Exploiting iterative-ness for parallel ML computations

Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky,
Qirong Ho, Gregory R. Ganger, Phillip B. Gibbons*, Garth A. Gibson, Eric P. Xing

Carnegie Mellon University, *Intel Labs

Abstract

Many large-scale machine learning (ML) applications use iterative algorithms to converge on parameter values that make the chosen model fit the input data. Often, this approach results in the same sequence of accesses to parameters repeating each iteration. This paper shows that these repeating patterns can and should be exploited to improve the efficiency of the parallel and distributed ML applications that will be a mainstay in cloud computing environments. Focusing on the increasingly popular "parameter server" approach to sharing model parameters among worker threads, we describe and demonstrate how the repeating patterns can be exploited. Examples include replacing dynamic cache and server structures with static pre-serialized structures, informing prefetch and partitioning decisions, and determining which data should be cached at each thread to avoid both contention and slow accesses to memory banks attached to other sockets. Experiments show that such exploitation reduces per-iteration time by 33–98%, for three real ML workloads, and that these improvements are robust to variation in the patterns over time.

1. Introduction

Data analytics (a.k.a. Big Data) has emerged as a primary cloud computing activity for business, science, and online services that attempt to extract insight from quantities of observation data. Increasingly, such analytics center on statistical machine learning (ML), in which an algorithm induces a statistical model conforming to input data. Such models can expose relationships among data items (e.g., for grouping documents into topics), predict outcomes for new data items based on selected characteristics (e.g., for recommendation systems), correlate effects with causes (e.g., for genomic analyses of diseases), and so on.

This paper focuses on the major subset of ML approaches that employ iterative algorithms to determine model parameters that best fit a given set of input (training) data. Such an algorithm iterates over the input data, refining its current best estimate of the parameter values to converge on a final solution. In each iteration, input data items are evaluated against the current parameters, some of which may be adjusted to better fit those data items.

The common approach to parallelizing such computations is to partition the input data among worker threads across cores and machines, sharing only the parameter values and values associated with determining them. While other designs can be used, an increasingly popular design for maintaining the distributed values is to use a so-called *parameter server* accessed via a simple key-value interface [1, 24, 35]. Not surprisingly, even with the loose synchronization typically used in parallel ML computations, such as synchronizing each iteration with a barrier, a.k.a. a Bulk Synchronous Parallel (BSP) style, the efficiency of maintaining the shared state is a major runtime determinant.

This paper explores an opportunity created by the iterativeness of the iterative algorithms: knowable repeating patterns of access to the share state. Often, each thread processes its portion of the input data in the same order in each iteration, and the same subset of parameters are read and updated any time a particular data item is processed. So, each iteration involves the same pattern of reads and writes to the shared state.

Knowledge of these patterns can be exploited to improve efficiency, both within a multi-core machine and for communication across machines. For example, within a machine, state used primarily by one thread can be placed in the memory NUMA zone closest to the core on which it runs, while significantly write-shared state can be replicated in thread-private memory to reduce lock contention and synchronized only when required. Naturally, cross-machine overheads can be reduced by appropriate partitioning and prefetching of parameter state. But, one can gain further efficiency by constructing static structures for both the servers' and workers' copies of the shared state, rather than the general dynamic

structures that would typically be used, organized according to the prefetch and update batches that will occur each iteration so as to minimize marshaling costs. Indeed, even those batch prefetch and update requests can be constructed statically.

Using three real and oft-studied ML applications (Topic Modeling, Collaborative Filtering, and PageRank), we experiment extensively with a broad collection of iteration-aware specializations in a parameter server (called IterStore) that was originally designed assuming arbitrary access patterns. Altogether, we find that the specializations reduce per-iteration runtimes by 33–98%. Measuring their individual impacts shows that different specializations are most significant for different ML applications, based on their use of shared state, and that there can be synergistic dependencies among them. For example, informed prefetching is crucial for all of the applications, but must be coupled with static pre-marshaled cache structures to achieve high performance on PageRank.

We also evaluate costs associated with attempting to exploit iterative-ness. Naturally, the work of detecting and processing per-iteration patterns is part of an application's overall execution time. We find that most of the specializations compensate for those costs, though cross-machine partitioning surprisingly sometimes does not. We also find that, although capturing the patterns during execution of the first iteration does work, it is more efficient if the application instead does an explicit *virtual iteration* that performs the reads and updates without affecting any state. Finally, we find that the benefits of the specializations are robust to imperfect information about the pattern, as might occur when converged values are no longer modified or responsibility for processing some input data items is shifted to another thread.

This paper makes three primary contributions. First, it identifies iterative-ness as a property of many parallel ML algorithms that can and should be exploited to improve efficiency significantly. Second, it describes and evaluates a broad collection of specializations that exploit iterative-ness effectively, showing their overall and individual values for several real ML applications. Third, it describes and evaluates the concept of a *virtual iteration* and compares it with detection of patterns during the first iteration.

2. Iterative ML and systems

Despite the use of a common name, a wide variety of activities falls under the broad terms "data analytics" and "Big Data", ranging from aggregation queries and summary report generation to advanced statistical machine learning (ML). Even within statistical ML, a variety of models and algorithmic styles is used for data analytics, and various specialized execution frameworks continue to be explored. This section describes the particular context for our work, including iterative ML, computational model, and example applications.

2.1 Iterative fitting of model parameters

This paper focuses on the major subset of ML approaches that we refer to as iterative ML. Generally speaking, statistical ML consists of algorithmically processing a set of input data to identify a mathematical model that fits that data. Iterative ML approaches assume a particular mathematical model will describe the input data and use an algorithm to identify parameter values for that model that make it fit the input data most closely. That is, the computation attempts to optimize the model by maximizing an objective function, which generally describes the error.

In iterative ML, the algorithm starts with some initial parameter value guesses, and then performs a number of iterations to refine them. Each iteration evaluates each input datum, one by one, against current model parameters and adjusts parameters to better fit that datum. Various stopping conditions may be used, such as when the objective function improvement slows sufficiently or just a given amount of time has been spent refining.

2.2 Parallel computation model

Increasingly, iterative ML applications process large data sets to induce detailed models with large numbers of parameters; the term "Big Data" arose in large part from this trend. When the problem size is too big for a single thread to find a solution in an acceptable amount of time, one can use multiple worker threads executing on the cores of one or more machines. Generally speaking, parallel realizations of iterative ML partition input data among the worker threads that each contribute to computing the derived parameter values.

While other models are being explored [21, 24, 30], the common parallel execution model for iterative ML is based on the Bulk Synchronous Parallel (BSP) model. In BSP, each thread executes a given amount of work on a private copy of shared state and barrier synchronizes with the others. Once all threads reach the barrier, updates are exchanged among threads, and a next amount of work is executed. Commonly, in iterative ML, one iteration over the input data is performed between each pair of barriers.

