Towards Staleness-Tolerant Asynchronous Data Processing

Jacob Garby
Supervised by Philippas Tsigas
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden

ACM Reference Format:
Jacob Garby. 2025. Towards Staleness-Tolerant Asynchronous Data Processing. In *Proceedings of 19th ACM International Conference on Distributed and Event-based Systems (DEBS '25).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As data processing is becoming increasingly important for both science and industry, the amount of data required to be processed is also increasing. Common applications which require processing large amounts of data include graph processing and of course machine learning. Much larger datasets as well as more advanced models result in the processing taking ever longer to complete, and requiring more memory to do so.

To combat this, parallel execution is a common optimisation. It is not trivial, however, to ensure that an algorithm run in parallel will be sufficiently scalable to the available number of cores. In an implementation in which all of the threads (let's assume one thread per core) process iterations in lockstep (i.e. a *synchronous* implementation), the synchronisation overhead means that the throughput (total iterations per second) cannot scale even close to linearly with respect to the number of threads.

To improve scalability, it can be preferable to devise *asynchronous* implementations, the simplest case of which has threads working entirely independently of one another, applying updates to a shared global state whenever they finish one iteration. This comes with its own issues, however: asynchronous execution means that the results of individual iterations may be computed based on an outdated view of the global state. Such iterations are referred to as *stale*, and are both extremely common and unavoidable in this type of asynchronous processing. Stale updates have a substantial negative effect on the convergence rate of the algorithm, since the statistical effectiveness of each update is reduced.

DEBS '25, June 10–13, 2025, Gothenburg, Sweden

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Problem Statement

The above discussion applies to any data processing algorithm which consists of a sequence of iterations, each of which uses the result of the previous iteration and produces a result which is used for the next. We aim to address different types of iterative data processing systems, primarily machine learning and graph processing.

Overall, we want to design a system, or a collection of systems, which are able to dynamically manage the trade-offs between asynchronous and synchronous processing. This involves the design of new thread scheduling and synchronisation mechanisms, as well as strategies to choose and evaluate hyperparameters.

Additionally, many such algorithms use a set of *hyperparameters*, and in order to improve their performance we aim to consider how these hyperparameters may be adjusted *at runtime*.

1.2 Questions

As mentioned earlier, many iterative data processing algorithms may be implemented synchronously or asynchronously. There are other possibilities, but most existing works fall into one of those two categories. It is not uncommon to execute asynchronously with some additional constraints on top, for example dynamically adjusting the number of threads at runtime[2], or rejecting updates above a certain staleness [4]. It has also been proposed to dynamically switch between fully synchronous and fully asynchronous [9]. However, not much work exists studying execution modes *somewhere in the middle*, which could potentially work better than either extreme. Hence, we have our first research question:

Research Question A

What is the best way to synchronise and schedule large numbers of threads in a way that is scalable while either limiting the expected update staleness or softening the impact of stale updates? ?

Of course, it is likely that there will be differences in the best thread synchronisation strategy between different iterative data processing tasks. It would be interesting to study these differences and determine if the answer to *Research Question* ***A*** is applicable to all, or many, different such tasks.

Research Question B

What are the significant differences and considerations between different large-scale data processing applications, such as machine learning and graph processing? ?

As mentioned, we want to look into the effect of hyperparameters and determine in what ways adjustment thereof can improve the performance of these algorithms. *Hyperparameter tuning* is a well-studied field, but prior work deals with finding values for hyperparameters that are then constant over the entire execution. This is usually achieved via repeated restarting of the algorithm in question with different parameters, potentially with some early-stopping mechanism to abort hopeless runs. We have seen that many hyperparameters benefit from constant adjustment (e.g. batch size, learning rate, number of threads), and suspect that this is true of many others.

Research Question C

How can we adjust multiple hyperparameters such a way that we produce executions during which the hyperparameters vary? How can we determine whether a given execution state should advance with an adjusted set of hyperparameters, be restarted with a new initial set, or continue with no change? ?

2 Approach

2.1 Background

2.1.1 Stochastic Gradient Descent. Stochastic gradient descent (SGD) is a widely used iterative algorithm for optimising the parameters of some model to minimise a given loss function by repeatedly computing the *gradient* with respect to the current parameters. A simple sequential formulation of this algorithm is as follows:

$$\theta_{i+1} := \theta_i - \eta \nabla L_{B_i}(\theta_i) \tag{1}$$

Where θ_i is the parameter vector of the model following iteration i; $L_{B_i}(\theta_i)$ evaluates the loss of the model given parameters θ_i evaluated on a *minibatch* B_i; and η is the learning rate, controlling the impact of an individual gradient.

Each minibatch B_i consists of a (typically but not always fixed) number of training samples from the entire training dataset.

2.1.2 Parallelising SGD. As mentioned before, a popular approach to speeding up iterative algorithms such as SGD is parallelism. We mainly consider data-parallelism, in which each worker thread independently considers its own minibatch and computes a gradient. Once *all* of the gradients have been computed (for synchronous execution), they can simply be aggregated together (e.g. the average) and used to update θ.

