DEMYSTIFYING PIPELINE PARALLELISM: FIRST THEORY FOR PIPEDREAM

Anonymous authors
Paper under double-blind review

ABSTRACT

Training modern machine learning models increasingly requires computation to be distributed across many accelerators. Data parallelism remains the default choice and is often paired with tensor-parallel sharding, but model parallelism becomes unavoidable once parameters, activations, or optimizer states no longer fit on a single device. This paper studies pipeline model parallelism through the lens of PipeDream (PD) (Harlap et al., 2018). Our first contribution is theoretical: we introduce Randomized PipeDream (RPD), a stale block-SGD abstraction that yields, to our knowledge, the first clean nonconvex convergence guarantee for a PD-style method. Our second contribution is a scaling diagnosis: we prove that the delay induced by steady-state PD grows as $S^2 - S/2 + O(1)$ for S stages, so the stale-read contribution in the convergence theorem scales as $\Theta(\gamma^2 S^4)$, equivalently as $\Theta(S^4/K)$ in the tuned-rate form. Our third contribution is a comparison with LocalSGD, whose periodic model averaging trades weight staleness for synchronization bubbles. In our reported simulated-time experiments, the better-performing method depends on the objective: PD performs better on the quadratic objective and on a small language-modeling training-loss task, while for logistic regression LocalSGD becomes superior as the number of stages increases.

1 INTRODUCTION

The recent success of deep learning has been driven in part by scaling: larger models, larger datasets, and longer training runs typically deliver better downstream performance. That scaling pressure immediately creates a systems problem. Even when a model fits on one accelerator, training time can be prohibitive; once the parameters, activations, and optimizer states no longer fit on one accelerator, single-device training is impossible. Distributed optimization is therefore not an implementation detail but a prerequisite for modern large-scale learning (Dean et al., 2012; Goyal et al., 2017).

The most widely used training strategy is data parallelism: each worker holds a model replica, computes gradients on different mini-batches, and synchronizes updates across workers. In practice, large training stacks often combine this outer-loop data parallelism with tensor-parallel sharding inside layers to spread matrix multiplications across multiple devices (Shoeybi et al., 2019). By contrast, *model parallelism* partitions the model itself across devices. This partition can be even or uneven, and it may follow layers, tensor dimensions, or other architectural boundaries. Model parallelism is less ubiquitous than pure data parallelism, but it is equally important whenever the model simply cannot be stored or executed on a single device (Huang et al., 2019; Narayanan et al., 2019).

This paper focuses on pipeline model parallelism. The basic idea is simple: split a deep network into S stages, place stage s on device s, and stream microbatches through the stage sequence. However, the naive schedule is painfully inefficient. If a model is partitioned across four devices, a single microbatch moves forward from stage 1 to stage 4 and then backward from stage 4 to stage 1. During much of that time, several machines are idle (Figure 1a). These idle regions are the classic *pipeline bubbles*. PipeDream (PD) (Narayanan et al., 2019) was one of the first influential methods to attack this issue: it overlaps multiple microbatches and uses a one-forward-one-backward (1F1B) steady-state schedule so that machines spend far less time doing nothing (Figure 1b).

The systems advantage of PD is clear, but its optimization behavior is hard to analyze. The core obstacle is not merely delay; it is *structured*, stage-dependent delay produced by weight stashing

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

and by the deterministic 1F1B schedule. Each backward pass uses a collection of stage-local weight versions encountered earlier during the corresponding forward pass, and these versions are coupled through the pipeline timeline. Existing literature provides excellent systems insight into pipeline-parallel training, yet a tractable convergence theory for PD itself remains largely missing.

This paper has three main contributions.

1. We introduce Randomized PipeDream (RPD), a mathematical abstraction in which the active stage and mini-batch are sampled uniformly and the stage-wise staleness is arbitrary but bounded by a hyperparameter δ. For this abstraction we prove a clean nonconvex convergence rate, and we then connect the theory back to the original method by showing that steady-state PD is described by $\delta = S^2 - S/2 + O(1)$.
2. We use this theory to explain why PD scales poorly with the number of stages. Because the theorem contains a δ^2 term and because $\delta = \Theta(S^2)$, the stale-read term grows as $\Theta(\gamma^2 S^4)$, or $\Theta(S^4/K)$ after the tuned stepsize substitution. Under a fixed iteration budget, increasing the number of stages slows objective decrease substantially.
3. We study LocalSGD as an alternative schedule that trades PipeDream-style stale weight versions for synchronization barriers and local-replica drift. The resulting comparison is not one-sided: our experiments on quadratic objectives, logistic regression, and an 11.1M-parameter NanoChat-style Transformer show that the better schedule depends on the objective and scaling regime. PD is stronger on the quadratic and language-modeling tasks, while LocalSGD outperforms it on logistic regression when the number of stages is large and gradients are noisy.

The implementation and code for reproducing our experiments are provided as supplementary material and will be released publicly upon publication.

1.1 Notation

All key notation used in this paper is summarized in a tabular form in Section A; see Table 2.

2 Related Work

Distributed and pipeline-parallel training. Distributed training scales modern deep networks either by replicating models across workers in data-parallel SGD (Dean et al., 2012; Goyal et al., 2017), or by partitioning computation, parameters, activations, and optimizer states across devices. Tensor-parallel systems such as Megatron-LM split large Transformer layers across accelerators (Shoeybi et al., 2019); ZeRO instead reduces data-parallel memory redundancy by partitioning optimizer states, gradients, and parameters (Rajbhandari et al., 2020); and FlexFlow/Alpa search over hybrid parallelization plans combining data, operator, tensor, and pipeline parallelism (Jia et al., 2019; Zheng et al., 2022). Our work is orthogonal to these systems contributions: we isolate the optimization effect of the stale weights induced by pipeline-parallel execution.

Pipeline parallelism partitions a model into consecutive stages and streams microbatches through them. GPipe pipelines microbatches synchronously and updates weights only after a mini-batch has flushed through (Huang et al., 2019), whereas PD uses inter-batch pipelining with a one-forward-one-backward schedule to keep stages active (Harlap et al., 2018; Narayanan et al., 2019); its weight-stashing mechanism preserves forward/backward consistency per microbatch, but gradients are evaluated using stale stage-local weights. Subsequent systems such as PipeDream-2BW and DAPPLE reduce memory overhead, improve scheduling, or combine pipeline and data parallelism (Narayanan et al., 2021; Fan et al., 2021), primarily optimizing throughput, memory, placement, and scheduling. In contrast, we study the nonconvex optimization cost of the stale weight versions created by PipeDream-style execution.

Stale-gradient and randomized block optimization. Pipeline-induced stale gradients are related to asynchronous optimization: Hogwild! established convergence of lock-free stochastic updates under sparsity assumptions (Recht et al., 2011), later extended to nonconvex objectives under bounded delays (Lian et al., 2015). The perturbed-iterate framework models stale reads as gradients evaluated at perturbed iterates (Mania et al., 2017), and delay-compensation methods correct staleness via Taylor or Hessian-based approximations (Zheng et al., 2017). Our setting differs from generic

asynchronous SGD because the delay structure is generated by a deterministic pipeline schedule, stage-local weight stashing, and forward/backward ordering; RPD keeps the stale block-gradient structure while removing this scheduling combinatorics. Our abstraction also connects to randomized (block-)coordinate methods (Nesterov, 2012; Richtárik & Takáč, 2014; Xu & Yin, 2015), but unlike standard block-coordinate methods, our active block gradient is evaluated at a stale model assembled from different stage-local weight versions.

Local SGD and model averaging. LocalSGD reduces communication by letting workers take several local SGD steps before averaging models, going back to parallelized SGD with model averaging (Zinkevich et al., 2010); later work matched mini-batch SGD rates with fewer communication rounds in convex settings (Stich, 2018) and analyzed nonconvex deep learning objectives (Yu et al., 2019), with empirical evidence that local updates can improve efficiency and generalization over simply increasing the batch size (Lin et al., 2018). We use LocalSGD as an alternative stage-distributed policy: several full-model trajectories run through the same pipeline stages and are periodically averaged, avoiding PD's cross-stage staleness at the cost of synchronization barriers and local-replica drift.

3 Background: Why Pipeline Parallelism?

We study the finite-sum problem

$$\min_{w\in\mathbb{R}^d} f(w), \qquad f(w) = \frac{1}{M}\sum_{m=1}^{M} f_m(w), \tag{1}$$

where the model is partitioned into S blocks

$$w = \big(w^{(1)}, \ldots, w^{(S)}\big), \qquad w^{(s)} \in \mathbb{R}^{d_s}, \qquad \sum_{s=1}^{S} d_s = d.$$

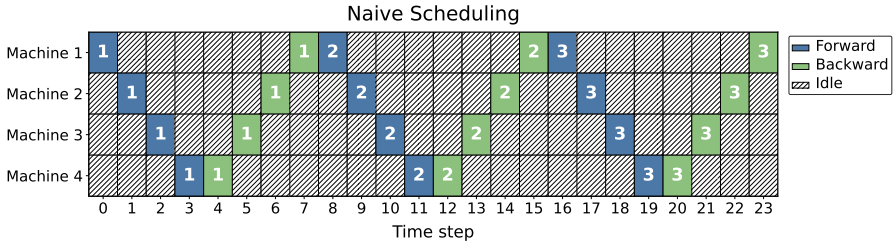
Each block corresponds to one pipeline stage. In practice, the partition may be balanced or intentionally uneven to match heterogeneous devices or layer costs.

Naive pipelining. If we send a single microbatch through the pipeline, the forward pass proceeds stage-by-stage from 1 to S, and the backward pass returns from S to 1. This schedule is functionally correct but extremely inefficient because most devices are idle during startup and drain. The result is a large bubble fraction, especially for deep pipelines (Figure 1a).

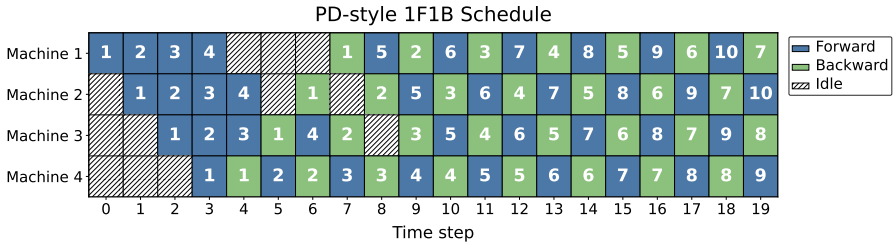
PipeDream. PD reduces these bubbles by admitting multiple microbatches into the pipeline and switching each stage into a 1F1B steady state after startup (Narayanan et al., 2019). The price is that gradients become stale: a backward pass does not use the newest global model, but rather the stage-local weights that were seen earlier during the corresponding forward pass. Weight stashing preserves forward/backward consistency within each stage, but it also creates a complicated pattern of cross-stage version dependencies.

Figure 1 visualizes the contrast between PD and a naive pipeline. The systems motivation for PD is therefore immediate: it converts idle pipeline time into useful work. The theoretical question is more subtle: what is the optimization cost of the induced staleness, and how does that cost scale with the number of stages? The discrete-event procedure used to generate the PD timelines in our experiments is described in Appendix E.1.

From schedules to optimization updates. A pipeline schedule determines not only which device is active, but also which parameter version is used when a gradient is computed. We distinguish two notions of progress. The first is the *block-update budget* K, which counts how many individual stage/block updates have been applied. This is the natural unit for the convergence theory. The second is *simulator time*, which counts discrete pipeline time steps and is closer to a wall-clock proxy: in one simulator step, several stages may perform work in parallel. We use K for theoretical comparisons and simulator time for experiments that study hardware-utilization tradeoffs. This distinction is important because increasing the number of stages can increase parallel work per time step while also increasing the staleness of the gradients being applied.



(a) Naive pipeline for one microbatch. The blank area is pure idle time.



(b) PD 1F1B schedule fills most of the bubble after startup.

Figure 1: Pipeline scheduling for a four-stage model. The naive schedule is sequential and wastes hardware; PD overlaps multiple microbatches to keep stages busy.

4 Randomized PipeDream: A Tractable Theory

The exact deterministic PD schedule is difficult to analyze because the active stage, the microbatch identity, and the stage-wise weight versions are all jointly coupled by the 1F1B timeline. We therefore introduce Randomized PipeDream (RPD), a simplified model that keeps the essential stale block-gradient structure while removing the deterministic scheduling combinatorics.

At iteration k, let

$$w_k = \left(w_k^{(1)}, \ldots, w_k^{(S)}\right)$$

denote the current model after k block updates, where $w_k^{(s)}$ is the current value of the s^{th} stage/block. RPD first chooses a delay vector

$$j_k = \left(j_k^{(1)}, \ldots, j_k^{(S)}\right)$$

using only randomness available before the current active stage and mini-batch are sampled, with

$$0 \le k - j_k^{(s)} \le \delta$$

for all s. It then forms a stale model

$$z_k = \left(w_{j_k^{(1)}}^{(1)}, \ldots, w_{j_k^{(S)}}^{(S)}\right),$$

where $j_k^{(s)}$ is the past global iteration whose stage-s block is used in the stale read. Conditional on this stale read, RPD samples a pair (s_k, m_k) uniformly from $\{1, \ldots, S\} \times \{1, \ldots, M\}$ and updates only the active block:

$$w_{k+1}^{(s_k)} = w_k^{(s_k)} - \gamma \nabla_{w^{(s_k)}} f_{m_k}(z_k), \qquad w_{k+1}^{(i)} = w_k^{(i)} \quad \forall i \neq s_k. \tag{2}$$

Here δ is a tunable staleness bound. Unlike original PD, which visits stages and microbatches in a highly structured order, RPD randomizes both. This is the key simplification that makes a rigorous analysis possible. The resulting randomized version of PD is summarized in Algorithm 1.

