Rebuttal NeurIPS 2019 – Paper ID: 319
SySCD: A System-Aware Algorithm for Machine Learning

We thank the reviewers for their comments and suggestions. In the following we will first detail what we commit to improve in a final version and then address each reviewer's comments individually.

- We are happy to add more guidance on the practical choice of hyperparameters used for our experiments to the main part of the paper and move Remark 1 from the appendix to Section 4, as suggested by **Reviewer 3**.
- We will also add additional experiments with more threads and corresponding discussion to the final version of the paper, as requested by **Reviewer 3**. See the dedicated paragraph below for some initial results and a corresponding discussion.
- We will not be able to open source the production code of our algorithm as requested by **Reviewer 2** and we feel that a prototypical version of the algorithm would not add much value. This algorithm is part of a free, commercial software package. Thus, what we can and will do is to provide detailed instructions on how to use the software and reproduce all our results.

@Reviewer 1: We thank Reviewer 1 for the feedback. However, we do not agree that the contribution of our paper can be reduced to a set of *practical tricks* and we would like to expand on this: Our contribution is a novel, principled algorithm with convergence guarantees whose design is motivated by a systematic analysis of the performance bottlenecks of a state-of-the-art coordinate descent (CD) implementation. What is unique to our work is that we design a parallel CD algorithm from a system point of view by incorporating several critical performance bottlenecks into the algorithm design. Previous CD methods had been designed with an idealized system model in mind which did not take memory access bottlenecks, non-uniformity across cores, or cache assess patterns into consideration. We show that this inherently limits their potential performance and scalability on real systems. The algorithmic modifications we propose can address these limitations very effectively (up to 12 $\times$ faster runtime) and, equally important, they can be shown to lead to a new algorithm with provable convergence guarantees. These guarantees are stronger than those of popular asynchronous methods, such as (Hsieh et al., 2015a), which often fail to convergence when scaled to many threads because their analysis relies on a strong assumptions on the delay between individual updates. This convergence issue is not only an artifact of their analysis but also observed in practice, see Fig 1(a) in our paper. Our algorithm, on the other hand, inherits strong convergence guarantees of popular distributed methods, is thus guaranteed to converge for any degree of parallelism and, at the same time, demonstrates superior performance.

In particular, we feel that the issue of coherence traffic which can be solved by transferring ideas from distributed learning, together with our novel dynamic repartitioning scheme of data across threads (as described in §4.2), in order to counter resulting convergence slow downs, could potentially lead to a rethinking of how we design parallel solvers.

@Reviewer 2: We want to thank Reviewer 2 for appreciating our work and its contributions.

@Reviewer 3: We thank Reviewer 3 for the positive assessment and the suggestions to improve our manuscript. We would like to further discuss the requested experiment: In all of our experiments in the paper, we have scaled the number of threads up to the maximum number of physical cores per machine. In all experiments simultaneous multi-threading (SMT) was disabled, since in practice we often find enabling SMT leads to worse overall performance. We do not have access to any machines with more cores than those presented in the paper (40 for the P9 and 32 for the x86 node). However, for the purpose of addressing your question, we re-ran the experiment on the P9 using 4 hardware threads per core (SMT4). The results are shown in Figure 1 where we plot the speed-up in terms of time-per-epoch as a function of the number of threads. As expected, we see linear scaling up to the number of physical CPU cores (in this case 40), after which we start to see diminishing returns due to the inherent inefficiency of SMT4 operation.

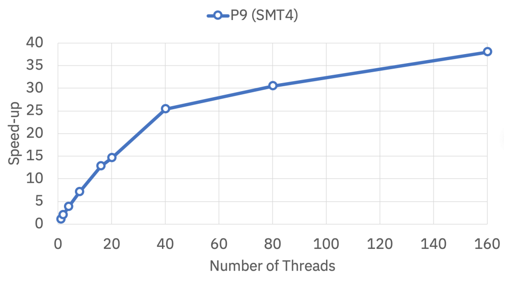

Figure 1: Scaling behaviour of time-per-epoch when using SMT4 mode.