Parallel SGD has traditionally been, and still often is, performed in this synchronous manner. This means that, before aggregating the workers' gradients and updating the model parameters, *all* of the gradient computations must have completed. The primary advantage of this approach is that the statistical semantics are identical to that of purely sequential SGD (Equation 1) (which itself has been proven to converge almost certainly to at least a local minimum, or a global minimum given some fairly weak assumptions [5, 7, 10]).

Unfortunately, synchronous SGD suffers from scalability issues for sufficiently high numbers of threads. Since all threads must finish step i before step $i+1$ can begin (which necessarily relies on the resultant parameters from step i), if some thread completes their iteration before another one does then it can do nothing but sit and wait for the next step to begin. Threads that take longer to finish their steps are referred to as *stragglers*. The impact of stragglers on CPU utilisation and hence on convergence rate is affected by the distribution of step durations: if every step took precisely the same amount of time, the overhead would be minimal, since there would be practically no time spent waiting. This is never the case in practice though; even for a system whose threads are homogeneous in processing speed, small variations cause step durations to vary.

In order to better utilise the CPU, some more recent works as well as machine learning software libraries make use of *asynchronous* processing [1–3, 6, 8]. This relaxes the semantics of the SGD update formula, 1; specifically, threads are allowed to apply their computed gradients to the model independently (without averaging between threads) as soon as they are finished, immediately starting a new step afterwards. In this way, the gradient applied to θ_i in order to compute θ_{i+1} is no longer required to be the gradient of θ_i:

$$\theta_{i+1} := \theta_i - \eta \nabla L_{B_i}(\theta_{i-\tau_i}) \tag{2}$$

Here, τ_i refers to the *staleness* of step i, i.e. the number of versions the parameters θ have gone through since the version that was used to compute this gradient was observed. If τ_i is always 0, then we have an algorithm which is equivalent to the sequential and synchronous formulations. When $\tau_i > 0$, we are applying a gradient which is likely to not be as relevant now as it would have been to the state of the model τ_i steps ago, and so the statistical efficiency of this step is therefore worse.

The benefit of asynchronous processing is that threads no longer have to wait for stragglers, which is the main bottleneck in synchronous processing. As such, when considering synchronous vs asynchronous execution, we are presented with a tradeoff: we can easily achieve scalable throughput (number of steps finished per second) at the expense of degraded statistical efficiency, and similarly we can achieve a relatively large improvement in model accuracy for each completed step at the expense of worsened scalability.

2.2 What We Have Done

Dynamic Parallelism. Recent work [2] has shown that it's not optimal, when executing SGD asynchronously, to always use as many threads as possible. This is due to the fact that the amount of noise introduced by stale updates is correlated with the number of asynchronous threads. Executing with higher than optimal parallelism leads to excessive asynchrony induced noise, which not only removes the throughput benefit of more threads, but can even lead to a worse convergence rate than that achieved by fewer threads. If the parallelism is lower than optimal, it means that we could improve our throughput by some amount without having the convergence rate decrease. Importantly, the optimal number of threads varies over time.

The aforementioned work proposed a probing-based local optimisation approach to follow the time-varying optimal number of threads, but probing all values within a certain range means that either a large proportion of the total execution is spent probing (hence less time spent at the actual optimal value), or the probing range has to be small (meaning that the optional value may drift too quickly for the probing to catch up).

One alternative we considered is a ternary-search approach, which is a popular algorithm for finding the minimum of an unknown function by making a number of "probes". In this case, the unknown function in question is the mapping of parallelism to convergence rate. Theoretically, ternary-search can give us better performance than the existing window probing method, since the optimal can be found in fewer steps, resulting in both a more accurate approximation (due to less change in the model over the probing period) and a higher proportion of the execution being taken up by the execution phases.

An alternative method is to initialise some parameterised function (say, a polynomial), and then – based on a number of probes – adjust the parameters to make the function fit the observed performance of different levels of parallelism. A function can be selected such that its minimum can be directly computed.

A challenge with these approaches is that the model loss we measure during probing can be noisy, when measuring loss based only on the minibatches that were used for training in that period. A workaround is to allow a more accurate loss to be obtained when needed (after each probe) by computing the accuracy across all of the test samples in the dataset. Note that it's okay to use the test data here, rather than the training data, because these samples are not actually being used to update the model.

We found that neither of these approaches provided a significant improvement over prior work, though, most likely due to the aforementioned measurement inaccuracy as well as the limited gains we can expect even given an absolutely optimal sequence of parallelism.

Interval-Asynchrony. In an attempt to take the best of both worlds of asynchronous and synchronous execution (namely high throughput and low staleness, respectively), we've developed a novel execution strategy for parallel SGD, which we refer to as *Interval-Asynchrony*.

Using this method, the whole execution is logically divided into a number of *asynchronous periods*, during which threads are free to begin SGD steps in an asynchronous manner. After a set number of steps have been accepted (both begun and ended) within a certain period, all the threads are then forced to synchronise before the next period can begin. We refer to the period length as y.

