SySCD: A System-Aware Parallel Coordinate Descent Algorithm

Nikolas Ioannou[*]
IBM Research
Zurich, Switzerland
nio@zurich.ibm.com

Celestine Mendler-Dünner[*†]
UC Berkeley
Berkeley, California
mendler@berkeley.edu

Thomas Parnell
IBM Research
Zurich, Switzerland
tpa@zurich.ibm.com

Abstract

In this paper we propose a novel parallel stochastic coordinate descent (SCD) algorithm with convergence guarantees that exhibits strong scalability. We start by studying a state-of-the-art parallel implementation of SCD and identify scalability as well as system-level performance bottlenecks of the respective implementation. We then take a principled approach to develop a new SCD variant which is designed to avoid the identified system bottlenecks, such as limited scaling due to coherence traffic of model sharing across threads, and inefficient CPU cache accesses. Our proposed system-aware parallel coordinate descent algorithm (SySCD) scales to many cores and across numa nodes, and offers a consistent bottom line speedup in training time of up to $\times 12$ compared to an optimized asynchronous parallel SCD algorithm and up to $\times 42$, compared to state-of-the-art GLM solvers (scikit-learn, Vowpal Wabbit, and H2O) on a range of datasets and multi-core CPU architectures.

1 Introduction

Today's individual machines offer dozens of cores and hundreds of gigabytes of RAM that can, if used efficiently, significantly improve the training performance of machine learning models. In this respect parallel versions of popular machine learning algorithms such as stochastic gradient descent (Recht et al., 2011) and stochastic coordinate descent (Liu et al., 2015; Hsieh et al., 2015a; Richtarik & Takac, 2016b) have been developed. These methods either introduce asynchronicity to the sequential algorithms, or they use a mini-batch approach, in order to enable parallelization and better utilization of compute resources. However, all of these methods treat machines as a simple, uniform, collection of cores. This is far from reality. While modern machines offer ample computation and memory resources, they are also elaborate systems with complex topologies, memory hierarchies, and CPU pipelines. As a result, maximizing the performance of parallel training requires algorithms and implementations that are aware of these system-level characteristics and respect their bottlenecks.

Setup. In this work we focus on the training of generalized linear models (GLMs). Our goal is to efficiently solve the following partially separable convex optimization problem using the full compute power available in modern CPUs:

$$\min_{\boldsymbol{\alpha}\in\mathbb{R}^n} F(\boldsymbol{\alpha}) \quad \text{where} \quad F(\boldsymbol{\alpha}) := f(A\boldsymbol{\alpha}) + \sum_i g_i(\alpha_i). \tag{1}$$

The model vector $\boldsymbol{\alpha} \in \mathbb{R}^n$ is learned from the training data $A \in \mathbb{R}^{d\times n}$, the function f is convex and smooth, and g_i are general convex functions. The objective (1) covers primal as well as dual formulations of many popular machine learning models which are widely deployed in industry

[*]Equal contribution.
[†]Work conducted while at IBM Research, Zurich.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

(Kaggle, 2017). For developing such a system-aware training algorithm we will build on the popular stochastic coordinate descent (SCD) method (Wright, 2015; Shalev-Shwartz & Zhang, 2013). We first identify its performance bottlenecks and then propose several algorithmic optimizations to alleviate them.

Contributions. The main contributions of this work can be summarized as follows:

1. We propose SySCD, the first system-aware coordinate descent algorithm that is optimized for
   - *cache access patterns:* We introduce *buckets* to design data access patterns that are well aligned with the system architecture.
   - *thread scalability:* We increase data parallelism across worker threads to avoid data access bottlenecks and benefit from the buckets to reduce permutation overheads.
   - *numa-topology:* We design a hierarchical numa-aware optimization pattern that respects non-uniform data access delays of threads across numa-nodes.
2. We give convergence guarantees for our optimized method and motivate a *dynamic re-partitioning* scheme to improve its sample efficiency.
3. We evaluate the performance of SySCD on diverse datasets and across different CPU architectures, and we show that SySCD drastically improves the implementation efficiency and the scalability when compared to state-of-the-art GLM solvers (scikit-learn Pedregosa et al. (2011), Vowpal Wabbit Langford (2007), and H2O The H2O.ai team (2015)), resulting in $\times 12$ faster training on average.

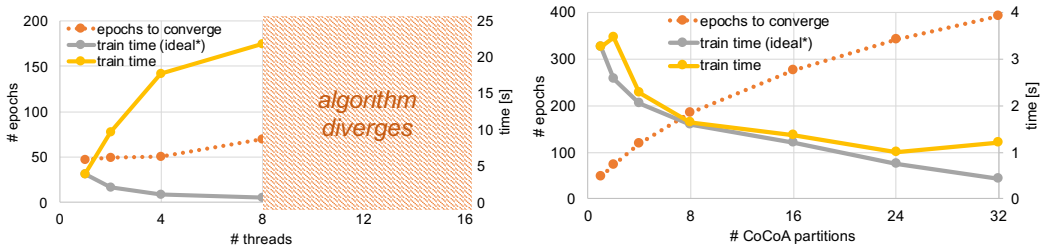
2 Background

Stochastic coordinate descent (SCD) methods (Wright, 2015; Shalev-Shwartz & Zhang, 2013) have become one of the key tools for training GLMs, due to their ease of implementation, cheap iteration cost, and effectiveness in the primal as well as in the dual. Their popularity has been driving research beyond sequential stochastic solvers and a lot of work has been devoted to map these methods to parallel hardware. We will give a short summary in the following, putting emphasis on the assumptions made on the underlying hardware.

