Consistent Bounded-Asynchronous Parameter Servers for Distributed ML

Jinliang Wei, Wei Dai, Abhimanu Kumar,
Xun Zheng, Qirong Ho and Eric P. Xing
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jinlianw@cs,wdai@cs,abhimank@andrew,
xunzheng@cs,qho@cs,epxing@cs.cmu.edu

January 3, 2014

Abstract

In distributed ML applications, shared parameters are usually replicated among computing nodes to minimize network overhead. Therefore, proper consistency model must be carefully chosen to ensure algorithm's correctness and provide high throughput. Existing consistency models used in general-purpose databases and modern distributed ML systems are either too loose to guarantee correctness of the ML algorithms or too strict and thus fail to fully exploit the computing power of the underlying distributed system.

Many ML algorithms fall into the category of *iterative convergent algorithms* which start from a randomly chosen initial point and converge to optima by repeating iteratively a set of procedures. We've found that many such algorithms are to a bounded amount of inconsistency and still converge correctly. This property allows distributed ML to relax strict consistency models to improve system performance while theoretically guarantees algorithmic correctness. In this paper, we present several relaxed consistency models for asynchronous parallel computation and theoretically prove their algorithmic correctness. The proposed consistency models are implemented in a distributed parameter server and evaluated in the context of a popular ML application: topic modeling.

1 Introduction

In response to the rapidly increasing interests in big data analytics, various system frameworks have been proposed to scale out ML algorithms to distributed systems, such as Pregel [8], Piccolo [9], GraphLab [7] and YahooLDA [1]. Amongst them, parameter server [10, 5, 3, 9, 10] is a widely used system architecture and programming abstraction that may support a broad range of ML algorithms. Parameter server can be conceptualized as a (usually distributed) key-value store that stores model parameters and supports concurrent read and write accesses from distributed clients [5]. In order to minimize the network overhead of remote accesses, shared parameters are (partially) replicated on client nodes and accesses are serviced from local replicas. Thus proper level of consistency guarantees must be ensured. A desirable consistency model for parameter server must meet two requirements: 1) Correctness of the distributed algorithm can be theoretically proven; 2) Computing power of the system is fully utlized. The consistency model effectively decouples the system implementations from ML algorithms and can be used to reason about the quality of the solution (such as the rate of variance reduction).

Maintaining consistency among replicas is a classic problem in database research. Various consistency models have been used to provide different levels of guarantees for different applications [11]. However, directly applying them usually fails to meet the requirements of a parameter server. It is difficult or impossible to prove algorithm correctness based on a naive eventual consistency model as it fails to bound the delay of seeing other clients' updates while one client keeps making progress and that may lead to divergence.

Stronger consistency models such as sequential consistency and linearizability require serializable updates and that significantly restricts parallelism of the application.

Consistency models employed in modern distributed ML system tend to fall into two extremes: either sequential consitency or no consistency guarantee at all. For example, Distributed GraphLab serializes read and write accesses to vertices and edges by scheduling vertex programs according to a carefully colored graph or by locking [7]. Altougth such a model guarantees the correctness of the algorithm, it may under-utilize the distributed system's computing power. At the other extreme, YahooLDA [1] employs a best-effort consistency model where the system makes best effort to delivery updates but does not make any guarantee. Although the system empirically achieves good performance for LDA, there is no theoretical guarantee that the distributed implementation of the algorithm is correct. It is also unclear whether such system with loose consistency can generalize to a broad range of ML algorithms. In fact, the system can potentially fail if stragglers present or the network bandwith is saturated.

Recently, [5] presented the Stale Synchrnous Parallel (SSP) model, which is a variant of bounded staleness consistency model – a form of eventual consistency. Simimlar to Bulk Synchrnous Parallel (BSP), an execution of SSP consists of multiple iterations and each iteration is composed of a computation phase and a synchronization phase. Updates are sent out only during the synchronization phase. However, in SSP, a client is allowed to go beyond other clients by at most s iterations, where s is a threshold set by the application. In SSP, accesses to shared parameters are usually serviced by local replicas and network accesses only occur in case the local replica is more than s iterations stale. SSP delivers high throughput as it reduces network communication cost. [5] shows SSP is theoretically sound by proving the convergence of stochastic gradient descent.

