PARMAC: DISTRIBUTED OPTIMISATION OF NESTED FUNCTIONS, WITH APPLICATION TO LEARNING BINARY AUTOENCODERS

Miguel Á. Carreira-Perpiñán [1] **Mehdi Alizadeh** [1]

ABSTRACT

Many powerful machine learning models are based on the composition of multiple processing layers, such as deep nets, which gives rise to nonconvex objective functions. A general, recent approach to optimise such "nested" functions is the *method of auxiliary coordinates (MAC)*. MAC introduces an auxiliary coordinate for each data point in order to decouple the nested model into independent submodels. This decomposes the optimisation into steps that alternate between training single layers and updating the coordinates. It has the advantage that it reuses existing single-layer algorithms, introduces parallelism, and does not need to use chain-rule gradients, so it works with nondifferentiable layers. We describe ParMAC, a distributed-computation model for MAC. This trains on a dataset distributed across machines while limiting the amount of communication so it does not obliterate the benefit of parallelism. ParMAC works on a cluster of machines with a circular topology and alternates two steps until convergence: one step trains the submodels in parallel using stochastic updates, and the other trains the coordinates in parallel. Only submodel parameters, no data or coordinates, are ever communicated between machines. ParMAC exhibits high parallelism, low communication overhead, and facilitates data shuffling, load balancing, fault tolerance and streaming data processing. We study the convergence of ParMAC and its parallel speedup, and implement ParMAC using MPI to learn binary autoencoders for fast image retrieval, achieving nearly perfect speedups in a 128-processor cluster with a training set of 100 million images.

1 INTRODUCTION

Serial computing has reached a plateau and parallel, distributed architectures are becoming widely available, from machines with a few cores to cloud computing with 1000s of machines. The combination of powerful nested models with large datasets is a key ingredient to solve difficult problems in machine learning, computer vision and other areas, and it underlies recent successes in deep learning (Hinton et al., 2012; Le et al., 2012; Dean et al., 2012). Unfortunately, parallel computation is not easy, and many good serial algorithms do not parallelise well. The cost of communicating (through the memory hierarchy or a network) greatly exceeds the cost of computing, both in time and energy, and will continue to do so for the foreseeable future. Thus, good parallel algorithms must minimise communication and maximise computation per machine, while creating sufficiently many subproblems (ideally independent) to benefit from as many machines as possible. The load (in runtime) on each machine should be approximately equal. Faults become more frequent as the number of machines increases, particularly if they are inexpensive machines. Machines may be heterogeneous and differ in CPU and memory; this is the case with initiatives such as SETI@home (Anderson et al., 2002), which may become an important source of distributed computation in the future. Big data applications have additional restrictions. The size of the data means it cannot be stored on a single machine, so distributed-memory architectures are necessary. Sending data between machines is prohibitive because of the size of the data and the high communication costs. In some applications, more data is collected than can be stored, so data must be regularly discarded. In others, such as sensor networks, limited battery life and computational power imply that data must be processed locally.

In this paper, we focus on machine learning models of the form $\mathbf{y} = \mathbf{F}_{K+1}(\dots \mathbf{F}_2(\mathbf{F}_1(\mathbf{x}))\dots)$, i.e., consisting of a nested mapping from the input $\mathbf{x}$ to the output $\mathbf{y}$. Such *nested models* involve multiple parameterised layers of processing and include deep neural nets, cascades for object recognition in computer vision or for phoneme classification in speech processing, wrapper approaches to classification or regression, and various combinations of feature extraction/learning and preprocessing prior to some learning task. Nested and hierarchical models are ubiquitous

[1]EECS, School of Engineering, University of California, Merced, USA. Correspondence to: Miguel Á. Carreira-Perpiñán <mcarreira-perpinan@ucmerced.edu>.

Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA, 2019.

in machine learning because they provide a way to construct complex models by the composition of simple layers. However, training nested models is difficult even in the serial case because *function composition generally produces nonconvex functions*, which makes gradient-based optimisation difficult and slow, and sometimes inapplicable (e.g. with nonsmooth or discrete layers).

Our starting point is a recently proposed technique to train nested models, the *method of auxiliary coordinates (MAC)* (Carreira-Perpiñán & Wang, 2012; 2014). This reformulates the optimisation into an iterative procedure that alternates training submodels independently with coordinating them. It introduces significant model and data parallelism, can often train the submodels using existing algorithms, and has convergence guarantees with differentiable functions to a local stationary point, while it also applies with nondifferentiable or even discrete layers, such as binary autoencoders (Carreira-Perpiñán & Raziperchikolaei, 2015). MAC has been applied to various nested models (Carreira-Perpiñán & Wang, 2014; Wang & Carreira-Perpiñán, 2014; Carreira-Perpiñán & Raziperchikolaei, 2015; Raziperchikolaei & Carreira-Perpiñán, 2016; Carreira-Perpiñán & Vladymyrov, 2015), and several variations of it have been proposed (e.g. Lee et al., 2015; Taylor et al., 2016; Jaderberg et al., 2017; Askari et al., 2018; Ororbia et al., 2018). However, the original papers proposing MAC (Carreira-Perpiñán & Wang, 2012; 2014) did not address how to run MAC on a distributed computing architecture, where communication between machines is far costlier than computation. This paper proposes *ParMAC*, a parallel, distributed framework to learn nested models using MAC, analyses its parallel speedup and convergence, implements it in MPI for the problem of learning binary autoencoders, and demonstrates its ability to train on large datasets and achieve large speedups on a distributed cluster.

2 Related Work

Distributed optimisation and large-scale machine learning have been steadily gaining interest in recent years, with the development of parallel computation abstractions tailored (or applicable to) to machine learning, such as Spark (Zaharia et al., 2010), GraphLab (Low et al., 2012), Petuum (Xing et al., 2015) or TensorFlow (Abadi et al., 2015), which have the goal of making cloud computing easily available to train machine learning models. Most work has centred on *convex* optimisation, particularly when the objective function has the form of empirical risk minimisation (data fitting term plus regulariser) (Cevher et al., 2014). This includes many important models in machine learning, such as linear regression, LASSO, logistic regression or SVMs. Such work is typically based on stochastic gradient descent (SGD) (Bottou, 2010), coordinate descent (CD) (Wright, 2016) or the alternating direction method of multipliers (ADMM) (Boyd et al., 2011). This has resulted in several variations of parallel SGD (Bertsekas, 2011; Zinkevich et al., 2010; Niu et al., 2011), parallel CD (Bradley et al., 2011; Richtárik & Takáč, 2013; Liu & Wright, 2015) and parallel ADMM (Boyd et al., 2011; Ouyang et al., 2013; Zhang & Kwok, 2014).

Little work has addressed *nonconvex* models. Most of it has focused on deep nets (Dean et al., 2012; Le et al., 2012). For example, Google's DistBelief (Dean et al., 2012) uses asynchronous parallel SGD (with gradients for the full model computed with backpropagation) to achieve data parallelism, and some form of model parallelism. The latter is achieved by carefully partitioning the neural net into pieces and allocating them to machines to compute gradients. This is difficult to do and requires a careful match of the neural net structure (number of layers and hidden units, connectivity, etc.) to the target hardware. Also, parallel SGD can diverge with nonconvex models, which requires heuristics to make sure we average replica models that are close in parameter space and thus associated with the same optimum. Although this has managed to train huge nets on huge datasets by using tens of thousands of CPU cores, the speedups achieved were very modest. Other work has used similar techniques but for GPUs (Coates et al., 2013; Seide et al., 2014). At present, TensorFlow does data parallelism automatically, but more complex forms of parallelism must be programmed by the user.

Finally, there also exist specific approximation techniques for certain types of large-scale machine learning problems, such as spectral problems, using the Nyström formula or other landmark-based methods (Williams & Seeger, 2001; Bengio et al., 2004; Drineas & Mahoney, 2005; Talwalkar et al., 2008; Vladymyrov & Carreira-Perpiñán, 2013; 2016).

