UNIVERSITY OF CAMBRIDGE

Department of Computer Science and Technology

Research project report title page

Candidate **2475E**

"Probabilistic Synchronous Parallel: a barrier control method for distributed machine learning"

Submitted in partial fulfilment of the requirements for the Computer Science Tripos, Part III

Graduate Education Office, Room FS03
Department of Computer Science and Technology, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD

Probabilistic Synchronous Parallel: a barrier control method for distributed machine learning

Abstract

The synchronisation scheme used to manage parallel updates of a distributed machine learning model can dramatically impact performance. System and algorithm designers need methods which allow them to make trade-offs between fully asynchronous and fully deterministic schemes. Barrier control methods represent one possible solution. In this report, I present Probabilistic Synchronous Parallel (PSP), a barrier control method for distributed machine learning. I provide analytical proofs of convergence and carry out an experimental verification of the method using a bespoke simulator. I find that PSP improves the convergence speed and iteration throughput of more traditional barrier control methods. Furthermore, I demonstrate that PSP provides stronger convergence guarantees than a fully asynchronous design whilst maintaining the general characteristics of stronger methods.

Word count: 11,897

Contents

List of Figures

Chapter 1

Introduction

The rising size and complexity of datasets, along with a deepening desire to understand them using more flexible models, means that typical machine learning applications now require more computation resources [1]. Distributed machine learning techniques and frameworks solve this problem by spreading work across connected computers.

Traditionally, researchers focussed on specific machine learning algorithms [2]. However, literature is now emerging on more general machine learning platforms [4, 5, 6, 7]. This follows the thinking of Xing et al. [2] who propose that machine learning and systems approaches should be combined to provide general purpose distributed machine learning systems. They suggested such platforms offer a ready-made set of algorithms e.g., Stochastic Gradient Descent and Markov Chain Monte Carlo methods, which can be executed on top of a distributed system. Many different architectures have been proposed including MapReduce [7], GraphLab [8], Spark [6] and Parameter Servers [9]. Unfortunately, network delays, synchronization costs, a lack of scalability and non-homogeneous computation environments can make it hard to realise improvements [2].

To build these systems, intrinsic properties of machine learning algorithms need to be considered. These include:

- The *iterative-convergent* nature of many machine learning algorithms.
- Tolerance of errors in intermediate calculations.
- Dynamic and static structural dependencies. For example, changing correlations between model parameters.
- Non-uniform convergence: that is, parameters converging at different rates.

An iterative-convergent algorithm executes a sequence of iterations and has provable convergence properties which ensure it achieves an optimal solution. Such algorithms are typically error-tolerant [1], so will converge to local optimum even in the presence of errors or noise. They are typically characterised by the following formulation:

$$\max_A(\mathcal{L}(x, a)) \text{ OR } \min_A(\mathcal{L}(x, A))\,, \tag{1.1}$$

where $\mathcal{L}(x, A) = f(\{x_i, y_i\}_{i=1}^N; A) + r(A)$. Here, A represents the model and x the N data samples, $(x = \{x_i, y_i\})_{i=1}^N$. Note that y_i is only present for labelled data. The overall

objective function, $\mathcal{L}$, is composed of a loss function, f, which characterises how well the model fits the data, and a structure-inducing function, r, used to incorporate domain specific knowledge and limit the values parameters can take.

Different machine learning families have unique constructions and can be very complicated. Finding the optimal model requires algorithmic techniques such as Stochastic Gradient Descent (SGD), Markov Chain Monte Carlo (MCMC) and Variational Inference. These methods can be used to construct iterative-convergent algorithms which can be executed to find a suitable model.

To run a machine learning algorithm on multiple machines, the data and/or model needs to be partitioned. Data parallelism and model parallelism refer to the properties of an algorithm, and a set of techniques, which can be leveraged to yield viable partitions.

In model parallelism, the model is divided between workers and updated in parallel. A *scheduling* function is responsible for selecting a subset of the model for a worker to operate on. Its strategy must enable the machine learning algorithm to provably terminate with a reasonable quality of convergence.

Data parallelism is used to separate data into batches which can be operated on independently. As a result, data can be load-balanced across workers to handle heterogeneous computing power and performance fluctuations.

The two methods are complementary as sometimes both are applicable. However, they are also asymmetric as data often exhibits independence properties not enjoyed by model parameters.

Typically, the update equation has the following form:

$$A(t) = F(A(t-1), \sum_{p=1}^{P} \Delta_{\mathcal{L}}(A(t-1), x_p)) \,, \tag{1.2}$$

where t is the current time step. Here, F is a function which updates the model, A, using updates calculated on A at time $t-1$ and the data on the pth worker, x_p.

Once viable model and/or data partitions have been identified, careful consideration of the necessary synchronisation mechanisms is required to ensure correct and efficient distributed computation [10].

Barrier control methods offer one possible solution. These are algorithms whose purpose is to determine if a worker should proceed to the next iteration of a computation or if it should wait. The most dominant schemes are *Bulk Synchronous Parallel* (BSP) [11], *Stale Synchronous Parallel* (SSP) [4] and *Asynchronous Parallel* (ASP) [2].

BSP is a fully deterministic solution which requires all workers to complete an iteration before any worker can move on. ASP is fully asynchronous, providing no synchronisation at all. SSP offers a middle ground between the two, using a *staleness* parameter to control how far behind in the computation a worker is allowed to fall before other workers are forced to wait for it. This allows a designer to tune the system between BSP and ASP.

BSP produces high quality iterations [2]. That is, the noise/error in the model updates, Δ, is minimal. However, the presence of even a single sporadically delayed worker can have disastrous consequences for iteration throughput [4, 2]. SSP permits some delays but at the cost of decreased quality of convergence. Furthermore, as SSP enforces a hard limit on delays, if a single worker is sufficiently delayed, all the other workers must wait, as in

BSP. ASP works well in environments where workers are not delayed severely but suffers rapidly decreasing iteration quality if many and/or highly delayed workers appear [2].

SSP is typically seen as the best compromise between BSP and ASP [5]. Unfortunately, it lacks flexibility as it provides only one, rather coarse, parameter. Furthermore, increasing staleness too much leads to the same problems which ASP suffers from. The aim is thus to find methods which can maintain iteration quality and speed.

I present a new barrier control method: Probabilistic Synchronous Parallel (PSP). The idea is to introduce a sampling primitive which selects a subset of the workers upon which traditional barrier control methods are applied. This yields **pBSP** and **pSSP**: Probabilistic Synchronous Parallel for BSP and Probabilistic Synchronous Parallel for SSP respectively.

PSP can be viewed as providing an additional axis in the space of parameters currently available. This gives designers more flexibility in trading off between BSP and ASP. Furthermore, PSP can be executed independently on the workers of the system, rather than requiring a central oracle to make decisions. This reduces the need for synchronisation and communication.

My contribution includes analytical proofs of convergence which reveal that PSP provides stronger convergence guarantees than ASP. I also implemented two simulators which I used to validate the method. Unfortunately, my parameter server simulator provided insufficient fidelity so I only present results from my barrier control simulator. I evaluated PSP using Stochastic Gradient Descent (SGD) and Latent Dirichlet Allocation (LDA) due to the frequent attention they receive in the barrier control literature [2, 12, 13].

My experiments reveal that PSP is able to improve convergence speed and iteration throughput of traditional barrier control methods. I find that only a small proportion of workers need to be sampled to enforce similar step distributions to current methods. Furthermore, I demonstrate that the sampling primitive enables greater control of synchronisation trade-offs than provided by current methods.

Chapter 2

Background

A rich collection of designs and ideas have been proposed to build frameworks for programmers to develop distributed machine learning algorithms on [2]. My contribution is to provide theoretical and experimental evaluation of a new synchronisation method. In this section, I provide the academic context for my work. I cover the different synchronisation methods currently available and describe systems which employ them. I conclude the chapter with a discussion of parameter servers which I explored as an evaluation platform.

2.1 Barrier Control Methods

In the following sections, I describe each barrier control method in more detail, summarise related research and provide examples of systems using them (or a hybrid). Table 2.1 shows how the synchronisation methods of these systems can be classified.

System	Synchronisation	Barrier Method
MapReduce [7]	Requires map to complete before reducing	BSP
Spark [6]	Aggregate updates after task completion	BSP
Pregel [14]	Superstep model	BSP
Hogwild! [15]	ASP but system-level bounds on delays	ASP, SSP
Parameter Servers [9]	Swappable synchronisation method	BSP, ASP, SSP
Cyclic Delay [16]	Updates delayed by up to $N-1$ steps	SSP
Yahoo! LDA [13]	Checkpoints	SSP, ASP

Table 2.1: Classification of the synchronisation methods used by different systems.

Bounded Synchronous Parallel (BSP) BSP is a deterministic scheme where workers perform a computation phase followed by a synchronisation/communication phase where they exchange updates [2]. The method ensures that all workers are on the same iteration of a computation by preventing any worker from proceeding to the next step until all can. Furthermore, the effects of the current computation are not made visible to other workers until the barrier has been passed. Provided the data and model of a distributed algorithm have been suitably scheduled, BSP programs are often serializable — that is, they are

equivalent to sequential computations. This means that the correctness guarantees of the serial program are often realisable making BSP the strongest barrier control method [5]. Unfortunately, BSP does have a disadvantage. As workers must wait for others to finish, the presence of *stragglers*, workers which require more time to complete a step due to random and unpredictable factors [2], limit the computation efficiency to that of the slowest machine. This leads to a dramatic reduction in performance. Overall, BSP tends to offer high computation accuracy but suffers from poor efficiency in unfavourable environments.

Asynchronous Parallel (ASP) ASP takes the opposite approach to BSP, allowing computations to execute as fast as possible by running workers completely asynchronously. In homogeneous environments, wherein the workers have similar configurations, ASP enables fast convergence because it permits the highest iteration throughputs. Typically, P-fold speed-ups can be achieved [2]. However, such asynchrony causes delayed updates: updates calculated on an old model state which should have been applied earlier but were not. Applying them introduces noise and error into the computation. Consequently, ASP suffers from decreased iteration quality and may even diverge in unfavourable environments. Overall, ASP offers excellent speed-ups in convergence but has a greater risk of diverging.

Stale Synchronous Parallel (SSP) SSP is a bounded-asynchronous model which can be viewed as a relaxation of BSP. Rather than requiring all workers to be on the same iteration, the system decides if a worker may proceed based on how far behind the slowest worker is. Specifically, a worker which is more than s iterations behind the fastest worker is considered too slow. If such a worker is present, the system pauses faster workers until the straggler catches up. This s is known as the *staleness* parameter.

More formally, each machine keeps an iteration counter, c, which it updates whenever it completes an iteration. Each worker also maintains a local view of the model state. After each iteration, a worker commits updates, i.e., Δ, which the system then sends to other workers, along with the worker's updated counter. The bounding of clock differences through the staleness parameter means that the local model cannot contain updates older than $c - s - 1$ iterations. This limits the potential error. Note that systems typically enforce a read-my-writes consistency model.

The staleness parameter allows SSP to provide deterministic convergence guarantees [2, 1, 5]. Note that SSP is a generalisation of BSP: setting $s = 0$ yields the BSP method. Furthermore, setting $s = \infty$ produces ASP. Overall, SSP offers a good compromise between fully deterministic BSP and fully asynchronous ASP [5].

In the following sections, I provide a summary of the barrier control literature and describe systems using these methods.

2.1.1 Bulk Synchronous Parallel (BSP)

BSP was introduced in 1990 by Valiant [11] who applied it to parallel computations. His system's architecture consists of a set of *components* which perform processing, a *router*

which delivers messages, and a facility for synchronising all/some of the components at regular intervals of L time units. Computations are modelled as sequences, composed of *supersteps*.

Every L time units, the system performs a global check to determine if all components have finished their current superstep. If so, all machines proceed to the next superstep. Otherwise, faster workers wait until the remaining components have finished. This is clearly a BSP formulation. Valiant also discusses an ASP mode which is used when selected supersteps are independent.

More recently, Bradley et al. [17] parallelized coordinate descent for L_1-regularised losses, parallelizing over the parameters of the model. In each iteration, P updates are computed and no further work is done until all of these updates are finished. Again, this fits the BSP scheme. Their work is notable as they proved their method converges which, to their knowledge, provided the first proofs for parallelization methods on these algorithms. This analysis was later refined by others such as Ho et al. [5].

I now present several use cases of BSP.

MapReduce

MapReduce is a programming model designed for processing and generating large datasets [7]. Programmers specify a *map* function which processes key/value pairs to generate a set of intermediate key/value pairs. A *reduce* function is defined which combines intermediate values associated with the same intermediate key. More abstractly, this can be viewed as specifying acyclic data flow graphs and a set of operators, through which input is passed [6].

The runtime system is responsible for partitioning the input data, scheduling execution across a group of machines and for handling system failures. Programmers are thus relieved of the need to consider such issues.

The programming model requires that all map tasks complete before any reduce tasks are executed to ensure that all required data is available at the next step[18]. This requirement implements the concept of the BSP method.

Spark

Many iterative machine learning algorithms reuse a common working set of data across multiple parallel operations [6]. Spark is a framework which supports such applications whilst retaining the scalability and fault tolerance of MapReduce [6]. Indeed, Spark has been shown to outperform Hadoop (a MapReduce platform) by up to 10x in iterative machine learning jobs [6].

Spark provides an abstraction called *resilient distributed datasets* (RDDs): a read-only collection of objects, partitioned and then distributed across a group of machines. Developers write driver programs which apply parallel operations such as reduce, collect and foreach to RDDs. Furthermore, map, filter and reduce can be invoked on shared variables which are either *broadcast* variables or *accumulators*. Broadcast variables are containers that handle distribution of read-only data to workers. Accumulators collect the overall results of a computation.

When creating a task, Spark partitions the data and then distributes these, along with the required sequence of operations, amongst the workers. A task stores any updates it makes to accumulators locally. The system reads these updates at a later time and applies them to the global accumulator. This design is in keeping with the BSP paradigm.

Pregel

The Pregel [14] system is designed for graph-based computations. Such programs consist of a sequence of traversals through a graph. Pregel is thus particularly useful for graph processing applications. However, the programmer needs to be able to express the program in a graph-based model which is not always possible [1].

Pregel uses the Bulk Synchronous model by Valiant [11]. In a superstep, a user-defined function is invoked which specifies the behaviour at a single vertex, v, for the current superstep, s. This function uses messages sent to the vertex in the previous superstep to execute the current computation. The result is a separation of updates between iterations, as in BSP.

2.1.2 Asynchronous Parallel (ASP)

The ASP model is less popular in distributed machine learning frameworks than BSP since its convergence properties depend heavily on the computation and networking environment. However, the recently introduced parameter server paradigm has led to fresh evaluation of the method [9, 1].

Niu et al. [15] presented Hogwild! in 2011, a lock-free approach to parallelizing stochastic gradient descent (SGD) over many processors. The method sees SGD run in a multi-core, shared memory system without deploying any form of locking or synchronisation. Consequently, the algorithm performs as if it were using ASP.

The key insight was that memory accesses are typically random and sparse meaning that accesses are unlikely to collide. Thus, updates based on old gradients are improbable enough that, when combined with SGD's inherent error-tolerance, convergence can be achieved. However, Niu et al. assume that the maximum delay of an update is bounded. Thus, although they claim to require no synchronisation, their method actually relies on an SSP-like system-level guarantee. Careful system design, or additional synchronisation, is thus required to ensure convergence. The authors argue that homogeneous computation environments, such as multi-core chips, meet these requirements.

2.1.3 Stale Synchronous Parallel (SSP)

In 2013, Cipar et al. [4] introduced the Stale Synchronous Parallel (SSP) model as a generalisation of BSP. The intention is to overcome the straggler problem. Previous work had tried reducing the number of stragglers caused by hardware or operating system effects [4]. Unfortunately, such methods do not solve the root problem. This is being able to compute correctly and efficiently in the presence of stragglers. Cipar et al. suggested allowing workers to lag up to s steps behind the fastest worker. They experimentally analysed the convergence behaviour of this method using Latent Dirichlet Allocation and

found that it did indeed converge. Although the rate of convergence decreased with increasing s. The authors stopped short of carrying out a theoretical analysis which others, such as Ho et al. [5], later provided.

Cyclic delay

In 2009, Langford et al. [16] presented the first theoretical guarantees for online learning algorithms which operate on a shared parameter vector. They specifically examined stochastic gradient descent (SGD).

Workers compute a gradient, g_t, and then update the parameter vector, x, in a round-robin fashion. A *cyclic delay* of $n-1$, where n is the number of workers, is thus present between the computation of a gradient and a parameter update. This process is similar to an SSP method.

Langford et al. provide deterministic bounds for the error between the current estimate, x_t, and the optimal estimate, $x^\star$. They explored the worst case which occurs when an adversary picks the most harmful update order. However, the bounds can be improved if this assumption is relaxed.

Agarwal et al. [19] researched delayed asynchronous updates using a cyclic delay update architecture. They presented convergence rates for delayed optimisation of smooth functions but went further than Langford et al. [16] by showing that, for smooth stochastic problems, the delay is asymptotically negligible. That is, the time, τ, that an update is delayed does not matter provided most processors remain synchronised for most of the time. However, this statement imposes SSP-like restrictions on the network and delay processes which, in practice, require synchronisation overheads or extremely careful system design.

Yahoo! LDA

In 2012, Ahmed et al. [13] presented Yahoo! LDA, a distributed machine learning system which used the Latent Dirichlet Allocation model to assign topics to a set of documents.

The global model is distributed amongst a set of machines and kept synchronised. Clients update local copies of parameters and notify servers of the updates. These servers aggregate updates from multiple workers before forwarding them onto other clients. This process requires the parameters to be closed under addition/subtraction in $\mathbb{R}$ or for multiplication/division in $\mathbb{R}^+$, defining a ring [13]. Their design is an early form of the recent parameter server paradigm.

To synchronize all local models, and checkpoint the system, all communication is halted. The system then waits for each worker's message queue to empty, ensuring that all updates have been processed. Checkpoints are performed at regular intervals and no worker is allowed to move past one until it completes. Furthermore, a client can only have a fixed number of messages in-flight at one time and cannot operate on the same parameter in parallel. This is a bounded best-effort scheme similar to SSP.

2.2 Parameter Server

To evaluate PSP, I explored a peer-to-peer parameter server implementation. In this section, I summarise the relevant literature.

Parameter servers distribute the parameters of a model across a set of servers and the data across a set of clients. Clients run a machine learning algorithm on their local data, requesting parameters and sending updates to the servers.

Ho et al. [5] and Cipar et al. [4] presented distributed, shared memory parameter server designs whilst others, such as Li et al. [9] and Ahmed et al. [13], presented a distributed push/pull model. Trade-offs exist between the two methods. The former reduces the programming complexity of implementing new algorithms but superior efficiency gains can be achieved with the later [1].

Ho et al. [5] used SSP to provide convergence guarantees in their system. They proved that, using SSP, stochastic gradient descent converges. In particular, they demonstrated that it converges at least as quickly as cyclic delay systems [5]. Their paper also evaluated the system using topic modelling and matrix factorisation algorithms. They found that SSP converged faster than BSP and ASP. Ho et al. concluded that their use of SSP hits a sweet spot between progress per iteration and iteration throughput.

Li et al. [9] provided a more explicit parameter server design, particularly applicable to machine learning algorithms where both data and model parallelism are present. Such models include Latent Dirichlet Allocation, risk minimisation by distributed subgradient iterations and Deep Learning [9]. Their design stores parameters in a key-value vector and provides both a push and a pull operation. Push sends local modifications of shared parameters to servers and pull retrieves them. The communication model is asynchronous and can either follow a best effort, ASP-like model or a bounded delay, SSP-like model. User-defined filters can be specified which can further restrict model updates. For example, forcing a push if a parameter change exceeds a specified threshold value.

Li et al. [9] provided theoretical analysis of their design but assumed that in-transmit updates were bounded by guarantees not explicitly provided by a feature of the parameter server. Dai et al. [1] extended their work, presenting the Value-bounded Asynchronous Parallel (VAP) method. They propose this as an ideal model because it approximates a strong consistency model .

2.3 Summary

Current barrier control methods offer the choice between completely deterministic computation (BSP), fully asynchronous computation (ASP) and a bounded non-deterministic scheme (SSP).

To fully support BSP and SSP, systems need to have a central copy of the entire system's state. However, in distributed settings, such centralisation of state requires additional communication and synchronisation overhead. ASP has no need for an oracle but offers poorer convergence guarantees.

Consequently, there is a need for a scalable synchronisation scheme which can operate on distributed state and provide stronger convergence properties than ASP. I present

Probabilistic Synchronous Parallel (PSP) as one possible solution. In the next chapter, I present a formal analysis of its convergence properties.

