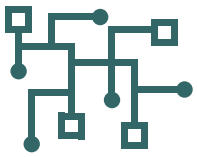

ISE

Industrial and Systems Engineering

Scaling Up Quasi-Newton Algorithms: Communication Efficient Distributed SR1

Majid Jahani[1], Mohammadreza Nazari[1], Sergey Rusakov[1], Albert S Berahas[1], and Martin Takáč[1]

[1]Department of Industrial and Systems Engineering, Lehigh University

ISE Technical Report 20T-006

LEHIGH UNIVERSITY

Scaling Up Quasi-Newton Algorithms: Communication Efficient Distributed SR1

Majid Jahani
Lehigh University
Bethlehem, PA
maj316@lehigh.edu

Mohammadreza Nazari
Lehigh University
Bethlehem, PA
mon314@lehigh.edu

Sergey Rusakov
Lehigh University
Bethlehem, PA
ser318@lehigh.edu

Albert S Berahas
Lehigh University
Bethlehem, PA
albertberahas@gmail.com

Martin Takáč
Lehigh University
Bethlehem, PA
Takac.MT@gmail.com

Abstract

In this paper, we present a scalable distributed implementation of the sampled LSR1 (S-LSR1) algorithm. First, we show that a naive distributed implementation of S-LSR1 requires multiple rounds of expensive communications at every iteration and thus is inefficient. We then propose DS-LSR1, a communication-efficient variant of the S-LSR1 method, that drastically reduces the amount of data communicated at every iteration, that has favorable work-load balancing across nodes and that is matrix-free and inverse-free. The proposed method scales well in terms of both the dimension of the problem and the number of data points. Finally, we illustrate the performance of DS-LSR1 on standard neural network training tasks.

1 Introduction

In the last decades, a significant amount of research has been devoted to the development of optimization algorithms for machine learning. Currently, due to its fast learning properties, low per-iteration cost, and ease of implementation, the stochastic gradient (SG) method [10, 47], and its adaptive [23, 33, 62], variance-reduced [22, 30, 41, 50] and distributed [21, 34, 45, 46, 58, 64] variants are the preferred optimization methods for large-scale machine learning applications. Nevertheless, these methods have several drawbacks; they are highly sensitive to the choice of hyper-parameters (e.g., step size parameter) and are cumbersome to tune, and they suffer from ill-conditioning [3, 11, 49, 60]. More importantly, these methods offer a limited amount of benefit in distributed computing environments. Since these methods are usually implemented with small mini-batches, they spend more time communicating instead of performing "actual" computations. This shortcoming can be remedied to some extent by increasing the batch sizes, however, there is a point after which the increase in computation is not offset by the faster convergence [55].

Recently, there has been an increased interest in second-order and quasi-Newton methods by the machine learning community, and several stochastic variants have been proposed; see e.g. [4, 6–8, 13, 14, 20, 27–29, 31, 39, 40, 49, 51, 61]. These methods judiciously incorporate curvature information, and thus mitigate some of the issues that plague first-order methods. Another benefit of these methods is that they are usually implemented with larger batches, and thus better balance the communication and computation costs. Of course, this does not come for free; (stochastic) second-order and quasi-Newton methods are more memory intensive and more expensive (per iteration) than first-order methods. This naturally calls for distributed implementations of these methods.

In this paper, we propose an efficient distributed variant of the sampled L-SR1 (S-LSR1) method [5]—which we call DS-LSR1—that operates in the master-worker framework illustrated in Figure 1.

Preprint. Under review.

Each worker node has a portion of the dataset, and performs local computations using solely that information and information received from the master node. The proposed method is matrix-free (the Hessian approximation is never explicitly constructed) and inverse-free (no matrix is inverted). To this end, we leverage the compact form of the updating formula of the SR1 Hessian approximations [16], and utilize sketching techniques [36, 59] to approximate several required quantities. We show that, contrary to a naive distributed implementation of S-LSR1, the method is communication-efficient and has favorable work-load balancing across nodes. Specifically, the naive implementation requires communicating $\mathcal{O}(md)$ quantities, whereas our approach only requires communicating $\mathcal{O}(m^2)$ quantities, where d is the dimension of the problem and m is the LSR1 memory.[1] Furthermore, in our approach the heavy computations are done by the worker nodes and the master node performs only simple aggregations, whereas in the naive approach the most computationally intensive operations, e.g., CG and Hessian-vector products, are computed locally by the master node. Finally, we show empirically that DS-LSR1 has good strong and weak scaling properties, and illustrate the performance of the proposed method on standard neural network training tasks.

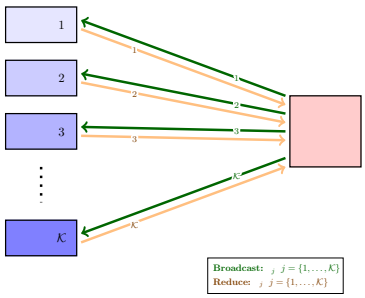

Figure 1: Distributed Computing Schematic.

Problem Formulation and Notation We focus on machine learning empirical risk minimization problems that can be expressed as:

$$\min_{w \in \mathbb{R}^d} F(w) := \frac{1}{n} \sum_{i=1}^{n} f(w; x^i, y^i) = \frac{1}{n} \sum_{i=1}^{n} f_i(w), \tag{1.1}$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is the composition of a prediction function (parametrized by w) and a loss function, and (x^i, y^i), for $i = 1, \ldots, n$, denote the training examples (samples). Specifically, we focus on deep neural network training tasks where the function F is nonconvex, and the dimension d and number of samples n are large.

The paper is organized as follows. We conclude this section with a discussion of related work. We describe the classical (L)SR1 and sampled LSR1 (S-LSR1) methods in Section 2. In Section 3, we present DS-LSR1, our proposed distributed variant of the sampled LSR1 method. We illustrate the scaling properties of DS-LSR1 and the empirical performance of the method on deep learning tasks in Section 4. Finally, in Section 5 we provide some final remarks.

Related Work The symmetric-rank-1 (SR1) method [15, 19, 32] and its limited-memory variants (LSR1) [12, 38] are quasi-Newton methods that have gained significant attention by the machine learning community in recent years [5, 24, 25]. These methods incorporate curvature (second-order) information using only gradient (first-order) information. Contrary to arguably most popular quasi-Newton method, (L)BFGS [37, 42, 43], the (L)SR1 method does not enforce that the Hessian approximations are positive definite, and as such is usually implemented with a trust-region [43]. This has several benefits: (1) the method is able to exploit negative curvature, and (2) the method is able to efficiently escape saddle points.

There has been a significant volume of research on distributed algorithms for machine learning; specifically, distributed gradient methods [9, 18, 21, 45, 58, 64], distributed Newton methods [2, 29, 52, 63] and distributed quasi-Newton methods [1, 17, 21]. General distributed optimization methods close to our work are the approaches based on parallel gradient computation followed by a centralized algorithm [17, 26, 56]. Possibly the closest work to ours is VF-BFGS [17], in which the authors propose a vector-free implementation of the classical LBFGS method. We leverage several of the techniques proposed in [17], however, what differentiates our work is that we focus on the S-LSR1 method. Developing an efficient distributed implementation of the S-LSR1 method is not as straight-forward as LBFGS for several reasons: (1) the construction and acceptance of the curvature pairs, (2) the trust-region subproblem, and (3) the step acceptance procedure.

[1] Note, these costs are on top of the communications that are common to both approaches.

2 Sampled limited-memory SR1 (S-LSR1)

In this section, we review the sampled LSR1 method [5], and discuss the components that can be distributed. We begin by describing the classical (L)SR1 method as this will set the stage for the presentation of the S-LSR1 method. At the kth iteration, the SR1 method computes a new iterate via

$$w_{k+1} = w_k + p_k,$$

where p_k is the minimizer of the following subproblem

$$\min_{\|p\| \leq \Delta_k} m_k(p) = F(w_k) + \nabla F(w_k)^T p + \tfrac{1}{2} p^T B_k p, \tag{2.1}$$

Δ_k is the trust region radius, B_k is the SR1 Hessian approximation computed as

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}, \tag{2.2}$$

and $(s_k, y_k) = (w_k - w_{k-1}, \nabla F(w_k) - \nabla F(w_{k-1}))$ are the curvature pairs. In the limited memory version, the matrix B_k is defined at each iteration as the result of applying m SR1 updates to a multiple of the identity matrix using the set of m most recent curvature pairs $\{s_i, y_i\}$ kept in storage.