ParMAC (and MAC) is specifically designed for nested models, which are typically nonconvex and include deep nets and many other models, some of which have nondifferentiable layers. As we describe below, ParMAC has the advantages of being simple and relatively independent of the target hardware, while achieving high speedups.

3 Optimising Nested Models Using Auxiliary Coordinates (MAC)

Many optimisation problems in machine learning involve mathematically "nested" functions of the form $\mathbf{F}(\mathbf{x};\mathbf{W}) = \mathbf{F}_{K+1}(\dots \mathbf{F}_2(\mathbf{F}_1(\mathbf{x};\mathbf{W}_1);\mathbf{W}_2)\dots;\mathbf{W}_{K+1})$ with parameters $\mathbf{W}$, such as deep nets. Such problems are traditionally optimised using methods based on gradients computed using the chain rule. However, such gradients may some-

times be inconvenient to use, or may not exist (e.g. if some of the layers are nondifferentiable, as with binary autoencoders). Also, they are hard to parallelise, because of the inherent sequentiality in the chain rule. The *method of auxiliary coordinates (MAC)* (Carreira-Perpiñán & Wang, 2012; 2014) is designed to optimise nested models without using chain-rule gradients while introducing parallelism. The idea is to break nested functional relationships judiciously by introducing new variables (the *auxiliary coordinates*) as equality constraints. These are then solved by optimising a penalised function using alternating optimisation over the original parameters (which we call the $\mathbf{W}$ step) and over the coordinates (which we call the $\mathbf{Z}$ step). The result is a *coordination-minimisation (CM) algorithm*: the minimisation ($\mathbf{W}$) step updates the parameters by splitting the nested model into independent submodels and training them using existing algorithms, and the coordination ($\mathbf{Z}$) step ensures that corresponding inputs and outputs of submodels eventually match. MAC algorithms have been developed for several nested models so far: deep nets (Carreira-Perpiñán & Wang, 2014), low-dimensional SVMs (Wang & Carreira-Perpiñán, 2014), binary autoencoders (Carreira-Perpiñán & Raziperchikolaei, 2015), affinity-based loss functions for binary hashing (Raziperchikolaei & Carreira-Perpiñán, 2016) and parametric nonlinear embeddings (Carreira-Perpiñán & Vladymyrov, 2015). Although this paper proposes and analyses ParMAC in general, our MPI implementation is for the particular case of binary autoencoders. These define a nonconvex nondifferentiable problem, yet its MAC algorithm is simple and effective. We briefly describe it next.

3.1 MAC Algorithm for Binary Autoencoders

A *binary autoencoder (BA)* is a usual autoencoder but with a binary code layer. It consists of an encoder $\mathbf{h}(\mathbf{x})$ that maps a real vector $\mathbf{x} \in \mathbb{R}^D$ onto a *binary* code vector with $L < D$ bits, $\mathbf{z} \in \{0,1\}^L$, and a linear decoder $\mathbf{f}(\mathbf{z})$ which maps $\mathbf{z}$ back to $\mathbb{R}^D$ in an effort to reconstruct $\mathbf{x}$. We will call $\mathbf{h}$ a *binary hash function* (see later). Let us write $\mathbf{h}(\mathbf{x}) = \curlywedge(\mathbf{A}\mathbf{x})$ ($\mathbf{A}$ includes a bias by having an extra dimension $x_0 = 1$ for each $\mathbf{x}$) where $\mathbf{A} \in \mathbb{R}^{L \times (D+1)}$ and $\curlywedge(t)$ is a step function applied elementwise, i.e., $\curlywedge(t) = 1$ if $t \geq 0$ and $\curlywedge(t) = 0$ otherwise. Given a dataset of D-dimensional patterns $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, our objective function, which involves the nested model $\mathbf{y} = \mathbf{f}(\mathbf{h}(\mathbf{x}))$, is the usual least-squares reconstruction error:

$$E_{\text{BA}}(\mathbf{h}, \mathbf{f}) = \sum_{n=1}^{N} \|\mathbf{x}_n - \mathbf{f}(\mathbf{h}(\mathbf{x}_n))\|^2. \tag{1}$$

Optimising this nonconvex, nonsmooth function is NP-hard (Carreira-Perpiñán & Raziperchikolaei, 2015). Where the gradients do exist wrt $\mathbf{A}$ they are zero, so optimisation of $\mathbf{h}$ using chain-rule gradients does not apply. We introduce as auxiliary coordinates the outputs of $\mathbf{h}$, i.e., the codes for each of the N input patterns, and obtain the following equality-constrained problem:

$$\min_{\mathbf{h},\mathbf{f},\mathbf{Z}} \sum_{n=1}^{N} \|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 \text{ s.t. } \begin{cases} \mathbf{z}_n = \mathbf{h}(\mathbf{x}_n) \in \{0,1\}^L, \\ n = 1, \dots, N. \end{cases} \tag{2}$$

Note the codes are binary. We now apply a penalty method to bring the equality constraints into the objective function. The best method generally uses the augmented Lagrangian (Nocedal & Wright, 2006; Carreira-Perpiñán & Wang, 2012; 2014), but for simplicity of notation we describe here the quadratic penalty, which is identical except it lacks the Lagrange multiplier parameters. This minimises the following objective while progressively increasing μ, so the constraints are eventually satisfied:

$$E_Q(\mathbf{h}, \mathbf{f}, \mathbf{Z}; \mu) = \sum_{n=1}^{N} \left(\|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2 \right)$$

s.t. $\mathbf{z}_n \in \{0,1\}^L$, $n = 1, \dots, N$. Finally, we apply alternating optimisation over $\mathbf{Z}$ and $\mathbf{W} = (\mathbf{h}, \mathbf{f})$. This gives the following steps:

Coordinates Over $\mathbf{Z}$ for fixed $(\mathbf{h}, \mathbf{f})$, this is a binary optimisation on NL variables, but it separates into N independent optimisations each on only L variables, with the form of a binary proximal operator (where we omit the index n): $\min_{\mathbf{z}} \|\mathbf{x} - \mathbf{f}(\mathbf{z})\|^2 + \mu \|\mathbf{z} - \mathbf{h}(\mathbf{x})\|^2$ s.t. $\mathbf{z} \in \{0,1\}^L$. This can be solved approximately by alternating optimisation over bits.

Submodels Over $\mathbf{W} = (\mathbf{h}, \mathbf{f})$ for fixed $\mathbf{Z}$, we obtain $L + D$ independent problems: for each of the L single-bit hash functions (which try to predict $\mathbf{Z}$ optimally from $\mathbf{X}$), each solvable by fitting a linear SVM; and for each of the D linear decoders in $\mathbf{f}$ (which try to reconstruct $\mathbf{X}$ optimally from $\mathbf{Z}$), each a linear least-squares problem.

The user must choose a schedule for the penalty parameter μ (sequence of values $0 < \mu_1 < \cdots < \mu_\infty$). This should increase slowly enough that the binary codes can change considerably and explore better solutions before the constraints are satisfied and the algorithm stops. With BAs, MAC stops for a finite value of μ, which occurs whenever $\mathbf{Z}$ does not change compared to the previous $\mathbf{Z}$ step. This gives a practical stopping criterion. Carreira-Perpiñán & Raziperchikolaei (2015) give proofs of these statements and further details about the algorithm. Fig. 1 gives the MAC algorithm for BAs.

The BA was proposed as a way to learn good binary hash functions for fast, approximate information retrieval (Carreira-Perpiñán & Raziperchikolaei, 2015). Binary hashing (Grauman & Fergus, 2013) has emerged in recent years as an effective way to do fast, approximate nearest-neighbour searches in image databases. The real-valued,

high-dimensional image vectors are mapped onto a binary space with L bits and the search is performed there using Hamming distances at a vastly faster speed and smaller memory (e.g. $N = 10^9$ points with $D = 500$ take 2 TB, but only 8 GB using $L = 64$ bits, which easily fits in RAM). As shown by Carreira-Perpiñán & Raziperchikolaei (2015), training BAs with MAC beats approximate optimisation approaches such as relaxing the codes or the step function in the encoder, and yields state-of-the-art binary hash functions $\mathbf{h}$ in unsupervised problems, improving over established approaches such as iterative quantisation (ITQ) (Gong et al., 2013). We focus mostly on linear hash functions because these are, by far, the most used type of hash functions in the literature of binary hashing, due to the fact that computing the binary codes for a test image must be fast at run time.

