2015 International Conference on Parallel Architecture and Compilation

BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models

Joo Hwan Lee
School of Computer Science
Georgia Institute of Technology
Atlanta, United States
joohwan.lee@gatech.edu

Jaewoong Sim
School of ECE
Georgia Institute of Technology
Atlanta, United States
jaewoong.sim@gatech.edu

Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
Atlanta, United States
hyesoon.kim@gatech.edu

***Abstract*—Parallel machine learning workloads have become prevalent in numerous application domains. Many of these workloads are iterative convergent, allowing different threads to compute in an asynchronous manner, relaxing certain read-after-write data dependencies to use stale values. While considerable effort has been devoted to reducing the communication latency between nodes by utilizing asynchronous parallelism, inefficient utilization of relaxed consistency models within a single node have caused parallel implementations to have low execution efficiency. The long latency and serialization caused by atomic operations have a significant impact on performance. The data communication is not overlapped with the main computation, which reduces execution efficiency. The inefficiency comes from the data movement between where they are stored and where they are processed.**

In this work, we propose Bounded Staled Sync (BSSync), a hardware support for the bounded staleness consistency model, which accompanies simple logic layers in the memory hierarchy. BSSync overlaps the long latency atomic operation with the main computation, targeting iterative convergent machine learning workloads. Compared to previous work that allows staleness for read operations, BSSync utilizes staleness for write operations, allowing stale-writes. We demonstrate the benefit of the proposed scheme for representative machine learning workloads. On average, our approach outperforms the baseline asynchronous parallel implementation by 1.33x times.

***Keywords*-Iterative Convergent Machine Learning Workloads; Bounded Staleness Consistency Model; Asynchronous Parallelism; Atomic Operation.**

I. Introduction

Machine learning (ML) workloads have become an important class of applications these days. ML provides an effective way to model relationships in physical, biological, and social systems, and thus is actively used in a variety of application domains such as molecular models, disease propagation, and social network analysis. As more data becomes available for many tasks, ML is expected to be applied to more domains, thereby making efficient execution of ML workloads on architectures increasingly important.

In ML, the key phase is *learning*; ML inductively learns a model by examining the patterns in massive amounts of data. This requires significant amounts of computation and thus easily takes several days or even months with sequential execution on a single node. As such, most of ML workloads are parallelized to be executed on large-scale nodes, and prior work [1], [2], [3], [4], [5], [6] has mainly focused on reducing the synchronization cost among multiple nodes by utilizing asynchronous parallelism.

While a considerable amount of efforts has been focused on inter-node synchronization in large-scale nodes, there has been little focus on the performance of ML workloads on a single node. As we will discuss in Section II, however, inefficient execution of parallel ML workloads within a node under-utilizes the performance potential on multi-threaded architectures and reduces overall performance. In this work, we focus on single-node performance.

We observe that the greatest inefficiency within a node also comes from a synchronization overhead, which is implemented with atomic operations on a single-node machine such as shared memory processors. In parallel ML implementations, atomic operations are typically used to ensure the convergence of lock-free iterative convergent algorithms. However, atomic operations occupy a large portion of overall execution time and become the biggest inefficiency within a node. The inefficiency comes from non-overlapped synchronization with the main computation thus wasting computing time.

The recent emergence of 3D-stacking technology has motivated researchers to revisit near-data processing architectures targeting the benefit both from the efficiency of the logic layer and data movement cost reduction [7], [8], [9]. In this paper, we propose Bounded Staled Sync (BSSync), hardware support integrating logic layers on the memory hierarchy to reduce the atomic operation overhead in parallel ML workloads. BSSync offloads atomic operations onto logic layers on memory devices, to fully utilize the performance potential of parallel ML workloads. Atomic operations are now overlapped with the main computation to increase execution efficiency. BSSync revisits the hardware/software interfaces to exploit parallelism.

BSSync utilize a unique characteristic of iterative convergent ML algorithms. ML workloads start with some guess as to the problem solution and proceed through a number

1089-795X/15 $31.00 © 2015 IEEE
DOI 10.1109/PACT.2015.42

of iterations until the computed solution converges. The key property of parallel implementations for such ML workloads is that workers on each core are allowed to compute using stale values in intermediate computations. BSSync utilizes this property to reduce atomic operation overhead.

In summary, the key contributions of our work are as follows:

1) We evaluate ML workloads within a single node, unlike previous works that focus on communication latency between nodes.
2) We observe that the atomic operation overhead causes inefficiencies within a single node for parallel ML workloads. We quantify how much the overhead contributes to the overall execution time.
3) Compared to previous works that allow staleness for read operations, BSSync utilizes staleness for writes, allowing stale-writes that accompany simple logic layers at the memory hierarchy.
4) We propose hardware support that reduces the atomic operation overhead. Our evaluation reveals that our proposal outperforms a baseline implementation that utilizes the asynchronous parallel programming model by 1.33x times.

The rest of the paper is organized as follows. Section II provides the motivation for our proposal and identifies the key performance bottleneck as the atomic operation overhead. Section III presents our mechanism, BSSync. After presenting our evaluation methodology in Section IV, Section V evaluates our mechanism. Section VI discusses related work. Section VII concludes the paper.

II. Background and Motivation

In this section, we describe the benefit of utilizing asynchronous parallelism within a single node as background and then show the performance overhead of the atomic operations in ML workloads.

A. Asynchronous Parallelism

Exploiting asynchronous parallelism has the benefit of addressing the issue of workload imbalance between threads that introduce a significant overhead for iterative convergent ML workloads [5]. For the Bulk Synchronous Parallel (BSP) model [10], all threads must execute the same iteration at the same time, and barrier synchronization is used at every iteration to guarantee that all running threads are in the same phase of execution. On the contrary, with asynchronous execution, threads can perform computation instead of waiting for other threads to finish [11].

Figure 1 illustrates the problem of wasted computing time for the BSP model due to straggler threads. The white arrows represent the wasted computing time and the gray arrows represent when each thread performs computation. With barrier synchronization, each thread waits for the others at every iteration. As such, even when only a single straggler thread has not reached the barrier, other threads must stay idle as they wait for the straggler thread to finish before they can start the next iteration, thereby leading to wasted computing time. Relaxing the barrier can reduce the stall time, which leads to significant speedups for a variety of iterative convergent ML workloads.

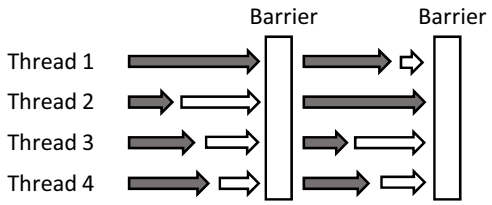


Figure 1: Straggler Problem of the BSP Model.