The main idea of the S-LSR1 method is to use the SR1 updating formula, but to construct the Hessian approximations using sampled curvature pairs instead of pairs that are constructed as the optimization progresses. Specifically, at every iteration, m curvature pairs are constructed via random sampling around the current iterate; see Algorithm 2. The S-LSR1 method is outlined in Algorithm 1. The components of the algorithms that can be distributed are highlighted in magenta.

Algorithm 1 Sampled LSR1 (S-LSR1)

Input: w_0 (initial iterate), Δ_0 (initial trust region radius), m (memory), r (sampling radius).

1: **for** $k = 0, 1, 2, \ldots$ **do**
2: Compute $F(w_k)$ and $\nabla F(w_k)$
3: Compute new (S_k, Y_k) pairs via Algorithm 2
4: Compute p_k by solving the subproblem (2.1)
5: Compute $\rho_k = \frac{F(w_k) - F(w_k + p_k)}{m_k(0) - m_k(p_k)}$
6: **if** $\rho_k \geq \eta_1$ **then** Set $w_{k+1} = w_k + p_k$
7: **else** Set $w_{k+1} = w_k$
8: $\Delta_{k+1} = \texttt{adjustTR}(\Delta_k, \rho_k)$ [Appendix B.3]
9: **end for**

Algorithm 2 Construct new (S_k, Y_k) curvature pairs

Input: w_k (current iterate), m (memory), r (sampling radius), $S_k = [\,]$, $Y_k = [\,]$ (curvature pair containers).

1: **for** $i = 1, 2, \ldots, m$ **do**
2: Sample a random direction of unit length σ_i
3: Sample point $\bar{w}_i = w_k + r\sigma_i$
4: Set $s_i = w_k - \bar{w}_i$ and $y_i = \nabla^2 F(w_k) s_i$
5: Set $S_k = [S_k \; s_i]$ and $Y_k = [Y_k \; y_i]$
6: **end for**

Output: S, Y

As is clear, several components of the above algorithms can be distributed. Before we present the distributed implementations of the S-LSR1 method, we discuss several key elements of the method: (1) Hessian-vector products; (2) curvature pair construction; (3) curvature pair acceptance; (4) search direction computation; (5) step acceptance procedure; and (6) initial Hessian approximations.

For the remainder of the paper, let $S_k = [s_{k,1}, s_{k,2}, \ldots, s_{k,m}] \in \mathbb{R}^{d \times m}$ and $Y_k = [y_{k,1}, y_{k,2}, \ldots, y_{k,m}] \in \mathbb{R}^{d \times m}$ denote the curvature pairs constructed at the kth iteration, S_k^i and Y_k^i denote the curvature pairs constructed at the kth iteration by the ith node, and $B_k^{(0)} = \gamma_k I \in \mathbb{R}^{d \times d}$, $\gamma_k \geq 0$, denote the initial Hessian approximation at the kth iteration.

Hessian-vector products Several components of the algorithms above require the calculation of Hessian vector products of the form $B_k v$. In the large-scale setting, it is not memory-efficient, or even possible for some applications, to explicitly compute and store the $d \times d$ Hessian approximation matrix B_k. Instead, one can exploit the compact representation of the SR1 matrices [16] and compute:

$$B_{k+1} v = B_k^{(0)} v + (Y_k - B_k^{(0)} S_k) \underbrace{(D_k + L_k + L_k^T - S_k^T B_k^{(0)} S_k)}_{M_k}{}^{-1} (Y_k - B_k^{(0)} S_k)^T v, \tag{2.3}$$

$$D_k = diag[s_{k,1}^T y_{k,1}, \ldots, s_{k,m}^T y_{k,m}], \quad (L_k)_{j,l} = \begin{cases} s_{k,j-1}^T y_{k,l-1} & \text{if } j > l, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

Computing $B_{k+1} v$ via (2.3) is both memory and computationally efficient; the complexity of computing $B_{k+1} v$ is $\mathcal{O}(m^2 d)$.

Curvature pair construction For ease of exposition, we presented the curvature pair construction algorithm (Algorithm 2) as a sequential process. Of course, this need not be the case; all curvature pairs can be constructed simultaneously. First, generate a random matrix $S_k \in \mathbb{R}^{d \times m}$, and then compute $Y_k = \nabla^2 F(w_k) S \in \mathbb{R}^{d \times m}$. We discuss how this may be done in a distributed manner in the following sections.

Curvature pair acceptance In order for the S-LSR1 Hessian update (2.2) to be well defined and for numerical stability we require certain conditions on the curvature pairs employed; see [43, Chapter 6]. Namely, for a given $\eta > 0$, we impose that the Hessian approximation B_{k+1} is only updated using the curvature pairs that satisfy the following condition:

$$|s_{k,j}^T(y_{k,i} - B_k^{(j-1)}s_{k,j})| \geq \eta \|s_{k,j}\| \|y_{k,i} - B_k^{(j-1)}s_{k,j}\|, \tag{2.5}$$

for $j = 1, \ldots, m$, where $B_k^{(0)}$ is the initial Hessian approximation and $B_k^{(j-1)}$, for $j = 2, \ldots, m$, is the Hessian approximation constructed using only curvature pairs $\{s_l, y_l\}$, for $l < j$, that satisfy (2.5). Note, $B_{k+1} = B_k^{(m)}$. Thus, potentially, not all curvature pairs returned by Algorithm 2 are used to update the S-LSR1 Hessian approximation. Checking this condition is not trivial and requires m Hessian vector products. In [5, Appendix B.5], the authors propose a recursive memory-efficient way to check the condition and retain only the pairs that satisfy (2.5).

Search direction computation The search direction p_k is computed by solving subproblem (2.1) using CG-Steihaug [43, Chapter 7]; see Appendix B.1 Algorithm 5. This procedure requires the computation of Hessian vectors products of the form (2.3).

Step acceptance procedure In order to determine if a step is successful (Line 6, Algorithm 1) one has to compute the function value at the trial iterate and the predicted model reduction. This entails a function evaluation and a Hessian vector product. The acceptance ratio ρ_k determines if a step is successful, after which the trust region radius has to be adjusted accordingly. For brevity we omit the details from the paper and refer the interested reader to Appendix B.3.

Initial Hessian approximations $B_k^{(0)}$ In practice, it is not clear how the initial Hessian approximation should be chosen. We argue, that in the context of the S-LSR1 method, a good choice is $B_k^{(0)} = 0$. In Figure 2 we show the eigenvalues of the true Hessian and the eigenvalues of the S-LSR1 matrices for different values of γ_k for a toy classification problem [5]. As is clear, the eigenvalues of the S-LSR1 matrices with $\gamma_k = 0$ better match the eigenvalues of the true Hessian. Similar results were observed for other datasets; see Appendix C.1. Moreover, by setting $\gamma_k = 0$, the rank of the approximation is at most m and thus the CG algorithm will terminate in at most m iterations, whereas the CG algorithm may require as many as $d \gg m$ iterations when $\gamma_k \neq 0$. Another reason for making this choice is that it removes a hyper-parameter. Henceforth, in our presentation of the algorithms we assume that $B_k^{(0)} = 0$, however, we note that our method can be extended to $B_k^{(0)} \neq 0$.