Asynchronous parallel model improves system performance over BSP because 1) it uses CPU and network bandwidth in parallel (e.g. sending out updates whenever network bandwidth is available); 2) it does not enforce a synchronization barrier. Also, since the system makes best effort to send out updates (instead of waiting for synchoronization barrer), clients are more likely to compute with fresh data. Thus it may bring algorithmic benefits. It is more difficult to maintain consistency in an asynchronous system as communcation may happen anytime.

We have found that many ML algorithms are sufficiently robust to a bounded amount of inconsistency and thus admits consistency models weaker than serializability. By relaxing the consistency guarantees properly, the system may gain significant throughput improvements. In this paper, we present a few high-throughput consistency models that takes advantange of the robustness of ML algorithms. As shown in [5], even though relaxed consistency improves system throughput, it may result in reduced algorithmic progress per-iteration (e.g., smaller covergence rate). All our proposed consistency models allows application developers to tune the strictness of the consistency model to achieve the sweet spot. We present the following consistency models:

Clock-bounded Asynchronous Parallel (CAP): We apply the concept of "clock-bounded" consistency of SSP to asynchronous parameter server. Unlike SSP where updates are sent out only during the synchrnonization phase, CAP propgates updates whenever the netowrk bandwidth is available. Similar to SSP, CAP guarantees bounded staleness - a client must see all updates older than certain timestamp.

Value-bounded Asynchronous Parallel (VAP) This consistency model guarantees a bound on the absolute deviation between any two parameter replicas.

Clock-Value-bounded Asnychronous Parallel (CVAP) This model combines CAP and VAP to provide stronger consistency guarantee. With such a guarantee, the solution's quality (e.g. asymptotic variance) can be assessed.

2 Consistency Models

In this section, we present the formal definitions of CAP, VAP and CVAP. Consisder a collection of P workers which share access to a set of parameters. A worker writes to a parameter θ by applying an update in the form of an associate and commutative operation $\theta \leftarrow \theta + \delta$. The asynchronous parameter server propogates out the update at some point of time and the worker usually proceeds without waiting but may block occasionally to maintain consistency. Thus the parameter server may accumulate a set of updates generated by a worker which are not yet synchronized acroos all workers. All our consistency models ensure read-my-writes consistency. That is, a worker sees all its previous writes whether or not the updates are

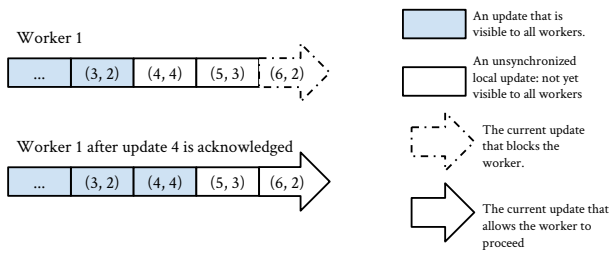


Figure 1: An illustration of VAP. The arrow represents a series of updates that a worker applies to a parameter. An update is representd by a pair of numbers (i, j) where i is the update's sequence number and j is the value of the update. In this example, the value bound is set to 8. Applying the 6th update blocks the worker as it would bring the accumulated unsynchronized updates beyond 8. After the 4th update becomes visible to all workers, the worker may procee with update $(6, 2)$.

synchronized with other workers. Intuitively, read-my-write consistency is desirable as it allows a worker to proceed with more fresh parameter. Our consistency models also ensure FIFO consistency [6] - updates from a single worker are seen by all other workers in the order in which they are issued. Intuitively, this consistency guarantee ensures that the updates are handled as fairly as the network ordering and prevents a worker from being biased by a particular subset of updates from another worker.

2.1 Clock-bounded Asynchronous Parallel (CAP)

Informally, Clock-bounded Asynchronous Parallel (CAP) ensures all workers are making sufficient progress forward, otherwise, faster workers are blocked to wait for the slow ones. Progress of a worker is represented by "clock", which is an integer which starts from 0 and is incremented at regular intervals. Updates generated between clock $[c-1, c]$ are timestamped with c. The consistency model guarantees that a worker with clock c sees all other clients' updates in the range of $[0, c-s-1]$, where s is a user-defined threshold.