Previous works on *parallel* coordinate descent (Hsieh et al., 2015a; Parnell et al., 2017; Richtarik & Takac, 2016b; Liu et al., 2015) assume that parallel processes are homogeneous and data as well as model information resides in shared memory which is accessible by all processes. Building on these assumptions, Hsieh et al. (2015a); Liu et al. (2015); Liu & Wright (2015) propose asynchronous methods for scaling up SCD: the model resides in shared memory and all processes simultaneously read and write this model vector. A fundamental limitation of such an approach is that its convergence relies on the fact that the model information used to compute each update is not too stale. Thus, asynchronous algorithms are prone to diverge when scaled up to a large number of processes. In addition, the heavy load on the model vector can cause significant runtime delays. Both limitations are more pronounced for dense data, thus we use a dense synthetic dataset to illustrate these effects in Fig 1a; the orange, dashed line shows that convergence suffers from staleness, the gray line shows the respective runtime assuming perfect thread scalability and the yellow lines depicts the measured runtime. The algorithm diverges when scaled across more than 8 threads. Taking another route, Richtarik & Takac (2016b); Bradley et al. (2011) propose a synchronous approach for parallelizing SCD. Such methods come with more robust convergence properties. However, depending on the inherent separability of f, the potential of acceleration can be small. For synthetic, well separable problems, mini-batch SDCA proposed by Richtarik & Takac (2016b) show almost linear scaling, whereas for correlated objectives or dense datasets, the potential for acceleration, as given in their theory diminishes. In addition, updates to the shared vector in the synchronous setting are guaranteed to conflict across parallel threads – mini-batch SDCA uses atomic operations[3] to serialize those updates; this does not scale as the thread count increases, and especially not in numa machines. We have applied this method to the same synthetic example used in Fig 1 and we observed virtually no speedup (5%) when using 32 threads.

Orthogonal to parallel methods, *distributed* coordinate-based methods have also been the focus of many works, including (Yang, 2013; Ma et al., 2015; Richtarik & Takac, 2016a; Dünner et al.,

[3]code available at `https://code.google.com/archive/p/ac-dc/downloads`



(a) PASSCoDe (b) CoCoA

Figure 1: Scalability of existing methods: Training of a logistic regression classifier on a synthetic dense dataset with 100k training examples and 100 features – (a) training using PASSCoDe-wild (Hsieh et al., 2015a) and (b) training using CoCoA (Smith et al., 2018) deployed across threads. Details can be found in the appendix.

2018; Smith et al., 2018; Lee & Chang, 2018). Here the standard assumption on the hardware is that processes are physically separate, data is partitioned across them, and communication is expensive. To this end, state-of-the-art distributed first- and second-order methods attempt to pair good convergence guarantees with efficient distributed communication. However, enabling this often means trading convergence for data parallelism (Kaufmann et al., 2018). We have illustrated this tradeoff in Fig 1b where we employ CoCoA Smith et al. (2018) across threads; using 32 threads the number of epochs is increased by $\times 8$ resulting in a speedup of $\times 4$ assuming perfect thread scalability. This small payback makes distributed algorithms generally not well suited to achieving acceleration; they are primarily designed to enable training of large datasets that do not fit into a single machine (Smith et al., 2018).

The fundamental trade-off between statistical efficiency (how many iterations are needed to converge) and hardware efficiency (how efficient they can be executed) of deploying machine learning algorithms on modern CPU architectures has previously been studied in Zhang & Ré (2014). The authors identified data parallelism as a critical tuning parameter and demonstrate that its choice can significantly affect performance of any given algorithm.

The goal of this work is to go one step further and enable better trade-offs by directly incorporate mitigations to critical system-level bottlenecks into the algorithm design. We exploit the shared memory performance available to worker threads within modern individual machines to enable new algorithmic features that improve scalability of parallel coordinate descent, while at the same time maintaining statistical efficiency.

3 Bottleneck Analysis

We start by analyzing state-of-the-art implementations of sequential and parallel coordinate descent to identify bottlenecks and scalability issues. For the parallel case, we use an optimized implementation of PASSCoDe (Hsieh et al., 2015a) as the baseline for this study, which is vectorized and reasonably efficient. The parallel algorithm operates in epochs and repeatedly divides the n shuffled coordinates among the P parallel threads. Each thread then operates asynchronously: reading the current state of the model $\boldsymbol{\alpha}$, computing an update for this coordinate and writing out the update to the model α_j as well as the shared vector $\mathbf{v}$. The auxiliary vector $\mathbf{v} := A\boldsymbol{\alpha}$ is kept in memory to avoid recurring computations. Write-contention on $\mathbf{v}$ is solved opportunistically in a wild fashion, which in practice is the preferred approach over expensive locking (Parnell et al., 2017; Hsieh et al., 2015a). The parallel SCD algorithm is stated in Appendix A.1 for completeness.