While early parallel ML implementations used direct message passing among threads for update exchanges, a *parameter server* architecture has become a popular approach to making it easier to build and scale ML applications [1, 2, 14, 16, 24, 35, 40, 42]. Figure 1 illustrates this architecture in which all state shared among worker threads is kept in a key-value store, which is commonly sharded across the same machines used to execute the worker threads. Worker threads process their assigned input data using simple READ and UPDATE methods to check and adjust parameter values. To avoid constant remote communication, workers can cache parameter values locally, and both READ and UPDATE these cached values. A CLOCK method is used to identify a point at which a worker's cached updates must be pushed to the shared key-value store and its local cache

state must be refreshed, and it is usually implemented as a barrier. `UPDATE` operations are assumed to be sufficiently commutative and associative that concurrent `UPDATE`s by different workers can be applied to the shared store in any order; thus, there are no update conflicts, by definition. Similarly, having `READ`s return cached values that do not reflect all recent `UPDATE`s by other workers is assumed to not unduly impact the convergence rate. Fortunately, many iterative ML applications, including our example applications, satisfy these assumptions [24, 28]. We consider the case where there is a `CLOCK` after each iteration, and optionally additional `CLOCK`s within an iteration.
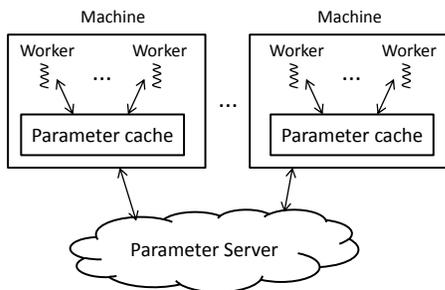

Figure 1. Parallel ML with parameter server.

2.3 Example applications

We use three real ML applications that use different oft-used iterative ML approaches in our experiments, described here to reinforce the general background.

Collaborative Filtering (CF) is a technique commonly used in recommender systems (e.g. recommending movies to users on Netflix). The key idea is to discover latent interactions between the two entities (e.g., users and movies) via matrix factorization. Given a partially filled matrix X (e.g., a rating matrix where entry (i, j) is user i's rating of movie j), matrix factorization factorizes X into matrices L and R such that their product approximates X ($X \approx LR$). We use stochastic gradient descent (SGD) [18, 28], which is a popular method for large scale CF. Each worker thread is assigned a subset of the observed entries in X; in each iteration, each worker processes every element of its assigned subset and updates the corresponding row of L and column of R based on the gradient. L and R are stored in the parameter server so that entries can be accessed by any worker, as needed.

Topic Model (TM) is an unsupervised method for discovering hidden semantic structures ("topics") in a (unstructured) collection of "documents". Popular applications include news categorization, visual pattern discovery in images, ancestral grouping from genetics data, and community detection in social networks. To implement topic modeling, we divide the set of input documents evenly among worker threads and initialize values in the parameter server for the document-topic and word-topic distributions (tables) being derived; there is also a special word-topic "summation row" that provides the

summation of all other rows. Then, we execute the collapsed Gibbs sampling algorithm [23]. In each iteration, each worker thread passes through all the words in its input documents. For a word with *word_id* in a document with *doc_id*, the thread will read the *doc_id*-th row of the document-topic table and the *word_id*-th row, as well as the summation row of the word-topic table. The thread then calculates a new topic assignment and updates these rows.

PageRank (PR) assigns a weighted score (PageRank) to every vertex in a graph [5]. A vertex's score measures its importance in the graph, with higher scores indicating higher importance. The set of scores can be treated as a table where each row corresponds to the PageRank score of one vertex, and we apply the power iteration approach until the PageRanks stop changing. In each iteration, the algorithm passes through all edges in the graph and updates the PageRank of the destination node according to the current PageRank of the source node. The set of edges is partitioned evenly among worker threads, and the scores are kept in the parameter server. PageRank lends itself well to an explicit graph processing model of computation, such as that of GraphLab [21, 30], which makes it a usefully challenging test case for more general architectures like those built around a parameter server.

3. Exploiting iterative-ness for performance

The iterative-ness discussed above creates an opportunity: when per-thread sequences of reads and updates repeat each iteration, they can be known in advance and used to reduce the overheads associated with maintaining the shared state. This section discusses approaches to identifying the sequences and a variety of ways that they can be exploited.

3.1 Obtaining per-iteration access sequence

There are several ways that a parameter server can obtain the access sequences, with different overheads and degrees of help from the application writer. Of course, an ideal solution would have no overhead and need no help, but realistic options are non-optimal on one or both of these axes.

At one end of the spectrum would be completely transparent detection, in which the parameter server gathers the pattern between pairs of `CLOCK` calls. Although this seems straightforward, it is not for two primary reasons. First, many ML applications use the parameter server before beginning to iterate, such as to initialize parameter values appropriately. Because the initialization access pattern likely does not match the per-iteration pattern, and may involve several calls to `CLOCK`, identifying the right pattern would require comparing inter-`CLOCK` patterns until a repeat is found. Second, not every iterative ML application is perfectly repetitive, so such a repeat may never be found, either because there are no exact matches or perhaps even no significant repetitiveness at all. Third, exploitation of repeating patterns can only begin after they are known, so a significant portion of the applica-

tion may be executed inefficiently until then. And, of course, the shared state must be retained in any conversion of the parameter server to a more efficient configuration. These three issues make the fully transparent approach high overhead and not robust; we do not consider it further.

Instead, we explore two options that involve some amount of assistance from the application programmer, illustrated in Figure 2: explicit reporting of the sequence (right-most pseudo-code) and explicit reporting of the iteration boundaries (middle pseudo-code). Both options are described in terms of the access sequence being reported once, at the beginning of the application. But, detecting and specializing can be repeated multiple times in an execution, if the access pattern changes dramatically. Section 5.6 shows that doing so is unnecessary for moderate access pattern changes.

```
// Original            // Gather in first iter    // Gather in virtual iter
init_params()          init_params()              ps.start_gather(virtual)
ps.clock()             ps.clock()                 do_iteration()
do {                   do {                       ps.finish_gather()
  do_iteration()         if (first iteration)     init_params()
  ps.clock()               ps.start_gather(real)  ps.clock()
} while (not stop)       do_iteration()           do {
                         if (first iteration)       do_iteration()
                           ps.finish_gather()       ps.clock()
                         ps.clock()               } while (not stop)
                       } while (not stop)
```

Figure 2. Two ways of collecting access information. The left-most pseudo-code illustrates a simple iterative ML program flow, for the case where there is a CLOCK after each iteration. The middle pseudo-code adds code for informing the parameter server of the start and end of the first iteration, so that it can record the access pattern and then reorganize (during ps.finish_gather) to exploit it. The right-most pseudo-code adds a virtual iteration to do the same, re-using the same do_iteration code as the real processing.

Explicit virtual iteration. The first, and most efficient, option involves having the application execute what we call a *virtual iteration*. In a virtual iteration, each application thread reports their sequence of parameter server operations (READ, UPDATE, and CLOCK) for an iteration. The parameter server logs the operations and returns success, without doing any reads or writes. Naturally, because no real values are involved, the application code cannot have any internal side-effects or modify any state of its own when performing the virtual iteration. So, ideally, the code involved in doing an iteration would be side-effect free (at least optionally) with respect to its local state; our example applications accommodate this need. If the per-iteration code normally updates local state, but still has repeating patterns, then a second side-effect free version of the code would be needed for executing the virtual iteration to expose them. Moreover, because no real values are involved, the application's sequence of parameter server requests must be independent of any parameter values read. Note that if the sequence were to depend on the parameter values, then the sequence would likely vary from iteration to iteration. Thus, the independence property is expected in applications that meet our overall requirement that the request sequence is (roughly) the same from iteration to iteration, and indeed, our example applications satisfy this property.