We omit the proof of correctness for CAP as the analysis in [5] applies to CAP as well.

2.2 Value-bounded Asynchronous Parallel (VAP)

Since the asynchronous parameter server does not block workers when propagating out updates, a worker may accumulate a set of updates that are only visiable to itself. We refer this set of updates as *unsynchronized local updates*. As the update operation is associative and commutative, updates may be aggregated by summing them up. The Value-bounded Asynchronous Parallel (VAP) model guarantees that for any worker the accumulated sum s of *unsynchronized local updates* of any parameter is less than v_{thr} where v_{thr} is a user-defined threshold. When a worker attempts to apply an update that makes the accumulated sum exceed the threshold v_{thr}, the worker is blocked until the system has made a sufficient number of its updates visible to all workers. The VAP model is illustrated by Figure 1.

We should note that the VAP model described above still allows two workers to see two very different set of updates. For any pair of workers, say A and B, even though VAP restricts how they see updates generated within this pair, it makes no guarantees about seeing updates that are generated by other peer workers. For example, it can happen that worker A has seen one update from each other worker, but B hasn't seen any of them. Thus, VAP only provides a very loose bound on how two workers may read different values: for any two workers A and B, let θ_A and θ_B be the sum over all updates seen by A and B respectively. Then, the absolute difference between θ_A and θ_B, $|\theta_A - \theta_B|$, is upper bounded by $max(u, v_{thr}) \times P$, where u is an upper bound on the magnitude of any update, v_{thr} is the aforementioned user-defined threshold, and P is the num

We may provide a stronger consistency guarantee by restricting how workers see other workers' updates. We refer to an update as a *half-synchronized update* if the update is seen by at least other one worker (that did not generate the update), but has not yet been seen by all other workers. In addition to the guarantees provided by weak VAP, the strong VAP model guarantees that for any parameter, the total magnitude of all

half-synchronized updates is bounded by $max(u, v_{thr})$, where u and v_{thr} are as defined before. Thus, strong VAP guarantees that for any two workers A and B, $|\theta_A - \theta_B|$ is upper bounded by $2 * max(u, v_{thr})$. Note that this is independent of P, the number of workers.

2.3 Clock-Value-bounded Asynchronous Parallel (CVAP)

CAP may be combined with VAP to provide stronger consistency guarantees. The idea is that CVAP ensure all workers make enough progress but bounds the absolute difference between replicas. CVAP provides the consistency guarantees of both CAP and VAP. As VAP has a strong and a weak version, there are two versions of CVAP correspondingly. As we shown in Section 3, CVAP allows application developers to guarantee a certain level of solution quality.

3 Theoretical Analysis

We choose the Stochastic Gradient Descent algorithm as our example application and prove its convergence under VAP. Our proof utilizes the techniques developed in [5]. As in [5], the VAP model supports operations $\mathbf{x} \leftarrow \mathbf{x} \oplus (z \cdot \mathbf{y})$, where $\mathbf{x}, \mathbf{y}$ are members of a ring with an abelian operator $\oplus$ (such as addition), and a multiplication operator $\cdot$ such that $z \cdot \mathbf{y} = \mathbf{y}'$ where $\mathbf{y}'$ is also in the ring. We shall informally refer to $\mathbf{x}$ as the "system state", $\mathbf{u} = z\mathbf{y}$ as an "update", and to the operation $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{u}$ as "writing an update".

