Coordinate Descent Algorithms With Coupling Constraints: Lessons Learned

Ahmed Hefny
ahefny@cs.cmu.edu

Sashank Reddi
sjakkamr@cs.cmu.edu

Suvrit Sra

Abstract

Coordinate descent methods are enjoying renewed interest due to their simplicity and success in many machine learning applications. Given recent theoretical results on random coordinate descent with linear coupling constraints, we develop a software architecture for this class of algorithms. A software architecture has to (1) maintain solution feasibility, (2) be applicable to different execution environments, whether local or distributed and (3) decouple problem-specific logic from the execution environment. We demonstrate that due to the nature of these algorithms, these requirements raise some issues that are not present in many other classes of machine learning algorithms and thus can be overlooked when designing a generic machine learning system.

1 Introduction

Coordinate descent (CD) methods are conceptually among the simplest schemes for unconstrained optimization—they have been studied for a long time (see e.g., [2, 3]), and are now enjoying greatly renewed interest. Their resurgence is rooted in successful applications in machine learning [6, 7], statistics [5, 8], and many other areas—see [15, 16, 18] and references therein for more examples.

Parallel coordinate descent has been the focus of increasing interest. Numerous previous works (e.g. [17][4]) addressed parallel coordinate descent where the constraints are absent or block-separable; that is, each block can be updated independently while maintaining feasibility.

The case of coupling constraints has been addressed in [13], which proposes a distributed algorithm for separable objective functions with Lipschitz continuous gradients in the presence of a (weighted) sum constraint. Neocara et. el. [12] extended the formulation to a (not-necessarily separable) composite objective function with general linear constraints but did not detail on parallel and distributed implementations. In these implementations, care has to be taken to maintain feasibility of the solution by ensuring consistent updates.

In this work, we propose a software architecture for coordinate descent algorithms with linear constraints. The architecture we propose applies to several execution environments, including sequential, parallel and distributed implementations (with and without a central server node). In addition, it is agnostic to the exact problem as it decouples variable update logic from the underlying execution environment, making it easy to solve new problems once the required environment is realized. We highlight some of the issues that rise due to the nature of coordinate descent algorithms and illustrate how we dealt with them.

In the rest of the paper we describe our architecture in detail and then we demonstrate how to implement two problems (a generic smooth objective function and SVM) and two environments (local parallel and distributed with a parameter server).

2 Architecture

We consider problems of the form

$$\min_x f(x) + \sum_{i=1}^N h_i(x_i)$$
$$\text{s.t. } \sum_{i=1}^N A_i x_i = 0, \tag{1}$$

where $x_i \in \mathbb{R}^n$, $A_i \in \mathbb{R}^{m \times n}$ for some $m \leq 2n$, $h_i(x_i) : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ is a convex function and $f(x) : \mathbb{R}^{Nn} \to \mathbb{R}$ is a smooth convex function with Lipschitz continuous gradient satisfying

$$\|\nabla_{x_i} f(x') - \nabla_{x_i} f(x)\| \leq L_i \|x' - x\|,$$

where x' and x differ only in x_i.

The presence of coupling constraints mandates updating at least two blocks to maintain feasibility. The *pair update* outlined below is the core operation upon which our architecture is based

$$d_i, d_j = \operatorname*{arg\,min}_{A_i d_i + A_j d_j = 0} f(x) + \langle \nabla_{x_i} f(x), d_i \rangle + \langle \nabla_{x_j} f(x), d_j \rangle + \frac{1}{2\alpha} \|d_i\|^2 + \frac{1}{2\alpha} \|d_j\|^2 + h(x')$$
$$x_j \leftarrow x_j + d_j, x_i \leftarrow x_i + d_i \tag{2}$$

for some $i, j \leq N$, where x' is the result of adding d_i to x_i and d_j to x_j and $\alpha \leq \frac{1}{L_i + L_j}$ is a step size parameter. In principle, equation 2 defines a template for coordinate descent algorithms with linear constraints; essentially one specifies a *scheduler* to choose block pairs and a solver for the minimization problem. In practice, different environments impose additional considerations such as whether different blocks are stored on different machines, the format of information required to be exchanged and whether blocks are updated in an asynchronous manner. We would like to have a software architecture that

- Maintains feasibility in the presence of multiple asynchronous updates.
- Isolates update computation logic from the execution environment, including where the blocks are stored and whether external information are needed to compute the update.
- Allows flexibility in building the execution environment.