Chapter 3

Analytical Proofs of Convergence for PSP

In this chapter, I formalise each barrier control method and then provide analytical proofs of convergence for both ASP and PSP.

3.1 Formal Barrier Control Methods

To pass a barrier, a worker must satisfy the specified conditions of the barrier control method. Otherwise, the worker needs to wait until the conditions become true. In the following definitions, let V be the set of all workers in the system and s_i the step of worker i.

BSP BSP requires all workers to be on the same step:

$$\forall i, j \in V . s_i = s_j \,. \tag{3.1}$$

SSP SSP enforces an upper limit on the lag a worker can experience:

$$\forall i, j \in V . |s_i - s_j| \leq s \,, \tag{3.2}$$

where s is a parameter known as *staleness*.

ASP No synchronisation is performed in ASP.

$$\top \tag{3.3}$$

pBSP PSP generalises the previous methods. Specifically, only a subset, $S \subseteq V$, of the workers are tested. For pBSP, this yields:

$$\forall i, j \in S \subseteq V . s_i = s_j \,. \tag{3.4}$$

If $S = V$, then pBSP reduces to BSP and if $S = \emptyset$, then it reduces to ASP.

pSSP The most general method of PSP is pSSP. Here, sampling a subset of workers, $S \subseteq V$, leads to:

$$\forall i, j \in S \subseteq V. |s_i - s_j| \leq s \,. \tag{3.5}$$

Clearly, if $S = V$, then the method becomes SSP and if $S = \emptyset$ or $s = \infty$, then the method reduces to ASP. Furthermore, if $s = 0$ then pSSP reduces to pBSP.

3.2 General Proof Framework

To prove convergence, I first consider a general form of a distributed machine learning algorithm. I formalise the process of updating the model by casting this as a sequence of updates. The true sequence of updates encodes the results expected from a fully deterministic barrier control system, such as BSP. A noisy sequence represents scenarios where updates are reordered due to sporadic and random network and system delays. In my analysis, I consider the different restrictions each barrier control method enforces on this noisy sequence.

With these notions, the convergence properties of barrier control methods can be analysed. Specifically, I find a bound on the difference between the true sequence and a noisy sequence. By showing that this bound has certain properties, I prove convergence.

As a proof of convergence for ASP requires much of the work needed in a proof for PSP, I first prove that SGD under ASP converges and then refine this proof for PSP.

Dai et al. [1] presented bounds for SSP demonstrating that, with suitable conditions, SGD under SSP converges deterministically. Ho et al. [5] presented an alternative method and proved probabilistic convergence for SSP. My analysis extends the method used for the deterministic bounds but differs when forming the probabilistic bounds.

In distributed machine learning, a series of updates are applied to a stateful model, represented by a d-dimensional vector, $\mathbf{x} \in \mathbb{R}^d$, at clock times denoted by $c \in \mathbb{Z}$. Let the number of workers in the system be P and let $\mathbf{u}_{p,c} \in \mathbb{R}^d$ be the pth worker's updates at clock time c. These updates are applied to the system state, $\mathbf{x}$. Let t be the index of the serialised updates.

Define the true sequence of updates as taking the sum over all the workers t mod P and then over all iterations, $\lfloor \frac{t}{P} \rfloor$:

$$\mathbf{x}_t = \mathbf{x}_0 + \sum_{j=0}^{t} \mathbf{u}_j \,, \tag{3.6}$$

where $\mathbf{u}_j = \mathbf{u}_{t \bmod P, \lfloor \frac{t}{P} \rfloor}$. The sum is taken as SGD aggregates updates by summing them before applying them to the model. The ordering of the sequence is chosen to represent a deterministic BSP-like system, which would aggregate all updates from the workers and then proceed to the next step. It is worth emphasizing that multiple values of t occur in a given iteration, $\lfloor \frac{t}{P} \rfloor$, and clock time, c.

As some updates will be in progress in the network, we have a noisy state, $\tilde{\mathbf{x}}_{p,c}$, which is read by worker p at time c. Let $\tilde{\mathbf{x}}_t = \tilde{\mathbf{x}}_{t \bmod P, \lfloor \frac{t}{P} \rfloor}$ so that,

$$\tilde{\mathbf{x}}_t = \mathbf{x}_t - \sum_{i \in \mathcal{A}_t} \mathbf{u}_i + \sum_{i \in \mathcal{B}_t} \mathbf{u}_i \,. \tag{3.7}$$

Here, $\mathcal{A}_t$ and $\mathcal{B}_t$ are the index sets of updates where $\mathcal{A}_t$ holds missing updates (those missing from the noisy representation but which are in the true sequence) and $\mathcal{B}_t$ holds the extra updates (those in the noisy sequence but which are not in the true sequence).

To perform convergence analysis, the difference between the true sequence, $\mathbf{x}_t$, and the noisy sequence, $\tilde{\mathbf{x}}_{p,c}$, can be examined.

3.3 Convergence of SGD under ASP

Theorem 1: SGD under ASP

Let $f(\mathbf{x}) = \sum_{t=1}^{T} f_t(\mathbf{x})$ be a convex function where each $f_t \in \mathbb{R}$ is also convex and $\mathbf{x} \in \mathbb{R}^d$. Let $\mathbf{x}^\star \in \mathbb{R}^d$ be the minimizer of this function. Assume that f_t are L-Lipschitz, where L is constant, and that the distance between two points $\mathbf{x}$ and $\mathbf{x}'$ is bounded: $D(\mathbf{x}||\mathbf{x}') = \frac{1}{2}||\mathbf{x} - \mathbf{x}'||_2^2 \leq F^2$, where F is constant. This distance measure can be used to bound the magnitude of the differences between two states.

Let an update be given by $\mathbf{u}_t = -\eta_t \nabla f_t(\tilde{\mathbf{x}}_t)$ and the learning rate by $\eta_t = \frac{\sigma}{\sqrt{t}}$ with constant σ. Here, σ can be used to adjust the learning rate to ensure convergence.

Represent the lag of updates due to network overheads and the different execution speeds of the P workers by a vector, $\gamma_t \in \mathbb{R}^d$, which consists of random variables, Y_i. These Y_i are assumed i.i.d and independent of $\mathbf{u}_t$ and $\tilde{\mathbf{x}}_t$. Assume the same distribution for each γ_t so that $\forall t.\mathbb{E}(\gamma_t) = \mu$ and $\forall t.\mathbb{E}(\gamma_t^2) = \phi$. That is, they have constant mean and variance.

Following the presentation of regret by Ho et al. [5], let $R[X] = \sum_t^T f_t(\tilde{\mathbf{x}}_t) - f_t(\mathbf{x}^\star)$. This is the sum of the differences between the optimal value of the function and the current value given a noisy state. A probabilistic bound on the regret allows us to infer if the noisy system state, $\tilde{\mathbf{x}}_t$, is expected to converge towards the optimal, $\mathbf{x}^\star$, in probability. One such bound on the regret is given by:

$$\mathbb{P}\left(\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\sigma L^2 - \frac{2F^2}{\sigma}\right) - 4P\sigma L\mu \geq \delta\right) \leq \exp\left(-\frac{T\delta^2}{16P^2\sigma^2L^2\phi + \frac{b\delta}{3}}\right), \quad (3.8)$$

for constant δ and $b \leq 4PT\sigma L$.

The b term here is the upper bound on the random variables which are drawn from the lag distribution. If we assume with probability Φ that $\forall t.4PL\sigma\gamma_t < O(T)$, then $b < O(T)$ so $\frac{R[X]}{T}$ converges to $O(T^{-1/2})$ in probability with an exponential tail bound with probability Φ.

Proof

The start of the proof proceeds as in Dai et al. [1] and Ho et al. [5]. After some manipulation, the regret term reduces into some deterministic terms and a probabilistic term. I find bounds for the deterministic terms and then the probabilistic term. Finally, I use a one-sided Bernstein inequality, and the bound on the regret, to show that ASP will converge in probability.

Manipulating the regret Starting with the definition of the regret:

$$R[X] = \sum_{t=1}^{T} f_t(\tilde{\mathbf{x}}_t) - f_t(\mathbf{x}^\star) \tag{3.9}$$

$$\leq \sum_{t=1}^{T} \langle \nabla f_t(\tilde{\mathbf{x}}_t), \tilde{\mathbf{x}}_t - \mathbf{x}^\star \rangle \text{ (by } f_t \text{ convex)} \tag{3.10}$$

$$= \sum_{t=1}^{T} \langle \tilde{\mathbf{g}}_t, \tilde{\mathbf{x}}_t - \mathbf{x}^\star \rangle \,. \tag{3.11}$$

Here, $\tilde{\mathbf{g}}_t = \nabla f_t(\tilde{\mathbf{x}}_t)$ and $\langle \mathbf{a}, \mathbf{b} \rangle$ is the dot product of $\mathbf{a}$ and $\mathbf{b}$.

Now, find a bound so that $R[X] < O(T)$, meaning that $\mathbb{E}(f_t(\tilde{\mathbf{x}}_t) - f_t(\mathbf{x}^\star)) \to 0$ as $t \to \infty$. A bound involving steps t and $t+1$ is particularly useful. Ho et al. [5] prove the following lemma:

Lemma 1 Let $\tilde{\mathbf{g}}_t$ be as defined above, $D(\mathbf{x}||\mathbf{x}') = \frac{1}{2}||\mathbf{x} - \mathbf{x}'||_2^2$, and $\mathcal{A}_t$ and $\mathcal{B}_t$ are the index sets. The dot product between $\tilde{\mathbf{g}}_t$ and the difference between the current state, $\tilde{\mathbf{x}}_t$, and the optimal, $\mathbf{x}^\star$, is given by:

$$\langle \tilde{\mathbf{g}}_t, \tilde{\mathbf{x}}_t - \mathbf{x}^\star \rangle = \frac{1}{2}\eta_t ||\tilde{\mathbf{g}}_t||^2 + \frac{D(\mathbf{x}^\star||\mathbf{x}_t) - D(\mathbf{x}^\star||\mathbf{x}_{t+1})}{\eta_t} + \left[\sum_{i \in \mathcal{A}_t} \eta_i \langle \tilde{\mathbf{g}}_i, \tilde{\mathbf{g}}_t \rangle - \sum_{i \in \mathcal{B}} \eta_i \langle \tilde{\mathbf{g}}_i, \tilde{\mathbf{g}}_t \rangle \right].$$

The first term incorporates the updates from the current iteration, the second is the distance between two successive states at t and $t+1$ compared to the optimal $\mathbf{x}^\star$, and the final term incorporates the extra updates and missed updates at the current step.

□

By application of Lemma 1:

$$\begin{aligned} R[X] \leq \sum_{t=1}^{T} \langle \tilde{\mathbf{g}}_t, \tilde{\mathbf{x}}_t - \mathbf{x}^\star \rangle &= \sum_{t=1}^{T} \left[\frac{1}{2}\eta_t ||\tilde{\mathbf{g}}_t||^2 + \frac{D(\mathbf{x}^\star||\mathbf{x}_t) - D(\mathbf{x}^\star||\mathbf{x}_{t+1})}{\eta_t} \right] \\ &\quad + \sum_{t=1}^{T} \left[\sum_{i \in \mathcal{A}_t} \eta_i \langle \tilde{\mathbf{g}}_i, \tilde{\mathbf{g}}_t \rangle - \sum_{i \in \mathcal{B}} \eta_i \langle \tilde{\mathbf{g}}_i, \tilde{\mathbf{g}}_t \rangle \right] \\ &= \sum_{t=1}^{T} \left[\frac{1}{2}\eta_t ||\tilde{\mathbf{g}}_t||^2 \right] \\ &\quad + \left[\frac{D(\mathbf{x}^\star||\mathbf{x}_1)}{\eta_1} - \frac{D(\mathbf{x}^\star||\mathbf{x}_{T+1})}{\eta_T} + \sum_{t=2}^{T} D(\mathbf{x}^\star||\mathbf{x}_t) \left(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}} \right) \right] \\ &\quad + \sum_{t=1}^{T} \left[\sum_{i \in \mathcal{A}_t} \eta_i \langle \tilde{\mathbf{g}}_i, \tilde{\mathbf{g}}_t \rangle - \sum_{i \in \mathcal{B}_t} \eta_i \langle \tilde{\mathbf{g}}_i, \tilde{\mathbf{g}}_t \rangle \right]. \end{aligned}$$

Deterministic bounds I now find bounds for each term of $R[X]$.

Starting with the first term in the regret:

$$\begin{aligned}\sum_{t=1}^{T}\frac{1}{2}\eta_t||\tilde{\mathbf{g}}_t||^2 &\leq \sum_{t=1}^{T}\frac{1}{2}\eta_t L^2 \text{ (by Lipschitz continuity)}\\ &\leq \sum_{t=1}^{T}\frac{1}{2}\frac{\sigma}{\sqrt{t}}L^2\\ &\leq \sigma L^2\sqrt{T}\,.\end{aligned}$$

The second term:

$$\begin{aligned}&\frac{D(\mathbf{x}^\star||\mathbf{x}_1)}{\eta_1}-\frac{D(\mathbf{x}^\star||\mathbf{x}_{T+1})}{\eta_t}+\sum_{t=2}^{T}D(\mathbf{x}^\star||\mathbf{x}_t)\left(\frac{1}{\eta_t}-\frac{1}{\eta_{t-1}}\right)\\ &\leq \frac{F^2}{\sigma}+\frac{F^2\sqrt{T}}{\sigma}+\frac{F^2}{\sigma}\sum_{t=2}^{T}\left(\sqrt{t}-\sqrt{t-1}\right)\\ &\leq \frac{F^2}{\sigma}+\frac{F^2\sqrt{T}}{\sigma}+\frac{F^2}{\sigma}(\sqrt{T}-1)\\ &\leq \frac{2F^2}{\sigma}\sqrt{T}\,.\end{aligned}$$

Now for the final term. I first provide a general form which is applicable to both ASP and PSP. The final bounds require assumptions over the distribution of lags. Let $\bar{u}_t = \frac{1}{|\mathcal{A}||\mathcal{B}|}\sum_{i\in\mathcal{A}\cup\mathcal{B}}||\tilde{\mathbf{u}}_t||_2$ be the average l^2-norm of the updates. The update equation is now (see Ho et al. [5]):

$$\tilde{\mathbf{x}}_t = \mathbf{x}_t + \bar{u}_t\gamma_t\,. \tag{3.12}$$

With this and $\bar{u}_t$, we can simplify the notation as follows:

$$\begin{aligned}\sum_{t=1}^{T}\left[\sum_{i\in\mathcal{A}_t}\eta_i\langle\tilde{\mathbf{g}}_i,\tilde{\mathbf{g}}_t\rangle-\sum_{i\in\mathcal{B}_t}\eta_i\langle\tilde{\mathbf{g}}_i,\tilde{\mathbf{g}}_t\rangle\right] &\leq \sum_{t=1}^{T}\eta_t\langle\bar{u}_t\gamma_t,\tilde{\mathbf{g}}_t\rangle\\ &\leq \sum_{t=1}^{T}\eta_t\bar{u}_t||\gamma_t||_2||\tilde{\mathbf{g}}_t||_2\,.\end{aligned}$$

This final term has many different components which require bounds. I now prove several lemmas for this purpose.

Lemma 2 The average l^2-norm of the updates is bounded: $\bar{u}_t \leq 4PL\sqrt{t}$. Additionally, $||\tilde{\mathbf{u}}_t||_2 = ||\eta_t\tilde{\mathbf{g}}_t|| \leq \eta_t L$ by Lipschitz continuity. By definition:

$$\begin{aligned}\bar{u}_t &= \frac{1}{|\mathcal{A}||\mathcal{B}|}\sum_{i\in\mathcal{A}\cup\mathcal{B}}||\tilde{\mathbf{u}}_i||_2\\ &= \frac{1}{|\mathcal{A}||\mathcal{B}|}\sum_{i\in\mathcal{A}\cup\mathcal{B}}||\eta_i\tilde{\mathbf{g}}_i||_2\\ &\leq \frac{1}{|\mathcal{A}||\mathcal{B}|}\sum_{i\in\mathcal{A}\cup\mathcal{B}}\eta_i L\,.\end{aligned}$$

The worst case is if all updates are missed. Thus, bounds for the size of the index sets are given by: $1 \leq |\mathcal{A}| \leq Pt$ and $1 \leq |\mathcal{B}| \leq Pt$ as t steps have been performed and P workers are present. Note that the total sizes of the two sets is going to be $\leq Pt$ but I ignore this in the following bound. So,

$$\begin{aligned}\bar{u}_t \leq & \frac{1}{|\mathcal{A}||\mathcal{B}|} 2PL \sum_{j=0}^{t} \frac{\sigma}{\sqrt{j}} \\ \leq & 2PL \sum_{j=0}^{t} \frac{\sigma}{\sqrt{j}} \,.\end{aligned}$$

Using the identity $\sum_{i=a}^{b} \frac{1}{\sqrt{i}} \leq 2\sqrt{b-a-1}$,

$$\begin{aligned}\bar{u}_t \leq & 4PL\sqrt{t-1} \\ \leq & 4PL\sqrt{t} \,.\end{aligned}$$

□

Lemma 3 The L2 norm of γ_t is bounded: $||\gamma_t||_2 \leq TP$.

In the worst case, an update can experience a lag of T and there are P workers who could lag. If all P entries of γ_t are T then $||\gamma_t||_2 \leq TP$.

□

Back to the theorem. Using Lemma 2 on the final term of the regret yields:

$$\begin{aligned}\sum_{t=1}^{T} \eta_t \bar{u}_t ||\gamma_t||_2 ||\tilde{g}_t||_2 \leq & \sum_{t=1}^{T} \frac{\sigma}{\sqrt{t}} 4PL\sqrt{t} L ||\gamma_t||^2 \\ = & 4PL\sigma \sum_{t=1}^{T} \gamma_t \,.\end{aligned}$$

Substituting the new bounds back into $R[X]$ yields:

$$R[X] \leq \sigma L^2 \sqrt{T} + \frac{2F^2}{\sigma}\sqrt{T} + 4P\sigma L \sum_{t=1}^{T} \gamma_t \,.$$

Dividing through by T:

$$\frac{R[X]}{T} - \frac{\sigma L^2}{\sqrt{T}} - \frac{2F^2}{\sigma\sqrt{T}} \leq \frac{4P\sigma L \sum_{t=1}^{T} \gamma_t}{T} \,. \tag{3.13}$$

This represents a bound on the regret. However, the bound from Lemma 3 on γ_t would lead to an upper bound of $4PL\sigma \sum_{t=1}^{T} TP$ which is $O(T^2)$. Clearly, this leads to $R[X] = O(T^2)$ which means we cannot guarantee convergence. However, if stronger assumptions are made on the distribution of γ_t, probabilistic bounds can be found.

Probabilistic Bound To bound the regret in a more conservative fashion than Lemma 3, it is necessary to consider a probabilistic bound. Denote Z_t as a random variable where each Z_t is bounded, $Z_t \leq b \leq 4PL\sigma T$, and independent. A one-sided Bernstein inequality from Pollard [20] is:

$$\mathbb{P}\left(\sum_{t=1}^{T}(Z_t - \mathbb{E}(Z_t)) \geq T\delta\right) \leq \exp\left(-\frac{T\delta^2}{\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}(Z_t^2) + \frac{b\delta}{3}}\right),$$

where δ is a constant. This can be used to bound the differences between the mean of the distribution and the random variables which will enable the derivation of a bound on the regret.

Letting $Z_t = 4P\sigma L\gamma_t$ yields:

$$\mathbb{P}\left(\frac{\sum_{t=1}^{T}4P\sigma L\gamma_t - \sum_{t=1}^{T}\mathbb{E}(4P\sigma L\gamma_t)}{T} \geq \delta\right) \leq \exp\left(-\frac{T\delta^2}{\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}((4P\sigma L\gamma_t)^2) + \frac{b\delta}{3}}\right).$$

Using 3.13, the regret can be introduced into the expression:

$$\mathbb{P}\left(\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\sigma L^2 - \frac{2F^2}{\sigma}\right) - \frac{4P\sigma L}{T}\left(\sum_{t=1}^{T}\mathbb{E}(\gamma_t)\right) \geq \delta\right) \leq \exp\left(-\frac{T\delta^2}{\frac{16P^2\sigma^2L^2}{T}\sum_{t=1}^{T}\mathbb{E}(\gamma_t^2) + \frac{b\delta}{3}}\right).$$

Convergence is dependent upon the mean and variance of the lag distribution. However, for distributions where $\sum_{t=1}^{T}\mathbb{E}(\gamma_t) \leq \sqrt{T}$, $\frac{R[X]}{T}$ converges to $O(T^{-1/2})$ with an exponential tail bound. This implies probabilistic convergence. Unfortunately, this limit on the sum of the means is rather tight. Instead, let us assume the same distribution for each γ_t so that $\forall t.\mathbb{E}(\gamma_t) = \mu$ and $\forall t.\mathbb{E}(\gamma_t^2) = \phi$, that is constant mean and variance. We can relax our bound by letting $\delta' = \delta + 4PL\mu\sigma$, which is constant:

$$\mathbb{P}\left(\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\sigma L^2 - \frac{2F^2}{\sigma}\right) - 4P\sigma L\mu \geq \delta\right) \leq \exp\left(-\frac{T\delta^2}{16P^2\sigma^2L^2\phi + \frac{b\delta}{3}}\right)$$

$$\mathbb{P}\left(\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\sigma L^2 - \frac{2F^2}{\sigma}\right) \geq \delta'\right) \leq \exp\left(-\frac{T\delta'^2}{16P^2\sigma^2L^2\phi + \frac{b\delta}{3}}\right).$$

If we assume with probability Φ that $\forall t.4PL\sigma\gamma_t < O(T)$, then $b < O(T)$ so $\frac{R[X]}{T}$ converges to $O(T^{-1/2})$ in probability with an exponential tail bound with probability Φ.