3.2 MAC in General

With a nested function with K layers, we can introduce auxiliary coordinates at each layer. For example, with a neural net, this decouples the weight vector of every hidden unit in the $\mathbf{W}$ step, which can be solved as a logistic regression (see Carreira-Perpiñán & Alizadeh, 2016). For a large net with a large dataset, this affords an enormous potential for parallel computation.

3.3 MAC and EM

MAC is very similar to expectation-maximisation (EM) at a conceptual level. We briefly explain this here; see Carreira-Perpiñán (2019b) for more details. EM (McLachlan & Krishnan, 2008) applies generally to many probabilistic models. The resulting algorithm can be very different (e.g. EM for Gaussian mixtures vs EM for hidden Markov models), but it always alternates two steps that conceptually do the following. The E step updates in parallel the posterior probabilities. This separates over data points and is like the $\mathbf{Z}$ step in MAC, where the posterior probabilities are the auxiliary coordinates, and where the step may be in closed form or require optimisation, depending on the model. The M step updates in parallel the "submodels". For a mixture with M components, these are the M Gaussians (means, covariances, proportions). This separates over submodels and is like the $\mathbf{W}$ step in MAC. For BAs, the submodels are the L encoders (linear SVMs) and the D decoders (linear regressors); for a neural net, each weight vector of a hidden unit is a submodel (a logistic regressor). For Gaussian mixtures, the M step can be done exactly in one "epoch" because it is a simple average. For MAC, it usually requires optimisation, and so multiple epochs. In fact, ParMAC applies to EM by using $e = 1$ epoch: in the $\mathbf{W}$ step, the Gaussians visit each machine circularly and (their averages) are updated on its data; in the $\mathbf{Z}$ step, each machine updates its posterior probabilities.

In the rest of the paper, some readers may find this analogy useful and think of EM for Gaussian mixtures instead of MAC, replacing "submodels" and "auxiliary coordinates" in MAC with "Gaussians" and "posterior probabilities" in EM, respectively.

4 ParMAC: A Parallel, Distributed Computation Model for MAC

A specific MAC algorithm depends on the model and objective function and on how the auxiliary coordinates are introduced. We can achieve steps that are closed-form, convex, nonconvex, binary, or others. However, we will assume the following always hold: (1) **Separability over data points**. In the $\mathbf{Z}$ step, *the N subproblems for $\mathbf{z}_1, \dots, \mathbf{z}_N$ are independent, one per data point*. Each $\mathbf{z}_n$ step depends on the current model. (2) **Separability over submodels**. In the $\mathbf{W}$ step, there are M *independent submodels*, where M depends on the problem. For example, M is the number of hidden units in a deep net, or the number of hash functions and linear decoders in a BA. Each submodel depends on all the data and coordinates. We now show how to turn this into a distributed, low-communication *ParMAC* algorithm.

The basic idea in ParMAC is as follows. With large datasets in distributed systems, it is imperative to minimise data movement over the network because the communication time generally far exceeds the computation time in modern architectures. In MAC we have 3 types of data: the original training data of inputs $\mathbf{X}$ and outputs $\mathbf{Y}$, the auxiliary coordinates $\mathbf{Z}$, and the model parameters (the submodels). Usually, the latter type is far smaller. *In ParMAC, we never communicate training or coordinate data; each machine keeps a disjoint portion of $(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ corresponding to a subset of the points. Only model parameters are communicated, during the $\mathbf{W}$ step, following a circular topology*[1]*, which implicitly implements a stochastic optimisation*. The model parameters are the hash functions $\mathbf{h}$ and the decoder $\mathbf{f}$ for BAs, and the weight vector $\mathbf{w}_h$ of each hidden unit h for deep nets. Let us see this in detail (refer to fig. 2).

Assume we have P identical processing machines, each with its own memory and CPU, connected through a network in a circular unidirectional topology. Each machine stores a subset of the data points and corresponding coordinates $(\mathbf{x}_n, \mathbf{y}_n, \mathbf{z}_n)$ such that the subsets are disjoint and their union is the entire data. Before the $\mathbf{Z}$ step starts, each machine contains all the (just updated) submodels. This means that in the $\mathbf{Z}$ step each machine processes its auxiliary coordinates $\{\mathbf{z}_n\}$ independently of all other machines, i.e., no communication occurs. The $\mathbf{W}$ step is more subtle. At the beginning of the $\mathbf{W}$ step, each machine will

[1] We discuss other topologies in section 4.4.

input $\mathbf{X}_{D\times N} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, $L \in \mathbb{N}$
Initialise $\mathbf{Z}_{L\times N} = (\mathbf{z}_1, \dots, \mathbf{z}_N) \in \{0,1\}^{LN}$
for $\mu = 0 < \mu_1 < \cdots < \mu_\infty$
  parfor $l = 1, \dots, L$ — **W** step: **h**
    $h_l(\cdot) \leftarrow$ fit SVM to $(\mathbf{X}, \mathbf{Z}_{l\cdot})$
  parfor $d = 1, \dots, D$ — **W** step: **f**
    $f_d(\cdot) \leftarrow$ least-squares fit to $(\mathbf{Z}, \mathbf{X}_{d\cdot})$
  parfor $n = 1, \dots, N$ — **Z** step
    $\mathbf{z}_n \leftarrow \arg\min_{\mathbf{z}_n \in \{0,1\}^L} \|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2$
  if no change in $\mathbf{Z}$ and $\mathbf{Z} = \mathbf{h}(\mathbf{X})$ **then** stop
return $\mathbf{h}$, $\mathbf{Z} = \mathbf{h}(\mathbf{X})$

$\mathbf{X}$ training points

$\mathbf{Z}$ auxiliary coordinates

$\mathbf{h}$: $\mathbb{R}^D \to \{0,1\}^L$,
$\mathbf{h} = (h_1, \dots, h_L)$
encoders (hash function)

$\mathbf{f}$: $\mathbb{R}^L \to \mathbb{R}^D$,
$\mathbf{f} = (f_1, \dots, f_D)$
decoders

Figure 1. MAC algorithm for binary autoencoders. "**parfor**" indicates a **for** loop whose iterations are carried out in parallel. The steps over $\mathbf{h}$ and $\mathbf{f}$ can be run in parallel as well.

contain all the submodels and its portion of the data and (just updated) coordinates. Each submodel must have access to the entire data and coordinates in order to update itself and, since the data cannot leave its home machine, the submodel must go to the data. We achieve this in the circular topology with an asynchronous processing, as follows. Each machine keeps a queue of submodels to be processed, and repeatedly performs the following operations: extract a submodel from the queue, process it on its data and send it to the machine's successor (which will insert it in its queue). If the queue is empty, the machine waits until it is nonempty. The queue of each machine is initialised with a portion M/P of submodels associated with that machine (e.g. in fig. 2, machine 1's queue contains submodels 1–3, machine 2 submodels 4–6, etc.). Each submodel carries a counter that is initially 1 and increases every time it visits a machine. When it reaches P, the submodel has visited all machines in sequence and has completed an *epoch*. We repeat this for e epochs and, to ensure all machines have all final submodels before starting the $\mathbf{Z}$ step, we run a communication-only epoch $e+1$ (without computation), where submodels simply move from machine to machine.

Since each submodel is updated as soon as it visits a machine, rather than computing the exact gradient once it has visited all machines and then take a step, the $\mathbf{W}$ step is really carrying out *stochastic steps for each submodel*. For example, if the update is done by a gradient step, we are actually implementing stochastic gradient descent (SGD) where the minibatches are of size N/P (or smaller, if we subdivide a machine's data portion into minibatches, which should be typically the case in practice). From this point of view, we can regard the $\mathbf{W}$ step as doing SGD on each submodel in parallel by having each submodel visit the minibatches in each machine.