A virtual iteration can be very fast, since operations are simply logged, and does not require any inefficient shared state maintenance. In particular, the virtual iteration can be done before even the initialization of the shared state. So, not only is every iteration able to benefit from iterative-ness specializations, no transfer of state from an inefficient to an efficient configuration is required. Moreover, the burden of adding a virtual iteration is modest—only ≈10 lines of annotation code for our ML applications.

Explicit identification of iteration boundaries. If a virtual iteration would require too much coding effort, an application writer can instead add start and end breadcrumb calls to identify the start and end of an iteration. Doing so removes the need for pattern recognition and allows the parameter server to transition to more efficient operation after just one iteration. This option does involve some overheads, as the initialization and first iteration are not iterative-ness specialized, and the state must be retained and converted as specializations are applied. But, it involves minimal programmer effort.

3.2 Exploiting access information

In this section we detail parameter server specializations afforded by the knowledge of repeating per-iteration access patterns.

Data placement across machines. When parameter state is sharded among multiple machines, both communication demands and latency can be reduced if parameters are co-located with computation that uses them. As others have observed, the processing of each input data item usually involves only a subset of the parameters, and different workers may access any given parameter with different frequencies. Systems like GraphLab [21, 30] exploit this property aggressively, partitioning both input data and state according to programmer-provided graphs of these relationships. Even without such a graph, knowledge of per-iteration access patterns allows a subset of this benefit. Specifically, given the access sequences, the system can decide in which machine each parameter would best be stored by looking at the access frequency of the workers in each machine.

Data placement inside a machine. Modern multi-core machines, particularly larger machines with multiple sockets, have multiple memory NUMA zones. That is, a memory access from a thread running on given core will be faster or slower depending on the "distance" to the corresponding physical memory. For example, in the machines used in our experiments (see Section 5), we observe that an access to memory attached to a different socket from the core can be as much as 2.4x slower than an access to the memory attached the local socket. Similar to the partitioning of parameters across machines, knowledge of the access sequences can be exploited to co-locate worker threads and data that they access frequently to the same NUMA memory zone.

Static per-thread caches. Caching usually improves performance. Beyond caching state from remote server shards, per-worker-thread caching can improve performance in two ways: reducing contention between worker threads (and thus locking overheads on a shared client cache) and reducing accesses to remote NUMA memory zones. But, when cache capacity is insufficient to store the whole working set, requiring use of a cache replacement policy (e.g., LRU), we have observed that per-thread caches hurt performance rather than help. The problem we observe is that doing eviction (including propagating updates) slows progress significantly, by resulting in much more data propagation between data structures than would otherwise be necessary. Given the access patterns, one can employ a *static cache policy* that determines beforehand the best set of entries to be cached and never evicts them. The size of this per-thread cache can also be optimized—see Section 4.6.

Efficient data structures. The client library is usually multi-threaded, with enough application worker threads to use all cores as well as background threads for communication, and the parameter server is expected to store arbitrary keys as they are inserted, used, and deleted by the application. As a result, a general-purpose implementation must use thread-safe data structures, such as concurrent hash maps [25] for the index. However, given knowledge of the access patterns, one can know the full set of entries that each data structure needs to store, allowing use of more efficient less-general data structures. For example, one can instead use non-thread-safe data structures for the index and construct all the entries in a preprocessing stage, as opposed to inserting the entries dynamically. Moreover, a data structure that does not require support for insertion and deletion can be organized in contiguous memory in a format that can be copied directly to other machines, reducing marshaling overhead by eliminating the need to extract and marshal each value one-by-one. As noted earlier, the first iteration may not provide perfect information about the pattern in all subsequent iterations. To retain the above performance benefits while preserving correctness, one can fall back to using a thread-safe dynamic data structure solely for the part of the pattern that deviates from the first iteration, as discussed in Section 4.3.

Prefetching. Under BSP, each worker thread must use updated values after each `clock`, requiring that each cached value be refreshed before use in the new iteration. Naturally, read miss latencies that require fetching values from remote server shards can have significant performance impact. Prefetching can help mask the high latency, and of course knowing the access pattern maximizes the potential value of prefetching. One can go beyond simply fetching all currently cached values by constructing large batch prefetch requests once and using them each iteration, with multiple prefetch requests used to pipeline the communication and computation work. So, for example, a first prefetch request can get values

used at the beginning of the iteration, while a second prefetch request gets the values used in the remainder of the iteration.

4. Implementation

IterStore is a distributed parameter server that maintains globally shared data for applications. It is an improved version of LazyTable [15, 24], following the computational model described in Section 2.2, and it employs our optimizations (optionally) when informed of the per-iteration access patterns. Although our descriptions and experiments in this paper focus on the traditional BSP execution model, the same optimizations apply to the more flexible Stale Synchronous Parallel (SSP) [13, 24] model supported by LazyTable, and we observe approximately the same relative improvements; that is, the benefits of exploiting iterative-ness apply equally to SSP.
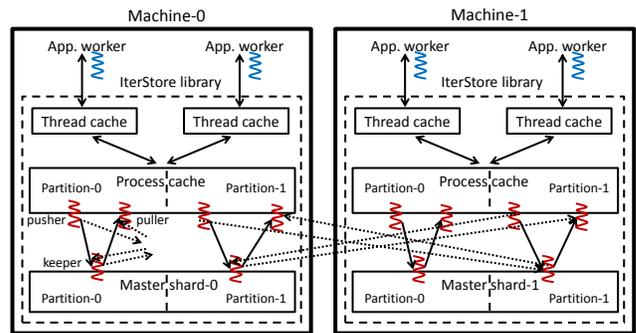
4.1 System architecture

Figure 3. IterStore with two partitions, running on two machines with two application threads each.

The parameter data in IterStore is managed as a collection of *rows* indexed by *keys*, accessed by application threads as described in Section 2.2. A row is a user-defined data type and is required to be serializable and be defined with an associative aggregation operation, allowing updates to be applied in any order. [1]

The IterStore architecture is depicted in Figure 3. The distributed application program usually creates one process on each machine and each process links to one instance of the IterStore library. Each IterStore instance stores one shard of the master version of the data in its *master store*. The data is not replicated, and fault tolerance is handled by checkpointing [15]. The application threads don't directly access the master data. Instead, they access the *process cache* that is shared by all threads in the same instance. One level above the process cache, each application thread has a private *thread cache*. For simplicity, the current implementation assumes that each physical machine has a single IterStore instance, and that each process cache is large enough to cache

---

[1] In order to use the "efficient data structures" specialization, the row must be fixed sized.

all values used by local worker threads. To accommodate cases where there is not that much memory capacity available, the process cache would store only a subset of said values, exploiting the static cache approach used for thread caches; the iteration-aware specializations discussed in this section would still be effective.