2.1 Incremental Updates

A key ingredient in our architecture to maintain consistency in a potentially parallel asynchronous environment is *incremental updates*. Specifically, the module responsible for solving the minimization in (2) outputs the increments d_i and d_j rather than the new values of x_i and x_j. These increments are applied to the existing values of x_i and x_j. We require that, for each scalar component x_{ik}, the update $x_{ik} \leftarrow x_{ik} + d_{ik}$ is executed as an *atomic* operation. This can be achieved on modern processors without additional locking structure using the compare-and-swap instruction. Algorithm 4 in the appendix demonstrates this via a C++ code snippet. Since the updates are themselves consistent and since they are applied in a sequentially consistent manner, the net result of T updates remains consistent. This update formulation is in contrast with architectures where user-specified modules are expected to specify the new values of variables under consideration, which may cause concurrent overwrites to destroy feasibility in an asynchronous environment.

2.2 Modular Decomposition

The proposed architecture consists of three main components: the *scheduler*, *worker nodes* and *parameter clients*.

The scheduler is responsible for selecting pairs of blocks to update. For flexibility in that aspect, a typical scheduler is equipped with a *pair selection policy* whose implementation determines the pair selection logic. A straightforward policy is to select any pair at random with equal probability.

Customized policies could restrict pairs that can interact or bias the distribution over pairs to reduce communication cost or enhance convergence. It is convenient to encode pair restriction in the form of a *communication graph*, where each vertex corresponds to a block x_i and two vertices are connected if they can be selected as a pair.

The scheduler is also responsible for exchanging information among other modules (either locally or through the network). The scheduler implements the execution environment but should be agnostic to the specific problem and update logic.

A *parameter read client* is an interface to access the information required to compute the update for a given block. Such information typically consists of the current value of the parameters. We refer to this information as $node\_input$. Some information, such as the constraint submatrices A_i do not change and will be referred to as $static\_node\_input$. A concrete parameter read client is required to implement the methods

$$Read(block\_id) : node\_input$$
$$ReadStatic(block\_id) : node\_static$$

A parameter read client therefore provides an abstraction of the underlying representation of blocks, whether it is local or remote.

A *parameter update client* is an interface to apply a pair update. A concrete client is realized by providing an implementation for the method

$$Update(block\_id\_1, block\_id\_2, update\_1, update\_2, is\_async\_update)$$

The reason for providing is_async_update flag is to allow the client to optimize for the case where the scheduler prevents concurrent updates to the same block.

The third main component is *worker nodes*. Worker nodes implement the update logic (i.e. the minimization subproblem in equation 2). Each worker node corresponds to a block. Multiple nodes on different machines can correspond to the same block. A pair update requires the interaction of two worker nodes, which is implemented in the following master/slave scheme:

1. Master node computes information needed for the update and encapsulates it in $master\_message$.
2. Based on $master\_message$, slave computes information needed for the update and encapsulates information used by master in $slave\_message$. Slave nodes also computes its update ($slave\_update$).
3. Based (solely) on information computed in steps 1 and 2, the master can compute its update ($master\_update$).

Given that scheme, a worker node implementation needs to provide the following methods:

- $Init(block\_id, static\_node\_input)$
- $GetMasterMessage(slave\_block\_id, node\_input) : master\_message$
- $GetSlaveMessage(master\_block\_id, node\_input, master\_message)$ $: (slave\_message, slave\_update)$
- $GetMasterUpdate(master\_message, slave\_message)$

It is possible to have the slave node perform all computations required to obtain the master and slave updates, with the master only providing necessary information that is not available to the slave node. The proposed architecture provides a additional flexibility, by allowing to split computations between master and slave, as will be exemplified in section 3.1.

