PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates

Anurag Mukkara
anuragm@csail.mit.edu
MIT CSAIL

Nathan Beckmann
beckmann@cs.cmu.edu
CMU SCS

Daniel Sanchez
sanchez@csail.mit.edu
MIT CSAIL

ABSTRACT

Many applications perform frequent *scatter update* operations to large data structures. For example, in push-style graph algorithms, processing each vertex requires updating the data of all its neighbors. Neighbors are often scattered over the whole graph, so these scatter updates have poor spatial and temporal locality. In current systems, scatter updates suffer *high synchronization costs* and *high memory traffic*. These drawbacks make push-style execution unattractive, and, when algorithms allow it, programmers gravitate towards pull-style implementations based on *gather reads* instead.

We present PHI, a push cache hierarchy that makes scatter updates synchronization- and bandwidth-efficient. PHI adds support for pushing sparse, commutative updates from cores towards main memory. PHI adds simple compute logic at each cache level to buffer and coalesce these commutative updates throughout the hierarchy. This avoids synchronization, exploits temporal locality, and produces a load-balanced execution. Moreover, PHI exploits spatial locality by *selectively* deferring updates with poor spatial locality, batching them to achieve sequential main memory transfers.

PHI is the first system to leverage both the temporal and spatial locality benefits of commutative scatter updates, some of which do not apply to gather reads. As a result, PHI not only makes push algorithms efficient, but *makes them consistently faster than pull ones*. We evaluate PHI on graph algorithms and other sparse applications processing large inputs. PHI improves performance by 4.7× on average (and by up to 11×), and reduces memory traffic by 2× (and by up to 5×).

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**.

KEYWORDS

graph analytics, multicore, caches, locality, specialization

ACM Reference Format:
Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358254

1 INTRODUCTION

Sparse algorithms such as graph analytics, sparse linear algebra, and sparse neural networks are an increasingly important workload domain [22, 30, 53]. Unfortunately, sparse algorithms work poorly on existing memory systems, as they perform frequent, indirect memory accesses to small chunks of data scattered over a large footprint. For example, graph algorithms often process large graphs with 10-100 GB footprints [16, 48], which do not fit on-chip. Graph algorithms execute few instructions per edge, and processing each edge requires accessing a small (e.g., 4- or 8-byte) object over the whole graph, resulting in poor spatial and temporal locality.

Sparse algorithms are very diverse, but they can be broadly classified into two styles: *push-based* or *pull-based* execution. This nomenclature stems from graph analytics, where most algorithms proceed in iterations, and on each iteration the data of each vertex is updated based on the data of neighboring vertices. In push algorithms, source vertices (i.e., those whose values need to be propagated) are processed one by one, and each vertex propagates (pushes) its update to all its outgoing neighbors. Thus, in push algorithms, indirect accesses are *scatter updates*. By contrast, in pull algorithms, destination vertices are processed one by one, and each vertex reads (pulls) updates from its incoming neighbors. Thus, in pull algorithms, indirect accesses are *gather reads*.

While some algorithms admit both pull and push implementations, in many cases the algorithm requires or is asymptotically more efficient with a push implementation. For example, many algorithms such as PageRank Delta [36] process a small set of *active* vertices each iteration, and only active vertices push updates to neighbors; BFS is most efficient with a combination of push and pull iterations [8]; and push (i.e., outer-product) sparse matrix multiplication has higher locality [46]. Therefore, it is important for systems to support both styles of execution efficiently.

Unfortunately, in current systems, push algorithms suffer two major drawbacks over pull ones: *higher synchronization costs* and *worse memory traffic*. These drawbacks both happen because the indirect accesses in push algorithms are updates, whereas the indirect accesses in pull algorithms are reads. In parallel pull algorithms, different threads update disjoint vertices, and thus updating each vertex requires no synchronization. By contrast, push algorithms must support concurrent updates to the same vertex from multiple threads. Typical implementations use atomic read-modify-writes and incur significant serialization and cache line ping-ponging. Additionally, because many scatter updates have no locality, each such update requires fetching and writing back an entire line from main memory—twice the traffic required for reads. Given these drawbacks, the conventional wisdom is that pull algorithms are inherently more efficient, and hence, when an algorithm admits push and pull implementations, pull should be used [11, 18, 19, 66].

In this paper, *we disprove this conventional wisdom.* We introduce novel architectural support in the memory hierarchy to make scatter updates efficient both in terms of synchronization and memory bandwidth. Surprisingly, we show that push algorithms *can be made more bandwidth-efficient than pull ones* because updates can be reordered in ways that reads cannot, uncovering more opportunities to exploit locality. This flexibility gives push algorithms a fundamental advantage over pull ones, and enables small on-chip hierarchies to incur close to minimal memory traffic.

We leverage the insight that many algorithms perform *commutative* scatter updates, such as addition or logical operations. Commutative updates produce the same result regardless of the order they are applied in, which enables the system to reorder and coalesce them to improve performance.

Prior work has used commutativity to partially address *either* the synchronization *or* the bandwidth costs of scatter updates, but has not tackled both, and these techniques also introduce drawbacks. First, Remote Memory Operations (RMOs) [28, 49], Coup [62], and CCache [6] add hardware support to reduce synchronization overheads: RMOs push updates to shared caches, where they are performed atomically, whereas Coup and CCache enable private caches to buffer commutative updates. However, these techniques still incur the main memory traffic blowup of scatter updates, and also suffer from more on-chip traffic than required.

Second, *update batching* techniques like MILK [29] and Propagation Blocking [10] first batch updates to cache-fitting graph slices, then apply each batch. These techniques transform indirect accesses into efficient streaming accesses, achieving great spatial locality, but they *sacrifice all temporal locality*, causing much more memory traffic than needed. This prior work also performs this batching in software, which adds significant overheads.

To address these problems we propose PHI, a *push* cache hierarchy that makes commutative scatter updates efficient. PHI achieves this through three key contributions:

(1) PHI extends caches to buffer and coalesce updates, acting as large coalescing write buffers. Cores push updates towards main memory through the cache hierarchy in a unidirectional fashion. By coalescing updates, PHI exploits temporal locality.
(2) PHI *selectively* employs update batching when evicting partial updates from the last-level cache: it streams updates to in-memory batches when there is little spatial locality, and applies updates *in-place* when there is significant spatial locality. By batching updates, PHI exploits spatial locality. Coalescing and selective batching work together to exploit *both* temporal and spatial locality, achieving the benefits of prior techniques like update batching while avoiding their drawbacks.
(3) PHI performs hierarchical buffering and coalescing: private and shared caches both buffer and coalesce updates. This approach avoids synchronization, reduces on-chip traffic, and balances the load among shared cache banks.

Fig. 1 illustrates the benefits of PHI for the PageRank algorithm on the `uk-2005` web graph [15]. We compare PHI with conventional *Push* and *Pull* implementations, and with a push implementation that uses update batching (*UB*). Pull not only has lower memory traffic than Push, but avoids frequent synchronization among cores. Thus, it improves performance over Push by 3.3×. Although UB has lower memory traffic than Pull, it is only slightly faster, as the memory traffic reduction comes at the cost of extra instructions. PHI further reduces memory traffic over UB by coalescing updates in the cache hierarchy while also avoiding UB's instruction overheads. Thus, PHI improves performance over Push by 6.5×. PHI's memory traffic is only 3.1× higher than the compulsory traffic, i.e., the traffic incurred with unbounded caches (the dotted line in Fig. 1).

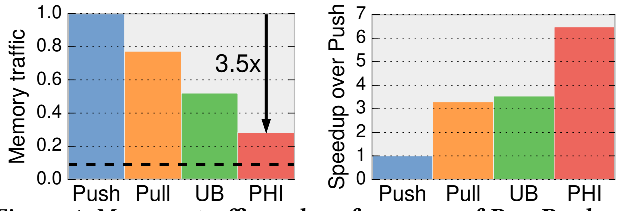


Figure 1: Memory traffic and performance of PageRank on `uk-2005`. The dotted line indicates compulsory traffic.

We evaluate PHI using detailed microarchitectural simulation. We focus our evaluation on graph algorithms, which are particularly memory-intensive due to the large footprint of real-world graphs, but PHI's benefits apply to other sparse applications that use scatter updates like sparse linear algebra and in-memory databases [7]. On a 16-core system, PHI reduces main memory traffic by up to 5× and by 2× on average over Push. On average, PHI incurs less than 2× the traffic of an ideal memory hierarchy. Consequently, PHI improves performance by up to 11× and by 4.7× on average. While we evaluate PHI on a general-purpose multicore, its techniques are easily applicable to an FPGA or ASIC accelerator.

2 BACKGROUND AND MOTIVATION

2.1 Push versus pull execution

Sparse algorithms are those that operate on sparse data structures, i.e., structures that efficiently encode collections of mostly zero values by representing only the nonzero values and their coordinates. Sparse algorithms perform *pull-* or *push-based* indirect accesses on sparse data structures.