□

3.4 Convergence of SGD under PSP

In PSP, either a central oracle tracks the progress of each worker or the workers each hold their own local view. At the barrier control point, a worker samples β out of P workers without replacement. If a single one of these lags more than s steps behind the current

worker then it waits. This process is pBSP (based on BSP) if $s = 0$ and pSSP (based on SSP) if $s > 0$. However, if $s = \infty$ then PSP reduces to ASP.

PSP improves on ASP by providing probabilistic guarantees about convergence with tighter bounds and less restrictive assumptions. pSSP relaxes SSP's inflexible staleness parameter by permitting some workers to fall further behind. This works because machine learning algorithms can typically tolerate the additional noise [4]. pBSP relaxes BSP by allowing some workers to lag slightly, yielding a BSP-like method which is more resistant to stragglers but no longer deterministic.

I now formalise the PSP method.

Theorem 2: PSP sampling

Take a set of P workers whose probabilities of lagging r steps are drawn from a distribution with probability mass function $f(r)$ and cumulative distribution function (CDF) $F(r) = \mathbb{P}(x \leq r) = \sum_{x=0}^{r} f(r)$. Sample without replacement β workers where $\beta \in \mathbb{Z}^+$ and $\beta \leq P$. Impose the sampling constraint that if a single one of the β workers has lag greater than r then the sampler must wait. This leads to the following distribution for lags:

$$p(s) = \left\{ \begin{array}{ll} \alpha f(s) & \text{for } s \leq r \\ \alpha (F(r)^{\beta})^{s-r} & \text{for } s > r \end{array} \right\},$$

where α is some normalizing constant.

Proof

Let n be one of the sampled workers. For $s \leq r$, the probability that a worker will lag s steps depends entirely upon its lag distribution as the sampling primitive only considers workers with $s > r$.

For $s > r$, a worker waits if at least one of the β workers lags more than r steps and otherwise it proceeds. The probability that a worker's step count increases by one beyond r is thus given by:

$$\mathbb{P}(\forall n.\text{lag}(n) \leq r) = F(r)^{\beta}\,.$$

This assumes that the probability of a worker lagging is independent of the state of the other workers. In order for a worker to lag $s - r$ steps, where $s > r$, it has to have been missed $s - r$ times by the sampling primitive. Assuming each sampling event is independent yields:

$$\forall n.\forall s > r.\mathbb{P}(\text{lag}(n) = s) = (F(r)^{\beta})^{s-r}\,.$$

Now, to make $p(s)$ a valid probability distribution, it needs to be normalised over $[0, T]$ inclusive:

$$\alpha \sum_{s=0}^{\infty} p(s) = 1\,. \tag{3.14}$$

Evaluating the sum:

$$\sum_{s=0}^{T} p(s) = \sum_{s=0}^{r} f(s) + \sum_{s=r+1}^{T} (F(r)^{\beta})^{s-r} \tag{3.15}$$

$$= \sum_{s=0}^{r} f(s) + \sum_{s=1}^{T-r} (F(r)^{\beta})^{s} \,. \tag{3.16}$$

If $F(r)^{\beta} < 1$ then we have a geometric series:

$$\sum_{s=0}^{T} p(s) = \sum_{s=0}^{r} f(s) + \frac{(1 - (F(r)^{\beta})^{T-r+1})}{1 - F(r)^{\beta}} - 1 \,. \tag{3.17}$$

Otherwise, if $F(r)^{\beta} = 1$:

$$\sum_{s=0}^{T} p(s) = \sum_{s=0}^{r} f(s) + T - r \,. \tag{3.18}$$

A substitution and rearrangement yields α in 3.14.

To bound α, examine the sum in 3.16. For $T > r + 1$,

$$\sum_{s=0}^{T} p(s) \leq F(r) + F(r)^{\beta} \,, \tag{3.19}$$

because each term of the geometric sum is positive. Thus, by 3.14:

$$\alpha \geq \frac{1}{F(r) + F(r)^{\beta}} \,. \tag{3.20}$$

If $F(r)^{\beta} = 1$ then,

$$\alpha \leq \frac{1}{T - r} \,. \tag{3.21}$$

□

Theorem 3: SGD under PSP

Let $f(\mathbf{x}) = \sum_{t=1}^{T} f_t(\mathbf{x})$ be a convex function where each $f_t \in \mathbb{R}$ is also convex. Let $\mathbf{x}^{\star} \in \mathbb{R}^d$ be the minimizer of this function. Assume that f_t are L-Lipschitz and that the distance between two points $\mathbf{x}$ and $\mathbf{x}'$ is bounded: $D(\mathbf{x}||\mathbf{x}') = \frac{1}{2}||\mathbf{x} - \mathbf{x}'||_2^2 \leq F^2$, where F is constant.

Let an update be given by $\mathbf{u}_t = -\eta_t \nabla f_t(\tilde{\mathbf{x}}_t)$ and the learning rate by $\eta_t = \frac{\sigma}{\sqrt{t}}$.

Represent the lag of updates due to network overheads and the different execution speeds of the P workers by a vector, $\gamma_t \in \mathbb{R}^d$, which consists of random variables, Y_i. These Y_i are i.i.d and are independent of $\mathbf{u}_t$ and $\tilde{\mathbf{x}}$.

Following the presentation of regret by Ho et al. [5], let $R[X] = \sum_t^T f_t(\tilde{\mathbf{x}}_t) - f_t(\mathbf{x}^{\star})$. This is the sum of the differences between the optimal value of the function and the current value given a noisy state. A probabilistic bound on the regret allows us to determine if

the noisy system state, $\tilde{\mathbf{x}}_t$, converges towards the optimal, $\mathbf{x}^\star$, in probability. One such bound on the regret is given by:

$$\mathbb{P}\left(\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\sigma L^2 - \frac{2F^2}{\sigma}\right) - q \geq \delta\right) \leq \exp\left(-\frac{T\delta^2}{c + \frac{b\delta}{3}}\right), \tag{3.22}$$

where δ is a constant and $b \leq 4PTL\sigma$. The b term here is the upper bound on the random variables which are drawn from the lag distribution.

Let $a = F(r)^\beta = (\sum_{s=0}^{r} p(s))^\beta$. If we assume that $0 < a < 1$, then:

$$q \leq \frac{4P\sigma L(1-a)}{F(r)(1-a) + a - a^{T-r+1})}\left(\frac{r(r+1)}{2} + \frac{a(r+2)}{(1-a)^2}\right). \tag{3.23}$$

Again, assuming that $0 < a < 1$, then the value of c is bounded by the following expression:

$$c \leq \frac{16P^2\sigma^2L^2(1-a)}{F(r)(1-a) + a - a^{T-r+1}}\left(\frac{r(r+1)(2r+1)}{6} + \frac{a(r^2+4)}{(1-a)^3}\right). \tag{3.24}$$

If we further assume with probability Φ that $\forall t.4PL\sigma\gamma_t < O(T)$, then $b < O(T)$ so $\frac{R[X]}{T}$ converges to $O(T^{-1/2})$ in probability with an exponential tail bound with probability Φ.

Proof

The initial parts of my convergence proof for ASP provide the starting point for my analysis of PSP. I begin with equation 3.13 in Theorem 1 and construct one-sided probabilistic Bernstein bounds on the regret. Next, I consider how PSP impacts this bound by examining how it bounds the average of the means and then the average of the variances.

Probabilistic Bound Following the application of Lemma 2 in Theorem 1, we have:

$$\frac{R[X]}{T} - \frac{\sigma L^2}{\sqrt{T}} - \frac{2F^2}{\sigma\sqrt{T}} \leq \frac{4P\sigma L\sum_{t=1}^{T}\gamma_t}{T}. \tag{3.25}$$

Assume $Z_t \leq b \leq 4PTL\sigma$ and that each Z_t is independent. Then the following one-sided Bernstein inequality from Pollard [20] can be used:

$$\mathbb{P}\left(\sum_{t=0}^{T}(Z_t - \mathbb{E}(Z_t)) \geq T\delta\right) \leq \exp\left(-\frac{T\delta^2}{\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(Z_t^2) + \frac{b\delta}{3}}\right). \tag{3.26}$$

Using the bounds in Lemma 3, let $Z_t = 4P\sigma L\gamma_t$ so,

$$\mathbb{P}\left(\frac{\sum_{t=0}^{T}4P\sigma L\gamma_t - \sum_{t=0}^{T}\mathbb{E}(4P\sigma L\gamma_t)}{T} \geq \delta\right) \leq \exp\left(-\frac{T\delta^2}{\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}((4P\sigma L\gamma_t)^2) + \frac{b\delta}{3}}\right).$$

Substituting in equation 3.13,

$$\mathbb{P}\left(\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\sigma L^2 - \frac{2F^2}{\sigma}\right) - \frac{4P\sigma L}{T}\left(\sum_{t=1}^{T}\mathbb{E}(\gamma_t)\right) \geq \delta\right) \leq \exp\left(-\frac{T\delta^2}{\frac{16P^2\sigma^2L^2}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) + \frac{b\delta}{3}}\right).$$

Clearly the probability is dependent upon the mean, $\mathbb{E}(\gamma_t)$, and variance, $\mathbb{E}(\gamma_t^2)$, of the lag distribution. Currently, we need $\sum_{t=1}^{T}\mathbb{E}(\gamma_t) \leq \sqrt{T}$. However, this limit on the sum of the means is unfortunately tight. In ASP, an assumption that the mean and variance were constant yielded a bound but we had to increase our constant, δ, by adding $4PL\mu\sigma$ which could make the constant far too large to be practical. For example, if μ is T. The aim is to see how using the PSP sampling primitive impacts this bound. I focus first on the term with the means and then the term with the variances.

Bounding the average of the means First the $\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}(\gamma_t)$ term:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) = \frac{1}{T}\sum_{t=0}^{T}\sum_{s=1}^{t}p(s)s\,. \tag{3.27}$$

By the definition of PSP in Theorem 2,

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) = \frac{\alpha}{T}\left(\sum_{t=0}^{r}\left(\sum_{s=0}^{t}f(s)s\right) + \sum_{t=r+1}^{T}\left(\sum_{s=0}^{r}f(s)s + \sum_{s=r+1}^{t}(F(r)^{\beta})^{s-r}s\right)\right)\,. \tag{3.28}$$

Letting $a = F(r)^{\beta} = \left(\sum_{i=1}^{r}p(i)\right)^{\beta}$ where $0 \leq a \leq 1$, bounding summations, and performing some rearrangements:

$$\begin{aligned}
\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq& \frac{\alpha}{T}\left(r\sum_{s=0}^{r}f(s)s + \sum_{t=r+1}^{T}\left(\sum_{s=0}^{r}f(s)s + \sum_{s=r+1}^{t}a^{s-r}s\right)\right) && (3.29)\\
\leq& \frac{\alpha}{T}\left(T\sum_{s=0}^{r}f(s)s + \sum_{t=r+1}^{T}\left(\sum_{s=r+1}^{t}a^{s-r}s\right)\right) && (3.30)\\
\leq& \alpha\sum_{s=0}^{r}f(s)s + \frac{\alpha}{T}\sum_{t=r+1}^{T}\left(\sum_{s=r+1}^{t}a^{s-r}s\right)\,. && (3.31)
\end{aligned}$$

The indexing of the inner sum over $[r+1, t]$ can be rewritten so that a closed-form solution can be used:

$$\leq \alpha\sum_{s=0}^{r}f(s)s + \frac{\alpha}{T}\sum_{t=r+1}^{T}\left(a\sum_{s=1}^{t-r}a^{s-1}(s+r)\right)\,. \tag{3.32}$$

Specifically, the inner summation on $[1, t-r]$ is over an arithmetico-geometric series.

Arithmetico-geometric series An arithmetico-geometric series takes the following form:

$$S_n = \sum_{k=1}^{n}(a + (k-1)d)r^{k-1}\,. \tag{3.33}$$

There exists a closed-form solution to this partial sum. See D. Khattar's [21] book for the proof.

Provided $r \neq 1$ then,

$$S_n = \frac{a - (a + (n-1)d)r^n}{1-r} + \frac{dr(1-r^{n-1})}{(1-r)^2}\,. \tag{3.34}$$

If $r = 1$ then,

$$S_n = \frac{n}{2}(2a + (n-1)d)\,. \tag{3.35}$$

□

Back to the proof. Make the substitutions $a = r + 1$, $d = 1$, $n = t - r$ and $r = a$ in equation 3.33.

For the first case in the arithmetico-geometric solution we have $r \neq 1$. Thus, $a < 1$ and $(1 - a^{t-r-1}) \leq 1$. Substituting this result into the bound yields:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq \alpha\sum_{s=0}^{r} f(s)s + \frac{\alpha a}{T}\sum_{t=r+1}^{T}\left(\frac{r+1-ta^{t-r}}{1-a} + \frac{a(1-a^{t-r-1})}{(1-a)^2}\right) \tag{3.36}$$

$$\leq \alpha\sum_{s=0}^{r} f(s)s + \frac{\alpha a}{T}\left(\frac{a(T-r)}{(1-a)^2} + \frac{(T-r)(r+1)}{1-a} - \sum_{t=r+1}^{T}\left(\frac{ta^{t-r}}{1-a}\right)\right)\,. \tag{3.37}$$

Examining $\sum_{t=r+1}^{T}\left(\frac{ta^{t-r}}{1-a}\right)$. For $T > r + 1$, this summation is at least $\frac{Ta^{T-r}}{1-a}$ as each term in the summation is positive. Removing some negative terms and using the bound on the sum:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq \alpha\sum_{s=0}^{r} f(s)s + \frac{\alpha a}{T}\left(\frac{T(a + (1-a)(r+1) - (1-a)a^{T-r})}{(1-a)^2}\right) \tag{3.38}$$

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq \alpha\sum_{s=0}^{r} f(s)s + \frac{\alpha a}{(1-a)^2}\left(a + (1-a)(r+1) - (1-a)a^{T-r}\right) \tag{3.39}$$

$$\leq \alpha\sum_{s=0}^{r} f(s)s + \frac{\alpha a}{(1-a)^2}\left(r+2\right)\,. \tag{3.40}$$

By Theorem 2, α is defined as:

$$\alpha = \frac{1 - F(r)^{\beta}}{F(r)(1 - F(r)^{\beta}) + (1 - (F(r)^{\beta})^{T-r+1}) - (1 - F(r)^{\beta})} \tag{3.41}$$

$$= \frac{1-a}{F(r)(1-a) + a - a^{T-r+1}}\,. \tag{3.42}$$

The first term in 3.40 can be bounded as $f(s) \leq 1$. Specifically, letting $\forall s.f(s) = 1$ yields an arithmetic series. Taking the partial sum and substituting in α:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq \frac{1-a}{F(r)(1-a) + a - a^{T-r+1}}\left(\frac{r(r+1)}{2} + \frac{a(r+2)}{(1-a)^2}\right)\,. \tag{3.43}$$

This is a bound on the average of the means of the lags which relies on the sampling count, β (in the a term), the staleness parameter, r, and the length of the update sequence, T.

Considering now the second case of the arithmetico-geometric series closed-form solution. Specifically with $r = 1$ so that $a = 1$. Using 3.32 we arrive at the following:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq \alpha\frac{r(r+1)}{2} + \frac{\alpha}{T}\sum_{t=r+1}^{T}\left(\frac{t-r}{2}(2(r+1)+(t-r-1))\right) \tag{3.44}$$

$$\leq \alpha\frac{r(r+1)}{2} + \frac{\alpha}{T}\left(\frac{-1}{2}(T-r)(r^2+r) + \frac{1}{2}\sum_{t=r+1}^{T}t^2 + \frac{1}{2}\sum_{t=r+1}^{T}t\right). \tag{3.45}$$

Assuming $T > r$, removing some negative terms and substituting in solutions to the partial sums over squared arithmetic and arithmetic series yields:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq \alpha\frac{r(r+1)}{2} + \frac{\alpha}{T}\left(\frac{T(T+1)(2T+1)}{12}\right. \tag{3.46}$$

$$\left. + \frac{T(T+1)}{4} - \frac{r(r+1)(2r+1)}{12} - \frac{r(r+1)}{4}\right). \tag{3.47}$$

By Theorem 2, as $a = 1$, $\alpha \leq \frac{1}{T-r}$ so,

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) < \frac{1}{T-r}\left(\frac{r(r+1)}{4} + \frac{(T+1)(2T+1)}{12} + \frac{T+1}{4}\right) \tag{3.48}$$

$$< \frac{1}{T-r}\left(\frac{r(r+1)}{2} + T^2 + T + Tr + r\right). \tag{3.49}$$

This is $O(T)$, indicating that when $a = (\sum_{s=0}^{r}p(s))^{\beta} = 1$, PSP does not converge in probability. Thus, at least one worker needs to be sampled and some probability mass needs to be present in the first r steps.

Bounding the average of the variances Now for the $\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2)$ term. This is the same process used to bound the average of the means but with s^2 rather than s as the variable:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) \leq \alpha\sum_{s=0}^{r}f(s)s^2 + \frac{\alpha}{T}\sum_{t=r+1}^{T}\left(a\sum_{s=1}^{t-r}a^{s-1}(s+r)^2\right). \tag{3.50}$$

The inner summation on $[1, t-r]$ is no longer over an arithmetico-geometric series but a squared arithmetico-geometric series.

