



Interval-Asynchrony: Delimited Intervals of Localised Asynchrony for Fast Parallel SGD

Jacob Garby^{1,2}   and Philippas Tsigas^{1,2} 

¹ Chalmers University of Technology, Gothenburg, Sweden

² University of Gothenburg, Gothenburg, Sweden

{garby,tsigas}@chalmers.se

Abstract. Stochastic gradient descent (SGD) is a crucial optimisation algorithm due to its ubiquity in machine learning applications. Parallelism is a popular approach to scale SGD, but the standard synchronous formulation struggles due to significant synchronisation overhead. For this reason, asynchronous implementations are increasingly common. These provide an improvement in throughput at the expense of introducing stale gradients which reduce model accuracy. Previous approaches to mitigate the downsides of asynchronous processing include adaptively adjusting the number of worker threads or the learning rate, but at their core these are still fully asynchronous and hence still suffer from lower accuracy due to more staleness.

We propose *Interval-Asynchrony*, a semi-asynchronous method which retains high throughput while reducing gradient staleness, both on average as well as with a hard upper bound. Our method achieves this by introducing periodic *asynchronous intervals*, within which SGD is executed asynchronously, but between which gradient computations may not cross. The size of these intervals determines the degree of asynchrony, providing us with an adjustable scale. Since the optimal interval size varies over time, we additionally provide two strategies for dynamic adjustment thereof. We evaluate our method against several baselines on the CIFAR-10 and CIFAR-100 datasets, and demonstrate a 32% decrease in training time as well as improved scalability up to 128 threads.

Keywords: Parallel Algorithms · Parallel SGD · Staleness · Asynchronous Data Processing

1 Introduction

Stochastic gradient descent (SGD) is a classic and widely used algorithm for optimising the parameters of some model to minimise a given loss function by iteratively computing its *gradient* with respect to the current parameters. In the simplest sequential formulation, a series of iterations are carried out following $\theta_{i+1} \leftarrow \theta_i - \eta \nabla L_{B_i}(\theta_i)$ [15], where θ_i is the parameter vector of the model following iteration i ; $L_{B_i}(\theta_i)$ is the target function which evaluates the loss of the model given parameters θ_i evaluated on a *minibatch* B_i of training samples;

and η is the learning rate, controlling the impact of an individual gradient. Each minibatch B_i consists of a subset of samples from the entire training dataset.

The convergence rate of SGD can be increased through *data-parallelism*. Data-parallel SGD is traditionally formulated synchronously: workers run in lockstep with each other, at each iteration i computing a gradient from a subset of B_i . Between steps all workers synchronise, and their individual gradients are aggregated and used to update θ_i . Semantically, this is exactly equivalent to the aforementioned sequential formulation [2, 9] (which itself has desirable convergence properties even for certain non-convex optimisation problems, such as training deep neural networks [12, 15, 20]).

Unfortunately, synchronous parallel SGD does not scale well to large numbers of threads. Since steps are processed in lockstep, a thread which finishes its gradient computation early can do nothing but sit idle until all the others finish too. This can significantly reduce the *throughput* (i.e. the number of training samples processed per unit time) leading to slower convergence.

In order to better utilise the CPU, many machine learning algorithms, frameworks, and software libraries make use of *asynchronous* processing [2–4, 14, 18]. This relaxes the semantics of the sequential SGD formula; specifically, threads are allowed to apply their computed gradients to the model independently as soon as they are finished, immediately starting a new step afterwards. In this way, the gradient used to compute θ_{i+1} is no longer necessarily based on θ_i . Instead, asynchronous updates follow $\theta_{i+1} \leftarrow \theta_i - \eta \nabla L_{B_i}(\theta_{i-\tau_i})$. Here, τ_i refers to the *staleness* of step i , i.e. the number of intermediate versions the parameters θ have gone through since the state that was used to compute this gradient. Higher values of τ correspond with worse statistical efficiency, defined as the improvement in training loss per step ($\frac{dL}{dt}$). $\mathbb{E}[\tau]$ increases linearly with the number of threads, and therefore plain asynchronous SGD does not scale well. When considering synchronous vs. asynchronous execution, we can either achieve scalable throughput or good statistical efficiency, but not both at the same time.

Previous efforts [2–4, 13, 19] to manage this trade-off include dynamically adjusting the number of active worker threads and the training batch size, as well as scaling the impact of gradients based on observed staleness. These all demonstrate impressive performance, recovering from the impact of stale updates by adaptively adjusting different parameters in response – either explicitly or implicitly – to the distribution and effect of staleness. We consider an orthogonal approach in which the synchronisation semantics are adjusted such that the actual distribution of staleness is improved. Our contributions are as follows:

- We propose a novel semi-asynchronous execution strategy, *Interval-Asynchrony*, which results in shorter convergence time for high-parallelism SGD.
- We propose and evaluate strategies for setting *Interval-Asynchrony*’s interval size; in particular, we describe an online probing method which aims to dynamically optimise the interval size during a single execution.
- We provide a comprehensive evaluation of our method against two baseline asynchronous approaches and one synchronous one.