One would expect that, especially for large datasets (e.g., datasets that do not fit in the CPU caches), the runtime would be dominated by (a) the time to compute the inner product required for the coordinate update computation and (b) retrieving the data from memory. While these bottlenecks can generally not be avoided, our performance analysis identified four other bottlenecks that in some cases vastly dominate the runtime:

(B1) Access to model vector. When the model vector $\boldsymbol{\alpha}$ does not fit in the CPU cache, a lot of time is spend in accessing the model. The origin of this overhead is the random nature of the accesses to $\boldsymbol{\alpha}$, there is very little cache line re-use: a cache line is brought from memory (64B or 128B), out of which only 8B are used. This issue affects both the parallel and the sequential implementation. For

Algorithm 1 SySCD for minimizing (1)

Input: Training data matrix $A = [\mathbf{x}_1, ..., \mathbf{x}_n] \in \mathbb{R}^{d \times n}$
Initialize model $\boldsymbol{\alpha}$ and shared vector $\mathbf{v} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i$.
Partition coordinates into buckets of size B.
Partition buckets across numa nodes according to $\{\mathcal{P}_k\}_{k=1}^{K}$.
for $t = 1, 2, \ldots, T_1$ **do**
    parfor $k = 1, 2, \ldots, K$ across numa nodes **do**
        $\mathbf{v}_k = \mathbf{v}$
        for $t = 1, 2, \ldots, T_2$ **do**
            create random partitioning of local buckets across threads $\{\mathcal{P}_{k,p}\}_{p=1}^{P}$
            parfor $p = 1, 2, \ldots, P$ across threads **do**
                $\mathbf{v}_p = \mathbf{v}_k$
                for $j = 1, 2, \ldots, T_3$ **do**
                    randomly select a bucket $\mathcal{B} \in \mathcal{P}_{k,p}$
                    for $i = 1, 2, \ldots, T_4$ **do**
                        randomly sample a coordinate j in bucket $\mathcal{B}$
                        $\delta = \arg\min_{\delta \in \mathbb{R}} \bar{f}(\mathbf{v}_p + \mathbf{x}_j \delta) + \bar{g}_j(\alpha_j + \delta)$
                        $\alpha_j = \alpha_j + \delta$
                        $\mathbf{v}_p = \mathbf{v}_p + \delta \mathbf{x}_j$
                    end for
                end for
            end parfor
            $\mathbf{v}_k = \mathbf{v}_k + \sum_p (\mathbf{v}_p - \mathbf{v}_k)$
        end for
    end parfor
    $\mathbf{v} = \mathbf{v} + \sum_k (\mathbf{v}_k - \mathbf{v})$
end for

the latter this bottleneck dominates and we found that, by accessing the model in a sequential manner, we can reduce the runtime by $\times 2$.

(B2) Access to the shared vector. For the parallel implementation, we found that writing the updates to the shared vector $\mathbf{v}$ across the different threads was the main bottleneck. On top of dominating the runtime, staleness in the shared vector can also negatively impact convergence.

(B3) Non-uniform memory access. When the parallel implementation is deployed across multiple numa nodes, bottleneck (B2) becomes catastrophic, often leading to divergence of the algorithm (see Fig. 1a). This effect can be explained by the fact that the inter-node delay when writing updates is far more pronounced than the intra-node delay.

(B4) Shuffling coordinates. A significant amount of time is spent permuting the coordinates before each epoch in both the parallel and the sequential case. For the latter, we found that by removing the permutation, effectively performing cyclic coordinate descent, we could achieve a further 20% speed-up in runtime on top of removing (B1).

4 Algorithmic Optimizations

In this section we present the main algorithmic optimizations of our new training algorithm which are designed to simultaneously address the system performance bottlenecks (B1)-(B4) identified in the previous section as well as the scalability issue demonstrated in Fig. 1b. Our system-aware parallel training algorithm (SySCD) is summarized in Alg. 1 and its convergence properties are analyzed in Sec. 4.4. The following subsections will be accompanied by experimental results illustrating the effect of the individual optimizations. They show training of a logistic regression classifier on the criteo-kaggle dataset (Criteo-Labs, 2013) on a 4 node system with 8 threads per numa node (for the experimental setup, see Sec 5) . Results for two additional datasets can be found in the appendix.

4.1 Bucket Optimization

We have identified the cache line access pattern (B1) and the random shuffling computation (B4) as two critical bottlenecks in the sequential as well as the parallel coordinate descent implementation. To alleviate these in our new method, we introduce the concept of *buckets*: We partition the coordinates

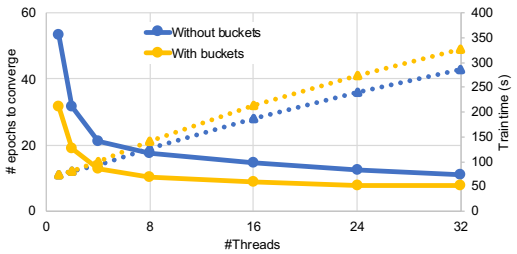


Figure 2: Bucket Optimization: Gain achieved by using buckets. Solid lines indicate time, and dashed-lines depict number of epochs.