The rationale behind this approach is that we are still able to achieve a high throughput due to a good proportion of the total execution consisting of asynchronous execution. The advantage is that we're able to both bound the maximum staleness ($\tau <_\epsilon y$) and shift the distribution of staleness, as shown in Figure 1. This plot shows several interesting trends: most importantly, we can see that by adjusting y we can shift the expected staleness smoothly between that of a synchronous execution ($\tau = 0$) to an asynchronous execution, in which τ follows a Gaussian distribution with $\mu < m$. As we shift away from asynchronous, by reducing y, the left tail of this distribution grows more significant, as we get more and more low-staleness steps for each asynchronous period.

This semi-synchronous execution is performant on its own, using a static value for y, but delivers even better results when y varies over time. This is similar to the time-varying optimal parallelism mentioned earlier. We proposed two different methods for adjusting y: one in which it decays over time at a fixed rate, and one which uses a local probing technique.

We evaluated this algorithm against several baselines, including the aforementioned dynamic parallelism approach, on two popular image recognition datasets (CIFAR10 and CIFAR100). A summary of our results is that we cut the time to train to a given accuracy by 32%, and demonstrated significantly improved scalability up to 128 threads.

Our work on *Interval-Asynchrony* has been accepted for publication at Euro-Par 2025.

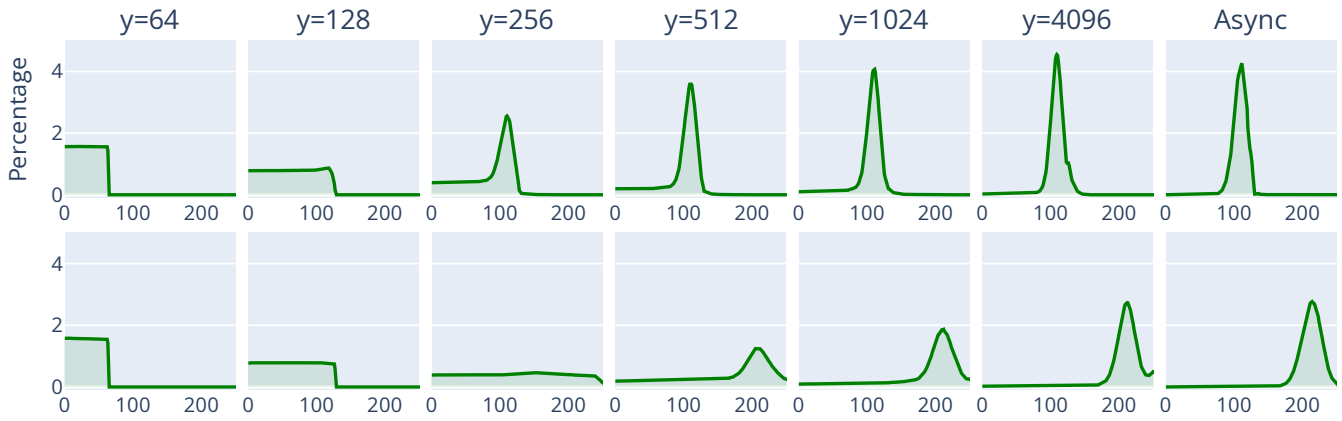

Figure 1. τ-distribution with varying semisync period y. The top row uses 128 threads, and the bottom uses 256.

2.3 Improvements & Future Work

Although we propose an effective window-probing approach for automatically controlling the interval size of *Interval-Asynchronous* SGD, 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, as mentioned in *Research Question* ***C***, 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.

2.4 Conclusion

This project aims to produce algorithms and frameworks which can improve the speed of various data processing algorithms, with a focus on machine learning and graph algorithms. The central theme for such optimisations is trading strict semantics in thread synchronisation and data structures for improved throughput. This trade-off introduces problems with reduced accuracy, which we must find ways to mitigate.

References

[1] Karl Backstrom, Marina Papatriantafilou, and Philippas Tsigas. MindTheStep-AsyncPSGD: Adaptive asynchronous parallel stochastic gradient descent. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 16–25. IEEE, 2019.

[2] Karl Bäckström. Adaptiveness, asynchrony, and resource efficiency in parallel stochastic gradient descent. ISBN: 9789179058555.

[3] Karl Bäckström, Marina Papatriantafilou, and Philippas Tsigas. ASAP.SGD: Instance-based adaptiveness to staleness in asynchronous SGD. In *Proceedings of the 39th International Conference on Machine Learning*, pages 1261–1276. PMLR, 2022. ISSN: 2640-3498.

[4] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[5] Ahmed Khaled and Peter Richtárik. Better theory for SGD in the nonconvex world, 2020.

[6] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent.

[7] Herbert Robbins and Sutton Monro. A stochastic approximation method. 22(3):400–407, 1951.

[8] 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].

[9] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: time to fuse for distributed graph-parallel computation. *SIGPLAN Not.*, 50(8):194–204, jan 2015.

[10] Yi Zhou, Junjie Yang, Huishuai Zhang, Yingbin Liang, and Vahid Tarokh. SGD converges to global minimum in deep learning via star-convex path, 2019.