The performance of iterative convergent ML workloads is determined by how much the solution has progressed within a certain time, which is the product of both 1) the number of iterations per time and 2) the progress per each iteration. A small iteration difference between threads can lead to the large progress per iteration. Relaxing the barrier yields more iterations per time but lowers the progress per iteration, therefore increasing the number of iterations required to converge to the final solution.[1]

The Stale Synchronous Parallel (SSP) model [5] is a type of programming/execution model that makes use of asynchronous parallelism. Figure 2 shows a pseudo-code example of parallel ML workloads utilizing the SSP model. At a high level, a loop iteration consists of five stages of operation. First, a loop iteration starts with read operations fetching inputs (stage 1), followed by the computation on the inputs to generate new data (stage 2). The read operations may fetch stale values, and the stale values can be used for computation. Then, after executing an atomic update operation to store a new computation result (stage 3), a synchronization operation is performed (stage 4). Unlike the barrier operation in the BSP model, the synchronization operation is used to guarantee that the iteration counts of different threads are within the specified ranges. With the user-specified staleness threshold s, the fast thread should stall if not all threads have progressed for enough iterations; the thread should wait for slower threads until they finish iteration $i - s$. At the end of the iteration, a convergence check is performed (stage 5). If the values have not converged, the threads proceed to another iteration. The computed results from the iteration are used as inputs for the other threads and for the later iterations from the thread. This process continues until the computed values converge.

The SSP model provides the benefit of both the BSP and asynchronous execution models for iterative convergent ML

[1]Iterative convergent ML workloads continue to iterate while the computed solution keeps changing from iteration to iteration. Convergence check performs whether or not the solution has converged (unchanged).

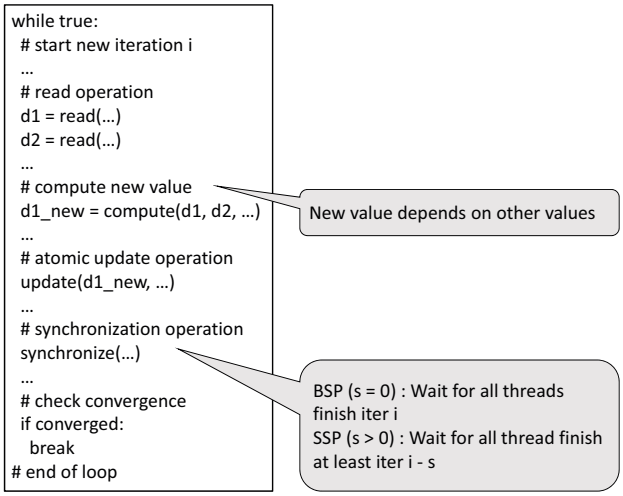


Figure 2: Stages of ML Workloads.

workloads. It alleviates the overhead of straggler threads because threads can perform computation instead of waiting for other threads to finish. At the same time, the bound on staleness enables a faster progress toward the final solution within an iteration. We utilize the SSP model in our evaluation for asynchronous parallel workloads. Figure 3 compares the performance of the BSP model and the SSP model with a staleness threshold of two. The staleness threshold is selected through our experiments to find the value that provides the best speedup. Figure 3 shows that the SSP model outperforms the BSP model by 1.37x times.

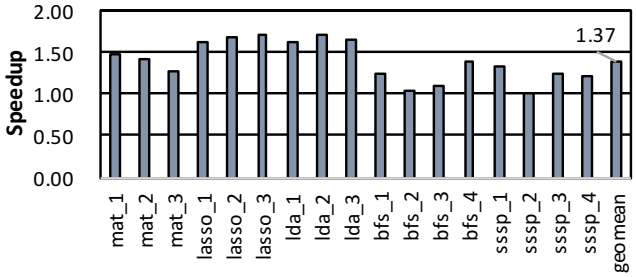


Figure 3: Speedups of the SSP model over the BSP model.

B. Atomic Operation Overhead

Atomic read-modify-write operations capable of reading, modifying, and writing a value to the memory without interference from other threads are frequently used in workloads where threads must sequentialize writes to a shared variable. They provide serializability so that memory access appears to occur at the same time to every thread. Atomic operations affect how threads see updates from other threads for the shared memory address.

Atomic operations are used in parallel ML workloads for reduction operations. Reduction is used to combine the result of computation from each thread. Atomic operations enable different threads to update the same memory address in parallel code. For example, to implement Matrix Factorization, atomic-inc/dec operations are used. Atomic-inc/dec reads a word from a memory location, increments/decrements the word, and writes the result back to the memory location without interference from other threads. No other thread can access this address until the operation is complete. If atomicity is not provided, multi-threaded systems can read/write in shared memory thus inducing the data race. The data race can lead to reduction operation failure, which can slow down progress per iteration and even break the convergence guarantee.

While previous studies show that exploiting asynchronous parallelism could improve performance significantly, the atomic operation overhead also has a huge impact on performance. Figures 4 and 5 show the execution breakdown of the BSP model and the SSP model with a staleness threshold of two. We measure the execution time of different stages from each thread and use the sum of all threads as the execution time of that stage for the workload. (See Section IV for detailed explanations of workloads and the hardware).

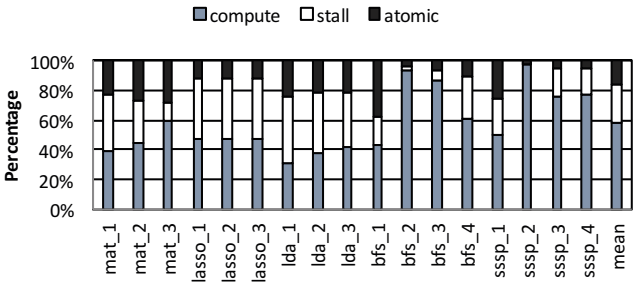


Figure 4: Portion of Each Passes on the BSP Model.

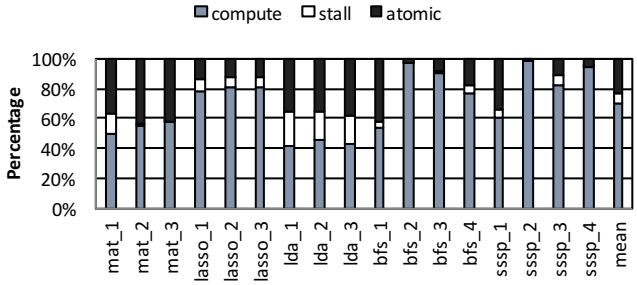


Figure 5: Portion of Each Passes on the SSP Model.