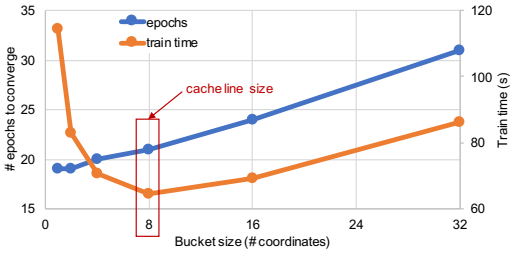


Figure 3: Sensitivity analysis on the bucket size w.r.t. training time and epochs for convergence.

and the respective columns $\mathbf{x}_i$ of A into buckets and then train a bucket of B consecutive coordinates at a time. Thus, instead of randomizing all coordinates at once, the order in which buckets are processed is randomized, as well as the order of coordinates within a bucket. This modification to the algorithm improves performance in several ways; (i) the model vector $\boldsymbol{\alpha}$ is accessed in a cache-line efficient manner, (ii) the computation overhead of randomizing the coordinates is reduced by $1/B$, and (iii) CPU prefetching efficiency on accessing the different coordinates of $\mathbf{x}_i$ is implicitly improved. For our test case this optimization leads to an average speedup of 63% with only a small toll on convergence, as depicted in Fig. 2.

The bucket size B will appear in our convergence rate (Theorem 1) and can be used to control the scope of the randomization to trade-off between convergence speed and implementation efficiency. We illustrate the sensitivity of our algorithm to the bucket size B in Fig. 3. We see that the bottom line training time decreases significantly across the board by introducing buckets. The optimal bucket size in Fig. 3 is eight which coincides with the cache line size of the CPU with respect to the model vector α accesses. Thus we do not need to introduce an additional hyperparameter and can choose the bucket size B at runtime based on the cache line size of the CPU, using linux `sysfs`.

4.2 Increasing Data Parallelism

Our second algorithmic optimization mitigates the main scalability bottleneck (B2) of the asynchronous implementation: write-contention on the shared vector $\mathbf{v}$. We completely avoid this write-contention by replicating the shared vector across threads to increase data parallelism. To realize this data parallelism algorithmically we transfer ideas from distributed learning. In particular, we employ the CoCoA method (Smith et al., 2018) and map it to a parallel architecture where we partition the (buckets of) coordinates across the threads and replicate the shared vector in each one. The global shared vector is therefore only accessed at coarse grain intervals (e.g., epoch boundaries), where it is updated based on the replicas and broadcasted anew to each thread. Similar to CoCoA we can exploit the typical asymmetry of large datasets and map our problem such that the shared vector has dimensionality $d = \min(\#\text{features}, \#\text{examples})$.

We have seen in Sec 2 that distributed algorithms such as CoCoA are generally not suited to achieve significant acceleration with parallelism. This behavior of distributed methods is caused by the static partitioning of the training data across workers which increases the epochs needed for convergence (Smith et al., 2018; Kaufmann et al., 2018) (e.g., see Fig 1b). To alleviate this issue, we propose to combine our multi-threaded implementation with a *dynamic re-partitioning* scheme. That is, we shuffle all the (buckets of) coordinates at the beginning of each local optimization round (Step 9 of Alg. 1), and then, each thread picks a different set of buckets each time. Such a re-partitioning approach is very effective for convergence when compared to a default static partitioning, as depicted in Fig. 4. It reduces the number of epochs by 54% at the cost of only a small implementation overhead. To the best of our knowledge we are the first to consider such a re-partitioning approach in combination with distributed methods and demonstrate a practical scenario where it pays off – in a classical distributed setting the cost of re-partitioning would be unacceptable.

The intuition behind this approach is the following: In CoCoA (Smith et al., 2018) a block-separable auxiliary model of the objective is constructed. In this model the correlation matrix $M = A^\top A$ is approximated by a block-diagonal version where the blocks are aligned with the partitioning of the data. This allows one to decouple local optimization tasks. However, this also means that correlations

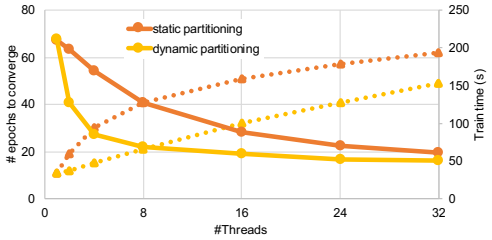


Figure 4: Static and dynamic partitioning: Gain achieved by dynamic re-partitioning. Solid lines indicate time, and dashed-lines depict number of epochs.