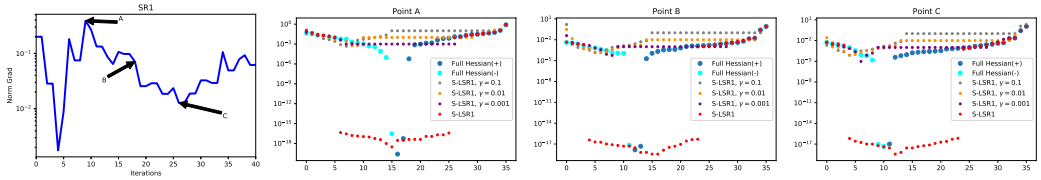

Figure 2: Comparison of the eigenvalues of S-LSR1 for different γ (@ A, B, C) for a toy classification problem.

2.1 Naive Distributed Implementation of S-LSR1

In this section, we describe a naive distributed implementation of the S-LSR1 method, where the data is stored across $\mathcal{K}$ machines. In order to implement Algorithm 1 in a distributed manner, at each iteration k, we broadcast the current iterate w_k to every worker node. The worker nodes then calculate the local objective function and gradient, and construct local curvature pair S_k^i and Y_k^i. The local information is then reduced to the master node to form $F(w_k)$, $\nabla F(w_k)$, S_k and Y_k. The SR1 curvature pair condition (2.5) is then recursively checked on the master node. Given a set of accepted curvature pairs, the search direction p_k is computed on the master node. We should note that the last two step could potentially be done in a distributed manner at the cost of $m+1$ extra expensive rounds of communication. Finally, given a search direction the trial iterate is broadcast to the worker nodes where the local objective function is computed and reduced to the master node, and a step is taken.

As is clear, in this distributed implementation of the S-LSR1 method, the amount of information communicated is large, and the amount of computation performed on the master node is significantly larger than that on the worker nodes. Note, all the Hessian vector products, as well as the computations of the M_k^{-1} are performed on the master node. The precise communication and computation details are summarized in Tables 1 and 2.

3 Efficient Distributed S-LSR1 (DS-LSR1)

The naive distributed implementation of S-LSR1 has several significant deficiencies. We propose a distributed variant of the S-LSR1 method that alleviates these issues, is communication-efficient, has favorable work-load balancing across nodes and is inverse-free and matrix-free. To do this, we leverage the form of the compact representation of the S-LSR1 updating formula

$$B_{k+1}v = Y_k M_k^{-1} Y_k^T v, \tag{3.1}$$

($B_k^{(0)} = 0$), and the form of the SR1 condition

$$|s_{k,j}^T (y_{k,i} - B_k^{(j-1)} s_{k,j})| \geq \eta \|s_{k,j}\| \|y_{k,i} - B_k^{(j-1)} s_{k,j}\|, \tag{3.2}$$

for $j = 1, \ldots, m$. We observe the following: one need not communicate the full S_k and Y_k, rather one can communicate $S_k^T Y_k$, $S_k^T S_k$ and $Y_k^T Y_k$. We now discuss the means by which: (1) we reduce the amount of information communicated and (2) we balance the computation across the nodes.

3.1 Reducing the Amount of Information Communicated

As mentioned above, communicating curvature pairs is not necessary; instead one can just communicate inner products of the pairs, reducing the amount of communication from $2md$ to $3m^2$. In this section, we show how this can be achieved, and in fact show that this can be further reduced to m^2.

Construction of $S_k^T S_k$ and $S_k^T Y_k$ Since the curvature pairs are scale invariant [5], S_k can be any random matrix. Therefore, each worker node can construct this matrix by simply sharing random seeds. In fact, the matrix $S_k^T S_k$ need not be communicated to the master node as the master node can construct and store this matrix. With regards to the $S_k^T Y_k$, each worker node can construct local versions of the Y_k curvature pair, Y_k^i, and send $S_k^T Y_k^i$ to the master node for aggregation, i.e., $S_k^T Y_k = {}^1/_{\mathcal{K}} \sum_{i=1}^{\mathcal{K}} S_k^T Y_k^i$. Thus, the amount of information communicated to the master node is m^2.

Construction of $Y_k^T Y_k$ Constructing the matrix $Y_k^T Y_k$ in distributed fashion, without communicating local Y_k^i matrices, is not that simple. In our communication-efficient method, we propose that the matrix is approximated via sketching [36, 59], using quantities that are already computed, i.e., $Y_k^T Y_k \approx Y_k^T S_k S_k^T Y_k$. In order for the sketch to be well defined, $S_k \sim \mathcal{N}(0, I/m)$, thus satisfying the conditions of sketching matrices [59]. By using this technique, we construct an approximation to $Y_k^T Y_k$ with no additional communication. Figure 3 illustrates the dependence of the error on the sketch size. Note, sketch size in our setting is the memory size m. We should note that this approximation is only used in checking the SR1 condition (3.2), which is not sensitive to approximation errors, and not in the Hessian vector products.

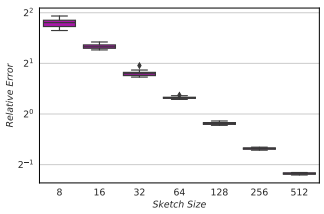

Figure 3: Error in the approximation as a function of the sketch size m.

3.2 Balancing the Computation Across the Nodes

Balancing the computation across the nodes does not come for free. We propose the use of a few more rounds of communication. The key idea is to exploit the compact representation of the SR1 matrices and perform as much computation as possible on the worker nodes.

Computing Hessian vector products $B_{k+1}v$ The Hessian vector products (3.1), require products between the matrices Y_k, M_k^{-1} and a vector v. Suppose that the we have M_k^{-1} on the master node, and that it broadcasts this information as well as the vector v to the worker nodes. The worker nodes then locally compute $M_k^{-1}(Y_k^i)^T v$, and send this information back to the master node. The master node then reduces this to form $M_k^{-1}(Y_k)^T v$, and broadcasts this vector back to the worker nodes. This time the worker nodes compute $Y_k^i M_k^{-1}(Y_k)^T v$ locally, and then this quantity is reduced by the master node; the cost of this communication is d. Namely, in order to compute Hessian vector products, the master node performs two aggregation, the bulk of the computation is done on the worker nodes and the communication cost is $m + 2d$.

Checking the SR1 Condition 3.2 As proposed in [5], at every iteration condition (3.2) is checked recursively by the master node. For each pair in memory, checking this condition amounts to a Hessian vector product as well as the use of inner products of the curvature pairs. Moreover, it requires the computation of $(M_k^{(j)})^{-1} \in \mathbb{R}^{j \times j}$, for $j = 1, \ldots, m$, where $M_k^{-1} = (M_k^{(m)})^{-1}$.

Inverse-Free Computation of M_k^{-1} The matrix M_k^{-1} is non-singular, depends solely on inner products of the curvature pairs, and is used in the the computation of Hessian vector products (3.1). This matrix is constructed recursively (its dimension grows with the memory) by the master node as condition (3.2) is checked. We propose an inverse-free approach for constructing this matrix. Suppose we have the matrix $(M_k^{(j)})^{-1}$, for some $j = 1, \dots, m-1$, and that the new curvature pair $(s_{k,j+1}, y_{k,j+1})$ satisfies 3.2. One can show that

$$(M_k^{(j+1)})^{-1} = \begin{bmatrix} (M_k^{(j)})^{-1} + \zeta (M_k^{(j)})^{-1} u v^T (M_k^{(j)})^{-1} & -\zeta (M_k^{(j)})^{-1} u \\ -\zeta v^T (M_k^{(j)})^{-1} & \zeta \end{bmatrix}$$

where $\zeta = 1/c - v^T (M_k^{(j)})^{-1} u$, $v^T = s_{k,j+1}^T Y_{k,1:l}$ and $Y_{k,1:l} = [y_{k,1}, \dots, y_{k,l}]$ for $l \leq j$, $u = v$, and $c = s_{k,j+1}^T y_{k,j+1}$; see Appendix A for the proof. The issue with the aforementioned approach is that it can be numerically unstable. Therefore, we propose another inverse-free approach that uses a QR decomposition of $M_k^{(j)}$ and updates this decomposition with every new curvature pair. The idea is based on utilizing and updating QR decompositions for solving a system of equations. Since the matrices $M_k^{(j)}$ are non-singular, the QR decompositions are well defined [54]. Notice that we do not require the explicit formation of $(M_k^{(j)})^{-1}$ or $(M_k)^{-1}$, rather we need $(M_k^{(j)})^{-1} Y_{k,1:l}^T v$ (for $l \leq j$) and $(M_k)^{-1} Y_k^T v$ to calculate $B_k^{(j)} v$ and $B_k v$, respectively. For a given j we do this as follows: (1) construct $M_k^{(j)}$ by updating the QR factorization of $M_k^{(j-1)}$; (2) solve the system $M_k^{(j)} \tilde{x} = Y_{k,1:l}^T v$, (3) set $B_k^{(j)} v = Y_{k,1:l}^T \tilde{x}$. We construct the QR factorization of M_k by updating the factorization of $M_k^{(m-1)}$ using the pairs $(s_{k,m}, y_{k,m})$. In our numerical experiments we use this approach, however, in the presentation of DS-LSR1 we use $(M_k)^{-1}$ explicitly since this makes the presentation clearer.