On the BSP models, a major portion of the execution time is not spent on the main computation; only 58% is spent for the main computation, 16% for atomic operations, and 26% for stall time. As explained in Section II-A, stall occurs due to the imbalanced progress of each thread in terms of iterations. The main reason for the thread imbalance is due to the sparse nature of the data and value-dependent

computation in ML workloads; that is, different threads have non-uniform workload distributions. For example, in Breadth First Search (BFS), each vertex has a different number of neighbors. The task is typically partitioned so that each thread processes a disjoint subset of vertices, therefore different threads can execute a different number of instructions.

The stall time is reduced on the SSP model by allowing asynchronous execution; on the SSP model as an average, stall time is reduced to 7% from 26% of the execution time on the BSP model. The atomic operation overhead now becomes the dominant performance bottleneck on the SSP model. On average, the atomic operation overhead consists of 23% of execution time on the SSP model. Still, 30% of the time is wasted, not performing the main computation.

```
void atomic_operation(int* address, int val){
  // normal memory load operation
  int old = *address;

  // compute new value to store
  int new_val = compute(old, val);

  int assumed;
  do {
    assumed = old;
    // decide whether to perform CAS operation
    if(good_to_compare_and_swap(assumed, new_val)){
      // CAS operation
      old = compare_and_swap(address, assumed, new_val);
    }else{
      return;
    }
  }while(assumed != old);
}
```

Figure 6: Pseudo Code of Atomic Operation.

Figure 6 shows the pseudo-code of atomic operations using the compare-and-swap (CAS) operation.[2] The CAS operation is provided as a single instruction in many architectures such as x86. The atomic operation consists of four steps. First, a normal memory load operation is performed. This load operation does not allow reading the stale value unlike the load operation fetching the value for main computation (stage 2 in Figure 2). Second, the new value to store (new_val) is calculated by the program using the value computed from main computation (val) and the loaded value (old). Third, a check operation is performed to decide whether to perform a CAS operation. Fourth, the CAS operation for the same memory location is performed, which compares the memory contents with the original loaded value. Only when the values match, does the swap operation occur, updating memory.

The memory-intensive atomic operation incurs high overhead with non-overlapped multiple transactions on the lower level of the memory hierarchy. The transactions of reading, modifying, and writing a value back to memory are done by the core, and the data movement is not overlapped with the main computation, thus increasing execution time. Fetching data from the lower level to the L1 cache increases the latency of the atomic operation, which can become worse with large data so that the line should be fetched from DRAM rather than shared cache. Other cores trying to modify the same cache line can cause the repetition of the memory load and CAS operation, which increases the L1 access, which will be mostly cache misses.

The increased latency resulting from invalidation also increases the overhead, and the invalidation traffic will also be problematic on the shared cache with many cores. When multiple threads try to modify the same line, a lot of invalidations result since every write will send the invalidation.

Also, all threads that try to access the same location are sequentialized to assure atomicity. Possible collisions can cause poor performance as threads are sequentialized. As the atomics serialize accesses to the same element, the performance of atomic instructions will be inversely proportional to the number of threads that access the same address. As more cores become available on chip, performance degradation due to serialization will increase.

While overlapping atomic operations with computation is possible with launching extra workers, launching a background thread is neither realistic nor beneficial for ML workloads since launching more threads for computation is typically more helpful than launching background threads. When executing highly parallel ML workloads on many cores, despite the reduced stall time with asynchronous parallelism, strict consistency with atomic operations has caused slow execution. Therefore, we conclude that the performance of ML workloads is atomic operation bound.

III. Mechanism

In this section, we propose *BSSync*, hardware support for offloading the atomic operation onto logic layers on memory devices. We describe the bounded staleness consistency model that BSSync utilizes and the key idea of our proposal. We then explain how data communication is performed with our hardware implementation.

A. Bounded Staleness Consistency Model

BSSync supports the bounded staleness consistency model [12] to reduce the atomic operation overhead. The bounded staleness consistency model is a variant of the relaxed consistency models in which data read by a thread may be stale, missing some recent updates. The degree of staleness, the delay between when an update operation completes from a thread, and when the effects of that update are visible to other threads is defined under the user-specified threshold. The bounded staleness consistency model allows reads to use stale values unless they are too stale. The staleness is bounded so that it cannot be larger than the

[2]We omit the Load-Linked/Store-Conditional (LL/SC) implementation that incurs similar cost as CAS implementation for brevity.

user-specified threshold. The bounded staleness consistency model has been widely used for its combined benefits of communication latency tolerance and minimization of the negative impact of stale values on convergence on multiple-nodes configurations [6].

1) Staleness Definition: Staleness can be measured in multiple ways. Here, we use *version numbers* in terms of the number of iterations, as in the SSP model [5]. The version number v of a datum represents that the value of a particular datum has been read by the thread at iteration v. Staleness is defined using the version number and the current iteration of the thread. For example, if the thread is at iteration i, the staleness of the data with version number v is equal to $i - v$.

We change the validity of a datum and redefine the meaning of a cache hit/miss as in ASPIRE [6]. With the user-defined staleness threshold s, as in the SSP model, the read request to a datum can be

- Stale-hit: datum in cache and staleness $\leq$ s
- Stale-miss: datum in cache and staleness $>$ s
- Cache-miss: datum not in cache

2) Operations: Here, we explain how the bounded staleness consistency model defines the read operation for asynchronous parallel ML workloads. Different threads can have different views of a shared datum; the read operation is only guaranteed to obtain the value whose staleness is less than or equal to the staleness threshold s. When a thread reads a shared datum d at iteration i, the thread is guaranteed to see the effect of all updates on d from the first iteration to iteration $i - s$.

B. Key Idea

BSSync reduces the atomic operation overhead by utilizing the characteristic of the iterative convergent ML workloads that allows the use of stale values in the computation. Compared to previous studies utilizing the characteristic for read operations, BSSync utilizes the characteristic to allow stale-writes to minimize the adverse impact of long latency atomic operations. BSSync is based on the following two ideas regarding the iterative convergent algorithms and the-state-of-art hardware implementations.

- First, atomic operations in ML workloads are for other threads to see the updates within a certain staleness bound. The atomic update stage is separate from the main computation, and it can be overlapped with the main computation. Since these workloads do not enforce strict data dependence constraints (threads can miss a certain number of updates to use stale values), the update operation can be performed asynchronously.
- Second, atomic operations are a very limited, pre-specified set of operations that don't require the full flexibility of a general-purpose core. The hard-wired implementation of atomic operations on the memory hierarchy can be more efficient.

A key observation is that offloading atomic operations to asynchronously execute in parallel with the CPU core can eliminate the overhead of atomic operations. Atomic operations are immediately retired from the core side but the logic layer guarantees the atomicity. Atomic operations only need to guarantee the atomicity of the update. Assuming the atomicity is provided, that is, if the update is not dropped, it is all right when the atomic update operation is performed asynchronously so that other threads can observe the recent value later.