IterStore follows the BSP model, wherein readers are only required to see updates from before the most recent clock. As a result, data stored in master stores, process caches, and thread caches can be inconsistent inside a clock. To reduce communication overhead, IterStore batches updates in thread caches and process caches, and propagates them to master stores at the edge of one clock. We attach a data age field to each process cache (and also thread cache) row, so that readers can detect out-of-date data and refresh from master stores.

IterStore performs communication asynchronously with three types of background threads: *keeper* threads manage the data in master stores; *pusher* threads move data from process caches to master stores by sending messages to keeper threads; *puller* threads move data from master stores to process caches by receiving messages from keeper threads.

To increase communication throughput, IterStore globally divides the key space (by the hash of the key) into M partitions; each of the N IterStore machines manages the rows falling in different partitions with different threads in different data structures. For each of the M partitions, each machine launches one keeper thread, one pusher thread, and one puller thread dedicated to the rows of that partition. Each machine's shard of the master data is divided into M pieces in its master store, thus $N \times M$ pieces in total. (The sharding scheme is discussed in Section 4.4.) The keeper thread of `partition-j` of `machine-i` exchanges messages with the pusher thread and puller thread of `partition-j` of all N machines.

4.2 Access information gathering

IterStore supports both ways of gathering access information described in Section 3.1, allowing the application to report the access information either in a virtual iteration or in a real iteration (e.g., the first). When `finish_gather()` is called, IterStore summarizes the gathered information, including the generation of per-thread and machine-wide access rates of all rows, and applies the specializations accordingly.

4.3 Efficient data structures

Because the application can use arbitrary keys to index the stored data, IterStore uses hash maps to store the rows (or their pointers) in master stores, process caches, and thread caches. If not using the virtual iteration approach, IterStore has to start with empty data structures and have the worker threads insert rows dynamically. For master stores and thread caches, dynamic row insertion might not be a big problem, because these data structures are only accessed by one thread.

Process caches, however, are shared. Each process cache partition is accessed by the application worker threads, pusher thread, and puller thread. To support dynamic row insertion, we have to either always grab a global lock on the index before accessing or use a thread-safe data structure, and both approaches are expensive. Moreover, when the background threads exchange updates between process caches and master stores, they need to go through every updated row in their partitions, incurring significant marshaling and update costs.

When IterStore is informed with the set of rows to be stored, it can construct a static data structure with all the rows in it and an immutable index, so that only per-row locks are needed. When the size of each row is fixed (true for all our benchmarks), IterStore will allocate contiguous memory to store the rows for each storage module. Row keys are stored together with the values, making the data structure self-explanatory, so that it can be sent in bulk without per-row marshaling work. The receiving sides still need to unmarshall the message row-by-row, because the layout of the receiver might not be the same as the sender. For random key access, IterStore uses the hash maps to store the mapping from row keys to memory indices, but these hash maps are immutable and can be non-thread-safe.

To deal with the situation where the application needs to access unreported rows (i.e., the current iteration has deviated from the first iteration's pattern), IterStore uses two sets of data structures for each process cache partition, a static part and a dynamic part. The static part is what we described above, and the dynamic part uses a thread-safe hash map to store the additional unreported rows. When a thread fails to find a desired row in the static part, it checks the dynamic part, creating a new row if not found there either.

4.4 Data placement across machines

IterStore determines the partition ID of a row by the hash of its key, but the master version of each row can still be stored in any of the master stores among N machines. Without the access information from the application, IterStore determines the machine ID of each row using another hash, so that all rows will be stored uniformly on all $N \times M$ master stores. With the access information, IterStore assigns each row to machines in a way that minimizes cross-machine traffic, decided in the preprocessing stage, as described next.

Because of the process caches, each machine sends to master stores at most one read and one update request for each row, per clock. Since the *batched access frequency* is either one or zero,[2] IterStore simply places each row on any one of the machines that has access to it. When there are multiple machines accessing the same row, IterStore greedily chooses the machine that has the least number of rows.

---

[2] In case of iterations with multiple clocks, where the work done varies between the clocks of an iteration, the batched access frequency of `machine-i` to `row-j` is the number of clocks in which it accesses `row-j`, divided by the total number of clocks.

The placement decision is accomplished in a distributed manner, by a metadata storage module of the master stores. The master stores of `partition-i` decide and keep the machine placement of rows that are hashed to `partition-i`. For a particular `row-k` in `partition-i`, we choose which master store makes the decision by hashing its key. Suppose the hash is p, in the preprocessing stage, all machines send their batched access frequency of `row-k` to `machine-p`, which chooses the machine to store it based on the frequencies. Suppose the chosen one is `machine-q`, it would inform all machines to send further `READ` and `UPDATE` requests to `machine-q`. Each machine maintains a local mapping of these placement decisions.

4.5 Data placement inside a machine

In our implementation, most of the memory accesses are by application worker threads to thread caches, pusher/puller threads to process caches, and keeper threads to master stores. While many systems assume that access latencies to all memory regions are the same and allocate memory blindly, it is beneficial to allocate memory close to the execution in modern multi-core systems [3]. Because the data structures are allocated statically in the preprocessing stage, IterStore can co-locate the data structures in the same NUMA zones with the thread accessing them most.

Suppose each machine has C CPU cores and Z NUMA zones. We encourage the application writer to create one process per machine and C application threads per process. IterStore will divide the key space into $\frac{C}{2}$ partitions, so that with three background threads per partition, IterStore can (empirically) fully utilize all CPU cores when the application threads are blocked for communication. For each set of $\frac{C}{Z}$ cores in a NUMA zone, we have $\frac{C}{Z}$ application threads and the background threads of $\frac{C}{2Z}$ partitions run on these cores and only allocate memory in the local NUMA zone. Note that each thread cache is allocated by its corresponding application thread, each process cache partition by its pusher thread, and each master store partition by its keeper thread. As a result, all data structures can be accessed locally by the threads accessing them most often.

4.6 Contention and locality-aware thread caches

Thread caches, in addition to a shared process cache, can improve parameter server performance in two ways: reducing contention and increasing memory access locality. To assist with experimentation, the capacity of each thread cache is specified as an input parameter, which might otherwise be determined by dividing the total amount of memory remaining for the application (after initialization, process cache allocation, and master store allocation) by the number of threads per machine.

When access information is not provided, IterStore defaults to an LRU policy for cache eviction. However, we find there is sufficient overhead in doing row eviction and LRU list maintenance that dynamic thread caches often hurt per-

formance rather than help. To avoid this overhead, IterStore uses a static cache policy when it is provided with the access information of the application; IterStore determines the set of rows to be cached in the preprocessing stage and never evicts them. A first small number of rows address contention and the remainder address locality. The remainder of this section describes the selection process for each.

As described in Section 4.1, each process cache partition is accessed by all application threads, one pusher thread, and one puller thread. IterStore explicitly uses thread caches to reduce contention on the process cache. In the preprocessing stage, IterStore estimates the contention probability of each row accessed by each application thread. If the estimated probability is higher than a predetermined threshold, IterStore caches that row in the thread cache.