The rest of this paper is organised like so: in Section 2 we give an outline of some related work, which helps to motivate our algorithm described in the following Section 3. There we explain the algorithm from a conceptual point of view and provide an efficient lock-free implementation. In Section 4 we discuss in more detail the effect of the interval size and suggest methods by which it can be chosen and adjusted. Section 5 provides a comprehensive evaluation of *Interval-Asynchrony*, primarily by comparing its convergence rate against a number of baseline methods.

2 Related Work

Elastic Parallelism. Recent work [3] demonstrated that an effective way to increase the convergence rate for asynchronous parallel SGD is through dynamically adjusting the number of active worker threads. This work observes that the optimal number of threads varies throughout an execution; much lower staleness is typically required closer to convergence. Their strategy attempts to find the time-varying optimal number of threads using probing to estimate the convergence rate of candidate values.

While this was shown to provide a significant speed-up in many cases, a downside of this type of method is that it does not facilitate scalability above the maximum number of threads that it deems optimal.

Staleness Adaptiveness. Instead of adapting the number of workers, some works have proposed different ways to adjust the learning rate, either over time [7] or based on measured staleness [4]. Additionally, a technique was proposed for distributed parallel SGD which relaxes the synchrony in a way similar to asynchronous SGD, while also reducing gradient staleness [10].

Concurrent Model Access. Threads performing asynchronous SGD contend for access to the shared global model. The naïve implementation involves using a lock to ensure mutual exclusion for shared model access, guaranteeing consistency at the expense of throughput. The HOGWILD! algorithm [14] takes a different approach, giving threads unrestricted concurrent access to the shared model. Although HOGWILD! introduces inconsistency when updating the model, it is proven to converge in general given bounded staleness [14, 17]. Additionally, atomic operations such as *Compare-and-Swap* have been used to provide lock-free consistent model updates [1, 5].

3 Interval-Asynchronous Execution

Our main contribution is a thread scheduling and synchronisation algorithm called *Interval-Asynchrony*. In this section we give a description of *Interval-Asynchrony* and justify its design with respect to scalability. We then discuss various online methods for selecting values for the asynchrony interval size.

Algorithm 1 One SGD Worker w_{id} , illustrating invocation of the *Dispatcher*

```

1: while !ISFINISHED() do
2:    $\text{can\_start}, i \leftarrow \text{TRYSTARTSTEP}(w_{id})$ 
3:   if  $\text{can\_start}$  then  $\triangleright$  Dispatchers may be restrictive about start conditions.
4:      $\theta_{local} \leftarrow \theta_{global}$   $\triangleright$  Make a local copy of the model state.
5:      $B_i \leftarrow \text{GETBATCH}(i)$   $\triangleright$  Retrieve the  $i$ th batch of training data.
6:      $g_i \leftarrow \nabla \tilde{L}_{B_i}(\theta_{local})$   $\triangleright$  The “slow” part – computing a gradient.
7:     if  $\text{FINISHSTEP}(w_{id}, i)$  then  $\triangleright$  If the Dispatcher allows it...
8:        $\theta_{global} \leftarrow \theta_{global} + \eta g_i$   $\triangleright$  ...we apply our gradient to the global model.
9:     end if
10:  end if
11: end while

```

In order to make the best use of the available threads while limiting the impact of stale updates, our strategy lets us smoothly adjust the degree of asynchrony. We achieve this by logically splitting the execution into intervals, during which threads execute SGD asynchronously, and between which synchronisation occurs. We introduce a parameter called the *asynchrony interval*, which is the number of SGD steps that make up one such interval. We call this value y .

In order to describe *Interval-Asynchrony*, we first introduce the concept of a *Dispatcher*. A *Dispatcher* is an interface which exposes two functions to worker threads. $\text{TRYSTARTSTEP}(w_{id})$ determines whether a given thread with id w_{id} may begin a step, and $\text{FINISHSTEP}(w_{id}, i)$ determines whether the gradient computed by a given step i is allowed to be applied to the model. TRYSTARTSTEP returns a pair: a boolean value for whether the step may begin, and a step *start index*, i . FINISHSTEP simply returns a boolean for whether a certain step’s gradient is accepted or rejected. Accepted steps’ gradients are processed like normal, i.e. used to update the global set of parameters, whereas the gradient of a rejected step is simply discarded.

Workers interact with the *Dispatcher* according to Algorithm 1. The function ISFINISHED decides when the entire execution has finished; this may be based on time, number of epochs, or model performance. We use HOGWILD!-semantics [14] for concurrent global model updates (*Line 8*), such that multiple model updates can be interleaved.