As described, and as implemented in our experiments, the entire model parameters are communicated $e+1$ times in a MAC iteration if running e epochs in the $\mathbf{W}$ step. We can also run e epochs with only 2 rounds of communication *by having a submodel do e consecutive passes within each machine's data*. This reduces the amount of shuffling, but should not be a problem if the data are randomly distributed over machines.

4.1 Extensions of ParMAC

Data shuffling, which improves the SGD convergence speed, can be achieved without data movement by accessing the local data in random order at each epoch (within-machine), and by randomising the circular topology at each epoch (across-machine). *Load balancing* is simple because the work in both $\mathbf{W}$ and $\mathbf{Z}$ steps is proportional to the number of data points N. Hence, if the processing power of machine p is proportional to $\alpha_p > 0$, we allocate to it $N\alpha_p/(\alpha_1 + \cdots + \alpha_P)$ data points. *Streaming*, i.e., discarding old data and adding new data during training, can be done by adding/removing data within-machine, or by adding/removing machines and updating the circular topology. *Fault tolerance* is possible because we can still learn a good model even if we lose the data from a machine that fails, and because in the $\mathbf{W}$ step we can revert to older copies of the lost submodels residing in other machines. See further details in Carreira-Perpiñán & Alizadeh (2016).

4.2 A Theoretical Model of the Parallel Speedup

We can estimate the runtime of the $\mathbf{W}$ and $\mathbf{Z}$ steps assuming there are M independent submodels of the same size in the $\mathbf{W}$ step, using e epochs, on a dataset with N training points, distributed over P identical machines (each with N/P points). Let $t_r^{\mathbf{W}}$ be the computation time per submodel and data point in the $\mathbf{W}$ step, $t_r^{\mathbf{Z}}$ the computation time per data point in the $\mathbf{Z}$ step, and $t_c^{\mathbf{W}}$ the communication time per submodel in the $\mathbf{W}$ step. Then the runtime of the $\mathbf{W}$ and $\mathbf{Z}$ steps is $T^{\mathbf{W}}(P) = \lceil M/P \rceil (t_r^{\mathbf{W}} \frac{N}{P} + t_c^{\mathbf{W}}) Pe + \lceil M/P \rceil t_c^{\mathbf{W}} P$ and $T^{\mathbf{Z}}(P) = M\frac{N}{P} t_r^{\mathbf{Z}}$, respectively, and the total runtime per iteration is $T(P) = T^{\mathbf{W}}(P) + T^{\mathbf{Z}}(P)$. Hence the parallel speedup is

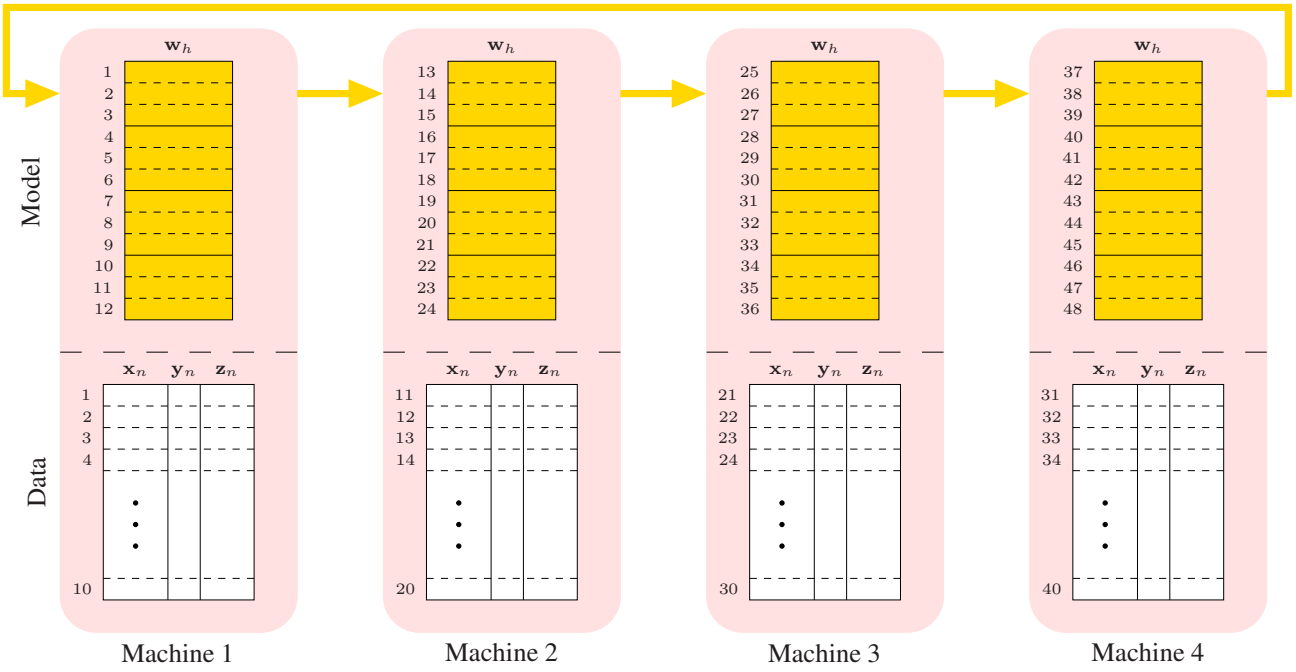

Figure 2. ParMAC model with $P = 4$ machines, $M = 12$ submodels "$\mathbf{w}_h$" and $N = 40$ data points. Submodels h, $h + M$, $h + 2M$ and $h + 3M$ are copies of submodel h, but only one of them is the most currently updated. At the end of the $\mathbf{W}$ step all copies are identical.

(see details in Carreira-Perpiñán, 2019a):

$$S(P) = \frac{T(1)}{T(P)} = \frac{\rho \frac{1}{\lceil M/P \rceil} MP}{\frac{1}{N}P^2 + \rho_2 P + \rho_1 \frac{1}{\lceil M/P \rceil} M} \quad (3)$$

$$\rho_1 = t_r^{\mathbf{Z}}/(e+1)t_c^{\mathbf{W}}, \ \rho_2 = et_r^{\mathbf{W}}/(e+1)t_c^{\mathbf{W}} \quad (4)$$

$$\rho = \rho_1 + \rho_2 = (et_r^{\mathbf{W}} + t_r^{\mathbf{Z}})/(e+1)t_c^{\mathbf{W}} \quad (5)$$

where ρ, ρ_1 and ρ_2 are ratios of computation vs communication, dependent on the optimisation algorithm in the $\mathbf{W}$ and $\mathbf{Z}$ steps, and on the performace of the distributed system and libraries (MPI in our implementation).

Hence, if $P \leq M$ and M is divisible by P we have $S(P) = P/(1 + \frac{P}{\rho N})$ and if $P > M$ we have $S(P) = \rho M/(\rho_2 + \rho_1 \frac{M}{P} + \frac{P}{N})$. In practice, typically we have $\rho \ll 1$ (because communication dominates computation in current architectures) and $\rho_2 N \gg 1$ (large dataset). If we take $P \ll \rho_2 N$, then $S(P) \approx P$ if $P \leq M$ and $S(P) \approx \rho M/(\rho_2 + \rho_1 \frac{M}{P})$ if $P > M$. Hence, *the speedup is nearly perfect if using fewer machines than submodels, and otherwise it peaks at* $S_1^* = \rho M/(\rho_2 + 2\sqrt{\rho_1 M/N}) > M$ *for* $P = P_1^* = \sqrt{\rho_1 MN} > M$ *and decreases thereafter.* This affords very large speedups for large datasets and large models. This theoretical speedup matches well our measured ones (see the experiments section), and can be used to determine optimal values for the number of machines P to use in practice (subject to additional constraints, e.g. cost of the machines).

Eq. (3) also shows that we can leave the speedup unchanged by trading off dataset size and computation/communication times, as long as one of these holds: $Nt_r^{\mathbf{W}}$ and $Nt_r^{\mathbf{Z}}$ remain constant; or $N/t_c^{\mathbf{W}}$ remains constant; or $t_r^{\mathbf{W}}/t_c^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}/t_c^{\mathbf{W}}$ remain constant.

