ParMAC: distributed optimisation of nested functions, with application to learning binary autoencoders

Miguel Á. Carreira-Perpiñán Mehdi Alizadeh
Electrical Engineering and Computer Science, University of California, Merced
http://eecs.ucmerced.edu

May 30, 2016

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)* (Carreira-Perpiñán and Wang, 2014). 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. With large-scale problems, or when distributing the computation is necessary for faster training, the dataset may not fit in a single machine. It is then essential to limit the amount of communication between machines so it does not obliterate the benefit of parallelism. We describe a general way to achieve this, *ParMAC*. ParMAC works on a cluster of processing 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 propose a theoretical model of its runtime and parallel speedup. To illustrate our general results in a specific algorithm, we develop ParMAC to learn binary autoencoders with application to fast, approximate image retrieval. We implement this using Message Passing Interface (MPI) in a distributed system and demonstrate nearly perfect speedups in a 128-processor cluster with a training set of 100 million high-dimensional points. The speedups achieved agree well with the prediction of our theoretical speedup model.

1 Introduction: big data, parallel processing and nested models

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 (Fuller and Millett, 2011; Graham et al., 2004). 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, 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 (Hinton and Salakhutdinov, 2006), cascades for object recognition in computer vision (Serre et al., 2007; Ranzato et al., 2007) or for phoneme classification in speech processing (Gold and Morgan, 2000; Saon and Chien, 2012), wrapper approaches to classification or regression (Kohavi and John, 1997), and various combinations of feature extraction/learning and preprocessing prior to some learning task. Nested and hierarchical models are ubiquitous 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 produces inherently 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 and 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. MAC has been applied to various nested models (Carreira-Perpiñán and Wang, 2014; Wang and Carreira-Perpiñán, 2014; Carreira-Perpiñán and Raziperchikolaei, 2015; Raziperchikolaei and Carreira-Perpiñán, 2016; Carreira-Perpiñán and Vladymyrov, 2015). However, the original papers proposing MAC (Carreira-Perpiñán and 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, implements it in Message Passing Interface (MPI) for the problem of learning binary autoencoders (BAs), and demonstrates its ability to train on large datasets and achieve large speedups on a distributed cluster. We first review related work (section 2), describe MAC in general and for BAs (section 3) and introduce the ParMAC model and some extensions of it (section 4). Then, we analyse theoretically ParMAC's parallel speedup (section 5) and convergence (section 6). Finally, we describe our MPI implementation of ParMAC for BAs (section 7) and show experimental results (section 8). Although our MPI implementation and experiments are for a particular ParMAC algorithm (for binary autoencoders), we emphasise that our contributions (the definition of ParMAC and the theoretical analysis of its speedup and convergence) apply to ParMAC in general for any situation where MAC applies, i.e., nested functions with K layers.

2 Related work

Distributed optimisation and large-scale machine learning have been steadily gaining interest in recent years. 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 (McDonald et al., 2010; Bertsekas, 2011; Zinkevich et al., 2010; Gemulla et al., 2011; Niu et al., 2011), parallel CD (Bradley et al., 2011; Richtárik and Takáč, 2013; Liu and Wright, 2015) and parallel ADMM (Boyd et al., 2011; Ouyang et al., 2013; Zhang and Kwok, 2014).

It is instructive to consider the parallel SGD case in some detail. Here, one typically runs SGD independently on data subsets (done by P worker machines), and a parameter server regularly gathers the replica parameters from the workers, averages them and broadcasts them back to the workers. One can show (Zinkevich et al., 2010) that, for a small enough step size and under some technical conditions, the distance to the minimum in objective function value satisfies an upper bound. The upper bound has a term that decreases as the number of workers P increases, so that parallelisation helps, but it has another term that

is independent of P, so that past a certain point parallelisation does not help. In practice, the speedups over serial SGD are generally modest. Also, the theoretical guarantees of parallel SGD are restricted to *shallow* models, as opposed to *deep* or *nested* models, because the composition of functions is nearly always nonconvex. Indeed, parallel SGD can diverge with nonconvex models. The intuitive reason for this is that, with local minima, the average of two workers can have a larger objective value than each of the individual workers, and indeed the average of two minima need not be a minimum. In practice, parallel SGD can give reasonable results with nonconvex models if one takes care to average replica models that are close in parameter space and thus associated with the same optimum (e.g. eliminating "stale" models and other heuristics), but this is not easy (Dean et al., 2012).

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 (with the caveat above), 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. 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 (Chen et al., 2012; Zhang et al., 2013; Coates et al., 2013; Seide et al., 2014).

Another recent trend is on parallel computation abstractions tailored 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), with the goal of making cloud computing easily available to train machine learning models. Again, this is often based on shallow models trained with gradient-based convex optimisation techniques, such as parallel SGD. Some of these systems implement some form of deep neural nets.

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 and Seeger, 2001; Bengio et al., 2004; Drineas and Mahoney, 2005; Talwalkar et al., 2013; Vladymyrov and Carreira-Perpiñán, 2013, 2016).

ParMAC 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 the method of auxiliary coordinates (MAC)

Many machine learning architectures share a fundamental design principle: *mathematically, they construct a (deeply) nested mapping from inputs to outputs*, 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 or binary autoencoders consisting of multiple processing layers. Such problems are traditionally optimised using methods based on gradients computed using the chain rule. However, such gradients may sometimes be inconvenient to use, or may not exist (e.g. if some of the layers are nondifferentiable). Also, they are hard to parallelise, because of the inherent sequentiality in the chain rule.

The *method of auxiliary coordinates (MAC)* (Carreira-Perpiñán and Wang, 2012, 2014) is designed to optimise nested models without using chain-rule gradients while introducing parallelism. It solves an equivalent but in appearance very different problem to the nested one, which affords embarrassing parallelisation. 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 and Wang, 2014), low-dimensional SVMs (Wang and Carreira-Perpiñán, 2014), binary autoencoders (Carreira-Perpiñán and Raziperchikolaei, 2015), affinity-based loss functions for binary hashing (Raziperchikolaei and

Carreira-Perpiñán, 2016) and parametric nonlinear embeddings (Carreira-Perpiñán and Vladymyrov, 2015). In this paper we focus mostly on the particular case of binary autoencoders. These define a nonconvex nondifferentiable problem, yet its MAC algorithm is simple and effective. It allows us to demonstrate, in an actual implementation in a distributed system, the fundamental properties of ParMAC: how MAC introduces parallelism; how ParMAC keeps the communication between machines low; the use of stochastic optimisation in the $\mathbf{W}$ step; and the tradeoff between the different amount of parallelism in the $\mathbf{W}$ and $\mathbf{Z}$ steps. It also allows us to test how good our theoretical model of the speedup is in experiments. We first give the detailed MAC algorithm for binary autoencoders, and then generalise it to $K > 1$ hidden layers.

3.1 Optimising binary autoencoders using MAC

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 $\mathbf{h}(\mathbf{x}) = s(\mathbf{A}\mathbf{x})$ ($\mathbf{A}$ 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 $s(t)$ is a step function applied elementwise, i.e., $s(t) = 1$ if $t \geq 0$ and $s(t) = 0$ 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:

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

Optimising this nonconvex, nonsmooth function is NP-complete. Where the gradients do exist with respect to $\mathbf{A}$ 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:

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

Note the codes are binary. We now apply the quadratic-penalty method (it is also possible to apply the augmented Lagrangian method; Nocedal and Wright, 2006) and minimise the following objective function while progressively increasing μ, so the constraints are eventually satisfied:

$$E_Q(\mathbf{h}, \mathbf{f}, \mathbf{Z}; \mu) = \sum_{n=1}^{N} \left(\|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2 \right) \text{ s.t. } \mathbf{z}_n \in \{0,1\}^L,\ n = 1, \dots, N. \tag{3}$$

Finally, we apply alternating optimisation over $\mathbf{Z}$ and $\mathbf{W} = (\mathbf{h}, \mathbf{f})$. This results in the following two steps:

- 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. $\mathbf{z} \in \{0,1\}^L$. After some transformations, this problem can be solved exactly for small L by enumeration or approximately for larger L by alternating optimisation over bits, initialised by solving the relaxed problem to $[0,1]$ and truncating its solution (see Carreira-Perpiñán and Raziperchikolaei, 2015 for details).
- 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. With linear $\mathbf{h}$ and $\mathbf{f}$ this simply involves fitting L SVMs to $(\mathbf{X}, \mathbf{Z})$ and D linear regressors to $(\mathbf{Z}, \mathbf{X})$.

The user must choose a schedule for the penalty parameter μ (sequence of values $0 < \mu_1 < \cdots < \infty$). 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

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 $\mu = 0 < \mu_1 < \cdots < \mu_\infty$
  for $l = 1, \dots, L$ — **W** step: **h**
    $h_l \leftarrow$ fit SVM to $(\mathbf{X}, \mathbf{Z}_{\cdot l})$
  $\mathbf{f} \leftarrow$ least-squares fit to $(\mathbf{Z}, \mathbf{X})$ — **W** step: **f**
  for $n = 1, \dots, N$ — **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 $\mathbf{h}$, $\mathbf{Z} = \mathbf{h}(\mathbf{X})$

Figure 1: Binary autoencoder MAC algorithm.

(Carreira-Perpiñán and Raziperchikolaei, 2015). This occurs whenever $\mathbf{Z}$ does not change compared to the previous $\mathbf{Z}$ step, which gives a practical stopping criterion. Also, in order to generalise well to unseen data, we stop iterating for a μ value not when we (sufficiently) optimise $E_Q(\mathbf{h}, \mathbf{f}, \mathbf{Z}; \mu)$, but when the precision of the hash function in a validation set decreases. This is a form of early stopping that guarantees that we improve (or leave unchanged) the initial $\mathbf{Z}$, and besides is faster. We also have to initialise $\mathbf{Z}$. This can be done by running PCA and binarising its result, for example. 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 and Raziperchikolaei, 2015). Binary hashing (Grauman and 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 and 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).

In this paper, we focus 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. We also provide an experiment with nonlinear hash functions (RBF network).

3.2 MAC with K layers

We now consider the more general case of MAC with K hidden layers (Carreira-Perpiñán and Wang, 2012, 2014), inputs $\mathbf{x}$ and outputs $\mathbf{y}$ (for a BA, $\mathbf{x} = \mathbf{y}$). It helps to think of the case of a deep net and we will use it as a running example, but the ideas apply beyond deep nets. Consider a regression problem of mapping inputs $\mathbf{x}$ to outputs $\mathbf{y}$ (both high-dimensional) with a deep net $\mathbf{f}(\mathbf{x})$ given a dataset of N pairs $(\mathbf{x}_n, \mathbf{y}_n)$. We minimise the least-squares error (other loss functions are possible):

$$E(\mathbf{W}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{f}(\mathbf{x}_n; \mathbf{W})\|^2 \qquad \mathbf{f}(\mathbf{x}; \mathbf{W}) = \mathbf{f}_{K+1}(\dots \mathbf{f}_2(\mathbf{f}_1(\mathbf{x}; \mathbf{W}_1); \mathbf{W}_2) \dots; \mathbf{W}_{K+1}) \tag{4}$$

where each layer function has the form $\mathbf{f}_k(\mathbf{x}; \mathbf{W}_k) = \sigma(\mathbf{W}_k \mathbf{x})$, i.e., a linear mapping followed by a squashing nonlinearity ($\sigma(t)$ applies a scalar function, such as the sigmoid $1/(1+e^{-t})$, elementwise to a vector argument, with output in $[0, 1]$). We introduce one auxiliary variable per data point and per hidden unit and define the following equality-constrained optimisation problem:

$$\frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{f}_{K+1}(\mathbf{z}_{K,n}; \mathbf{W}_{K+1})\|^2 \text{ s.t. } \left.\begin{cases} \mathbf{z}_{K,n} = \mathbf{f}_K(\mathbf{z}_{K-1,n}; \mathbf{W}_K) \\ \cdots \\ \mathbf{z}_{1,n} = \mathbf{f}_1(\mathbf{x}_n; \mathbf{W}_1) \end{cases}\right\} n = 1, \dots, N. \tag{5}$$

Each $\mathbf{z}_{k,n}$ can be seen as the coordinates of $\mathbf{x}_n$ in an intermediate feature space, or as the hidden unit activations for $\mathbf{x}_n$. Intuitively, by eliminating $\mathbf{Z}$ we see this is equivalent to the nested problem (4); we

can prove under very general assumptions that both problems have exactly the same minimisers (Carreira-Perpiñán and Wang, 2012). Applying the quadratic-penalty method, we optimise the following function:

$$E_Q(\mathbf{W}, \mathbf{Z}; \mu) = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{f}_{K+1}(\mathbf{z}_{K,n}; \mathbf{W}_{K+1})\|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} \|\mathbf{z}_{k,n} - \mathbf{f}_k(\mathbf{z}_{k-1,n}; \mathbf{W}_k)\|^2 \tag{6}$$

over $(\mathbf{W}, \mathbf{Z})$ and drive $\mu \to \infty$. This defines a continuous path $(\mathbf{W}^*(\mu), \mathbf{Z}^*(\mu))$ which, under mild assumptions (Carreira-Perpiñán and Wang, 2012), converges to a minimum of the constrained problem (5), and thus to a minimum of the original problem (4). In practice, we follow this path loosely. The quadratic-penalty objective function can be seen as breaking the functional dependences in the nested mapping $\mathbf{f}$ and unfolding it over layers. Every squared term involves only a shallow mapping; all variables $(\mathbf{W}, \mathbf{Z})$ are equally scaled, which improves the conditioning of the problem; and the derivatives required are simpler: we require no backpropagated gradients over $\mathbf{W}$, and sometimes no gradients at all. We now apply alternating optimisation of the quadratic-penalty objective over $\mathbf{Z}$ and $\mathbf{W}$:

W step (submodels) Minimising over $\mathbf{W}$ for fixed $\mathbf{Z}$ results in a separate minimisation over the weights of each hidden unit—each a single-layer, single-unit *submodel* that can be solved with existing algorithms (logistic regression).

Z step (coordinates) Minimising over $\mathbf{Z}$ for fixed $\mathbf{W}$ separates over the coordinates $\mathbf{z}_n$ for each data point $n = 1, \dots, N$ and can be solved using the derivatives with respect to $\mathbf{z}$ of the single-layer functions $\mathbf{f}_1, \dots, \mathbf{f}_{K+1}$ (omitting the subindex n): $\min_{\mathbf{z}} \|\mathbf{y} - \mathbf{f}_{K+1}(\mathbf{z}_K)\|^2 + \mu \sum_{k=1}^{K} \|\mathbf{z}_k - \mathbf{f}_k(\mathbf{z}_{k-1})\|^2$.

Thus, the $\mathbf{W}$ step results in many independent, single-layer single-unit submodels that can be trained with existing algorithms, without extra programming cost. The $\mathbf{Z}$ step is new and has the form of a "generalised" proximal operator (Rockafellar, 1976; Combettes and Pesquet, 2011). MAC reduces a complex, highly-coupled problem—training a deep net—to a sequence of simple, uncoupled problems (the $\mathbf{W}$ step) which are coordinated through the auxiliary variables (the $\mathbf{Z}$ step). For a large net with a large dataset, this affords an enormous potential for parallel computation.

4 ParMAC: a parallel, distributed computation model for MAC

We now turn to the contribution of this paper, the distributed implementation of MAC algorithms. As we have seen, 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, the following always hold: (1) 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 all or part of the current model. (2) 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 (usually, a given submodel depends, for each n, on only a portion of the vector $(\mathbf{x}_n, \mathbf{y}_n, \mathbf{z}_n)$). We now show how to turn this into a distributed, low-communication *ParMAC* algorithm, give an MPI implementation of ParMAC for BAs, and discuss the convergence of ParMAC. Throughout the paper, unless otherwise indicated, we will use the term "machine" to mean a single-CPU processing unit with its own local memory and disk, which can communicate with other machines in a cluster through a network or shared memory.