Algorithm 1 Randomized PipeDream (RPD)

1: Input: stepsize $\gamma > 0$, number of stages S, delay bound δ, initial point $w_0 \in \mathbb{R}^d$
2: Initialize: $k = 0$, $j_k = (j_k^{(1)}, \dots, j_k^{(S)}) = (0, \dots, 0) \in \mathbb{Z}^S$.
3: **for** $k = 0, 1, 2, \dots$ **do**
4: Select a delay vector: $j_k = (j_k^{(1)}, \dots, j_k^{(S)})$, with $0 \le k - j_k^{(i)} \le \delta \quad \forall i$
5: Form a stale model: $z_k := \left(w_{j_k^{(1)}}^{(1)}, \dots, w_{j_k^{(S)}}^{(S)}\right)$
6: Sample (s_k, m_k) uniformly from $\{1, \dots, S\} \times \{1, \dots, M\}$, cond. independently of z_k.
7: Update the active stage block: $w_{k+1}^{(s_k)} = w_k^{(s_k)} - \gamma \nabla_{w^{(s_k)}} f_{m_k}(z_k)$
8: Keep the remaining blocks unchanged: $w_{k+1}^{(i)} = w_k^{(i)} \qquad \forall i \neq s_k$
9: **end for**

We now analyze Algorithm 1 as a randomized stale block-SGD method. Let $\Delta_0 := f(w_0) - f_\star$. Two elementary facts drive the proof. First, because (s_k, m_k) is sampled uniformly after the stale read is fixed, the randomized block update is an unbiased estimate of the full gradient direction up to the factor $1/S$. Second, trajectory-bounded block gradients and bounded delay imply that the stale read cannot be too far from the current iterate: $\|w_k - z_k\| \le \gamma\delta G$. Combining these facts with L-smoothness gives the following convergence guarantee.

Theorem 4.1 (Convergence of RPD).

Consider (1) *and the* RPD *update* (2)*. Suppose that, at each iteration, the delay vector is fixed before sampling* (s_k, m_k)*, and that conditional on the past and this delay vector,* (s_k, m_k) *is sampled uniformly. Suppose further that*

(i) f *is lower bounded,*

(ii) L*-smooth,*

(iii) there exists $G \ge 0$ *such that the block gradients along the stale-read trajectory satisfy*

$$\|\nabla_{w^{(s)}} f_m(z_k)\| \le G$$

for all k, m, s *almost surely, and*

(iv) the stage-wise staleness is bounded by $\delta \ge 0$*.*

If the stepsize satisfies $0 < \gamma \le 1/L$*, then for every* $K \ge 1$*, the iterates of* RPD *satisfy*

$$\frac{1}{K} \sum_{k=0}^{K-1} \mathbf{E}\left[\|\nabla f(w_k)\|^2\right] \le \frac{2S\Delta_0}{\gamma K} \; + \; \gamma SLG^2 \; + \; \gamma^2 L^2 \delta^2 G^2. \tag{3}$$

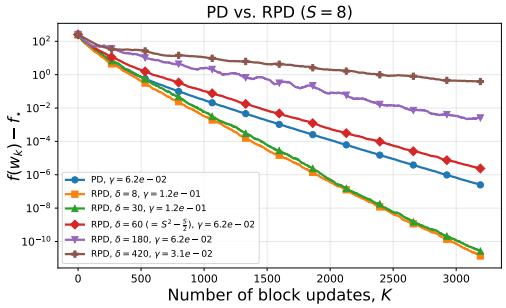
The proof is given in Appendix B. The bound separates three effects. The first term is the usual optimization term and decreases with the number of block updates K. The second term is the stochastic block-update price: at each iteration, RPD updates only one of the S blocks. The third term is the stale-read penalty. Its dependence on δ^2 is the key structural fact that will later explain the scaling behavior of PD.

For later comparison, it is useful to tune the stepsize in (3). If we choose

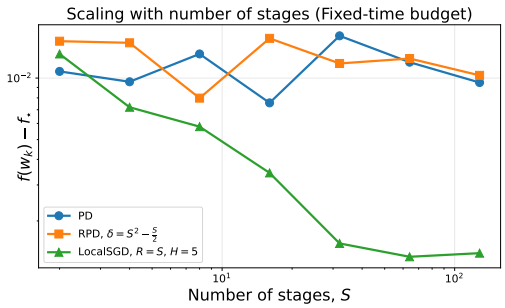
$$\gamma = \min\left\{\frac{1}{L}, \sqrt{\frac{2\Delta_0}{LG^2K}}\right\},$$

then the RPD rate becomes

$$\frac{1}{K} \sum_{k=0}^{K-1} \mathbf{E}\left[\|\nabla f(w_k)\|^2\right] = \mathcal{O}\left(\frac{S}{\sqrt{K}} + \frac{\delta^2}{K}\right).$$



(a) Random quadratic objective: best-tuned PD and RPD trajectories. $S = 8$, batch size is 10, $M = 60$, trained for 5 epochs.



(b) Logistic regression: final objective versus the number of stages S under a fixed simulator-time budget. Batch size is 10, $M = 60$, trained for 5 epochs. $H = 5$, $\lambda = 10^{-4}$.

Figure 2: The left plot validates RPD as a theoretical proxy for PD: when RPD is instantiated in the delay regime predicted for steady-state 1F1B execution, its trajectory closely matches that of PD on the quadratic objective. The right plot studies scaling on logistic regression: for each method and each stage count S, we measure the final objective reached under the same fixed simulator-time budget. $d = 512$.

From RPD back to PD. The point of RPD is not to claim that the deterministic PD scheduler samples stages and mini-batches uniformly. Rather, RPD is a deliberately simplified stale block-SGD model: it preserves the feature of PD that matters for the analysis—gradients are computed on stage-wise stale models—while replacing the deterministic 1F1B combinatorics by uniform sampling.

The link back to the original 1F1B method is captured by the next result.

Proposition 4.1 (Steady-state PD delay).

Embed a steady-state S-stage PD 1F1B *execution into the global-history model of* RPD*. Under the slot-batched global-history embedding used by our simulator, the worst-case delay satisfies*

$$\delta_{\mathrm{PD}} := S^2 - \frac{S}{2} + O(1). \tag{4}$$

For even S, this becomes the exact identity

$$\delta_{\mathrm{PD}} = S^2 - \frac{S}{2}.$$

The proof is given in Appendix C. This proposition identifies the delay parameter that should be used when RPD is used as a proxy for the original deterministic PD schedule. In particular, although RPD randomizes the active stage and mini-batch, the worst-case stage-wise staleness produced by steady-state 1F1B PD scales as $\delta_{\mathrm{PD}} = S^2 - S/2 + O(1)$. Under the slot-batched convention, each backward operation in a simulator slot appends one block update to the global history. Thus, to model the delay scale of real PD within the RPD abstraction, we should instantiate RPD with $\delta = \delta_{\mathrm{PD}}$.

Thus Theorem 4.1 should be read as a convergence theorem for a randomized PD-style abstraction, not as a direct theorem for deterministic PD. The role of Proposition 4.1 is to transfer the scheduler-induced delay scale of PD into this abstraction.

5 Scaling Limits of PipeDream

By Proposition 4.1, steady-state PD induces $\delta = \delta_{\mathrm{PD}} = \Theta(S^2)$. Plugging this into Equation (3) yields the staleness contribution $\Theta(\gamma^2 S^4)$, or $\Theta(S^4/K)$ in the tuned-rate form. Thus the stale-read part of the worst-case theorem has quartic dependence on the number of pipeline stages.

This does *not* mean that throughput scaling disappears; PD still reduces idle time very effectively. The point is instead that better utilization can come with a rapidly growing optimization penalty. Under a fixed computational budget, i.e., a fixed number of block updates or iterations, increasing the number of stages makes the final objective decrease more slowly. Reaching a target solution quality therefore requires substantially more optimization steps as S grows.

6 LocalSGD as an Alternative

The scaling diagnosis above suggests that the central weakness of PD is not pipelining itself, but the particular way in which PD buys utilization: it keeps stages busy by allowing different parts of the model to participate in gradients computed from old stage-local versions. A natural alternative is to recover consistency through occasional synchronization. This leads to LocalSGD: instead of maintaining one stale pipeline trajectory, we maintain several full-model trajectories, execute them through the same pipeline stages, and periodically average them.

In standard LocalSGD, each replica stores a full parameter vector, performs several local SGD steps, and then synchronizes with the other replicas by model averaging. In our pipeline-parallel setting, each full replica is itself partitioned across the same S stages used by the pipeline. We write

$$w_{q,r} = \left(w_{q,r}^{(1)}, \ldots, w_{q,r}^{(S)}\right) \in \mathbb{R}^d, \qquad w_{q,r}^{(s)} \in \mathbb{R}^{d_s},$$

for the logical model of replica r after q local steps. Here s indexes the pipeline stage, r indexes the replica, and q indexes local full-model steps performed by each replica.

This model uses additional replica memory: LocalSGD stores R logical copies of the model, partitioned across the stages, whereas the basic PD simulator follows a single stale trajectory with weight stashing. Our theory and plots compare optimization progress under matched block-update or simulated-time budgets; they do not charge this extra memory as a separate resource cost.

A logical LocalSGD step of replica r is realized by a full forward/backward pass through the S-stage pipeline. During this pass, each stage updates its own block $w_{q,r}^{(s)}$ when the corresponding backward computation is executed. After every H local steps, the replicas are synchronized by stage-wise averaging:

$$\bar{w}_q^{(s)} = \frac{1}{R}\sum_{r=1}^{R} w_{q,r}^{(s)}, \qquad w_{q,r}^{(s)} \leftarrow \bar{w}_q^{(s)} \qquad \forall r \in \{1, \ldots, R\},\ s \in \{1, \ldots, S\}.$$

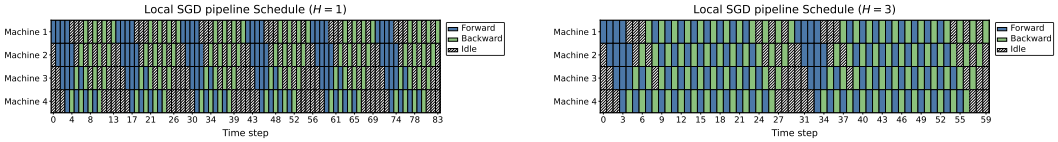
The resulting LocalSGD algorithm is summarized in Algorithm 2 located in Appendix D.

In all experiments, we choose $R = S$ in order to keep the pipeline highly utilized. The corresponding 1F1B timeline generation procedure, including the local-step dependencies and synchronization barriers, is given in Appendix E.2.

Relation to standard LocalSGD. Algorithm 2 is a stage-distributed version of the usual LocalSGD. At the optimization level, each replica performs H local stochastic-gradient steps and then the replicas are averaged. At the systems level, one such local step is not executed as one atomic full-vector operation: because the model is split across S stages, the forward and backward passes are scheduled through the pipeline, and each stage updates its own block when its backward computation is executed.

This distinction matters for the iteration count. For RPD and PD, one iteration means one stage/block update. We therefore let K denote the common block-update budget used in the theory comparison. For LocalSGD, we let T denote the number of logical full-model local SGD steps performed by each replica. Since one logical local step updates all S blocks of one replica, and there are R replicas, the common block-update budget is $K = RST$.

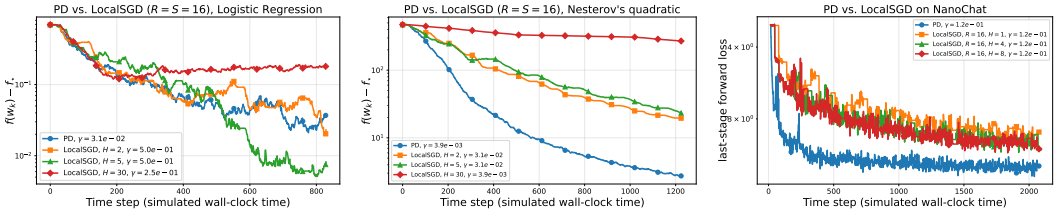
The parameter H is the main scheduling knob. If H is small, synchronization is frequent and the system incurs visible synchronization bubbles (Figure 3a). If H is large, synchronization becomes rare and the execution visually starts to resemble PD: machines spend most of their time computing, with only occasional global coordination (Figure 3b).



(a) $H = 1$, $R = S = 4$: frequent synchronization introduces visible bubbles.

(b) $H = 3$, $R = S = 4$: synchronization is sparse and the schedule approaches pipeline-style utilization.

Figure 3: Scheduling intuition for LocalSGD. Increasing the number of local steps H shrinks synchronization overhead but increases replica drift between averaging events.



(a) Logistic regression.

(b) Nesterov-style tridiagonal quadratic objective.

(c) NanoChat character language modeling.

Figure 4: PD versus LocalSGD at $S = 16$. Panels (a,b) use synthetic finite-sum objectives with $d = 512$, batch size 10, and $M = 60$; panel (c) uses an 11.1M-parameter NanoChat-style Transformer on Tiny Shakespeare with sequence length 128, batch size 32, embedding dimension 256, 8 attention heads, and tuned learning rates. PD is stronger on the quadratic and NanoChat tasks, while LocalSGD is stronger on logistic regression.

This trade-off is qualitatively different from the trade-off in PD. PD reduces hardware idle time by accepting stale gradients tied to old weight versions. LocalSGD avoids that particular staleness mechanism, but it pays for it with synchronization pauses and with drift between replicas.

Convergence analysis. We use the standard nonconvex LocalSGD analysis of Yu et al. (2019) in the homogeneous-worker case: each replica's local objective is a copy of f, so the worker count in Yu et al. (2019) corresponds to our R. Let $\bar{w}_q := \frac{1}{R}\sum_{r=1}^{R} w_{q,r}$ be the averaged logical model after q local steps per replica. Under the standard assumptions—f lower bounded and L-smooth, unbiased mini-batch gradients, and bounded gradient variance σ^2 and second moment B^2—choosing $R = S$ gives $T = K/S^2$ for the common block-update budget $K = RST$, and the standard LocalSGD guarantee ($0 < \gamma \leq 1/L$) becomes the bound reported in Table 1, whose three terms are the usual optimization term, the local-drift penalty from taking H unsynchronized local steps, and the stochastic-gradient variance term reduced by averaging across the S replicas. The full derivation, together with the substitution $T = K/(RS)$, is given in Appendix D. The LocalSGD guarantee is for the logical averaged model $\bar{w}_q$, whereas RPD/PD are measured along a single stale block-update trajectory; in contrast to that abstraction, the LocalSGD bound does not contain a stale-weight term of the form δ^2/K, its dominant additional error instead being the local-replica drift from taking H unsynchronized steps.