Pseudocode for an efficient implementation of our Interval-Asynchronous *Dispatcher* is given in Algorithm 2. The variables I_{first} and I_{done} keep track of the state of the current asynchronous interval; they are the step *start index* which initiated the interval and the number of steps that have been accepted so far during the interval, respectively. $s_{started}$ is the total number of steps started, and s_{done} is the number of steps that have been accepted by the *Dispatcher*.

The implementation of TRYSTARTSTEP is straightforward because steps may begin unrestricted. Still, we have to make sure that the thread is within the current parallelism bound ($w_{id} < m$, where m is the number of active threads). This condition is not strictly necessary for Interval-Asynchronous execution, but we include it anyway since it allows for runtime adjustment of the number of

Algorithm 2 Interval-Asynchronous Dispatcher

```

1:  $I_{first} \leftarrow 0$   $\triangleright$  Interval's first step
2:  $I_{done} \leftarrow 0$   $\triangleright$  Interval's accepted count
3:  $s_{started} \leftarrow 0$   $\triangleright$  Total steps started
4:  $s_{done} \leftarrow 0$   $\triangleright$  Total steps completed
5:  $y \leftarrow y_0$   $\triangleright$  Initial interval size

6: function TRYSTARTSTEP( $w_{id}$ )
7:   if  $w_{id} \geq m$  then
8:     return false, -1
9:   else
10:    return true, FAA( $s_{started}$ , 1)
11:   end if
12: end function

13: function UPDATEINTERVAL( $y$ )
14:   return  $y - 1$   $\triangleright$  Simple  $y$ -decay
15: end function

16: function FINISHSTEP( $w_{id}, i$ )
17:   if  $i < I_{first}$  then
18:     return false
19:   end if
20:   repeat
21:      $old \leftarrow I_{done}$ 
22:     if  $old \geq y$  or  $i < I_{first}$  then
23:       return false
24:     end if
25:      $new \leftarrow old + 1$ 
26:     until CAS( $I_{done}, old, new$ )
27:     if  $new = y$  then
28:        $y \leftarrow$  UPDATEINTERVAL()
29:        $I_{first} \leftarrow s_{started}$ 
30:        $I_{done} \leftarrow 0$ 
31:     end if
32:     FAA( $s_{done}$ , 1)
33:   return true
34: end function

```

threads, m , highlighting the versatility of the *Dispatcher* interface and demonstrating how our method can be integrated with existing techniques (specifically *ElAsyncSGD*). The use of *fetch-and-add* (FAA) atomically increments $s_{started}$ and returns its prior value.

The FINISHSTEP implementation for *Interval-Asynchrony* checks if a certain SGD step is contained entirely by the current interval and, if required, ends the current interval and sets up the next one. A step is contained by the current interval iff $i \geq I_{first}$ and $I_{done} < y$. We use *compare-and-swap* (atomic $I_{done} \leftarrow new$ iff $I_{done} = old$, returning *true* on success) to ensure consistency when checking the above two conditions and incrementing I_{done} . This approach is more scalable than a simpler implementation with a lock around FINISHSTEP. new refers to the order in the current interval with which this step *finished*. When an accepted step finishes an interval ($new = y$), we start the next one by updating I_{first} . We also provide the option of changing the size of the next interval with UPDATEINTERVAL. In Section 4.1 we justify why and how we may wish to do so.

Values for y can be any integer. Higher values of y bring the execution closer to fully asynchronous semantics (and of course if $y \geq s_{target}$, where s_{target} is the total number of SGD steps we wish to run, then the entire execution fits within just one interval, which is then exactly equivalent to asynchronous execution). On the other hand, $y = 1$ is not exactly the same as synchronous data-parallel execution; instead, it is semantically similar to a sequential execution. In practice, we select values for y somewhere between these two extremes.

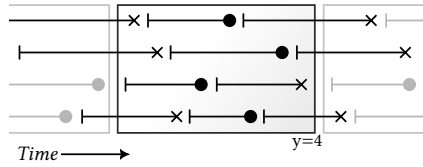


Fig. 1: One interval within an Interval-Asynchronous execution.

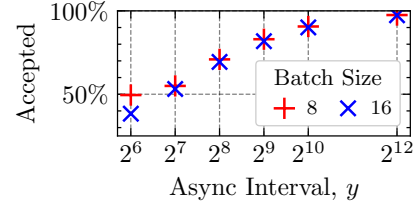


Fig. 2: Acceptance rate *vs.* async interval, $m = 128$.

An illustrative example of *Interval-Asynchronous* execution is shown in Figure 1. The shaded rectangle refers to one asynchronous interval with $y = 4$. Each row depicts the execution of one thread (hence $m = 4 = y$, in this case), within which each line depicts the *start index* and end order of one step. Lines terminating in a circle represent steps whose gradient was accepted by the *Dispatcher*, whereas those ending in a cross were rejected.