4.1 Description of ParMAC

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 $(\mathbf{X}, \mathbf{Y})$, 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, which implicitly implements a stochastic optimisation.* The model

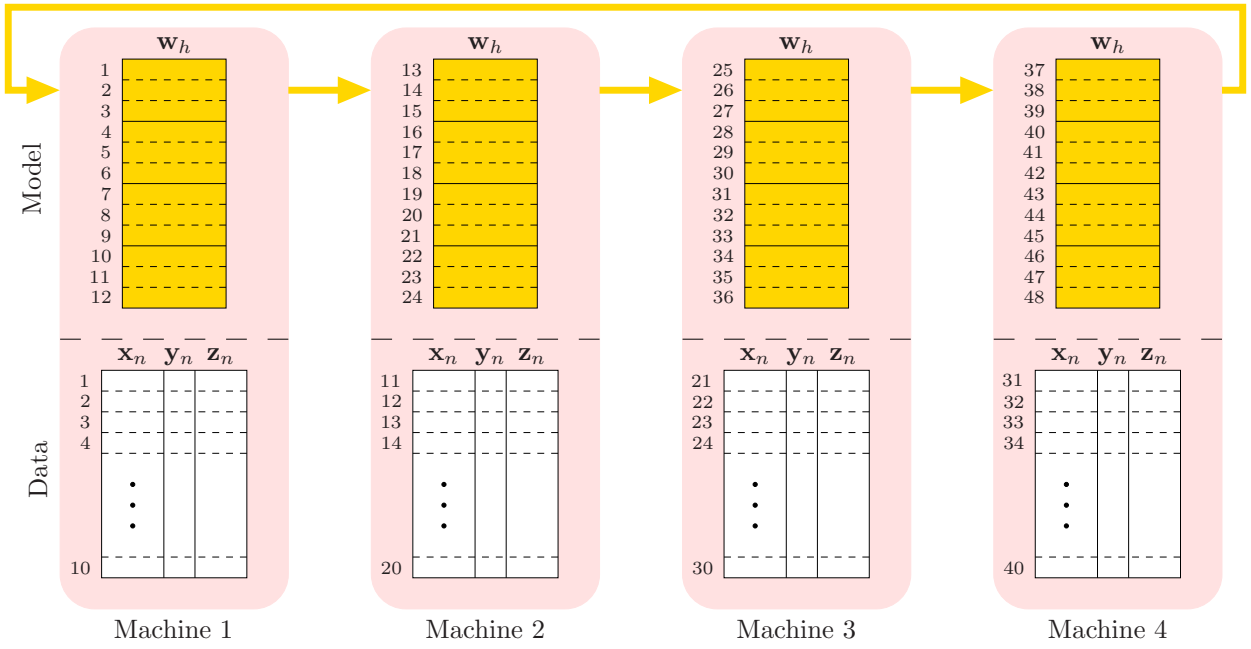

Figure 2: ParMAC model with $P = 4$ machines, $M = 12$ submodels and $N = 40$ data points. "$\mathbf{w}_h$" represents the submodels (hash functions and decoders for BAs, hidden unit weight vectors for deep nets). Submodels h, $h + M$, $h + 2M$ and $h + 3M$ are copies of submodel h, but only one of them is the most currently updated. At the end of the $\mathbf{W}$ step all copies are identical.

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 for simplicity we have P identical processing machines, each with their own memory and CPU, which are connected through a network. The machines are connected in a circular (ring) unidirectional topology, i.e., machine 1 $\rightarrow$ machine 2 $\rightarrow \cdots \rightarrow$ machine $P \rightarrow$ machine 1, where "machine $p \rightarrow$ machine q" means machine p can send data directly to machine q (and we say machine q is the successor of machine p). Call $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n, \mathbf{z}_n)\colon\ n \in \{1, \dots, N\}\}$ the entire dataset and corresponding coordinates. Each machine p will store a subset $\mathcal{D}_p = \{(\mathbf{x}_n, \mathbf{y}_n, \mathbf{z}_n)\colon\ n \in \mathcal{I}_p\}$ such that the subsets are disjoint and their union is the entire data, i.e., the index sets satisfy $\mathcal{I}_p \cap \mathcal{I}_q = \varnothing$ if $p \neq q$ and $\cup_{p=1}^{P} \mathcal{I}_p = \{1, \dots, N\}$.

The $\mathbf{Z}$ step is very simple. Before the $\mathbf{Z}$ step starts[1], each machine will contain all the (just updated) submodels. This means that in the $\mathbf{Z}$ step each machine p processes its auxiliary coordinates $\{\mathbf{z}_n\colon\ n \in \mathcal{I}_p\}$ 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 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 (this contrasts with the intuitive notion of the model sitting in a computer while data arrive and are processed). We achieve this in the circular topology as follows. We assume synchronous processing for simplicity, but in practice one would implement this asynchronously. Assume arithmetic modulo P and an imaginary clock whose period equals the time that any one machine takes to process its portion M/P of submodels. At each clock tick, the P machines update each a different portion M/P of the submodels. For example, in fig. 2, at clock tick 1 machine 1 updates submodels 1–3 using its data $\mathcal{D}_{\mathcal{I}_1}$ (where $\mathcal{I}_1 = \{1, \dots, 10\}$); machine 2 updates submodels 4–6; machine 3 updates submodels 7–9; and machine 4 updates submodels 10–12. This happens in parallel. Then each machine sends the submodels updated to its successor, also in parallel. In the next tick, each machine updates the submodels it just

[1]Also, the machines need not start all at the same time in the $\mathbf{Z}$ step. A machine can start the $\mathbf{Z}$ step on its data as soon as it has received all the updated submodels in the $\mathbf{W}$ step. Likewise, as soon as a machine finishes its $\mathbf{Z}$ step, it can start the $\mathbf{W}$ step immediately, without waiting for all other machines to finish their $\mathbf{Z}$ step. However, in our implementation we consider the $\mathbf{W}$ and $\mathbf{Z}$ steps as barriers, so that all machines start the $\mathbf{W}$ or $\mathbf{Z}$ step at the same time.

received, i.e., machine 1 updates 10–12, machine 2 updates submodels 1–3, machine 3 updates submodels 4–6; and machine 4 updates submodels 7–9 (and each machine always uses its data portion, which never changes). This is repeated until each submodel has visited each machine and thus has been updated with the entire dataset $\mathcal{D}$. This happens after P ticks, and we call this an *epoch*. This process may be repeated for e epochs in eP ticks. At this time, each machine contains M/P submodels that are finished (i.e., updated e times over the entire dataset), and the remaining $M(1-1/P)$ submodels it contains are not finished, indeed the finished versions of those submodels reside in other machines. Finally, before starting with the **Z** step, each machine must contain all the (just updated) submodels (i.e., the parameters for the entire nested model). We achieve this[2] by running a final round of communication without computation, i.e., each machine sends its just updated submodels to its successor. Thus, after one clock tick, machine p sends M/P final submodels to machine $p+1$ and receives M/P submodels from machine $p-1$. After $P-1$ clock ticks, each machine has received the remaining $M(1-1/P)$ submodels that were finished by other machines, hence each machine contains a (redundant) copy of all the current submodels. Fig. 3 illustrates the sequence of operations during one epoch for the example of fig. 2.

In practice, we use an asynchronous implementation. Each machine keeps a queue of submodels to be processed, and repeatedly performs the following operations: extract a submodel from the queue, process it (except in epoch $e+1$) 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 the portion of submodels associated with that machine. Each submodel carries a counter that is initially 1 and increases every time it visits a machine. When it reaches Pe then the submodel is in the last machine and the last epoch. When it reaches $P(e+1)-1$, it has undergone e epochs of processing and all machines have a copy of it, so it has finished the **W** step.

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 **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 **W** step as doing SGD on each submodel in parallel by having each submodel visit the minibatches in each machine.

In summary, using P machines, ParMAC iterates as follows:

W step The submodels (hash functions and decoders for BAs) visit each machine. This implies we train them with stochastic gradient descent, where one "epoch" for a submodel corresponds to that submodel having visited all P machines. All submodels are communicated in parallel, asynchronously with respect to each other, in a circular topology. With e epochs, the entire model parameters are communicated $e+1$ times. The last round of communication is needed to ensure each machine has the most updated version of the model for the **Z** step.

Z step Identical to MAC, each data point's coordinates $\mathbf{z}_n$ are optimised independently, in parallel over machines (since each machine contains $\mathbf{x}_n$, $\mathbf{y}_n$, $\mathbf{z}_n$, and all the model parameters). No communication occurs at all.

4.2 A W step with only two rounds of communication

As described, and as implemented in our experiments, running e epochs in the **W** step requires e rounds of communication (plus a final round). However, we can run e epochs with only 1 round of communication *by having a submodel do e consecutive passes within each machine's data*. In the example of fig. 2, running $e=2$ epochs for submodel $\mathbf{w}_1$ means the following: instead of visiting the data as 1,...,10, 11,...,20, 21,...,30, 31,...,40, 1,...,10, 11,...,20, 21,...,30, 31,...,40, it visits the data as 1,...,10, 1,...,10, 11,...,20, 11,...,20, 21,...,30, 21,...,30, 31,...,40, 31,...,40. (We can also have intermediate schemes such as doing between 1 and e within-machine passes.) This reduces the amount of shuffling, but should not be a problem if the data are randomly distributed over machines (and we can still do within-machine shuffling). This effectively

[2]In MPI, this can be directly achieved with `MPI_Alltoall` broadcasting, which scatters/gathers data from all members to all members of a group (a complete exchange). However, in this paper we implement it using the circular topology mechanism described.

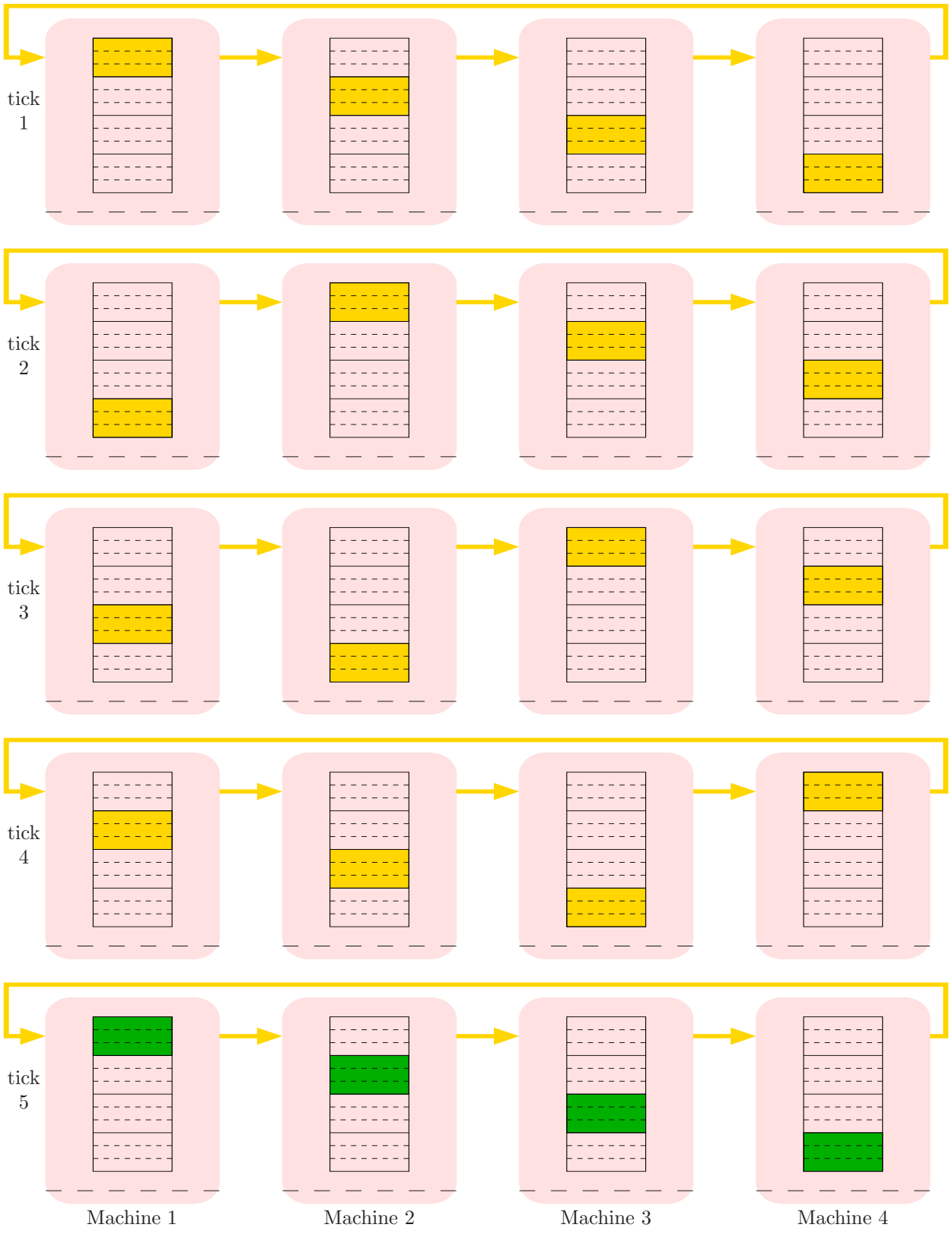

Figure 3: Illustration of one epoch of the synchronous version of ParMAC's **W** step for the example of fig. 2 with $P = 4$ machines and $M = 12$ submodels (we only show the "model" part of the figure). Each row corresponds to one clock tick, within which each machine computes on its portion of M/P submodels (coloured gold) and sends them to its successor. The last tick ($= P + 1$) is the start of the next epoch, at which point all submodels (coloured green) have been updated over the entire dataset.

reduces the total communication in the **W** step to 2 rounds regardless of the number of epochs e (with the second round needed to ensure each machine has the most updated submodels).

4.3 Extensions of ParMAC

In addition, the ParMAC model offers good potential for data shuffling, load balancing, streaming and fault tolerance, which make it attractive for big data. We describe these next.

Data shuffling It is well known that shuffling (randomly reordering) the dataset prior to each epoch improves the SGD convergence speed. With distributed systems, this can sometimes be a problem and require data movement across machines. Shuffling is easy in ParMAC. Within a machine, we can simply access the local data (minibatches) in random order at each epoch. Across machines, we can simply reorganise the circular topology randomly (while still circular) at the beginning of each new epoch (by generating a random permutation and resetting the successor's address of each machine). We could even have each submodel follow a different, random circular topology. However, we do not implement this because it is unlikely to help (since the submodels are independent) and can unbalance the load over machines.

Load balancing This is simple because the work in both the **W** and **Z** steps is proportional to the number of data points N. Indeed, in the **W** step each submodel must visit every data point once per epoch. So, even if the submodels differ in size, the training of any submodel is proportional to N. In the **Z** step, each data point is a separate problem dependent on the current model (which is the same for all points), thus all N problems are formally identical in complexity. Hence, in the assumption that the machines are identical and that each data point incurs the same runtime, load balancing is trivial: the N points are allocated in equal portions of N/P to each machine. If the processing power of machine p is proportional to $\alpha_p > 0$ (where α_p could represent the clock frequency of machine p, say), then we allocate to machine p a subset of the N points proportional to α_p, i.e., machine p gets $N\alpha_p/(\alpha_1 + \cdots + \alpha_P)$ data points. This is done once and for all at loading time.

In practice, we can expect some degradation of the parallel speedup even with identical machines and submodels of the same type. This is because machines do vary for various reasons, e.g. the runtime can be affected by differences in ventilation across machines located in different areas of a data centre, or because machines are running other user processes in addition to the ParMAC optimisation. Another type of degradation can happen if the submodels differ significantly in runtime (e.g. because there are different types of submodels): the runtime of the **W** step will be driven by the slow submodels, which become a bottleneck. As discussed in section 5.4, we can group the M submodels into a smaller number $M' < M$ of approximately equal-size aggregate submodels, for the purpose of estimating the speedup in theory. This need not be the fastest way to schedule the jobs, and in practice we still process the individual submodels asynchronously.

Streaming Streaming refers to the ability to discard old data and to add new data from training over time. This is useful in online learning, or to allow the data to be refreshed, but also may be necessary when a machine collects more data than it can store. The circular topology allows us to add or remove machines on the fly easily, and this can be used to implement streaming.

We consider two forms of streaming: (1) new data are added within a machine (e.g. as this machine collects new data), and likewise old data are discarded within a machine. And (2) new data are added by adding a new machine to the topology, and old data are discarded by removing an existing machine from the topology. Both forms are easily achieved in ParMAC. The first form, within-machine, is trivial: a machine can always add or remove data without any change to the system, because the data for each note is private and never interacts with other machines other than by updating submodels. Adding or discarding data is done at the beginning of the **Z** step. Discarding data simply means removing the corresponding $\{(\mathbf{x}_n, \mathbf{y}_n, \mathbf{z}_n)\}$ from that machine. Adding data means inserting $\{(\mathbf{x}_n, \mathbf{y}_n)\}$ in that machine and, if necessary, creating within that machine coordinate values $\{\mathbf{z}_n\}$ (e.g. by applying the nested model to $\mathbf{x}_n$). We never upload or send any $\mathbf{z}$ values over the network.

The second form, creating a new machine or removing an existing one, is barely more complicated, assuming some support from the parallel processing library. We describe it conceptually. Imagine we

currently have P machines. We can add a new machine, with its own preloaded data $\{(\mathbf{x}_n, \mathbf{y}_n)\}$, as follows. Adding it to the circular topology simply requires connecting it between any two machines (done by setting the address of their successor): before we have "machine $p \rightarrow$ machine $p+1$", afterwards we have "machine $p \rightarrow$ new machine $\rightarrow$ machine $p+1$". We add it in the **W** step, making sure it receives a copy of the final model that has just been finished. The easiest way to do this is by inserting it in the topology at the end of the **W** step, when each machine is simply sending along a copy of the final submodels. In the **Z** step, we proceed as usual, but with $P+1$ machines. Removing a machine is easier. To remove machine p, we do so in the **Z** step, by reconnecting "machine $p-1 \rightarrow$ machine $p+1$" and returning machine p to the cluster. That is all. In the subsequent **W** step, all machines contain the full model, and the submodels will visit the data in each machine, thus not visiting the data in the removed machine.