3.3 The Distributed S-LSR1 (DS-LSR1) Algorithm

We are now ready to present our proposed distributed variant of the S-LSR1 method. Pseudo-code for the DS-SLR1 method and the curvature pair sampling procedure are given in Algorithms 3 and 4, respectively. The distributed version of CG-Steihaug is given in Appendix B.2 Algorithm 6.

Algorithm 3 Distributed Sampled LSR1 (DS-LSR1)

Input: w_0 (initial iterate), Δ_0 (initial trust region radius), m (memory).

Master Node: **Worker Nodes ($i = 1, 2, \dots, \mathcal{K}$):**

for $k = 0, 1, 2, \dots$ **do**
  Broadcast: w_k $\longrightarrow$ Compute $F_i(w_k)$, $\nabla F_i(w_k)$
  Reduce: $F_i(w_k), \nabla F_i(w_k)$ to $F(w_k), \nabla F(w_k)$ $\longleftarrow$
  Compute new (M_k^{-1}, Y_k, S_k) pairs via Algorithm 4
  Compute p_k via Algorithm 6
  Broadcast: p_k, M_k^{-1} $\longrightarrow$ Compute $M_k^{-1}(Y_k^i)^T p_k$, $\nabla F_i(w_k)^T p_k$, $F_i(w_k + p_k)$
  Reduce: $M_k^{-1}(Y_k^i)^T p_k, \nabla F_i(w_k)^T p_k, F_i(w_k + p_k)$ to $M_k^{-1} Y_k^T p_k, \nabla F(w_k)^T p_k, F(w_k + p_k)$ $\longleftarrow$
  Broadcast: $M_k^{-1} Y_k^T p_k$ $\longrightarrow$ Compute $(Y_k^i)^T M_k^{-1} Y_k^T p_k$
  Reduce: $(Y_k^i)^T M_k^{-1} Y_k^i p_k$ to $B_k p_k = (Y_k)^T M_k^{-1} Y_k p_k$ $\longleftarrow$
  Compute $\rho_k = \frac{F(w_k) - F(w_k + p_k)}{m_k(0) - m_k(p_k)}$
  if $\rho_k \geq \eta_1$ **then** Set $w_{k+1} = w_k + p_k$ **else** Set $w_{k+1} = w_k$
  $\Delta_{k+1} = \texttt{adjustTR}(\Delta_k, \rho_k)$ [Appendix B.3]
end for

Algorithm 4 Construct new (S_k, Y_k) curvature pairs

Input: w_k (iterate), m (memory), $S_k = [\,]$, $Y_k = [\,]$ (curvature pair containers).

Master Node: **Worker Nodes ($i = 1, 2, \dots, \mathcal{K}$):**

Broadcast: $\bar{S}_k$ and w_k $\longrightarrow$ Compute $\bar{Y}_{k,i} = \nabla^2 F_i(w_k) \bar{S}_k$
Reduce: $\bar{S}_k^T \bar{Y}_{k,i}$ to $\bar{S}_k^T \bar{Y}_k$ and $\bar{Y}_k^T \bar{S}_k \bar{S}_k^T \bar{Y}_k$ $\longleftarrow$ Compute $\bar{S}_k^T \bar{S}_k$ and $\bar{S}_k^T \bar{Y}_{k,i}$
Check the SR1 condition (2.5) and construct M_k^{-1} recursively
  using $\bar{S}_k^T \bar{S}_k$, $\bar{S}_k^T \bar{Y}_k$ and approximation of $\bar{Y}_k^T \bar{Y}_k$ and construct list of accepted pairs S_k and Y_k
Broadcast: the list of accepted curvature pairs

Output: M^{-1}, Y_k, S_k

3.4 Complexity Analysis - Comparison of Methods

In this section, we compare the naive distributed implementation of the S-LSR1 method and the DS-LSR1 method. Specifically, we discuss the amount of information communicated at every iteration and the amount of computation performed by the nodes. Tables 1 and 2 summarize the communication and computation costs, respectively; see Appendix B.5 for details on the quantities presented in the tables.

Table 1: Communication Details.

	Naive DS-LSR1	DS-LSR1
Broadcast:	w_k	w_k, p_k, M^{-1}
Reduce:	$\nabla F_i(w_k), F_i(w_k), S_{k,i}, Y_{k,i}$	$\nabla F_i(w_k), F_i(w_k), S_k^T Y_{k,i}, Y_{k,i} M_k^{-1} Y_{k,i} p_k, M_k^{-1} Y_{k,i}^T p_k$

Table 2: Computation Details.

	Naive DS-LSR1	DS-LSR1
Worker:	$\nabla F_i(w_k), F_i(w_k), Y_{k,i}$	$\nabla F_i(w_k), F_i(w_k), Y_{k,i}, S_{k,i}^T Y_{k,i}$ $M_k^{-1} Y_{k,i}^T p_k, Y_{k,i} M_k^{-1} Y_k^T p_k$, CG
Master:	$M_k^{-1}, w_{k+1}, B_k d$, CG	M_k^{-1}, w_{k+1}

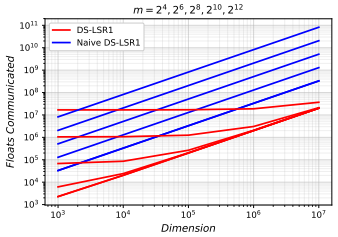

Figure 4: Number of floats communicated ate every iteration for different dimension d and memory size m.

As is clear from Tables 1 and 2 the amount of information communicated in the naive implementation $(2md + d + 1)$ is significantly larger than that in the DS-LSR1 method $(m^2 + 2d + m + 1)$. Note, $m < d$. This can also be seen in Figure 4 where we show for different dimension d and memory m the number of floats communicated at every iteration; see Appendix B.6. To put this into perspective, consider a training problem where $d = 9.2M$ (e.g., VGG11 network [53]) and $m = 256$, DS-LSR1 and Naive DS-LSR1 need to communicate $0.0688\,GB$ and $8.8081\,GB$, respectively, per iteration. In terms of computation, it is clear that in the naive approach the amount of computation is not balanced between the master node and the worker nodes, whereas for DS-LSR1 the quantities are balanced.

4 Numerical Experiments

In this section, we present a thorough numerical investigation of the proposed DS-LSR1 method.[2] We first show the scaling properties of the method and compare it to the naive implementation. We then deconstruct the main computational elements of the method and show how they scale in terms of memory. Finally, we illustrate the performance of DS-LSR1 on a neural network training task.

4.1 Scaling

In this section, we present the weak and strong scaling properties of the DS-LSR1 method.

Weak Scaling - Different networks We begin with the weak scaling properties of the method. We considered two different networks: (1) `Shallow`, one hidden layer with different number of nodes, and (2) `Deep`, 7 hidden layers with different number on nodes in each layer, and the MNIST dataset [35]; see Appendix C.2 for details. For these experiments the memory was set to $m = 64$. Figure 5 shows the time per iteration for the DS-LSR1 method for different number of variables and batch sizes.

