported by the following grants: NSF-CCF-1717754, NSF-CNS-1717984 and National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, U1435216, 61402198), National Basic Research (973) Program of China (2014CB340402). Contact: Mingxing Zhang (zhangmx12@mails.tsinghua.edu.cn), Yongwei Wu (wuyw@tsinghua.edu.cn).

REFERENCES

[1] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and E. Sha, "Pursuing a petaflop: point designs for 100 tf computers using pim technologies," in *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers '96., Sixth Symposium on the*, pp. 88–97, Oct 1996.

[2] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, ICPP '94, (Washington, DC, USA), pp. 77–84, IEEE Computer Society, 1994.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2012.

[5] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, *et al.*, "Die stacking (3d) microarchitecture," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 469–479, IEEE, 2006.

[6] Hybrid Memory Cube Consortium, *Hybrid Memory Cube Specification 2.1.* 2015.

[7] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux^2: Distributed Graph Computation for Machine Learning," in *The 14th USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[8] A. Alexandrescu and K. Kirchhoff, "Data-driven graph construction for semi-supervised graph-based learning in nlp.," in *HLT-NAACL*, pp. 204–211, 2007.

[9] A. Goyal, H. Daumé III, and R. Guerra, "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*, pp. 1069–1080, Association for Computational Linguistics, 2012.

[10] T. Zesch and I. Gurevych, "Analysis of the wikipedia category graph for nlp applications," in *Proceedings of the TextGraphs-2 Workshop (NAACL-HLT 2007)*, pp. 1–8, 2007.

[11] W. M. Campbell, C. K. Dagli, and C. J. Weinstein, "Social network analysis with content and graphs," *Lincoln Laboratory Journal*, vol. 20, no. 1, pp. 61–81, 2013.

[12] L. Tang and H. Liu, "Graph mining applications to social network analysis," in *Managing and Mining Graph Data*, pp. 487–513, Springer, 2010.

[13] T. Wang, Y. Chen, Z. Zhang, T. Xu, L. Jin, P. Hui, B. Deng, and X. Li, "Understanding graph sampling algorithms for social network analysis," in *2011 31st International Conference on Distributed Computing Systems Workshops*, pp. 123–128, IEEE, 2011.

[14] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *Proceeding of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Mar 2016.

[15] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *Proceeding of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2017.

[16] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 105–117, ACM, 2015.

[17] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[18] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A processing in memory taxonomy and a case for studying fixed-function pim," in *Workshop on Near-Data Processing (WoNDP)*, 2013.

[19] M. OĆonnor, "Highlights of the high-bandwidth memory (hbm) standard," in *Memory Forum Workshop*, 2014.

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

[21] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. N. Udipi, "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*, pp. 198–209, ACM, 2013.

[22] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 145–156, IEEE Press, 2013.

[23] J. Kim, W. Dally, S. Scott, and D. Abts, "Cost-efficient dragonfly topology for large-scale systems," *IEEE micro*, vol. 29, no. 1, pp. 33–40, 2009.

[24] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: a cost-efficient topology for high-radix networks," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 126–137, ACM, 2007.

[25] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 375–386, 2015.

[26] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 475–486, ACM, 2013.

[27] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 113–124, IEEE, 2015.

[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[29] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed networks in social media," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, (New York, NY, USA), pp. 1361–1370, ACM, 2010.

[30] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, (USA), pp. 539–547, Curran Associates Inc., 2012.

[31] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[32] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, p. 5, 2007.

[33] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A power-area simulator for interconnection networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 1, pp. 191–196, 2012.

[34] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "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*, pp. 469–480, 2009.

[35] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Sotware-defined cache hierarchies," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 652–665, ACM, 2017.

[36] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 166–177, IEEE, 2016.

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

[38] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 457–468, IEEE, 2017.

[39] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[40] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "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)*, pp. 17–30, 2012.

[41] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[42] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 472–488, ACM, 2013.

[43] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John, "Proxy-guided load balancing of graph processing workloads on heterogeneous clusters," in *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 77–86, Aug 2016.

[44] S. Song, X. Zheng, A. Gerstlauer, and L. K. John, "Fine-grained power analysis of emerging graph processing workloads for cloud operations management," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 2121–2126, Dec 2016.

[45] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, "Data partitioning strategies for graph workloads on heterogeneous clusters," in *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Nov 2015.

[46] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 1, ACM, 2015.

[47] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.