Software-based Buffering of Associative Operations on Random Memory Addresses

Matthias Hauck*†, Marcus Paradies‡, Holger Fröning*
* Institute of Computer Engineering, Ruprecht-Karls University of Heidelberg, Germany
Email: holger.froening@ziti.uni-heidelberg.de
† SAP SE
Email: matthias.hauck@sap.com
‡ DLR
Email: marcus.paradies@dlr.de

***Abstract*—An important concept for indivisible updates in parallel computing are atomic operations. For most architectures, they also provide ordering guarantees, which in practice can hurt performance. For associative and commutative updates, in this paper we present software buffering techniques that overcome the problem of ordering by combining multiple updates in a temporary buffer and by prefetching addresses before updating them. As a result, our buffering techniques reduce contention and avoid unnecessary ordering constraints, in order to increase the amount of memory parallelism. We evaluate our techniques in different scenarios, including applications like histogram and graph computations, and reason about the applicability for standard systems and multi-socket systems.**

I. Introduction

Modern computer systems are relying on an increase of available parallelism to achieve performance scaling, and technical constraints demand for a continuation of this trend. Besides the growing amount of homogeneous parallelism, such as instruction-level parallelism (ILP), multicore, and SIMD, heterogeneity also increases due to specialized architectures (GPGPU, TPU, FPGA). Similarly, memory is growing in capacity and performance, albeit at a lower rate. Emerging memory technologies like Storage Class Memory (SCM) promise to continue this trend by providing large, persistent memory. However, these improvements come with certain trade-offs regarding memory access latency.

The pervasive use of concurrency, especially multithreading, requires efficient solutions for concurrency control. A well-known concept for this purpose are atomic operations (atomics), which allow lock-free programming. An atomic is indivisible regarding other simultaneously applied operations on the same memory address, which makes atomics particularly suitable for update operations based on a read-modify-write scheme.

Practical algorithms can apply update operations on a single or multiple different shared memory addresses. There are multiple important algorithms that scatter updates across many memory addresses like push-based graph algorithms (PageRank, Breadth-First Search (BFS)[1]), histogram generation, or hash-based aggregation. Because of their cost and to leverage all available system capabilities, there exist dozens of parallel implementations of these algorithms. Simple parallelization schemes often rely on atomics to resolve data dependencies without the overhead of explicit locking, making atomics for these cases useful and appropriate.

However, atomics, as found in common microprocessor architectures like x86 [1] or ARMv8-A [2], usually guarantee more than only mutual exclusion. In particular, they come with ordering guarantees, adhering to the memory consistency model, and are executed sequentially. Even though there are architectures like IBM POWER [3] that support resortable atomics, these atomics usually lack strong progress guarantees.

As a result, atomics have to be executed in order—often even with memory fence semantics—and are blocking in the context of their originating thread. The execution order is serialized, so operations that would use low-latency cache copies might have to wait for operations on slow main memory. Similarly, the lack of strong progress guarantees can lead to many replays in high-contention scenarios. Consequently, even threads with high cache hit rates might observe a memory performance degradation, as the average memory access latency increases and the amount of memory-level parallelism deteriorates. The trends of higher parallelism, and the increasing average memory access latency due to emerging technologies like SCM, suggest that the implications of atomics on overall performance are increasing dramatically.

A well-known technique to tolerate latency is prefetching data into a higher level of the memory hierarchy. When all relevant addresses are already in the cache, the cost of the limited dynamic reorder capability is reduced. The problem of using prefetching is that it needs to be done tens to thousands of cycles before the update operation to ensure that the value is cached. However, prefetching is difficult as the prefetching distance, i.e., the distance from prefetch instruction to memory operation, can be either too long or too short. Thus, it is very desirable to decouple the execution of an update operation from its issue, allowing to optimize prefetching effectiveness.

In this work, we design and analyze software techniques that help to overcome limitations of atomics for associative and commutative updates. This class of operations is commonly used in push-based graph algorithms and is insensitive to the update order. We propose a series of software buffering

[1]The use of a single data structure for duplicate elimination is common.

techniques for update operations to reduce memory contention and to increase memory-level parallelism using prefetching. As the usefulness of such buffering techniques highly depends on the system, including parallelism, memory latency and data-induced memory contention, we put attention on reporting and analyzing applicability constraints. We start with a description of our software-based buffering techniques (section III, section III-D: Implementation), which tolerate the memory access latency of associative and commutative updates in multi-threaded environments.

Second, we analyze performance and applicability of these techniques for different usage scenarios (section IV: Methodology), with varying sizes and types of input data, contention patterns and memory access latency (section V for commodity x86 and DRAM technology of different scale). By using them, we outline how different hardware properties impact the applicability of such buffering techniques. We show how to apply the buffering techniques to two fundamental graph algorithms (BFS, PAGERANK) that rely on associative and commutative updates (section VI). Before we conclude, we briefly discuss applicability and application integration of the proposed buffering techniques (section VII).

II. BACKGROUND

Atomics are one of the fundamental synchronization techniques in modern multi-core CPUs. These operations update a memory location such that the operation appears indivisible.

The x86 instruction set architecture (ISA) [1] provides two types of atomics—direct-fetch and compare-and-swap (CAS). Fetch-atomics apply an indivisible update directly on a memory address, but they are only defined for `integer` values. CAS can be applied to all data types of up to 8 B (with CMPXCHG 16 B). To achieve this, the atomic operation loads a memory address, updates the value and writes this result to the memory address, if the value at the memory address has not been changed in the meantime. If the value has been changed, the CAS operation has to retry. In contrast, a fetch-atomic locks the cache line that will be updated during the complete update from the first load until the result is written to the memory.

In a multi-threaded environment with a single shared address space not only the atomicity of updates is important, but also the order in which they become visible to other threads. Thus, programming languages like C++ [4] provide options to specify in which order atomics can become visible and how they can be reordered. ISAs provide ordering guarantees or mechanisms (e.g., fences) to implement the desired memory ordering. The guarantees made at programming language level not necessarily have to match the guarantees at ISA level, as long as the ISA guarantees are stronger. For example, X86 is restrictive as an atomic cannot be reordered with any other memory operation (loads and stores). As a consequence, even a relaxed atomic at C++ level is often executed with stronger guarantees by the architecture.