As defined in section 2.2 the updates $\mathbf{u}$ are accumulated and are propagated to server when they are greater than v_{thr}. We call these propagation times as t_p. We define $\mathbf{u}_{p,t_p}$ as the accumulated update written by worker p at propagation time t_p through the write operation $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{u}_{p,t_p}$. The updates $\mathbf{u}_{p,t_p}$ are a function of the system state $\mathbf{x}$, and under the VAP model, different workers will "see" different, noisy versions of the true state $\mathbf{x}$. $\tilde{\mathbf{x}}_{p,t_p}$ is the noisy state read by worker p at time t_p, implying that $\mathbf{u}_{p,t_p} = G(\tilde{\mathbf{x}}_{p,t_p})$ for some function G. Formally, in VAP $\tilde{\mathbf{x}}_{p,t_p}$ can take:

Value Bounded Staleness: Fix a staleness s. Then, the noisy state $\tilde{\mathbf{x}}_{p,t_p}$ is equal to

$$\tilde{\mathbf{x}}_{p,t_p} = \mathbf{x}_0 + \underbrace{\left[\sum_{p'=1}^{P}\sum_{t=1}^{t_{p'}-1} \mathbf{u}_{p',t}\right]}_{\text{guaranteed pre-propagation updates}} + \underbrace{\mathbf{u}_{p,t_p}}_{\text{guaranteed read-my-writes updates}} + \underbrace{\left[\sum_{(p',t_{p'})\in\mathcal{S}_{p,t_p}} \mathbf{u}_{p',t_{p'}}\right]}_{\text{best-effort in-propagation updates}}, \quad (1)$$

where $\mathcal{S}_{p,t_p} \subseteq \mathcal{W}_{p,t} = ([1,P] \setminus \{p\}) \times \{t_1, t_2, ..., t_P\}$ is some subset of the updates $\mathbf{u}$ written in between propagations $t_p - 1$ and t_p "window" and does not include updates from worker p. In other words, the noisy state $\tilde{\mathbf{x}}_{p,t_p}$ consists of three parts:

1. Guaranteed "pre-propagation" updates from begining to $t_{p'} - 1$, for every worker p'.
2. Guaranteed "read-my-writes" set $\mathbf{u}_{p,t_p}$ that covers all "in-window" updates made by the querying worker p.
3. Best-effort "in-window" updates $\mathcal{S}_{p,t_p}$ (not counting updates from worker p).

As with [5], VAP also generalizes the Bulk Synchronous Parallel (BSP) model:

BSP Lemma: Under zero staleness VAP reduces to BSP. **Proof:** $\tilde{\mathbf{x}}_{p,t_p}$ exactly consists of all updates until time t_p. □

We now define a reference sequence of states $\mathbf{x}_t$, informally referred to as the "true" sequence :

$$\mathbf{x}_t = \mathbf{x}_0 + \sum_{t'=0}^{t} \mathbf{u}_{t'}, \qquad \text{where} \quad \mathbf{u}_t := \mathbf{u}_{t \bmod P, \lfloor t/P \rfloor}.$$

In other words, we sum updates by first looping over workers (t mod P), then over time-intervals $\lfloor t/P \rfloor$. Now, let us use VAP to bound the difference between the "true" sequence $\mathbf{x}_t$ and the noisy views $\tilde{\mathbf{x}}_{p,t_p}$:

Lemma 1: Assume $s \geq 1$, and let $\tilde{\mathbf{x}}_t := \tilde{\mathbf{x}}_{t \bmod P, \lfloor t/P \rfloor}$, so that

$$\tilde{\mathbf{x}}_t = \mathbf{x}_t + \Delta_t, \tag{2}$$

where Δ_t is the the difference (i.e. error) between $\tilde{\mathbf{x}}_t$ and $\mathbf{x}_t$. We claim that $\|\Delta_t\| \leq 2v_{thr}\sqrt{K}(P-1)$, where $\|\cdot\|$ is the ℓ_2 norm, and K is the dimension of $\mathbf{x}$. **Proof**: As argued earlier, strong VAP implies $\|\Delta_t\|_\infty \leq 2v_{thr}(P-1)$. The result immediately follows since $\|y\| \leq \sqrt{K}\|y\|_\infty$ for all y.