Which method is better in which regime? The two bounds in Table 1 should not be read as proving that one schedule is uniformly better than the other; instead, they expose two different failure modes under the common block-update budget K. For PD, substituting $\delta = \delta_{\mathrm{PD}} = \Theta(S^2)$ into the RPD bound gives a staleness contribution of $\Theta(S^4/K)$, so PD becomes increasingly sensitive to stale gradients as S grows. LocalSGD avoids this particular stale-version mechanism, but instead pays a local-drift term of order S^3H^2/K. The empirical question is therefore not whether LocalSGD always improves over PD, but which error source—staleness or drift—is more damaging for a given objective and scaling regime.

Method	Bound for block-update budget K	Tuned rate	Assumptions
RPD	$\frac{2S\Delta_0}{\gamma K} + \gamma SLG^2 + \gamma^2 L^2 \delta^2 G^2$	$\mathcal{O}\left(\frac{S}{\sqrt{K}} + \frac{\delta^2}{K}\right)$	L-smooth, lower bounded, trajectory-bounded block gradients, uniform stage-batch sampling after delay selection, delay $\leq \delta$, $0 < \gamma \leq 1/L$
RPD proxy for PD	$\frac{2S\Delta_0}{\gamma K} + \gamma SLG^2 + \gamma^2 L^2 \delta_{\mathrm{PD}}^2 G^2$	$\mathcal{O}\left(\frac{S}{\sqrt{K}} + \frac{S^4}{K}\right)$	same as RPD, with $\delta = \delta_{\mathrm{PD}}$
LocalSGD	$\frac{2S^2\Delta_0}{\gamma K} + 4\gamma^2 H^2 B^2 L^2 + \frac{L\gamma\sigma^2}{S}$	$\mathcal{O}\left(\sqrt{\frac{S}{K}} + \frac{S^3H^2}{K}\right)$	L-smooth, lower bounded, unbiased gradients, bounded variance and second moments, $0 < \gamma \leq 1/L$

Table 1: Comparison of rates for the studied optimization models. The PD row is obtained by inserting the steady-state PD delay scale $\delta = \delta_{\mathrm{PD}}$ into the RPD theorem. For RPD and PD, K directly counts stage/block updates. For LocalSGD, T denotes the number of logical full-model local SGD steps per replica, and the staged implementation satisfies $K = RST$.

Table 1 also clarifies the role of H: it trades a smaller drift penalty (small H, more synchronization barriers) for faster, pipeline-like execution (large H, quadratically larger drift; Figure 3), so we tune H empirically.

7 Experiments

Our experiments test whether RPD predicts deterministic 1F1B PD at $\delta = \delta_{\mathrm{PD}}$, whether the δ^2-driven degradation becomes visible as S grows, and whether the PD/LocalSGD tradeoff is objective-dependent, using three regimes—quadratic objectives, logistic regression, and an 11.1M-parameter NanoChat-style Transformer task (Karpathy, 2025; 2022)—with PD, RPD, and LocalSGD each tuned separately and PD/LocalSGD run via static 1F1B timelines in a discrete-event simulator (full setup in Appendix G).

PD versus RPD, and scaling with the number of stages. Figure 2 (left) shows that RPD closely tracks deterministic PD on the quadratic objective at $\delta = \delta_{\mathrm{PD}}$ (Proposition 4.1), supporting RPD as a tractable proxy for 1F1B's dominant optimization effect: stage-wise stale model versions. The right panel studies logistic regression under a fixed simulator-time budget while varying S—an empirical wall-clock-proxy experiment, since Theorem 4.1 compares methods under the block-update budget K rather than simulator time (block updates are matched across methods as detailed in Appendix G.6). Larger S gives RPD/PD more updates within the same horizon but also a larger delay scale; even so, stale-gradient pipeline methods deteriorate as S grows—consistent with $\delta_{\mathrm{PD}} = \Theta(S^2)$ and the δ^2 term in (3)—while LocalSGD scales more favorably.

PD versus LocalSGD and results summary. Figure 4 additionally compares PD and LocalSGD at $S = 16$ on these objectives and on NanoChat-style Tiny Shakespeare (Karpathy, 2025; 2022; 2015) (Appendix G): PD wins on the quadratic and NanoChat tasks, LocalSGD on logistic regression—so synchronization is not uniformly preferable.

8 Conclusion

PD is an optimization algorithm induced by a pipeline schedule whose active stages, microbatches, and stage-wise weight versions are tightly coupled. We introduced RPD, a randomized stale block-SGD proxy for PD, proved a nonconvex convergence guarantee, and showed the induced worst-case delay of real 1F1B PD scales as $\delta_{\mathrm{PD}} = S^2 - S/2 + O(1)$, exposing a $\Theta(\gamma^2 S^4)$ stale-read term ($\Theta(S^4/K)$ tuned) growing rapidly with S. Empirically, PD is stronger on quadratic/NanoChat tasks, LocalSGD on logistic regression, motivating direct 1F1B analysis of deterministic PD.

Ethics Statement

This work is theoretical and simulation-based. It does not involve human subjects, personal data, or proprietary/private datasets. The only dataset used, Tiny Shakespeare, is a standard openly licensed character-level language-modeling corpus (Section G.9), and the pretrained-model code we build on is released under the MIT License. We do not foresee direct harmful applications of this work beyond the general dual-use considerations common to optimization and systems research for large-scale model training.

Reproducibility Statement

Complete proofs of all theoretical results, together with the assumptions they rely on, are given in the appendix: the RPD convergence guarantee is proved in Section B, the delay-scaling result for PD is proved in Section C, and the LocalSGD rate is derived in Section D. The discrete-event 1F1B simulators used to generate the PD and LocalSGD timelines are described in Section E, and full experimental details, including objectives, hyperparameter grids, seeds, and compute resources, are given in Section G. Code for reproducing all experiments is included in the supplementary material.

AI Use Statement

We used generative AI tools (an LLM-based coding assistant) to help with code implementation of the experiments described in this paper, and to assist with editing/polishing the writing of the manuscript. We did not use generative AI tools to generate research ideas, theoretical results, or experimental findings. All AI-assisted code and text were reviewed and verified by the authors, who take full responsibility for the final content of this work.

References

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 431–445, 2021.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

Andrej Karpathy. char-rnn. `https://github.com/karpathy/char-rnn`, 2015. Tiny Shakespeare dataset source.

Andrej Karpathy. nanogpt. `https://github.com/karpathy/nanoGPT`, 2022. MIT License.

Andrej Karpathy. nanochat: The best chatgpt that $100 can buy. `https://github.com/karpathy/nanochat`, 2025. MIT License.

Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. *Advances in neural information processing systems*, 28, 2015.

Tao Lin, Sebastian U Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large mini-batches, use local sgd. *arXiv preprint arXiv:1808.07217*, 2018.

Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *SIAM Journal on Optimization*, 27(4):2202–2229, 2017.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pp. 1–15, 2019.

Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pp. 7937–7947. PMLR, 2021.

Yu Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pp. 1–16. IEEE, 2020.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011.

Peter Richtárik and Martin Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *Mathematical Programming*, 144(1):1–38, 2014.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Sebastian U Stich. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

Yangyang Xu and Wotao Yin. Block stochastic gradient iteration for convex and nonconvex optimization. *SIAM Journal on Optimization*, 25(3):1686–1716, 2015.

Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pp. 5693–5700, 2019.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. In *International conference on machine learning*, pp. 4120–4129. PMLR, 2017.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized stochastic gradient descent. *Advances in neural information processing systems*, 23, 2010.

A TABLE OF FREQUENTLY USED NOTATION

Table 2: Notation table

d	–	Total dimension of the model parameters.
M	–	Number of finite-sum components, corresponding to data mini-batches in our experiments.
S	–	Number of pipeline stages (or partitioned blocks of the model).
d_s	–	Dimension of the s^{th} block of the model, such that $\sum_{s=1}^{S} d_s = d$.
w	–	Full model parameter vector, $w \in \mathbb{R}^d$.
$w^{(s)}$	–	The s^{th} block of the model weights, corresponding to the s^{th} stage, $w^{(s)} \in \mathbb{R}^{d_s}$.
$f(w)$	–	Global objective function to minimize.
$f_m(w)$	–	Loss function evaluated on the m^{th} data batch.
$f_\star$	–	Global lower bound of the objective function f.
Δ_0	$\coloneqq$	Initial objective gap; for RPD/PD, $\Delta_0 = f(w_0) - f_\star$, and for LocalSGD, $\Delta_0 = f(\bar{w}_0) - f_\star$.
k	–	Global iteration counter (for block updates).
K	–	Block-update budget. For RPD and PD, K counts stage/block updates directly. For LocalSGD, $K = RST$, where T is the number of logical local steps per replica.
s_k, m_k	–	Active stage index and batch index sampled uniformly at iteration k, after the delay vector is selected.
$j_k^{(s)}$	–	The global history index whose stage-s block is used in the stale read at iteration k.
z_k	$\coloneqq$	$\big(w^{(1)}_{j_k^{(1)}}, \dots, w^{(S)}_{j_k^{(S)}}\big)$ – Stale model read at iteration k.
$\widehat{z}_k$	–	Extrapolated stale model used in the NAG-RPD.
δ	–	Upper bound on the stage-wise staleness/delay, such that $k - j_k^{(s)} \le \delta$.
δ_{PD}	–	Exact worst-case global-history delay of the steady-state PD 1F1B schedule.
$\delta_k^{(s)}$	–	Exact global-history delay of stale block s at backward-update event k for a PD 1F1B schedule.
a_k	–	Active backward stage at the k^{th} PD backward-update event.
$\bar{\delta}$	–	Average global-history delay over the steady-state execution for a PD 1F1B schedule.
γ	–	Stepsize (learning rate).
L	–	Smoothness constant of the objective function f.
G	–	Trajectory bound on the norm of the block gradients used in the analysis.
μ	–	Strong convexity or Polyak–Łojasiewicz (PL) condition parameter.
α	–	Prediction/momentum parameter used in NAG-RPD.
U_s	–	Canonical embedding matrix from the block space $\mathbb{R}^{d_s}$ into the full space $\mathbb{R}^d$.
g_k	$\coloneqq$	$U_{s_k} \nabla_{w^{(s_k)}} f_{m_k}(z_k)$ – Full-space update vector at iteration k.
R	–	Number of independent replicas in LocalSGD; in the experiments we use $R = S$.
T	–	Number of logical local SGD steps performed by each replica. Block-update budget $K = RST$.
H	–	Synchronization period in LocalSGD; replicas are averaged every H local steps.
q	–	Logical local-step index for LocalSGD.
r	–	Replica index, $r \in \{1, \dots, R\}$.
$m_{q,r}$	–	Mini-batch/component index sampled by replica r at local step q.
$w_{q,r}$	–	Full model of replica r after q local steps.
$w_{q,r}^{(s)}$	–	Stage-s block of replica r after q local steps.
$\bar{w}_q$	$\coloneqq$	$\frac{1}{R}\sum_{r=1}^{R} w_{q,r}$ – Averaged logical model after q local steps per replica.
B	–	Uniform second-moment bound for stochastic gradients in the LocalSGD analysis.
σ^2	–	Uniform variance bound for stochastic gradients in the LocalSGD analysis.

B CONVERGENCE OF RPD

We will analyze RPD (Algorithm 1) under a random pair sampling assumption. The original PD schedule is deterministic and couples stage order, mini-batch order, and delay vectors through the 1F1B timeline. In this appendix we analyze the randomized abstraction used in the main text, where the delay vector is selected first and then the active stage and mini-batch are sampled uniformly. This abstraction is designed to isolate the effect of bounded stale block gradients while avoiding the combinatorial complexity of the exact scheduler.

We assume that the delay vector used at iteration k is chosen before the current active stage and mini-batch. We then sample the pair

$$(s_k, m_k) \in \{1, \dots, S\} \times \{1, \dots, M\}$$

randomly.

Assumption B.1 (Uniform random pair sampling after delay selection)**.** *Let $\mathcal{F}_k$ denote the sigma-field generated by all randomness before the current stage/mini-batch draw, including the current delay vector j_k and therefore the stale point z_k. For every $k \geq 0$, conditioned on $\mathcal{F}_k$, the pair (s_k, m_k) is sampled uniformly from*

$$\{1, \dots, S\} \times \{1, \dots, M\},$$

and is independent of all previous randomness not already contained in $\mathcal{F}_k$.

Assumption B.2 (Bounded staleness)**.** *There exists an integer $\delta \geq 0$ such that for every $k \geq 0$,*

$$z_k = \big(w^{(1)}_{j_k^{(1)}}, \dots, w^{(S)}_{j_k^{(S)}}\big), \qquad j_k^{(s)} \leq k, \qquad k - j_k^{(s)} \leq \delta \quad \forall s \in \{1, \dots, S\}.$$

B.1 ASSUMPTIONS ON THE PROBLEM

Assumption B.3 (Lower boundedness)**.** *The function f is bounded from below:*

$$f(w) \geq f_\star \qquad \forall w \in \mathbb{R}^d.$$

Assumption B.4 (L-smoothness)**.** *The function $f : \mathbb{R}^d \to \mathbb{R}$ is differentiable and L-smooth:*

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \qquad \forall x, y \in \mathbb{R}^d.$$

Assumption B.5 (Trajectory-bounded block gradients)**.** *There exists $G \geq 0$ such that*

$$\|\nabla_{w^{(s)}} f_m(z_k)\| \leq G \qquad \forall k \geq 0,\ \forall s \in \{1, \dots, S\},\ \forall m \in \{1, \dots, M\},$$

almost surely along the trajectory generated by the algorithm.