Asynchronous execution of atomic operations reduces the overhead of long latency reading the value from the lower level of the memory hierarchy. It reduces the overhead by transforming blocking operations into non-blocking operations. The CPU core does not wait for the atomic operations to finish but proceeds to the main computation, enabling high performance. Asynchronous execution of atomic operations also reduces the overhead of redundant computation for retrying in the case of conflict and the serialization that blocks cores from proceeding for computation.

Instead of general-purpose cores, BSSync utilizes simple logic layers at the memory hierarchy to perform atomic operations. The CMOS-based logic layer can provide an efficient implementation for atomic operations and can help the CPU core to be latency tolerant. While the atomic operation is basically read-modify-write, reading the value from the thread performing the atomic operation is not needed since the value will not be used by the thread. The read can probably incur fetching the value from the lower level of the memory hierarchy. Since the data has low temporal locality, it is inefficient to fetch the data to be located near the core expecting short latency with a cache hit. The second generation of Hybrid Memory Cube (HMC) also supports simple atomic operations [13]. The implementation is quite straight-forward without the requirement of full flexibility of the CPU core so that it can be a hard-wired implementation.

C. Structure

Figure 7 shows the hardware extension of BSSync. Each core is extended with a region table, control registers (iteration register, threshold register), and an atomic request queue (ATRQ). The cache hierarchy consists of per-core private L1 data caches and an inclusive shared L2 cache. The traditional directory-based coherence mechanism is extended to control the degree of coherence. Logic layers are extended on each level of memory hierarchy: L1 data cache, L2 cache, and DRAM.

BSSync changes the conventional hardware/software interfaces to exploit parallelism between the host core and the logic layer at the memory hierarchy. The programmer needs to provide the staleness threshold and the thread progress in terms of iteration counts. The programmer also needs to modify the code to invoke the assembly-level atomic instruction and annotate the shared memory object that

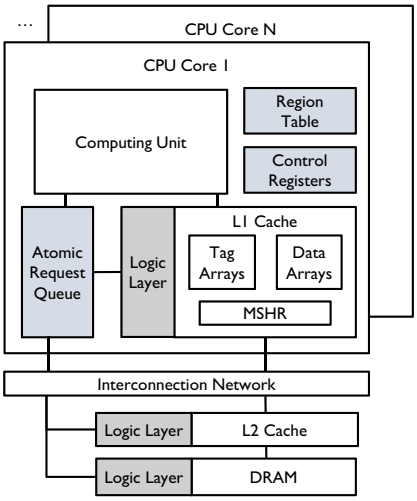


Figure 7: Overview of BSSync.

allows bounded staleness consistency. While this requires certain changes, these changes are mostly straight forward so they can be easily done by the programmer.

The information provided by the programmer is used by the BSSync hardware. The region table for each core contains the address range of annotated memory objects, one entry for each memory object. In many ML workloads, these regions are contiguous memory regions and not many objects exist in most workloads. The iteration register tracks the progress of each thread storing the iteration count that the core is currently executing. Before starting a new iteration, each thread updates the iteration register for the thread with the iteration number that the thread will start. The threshold register stores the staleness threshold, which is provided by the programmer.

ISA is extended to include the fixed-function, bounded-operand atomic instructions executing on the logic layer. We extend the opcode to encode the atomic operation. The format of the atomic instruction follows the format of the current load and store instructions where the size of the operation is encoded.[3]

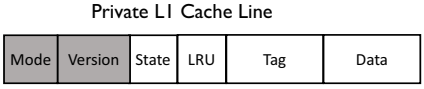


Figure 8: Cache Tag Extension of Private L1 Data Cache Line.

BSSync utilizes different cache protocols, depending on the type of memory object. The decision for which protocol to use is made at the cache line granularity on the private L1 data cache. Figure 8 shows the cache tag extension of the L1 cache line. The tag entry for each L1 cache line is extended to include additional bits for tracking (a) coherence mode, and (b) version number of the cache line. The mode bit is used to define the coherence mode of the cache line. The coherence mode is bi-modal: Normal (N = 0), and Bounded staled (B = 1). The normal line follows the conventional coherence protocol: write-back policy with write-allocate, and the MESI coherence protocol, but the bounded staled line follows a different protocol described in later sections. The version bits in the L1 cache line are used to track the time until this cache line is valid, provide the time when the cache line should be invalidated and get the new update from the lower level of the memory hierarchy. The length for version bits depend on the maximal staleness threshold with the use of modulo operation, which incurs negligible overhead.

The ATRQ decouples the atomic operations from the conventional coherence tracking structures. The ATRQ is for the computing unit to send the atomic operation requests to logic layers at the memory hierarchy. It is placed between the computing units and the logic layer on the L1 data cache. It is also connected to the shared L2 cache and DRAM through an interface to the interconnection network. This interface is similar to the one in many cache bypass mechanisms. The ATRQ shares an interface to the core's MMU and uses the physical address translated from the MMU to send the request to the logic layers of the shared L2 cache and DRAM.

D. Operations

In BSSync, there are two different ways to handle memory instruction: memory accesses for normal memory objects and memory accesses for objects allowed to read stale values. The CPU core identifies the memory access type by using a region table and sends the memory request to different memory units.

The normal memory accesses are handled the same way as conventional directory-based MESI protocols. When a L1 cache miss occurs, a new miss status-holding register (MSHR) entry and cache line are reserved for the line if there is no prior request to the same cache line. The MSHR entry is released when the cache line arrives from the L2 cache. When invalidation occurs, the invalidation requests are sent to all sharers and the L1 cache tag array and MSHR table are read upon receiving the request.

On the contrary, memory accesses for the objects allowed to read stale values do not follow conventional protocol. The accesses are further decomposed into the memory read requests for the objects, and the atomic reduction operation requests for those objects.

1) Handling Read Request: The data that are modified through atomic operations are also read by each thread for computation. Each thread needs to fetch recent changes on

[3]While we consider the bounded operand as the single word of data, the operation might operate on multiple words using pre-defined vectors as SIMD extensions are supported, which is straight-forward.

the shared data into each thread's private L1 data cache. When a core makes a read request for the data on its private L1 cache, it checks whether the data is too stale (the staleness is larger than the staleness threshold). The version number of the cache line is used to decide stale-hit/miss; therefore it is used to define the limit until the line is used. If it is a stale-hit, no further action is required, the core keeps accessing the line in the L1 cache.

On the contrary, in the case of a stale-miss, the read request is blocked and the line in the lower level of the memory hierarchy should be fetched into the private L1 cache even if the line resides in the L1 cache. The cache line is invalidated from the L1 cache and the fetch request is sent to the shared L2 cache. The version bit in the tag array is updated with the current iteration count that is stored in the iteration register of the core.