For concreteness, we focus on graph analytics, where the sparse structure is the adjacency matrix that encodes the edges among vertices. Graphs are large and highly sparse, and graph algorithms require little processing per edge, making these algorithms more memory-intensive than other sparse ones (e.g., sparse tensor operations [30]). However, the techniques we develop apply to a wide range of sparse algorithms beyond graphs, as we will see in Sec. 4.

In push-based execution, the graph encodes the *outgoing* edges of each vertex, and each processed vertex (source) pushes updates to its out-neighbors (destinations). In pull-based execution, the graph format encodes the *incoming* edges to each vertex, and each processed vertex (destination) pulls updates from its in-neighbors (sources).

While many sparse algorithms admit push and pull implementations, there are often algorithmic reasons that force either approach. For example, *non-all-active* graph algorithms [50] maintain a small set of active vertices, and only these vertices update neighbors on each iteration. Push versions of non-all-active algorithms are more work-efficient because they only traverse the outgoing edges of active vertices, whereas pull versions traverse the incoming edges of all vertices. Direction-optimizing implementations of BFS [8] and other algorithms [50] switch between push and pull modes across iterations to reduce work. And in other sparse algorithms, such as degree counting, the push version is asymptotically more efficient (Sec. 4). It is thus important for systems to support both types of execution efficiently.

Unfortunately, in current systems, push implementations are hampered by two main drawbacks: *synchronization costs* and *memory traffic overheads*.

First, in a push implementation, multiple source vertices scatter updates to the same destination vertex concurrently. For example, in Fig. 2, vertices 0 and 1 both update vertex 2. Thus, typical push implementations use locks or atomic read-modify-writes, causing heavy synchronization and cache-line ping-ponging. This does not happen in pull implementations. For example, in Fig. 2, a single thread would process vertex 2 by gathering the updates from vertices 0 and 1.

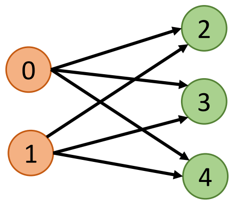


Figure 2: Example graph.

Second, when graphs are large and do not fit in caches, many scatter updates have no reuse and cause two memory accesses per update: the line is first fetched from memory, modified, and later written back. For example, suppose that the graph in Fig. 2 was large and vertices 0 and 1 were processed far away in time, so that the cache could not retain vertex 2's cache line between the processing of vertices 0 and 1. In this case, vertex 2 would be fetched from memory when updated by vertex 0, written back, then fetched again by vertex 1, and finally written back. By contrast, pull implementations suffer from poor reuse on reads, but writes are sequential and thus have great locality. In our example, updating vertex 2 would require reading the lines for vertices 0 and 1, incurring nearly half the memory traffic.

Given these drawbacks, graph frameworks often prefer pull implementations over push ones [11, 18, 19, 66]. But this choice *stems from architectural limitations*, not algorithmic ones: push implementations are slower due to their mismatch with the pull nature of existing memory hierarchies. PHI makes push mode efficient by matching the algorithmic direction of information flow with that of the memory hierarchy. PHI builds on insights from prior work, which we review next. Table 1 gives a qualitative comparison with prior proposals.

Scheme	Synchronization	Memory traffic
Push	Very high	Very high
Pull	Low	High
RMO	Medium	Very high
Coup	Low	Very high
Update batching	Low	Medium
PHI	**Low**	**Low**

Table 1: Comparison of PHI with prior techniques.

2.2 Hardware support for updates

Prior work has observed that updates to shared data are inefficient in conventional cache hierarchies, and has proposed several techniques to make them efficient.

Remote memory operations (RMOs) send and perform update operations at a fixed location. RMOs were first proposed in the NYU Ultracomputer [17]. The Cray T3D [28], T3E [49], and SGI Origin [32] implemented RMOs at the memory controllers, while recent GPUs [56] and the TilePro64 [23] implement RMOs in shared caches. While RMOs avoid ping-ponging cache lines, they send every update to a shared, fixed location. This causes global traffic and hotspots. By contrast, PHI coalesces updates in a hierarchical fashion to avoid hotspots. This is especially important for graph analytics, where high-degree vertices cause significant imbalance.

Coup [62], CommTM [61], and CCache [6] perform commutative updates in a distributed fashion. Coup modifies the coherence protocol to allow multiple private caches to hold update-only permission to the same cache line. Private caches buffer and coalesce updates, reducing update traffic. A read triggers a reduction of the private copies in the shared cache, and the result is the sent to the requesting core. Coup supports a limited number of operations, whereas CommTM and CCache add support for user-defined operations.

While these techniques avoid the synchronization overheads of scatter updates, they do not improve their locality. Thus, when the data being updated is too large to fit in the cache, these techniques incur the same high memory traffic as conventional push algorithms, as discussed in Sec. 2.1.

Like the above techniques, PHI exploits commutativity to buffer scatter updates in private caches. Unlike the above techniques, PHI also leverages the fact that scatter updates are applied in bulk to avoid onerous coherence protocol changes and to reduce on-chip traffic further.

2.3 Update batching

Prior work has proposed *update batching* techniques to improve the spatial locality of push algorithms.[1] MILK's DRAM-conscious Clustering [29] and Propagation Blocking [10] improve locality by translating indirect memory references into batches of efficient sequential main memory accesses. While Propagation Blocking was designed specifically for the PageRank algorithm, MILK is a compiler that handles a broad set of commutative operations on indirectly accessed data.

```
def PageRank(Graph G):
  # Binning phase
  for src in range(G.numVertices):
    update = genUpdate(G.vertex_data[src])
    for dst in G.outNeighbors[src]:
      binId = dst / neighborsPerBin
      bins[binId].append({dst, update})

  # Accumulation phase
  for bin in bins:
    for dst, update in bin:
      G.vertex_data[dst].newScore += update
```

Listing 1: Push PageRank using update batching.

Listing 1 shows pseudocode for a push version of PageRank[2] using update batching (UB). UB splits execution into two phases, *binning* and *accumulation*. In the binning phase, UB accesses the graph edges sequentially to generate the updates to destination

[1]Though we adopt graph analytics terminology in this paper, we use the term update batching instead of the more common Propagation Blocking to avoid confusion with graph blocking/tiling, a different optimization [55, 60] (Sec. 5).
[2]There are more efficient PageRank variants that do not process all edges on each iteration. We later evaluate PageRank Delta, which performs this optimization.

vertices and writes them to bins. Each bin holds updates for a cache-fitting fraction of vertices. In the accumulation phase, the updates are read bin-by-bin and applied to destination vertices.

UB reduces traffic because it enjoys perfect spatial locality. The binned updates are very large, so they are spilled to main memory, but because each bin is written sequentially, scatter updates are transformed into efficient streaming writes. These updates are then fetched and applied, but because each bin contains updates for a cache-fitting fraction of vertices, they enjoy great locality.

Fig. 3 illustrates UB with a small example, where a graph with 16 vertices (8 cache lines) is processed in a 4-line cache. Destination vertices are divided into two 4-line bins. Each source vertex generates an update and scatters it to each of its neighbors. In this example, all updates to vertices 0-7 are collected in the first bin and those to vertices 8-15 are collected in the second bin. Each bin holds *(destination id, update)* pairs in consecutive memory locations (e.g., *(0, A)*... in bin 0). Once all the updates are collected in bins, updates are applied bin by bin. When applying updates from bin 0, only destination vertices 0-7 are fetched into the cache. Similarly, when updates from bin 1 are being applied, destination vertices 0-7 are evicted while vertices 8-15 are fetched into the cache. Thus, each slice of destination vertices is fetched into the cache only once, incurring the minimal memory traffic.

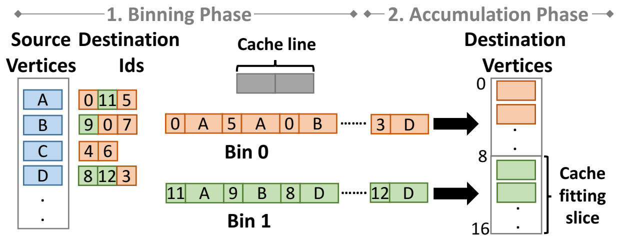
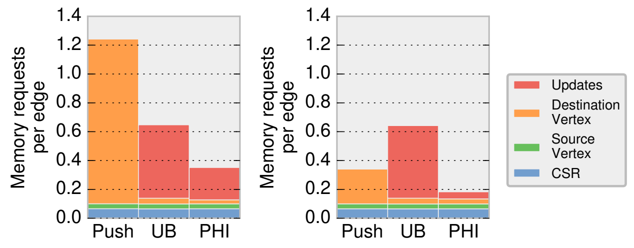


Figure 3: Example of update batching on a graph with 16 vertices and a 4-line cache that can hold 8 vertices (2 vertices per cache line).