We use the following model to estimate the contention probability of a row. We define the *access frequency* $\text{AF}_i^{(j)}$ as the number of times (0 or 1) that `thread-i` accesses `row-j`, divided by the number of times that `thread-i` accesses any row, in one iteration. If we assume the time it takes to access each row is the same, access frequency equals *access probability* $\text{AP}_i^{(j)}$, which is the probability that at a given point of time, `thread-i` is accessing `row-j`. Consider a `row-j` that is accessed by `thread-i`, and let $n_j \geq 2$ be the number of threads that access `row-j`. Let $\text{CP}_i^{(j)}$ be the probability that the access of `thread-i` to `row-j` overlaps with one or more accesses to `row-j` by other threads. Our goal is to use thread caches to reduce all $\text{CP}_i^{(j)}$ to below a target bound CPB. To do this, we calculate a threshold $\text{AFT}_i^{(j)}$ for each access frequency such that if some access frequency $\text{AF}_i^{(j)}$ is larger than $\text{AFT}_i^{(j)}$ we will have `thread-i` cache `row-j` in its thread cache. Under the model, it suffices to set $\text{AFT}_i^{(j)} = \frac{\text{CPB}}{n_j - 1}$ for all i and j, as shown in Equation 1.

$$
\begin{aligned}
\text{CP}_i^{(j)} &= 1 - \prod_{i' \neq i}(1 - \text{AP}_{i'}^{(j)}) \\
&= 1 - \prod_{i' \neq i}(1 - \text{AF}_{i'}^{(j)}) \\
&\leq 1 - (1 - \max_{i' \neq i} \text{AF}_{i'}^{(j)})^{n_j - 1} \\
&\leq 1 - (1 - \max_{i' \neq i} \text{AFT}_{i'}^{(j)})^{n_j - 1} \quad (1) \\
&= 1 - (1 - \frac{\text{CPB}}{n_j - 1})^{n_j - 1} \\
&\approx (n_j - 1) \times \frac{\text{CPB}}{n_j - 1} \\
&= \text{CPB}
\end{aligned}
$$

The number of rows cached as a result of these thresholds is bounded by $\max_j(1/\text{AFT}_i^{(j)})$, which is no greater than $\frac{n-1}{\text{CPB}}$, where n is the total number of threads in this machine. The remainder of available IterStore thread cache capacity is used to reduce the amount of remote memory access. As described in Section 4.5, memory access from application

threads to process caches is not necessarily local. The static cache selection policy picks additional rows in decreasing order of access frequency, skipping those stored in the local memory NUMA zone at the process cache level.

4.7 Prefetching

In the preprocessing stage, each IterStore process cache partition summarizes the union set of rows that application worker threads on that machine read and uses it to construct prefetch requests. At the beginning of each clock, a prefetch module sends these pre-constructed prefetch messages to master stores, requesting a refresh of the rows.

To better pipeline the prefetch with computation work, we can do *pipelined prefetch* when we know the ordering that each row is read. Each IterStore process cache partition creates two prefetch requests for each clock, controlled by an *early-ratio* parameter. The *early prefetch* requests contain the rows that satisfy the first early-ratio READ operations, and the *normal prefetch* requests contain the other rows used. The early prefetch requests are prioritized over the normal ones, so that we can reduce the waiting time at the beginning of each clock. We generally observe that an early-ratio of 10% works well, and Section 5.5 describes some experimental results.

5. Evaluation

This section describes our experiment results, showing that exploiting iterative-ness significantly improves efficiency, quantifying individual benefits of the specializations, and showing that they are robust to some deviations from expected per-iteration access patterns.

5.1 Experimental setup

Hardware Information. All experiments use an 8-node cluster of 64-core machines. Each node has four 2-die 2.1 GHz 16-core AMD® Opteron 6272 packages, with a total of 128GB of RAM and eight memory NUMA zones. The nodes run Ubuntu 12.04 and are connected via an Infiniband network interface (40Gbps spec; ≈13Gbps observed).

Application benchmarks. We use the following three example applications: PageRank (PR), Collaborative Filtering (CF), and Topic Modeling (TM). For CF, we use the *Netflix* dataset, which is a 480k-by-18k sparse matrix with 100m known elements. The application is configured to factor it into the product of two matrices with rank 1000. For TM, we use the *Nytimes* dataset [39], containing 100m tokens in 300k documents with a vocabulary size of 100k, and we configure the application to generate 1000 topics. For PageRank, we use the *twitter-graph* dataset [26], which is a web graph with 40m nodes and 1.5b edges. For each application, the virtual iteration annotation requires only ≈10 lines of code.

IterStore setup. We run one application process on each machine. Each machine creates 64 computation threads and is linked to one instance of IterStore library with 32 partitions.

We assume each machine has enough memory to not need replacement in its process cache.

Experiment methodology. Note that our specializations don't change the convergence rate as a function of iterations completed. Only the per-iteration execution time is affected. Therefore, we report on the time required to complete a given number of iterations in the bulk of our analysis. We run each experiment at least thrice (except for Figure 5), report arithmetic means, and use error bars (often too small to see) to show standard deviations.

5.2 Overall performance

Figure 4 shows performance for each of the three ML applications running on four system setups: IterStore without any iterative-ness specializations ("IS-no-opt"), IterStore with all of the specializations and obtaining the access pattern from the first real iteration ("IS-no-viter"), IterStore with all specializations and use of a virtual iteration ("IterStore"), and GraphLab [21, 30][3] using its synchronous engine. Comparing different ones of the four setups allow evaluation of different key aspects, including how much benefit IterStore realizes from iterative-ness specializations, how much more efficient using a virtual iteration is than obtaining the patterns in the first iteration, and, to put the numbers in a broader context, how IterStore compares to a popular efficient system with and without the specializations.

Each bar shows the time required for the application to initialize its data structures and execute 5 iterations, broken into four parts: preprocessing, initialization, first iteration, and next four iterations. Preprocessing time includes gathering the access information and setting up data structures according to them, which is zero for IS-no-opt; GraphLab's preprocessing time is its graph finalization step, during which it uses the application-supplied graph of dependencies between data-items to partition those data-items across machines and construct data structures. Initialization includes setting initial parameter values as well as some other application-level initialization work. The first iteration is shown separately, because it is slower for IS-no-viter and because that setup performs preprocessing after the first iteration; all five iterations run at approximately the same speed for the other three systems.