Squared Arithmetico-geometric series See A.14 in the Appendix for my proof of the closed-form solution of the partial sum. This is given by:

$$S_n = \begin{cases} \left[\frac{a^2 - 2ad - d^2 - (a+(n-1)d)^2r^n}{1-r} + \frac{1-r^{n-1}}{(1-r)^2}\right. & \\ \left. + \frac{d^2}{1-r}\left(\frac{1-(1+2(n-2))r^{n-1}}{1-r} + \frac{2r(1-r^{n-2})}{(1-r)^2}\right)\right], & \text{for } |r| < 1 \\ na^2 + \frac{2dn(n-1)}{2} + \frac{d^2n(n-1)(2n-1)}{6} & \text{for } r = 1 \end{cases} \tag{3.51}$$

For the case of $|r| < 1$ in equation 3.51, we have $a < 1$. Make the substitutions: $a = r + 1$, $d = 1$, $n = t - r$ and $r = a$. Substituting this into 3.50 yields:

$$\begin{aligned}
\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) \leq & \alpha\sum_{s=0}^{r} f(s)s^2 + \frac{\alpha a}{T}\sum_{t=r+1}^{T}\Bigg[\frac{(r+1)^2 - 2(r+1) - 1 - (r+1+(t-r-1))^2 a^{t-r}}{1-a} \\
& + \frac{1-a^{t-r-1}}{(1-a)^2} \\
& + \frac{1}{1-a}\left(\frac{1-(1+2(t-r-2))a^{t-r-1}}{1-a} + \frac{2a(1-a^{t-r-2})}{(1-a)^2}\right)\Bigg] \\
\leq & \alpha\sum_{s=0}^{r} f(s)s^2 + \frac{\alpha a}{T}\sum_{t=r+1}^{T}\Bigg[\frac{r^2 - 2 - t^2 a^{t-r}}{1-a} + \frac{1-a^{t-r-1}}{(1-a)^2} \\
& + \frac{1}{1-a}\left(\frac{1-(2t-2r-3)a^{t-r-1}}{1-a} + \frac{2a(1-a^{t-r-2})}{(1-a)^2}\right)\Bigg] \\
\leq & \alpha\sum_{s=0}^{r} f(s)s^2 + \frac{\alpha a}{T(1-a)^3}\sum_{t=r+1}^{T}\Bigg[(1-a)^2(r^2 - 2 - t^2 a^{t-r}) + (1-a)(1-a^{t-r-2}) \\
& + (1-a)(1-(2t-2r-3)a^{t-r-1}) + 2a(1-a^{t-r-2})\Bigg].
\end{aligned}$$

Assume $T > r + 2$ and use $a < 1$ to bound some terms:

$$\begin{aligned}
\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) \leq & \alpha\sum_{s=0}^{r} f(s)s^2 + \frac{\alpha}{T}\sum_{t=r+1}^{T}\left(\frac{a}{(1-a)^3}\Big[(1-a)^2 r^2 + (1-a) + (1-a) + 2a\Big]\right) \\
\leq & \alpha\sum_{s=0}^{r} f(s)s^2 + \frac{\alpha a(T-r)}{T(1-a)^3}\Big[r^2 + 4\Big].
\end{aligned}$$

As $a < 1$, $\forall s.f(s) \leq 1$ so the first term can be bounded by setting $f(s) = 1$, yielding a squared arithmetic series. Take the partial sum:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) < \alpha\frac{r(r+1)(2r+1)}{6} + \frac{\alpha a(r^2+4)}{(1-a)^3}. \tag{3.52}$$

Substituting in α from Theorem 2 yields a bound on the average of the variances of the lag distribution given by:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) < \frac{1-a}{F(r)(1-a) + a - a^{T-r+1}}\left(\frac{r(r+1)(2r+1)}{6} + \frac{a(r^2+4)}{(1-a)^3}\right). \tag{3.53}$$

Consider now the case in 3.51 where $r = 1$, meaning that $a = 1$. Make the substitutions: $a = r + 1$, $d = 1$, $n = t - r$ and $r = a$. This achieves:

$$\begin{aligned}
\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) \leq & \alpha\sum_{s=0}^{r} f(s)s^2 \\
& + \frac{\alpha a}{T}\sum_{t=r+1}^{T}\left[na^2 + \frac{2dn(n-1)}{2} + \frac{d^2n(n-1)(2n-1)}{6}\right] \\
\leq & \alpha\sum_{s=0}^{r} f(s)s^2 \\
& + \frac{\alpha a}{T}\sum_{t=r+1}^{T}(t-r)\left[(r+1)^2 + \frac{2(t-r-1)}{2} + \frac{(t-r-1)(2(t-r)-1)}{6}\right] \\
< & \alpha\sum_{s=0}^{r} f(s)s^2 + \frac{\alpha a}{T}\sum_{t=r+1}^{T}\left[t(r+1)^2 + t^2 + \frac{t^3}{3}\right] \\
< & \alpha\sum_{s=0}^{r} f(s)s^2 + \alpha a\left[\frac{(T+1)(r+1)^2}{2} + \frac{(T+1)(2T+1)}{6} + \frac{T(T+1)^2}{12}\right].
\end{aligned}$$

Using α from Theorem 2, when $a = 1$, $\alpha \leq \frac{1}{T-r}$, which provides a bound on the average of the variances of the lag distribution given by:

$$\begin{aligned}
\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) < & \frac{1}{T-r}\Bigg(\frac{r(r+1)(2r+1)}{6} \\
& + a\left[\frac{(T+1)(r+1)^2}{2} + \frac{(T+1)(2T+1)}{6} + \frac{T(T+1)^2}{12}\right]\Bigg).
\end{aligned}$$

This is $O(T^2)$ which indicates that if $a = 1$, then SGD under PSP will not converge in probability. This is expected as $a = 1$ if $\beta = 0$.

□

3.5 Discussion of PSP Bounds

To conclude, I have shown that using PSP with SGD yields a probabilistic bound on the regret, implying convergence in probability. This happens in two scenarios. The first is when the average of the means and the average of the variances are constant, or bounded by a constant, causing the constant term in the probabilistic bound to be increased. The second case is that both sums are $< O(\sqrt{T})$.

I derived bounds for both the average of the means and the average of the variances. These can be treated as constants for fixed a, T, r and β, meeting the first case for convergence in probability.

More specifically, the average of the means is bounded by:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t) \leq \frac{1-a}{F(r)(1-a)+a-a^{T-r+1})}\left(\frac{r(r+1)}{2} + \frac{a(r+2)}{(1-a)^2}\right). \tag{3.54}$$

The average of the variances has a similar bound:

$$\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\gamma_t^2) < \frac{1-a}{F(r)(1-a)+a-a^{T-r+1}}\left(\frac{r(r+1)(2r+1)}{6}+\frac{a(r^2+4)}{(1-a)^3}\right). \tag{3.55}$$

These bounds rely on the sampling count, β (in a), the staleness parameter, r, the length of the update sequence, T, and the probability mass in the first r steps of the lag distribution. They provide a means to quantify the impact of the PSP sampling primitive and provide stronger convergence guarantees than ASP. Specifically, they do not depend upon the entire lag distribution, which ASP does.

The bounds demonstrate the impact of the sampling primitive. Figures 3.1 and 3.2 show how increasing the sampling count, β, yields tighter bounds. Notably, only a small number of nodes need to be sampled to yield bounds close to the optimal. This is verified by my experiments in section 5.2.

The discontinuities at $a = 0$ and $a = 1$ reflect edge cases of the behaviour of the barrier method control. Specifically, with $a = 0$, no probability mass is in the initial r steps so no progress can be achieved if the system requires $\beta > 0$ workers to be within r steps of the fastest worker. If $a = 1$ and $\beta = 0$, then the system is operating in ASP mode so the bounds are expected to be large. However, these are overly generous. Better bounds are $O(T)$ for the mean and $O(T^2)$ for the variance, which I give in my proof. When $a = 1$ and $\beta \neq 0$, the system should never wait and workers could slip as far as they like as long as they returned to be within r steps before the next sampling point.

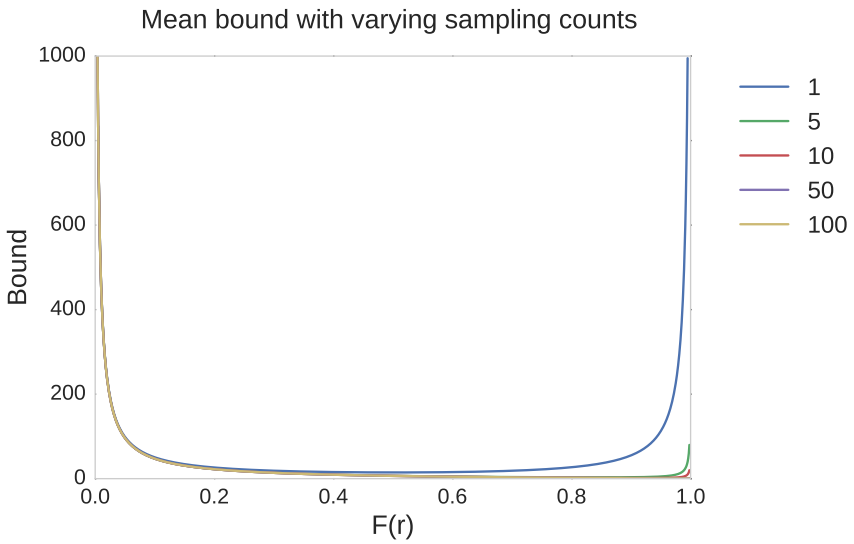


Figure 3.1: Plot showing the bound on the average of the means of the sampling distribution. The sampling count is varied and the staleness, r, is set to 4 with T equal to 10000.

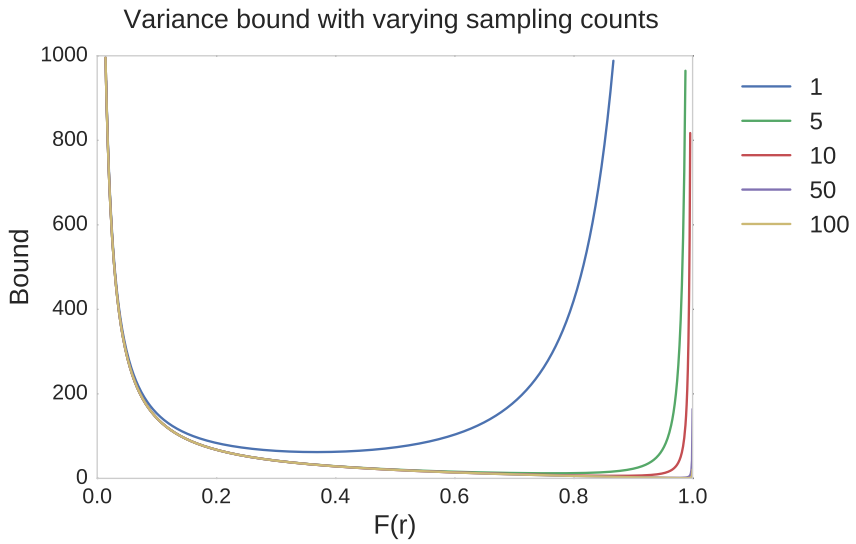


Figure 3.2: Plot showing the bound on the average of the variances of the sampling distribution. The sampling count is varied and the staleness, r, is set to 4 with T equal to 10000.

3.6 Discussion of PSP vs. ASP

Both PSP and ASP are able to achieve convergence in probability as long as the δ term in the bounds is relaxed. Provided the algorithms are executed for at least $r + 2$ steps ($T > r+1$) and $T > \beta$, then PSP bounds the relaxation of the δ term independently of the mean of the lag distribution. Furthermore, it only relies on the distribution where the lag is less than r. In ASP, the δ relaxation relies heavily upon the mean of the distribution. Thus, the bound can deteriorate over time if the mean increases and can be far larger than the PSP bound.

PSP also reduces the impact that the lag distribution can have on the system. For example, PSP can mitigate the impact of long and heavy tailed distributions. Furthermore, PSP is able to provide the same bound even if the lag distribution is changing over time. This is an excellent result for real-world computing environments where fluctuating performance characteristics are typical.

3.7 Discussion of PSP vs. SSP

SSP provides deterministic convergence bounds [5] due to the guaranteed pre-window updates in the interval $[0, c - s + 1]$. PSP provides no such guarantee. Instead, the SSP update window is replaced by a lag distribution over updates which yields a probabilistic bound on convergence.

To compare the two methods, I run experiments where I alter the sample size to determine the proportions which achieve SSP-like iteration distributions.

Chapter 4

Implementation

In this chapter I discuss the parameter server implementation I investigated using to evaluate PSP. I detail my implementation of a simulator of this system and present an analysis of its results. I discuss why I subsequently developed a higher fidelity simulator. I also describe the theory, and my implementations, of the Stochastic Gradient Descent and Latent Dirichlet Allocation algorithms. I conclude with a brief overview of the tools I employed.

4.1 Peer-to-Peer Parameter Server

I evaluated using a peer-to-peer parameter server architecture to analyse PSP because of the increasing academic focus on this paradigm [2]. I used an implementation by Dr Liang Wang [22] which follows the general design decisions of Li et al. [9] but is structured as a peer-to-peer system. His system is implemented in OCaml [23], a functional programming language.

To ensure my experiments produce reliable and accurate results, fine-grained control of the computation is required. Network delays need to be configurable to assess their impact. Per-operation computation time needs to be controlled to ensure experiments assess the performance of the system, rather than the operating system and its scheduler.

To satisfy these requirements, I implemented a discrete-event simulation [24] of the parameter server. This process involved breaking up worker's operations into events which required a detailed analysis of the codebase. System initialisation also needed to be carefully controlled as different operation orderings yield different model and data partitions. In this section, I explain these processes and the parameter server's architecture. I also describe Distributed Hash Tables (DHT) [25] which the system uses to resolve parameter ownership. I conclude with a discussion of the ZeroMQ library which the parameter server and simulator use for networking functionality.

4.1.1 General Architecture

The peer-to-peer parameter server consists of a *manager* and a collection of peers a.k.a workers. Each worker runs a server and a client process. This differs from other designs

which typically run disjoint sets of servers and clients [9, 2]. Figure 4.2 shows the general communication architecture.

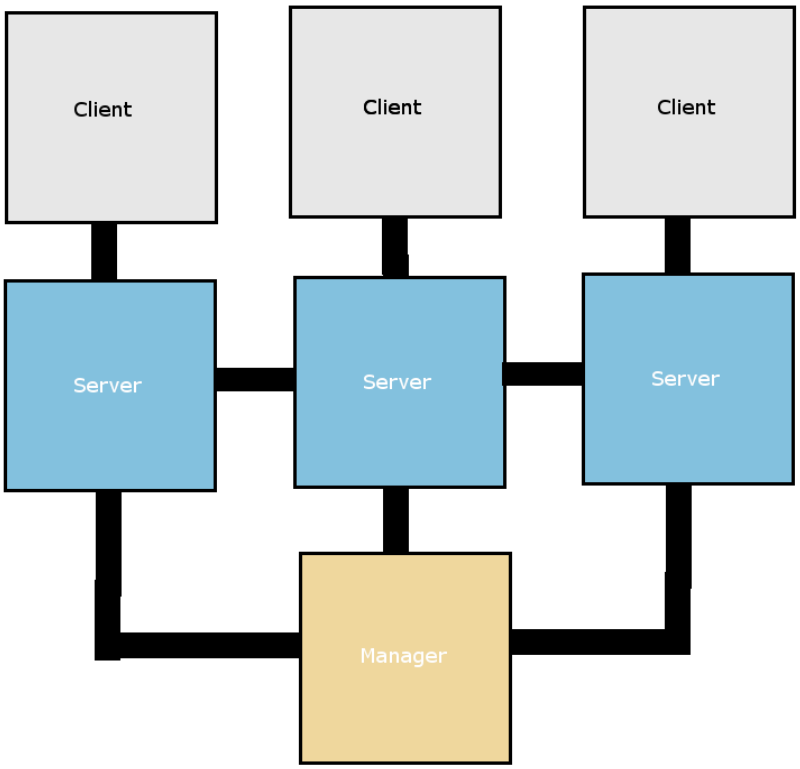


Figure 4.1: Architecture of the peer-to-peer parameter server. Thick black lines indicate connections.

Servers operate as parameter servers whilst clients execute machine learning algorithms. Each parameter server holds a part of the global model which can be queried and updated by the local client, and other servers. The manager tracks which workers are available and their current job assignment.

4.1.2 Operations

In order to perform user-defined computations, the system provides the following operations:

push Performs computations on the local model and data, yielding a set of model updates. The underlying system forwards these to the owning server.

pull Retrieves the latest updates from the paired parameter server, updating the local model.

schedule Selects the next set of work for the user-defined program. For SGD, this is typically the set of parameters to operate on and for LDA it is often a subset of the text corpus.

stop Indicates if the stopping criteria for the user program have been met. For example, if the desired number of iterations have been performed.

barrier Runs a barrier control method to determine if this particular client should continue onto its next iteration or if it should wait.

clean up Executed once the stopping criteria are met. I use this to calculate the loss on the final model.

4.1.3 Initialisation

On start up, the system executes a user-defined function which loads the local state needed for the machine learning algorithm. Next, the user program registers user-defined functions (*schedule, barrier, push, pull, stop* and *clean up*) with the system. The system's own initialisation routine is then called which binds a socket to listen for messages from other workers. If the simulator is enabled, the worker connects to it and, from that point on, the simulator controls all operations.

The worker then connects to the manager, passing it a job id. It receives back a list of all workers registered on the same job. The process then forks with the parent running as a parameter server and the child as a client.

Server Initialisation

The server constructs a Distributed Hash Table (DHT) using the list of peers received from the manager. The server then enters its service loop which will indefinitely listen for, and process, messages.

Client Initialisation

If the simulator is running, the client creates a new connection to it. All operations after this point would then be controlled by the simulator. Next, the client connects to its local server using a **P2P_Connect** message. The client is now ready and enters its service loop which performs the following operations:

```
1 while not stop():
2     schedule ()
3     pull_model ()
4     run_user_program ()
5     push_model ()
6     barrier ()
```

Listing 4.1: Client service loop.

4.1.4 Distributed Hash Table

The core component of the peer-to-peer parameter server implementation is a distributed hash table (DHT) which facilitates fast resolution of parameter ownership.

A seminal paper on Distributed Hash Tables was due to Stoica et al. [25] for the Chord system. Therein, they identified that a core problem for peer-to-peer systems is efficient lookup of data items. As a solution, they presented distributed hash tables using consistent hashing. Consistent hashing was introduced by Karger et al. [26] as a way of distributing requests amongst a changing population of web servers. It has the property that when a hash table is resized, only K/n keys need to be remapped on average. Here, K is the number of keys and n is the number of entries.

Specifically, a consistent hash function assigns each worker an m-bit identifier using a base hash function (such as SHA-512) applied to the worker's IP address [25]. A key's identifier is the hash of the key itself. The identifiers can be viewed as ordered using an *identifier circle*, modulo 2^m, as in Figure 4.2. Key k is assigned to the worker with the nearest identifier in a clockwise direction on the circle. Note that the size of m needs to be sufficiently large that the probability of a clash between two separate keys is negligible.

Using DHTs, workers require minimal routing information as each worker only needs to know its successor. Queries can be passed around the circle until a worker's identifier exceeds the hashed key: this is the worker which owns that data [25]. However, to speed up routing, each worker can hold a routing table, the *finger table*, which holds the IP addresses of certain workers [25]. This table means that lookups become $O(lg(n))$ with high probability[25].

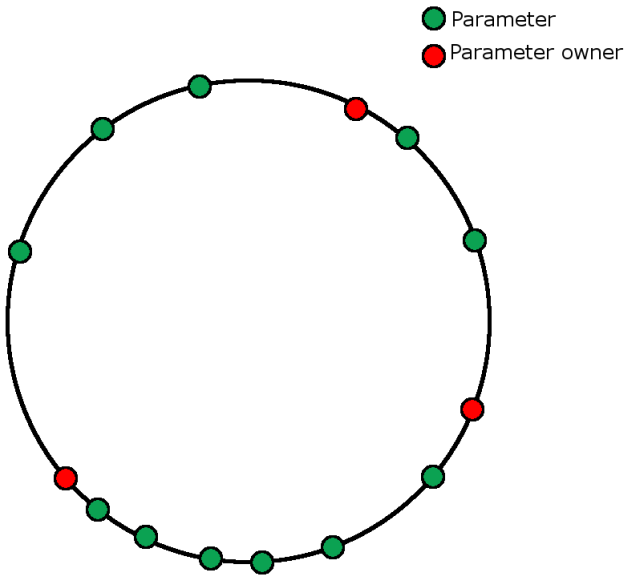


Figure 4.2: Conceptual view of the distribution of keys and workers in a Distributed Hash Table. The parameters are owned by the server proceeding them in a clockwise direction.

4.1.5 ZeroMQ

ZeroMQ [27] (ZMQ) is an efficient, embeddable networking library which provides socket implementations that carry atomic messages across transports such as TCP. ZMQ provides an asynchronous I/O model that aims to solve the difficulties of connecting code in today's

applications by providing simple building blocks which are easy to understand and use [27]. Dr Wang's implementation uses ZMQ to handle its networking operations.

ZeroMQ messages can be sent between workers using a variety of socket configurations. The ones which I make use of are:

- **REQ-REP**: The REQ-REP socket pair operates in lock-step. The REQ socket first sends a request, using zmq_send(), to the REP socket which subsequently sends a response back [27].
- **DEALER**: A DEALER socket can be plugged in anywhere a REQ socket is used. However, a DEALER socket can send multiple messages before receiving a reply [27].
- **ROUTER**: A ROUTER socket can be plugged in anywhere a REP socket is used. They accept and track connections, giving each connection a unique identity [27].

4.2 Peer-to-Peer Parameter Server Simulator

I designed and implemented a discrete-event simulation of Dr Liang Wang's [22] peer-to-peer parameter server using OCaml. This is written as a plug-in component of the actual parameter server which can be enabled at compile time.

A discrete-event simulation models the system's operations as a discrete sequence of events [24]. In my design, workers connect to a central simulator and register events for the simulator to trigger. The simulator uses event queues and an internal clock to determine which events to execute. Each worker is identified by a unique, deterministic IP address and given a random number generator seed. These steps help with ensuring deterministic and repeatable simulations.

To control networking behaviour, all network operations are modelled as events, whose durations are controlled by the simulator. To facilitate this, I introduced a middle layer between the ZeroMQ networking calls and the system which allows the underlying networking code to be altered for simulation without affecting the main codebase.

4.2.1 Initialisation and General Operation

The simulator starts up the designated number of workers with the desired configuration. This include the workers' IP addresses, random number generator seeds, machine learning parameters and barrier control parameters. Each worker initialises its base context and the machine learning algorithm using this information. Next, the worker creates a connection to the simulator and waits to begin executing events. The simulator adds each new worker to a set of available workers.