When the data is not cached in the L1 cache, thus incurring a cache miss, the core brings the data from the lower level of the memory hierarchy into the private L1 cache and updates the version bit with the current iteration count. The behavior is same as if the version number in the L1 cache were -∞.

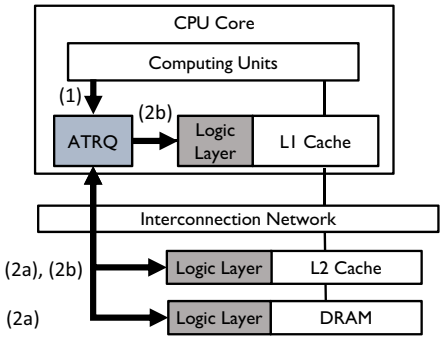


Figure 9: Handling Atomic Operation Request.

2) Handling Atomic Operation Request: Figure 9 shows how BSSync handles the atomic request. First, the core issues an atomic operation request into the ATRQ (1). In BSSync, the atomic reduction operation is a non-blocking operation. The operation completes immediately after the core simply puts the request into the ATRQ. The ATRQ holds the information of the requests that are not completed, so the size of the ATRQ varies dynamically with the number of in-flight atomic operation requests. The logic layers on the cache and the DRAM perform atomic operations and notify the ATRQ to release its entry, when the atomic operation is finished (2).

The processing of the atomic operation request depends on whether or not the datum resides on the L1 data cache. In the case of an L1 cache miss (2a), the ATRQ diverts the atomic operation requests to bypass the L1 D-cache and directly sends requests through the interconnection network into the lower level of the memory hierarchy. If the line resides on the L2 cache, the logic layer on the L2 cache performs the atomic operation, sets the L2 cache line as "dirty" (changing the state bits as "modified") and notifies the ATRQ. In the case of L2 cache miss, the logic layer on the DRAM performs the operation and notifies the ATRQ.

When the line resides in the L1 cache (2b), the ATRQ sends the atomic operation request to both logic layers on the private L1 cache and the shared L2 cache. Since we assume inclusive cache, the line resides on the L2 cache if it resides on the L1 cache. The state bits of the L1 cache line still remain as "clean," unlike conventional coherence protocols, but changes only the state bits on the L2 cache line. The ATRQ entry is released when the logic layer at the L2 cache notifies the ATRQ after finishing the atomic operation.

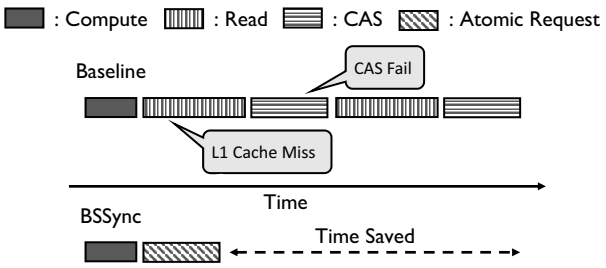


Figure 10: Comparison to Conventional Protocol

Figure 10 compares how BSSync changes the way the atomic operation is handled. While the atomic operation is completed by sending the atomic operation request into the ATRQ with BSSync, conventional implementation requires memory load and the CAS operation. The memory load can miss on the L1 cache, thus fetching the cache line from the lower level of the memory hierarchy. The CAS operation can fail incurring the repetition of the memory load and CAS.

BSSync supports two different types of atomic operations: atomic-inc/dec and atomic-min/max. The value accompanied by the atomic operation request is different depending on the type. For atomic-inc/dec, the ATRQ entry holds the delta for inc/dec, and the logic layer performs the inc/dec operation with the delta. For atomic-min/max, the logic layer compares the value for the datum in the request and the one in the memory hierarchy and stores the minimum/maximum value from the comparison.

It should be noted that our mechanism is different from the studies that bypass the private L1 cache to reduce the contention at the private L1 cache. Bypassing can help since it can reduce the contention of invalidation and serialization due to multiple writers and reduce cache thrashing that evicts the lines that may be reused shortly. However, bypassing loses the opportunity of the shorter access latency when the requested data resides in the L1 cache. When only fetching

data from the L2 cache, the latency of accessing data will increase. On the contrary, BSSync increases the reuse at the private L1 cache reducing cache thrashing and invalidation by allowing multiple values for the same data and not using the L1 cache for the no-reuse data.

Our mechanism is also different from write-through policy where all writes directly go to the shared cache. BSSync eliminates the memory load within atomic operations for the low-reuse data, not the store operation, which is required for whatever policy is used for write.

3) Handling Evictions: BSSync changes how eviction is handled. When eviction occurs, the memory hierarchy uses the dirty bit (state bit as modified) to identify if write-back needs to be performed. Since the state bits of the L1 cache line of annotated memory objects are always "clean," the line is just evicted from the private cache without performing write-back on the L2 cache. In fact, the new value should have already been applied to the L2 cache by performing the atomic operation on the L2 cache.

When the cache lines in the shared L2 cache are evicted, BSSync performs similar tasks as in the conventional protocol. The dirty lines on the L2 cache are written back to the DRAM. The atomic operation performed on the L2 cache changes the state bits of the line so that the new value can be stored onto DRAM when eviction occurs.

IV. Methodology

A. Benchmarks and Inputs

We evaluate five ML applications with different inputs: Least Squares Matrix Factorization (MAT), LASSO regression (LASSO), Latent Dirichlet Allocation (LDA), Breadth First Search (BFS), and Single Source Shortest Path (SSSP). MAT, LASSO, and LDA are from Petuum [5] and utilize the atomic-inc/dec operation. BFS and SSSP are taken from implementations provided by Harish et al. [14], [15] and utilize the atomic-min operation. MAT learns two matrices, L and R, such that $L * R$ is approximately equal to an input matrix X. Given a design matrix X and a response vector Y, LASSO estimates the regression coefficient vector β, where coefficients corresponding to the features relevant to Y become non-zero. LDA automatically finds the topics, and the topics of each document from a set of documents using Gibbs sampling. BFS expands the Breadth First Search tree from a given source node, and SSSP calculates the shortest path for each vertex from the given source vertex. We port original workloads into pthread workloads utilizing all available threads.

Tables I and II show the input for our evaluation. Each data set has varying properties. For each workload, we measure the workload completion time. Each thread iterates on a loop until the solution converges. We sum the execution time of each iteration and use it for performance comparison.

Workloads	Inputs
Matrix Factorization (MAT)	729 X 729 matrix rank : 9, 27, 81
LASSO Regression (LASSO)	50 samples X 1M features lambda : 0.1, 0.01, 0.001
Latent Dirichlet Allocation (LDA)	20-news-groups data set topics : 4, 8, 16

Table I: Inputs for MAT, LASSO, and LDA from Petuum [5].