Note that the worker node is not responsible for the actual communication of messages, retrieval of node inputs or applying the updates. By delegating these responsibilities to the scheduler, the worker node is independent from execution environment.

As an example of interaction between these modules, Algorithm 1 depicts a scheduler in the simple setting of a single processor.

```
1: for all $1 \le i \le N$ do
2:     $static\_node\_input \leftarrow ParameterReadClient.ReadStatic(i)$
3:     $Node(i).Init(static\_node\_input)$
4: end for
5: while Stopping Criteria Not Met do
6:     Select a master block $i$ and slave block $j$ using PairSelectionPolicy
7:     $master\_input \leftarrow ParameterReadClient.Read(i);$
8:     $slave\_input \leftarrow ParameterReadClient.Read(j);$
9:     $master\_to\_slave\_message \leftarrow Node(i).GetMasterMessage(j, master\_input)$
10:    $[slave\_to\_master\_message, slave\_update] \leftarrow$
       $Node(j).GetSlaveMessage(i, slave\_input, master\_to\_slave\_message)$
11:    $master\_update \leftarrow Node(i).GetMasterUpdate($
       $master\_to\_slave\_message, j, slave\_to\_master\_message)$
12:    $ParameterUpdateClient.Update(i, j, master\_update, slave\_update)$
13: end while
```

Algorithm 1: Sequential Scheduler

3 Problem Examples

In this section we show how to adapt our framework to two optimization problems. An adaptation to a problem amounts to providing an implementation of worker node methods. as well as specifying the the corresponding $node\_info$, $static\_node\_info$, $master\_message$ and $slave\_message$ objects. A part of this specification is providing encoding and decoding methods required to communicate these objects over a network.

3.1 Smooth Objective Function

In the case where the non-smooth component $h(x)$ is 0, the solution to 2 is given by

$$\begin{aligned} \lambda &= \alpha(A_i A_i^\top + A_j A_j^\top)^+ \left(A_i \nabla_i f(x) + A_j \nabla_j f(x)\right) \\ d_i &= \alpha \nabla_i f(x) - A_i^\top \lambda \\ d_j &= \alpha \nabla_j f(x) - A_j^\top \lambda \end{aligned} \tag{3}$$

If the worker node for x_i can compute $\nabla_i f(x)$ (or a stochastic version thereof) and has access to A_i then we can have the following master/slave update sequence:

1. The master i computes $y = A_i \nabla_i f(x)$ and sends both y and A_i to the slave.
2. The slave j computes λ and d_j and sends back λ to the master.
3. The master computes d_i.

This sequence fits within our framework. For concreteness, Node info and message representations are outlined in Algorithm 2.

Given these representation a worker node implementation can be obtained based on the aforementioned master/slave update.

In a practical implementation, the node x_i does not need to send A_i to a node x_j more than once. This can be done by having each node maintain a list of previous slaves and include A_i in the master message only if it is directed to a new slave. On the other hand, each node x_i has access to a cache to store the values of A_j and $(A_i A_i^\top + A_j A_j^\top)^+$ for each other node j that interacted with i as a master. To avoid redundant storage, the same cache can be shared between all worker nodes on the same machine. The cache has to support concurrent access without adding a locking structure. A simple way to achieve that purpose is to pre-allocate cache entries (as pointers) and update these entries using atomic lock-free operations, as detailed in the appendix.

```
$smoothobj\_static\_node\_input$ {
    $A$ : $m \times n$ constraint submatrix
}
$smoothobj\_node\_input$ {
    $x$ : $Nn$-dimensional vector
}
$smoothobj\_master\_message$ {
    $y$ : $m$-dimensional vector (to store $A_i \nabla_i f(x)$)
    $A$ : $m \times n$ constraint submatrix (can be empty)
}
$smoothobj\_slave\_message$ {
    $\lambda$ : $m$-dimensional vector (Lagrange multipliers).
}
```