In the BA, we have submodels of different size: encoders of size D and decoders of size $L < D$. We can model this by "grouping" the D decoders into L groups of D/L decoders each, resulting in $M = 2L$ equal-size submodels (assuming the ratio of computation and communication times of decoder vs encoder is $L/D < 1$).

4.3 Convergence of ParMAC

The only approximation that ParMAC makes to the original MAC algorithm is using SGD in the $\mathbf{W}$ step. Since we can guarantee convergence of SGD under certain conditions (e.g. Robbins-Monro schedules), we can recover the original convergence guarantees for MAC to a local stationary point with differentiable layers (see details in Carreira-Perpiñán & Alizadeh, 2016). This convergence guarantee is independent of the number of models and processors. With nondifferentiable layers, the convergence properties of MAC (and ParMAC) are not well known. In particular, for the binary autoencoder the encoding layer is discrete and the problem is NP-hard. While convergence guarantees are important theoretically, in practical applications with large datasets in a distributed setting one typically runs

SGD for just a few epochs, even one or less than one (i.e., we stop SGD before passing through all the data). This typically reduces the objective function to a good enough value as fast as possible, since each pass over the data is very costly. In our experiments, 1–2 epochs in the $\mathbf{W}$ step make ParMAC very similar to MAC using an exact step.

4.4 Circular vs Parameter-Server Topologies

We also considered implementing ParMAC (in the $\mathbf{W}$ step) using a parameter-server (PS) topology rather than a circular one, but the latter is better. To see this, focus on how a single submodel $m \in \{1, \dots, M\}$ is processed (since different submodels are processed independently and in parallel in either topology). With a PS we do parallel SGD on m, i.e., each worker runs SGD on its own replica of m for a while, sends it to the PS, and this broadcasts an "average" m back to the workers, asynchronously. The circular topology does true SGD directly on m, with no replicas. We can show (Carreira-Perpiñán, 2019a) the runtime per iteration using a PS is equal to that of the circular topology only if the server can communicate with P workers simultaneously (rather than sequentially), otherwise it is slower. The reason is the PS has more communication. Also importantly, parallel SGD converges more slowly than true SGD and is difficult to apply if the $\mathbf{W}$ step is nonconvex. Finally, the PS needs extra machine(s) to act as parameter server(s). Considering now all M submodels, the fundamental difference between both topologies is in how they employ the available parallelism: the circular topology updates submodels directly and communicates them, while the PS updates replicas (of each submodel), communicates them and averages them.

It may be possible to use other topologies that do true SGD on the submodels but we did not explore them.

5 Experiments[2]

MPI implementation of ParMAC for BAs. We have used C/C++, the GSL and BLAS libraries for mathematical operations, and the Message Passing Interface (MPI) (Gropp et al., 1999) for interprocess communication. MPI is a widely used framework for high-performance parallel computing, available in multiple platforms. It is particularly suitable for ParMAC because of its support of the SPMD (single program, multiple data) model. In MPI, processes in different machines communicate through messages. To receive data, we use the synchronous blocking receive function `MPI_Recv`; the process calling this blocks until the data arrives. To send data we use the buffered blocking send function `MPI_Bsend`. We allocate enough memory and attach it to the system. The process calling `MPI_Bsend` blocks until the buffer is copied to the MPI internal memory; after that, the MPI library takes care of sending the data.

The code snippet in figure 3 shows the main steps of the ParMAC algorithm for the BA. All the functions starting with `MPI_` are API calls from the MPI library. As with all MPI programs, we start the code by initialising the MPI environment and end by finalising it. To receive data we use the synchronous[3], blocking MPI receive function `MPI_Recv`. The process calling this blocks until the data arrives. To send data we use the buffered blocking version of the MPI send functions, `MPI_Bsend`. This requires that we allocate enough memory and attach it to the system in advance. The process calling `MPI_Bsend` blocks until the buffer is copied to the MPI internal memory; after that, the MPI library takes care of sending the data appropriately. The benefit of using this version of send is that the programmer can send messages without worrying about where they are buffered, so the code is simpler.

Distributed-memory cluster We used General Computing Nodes from the UCSD Triton Shared Computing Cluster (TSCC), available to the public for a fee. Each node contains 2 8-core Intel Xeon E5-2670 processors (16 cores in total), 64GB RAM (4GB/processor) and a 500GB hard drive. The nodes are connected through a 10GbE network. We used up to $P = 128$ processors. Carreira-Perpiñán & Alizadeh (2016) give detailed specs as well as experiments in a shared-memory machine.

Datasets We have used 3 well-known colour image retrieval benchmarks. (1) CIFAR (Krizhevsky, 2009) contains 60 000 images ($N = 50\,000$ training and 10 000 test), represented by $D = 320$ GIST features. (2) SIFT-1M (Jégou

[2]Our implementation is available in the authors' webpage.

[3]Note that the word "synchronous" here does not refer to how we process the different submodels, which as we stated earlier are not synchronised to start or end at specific clock ticks, hence are processed asynchronously with respect to each other. The word "synchronous" here refers to MPI's handling of an *individual* receive function. This can be done either by calling `MPI_Recv`, which will block until the data is received (synchronous blocking function), as in the pseudocode in fig. 3; or by calling `MPI_Irecv` (asynchronous nonblocking function) followed by a `MPI_Wait`, which will block until the data is received, like this:

```
MPI_Irecv(receivebuffer, commbuffsize,
  MPI_CHAR, MPI_ANY_SOURCE, MODEL_MSG_TAG,
  MPI_COMM_WORLD, &recvRequest);
MPI_Wait(&recvRequest, &recvStatus);
```

Both options are equivalent for our purpose, which is to ensure we receive the submodel before starting to train it. The `MPI_Irecv`/`MPI_Wait` option is slightly more flexible in that it would allow us to do some additional processing between `MPI_IRecv` and `MPI_Wait` and possibly achieve some performance gain.

```
MPI_Init(&argc, &argv);                    // initialise MPI execution environment
MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
loadsettings();                            // μ, epochs, dataset path, etc.
loaddatasets();                            // datasets and initial auxiliary coordinates
initializelayers();                        // initialise f, h and Z steps
// allocate big enough buffer for MPI_Bsend
MPI_Pack_size(commbuffsize, MPI_CHAR, MPI_COMM_WORLD,
 &mpi_attach_buff_size);
mpi_attach_buff = malloc(totalsubmodelcount*
 (mpi_attach_buff_size+MPI_BSEND_OVERHEAD));
MPI_Buffer_attach(mpi_attach_buff, mpi_attach_buff_size);

for (iter=1 to length(μ)) {
 // begin W-step
 visitedsubmodels = 0;
 // each process visits all the submodels, epochs + 1 times
 while (visitedsubmodels <= totalsubmodelcount*epochs) {
  // stepcounter indicates how far trained each submodel is
  if (stepcounter > 0) {                   // not 1st submodel? wait to receive
   // MPI_Recv blocks until requested data is available
   MPI_Recv(receivebuffer, commbuffsize,
    MPI_CHAR, MPI_ANY_SOURCE, MODEL_MSG_TAG,
                MPI_COMM_WORLD, &recvStatus);
   savesubmodel(receivebuffer);
  }
  if (stepcounter < epochs*mpisize) {      // not in last round
   switch(submodeltype)                    // train submodel according to type
   case 'SVM': HtrainSGD();
   case 'linlayer': FtrainSGD();
  }
  if (stepcounter < (ringepochs+1)*mpisize) {
   // pick the successor process from the lookup table
   successor = next_in_lookuptable();
   loadsubmodel(sendbuffer);
   MPI_Bsend(sendbuffer, taskbufsize*sizeof(double),
    MPI_CHAR, successor, MODEL_MSG_TAG, MPI_COMM_WORLD);
  }
  visitedsubmodels++;
 }
 // end W-step

 // begin Z-step
 updateZ();                                // optimise auxiliary coordinates
 // end Z-step
}

MPI_Buffer_detach(&mpi_attach_buff, &mpi_attach_buff_size);
free(mpi_attach_buff);                     // free the allocated memory
MPI_Finalize();                            // terminate MPI execution environment
```

Figure 3. Binary autoencoder ParMAC algorithm (fragment), showing important MPI calls.

et al., 2011a) contains $N = 10^6$ training and 10^4 test images, each represented by $D = 128$ SIFT features. (3) SIFT-1B (Jégou et al., 2011a) has three subsets: 10^9 base vectors where the search is performed, $N = 10^8$ learning vectors used to train the model and 10^4 query vectors.