Graph	Vertices	Edges	Characteristic
coAuthorsDBLP (1)	299067	977676	Co-authorship network generated by DBLP
PGPgiantcompo (2)	10680	24316	PGP trust network
cond-mat-2003 (3)	30460	120029	Co-authorship network of condensed matter publications
ny_sandy (4)	44062	77348	Live Twitter event

Table II: Input Graphs for BFS, and SSSP from Dyno-Graph [16].

Number of x86 cores	64
x86 core	x86 instruction set(user space) Out of order execution, 2.4 GHz
On-chip caches	MESI coherence, 64B line, LRU replacement L1I cache: 16 KB, 8-way assoc, 3 cycles L1D cache: 16 KB, 4-way assoc, 2 cycles L2 cache: 2 MB, 4 bank, 16-way assoc, 27 cycles
DRAM	Single controller, 4 ranks/channel, 8 banks/rank Closed page policy Latency : 100 cycles

Table III: Baseline Hardware Configuration.

B. Hardware Configurations

For evaluating our hardware mechanism, we utilize ZSIM [17]. Table III shows the baseline hardware configuration on the ZSIM simulator. Not just the core but also the memory model are modeled in detail. Each core runs x86 instructions and consists of the execution pipeline, the private L1 instruction and data caches. In BSSync, each core is also extended to include the region table, and the control register. The ATRQ has 64 entries per core and we did not see resource contention for the ATRQ in the evaluation. The coherence mode and version number for each cache line are integrated by extending tag arrays to track the status of the cache line in the caches. The caches use LRU replacement policy. A simple logic layer performing atomic operations is integrated on the caches and DRAM. We assume the single-cycle latency of performing the atomic operation when the request reaches the logic layer.

V. Results

A. Overall Performance

Figure 11 shows the speedup of BSSync with the BSP model and the SSP model with a staleness threshold of two. Not only with the SSP model, but also with the BSP model, BSSync reduces the atomic operation overhead. On average, BSSync outperforms the baseline implementation of SSP model by $1.83 \div 1.37$ = 1.33x times and the one of the BSP model by 1.15x times.

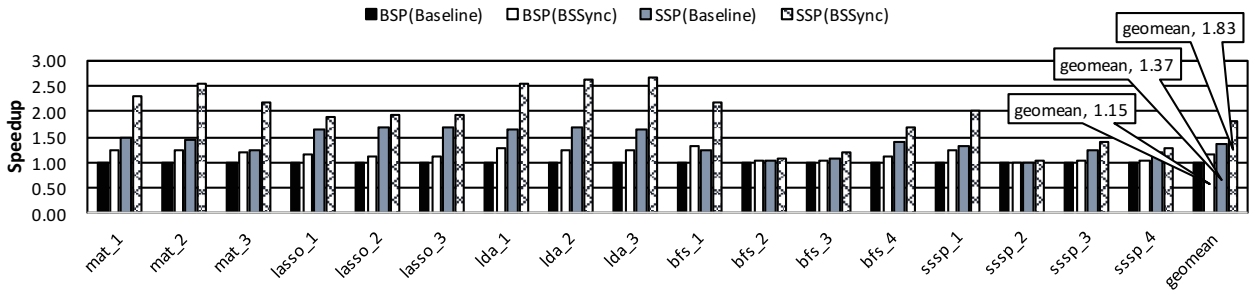


Figure 11: Speedup of BSSync.

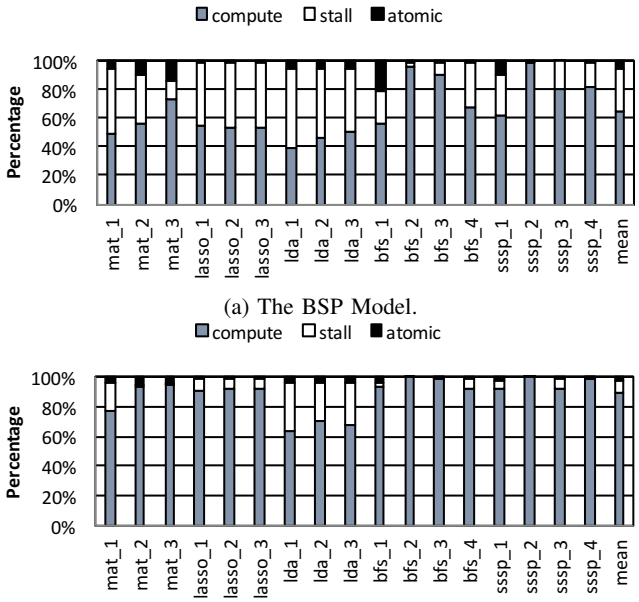
BSSync shows greater benefit on the SSP model due to two reasons. First, it is because the portion of the atomic operation for overall execution time is greater on the SSP model. On the BSP model, stall time still consumes a large portion of the execution time and has less benefit. Second, it is because BSSync can better overlap atomic operation execution with main computation on the SSP model than the BSP model. On the BSP model, the computation result of an iteration should be visible to other threads before starting next iteration, so atomic operations are not fully overlapped with main computation.

The benefit of BSSync varies depending on workloads with regard to words per instructions. In general, if a thread touches more shared data per instruction, the degree of benefit from BSSync increases. MAT and LDA show a higher ratio for atomic operations than LASSO; therefore, they have more benefit with BSSync. LDA shows greater benefit with increasing topic counts. BFS and SSSP show the highest benefit with coAuthorsDBLP (1) input, which is the largest graph in our input sets.

B. Analysis

Figures 12 shows the reduced portion of the atomic pass in total execution time. Figures 12a shows that the atomic operation overhead consists of 5% of the execution time on the BSP model with BSSync. In Figures 12b, the atomic operation overhead is reduced to 2.3% from 23% of the baseline implementation of the SSP model. Now, 89% of the time is spent on the main computation with BSSync for the SSP model.

A major portion of performance improvements of BSSync can be attributed to reduced number of memory loads. Performing atomic operations at the core incurs a large number of memory loads, which inefficiently handles data with low locality. Contention from multiple cores can incur the repetition of atomic operations and therefore more memory loads. Figure 13 shows how BSSync reduces the number of memory loads by offloading atomic operations to the logic layer for the SSP model with a staleness threshold of two. BSSync reduces the number of loads by 33%. Benchmarks



(a) The BSP Model.

(b) The SSP Model.

Figure 12: Portion of Each Passes with BSSync.

such as LDA show less than half the number of memory loads from the baseline with BSSync.