To complement automatic hardware prefetching, ISAs like x86 or ARMv8-A provide prefetch instructions to partially or completely hide memory access latency. These prefetch instructions can provide additional information about an optimal cache level, if there is temporal reuse, or which type of operation (read/write) will be executed. Lee et al. [5] provide an overview about software and hardware prefetching and how they can interact.

However, in comparison to a load, a prefetch does not change the state of the program as it only interacts with the cache. When a thread writes to a memory address that another thread had successfully prefetched, but not loaded, the cache coherence protocol simply invalidates the prefetched entry. While load and store operations on x86 are serialized for atomics, nothing indicates in the reference manual [1] that this also holds true for prefetches[2].

III. SOFTWARE BUFFERING FOR COMMUTATIVE AND ASSOCIATIVE UPDATES

Many applications like push-based graph algorithms, histogram computations, or hash-based aggregations perform update operations that are scattered across many memory locations. In parallel implementations, these scattered updates are often realized using atomic operations and can be a root cause of poor performance. To understand this potential performance problem, let us analyze a basic graph processing problem as motivating example.

A. Motivating Example

Graphs that represent the relationship between different vertices can be represented as a simple list of edges (edge list), where each edge is a pair of source and target vertex IDs. For example, a Compressed Sparse Row (CSR) data structure, a commonly used graph representation, includes a prefix sum of the vertex degrees, i.e., the number of incoming or outgoing edges per vertex. The vertex degrees can be obtained by counting the occurrence of the vertex IDs in the edge list. In parallel implementations, multiple threads read portions of an edge list and update the counters of these vertices. This has several consequences:

- Data-driven: the counter to update is not known before the vertex ID is read from the edge list.
- Ordering: besides the update itself, the algorithm does not depend on intermediate counter values, consequently the order of updates is irrelevant.
- Atomicity: because multiple threads could try to update the same counter concurrently, these updates need to be atomic to prevent lost updates.
- Contention: as part of the atomic update, all other cache copies are invalidated. Depending on the data distribution of the edge list, some counters might be heavily updated, which causes cache contention.

Since the algorithm is memory-bound, performance can usually be improved by two approaches: the reduction of cache contention and by hiding memory access latency. Our approach in general is to address these approaches using per-thread

[2]According to our experiments, this applies also to hardware prefetching of normal loads. The (hardware) prefetch of the initial load in a CAS style atomic can therefore improve performance

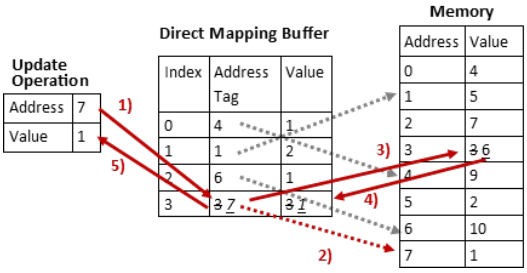


Fig. 1: Activities of the direct mapping buffer during a buffer miss (red) and mapping to target data structure (dashed)

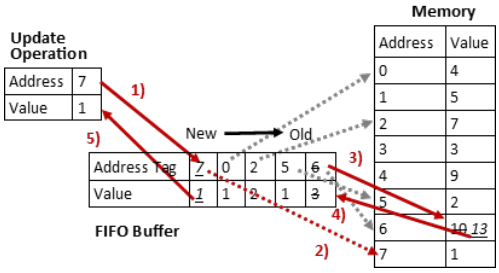


Fig. 2: Activities of the FIFO buffer during a buffer miss(red) and mapping to target data structure(dashed)

buffering techniques. Such techniques exploit the fact that updates scattered across many addresses are often associative and commutative. We propose four major buffering techniques that address these optimization directions: three small buffer techniques (direct mapping buffer, a FIFO buffer, a combination of both), and thread-local, fully replicated data structures.

B. Small Buffers

In this subsection, we present a direct mapping buffer, a FIFO buffer, and a buffer that combines both, which all have some common properties: every thread has its own local, isolated buffer, i.e., within the buffer no synchronizing operations are required. Synchronization operations only become necessary when an (update) entry is evicted from the buffer and applied to memory. In addition, it can be assumed that for reasonable small sizes the buffer fits into L1 cache, while the buffered data structure resides on a lower and slower level (main memory).

1) Direct Mapping Buffer: The main purpose of the direct mapping buffer (cf. figure 1) is to combine updates on frequently used memory addresses and has many similarities with a hardware cache: to access the buffered values fast, a function I maps every memory address A of the buffered data structure to an index of an entry in the buffer. I can be chosen arbitrarily but must be efficiently computable. To distinguish to which address a buffer entry maps, the buffer internally stores an address tag T in addition to the buffered value V for each of the N entries. The memory consumption M of the buffer per thread is therefore $M = N \cdot (T_{\text{size}} + V_{\text{size}})$. Because we assume no prior knowledge about the data, we choose as mapping function $I(A) = A \bmod N$.

In the case of an update the buffer checks if it already has an entry for the related address by applying the mapping function and checking if the address tags match (cf. figure 1 Step 1). If true, the update is applied to the entry value. If false and this entry is used, the old entry is evicted and replaced (cf. figure 1 Step 3 & 4). If there are no updates anymore, the buffer is flushed and all entries are evicted.

Essentially, the direct mapping buffer is a cache with an associativity of 1. In general, it is possible to use a higher associativity or to buffer multiple values per entry (e.g., a full cache line). The problem is to implement such a buffer efficiently in software. A higher associativity requires more address comparisons for every access and a more complex

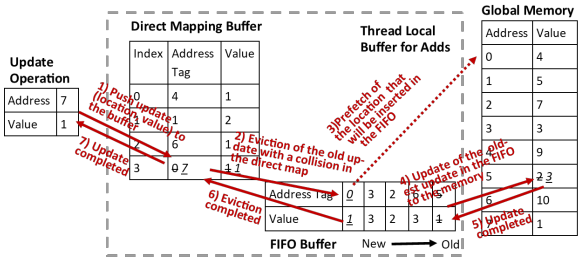


Fig. 3: Activities of the combined buffer during a buffer miss

eviction strategy to decide which entry to replace. Therefore, a possible performance improvement would be easily offset by an increased instruction overhead.