Theorem 1 (SGD under VAP): Suppose we want to find the minimizer $\mathbf{x}^*$ of a convex function $f(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} f_t(\mathbf{x})$, via gradient descent on one component ∇f_t at a time. We assume the components f_t are also convex. Let the updates $\mathbf{u}_t := -\eta_t \nabla f_t(\tilde{\mathbf{x}}_t)$, with decreasing step size $\eta_t = \frac{\sigma}{\sqrt{t}}$. Also let the VAP threshold v_{thr} decrease according to $v_t = \frac{\delta}{\sqrt{t}}$. Then, under suitable conditions (f_t are L-Lipschitz and the distance between two points $D(x\|x') \leq F^2$),

$$R[\mathbf{X}] := \sum_{t=1}^{T} [f_t(\tilde{\mathbf{x}}_t) - f(\mathbf{x}^*)] \leq \sigma L^2 \sqrt{T} + F^2 \frac{\sqrt{T}}{\sigma} + 4\delta LP\sqrt{KT}$$

Dividing both sides by T, we see that VAP converges at rate $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$ when all other quantities are fixed.

3.1 Proof of Theorem 1

We follow the proof of [5]. Define $D(x\|x') := \frac{1}{2}\|x - x'\|^2$, where $\|\cdot\|$ is the ℓ_2 norm.

Proof: We have ,

$$\begin{aligned} R[X] &:= \sum_{t=1}^{T} f_t(\tilde{x}_t) - f_t(x^*) \\ &\leq \sum_{t=1}^{T} \langle \nabla f_t(\tilde{x}_t), \tilde{x}_t - x^* \rangle \qquad (f_t \text{ are convex}) \\ &= \sum_{t=1}^{T} \langle \tilde{g}_t, \tilde{x}_t - x^* \rangle . \end{aligned}$$

where we have defined $\tilde{g}_t := \nabla f_t(\tilde{x}_t)$. The high-level idea is to show that $R[X] \leq o(T)$, which implies $\mathbb{E}_t[f_t(\tilde{x}_t) - f_t(x^*)] \to 0$ and thus convergence. First, we shall say something about each term $\langle \tilde{g}_t, \tilde{x}_t - x^* \rangle$.

Lemma 2: If $X = \mathbb{R}^n$, then for all x^*,

$$\langle \tilde{x}_t - x^*, \tilde{g}_t \rangle = \frac{1}{2}\eta_t \|\tilde{g}_t\|^2 + \frac{D(x^*\|x_t) - D(x^*\|x_{t+1})}{\eta_t} + \langle \Delta_t, \tilde{g}_t \rangle$$

Proof:

$$\begin{aligned} D(x^*\|x_{t+1}) - D(x^*\|x_t) &= \frac{1}{2}\|x^* - x_t + x_t - x_{t+1}\|^2 - \frac{1}{2}\|x^* - x_t\|^2 \\ &= \frac{1}{2}\|x^* - x_t + \eta_t \tilde{g}_t\|^2 - \frac{1}{2}\|x^* - x_t\|^2 \\ &= \frac{1}{2}\eta_t^2 \|\tilde{g}_t\|^2 - \eta_t \langle x_t - x^*, \tilde{g}_t \rangle \\ &= \frac{1}{2}\eta_t^2 \|\tilde{g}_t\|^2 - \eta_t \langle \tilde{x}_t - x^*, \tilde{g}_t \rangle - \eta_t \langle x_t - \tilde{x}_t, \tilde{g}_t \rangle \\ &= \frac{1}{2}\eta_t^2 \|\tilde{g}_t\|^2 - \eta_t \langle \tilde{x}_t - x^*, \tilde{g}_t \rangle - \eta_t \langle -\Delta_t, \tilde{g}_t \rangle \end{aligned}$$

Thus,

$$\begin{aligned} D\left(x^*\|x_{t+1}\right) - D\left(x^*\|x_t\right) &= \frac{1}{2}\eta_t^2 \left\|\tilde{g}_t\right\|^2 - \eta_t \left\langle \tilde{x}_t - x^*, \tilde{g}_t \right\rangle - \eta_t \left\langle -\Delta_t, \tilde{g}_t \right\rangle \\ \frac{D\left(x^*\|x_{t+1}\right) - D\left(x^*\|x_t\right)}{\eta_t} &= \frac{1}{2}\eta_t \left\|\tilde{g}_t\right\|^2 - \left\langle \tilde{x}_t - x^*, \tilde{g}_t \right\rangle - \left\langle -\Delta_t, \tilde{g}_t \right\rangle \\ \left\langle \tilde{x}_t - x^*, \tilde{g}_t \right\rangle &= \frac{1}{2}\eta_t \left\|\tilde{g}_t\right\|^2 + \frac{D\left(x^*\|x_t\right) - D\left(x^*\|x_{t+1}\right)}{\eta_t} + \left\langle \Delta_t, \tilde{g}_t \right\rangle. \end{aligned}$$