Because UB does not exploit graph structure, its memory traffic is easy to analyze. Assume that caches are much smaller than the processed graph, so that there is negligible opportunity for reuse across iterations. In this case, there is some *compulsory traffic* that every scheme must incur: the source vertices and adjacency matrix must be read from memory at least once, and the destination vertices must be written to once. A conventional push implementation achieves this minimum, compulsory traffic on reads, but is hampered by the large overheads of scatter updates to destination vertices. By contrast, UB also achieves the minimum amount of traffic on scatter updates (one read and one writeback per line), but at the cost of streaming all updates to memory. Consider a graph with V vertices, E edges, and U bytes per update. Each logged update requires $I = log_2(V)/8$ bytes for the vertex id, so all the bins consume a total of $(I + U) \cdot E$ bytes, and UB incurs an extra traffic of $2 \cdot (I + U) \cdot E$ bytes over compulsory to write and read updates.

Fig. 4a shows this tradeoff by analyzing the memory traffic of PageRank on the uk-2005 graph (same experiment as Fig. 1). Each bar shows a breakdown of memory accesses to the different data structures (CSR is the adjacency matrix). In the Push implementation, scatter updates contribute 92% of memory traffic. In UB, scatter updates take 29× less traffic—the minimum amount. However, 78% of the traffic is now spent on batched updates. This is 1.9× better than Push overall, but still a far cry from the compulsory traffic.



(a) Original graph. (b) Preprocessed graph.

Figure 4: Memory traffic breakdown by data structure for PageRank on (a) the `uk-2005` graph, and (b) a preprocessed version of the same graph.

UB's key limitation is its disregard for temporal locality. If a vertex has thousands of incoming neighbors, spilling all its updates to main memory is wasteful—it is far more efficient to coalesce the updates to that vertex in the on-chip caches. PHI does just that by applying UB selectively upon eviction, dramatically reducing batched update traffic. Fig. 4a shows that PHI's update traffic is 2.3× smaller in this case, and overall, memory traffic is 1.8× lower than UB.

2.4 Preprocessing algorithms

Prior work has proposed several *preprocessing algorithms* to reorder sparse data structures to improve locality [20, 54, 59, 64, 65, 67]. For example, graph preprocessing techniques reorder vertices in memory so that closely related vertices are stored nearby. In sparse linear algebra, these are known as fill-reducing permutations.

Preprocessing algorithms improve locality, but they are expensive, often taking many times longer than the algorithm itself. This makes preprocessing algorithms impractical for many use cases, e.g., on single-use graphs or simple algorithms [35, 39]. But if a graph is reused many times, preprocessing can be beneficial, so prior work has proposed to use preprocessing selectively [5].

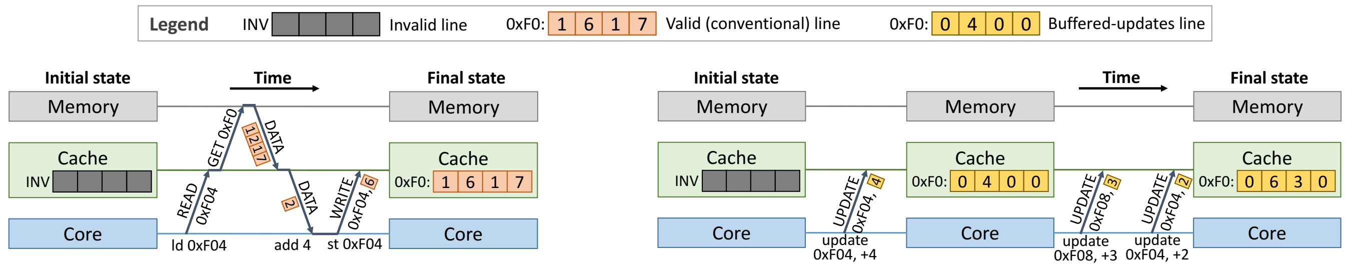
Because PHI leverages temporal and spatial locality, it is complementary to preprocessing algorithms. Fig. 4b shows this tradeoff, reporting memory traffic for the same experiment from Fig. 4a, but where the graph is preprocessed by sorting vertices by incoming degree [64]. Because UB does *not* exploit temporal locality, it incurs the same memory traffic as with the non-preprocessed graph, and is now worse than Push. By contrast, PHI achieves 2× lower traffic on this graph, just 36% higher than compulsory traffic.

3 PHI DESIGN

PHI consists of three key techniques:

(1) **In-cache update buffering and coalescing** extends caches to act as coalescing write buffers for partial updates. If the cache receives an update for a non-cached line, it does *not* fetch the line. Instead, it buffers the partial update. Moreover, caches coalesce (i.e., merge) multiple updates to the same cache line. This technique *exploits temporal locality* and *enables PHI's other optimizations*.

(2) **Selective update batching** extends the last-level cache's eviction process to apply buffered updates adaptively. When a line with partial updates is evicted, the cache first counts the number of elements in the line holding updates. If few elements hold



(a) In a conventional system, every missing update causes a fetch from memory.

(b) In PHI, an update that misses allocates a *buffered-updates line* without fetching the line from memory.

Figure 5: Update buffering and coalescing operation in the single-core system.

updates (i.e., there is poor spatial locality), the cache performs update batching (Sec. 2.3), streaming them to memory sequentially. But to avoid the overheads of update batching, if most elements hold updates (i.e., there is high spatial locality), the cache applies the updates in-place, fetching and writing back the line. This technique *achieves high spatial locality* in all cases.

(3) **Hierarchical buffering and coalescing** applies to multi-level hierarchies. In this scenario, private caches act as buffers for the shared cache, locally buffering and coalescing updates without any per-line synchronization or coherence protocol changes. This technique *eliminates synchronization overheads*, *enables update parallelism*, and *produces a load-balanced execution.*

We now introduce these techniques progressively using two scenarios. First, we consider a single-core system with a single cache, and introduce the first two techniques, whose goal is to reduce memory traffic (Sec. 3.1). Then, we extend PHI to a parallel system with private and shared caches, introducing the third technique, whose goal is to eliminate synchronization overheads (Sec. 3.2).

3.1 Making scatter updates bandwidth-efficient

Consider the system on the left of Fig. 5a, with a single core and cache. We first explain PHI in this simplified system to focus on the memory bandwidth problem. As we saw in Sec. 2, prior work incurs different kinds of memory traffic overheads. On the one hand, the conventional push implementation suffers from poor spatial locality, and each update that misses incurs a fetch and a writeback to memory, doubling traffic over that of a pull implementation. On the other hand, update batching achieves perfect spatial locality, but sacrifices temporal locality, producing large streams of updates that are spilled to main memory, then read back.

PHI combines the benefits of both approaches while avoiding their drawbacks. PHI can be seen as an adaptive version of update batching that exploits the significant temporal and spatial locality available in updates to drastically reduce the number of updates batched and streamed to memory.

3.1.1 Execution phases. Like in update batching, in PHI, algorithm execution is divided into two phases. In the first phase, the core pushes updates to the memory hierarchy, which has the option of applying them directly (in-place) or batching them for the second phase. Batching streams updates in bins, with each bin corresponding to a cache-fitting fraction of vertices. In the second phase, the core applies the batched updates bin-by-bin to achieve good locality. We first explain how PHI's techniques work on the first phase, then describe the second phase, and finally present PHI's concrete API.

In-cache update buffering and coalescing: To reap the locality of scatter updates, the first step is to let the cache buffer updates *without fetching the data being updated from memory*, avoiding the behavior shown in Fig. 5a. In other words, the cache should act as a very large *coalescing write buffer* [26] for updates that miss in the cache. Conventional write buffers alone cannot be used for this purpose, as they have a small number of entries (8-16) and cannot capture update reuse, which occurs over a longer timescale.

Enabling this behavior requires two simple changes, shown in Fig. 6. First, each line is extended with a *buffered updates bit* that denotes whether the line holds buffered updates. Second, the cache controller is extended with a *reduction unit*, a simple ALU that can perform the set of supported commutative operations (e.g., integer and floating-point additions, min/max, and bitwise logical operations).

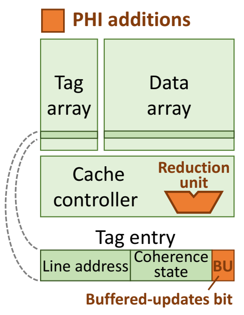


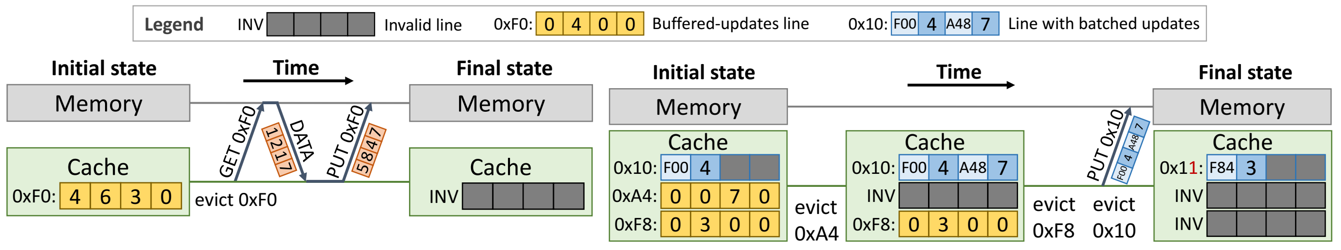
Figure 6: PHI hardware additions.