B.2 ANALYSIS

Recall that the RPD update is

$$w_{k+1}^{(s_k)} = w_k^{(s_k)} - \gamma\, \nabla_{w^{(s_k)}} f_{m_k}(z_k), \qquad w_{k+1}^{(r)} = w_k^{(r)} \ \ \forall r \neq s_k.$$

For convenience, define the full-space update vector

$$g_k \coloneqq U_{s_k} \nabla_{w^{(s_k)}} f_{m_k}(z_k),$$

where $U_s : \mathbb{R}^{d_s} \to \mathbb{R}^d$ is the canonical embedding of block s into the full space. Then the update can be written compactly as

$$w_{k+1} = w_k - \gamma g_k.$$

Lemma B.1 (Unbiasedness of the RPD update).

Let Assumption B.1 (uniform random pair sampling) hold. Then

$$\mathbb{E}\left[Sg_k \mid \mathcal{F}_k\right] = \nabla f(z_k),$$

where $\mathcal{F}_k$ is the sigma-field in Assumption B.1.

Proof. Conditioned on $\mathcal{F}_k$, the stale point z_k is fixed. Hence

$$\begin{aligned}
\mathbb{E}\left[g_k \mid \mathcal{F}_k\right] &= \frac{1}{SM}\sum_{s=1}^{S}\sum_{m=1}^{M} U_s \nabla_{w^{(s)}} f_m(z_k) \\
&= \frac{1}{S}\sum_{s=1}^{S} U_s \left(\frac{1}{M}\sum_{m=1}^{M} \nabla_{w^{(s)}} f_m(z_k)\right) \\
&= \frac{1}{S}\sum_{s=1}^{S} U_s \nabla_{w^{(s)}} f(z_k) = \frac{1}{S}\nabla f(z_k).
\end{aligned}$$

□

Lemma B.2 (Distance between current and stale models).

Let Assumption B.2 (δ-bounded staleness) and Assumption B.5 (trajectory-bounded block gradients) hold. Then for every $k \geq 0$,

$$\|w_k - z_k\| \leq \gamma\delta G.$$

Proof. At each iteration, only one block is updated, so

$$\|w_{t+1} - w_t\| = \gamma\|g_t\| = \gamma\|\nabla_{w^{(s_t)}} f_{m_t}(z_t)\| \leq \gamma G.$$

Now z_k is assembled from block values that are at most δ iterations old. Hence $w_k - z_k$ consists only of block increments accumulated during the last δ iterations. Therefore

$$\|w_k - z_k\| \leq \sum_{t=k-\delta}^{k-1} \|w_{t+1} - w_t\| \leq \sum_{t=k-\delta}^{k-1} \gamma G = \gamma\delta G.$$

□

We can now proceed to the main theorem.

Theorem B.1 (Convergence of RPD).

Let Assumption B.1 (uniform random pair sampling after delay selection), Assumption B.3 (lower boundedness), Assumption B.4 (L-smoothness), Assumption B.2 (δ-bounded staleness), Assumption B.5 (trajectory-bounded block gradients) hold. Assume

$$0 < \gamma \leq \frac{1}{L}.$$

Then for every integer $K \geq 1$,

$$\frac{1}{K}\sum_{k=0}^{K-1} \mathbb{E}\left[\|\nabla f(w_k)\|^2\right] \leq \frac{2S\big(f(w_0) - f_\star\big)}{\gamma K} + \gamma S L G^2 + \gamma^2 L^2 \delta^2 G^2. \tag{5}$$

Proof. By L-smoothness of f,

$$f(w_{k+1}) \leq f(w_k) + \langle \nabla f(w_k), w_{k+1} - w_k \rangle + \frac{L}{2}\|w_{k+1} - w_k\|^2.$$

Since $w_{k+1} = w_k - \gamma g_k$, this becomes

$$f(w_{k+1}) \leq f(w_k) - \gamma \langle \nabla f(w_k), g_k \rangle + \frac{\gamma^2 L}{2}\|g_k\|^2.$$

Taking conditional expectation with respect to $\mathcal{F}_k$, and using Lemma B.1,

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \leq f(w_k) - \frac{\gamma}{S}\langle \nabla f(w_k), \nabla f(z_k) \rangle + \frac{\gamma^2 L}{2}\mathbb{E}\left[\|g_k\|^2 \mid \mathcal{F}_k\right]. \tag{6}$$

In view of Assumption B.5 (trajectory-bounded block gradients), we have $\|g_k\| = \|\nabla_{w^{(s_k)}} f_{m_k}(z_k)\| \leq G$, and hence

$$\mathbb{E}\left[\|g_k\|^2 \mid \mathcal{F}_k\right] \leq G^2.$$

Plugging this estimate into (6), we get

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \leq f(w_k) - \frac{\gamma}{S}\langle \nabla f(w_k), \nabla f(z_k) \rangle + \frac{\gamma^2 L G^2}{2}. \tag{7}$$

Let us now use the elementary inequality

$$\langle a, b \rangle = \frac{1}{2}\|a\|^2 + \frac{1}{2}\|b\|^2 - \frac{1}{2}\|a - b\|^2 \geq \frac{1}{2}\|a\|^2 - \frac{1}{2}\|a - b\|^2,$$

with $a = \nabla f(w_k)$ and $b = \nabla f(z_k)$ to estimate the inner product in (7). This gives

$$-\frac{\gamma}{S}\langle \nabla f(w_k), \nabla f(z_k) \rangle \leq -\frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{\gamma}{2S}\|\nabla f(w_k) - \nabla f(z_k)\|^2.$$

Plugging this estimate into (7), we get

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \leq f(w_k) - \frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{\gamma}{2S}\|\nabla f(w_k) - \nabla f(z_k)\|^2 + \frac{\gamma^2 L G^2}{2}. \tag{8}$$

Since f is L-smooth, we know that $\|\nabla f(w_k) - \nabla f(z_k)\| \leq L\|w_k - z_k\|$. Further, using Lemma B.2, we get $\|w_k - z_k\| \leq \gamma\delta G$. Therefore,

$$\|\nabla f(w_k) - \nabla f(z_k)\|^2 \leq \gamma^2 A, \qquad A := L^2\delta^2 G^2.$$

Substituting this into (8) yields

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \leq f(w_k) - \frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{\gamma^3 A}{2S} + \frac{\gamma^2 L G^2}{2}.$$

Taking expectation and applying the tower property, we get

$$\mathbb{E}\left[f(w_{k+1})\right] \leq \mathbb{E}\left[f(w_k)\right] - \frac{\gamma}{2S}\mathbb{E}\left[\|\nabla f(w_k)\|^2\right] + \frac{\gamma^3 A}{2S} + \frac{\gamma^2 L G^2}{2}.$$

Summing from $k = 0$ to $K - 1$ gives

$$\mathbb{E}\left[f(w_K)\right] \leq f(w_0) - \frac{\gamma}{2S}\sum_{k=0}^{K-1}\mathbb{E}\left[\|\nabla f(w_k)\|^2\right] + K\frac{\gamma^3 A}{2S} + K\frac{\gamma^2 L G^2}{2}.$$

Since we know that $\mathbb{E}\left[f(w_K)\right] \geq f_\star$, we obtain

$$\frac{\gamma}{2S}\sum_{k=0}^{K-1}\mathbb{E}\left[\|\nabla f(w_k)\|^2\right] \leq f(w_0) - f_\star + K\frac{\gamma^3 A}{2S} + K\frac{\gamma^2 L G^2}{2}.$$

Multiply by $2S/(\gamma K)$ to get

$$\frac{1}{K}\sum_{k=0}^{K-1}\mathbb{E}\left[\|\nabla f(w_k)\|^2\right] \leq \frac{2S(f(w_0) - f_\star)}{\gamma K} + \gamma^2 A + \gamma S L G^2,$$

which establishes (5). □

Corollary B.1 (Stepsize choice).

Let the assumptions of Theorem B.1 hold, fix $K \geq 1$, and choose the stepsize

$$\gamma = \min\left\{\frac{1}{L}, \sqrt{\frac{2(f(w_0) - f_\star)}{LG^2K}}\right\}.$$

Then

$$\frac{1}{K}\sum_{k=0}^{K-1}\mathbb{E}\left[\|\nabla f(w_k)\|^2\right] = \mathcal{O}\left(\frac{\delta^2}{K} + \frac{S}{\sqrt{K}}\right).$$

For fixed S and fixed delay bound δ, this recovers the standard $\mathcal{O}(K^{-1/2})$ stationarity rate of gradient-type methods for smooth nonconvex optimization.

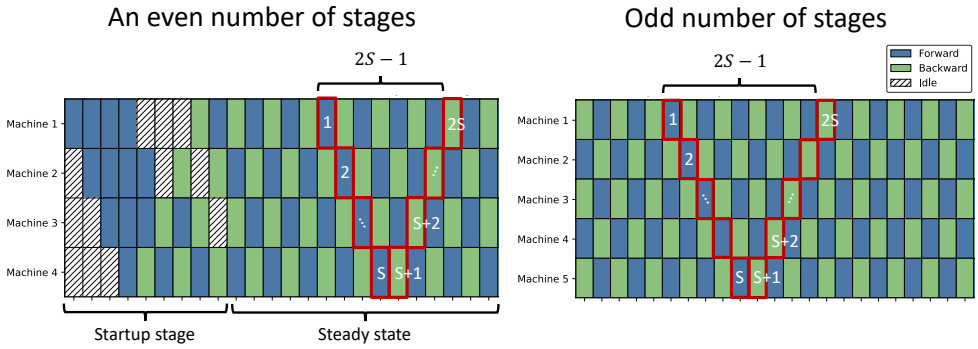


Figure 5: Steady-state counting argument for the exact global-history delay in PD 1F1B. The red staircase highlights the forward-to-backward traversal that determines a stale-read window. For a stale block from stage r used at an active backward stage a, the window length is $2S - r - a + 1$; the worst case is $r = a = 1$, with length $2S - 1$. In each steady-state pipeline slot, approximately half of the stages perform backward updates: exactly $S/2$ when S is even (left), and alternating between $(S-1)/2$ and $(S+1)/2$ when S is odd (right).

C Scaling of delay for PD

The global-history embedding used in this appendix is the same one used by the simulator. A pipeline slot is first generated as a row of simultaneous stage operations; every backward operation in that row then appends one block update to the global history in stage order. The counting below is exact for this slot-batched linearization. Changing only the within-slot ordering can move individual finite-S delays by lower-order terms, but it does not change the quadratic delay law that drives the convergence comparison.

Lemma C.1 (Steady-state scaling of the exact PD delay).

Consider a PD 1F1B pipeline with S stages, embedded into the global-history RPD model. For each steady-state backward update iteration k and each stage $s \in \{1, \dots, S\}$, let $\delta_k^{(s)}$ denote the exact global-history delay of stage s at iteration k. Define

$$\delta_{\mathrm{PD}} := \max_{k\in\{1,\dots,K\}} \max_{r\in\{1,\dots,S\}} \delta_k^{(r)}, \qquad \bar{\delta} := \frac{1}{KS} \sum_{k=1}^{K} \sum_{r=1}^{S} \delta_k^{(r)},$$

where K is the number of steady-state backward-update iterations under consideration.

Under the slot-counting convention used in the global-history embedding, the delay window for stale block r at active stage a_k has length

$$L_{r,a_k} = 2S - r - a_k + 1.$$

Consequently, over a full steady-state period,

$$\delta_{\mathrm{PD}} = S^2 - \frac{S}{2} + O(1), \qquad \bar{\delta} = \frac{S^2}{2} + O(1).$$

For even S, the leading formula is exact:

$$\delta_k^{(r)} = \frac{S}{2}\big(2S - r - a_k + 1\big), \qquad \delta_{\mathrm{PD}} = S^2 - \frac{S}{2}, \qquad \bar{\delta} = \frac{S^2}{2}.$$

Proof. We work in the global-history RPD embedding of PD. Thus, every backward update appends one new entry to the global history, and the delay of a stale block is the number of global-history

increments between the forward use of that block and the backward event that consumes the resulting stale model.

Fix a steady-state backward event whose active stage is $a \in \{1, \dots, S\}$, and fix a stale block index $r \in \{1, \dots, S\}$. The microbatch whose gradient is being applied at stage a saw block r during its forward pass on stage r. From that forward use, the microbatch must traverse the remaining forward stages $r+1, \dots, S$ and then traverse the backward path from stage S down to stage a. Including the active backward slot, this gives the window length

$$L_{r,a} = (S - r) + (S - a + 1) = 2S - r - a + 1.$$

The longest window occurs at $r = a = 1$ and has length $2S - 1$.

Let b_t denote the number of backward operations performed in steady-state pipeline slot t. Since the global history increases by one after each backward update, the delay is obtained by summing b_t over the corresponding window of length $L_{r,a}$.