A direct mapping buffer can also be combined with prefetching (cf. figure 1 Step 2). Then, a prefetch instruction is issued when a new entry is inserted into the direct mapping buffer. When later an entry is evicted from the buffer and its update applied to memory, the probability of a cache hit increases.

2) FIFO Buffer: The main purpose of the first-in first-out FIFO buffer (cf. figure 2) is to defer update operations on a shared, memory-resident data structure, such that multiple updates on the same location can be combined and the corresponding address prefetched. New updates are inserted at the end (cf. figure 2 Steps 1 & 5), while old updates at the front of the buffer are applied to the buffered data structure (cf. figure 2 Steps 3 & 4). This allows prefetching memory addresses (cf. figure 2 Step 2) such that the deferred update will likely result in a cache hit.

As all updates are assumed to be associative, late combining in the FIFO buffer is used to combine multiple updates to the same location into one single update. During insert, the already existing entry will be updated instead of inserting a new entry. This reduces the buffer contention, and as a result, entries can remain longer in the buffer and increase the probability of additional combining. The disadvantage is that, for every insert, additional work proportional to the buffer size is required, so only small buffers are reasonable. As each buffer entry needs an address tag and a value, the memory consumption formula is equivalent to the direct mapping buffer formula.

3) Combined Buffer: This technique combines the two previous building blocks: the direct mapping buffer as a front end and the FIFO buffer as an eviction buffer. Figure 3 shows

how both stages interact by depicting the activity sequence and the data movement between the buffer stages, when a buffer miss occurs. This approach combines the advantages of both buffering techniques. The direct map combines updates on often-used addresses to reduce contention, while the FIFO buffer hides latency using prefetches. A disadvantage of this approach is a slightly higher instruction count in case of a miss compared to a direct mapping buffer or FIFO buffer alone. In addition, the memory consumption M is $M = M_{\text{direct mapping}} + M_{\text{FIFO}}$, so it is typically higher then both buffer in isolation.

Note that while both, direct mapping buffer and FIFO buffer, already make use of update combining, which might seem redundant, the combined buffer maintains this. Even though the main purpose of the direct mapping buffer is to combine updates, the FIFO buffer serves as an extension of the (limited) associativity for often-used indices.

C. Thread-Local Fully Replicated Data Structures

For associative and commutative update operations, the main alternative to use small local buffers is to use fully replicated data structures, which replicate the target data structure on a per-thread basis. For hash-based aggregation, this approach already has been used by Cieslewicz and Ross [6]. By using fully replicated data structures, all updates are applied to the local copy without the need of synchronization. When a globally consistent state is required, all thread-local copies are merged.

As a result, fully replicated data structures mainly avoid the use of synchronization, in particular atomics, as updates on replicated structures do not result in data dependencies. Non-atomic updates are typically cheaper than their atomic counterparts as they provide no mutual exclusion or ordering guarantees, which allows reordering and speculative execution. Also, this approach can improve cache usage as other threads will not invalidate cached entries. This concept, however, has multiple downsides, especially when the ratio of updates to update targets is low. The most important one is clearly space complexity, which is linear to the number of threads. For example, we would require $14\,\text{GB}$ for the counter, if we count the occurrence of 64 million IDs in a list using $4\,\text{B}$ counters and 56 threads in parallel. If the list itself has only $32 \cdot 64$ million entries (each $4\,\text{B}$, or a total size of $8\,\text{GB}$), the required space would increase by an additional factor of 1.75 to $22\,\text{GB}$.

The second main disadvantage is the merging of thread-local results into a single, global form. With an increasing number of threads, also the number of local results to be merged increases. Similar, merge complexity increases linearly with the number of elements to be merged. While merging can be typically done in parallel[3], the merge of complex data structures like hash maps is complicated, in particular when done thread-parallel.

However, when an algorithm is iterative and requires the preparation of a new target data structure for each iteration anyways, using fully replicated data structures is usually applicable as the downsides are partially compensated.

[3]In principle the number of threads in the update phase and the merge phase can be chosen independently.

Listing 1: Buffer API Usage.

```
1  TBuffer< // Direct mapping buffer
2          countMap,// Type of buffered data structure
3          Sum, // Update operation
4          FIFO, // Eviction strategy
5          16 // Direct buffer size
6  > countBuffer(
7          edgesPerVertex ); // Buffered data structure
8
9  // Update operation
10 countBuffer(
11         id,  // Entry to update
12         1); // Value to add
```

D. Implementation Details

One of the main objectives of software buffering is an efficient implementation with low overhead, as otherwise performance gains originating to reduced memory access latency can be easily exceeded by instruction overhead.

1) Small Buffers: The software buffers (FIFO, direct mapping, and combined) are realized in C++ using templates (cf. listing 1) to be fast and configurable. The template arguments are used to describe statically how the buffer will be used (buffered object type, update operation), and to define parameters like eviction strategy or buffer size(s). By using templates, the methods of complex operations like *update* can be completely inlined, and arithmetic operations (e.g., divisions) can be replaced with cheaper equivalent operations, if applicable.

All three types of the buffer are realized as combinations of direct mapping and FIFO as eviction strategy. A *dummy write strategy* even allows creating a buffer variant that forwards updates directly to memory. The *direct map* essentially consists of two arrays, one for the address tags and one for the values. Every address belongs to a single index in the arrays, where the index is the remainder of the address divided by the buffer size. The FIFO is essentially a ring buffer that, like the *direct map*, consists of two arrays, one for the address tags and one for the values.

Both, direct map and FIFO, perform prefetching of addresses that will be updated using prefetch instructions. Prefetches are issued when new updates are inserted into the buffer or pushed to the eviction stage for the combined buffer. To guarantee a global state at specific points, the buffers use a flush method, which evicts all entries in the buffer. The flush method is automatically triggered when a buffer gets destructed.

2) Thread-Local Fully Replicated Data Structures: The implementation of the thread-local fully replicated data structures differs from the buffer realization as here every thread owns a complete copy of the data structure. During the main computational phase every thread applies updates to this local structure to generate a partial result. These partial results are merged afterwards.

In our experiments, we only merge `std::vector` instances and restrict the update operations to adds. Therefore, every thread receives an equal-sized ID range of the result vector that it merges using the partial results from all other threads. Every thread then computes the sum of all partial results for all its IDs and stores them in the final result vector.