Fault tolerance This situation is similar to discarding a machine in streaming, except that the fault can occur at any time and is not intended. We can handle it with a little extra bookkeeping, and again assuming some support from the parallel processing library. Imagine a fault occurs at machine p and we need to remove it. If it happens during the **Z** step, all we need to do is discard the faulty machine and reconnect the circular topology. If it happens during the **W** step, we also discard and reconnect, but in addition we need to rescue the submodels that were being updated in p, which we lose. To do this, we revert to the previously updated copy of them, which resides in the predecessor of p in the circular topology (if no predecessor, we are at the beginning of the **W** step and we can use any copy in any machine). As for the remaining submodels being updated in other machines, some will have already been updated in p (which require no action) and some will not have been updated in p yet (which should not visit p anymore). We can keep track of this information by tagging each submodel with a list of the machines it has not yet visited. At the beginning of the **W** step the list of each submodel contains $\{1, \ldots, P\}$, i.e., all machines. When this list is empty, for a submodel, then that submodel is finished and needs no further updates.

Essentially, the robustness of ParMAC to faults comes from its in-built redundance. In the **Z** (and **W**) step, we can do without the data points in one machine because a good model can still be learned from the remaining data points in the other machines. In the **W** step, we can revert to older copies of the lost submodels residing in other machines.

The asynchronous implementation of ParMAC we described earlier relied on tagging each submodel with a counter in order to know whether it needs processing and communicating. A more general mechanism to run ParMAC asynchronously is to tag each submodel with a list (per epoch) of machines it has to visit. All a machine p needs to do upon receiving a submodel is check its list: if p is not in the list, then the submodel has already visited machine p and been updated with its data, so machine p simply sends it along to its successor without updating it again. If p is in the list, then machine p updates the submodel, removes p from its list, and sends it along to its successor. This works even if we use a different communication topology for each submodel at each epoch.

5 A theoretical model of the parallel speedup for ParMAC

In this section we give a theoretical model to estimate the computation and communication times and the parallel speedup in ParMAC. Specifically, eq. (12) gives the speedup $S(P)$ as a function of the number of machines P and other parameters, which seems to agree well with our experiments (section 8.3). In practice, this model can be used to estimate the optimal number of machines P to use, or to explore the effect on the speedup of different parameter settings (e.g. the number of submodels M). Throughout the rest of the paper, we will call "speedup" $S(P)$ the ratio of the runtime using a single machine (i.e., the serial code) vs using $P > 1$ machines (the parallel code), and "perfect speedup" when $S(P) = P$. Our theoretical model applies to the general ParMAC case of K layers, whether differentiable or not; it only assumes that the resulting submodels after introducing auxiliary coordinates are of the same "size," i.e., have the same computation and communication time (this assumption can be relaxed, as we discuss at the end of the section).

We can obtain a quick, rough understanding of the speedup appealing to (a generalisation of) Amdahl's law (Goedecke and Hoisie, 2001). ParMAC iterates the **W** and **Z** steps as follows (where M is the number of submodels and N the number of data points):

Roughly speaking, the **W** step has M independent problems so its speedup would be $\min(M, P)$, while the **Z** step has N independent problems so its speedup would be $\min(N, P) = P$ (because in practice $N \geq P$). So the overall speedup would be between M and P depending on the relative runtimes of the **W** and **Z** steps. This suggests we would expect a nearly perfect speedup $S \approx P$ with $P \leq M$ and diminishing returns for $P > M$. This simplified picture ignores important factors such as the ratio of computation vs communication (which our model will make more precise), but it does capture the basic, qualitative behaviour of the speedup.

5.1 The theoretical model of the speedup

Let us now develop a more precise, quantitative model. Consider a ParMAC algorithm, operating synchronously, such that there are M independent submodels of the same size in the **W** step, on a dataset with N training points, distributed over P identical machines (each with N/P points). The ParMAC algorithm runs a certain number of iterations, each consisting of a **W** and a **Z** step, so if we ignore small overheads (setup and termination), we can estimate the total runtime as proportional to the number of iterations. Hence, we consider a theoretical model of the runtime of one iteration of the ParMAC algorithm, given the following parameters:

- P: number of machines.
- N: number of training points.
  We assume $N > P$ is divisible by P. This is not a problem because $N \gg P$ in practice (otherwise, there would be no reason to distribute the optimisation).
- M: number of submodels in the **W** step.
  This may be smaller than, equal to or greater than P.
- e: number of epochs in the **W** step.
- $t_r^{\mathbf{W}}$: computation time per submodel and data point in the **W** step.
  This is the time to process (within the current epoch) one data point by a submodel, i.e., the time do an SGD update to a weight vector, per data point (if we use minibatches, then this is the time to process one minibatch divided by the size of the minibatch).
- $t_c^{\mathbf{W}}$: communication time per submodel in the **W** step.
  This is the time to send one submodel from one machine to another, including overheads such as buffering, partitioning into messages or waiting time. We assume communication does not overlap with computation, i.e., a machine can either compute or communicate at a given time but not both. Also, communication involves time spent both by the sender and the receiver; we interpret $t_c^{\mathbf{W}}$ as the time spent by a given machine in first receiving a submodel and then sending it.
- $t_r^{\mathbf{Z}}$: computation time per data point in the **Z** step.
  This is the time to finish one data point entirely, using whatever optimisation algorithm performs the **Z** step.

P, N, M and e are integers greater or equal than 1, and $t_r^{\mathbf{W}}$, $t_c^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}$ are real values greater than 0. This model assumes that $t_r^{\mathbf{W}}$, $t_c^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}$ are constant and equal for every submodel or and data point. In reality, even if the submodels are of the same mathematical form and dimension (e.g. each submodel is a weight vector of a linear SVM of dimension D), the actual times may vary somewhat due to many factors. However, as we will show in section 8.3, the model does agree quite well with the experimentally measured speedups.

Let us compute the runtimes in the **W** and **Z** step under these model assumptions. The runtime in the **Z** step equals the time for any one machine to process its N/P points on all M submodels, i.e.,

$$T^{\mathbf{Z}}(P) = M \tfrac{N}{P} t_r^{\mathbf{Z}} \tag{7}$$

since all machines start and end at the same time and do the same amount of computation, without communication. To compute the runtime in the **W** step, we again consider the synchronous procedure of section 4. At each tick of an imaginary clock, each machine processes its portion M/P of submodels and sends it to its successor. After P ticks, this concludes one epoch. This is repeated for e epochs, followed by a final round of communication of all the submodels. If M is not divisible by P, say $M = QP + R$ with $Q, R \in \mathbb{N}$ and $0 < R < P$, we can apply this procedure pretending there are $P - R$ fictitious submodels[3] (on which machines do useless work). Then, the runtime in each tick is $\lceil M/P \rceil \frac{N}{P} t_r^{\mathbf{W}}$ (time for any one machine to process its N/P points on its portion $\lceil M/P \rceil$ of submodels) plus $\lceil M/P \rceil t_c^{\mathbf{W}}$ (time for any one machine to send its portion of submodels). The total runtime of the **W** step is then Pe times this plus the time of the final round of computation:

$$T^{\mathbf{W}}(P) = \lceil M/P \rceil \left(t_r^{\mathbf{W}} \tfrac{N}{P} + t_c^{\mathbf{W}} \right) Pe + \lceil M/P \rceil t_c^{\mathbf{W}} P. \tag{8}$$

(The final round actually requires $P - 1$ ticks, but we take it as P ticks to simplify the equation a bit.) Finally, the total runtime $T^{\mathbf{W}}(P) + T^{\mathbf{Z}}(P)$ of one ParMAC iteration (**W** and **Z** step) with P machines is:

$$T(P) = M \tfrac{N}{P} t_r^{\mathbf{Z}} + P \lceil M/P \rceil \left(e \left(t_r^{\mathbf{W}} \tfrac{N}{P} + t_c^{\mathbf{W}} \right) + t_c^{\mathbf{W}} \right), \ P > 1 \tag{9}$$

$$T(1) = MNt_r^{\mathbf{Z}} + MNet_r^{\mathbf{W}} \tag{10}$$

where for $P = 1$ machine we have no communication ($t_c^{\mathbf{W}} = 0$). Hence, the parallel speedup is

$$S(P) = \frac{T(1)}{T(P)} = \frac{M/P}{\lceil M/P \rceil} \frac{et_r^{\mathbf{W}} + t_r^{\mathbf{Z}}}{\frac{1}{P}\left(et_r^{\mathbf{W}} + \frac{M/P}{\lceil M/P \rceil} t_r^{\mathbf{Z}} \right) + \frac{1}{N}(e+1)t_c^{\mathbf{W}}} = \frac{\frac{1}{\lceil M/P \rceil} M(et_r^{\mathbf{W}} + t_r^{\mathbf{Z}})P}{\frac{1}{N}(e+1)t_c^{\mathbf{W}} P^2 + et_r^{\mathbf{W}} P + \frac{1}{\lceil M/P \rceil} M t_r^{\mathbf{Z}}} \tag{11}$$

which can be written more conveniently as

$$S(P) = \frac{\rho \frac{1}{\lceil M/P \rceil} MP}{\frac{1}{N}P^2 + \rho_2 P + \rho_1 \frac{1}{\lceil M/P \rceil} M} \tag{12}$$

by defining the following constants:

$$\rho_1 = \frac{t_r^{\mathbf{Z}}}{(e+1)t_c^{\mathbf{W}}} \qquad \rho_2 = \frac{et_r^{\mathbf{W}}}{(e+1)t_c^{\mathbf{W}}} \qquad \rho = \rho_1 + \rho_2 = \frac{et_r^{\mathbf{W}} + t_r^{\mathbf{Z}}}{(e+1)t_c^{\mathbf{W}}}. \tag{13}$$

These constants can be understood as ratios of computation vs communication, independent of the training set size, number of submodels and number of machines. These ratios depend on the actual computation within the **W** and **Z** step, and on the performance of the distributed system (computation power of each machine, communication speed over the network or shared memory, efficiency of the parallel processing library that handles the communication between machines). The value of these ratios can vary considerably in practice, but it will typically be quite smaller than 1 (say, $\rho \in [10^{-4}, 1]$), because communication is much slower than computation in current computer architectures.

5.2 Analysis of the speedup model

We can characterise the speedup $S(P)$ of eq. (12) in the following three cases:

- *If $M \geq P$ and M is divisible by P*, then we can write the speedup as follows:

$$\text{if } M \text{ divisible by } P\text{:} \quad S(P) = 1/\left(\frac{1}{P} + \frac{1}{\rho N} \right) = P/\left(1 + \frac{P}{\rho N} \right) \leq P. \tag{14}$$

[3]This means that our estimated runtime is an upper bound, because when M is not divisible by P, there may be a better way to organise the computation in the **W** step that reduces the time when any machine is idle. In practice this is irrelevant because we implement the computation asynchronously. Each machine keeps a queue of incoming submodels it needs to process, from which it repeatedly takes one submodel, processes it and sends it to the machine's successor.

Here, the function $S(P)$ is independent of M and monotonically increasing with P. It would asymptote to $\lim_{P\to\infty} S(P) = \rho N$, but the expression is only valid up to $P = M$. From (14) we derive the following condition for perfect speedup to occur (in the limit)[4]:

$$S \approx P \iff P \ll \rho N. \tag{15}$$

This gives an upper bound on the number P of machines to achieve an approximately perfect speedup. Although ρ is quite small in practice, the value of N is very large (typically millions or greater), otherwise there would be no need to distribute the data. Hence, we expect $\rho N \gg 1$, so P could be quite large. In fact, the limit in how large P can be does not come from this condition (which assumes $P \leq M$ anyway) but from the number of submodels, as we will see next.
In summary, we conclude that if $M \geq P$ and M is divisible by P then the speedup $S(P)$ is given by (14), and in practice $S(P) \approx P$ typically.

- *If $M \geq P$ and M is not divisible by P*, then $S(P)$ is given by the full expression (12), which is studied in appendix A. $S(P)$ is piecewise continuous on M intervals of the form

$$\left[1, \tfrac{M}{M-1}\right),\ \left[\tfrac{M}{M-1}, \tfrac{M}{M-2}\right),\ \ldots,\ \left[\tfrac{M}{2}, M\right),\ [M, \infty). \tag{16}$$

Within each interval $P \in \left[\frac{M}{k}, \frac{M}{k-1}\right)$ for $k = 1, 2, 3 \ldots, M$ we have $\lceil M/P \rceil = k$ and we obtain that $S(P)$ either is monotonically increasing, or is monotonically decreasing, or achieves a single maximum at

$$P_k^* = \sqrt{\rho_1 MN/k} \qquad S_k^* = S(P_k^*) = \frac{\rho M/k}{\rho_2 + 2\sqrt{\rho_1 M/Nk}}. \tag{17}$$

The parallelisation ability in this case is less than if M is divisible by P, since now some machines are idle at some times during the **W** step.

- *If $M < P$*, then we can write the speedup as follows:

$$\text{if } M < P\text{:} \quad S(P) = \rho / \left(\frac{\rho_1}{P} + \frac{\rho_2}{M} + \frac{P}{MN}\right) = \rho M / \left(\rho_2 + \rho_1 \frac{M}{P} + \frac{P}{N}\right) \tag{18}$$

which corresponds to the last interval $P \in [M, \infty)$ (for $k = 1$) over which S is continuous. We obtain that $S(P)$ either is monotonically decreasing (if $M \geq P_1^*$), or it increases from $P = M$ up to a single maximum at $P = P_1^*$ and then decreases monotonically, with

$$P_1^* = \sqrt{\rho_1 MN} \qquad S_1^* = S(P_1^*) = \frac{\rho M}{\rho_2 + 2\sqrt{\rho_1 M/N}}. \tag{19}$$

As $P \to \infty$ we have that $S(P) \approx \rho NM/P \to 0$ (assuming $t_c^{\mathbf{W}} > 0$ so $\rho < \infty$). This decrease of the speedup for large P is caused by the communication overhead in the **W** step, where $P - M$ machines are idle at each tick in the **W** step.
In the impractical case where there is no communication cost ($t_c^{\mathbf{W}} = 0$ so $\rho = \infty$) then $S(P)$ is actually monotonically increasing and $\lim_{P\to\infty} S(P) = S_1^* = \frac{\rho}{\rho_2} M > M$, so the more machines the larger the speedup, although with diminishing returns.

Theorem A.1 shows that $S(P)$ at the beginning of each interval is greater than anywhere before that interval, i.e., $S(M/k) > S(P)\ \forall P < M/k$, for $k = 1, 2, \ldots, M$. That is, although the speedup $S(P)$ is not necessarily monotonically increasing for $P \geq 1$, it is monotonically increasing for $P \in \left\{1, \frac{M}{M-1}, \frac{M}{M-2}, \ldots, \frac{M}{2}, M\right\}$. This suggests selecting values of P that make M/P integer, in particular when M is divisible by P.

[4]Note that if $t_c^{\mathbf{W}} = 0$ (no communication overhead) then $\rho = \infty$ and there is no upper bound in (15), but $P \leq N$ still holds, because we have to have at least one data point per machine.

Globally maximum speedup $S^* = \max_{P \geq 1} S(P)$ This is given by (see appendix A):

- If $M \geq \rho_1 N$: $S^* = M / \left(1 + \frac{M}{\rho N}\right) \leq M$, achieved at $P = M$.
- If $M < \rho_1 N$: $S^* = S_1^* = \frac{\rho M}{\rho_2 + 2\sqrt{\rho_1 M/N}} > M$, achieved at $P = P_1^* = \sqrt{\rho_1 MN} > M$.

In practice, with large values of N, the more likely case is $S^* = S_1^* > M$ for $P = P_1^* > M$. In this case, the maximum speedup is achieved using more machines than submodels (even though this means some machines will be idle at some times in the **W** step), and is bigger than M. Since diminishing returns occur as we approach the maximum, the practically best value of P will be somewhat smaller than P_1^*.

The "large dataset" case The case where N is large is practically important because the need for distributed optimisation arises mainly from this. Specifically, if we take $P \ll \rho_2 N$, the speedup becomes (see appendix A):

$$\text{if } M \text{ divisible by } P\text{:} \quad S(P) \approx P; \qquad \text{if } M > P\text{:} \quad S(P) \approx \rho / \left(\frac{\rho_1}{P} + \frac{\rho_2}{M}\right) \tag{20}$$

so that the speedup is almost perfect up to $P = M$, and then it is approximately the weighted harmonic mean of M and P (hence, $S(P)$ is monotonically increasing and between M and P). For $P \gg \rho_1$, we have $S(P) \approx \frac{\rho}{\rho_2} M > M$.

The "dominant Z step" case If we take $t_r^{\mathbf{Z}} \gg t_r^{\mathbf{W}}, t_c^{\mathbf{W}}$ or equivalently $\rho \approx \rho_1$ very large, which means the **Z** step dominates the runtime, then $S(P) \approx P$. This is because the **Z** step parallelises perfectly (as long as $P < N$).