Performance measures Regarding the quality of the BA and hash functions learnt, we report the retrieval precision (%) in the test set using as true neighbours the K nearest images in Euclidean distance in the original space, and as retrieved neighbours in the binary space we use the k nearest images in Hamming distance. We set $(K, k) =$ (1 000,100) for CIFAR and (10 000,10 000) for SIFT-1M. For SIFT-1B, as suggested by the dataset creators, we report the recall@R: the average number of queries for which the nearest neighbour is ranked within the top R positions (for varying values of R); in case of tied distances, we place the query as top rank. All these measures are computed offline once the BA is trained. Carreira-Perpiñán & Alizadeh (2016) give additional measures and experiments.

Models and their parameters We use BAs with linear encoders (linear SVM) except with SIFT-1B, where we also use kernel SVMs. The decoder is always linear. We set $L = 16$ bits (hash functions) for CIFAR and SIFT-1M and $L = 64$ bits for SIFT-1B. We initialise the binary codes from truncated PCA ran on a subset of the training set (small enough that it fits in one processor). To train the encoder (L SVMs) and decoder (D linear mappings) with stochastic optimisation, we used the SGD code from (Bottou & Bousquet, 2008), using its default parameter settings. The SGD step size is tuned automatically in each iteration by examining the first 1 000 datapoints. We use a multiplicative μ schedule $\mu_i = \mu_0 a^i$ where the initial value μ_0 and the factor $a > 1$ are tuned offline in a trial run using a small subset of the data. For CIFAR we use $\mu_0 = 0.005$ and $a = 1.2$ over 26 iterations ($i = 0, \dots, 25$). For SIFT-1M and SIFT-1B we use $\mu_0 = 10^{-4}$ and $a = 2$ over 10 iterations.

5.1 Effect of Stochastic Steps in the W Step

Fig. 4 shows the effect on the precision on CIFAR of varying the number of epochs within the **W** step and shuffling the data as a function of the number of processors P. As the number of epochs increases, the **W** step is solved more exactly (8 epochs is practically exact in this data). Fewer epochs, even just one, cause only a small degradation. The reason is that, although these are relatively small datasets, they contain sufficient redundance that few epochs are sufficient to decrease the error considerably. This is also helped by the accumulated effect of epochs over MAC iterations. Running more epochs increases the runtime and lowers the parallel speedup in this particular model, because we use few bits ($L = 16$) and therefore few submodels ($M = 2L = 32$) compared to the number of machines (up to $P = 128$), so the **W** step has less parallelism. The positive effect of data shuffling in the **W** step is clear: shuffling generally increases the precision with no increase in runtime.

5.2 Speedup

The fundamental advantage of ParMAC and distributed optimisation in general is the ability to train on datasets that do not fit in a single machine, and the reduction in runtime because of parallel processing. Fig. 5 shows the "strong scaling" speedups achieved, as a function of the number of machines P for fixed problem size (dataset and model), in CIFAR and SIFT-1M ($N =$ 50K and 1M training points, respectively). Even though these datasets and especially

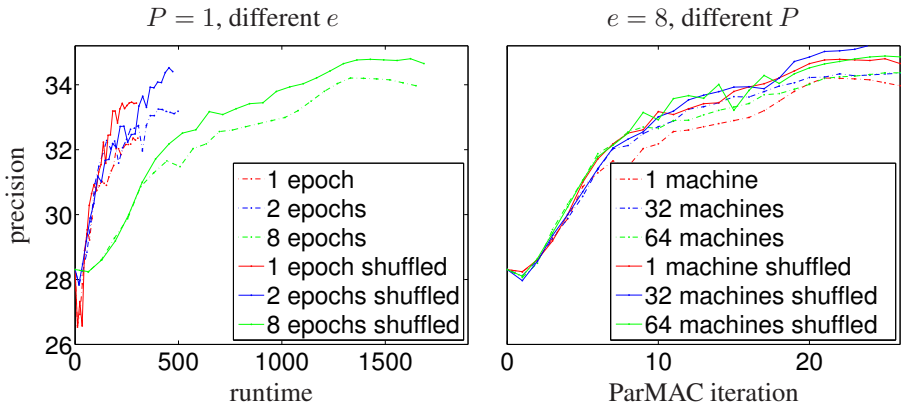

Figure 4. Precision in CIFAR dataset.

the number of independent submodels ($M = 2L = 32$ effective submodels of the same size, as discussed earlier) are relatively small, the speedups we achieve are nearly perfect for $P \leq M$ and hold very well for $P > M$ up to the maximum number of machines we used ($P = 128$ in the distributed system). The speedups flatten as the number of $\mathbf{W}$-step epochs (and consequently the amount of communication) increases, because for this experiment the bottleneck is the $\mathbf{W}$ step, whose parallelisation ability (i.e., the number of concurrent processes) is limited by $M = 2L$ (the $\mathbf{Z}$ step has N independent processes and is never a bottleneck, since N is very large). However, as noted earlier, using 1 to 2 epochs gives a good enough result, very close to doing an exact $\mathbf{W}$ step. The runtime for SIFT-1M on $P = 128$ machines with 1 epoch was 12 minutes and its speedup 100×. This is particularly remarkable given that the original, nested model did not have model parallelism.

Fig. 5 also shows the speedups predicted by our theoretical model. We set the parameters e and N to their known values, and $M = 2L = 32$ for CIFAR and SIFT-1M and $M = 2L = 128$ for SIFT-1B. For the time parameters, we set $t_r^{\mathbf{W}} = 1$ to fix the time units, and we set $t_c^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}$ by trial and error to achieve a reasonably good fit to the experimental speedups: $t_c^{\mathbf{W}} = 10^4$ for both datasets, and $t_r^{\mathbf{Z}} = 200$ for CIFAR and 40 for SIFT-1M. Although these are fudge factors, they are in rough agreement with the fact that communicating a weight vector over the network is orders of magnitude slower than updating it with a gradient step, and that for BAs the $\mathbf{Z}$ step is quite slower than the $\mathbf{W}$ step because of the binary optimisation it involves.

5.3 Large-Scale Experiment

SIFT-1B is one of the largest datasets, if not the largest one, that are publicly available for comparing nearest-neighbour search algorithms with known ground-truth (i.e., precomputed exact Euclidean distances for each query to its k nearest vectors in the base set). The training set contains $N =$ 100M vectors, each consisting of 128 SIFT features. We used $L = 64$ hash functions ($M = 128$ submodels): linear SVMs as before, and kernel SVMs. These have fixed Gaussian radial basis functions (2 000 centres picked at random from the training set and bandwidth $\sigma = 160$), so the only trainable parameters are the weights, and the MAC algorithm does not change except that it operates on a 2 000-dimensional input vector of kernel values, instead of the 128 SIFT features. We use $e = 2$ epochs with shuffling. All these decisions were based on trials on a subset of the training dataset. We initialised the binary codes from truncated PCA trained on a subset of size 1M (recall@R=100: 55.2%), which gave results comparable to the baseline in (Jégou et al., 2011b).

We ran ParMAC on the whole training set in the distributed system with 128 processors for 6 iterations and achieved a recall@R=100 of 61.5% in 29 hours (linear SVM) and 66.1% in 83 hours (kernel SVM). Using a scaled-down model and training set, we estimated that training in one machine (with enough RAM to hold the data and parameters) would take months. The theoretical speedup (fig. 5 right plot, using the same parameters as in SIFT-1M), is nearly perfect (note the plot goes up to $P = 1\,024$ machines, even though our experiments are limited to $P = 128$). This is because M is quite larger and N is much larger than in the previous datasets.