IV. Micro-Benchmarking Methodology

The focus of our initial experiments is to show, which of the buffer techniques are applicable for a specific scenario and how they address the implications of atomics in terms of memory contention and parallelism. As a test algorithm, we use the degree-counting algorithm as described as follows:

A. Test Algorithm

For our experiments we use a degree-counting algorithm, a variant of histogram computation (cf. section III). We choose this algorithm, because its parameters can be varied almost arbitrarily to model different scenarios for a thorough analysis of different buffering techniques. This algorithm counts the occurrence of vertex IDs in an edge list as source or target vertex, and stores the result in a counter array. When executed in parallel, the input edge list is tiled in non-overlapping partitions of 16k elements each. If this results in fewer partitions than cores, we reduce the number of threads accordingly.

The partitions are dynamically dispatched as work packages to the threads using OpenMP's *parallel for* dynamic scheduling construct. This dynamic scheduling keeps all threads utilized during the algorithm execution even if the elapsed time of different work packages varies. The updates of the counter array are implemented using either direct updates, our buffering techniques, or thread-local, fully replicated data structures. As buffer size we use for direct mapping 16, and for FIFO 8, so by using $64\,\mathrm{bit}$ counter and address tags per thread the buffer have the following size: direct mapping $256\,\mathrm{B}$, FIFO $128\,\mathrm{B}$, and combined $384\,\mathrm{B}$.

B. Test Data

Another important factor is the selection of input data as it determines the update pattern on the counter array, and therefore the amount of contention. As representatives of a skewed data set, which typically exhibits a large amount of contention, we choose RMAT graphs using a parameter set similar to the Graph500 Benchmark [7]. These graphs are scale-free [8], so many vertices have few in- and outgoing edges, while few vertices have many in- and outgoing edges. The number of edges of a vertex is equivalent to the occurrence in the edge list. Because numerous vertices are isolated and have no incoming or outgoing edges at all, we encode the vertex IDs upfront using a dictionary to obtain a dense ID domain. Every vertex with an ID occurs at least once in the edge list.

As non-skewed data sets typically cause low contention, we furthermore generate edge lists, in which all vertices occur as source or target with an identical frequency. The vertices of source and target are randomly shuffled using different random number generation functions.

For our measurements, we generate for both types of test data edge lists at different scale factors (SF), with $|V| = 2^{SF}$, $|E| = 16 \cdot |V|$ and 32 Bit vertex IDs.

C. Test Systems

Nowadays NUMA systems with one or two sockets are commonplace in server environments. Because of the limited number of sockets, there are no expensive remote or multi-hop accesses. We expect systems with such a low memory access latency to be the worst case for buffering techniques, which try to hide memory access latency.

For most experiments, unless noted otherwise, we use a two-socket system equipped with Intel Xeon E5-2660 v4 processors, each with $2 \cdot 14$ cores @$2.0\,\mathrm{GHz}$ and Hyper-Threading enabled, $35\,\mathrm{MB}$ last-level cache for each socket, and $128\,\mathrm{GB}$ DDR4 RAM (System A). In general, we use GCC 7.1 with OpenMP, optimization flags `-O3` and `march=native`. All tests use Linux systems without the kernel-level page table isolation patch[4]. Besides availability reasons, our code does not cause many context switches, syscalls or I/O, so we do not expect a significant impact.

D. Experimental methodology

We measure the elapsed time from the point where all necessary data structures are initialized until the algorithm completed and is able to return a result. The elapsed time for every configuration of buffering technique and input data is measured multiple times, so that the combined elapsed time equals $60\,\mathrm{s}$, with a maximum of 120 runs and a minimum of 8 runs. The reported elapsed time, update rate, and other values derived from it, is the mean value of these runs. For a description of the computing system, please refer to the experiments.

V. Experiments

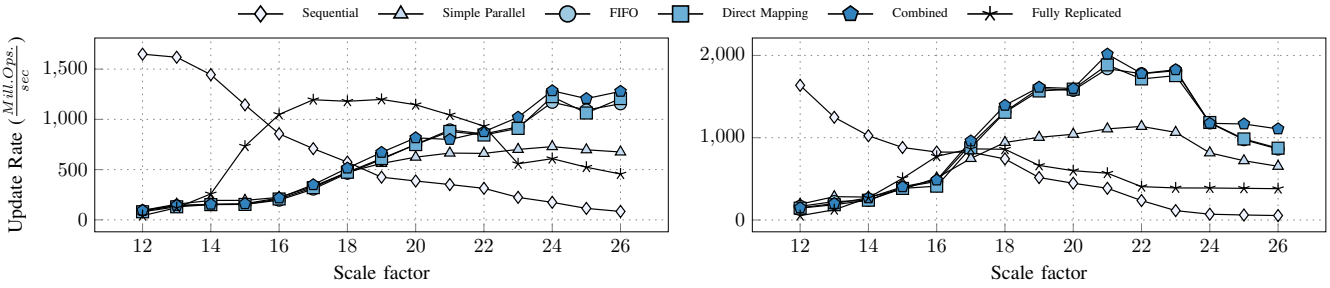
As part of micro-benchmarking, we evaluate the degree counting workload on a standard CPU system (System A). Our expectation is to observe three types of effects besides performance improvements: cache capacity effects, contention effects, and overhead-related effects. Besides the previously introduced four buffering techniques (Direct Mapping, FIFO, Combined and thread-local fully replicated data structures), we furthermore compare against a plain sequential implementation and a simple parallel implementation that uses no buffering.

Sequential and fully replicated data structures have low instruction overhead and are not prone to contention due to updates from other threads. Both should be able to exploit caches well, but they are also affected by increased memory access latency in the case of capacity misses. For the simple parallel implementation, we expect a similar behavior except that it will be affected by contention.

The buffers in contrast have a high overhead as they need to check buffer contents for every update. In addition, there is update contention, which increases access latency as there is only one global data structure that will be updated. As all buffers are capable of hiding latency, capacity misses when the main data structure exceeds cache size should be damped.

Regarding the data sets, we expect that they will cause different cache capacity and contention effects: the skewed data set (RMAT) will cause contention and its often-updated part is much smaller than the complete data structure. The update pattern of the non-skewed data set (EQUAL) has virtually no hot spots that cause contention.