Fig. 5b shows how update buffering and coalescing works. The core sends updates to the cache. For each update, the cache first performs a tag lookup like in a conventional access. Then, if the access is a miss (e.g., the first update in Fig. 5b), the cache inserts a new *buffered-updates* line that contains the update at the right offset. If the access is a hit (e.g., the second and third updates in Fig. 5b), the cache uses the ALU to apply the update to the existing line, coalescing it. Note that, on a hit, the existing line may or may not be a *buffered-updates* line—operation is identical in both cases.

Unlike in conventional write buffers, caches need not track which elements or words of a line hold updates. Instead, PHI leverages that every commutative operation has an *identity element* (e.g., zero for addition and XOR, all-ones for AND, etc.). Buffered-updates lines are initialized with identity elements, as shown in Fig. 5b.

Selective update batching: Cache lines with buffered updates cannot be evicted like normal lines (i.e., they cannot be simply written back). On an eviction, PHI either applies the buffered updates in-place or performs update batching. Fig. 7 illustrates this process.

When a *buffered-updates* line is evicted, the cache controller counts the number of elements with updates (by comparing with the identity element). If this number exceeds a particular threshold, PHI performs the updates in-place: it fetches the line from memory, applies the updates using the ALU, and writes it back, as shown



(a) A buffered-updates line with high spatial locality is applied *in-place*. (b) Two buffered-updates lines without spatial locality are *batched*.

Figure 7: Selective update batching operation in the single-core system.

in Fig. 7a. If the number of updates is below the threshold, PHI performs update batching, as shown in Fig. 7b.

Selective batching reduces memory traffic because it avoids the overheads of update batching for lines with good spatial locality. In the extreme, consider the overheads of applying update batching to a line where all the elements have updates. An in-place update requires reading and writing one line to memory, but update batching requires tagging each element with its vertex id, so it requires writing all the elements in the line *and their ids* to an in-memory buffer, then reading the elements back. Therefore, in the best case (i.e., discounting all the potential vertex data misses incurred when applying the updates in the second phase, as these misses may be heavily amortized), update batching will incur extra traffic overhead of $(I+U)/U$, where I and U are the sizes of the vertex id and the updated value, respectively (e.g., with 32-bit vertex ids and 64-bit elements, this overhead is 12/8=1.5×). Conversely, update batching is beneficial when the fraction of elements in the line that have updates is below $U/(I+U)$. PHI uses this as the threshold to perform update batching.

PHI extends the cache controller with simple logic to perform update batching. All the data required for batching (bin pointers and partially filled bin cache lines) is kept in the cache array, in normal lines. Fig. 7b shows how this process works on a single bin. Initially, the cache has two buffered-updates lines with one update each, both of which map to the same bin, and another line, `0x10`, that holds batched updates for that bin. First, buffered-updates line `0xA4` is evicted, and its single update is logged into the bin. This modifies the bin's tail cache line, which is now full with updates. Then, buffered-updates line `0xF8` is evicted, and its single update is logged to the bin. This allocates a new line for the bin, `0x11`, which becomes partially filled. After some time, `0x10` is evicted and written to memory. Note how this process incurs a cache line write for every two updates, i.e., it has good spatial locality. (In this example lines are short, so only two updates fit per line; real systems with longer lines achieve larger savings.)

An alternative to making update batching decisions on a per-line basis would be to let the programmer turn off update batching when deemed beneficial (e.g., for small inputs and/or sparser iterations). However, this static approach would increase programmer burden and lose some locality benefits, as even for large footprints some updates can be applied in-place.

Applying batched updates: In the second phase, the updates batched to memory in the first phase are applied bin-by-bin. PHI again uses the cache's reduction unit for the second phase by changing the cache eviction process slightly.

For each bin, the core fetches the updates from memory using conventional loads and pushes them to the memory hierarchy. Similar to the first phase, the cache buffers and coalesces these updates. Unlike in the first phase, the cache controller does not perform further update batching on evictions: it always applies the buffered updates in-place, fetching the original line and writing it back (i.e., as shown in Fig. 7a). Since each bin holds updates to a cache-fitting fraction of vertices, in the common case, there is only one eviction per buffered-updates line in the second phase.

After the second phase finishes, all batched updates have been applied. However, the cache may still have some buffered-updates lines. To ensure correct behavior, if a line with buffered updates receives a read request, *the cache automatically applies the buffered updates in-place*: it fetches the line from main memory and merges the buffered-updates line to produce the final values. This process is almost the same as the one shown in Fig. 7a for evictions, except that the line is not written back to memory at the end, but stays in the cache.

PHI exploits temporal and spatial locality: In summary, PHI's combination of coalescing and selective update batching avoids the pitfalls of conventional update batching (UB). As we saw in Sec. 2, UB *(i)* does not exploit temporal locality; and *(ii)* by streaming updates to memory, it incurs far more traffic than needed on small graphs, preprocessed graphs, or on algorithms that process a small, cache-fitting region of the graph per iteration. By contrast, *(i)* PHI coalesces updates on-chip to exploit temporal locality, and exploits spatial locality through update batching when profitable; and *(ii)* PHI does not incur traffic for the lines of the graph that the cache can retain, so smaller or structured graphs enjoy intra- and inter-iteration reuse, unlike in UB.

3.1.2 Programming interface. Listing 2 illustrates PHI's API by showing the single-thread implementation of the PageRank algorithm. This code is similar to that of the UB implementation in Listing 1, with two phases. We now describe the interface only; Sec. 3.3 explains our specific implementation of PHI's primitives. We show this code for illustration purposes, but do not expect application programmers to change their code, as graph processing frameworks (Sec. 4.1) and sparse compilers [29, 30] can be easily changed to leverage PHI.

The algorithm begins by configuring PHI through the `phi_configure()` call. This specifies two pieces of information:

(1) The type of commutative operation (e.g., 32-bit addition).
(2) The number of bins and their starting addresses.

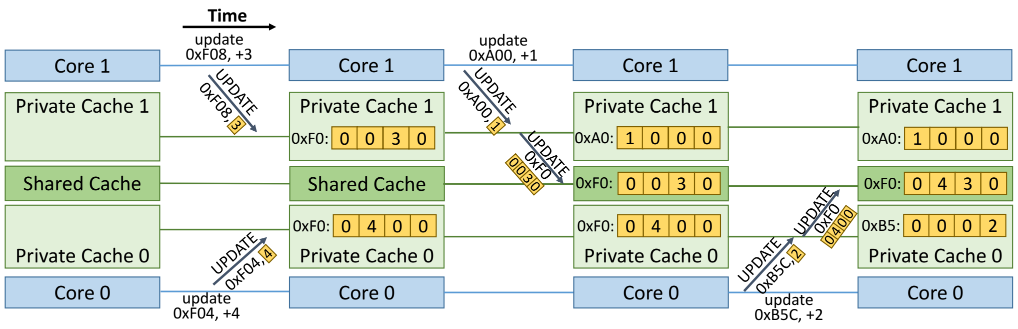


Figure 8: Hierarchical buffering and coalescing in a multicore system.

```
def PageRank(Graph G):
 phi_configure(...)
 phi_enable_batching()
 for src in range(G.numVertices):
  update = genUpdate(G.vertex_data[src])
  for dst in G.outNeighbors[src]:
   phi_push_update(G.vertex_data[dst], update)

 phi_disable_batching()
 for bin in bins:
  for dst, update in bin:
   phi_push_update(G.vertex_data[dst], update)
```

Listing 2: Serial PageRank implementation using PHI.

The algorithm then starts the first phase (lines 3–7). It first calls `phi_enable_batching()`, which enables selective update batching on cache evictions. Then, the algorithm traverses the graph edges and pushes updates using `phi_push_update()`.

Finally, the algorithm performs the second phase (lines 9–12). It calls `phi_disable_batching()` to disable update batching on cache evictions. Then, the algorithm fetches the updates that were batched to memory in the first phase, bin-by-bin, and applies them using `phi_push_update()`.

3.2 Making parallel scatter updates synchronization-efficient

We now extend PHI to parallel systems. Consider the system in Fig. 9, where each core has a private cache and all cores share a banked last-level cache. PHI's third technique, **hierarchical buffering and coalescing**, addresses the synchronization and scalability challenges of scatter updates in this system.

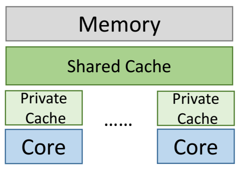


Figure 9: Example multicore system.