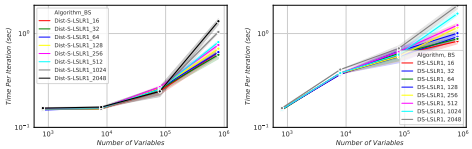

Figure 5: **Weak Scaling:** Time per iteration (sec) versus number of variables for `Shallow` (left) and `Deep` (right) networks.

Strong Scaling - Increasing number of nodes Next, we show the strong scaling properties of DS-LSR1. Here, we fix the problem size (LeNet, CIFAR10, $d = 62006$ [35]), vary the number of compute nodes and measure the speed-up achieved. Figure 6 illustrates the speedup of our proposed method as well as the naive distributed implementation for $m = 256$. As is clear, our method achieves near linear speedup as the number of nodes increases, and the speedup is better than that of the naive approach.

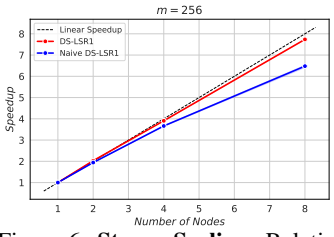

Figure 6: **Strong Scaling:** Relative speedup.

[2] All algorithms are implemented in Python (PyTorch library [44]), using the MPI for Python distributed environment. The experiments were conducted on XSEDE clusters [57] using GPU nodes. Each physical node includes 4 K80 GPUs, and each MPI process is assigned to a distinct GPU, i.e., 4 MPI processes for each node. We will release our source code upon publication of the paper.

We normalized the speedup of each method with respect to the performance of that method with a single node, i.e., Figure 6 depicts the relative speedup for each method. We should note, however, that the times of our proposed method are lower than the respective times for the naive implementation. The reasons for this are: (1) DS-LSR1 is inverse free, and (2) the amount of information communicated is significantly smaller. See Appendix C.3 for more results.

Scaling of Different Components of DS-LSR1 Here we deconstruct the main components of the DS-LSR1 method and illustrate the scaling with respect to memory. Specifically, Figure 7 shows the scaling for: (1) reduce time/iteration; (2) time/iteration; (3) CG time/iteration; (4) time to sample S, Y pairs/iteration. For all these plots, we ran 10 iterations and averaged the time, and also show the variability. As is clear for the figure, our proposed method has lower times for all components of the algorithm. Again, we attribute this to the fact that our approach: (1) requires less information exchange (communication) per iteration; and (2) is inverse-free.

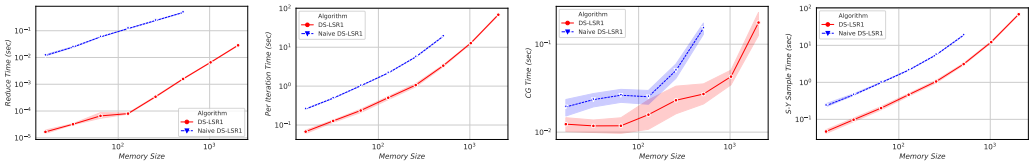

Figure 7: Time (sec) for different components of the DS-LSR1 method with respect to memory.

To further highlight the efficiency of our proposed method, in terms of communications, we plot the ratio of the reduce time per iteration of DS-SLR1 to the reduce time per iteration of the naive distributed implementation in Figure 8. For these experiments we set the memory size to $m = 64$. As is clear, the reduce time for DS-LSR1 is significantly smaller than that of the naive approach. As expected, this is especially true when the number of variables in the problem d is large.

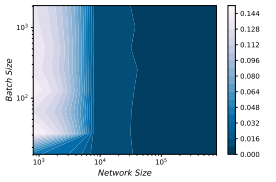

Figure 8: Ratios of reduce times per iteration with respect to batch size and dimension.

4.2 Performance of DS-LSR1

In this section, we show the actual performance of the DS-LSR1 method on a neural network training task; LeNet [35], CIFAR10, $n = 50000$, $d = 62006$. For this experiment we set memory to $m = 256$. In Figure 9, we illustrate the training accuracy in terms of wall clock time and amount of data (GB) communication (left and center plots, respectively), for different number of nodes. As expected, when using larger number of compute nodes training is faster, i.e., given a fixed time budget, the accuracy achieved when using more nodes is higher. Similar results were obtained for testing accuracy; see Appendix C.4. We also plot, the performance of the naive implementation. Firstly, to show that the accuracy achieved is comparable, and thus the two approaches are identical. And, secondly, to show that one can train faster using our proposed method.

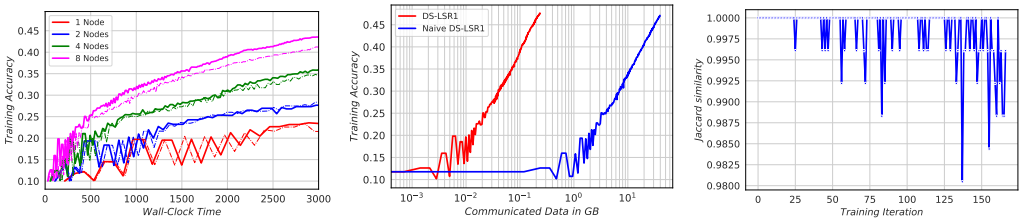

Figure 9: Performance of DS-LSR1 on CIFAR10 dataset with different number of nodes.

The final thing we show in this experiment is that the curvature pairs chosen by our approach are almost identical to those chosen by the naive approach even though we use an approximation (via sketching) when checking the SR1 condition. To this end, for all the runs in Figure 9, we show the Jaccard similarity for the sets of curvature pairs selected by the methods. As is clear, the pairs are almost identical, with slight differences on only a small fraction of iterations; see Figure 9 right plot.

5 Final Remarks

This paper describes a scalable distributed implementation of the sampled LSR1 method which is communication-efficient, has favorable work-load balancing across nodes and that is matrix-free and inverse-free. The method leverages the compact representation of SR1 matrices and uses sketching techniques to drastically reduce the amount of data communicated at every iteration as compared to a naive distributed implementation. The DS-LSR1 method scales well in terms of both the dimension of the problem and the number of data points and performs well on standard neural network tasks.

Acknowledgements

This work was partially supported by the U.S. National Science Foundation, under award numbers NSF:CCF:1618717, NSF:CMMI:1663256 and NSF:CCF:1740796, DARPA Lagrange award HR-001117S0039, and XSEDE Startup grant IRI180020.

References

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[2] Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using kronecker-factored approximations. 2016.

[3] Albert S Berahas, Raghu Bollapragada, and Jorge Nocedal. An investigation of newton-sketch and subsampled newton methods. *arXiv preprint arXiv:1705.06211*, 2017.

[4] Albert S Berahas, Majid Jahani, and Martin Takác. Sampled quasi-newton methods for deep learning.

[5] Albert S. Berahas, Majid Jahani, and Martin Takác. Quasi-newton methods for deep learning: Forget the past, just sample. *arXiv preprint arXiv: 1901.09997*, 2019.

[6] Albert S Berahas, Jorge Nocedal, and Martin Takác. A multi-batch l-bfgs method for machine learning. In *Advances in Neural Information Processing Systems*, pages 1055–1063, 2016.

[7] Albert S Berahas and Martin Takáč. A robust multi-batch l-bfgs method for machine learning. *arXiv preprint arXiv:1707.08552*, 2017.

[8] Raghu Bollapragada, Richard H Byrd, and Jorge Nocedal. Exact and inexact subsampled newton methods for optimization. *IMA Journal of Numerical Analysis*, 2016.

[9] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[10] Léon Bottou and Yann L Cun. Large scale online learning. In *Advances in neural information processing systems*, pages 217–224, 2004.

[11] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[12] Johannes Brust, Jennifer B Erway, and Roummel F Marcia. On solving l-sr1 trust-region subproblems. *Computational Optimization and Applications*, 66(2):245–266, 2017.