Algorithm 2: Info and message objects used for the smooth objective function example

3.2 Support Vector Machine Training

As another example of an optimization problem that fits within our framework, we consider the SVM dual optimization problem, given by

$$\min_{\lambda} \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{ij} \lambda_i \lambda_j G_{ij} + \sum_{i} \mathbb{I}_{[0,C]}(\lambda_i)$$
$$\text{s.t.} \sum_{i=1}^{N} \lambda_i y_i = 0, \tag{4}$$

where x_i and y_i are training examples and corresponding labels, G is the matrix given by $G_{ij} = y_i y_j k(x_i, x_j)$ for some kernel k and $\mathbb{I}_A(w)$ is the indicator function which is 0 for $\omega \in A$ and ∞ otherwise. This matches the formulation in (1). The master/slave update scheme we developed for SVM is based on sequential minimum optimization (SMO) [14]. The idea of SMO is to optimize (4) over a pair of dual variables λ_i and λ_j keeping other dual variables fixed. This subproblem is fully specified by λ, y, C in addition to G_i and G_j, the i^{th} and j^{th} columns of G.

That makes the representation outlined in Algorithm 3. Here we assume that SMO solution will be computed by the slave, with the master only providing the necessary input.

```
$svm\_static\_node\_input$ {
    $Gc$ : $N$-dimensional corresponding column of $G$
    $y$ : Corresponding label
}
$svm\_node\_input$ {
    $\lambda$ : $N$-dimensional vector
}
$svm\_master\_message$ {
    $Gc$ : $N$-dimensional corresponding column of $G$
    $y$ : Corresponding label
}
$svm\_slave\_message$ {
    $master\_update$
}
```

Algorithm 3: Info and message objects used for the SVM example

4 Execution Environment Examples

In this section we look at the other side of adapting our proposed architecture; adapting to the execution environment. This amounts to providing a scheduler and adapting parameter read and update clients.

4.1 Local Parallel Execution

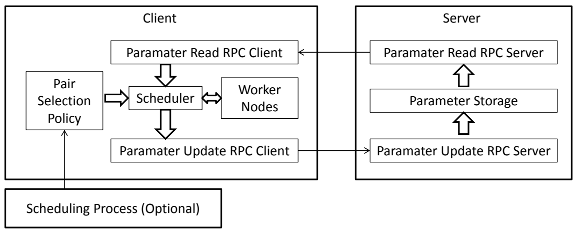
In a multiprocessor with shared memory setting, parameter read and update clients are simple accessors to shared variables. It is important to use atomic increments to implement the $Update$ method in order to maintain consistency in an asynchronous setting.

A scheduler is obtained by running algorithm 1 on multiple threads. Different levels of synchronization can be provided; an asynchronous scheduler is obtained by simply having each thread run at will. A fully synchronous scheduler can be obtained by placing a synchronization barrier after each iteration and using a pair selection policy that provides different threads with disjoint pairs. There are other synchronization models along the spectrum. For example, in a *pair-locking* model, each thread locks each block in the pair to be updated.

4.2 Distributed Setting With a Parameter Server

In this setting, we have a central parameter server [1][9] storing the variables to be optimized as well as a set of clients that perform the computations. The updates are committed to the server which provides the logic to apply them (which may include, for example, rejecting updates whose staleness exceeds some threshold). Again, server-side updates need to use atomic increments to allow asynchronous operation.

In the parameter server setting, parameter read and update clients encapsulate communication with the server to retrieve parameters and commit updates. The communication graph is divided into overlapping connected subgraphs each of which is assigned to a client machine, which creates corresponding worker nodes. A (possibly multithreaded) scheduler based on algorithm 1 can then be run on each machine. The pair selection policy can be implemented within the client or by remotely querying a central scheduling process. The overall structure is depicted in figure 1.



Client
Server
Paramater Read RPC Client
Paramater Read RPC Server
Pair Selection Policy
Scheduler
Worker Nodes
Parameter Storage
Paramater Update RPC Client
Paramater Update RPC Server
Scheduling Process (Optional)

Figure 1: Block diagram of a distributed setting with a parameter server. Arrows represent direction of information flow.