Once all workers are connected, the simulator starts executing events. At each time step, each worker runs all of its events tagged for execution. To ensure a deterministic simulation, workers are processed in the strict order defined by their IP addresses. Once these events finish, the simulator executes all events queued for the current time step on its internal event queue. This holds events delayed for future execution (e.g. packet transmissions).

After workers finish executing their user-defined computations, the simulator picks the worker with the lowest IP address and releases it to provide a summary of the algorithm's results. This involves fetching the latest model and then evaluating it using a test set. Servers are allowed to continue executing so that they can respond to queries for parameters.

4.2.2 Networking Swap Out

Whenever a process creates a connection to another process, it calls into the networking system. If the simulator is running, the process establishes a new connection with the simulator using a **SIM_Connect** message. This takes the address of the destination process as an argument. The simulator creates a new connection between itself and that endpoint for use by the initiating process (if one does not already exist). All future traffic sent to the destination by this process will now be routed via the simulator, enabling it to enforce networking delays using the event system.

During simulation, all networking calls become blocking and require an ACK from the simulator to unblock. This is necessary to create a deterministic simulation by ensuring that nodes and the simulator receive packets.

4.2.3 Simulator Control Messages

Whilst the simulator is running, the simulator and nodes communicate using control messages. Table 4.1 lists the main ones.

Message	Parameters	Description
SIM_Connect_Manager	src	Create a connection between *src* and the manager
SIM_Connect	src, dst	Create a connection between *src* and *dst* by creating a connection from the simulator to *dst* and then storing this socket in the routing table.
SIM_Await_Release	src	The *src* node informs the simulator it has finished its event for this time step and is ready to run its next event.
SIM_Release	dst	The *dst* node is told to run its next event.
SIM_Waiting_For_Packet	src	Tell the simulator that the *src* node is waiting for a packet.
SIM_Received_Packet	src	Tell the simulator that the *src* node received a packet it was waiting for.
SIM_Do_Summary	dst	The *dst* node is told to perform a summary of the computation.

Table 4.1: Simulator control messages.

4.2.4 ZeroMQ Workarounds

Messages are routed via the simulator which passes them to the correct destination at the specified time. However, as ZeroMQ is asynchronous, a protocol is required to ensure message delivery.

Nodes notify the simulator when their current event is to listen for a message and also acknowledge the receipt of a message. The **SIM_Waiting_For_Packet** and **SIM_Received_Packet** control messages are used for these purposes. The simulator tracks the state of message delivery. Consequently, if the simulator delivers a message and the client does not receive it that time step, the simulator can re-run the node's listening event to ensure the node receives it. The state machine in Figure 4.3 demonstrates the protocol and the following list shows the interpretation of the states:

- **No Messages**: No packets have been sent to a node and it is not listening for a packet.
- **Packet Waiting**: A packet has been sent to a node which it should receive the next time it listens.
- **Node Listening**: A node is ready to receive packets.
- **Deliver**: A node is listening for packets and a packet has been sent to it. The packet needs to be picked up on this time step.

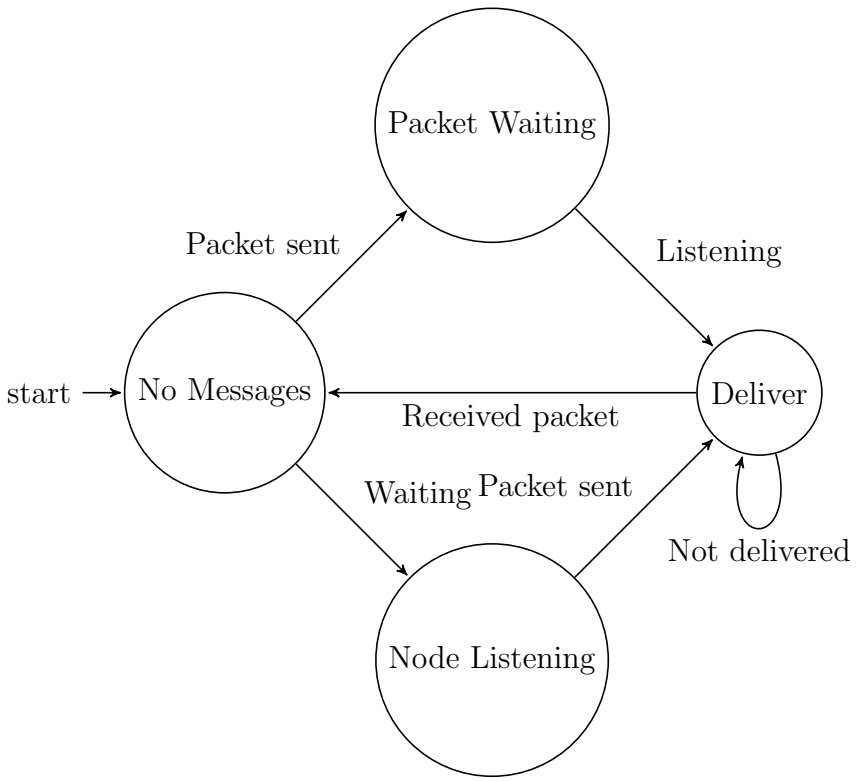


Figure 4.3: State machine showing the packet delivery protocol.

4.2.5 Event Queue

The event queue is implemented as a priority queue where event times are the priorities. Each entry holds a tuple which contains the event duration, event function and meta data used for logging. Listing 4.2 shows the OCaml signature.

```
module type PriorityQueue_type = sig
    type 'a queue    = Empty | Node of float * 'a * 'a queue
    exception Queue_is_empty
    val empty        : 'a queue
    val insert       : 'a queue ref -> float -> 'a -> 'a queue
    val extract_min  : 'a queue ref-> float * 'a * 'a queue
    val _insert      : 'a queue -> float -> 'a -> 'a queue
    val _extract_min : 'a queue -> float * 'a * 'a queue
    val is_empty     : 'a queue -> bool
end
```

Listing 4.2: Event queue signature.

Whilst the simulator is running, the next event is repeatedly fetched from the event queue and its time compared with the current simulation time. If the event needs to be executed, its event function, an anonymous function, will be evaluated. This approach is highly flexible as it enables complete customisation of events. Consider modelling network delays. When node A wishes to send a packet to node B with network delay d, the simulator adds an event to the queue with event time $t + d$. This event will then deliver the message — see Listing 4.3.

```
event_queue := PriorityQueue.insert event_queue !event_time
      (!event_duration, (fun x -> send !sock m.bar m.typ params),
      src, dst,  m.typ, m.par)
```

Listing 4.3: Example of a networking event.

4.2.6 Routing Table

The simulator uses a routing table to route packets between nodes, see listing 4.4. This is a mapping from an ordered two-tuple, (source address, destination address), to a ZeroMQ socket instance which connects the simulator to the destination on behalf of the source.

```
module AddressMap = Map.Make(StringPairs)
let _addr_pair_to_socket_map : [`Req | `Dealer] ZMQ.Socket.t ref
       AddressMap.t ref = ref AddressMap.empty
let find_addr_map id map     = AddressMap.find id !map
let mem_addr_map id map      = AddressMap.mem id !map
let add_addr_map id item map =
    if mem_addr_map id map then
        Logger.error "SIM: Address already mapped (AddrMap)"
    else
        map := AddressMap.add id item !map
```

Listing 4.4: The simulator's routing table.

4.2.7 Simulator Clock

The simulator tracks time using an internal clock. Listing 4.5 shows the module's signature.

```
type clock =
{
    mutable time    : float
}
module type EventClock_type = sig
    val get_time    : clock  -> float
    val set_time    : clock -> float -> unit
    val incr_time   : clock -> float -> unit
end
let sim_clock = empty_simulator_clock ()
```

Listing 4.5: Simulator clock signature.

4.2.8 Logging

The simulator gathers information to enable analysis of system performance. This includes a complete trace of events, a log of all messages and the step of each client at each clock tick. The event trace is particularly useful in ensuring that simulations are completely deterministic as a *diff* between different runs with the same parameters should be empty.

4.2.9 Results

Figure 4.4 shows an example run using my parameter server simulator. At this scale, all of the barrier control methods' curves have overlapped. Figure 4.5 shows the relative differences in loss between each of the methods and the one with the lowest loss at each iteration. The greater the difference, the worse the method has performed. These differences roughly follow the expected pattern. Specifically, ASP performs worst, followed by pSSP and SSP with pBSP often performing the best. BSP exhibits interesting behaviour, starting out as the worst method but, as the computation proceeds, it ends up with the lowest loss.

However, the differences shown in figure 4.5 are in the 5th decimal place. Previous research [5] has shown that they should be far larger. This indicates that the simulator lacks sufficient randomness to accurately model the real world. Furthermore, my experiments revealed that the simulator is prohibitively expensive to run. With 25 workers it can take hours to run LDA for 100 iterations. This is because every operation the full parameter server executes is simulated.

The aim of my research is to evaluate PSP and compare it to other barrier control methods. To do so, I need to be able to adequately explore the parameter space. For this purpose, I implemented a faster, higher fidelity simulator.

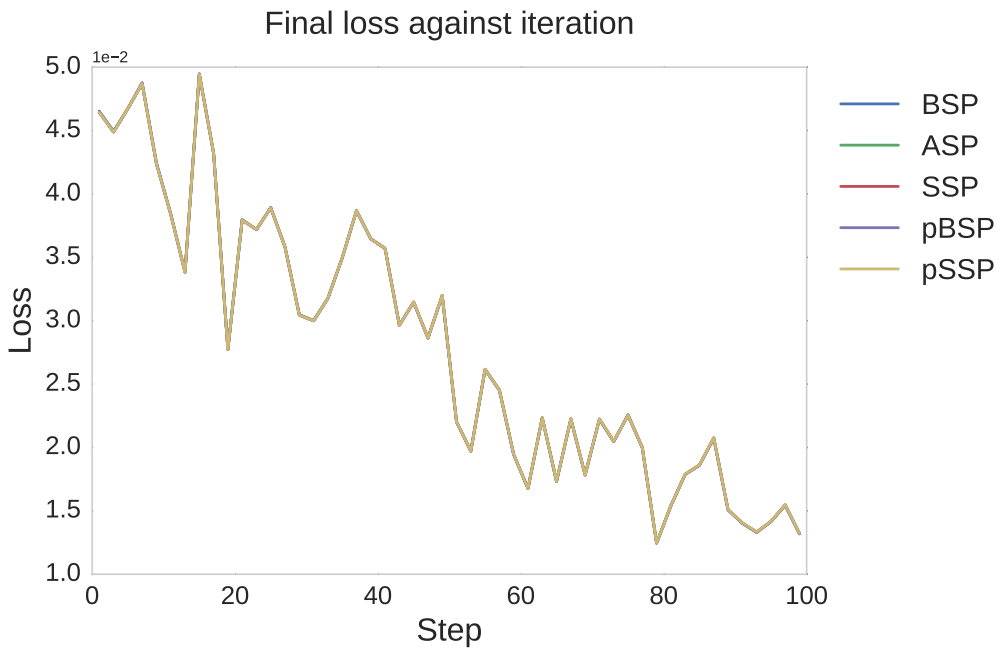


Figure 4.4: Plot showing loss against iteration for different barrier control methods. A total of 15 nodes are present with PSP sampling set to 12 and staleness to 4. Eight nodes have link delays of 50 and the rest have a delay of 1.

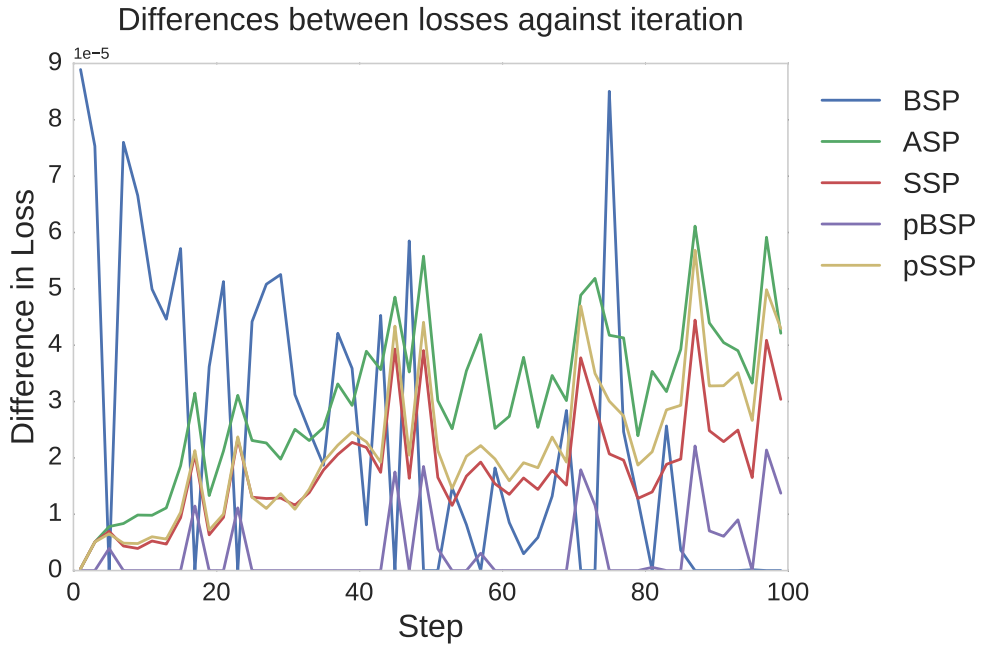


Figure 4.5: Plot showing the loss differences.

4.3 Barrier Control Simulator

My peer-to-peer parameter server simulator was designed to provide as realistic a simulation as possible. Unfortunately, it is prohibitively expensive to run simulations with many workers or for more than a hundred iterations. I thus created a specialised discrete-event simulator in OCaml to fully explore the impacts of different parameters on PSP. This

simulator abstracts away the details of a distributed machine learning framework and focusses on the impacts of delays to updates for different machine learning algorithms. This is the core issue which I need to explore to fully evaluate PSP.

The simulator is configurable, allowing parameters such as network size, update delays, PSP sample size, staleness and random number generator seeds to be set. Furthermore, the framework has a plug-in system to enable different machine learning algorithms to be implemented using the same underlying system.

Events in the simulator consist of computing model updates and applying these to the global model. The event duration covers both the computation and communication time.

The simulator's state is maintained in a record, see listing 4.6. Each node also has associate state, see listing 4.7. The types are polymorphic over the update type and the model parameters. This means that different machine learning algorithms can be plugged into the framework.

```
type ('a,'b) sim_state = {
    mutable clock            : float;
    mutable nodes            : 'a node_type array;
    mutable num_psp_to_check : int;
    mutable staleness        : int;
    mutable stop_time        : float;
    mutable accuracy_log     : log_type array;
    mutable params           : 'b;
    mutable step_size        : float;
}
```

Listing 4.6: Simulator state type.

```
type 'a node_type = {
    mutable id    : int;
    mutable t0    : float;  (* Last step's end time *)
    mutable t1    : float;  (* Current step's end time*)
    mutable step  : int;
    mutable delta : 'a;     (* Updates *)
}
```

Listing 4.7: Node type.

The main loop of the simulator is in listing 4.8. Here, s is the simulator's state. To run a machine learning algorithm, a set of methods needs to be registered with the system as in listing 4.9. The simulator calls these to perform operations such as initialising algorithm state, sampling from the delay distribution and calculating updates to the model.

```
while s.clock < endtime do (
    log_model_accuracy s;
    update_nodes_times s barrier;
    let update_nodes_model : ('a,'b) sim_state -> unit  =
        Marshal.from_string !_update_nodes_model 0 in
    update_nodes_model s;
    s.clock <- get_earliest_time s;
) done;
```

Listing 4.8: The main loop in the barrier control simulator.

```
Barrier_simulator.register_init_experiment init_experiment;
Barrier_simulator.register_apply_model_updates  apply_model_updates;
Barrier_simulator.register_update_nodes_model update_nodes_model;
Barrier_simulator.register_calc_model_accuracy calc_model_accuracy;
Barrier_simulator.register_get_random_time get_random_time;
Barrier_simulator.run_experiment barrier_type endtime numnodes staleness
   num_psp_to_check num_model_parameters;
```

Listing 4.9: Registering client machine learning methods.

To determine if a node can progress to the next iteration, the simulator calls a specified barrier control method. Listing 4.10 shows my implementation of the conditions I defined in section 3.1.

```
let p2p_asp s node = true

let p2p_bsp s node =
  let b = Array.for_all (fun x ->
      (node.step < x.step) || (node.step = x.step && s.clock >= x.t1)
  ) s.nodes
  in
  s.clock >= node.t1 && b

let p2p_ssp s staleness node =
  let slowest = ref node.step in
  Array.iter (fun x ->
      if x.step < !slowest then
          slowest := x.step
  ) s.nodes;
  s.clock >= node.t1 && (node.step - !slowest <= staleness)

let p2p_pspB s num_to_check node =
  let samples = Stats.choose s.nodes num_to_check in
  let b = Array.for_all (fun x ->
      (x.step > node.step)  || (x.step == node.step && s.clock >= x.t1)
  ) samples in
  s.clock >= node.t1 && b

let p2p_pspS s num_to_check staleness node =
  let samples = Stats.choose s.nodes num_to_check in
  let slowest = ref node.step in
  Array.iter (fun x ->
      if x.step < !slowest then
          slowest := x.step
  ) samples;
  s.clock >= node.t1 && (node.step - !slowest <= staleness)
```

Listing 4.10: Implementation of the barrier control methods.

4.3.1 Stochastic Gradient Descent

I use stochastic gradient descent to evaluate the proposed barrier control methods.

The standard gradient descent framework consists of a set of examples, $\{z\}$, where each z is a pair, (x, y), consisting of an arbitrary input x and a scalar output y. A loss function, $l(\hat{y}, y)$, measures the cost of predicting $\hat{y}$ when the actual answer is y. The aim is to find a function, f, and model, w, which minimises the loss $Q(z, w) = l(f_w, y)$, averaged over the examples [28]:

$$E_n(f_w) = \frac{1}{n} \sum_{i=1}^{n} l(f_w(x_i), y_i) \,. \tag{4.1}$$

Here $E_n(f_w)$ measures the training set performance. Provided f is chosen from a sufficiently restricted family, statistical learning theory justifies minimising this average, by altering the weights, to find the optimal model [28]. Gradient descent is one method which can be used to minimise $E_n(f)$. Each iteration of this algorithm updates the set of weights, w, using the gradient of $E_n(f_w)$:

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t) \,, \tag{4.2}$$

where t is the iteration counter, w_t is the model at iteration t and γ is typically a constant or a function of t. Provided suitable γ and w_0 are chosen, this algorithm achieves linear convergence [28]. Convergence results typically require that $\sum_t \gamma_t^2 < \infty$ and $\sum_t \gamma_t = \infty$.

Stochastic gradient descent is a simplification of gradient descent. Rather than computing the gradient of $E_n(f_w)$, each iteration estimates the gradient based on a single randomly selected example, z_t [28]:

$$w_{t+1} = w_t - \gamma_t \nabla_w Q(z_t, w_t) \,. \tag{4.3}$$

The intention is that this form behaves like its expectation, $E_n(f)$, despite the additional noise [28].

Implementation

Listing 4.11 shows excerpts from my SGD implementation in the barrier control simulator. This code is registered with the simulator and called when the simulator runs a model update event. Here, s is the simulator's state (see listing 4.6).

```
let data_size, model_size = MX.shape s.data_x in
let slab_size = data_size / (Array.length s.nodes) in
Array.iter (fun x ->
  if s.clock = x.t0 then (
    let start_off = slab_size * x.id in
    let end_off = start_off + slab_size - 1 in
    let i = Stats.Rnd.uniform_int start_off end_off () in
    let xt = MX.row s.data_x i in
    let yt = MX.row s.data_y i in
    let yt' = MX.(xt $@ s.model) in (* dot product of two matrices *)
    let d = MX.((transpose xt) $@ ((yt' -@ yt) *$ s.step_size)) in
    x.delta <- d
  )
) s.nodes
```

Listing 4.11: Stochastic gradient descent implementation.

4.3.2 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a probabilistic topic model I use to evaluate the proposed barrier control methods. These models aim to discover and annotate large collections of documents with topical information without prior labelling.

Specifically, LDA is a generative Bayesian model, introduced by Blei et al. [29], in which each document is modelled as a mixture over a set of topics. Each topic is modelled as a distribution over an underlying set of words.

A *word* is the basic unit of discrete data, drawn from a vocabulary indexed by the set $\{1, \ldots, V\}$. A *document* is a sequence of N words, $(w_1, w_2, \ldots, w_N)$, where w_n is the nth word in the sequence. The *corpus* is a collection of M documents, denoted by $\mathcal{D}$. The original generative process, as described by Blei et al. [29], is as follows:

1. Choose $N \sim \text{Poisson}(\zeta)$
2. Choose $\theta^d \sim \text{Dir}(\alpha)$
3. For each of the N desired words:
   (a) Choose a topic $z_n \sim \text{Multinomial}(\theta^d)$.
   (b) Choose a word, w_n, from $p(w_n|z_n, \beta)$, a multinomial probability conditioned on the topic z_n.