[13] Richard H Byrd, Gillian M Chin, Will Neveitt, and Jorge Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

[14] Richard H Byrd, Samantha L Hansen, Jorge Nocedal, and Yoram Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[15] Richard H Byrd, Humaid Fayez Khalfan, and Robert B Schnabel. Analysis of a symmetric rank-one trust region method. *SIAM Journal on Optimization*, 6(4):1025–1039, 1996.

[16] Richard H. Byrd, Jorge Nocedal, and Robert B. Schnabel. Representations of quasi-newton matrices and their use in limited memory methods. *Math. Program.*, 63:129–156, 1994.

[17] Weizhu Chen, Zhenghao Wang, and Jingren Zhou. Large-scale l-bfgs using mapreduce. In *Advances in Neural Information Processing Systems*, pages 1332–1340, 2014.

[18] Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

[19] Andrew R Conn, Nicholas IM Gould, and Ph L Toint. Convergence of quasi-newton matrices generated by the symmetric rank one update. *Mathematical programming*, 50(1-3):177–195, 1991.

[20] Frank Curtis. A self-correcting variable-metric algorithm for stochastic optimization. In *International Conference on Machine Learning*, pages 632–641, 2016.

[21] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[22] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in neural information processing systems*, pages 1646–1654, 2014.

[23] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[24] Jennifer B Erway, Joshua Griffin, Roummel F Marcia, and Riadh Omheni. Trust-region algorithms for training responses: Machine learning methods using indefinite hessian approximations. *arXiv preprint arXiv:1807.00251*, 2018.

[25] Jennifer B Erway, Joshua Griffin, Riadh Omheni, and Roummel Marcia. Trust-region optimization methods using limited-memory symmetric rank-one updates for off-the-shelf machine learning. 2017.

[26] Siddharth Gopal and Yiming Yang. Distributed training of large-scale logistic models. In *International Conference on Machine Learning*, pages 289–297, 2013.

[27] Robert Gower, Donald Goldfarb, and Peter Richtárik. Stochastic block bfgs: Squeezing more curvature out of data. In *International Conference on Machine Learning*, pages 1869–1878, 2016.

[28] Majid Jahani, Xi He, Chenxin Ma, Aryan Mokhtari, Dheevatsa Mudigere, Alejandro Ribeiro, and Martin Takác. Grow your samples and optimize better via distributed newton cg and accumulating strategy.

[29] Majid Jahani, Xi He, Chenxin Ma, Aryan Mokhtari, Dheevatsa Mudigere, Alejandro Ribeiro, and Martin Takáč. Efficient distributed hessian free algorithm for large-scale empirical risk minimization via accumulating sample strategy. *arXiv preprint arXiv:1810.11507*, 2018.

[30] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[31] Nitish Shirish Keskar and Albert S Berahas. adaqn: An adaptive quasi-newton algorithm for training rnns. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 1–16. Springer, 2016.

[32] H Fayez Khalfan, Richard H Byrd, and Robert B Schnabel. A theoretical and experimental study of the symmetric rank-one update. *SIAM Journal on Optimization*, 3(1):1–24, 1993.

[33] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[34] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Asaga: Asynchronous parallel saga. In *Artificial Intelligence and Statistics*, pages 46–54, 2017.

[35] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[36] Edo Liberty. Simple and deterministic matrix sketching. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–588. ACM, 2013.

[37] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[38] Xuehua Lu. *A study of the limited memory SR1 method in practice*. University of Colorado at Boulder, 1996.

[39] James Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[40] Aryan Mokhtari and Alejandro Ribeiro. Global convergence of online limited memory bfgs. *The Journal of Machine Learning Research*, 16(1):3151–3181, 2015.

[41] Lam M Nguyen, Jie Liu, Katya Scheinberg, and Martin Takáč. Sarah: A novel method for machine learning problems using stochastic recursive gradient. In *International Conference on Machine Learning*, pages 2613–2621, 2017.

[42] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

[43] Jorge Nocedal and Stephen J Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, second edition, 2006.

[44] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[45] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[46] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alexander J Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. In *Advances in Neural Information Processing Systems*, pages 2647–2655, 2015.

[47] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[48] David P Rodgers. Improvements in multiprocessor system design. *ACM SIGARCH Computer Architecture News*, 13(3):225–231, 1985.

[49] Farbod Roosta-Khorasani and Michael W. Mahoney. Sub-sampled newton methods. *Mathematical Programming*, 2018.

[50] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

[51] Nicol N Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-newton method for online convex optimization. In *Artificial Intelligence and Statistics*, pages 436–443, 2007.

[52] Ohad Shamir, Nati Srebro, and Tong Zhang. Communication-efficient distributed optimization using an approximate newton-type method. In *International conference on machine learning*, pages 1000–1008, 2014.

[53] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[54] Gilbert Strang. *Introduction to linear algebra*, volume 3.

[55] Martin Takác, Avleen Singh Bijral, Peter Richtárik, and Nati Srebro. Mini-batch primal and dual methods for svms. In *ICML (3)*, pages 1022–1030, 2013.

[56] Choon Hui Teo, Alex Smola, SVN Vishwanathan, and Quoc Viet Le. A scalable modular convex solver for regularized risk minimization. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 727–736. ACM, 2007.

[57] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. Xsede: accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, 2014.

[58] John Tsitsiklis, Dimitri Bertsekas, and Michael Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control*, 31(9):803–812, 1986.

[59] David P Woodruff. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 10(1–2):1–157, 2014.

[60] Peng Xu, Farbod Roosta-Khorasan, and Michael W Mahoney. Second-order optimization for non-convex machine learning: An empirical study. *arXiv preprint arXiv:1708.07827*, 2017.

[61] Peng Xu, Farbod Roosta-Khorasani, and Michael W Mahoney. Newton-type methods for non-convex optimization under inexact hessian information. *arXiv preprint arXiv:1708.07164*, 2017.

[62] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[63] Yuchen Zhang and Lin Xiao. Communication-efficient distributed optimization of self-concordant empirical loss. *arXiv preprint arXiv:1501.00263*, 2015.

[64] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.

A Theoretical Results and Proofs

In this section, we prove a theoretical result about the matrix $(M_k^{(j)})^{-1}$.

Lemma A.1. *The matrix $M_k^{(j+1)}$, for $j = 0, \ldots, m-1$, has the form:*

$$M_k^{(j+1)} = \left[\begin{array}{c|c} M_k^{(j)} & u \\ \hline v^T & c \end{array}\right], \tag{A.1}$$

where $v^T = s_{k,j+1}^T Y_{k,1:l}$ and $l \leq j$, $u = v$ and $c = s_{k,j+1}^T y_{k,j+1}$, and is nonsingular. Moreover, its inverse can be calculated as following:

$$(M_k^{(j+1)})^{-1} = \left[\begin{array}{c|c} (M_k^{(j)})^{-1} + \zeta (M_k^{(j)})^{-1} u v^T (M_k^{(j)})^{-1} & -\zeta (M_k^{(j)})^{-1} u \\ \hline -\zeta v^T (M_k^{(j)})^{-1} & \zeta \end{array}\right] \tag{A.2}$$

where $\zeta = \dfrac{1}{c - v^T (M_k^{(j)})^{-1} u}$.