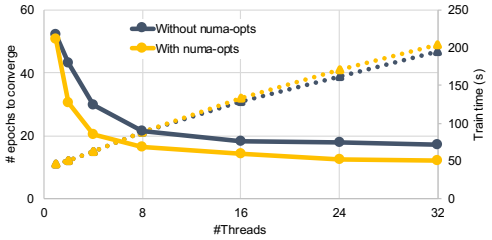


Figure 5: Numa-level Optimizations: Gain achieved by numa-awareness. Solid lines indicate time, and dashed-lines depict number of epochs.

between data points on different worker nodes are not considered. A dynamic re-partitioning scheme has the effect of choosing a different block diagonal approximation of M in each step. By randomly re-partitioning coordinates, the off-diagonal elements of M are sampled uniformly at random and thus in expectation a good estimate of M is used. A rigorous analysis of this effect would be an interesting study for future work. However, note that SySCD inherits the strong convergence guarantees of the CoCoA method, independent of the partitioning, and can thus be scaled up safely to a large number of cores in contrast to our asynchronous reference implementation.

4.3 Numa-Level Optimizations

Subsequently, we focus on optimizations related to the numa topology in a multi-numa node system. Depending on the numa node where the data resides and the node on which a thread is running, data access performance can be non-uniform across threads. As we have seen in Fig. 1b and discussed in Sec. 3 this amplifies bottleneck (B3). To avoid this in SySCD, we add an additional level of parallelism and treat each numa node as an independent training node, in the distributed sense. We then deploy a hierarchical scheme: we statically partition the buckets across the numa nodes, and within the numa nodes we use the dynamic re-partitioning scheme introduced in Sec 4.2. We exploit the fact that the training dataset is read-only and thus it does not incur expensive coherence traffic across numa nodes. We do not replicate the training dataset across the nodes and the model vector $\boldsymbol{\alpha}$ is local to each node which holds the coordinates corresponding to its partition $\mathcal{P}_k$. Crucially, each node holds its own replica of the shared vector, which is reduced across nodes periodically. The frequency of synchronization can be steered in Alg. 1 by balancing the total number of updates between T_1 and T_2. This again offers a trade off between fast convergence (see Theorem 1) and implementation efficiency. This hierarchical optimization pattern that reflects the numa-topology results in a speedup of 33% over a numa-oblivious implementation, as shown in Fig 5. To avoid additional hyperparameters, we dynamically detect the numa topology of the system, as well as the number of physical cores per node, using `libnuma` and the `sysfs` interface. If the number of threads requested by the user is less or equal to the number of cores in one node, we schedule a single node solver. We detect the numa node on which the dataset resides using the `move_pages` system call.

4.4 Convergence Analysis

We derive an end-to-end convergence rate for SySCD with all its optimizations as described in Alg. 1. We focus on strongly convex g_i while every single component of SySCD is also guaranteed to converge for general convex g_i, see Remark 2 in the Appendix.

Theorem 1. *Consider Algorithm 1 applied to* (1). *Assume f is γ-smooth and g_i are μ-strongly convex functions. Let K be the number of numa nodes and P the number of threads per numa node. Let B be the bucket size. Denote T_4 the number of SDCA updates performed on each bucket, let T_3 be the number of buckets processed locally in each iteration and let T_2 be the number of communication rounds performed independently on each numa node before global synchronization. Then, after T_1 outer rounds the suboptimality $\varepsilon = F(\boldsymbol{\alpha}) - \min_{\boldsymbol{\alpha}} F(\boldsymbol{\alpha})$ can be bounded as*

$$\mathrm{E}[\varepsilon] \leq \left(1 - \left[1 - \left(1 - (1-\theta)\frac{\gamma K c_A + \mu}{\gamma K P c_A + \mu}\right)^{T_2}\right] \frac{\mu}{\mu + K\gamma c_A}\right)^{T_1} \varepsilon_0 \tag{2}$$

where $c_A := \|A\|_{op}$ *and*

$$\theta = \left(1 - \left[1 - \left(1 - \frac{1}{n}\frac{\mu}{\mu + \gamma K P}\right)^{T_4}\right]\frac{B}{n}\frac{\mu}{\mu + c_A \gamma K P}\right)^{T_3}. \tag{3}$$

Proof Sketch. To derive a convergence rate of Alg. 1 we start at the outer most level. We focus on the two nested for-loops in Step 6 and Step 10 of Alg. 1. They implement a nested version of CoCoA where the outer level corresponds to CoCoA across numa nodes and the inner level to CoCoA across threads. The number of inner iterations T_2 is a hyper-parameter of our algorithm steering the accuracy to which the local subproblem assigned to each numa node is solved. Convergence guarantees for such a scheme can be derived from a nested application of (Smith et al., 2018, Theorem 3) similar to (Dünner et al., 2018, Appendix B). Subsequently, we combine this result with the convergence guarantees of the local solver used by each thread. This solver, implementing the bucketing optimization, can be analyzed as a randomized block coordinate descent method, similar to (Dünner et al., 2017, Theorem 1), where each block corresponds to a bucket of coordinates. Each block update is then computed using SDCA (Shalev-Shwartz & Zhang, 2013). Again, the number of coordinate descent steps T_4 forms a hyper-parameter to steer the accuracy of each block update. Combining all these results in a nested manner yields the convergence guarantee presented in Theorem 1. We refer to the Appendix A.3 for a detailed proof. □