The *scalability* of an algorithm describes the performance benefit of using additional threads in parallel to execute it. Plain synchronous and asynchronous SGD both struggle to scale to high thread counts, although for different reasons: the former suffers significant overhead due to between-iteration synchronisation, while the latter eliminates synchronisation overhead at the expense of unbounded staleness, reducing the effectiveness of individual iterations.

Interval-Asynchrony is similarly able to reduce the synchronisation-induced overhead, in part due to the asynchronous execution within each interval. The other reason is that the synchronisation points (i.e. interval boundaries) don't cause threads to wait in the same way that they do between iterations of synchronous SGD: a new interval begins as soon as enough gradients are accepted in the previous one, and so threads don't need to wait for each other. On the other hand, while *Interval-Asynchrony* does induce stale gradients, the expected staleness of a given gradient is significantly less than that of asynchronous executions. For these reasons, we expect that *Interval-Asynchrony* will scale up to a higher number of threads than either synchronous or asynchronous SGD.

4 Asynchronous Interval Size

Now we discuss the impact of the choice of y , and in particular how it relates to the number of threads, m . Figure 2 shows that the proportion of accepted steps increases with y . This happens because if the interval is larger then we expect fewer steps to overlap interval boundaries due to longer periods of asynchronous execution. The acceptance rate is affected by the batch size because larger batches beget slower steps, leading to a higher chance that a step crosses an interval boundary.

When $y \leq m$ it is very likely that no thread will compute more than one *accepted* gradient in a given interval. Although the effect of this is comparable

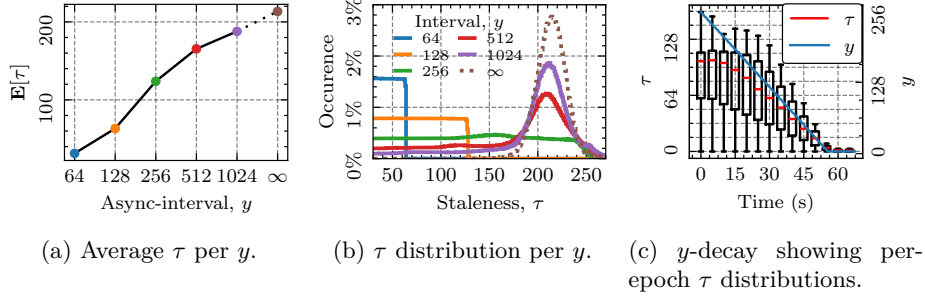


Fig. 3: The effect of y on the distribution and expectation of τ .

to synchronous parallel SGD with $m_{\text{synchronous}} = y$ (assuming step duration variance \ll mean step duration, which is always true in practice in a shared-memory setting), we can expect improved throughput. This is because interval-async execution reduces the average step duration, since instead of waiting for a specific set of m steps to complete, we dispatch m steps and stop once y of them have finished, resulting in a lower expected wait time, even when $m_{\text{synchronous}} = y$, i.e. accepting the same number of steps in each interval as the total number of workers in a comparable synchronous execution.

Conversely, when $y > m$, it is guaranteed that at least one thread will produce more than one accepted step in a given interval. However, the distribution of staleness in this case is still lower than for a fully asynchronous execution, in addition to there being a *bound* on maximum staleness: for any y, m , an accepted step will always have $\tau \leq y$, and even when $y > m$ we can expect a lower staleness than that of fully asynchronous execution.

The relationship between the asynchronous interval y and the measured distribution of staleness, using 256 threads, is presented in Figures 3a and 3b. For $y \leq m = 256$, we get mostly uniform τ -distributions. This is because in these cases, due to the behaviour described above, the first step has $\tau_0 = 0$, the second will have $\tau_1 = 1$, and so on, because it is most likely that no thread computes more than one step. The distribution is slightly non-uniform for $y = 256$ because with a larger asynchronous interval it becomes more likely that steps that began later will get a chance to finish. When $y \geq 512$ we see skewed Gaussian distributions but with a “fat” lower tail; the lower tail is due to the expected initial sequence of $\tau_i = i$ for some number of steps at the beginning of each interval.

In order to compare the staleness distributions for interval-async execution in Figure 3b to that of fully asynchronous execution, which is presented in the same plot as a dotted line, the expected (mean) staleness for each interval size is shown in Figure 3a. Note that the x -axis is logarithmic, hence a linear increase in y causes $\mathbf{E}[\tau]$ to increase roughly logarithmically.

4.1 Selecting y

The asynchronous interval size y influences the degree of asynchrony of the execution, and therefore choosing a suitable value is important. If y is too low, the throughput is throttled too much due to the increased proportion of rejected gradients, in which case the convergence rate may be suboptimal. If y is too high, convergence may take longer due to the impact of noise from stale updates.