The Dirichlet probability distribution has the following definition:

$$p(\theta|\alpha) = \frac{\Gamma(\sum_{i=1}^{k} \alpha_i)}{\prod_{i=1}^{k} \Gamma(\alpha_i)} \theta_1^{\alpha_1 - 1} \ldots \theta_k^{\alpha_k - 1}, \tag{4.4}$$

where α is a k-vector with components such that $\forall i.\alpha_i > 0$ and $\Gamma(x)$ is the Gamma function.

It is assumed that the dimensions of the Dirichlet distribution are fixed and known. The word probabilities are held in β as a k x V matrix where $\beta_{ij} = p(w^j = 1|z^i = 1)$. Here, k is the number of topics and V is the size of the vocabulary.

The LDA model was further refined by Griffiths and Steyvers [30] who introduced a symmetric Dirichlet prior, ϕ^z, over the word-topic probabilities, parameterised by β_k. Typically, symmetric Dirichlet priors are used such that $\beta_1 = \beta_2 = \cdots = \beta$ and $\alpha_1 = \alpha_2 = \cdots = \alpha$. Values of $\alpha = \frac{50}{K}$ and $\beta = 0.01$ are suggested by Steyvers et al. [30].

More formally, let $\mathbf{z}_{-i}$ denote the topic assignments of all words except the ith one. Let w_i be the ith word token and $z_i = j$ be the assignment of word token i to topic j. Let $n_{-i}^{(.)}$ be a count that does not include the current instance of z_i. So, $n_{-i,j}^{(w_i)}$ is a count of the number of times a word is assigned to topic j, excluding the ith word and $n_{-i,j}^{(d_i)}$ is a count for the number of times topic j is assigned in the current document, excluding the current assignment. Let $n_{-i}^{(d_i)}$ be the sum over all topics and $n_{-i,j}^{(.)}$ the sum over all words. The conditional distribution is then given by [31]:

$$p(z_i = j|\mathbf{z}_{-i}, \mathbf{w}) \propto \frac{n_{-i,j}^{(w_i)} + \beta}{n_{-i,j}^{(.)} + W\beta} \frac{n_{-i,j}^{(d_i)} + \alpha}{n_{-i}^{d_i} + T\alpha}\,. \tag{4.5}$$

The Gibbs sampling algorithm, a Markov Chain Monte Carlo (MCMC) method, is an iterative procedure which enables sampling from complex distributions [30]. This is achieved by sampling on lower-dimensional subsets of variables, conditioned on the values of all others[32]. For LDA, the Gibbs sampler considers each word token in the corpus and estimates the probability of assigning it to a topic, conditioned on the topic assignments to all other word tokens. A sample drawn from this probability distribution is the new topic assignment for the word token.

The equation given by Griffiths et al. [31], (eq. 4.5), can be reduced further, as shown by Wang [33]. The key is that the second term's denominator is independent of z_i and, as the Gibbs sampler only requires a function to be proportional to $p(z_i)$, this factor can be dropped. This yields a new sampling distribution:

$$p(z_i = j|\mathbf{z}_{-i}, \mathbf{w}) \propto \frac{n_{-i,j}^{(w_i)} + \beta}{n_{-i,j}^{(.)} + W\beta}(n_{-i,j}^{(d_i)} + \alpha)\,. \tag{4.6}$$

Approximate Distributed LDA

Newman et al. [34] proposed an approximate LDA algorithm for distributed computation. First, the documents of the text corpus are distributed over a set of processors. Each machines then takes a local copy of the model and performs Gibbs sampling on its local corpus using this local model. Once all machines have sampled their collection, the local model updates are merged back into the global model. As sampling is done using local models, the sampling method is not sampling from the true distribution but an approximation. However, Newman et al. found experimentally that this method is still able to produce very good topic models.

Implementation

I implemented the approximate distributed LDA method of Newman et al. [34]. Listing 4.12 contains an excerpt from the sampling method. This is registered with the simulator to calculate model updates.

```
Array.iter (fun d ->
  Array.iteri (fun i w ->
      let k = t__z'.(d).(i) in
      (* track local model updates *)
      oldidx := Array.append !oldidx [|(w,d,k)|];
      exclude_token t__k' t_wk' t_dk' w d k;
      (* make cdf function *)
      let x = ref 0. in
      for j = 0 to s.params.n_k - 1 do
          x := !x +. (t_dk'.(d).(j) +. s.params.alpha_k)
               *. (t_wk'.(w).(j) +. s.params.beta) /. (t__k'.(j)
               +. s.params.beta_v);
          MX.set p 0 j !x;
      done;
      (* draw a sample *)
      let u = Stats.Rnd.uniform () *. !x in
      let k = ref 0 in
      while (MX.get p 0 !k) < u do k := !k + 1 done;
      (* track local model updates *)
      newidx := Array.append !newidx [|(w,d,!k)|];
      include_token t__k' t_wk' t_dk' w d !k;
      t__z'.(d).(d).(i) <- !k;
      ) s.params.data.(d)
    ) docs;
(* updates for global model *)
x.delta <- (oldidx,newidx)
```

Listing 4.12: Barrier control simulator LDA sampling implementation.

4.4 Tools

I use Owl [35], an OCaml library, for matrix and mathematical operations. For version control, I use Git [36] and Bitbucket [37]. I developed Python [38] scripts to automate the execution and analysis of my experiments. These scripts set up the necessary simulation configuration, run the experiments and then gather and analyse the results. This includes automatically plotting graphs using Matplotlib [39].

Chapter 5

Results and Analysis

In this chapter, I present and analyse results from my barrier control simulator.

5.1 Experimental Setup

The random number generators are initialised by the simulator to enable repeatable simulations. In SGD, the model size is fixed regardless of the number of nodes but each node brings with it additional data to operate on. For LDA, the model size, training set and test set are fixed. A random schedule of document identifiers is used to pick the next document.

Event times are sampled from an exponential distribution with a value of 1 for λ. Events thus follow a Poisson process: a random process where events occur independently and continuously at a constant average rate [40].

Unless stated otherwise, the error bars in my plots represent the Inter Quartile Range (IQR) of a set of simulator runs. Each run is started with a different random number seed. I take 12 runs for SGD and 5 runs for LDA.

5.2 Stochastic Gradient Descent

I now review how the different barrier control methods impact stochastic gradient descent.

5.2.1 Loss and Time

Figure 5.1 demonstrates how loss varies with time for each barrier control method. Clearly, using PSP with BSP or SSP improves the convergence rate.

The error bars on pBSP and BSP are rather large because of the stepping behaviour both mechanisms exhibit, see figure 5.2. pBSP improves on BSP because sampling a subset of workers enables it to better handle sporadic delays.

SSP offers faster convergence than BSP and pBSP because the staleness parameter allows it to be more accommodating to stragglers. However, stragglers do still occur which cause the curve to be less smooth than ASP and the error bars wider. pSSP yields

the closest results to ASP. This is due to the additional leniency the sampling primitive provides to stragglers.

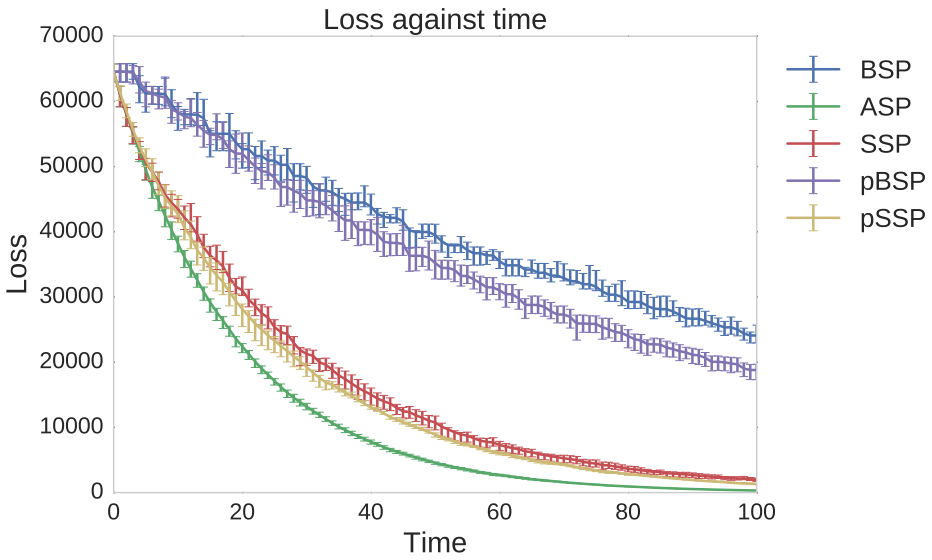


Figure 5.1: Plot showing loss against simulation time for different barrier control methods. 100 nodes are present with a staleness of 4 and a PSP sample size of 80.

Figure 5.2 shows the results from a single simulation run, highlighting details lost in figure 5.1. The stepping behaviour exhibited by BSP reveals how iteration progress is delayed by slow workers, with the step's varying widths indicating the severity of the delays. The impact of stragglers is also obvious in SSP where the curve shows definite steps. Conversely, pSSP exhibits a nearly continuous curve. This means that updates are being committed nearly unfalteringly with very short delays. The presence of the sampling primitive thus helps to reduce the impact of stragglers.

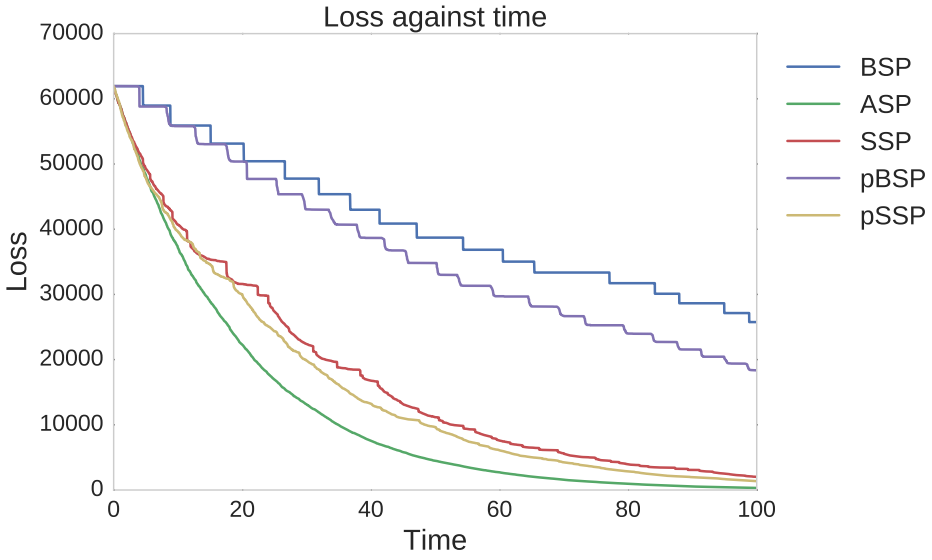


Figure 5.2: Plot showing a single simulation run with 100 nodes, a staleness of 4 and PSP sampling 80 nodes.

Figure 5.3 shows the step distributions of the barrier control methods at different times. Notably, pBSP progresses at a faster rate than BSP and pSSP moves quicker than SSP. This indicates that the sampling primitive improves the iteration throughput of each method. Furthermore, the spreads of each pair of methods (pBSP and BSP, and pSSP and SSP), remain fairly similar. This demonstrates that the additional flexibility offered by PSP does not degrade the inherent constraints of the underlying barrier method.

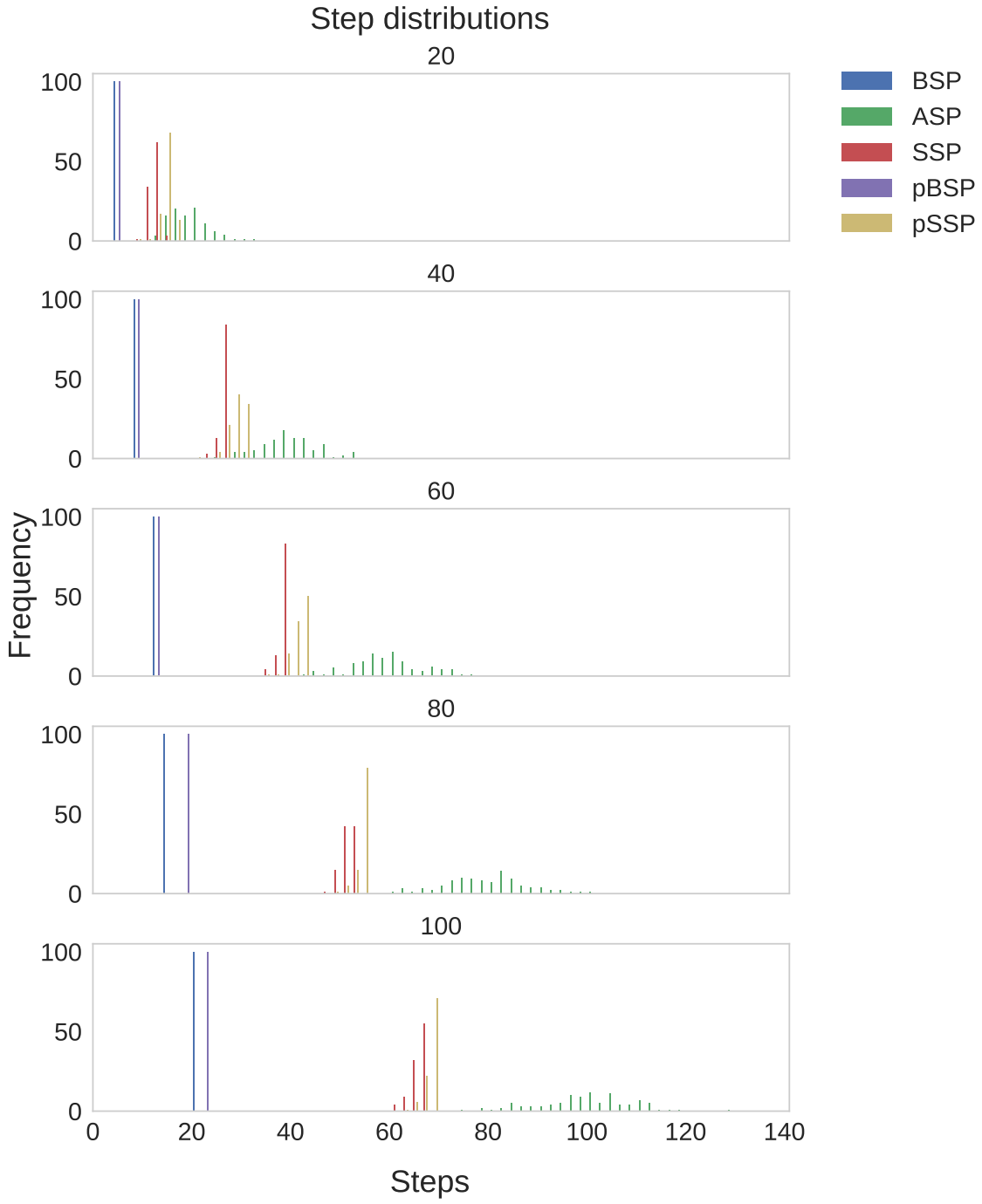


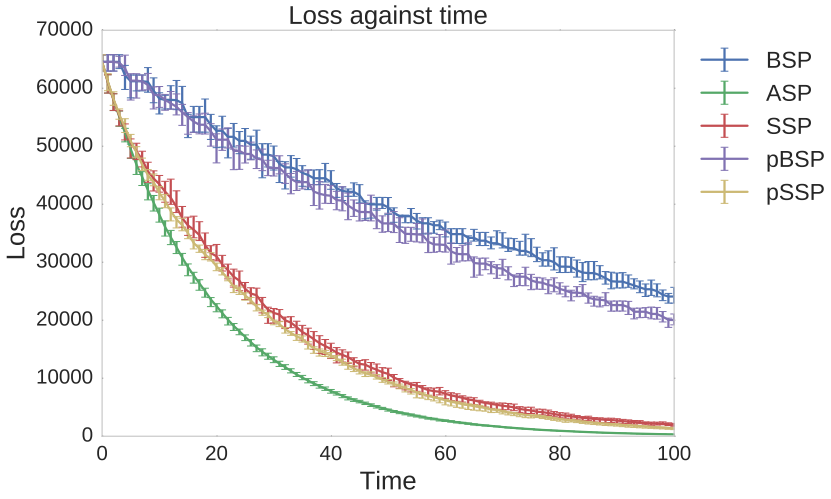
Figure 5.3: Step distributions at a range of times. 100 clients are present, staleness is set to 4 and the PSP sample size is 90.

5.2.2 Network Size

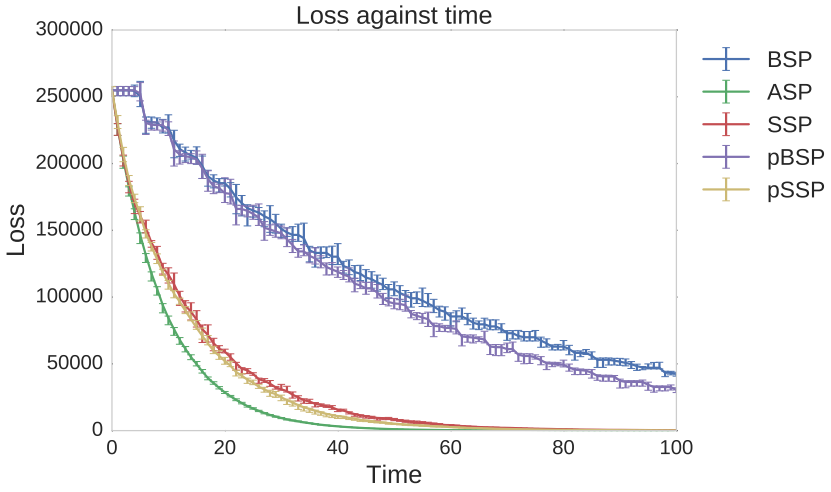
My results demonstrate that the size of the network impacts the rate of convergence, see figure 5.4. Notably, as more nodes and data are added to the system, the loss increases. However, the rate of convergence also improves, likely due to the additional compute power, and data, available to the system, which enables more progress to be made in an iteration. The more dramatic stepping behaviour of BSP and pBSP, exhibited as network size increases, supports this hypothesis. Specifically, it indicates that iteration quality improves as more nodes are added to the system.

Network size also affects iteration throughput. Figure 5.5 reveals that the iteration throughputs of BSP, pBSP, SSP and pSSP fall as network size increases. This is because of the higher probability of stragglers. ASP's iteration throughput remains reasonably constant which is expected due to the lack of synchronisation. The spreads of the distributions remain fairly consistent across network sizes.

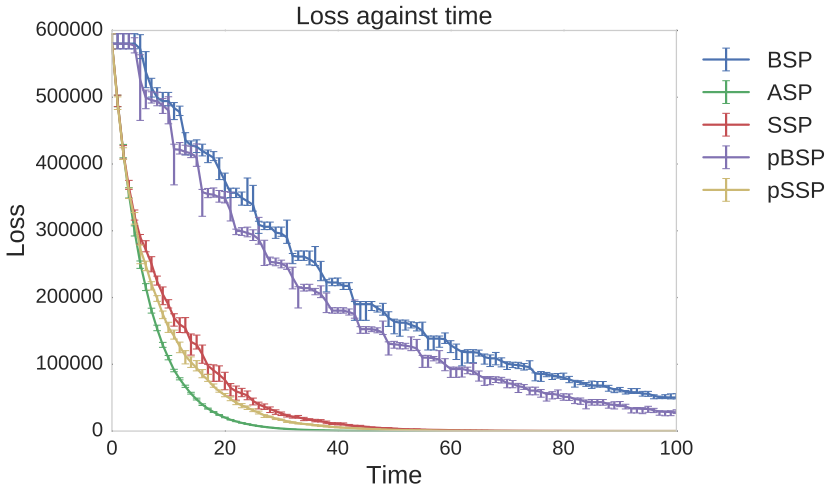
Examining these results together with the convergence rates, indicates that the increase in compute power from more nodes, balances out the resulting drop off in iteration throughput.



(a) SGD loss with 100 nodes.