Proof. It is trivial to show that $M_k^{(j+1)}$ shown in (A.1) is equivalent to the corresponding matrix in (2.3). Moreover, the second part of the lemma follows immediately from the fact that $M_k^{(i+1)}$ is itself non-singular and symmetric as shown in [16]. Lets consider the following matrix $M_k^{(i+1)}$:

$$M_k^{(j+1)} = \left[\begin{array}{c|c} M_k^{(j)} & u \\ \hline v^T & c \end{array}\right] \tag{A.3}$$

We know that $M_k^{(i)}$ is invertible, and in the following by simple linear algebra, we calculate the inverse of $M_k^{(i+1)}$:

$$\left[\begin{array}{c|c|c|c} M_k^{(j)} & u & I & 0 \\ \hline v^T & c & 0 & 1 \end{array}\right] \Rightarrow \left[\begin{array}{c|c|c|c} I & (M_k^{(j)})^{-1} u & (M_k^{(j)})^{-1} & 0 \\ \hline v^T & c & 0 & 1 \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{c|c|c|c} I & (M_k^{(j)})^{-1} u & (M_k^{(j)})^{-1} & 0 \\ \hline 0 & c - v^T (M_k^{(j)})^{-1} u & -v^T (M_k^{(j)})^{-1} & 1 \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{c|c|c|c} I & (M_k^{(j)})^{-1} u & (M_k^{(j)})^{-1} & 0 \\ \hline 0 & 1 & \dfrac{-v^T (M_k^{(j)})^{-1}}{c - v^T (M_k^{(j)})^{-1} u} & \dfrac{1}{c - v^T (M_k^{(j)})^{-1} u} \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{cc|c|c} I & 0 & (M_k^{(j)})^{-1} + \dfrac{(M_k^{(j)})^{-1} u v^T (M_k^{(j)})^{-1}}{c - v^T (M_k^{(j)})^{-1} u} & \dfrac{-(M_k^{(j)})^{-1} u}{c - v^T (M_k^{(j)})^{-1} u} \\ \hline 0 & 1 & \dfrac{-v^T (M_k^{(j)})^{-1}}{c - v^T (M_k^{(j)})^{-1} u} & \dfrac{1}{c - v^T (M_k^{(j)})^{-1} u} \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{cc|c|c} I & 0 & (M_k^{(j)})^{-1} + \zeta (M_k^{(j)})^{-1} u v^T (M_k^{(j)})^{-1} & -\zeta (M_k^{(j)})^{-1} u \\ \hline 0 & 1 & -\zeta v^T (M_k^{(j)})^{-1} & \zeta \end{array}\right]$$

The last line is by putting $\zeta = \dfrac{1}{c - v^T (M_k^{(j)})^{-1} u}$.

□

Lemma A.1 describes a recursive method for computing $(M_k^{(j)})^{-1} \in \mathbb{R}^{j \times j}$, for $j = 1, \ldots, m$. Specifically, one can calculate $(M_k^{(j+1)})^{-1}$ using $(M_k^{(j)})^{-1}$. We should note, that the first matrix $(M_k^{(1)})^{-1}$ is simply a number. Overall, this procedure allows us to compute $(M_k^{(j)})^{-1}$ without explicitly computing an inverse.

B Additional Algorithm Details

In this section, we present additional details about the S-LSR1 and DS-LSR1 algorithms discussed in the Sections 2 and 3.

B.1 CG Steihaug Algorithm - Serial

In this section, we describe CG-Steihaug Algorithm [43, Chapter 7] which is used for computing the search direction p_k.

Algorithm 5 CG-Steihaug (Serial)

Input: ϵ (termination tolerance), $\nabla F(w_k)$ (current gradient).

Set $z_0 = 0$, $r_0 = \nabla F(w_k)$, $d_0 = -r_0$
if $\|r_0\| < \epsilon$ **then**
  return $p_k = z_0 = 0$
end if
for $j = 0, 1, 2, \ldots$ **do**
  if $d_j^T B_k d_j \leq 0$ **then**
    Find $\tau \geq 0$ such that $p_k = z_j + \tau d_j$ minimizes $m_k(p_k)$ and satisfies $\|p_k\| = \Delta_k$
    return p_k
  end if
  Set $\alpha_j = \dfrac{r_j^T r_j}{d_j^T B_k d_j}$ and $z_{j+1} = z_j + \alpha_j d_j$
  if $\|z_{j+1}\| \geq \Delta_k$ **then**
    Find $\tau \geq 0$ such that $p_k = z_j + \tau d_j$ and satisfies $\|p_k\| = \Delta_k$
    return p_k
  end if
  Set $r_{j+1} = r_j + \alpha_j B_k d_j$
  if $\|r_{j+1}\| < \epsilon_k$ **then**
    return $p_k = z_{j+1}$
  end if
  Set $\beta_{j+1} = \dfrac{r_{j+1}^T r_{j+1}}{r_j^T r_j}$ and $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$
end for

B.2 CG Steihaug Algorithm - Distributed

In this section, we describe a distributed variant of CG Steihaug algorithm that is used as a subroutine of the DS-LSR1 method. The manner in which Hessian vector products are computed was discussed in Section 3.

Algorithm 6 CG-Steihaug (Distributed)

Input: ϵ (termination tolerance), $\nabla F(w_k)$ (current gradient), M_k^{-1}.

Master Node: **Worker Nodes ($i = 1, 2, \ldots, \mathcal{K}$):**

Set $z_0 = 0$, $r_0 = \nabla F(w_k)$, $d_0 = -r_0$
if $\|r_0\| < \epsilon_k$ **then**
  return $p_k = z_0 = 0$
end if
for $j = 0, 1, 2, \ldots$ **do**
  Broadcast: d_j, M_k^{-1} $\longrightarrow$ Compute $M_k^{-1}(Y_k^i)^T d_j$
  Reduce: $M_k^{-1}(Y_k^i)^T d_j$ to $M_k^{-1} Y_k^T d_j$ $\longleftarrow$
  Broadcast: $M_k^{-1} Y_k^T d_j$ $\longrightarrow$ Compute $Y_k^i M_k^{-1} Y_k^T d_j$
  Reduce: $Y_k^i M_k^{-1} Y_k^T d_j$ to $B_k d_j = Y_k M_k^{-1} Y_k^T d_j$ $\longleftarrow$
  if $d_j^T B_k d_j \leq 0$ **then**
    Find $\tau \geq 0$ such that $p_k = z_j + \tau d_j$ minimizes $m_k(p_k)$ and satisfies $\|p_k\| = \Delta_k$
    return p_k
  end if
  Set $\alpha_j = \dfrac{r_j^T r_j}{d_j^T B_k d_j}$ and $z_{j+1} = z_j + \alpha_j d_j$
  if $\|z_{j+1}\| \geq \Delta_k$ **then**
    Find $\tau \geq 0$ such that $p_k = z_j + \tau d_j$ and satisfies $\|p_k\| = \Delta_k$
    return p_k
  end if
  Set $r_{j+1} = r_j + \alpha_j B_k d_j$
  if $\|r_{j+1}\| < \epsilon_k$ **then**
    return $p_k = z_{j+1}$
  end if
  Set $\beta_{j+1} = \dfrac{r_{j+1}^T r_{j+1}}{r_j^T r_j}$ and $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$
end for

B.3 Trust-Region Management Subroutine

In this section, we present the Trust-Region management subroutine $\Delta_{k+1} = \texttt{adjustTR}(\Delta_k, \rho_k)$. See [43, Chapter 5] for further details.

Algorithm 7 $\Delta_{k+1} = \texttt{adjustTR}(\Delta_k, \rho_k, \eta_2, \eta_3, \gamma_1, \zeta_1, \zeta_2)$: Trust-Region management subroutine

Input: Δ_k (current trust region radius), $0 \leq \eta_3 < \eta_2 < 1$, $\gamma_1 \in (0, 1)$, $\zeta_1 > 1$, $\zeta_2 \in (0, 1)$ (trust region parameters).

if $\rho_k > \eta_2$ **then**
  if $\|p_k\| \leq \gamma_1 \Delta_k$ **then**
    Set $\Delta_{k+1} = \Delta_k$
  else
    Set $\Delta_{k+1} = \zeta_1 \Delta_k$
  end if
else if $\eta_3 \leq \rho_k \leq \eta_2$ **then**
  Set $\Delta_{k+1} = \Delta_k$
else
  $\Delta_{k+1} = \zeta_2 \Delta_k$
end if

B.4 Load Balancing

In distributed algorithms, it is very important to have work-load balancing across nodes. In order for an algorithm to be scalable, every machine (worker) should have similar amount of assigned computation, and each machine should be equally busy. According to Amdahl's law [48] if the parallel/distributed algorithm runs t portion of time only on one of the machines (e.g., the master node), the theoretical speedup (SU) is limited to at most

$$SU \leq \frac{1}{t + \frac{(1-t)}{\mathcal{K}}}. \tag{B.1}$$

As is clear from Tables 1 and 2, the DS-LSR1 method makes each machine almost equally busy, and as a result DS-LSR1 has a near linear speedup. On the other hand, in the naive DS-LSR1 approach the master node is significantly busier than the remainder of the nodes, and thus by Adamhl's law, the speedup will not be linear and is bounded above by (B.1).