5 Evaluation

In this section, we evaluate the performance of SySCD in two different single-server multi numa-node environments. We first analyze the scalability of our method and the performance gains achieved over the reference implementation. Then, we compare SySCD with other state-of-the-art GLM solvers available in scikit-learn (Pedregosa et al., 2011)(0.19.2), H2O (The H2O.ai team, 2015) (3.20.0.8), and Vowpal Wabbit (VW) (Langford, 2007) (commit: `5b020c4`). We take logistic regression with L_2 regularization as a test case. We use two systems with different CPU architectures and numa topologies: a 4-node Intel Xeon (E5-4620) with 8 cores and 128GiB of RAM in each node, and a 2-node IBM POWER9 with 20 cores and 512GiB in each node, 1TiB total. We evaluate on three datasets: (i) the sparse dataset released by Criteo Labs as part of their 2014 Kaggle competition (Criteo-Labs, 2013) (criteo-kaggle), (ii) the dense HIGGS dataset (Baldi et al., 2014) (higgs), and (iii) the dense epsilon dataset from the PASCAL Large Scale Learning Challenge (Epsilon, 2008) (epsilon). Results on epsilon and additional details can be found in the appendix.

Remark 1 (Hyperparameters)**.** *The hyperparameters* T_2, T_3, T_4 *in Alg 1 can be used to optimally tune SySCD to different CPU architectures. However, a good default choice is*

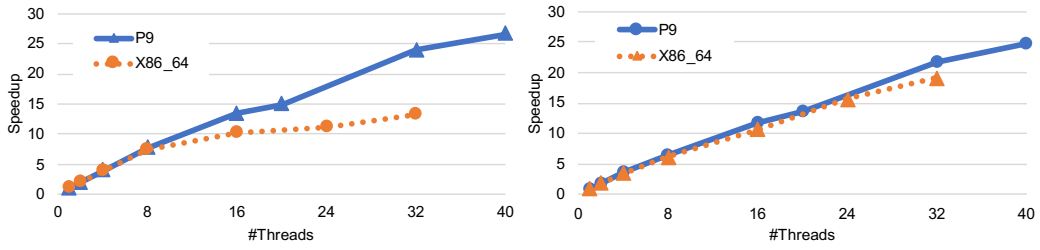
$$T_4 = B, \qquad T_3 = \frac{n}{PB} \qquad T_2 = 1 \tag{4}$$

such that one epoch (n *coordinate updates) is performed across the threads before each synchronization step. We will use these values for all our experiments and did not further tune our method. Further, recall that the bucket size* B *is set to be equal to the cache line size of the CPU and the number of numa nodes* K *as well as the number of threads* P *is automatically detected.*

5.1 Scalability

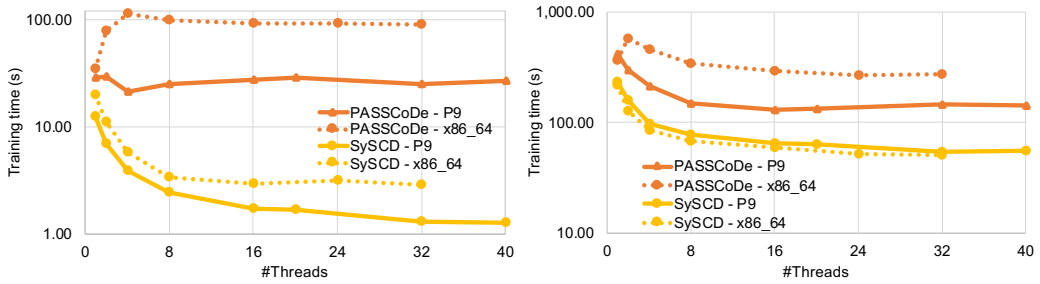
We first investigate the thread scalability of SySCD. Results, showing the speedup in time per epoch (an epoch corresponds to n coordinate updates) over the sequential version, are depicted in Fig 6. We see that SySCD scales almost linearly across the two systems and thus the main scalability bottleneck (B2) of our reference implementation is successfully mitigated. The 4 node system show a slightly lower absolute speedup beyond 1-node (8 threads), which is expected due to the higher overhead when accessing memory on different numa nodes compared to the 2-node system.

Note that in our experiments we disable simultaneous multi-threading (SMT), since in practice we often find enabling SMT leads to worse overall performance. Therefore, the maximal thread count corresponds to the number of physical cores present in the machine. In order to illustrate how SySCD scales when the number of threads exceeds the number of physical cores, we enabled SMT4 (4



(a) higgs (b) criteo-kaggle

Figure 6: Strong thread scalability of SySCD w.r.t runtime per epoch with increasing thread counts for the two different systems: a 2 node (P9) and a 4 node (X86_64) machine.



(a) higgs (b) criteo-kaggle

Figure 7: Training time w.r.t. thread count for the reference PASSCoDe and our optimized SySCD implementation on a 2 node (P9) and a 4 node (X86_64) machine.