The results show that the specializations decrease per-iteration times substantially (by 33–98%). Even when accounting for preprocessing times, exploiting iterative-ness reduces the time to complete five iterations by 29–82%. As more iterations are run, the improvement can approach the per-iteration improvement, although early convergence of some parameters can change the access pattern over time and thereby reduce the effectiveness of exploiting iterative-ness. Figure 5 shows performance for the same workloads completing 100 iterations, instead of just five. For these applications,

---

[3] The GraphLab code was downloaded from `https://github.com/graphlab-code/graphlab/`, with the last commit on Jan 27, 2014.
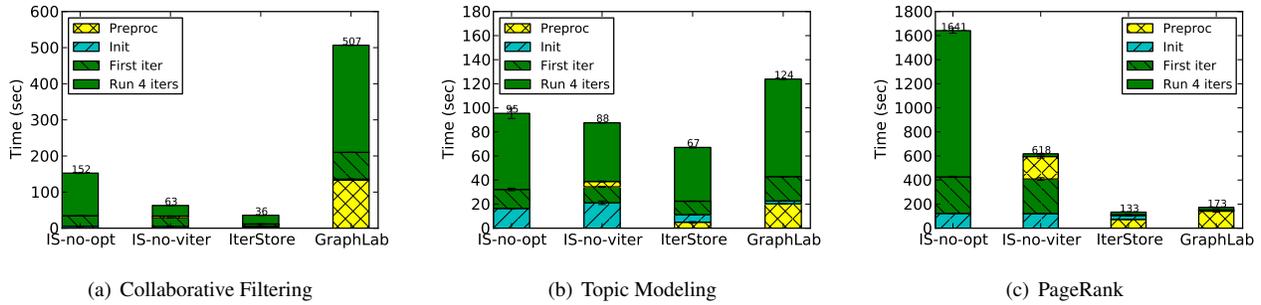
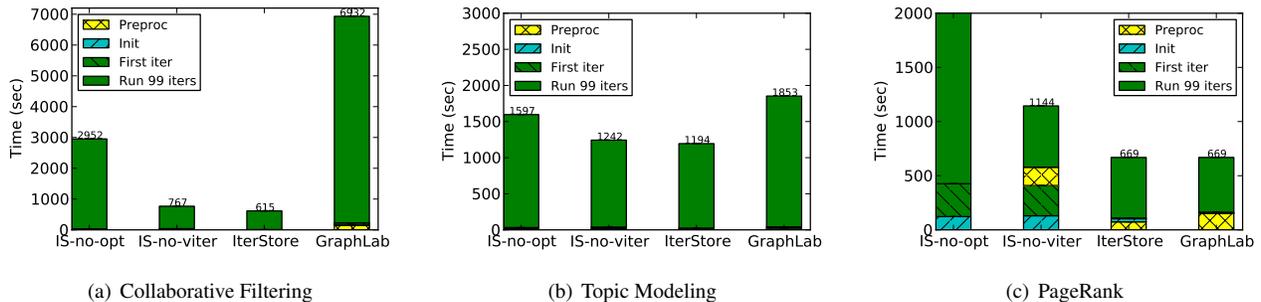**Figure 4.** Performance comparison, running 5 iterations.

Figure 5. Performance comparison, running 100 iterations. The "IS-no-opt" bar in the PageRank figure is cut off at 2000 sec, because it's well over an order of magnitude worse than the other three.

greater benefit is achieved with more iterations, because they don't change their patterns significantly and the preprocessing work is better amortized. Section 5.6 explores further the sensitivity of exploiting iterative-ness to such changes. Note that the entire range of few to 100+ iterations can be of interest in practice, depending on the available time budget and the convergence rate.

The results (more easily seen in Figure 4) also show that using a virtual-iteration is more efficient than collecting patterns in the first real iteration, because the latter causes the initialization and first iteration to be inefficient. Moreover, doing preprocessing after the first iteration requires copying the parameter server state from the original dynamic data structures to the new static ones, making the preprocessing time longer.

With optimizations and virtual-iteration turned on, Iter-Store out-performs GraphLab for all of the three benchmarks, even PageRank which fits GraphLab's graph-oriented execution style very well. For CF and TM, IterStore out-performs GraphLab even without the optimizations, and by more with them. IterStore's performance advantages have two sources. First, the GraphLab abstraction couples parameter data with computation making it less suitable for CF and TM. Second, though the GraphLab implementation implicitly uses some of our proposed specializations, it does not use NUMA-aware data placement or contention-aware thread caches. Note that we under-estimate GraphLab's initialization and preprocess-

ing time in these results, because GraphLab does some of its preprocessing work together with its data loading. We are not showing the data loading time of these systems, because it is huge for GraphLab, due to the fact that GraphLab uses unpartitioned text file as input, while IterStore uses partitioned binary input.

Table 1 summarizes some features of the three benchmarks. We show the total number of rows, size of each row, and the average degree of each node when we express them with a sparse graph.[4] A graph is more sparse when it has a smaller average node degree. The PR benchmark gets huge benefit from our optimizations because its corresponding graph is very sparse, with a huge number of tiny rows, and both features cause our previous IterStore implementation to be inefficient.

| App.	# of rows	Row size (bytes)	Ave. node degree
CF	500k	8k	200
TM	400k	8k	250
PR	40m	8	75

Table 1. Features of benchmarks.

---

[4] Though CF and TM are not graphical applications, some frameworks, such as GraphLab, will still express the computation using graphical structures, and here we use the same graphical structures as those used in GraphLab toolkits to calculate the average node degree.
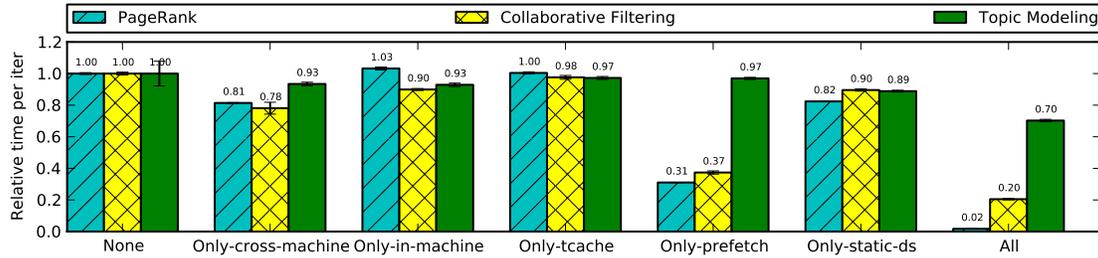
(a) Turn on one of the optimizations.
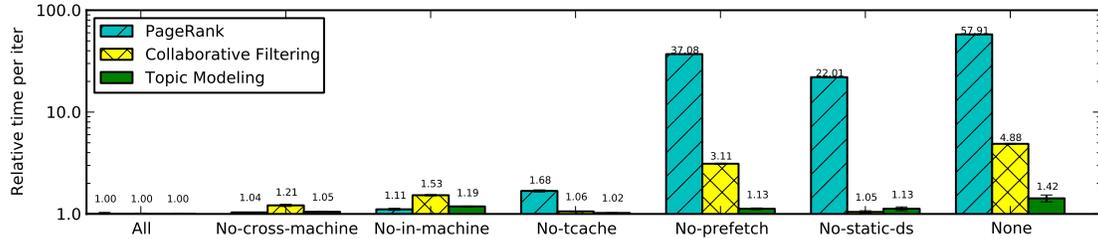
(b) Turn off one of the optimizations.
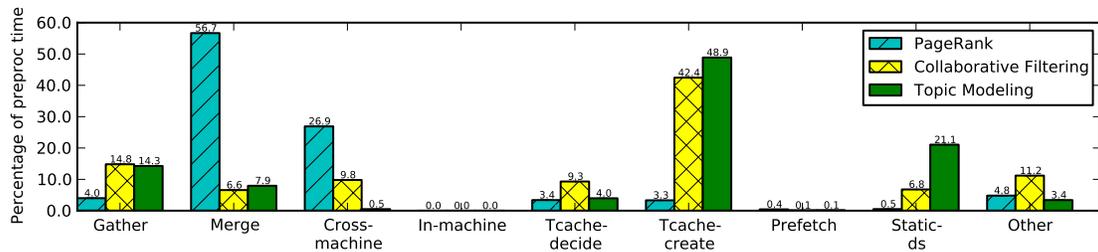
(c) Preprocessing time break down.