Case 1: S even. In steady state, exactly half of the stages execute backward and half execute forward in every pipeline slot; see the left panel of Figure 5. Therefore $b_t = S/2$ for every steady-state slot t, and hence

$$\delta_k^{(r)} = \frac{S}{2} L_{r,a_k} = \frac{S}{2}\big(2S - r - a_k + 1\big).$$

Taking $r = 1$ and $a_k = 1$ gives

$$\delta_{\mathrm{PD}} = \frac{S}{2}(2S - 1) = S^2 - \frac{S}{2}.$$

To compute the average delay over a full steady-state period, note that each active stage appears equally often. Thus

$$\begin{aligned}\bar{\delta} &= \frac{1}{S^2} \sum_{a=1}^{S} \sum_{r=1}^{S} \frac{S}{2}\big(2S - r - a + 1\big) \\ &= \frac{S}{2}\left(2S + 1 - \frac{S+1}{2} - \frac{S+1}{2}\right) = \frac{S^2}{2}.\end{aligned}$$

Case 2: S odd. In steady state, the number of backward operations per slot alternates between

$$\left\lfloor \frac{S}{2} \right\rfloor = \frac{S-1}{2} \qquad \text{and} \qquad \left\lceil \frac{S}{2} \right\rceil = \frac{S+1}{2},$$

as shown in the right panel of Figure 5. Therefore, over any contiguous window of length L,

$$\sum_{t=1}^{L} b_t = \frac{S}{2} L + \varepsilon, \qquad |\varepsilon| \le \frac{1}{2}.$$

Applying this with $L = L_{r,a_k}$ yields

$$\delta_k^{(r)} = \frac{S}{2}\big(2S - r - a_k + 1\big) + \varepsilon_k^{(r)}, \quad |\varepsilon_k^{(r)}| \le \frac{1}{2}.$$

The maximum again occurs at $r = 1$ and $a_k = 1$, up to the bounded parity correction, so

$$\delta_{\mathrm{PD}} = S^2 - \frac{S}{2} + O(1).$$

Averaging the leading term over stale blocks and active stages gives $S^2/2$, and the averaged parity corrections remain bounded. Hence

$$\bar{\delta} = \frac{S^2}{2} + O(1).$$

Combining the two parity cases proves the result. □

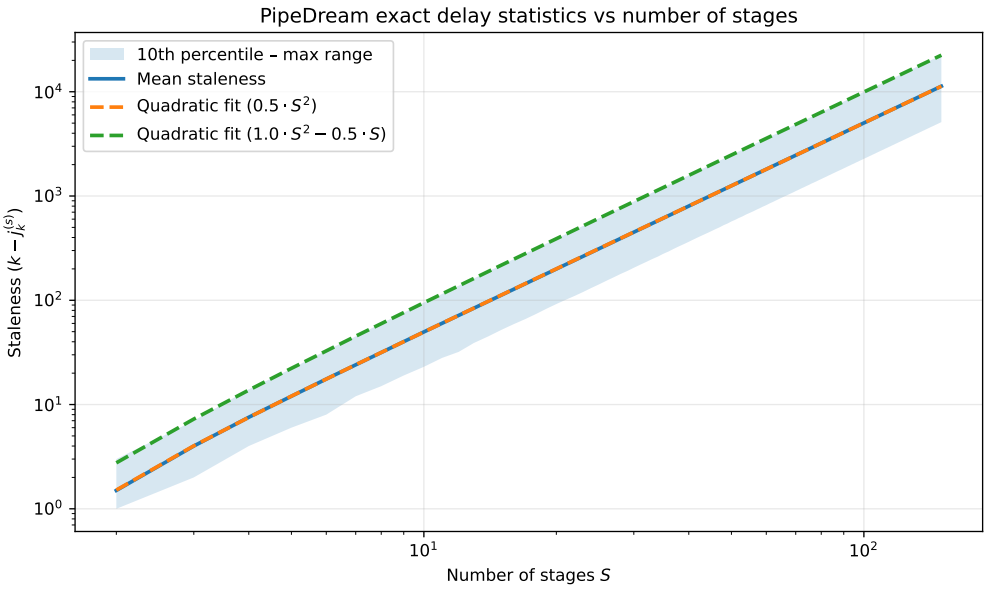


Figure 6: Steady-state statistics of the exact global-history delay $\delta_k^{(s)}$ induced by PD 1F1B as a function of the number of pipeline stages S. For each value of S, we generated the corresponding 1F1B timeline, reconstructed the exact delays $\delta_k^{(s)}$ directly from the timeline, discarded the first half of the resulting delay sequence to remove the startup transient, and then computed statistics on the remaining steady-state part. The solid blue curve shows the empirical mean delay $\bar{\delta}$, while the shaded region spans from the 10th percentile to the maximum delay. Both statistics exhibit quadratic growth in S. The orange dashed curve corresponds to the fit $\bar{\delta}(S) \approx \frac{1}{2}S^2$, while the green dashed curve corresponds to $\delta(S) \approx S^2 - \frac{1}{2}S$, where $\delta = \max_k \max_s \delta_k^{(s)}$.

Corollary C.1 (Quadratic scaling of PD staleness).

Consider PD 1F1B *embedded into the global-history* RPD *model. Let*

$$\delta_{\mathrm{PD}} := \max_{k\in\{1,\dots,K\}} \max_{s\in\{1,\dots,S\}} \delta_k^{(s)}, \qquad \bar{\delta} := \frac{1}{KS}\sum_{k=1}^{K}\sum_{s=1}^{S} \delta_k^{(s)},$$

where $\delta_k^{(s)}$ is the exact global-history delay of stale block s at backward-update event k, and K is the number of steady-state backward-update iterations under consideration. Then

$$\delta_{\mathrm{PD}} = S^2 - \frac{S}{2} + O(1), \qquad \bar{\delta} = \frac{S^2}{2} + O(1).$$

Proof. This follows directly from Lemma C.1. For even S, the identities

$$\delta_{\mathrm{PD}} = S^2 - \frac{S}{2}, \qquad \bar{\delta} = \frac{S^2}{2}$$

hold exactly. For odd S, Lemma C.1 gives

$$\delta_{\mathrm{PD}} = S^2 - \frac{S}{2} + O(1), \qquad \bar{\delta} = \frac{S^2}{2} + O(1).$$

Combining the two cases yields the claim. □

To validate the asymptotic scaling predicted by Lemma C.1, we computed the exact global-history delays induced by PD 1F1B over a wide range of pipeline depths. For each stage count S, we first

generated the corresponding 1F1B schedule and then reconstructed the exact delay matrix directly from the timeline. More precisely, for every steady-state backward-update iteration k and every stage s, we computed

$$\delta_k^{(s)} = k - j_k^{(s)},$$

where $j_k^{(s)}$ is the global history index of the snapshot used by the corresponding microbatch during its forward pass on stage s. Since our goal is to measure the steady-state behavior rather than the startup transient, we discarded the first half of the resulting exact-delay sequence and analyzed only the remaining part, where the pipeline has entered its periodic 1F1B regime.

Figure 6 reports the resulting statistics as functions of S. The solid blue curve plots the empirical average delay

$$\bar{\delta} = \frac{1}{KS} \sum_{k=1}^{K} \sum_{s=1}^{S} \delta_k^{(s)},$$

while the shaded region shows the spread from the 10th percentile to the largest observed delay. The two dashed curves correspond to the quadratic laws predicted by the steady-state counting argument:

$$\bar{\delta}(S) \approx \frac{S^2}{2} = \bar{\delta}, \qquad \delta(S) \approx S^2 - \frac{S}{2} = \delta_{\mathrm{PD}},$$

where

$$\delta_{\mathrm{PD}} = \max_{k \in \{1,\dots,K\}} \max_{s \in \{1,\dots,S\}} \delta_k^{(s)}.$$

The agreement is remarkably tight. In particular, the average delay grows as $\frac{1}{2}S^2$, while the worst-case delay grows as $S^2 - \frac{1}{2}S$, confirming that the exact global-history staleness induced by PD scales quadratically with the number of stages.

This observation is important for the RPD analysis. In our theoretical convergence bound, the staleness parameter δ appears explicitly, so a quadratic growth of δ with S implies a corresponding degradation of the theoretical guarantee as the pipeline depth increases. Thus, even though PD 1F1B achieves high hardware utilization, in the global-history RPD abstraction it induces a staleness regime that becomes progressively less favorable for optimization as S grows. This motivates the search for alternative scheduling policies that preserve good utilization while reducing the effective global-history delay.

D DERIVATION OF THE LOCALSGD RATE UNDER THE BLOCK-UPDATE BUDGET

We study the following variation of the LocalSGD algorithm for solving (1):

Algorithm 2 Stage-distributed LocalSGD

Input: stepsize $\gamma > 0$, number of stages S, number of replicas R, synchronization period H, initial model $w_0 = (w_0^{(1)}, \dots, w_0^{(S)})$
Initialize replicas: $w_{0,r}^{(s)} = w_0^{(s)} \qquad \forall r \in \{1, \dots, R\}, \quad \forall s \in \{1, \dots, S\}.$
for $q = 0, 1, \dots$ **do**
  for each replica $r = 1, \dots, R$ **do**
    Sample $m_{q,r} \in \{1, \dots, M\}$.
    Run a 1F1B pass for replica r on $m_{q,r}$ across S stages.
    Update replica r: $w_{q+1,r}^{(s)} = w_{q,r}^{(s)} - \gamma \nabla_{w^{(s)}} f_{m_{q,r}}(w_{q,r}) \qquad \forall s \in \{1, \dots, S\}.$
  end for
  if $q + 1$ is divisible by H **then**
    Average weights between replicas: $\bar{w}_{q+1}^{(s)} = \frac{1}{R} \sum_{r=1}^{R} w_{q+1,r}^{(s)} \qquad \forall s \in \{1, \dots, S\}.$
    Synchronize replicas: $w_{q+1,r}^{(s)} \leftarrow \bar{w}_{q+1}^{(s)} \qquad \forall r \in \{1, \dots, R\}, \quad \forall s \in \{1, \dots, S\}.$
  end if
end for

We use the standard nonconvex LocalSGD analysis of Yu et al. (2019) in the homogeneous-worker case. To avoid overloading the finite-sum components f_m from (1), denote the objective assigned to replica r in the standard LocalSGD formulation by ϕ_r. In our setting,

$$\phi_1 = \cdots = \phi_R = f,$$

where R is the number of replicas, i.e., the number of parallel local trajectories. Thus, the worker count in Yu et al. (2019) corresponds to our R, not to the finite-sum size M in (1).

Let

$$\bar{w}_q := \frac{1}{R} \sum_{r=1}^{R} w_{q,r}$$

be the averaged logical model after q local steps per replica. Assume that f is lower bounded by $f_\star$ and L-smooth. For uniformly sampled mini-batches $m_{q,r} \in \{1, \dots, M\}$, assume

$$\mathbf{E}\big[\nabla f_{m_{q,r}}(w_{q,r}) \mid w_{q,r}\big] = \nabla f(w_{q,r}),$$

and

$$\mathbf{E}\Big[\big\|\nabla f_{m_{q,r}}(w_{q,r}) - \nabla f(w_{q,r})\big\|^2\Big] \le \sigma^2, \qquad \mathbf{E}\Big[\big\|\nabla f_{m_{q,r}}(w_{q,r})\big\|^2\Big] \le B^2.$$

We use B for the second-moment bound in order to avoid confusion with the block-gradient bound G used in the RPD analysis.

The standard periodic-averaging LocalSGD guarantee gives that, for $0 < \gamma \le 1/L$ and T local steps per replica,

$$\frac{1}{T} \sum_{q=0}^{T-1} \mathbf{E}\big[\|\nabla f(\bar{w}_q)\|^2\big] \le \frac{2\big(f(\bar{w}_0) - f_\star\big)}{\gamma T} + 4\gamma^2 H^2 B^2 L^2 + \frac{L\gamma\sigma^2}{R}. \tag{9}$$

In our staged implementation, one logical local SGD step updates all S blocks of one replica. Since there are R replicas, T local steps per replica correspond to

$$K = RST$$

stage/block updates. Equivalently,

$$T = \frac{K}{RS}.$$

Substituting this relation into (9) gives the bound under the common block-update budget:

$$\frac{RS}{K}\sum_{q=0}^{K/(RS)-1}\mathbf{E}\big[\|\nabla f(\bar{w}_q)\|^2\big] \le \frac{2RS\big(f(\bar{w}_0)-f_\star\big)}{\gamma K} + 4\gamma^2 H^2 B^2 L^2 + \frac{L\gamma\sigma^2}{R}. \tag{10}$$

Using the stepsize choice from the corollary of Yu et al. (2019),

$$\gamma = \frac{\sqrt{R}}{L\sqrt{T}},$$

truncated at $1/L$ when necessary, balances the optimization term and the stochastic-variance term in (9) and yields

$$\frac{1}{T}\sum_{q=0}^{T-1}\mathbf{E}\big[\|\nabla f(\bar{w}_q)\|^2\big] = \mathcal{O}\left(\frac{1}{\sqrt{RT}} + \frac{RH^2}{T}\right).$$

Their linear-speedup corollary additionally requires $H \le T^{1/4}/R^{3/4}$ to make the local-drift contribution no larger than the $1/\sqrt{RT}$ term. We do not impose this restriction in the displayed comparison; instead we retain the explicit H-dependent term.

Finally, using $T = K/(RS)$, we obtain

$$\frac{1}{\sqrt{RT}} = \sqrt{\frac{S}{K}}, \qquad \frac{RH^2}{T} = \frac{R^2SH^2}{K}.$$

Therefore,

$$\frac{RS}{K}\sum_{q=0}^{K/(RS)-1}\mathbf{E}\big[\|\nabla f(\bar{w}_q)\|^2\big] = \mathcal{O}\left(\sqrt{\frac{S}{K}} + \frac{R^2SH^2}{K}\right). \tag{11}$$

For the experimental choice $R = S$, this reduces to

$$\frac{RS}{K}\sum_{q=0}^{K/(RS)-1}\mathbf{E}\big[\|\nabla f(\bar{w}_q)\|^2\big] = \mathcal{O}\left(\sqrt{\frac{S}{K}} + \frac{S^3H^2}{K}\right).$$

E DISCRETE 1F1B SCHEDULING SIMULATORS

This appendix describes the discrete-event schedulers used to generate the pipeline timelines in our experiments. The purpose of these schedulers is to separate the systems-level ordering of forward and backward computations from the optimization dynamics. We first generate a static timeline of pipeline actions and then replay this timeline during optimization. This makes the comparison between PD and LocalSGD independent of low-level runtime effects such as kernel launch overheads, communication latency, or device-specific implementation details.

We consider a straight pipeline with S stages. Time is discrete. At every time step, each stage can execute at most one action:

$$a_{t,s} \in \{\emptyset\} \cup \{F_m : m \in \mathcal{M}\} \cup \{B_m : m \in \mathcal{M}\},$$

where F_m denotes the forward computation of microbatch m, B_m denotes the corresponding backward computation, and $\emptyset$ denotes an idle slot. The complete schedule is therefore a timeline

$$\mathcal{T} = \big(a_{t,1}, \ldots, a_{t,S}\big)_{t \geq 0}.$$

In all schedulers, data movement is strictly causal: a computation that depends on another stage may start only at a later discrete time step, not in the same slot.

E.1 PD 1F1B SCHEDULE

The PD scheduler follows a one-forward-one-backward (1F1B) policy after pipeline startup. The input stage admits at most A active microbatches into the pipeline, where A is the number of outstanding active microbatches allowed by the schedule. In our experiments we use the standard one-machine-per- stage choice $A = S$, corresponding to the NOAM (number of outstanding active microbatches) parameter in the original PD implementation.