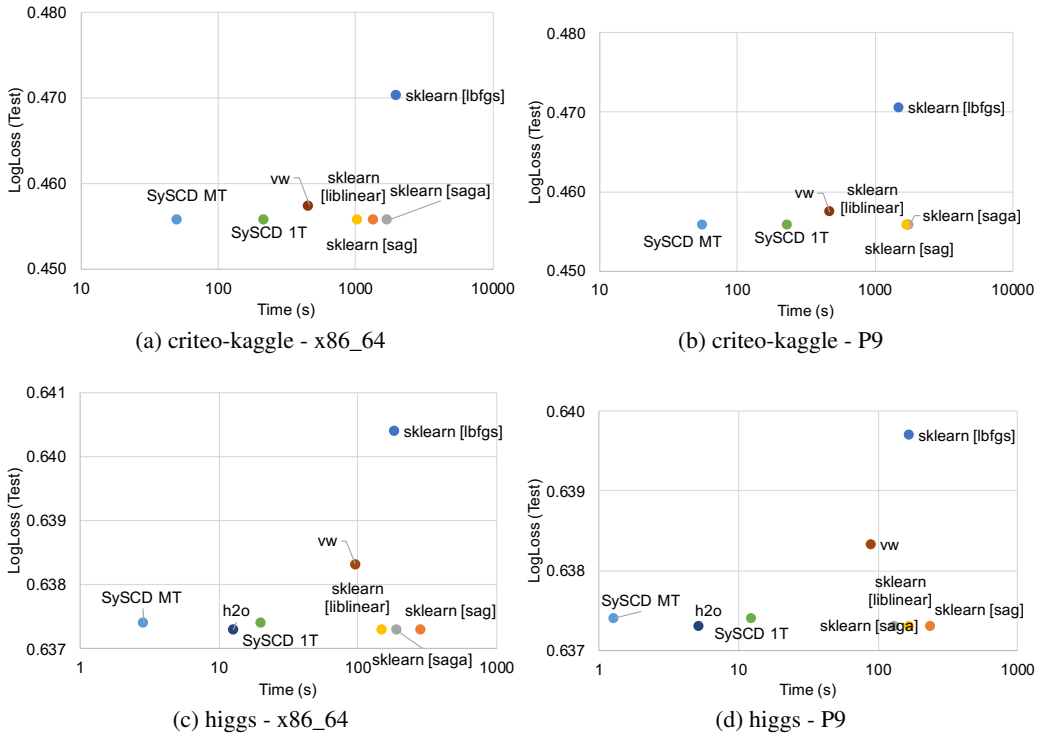
hardware threads per core) on the P9 machine and re-ran the experiment from Fig. 6b. The results are shown in Figure 16 in the appendix. As expected, we see linear scaling up to the number of physical CPU cores (in this case 40), after which we start to see diminishing returns due to the inherent inefficiency of SMT4 operation. We thus recommend disabling SMT when deploying SySCD.

5.2 Bottom Line Performance

Second, we compare the performance of our new SySCD algorithm to the PASSCoDe baseline implementation. Convergence is declared if the relative change in the learned model across iterations is below a threshold. We have verified that all implementations exhibit the same test loss after training, apart from the PASSCoDe implementation which can converge to an incorrect solution when using many threads (Hsieh et al., 2015b). Fig 7 illustrates the results for two different systems. Comparing against PASSCoDe with the best performing thread count, SySCD achieves a speedup of $\times 5.4$ (P9) and $\times 4.8$ (X86_64) on average across datasets. The larger performance improvement observed for the 2-node system relative to the 4-node system, in particular on the higgs dataset, can be attributed to the increased memory bandwidth.

5.3 Comparison with sklearn, VW, and H2O

We finally compare the performance of our new solver against widely used frameworks for training GLMs. We compare with scikit-learn (Pedregosa et al., 2011), using different solvers (`liblinear`, `lbfgs`, `sag`), with H2O (The H2O.ai team, 2015), using its multi-threaded `auto` solver and with VW (Langford, 2007), using its default solver. Care was taken to ensure that the regularization strength was equivalent across all experiments, and that the reported time did not include parsing of text and loading of data. Results showing training time against test loss for the different solvers, on the two systems, are depicted in Fig 8. We add results for SySCD with single (*SySCD 1T*) and maximum (*SySCD MT*) thread counts. Overall *SySCD MT* is over $\times 10$ faster, on average, than the best performing alternative solver. The best competitor is VW for criteo-kaggle and H2O for higgs. H20 results are not shown in Fig 8a and 8b because we could not train the criteo-kaggle dataset in a reasonable amount of time ($>$ 16 hours), even by using the max_active_predictors option.



(a) criteo-kaggle - x86_64

(b) criteo-kaggle - P9

(c) higgs - x86_64

(d) higgs - P9

Figure 8: Comparing a single- and multi-threaded implementations of SySCD against state-of-the-art GLM solvers available in scikit-learn, VW, and H2O.