Figure 6. Optimization effectiveness break down.

5.3 Optimization effectiveness break down

The optimizations proposed in this paper can work independently. In this experiment, we break down the contribution coming from each of them, as shown in Figure 6. We employ two approaches to investigate that. First, we turn on each of the individual optimizations alone (Figure 6(a)) and measure speed up compared with having none of the optimizations. Second, we turn off each of the optimizations (Figure 6(b)), with the others on, and measure how much the execution is slowed down compared with having all of the optimizations. The results show that most of the optimizations are effective when deployed independently of others, and we also have several interesting findings. First, we find different optimizations are most significant for different applications, based on their use of shared states. PR benefits most from prefetching and static data structures, while CF benefits most from prefetching and in-machine memory management. Second, we find there can be synergistic dependencies among these optimizations. Comparing the "cross-machine" speed

up in both figures, we find its benefit becomes less when we have other optimizations. This is because there is less cross-machine communication overhead in the presence of other optimizations. We also find "prefetch" and "static-ds" offer much more speed up when applied collaboratively than individually. This is because, when both of them are present, IterStore local stores and master stores can exchange data in a batched form, with lower marshaling and unmarshaling overhead.

Figure 6(c) shows the costs of each of these optimizations in the preprocessing stage, together with some costs shared by all of them. "Gather" stands for collecting information via virtual iteration, "Merge" stands for merging information from all threads in each machine, "Cross-machine" to "Static-ds" refer to the respective optimizations, and "Other" stands for the rest of time spent, mostly on synchronization inside the procedure. We break the time for thread cache preparation into a decision part and a creation part. The creation part is the cost of creating the cache data structures, which is inevitable

in order to use the cache, and all "Static-ds" costs are for creating data structures for local stores. We find in CF and TM, these two bars account for most of the preprocessing time.

5.4 Contention and locality-aware caching

This set of experiments compares the performance of IterStore's static thread-caching policy with the LRU policy. We increase the capacity of the thread cache from zero to the size that is large enough to store all values accessed by a thread, and compare time per iteration, as shown in Figure 7. We are not showing the data point for cache capacity being zero in Figure 7(b), because it is too high. TM has one summation row that needs to be updated whenever the topic assignment of any word is changed; when the cache capacity is zero, each iteration takes 493 seconds. The results show that IterStore's cache policy based on the known access pattern outperforms LRU. For all three benchmarks, time per iteration curves for LRU first go up and then go down, which can be explained as follows. When the cache capacity is small, there are few cache hits because most rows are already evicted before being accessed a second time, and the time per iteration increases with increasing cache sizes due to the increasing overheads for cache insertions and evictions. On the other hand, when the cache capacity is sufficiently large, the benefit from increased hit rates outweighs these overheads, and the time per iteration decreases. For our static cache policy, even when we allocate only a tiny amount of memory for the cache, it almost always gives positive benefits from reduced contention and remote memory accesses. Moreover, it performs slightly, or in the case of PR significantly, better than LRU even when the thread cache capacity is large enough to store all values, for two reasons. First, even when there is enough capacity, IterStore's cache policy does not cache rows that have low contention probability and are stored in the local memory NUMA zone, because we find caching these rows results in unnecessary work for negligible benefit. Second, LRU cache needs to update its LRU list whenever an row is accessed, so it has additional bookkeeping overhead.
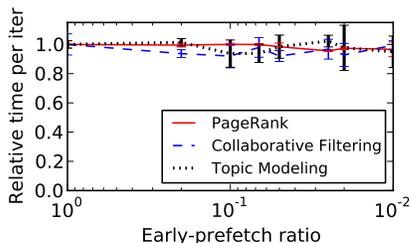
5.5 Pipelined prefetching

Figure 8. Pipelined prefetching.

Pipelined prefetch helps reduce the waiting time at the beginning of each clock by fetching the rows that are used

by the first *early-ratio* `READ` operations. We evaluate the effectiveness of pipelined prefetch by comparing the time per iteration with different early-prefetch ratios (Figure 8). To emphasize the effect, we use a one Gigabit Ethernet network, which has less bandwidth than our default setting. The results show that pipelined prefetch reduces time per iteration in general, but there is high variance across different runs. That is because each client sends one early prefetch request and one normal prefetch request for every master store partition, and the replies containing rows used for the first few reads can be delayed by non-early-prefetch requests from other machines.

5.6 Inaccurate information

This section investigates sensitivity of the specializations to application access patterns that differ from those initially reported, such as can occur when an application's access pattern changes over time due to early convergence of some parameters or due to work migration for load balancing. This section evaluates robustness of iterative-ness specializations when such changes occur, but in the absence of the application reporting a new pattern.

In a first set of experiments (Figure 9(a)), we have each application worker thread report an incomplete set of their accesses, which emulates the situation where these workers get the work of others during the run due to work reassignment. We use the case when we know all access information (missing info is 0) as the baseline, and compare the time per iteration with different amounts of missing information. Results show that IterStore can work well with small amounts of missing information. PR is slowed down by 16% when we have 5% information missing. For CF and TM, we observe almost no slow down even with only 50% of accesses reported. That is because most of the benefits that we achieve for these two applications come from informed prefetching, and because of the large number of application threads (64) in each machine that access overlapping subsets of values; as a result, reporting 50% of accesses from each thread is enough for IterStore to know most of the rows accessed by the machine.

In a second set of experiments (Figure 9(b)), we investigate the situation where the actual accesses are fewer than the reported ones, which can be caused by either work reassignment or by convergence of some parameters. We call the reported accesses that don't actually occur as *false information*. In this experiment, we use the accesses of another thread in a different machine as the false information. We change the fraction of false information and compare time per iteration with having no false information. Data point "false information is 50%" means that half of the accesses reported from each thread will not actually occur. The results show that the extra false information has minimal influence on the performance. The additional overhead comes from extra communication and occupation of thread caches.
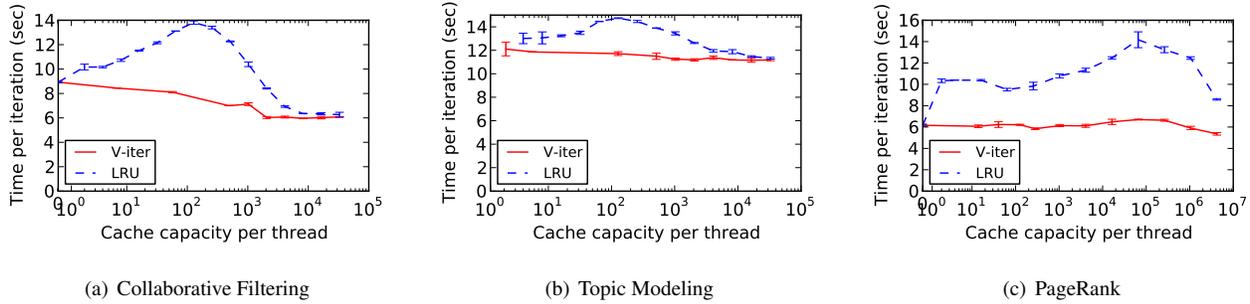
(a) Collaborative Filtering (b) Topic Modeling (c) PageRank