This completes the proof of Lemma 2. □

Back to Theorem 1: Returning to the proof of Theorem 1, we use Lemma 2 to expand the regret $R[X]$:

$$\begin{aligned} R\left[X\right] \leq \sum_{t=1}^{T} \left\langle \tilde{g}_t, \tilde{x}_t - x^* \right\rangle &= \sum_{t=1}^{T} \frac{1}{2}\eta_t \left\|\tilde{g}_t\right\|^2 + \sum_{t=1}^{T} \frac{D\left(x^*\|x_t\right) - D\left(x^*\|x_{t+1}\right)}{\eta_t} + \sum_{t=1}^{T} \left\langle \Delta_t, \tilde{g}_t \right\rangle \\ &= \sum_{t=1}^{T} \left[\frac{1}{2}\eta_t \left\|\tilde{g}_t\right\|^2 + \left\langle \Delta_t, \tilde{g}_t \right\rangle\right] \\ &\quad + \frac{D\left(x^*\|x_1\right)}{\eta_1} - \frac{D\left(x^*\|x_{T+1}\right)}{\eta_T} + \sum_{t=2}^{T} \left[D\left(x^*\|x_t\right)\left(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}}\right)\right] \end{aligned}$$

We now upper-bound each of the terms:

$$\begin{aligned} \sum_{t=1}^{T} \frac{1}{2}\eta_t \left\|\tilde{g}_t\right\|^2 &\leq \sum_{t=1}^{T} \frac{1}{2}\eta_t L^2 \qquad \text{(Lipschitz assumption)} \\ &= \sum_{t=1}^{T} \frac{1}{2}\frac{\sigma}{\sqrt{t}} L^2 \\ &\leq \sigma L^2 \sqrt{T}, \end{aligned}$$

and

$$\begin{aligned} &\frac{D\left(x^*\|x_1\right)}{\eta_1} - \frac{D\left(x^*\|x_{T+1}\right)}{\eta_T} + \sum_{t=2}^{T} \left[D\left(x^*\|x_t\right)\left(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}}\right)\right] \\ \leq\ & \frac{F^2}{\sigma} + 0 + \frac{F^2}{\sigma} \sum_{t=2}^{T} \left[\sqrt{t} - \sqrt{t-1}\right] \qquad \text{(Bounded diameter)} \\ =\ & \frac{F^2}{\sigma} + \frac{F^2}{\sigma}\left[\sqrt{T} - 1\right] \\ =\ & \frac{F^2}{\sigma}\sqrt{T}, \end{aligned}$$

and

$$\begin{aligned} &\sum_{t=1}^{T} \left\langle \Delta_t, \tilde{g}_t \right\rangle \\ \leq\ & \left[\sum_{t=1}^{T} 2 v_t \sqrt{K}(P-1)L\right] \qquad \text{(Lemma 1 plus Lipschitz assumption)} \\ =\ & \left[\sum_{t=1}^{T} 2\frac{\delta}{\sqrt{t}}\sqrt{K}(P-1)L\right] \\ \leq\ & 4\delta LP\sqrt{KT}. \end{aligned}$$

Hence,

$$R[X] \leq \sum_{t=1}^{T} \langle \tilde{g}_t, \tilde{x}_t - x^* \rangle \quad \leq \quad \sigma L^2 \sqrt{T} + F^2 \frac{\sqrt{T}}{\sigma} + 4\delta LP\sqrt{KT}.$$