Transformations that keep the speedup invariant We can rewrite the speedup of eq. (12) as:

$$S(P) = \frac{\rho' \frac{1}{\lceil M/P \rceil} MP}{P^2 + \rho_2' P + \rho_1' \frac{1}{\lceil M/P \rceil} M} \tag{21}$$

with

$$\rho' = \rho N = (\rho_1 + \rho_2) N = \frac{N(e t_r^{\mathbf{W}} + t_r^{\mathbf{Z}})}{(e+1) t_c^{\mathbf{W}}} \qquad \rho_1' = \rho_1 N = \frac{N t_r^{\mathbf{Z}}}{(e+1) t_c^{\mathbf{W}}} \qquad \rho_2' = \rho_2 N = \frac{N e t_r^{\mathbf{W}}}{(e+1) t_c^{\mathbf{W}}} \tag{22}$$

so that $S(P)$ is independent of N, which has been absorbed into the communication-computation ratios. This means that $S(P)$ depends on the dataset size (N) and computation/communication times ($t_r^{\mathbf{W}}$, $t_r^{\mathbf{Z}}$, $t_c^{\mathbf{W}}$) only through ρ', ρ_1' and ρ_2', and is therefore invariant to parameter transformations that leave these ratios unchanged. Such transformations are the following (where $\alpha > 0$):

- Scaling N, $t_r^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}$ as αN, $\frac{1}{\alpha} t_r^{\mathbf{W}}$ and $\frac{1}{\alpha} t_r^{\mathbf{Z}}$.
  "Larger dataset, faster computation," or "smaller dataset, slower computation."
- Scaling N and $t_c^{\mathbf{W}}$ as αN and $\alpha t_c^{\mathbf{W}}$
  "Larger dataset, slower communication," or "smaller dataset, faster communication."
- Scaling $t_r^{\mathbf{W}}$, $t_r^{\mathbf{Z}}$ and $t_c^{\mathbf{W}}$ as $\alpha t_r^{\mathbf{W}}$, $\alpha t_r^{\mathbf{Z}}$ and $\alpha t_c^{\mathbf{W}}$
  "Faster computation, faster communication," or "slower computation, slower communication."

5.3 Discussion and examples

Fig. 4 plots a "typical" speedup curve $S(P)$, obtained with a realistic choice of parameter values. It displays the prototypical speedup shape we should expect in practice (the experimental speedups of fig. 10 confirm this). For $P \leq M$ the curve is very close to the perfect speedup $S(P) = P$, slowly deviating from it as P approaches M. For $P > M$, the curve continues to increase until it reaches its maximum at $P = P_1^*$, and decreases thereafter.

Fig. 5 plots $S(P)$ for a wider range of parameter settings. We set the dataset size to a practically small value ($N = 50\,000$), otherwise the curves tend to look like the typical curve from fig. 4. The parameter settings are representative of different, potential practical situations (some more likely than others). We note the following observations:

- Again, the most important observation is that the number of submodels M is the parameter with the most direct effect on the speedup: near-perfect speedups ($S \approx P$) occur if $M \geq P$, otherwise the speedups are between M and P (and eventually saturate if $P \gg M$).
- When the time spent on communication is large in relative terms, the speedup is decreased. This can happen when the runtime of the $\mathbf{Z}$ step is low (small $t_r^{\mathbf{Z}}$), when the communication cost is large (large $t_c^{\mathbf{W}}$), or with many epochs (large e). Indeed, since the $\mathbf{Z}$ step is perfectly parallelisable, any decrease of the speedup should come from the $\mathbf{W}$ step.
- Some of the curves display noticeable discontinuities, caused by the function $(M/P)/\lceil M/P \rceil$, occurring at values of P of the form M/k for $k \in \{1, \dots, M\}$. At each such value, $S(P)$ is greater than for any smaller value of P, in accordance with theorem A.1. This again suggests selecting values of P that make M/P integer, in particular when M is a multiple of P (P, $2P$, $3P \dots$). This achieves the best speedup efficiency in that machines are never idle (in our theoretical model).
  Also, for fixed P, the function $(M/P)/\lceil M/P \rceil$ can take the same value for different values of M (e.g. $M = 32$ and $M = 64$ for $P = 60$). This explains why some curves (for different M) partly overlap.
- The maximum speedup is typically larger than M and occurs for $P > M$. It is possible to have the maximum speedup be smaller than M (in which case it occurs at $P = M$); an example appears in fig. 5, row 3, column 2 (for the larger M values). But this happens only when $M \geq \rho_1 N$, which requires an unusually small dataset and an impractically large number of submodels. Generally, we should expect $P > M$ to be beneficial. This is also seen experimentally in fig. 10.

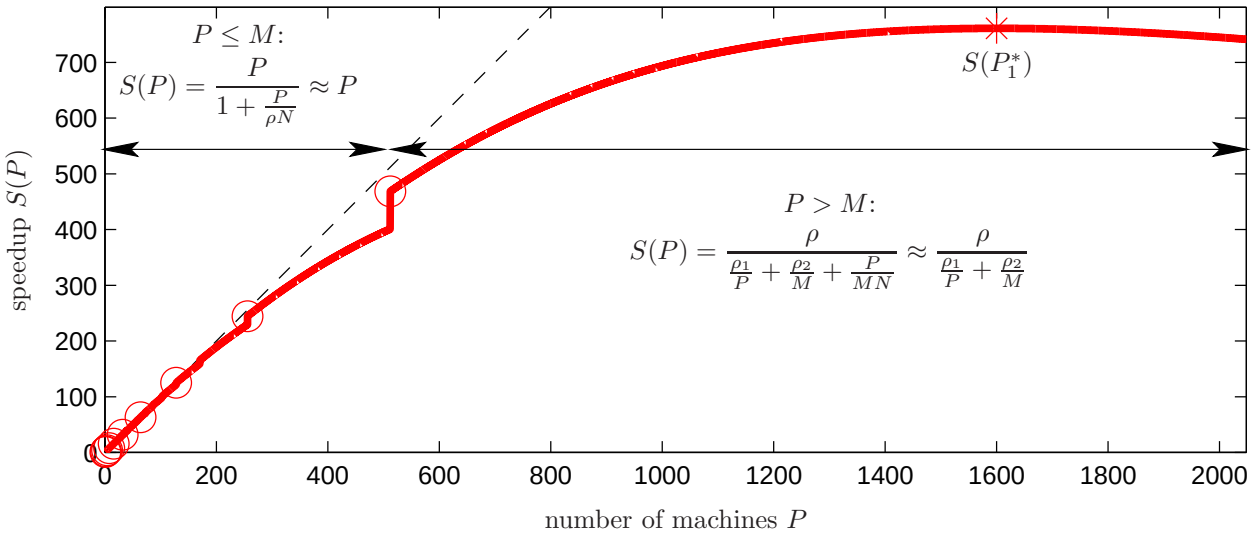

Figure 4: Typical form of the theoretical speedup curve for realistic parameter settings, specifically $N = 10^6$ data points, $M = 512$ submodels, $e = 1$ epoch in the $\mathbf{W}$ step, and $t_r^{\mathbf{W}} = 1$ (this sets the units of time), $t_r^{\mathbf{Z}} = 5$ and $t_c^{\mathbf{W}} = 10^3$ (so $\rho_1 = 0.0025$, $\rho_2 = 0.0005$ and $\rho = 0.003$). Some of the discontinuities of the curve (where $\lceil M/P \rceil$ is discontinuous) are visible. We mark the values for P such that M is divisible by P ($\circ$) and the maximum speedup ($*$), which occurs for $P = P_1^* > M$.

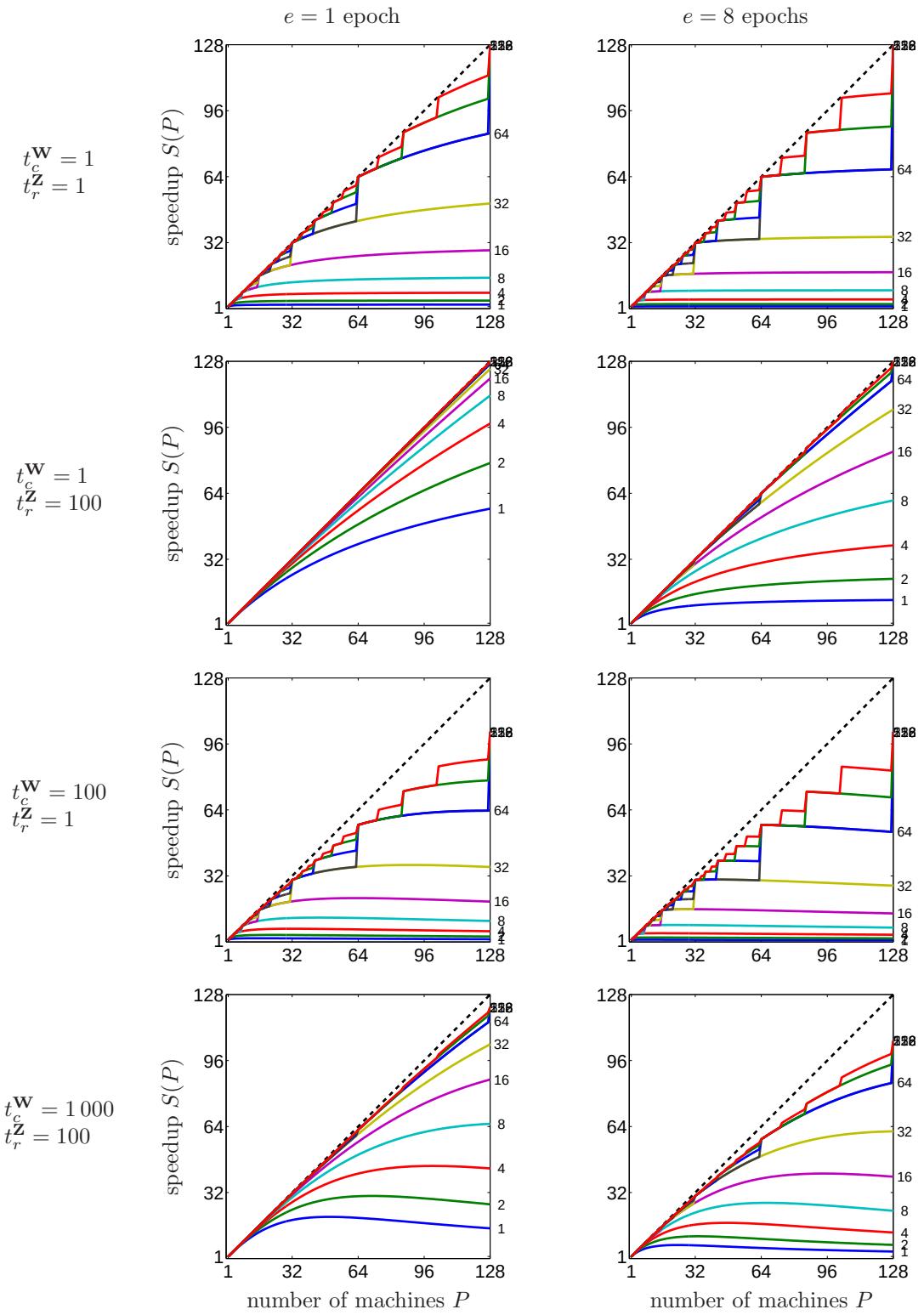

Figure 5: Theoretical speedup $S(P)$ as a function of the number of machines P for various settings of the parameters of ParMAC with a binary autoencoder. The parameters are: dataset size $N = 50\,000$ training points; number of submodels $M \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$; number of epochs in the $\mathbf{W}$ step $e \in \{1, 8\}$; $\mathbf{W}$ step computation time (per submodel and data point) $t_r^{\mathbf{W}} = 1$ (this sets the units of time); $\mathbf{W}$ step communication time (per submodel) $t_c^{\mathbf{W}} \in \{1, 100, 1\,000\}$; $\mathbf{Z}$ step computation time (per submodel and data point) $t_r^{\mathbf{Z}} \in \{1, 100\}$. Within each plot, each curve corresponds to one value of M, indicated on the right end of the plot.

5.4 Practical considerations

In practice, given a specific problem (with a known number of submodels M, epochs e and dataset size N), our theoretical speedup curves can be used to determine optimal values for the number of machines P to use. As seen in section 8.3, the theoretical curves agree quite well with the experimentally measured speedups. The theoretical curves do need estimates for the computation time and communication times of the **W** and **Z** steps. These are hard to obtain a priori; the computational complexity of the algorithm in $\mathcal{O}$-notation ignores constant factors that affect significantly the actual times. Their estimates should be measured from test runs.

As seen from eq. (21), we can leave the speedup unchanged by trading off dataset size (N) and computation/communication times ($t_r^{\mathbf{W}}$, $t_r^{\mathbf{Z}}$, $t_c^{\mathbf{W}}$) in various ways, as long as one of the three following holds: the products $Nt_r^{\mathbf{W}}$ and $Nt_r^{\mathbf{Z}}$ remain constant; or the quotient $N/t_c^{\mathbf{W}}$ remains constant; or the quotients $t_r^{\mathbf{W}}/t_c^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}/t_c^{\mathbf{W}}$ remain constant.

Theoretically, the most efficient operating points for P are values such that M is divisible by P, because this means no machine is ever idle. In practice with an asynchronous implementation and with $t_r^{\mathbf{Z}}$, $t_r^{\mathbf{W}}$ and $t_c^{\mathbf{W}}$ exhibiting some variation over submodels and data points, this is not true anymore. Still, if in a given application one is constrained to using $P \leq M$ machines, choosing P close to a divisor of M would probably be preferable.

One assumption in our speedup model is that the P machines are identical in processing power. The model does extend to the case where the machines are different, as noted in our discussion of load balancing (section 4.3). This is because the work performed by each machine is proportional to the number of data points it contains: in the **W** step, because every submodel runs (implicitly) SGD, and every submodel must visit each machine; in the **Z** step, because each data point is a separate problem, and involves all submodels (which reside in each machine). Hence, we can equalise the work over machines by loading each machine with an amount of data proportional to its processing speed, independent of the number of submodels.

Another assumption in our model (in the **W** step) is that all M submodels are identical in "size" (computation and communication time). This is sometimes not true. For example, in the BA, we have submodels of two types: the L encoders (each a binary linear SVM operating on a D-dimensional input) and the D decoders (each a linear regressor operating on an L-dimensional input). Since $D > L$, the encoders are bigger than the decoders and take longer to train and communicate. We can still apply our speedup model if we "group" smaller submodels into a single submodel of size comparable to the larger submodels, so as to equalise as much as possible the submodel sizes (the actual implementation does not need to group submodels, of course). For the BA, under the reasonable assumption that the ratio of computation times $t_r^{\mathbf{W}}$ (and communication times $t_c^{\mathbf{W}}$) of decoder vs encoder is $L/D < 1$, we can group the D decoders into L groups of D/L decoders each. Each group of decoders has now a computation and communication time equal to that of one encoder. This gives an effective number of independent submodels $M = 2L$, and this is what we use when applying the model to the experimental speedups in section 8.3.

Finally, we emphasise that the goal of this section was to characterise the parallel speedup of ParMAC quantitatively and demonstrate the runtime gains that are achievable by using P machines. In practice, other considerations are also important, such as the economic cost of using P machines (which may limit the maximum P available); the type of machines (obviously, we want all the computation and communication times as small as possible); the choice of optimisation algorithm in the **W** and **Z** steps; the fact that, because of its size, the dataset may need to be, or already is, distributed across P machines; etc. It is also possible to combine ParMAC with other, orthogonal techniques. For example, if each of the submodels in the **W** step is a convex optimisation problem (as is the case with the linear SVMs with the binary autoencoder), we could use the techniques described in section 2 for distributed convex optimisation to each submodel. This would effectively allow for larger speedups when $P > M$.

6 Convergence of ParMAC

The only approximation that ParMAC makes to the original MAC algorithm is using SGD in the **W** step. Since we can guarantee convergence of SGD under certain conditions, we can recover the original convergence guarantees for MAC. Let us see this in more detail. Convergence of MAC to a stationary point is given by theorem B.3 in Carreira-Perpiñán and Wang (2012), which we quote here:

Theorem 6.1. *Consider the constrained problem of eq.* (5) *and its quadratic-penalty function* $E_Q(\mathbf{W}, \mathbf{Z}; \mu)$ *of eq.* (6). *Given a positive increasing sequence* $(\mu_k) \to \infty$, *a nonnegative sequence* $(\tau_k) \to 0$, *and a starting point* $(\mathbf{W}^0, \mathbf{Z}^0)$, *suppose the quadratic-penalty method finds an approximate minimiser* $(\mathbf{W}^k, \mathbf{Z}^k)$ *of* $E_Q(\mathbf{W}^k, \mathbf{Z}^k; \mu_k)$ *that satisfies* $\left\|\nabla_{\mathbf{W},\mathbf{Z}} E_Q(\mathbf{W}^k, \mathbf{Z}^k; \mu_k)\right\| \leq \tau_k$ *for* $k = 1, 2, \ldots$ *Then,* $\lim_{k\to\infty} (\mathbf{W}^k, \mathbf{Z}^k) = (\mathbf{W}^*, \mathbf{Z}^*)$, *which is a KKT point for the problem* (5), *and its Lagrange multiplier vector has elements* $\boldsymbol{\lambda}_n^* = \lim_{k\to\infty} -\mu_k \left(\mathbf{Z}_n^k - \mathbf{F}(\mathbf{Z}_n^k, \mathbf{W}^k; \mathbf{x}_n)\right)$, $n = 1, \ldots, N$.