6 Conclusion

We have shown that the performance of existing parallel coordinate descent algorithms which assume a simplistic model of the parallel hardware, significantly suffers from system bottlenecks which prevents them from taking full advantage of modern CPUs. In this light we have proposed SySCD, a new system-aware parallel coordinate descent algorithm that respects cache structures, data access patterns and numa topology of modern systems to improve implementation efficiency and exploit fast data access by all parallel threads to reshuffle data and improve convergence. Our new algorithm achieves a gain of up to $\times 12$ compared to a state-of-the-art system-agnostic parallel coordinate descent algorithm. In addition, SySCD enjoys strong scalability and convergence guarantees and is thus suited to be deployed in production.

References

Baldi, P., Sadowski, P., and Whiteson, D. O. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.

Bradley, J. K., Kyrola, A., Bickson, D., and Guestrin, C. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pp. 321–328, 2011. ISBN 978-1-4503-0619-5.

Criteo-Labs. Terabyte click logs dataset. `http://labs.criteo.com/2013/12/download-terabyte-click-logs/`, 2013. Online; Accessed: 2018-01-25.

Dünner, C., Forte, S., Takac, M., and Jaggi, M. Primal-dual rates and certificates. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 783–792, New York, New York, USA, 20–22 Jun 2016. PMLR.

Dünner, C., Parnell, T., and Jaggi, M. Efficient use of limited-memory accelerators for linear learning on heterogeneous systems. In *Advances in Neural Information Processing Systems 30*, pp. 4258–4267. 2017.

Dünner, C., Lucchi, A., Gargiani, M., Bian, A., Hofmann, T., and Jaggi, M. A distributed second-order algorithm you can trust. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pp. 1358–1366, 2018.

Dünner, C., Parnell, T., Sarigiannis, D., Ioannou, N., Anghel, A., Ravi, G., Kandasamy, M., and Pozidis, H. Snap ML: A Hierarchical Framework for Machine Learning. In *Advances in Neural Information Processing Systems 31*, pp. 250–260. 2018.

Epsilon. Pascal large scale learning challenge. `http://www.k4all.org/project/large-scale-learning-challenge`, 2008.

Hsieh, C.-J., Yu, H.-F., and Dhillon, I. Passcode: Parallel asynchronous stochastic dual co-ordinate descent. In *International Conference on Machine Learning*, pp. 2370–2379, 2015a.

Hsieh, C.-J., Yu, H.-F., and Dhillon, I. Passcode: Parallel asynchronous stochastic dual co-ordinate descent. In Bach, F. and Blei, D. (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2370–2379, Lille, France, 07–09 Jul 2015b. PMLR.

Kaggle. Kaggle machine learning and data science survey, 2017. `https://www.kaggle.com/surveys/2017`.

Kaggle. Libsvm data: Classification, regression, and multi-label, 2019. `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

Kaufmann, M., Parnell, T. P., and Kourtis, K. Elastic cocoa: Scaling in to improve convergence. *arXiv:1811.02322*, 2018.

Langford, J. Vowpal wabbit. `https://github.com/JohnLangford/vowpal_wabbit/wiki`, 2007.

Lee, C.-p. and Chang, K.-W. Distributed block-diagonal approximation methods for regularized empirical risk minimization. *arXiv:1709.03043*, 2018.

Liu, J. and Wright, S. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM Journal on Optimization*, 25(1):351–376, 2015.

Liu, J., Wright, S. J., Ré, C., Bittorf, V., and Sridhar, S. An asynchronous parallel stochastic coordinate descent algorithm. *The Journal of Machine Learning Research*, 16(1):285–322, 2015.

Ma, C., Smith, V., Jaggi, M., Jordan, M. I., Richtárik, P., and Takáč, M. Adding vs. Averaging in Distributed Primal-Dual Optimization. In *Proceedings of the 32th International Conference on Machine Learning*, ICML, pp. 1973–1982, 2015.

Parnell, T. P., Dünner, C., Atasu, K., Sifalakis, M., and Pozidis, H. Large-Scale Stochastic Learning Using GPUs. *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 419–428, 2017.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

Richtarik, P. and Takac, M. Distributed coordinate descent method for learning with big data. *Journal of Machine Learning Research*, 17:1–25, 2016a.

Richtarik, P. and Takac, M. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156:433–484, 2016b.

Shalev-Shwartz, S. and Zhang, T. Stochastic dual coordinate ascent methods for regularized loss. *JMLR*, 14(1):567–599, 2013. ISSN 1532-4435.

Smith, V., Forte, S., Ma, C., Takáč, M., Jordan, M., and Jaggi, M. Cocoa: A general framework for communication-efficient distributed optimization. 18, 2018.

The H2O.ai team. h2o: Python interface for h2o. `http://www.h2o.ai`, 2015. Python package version 3.20.0.8.

Wright, S. J. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

Yang, T. Trading computation for communication: Distributed stochastic dual coordinate ascent. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 26*, pp. 629–637. Curran Associates, Inc., 2013.

Zhang, C. and Ré, C. Dimmwitted: A study of main-memory statistical analytics. *Proc. VLDB Endow.*, 7(12):1283–1294, August 2014. ISSN 2150-8097.