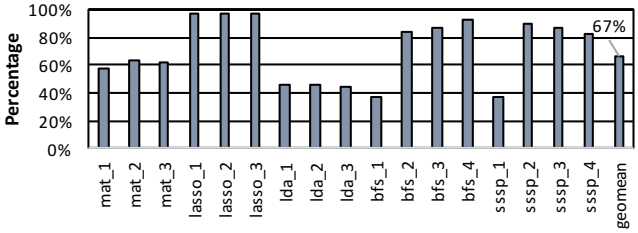


Figure 13: Reduced Memory Loads on the SSP model.

Contention from multiple cores to write the same cache line increases memory access latency. The memory latency is increased due to round trip time to invalidate other

cores and receive their acknowledgments, and for requesting and receiving synchronous write-backs. BSSync reduces memory access latency by reducing on-chip communication, reduces the upgrade misses by reducing exclusive request, and sharing misses by reducing invalidations. Figure 14 shows the reduced invalidation traffic with BSSync for the SSP model with a staleness threshold of two. Invalidation traffic is reduced to 43% of baseline implementation. On the majority of the benchmarks, the invalidation traffic is reduced more than 50% from the baseline, which translates into a lower completion time.

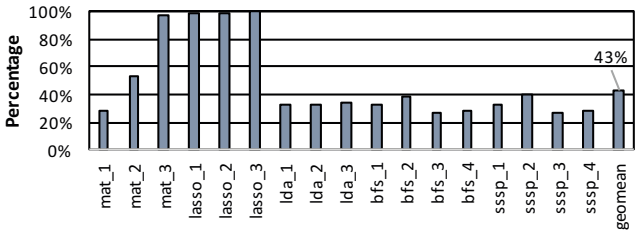
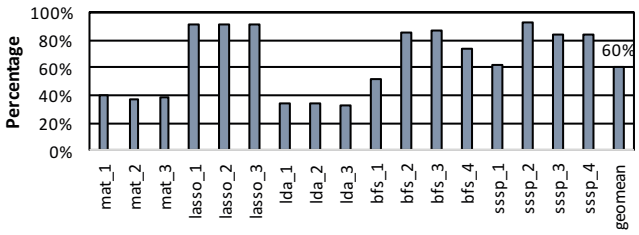


Figure 14: Reduced Invalidation Traffic on the SSP model.

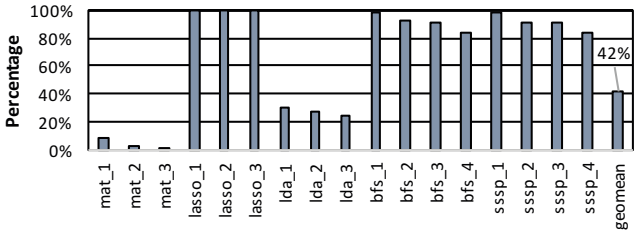
BSSync also increases the L1 private cache utilization efficiency. Performing atomic operations at the core can incur a large number of accesses to the lower level of the memory hierarchy due to memory loads for low locality data in atomic operation. Unnecessary fetches of a word with low locality can lead to L1 cache thrashing and the cores can suffer from expensive communication to the lower level of the memory hierarchy such as the L2 cache and DRAM. Access to the L2 cache and DRAM is costly since there is additional latency due to the time spent in the network, and the queuing delays.

Figure 15 shows a reduced number of accesses to the L2 cache and DRAM. BSSync better exploits L1 caches by efficiently controlling communication. BSSync reduces the capacity misses of the L1 cache by bypassing to the L2 cache or DRAM, thus reducing evictions of the other L1 cache lines. In Figures 15a, 15b, BSSync reduces the number of L2 cache read accesses by 40% and reduces DRAM accesses to 42% of the baseline implementation. Benchmarks like MAT and LDA benefit by reduced L2 cache read accesses and DRAM accesses. BFS and SSSP also benefit from lower L2 cache access frequency with coAuthorsDBLP (1) input.

Here, we model how BSSync reduces memory waiting cycles at the cores. An application's execution time can be partitioned into computing cycles, memory waiting cycles, and synchronization cycles. The speedup of BSSync comes from overlapping atomic operations with the main computation, thus reducing execution time. Figure 16 shows how memory waiting cycles are reduced with BSSync. On average, the memory waiting cycles are reduced by 33%.



(a) L2 Cache Read Accesses.



(b) DRAM accesses.

Figure 15: Reduced Accesses to the Lower Level of the Memory Hierarchy with BSSync.

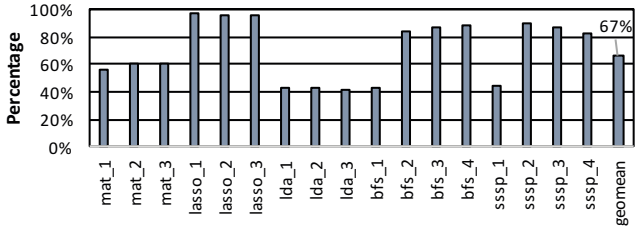


Figure 16: Total Memory Waiting Cycles on the SSP model.

C. Discussion

1) Comparison with Other Studies: Compared to other studies focusing on inter-node communication latency, BSSync tries to improve execution efficiency within a single node. While most studies relax the consistency so that workloads execute in an asynchronous manner, allowing stale-read, our work focuses on allowing stale-write to reduce the atomic operation overhead. BSSync can be easily combined with other studies to further improve overall performance.

2) Thread Migration: So far we have assumed that a thread executes on a pre-defined physical core and that the thread is not switched to other cores so that each physical core tracks the iteration count. If a thread migrates between cores, modification is required such that all hardware should use the thread id instead of the physical core id. However, overall, it is a minor modification and will not affect the benefit of our mechanism.

3) Future Work: As identified in our work, there exists workload imbalance between multiple threads in ML workloads due to the sparse nature of data and value-dependent computation. For simplicity, we assume a workload is statically distributed, so no dynamic work distribution

mechanism exists on this platform. However, a dynamic distribution of workload such as work stealing can resolve the workload imbalance issue and can improve performance. Thus, our future work will address the coordination of BSSync with dynamic workload distribution.

Currently, our evaluation is performed within a single node. Depending on workloads and optimization target, the workload partitioning decision such as number of nodes and number of threads within a node can vary. We will perform a trade-off analysis for each decision on how BSSync changes the trade-off.

VI. Related work

Due to the parallel nature of ML algorithms, considerable effort [18], [19], [20], [21], [22], [23], [24], [25] has been devoted to improving the performance and efficiency of ML workloads. Those efforts are categorized in to two categories: 1) system-oriented and 2) theory-oriented approaches.