3.2.1 Mechanisms and execution. PHI leverages private and shared caches to buffer and coalesce updates at multiple levels of the cache hierarchy, without synchronization among cache levels and without changing the coherence protocol. Fig. 8 shows hierarchical buffering and coalescing in action, in a system with two cores, each with its own private cache. Each core pushes updates to its private cache. If the private cache does not have write permission to the line, it allocates a buffered-updates line to hold the update. For example, in Fig. 8, both private caches allocate a buffered-updates line for `0xF0`, without any communication with the shared cache. The private caches can then locally coalesce other updates to the same line, just like before. When a private cache needs to evict a buffered-updates line, it simply sends its contents as an update message to the shared cache bank. The shared cache allocates a buffered-updates line if needed (as shown in Fig. 8, middle), or coalesces the updates with existing ones (as shown in Fig. 8, right).

PHI's parallel operation is nearly identical as before: the algorithm goes through two phases, and selective update batching is enabled only on the first phase. Only shared cache banks perform selective update batching; on evictions, private caches simply update the shared cache.

(In-)coherence and flushing: For simplicity, PHI leverages that scatter updates happen in bulk to avoid changing the coherence protocol. Buffered-updates lines are not tracked by the coherence protocol, so private caches may have updates for lines that do not exist in the shared cache or directory. This approach simplifies the design, as the coherence protocol does not require any changes, and avoids coherence traffic, as private caches need not request any coherence permissions for buffered-updates lines. Note that private caches are still kept coherent for data beyond buffered updates.

The drawback of this design decision is that, when the algorithm finishes, *updates buffered in private caches must be flushed* to ensure correct behavior. Specifically, at the end of the second phase, all private caches traverse their tag arrays and flush every buffered-updates line to its corresponding shared cache bank. This process can be done in parallel and is fast because private caches are small.

Though adding an explicit flush step to ensure correctness may seem limiting, note that these algorithms perform updates in bulk, without any intervening reads. In fact, updated data should not be read before updates are fully applied even without private caches, since *a line may have batched updates somewhere else in memory*.

Consistency: PHI requires some changes to the consistency model. Commutative updates by different threads are not ordered by synchronization and hence constitute a data race. These data races are harmful under the data-race-free (DRF) consistency model adopted in modern languages such as Java and C++.

Fortunately, prior work has already shown how to address this issue: Sinclair et al. [51] propose the DRFrlx (DRF-relaxed) model, which extends DRF to provide SC-centric semantics for the common use cases of relaxed atomics. Specifically, PHI can use the semantics of commutative relaxed atomic operations in the DRFrlx model. In DRFrlx, commutative update operations need to be explicitly annotated to use relaxed atomic semantics, and a fence is needed before any read to updated data. PHI achieves both of these conditions by *(i)* performing these updates through a different instruction (`phi_update`), and *(ii)* requiring software to perform an explicit flush step (called `phi_sync`, Sec. 3.2.2) after updates are applied and before any reads. Thus, to ensure that data races do not occur, it is sufficient for the explicit flush step to have full fence semantics (so that no accesses following the flush can be reordered before all updates are visible).

Much like weak consistency models offer no guarantees for data read or written without adequate fences, PHI offers no guarantees if programs do not follow the above requirements, e.g., if a read to updated data is performed before `phi_sync`, or if the program mixes conventional atomics with `phi_update`s. Thus, PHI requires some discipline from programmers.

Beyond these changes to incorporate relaxed-atomic semantics, PHI does not affect the consistency model for other loads and stores that do not touch updated data.

Comparison with prior work: PHI's hierarchical buffering and coalescing shares some of the same objectives as Coup, CommTM, and CCache, which perform commutative updates in multiple private caches and reduce synchronization (Sec. 2.2). However, PHI's implementation is simpler than these techniques. These prior techniques modify the coherence protocol, adding complexity, and need private caches to request and acquire update-only permissions to the line before applying any updates locally, adding traffic, serialization, and shared cache pollution when there is little reuse.

However, these prior techniques are more general than PHI, as they transparently satisfy reads to data receiving commutative updates: the coherence protocol is used to gather and merge all partial updates in response to a read. PHI does not do this, requiring an explicit flush step instead. We deem this is a good tradeoff, as the bulk scatter updates that PHI targets do not need this behavior as explained above.

While PHI moves complexity from the coherence protocol to software, only the framework code needs to interact with PHI, not the programmer. This is similar to how, with relaxed memory models, software libraries are in charge of fences in practice, and programmers mainly use higher-level primitives.

PHI avoids synchronization and achieves load balance: Overall, hierarchical buffering and coalescing design gives three key benefits. First, it increases the coalescing throughput of the system, since multiple private caches can coalesce updates to the same line in parallel. Moreover, this happens without any synchronization between caches.

Second, most cache accesses happen at smaller, energy-efficient lower cache levels due to hierarchical reuse patterns of real-world inputs. Without hierarchical coalescing, each update would require accessing a large cache.

Third, hierarchical coalescing balances traffic across shared cache banks, because private caches are especially effective at coalescing frequent updates to the same data. This is important for graph analytics, where graphs often have very few vertices that account for a large fraction of edges. Without hierarchical coalescing, these high-degree vertices can create uneven traffic among banks, as we will see in Sec. 4.4. With hierarchical coalescing, these frequent updates are all coalesced in private caches, avoiding imbalance.

3.2.2 Programming interface. We add small extensions to PHI's serial API described in Sec. 3.1.2. First, we extend `phi_configure()` to specify bin information for each shared cache bank, so shared banks can batch updates without synchronization. Second, we add a `phi_sync()` primitive to flush buffered-updates lines from private caches at the end of the second phase.

The parallel PHI implementation is very similar to the serial one from Listing 2. We exploit the trivial parallelism available in both phases by changing the for loops in lines 4 and 11 into **`parallel for`** loops; and `phi_sync()` is called at the end to ensure no updates remain in private caches.

3.3 PHI implementation details

ISA: PHI adds only one instruction, `push_update`. `push_update` is similar to a store; it has two source operands that specify the address and value of the update. Configuration through `phi_configure()`, `phi_enable_batching()` etc., is infrequent and happens using memory-mapped registers (e.g., the same way a DMA engine is configured).

Virtual memory and index computation: Updated data occupies a contiguous range of virtual memory, but caches are physically addressed. This introduces some subtleties for update batching. For simplicity, when batching updates, we would like to derive each update's index (e.g., vertex id) and bin from its cache line address. Our implementation achieves this by allocating update data in a contiguous physical memory region. (OSes already support allocating contiguous physical memory, e.g., for DMA buffers [41].) This way, computing an update's index requires subtracting the region's starting physical address from the update's physical address.

This approach needs some care when paging out update data: the OS must disable update batching before paging it out. A more complex alternative would be to add reverse TLBs to LLC banks, then compute indexes using virtual addresses.

Update batching: To perform update batching, the shared cache tracks two types of information for each bin: a pointer to the next address where updates will be written and a partially filled cache line at that address. Both are stored in normal, cacheable memory.

Because selective update batching is done at the last-level cache, it uses *physical addresses* to avoid having an in-cache TLB. Each bin is initially given a contiguous chunk of physical memory (e.g., 256 KB per bin). Bins need not be contiguous: when the cache exhausts a bin, it raises an interrupt to request another chunk (and disables batching in the interim). Coalescing makes these interrupts extremely rare.

When the last-level cache has multiple banks, each bank performs batching autonomously. To avoid communication among cache banks, we ensure that both the pointers and the bin cache lines accessed by any given bank are mapped to the bank itself. Since most systems interleave cache lines across banks, this simply results in a striped layout. Specifically, in a system with B banks, the bin pointers array is B times larger, and each bank uses one out of B cache lines for pointers. Then, each bank fills only the 1/B lines of the bin that map to itself. Each time the current line for the bin is exhausted, the bank bumps the pointer by B cache lines. This striping results in completely local operation, and thanks to coalescing, banks fill bins at about the same rate, achieving near-perfect space utilization.

Overhead analysis: PHI hardware adds small costs. The main overhead is the per-line buffered updates bit. With 64-byte lines,

Cores	16 cores, x86-64 ISA, 2.2 GHz, Haswell-like OOO [47]
L1 caches	32 KB per core, 8-way set-associative, split D/I, 3-cycle latency
L2 cache	256 KB, core-private, 8-way set-associative, 6-cycle latency
L3 cache	32 MB, shared, 16 banks, 16-way hashed set-associative, inclusive, 24-cycle bank latency, DRRIP replacement
Global NoC	4×4 mesh, 128-bit flits and links, X-Y routing, 1-cycle pipelined routers, 1-cycle links
Coherence	MESI, 64 B lines, in-cache directory, no silent drops
Memory	4 controllers, FR-FCFS, DDR4 1600 (12.8 GB/s per controller)

Table 2: Configuration of the simulated system.

this costs 0.17% additional storage in caches (e.g., 64 KB for the 32 MB LLC we use).

The reduction unit is small because it uses simple operations. We implement RTL for a reduction unit with 64-bit floating-point and integer additions and logical operations. We synthesize it using yosys [57] and the FreePDK45 [24] cell library, with a 1 GHz target frequency (higher ones are possible). Each reduction unit takes $0.09mm^2$ (i.e., about $0.008mm^2$ in 14 nm). In our 16-core system, reduction units take about 0.06% of chip area.