5 Adapting Parameter Clients

Sections 3 and 4 showed that worker nodes and schedulers can be independently implemented to fit the problem at hand and the execution environment respectively. A software coupling issue remains with parameter read and update clients; the parameter clients determine *what* information to read/write and *how* to read/write it. The *what* aspect depends on the problem while the *how* aspect depends on the environment.

To avoid the need for implementing parameter clients for each problem/environment combination we propose the following framework: Given a specific problem, a *local parameter client* implementation is provided which encapsulates local memory access (i.e. is compatible with local scheduler in section 4.1). This is independent from environment adaptation. Environment adaptation, on the other hand, includes providing generic wrappers for the local parameter clients. The implementation of these wrappers is independent of the implementation of the local parameter clients. Algorithm 6 illustrates this concept in more detail.

6 Discussion

In this paper we proposed a software architecture to address issues in coordinate descent algorithms with linear constraints. Two major issues resulted from the existence of coupling constraints; the first issue is the need to update block pairs rather than an individual blocks while the second issue is the necessity to specify an updates as a value increment rather than a value overwrite to facilitate asynchronous operation while maintaining feasibility. We believe that the formulation we proposed to tackle these issues can be incorporated into more general machine learning toolkits.

To demonstrate the effect of these issues, we discuss their effect on a popular generic machine learning framework. Perhaps the closest machine learning abstraction to the proposed architecture is GraphLab [10][11] in the sense that it abstracts computation in terms of interactions between a vertex and its neighborhood in a graph. To the best of our knowledge, Graphlab, in its current form, falls short in handling the two issues we discussed. In synchronous execution, GraphLab suffers from the fact that it uses *vertex update* as a scheduling unit, where each vertex corresponds to a block. Although a vertex update can change block pairs it specifies the new value of each block rather than an increment, which could result in an infeasible solution in an asynchronous setting. To avoid this problem, a higher consistency model can be enforced, but the least conservative model that can ensure feasibility requires locking a vertex with all its neighbors (in contrast to our pair-lock model which locks precisely the blocks to be updated). This can be serious if the communication graph is not sparse or contains high-degree nodes.

A Supplementary Code Snippets

A.1 Applying Incremental Updates

Algorithm 4 demonstrated applying atomic lock-free increments.

```
void increment(volatile double *target, double increment) {
    double value;
    do {
       value = *target;
    }while(value != CompareAndSwap(target, value, value+increment)); }
}
void incrementVector(volatile double vector[], int size,
                    double increment[]) {
   for(int i = 0; i < size; ++i) {
       increment(\&vector[i], increment[i]);
   }
}
```

Algorithm 4: Applying an increment. `CompareAndSwap(a,b,c)` is an atomic operation that compares the current content of memory location a to the value b and if they are equal replaces it with the value c. The output is the current content of memory location a (before replacement).

A.2 Lock-free Caching

Algorithm 5 demonstrates lock-free cache updates in a high-level C++ code snippet.

A.3 Parameter Client Wrapper

Algorithm 6 demonstrates a high-level example of a generic parameter update client wrapper for the parameter server scenario where communication is performed via a remote procedure call service. Note that the implementation of the RPC client and server wrappers is independent from the provided implementation of the parameter update client.

```
//Cache[i][j] is initalized to NULL
ptr = Cache[i][j];
if(ptr == NULL) { // Element does not exist
   ptr = AllocateMemory();
   *ptr = ComputeMatrix(i,j);
}
ptr2 = CompareAndSwap(&Cache[i][j], NULL, ptr);
if(ptr2 != NULL) {// Element already written by another thread
   delete ptr; ptr = ptr2;}
//Now ptr points to the cached element
```