This completes the proof of Theorem 1. □

4 Parameter Server Design and Implementation

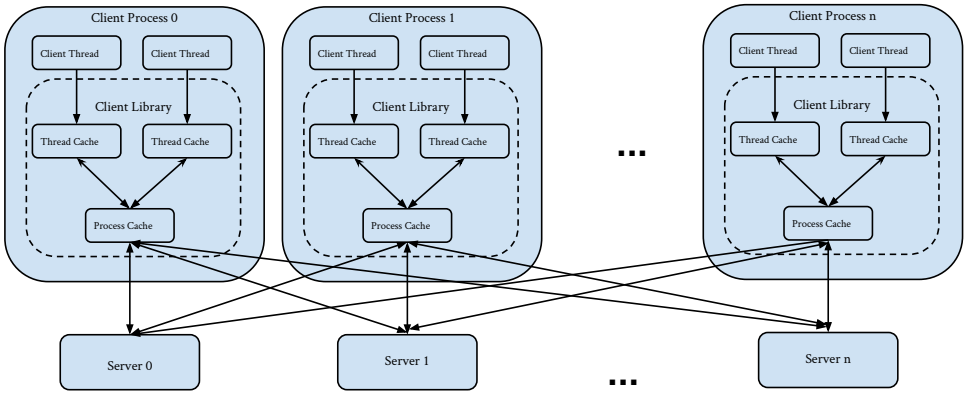


Figure 2: Parameter server architecture: a hierachical cache is used to minimize network communication overhead and reduce contention among application threads.

We implemented the proposed consitency models in a parameter server, called Petuum PS. For the purpose of comparison, Petuum PS also implemented SSP. Petuum PS is implemented in C++ and ZeroMQ is used for network communication.

Petuum PS contains a collection of server processes, which holds the shared parameters in a distributed fashion.. An application process accesses the shared parameters via a client library. The client library caches the parameters in order to minimize network communication overhead. An application process may contain multiple threads. The client library allocates a thread cache for each application thread to reduce contention among them. A thread is considered as a worker in our consistency mode description. The parameter server architecture is visualized in Fig 2.

4.1 System Abstraction

Petuum PS organizes shared parameters as tables. Thus a parameter stored in Petuum PS is identified by a triple of table id, row id and column id. Petuum PS supports both dense and sparse rows corresponding to dense and spares column index. A table is distributed across a set of server processes via hash partitioning and row is the unit of data distribution and transmission. The data stored in one table must be of the same type and Petuum PS can support an unlimited number of tables. It's worth mentioning that our implementation allows different tables to use different consistency model.

Petuum PS supports a set of straightforward APIs for data access. The main functions are:

- `Get(table_id, row_id, column_id)`: Retrieve an element from a table.
- `Inc(table_id, row_id, column_id, delta)`: Update an element by delta.
- `Clock()`: Increment the worker thread's clock by 1.

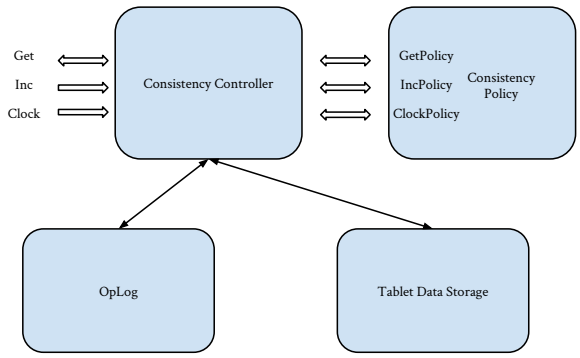


Figure 3: Consistency Controller and Consistency Policy

4.2 System Components

Petuum PS's client library employs a two-level cache hierachy to hide network access latency and minimize contension among application threads: all application threads within a process share a **process cache** which caches rows fetched from server. Each thread has its own **thread cache** and accesses to data are mostly serviced by thread cache as long as memory suffices. In order to support asynchronous computation, thread cache employs a write-back strategy.

In order to support Clock-based consistency models, the system needs to keep track of the clock of each worker. This is achieved via using vector clock. Each client library maintains a vector clock where each entity represents the clock of a thread. The minmum clock in the vector represents the progress of the process. Server treats a process as an entity and its vector clock keeps track of the progress of all processes.

Asynchronous system tends to congest the network with large volume of messages. Our client and server thus batch messages to achieve high throughput. Messages are sent out based on their priorities which might be application-dependent. We by default prioritize updates with larger magnitude as they are more likely to contribute to convergence.