This theorem applies to the general case of K differentiable layers, where the standard Karush-Kuhn-Tucker (KKT) conditions hold (Nocedal and Wright, 2006). It relies on a standard condition for penalty methods for nonconvex problems (Nocedal and Wright, 2006), namely that we must be able to reduce the gradient of the penalised function E_Q below an arbitrary tolerance $\tau_k \geq 0$ for each value μ_k of the penalty parameter (in MAC iterations $k = 1, 2, \ldots$). This can be achieved by running a suitable (unconstrained) optimisation method for sufficiently many iterations. How does this change in the case of ParMAC? The $\mathbf{Z}$ step remains unchanged with respect to MAC (the fact that the optimisation is distributed is irrelevant since the N subproblems $\mathbf{z}_1, \ldots, \mathbf{z}_N$ are independent). The $\mathbf{W}$ step does change, because we are now obliged to use a distributed, stochastic training. What we need to ensure is that we can reduce the gradient of the penalised function with respect to each submodel (since they are independent subproblems in the $\mathbf{W}$ step) below an arbitrary tolerance. This can also be guaranteed under standard conditions. In general, we can use convergence conditions from stochastic optimisation (Benveniste et al., 1990; Kushner and Yin, 2003; Pflug, 1996; Spall, 2003; Bertsekas and Tsitsiklis, 2000). Essentially, these are Robbins-Monro schedules, which require the learning rate η_t of SGD to decrease such that $\lim_{t\to\infty} \eta_t = 0$, $\sum_{t=1}^{\infty} \eta_t = \infty$, $\sum_{t=1}^{\infty} \eta_t^2 < \infty$, where t is the epoch number (SGD iteration, or pass over the entire dataset[5]). We can give much tighter conditions on the convergence and the convergence rate when the subproblems in the $\mathbf{W}$ step are convex (which is often the case, as with logistic or linear regression, linear SVMs, etc.). This is a topic that has received much attention recently (see section 2), and many such conditions exist, often based on techniques such as Nesterov accelerated algorithms and stochastic average gradient (Cevher et al., 2014). They typically bound the distance to the minimum in objective function value as $\mathcal{O}(1/t^\alpha)$ or $\mathcal{O}(1/\beta^t)$ where the coefficients $\alpha > 0$, $0 < \beta < 1$ and the constant factors in the $\mathcal{O}$-notation depend on the (strong) convexity properties of the problem, Lipschitz constant, etc.

In summary, *convergence of ParMAC to a stationary point is guaranteed by the same theorem as MAC, with an added SGD-type condition for the* $\mathbf{W}$ *step.* This convergence guarantee is independent of the number of layers and submodels (since they are independent in the $\mathbf{W}$ step) and the number of machines P (since effectively we are doing SGD on shuffled datasets of size N, even if they are partitioned on portions of size N/P).

We can also guarantee ParMAC's convergence with only the original MAC theorem, without SGD-type conditions, while still in the distributed setting and achieving significant parallelism. This can be done by computing the gradient in the $\mathbf{W}$ step exactly (as MAC assumes). First, each machine $p = 1, \ldots, P$ computes the exact sum of per-point gradients for each submodel (by summing over its data portion), in parallel. Then, we aggregate these P partial gradients into one exact gradient, for each submodel. This could be done via a parameter server, or by having each machine act as the parameter server for one submodel, and could be easily implemented with MPI functions. However, as is well known, this is far slower than using SGD.

With nondifferentiable layers, the convergence properties of MAC (and ParMAC) are not well understood. In particular, for the binary autoencoder the encoding layer is discrete and the problem is NP-complete. But, again, the only modification of ParMAC over MAC is the fact that the encoder and decoder are trained with SGD in the $\mathbf{W}$ step, whose convergence tolerance can be achieved with SGD-type conditions. Indeed, our experiments show ParMAC gives almost identical results to MAC.

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

[5]Note that it is not necessary to assume that the points (or minibatches) are sampled at random during updates. Various results exist that guarantee convergence with deterministic errors (e.g. Bertsekas and Tsitsiklis, 2000 and references therein), rather than stochastic errors. These results assume a bound on the deterministic errors (rather than a bound on the variance of the stochastic errors), and apply to general, nonconvex objective functions with standard conditions (Lipschitz continuity, Robbins-Monro schedules for the step size, etc.). They apply as a particular case to the "incremental gradient" method, where we cycle through the data points in a fixed sequence.

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, one to two epochs in the **W** step make ParMAC very similar to MAC using an exact step.

7 Implementation of ParMAC for binary autoencoders

We have implemented ParMAC for binary autoencoders in C/C++ using the GNU Scientic Library (GSL) (`http://www.gnu.org/s/gsl`) and Basic Linear Algebra Subroutines (BLAS) library (`http://www.netlib.org`) for mathematical operations and linear algebra, and the Message Passing Interface (MPI) (Gropp et al., 1999a,b; Message Passing Interface Forum, 2012) for interprocess communication.

GSL and BLAS provide a wide range of mathematical routines such as basic matrix operations, various matrix decompositions and least-squares fitting. We used the versions of GSL and BLAS that come with our Linux distribution (Ubuntu 14.04). Considerably better performance could be achieved by using LAPACK and an optimised version of BLAS (such as ATLAS, or as provided by a computer vendor for their specific architecture).

MPI is one of the most widely used frameworks for high-performance parallel computing today, and is the best option for ParMAC because of its support for distributed-memory machines and SPMD (single program, multiple data) model, its language independence, and its availability in multiple machines, from small shared-memory multiprocessor machines to hybrid clusters. In MPI, different processes cannot directly access each other's memory space, but data can be transferred by sending messages from one process to another, or collectively among multiple processes. The SPMD model, very useful in distributed machine learning, means that all processes share the same code (and executable file), and each of them can operate on different data with flow control using its individual process id.

MPI is an industry standard for which there are many implementations, such as MPICH or OpenMPI, mostly compatible with each other. We used MPICH on our UC Merced shared-memory cluster and OpenMPI on the UCSD TSCC distributed cluster (see section 8.1). Our ParMAC C++ code compiles and runs with both implementations. We used the highest compiler optimisation level, specifically we ran `mpicc -O3 -lgsl -lgslcblas -lm`. This calls the GNU C compiler with option `-O3`, which turns on all the available code optimisation flags. It results in a longer compilation time but more efficient code.

The code snippet in figure 6 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[6], 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. Appendix B briefly describes important MPI functions and their arguments.

[6]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 (see appendix B). 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. 6; 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 the MPI execution environment
MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);                         // get the rank of the calling MPI process
MPI_Comm_size(MPI_COMM_WORLD, &mpisize);                         // get the total number of MPI processes
loadsettings();                                                  // load parameters (μ, epochs, dataset path, etc.)
loaddatasets();                                  // load input and output datasets and initial auxiliary coordinates
initializelayers();                                              // allocate memory and initialise f, h and Z steps
// we use MPI_Bsend to avoid managing send buffers, so we need to allocate the required
// amount of buffer space into which data can be copied until it is delivered
MPI_Pack_size(commbuffsize, MPI_CHAR, MPI_COMM_WORLD, &mpi_attach_buff_size);
// allocate enough memory so it can store the whole model
mpi_attach_buff = malloc(totalsubmodelcount*(mpi_attach_buff_size+MPI_BSEND_OVERHEAD));
MPI_Buffer_attach(mpi_attach_buff, mpi_attach_buff_size);                    // attach the allocated buffer

for (iter=1 to length(μ)) {                                                  // iterate over all the values of μ
  // begin W-step
  visitedsubmodels = 0;
  // each process visits all the submodels, epochs + 1 times
  while (visitedsubmodels <= totalsubmodelcount*epochs) {
    // stepcounter is a number that each submodel carries and increases by one in each step.
    // Once it reaches a certain value we stop sending the submodel around and it stops.
    // We reset stepcounter for all the submodels in the beginning of each W-step.
    if (stepcounter > 0) {          // if this is not the first submodel to train in the iteration, we wait to receive
      // MPI_Recv blocks until the requested data is available in the application buffer in the receiving task
      MPI_Recv(receivebuffer, commbuffsize, MPI_CHAR, MPI_ANY_SOURCE, MODEL_MSG_TAG,
                  MPI_COMM_WORLD, &recvStatus);
      savesubmodel(receivebuffer);                               // save the received buffer into a suitable struct
    }
    if (stepcounter < epochs*mpisize) {              // we don't train the submodels in the last update round
      switch(submodeltype)                                       // train each submodel according to its type
      case 'SVM': HtrainSGD();
      case 'linlayer':  FtrainSGD();
    }
    if (stepcounter < (ringepochs+1)*mpisize) {                  // we still need to send this submodel around
      // the lookup table is created randomly and stores the path of each submodel over epochs and iterations
      successor = next_in_lookuptable();                         // pick the successor process from the lookup table
      loadsubmodel(sendbuffer);                        // load the submodel from its struct into the send buffer
      // MPI_Bsend returns after the data has been copied from application buffer space to the allocated send buffer
      MPI_Bsend(sendbuffer, taskbufsize*sizeof(double), MPI_CHAR, successor, MODEL_MSG_TAG,
                  MPI_COMM_WORLD);
    }
    visitedsubmodels++;
  }
  // end W-step

  // begin Z-step
  updateZ_relaxed();                    // initialise auxiliary coordinates based on a truncated, relaxed solution
  updateZ_alternate();                    // update auxiliary coordinates by alternating optimisation over bits
  // end Z-step
}

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

Figure 6: Binary autoencoder ParMAC algorithm (fragment), showing important MPI calls.

Table 1: Detailed hardware specification of the two machines used in our experiments.

	Distributed-memory (TSCC at UCSD)	Shared-memory (cluster at UC Merced)
CPU	Intel(R) Xeon(R) CPU E5-2670 0	Intel(R) Xeon(R) CPU E5-2699 v3
CPU cache	20 MB	45 MB
CPU max frequency	3.3 GHz	3.6 GHz
Cores/threads	8/16	8/16
Memory types	DDR3 800/1066/1333/1600	DDR4 1600/1866/2133
RAM bandwidth	51.2 GB/s	68 GB/s
Processor connection	10GbE	shared memory

8 Experiments

8.1 Setup

Computing systems We used two different computing systems, to which we will refer as *distributed* and *shared-memory*:

Distributed-memory This 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 DRAM (4GB/core) and a 500GB hard drive. The nodes are connected through a 10GbE network. We used up to $P = 128$ processors. Detailed specs are in table 1 (obtained by running `dmidecode` in the actual processor) and `http://idi.ucsd.edu/computing`.

Shared-memory This is a 72-processor machine (36 physical cores with hyperthreading) with 256GB RAM located at UC Merced. The processors communicate through shared memory. We used this only for the large-scale experiment, and we used 64 of the 72 processors. Detailed specs are in table 1 (obtained by running `dmidecode` in the actual processor).

In both systems, the interprocess communication is handled by MPI (OpenMPI on the TSCC cluster and MPICH on our shared-memory cluster). The shared-memory system has both faster processors and faster communication than the distributed one and this is seen in our experiments (3–4 times faster). This does not imply that shared-memory systems are necessarily superior in practice, it simply reflects characteristics of the equipment we had access to. The ParMAC speedups as a function of the number of processors are comparable in both systems.

Datasets We have used 4 datasets commonly used as image retrieval benchmarks. (1) CIFAR (Krizhevsky, 2009) contains 60 000 32×32 colour images in 10 object classes. We ignore the labels in this paper and use $N = 50\,000$ images as training set and 10 000 as test set. We extract $D = 320$ GIST features (Oliva and Torralba, 2001) from each image. (2) SIFT-10K (Jégou et al., 2011a) contains $N = 10\,000$ training high-resolution colour images and 100 test images, each represented by $D = 128$ SIFT features. (3) SIFT-1M (Jégou et al., 2011a) contains $N = 10^6$ training and 10^4 test images. (3) SIFT-1B (Jégou et al., 2011b; `http://corpus-texmex.irisa.fr`) 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 following. (1) The binary autoencoder error $E_{\text{BA}}(\mathbf{h}, \mathbf{f})$ which we want to minimise, eq. (1). (2) The quadratic-penalty function $E_Q(\mathbf{h}, \mathbf{f}, \mathbf{Z}; \mu)$ which we actually minimise for each value of μ, eq. (3). (3) 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, $(100, 100)$ for SIFT-10K 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.

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, SIFT-10K and SIFT-1M and $L = 64$ bits for SIFT-1B. The $\mathbf{Z}$ step uses enumeration for SIFT-10K and SIFT-1M, and alternating optimisation (initialised by a truncated relaxed solution) otherwise. We initialise the binary codes from truncated PCA ran on a subset of the training set (small enough that it fits in one machine).

To train the encoder (L SVMs) and decoder (D linear mappings) with stochastic optimisation, we used the SGD code from Bottou and Bousquet (2008) (`http://leon.bottou.org/projects/sgd`), 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-10K and SIFT-1M we use $\mu_0 = 10^{-6}$ and $a = 2$ over 20 iterations. For SIFT-1B we use $\mu_0 = 10^{-4}$ and $a = 2$ over 10 iterations.

8.2 Effect of stochastic steps in the W step

Figures 7 to 9 show the effect on SIFT-10K and CIFAR of varying the number of epochs and shuffling the data as a function of the number of machines P on the learning curves (errors E_Q and E_{BA}, precision). As the number of epochs increases, the $\mathbf{W}$ step is solved more exactly (8 epochs is practically exact in these datasets). 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 $\mathbf{W}$ step has less parallelism.

Fig. 9 shows the positive effect of data shuffling in the $\mathbf{W}$ step. To shuffle the minibatches, the successor of a machine is given by a random lookup table. Shuffling generally reduces the error (this is particularly clear in E_Q, which is what we actually minimise) and increases the precision with no increase in runtime.

Note that, even if we keep the circular topology fixed throughout the $\mathbf{W}$ step (i.e., we do not randomise the topology at each epoch), there is still a small amount of shuffling. This occurs because, although all submodels process the data minibatches in the same "direction", submodels in different machines start at different minibatches. Were it not for this (and if no shuffling was done within-machine), ParMAC would give an identical result no matter the number of machines. However, as seen from figures 7 to 9, it seems that this small intrinsic shuffling simply randomises the learning curves, but does not make them better than the learning curve for one machine.

8.3 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. 10 (top row) shows the strong scaling[7] speedups achieved experimentally, as a function of the number of machines P for fixed problem size (dataset and model), in CIFAR and SIFT-1M ($N = 50$K and 1M training points, respectively). Even though these datasets and especially the number of independent submodels ($M = 2L = 32$ effective submodels of the same size, as discussed in section 5.4) 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 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

[7] In "strong scaling", the total problem size is fixed and the problem size on each machine is inversely proportional to the number of machines P. In "weak scaling", the problem size on each machine is fixed, so the total problem size is proportional to P. High speedups are easier to obtain in weak scaling than in strong scaling (Goedecke and Hoisie, 2001).

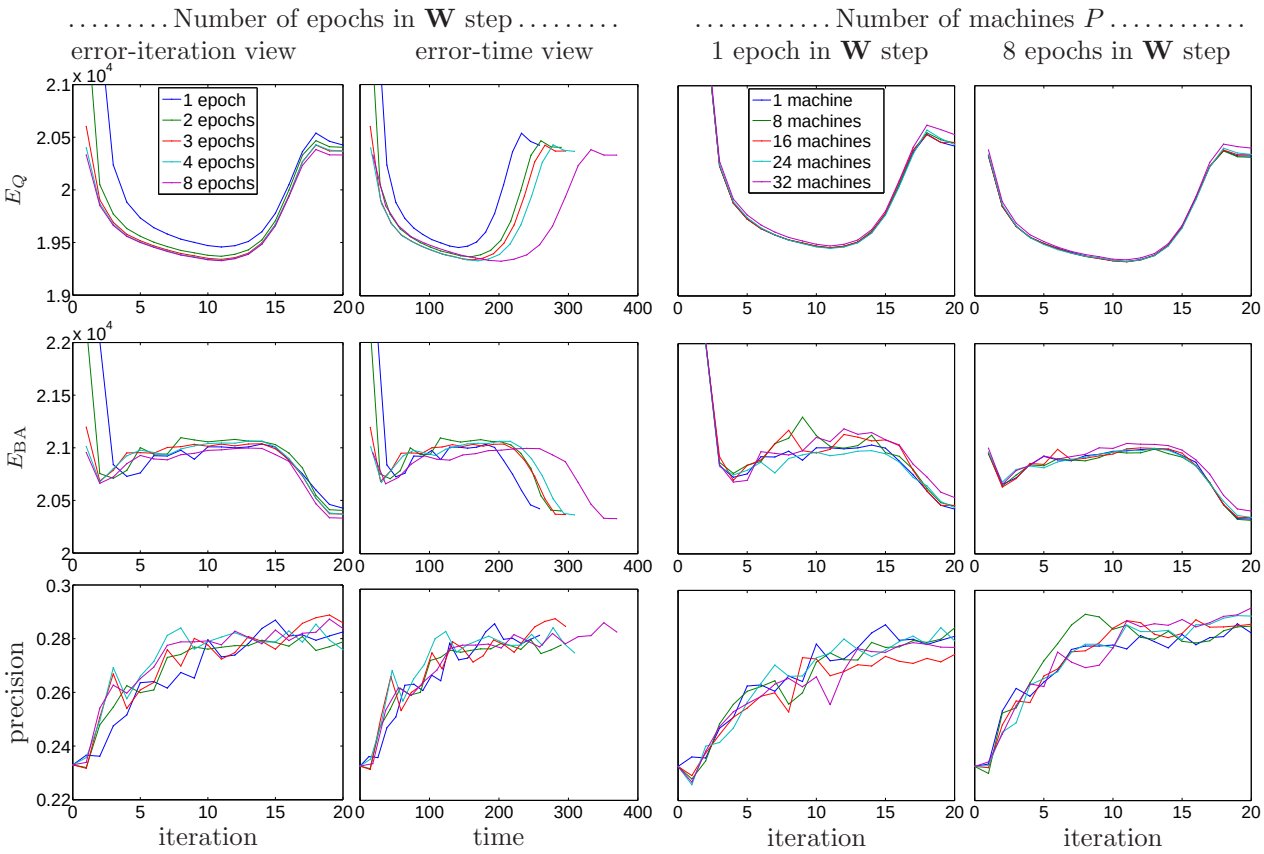

Figure 7: SIFT-10K dataset. *Left two columns*: single machine ($P = 1$) and different number of epochs e in the $\mathbf{W}$ step. *Right two columns*: fixed number of epochs (either 1 or 8) but different number of machines P.

runtime for SIFT-1M on $P = 128$ machines with 8 epochs was 12 minutes and its speedup 100×. This is particularly remarkable given that the original, nested model did not have model parallelism. These speedups are vastly larger than those achieved by earlier large-scale nonconvex optimisation approaches such as Google's DistBelief system (Le et al., 2012; Dean et al., 2012), although admittedly the deep nets trained there were far larger than our BAs.