[4]See https://lwn.net/Articles/741878/ for details.



(a) RMAT graphs (highly skewed)

(b) EQUAL graphs (not skewed)

Fig. 4: Update rate for the degree counting micro-benchmark (SYSTEM A)

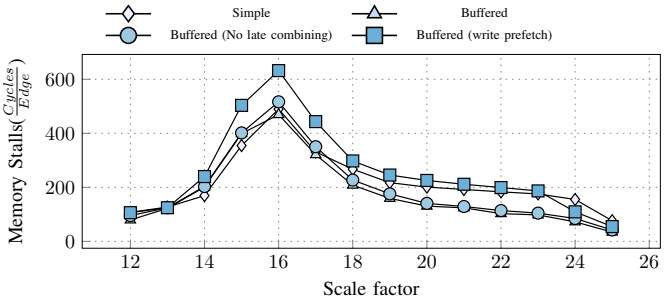


Fig. 5: Memory stall cycles per edge on an RMAT graph for the degree counting micro-benchmark (SYSTEM A)

A. Skewed Input Data (RMAT)

In our experiments with a skewed data set (cf. figure 4a) we can see that the behavior related to the input size can be divided in three different regimes: up to SF 16 with sequential being fastest, from SF 17 to SF 23 with fully replicated data structures being fastest, and above SF 23 with one of the small buffers (FIFO, direct mapping, combined) being fastest.

All three buffers show a similar performance behavior (cf. figure 4a), with a small advantage for the combined buffer. This is an expected behavior as they process data in a similar fashion. The FIFO acts as a fully associative cache, which reduces contention, because our FIFO implementation combines newly inserted updates with updates to the same address that are already in the buffer (late-combining). The direct mapping buffer on the other side hides latency by deferring updates and prefetching memory addresses. Between insertion and eviction, an update stays for some time in the direct mapping buffer. The combined buffer performs better as it combines the advantages of both approaches at a higher efficiency.

We expect that the cache size and contention effects have a significant impact on performance. At SF 16 this data set has a size of 365 kB, and at SF 23 a size of 35 MB, which is close to L2 and L3 size of 256 kB respectively 35 MB. The cache boundaries can therefore be an explanation for the border between the second and the third regime. This also explains why the performance of the fully replicated data structures approach drops earlier, as it normally utilizes more cache.

Another important fact for the explanation of the regime boundary at SF 16 is that the number of generated work packages starts to exceed the number of CPU threads. The memory stall cycles per edge are reported in figure 5, and these results suggest that memory access cost peeks at SF 16. It is reasonable to assume that the contention is high, caused by the large number of threads and the small size of the main data structure. This assumption is supported by the fact that the access latency increases further, when the write prefetch is used instead of a normal prefetch, which prepares a memory address for an update. In addition, it explains why the simple approach and the buffer both have a performance worse than the other approaches. Finally, for increasing graph sizes, the contention decreases and subsequently the performance increases.

B. Non-Skewed Input Data (EQUAL)

Results in terms of update rate for the non-skewed data set (cf. figure 4b) show a different behavior, in spite of similar points in terms of regime separation. For this input data, the size of the counting structures is 512 kB for SF 16 and 64 MB for SF 23. Thus, SF 16 requires twice the size of the L2, but SF 23 fits in the combined L3 caches of all sockets. In general, we can still apply the same reasoning for the boundaries as for the skewed data.

For sizes of SF 17 and larger, the buffered approaches are superior to all others, thus these results suggest to avoid using the fully replicated data structures approach on non-skewed data. For sizes of SF 17 or smaller, the sequential approach performs best, similar to results on skewed data.

Noticeable are two observations for SF 17 and larger: first, the poor performance of the fully replicated data structures approach can be explained by much less spatial and temporal reuse, as for non-skewed data updates are scattered uniformly across the complete data structure. Second, here the other buffer techniques are much faster compared to the performance using skewed data. The primary reason here is that the contention decreases so much that it virtually does not exist. This allows the buffers to fully utilize their prefetching capabilities to hide memory access latency.

C. Memory access latency and contention

Important factors for the atomic update performance are memory access latency and contention, which are prevalent for multi-socket systems due to inter-socket communication and cache coherence protocol overhead. We expect that the buffer

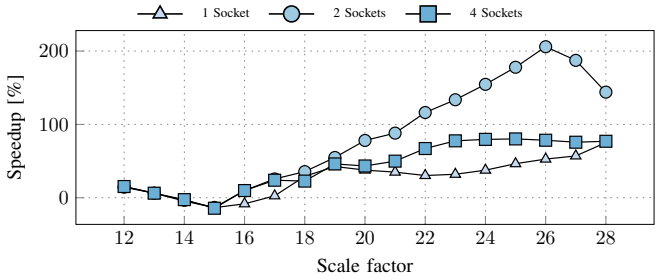


Fig. 6: Speedup of combined buffer compared to a simple parallel update scheme on RMAT graphs (SYSTEM B)

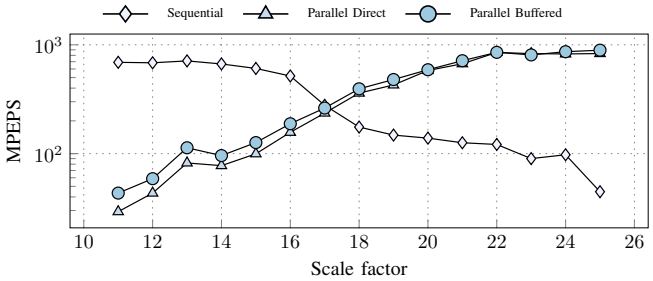


Fig. 7: Update throughput in millions of processed edges per second (MPEPS) for push-based PAGERANK on RMAT graphs (SYSTEM A)

is able to compensate parts of such a latency increase, therefore being inline with scaling and future technology trends.

However, this latency compensation is limited for two main reasons: first, there is inter-socket cache coherence overhead as additional coherence activity is required to lock the cache line during the update. Second, the bandwidth between the sockets is limited and the amount of transported data increases as some of the additional prefetches might be futile.