Although we could keep y constant throughout an entire execution, it is far better to dynamically adjust y so that we can maintain a good convergence rate throughout. Since the degree to which the model is susceptible to noise induced from high asynchrony and staleness is not constant, it is natural to respond with adjustments to the interval, y . It turns out that the model tends to become more sensitive to such “noisy” updates as it approaches convergence [4,6]. This can be seen intuitively: more precise control is required when we are close to a minimum, otherwise we will keep overshooting.

y-decay: Since SGD benefits from high asynchrony initially but requires less staleness later on in order to effectively reach some minimum, we propose a simple scheme for adjusting y : initialise $y \leftarrow y_0$ and gradually decrease it over time. A simple decay strategy like this one is common in literature, for example the popular approach of decaying the learning rate, η . Figure 3c shows an example of how the distribution of τ changes as y is decayed over time, starting from $y_0 = 256$ and decrementing gradually over time. The median decreases in accordance with y , but it is interesting to note that the lower quartile begins to decrease even before $y \leq m = 128$, demonstrating that *Interval-Asynchrony* provides not only an upper bound of $\tau \leq y$, but also an overall shift in the distribution. y -decay can be a very effective strategy, as we will show later, but its performance does depend on the speed at which y is decreased.

Adaptive y through window-probing: Rather than relying on manual selection of y -decay gradient, we propose an online adaptive method for selecting a suitable dynamic y at runtime. This approach employs a similar window-probing parameter search method as used by *ElAsyncSGD* [3], using w as a configurable window size, and p and x as the number of steps comprising a probing and an execution step, respectively.

Alternating *execution* and *probing* phases occur. During an *execution* phase, SGD is carried out as usual (following our *Interval-Asynchronous* semantics with the current y). After x steps are accepted, a probing phase begins. The purpose of a probing phase is to produce an estimation of which candidate interval size $y' \in [y - \frac{kw}{2}, y + \frac{kw}{2}]$ yields the best convergence rate at the current stage of the execution. During this phase, we execute SGD for p steps at each candidate y , keeping track of which yields the best convergence rate. This is then used for the subsequent execution phase.

5 Results & Evaluation

We now present an evaluation of our method compared to two asynchronous baselines as well as a fully synchronous one, with the intention of evaluating the training speed and accuracy, as well as scalability up to 128 concurrent threads.

We consider the use case of using SGDM (SGD with momentum [16], as is industry-standard) to train two different convolution neural network models, one for CIFAR10 and one for CIFAR100. For both datasets we use a LeNet5-like architecture, consisting of the following layers: *Convolution* \rightarrow *Pooling* \rightarrow *Convolution* \rightarrow *Pooling* \rightarrow *Dense* \rightarrow *Dense* \rightarrow *Dense*. We use an AMD EPYC 9754 128-core processor.

We compare our proposed *Interval-Asynchronous* execution (hereafter referred to in figures as *IntAsync*) to several baseline algorithms: fully asynchronous with constant number of active threads (*Async*); fully asynchronous with elastic parallelism due to the aforementioned window probing technique (*ELAsync*); and synchronous parallel SGD (*Sync*).

Except where otherwise specified, we use $\eta = 0.005$ (learning rate), $\mu = 0.5$ (momentum parameter), and $\|B_i\| = 16$ (batch size). We aim to select optimal parameters for the elastic parallelism baseline to challenge our algorithm as much as possible; to this end, we performed tests with many probing parameter combinations, and determined the best to be a window width of 12, and 1024 and 8192 steps per probe and execution phase respectively.

5.1 Accuracy

In order to evaluate the efficacy of *Interval-Asynchrony* we compare the accuracy it is able to achieve, as well as the time taken to reach it, to the two baselines. This is displayed in Table 1 in which we report the best accuracy achieved across all tested configurations (i.e. different numbers of threads) for each algorithm, and the average accuracy across all thread count levels. To provide a meaningful comparison we show the time at which each reached the highest accuracy achieved by all algorithms (50.9% for CIFAR-10, and 5.1% for CIFAR-100³). We also show the average time to reach this accuracy across all executions, but it is important to note that the baselines *Async* and *ELAsync* only get to this accuracy when using the minimal number of threads, 32. For this reason, we omit those averages from the table. The average time for *Sync* on CIFAR-100 is listed as “>5400s” because we only ran each experiment for ninety minutes, during which time *Sync* only reached the target accuracy in one instance. This is different to the *Async* runs which did not get to the target accuracy: with enough time, all *Sync* executions would reach the target, whereas many of the *Async* experiments *did* converge, but to a lesser accuracy.

For both datasets, *ELAsync* and *Interval-Async* are both capable of achieving a better accuracy than constant thread count *Async* with any tested number

³ The accuracy on CIFAR-100 is limited by the neural network architecture that we use (LeNet5), and is consistent with previous works using LeNet5 [11].