4 EVALUATION

4.1 Methodology

We now present our evaluation methodology, including the simulated system, applications and datasets we use.

Simulation infrastructure: We perform microarchitectural, execution-driven simulation using zsim [47]. We simulate a 16-core system with parameters given in Table 2. The system uses out-of-order cores modeled after and validated against Intel Haswell cores. Each core has private L1 and L2 caches, and all cores share a banked 32 MB last-level cache. The system has 4 memory controllers, like Haswell-EP systems [21]. We use McPAT [33] to derive the energy of chip components at 22 nm, and Micron DDR3L datasheets [37] to compute main memory energy.

Applications: We evaluate PHI on six sparse applications, listed in Table 3. All applications use objects that are much smaller than a cache line (64 B).

First, we use four graph algorithms from the widely used Ligra [50] framework. These include both all-active and non-all-active algorithms. All-active algorithms are ones in which each vertex and edge is processed in each iteration. *PageRank* computes the relative importance of vertices in a graph, and was originally used to rank webpages [45]. *PageRank Delta* is a variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their PageRank score [36]. *Connected Components* divides a graph's vertices into disjoint subsets (or components) such that there is no path between vertices belonging to different subsets [13]. *Radii Estimation* is a heuristic algorithm to estimate the radius of each vertex by performing multiple parallel BFS's from a small sample of vertices [34].

Application	Update Size	Reduction Operator	All-Active?
PageRank (**PR**)	8 B	double add	Yes
PageRank Delta (**PRD**)	8 B	double add	No
Conn. Components (**CC**)	4 B	integer min	No
Radii Estimation (**RE**)	8 B	bitwise or	No
Degree Counting (**DC**)	4 B	integer add	Yes
SpMV (**SP**)	8 B	double add	Yes

Table 3: Applications.

To avoid framework overheads, we tune the original Ligra code [50], incorporating several optimizations in the scheduling code like careful loop unrolling that yield significant speedups: our implementations outperform Ligra by up to 2.5×. We then change the framework's code to use PHI. Note that these optimizations affect only the baseline Push and Pull implementations and are agnostic to PHI.

Our approach lets us start with an optimized software baseline, which is important since it affects qualitative tradeoffs. In particular, well-optimized implementations are more memory-bound and saturate bandwidth more quickly.

Degree Counting computes the incoming degree for each vertex from an unordered list of graph edges and is often used in graph construction [9]. Whereas the other algorithms admit Pull and Push implementations, *Degree Counting* requires push-style execution.

Finally, *Sparse Matrix-Vector Multiplication (SpMV)* is an important sparse linear algebra primitive. The sparse matrix is stored in compressed sparse row (CSR) format. In the Pull version, the matrix is scanned row by row and values are gathered from the corresponding elements in the input vector. In the Push version, the matrix is scanned column by column and each column scatters partial sums to the result vector.

For update batching, we use the optimized implementation obtained from the authors of Propagation Blocking [10]. We modify the simulator to model non-temporal stores, which are crucial to reduce memory traffic of update batching.

Datasets: We evaluate the four graph algorithms and *Degree Counting* on five large web and social graphs shown in Table 4. For SpMV, we use a sparse matrix representative of structured optimization problems.

Graph	Vertices (M)	Edges (M)	Source
arb	22	640	arabic-2005 [15]
ukl	39	936	uk-2005 [15]
twi	41	1468	Twitter followers [31]
sk	51	1949	sk-2005 [15]
web	118	1020	webbase-2001 [15]
nlp	27	760	nlpkkt240 [15]

Table 4: Real-world datasets.

With the object sizes listed in Table 3, the vertex data footprint is much larger than the last-level cache. We represent graphs in memory in CSR format.

Graph algorithms are generally executed for several iterations until a convergence condition is reached. To avoid long simulation times, we use *iteration sampling*: we perform detailed simulation only for every 5th iteration and fast-forward through the other iterations (after skipping initialization). This yields accurate results since the execution characteristics of all algorithms change slowly over consecutive iterations. Even with iteration sampling, we simulate *over 100 billion instructions* for the largest graph.

4.2 PHI improves runtime, traffic, and energy

Performance: Fig. 10 summarizes the performance of different schemes. Each bar shows the speedup over the Push implementation (higher is better). Each group of bars reports results for a single application; the right-most group shows the gmean across applications. Most applications use multiple inputs, so each bar shows the gmean speedup across inputs (we will analyze per-input results later).

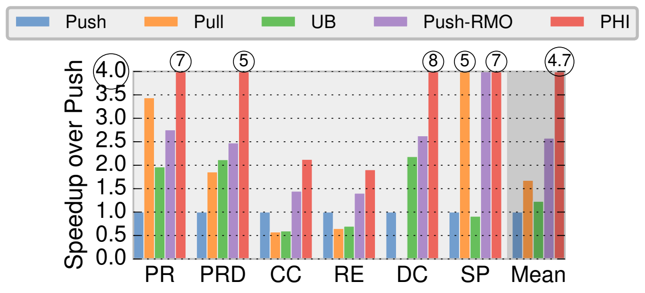


Figure 10: PHI achieves substantial speedups.

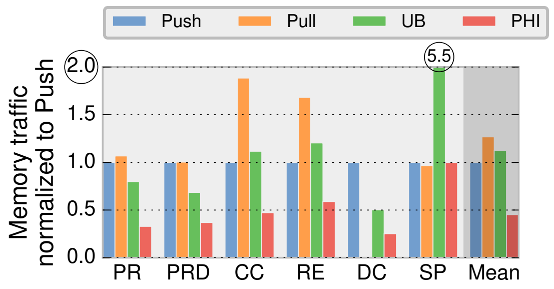


Figure 11: PHI reduces memory traffic significantly.

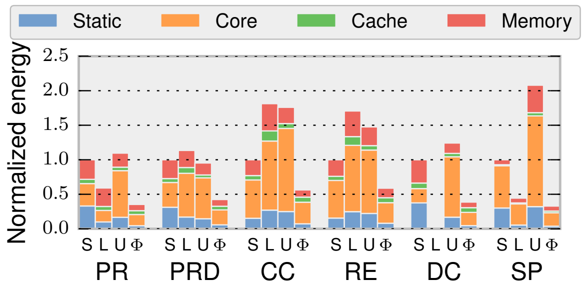


Figure 12: Energy breakdown normalized to Push. S:Push, L:Pull, U:UB, Φ:PHI.

We compare PHI with conventional Push and Pull implementations, update batching (UB), and Push-RMO, a variant of Push that uses hardware support to perform remote memory operations at LLC banks. DC does not have a Pull version.

PHI improves performance significantly and is the fastest scheme across all applications. PHI improves performance over Push by 4.7× on average and by up to 8.3× (DC). Moreover, no other scheme uniformly dominates the others across applications, and most have poor performance on some cases.

PHI substantially outperforms Push-RMO, the second-best scheme, thanks to its lower memory traffic. While Push-RMO has the same memory traffic as Push, it avoids synchronization overheads and achieves a 2.6× gmean speedup. On average, Pull and UB improve performance by 67% and 23%, respectively. Although we do not compare with Coup [62], we expect it would achieve the same performance as Push-RMO because Push-RMO is limited by memory bandwidth, not on-chip traffic or synchronization.

PHI's speedups are larger for all-active applications like PR and DC, since they are much more memory-bound. PRD is non-all-active but has many iterations where most of the graph is active. Thus, it has similarly high speedups.

Non-all-active applications CC and RE process a small fraction of the graph on each iteration, so Push achieves a moderately high cache hit ratio. For these applications, Pull and UB perform extra work, which causes poor performance even though they have no synchronization overheads.

For SP, the input has a regular structure with good locality and Push already achieves relatively low memory traffic. Pull, Push-RMO, and PHI improve performance by avoiding synchronization. By contrast, UB performs extra work and hurts performance slightly.

Memory traffic: Fig. 11 shows the average memory traffic across inputs for all applications (lower is better). Push-RMO's traffic is identical to Push, so it is not shown.

PHI substantially reduces memory traffic for all applications except SP, by up to 4× for DC and 2× on average. PHI achieves the lowest memory traffic among all schemes by exploiting both temporal and spatial locality.

For all-active algorithms like PR and DC, Push and Pull achieve similar memory traffic while UB reduces memory traffic considerably, by up to 2× for DC.

For CC and RE, Pull and UB both increase memory traffic. First, Pull processes each incoming edge of all vertices rather than the outgoing edges of only the active vertices as in Push. Thus, Pull accesses more edge data which increases memory traffic. Second, baseline Push already has good hit ratio and the extra update traffic of UB is not a good tradeoff.

Finally, SP's input is a structured matrix, so Push, Pull, and PHI achieve good locality and similar traffic. But UB's traffic is 5.5× worse, because UB does not take advantage of structure and streams many updates to memory.

Energy: Fig. 12 shows the energy breakdown for various schemes. For the software-only schemes (Push, Pull, UB), most of the energy comes from core and main memory, and the fraction of energy from cores depends on how compute-bound the application is.