4.3 Implementing Consistency Models

We implement SSP, CAP, VAP and CVAP in a unified, modular fashion. We realized that those consistency models can be implemented by performing different operations upon application threads accessing parameters. In other words, the exact sematics of the APIs depend on the consistency model being used. Each consistency model is expressed as a *Consistency Policy* data structure. Each table is associated with a *Consistency Controller*, which checks *Consistency Policy* and services user accesses accordingly. The consistency control logic is visualized in Fig 3.

Different semantics of the APIs include network communication and blocking wait for responses. Petuum PS uses three types of network communications:

- **Client Push**: Client pushes one or a batched set of updates to server.
- **Client Pull**: Client pull a row from server
- **Server Push**: Server pushes one or a batched set of updates to relevant clients.

Coupled with proper cache coherence mechanism, those APIs are sufficient for implementing our consistency models.

5 Evaluation

We evaluate our prototype parameter server via topic modeling. Latent Dirichlet Allocation (LDA) [2] is a popular unsupervised model that discovers a latent topic vector for each document. We implemented LDA

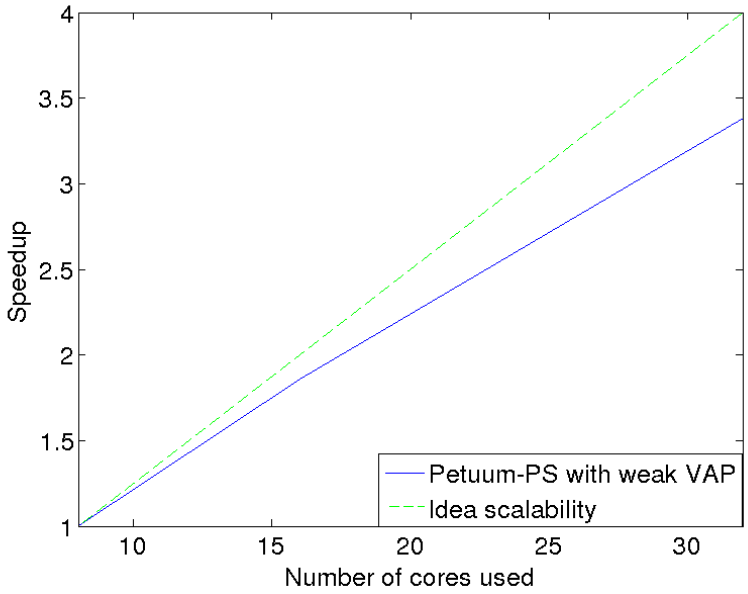
	20News
# of docs	11269
# of words	53485
# of tokens	1318299

Table 1: Summary statistics of two corpra used in LDA.

on our parameter server with the weak VAP model and conducted experiments on a 8-node cluster. Each node is equipped with 64 cores and 128GB of main memory and the nodes are connected with a 40 Gbps Ethernet network. We restrict our experiment to use at most 8 cores and 32GB of memory per machine to emmulate cluster with normal machines.

We used a relatively small dataset 20News and evaluted the strong scalability of the system in particular. Statistics of the 20News dataset are shown in Table 1.

We fixed the number of topcis to be 2000 while varying the number of workers. We assign each worker a core exclusively. We show the results in Figure 5 where the number of cores used ranges from 8 to 32. The curve showed the speed up using Petuum-PS vs. ideal linear scalability. Even though our experiments are conducted on a relatively small scale, the results show that it has a great potential to scale up.



Acknowledgments

We thank PRObE [4] and CMU PDL Consortium for providing testbed and technical support for our experiments.

References

[1] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 123–132, New York, NY, USA, 2012. ACM.

[2] David M. Blei, Andrew Ng, and Michael Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.

[3] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Proc. of the 14th Usenix Workshop on Hot Topics in Operating Systems*, HotOS '13. Usenix, 2013.

[4] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[5] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems 26*, pages 1223–1231. 2013.

[6] R. J. Lipton and J. S Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Sep 1988.

[7] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[8] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[9] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables.

[10] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, September 2010.

[11] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, December 2013.