For each stage s and microbatch m, we store two completion times:

$$\tau^F_{s,m}, \qquad \tau^B_{s,m},$$

where $\tau^F_{s,m}$ is the time at which the forward computation of microbatch m finishes at stage s, and $\tau^B_{s,m}$ is the time at which its backward computation finishes at stage s. A value of -1 means that the corresponding operation has not yet been completed.

At time t, a forward operation F_m is ready at stage s if

$$\begin{cases} m \text{ is the next unlaunched microbatch,} & \text{if } s = 1, \\ \text{and the number of active microbatches is less than } A, & \\ \tau^F_{s-1,m} \neq -1 \text{ and } \tau^F_{s-1,m} < t, & \text{if } s > 1. \end{cases}$$

A backward operation B_m is ready at stage s if the forward computation has already completed at that stage and either s is the last stage or the backward computation has already completed at the next stage:

$$\tau^F_{s,m} \neq -1, \qquad \tau^F_{s,m} < t,$$

and

$$\begin{cases} \text{true}, & s = S, \\ \tau^B_{s+1,m} \neq -1 \text{ and } \tau^B_{s+1,m} < t, & s < S. \end{cases}$$

The strict inequalities enforce that activations and gradients cannot propagate through multiple stages in a single discrete slot.

Each stage begins in a startup phase. During startup, the stage prioritizes forward work until the first backward operation becomes ready. Once a backward operation becomes available, the stage enters steady state. In steady state, the stage alternates between backward and forward work whenever both are ready. If only one type of work is ready, it executes that available operation rather than idling.

Equivalently, each stage stores a local preference variable

$$p_s \in \{F, B\}.$$

Initially $p_s = F$. After executing a backward operation, the stage sets $p_s = F$; after executing a forward operation in steady state, it sets $p_s = B$. Thus the local decision rule is:

$$a_{t,s} = \begin{cases} B_m, & \text{if backward work is ready and either} \\ & \text{the stage is not yet in steady state,} \\ & p_s = B, \text{ or no forward work is ready,} \\ F_m, & \text{if forward work is ready,} \\ \emptyset, & \text{otherwise.} \end{cases}$$

After all stages have selected their actions for time t, the completion-time arrays are updated simultaneously. This simultaneous update is important: it prevents one stage from using information produced by another stage in the same time slot.

The active-microbatch counter is updated only at the input stage. A microbatch becomes active when its forward computation at stage 1 completes, and it stops being active when its backward computation at stage 1 completes. The schedule terminates when the final microbatch has completed its backward computation at stage 1.

E.2 LocalSGD 1F1B schedule

The LocalSGD scheduler uses the same stage-level 1F1B execution rule, but the admissible forward operations are constrained by the local-training and synchronization structure. Instead of one pipeline trajectory, we run R independent local trajectories, which we call runs or replicas. Each synchronization round consists of H local steps for each replica. Therefore one round contains

$$RH$$

pipeline microbatch jobs.

We index the scheduled pipeline jobs by a single integer $m = 0, 1, \ldots, N-1$. This index can be decoded as

$$q(m) = \left\lfloor \frac{m}{RH} \right\rfloor, \qquad \ell(m) = \left\lfloor \frac{m \bmod RH}{R} \right\rfloor, \qquad r(m) = m \bmod R,$$

where $q(m)$ is the synchronization round, $\ell(m) \in \{0, \ldots, H-1\}$ is the local step inside that round, and $r(m) \in \{0, \ldots, R-1\}$ is the replica index. Thus the microbatches in a round are ordered as

$$(r = 0, \ell = 0), \ldots, (r = R-1, \ell = 0),$$
$$(r = 0, \ell = 1), \ldots, (r = R-1, \ell = 1), \ldots, (r = R-1, \ell = H-1).$$

The readiness rule for backward operations is identical to the PD scheduler. The difference is in the readiness rule for forward operations. A forward operation F_m at stage s must satisfy three conditions.

First, the usual pipeline dependency must hold:

$$\begin{cases} \text{true}, & s = 1, \\ \tau^F_{s-1,m} \neq -1 \text{ and } \tau^F_{s-1,m} < t, & s > 1. \end{cases}$$

Second, if m is not the first local step of its replica inside the current round, then the previous local step of the same replica must already have completed its backward computation at the same stage. Since replicas are ordered fastest within each local-step group, the previous local step of the same replica has index $m - R$. Hence, for $\ell(m) > 0$, we require

$$\tau^B_{s,m-R} \neq -1, \qquad \tau^B_{s,m-R} < t.$$

This condition enforces that a stage does not start the next local step of a replica before its own block has been updated by the previous local step.

Third, if m belongs to a new synchronization round, then the previous round must be fully complete before any microbatch from the new round is launched. Let $q(m) > 0$. The last microbatch of the previous round is

$$m_{\text{last}}(q(m) - 1) = q(m)RH - 1.$$

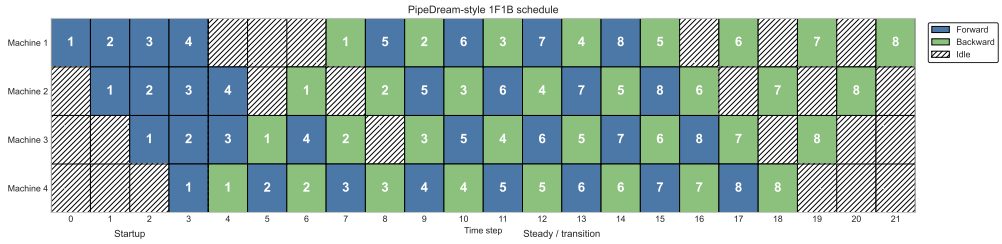


Figure 7: PD-style 1F1B schedule for a 4-stage pipeline with 8 microbatches. Rows correspond to pipeline stages (machines), and columns correspond to discrete time steps. Each numbered cell indicates the microbatch being processed at that stage and time. The figure illustrates how PD overlaps computation across stages to improve utilization after the initial fill phase.

Since the backward pass finishes at stage 1, it is sufficient to check

$$\tau^{B}_{1,\,q(m)RH-1} \neq -1, \qquad \tau^{B}_{1,\,q(m)RH-1} < t.$$

This implements the global model-averaging barrier between LocalSGD rounds. Only after all replicas have completed their H local steps is the averaged model broadcast and the next round allowed to begin.

Once the forward and backward readiness sets have been constructed, the same local 1F1B choice rule as in PD is applied at every stage. Thus LocalSGD differs from PD not in the local stage policy, but in the admissibility constraints for launching forward work. These constraints encode two optimization-level requirements: consecutive local steps of the same replica must respect the stage-local update order, and different synchronization rounds must be separated by a global averaging barrier.

E.3 Comparison of the two timelines

The two schedules share the same pipeline mechanics: forward computations move from stage 1 to stage S, backward computations move from stage S to stage 1, and each stage follows a 1F1B preference once backward work becomes available. The difference is the source of idle time.

In PD, idle time appears mainly during startup and drain. Once the pipeline is filled, the NOAM constraint allows many microbatches to remain active simultaneously, so the schedule has high utilization. The cost is optimization staleness: a backward computation may update a stage using weights and activations generated many pipeline slots earlier.

In LocalSGD, the schedule has additional idle regions at synchronization boundaries. No microbatch from round $q+1$ is allowed to start until all microbatches from round q have completed their backward pass at the input stage. This creates synchronization bubbles. However, the method avoids the same long cross-stage weight-version staleness mechanism that appears in PD; instead, its optimization error comes from local replica drift between averaging events.

This is the scheduling-level origin of the main tradeoff studied in the paper: PD maximizes pipeline utilization by accepting stale pipeline updates, whereas LocalSGD reduces stale-version effects by periodically paying for synchronization barriers.

F RELATIONSHIP BETWEEN PD AND RPD

To understand the connection between PD and our abstract RPD method (Algorithm 1), it is useful to distinguish between two notions of staleness that are easily conflated in implementation.

In PD, the model is partitioned into S stages, and each stage updates its own block of parameters asynchronously according to the pipeline schedule. When a microbatch performs its forward pass through stage s, the current weights of that stage are *stashed*, so that the corresponding backward pass later uses exactly the same stage-local weights. Thus, each microbatch carries a stage-wise record of the parameter values that it encountered during the forward pass. The resulting stale model seen by a given backward event is therefore not arbitrary: it is the collection of stage-wise weights that were actually used when that same microbatch traversed the pipeline on the forward pass.

By contrast, in RPD, the update rule is written in terms of a stale mixed model

$$z_k = \big(w^{(1)}_{j_k^{(1)}}, \dots, w^{(S)}_{j_k^{(S)}}\big),$$

where $j_k^{(s)} \le k$ specifies which historical version of block s is read at iteration k. The crucial observation is that RPD does not require all blocks to be read from the same past iterate. Different blocks may be taken from different history indices, which makes it expressive enough to represent stage-wise stale reads similar to those induced by pipeline parallelism.

Deterministic PD as a structured stale-block instance. Consider the sequence of backward-update events in PD, flattened in their actual execution order:

$$e_k = (s_k, m_k), \qquad k = 0, 1, 2, \dots,$$

where s_k is the stage updated at the k-th backward event and m_k is the corresponding microbatch. If, for each such event, one defines z_k to be exactly the collection of stage-wise weights that microbatch m_k saw on the forward pass in PD, then the resulting RPD update coincides with the PD update. In this sense, PD can be viewed as a structured special case of the stale mixed-model update template, in which the arriving stages, selected batches, and stale block reads are all determined by the pipeline schedule and by the forward-pass stashing mechanism.

This equivalence is at the level of the update template. It does not mean that deterministic PD satisfies the uniform random stage/mini-batch sampling assumption used in Theorem 4.1; that theorem applies to the randomized abstraction, and the deterministic scheduler enters the paper through the delay calculation.

This viewpoint is important because it clarifies what must be matched in order for RPD to reproduce PD exactly. Matching only the order of stage updates is not sufficient. Matching only the batch order is not sufficient. Even matching a simple stage-dependent delay pattern is not sufficient. Exact equivalence requires that, for each backward event, the stale model z_k contain the same stage-wise parameter values that the corresponding microbatch encountered on the forward pass in PD.

Global history versus local staleness. Our implementation maintains a global history

$$w_0, w_1, w_2, \dots,$$

where each new iterate is created after a single block update. Since only one block changes per update, the k-th history element stores the full parameter vector after the k-th global block update. This global history should be distinguished from the *local* version counters maintained by each stage in PD.

This distinction leads to two different notions of delay.

First, one may define the *local stage staleness* for microbatch m and stage s as

$$\Delta^{\mathrm{local}}(m, s) = v_s^{\mathrm{current}} - v_s^{\mathrm{forward}}(m),$$

where $v_s^{\mathrm{forward}}(m)$ is the local version of stage s used by microbatch m during its forward pass, and v_s^{current} is the current local version of stage s at the time of the backward update. This is the natural stage-local notion of staleness in PD.

Second, when representing PD inside RPD, one must work with a *global history delay*. For the backward event $e_k = (s_k, m_k)$, let

$$j_k^{(s)}$$

denote the global history index at which microbatch m_k saw stage s during its forward pass. Then the exact delay needed by RPD is

$$\delta_k^{(s)} = k - j_k^{(s)},$$

so that

$$w_{j_k^{(s)}}^{(s)} = w_{k-\delta_k^{(s)}}^{(s)}.$$

In other words, the delay used by RPD is measured with respect to the *global* sequence of block updates, not with respect to the local update count of stage s.

Why can the exact delays be large? A potentially confusing phenomenon is that the exact delay for a stage can be much larger than its local staleness, and can even be strictly positive for the final stage in steady state. This is not a contradiction. The reason is that the global history index advances after *every* block update, regardless of which stage was updated. Thus, even if a particular stage has not changed much locally, many global history entries may have been inserted by updates to other stages.

For example, suppose a microbatch reaches the final stage on the forward pass, and before its corresponding backward event occurs, several other stages perform backward updates. Even if the final-stage weights themselves are nearly unchanged in a local sense, the required forward-time snapshot for that stage may now lie several entries back in the global history. Consequently, the exact global delay can be larger than zero, and in practice may be substantially larger than the number of local updates experienced by that stage.

This explains an empirical phenomenon observed in our experiments: in the steady regime, the exact delay vectors extracted from PD may contain relatively large entries, even for stages that are intuitively "fresh." Such values should not be interpreted as large local staleness of that stage; rather, they represent how far one must rewind the *global* history in order to recover the exact forward-time block value used by the corresponding microbatch.

Steady-state structure. Once the pipeline reaches steady state, the exact delay vectors induced by PD exhibit a periodic pattern. This is a direct consequence of the deterministic structure of the 1F1B schedule. For a fixed number of stages, the sequence of backward-update events repeats cyclically, and therefore the associated exact stale-read vectors repeat as well. This steady-state periodicity is useful both conceptually and practically: conceptually, it shows that pipeline-induced staleness is highly structured rather than arbitrary; practically, it provides a way to construct RPD instances that reproduce PD exactly or approximately by feeding in the corresponding schedule-induced batch order, stage order, and exact delay vectors.

Practical consequence. These observations suggest three levels of approximation when relating RPD to PD. At the coarsest level, one may use simple synthetic delay rules, such as stage-dependent offsets, to obtain a rough pipeline-inspired stale model. At an intermediate level, one may match the stage-update order and batch-selection order to those of PD. At the finest level, one may additionally use the exact delay vectors extracted from a PD run, in which case RPD reproduces the same stale model seen by each backward event. This hierarchy provides a principled way to move from a mathematically convenient abstraction toward a faithful simulation of the original pipeline method.

G EXPERIMENTS AND SETUP

G.1 COMPUTE RESOURCES

We report the compute resources used to run the experiments in this paper. The synthetic experiments, including the quadratic objectives and logistic-regression objectives, were run on a MacBook Pro equipped with an Apple M1 Max chip and 64GB of RAM. These experiments were implemented in Python 3.12.

For the larger LLM experiments, we used a single NVIDIA A100 GPU with 40GB of VRAM. These experiments were also implemented in Python 3.12.

The synthetic experiments are lightweight and can be reproduced on a modern laptop or workstation with sufficient memory. The LLM experiments require GPU acceleration; in our setup, one NVIDIA A100 40GB GPU was sufficient.