To analyze such a scenario, we run our experiments on SYSTEM B with four Intel Xeon CPU E7-8870 v4 (Broadwell) CPUs, each with 20 Cores/40 Threads @ 2.1 GHz. Despite the fact that the four sockets are fully connected, the impact of the interconnect can be higher than on SYSTEM A as more parts of the target data structure might be stored in caches of other sockets. In figure 6 we report the sustained throughput for an increasing amount of active sockets, using the best configuration of the combined buffer. The number of active threads matches a full utilization of the used sockets, which also increases contention.

We can see that there is a substantial performance improvement of up to 200% for the combined buffer on multiple sockets. For this system, the sweet spot is a 2 socket configuration. For 4 sockets there are fewer benefits, probably because a larger core count results in more performance in terms of concurrent update operations, which again is advantageous for the simple approach, while it cannot be utilized by the buffering technique which saturates the inter-socket links earlier. Like in the previous experiments (cf. figure 4a), there is an advantage for all socket counts especially for large graphs.

VI. APPLICATION TO OTHER ALGORITHMS

In the previous section we showed that software buffering can improve performance of a simple histogram-like computation by up to 89% on the skewed data and 82% on the non-skewed data (SYSTEM A). Furthermore, we showed the applicability of this concept under increasing memory access latency. The combined buffer has proven to be very effective, and in contrast to the fully replicated data structures approach only imposes a very low memory overhead.

We now apply the software buffering to more complex applications, which requires an adapted use of the buffer and potentially introduces new constraints and preconditions. Furthermore, the performance improvement can differ as there is significant work that can overlap with update operations. In this section, we outline the use of small software buffers in the context of push-based versions of two popular graph algorithms: PAGERANK and breadth-first search (BFS).

Listing 2: Algorithmic core of a push-based PAGERANK

```
1  auto& neighbourhood = adj[vertex];
2  double delta = (damping * ranks[vertex]) /
3           static_cast<double>(neighbourhood.size());
4  for (auto targetVertex : neighbourhood){
5      //atomicIncrement(newRanks[targetVertex], delta); //
           not buffered
6      buffer(targetVertex, delta); // buffered
7  }
```

A. PAGERANK

The iterative, push-based PAGERANK algorithm is an ideal example where software buffering can be directly applied. As depicted in listing 2, the algorithm computes a per-vertex update and subsequently pushes this value to all its neighbors (line 5). To use a buffer, it is sufficient to buffer the push operation of the values to the neighbors (line 6). The buffer itself needs to be set up only once per worker thread, with each worker processing one or more work items of multiple vertices. At the end of each iteration, the buffers have to be flushed.

We implemented PAGERANK in the context of SAP HANA using its Jobs Executor Framework [9], and a parallelism scheme that uses optimized work packages to utilize all cores. The graph is represented as an adjacency list using a vector of vectors. As in the vertex degree counting example, the vertex IDs are dictionary encoded by the appearance of the IDs in the edge list. We do not use any enforced order like a BFS-order on the IDs that could further enhance the data locality[5].

The atomic update is realized by a combination of a normal load, a floating-point add, and a CAS operation as x86 does not directly support atomics for double floating-point values. As buffering technique we select a combination of a direct map with 16 entries and a FIFO with 8 entries. We evaluated this buffer configuration on SYSTEM A on several RMAT graphs of different size, with a damping factor of 0.85 [10] and a maximum error threshold of 10^{-6} as PAGERANK parameters. The RMAT graphs are undirected and are similarly generated

[5]Vertex ID orders like BFS-order can in practice improve the (sequential) performance of an algorithm as they can increase locality. The problem of reordering is that even simple techniques can take longer then the algorithm that they should accelerate.

as described in section IV-B, but with an edge factor of 32. We measure the whole execution of a full PAGERANK run after the creation of the adjacency list and its dictionary, from the setup of all supporting data structures until the algorithm converges and the result vector can be returned. Ten repeated full PAGERANK runs are executed and measured, and we report mean throughput in Processed Edges per Second (PEPS).

What we can see in figure 7 is that we achieve a performance improvement of more than 30% for small graphs, which likely contain update contention. For larger graphs, this improvement decreases as the contention also does, but even in this case the buffer causes no substantial disadvantage. Beyond SF 17, where parallel execution has a performance advantage over sequential execution, we observe a maximum improvement of 12% for SF 19 and a minimum of -2.7% for SF 23. SF 23 seems to be a local minimum as we see an improvement of 7.4% for SF 25.

B. BFS

The use of the buffer inside a push-based BFS algorithm is more difficult as it requires additional algorithmic engineering, which includes reduction on the prerequisite's strength for the buffer usage. A BFS, which uses a *visited* bitmap to eliminate duplicate discoveries of vertices, allows to perfectly buffer the updates to the *visited* bitmap.[6] For a block of 64 vertices inside of the visited bitmap, all potential updates are gathered in the buffer together with the address prefix of these vertices. When it is necessary to apply them to the bitmap, it is done once for the block with a single "atomic or".

As some bits set during the block update might have been already set previously, the set of newly marked vertices has to be determined.[7] We present the code in listing 3. The newly marked bits are the difference of the update block and the state in the bitmap before the update, which can be obtained as a byproduct of a CAS-based "atomic or" implementation[8]. However, this reduces the strength of buffering as the algorithm now has a dependency to a global representation of the buffered visited data structure at update time.

Similar to PAGERANK, we implemented a parallel, level-wise, push-based BFS, which computes the hop distance to each reachable vertex. As buffering technique, we select a combined buffer with 32 entries for the direct map, 6 entries for the FIFO, and late combining. We check before each (buffered) update of the global bitmap if an entry is already set, in order to reduce the number of updates on the global bitmap and accompanying cache invalidations. This optimization changes the majority of operations on the global bitmap to reads as there are many edges per vertex. We evaluate our implementation on SYSTEM A and SYSTEM B on RMAT graphs of different size. We measure the execution of the BFS algorithm after the construction of the adjacency list including the construction and destruction of all supporting data structures.