6 Discussion

Developing parallel, distributed optimisation algorithms for nonconvex problems in machine learning is challenging, as shown by recent efforts by large teams of researchers. One important advantage of ParMAC is its simplicity. Data and model parallelism arise naturally thanks to the introduction of auxiliary coordinates. The corresponding optimisation subproblems can often be solved reusing existing code as a black box (as with the SGD training of SVMs and linear mappings in the BA). A circular topology is sufficient to achieve a low communication between machines. There

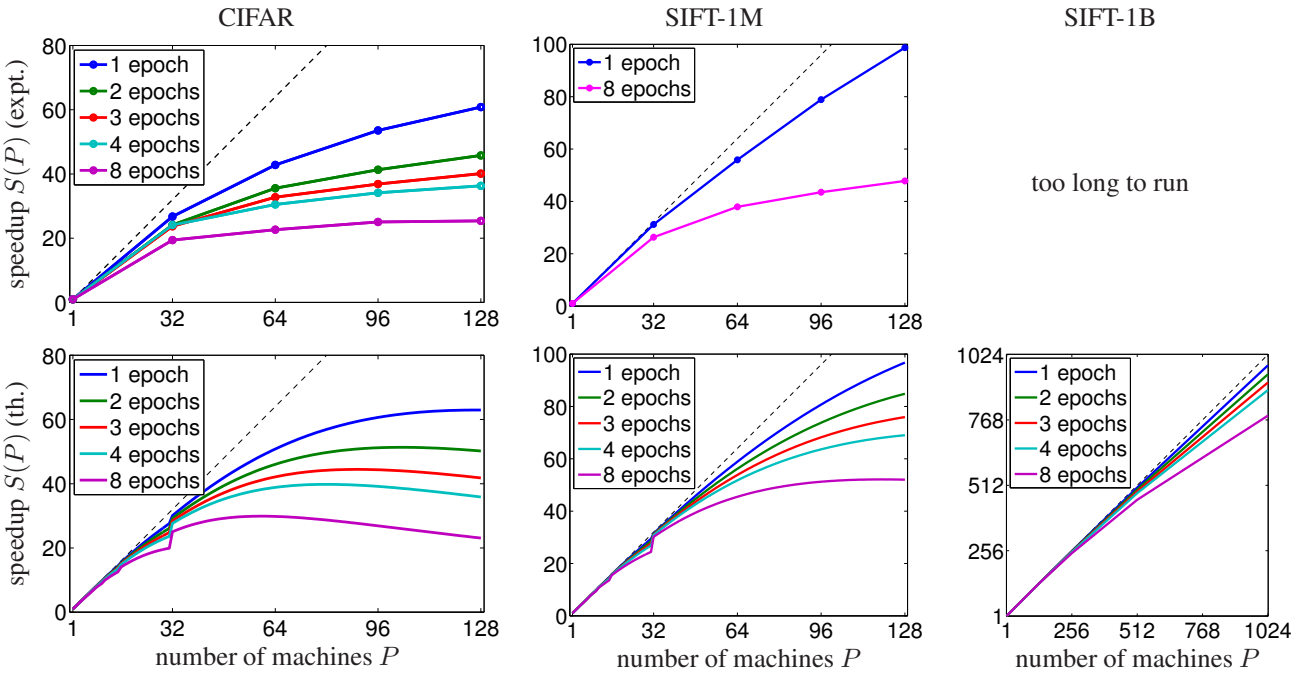

Figure 5. Speedup $S(P)$ as a function of the number of machines P (top: experiment, bottom: theory). The dataset size and number of submodels (N, M) is (50 000,32) for CIFAR, $(10^6, 32)$ for SIFT-1M and $(10^8, 128)$ for SIFT-1B.

is no close coupling between the model structure and the distributed system architecture. This makes ParMAC suitable for architectures as different as supercomputers, data centres or even IoT devices.

Further improvements can be made in specific problems. For example, we may have more parallelisation or less dependencies (e.g. the weights of hidden units in layer k of a neural net depend only on auxiliary coordinates in layers k and $k+1$). This may reduce the communication in the **W** step, by sending to a given machine only the model portion it needs, or by allocating cores within a multicore machine accordingly. The **W** and **Z** step optimisations can make use of further parallelisation by GPUs or by distributed convex optimisation algorithms. Many more refinements can be done, such as storing or communicating reduced-precision values with little effect of the accuracy. In this paper, we have kept our implementation as simple as possible, because our goal was to understand the parallelisation speedups of ParMAC in a setting as general as possible, rather than trying to achieve the very best performance for a particular dataset, model or distributed system.

7 Conclusion

We have proposed ParMAC, a distributed model for the method of auxiliary coordinates for training nested, non-convex models in general, analysed its parallel speedup and convergence, and demonstrated it with an MPI-based implementation for a particular case, to train binary autoencoders. MAC creates parallelism by introducing auxiliary coordinates for each data point to decouple nested terms in the objective function. ParMAC is able to translate the parallelism inherent in MAC into a distributed system by 1) using data parallelism, so that each machine keeps a portion of the original data and its corresponding auxiliary coordinates; and 2) using model parallelism, so that independent submodels visit every machine in a circular topology, effectively executing epochs of a stochastic optimisation, without the need for a parameter server and therefore no communication bottlenecks. The convergence properties of MAC remain essentially unaltered in ParMAC. The parallel speedup can be theoretically predicted to be nearly perfect when the number of submodels is comparable or larger than the number of machines, and to eventually saturate as one continues to increase the number of machines, and indeed this was confirmed in our experiments. ParMAC also makes it easy to account for data shuffling, load balancing, streaming and fault tolerance. Hence, we expect that ParMAC could be a basic building block, in combination with other techniques, for the distributed optimisation of nested models in big data settings.

Acknowledgments

Work supported by a Google Faculty Research Award and by NSF award IIS–1423515. We thank Dong Li (UC Merced) for discussions about MPI and performance evaluation on parallel systems and Quoc Le (Google) for discussions about Google's DistBelief system.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, Ł., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. TensorFlow WhitePaper.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. SETI@home: An experiment in public-resource computing. *Comm. ACM*, 45(11):56–61, November 2002.

Askari, A., Negiar, G., Sambharya, R., and Ghaoui, L. E. Lifted neural networks. arXiv:1805.01532, June 21 2018.

Bengio, Y., Paiement, J.-F., Vincent, P., Delalleau, O., Le Roux, N., and Ouimet, M. Out-of-sample extensions for LLE, Isomap, MDS, Eigenmaps, and spectral clustering. In Thrun, S., Saul, L. K., and Schölkopf, B. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 16. MIT Press, Cambridge, MA, 2004.

Bertsekas, D. P. Incremental gradient, subgradient, and proximal methods for convex optimization: A survey. In Sra, S., Nowozin, S., and Wright, S. J. (eds.), *Optimization for Machine Learning*. MIT Press, 2011.

Bottou, L. Large-scale machine learning with stochastic gradient descent. In *Proc. 19th Int. Conf. Computational Statistics (COMPSTAT 2010)*, pp. 177–186, Paris, France, August 22–27 2010.

Bottou, L. and Bousquet, O. The tradeoffs of large scale learning. In Platt, J. C., Koller, D., Singer, Y., and Roweis, S. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 20, pp. 161–168. MIT Press, Cambridge, MA, 2008.

Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

Bradley, J., Kyrola, A., Bickson, D., and Guestrin, C. Parallel coordinate descent for l_1-regularized loss minimization. In Getoor, L. and Scheffer, T. (eds.), *Proc. of the 28th Int. Conf. Machine Learning (ICML 2011)*, pp. 321–328, Bellevue, WA, June 28 – July 2 2011.

Carreira-Perpiñán, M. Á. Theoretical speedup of the ParMAC model for distributed optimisation of nested functions. arXiv, 2019a.

Carreira-Perpiñán, M. Á. The EM algorithm and the Method of Auxiliary Coordinates: Similarities and differences. arXiv, 2019b.