PHI reduces core energy over Push because it offloads both update processing and update batching to specialized hardware, reducing instruction count on general-purpose cores significantly. PHI reduces core energy by up to 3.1× (on SP). In contrast, UB increases core energy over Push as it adds instructions to log updates to memory. PHI's reduction in memory traffic causes proportional reductions in memory energy. Overall, PHI reduces energy by up to 3× (on SP).

4.3 PHI performs best across inputs

Beyond its raw performance benefits, PHI stands out for its consistency: whereas other techniques have weak spots, PHI uniformly performs best. We have seen this across applications; we now show PHI's consistency extends to inputs.

Memory traffic breakdown: Fig. 13 shows the main memory traffic breakdown of PageRank for each graph input. All schemes are normalized to Push. Both Push and Pull cause random, irregular accesses to neighbor vertices, which contribute a large fraction of the overall main memory traffic. For graphs with symmetric structure (twi, web), Push has higher overall traffic than Pull due to extra writeback traffic. For some non-symmetric graphs like sk, the graph's structure causes better locality for Push.

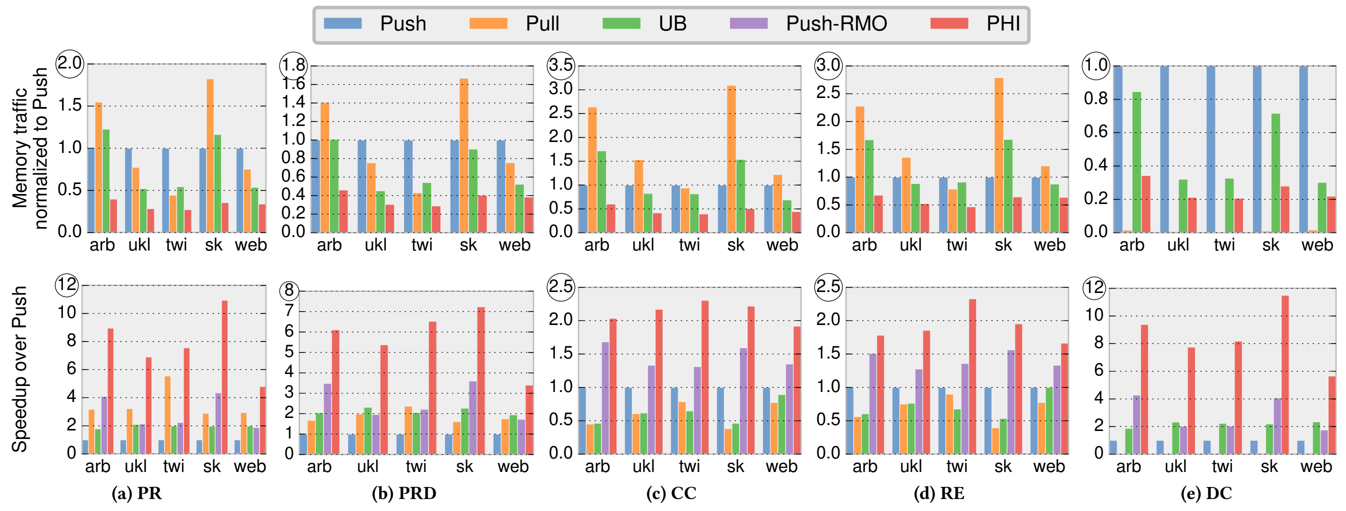


Figure 14: Per-input memory traffic (top) and performance (bottom) of the five graph applications, normalized to Push.

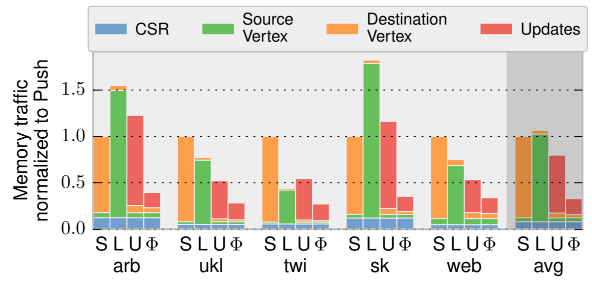


Figure 13: Breakdown of main memory traffic of PageRank by data structure. S:Push, L:Pull, U:UB, Φ:PHI.

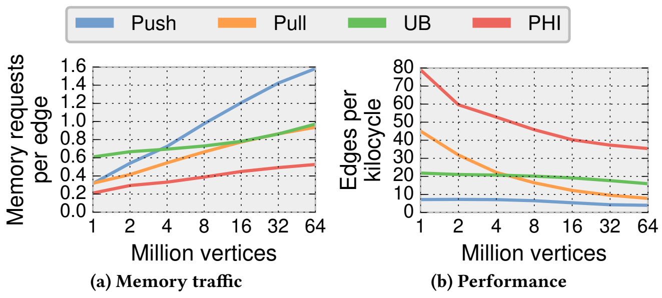


Figure 15: R-MAT graphs with different vertex counts and a fixed average degree.

UB incurs almost a constant amount of memory traffic irrespective of the graph structure. A large fraction of UB's memory traffic is caused by logging updates to main memory. As explained in Sec. 3.1, PHI exploits the locality caused by the graph's structure by coalescing updates in the cache hierarchy and reduces update traffic. Fig. 13 shows that these benefits hold across all graphs: while the best scheme among Push, Pull, or UB changes across graphs, PHI consistently outperforms other schemes. Due to selective update batching, PHI increases destination vertex traffic slightly in return for lower update traffic. Moreover, PHI's memory traffic is only 2× the compulsory traffic.

Performance across inputs: Fig. 14 shows the per-input memory traffic and performance for the four graph applications and *DC*. These graphs shows that the per-input results we have seen extend across applications: PHI achieves the highest performance and the lowest traffic on all inputs and applications, whereas others work poorly on some inputs. PHI improves performance by up to 11× (PR and DC on sk) and reduces traffic by up to 5× (DC on ukl and twi).

Performance across graph sizes and connectivities: To further analyze the effect of graph features on performance, we run PageRank on a wide range of synthetic R-MAT [12] graphs. R-MAT graphs mimic the properties of real-world social network graphs.

Fig. 15 shows the memory traffic and performance on R-MAT [12] graphs of different sizes and a fixed average degree of 16. PHI consistently incurs the lowest memory traffic across all sizes. In small graphs, Pull and Push achieve good locality, whereas UB is the worst due to its high update traffic. As the graph grows in size, UB becomes desirable over Push and Pull, and Push has the highest traffic, which increases at twice the rate as Pull's.

For both Pull and PHI, performance correlates well with memory traffic. For Push and UB, performance changes only slightly across graphs since they are bottlenecked by synchronization and cores respectively.

We also studied memory traffic by fixing the vertex count and varying the average degree from 4 to 32. PHI works best across all average degrees.

4.4 Sensitivity studies

Impact of preprocessing: Fig. 16 shows how graph preprocessing changes memory traffic and performance. We show results with non-preprocessed graphs (None), an inexpensive (DegreeSort [64]) and an expensive (GOrder [54]) preprocessing algorithm. Preprocessing improves temporal locality and reduces memory traffic of all schemes except UB. Moreover, PHI with preprocessing gets very close to the compulsory traffic, indicated by the dotted line.

In terms of performance, Push is bottlenecked by synchronization so preprocessing barely helps. Pull and PHI show consistent performance improvements with preprocessing. Preprocessing hurts

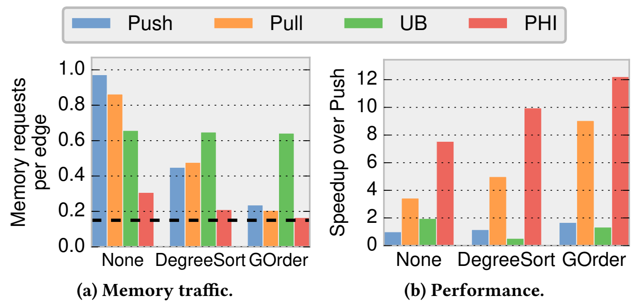


Figure 16: Impact of preprocessing. The dotted line indicates compulsory traffic.

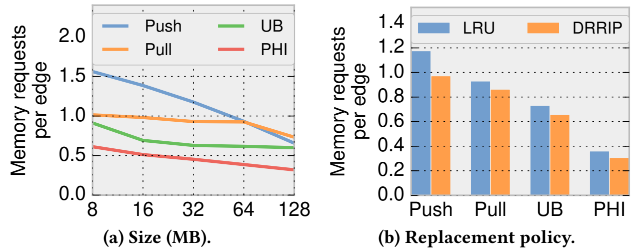


Figure 17: Sensitivity to LLC configuration.

UB's performance since it causes load imbalance (few bins account for a large fraction of edges).

Cache size: Fig. 17a shows the average memory traffic for each scheme at various cache sizes. PHI consistently outperforms other schemes at all cache sizes. With just a 8 MB cache, PHI achieves the same traffic as Push with a 128 MB cache. Moreover, whereas UB barely exploits caches beyond 16 MB, PHI continues to reduce misses throughout the range.