Listing 3: Block-wise update on visited bitmap with estimation of updated vertex.

```
1  atomic<uint64_t>& visitedEntry = bfsData.
       discoveredVertices[blockToUpdate];
2
3  //check if new vertex are discovered
4  uint64_t blockToUpdatePrevious = visitedEntry.load();
5  if ((blockToUpdatePrevious | blockUpdates) !=
       blockToUpdatePrevious){
6          blockToUpdatePrevious= visitedEntry.fetch_or(
               blockUpdates);
7
8      // detect bits that are really new
9      uint64_t changed = (~blockToUpdatePrevious) &
           blockUpdates;
10     [...]// regeneration of updated vertex ids and
           copying to next queue
11 }
```

The BFS algorithm has been executed and measured 250 times, and we report the throughput derived from the mean of these measurements in Traversed Edges per Second (TEPS). As the number of reads at runtime dominate and updates only interfere with them, and due to a higher share of overheads, the performance improvements are smaller compared to PAGERANK. On SYSTEM B for large graphs with scale factors 20 to 26, a speedup between 2% and 15% can be reached, similar to previous results. For small to medium size graphs with scale factors 12 to 18, a speedup of 2.5% to 4.5% is achieved. Due to low thread numbers and high locality for these graphs, also slowdowns between 2 and 8% can be observed.

On SYSTEM A the improvements are small as the number of physically available threads is lower and subsequently the exploitable contention. For large graphs with scale factors 20 to 26, the speedup ranges between 2% and 9%. For small to medium size graphs with scale factors 12 to 16, a speedup of 1.8% to 6.9% can be reached. We see for medium graphs with scale factors 17 to 19 slowdowns between 3% and 9%, due to low thread numbers and high locality for these graphs to a greater extend compared to SYSTEM B. In addition there are outlier in the areas with performance improvements with small slowdowns (SF 13: -2.6%, SF 22: -0.2%, SF 26: -2.3%).

VII. DISCUSSION

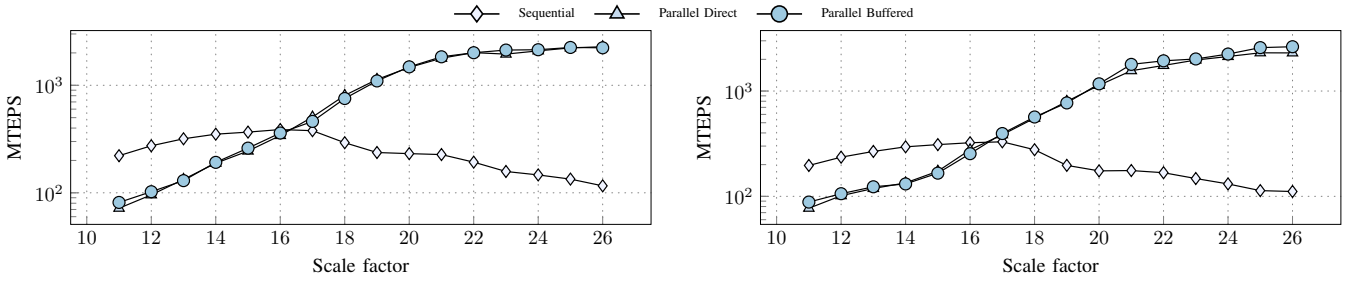
The most important observation is that software buffering can improve the performance of concurrent updates. According to our measurements on a low-latency system, buffered approaches like the combined buffer outperform standard parallel approaches, whenever a parallel execution has an advantage over sequential execution (i.e., when the buffered data structure exceeds last-level cache size). This improvement exists also for different types of update patterns, skewed or non-skewed, so it is also suitable for upfront unknown patterns. Additional costs for buffering techniques are mainly the memory consumption of the buffer itself. As this is independent of the size of the buffered data structure, the buffering techniques are perfectly suited for memory-constrained environments.

The main idea of the buffers is to either eliminate updates on global data structures by local buffering or to reduce the costs of atomic updates by prefetching. According to our evaluation

[6]Other implementations like [11] filter duplicates by checking set values in other data structures like a level map that are generated during a BFS run.

[7]There are BFS variants like the implementation related to Merrill et al. [12] that produce and tolerate a few duplicates as their goal is not to produce a list of unique vertex IDs in level order.

[8]The baseline implementation uses the fetch-atomic "lock bts"(bit test and set) to perform the update on the visited data structure.



(a) SYSTEM A (b) SYSTEM B

Fig. 8: Traversal throughput in millions of traversed edges per second (MTEPS) for push-based BFS on RMAT graphs

in section V, these expectations are fulfilled. Our approach is also capable of tolerating higher latency (cf. section V-C), but buffering parameters like the FIFO queue size have to be tuned for the particular hardware configuration.

Furthermore, the fully replicated data structures approach provides a performance advantage when sufficient work and contention exist, but likewise a significant part of the updated data structure has to fit into the cache. In this case, it takes advantage from the low per-update overhead, similar to the sequential approach. Furthermore, in high-latency settings it can improve performance by exploiting locality (same NUMA node), or leveraging fast DRAM caches temporally. However, fully-replicated data structures increase memory consumption proportionally to the number of threads and the total size of the updated data structure, so the applicability of this approach is highly limited.

VIII. RELATED WORK

In the literature there are several approaches to speed up updates on shared objects in parallel scenarios. We divide them into those rather based on software respectively hardware concepts:

A. Software-Focused Related Work

The most effective way in many cases is to avoid such updates at all. In the area of graph processing it is often possible to formulate an algorithm in a push or pull variant [13], [14], [15]. Push-based algorithms typically compute an update on a per-vertex base and scatter the result across the neighborhood of the vertex. Since the scatter operation is often performed in parallel, it requires synchronization techniques, such as atomics.

In contrast, pull-based graph algorithms gather data from the neighborhood of a vertex and update values for this vertex locally. Because the pull variant is local, it does not require synchronization while updating vertices. In practice, synchronization is only one factor of overall performance, so there are situations, e.g., because of the graph topology, where the performance of the pull-based algorithms is inferior to their push-based counterparts. This is the reason why prior work often recommends a hybrid approach, which combines the advantages of push-based and pull-based processing.

In the area of numerical optimization for machine learning, Niu et al. [16] propose a parallel stochastic gradient descent (SGD) update schema that is lock-free and does not use atomics as it accepts lost updates due to a lack of atomicity. They argue that the cost function of many machine learning problems that the SGD algorithm minimizes is rather sparse. Thus, individual SGD steps that are executed concurrently effect only small parts of the result, what makes lost updates insignificantly rare.