Carreira-Perpiñán, M. Á. and Alizadeh, M. ParMAC: Distributed optimisation of nested functions, with application to learning binary autoencoders. arXiv:1605.09114, May 30 2016.

Carreira-Perpiñán, M. Á. and Raziperchikolaei, R. Hashing with binary autoencoders. In *Proc. of the 2015 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'15)*, pp. 557–566, Boston, MA, June 7–12 2015.

Carreira-Perpiñán, M. Á. and Vladymyrov, M. A fast, universal algorithm to learn parametric nonlinear embeddings. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pp. 253–261. MIT Press, Cambridge, MA, 2015.

Carreira-Perpiñán, M. Á. and Wang, W. Distributed optimization of deeply nested systems. arXiv:1212.5921, December 24 2012.

Carreira-Perpiñán, M. Á. and Wang, W. Distributed optimization of deeply nested systems. In Kaski, S. and Corander, J. (eds.), *Proc. of the 17th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2014)*, pp. 10–19, Reykjavik, Iceland, April 22–25 2014.

Cevher, V., Becker, S., and Schmidt, M. Convex optimization for big data: Scalable, randomized, and parallel algorithms for big data analytics. *IEEE Signal Processing Magazine*, 31(5):32–43, September 2014.

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Ng, A. Deep learning with COTS HPC systems. In Dasgupta, S. and McAllester, D. (eds.), *Proc. of the 30th Int. Conf. Machine Learning (ICML 2013)*, pp. 1337–1345, Atlanta, GA, June 16–21 2013.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Large scale distributed deep networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 25, pp. 1232–1240. MIT Press, Cambridge, MA, 2012.

Drineas, P. and Mahoney, M. W. On the Nyström method for approximating a Gram matrix for improved kernel-based learning. *J. Machine Learning Research*, 6:2153–2175, December 2005.

Gong, Y., Lazebnik, S., Gordo, A., and Perronnin, F. Iterative quantization: A Procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 35 (12):2916–2929, December 2013.

Grauman, K. and Fergus, R. Learning binary hash codes for large-scale image search. In Cipolla, R., Battiato, S., and Farinella, G. (eds.), *Machine Learning for Computer Vision*, pp. 49–87. Springer-Verlag, 2013.

Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, second edition, 1999.

Hinton, G., Deng, L., Yu, D., Dahl, G., rahman Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, November 2012.

Jaderberg, M., Czarnecki, W. M., Osindero, S., Vinyals, O., Graves, A., Silver, D., and Kavukcuoglu, K. Decoupled neural interfaces using synthetic gradients. In Precup, D. and Teh, Y. W. (eds.), *Proc. of the 34th Int. Conf. Machine Learning (ICML 2017)*, pp. 1627–1635, Sydney, Australia, August 6–11 2017.

Jégou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 33(1):117–128, January 2011a.

Jégou, H., Tavenard, R., Douze, M., and Amsaleg, L. Searching in one billion vectors: Re-rank with source coding. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'11)*, pp. 861–864, Prague, Czech Republic, May 22–27 2011b.

Krizhevsky, A. Learning multiple layers of features from tiny images. Master's thesis, Dept. of Computer Science, University of Toronto, April 8 2009.

Le, Q., Ranzato, M., Monga, R., Devin, M., Corrado, G., Chen, K., Dean, J., and Ng, A. Building high-level features using large scale unsupervised learning. In Langford, J. and Pineau, J. (eds.), *Proc. of the 29th Int. Conf. Machine Learning (ICML 2012)*, Edinburgh, Scotland, June 26 – July 1 2012.

Lee, D.-H., Zhang, S., Fischer, A., and Bengio, Y. Difference target propagation. In Appice, A., Rodrigues, P. P., Costa, V. S., Soares, C., Gama, J., and Jorge, A. (eds.), *Proc. of the 26th European Conf. Machine Learning (ECML–15)*, pp. 498–515, Porto, Portugal, September 7–11 2015.

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

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endowment*, 5(8):716–727, April 2012.

McLachlan, G. J. and Krishnan, T. *The EM Algorithm and Extensions*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, second edition, 2008.

Niu, F., Recht, B., Ré, C., and Wright, S. J. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P., Pereira, F., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 24, pp. 693–701. MIT Press, Cambridge, MA, 2011.

Nocedal, J. and Wright, S. J. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, second edition, 2006.

Ororbia, A. G., Mali, A., Kifer, D., and Giles, C. L. Conducting credit assignment by aligning local representations. arXiv:1803.01834, July 12 2018.

Ouyang, H., He, N., Tran, L., and Gray, A. Stochastic alternating direction method of multipliers. In Dasgupta, S. and McAllester, D. (eds.), *Proc. of the 30th Int. Conf. Machine Learning (ICML 2013)*, pp. 80–88, Atlanta, GA, June 16–21 2013.

Raziperchikolaei, R. and Carreira-Perpiñán, M. Á. Optimizing affinity-based binary hashing using auxiliary coordinates. In Lee, D. D., Sugiyama, M., von Luxburg, U., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pp. 640–648. MIT Press, Cambridge, MA, 2016.

Richtárik, P. and Takáč, M. Distributed coordinate descent method for learning with big data. arXiv:1310.2059, October 8 2013.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *Proc. of Interspeech'14*, Singapore, September 14–18 2014.

Talwalkar, A., Kumar, S., and Rowley, H. Large-scale manifold learning. In *Proc. of the 2008 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'08)*, Anchorage, AK, June 23–28 2008.

Taylor, G., Burmeister, R., Xu, Z., Singh, B., Patel, A., and Goldstein, T. Training neural networks without gradients: A scalable ADMM approach. In Balcan, M.-F. and Weinberger, K. Q. (eds.), *Proc. of the 33rd Int. Conf. Machine Learning (ICML 2016)*, pp. 2722–2731, New York, NY, June 19–24 2016.

Vladymyrov, M. and Carreira-Perpiñán, M. Á. Locally Linear Landmarks for large-scale manifold learning. In Blockeel, H., Kersting, K., Nijssen, S., and Zelezný, F. (eds.), *Proc. of the 24th European Conf. Machine Learning (ECML–13)*, pp. 256–271, Prague, Czech Republic, September 23–27 2013.

Vladymyrov, M. and Carreira-Perpiñán, M. Á. The Variational Nyström method for large-scale spectral problems. In Balcan, M.-F. and Weinberger, K. Q. (eds.), *Proc. of the 33rd Int. Conf. Machine Learning (ICML 2016)*, pp. 211–220, New York, NY, June 19–24 2016.

Wang, W. and Carreira-Perpiñán, M. Á. The role of dimensionality reduction in classification. In Brodley, C. E. and Stone, P. (eds.), *Proc. of the 28th National Conference on Artificial Intelligence (AAAI 2014)*, pp. 2128–2134, Quebec City, Canada, July 27–31 2014.

Williams, C. K. I. and Seeger, M. Using the Nyström method to speed up kernel machines. In Leen, T. K., Dietterich, T. G., and Tresp, V. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 13, pp. 682–688. MIT Press, Cambridge, MA, 2001.

Wright, S. J. Coordinate descent algorithms. *Math. Prog.*, 151(1):3–34, June 2016.

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., and Yu, Y. Petuum: A new platform for distributed machine learning on big data. *IEEE Trans. Big Data*, 1(2):49–67, April–June 2015.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proc. 2nd USENIX Conf. Hot Topics in Cloud Computing (HotCloud 2010)*, 2010.

Zhang, R. and Kwok, J. Asynchronous distributed ADMM algorithm for global variable consensus optimization. In Xing, E. P. and Jebara, T. (eds.), *Proc. of the 31st Int. Conf. Machine Learning (ICML 2014)*, pp. 1701–1709, Beijing, China, June 21–26 2014.

Zinkevich, M., Weimer, M., Smola, A., and Li, L. Parallelized stochastic gradient descent. In Lafferty, J., Williams, C. K. I., Shawe-Taylor, J., Zemel, R., and Culotta, A. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 23, pp. 2595–2603. MIT Press, Cambridge, MA, 2010.