	CIFAR-10				CIFAR-100			
	Accuracy		Time to 50.9%		Accuracy		Time to 5.1%	
	Best	Avg.	Best	Avg.	Best	Avg.	Best	Avg.
<i>(Sync)</i>	52.7%	52.7%	1071.5s	1760.7s	6.7%	6.7%	4485s	>5400s
<i>Async</i>	50.9%	46.0%	43.9s	*	5.1%	4.5%	251.8s	*
<i>ELAsync</i>	52.6%	49.0%	54.4s	*	5.5%	4.6%	264.5s	*
IntAsync	51.8%	50.1%	30.0s	38.2s	5.4%	5.1%	193.3s	269.0s

*In these cases the target accuracy was never reached for some m .

Table 1: Accuracies achieved by each algorithm, and comparisons of training time to common accuracies.

of threads. In the best case, *ELAsync* manages a slightly higher accuracy than our method, because its best case is using only 32 threads *maximum*. *Interval-Async*'s best accuracy of 51.8% was reached using 64 threads, hinting at its better scalability. We can also see that *Interval-Asynchrony* delivers a significantly faster training time in the best case. It is 32% faster than *Async* to reach the threshold for CIFAR-10, and 23% faster for CIFAR-100.

5.2 Scalability

In addition to looking at the best case for each method, we observe the performance as we increase the number of threads. Looking at Figure 4 we can see that *Async* and *ELAsync* are only capable of reaching close to their best accuracy when relatively few threads are used. On the other hand, our *Interval-Asynchrony* is able to achieve close to its optimal accuracy across all tested numbers of threads, in the majority of configurations. Hence, if we were to use *Async* or *ELAsync* for a real-world training application we would have to carefully tune the number of threads to make sure we get an acceptable balance between speed and accuracy, but by using *Interval-Asynchrony* we no longer need to worry about this. This behaviour is shown both in Figure 4, as well as by observing the difference between the best and average accuracies achieved by the different algorithms: when these two values are closer, the accuracy degrades less with more threads.

Figure 4 further demonstrates the scalability of *Interval-Asynchrony* in its second row of plots. In these, we select a certain threshold accuracy (different to that used in Table 1, and different for each dataset) and present the time each configuration takes to reach this, looking individually at each number of threads. We select these thresholds (45% and 4%) such that for almost every number of threads, every algorithm reaches at least this accuracy. We exclude the synchronous results from these plots since its accuracy is unaffected by parallelism due to zero staleness, and its training time is so significantly higher than the asynchronous algorithms that it cannot be shown on the same scale, as can be seen in Table 1.

Figure 5 shows the accuracy and loss during the execution of each configuration from Figure 4. Although the accuracy and loss metrics both provide some

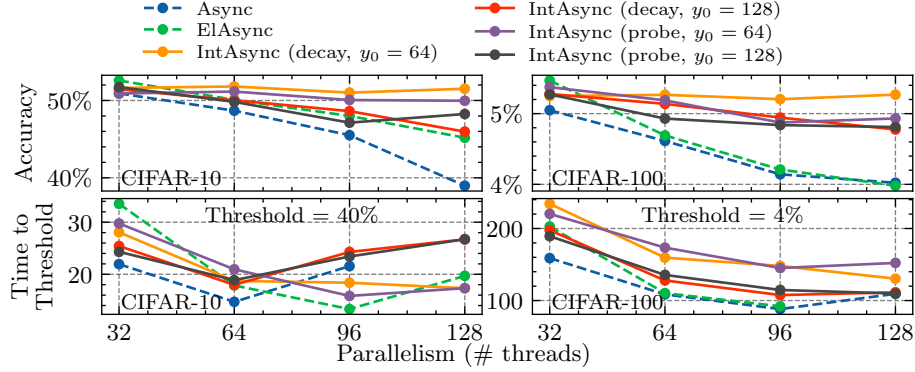


Fig. 4: Scalability of best accuracy and time to reach threshold accuracies.

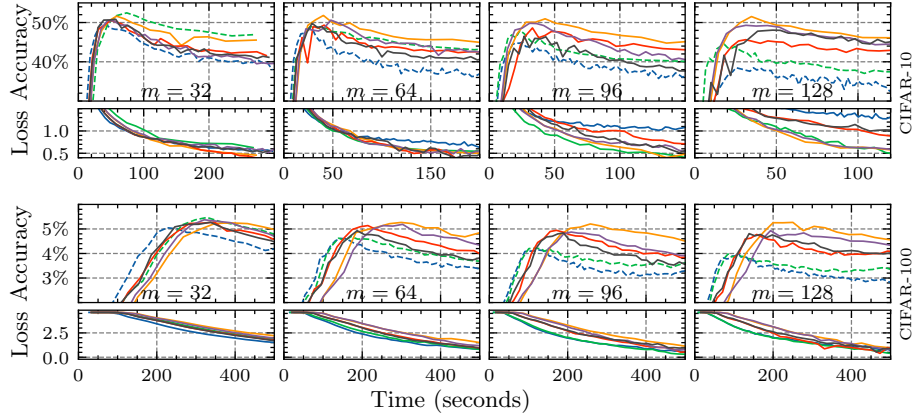


Fig. 5: Accuracy and loss over time for each algorithm and thread count.

indication of model quality, the latter is less accurate. Our *y-probing* considers loss rather than accuracy because it's much faster to compute at runtime. While lower loss does tend to imply better accuracy, we can see in Figure 5 that this is not an exact correlation. For this reason, *y-probing* does not necessarily produce the *best* value for *y*, but as shown in Figure 4 it is still an effective strategy.