Cache replacement policy: Fig. 17b compares memory traffic with the LRU and DRRIP [25] replacement policies (other experiments use DRRIP). A high-performance policy like DRRIP better exploits temporal locality in the update stream. Thus, PHI with DRRIP coalesces more updates in the cache hierarchy and reduces memory traffic over PHI with LRU. Other schemes benefit similarly from a better replacement policy.

Hierarchical coalescing: We measure imbalance in traffic to LLC banks on graphs preprocessed with degree-sorting. Without hierarchical coalescing, there is up to 2.1× imbalance across banks, while with hierarchical coalescing, the imbalance is limited to 20% and is often negligible.

5 ADDITIONAL RELATED WORK

Accelerators for sparse algorithms: Recent work has proposed specialized accelerators for graph processing for both FPGAs [14, 42, 43] and ASICs [2, 20, 40, 44, 52, 63]. While we evaluated PHI on a general-purpose multicore, its techniques are general and can be applied to accelerators too.

Graphicionado [20] employs graph tiling (also called blocking or slicing) to reduce memory traffic. Tiling is effective for graphs that are moderately (about 10×) larger than on-chip storage, but increasing the number of tiles adds work, and tiling eventually becomes unattractive [10, 39]. To avoid this, Graphicionado needs a large eDRAM that can fit a substantial part of the graph. Unlike tiling, update batching does not perform more work on larger graphs [10]. Thus, PHI does not lose efficiency with graph size and approaches the compulsory memory traffic with small on-chip caches.

GraFBoost [27] is a Flash-based accelerator for external (i.e., out-of-core) graph analytics. Similar to update batching, it logs updates to Flash before applying them in DRAM. It uses hardware-accelerated external sorting with interleaved reduction functions to reduce I/O traffic. PHI uses caches to perform coalescing, reducing traffic without sorting.

OMEGA [1] proposes a hybrid cache subsystem where a scratchpad holds the most popular vertices (identified by degree-sorting the graph) and a conventional cache hierarchy serves requests for other data structures. A specialized unit near each scratchpad performs atomic updates on vertex data.

HATS [38] adds a specialized hardware unit near each core to perform locality-aware graph traversals. HATS and PHI are complementary: HATS improves locality similar to preprocessing, and as we have seen PHI benefits from preprocessing.

Indirect prefetchers: Conventional stream or strided prefetchers are ineffective on the indirect memory accesses of sparse algorithms. Prior work [3, 4, 58] has proposed indirect prefetchers to handle such accesses. While these designs improve performance by hiding memory access latency, they quickly saturate memory bandwidth and become bandwidth-bound. By contrast, PHI reduces memory traffic, making better use of limited off-chip bandwidth. Moreover, PHI works well with stream prefetchers and, since updates do not fetch data, would not benefit from indirect prefetchers.

6 CONCLUSION

We have presented PHI, a *push* cache hierarchy that adds support for pushing commutative updates from cores towards main memory. PHI adds simple logic at each cache bank to buffer and coalesce updates throughout the hierarchy, performs selective update batching to exploit spatial locality, and avoids synchronization overheads.

PHI is the first system to exploit the temporal and spatial locality benefits of commutative scatter updates, On a set of demanding algorithms, PHI improves performance by 4.7× gmean and by up to 11×. Moreover, PHI consistently outperforms pull algorithms, showing for the first time that push algorithms have inherent locality advantages that can be exploited with the proper hardware support.

ACKNOWLEDGMENTS

We thank Maleen Abeydeera, Joel Emer, Mark Jeffrey, Hyun Ryong Lee, Po-An Tsai, Victor Ying, Guowei Zhang, and the anonymous reviewers for their feedback. This work was supported in part by DARPA SDH under contract HR0011-18-3-0007, NSF grants CAREER-1452994 and CAREER-1845986, and a Google Faculty Research Award. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. 2018. Heterogeneous Memory Subsystem for Natural Graph Analytics. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC).*

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd annual International Symposium on Computer Architecture (ISCA-42).*

[3] Sam Ainsworth and Timothy M Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the International Conference on Supercomputing (ICS'16).*

[4] Sam Ainsworth and Timothy M Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. In *Proceedings of the 23rd international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII).*

[5] Vignesh Balaji and Brandon Lucia. 2018. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC).*

[6] Vignesh Balaji, Dhruva Tirumala, and Brandon Lucia. 2017. Flexible Support for Fast Parallel Commutative Updates. *arXiv preprint arXiv:1709.09491* (2017).

[7] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2014. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2014).

[8] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC12).*

[9] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv:1508.03619 [cs.DC]* (2015).

[10] Scott Beamer, Krste Asanovic, and David Patterson. 2017. Reducing Pagerank communication via Propagation Blocking. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[11] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing.*

[12] 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.*

[13] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms* (3rd ed.). MIT press.

[14] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA—A Case Study of Breadth-First Search. In *Proceedings of the 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-24).*

[15] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM TOMS* 38, 1 (2011).

[16] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).*

[17] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. 1983. The NYU Ultracomputer? Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on computers* 2 (1983).

[18] Samuel Grossman and Christos Kozyrakis. 2019. A New Frontier for Pull-Based Graph Processing. *arXiv preprint arXiv:1903.07754* (2019).

[19] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making pull-based graph processing performant. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).*

[20] 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 *Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49).*

[21] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. 2014. Haswell: The fourth-generation intel core processor. *IEEE Micro* 34, 2 (2014).

[22] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations (ICLR-4).*

[23] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. 2010. Remote store programming. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers (HiPEAC).*

[24] Nangate Inc. 2008. The NanGate 45nm Open Cell Library. `http://www.nangate.com/?page_id=2325`.

[25] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA-37).*

[26] Norman P. Jouppi. 1993. Cache Write Policies and Performance. In *Proceedings of the 20th annual International Symposium on Computer Architecture (ISCA-20).*

[27] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th annual International Symposium on Computer Architecture (ISCA-45).*

[28] Richard E Kessler and James L Schwarzmeier. 1993. CRAY T3D: A new dimension for Cray Research. In *Digest of Papers. COMPCON Spring.*

[29] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing indirect memory references with milk. In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques (PACT-25).*

[30] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[31] 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 (WWW-19).*

[32] James Laudon and Daniel Lenoski. 1997. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA-24).*

[33] 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 *Proceedings of the 42nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-42).*

[34] Clémence Magnien, Matthieu Latapy, and Michel Habib. 2009. Fast computation of empirically tight bounds for the diameter of massive graphs. *JEA* 13 (2009).

[35] Jasmina Malicevic, Baptiste Joseph Eustache Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC).*

[36] Frank McSherry. 2005. A uniform approach to accelerated PageRank computation. In *Proceedings of the 14th International Conference on World Wide Web (WWW-14).*

[37] Micron. 2013. 1.35V DDR3L power calculator (4Gb x16 chips).

[38] 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).*

[39] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2017. Cache-Guided Scheduling: Exploiting caches to maximize locality in graph processing. In *AGP'17.*

[40] 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 *Proceedings of the 23rd IEEE international symposium on High Performance Computer Architecture (HPCA-23).*

[41] Michal Nazarewicz. 2012. A deep dive into CMA. LWN, `https://lwn.net/Articles/486301/`.

[42] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA framework for vertex-centric graph computation. In *Proceedings of the 22nd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM-22).*

[43] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A dataflow library for graph analytics acceleration. In *Proceedings of the 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-24).*

[44] 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 *Proceedings of the 43rd annual International Symposium on Computer Architecture (ISCA-43).*

[45] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[46] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator. In *Proceedings of the 24th IEEE international symposium on High Performance Computer Architecture (HPCA-24).*

[47] 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-40).*

[48] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on management of data (SIGMOD).*

[49] Steven L Scott. 1996. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*.

[50] Julian Shun and Guy E Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[51] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2017. Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*.

[52] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *Proceedings of the 24th IEEE international symposium on High Performance Computer Architecture (HPCA-24)*.

[53] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* (2015).

[54] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 ACM SIGMOD international conference on management of data (SIGMOD)*.

[55] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC07)*.

[56] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU architecture. *IEEE Micro* 31, 2 (2011).

[57] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.

[58] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect Memory Prefetcher. In *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*.

[59] Pingpeng Yuan, Changfeng Xie, Ling Liu, and Hai Jin. 2016. PathGraph: A path centric graph processing system. *IEEE TPDS* (2016).

[60] Albert-Jan Nicholas Yzelman and Dirk Roose. 2014. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE TPDS* (2014).

[61] Guowei Zhang, Virginia Chiu, and Daniel Sanchez. 2016. Exploiting semantic commutativity in hardware speculation. In *Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49)*.

[62] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*.

[63] 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 *Proceedings of the 24th IEEE international symposium on High Performance Computer Architecture (HPCA-24)*.

[64] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Matei Zaharia, and Saman Amarasinghe. 2017. Making caches work for graph analytics. *IEEE BigData* (2017).

[65] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. 2018. FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping. *IEEE TKDE* (2018).

[66] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[67] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*.