G.2 OBJECTIVES

Experimental objectives. We use synthetic finite-sum objectives of the form (1). In all cases, the parameter vector $w \in \mathbb{R}^d$ is partitioned into S contiguous blocks, corresponding to the S pipeline stages. The data are split into mini-batches, and each f_m is the loss evaluated on the m^{th} mini-batch.

Quadratic objective. Our first benchmark is a least-squares quadratic objective. We generate $X \in \mathbb{R}^{N\times d}$, a ground-truth vector $w^\star \in \mathbb{R}^d$, and targets

$$y = Xw^\star$$

with optional additive label noise. The objective is

$$f(w) = \frac{1}{2N}\|Xw - y\|_2^2.$$

Equivalently, this is a quadratic with Hessian

$$A = \frac{1}{N}X^\top X.$$

We use two variants. In the random quadratic, X has i.i.d. standard Gaussian entries and $w^\star$ is also sampled from a standard Gaussian distribution. In the tridiagonal quadratic, we construct a Nesterov-style tridiagonal matrix

$$T = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \\ & & -1 & 2 \end{pmatrix},$$

add a diagonal shift $A = T + \mu I$, and then construct X so that

$$\frac{1}{N}X^\top X = A$$

holds exactly. When a target condition number is specified, μ is chosen so that the resulting matrix A has that condition number; otherwise we use the default shift from the implementation. The optimal value $f_\star$ for this objective is computed by solving the corresponding least-squares problem.

The PD/RPD validation plot in Figure 2a uses the random quadratic variant. The quadratic panel in Figure 4 uses the Nesterov-style tridiagonal quadratic variant.

Logistic-regression objective. Our second benchmark is binary logistic regression. We sample $X \in \mathbb{R}^{N\times d}$ with i.i.d. standard Gaussian entries and generate a ground-truth vector $w^\star \in \mathbb{R}^d$. Labels are sampled according to

$$\mathbb{P}(y_i = 1 \mid x_i) = \sigma(x_i^\top w^\star), \qquad \sigma(t) = \frac{1}{1+\exp(-t)}.$$

The optimized objective is the regularized binary cross-entropy

$$f(w) = \frac{1}{N}\sum_{i=1}^{N}\left[\max(x_i^\top w, 0) - y_i x_i^\top w + \log\bigl(1+\exp(-|x_i^\top w|)\bigr)\right] + \frac{\lambda}{2}\|w\|_2^2,$$

where $\lambda \geq 0$ is the ℓ_2-regularization parameter. The implementation uses the numerically stable form above for both full-objective evaluation and mini-batch losses. For plots reported in objective-gap form, the reference value $f_\star$ is computed numerically using L-BFGS-B with the exact gradient.

Unless stated otherwise, we use $d = 512$ in the reported experiments. The number of stages S is varied in the scaling experiments and fixed to the value indicated in each figure caption for the direct method comparisons. Each method is tuned separately for the corresponding objective and stage count.

G.3 Weights stashing

Once the static timeline has been generated, we replay it to simulate training. Each stage maintains its current parameter vector and a local version counter. When a forward operation for microbatch m is executed at stage i, the stage uses the most recent locally available weights $w_i^{(t)}$. Immediately after this forward computation, that exact weight value and its version index are stored as part of the microbatch state for stage i. Later, when the matching backward operation for the same microbatch and stage is executed, the gradient is computed using the stashed version rather than the current version. After the gradient has been formed, the stage performs an SGD block update

$$w^{(s_k)} \leftarrow w^{(s_k)} - \gamma \nabla_{w^{(s_k)}} \ell_b,$$

and increments its local version counter.

This is exactly the PD weight-stashing rule: forward uses the latest available stage-local weights, and backward for a given microbatch reuses the same stage-local version that was used during forward. Importantly, this guarantees consistency only within a stage. It does not force all stages to use the same global parameter version for the same microbatch. PD calls the stronger cross-stage consistency mechanism vertical sync, but that is not part of its default semantics, and we did not include it in our initial simulator.

G.4 What was recorded

For reproducibility, we tracked four classes of quantities during replay.

First, we stored the full operation timeline, i.e. for each time step and each stage, whether the executed action was forward, backward, or idle, and for which microbatch. This timeline fully specifies the schedule.

Second, for every microbatch-stage pair, we stored the weight version used during forward and the weight version reused during backward. This allowed us to verify that the weight-stashing invariant held exactly:

$$\text{forward_version}(m, i) = \text{backward_version}(m, i).$$

Third, we measured stage-local staleness, defined as the difference between the current local version at backward time and the stashed version that had been used at forward time. This quantity captures how many local updates occurred at a stage between the forward and backward computations of the same microbatch.

Fourth, we tracked the full-dataset objective value over simulated pipeline time, as well as after each microbatch completed its backward pass at stage 1. The first view measures optimization progress against a wall-clock proxy given by pipeline time steps; the second measures optimization progress against the number of completed training examples.

G.5 Scope of the current simulator

The simulator is intentionally minimal. We assume one worker per stage, ignore communication delay, and treat each forward and backward stage operation as occupying one discrete time slot. We

also omit vertical sync. These choices are deliberate: the goal of the experiments is not to build a full systems simulator, but to isolate the effect of PD's scheduling and weight-stashing semantics on optimization in a setting where the ground-truth objective is simple and fully observable.

The simulator also does not charge memory as a separate axis in the plots. In particular, LocalSGD's R logical replicas require additional model storage compared with the single PD trajectory, and this extra memory is intentionally reported as a modeling difference rather than folded into the simulated-time budget.

G.6 Reproducibility details for the synthetic experiments

All synthetic experiments use zero initialization and seed 0 for the reported runs, with the deterministic training batch order generated from seed 1. When a method has internal sampling randomness, the same reported seed is passed to that method. Unless stated otherwise, the learning rate is selected by a grid sweep and the plotted curve is the stable selected run for that method/configuration. The main-paper synthetic plots are single-seed runs and therefore do not display error bars; the sweep utilities in the repository also support multi-seed aggregation and standard-deviation summaries.

For the PD/RPD validation in Figure 2a, we use the random quadratic objective with $S = 8$, $d = 512$, batch size 10, $M = 60$ data batches, and 5 epochs, so the PD timeline contains 300 microbatches. The learning-rate grid for both PD and RPD is

$$\{2^{-8}, 2^{-7}, \ldots, 2^{0}\}.$$

The delay sweep shown with RPD uses

$$\delta \in \{8, 30, 60, 180, 420\},$$

where $\delta = 60$ is the predicted steady-state PD value $S^2 - S/2$ for $S = 8$.

For the fixed-simulator-time logistic-regression scaling experiment in Figure 2b, we use $d = 512$, batch size 10, $M = 60$, $\lambda = 10^{-4}$, gradient-noise standard deviation 0.5, and stage counts

$$S \in \{2, 4, 8, 16, 32, 64, 128\}.$$

The common simulator-time target is 3684 discrete pipeline ticks, obtained as six times the $S = 8$, 300-microbatch PD timeline length used by the validation experiment. For each S, the PD and LocalSGD timelines use the smallest number of microbatches whose schedule reaches this target. RPD is assigned the number of block updates equal to the number of backward operations completed by the matched PD timeline. The learning-rate grids are $\{2^{-12}, 2^{-11}, \ldots, 2^{-3}\}$ for PD/RPD and $\{2^{-8}, 2^{-7}, \ldots, 2^{2}\}$ for LocalSGD. For this scaling plot, LocalSGD uses $R = S$ and $H = 5$.

For the $S = 16$ PD versus LocalSGD panels in Figure 4, we use the same simulator and learning-rate selection rule. The synthetic runs use $d = 512$ and zero initialization. LocalSGD uses $R = S$; the synchronization period H is swept in the corresponding notebook and then the displayed panel reports the selected tuned configuration for the plotted objective.

G.7 Experimental setup.

We compare PD and RPD on the same synthetic quadratic objective under a fixed block-update budget. For PD, we use the standard 1F1B schedule. For RPD, we run separate experiments for several values of the delay bound δ. In all cases, we start from the same initialization and evaluate progress using the full objective as a function of the number of block updates.

To make the comparison fair, we tune the stepsize independently for each method configuration. Concretely, for PD we sweep over a fixed grid of learning rates and keep the run with the smallest final full objective. For RPD, we repeat the same stepsize sweep separately for each tested value of δ, and again keep the run with the smallest final full objective. The corresponding best-tuned learning rate is reported in the figures.

Figure 2a, the left panel of Figure 2, compares the best-tuned PD trajectory against the best-tuned RPD trajectories for several delay bounds δ. The results show a strong dependence of optimization quality on the delay parameter. Small and moderate delay bounds yield substantially better objective decay, whereas larger values of δ lead to slower convergence and require smaller tuned learning rates in the same delay-sweep experiment.

Together with the fixed-time scaling panel in Figure 2b, this delay sweep highlights a tension between pipeline efficiency and optimization quality in the global-history RPD abstraction. On the one hand, PD 1F1B induces exact delays whose natural scale grows quadratically with the number of stages. On the other hand, RPD experiments show that large delay bounds are harmful for optimization and force more conservative stepsizes. Thus, while PD is highly attractive from the systems perspective, its induced global-history staleness becomes progressively less favorable as the pipeline depth increases.

G.8 NanoChat language-modeling experiment

We also compare PD and LocalSGD on a small autoregressive character-level language-modeling task implemented by the NanoChat-style causal Transformer used in our LLM simulator. The model is partitioned into pipeline stages: the first stage contains token and positional embeddings, the last stage contains the final layer normalization and language-modeling head, and the remaining stages contain Transformer blocks. We train on Tiny Shakespeare and measure last-stage forward cross-entropy loss as a function of simulated wall-clock time.

The model has $11{,}123{,}200$ trainable parameters and is split into $S = 16$ pipeline stages. We use sequence length 128, embedding dimension 256, 8 attention heads, mini-batch size 32, and 1024 data batches. PD is run with 1024 microbatches, producing a simulated-time budget of 2078 pipeline ticks and $16{,}384$ backward block updates. LocalSGD uses $R = 16$ parallel local runs. The selected LocalSGD synchronization period is $H = 5$, obtained by empirical tuning; the command below reproduces this selected configuration. Additional values of H can be reproduced by rerunning the same command with a different `-local-steps` value. Batches are shuffled during training.

The comparison uses a matched simulated-time budget. We first generate the PD timeline and use its length as the target horizon. For the selected LocalSGD setting $H = 5$, the smallest matching schedule uses 874 microbatches and has the same 2078-tick length. Thus the x-axis in Figure 4c is simulated wall-clock time rather than parameter-update count. Under this protocol, the tuned PD run obtains the lowest loss.

The experiment was run with:

```
python llm_pd_vs_sgd.py \
  --dataset tiny_shakespeare \
  --require-cuda \
  --num-stages 16 \
  --num-data-batches 1024 \
  --batch-size 32 \
  --seq-len 128 \
  --embed-dim 256 \
  --num-heads 8 \
  --pd-num-microbatches 1024 \
  --local-num-runs 16 \
  --local-steps 5 \
  --tune-stepsizes \
  --tuning-seeds 0 \
  --pd-lr-grid pow2:-4:2 \
  --local-sgd-lr-grid pow2:-4:2 \
  --shuffle-batches
```

The `pow2:a:b` grid denotes $\{2^a, 2^{a+1}, \ldots, 2^{b-1}\}$, so both PD and LocalSGD are tuned over $\{2^{-4}, 2^{-3}, \ldots, 2^{1}\}$ in this run. The script records the selected learning rates, final losses, schedules, curves, and learning-rate sweep summaries in the output directory.

G.9 Existing Assets and Licenses

We use several existing assets in the experimental part of the paper. For the larger LLM experiments, we use a small NanoChat/nanoGPT-style Transformer implementation based on the open-source educational codebases of Karpathy. NanoChat and nanoGPT are released under the MIT License. We cite these repositories and respect their license terms.

For the language-modeling data, we use Tiny Shakespeare, a standard small character-level language-modeling dataset consisting of approximately 40,000 lines of Shakespeare text. We credit the dataset source through Karpathy's `char-rnn`/Tiny Shakespeare reference. The dataset is used only for small-scale optimization experiments and not as a newly introduced dataset.

No proprietary datasets, scraped private data, user data, or restricted-access pretrained models are used in our experiments.

H A STRONGER CONVERGENCE THEOREM UNDER THE POLYAK–ŁOJASIEWICZ CONDITION

This section is not used in the main theoretical or empirical comparison. We include it only as a possible direction for future algorithms that use stale-read extrapolation.

We continue to work under the random pair-sampling model of the previous subsection.

Assumption H.1 (Polyak–Łojasiewicz condition)**.** *There exists $\mu > 0$ such that*

$$\frac{1}{2}\|\nabla f(w)\|^2 \geq \mu\big(f(w) - f_\star\big) \qquad \forall w \in \mathbb{R}^d.$$

Under this additional assumption, the RPD method enjoys a linear convergence guarantee up to a neighborhood whose size depends on the stepsize and the staleness level.

Theorem H.1 (Linear convergence to a neighborhood for RPD under PL).