Fig. 10 (bottom) shows the speedups predicted by our theoretical model of section 5. 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 (effective number of independent equal-size submodels). 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. Specifically, we set $t_c^{\mathbf{W}} = 10^4$ 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 the $\mathbf{Z}$ step is quite slower than the $\mathbf{W}$ step because of the binary optimisation it involves.

Fig. 10 (bottom, right plot) also shows the theoretical prediction for the SIFT-1B dataset ($N = 10^8$, $M = 128$), using the same parameters as in SIFT-1M (again assuming the distributed memory system): $t_c^{\mathbf{W}} = 10^4$ and $t_r^{\mathbf{Z}} = 40$. Since here M is quite larger and N is much larger, the speedup is nearly perfect over a very wide range (note the plot goes up to $P = 1\,024$ machines, even though our experiments are limited to $P = 128$).

8.4 Large-scale experiment: SIFT-1B dataset

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

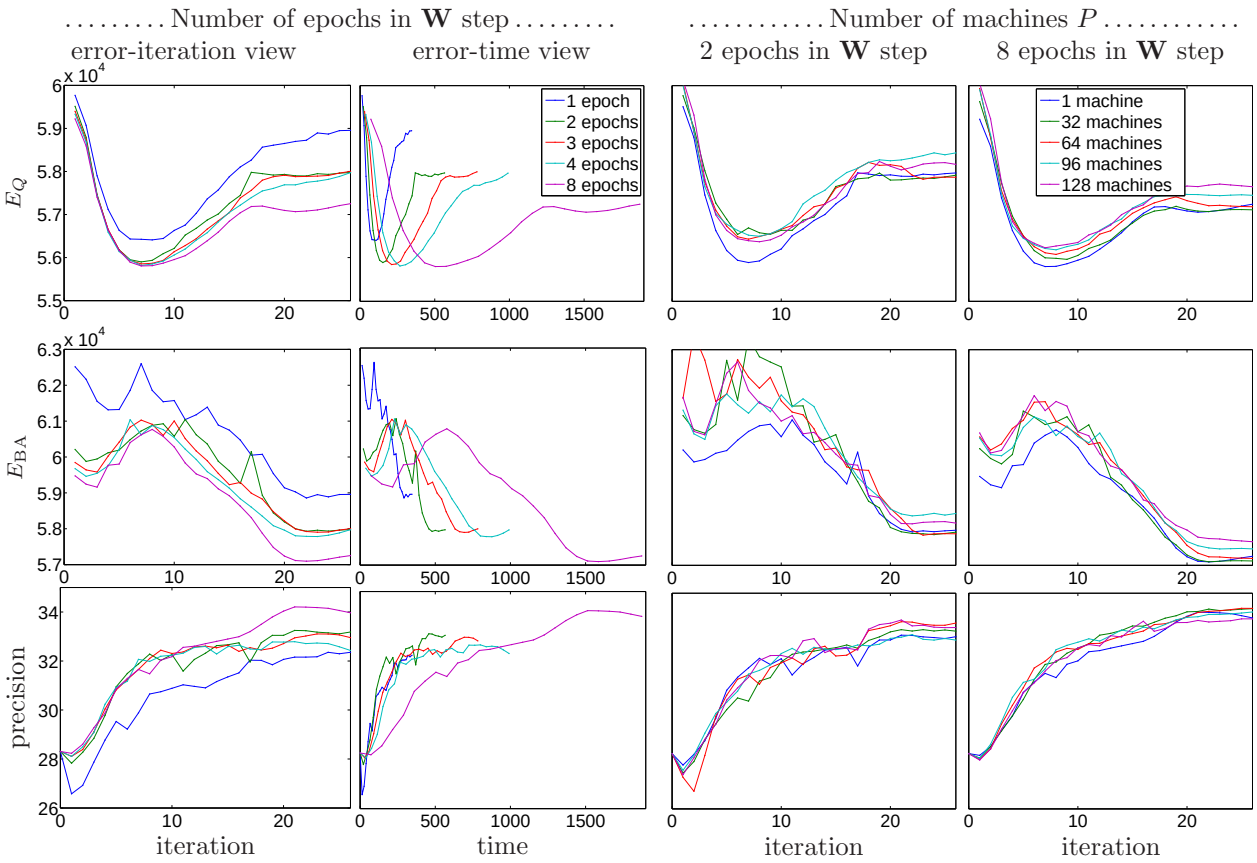

Figure 8: CIFAR dataset. *Left two columns*: single machine ($P = 1$) and different number of epochs e in the **W** step. *Right two columns*: fixed number of epochs (either 2 or 8) but different number of machines P.

for each query to its k nearest vectors in the base set). The training set contains $N = 100$M vectors, each consisting of 128 SIFT features.

Handling the SIFT-1B dataset required special care because of its size and the limited amount of memory (total 512GB for 128 processors in the distributed system and 256GB for 64 processors in the shared-memory one). Each vector has 128 SIFT features and each feature in the original dataset is stored in a single byte rather than as double-precision floats (8 bytes), as in our other experiments, totalling 12.8GB for the training set if using a linear hash function and 200GB if using an RBF one (see below). Rather than converting it to floats, which would exceed 1TB, we modified our code to convert each feature only as needed. In the **Z** step each datapoint is processed independently and the conversion to double is done one point at a time. In the **W** step it is done one minibatch at a time. It is of course possible to use hard disk as additional storage but this would slow down training. The auxiliary coordinates, which must be stored in MAC algorithms, take only 6.25% the memory of the data (64 bits per datapoint compared to 128 bytes).

We used $L = 64$ bits (hash functions). As hash function, we trained a linear SVM as before, and a kernel SVM using m Gaussian radial basis functions (RBF) with fixed bandwidth σ and centres. This means the only trainable parameters are the weights, so the MAC algorithm does not change except that it operates on an m-dimensional input vector of kernel values (stored as one byte each), instead of the 128 SIFT features. The Gaussian kernel values are in $(0, 1]$ but, as before, to save memory we store them as an unsigned one-byte integer value in $[0, 255]$. We used $m = 2\,000$ centres, the maximum we could fit in memory, picked at random from the training set. In trials with a subset of the training set, we set $\sigma = 160$. This worked well and was wide enough to ensure that, with our limited one-byte precision, no data point would produce m zeros as kernel values.

On trials on a subset of the training dataset, we set the number of epochs to $e = 2$ with shuffling (we observed no improvements by using more epochs, which is understandable given the size of the dataset).

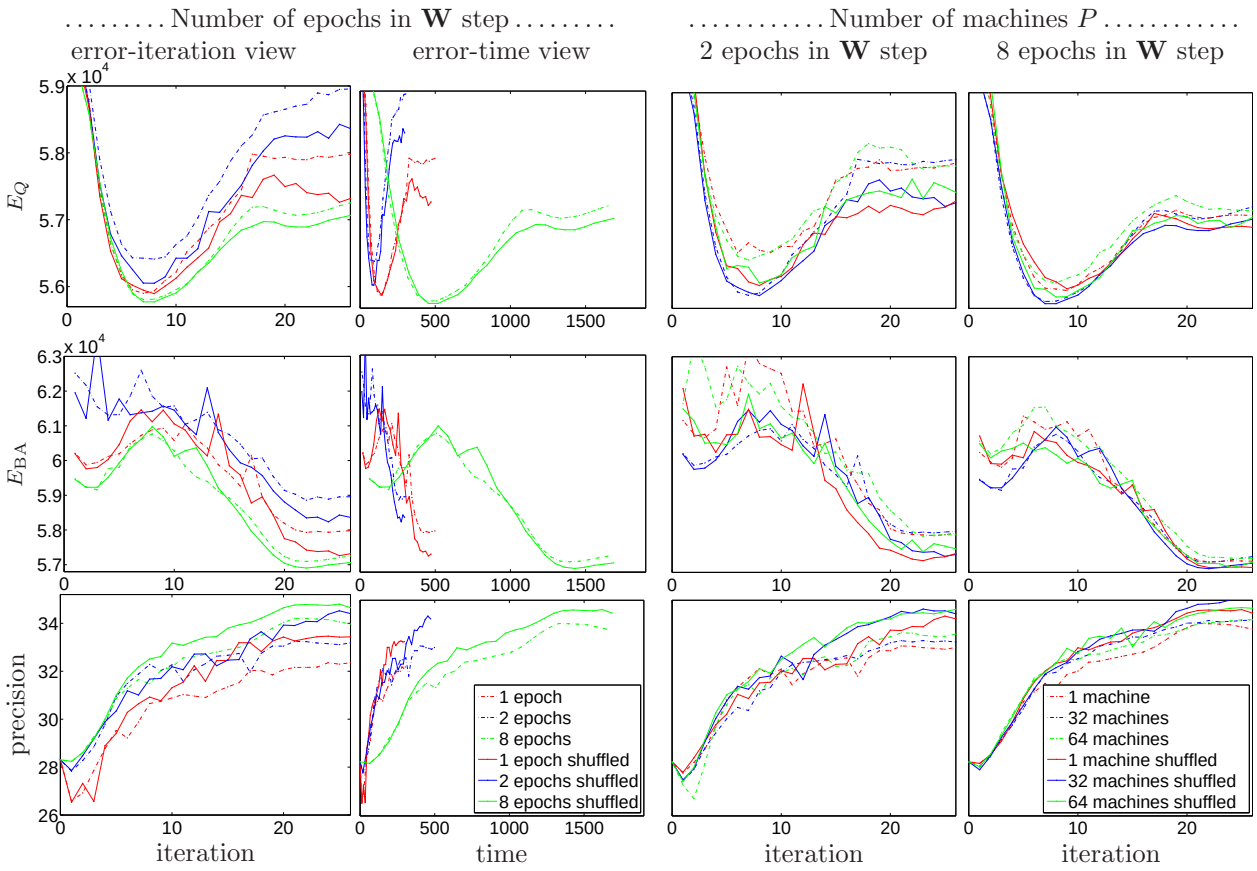

Figure 9: Like fig. 8 but with and without minibatch shuffling in the **W** step (solid and dashed lines, resp.).

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) if using 8 bytes per indexed vector (without postprocessing by reranking as done in that paper).

We ran ParMAC on the whole training set in the distributed system with 128 processors for 6 iterations and in the shared-memory one with 64 processors for 10 iterations. The results are given in the following table and figures 11–12:

Hash function (encoder)	Recall @R=100	Time (hours)	
		distrib.	shared
linear SVM	61.5%	29.30	11.04
kernel SVM	66.1%	83.44	32.19

The learning curves (fig. 11) are essentially identical over both systems. The nonlinear RBF hash function outperforms the linear one in recall, as one would expect. The improvement occurs across the whole range of R recall values (fig. 12). Note the error in the nested model, E_{BA}, does not decrease monotonically. This is because MAC optimises instead the penalised function E_Q, in an effort the minimise E_{BA} as μ increases.

Based on our previous results, the small number of epochs and the larger number of submodels in the **W** step, we expect nearly perfect speedups. We cannot compute the actual speedup because the single-machine runtime is enormous. 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.

Although the speedups are comparable on both the distributed and the shared-memory system, the former is 3–4 times slower. The reason is the distributed system has both slower processors and slower interprocessor communication (across a network); see also fig. 13.

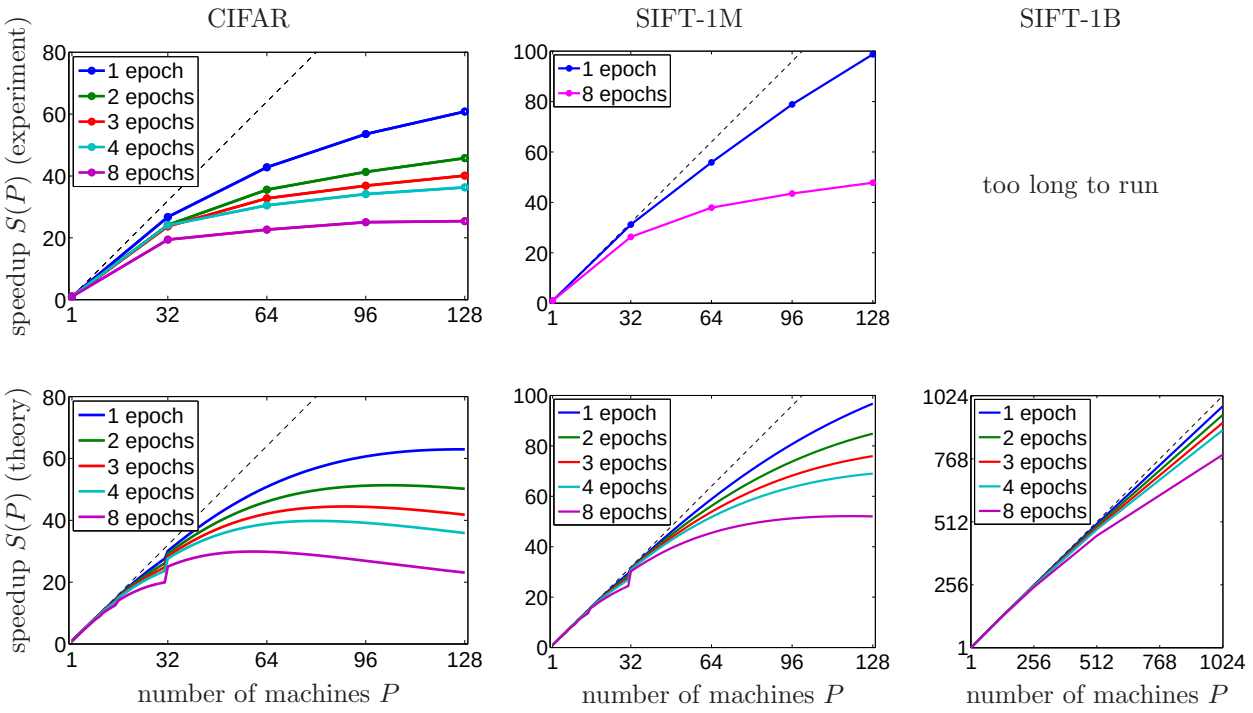

Figure 10: Speedup $S(P)$ as a function of the number of machines P for CIFAR, SIFT-1M and SIFT-1B. Top: experimental result in the distributed memory system. Bottom: theoretical result predicted by the model of section 5. The dataset size and number of submodels (N, M) is $(50\,000, 32)$ for CIFAR, $(10^6, 32)$ for SIFT-1M and $(10^8, 128)$ for SIFT-1B.

8.5 Shared-memory vs distributed systems

The TSCC distributed cluster consists of nodes containing 16 processors and 64GB RAM. These processors communicate through shared-memory within a node, and across a network otherwise (which is slower). In all our experiments up to now, we always allocated processors within the same node if possible. But, depending on whether a user requests processors within or across nodes, there is then a tradeoff between both communication modes. A full study of this issue is beyond our scope, which is to understand the ParMAC algorithm in general rather than for specific computer architectures. However, we ran a small experiment to evaluate the computation and communication time spent as a function of the number of processors per node. We set the total number of processors to $P = 16$ but allocated them in the following configurations: from a single node with all 16 processors (1×16, pure shared-memory) to 16 nodes each with 1 processor (16×1, pure distributed), and intermediate configurations such as 2 nodes each with 8 processors (2×8). We used the RBF hash function from the SIFT-1B experiment with all settings as before and trained it on a subset of 20K points for a single iteration. Figure 13 shows the resulting times. While the computation time remains constant, the communication time increases as we move from shared-memory to distributed settings, as expected. Hence, the effect on the ParMAC algorithm would be to increase the **W** step runtime correspondingly and lower the parallel speedup.