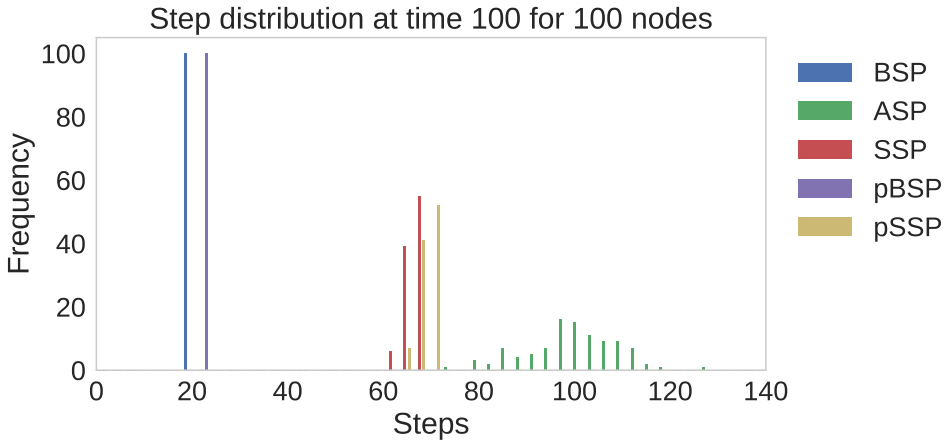


(b) SGD loss with 200 nodes.

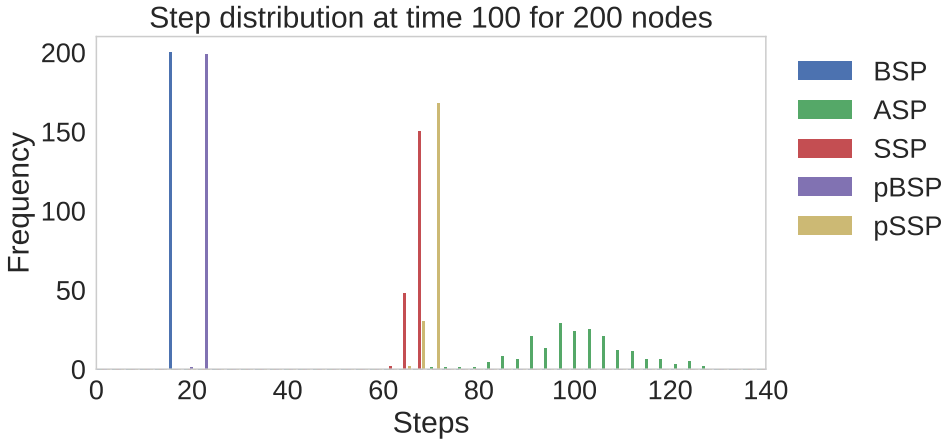


(c) SGD loss with 300 nodes.

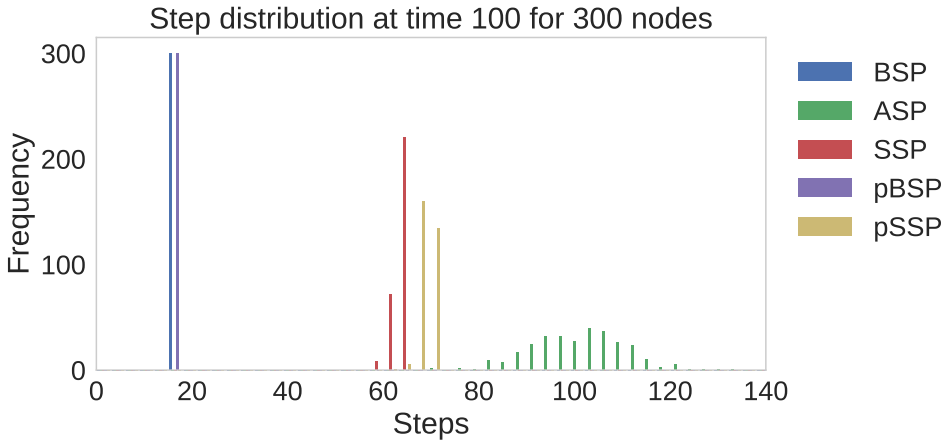
Figure 5.4: Plots of loss against time for different network sizes.



(a) SGD step distribution with 100 nodes.



(b) SGD step distribution with 200 nodes.



(c) SGD step distribution with 300 nodes.

Figure 5.5: Plots showing step distributions for different network sizes at time 100.

5.2.3 Model and Network Size

Understanding the relationship between network size and model size can aid designers in choosing the optimal network size given a barrier control method.

Figures 5.6, 5.7 and 5.8 show how loss varies with model size at different network sizes.

By increasing the model size, the magnitudes of updates should be reduced because more parameters are available to explain the data. This should decrease the potential error in the updates. In general, my results agree, indicating that, the larger the model, the smaller the final loss when all other parameters are kept the same.

Model size has the most dramatic effect on BSP and pBSP. I hypothesize that their lower iteration throughputs make them particularly susceptible to the size of the model. The other methods also benefit from increasing model size, but far less dramatically.

As network size is increased, BSP and pBSP exhibit an interesting trend where their loss curves translate left and scale up. This indicates that, if the model is too small, adding more nodes harms convergence due to the greater error in the updates. Conversely, with a large enough model, convergence can be dramatically improved by increasing network size. Designers thus need to carefully consider this trade-off.

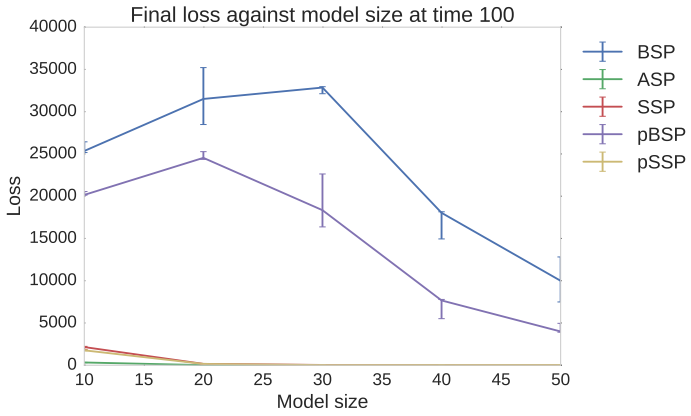


Figure 5.6: Plot of loss versus model size for 100 nodes, with a staleness of 4 and PSP sampling 90 nodes.

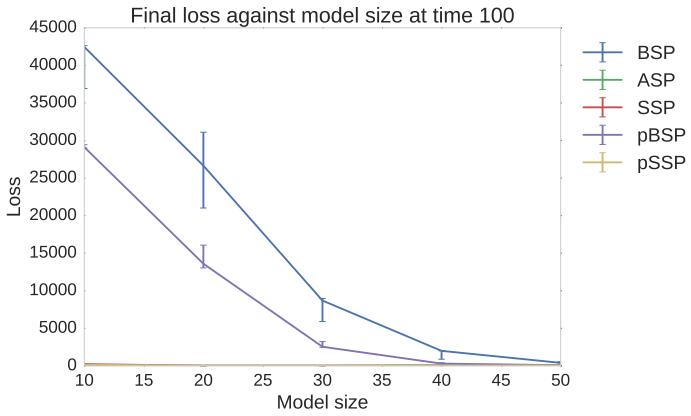


Figure 5.7: Plot of loss versus model size for 200 nodes, with a staleness of 4 and PSP sampling 180 nodes.

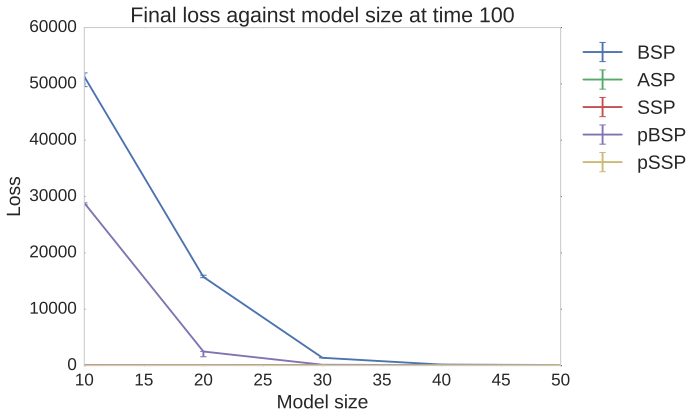


Figure 5.8: Plot of loss versus model size for 300 nodes, with a staleness of 4 and PSP sampling 270 nodes.

5.2.4 PSP Sample Size

By varying the number of nodes sampled by PSP, the leniency towards straggling nodes can be controlled. Figure 5.9 shows the final loss achieved at different sampling counts. As more nodes are sampled, the rate of convergence of pBSP decreases. In comparison, pSSP appears almost unaffected. These results support the hypothesis that pSSP has better intrinsic tolerance to stragglers. Furthermore, as pBSP is pSSP with staleness set to 0, this figure reveals some of the interplay between sample size and staleness. As staleness is increased, the reliance on the sampling count decreases for the same network conditions. However, it is clear that adding in the sampling primitive provides far greater flexibility than using just staleness alone.

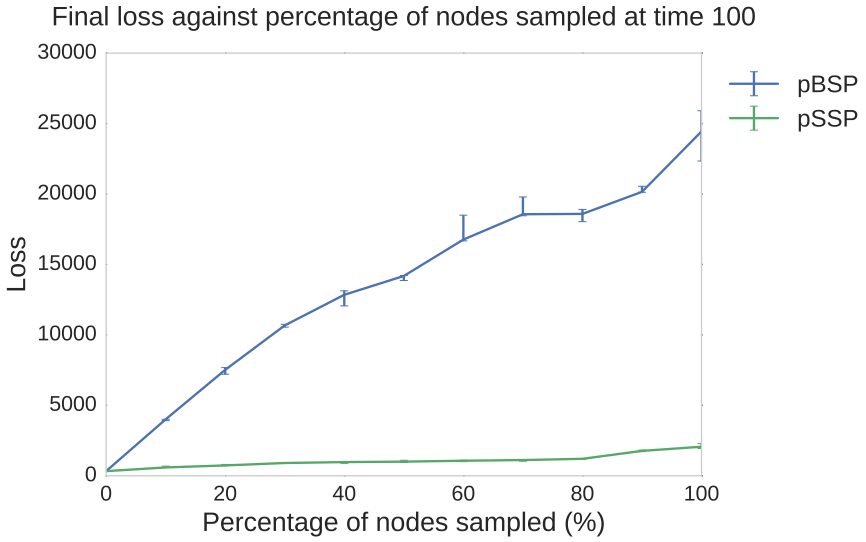
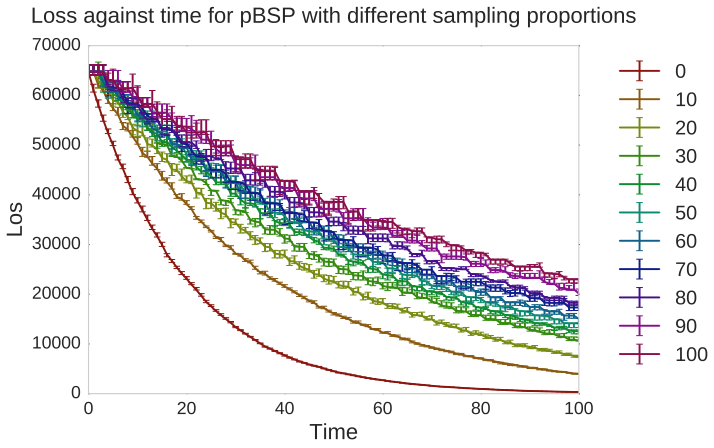


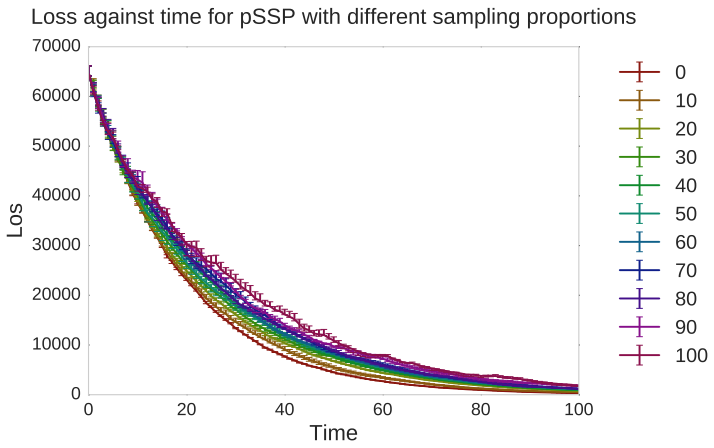
Figure 5.9: Plot of final loss against the proportion of nodes sampled. There are 100 nodes with pSSP's staleness set to 4.

Figure 5.10a provides more detail on the impact on pBSP's convergence rate. Clearly, the returns from decreasing the sample size are non-linear across a large range of times. It is only when the model is nearing convergence, as in figure 5.9, that the relationship

appears more linear. Figure 5.10b also reveals that sample size has more of an impact on pSSP than figure 5.9 initially indicates – albeit not as dramatically as with pBSP. The middle of the computation shows the greatest variation with sample size. This more stable behaviour is likely because the number of stragglers is not sufficient to overpower the leniency of the staleness parameter.



(a) pBSP: losses at different sample sizes.

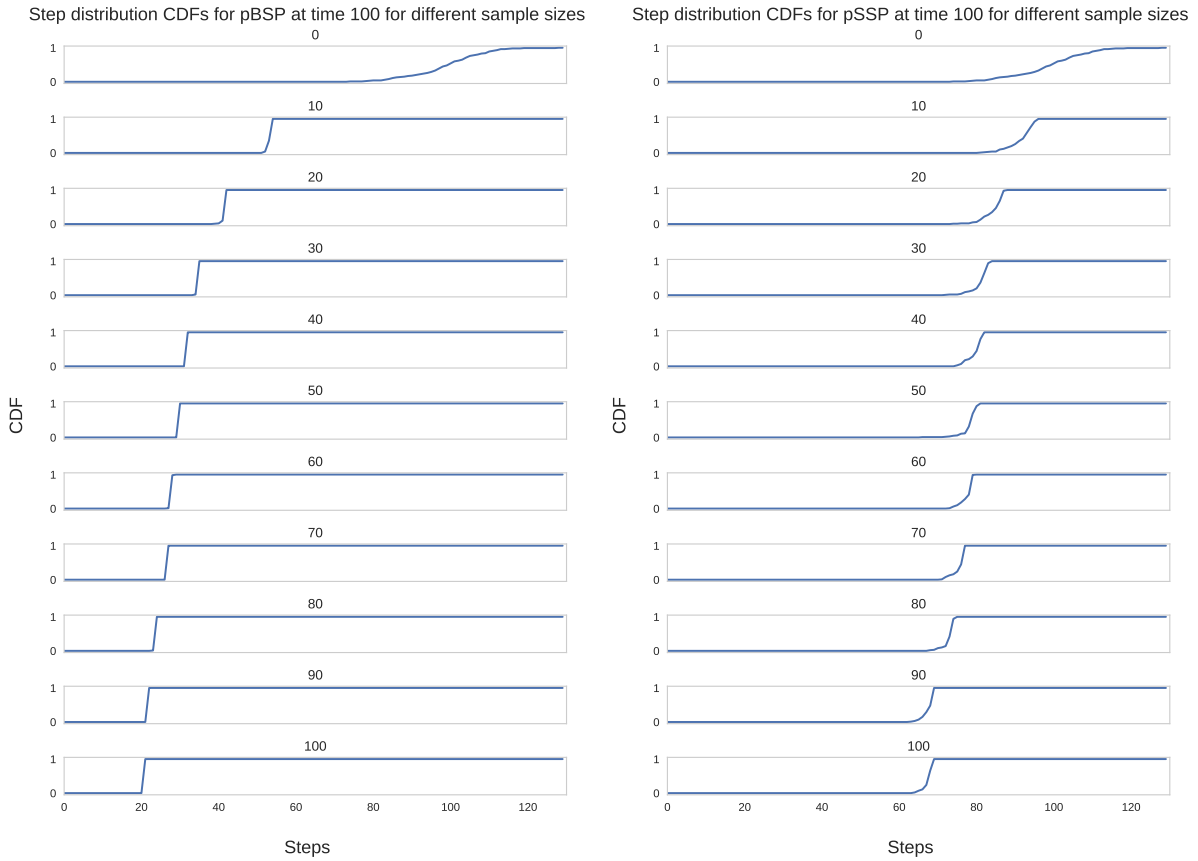


(b) pSSP: losses at different sample sizes.

Figure 5.10: Plots of loss vs. time with different sample sizes. A total of 100 nodes were used with a model size of 10. pSSP was run with a staleness parameter of 4.

Figure 5.11a indicates that taking only a small number of samples quickly forces the pBSP step distribution to approximate the full BSP distribution — roughly 20%. These results are supported by my convergence bounds, see section 3.5. Increasing the sample size further just shifts the distribution to the left, reducing iteration throughput and the convergence rate. This hints that sampling a small number of nodes strikes a good balance between a full BSP and a full ASP system.

The story is very similar for pSSP. However, the sample size needs to be larger to yield step distributions very close to SSP, see figure 5.11b.



(a) pBSP (b) pSSP

Figure 5.11: Step distribution CDFs for pBSP and pSSP. The system configuration is the same as in figure 5.10.

5.3 Latent Dirichlet Allocation

It is revealing to analyse how PSP fares with more complication machine learning algorithms. I use the Latent Dirichlet Allocation model for this purpose.

5.3.1 Log-likelihood and Time

Figure 5.12 shows how the log-likelihood varies with time under different barrier control methods. The results follow the same general patterns I identified with SGD. pBSP generally outperforms BSP and pSSP outperforms SSP, with ASP performing best. Figure 5.13 shows that the same stepping behaviour, observed using BSP and pBSP with SGD, is also present.

These results indicate that PSP is suitable for a range of different machine learning algorithms.

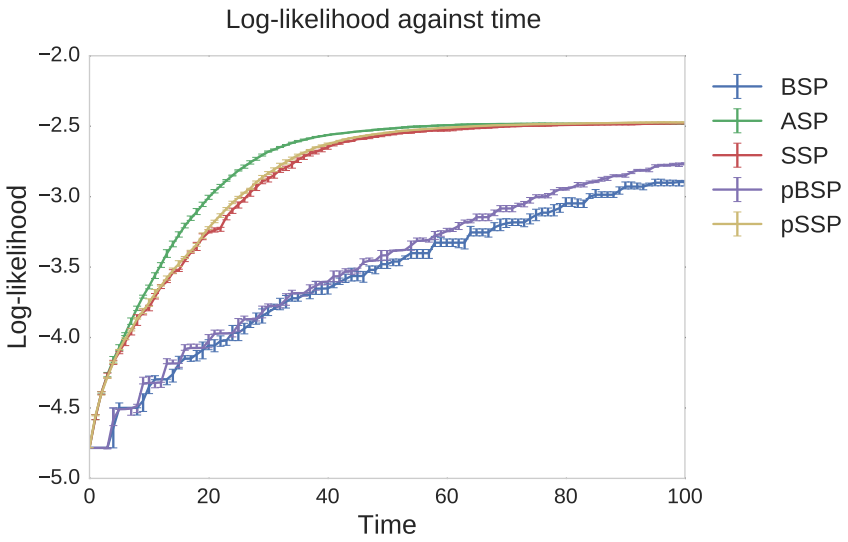


Figure 5.12: Plot showing how log-likelihood varies with time for different barrier control methods. The PSP sample size is 80 with a staleness of 4.

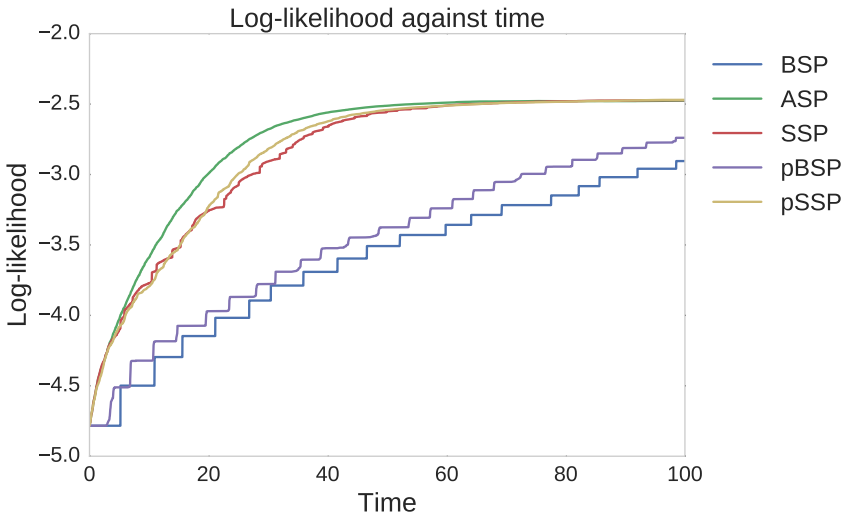


Figure 5.13: Plot showing the results from a single simulation run. The PSP sample size is set to 80 with a staleness of 4.