The general trend is an initial increase in speed w.r.t. parallelism due to increased throughput without yet too much staleness, followed by a slowdown as each algorithm is no longer able to make good use of the additional threads. For CIFAR-10, we see the two *Interval-Async* executions with initial $y_0 = 64$ excelling in terms of scalability, tending to continue speeding up as more threads are available. At the other end of the spectrum, *Async* struggles to speed up once more than 64 threads are used, and at 128 threads is not even capable of reaching the threshold accuracy. Note that although *ELAsync*'s scalability looks

promising since it is sometimes faster than *Interval-Asynchrony*, in these cases (and indeed in almost every case) it does not train the model to have as high an accuracy as *Interval-Asynchrony*. An initial fast increase in accuracy (even up to the threshold as shown here) is not always beneficial in the long run.

According to Figure 2, for the values of y we are using we should expect overall less than 50% of the total steps to be accepted, reducing the throughput by more than a half. Despite this, the total training time does not double, emphasizing that our method substantially increases the statistical efficiency, more than making up for the lower throughput. This improvement is due to lower staleness on average, as well as an absolute upper bound, as shown in Figure 3b.

5.3 Efficacy of Adaptive Interval Size

So far we have mostly considered all configurations of *Interval-Async* together, in order to discuss the method in general. We now provide a discussion of our two proposed techniques for adjusting the interval size, y . In Figures 4 and 5 we experiment with both y -decay and y -probing, and for each of these methods we consider initial interval sizes $y_0 \in \{64, 128\}$. For y -decay, we decrease y by 1 per every 4,096 accepted steps.

It turns out that *Interval-Async* SGD execution is sensitive to the initial y . Although the probing technique is intended to discover close to the best y , probing is restricted to a window and therefore will only find locally optimal values. If the initial value is so large that it causes significant inaccuracy, the probing may not be able to rectify this fast enough to be effective. A broader search strategy could improve y -probing results, at the expense of more time spent searching. More efficient searching thereof would make for interesting future work.

We propose $y_0 = 64$ as a good balance between training speed and scalability, but note that a method for picking a better y_0 could provide more consistent scaling at high thread counts. Although a suboptimal y_0 can have negative consequences on training time, it remains faster than the baseline algorithms. The maximum accuracy achieved also beats the baselines in almost every case.

6 Conclusions & Future Work

We demonstrate that our algorithm can reach a consistently higher accuracy than the baselines across two challenging datasets, suggesting that this trend holds in other settings too. Our method becomes increasingly useful as more threads are used: at higher numbers of threads the model accuracy reached stays constant, not suffering too much from asynchrony induced noise, whereas the baseline asynchronous algorithms experience substantially worse accuracy. Even more importantly we show that our method is capable of effectively making use of additional processors to reduce the time taken to reach a certain model accuracy. Our method achieves a consistently high accuracy, regardless of the number of threads used, by managing the distribution of staleness through self-contained intervals of asynchrony, and it does so at a competitive speed since the enforcement of these intervals does not sacrifice throughput too much.

Although we propose an effective window-probing approach for automatically controlling the interval size, it is often sensitive to the initial size, y_0 . Further work in this area is needed in order to design a more effective automatic controller for this parameter, for example incorporating a heuristic method to determine a suitable y_0 . More generally, there are a number of other parameters for which online control is conceivably beneficial to training performance. These are, at least: batch size, learning rate, thread count, and asynchronous interval size. An interesting piece of further work would produce a holistic controller for such parameters.

Acknowledgements and Artifact Availability This work was supported by the Marie Skłodowska-Curie Doctoral Network *RELAX-DN*, and funded by the EU under the Horizon Europe 2021-2027 Framework, Grant Agreement nr. 101072456. The artifact is available in the Zenodo repository [8].