9 Discussion

Developing parallel, distributed optimisation algorithms for nonconvex problems in machine learning is challenging, as shown by recent efforts by large teams of researchers (Le et al., 2012; Dean et al., 2012). 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 is no close coupling

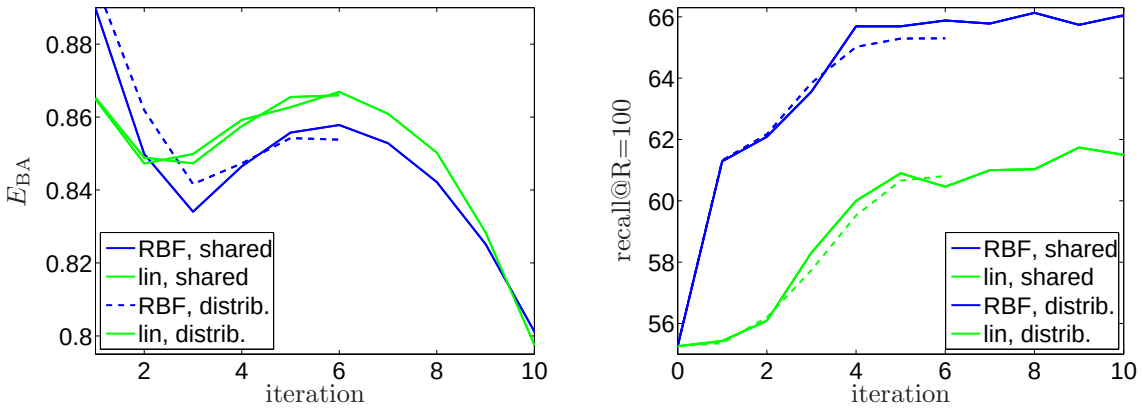

Figure 11: SIFT-1B dataset, using the shared-memory and distributed system (solid and dashed lines, resp.).

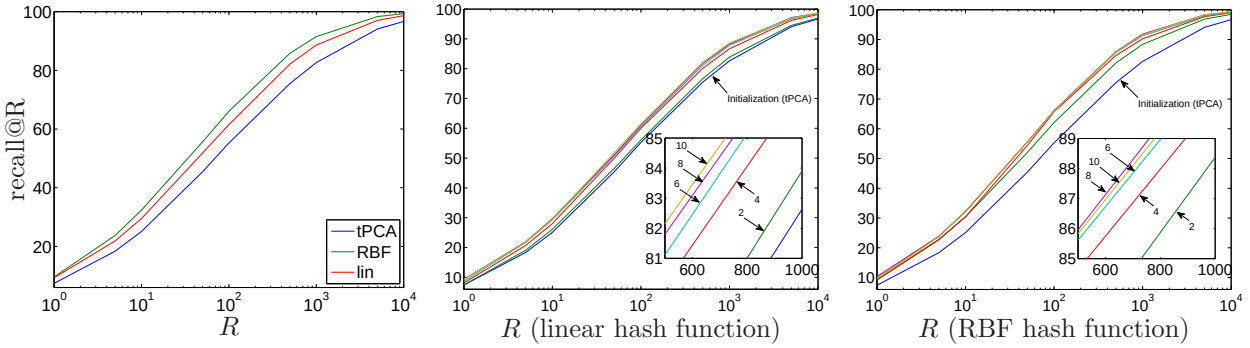

Figure 12: Recall@R on the SIFT-1B dataset for truncated PCA (initialisation), linear and kernel hash functions (left plot: final result; right two plots: over iterations, as labelled).

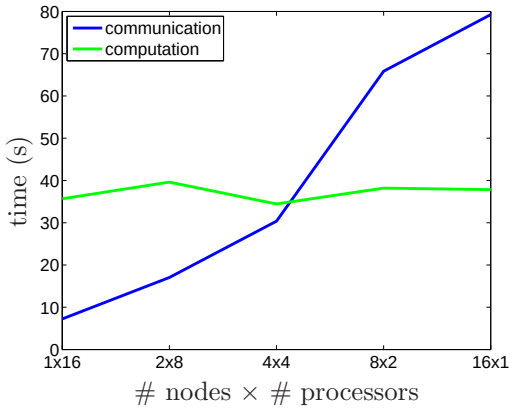

Figure 13: Time spent on communication and computation as a function of the number of processors per node in the TSCC cluster. The time for our shared-memory UC Merced cluster (also using 16 processors, i.e., corresponding to 1×16) is 2.57 and 8.76 seconds for communication and computation, respectively.

between the model structure and the distributed system architecture. The development and implementation of ParMAC for binary autoencoders on large datasets in a distributed cluster was achieved in a few months by the PI and one junior PhD student (both without prior experience in MPI).

Rather than an algorithm, MAC is a meta-algorithm that can produce a specific optimisation algorithm for a given nested problem, depending on how the auxiliary coordinates are introduced and on how the resulting subproblems are solved (in this sense, MAC is similar to expectation-maximisation (EM) algorithms; Dempster et al., 1977). For example, in the low-dimensional SVMs of Wang and Carreira-Perpiñán (2014), the **Z** step is a small quadratic program for each data point. However, regardless of these specifics, the resulting MAC algorithm typically exhibits a **W** step with M independent submodels and a **Z** step with N independent coordinates for the data points. Likewise, the specifics of a ParMAC algorithm (how the **W** and **Z** steps are optimised) will depend on its corresponding MAC algorithm. However, it will always split the data and auxiliary coordinates over machines and consist of a **Z** step with no communication between machines, and a **W** step where submodels visit machines in a circular topology, effectively training themselves by stochastic optimisation.

Further improvements can be made in specific problems. For example, it is possible to have further 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 that it needs, or by allocating cores within a multicore machine accordingly. Also, the **W** and **Z** step optimisations can make use of further parallelisation by GPUs or by distributed convex optimisation algorithms. And, if the submodels are small in size, it may be better for each machine to operate on a set of submodels and then send them all together in a larger message (rather than sending each submodel as it is finished), since this will reduce latency overheads (i.e., the setup cost of a message). Many more refinements can (and should) be done in an industrial implementation. For example, one can store and communicate reduced-precision values for data and parameters with little effect of the accuracy, as has been done in neural nets (e.g. Gupta et al., 2015; Han et al., 2015, 2016). Various system-dependent optimisations may be possible (beyond those that a good compiler may be able to do, such as loop unrolling or code inlining), such as improving the spatial or temporal locality of the code given the type and size of the cache in each machine. In this paper, we have tried to keep 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.

ParMAC is very efficient in communication: no data or coordinates are ever sent, only the entire model $e+1$ times per iteration. Using one epoch (which is sufficient in large datasets), or using e epochs but performing them within each machine, the entire model is sent twice per iteration. This is near optimal if we note the following. If the data cannot be communicated, then at every iteration each submodel must visit each machine (for it to be trained on the entire data). Hence, any correct algorithm will have to send the entire model at least once; ParMAC does so twice. Also, the circular topology is the minimal topology (considered as a directed graph on the P machines) that is necessary to be able to optimise a global model on the entire dataset with P machines, because each machine must be able to communicate with some other machine. It has P edges and is truly distributed, with each machine having the same importance.

A popular model for distributed optimisation (e.g. with parallel SGD) uses a worker-server connectivity: $W \gg 1$ workers that do actual parameter optimisation and $S \geq 1$ "parameter servers" that collect and broadcast parameters to worker machines. This is a bipartite graph with bidirectional edges between servers and workers, having SW edges, which is quite larger than the number of machines $P = S + W$. The entire model must be sent twice per iteration, to and from the parameter server, but this creates a bottleneck when multiple workers send data to the same server, and $S \ll W$ in practice. No such bottleneck occurs in ParMAC.

Parallel and distributed computing systems have been around for decades. One important class are supercomputers, which are carefully designed in terms of the processors, memory system and connection network. They have been traditionally used to solve a wide variety of large-scale scientific computation problems, such as weather prediction, nuclear reactor modelling, or astrophysical or molecular simulations. Another important class are clusters of inexpensive, heterogenous workstations connected through an Ethernet network, with workstations differing in speed, memory/disk capacity, number of cores/GPUs, etc. This is used in data centres in Google, Amazon and other companies, and also in distributed computation models

such as SETI@home that capitalise on the computation and Internet connectivity available to individuals, and their willingness to donate them to projects they find worthy. In these systems, the machine learning task may be one of other tasks running concurrently, such as web searches or email in a data centre (which may operate on the same data as the machine learning task), or personal applications in an individual's workstation. Supercomputers and clusters differ considerably across important factors: suitability for a particular problem, computation and communication speed, size of memory and disk, connection network, fault tolerance, load, cost, energy consumption, etc. At present it is unclear what the best choices will be for machine learning models (which exhibit a wide variety themselves), and we expect to see many different possibilities been researched in the immediate future. We suggest that ParMAC, by itself or in combination with other techniques, may play an important role with nested models because of the embarrassing parallelism it introduces and its loose demands on the underlying distributed system.

10 Conclusion

We have proposed ParMAC, a distributed model for the method of auxiliary coordinates for training nested, nonconvex 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 (weight vectors of a hash function or hidden unit) 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. This keeps the communication between machines to a minimum within each iteration. In this sense, ParMAC can be seen as a strategy to be able to use existing, well-developed (convex) distributed optimisation techniques—applicable to simple functions—to a setting where simple functions are coupled by nesting into a nonconvex function whose training data is distributed over machines. The convergence properties of MAC (to a stationary point of the objective function) 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.

Acknowledgements

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

A Theoretical analysis of the speedup: proofs

In section 5 we proposed the following theoretical estimate for the speedup $S(P)$:

$$S(P) = \frac{T(1)}{T(P)} = \frac{\rho \frac{1}{\lceil M/P \rceil} MP}{\frac{1}{N}P^2 + \rho_2 P + \rho_1 \frac{1}{\lceil M/P \rceil} M}. \tag{12'}$$

Consider $S(P)$ as a real function of a real variable $P \geq 1$ (keeping in mind that only integer values of P can occur in practice). The function $\lceil M/P \rceil$ is piecewise constant and takes the values $M, M-1, \ldots, 1$ as P increases from $P = 1$, with discontinuities where $M/P = k$ for $k = M, M-1, \ldots, 1$. Hence, $S(P)$ is piecewise continuous on M intervals of the form $\big[1, \frac{M}{M-1}\big), \big[\frac{M}{M-1}, \frac{M}{M-2}\big), \ldots, \big[\frac{M}{2}, M\big), [M, \infty)$. (Many of these intervals occur between integer values of P so they are actually unobserved in practice; for example, for $M = 16$ there are 8 intervals between $P = 1$ and $P = 2$.) Within each interval $P \in \big[\frac{M}{k}, \frac{M}{k-1}\big)$ we have $\lceil M/P \rceil = k$, hence we can equivalently write the speedup of eq. (12) as the following rational function of P:

$$S(P) = \frac{\frac{1}{k}\rho MP}{\frac{1}{N}P^2 + \rho_2 P + \frac{1}{k}\rho_1 M} \quad \text{for } P \in \big[\tfrac{M}{k}, \tfrac{M}{k-1}\big), \qquad k = M, M-1, \ldots, 1. \tag{23}$$

A.1 Characterisation of the speedup function $S(P)$

Our main theorem is theorem A.1 below. It characterises how the speedup grows as a function of P. Let us define

$$P_k^* = \sqrt{\rho_1 MN/k} \qquad S_k^* = S(P_k^*) = \frac{\rho M/k}{\rho_2 + 2\sqrt{\rho_1 M/Nk}} \qquad k = 1, 2, \ldots, M. \tag{17'}$$

Theorem A.1. *Consider the function $S(P)$ of eq. (23) and P_k^* and S_k^* as in eq. (17'). Then:*

1. *$S_k^* < S_{k-1}^*$ for $k = 2, \ldots, M$.*
2. *Within interval $\big[\frac{M}{k}, \frac{M}{k-1}\big)$ for $k = 1, 2, \ldots, M$, we have that $S(P)$ either is monotonically increasing, or is monotonically decreasing, or achieves a single maximum $S_k^* = S(P_k^*)$ at P_k^*.*
3. *$S\big(\frac{M}{k}\big) > S(P)$ for $1 \leq P < \frac{M}{k}$, for $k = 2, \ldots, M$.*

Proof. Part 1 is obvious by writing

$$S_k^* = \frac{\rho M/k}{\rho_2 + 2\sqrt{\rho_1 M/Nk}} = \frac{\rho M}{\rho_2 k + 2\sqrt{\rho_1 kM/N}}.$$

To prove part 2, we apply lemma A.4 to $S(P)$ within interval $\big[\frac{M}{k}, \frac{M}{k-1}\big)$ for $k = 1, 2, \ldots, M$. We obtain that $S(P)$ either is monotonically increasing, or is monotonically decreasing, or achieves a single maximum $S_k^* = S(P_k^*)$ at P_k^*.

To prove part 3, we apply theorem A.3 repeatedly for $k = 2, \ldots, M$. □

Remark A.2. As a particular case of theorem A.1 part 2 for $k = 1$, we obtain that for $P \in [M, \infty)$

$$P_1^* = \sqrt{\rho_1 MN} \qquad S_1^* = S(P_1^*) = \frac{\rho M}{\rho_2 + 2\sqrt{\rho_1 M/N}} \tag{19'}$$

and $S(P)$ is either monotonically decreasing with P if $M \geq P_1^*$, or it increases from $P = M$ up to a single maximum at $P = P_1^*$ and then decreases monotonically. In both cases, if $t_c^{\mathbf{W}} > 0$ we have that $S(P) \to 0$ as $P \to \infty$, and $S(P) \approx \rho NM/P$ for large P.

Theorem A.3. *Consider the function of eq. (23), written more simply using $\rho_1' = \rho_1 N > 0$, $\rho_2' = \rho_2 N > 0$ and $\rho' = \rho N = \rho_1' + \rho_2' > 0$:*

$$S(P) = \frac{\frac{1}{k}\rho' MP}{P^2 + \rho_2' P + \frac{1}{k}\rho_1' M} \quad \text{for } P \in \big[\tfrac{M}{k}, \tfrac{M}{k-1}\big), \qquad k = 2, \ldots, M. \tag{24}$$

Then, for $k = 2, \ldots, M$: $S\big(\frac{M}{k-1}\big) > S(P)$ $\forall P \in \big[\frac{M}{k}, \frac{M}{k-1}\big)$.

Proof. From theorem A.1 part 2, we know that exactly one of the following three cases holds for $P \in \left[\frac{M}{k}, \frac{M}{k-1}\right)$:

1. $S(P)$ is monotonically increasing. Then, it suffices to prove that $\lim_{P\to\frac{M}{k-1}} S(P) < S\left(\frac{M}{k-1}\right)$. Indeed,
$$\lim_{P\to\frac{M}{k-1}} S(P) = \frac{\rho' M}{\rho'(k-1)+\rho'_2+Mk/(k-1)} < \frac{\rho' M}{\rho'(k-1)+M} = S\left(\frac{M}{k-1}\right).$$
2. $S(P)$ is monotonically decreasing. Then, it suffices to prove that $S\left(\frac{M}{k}\right) < S\left(\frac{M}{k-1}\right)$. Indeed,
$$S\left(\frac{M}{k}\right) = \frac{\rho' M}{\rho' k+M} < \frac{\rho' M}{\rho'(k-1)+M} = S\left(\frac{M}{k-1}\right).$$
3. $S(P)$ achieves a single maximum $S_k^* = S(P_k^*)$ at P_k^* in the interior of the interval. Then, it suffices to prove that $S_k^* < S\left(\frac{M}{k-1}\right)$. This last case is more complicated. In the sequel, we provide a proof that has not technical difficulties, although it is somewhat cumbersome.

Let us then prove that $S_k^* < S\left(\frac{M}{k-1}\right)$. After a bit of algebra, from eqs. (24) and (17') we obtain the following:
$$S_k^* < S\left(\frac{M}{k-1}\right) \Leftrightarrow M+\rho'_1 k-\rho' < 2kP_k^* = 2k\sqrt{\rho'_1 M/k}.$$
If $M+\rho'_1 k-\rho' \le 0$, then the condition holds and the proof is done. Otherwise, assume $M+\rho'_1 k-\rho' > 0$ and take squares in the previous equation:
$$(M+\rho'_1 k-\rho')^2 < 4\rho'_1 kM \Leftrightarrow M^2+(\rho'_1 k-\rho')^2-2(\rho'_1 k+\rho')M < 0$$
$$\Leftrightarrow \rho'_1 k+\rho'-2\sqrt{\rho'_1\rho' k} < M < \rho'_1 k+\rho'+2\sqrt{\rho'_1\rho' k}.$$
Hence, we need to prove that the following inequalities hold:
$$\rho'_1 k+\rho'-2\sqrt{\rho'_1\rho' k} < M < \rho'_1 k+\rho'+2\sqrt{\rho'_1\rho' k} \tag{25}$$
under the following assumptions:

- $P_k^* = \sqrt{\rho'_1 M/k} \in \left(\frac{M}{k}, \frac{M}{k-1}\right)$, since case 3 above means that $S(P)$ achieves a maximum S_k^* at P_k^* in the interior of the interval. Equivalently, $\rho'_1 k > M > \rho'_1(k-1)^2/k = \rho'_1\left(k-2+\frac{1}{k}\right)$.
- $M \ge k \ge 2$, since k takes the values $2,3,\ldots,M$.
- $M > \rho'-\rho'_1 k$, from above.