B.5 Communication and Computation Details

In this section, we present details about the quantities that are communicated and computed at every iteration of the distributed S-LSR1 methods. All the quantities below are in Tables 1 and 2.

Table 3: Details of quantities communicated and computed.

Variable	**Dimension**
w_k	$d \times 1$
$F(w_k), F_i(w_k)$	1
$\nabla F(w_k), \nabla F_i(w_k)$	$d \times 1$
p_k	$d \times 1$
$S_k, S_{k,i}$	$d \times m$
$Y_k, Y_{k,i}$	$d \times m$
$S_k^T Y_{k,i}, S_{k,i}^T Y_{k,i}$	$m \times m$
M_k^{-1}	$m \times m$
$B_k d$	$d \times 1$
$M_k^{-1} Y_{k,i}^T p_k$	$m \times 1$
$Y_{k,i} M_k^{-1} Y_k^T p_k$	$d \times 1$
M_k^{-1}	$m \times m$

B.6 Floats Communicated per Iteration

In this section, we should the number of floats communicated per iteration for DS-LSR1 and naive DS-LSR1 for different memory size and dimension.

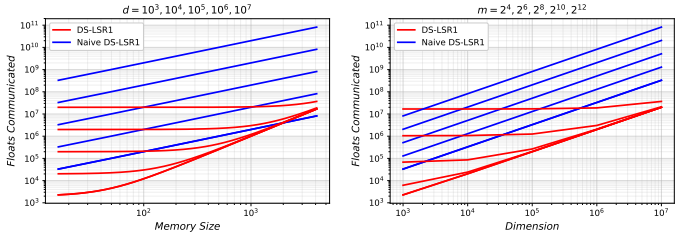

Figure 10: Number of floats communicated ate every iteration for different dimension d and memory size m.

C Additional Numerical Experiments and Experimental Details

In this section, we present additional experiments and experimental details.

C.1 Initial Hessian Approximation $B_k^{(0)}$

In this section, we show additional results motivating the use of $B_k^{(0)}$. Figure 11, is identical to Figure 2. Figure 12 shows similar results for a larger problem. See [5] for details about the problems.

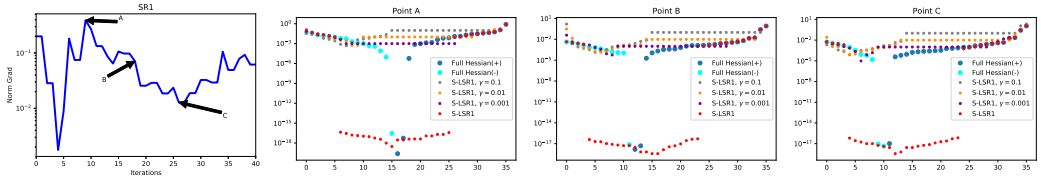

Figure 11: Comparison of the eigenvalues of S-LSR1 for different γ (@ A, B, C) for a `small` toy classification problem.

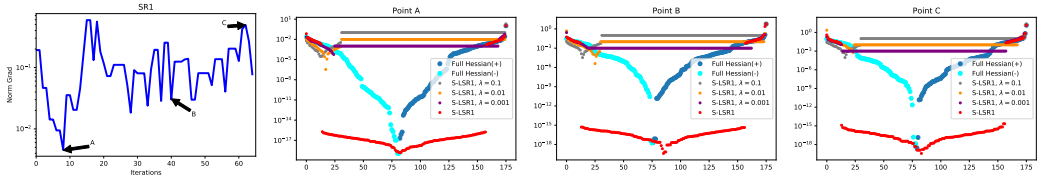

Figure 12: Comparison of the eigenvalues of S-LSR1 for different γ (@ A, B, C) for a `medium` toy classification problem.

C.2 `Shallow` and `Deep` Network Details

In this section, we describe the networks used in the weak scaling experiments (Section 4.1). For the problems corresponding to the Tables 4 and 5 we used ReLU activation functions and soft-max cross-entropy loss.

Table 4: Details for `Shallow` Networks.

Network	# Hidden Layers	# Nodes/ Layer	d
1	1	1	805
2	1	10	7960
4	1	100	79510
3	1	1000	795010

Table 5: Details for `Deep` Networks.

Network	# Hidden Layers	# Nodes/ Layer	d
1	7	2-2-2-2-2-2-2	817
2	7	10-10-10-10-10-10-10	8620
4	7	100-100-100-10-10-10-10	100150
3	7	1000-100-100-10-10-10-10	896650

C.3 Strong Scaling

In this section, we show the strong scaling properties of DS-LSR1 and naive DS-LSR1 for different memory sizes. The problem details for these experiments were as follows: LeNet, CIFAR10, $d = 62006$, [35].

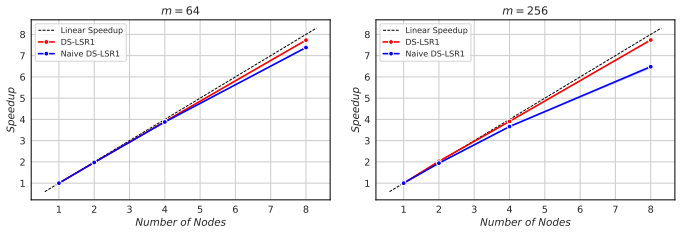

Figure 13: **Strong Scaling:** Relative speedup for different number of compute nodes and different memory levels: 64 (left), 256 (right).

C.4 Performance of DS-LSR1

In this section, we show training and testing accuracy in terms of wall clock time and amount of data communicated (in GB).

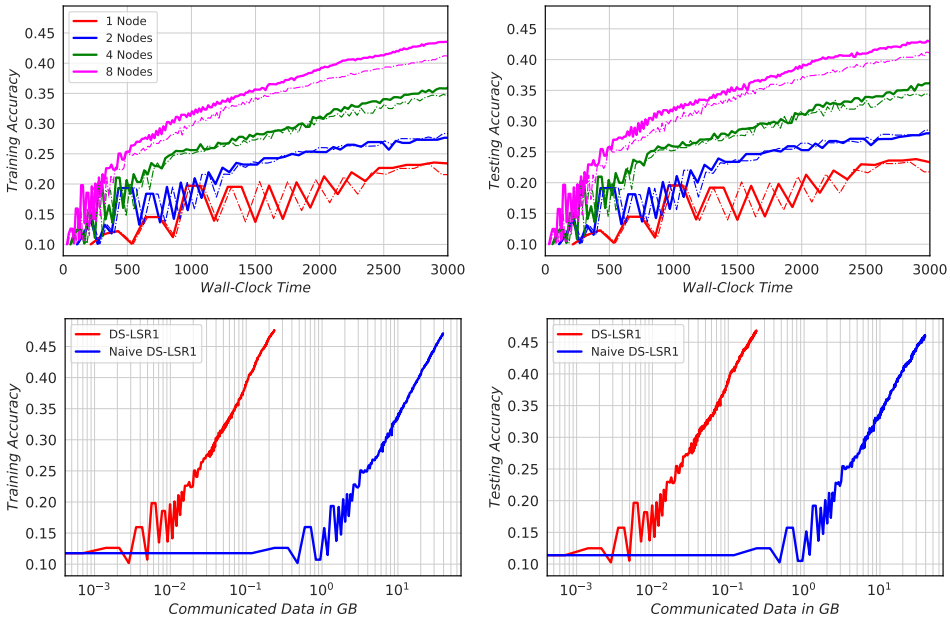

Figure 14: Performance of DS-LSR1 on CIFAR10 dataset with different number of nodes.