Disclosure of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ben-Nun, T., Hoefer, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis (2018). <https://doi.org/10.48550/arXiv.1802.09941>, <http://arxiv.org/abs/1802.09941>
2. Bäckström, K., Papatriantafylou, M., Tsigas, P.: MindTheStep-AsyncPSGD: Adaptive asynchronous parallel stochastic gradient descent. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 16–25. IEEE (2019). <https://doi.org/10.1109/BigData47090.2019.9006054>, <https://ieeexplore.ieee.org/document/9006054/>
3. Bäckström, K.: Adaptiveness, asynchrony, and resource efficiency in parallel stochastic gradient descent, <https://research.chalmers.se/en/publication/535694>, ISBN: 9789179058555
4. Bäckström, K., Papatriantafylou, M., Tsigas, P.: ASAP.SGD: Instance-based adaptiveness to staleness in asynchronous SGD. In: Proceedings of the 39th International Conference on Machine Learning. pp. 1261–1276. PMLR (2022), <https://proceedings.mlr.press/v162/backstrom22a.html>, ISSN: 2640-3498
5. Bäckström, K., Walulya, I., Papatriantafylou, M., Tsigas, P.: Consistent lock-free parallel stochastic gradient descent for fast and stable convergence. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 423–432 (2021). <https://doi.org/10.1109/IPDPS49936.2021.00051>, <https://ieeexplore.ieee.org/document/9460457>, ISSN: 1530-2075
6. Dai, W., Zhou, Y., Dong, N., Zhang, H., Xing, E.: Toward understanding the impact of staleness in distributed machine learning. In: 2019 International Conference on Learning Representations (ICLR) (2019). <https://doi.org/https://doi.org/10.48550/arXiv.1810.03264>
7. Dutta, S., Joshi, G., Ghosh, S., Dube, P., Nagpurkar, P.: Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. In: Storkey, A., Perez-Cruz, F. (eds.) Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research,

- vol. 84, pp. 803–812. PMLR (09–11 Apr 2018), <https://proceedings.mlr.press/v84/dutta18a.html>
8. Garby, J., Tsigas, P.: Artifact of the paper: Interval-asynchrony: Delimited intervals of localised asynchrony for fast parallel sgd (Jun 2025). <https://doi.org/10.5281/zenodo.15576941>, <https://doi.org/10.5281/zenodo.15576941>
 9. Gupta, S., Zhang, W., Wang, F.: Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In: 2016 IEEE 16th International Conference on Data Mining (ICDM). pp. 171–180 (2016). <https://doi.org/10.1109/ICDM.2016.0028>
 10. Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J.K., Gibbons, P.B., Gibson, G.A., Ganger, G., Xing, E.P.: More effective distributed ml via a stale synchronous parallel parameter server. In: Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K. (eds.) *Advances in Neural Information Processing Systems*. vol. 26. Curran Associates, Inc. (2013), https://proceedings.neurips.cc/paper_files/paper/2013/file/b7bb35b9c6ca2aee2df08cf09d7016c2-Paper.pdf
 11. Jeong, H., Son, H., Lee, S., Hyun, J., Chung, T.M.: Fedcc: Robust federated learning against model poisoning attacks (12 2022). <https://doi.org/10.48550/arXiv.2212.01976>
 12. Khaled, A., Richtárik, P.: Better theory for SGD in the nonconvex world (2020). <https://doi.org/10.48550/arXiv.2002.03329>, <http://arxiv.org/abs/2002.03329>
 13. Lee, S., Kang, Q., Madireddy, S., Balaprakash, P., Agrawal, A., Choudhary, A., Archibald, R., Liao, W.k.: Improving scalability of parallel CNN training by adjusting mini-batch size at run-time. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 830–839 (2019). <https://doi.org/10.1109/BigData47090.2019.9006550>, <https://ieeexplore.ieee.org/abstract/document/9006550>
 14. Niu, F., Recht, B., Re, C., Wright, S.J.: HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent (2011). <https://doi.org/10.48550/arXiv.1106.5730>, <http://arxiv.org/abs/1106.5730>
 15. Robbins, H., Monro, S.: A stochastic approximation method. *The Annals of Mathematical Statistics* **22**(3), 400–407 (1951). <https://doi.org/10.1214/aoms/1177729586>, <http://projecteuclid.org/euclid.aoms/1177729586>
 16. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. *Nature* **323**(6088), 533–536 (oct 1986). <https://doi.org/10.1038/323533a0>
 17. Sa, C.D., Zhang, C., Olukotun, K., Ré, C.: Taming the wild: a unified analysis of HOG WILD! -style algorithms. In: *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’15, vol. 2, pp. 2674–2682. MIT Press (2015)
 18. The PyTorch Foundation: Multiprocessing best practices – pytorch 2.6 documentation. <https://pytorch.org/docs/stable/notes/multiprocessing.html#asynchronous-multiprocess-training-e-g-hogwild>, [Online; Accessed: 12-02-2025]
 19. Tsitsiklis, J., Bertsekas, D., Athans, M.: Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control* **31**(9), 803–812 (1986). <https://doi.org/10.1109/TAC.1986.1104412>, <https://ieeexplore.ieee.org/document/1104412>, conference Name: IEEE Transactions on Automatic Control
 20. Zhou, Y., Yang, J., Zhang, H., Liang, Y., Tarokh, V.: SGD converges to global minimum in deep learning via star-convex path (2019). <https://doi.org/10.48550/arXiv.1901.00451>, <http://arxiv.org/abs/1901.00451>