Algorithm 5: C++ High-level code snippet for lazy computation of $(A_i A_i^\top + A_j A_j^\top)^+$, `CompareAndSwap(a,b,c)` is an atomic operation that compares the current content of memory location a to the value b and if they are equal replaces it with the value c. The output is the current content of memory location a (before replacement).

```
template<class Update>
class ParameterUpdateClient {
public:
    virtual void Update(int block_id_1, int block_id_2,
                        const Update &update_1, const Update &update_2,
                        bool is_async) = 0;
};

template<class Update>
class SVMParameterUpdateClient : public ParameterUpdateClient<Update> {
    // Implementation provided by problem adaptation
};

template<class Update>
class ParmaterUpdateClient_RPCClientWrapper
    : public ParameterUpdateClient<Update> {
public:
 ParameterUpdateClient(RPCConnectionInfo* rpc) {...};

 virtual void Update(int block_id_1, int block_id_2,
                       const Update &update_1, const Update &update_2,
                       bool is_async) {
        rpc->CallRPCService(block_id1, block_id2,
                            Update::Encode(update_1),
                            Update::Encode(update_2), is_async);
    }
};
template<class Update>
class ParmaterUpdateClient_RPCServerWrapper
public:
 ParmaterUpdateClient_RPCServerWrapper(
     ParameterUpdateClient<Update>* local_client) {...};

 void RPCService(int block_id_1, int block_id_2,
                   const char *encoded_update_1,
                   const char *encoded_update_2,
                   bool is_async) {
        local_client->Update(block_id1, block_id2,
                             Update::decode(encoded_update_1),
                             Update::decode(encoded_update_2),
                             is_async);
    }
};
```

Algorithm 6: RPC wrappers for parameter update client.

References

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 123–132, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0747-5. doi: 10.1145/2124295.2124312. URL `http://doi.acm.org/10.1145/2124295.2124312`.

[2] A. Auslender. *Optimisation Méthodes Numériques*. Masson, 1976.

[3] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, second edition, 1999.

[4] J. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for L1-regularized loss minimization. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning*, pages 321–328. Omnipress, 2011.

[5] J. Friedman, T. Hastie, H. Höfling, R. Tibshirani, et al. Pathwise coordinate optimization. *The Annals of Applied Statistics*, 1(2):302–332, 2007.

[6] C. J. Hsieh and I. S. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD)*, pages 1064–1072, August 2011.

[7] C. J. Hsieh, K. W. Chang, C. J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In W. Cohen, A. McCallum, and S. Roweis, editors, *ICML*, pages 408–415. ACM, 2008.

[8] C.-J. Hsieh, M. A. Sustik, I. S. Dhillon, and P. D. Ravikumar. Sparse inverse covariance matrix estimation using quadratic approximation. In *NIPS*, pages 2330–2338, 2011.

[9] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `http://blogs.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu`.

[10] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 340–349, 2010. URL `http://uai.sis.pitt.edu/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2126&proceeding_id=26`.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212354. URL `http://dx.doi.org/10.14778/2212351.2212354`.

[12] I. Necoara and A. Patrascu. A random coordinate descent algorithm for optimization problems with composite objective function and linear coupled constraints. *Comp. Opt. and Appl.*, 57(2):307–337, 2014.

[13] I. Necoara, Y. Nesterov, and F. Glineur. A random coordinate descent method on large optimization problems with linear constraints. Technical report, Technical Report, University Politehnica Bucharest, 2011, 2011.

[14] J. C. Platt. Advances in kernel methods. chapter Fast Training of Support Vector Machines Using Sequential Minimal Optimization, pages 185–208. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-19416-3. URL `http://dl.acm.org/citation.cfm?id=299094.299105`.

[15] P. Richtárik and M. Takáč. Distributed coordinate descent method for learning with big data. *ArXiv e-prints*, Oct. 2013.

[16] P. Richtárik and M. Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *arXiv:1107.2848v1*, July 2011.

[17] P. Richtárik and M. Takáč. Parallel coordinate descent methods for big data optimization. *arXiv:1212.0873v1*, Dec 2012.

[18] S. Shalev-Shwartz and T. Zhang. Stochastic dual coordinate ascent methods for regularized loss minimization. *JMLR*, 14, 2013.