5.3.2 PSP Sample Size

Figure 5.14 summarises the impact of sample size on the log-likelihood. The patterns in the results once again reflect those I observed with SGD. pSSP remains fairly flat whereas pBSP shows strong dependence on sample size. Figure 5.15 shows a more detailed view, demonstrating the same trends as PSP under SGD.

For the step distributions, figure 5.16a reveals that, as with SGD, only a small proportion of nodes need to be sampled in pBSP to yield distributions similar to BSP. pSSP requires a larger sample size to produce the same results. Iteration throughput once again decreases as sample size increases.

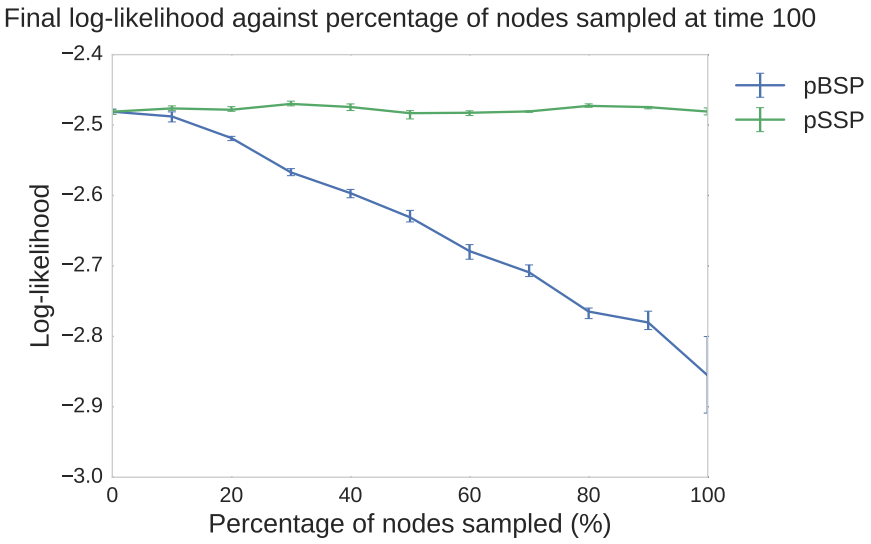
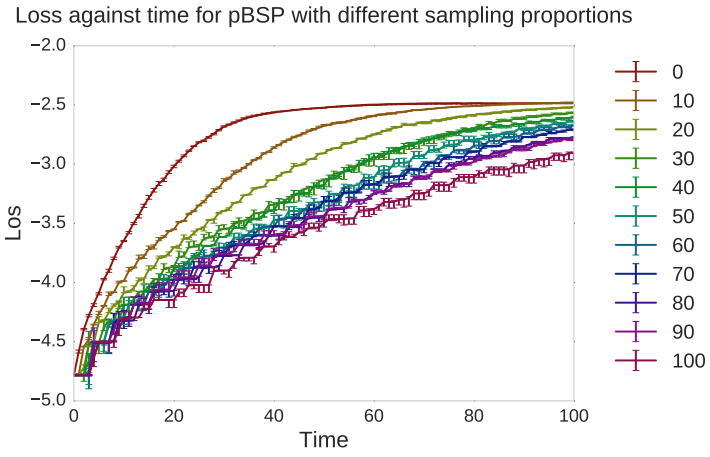
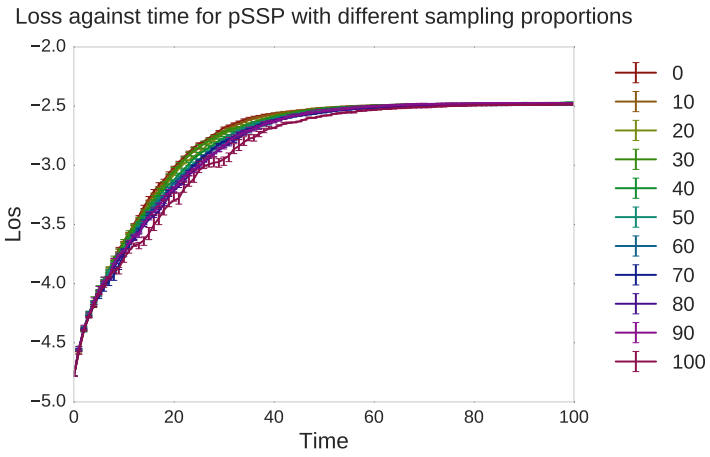


Figure 5.14: Plot showing the final log-likelihood against the proportion of nodes sampled. The system has 100 nodes and a staleness parameter of 4.

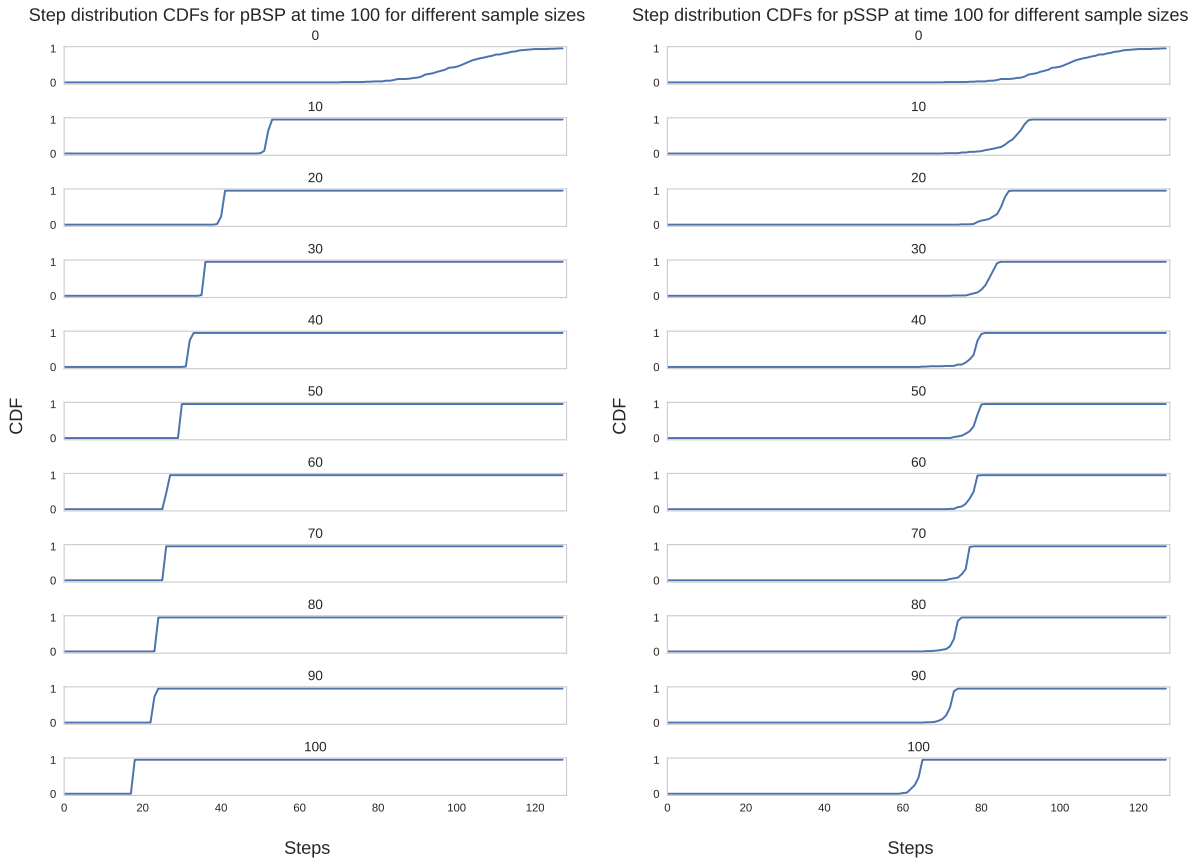


(a) pBSP: log-likelihood at different sample sizes.



(b) pSSP: log-likelihood at different sample sizes.

Figure 5.15: Plots of log-likelihood against time at different sampling proportions. The system has 100 nodes and a staleness parameter of 4.



(a) pBSP (b) pSSP.

Figure 5.16: Step distribution CDFs for pBSP and pSSP.

Chapter 6

Conclusion

In this report, I have presented the Probabilistic Synchronous Parallel (PSP) barrier control method, a new synchronisation scheme for distributed machine learning.

I have contributed analytical proofs of PSP's convergence properties which demonstrate that, provided reasonable conditions are met, algorithms using PSP are guaranteed to converge in probability.

I provided experimental verification of PSP, evaluating its performance using stochastic gradient descent (SGD) and latent dirichlet allocation (LDA). I developed a bespoke simulator for this purpose. My results revealed that PSP improves the convergence rate and iteration throughput of both BSP and SSP whilst maintaining their general characteristics. Furthermore, my simulations indicated that only a small proportion of nodes, roughly 20% for pBSP, need to be sampled to yield most of these benefits. This finding is supported by my analytical results on the convergence bounds. I also discovered that the relationship between network and model size is particularly important for BSP and pBSP.

My theoretical analysis demonstrated that PSP provides stronger convergence bounds than ASP and requires far weaker assumptions on the distribution of stragglers.

Methods such as BSP and SSP require an oracle to track the state of each worker in order to make accurate decisions. Conversely, PSP is theoretically able to function in a distributed setting where each worker has a local view of the current state of the system. This reduces communication and synchronisation overheads. My peer-to-peer parameter server simulator used a worker's local state to make barrier decisions and I found that PSP performed as expected. Further work could explore the impacts of distributed state in more detail.

Future work could also analyse PSP on real world systems and empirically evaluate its convergence properties under different lag distributions. A more detailed exploration of the interactions of sample size and staleness could also be performed.

The successful application of PSP to SGD and LDA, in a variety of different network configurations, demonstrates its viability as a synchronisation primitive. Overall, PSP gives distributed machine learning practitioners the flexibility they need to find the exact balance between a fully deterministic and a fully asynchronous synchronisation scheme.

Bibliography

[1] Wei Dai et al. "High-Performance Distributed ML at Scale through Parameter Server Consistency Models". In: *CoRR* (2014). arXiv: 1410.8043v1 [cs.LG].

[2] Eric P. Xing et al. "Strategies and Principles of Distributed Machine Learning on Big Data". In: *CoRR* (2016). arXiv: 1512.09295 [stat.ML].

[3] Xun Zheng et al. "Model-Parallel Inference for Big Topic Models". In: *CoRR* (2016). arXiv: 1411.2305 [cs.DC].

[4] James Cipar et al. "Solving the straggler problem with bounded staleness". In: *HotOS Usenix* (2013).

[5] Qirong Ho et al. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server". In: *NIPS'13: Advanced Neural Information Processing Systems* (2013).

[6] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010).

[7] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI: Sixth Symposium on Operating System Design and Implementation* (2004).

[8] Yucheng Low et al. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proceedings of the VLDB Endowment* 5 (8 2012). arXiv: 1204.6078v1 [cs.DB].

[9] Mu Li et al. "Parameter Server for Distributed Machine Learning". NIPS Working paper. 2013.

[10] Seunghak Lee et al. "Primitives for Dynamic Big modle Parallelism". In: *CoRR* (2014). arXiv: 1406.4580v1 [stat.ML].

[11] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Communications of the ACM 33(8)* (1990), pp. 103–111.

[12] Martin A. Zinkevich et al. "Parallelized Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems* (2010).

[13] Amr Ahmed et al. "Scalable Inference in Latent Variable Models". In: *WSDM* (2012), pp. 123–132.

[14] Grzegorz Malewicz et al. "Pregel: A System for Large-Scale Graph Processing". In: *Proceedings of the 2010 International Conference on Management of Data* (2010), pp. 135–146.

[15] Feng Niu et al. "Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *NIPS'11 Proceedings of the 24th International Conference on Neural Information Processing Systems* (2011), pp. 693–701. arXiv: 1106.5730 [math.OC].

[16] J.Langford, A. J. Smola, and M. Zinkevich. "Slow learners are fast". In: *Advances in Neural Information Processing Systems* (2009), pp. 2331–2339.

[17] Joseph K. Bradley et al. "Parallel Coordinate Descent for L_1-Regularised Loss Minimization". In: *Proceedings of the 28th international Conference on Machine Learning, Bellevue, WA, USA* (2011).

[18] Kyong-Ha Lee et al. "Parallel Data Processing with MapReduce: A Survey". In: *SIGMOD* 40 (4 December 2001).

[19] Alekh Agarwal and John C. Duchi. "Distributed Delayed Stochastic Optimization". In: *NIPS'11 Advances in Neural Information Processing Systems 24* (2011). arXiv: 1104.5525v1 [math.OC].

[20] David Pollard. *Convergence of Stochastic Processes*. Springer-Verlag new York Inc, 1984. ISBN: 0-387-90990-7.

[21] D. Khattar. *The Pearson Guide to Mathematics for the IIT-JEE*. 2nd ed. Pearson Eduction India. ISBN: 81-317-2876-5.

[22] Liang Wang. *OCaml Distributed Data Processing*. URL: https://github.com/ryanrhymes/actor (visited on 11/17/2016).

[23] *OCaml*. URL: ocaml.org (visited on 04/29/2017).

[24] Norm Matloff. *Introduction to Discrete-Event Simulation and the SimPy Language*. 2008. URL: http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf.

[25] Ion Stoica et al. "Chord: A Scalable peer-to-peer Lookup Service for Internet Applications". In: *SIGCOMM'01 August 27-31* (2001).

[26] David Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the29th Annual ACM Symposium on Theory of Computing* (1997), pp. 654–663.

[27] *ZMQ: Distributed Messaging*. URL: zeromq.org.

[28] Leon Bottou. "Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT* (2010).

[29] David M. Blei, Andrew Y. Ng, and Michael L. Jordan. "Latent Dirichlet Allocation". In: *Machine Learning Research* 3 (2003), pp. 993–1022.

[30] Mark Steyvers and Tom Griffiths. "Probabilistic Topic Models". In: *Latent Semantic Analysis: A Road to Meaning* (2007).

[31] Thomas L. Griffiths and Mark Steyvers. "Finding scienfitic Topics". In: *Proceedings of the National Academy of Sciences of The United States of America* (2004).

[32] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012. ISBN: 9780262306164.

[33] Yi Wang. *Distributed Gibbs Sampling of Latent Topic Models: The Gritty Details*. 2008. URL: `https : / / pdfs . semanticscholar . org / a166 / d65a5d5a2905b038288e59c4fd98864c6f44.pdf`.

[34] David Newman et al. "Distributed Algorithms for Topic Models". In: *Journal of Machine Learning Research 10* (2009), pp. 1801–1828.

[35] Liang Wang. *Owl - An OCaml Numerical Library*. URL: `https://github.com/ryanrhymes/owl` (visited on 11/17/2016).

[36] *Git*. URL: `https://git-scm.com/` (visited on 05/01/2017).

[37] *Bitbucket*. URL: `https://bitbucket.org/` (visited on 05/01/2017).

[38] *Python*. URL: `https://www.python.org/` (visited on 05/01/2017).

[39] *Matplotlib*. URL: `https://matplotlib.org/` (visited on 05/01/2017).

[40] Geoffrey Grimmett and David Stirzaker. *Probability and Random Processes*. 3rd. Oxford University Press, 2001. ISBN: 978-0-19-857222-0.

Appendix A

Additional Proofs

A.1 Proof of the closed form solution for the squared arithmetico-geometric series

The partial sum of the first n terms of the squared arithmetico-geometric series has the form:

$$S_n = \sum_{k=1}^{n} t_k = \sum_{k=1}^{n} (a + (k-1)d)^2 r^{k-1} \tag{A.1}$$

$$= a^2 + (a+d)^2 r + (a+2d)^2 r^2 + \cdots + (a + (n-1)d)^2 r^{n-1} \,. \tag{A.2}$$

Multiply S_n by r,

$$rS_n = a^2 r + (a+d)^2 r^2 + (a+2d)^2 r^3 + \cdots + (a + (n-1)d)^2 r^n \,. \tag{A.3}$$

Subtracting rS_n from S_n and rearranging:

$$\begin{aligned}
(1-r)S_n =& \left(a^2 + (a+d)^2 r + (a+2d)^2 r^2 + \cdots + (a + (n-1)d)^2 r^{n-1}\right) \\
& - \left(a^2 r + (a+d)^2 r^2 + (a+2d)^2 r^3 + \cdots + (a + (n-1)d)^2 r^n\right) \\
=& \left(a^2 + (a^2 + 2ad + d^2) r + (a^2 + 4ad + 4d^2) r^2 + \cdots + (a^2 + 2(n-1)ad + (n-1)^2 d^2) r^{n-1}\right) \\
& - \left(a^2 r + (a^2 + 2ad + d^2) r^2 + (a^2 + 4ad + 4d^2) r^3 + \cdots + (a^2 + 2(n-1)ad + (n-1)^2 d^2) r^n\right) \\
=& a^2 + 2ad(r + \cdots + r^{n-1}) + d^2(r + 3r^2 + 5r^3 + 7r^4 + \cdots + (2n-3))r^{n-1}) \\
& - (a + (n-1)d)^2 r^n \\
=& a^2 + 2ad(\sum_{k=0}^{n-1}(r^k) - 1) + d^2(\sum_{k=1}^{n-1}\left((1 + 2(k-1))r^{k-1}\right) - 1) - (a + (n-1)d)^2 r^n \,.
\end{aligned}$$

The $2ad$ term contains a geometric sum and the d^2 term an arithmetico-geometric sum. There exist closed-form solutions to both partial sums. For the arithmetico-geometric series where

$$S_n = \sum_{k=1}^{n} (a + (k-1)d) r^{k-1} \,, \tag{A.4}$$

if $r \neq 1$ then,

$$S_n = \frac{a - (a + (n-1)d) r^n}{1 - r} + \frac{dr(1 - r^{n-1})}{(1-r)^2} \,. \tag{A.5}$$

See D. Khattar's [21] book for the proof.

For the geometric series, its partial sum requires $|r| < 1$. So, for $r \neq 1$, make the substitutions $a = 1$, $d = 2$, $n = n - 1$ and $r = r$. This yields:

$$\begin{aligned}(1-r)S_n =& a^2 - 2ad - d^2 + \frac{1-r^{n-1}}{1-r} + d^2\left(\frac{1-(1+2(n-2))r^{n-1}}{1-r} + \frac{2r(1-r^{n-2})}{(1-r)^2}\right) \\ & - (a+(n-1)d)^2 r^n \,.\end{aligned}$$

Thus,

$$S_n = \frac{a^2 - 2ad - d^2 - (a+(n-1)d)^2 r^n}{(1-r)} + \frac{1-r^{n-1}}{(1-r)^2} \tag{A.6}$$

$$+ \frac{d^2}{1-r}\left(\frac{1-(1+2(n-2))r^{n-1}}{1-r} + \frac{2r(1-r^{n-2})}{(1-r)^2}\right). \tag{A.7}$$

Consider now the $r = 1$ case. We have:

$$S_n = \sum_{k=1}^{n} t_k = \sum_{k=1}^{n} (a+(k-1)d)^2 \tag{A.8}$$

$$= a^2 + (a+d)^2 + (a+2d)^2 + \cdots + (a+(n-2)d)^2 + (a+(n-1)d)^2 \tag{A.9}$$

$$= a^2 + a^2 + 2ad + d^2 + a^2 + 4ad + 4d^2 + \cdots + \tag{A.10}$$

$$a^2 + 2(n-2)d + (n-2)^2 d^2 + a^2 + 2(n-1)d + (n-1)^2 d^2 \,. \tag{A.11}$$

Collecting terms yields:

$$= na^2 + 2d\sum_{k=1}^{n-1} k + d^2 \sum_{k=1}^{n-1} k^2 \,. \tag{A.12}$$

Using arithmetic series and squared arithmetic series partial sum equations achieves:

$$= na^2 + \frac{2d(n-1)(n)}{2} + \frac{d^2 n(n-1)(2n-1)}{6} \,. \tag{A.13}$$

To conclude, the closed-form solution to the partial sum of the squared arithmetico-geometric sequence is as follows:

$$S_n = \begin{cases} \left[\dfrac{a^2 - 2ad - d^2 - (a+(n-1)d)^2 r^n}{(1-r)} + \dfrac{1-r^{n-1}}{(1-r)^2} \right. \\ \left. + \dfrac{d^2}{1-r}\left(\dfrac{1-(1+2(n-2))r^{n-1}}{1-r} + \dfrac{2r(1-r^{n-2})}{(1-r)^2}\right)\right], & \text{for } |r| < 1 \\ na^2 + \dfrac{2d(n-1)(n)}{2} + \dfrac{d^2 n(n-1)(2n-1)}{6} & \text{for } r = 1 \end{cases} \tag{A.14}$$

□