Let Assumption B.1, Assumption B.3, Assumption B.4, Assumption B.2, Assumption B.5, Assumption H.1 hold. Assume

$$0 < \gamma \leq \frac{1}{L}.$$

Then for every $k \geq 0$,

$$\mathbf{E}[f(w_{k+1}) - f_\star] \leq \Big(1 - \frac{\mu\gamma}{S}\Big)\,\mathbf{E}[f(w_k) - f_\star] + \frac{L\gamma^2}{2}G^2 + \frac{L^2\gamma^3\delta^2}{2S}G^2. \tag{12}$$

The above theorem implies that for every $K \geq 1$,

$$\begin{aligned}\mathbf{E}[f(w_K) - f_\star] &\leq \Big(1 - \frac{\mu\gamma}{S}\Big)^K \big(f(w_0) - f_\star\big) + \frac{\frac{L\gamma^2}{2}G^2 + \frac{L^2\gamma^3\delta^2}{2S}G^2}{\mu\gamma/S} \\ &= \Big(1 - \frac{\mu\gamma}{S}\Big)^K \big(f(w_0) - f_\star\big) + \frac{SL\gamma}{2\mu}G^2 + \frac{L^2\gamma^2\delta^2}{2\mu}G^2. \end{aligned} \tag{13}$$

Proof. From the proof of Theorem B.1, we already have

$$\mathbf{E}[f(w_{k+1}) \mid \mathcal{F}_k] \leq f(w_k) - \frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{\gamma}{2S}\|\nabla f(w_k) - \nabla f(z_k)\|^2 + \frac{L\gamma^2}{2}G^2. \tag{14}$$

By L-smoothness and Lemma B.2,

$$\|\nabla f(w_k) - \nabla f(z_k)\| \leq L\|w_k - z_k\| \leq L\gamma\delta G,$$

and therefore

$$\|\nabla f(w_k) - \nabla f(z_k)\|^2 \leq L^2\gamma^2\delta^2G^2.$$

Substituting this into (14), we get

$$\mathbf{E}[f(w_{k+1}) \mid \mathcal{F}_k] \leq f(w_k) - \frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{L^2\gamma^3\delta^2}{2S}G^2 + \frac{L\gamma^2}{2}G^2.$$

Now apply the PL inequality:

$$\frac{1}{2}\|\nabla f(w_k)\|^2 \geq \mu\big(f(w_k) - f_\star\big),$$

which implies

$$\frac{\gamma}{2S}\|\nabla f(w_k)\|^2 \geq \frac{\mu\gamma}{S}\big(f(w_k) - f_\star\big).$$

Hence

$$\mathbf{E}[f(w_{k+1}) \mid \mathcal{F}_k] \leq f(w_k) - \frac{\mu\gamma}{S}\big(f(w_k) - f_\star\big) + \frac{L\gamma^2}{2}G^2 + \frac{L^2\gamma^3\delta^2}{2S}G^2.$$

Subtract $f_\star$ from both sides:

$$\mathbf{E}[f(w_{k+1}) - f_\star \mid \mathcal{F}_k] \le \left(1 - \frac{\mu\gamma}{S}\right)(f(w_k) - f_\star) + \frac{L\gamma^2}{2}G^2 + \frac{L^2\gamma^3\delta^2}{2S}G^2.$$

Taking total expectation proves (12).

Now define

$$\Delta_k \coloneqq \mathbf{E}[f(w_k) - f_\star], \qquad a \coloneqq 1 - \frac{\mu\gamma}{S}, \qquad b \coloneqq \frac{L\gamma^2}{2}G^2 + \frac{L^2\gamma^3\delta^2}{2S}G^2.$$

Then

$$\Delta_{k+1} \le a\Delta_k + b.$$

Iterating this recursion gives

$$\Delta_K \le a^K\Delta_0 + b\sum_{t=0}^{K-1} a^t \le a^K\Delta_0 + \frac{b}{1-a}.$$

Since $1 - a = \mu\gamma/S$, we obtain

$$\Delta_K \le \left(1 - \frac{\mu\gamma}{S}\right)^K (f(w_0) - f_\star) + \frac{\frac{L\gamma^2}{2}G^2 + \frac{L^2\gamma^3\delta^2}{2S}G^2}{\mu\gamma/S},$$

which simplifies to (13). □

I EXPLORATORY NESTEROV-INSPIRED EXTRAPOLATED VARIANT OF RANDOMIZED PIPEDREAM (NAG-RPD)

We now describe a predictive variant of RPD in which the stale model is extrapolated before the gradient is evaluated. This appendix is exploratory and is not used in the main comparison. The worst-case bound below does not show that the extrapolation parameter improves the guarantee; it only records what follows from the same smoothness argument as the base RPD theorem.

Algorithm 3 Nesterov-inspired extrapolated variant of Randomized PipeDream (NAG-RPD)

1: Input: stepsize $\gamma > 0$, prediction parameter $\alpha \geq 0$, initial point $w_0 \in \mathbb{R}^d$
2: Initialize $w_{-1} \coloneqq w_0$
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: Select a delay vector j_k from the past, with $0 \leq k - j_k^{(s)} \leq \delta$ for all s
5: Read the stale model

$$z_k \coloneqq \big(w^{(1)}_{j_k^{(1)}}, \ldots, w^{(S)}_{j_k^{(S)}}\big)$$

6: Form the extrapolated stale model

$$\widehat{z}_k \coloneqq \big(\widehat{z}_k^{(1)}, \ldots, \widehat{z}_k^{(S)}\big), \qquad \widehat{z}_k^{(s)} \coloneqq w^{(s)}_{j_k^{(s)}} + \alpha\big(w^{(s)}_{j_k^{(s)}} - w^{(s)}_{j_k^{(s)}-1}\big),$$

with the convention $w_{-1} \coloneqq w_0$
7: Sample (s_k, m_k) uniformly from $\{1, \ldots, S\} \times \{1, \ldots, M\}$, conditionally independently of $\widehat{z}_k$
8: Update the active stage block:

$$w_{k+1}^{(s_k)} = w_k^{(s_k)} - \gamma \nabla_{w^{(s_k)}} f_{m_k}(\widehat{z}_k)$$

9: Keep the remaining blocks unchanged:

$$w_{k+1}^{(r)} = w_k^{(r)} \qquad \forall r \neq s_k$$

10: **end for**

The increment $w^{(s)}_{j_k^{(s)}} - w^{(s)}_{j_k^{(s)}-1}$ is the previous *global-history* increment in block s and may be zero if the previous global update changed another block. A version that extrapolates from the previous local update of block s would require different notation and a separate analysis.

Lemma I.1 (Unbiasedness of the NAG-RPD update).

Let Assumption B.1 hold, and define

$$\widehat{g}_k \coloneqq U_{s_k} \nabla_{w^{(s_k)}} f_{m_k}(\widehat{z}_k).$$

Then

$$\mathbb{E}\left[S\widehat{g}_k \mid \mathcal{F}_k\right] = \nabla f(\widehat{z}_k),$$

where $\mathcal{F}_k$ includes the current delay vector and extrapolated stale point, but not the current sampled pair (s_k, m_k).

Proof. Conditioned on $\mathcal{F}_k$, the extrapolated point $\widehat{z}_k$ is fixed. Hence

$$\begin{aligned}\mathbb{E}\left[\widehat{g}_k \mid \mathcal{F}_k\right] &= \frac{1}{SM}\sum_{s=1}^{S}\sum_{m=1}^{M} U_s \nabla_{w^{(s)}} f_m(\widehat{z}_k) \\ &= \frac{1}{S}\sum_{s=1}^{S} U_s \left(\frac{1}{M}\sum_{m=1}^{M} \nabla_{w^{(s)}} f_m(\widehat{z}_k)\right) \\ &= \frac{1}{S}\sum_{s=1}^{S} U_s \nabla_{w^{(s)}} f(\widehat{z}_k) = \frac{1}{S}\nabla f(\widehat{z}_k).\end{aligned}$$

□

Lemma I.2 (Distance between current and extrapolated stale models).

Let Assumption B.2 hold, and assume that all block gradients evaluated by the NAG-RPD *trajectory are bounded by G. Then for every $k \geq 0$,*

$$\|w_k - \widehat{z}_k\| \leq \gamma(\delta + \alpha\sqrt{S})G.$$

Proof. Fix $k \geq 0$. By definition,

$$\widehat{z}_k^{(s)} = w^{(s)}_{j_k^{(s)}} + \alpha\big(w^{(s)}_{j_k^{(s)}} - w^{(s)}_{j_k^{(s)}-1}\big).$$

Hence

$$w_k - \widehat{z}_k = (w_k - z_k) - \alpha\, u_k,$$

where

$$u_k \coloneqq \big(w^{(1)}_{j_k^{(1)}} - w^{(1)}_{j_k^{(1)}-1}, \ldots, w^{(S)}_{j_k^{(S)}} - w^{(S)}_{j_k^{(S)}-1}\big).$$

By the triangle inequality,

$$\|w_k - \widehat{z}_k\| \leq \|w_k - z_k\| + \alpha\|u_k\|.$$

The same argument as in Lemma B.2 gives

$$\|w_k - z_k\| \leq \gamma\delta G,$$

because every update changes a single block by norm at most γG.

It remains to bound $\|u_k\|$. At any iteration t, only one block is updated, and

$$\|w_{t+1} - w_t\| = \gamma\|\nabla_{w^{(s_t)}} f_{m_t}(\cdot)\| \leq \gamma G.$$

For each stage s, the vector

$$w^{(s)}_{j_k^{(s)}} - w^{(s)}_{j_k^{(s)}-1}$$

is exactly the increment of block s at iteration $j_k^{(s)} - 1$, or zero if that block did not change there. Thus u_k is obtained by collecting at most one block increment for each block. Each block increment has norm at most γG, so the concatenated vector satisfies

$$\|u_k\| \leq \gamma G\sqrt{S}.$$

Therefore

$$\|w_k - \widehat{z}_k\| \leq \gamma\delta G + \alpha\gamma G\sqrt{S} = \gamma(\delta + \alpha\sqrt{S})G.$$

□

Theorem I.1 (Nonconvex convergence of NAG-RPD**).**

Let Assumption B.1, Assumption B.3, Assumption B.4, Assumption B.2, and Assumption B.5 hold. Assume also that the trajectory-bounded gradient condition holds at the extrapolated stale

points $\widehat{z}_k$. Assume

$$0 < \gamma \le \frac{1}{L}, \qquad \alpha \ge 0.$$

Then for every integer $K \ge 1$,

$$\frac{1}{K}\sum_{k=0}^{K-1} \mathbb{E}\left[\|\nabla f(w_k)\|^2\right] \le \frac{2S\big(f(w_0) - f_\star\big)}{\gamma K} + \gamma S L G^2 + \gamma^2 L^2 (\delta + \alpha\sqrt{S})^2 G^2. \tag{15}$$

Proof. Define

$$\widehat{g}_k := U_{s_k} \nabla_{w^{(s_k)}} f_{m_k}(\widehat{z}_k).$$

Then the update is

$$w_{k+1} = w_k - \gamma \widehat{g}_k.$$

By L-smoothness of f,

$$f(w_{k+1}) \le f(w_k) - \gamma \left\langle \nabla f(w_k), \widehat{g}_k \right\rangle + \frac{\gamma^2 L}{2} \|\widehat{g}_k\|^2.$$

Taking conditional expectation and using Lemma I.1,

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \le f(w_k) - \frac{\gamma}{S} \left\langle \nabla f(w_k), \nabla f(\widehat{z}_k) \right\rangle + \frac{\gamma^2 L}{2} \mathbb{E}\left[\|\widehat{g}_k\|^2 \mid \mathcal{F}_k\right]. \tag{16}$$

By the trajectory-bounded gradient assumption at the extrapolated stale points,

$$\|\widehat{g}_k\| = \|\nabla_{w^{(s_k)}} f_{m_k}(\widehat{z}_k)\| \le G,$$

so

$$\mathbb{E}\left[\|\widehat{g}_k\|^2 \mid \mathcal{F}_k\right] \le G^2.$$

Substituting into (16),

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \le f(w_k) - \frac{\gamma}{S} \left\langle \nabla f(w_k), \nabla f(\widehat{z}_k) \right\rangle + \frac{\gamma^2 L G^2}{2}.$$

Now use

$$\langle a, b \rangle \ge \frac{1}{2}\|a\|^2 - \frac{1}{2}\|a - b\|^2$$

with

$$a = \nabla f(w_k), \qquad b = \nabla f(\widehat{z}_k).$$

This gives

$$-\frac{\gamma}{S} \left\langle \nabla f(w_k), \nabla f(\widehat{z}_k) \right\rangle \le -\frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{\gamma}{2S}\|\nabla f(w_k) - \nabla f(\widehat{z}_k)\|^2.$$

Hence

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \le f(w_k) - \frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{\gamma}{2S}\|\nabla f(w_k) - \nabla f(\widehat{z}_k)\|^2 + \frac{\gamma^2 L G^2}{2}.$$

By L-smoothness of f,

$$\|\nabla f(w_k) - \nabla f(\widehat{z}_k)\| \le L\|w_k - \widehat{z}_k\|.$$

Using Lemma I.2,

$$\|w_k - \widehat{z}_k\| \le \gamma(\delta + \alpha\sqrt{S})G,$$

and therefore

$$\|\nabla f(w_k) - \nabla f(\widehat{z}_k)\|^2 \le \gamma^2 L^2 (\delta + \alpha\sqrt{S})^2 G^2.$$

Substituting this gives

$$\mathbb{E}\left[f(w_{k+1}) \mid \mathcal{F}_k\right] \le f(w_k) - \frac{\gamma}{2S}\|\nabla f(w_k)\|^2 + \frac{\gamma^3 L^2 (\delta + \alpha\sqrt{S})^2 G^2}{2S} + \frac{\gamma^2 L G^2}{2}.$$

Taking expectation and summing from $k = 0$ to $K - 1$,

$$\mathbb{E}\left[f(w_K)\right] \leq f(w_0) - \frac{\gamma}{2S} \sum_{k=0}^{K-1} \mathbb{E}\left[\|\nabla f(w_k)\|^2\right] + K\frac{\gamma^3 L^2(\delta + \alpha\sqrt{S})^2 G^2}{2S} + K\frac{\gamma^2 L G^2}{2}.$$

Using $\mathbb{E}\left[f(w_K)\right] \geq f_\star$, we get

$$\frac{\gamma}{2S} \sum_{k=0}^{K-1} \mathbb{E}\left[\|\nabla f(w_k)\|^2\right] \leq f(w_0) - f_\star + K\frac{\gamma^3 L^2(\delta + \alpha\sqrt{S})^2 G^2}{2S} + K\frac{\gamma^2 L G^2}{2}.$$

Multiply by $2S/(\gamma K)$ to obtain

$$\frac{1}{K} \sum_{k=0}^{K-1} \mathbb{E}\left[\|\nabla f(w_k)\|^2\right] \leq \frac{2S\big(f(w_0) - f_\star\big)}{\gamma K} + \gamma S L G^2 + \gamma^2 L^2(\delta + \alpha\sqrt{S})^2 G^2,$$

which proves (15). □