From $M < \rho'_1 k$ it follows that $M < \rho'_1 k+\rho'+2\sqrt{\rho'_1\rho' k}$, and the RHS inequality in (25) is proven. Now let us prove the LHS inequality in (25), which states $M > \rho'_1 k+\rho'-2\sqrt{\rho'_1\rho' k}$. This can be derived from the assumption above that $M > \rho'_1\left(k-2+\frac{1}{k}\right)$. Specifically, we will prove that $\rho'_1\left(2-\frac{1}{k}\right) < -\rho'+2\sqrt{\rho'_1\rho' k}$, and this will complete the proof of the theorem.

From assumptions $\rho'_1 k > M > \rho'-\rho'_1 k$ we get that $\rho' < 2\rho'_1 k$, and since $\rho' \ge \rho'_1$, we have that $\rho' \in [\rho'_1, 2\rho'_1 k)$. Now, write $\rho' = a^2\rho'_1$ with $a \in [1, \sqrt{2k})$. Then
$$\rho'_1\left(2-\frac{1}{k}\right) < -\rho'+2\sqrt{\rho'_1\rho' k} \Leftrightarrow a^2+2-\frac{1}{k} < 2a\sqrt{k}.$$
Since $k \ge 2 \Leftrightarrow \frac{3}{2} \ge 2-\frac{1}{k}$, to prove $a^2+2-\frac{1}{k} < 2a\sqrt{k}$ it suffices to prove that $a^2+\frac{3}{2} < 2a\sqrt{k}$, or equivalently $a \in \left(\sqrt{k}-\sqrt{k-\frac{3}{2}}, \sqrt{k}+\sqrt{k-\frac{3}{2}}\right)$. This interval indeed contains $[1,\sqrt{2k})$: a little algebra shows that
$$\sqrt{k}-\sqrt{k-\frac{3}{2}} < 1 \Leftrightarrow k > \left(\frac{5}{4}\right)^2 \qquad \text{and} \qquad \sqrt{k}+\sqrt{k-\frac{3}{2}} > \sqrt{2k} \Leftrightarrow k > \frac{3}{4(\sqrt{2}-1)}$$
both of which hold because $k \ge 2$. ∎

Lemma A.4. *Given constants $\alpha, \beta, \gamma, \delta > 0$, define the following real function for $P \geq 0$:*

$$\psi(P) = \frac{\delta P}{\alpha P^2 + \beta P + \gamma} \tag{26}$$

and let $P^* = \sqrt{\gamma/\alpha}$ and $\psi^* = \psi(P^*) = \delta/(\beta + 2\sqrt{\alpha\gamma})$. Then, in the interval $[a, b)$ with $1 \leq a < b \leq \infty$,

ψ *is monotonically decreasing if $P^* \leq a$*
achieves a single maximum at P^* if $P^* \in (a, b)$
is monotonically increasing if $P^* \geq b$.

Proof. The derivatives of ψ with respect to P are:

$$\psi'(P) = \frac{\delta(-\alpha P^2 + \gamma)}{(\alpha P^2 + \beta P + \gamma)^2} \qquad \psi''(P) = \frac{-2\delta(-\alpha^2 P^3 + 3\alpha\gamma P + \beta\gamma)}{(\alpha P^2 + \beta P + \gamma)^3}.$$

Hence $\psi'(P^*) = 0$ at $P^* = \sqrt{\gamma/\alpha}$ and $\psi''(P^*) < 0$, and the lemma follows. □

A.2 Globally maximum speedup $S^* = \max_{P \geq 1} S(P)$

The maximum speedup can be determined as follows. Both $S(P)$ in (14) and S_k^* in (17) are monotonically increasing with P (note S_k^* is decreasing with k, and k is decreasing with P). Hence, the global maximum of $S(P)$ occurs in the last interval $[M, \infty)$, either at the beginning ($P = M$, if $P_1^* \leq M$) or in its interior ($P = P_1^*$, if $P_1^* > M$). This also follows from theorem A.1. Specifically, the global maximum S^* of $S(P)$ is:

- If $M \geq \rho_1 N$: $S^* = M/\left(1 + \frac{M}{\rho N}\right) \leq M$, achieved at $P = M$.
- If $M < \rho_1 N$: $S^* = S_1^* = \frac{\rho M}{\rho_2 + 2\sqrt{\rho_1 M/N}} > M$, achieved at $P = P_1^* = \sqrt{\rho_1 M N} > M$.
  Let us prove that $S_1^* > M$. Assume $S_1^* \leq M$, then

  $$\frac{\rho M}{\rho_2 + 2\sqrt{\rho_1 M/N}} \leq M \Leftrightarrow \rho \leq \rho_2 + 2\sqrt{\rho_1 M/N} \Leftrightarrow \rho_1 \leq 4M/N \Leftrightarrow M \geq \rho_1 N/4$$

  which contradicts the condition that $M < \rho_1 N$, hence $S_1^* > M$.

In practice, with large values of N, the more likely case is $S^* = S_1^*$ for $P = P_1^* > M$.

A.3 The "large dataset" case

If we take $P \ll \rho_2 N$ ("large dataset" case), the P^2 term in the speedup expression (12') becomes negligible, and the speedup becomes

$$S(P) = \frac{\rho \frac{1}{\lceil M/P \rceil} MP}{\frac{1}{N} P^2 + \rho_2 P + \rho_1 \frac{1}{\lceil M/P \rceil} M} \approx \rho / \left(\frac{\rho_1}{P} + k\frac{\rho_2}{M}\right)$$

where $k = \lceil M/P \rceil \in \{1, 2, \ldots, M\}$. Now, by taking $k = 1$ ($M < P$) and $k = M/P$ (M divisible by P) we obtain the following important cases:

$$\text{if } M \text{ divisible by } P\text{:} \quad S(P) \approx P; \qquad \text{if } M > P\text{:} \quad S(P) \approx \rho / \left(\frac{\rho_1}{P} + \frac{\rho_2}{M}\right) \tag{20'}$$

so that the speedup is almost perfect up to $P = M$, and then it is approximately the weighted harmonic mean of M and P (hence, $S(P)$ is monotonically increasing and between M and P). For $P \gg \rho_1$, we have $S(P) \approx \frac{\rho}{\rho_2} M > M$.

B Important MPI functions

For reference, we briefly describe important MPI functions and their parameters (Gropp et al., 1999a,b; Message Passing Interface Forum, 2012).

B.1 Environment Management Routines

- `MPI_Init(&argc,&argv)`: initialises the MPI execution environment. It must be called exactly once in every MPI program before calling any other MPI functions. For C programs, it may be used to pass the command-line arguments to all processes. Input: `argc`, pointer to the number of arguments; `argv`, pointer to the argument vector.
- `MPI_Comm_size(comm,&size)`: returns the total number of MPI processes in the specified communicator. If the communicator is `MPI_COMM_WORLD`, then it represents the number of MPI tasks available to your application. Input: `comm`, communicator (handle). Output: `size`, number of processes in the group of `comm` (integer).
- `MPI_Comm_rank(comm,&rank)`: returns the rank of the calling MPI process within the specified communicator. Initially, each process is assigned a unique integer rank between 0 and the number of tasks (1 within the communicator `MPI_COMM_WORLD`). This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well. Input: `comm`, communicator (handle). Output: `rank`, rank of the calling process in the group of `comm` (integer).
- `MPI_Finalize()`: terminates the MPI execution environment. It should be the last MPI function called in any MPI program.

B.2 Point to Point Communication Routines

MPI point-to-point operations involve message passing between exactly two MPI tasks. One task performs a send operation and the other task performs a matching receive operation. There are different types of send and receive functions, used for different purposes, such as synchronous send; blocking send, blocking receive; non-blocking send, non-blocking receive; buffered send; combined send-receive. Their argument list generally takes one of the following formats:

- Blocking send: `MPI_Send(buffer,count,type,dest,tag,comm)`.
- Non-blocking send: `MPI_Isend(buffer,count,type,dest,tag,comm,request)`.
- Blocking receive: `MPI_Recv(buffer,count,type,source,tag,comm,status)`.
- Non-blocking receive: `MPI_Irecv(buffer,count,type,source,tag,comm,request)`.

Here is a brief description of the parameters:

- `buffer`: program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is to be sent or received. For C programs, `buffer` is passed by reference and must be prepended with an ampersand: `&buffer`.
- `count`: indicates the number of data elements of type `type` to be sent.
- `type`: the data type that is sent or received. For reasons of portability, MPI predefines its elementary data types.
- `dest`: for send routines, it indicates the process to which a message should be delivered. Specified as the rank of the receiving process.
- `source`: for receive routines, it indicates the originating process of the message. Specified as the rank of the sending process. It may be set to the wild card `MPI_ANY_SOURCE` to receive a message from any task.

- `tag`: arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card `MPI_ANY_TAG` can be used to receive any message regardless of its tag.
- `comm`: it indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator `MPI_COMM_WORLD` is usually used.
- `status`: for receive routines, it indicates the source of the message and the tag of the message. In C, `status` is a pointer to a predefined structure `MPI_Status`. Additionally, the actual number of bytes received is obtainable from `status` via the `MPI_Get_count` routine.
- `request`: used by non-blocking send/receive. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system-assigned "handle" later (in a `Wait`-type routine) to determine completion of the non-blocking operation. In C, `request` is a pointer to a predefined structure `MPI_Request`.

These are the *blocking message passing routines*:

- `MPI_Send(&buf,count,datatype,dest,tag,comm)`: basic blocking send operation. It returns only after the application buffer in the sending task is free for reuse.
- `MPI_Recv(&buf,count,datatype,source,tag,comm,&status)`: receive a message and block until the requested data is available in the application buffer in the receiving task.
- `MPI_Ssend(&buf,count,datatype,dest,tag,comm)`: synchronous blocking send. It sends a message and blocks until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.
- `MPI_Bsend(&buf,count,datatype,dest,tag,comm)`: buffered blocking send. It allows the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered. It alleviates problems associated with insufficient system buffer space. It returns after the data has been copied from the application buffer space to the allocated send buffer. It must be used with the `MPI_Buffer_attach` routine.
- `MPI_Buffer_attach(&buffer,size)`, `MPI_Buffer_detach(&buffer,size)`: used by the programmer to allocate or deallocate message buffer space to be used by `MPI_Bsend`. The `size` argument is specified in actual data bytes (not a count of data elements). Only one buffer can be attached to a process at a time.
- `MPI_Wait(&request,&status)`: blocks until a specified non-blocking send or receive operation has completed.

These are the *non-blocking message passing routines*:

- `MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)`: identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to `MPI_Wait` or `MPI_Test` indicate that the non-blocking send has completed.
- `MPI_Irecv(&buf,count,datatype,source,tag,comm,&request)`: identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to `MPI_Wait` or `MPI_Test` to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.
- `MPI_Test(&request,&flag,&status)`: checks the status of a specified non-blocking send or receive operation. It returns in `flag` logical true (1) if the operation completed and logical false (0) otherwise.

B.3 Collective Communication Routines

- `MPI_Bcast(&buffer,count,datatype,root,comm)`: data movement operation. IT broadcasts (sends) a message from the process with rank `root` to all other processes in the group.
- `MPI_Gather(&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,root,comm)`: data movement operation. It gathers distinct messages from each task in the group to a single destination task. Its reverse operation is `MPI_Scatter`.
- `MPI_Allgather(&sendbuf,sendcount,sendtype,&recvbuf,recvcount,recvtype,comm)`: data movement operation. It concatenates data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.
- `MPI_Reduce(&sendbuf,&recvbuf,count,datatype,op,root,comm)`: collective computation operation. It applies a reduction operation on all tasks in the group and places the result in one task.
- `MPI_Allreduce(&sendbuf,&recvbuf,count,datatype,op,comm)`: collective computation and data movement operation. It applies a reduction operation and places the result in all tasks in the group. It is equivalent to `MPI_Reduce` followed by `MPI_Bcast`.

References

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

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

A. Benveniste, M. Métivier, and P. Priouret. *Adaptive Algorithms and Stochastic Approximations*, volume 22 of *Applications of Mathematics*. Springer-Verlag, Berlin, 1990.

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

D. P. Bertsekas and J. N. Tsitsiklis. Gradient convergence in gradient methods with errors. *SIAM Journal on Optimization*, 10(3):627–642, 2000.

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

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

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

J. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In L. Getoor and T. Scheffer, editors, *Proc. of the 28th Int. Conf. Machine Learning (ICML 2011)*, pages 321–328, Bellevue, WA, June 28 – July 2 2011.

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

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

M. Á. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. arXiv:1212.5921 [cs.LG], Dec. 24 2012.

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

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

X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide. Pipelined back-propagation for context-dependent deep neural networks. In *Proc. of Interspeech'12*, pages 26–29, Portland, OR, Sept. 9–13 2012.

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

P. L. Combettes and J.-C. Pesquet. Proximal splitting methods in signal processing. In H. H. Bauschke, R. S. Burachik, P. L. Combettes, V. Elser, D. R. Luke, and H. Wolkowicz, editors, *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, Springer Series in Optimization and Its Applications, pages 185–212. Springer-Verlag, 2011.

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

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the *EM* algorithm. *Journal of the Royal Statistical Society, B*, 39(1):1–38, 1977.

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

S. H. Fuller and L. I. Millett, editors. *The Future of Computing Performance: Game Over or Next Level?* National Academic Press, 2011.

R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proc. of the 17th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (SIGKDD 2011)*, pages 69–77, San Diego, CA, Aug. 21–24 2011.

S. Goedecke and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. Software, Environments and Tools. SIAM Publ., 2001.

B. Gold and N. Morgan. *Speech and Audio Signal Processing: Processing and Perception of Speech and Music*. John Wiley & Sons, New York, London, Sydney, 2000.

Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. 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, Dec. 2013.

S. L. Graham, M. Snir, and C. A. Patterson, editors. *Getting up to Speed: The Future of Supercomputing*. National Academic Press, 2004.

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

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

W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999b.

S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In F. Bach and D. Blei, editors, *Proc. of the 32nd Int. Conf. Machine Learning (ICML 2015)*, pages 1737–1746, Lille, France, July 6–11 2015.

S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 1135–1143. MIT Press, Cambridge, MA, 2015.

S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

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

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 28 2006.

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

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

R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1–2):273–324, Dec. 1997.

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

H. J. Kushner and G. G. Yin. *Stochastic Approximation and Recursive Algorithms and Applications*. Springer Series in Stochastic Modelling and Applied Probability. Springer-Verlag, second edition, 2003.

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

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

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

R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL HLT)*, pages 456—464, 2010.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS), Sept. 21 2012.

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

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

A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Computer Vision*, 42(3):145–175, May 2001.

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

G. C. Pflug. *Optimization of Stochastic Models: The Interface between Simulation and Optimization*. Kluwer Academic Publishers Group, 1996.

M. Ranzato, F. J. Huang, Y. L. Boureau, and Y. LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proc. of the 2007 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'07)*, pages 1–8, Minneapolis, MN, June 18–23 2007.

R. Raziperchikolaei and M. Á. Carreira-Perpiñán. Optimizing affinity-based binary hashing using auxiliary coordinates. arXiv:1501.05352 [cs.LG], Feb. 5 2016.

P. Richtárik and M. Takáč. Distributed coordinate descent method for learning with big data. arXiv:1310.2059 [stat.ML], Oct. 8 2013.

R. T. Rockafellar. Monotone operators and the proximal point algorithm. *SIAM J. Control and Optim.*, 14 (5):877–898, 1976.

G. Saon and J.-T. Chien. Large-vocabulary continuous speech recognition systems: A look at some recent advances. *IEEE Signal Processing Magazine*, 29(6):18–33, Nov. 2012.

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

T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio. Robust object recognition with cortex-like mechanisms. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 29(3):411–426, Mar. 2007.

J. C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. John Wiley & Sons, 2003.

A. Talwalkar, S. Kumar, M. Mohri, and H. Rowley. Large-scale SVD and manifold learning. *J. Machine Learning Research*, 14(1):3129–3152, 2013.

M. Vladymyrov and M. Á. Carreira-Perpiñán. Entropic affinities: Properties and efficient numerical computation. In S. Dasgupta and D. McAllester, editors, *Proc. of the 30th Int. Conf. Machine Learning (ICML 2013)*, pages 477–485, Atlanta, GA, June 16–21 2013.

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

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

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

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

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

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

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

S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for DNN training. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'13)*, pages 6660–6663, Vancouver, Canada, Mar. 26–30 2013.

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