Figure 7. Comparing IterStore's static cache to LRU, varying the cache size (log scale).
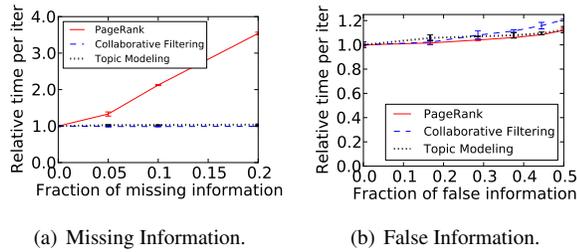

(a) Missing Information. (b) False Information.

Figure 9. Influence of inaccurate information.

5.7 Comparison w/ single thread baselines

App.	Single-threaded	IterStore (512 threads)	Speedup
CF	374.7 sec	6.02 sec	62x
TM	1105 sec	11.2 sec	99x
PR	41 sec	5.13 sec	8x

Table 2. Time per iteration comparison of single-threaded non-distributed implementations using one machine versus IterStore using 8 machines, 512 cores and 512 threads.

We compare our parameter server implementation of CF, TM, and PR against single thread baselines. We use open source GibbsLDA++[5] for TM, which uses the same standard Gibbs sampling algorithm as our implementation on IterStore. For CF and PR, we could not find fast open source implementations using the exact same model and algorithm and thus implemented our own. These single-threaded implementations take full advantage of having all data in memory and require no locking, no network communication, and no inter-processor communication.

Table 2 compares the time per iteration on our benchmarks. As expected, IterStore does not show speedups linear in the number of cores because of the overhead of parallelization and communication. Perhaps surprisingly, however, it does show speeds at least linear in the number of machines. This argues for the use of distributed ML implementations with IterStore techniques, when quick completion times are

important, even if the problem size is small enough to fit in the memory of a single node and good single-threaded code is available. The speed up for PageRank is smaller, because our single-threaded implementation assumes the webpage IDs are dense and stores their ranks in a vector instead of a hash map.

6. Additional Related Work

Frameworks for distributed ML have become an active area of systems research. Some rely on repeated executions of MapReduce for iterations [7, 12]. Some mix MapReduce iterations with multi-iteration in-memory state [41, 43]. Pregel applies the core BSP technique [19] to a long running iterative ML framework [31]. Percolator applies iterative refinement to a distributed key-value store with transactional updates and value triggers [34]. GraphLab [30] and PowerGraph [21] combine these abstractions with a graph-centric programming model, flexible dependency enforcement and optional asynchronous execution. And, several frameworks based on parameter servers have been developed [1, 24, 35]. Many of these systems could use the ideas explored in this paper, exploiting repeating per-iteration access patterns to improve their efficiency.

Other machine learning frameworks emphasize features outside the scope of this paper, such as out-of-core disk scheduling [27, 36]. Naiad generalizes iterative machine learning into a Dryad-like dataflow model for iterative and streaming computation of many forms [32].

Informed or application-aware caching and prefetching have been explored in file systems and databases [8, 9, 22, 29, 33]. In addition to exploring use of future access knowledge, researchers have explored a range of approaches to obtaining it, including explicit application calls [9, 33], compiler analysis [6], speculative execution [11, 17], and dynamic pattern detection [22, 29]. Some of our detection and exploitation of per-iteration patterns build on these ideas, adapting them to the specific characteristics of parameter servers for supporting parallel and distributed ML.

Data placement in NUMA systems was well-studied two decades ago for multiprocessor systems [4, 10]. The re-emergence of NUMA issues in multi-socket systems is well-

---

[5] http://gibbslda.sourceforge.net/

known [3], bringing back the value of carefully placing data and threads to increase locality. Similar access latency asymmetry has also been noted in single-socket many-core chips [37, 38]. Exploiting iterative-ness, as explored in this paper, allows one to orchestrate such locality without the dynamic identification and re-allocation overheads usually found in general-purpose solutions.

7. Conclusion

Many iterative ML applications make the same pattern of read and update accesses each iteration. Knowledge of this pattern can be exploited in parallel ML computations to reduce the overheads of maintaining the state shared among worker threads. With minimal application assistance, a parameter server can obtain each thread's pattern and specialize its data structures, data placement, caching, and prefetching policies. Experiments with ML applications show that such exploitation of iterative-ness can reduce per-iteration execution times by 33–98%.

Acknowledgments

References

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[2] S. Ahn, B. Shahbaba, and M. Welling. Distributed stochastic gradient MCMC. In *ICML*, 2014.

[3] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT*, 2010.

[4] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *SOSP*, 1989.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 1998.

[6] A. Brown, T. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. In *ACM Transactions on Computer Systems (TOCS)*, 2001.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proc. VLDB Endow*, 2010.

[8] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *ACM/IEEE Supercomputing*, 2008.

[9] P. Cao, E. Felton, and K. Li. Implementation and performance of application-controlled file caching. In *OSDI*, 1994.

[10] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS*, 1994.

[11] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *OSDI*, 1999.

[12] C.-T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

[13] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *HotOS*, 2013.

[14] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with COTS HPC systems. In *ICML*, 2013.

[15] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, 2014.

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[17] K. Fraser and F. Chang. Operating system I/O speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference*, 2003.

[18] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 2011.

[19] A. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. In *Scandinavian Workshop on Algorithm Theory*, 1992.

[20] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 2013.

[21] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[22] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Summer USENIX*, 1994.

[23] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 2004.

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[25] Intel. Intel® Threading Building Blocks. `https://www.threadingbuildingblocks.org`.

[26] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[27] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[28] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[29] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *USENIX Annual Technical Conference*, 1997.

[30] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[32] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[33] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, 1995.

[34] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[35] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[36] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, 2013.

[37] TILEPro. TILEPro processor family: TILEPro64 overview. `http://www.tilera.com/products/processors/TILEPro_Family`, 2013.

[38] A. Tumanov, J. Wise, O. Mutlu, and G. R. Ganger. Asymmetry-aware execution placement on manycore chips. In *Workshop on Systems for Future Multicore Architectures (SFMA)*, 2013.

[39] UCI. UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml/datasets/Bag+of+Words`.

[40] Y. Wang, X. Zhao, Z. Sun, H. Yan, L. Wang, Z. Jin, L. Wang, Y. Gao, J. Zeng, Q. Yang, et al. Towards topic modeling for big data. *arXiv preprint arXiv:1405.4402*, 2014.

[41] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *SOSP*, 2013.

[42] R. Zhang and J. Kwok. Asynchronous distributed ADMM algorithm for global variable consensus optimization. In *ICML*, 2014.

[43] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *SoCC*, 2011.