In the area of aggregations for relational databases, Cieslewicz and Ross [6] investigate hash-based aggregation on a single Sun UltraSPARC T1 that provides 32 threads/8 cores without out-of-order execution and limited cache (8 kB L1/ 3 MB shared L2). Similar to this work, they analyze the behavior of aggregation using a single shared hash table and a hash table per thread. In addition, they propose an approach they call "hybrid aggregation" that uses a small hash table per thread, which evicts aggregated values to a shared hash table, when it is full, and they propose an adaptive framework, which selects the best approach depending on the data to aggregate. The main difference to the small buffers of our approach is that they use entire hash tables per thread, in combination with a FIFO-based eviction policy for the buckets. Besides this difference in data structures, they furthermore do not use prefetching techniques and the used CPU requires different trade-offs than our multi-socket x86 CPUs.

Apart from accelerating updates on many shared objects, several techniques have been proposed to enhance updates and accesses on a single shared object, like a queue. One example for these techniques are delegates [17]. This approach uses a dedicated thread—the delegate—, which handles all accesses to a shared resource. All other threads send updates targeting the shared resource to the delegate, which processes them locally without additional synchronization. The advantage of this approach is that no synchronization is necessary when accessing the shared resource. A clear disadvantage is that it is not tailored towards handling independent updates on a large range of shared resources, and that the delegate can quickly become a bottleneck.

B. Hardware-Focused Related Work

There are also many hardware concepts to accelerate parallel synchronization. The BlueGene/Q architecture [18] provides several optimizations to the POWER architecture. One part of these optimizations is a set of specific atomic integer update operations that are pushed down to L2 cache slices. If multiple threads try to concurrently update addresses of the same slice

using these atomics, the updates will be sequentially applied, without cache invalidation if the updates address the same cache line. Then, the only overhead caused by update contention originates from the mandatory serialization of updates.

Additionally, the BlueGene/Q architecture has two hardware transactional memory modes that are supported by the L2 cache: the first one is a transaction memory mode, which allows threads to perform updates in isolation and makes them visible at once. If a conflict between multiple concurrent sessions occurs, the hardware can either arbitrarily choose to abort transactions, or allow the software to choose which transaction succeeds or is aborted. The second mode is called speculative execution mode and ensures that the execution of multiple parallel threads fulfills an expected sequential execution behavior as the hardware tracks data dependencies.

Not only CPUs provide atomic operations, but also modern GPUs. NVIDIA GPUs [19] for example provide a variety of atomic operations, e.g., add and CAS, that can operate on integer and floating-point (since compute capability 6.0) values using a relaxed memory model. Historically, GPUs have been added to systems as accelerators with their own address space, but current multi-GPU systems and technologies like NVLink allow a GPU to share its address space with other devices. To keep the cost for atomic operations low when there is no need to synchronize across devices, recent NVIDIA GPUs provide scoped atomic operations that guarantee atomicity on the scope of a thread block, a single device, or the entire system.

Finally, Schweizer et al. [20] provide an in-depth analysis of the behavior of CAS and fetch-atomic on different X86 architectures, including a performance model for atomics. They conclude that CAS and fetch-add atomics essentially have the same performance as long as there is no "wasted work" by the CAS operation. Another result is that atomics prevent ILP, which is one of the main motivations for this work.

IX. Conclusion

In this paper we show that parallel atomic update operations can be accelerated using software buffering techniques. The precondition is that updates have to be associative and commutative and need to be spread across a range of addresses.

The most commonly found technique for such problems is the use of thread-local, fully replicated data structures that are later merged, which can significantly improve update performance because it avoids atomics completely. The cost is that it requires a high ratio of updates to target structure size, and that memory consumption increases linearly with the number of threads and the size of the buffered data structure.

As an alternative, we introduce the concept of small, thread-local software buffers. In particular, the size of such buffers is much smaller than thread-local, fully replicated data structures, and it is not sensitive to update rate or pattern. Experimental results show that our technique can substantially improve performance, in particular for data sizes exceeding cache capacity and highly parallel executions. We also demonstrate that our technique is capable of tolerating an increasing memory access latency, as commonly found in multi-socket systems. Finally, our experiments with graph computations demonstrate the applicability to other real-world applications.

In conclusion, software-based buffering can be beneficial to the overall throughput, however, the right buffering technique depends on aspects including memory-level parallelism, problem size (target structure, number of updates), concurrency, and last but not least memory latency.

Acknowledgments

We would like to thank Intel and our colleagues Romans Kasperovics, Wolfgang Lehner, Roman Dementiev and Ismail Oukid for their support. The author Marcus Paradies was affiliated with SAP SE during the creation of this work.

References

[1] *Intel®64 and IA-32 Architectures Software Developer's Manual*, Intel, Oct. 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[2] *ARM®Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*, ARM Limited, Dec. 2017.

[3] *Power ISA Version 3.0B*, IBM, Mar. 2017.

[4] *Standard for Programming Language C++*, ISO/IEC Std. 14 882:2017.

[5] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012.

[6] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *33rd International Conference on Very large data bases*. VLDB Endowment, 2007, pp. 339–350.

[7] Graph 500 Steering Committee. Graph 500 benchmark 1 ("search"). [Online]. Available: http://www.graph500.org/specifications

[8] C. Seshadhri, A. Pinar, and T. G. Kolda, "An in-depth study of stochastic kronecker graphs," in *11th International Conference on Data Mining (ICDM)*. IEEE, 2011.

[9] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki, "Task scheduling for highly concurrent analytical and transactional main-memory workloads," in *4th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2013.

[10] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[11] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *22nd annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 303–314.

[12] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[13] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, 2013.

[14] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned," in *European Conference on Parallel Processing*. Springer, 2015.

[15] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017.

[16] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, pp. 693–701.

[17] S. Roghanchi, J. Eriksson, and N. Basu, "ffwd: delegation is (much) faster than you think," in *26th Symposium on Operating Systems Principles*. ACM, 2017.

[18] R. Haring, M. Ohmacht, T. Fox, M. Gschwind *et al.*, "The ibm bluegene/q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[19] *CUDA C Programming Guide*, v9.2 ed., Nvidia, May 2018. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[20] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015.