System-oriented approaches focus on high iteration throughput. Multiple platforms are examples of these efforts, including the Map-Reduce frameworks such as Hadoop [26], Spark [27], and graph-based platforms such as GraphLab [18]. Ahmed et al. [19] propose a parallel framework for efficient inference in latent variable models. They ignore consistency and synchronization altogether, and rely on a best-effort model for updating shared data. Most solutions based on NoSQL databases rely on this model. While this approach can work well in some cases, it may require careful design to ensure that the algorithm is operating correctly. Low et al. [20] focus on special communication patterns and propose Distributed GraphLab. GraphLab programs structure the computation as a graph, where data can exist on nodes and edges in which all communication occurs along the edges. Low et al. utilize the findings that if two nodes on the graph are sufficiently far apart, they may be updated without synchronization. While this finding significantly reduces synchronization in some cases, it requires the application programmer to specify the communication pattern explicitly. The problem of system-oriented approaches is that they provide little theoretical analysis and proof and that they require the programmer to convert ML algorithm to a new programming model.

On the contrary, theory-oriented approaches focus on algorithm correctness/convergence and number of iterations for a limited selection of ML models. Agarwal and Duchi [21] propose a distributed delayed stochastic optimization utilizing cyclic fixed-delay schemes. Recht et al. [22] propose an update scheme, Hogwild!, a lock-free approach to parallelizing stochastic gradient descent (SGD) utilizing the findings that most gradient updates modify only small parts of the decision variables. Zinkevich et al. [23] propose Naively-parallel SGD and Gemulla et al. [24] propose Partitioned SGD. The problem of theory-oriented approaches is that they oversimplify systems issues. The ideas proposed in those studies are simple to implement but tend to under exploit the full performance potential due to this simplification.

LazyBase [25] applies bounded staleness allowing staleness bounds to be configured on a per-query basis. The consistency model and delaying updates between parallel threads in LazyBase are similar to our work. ASPIRE [6] proposes a relaxed consistency model and consistency protocol coordinating the use of a best effort refresh policy and bounded staleness for communication latency tolerance and the stale value usage minimization. ASPIRE enforces only a single writer for each object which prohibits multiple machines/threads writes to the single object, while BSSync does not have such constraints. ASPIRE targets minimizing inter-node communication overhead, which can be combined with our work.

The recent emergence of 3D-stacking technology enabled high performance by incorporating different technologies: a logic and memory layer that are manufactured with different processes [7], [8], [9]. So, multiple vendors such as Micron, are revisiting the concept of processing data where the data lives, including in-memory atomic operations and ALU functions. Hybrid Memory Cube technology [13] has simple in-memory atomic operations. The Automata processor [28] directly implements non-deterministic finite automata in hardware to implement complex regular expression. Chu et al. [29] propose a high level, C++ template-based programming model for processor-in-memory that abstracts out low-level details from programmers. Nai and Kim [30] evaluates instruction offloading for graph traversals on HMC 2.0. Kim et al. [31] studies energy aspect of processor-in-memory for HPC workloads.

VII. Conclusion

The importance of parallel ML workloads for various application domains has been growing in the big data era. While previous studies focus on communication latency between nodes, the long latency and serialization caused by atomic operations have caused the workloads to have low execution efficiency within a single node.

In this paper, we propose BSSync, an effective hardware mechanism to overcome the overhead of atomic operations consisting of non-overlapped data communication, the serialization, and cache utilization inefficiency. BSSync accompanies simple logic layers at the memory hierarchy offloading atomic operations to asynchronously execute in parallel with the main computation, utilizing staleness for write operations. The performance results show that BSSync outperforms the asynchronous parallel implementation by 1.33x times.

References

[1] R. Zajcew, P. Roy, D. L. Black, C. Peak, P. Guedes, B. Kemp, J. Loverso, M. Leibensperger, M. Barnett, F. Rabii, and

D. Netterwala, "An OSF/1 UNIX for Massively Parallel Multicomputers," in *USENIX Winter*, 1993.

[2] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective Distributed Scheduling of Parallel Workloads," in *SIGMETRICS '96*, 1996.

[3] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti, "The Impact of System Design Parameters on Application Noise Sensitivity," *Cluster Computing*, 2013.

[4] D. Terry, "Replicated Data Consistency Explained Through Baseball," *Commun. ACM*, 2013.

[5] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *NIPS '13*, 2013.

[6] K. Vora, S. C. Koduru, and R. Gupta, "ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM," in *OOPSLA '14*, 2014.

[7] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC '14*, 2014.

[8] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch, "Sonic Millip3De: A Massively Parallel 3D-stacked Accelerator for 3D Ultrasound," in *HPCA '13*, 2013.

[9] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3DIC '13*, 2013.

[10] L. G. Valiant, "A Bridging Model for Parallel Computation," *Commun. ACM*, 1990.

[11] L. Liu and Z. Li, "Improving Parallelism and Locality with Asynchronous Algorithms," in *PPoPP '10*, 2010.

[12] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session Guarantees for Weakly Consistent Replicated Data," in *PDIS '94*, 1994.

[13] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *VLSIT '12*, 2012.

[14] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *HiPC'07*, 2007.

[15] P. Harish, V. Vineet, and P. J. Narayanan, "Large Graph Algorithms for Massively Multithreaded Architectures," International Institute of Information Technology Hyderabad, Tech. Rep., 2009.

[16] "DynoGraph," https://github.com/sirpoovey/DynoGraph, Georgia Institute of Technology, 2014.

[17] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *ISCA-40*, 2013.

[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "GraphLab: A New Framework For Parallel Machine Learning," in *UAI '10*, 2010.

[19] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable Inference in Latent Variable Models," in *WSDM '12*, 2012.

[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," in *VLDB '12*, 2012.

[21] A. Agarwal and J. Duchi, "Distributed Delayed Stochastic Optimization," in *NIPS '11*, 2011.

[22] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *NIPS '11*, 2011.

[23] M. Zinkevich, J. Langford, and A. J. Smola, "Slow Learners are Fast," in *NIPS '09*, 2009.

[24] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent," in *KDD '11*, 2011.

[25] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch, "LazyBase: Trading Freshness for Performance in a Scalable Database," in *EuroSys '12*, 2012.

[26] "Hadoop," http://hadoop.aparch.org.

[27] "Spark," https://spark.apache.org.

[28] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[29] M. L. Chu, N. Jayasena, D. P. Zhang, and M. Ignatowski, "High-level Programming Model Abstractions for Processing in Memory," in *WoNDP '13*, 2013.

[30] L. Nai and H. Kim, "Instruction Offloading with HMC 2.0 Standard - A Case Study for Graph Traversals," in *MEMSYS '15*, 2015.

[31] H. Kim, H. Kim, S. Yalamanchili, and A. F. Rodrigues, "Understanding Energy Aspect of Processing Near Memory for HPC Workloads," in *MEMSYS '15*, 2015.