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
March 8, 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 nodes 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 nodes. ParMAC exhibits high parallelism, low communication overhead, and facilitates data shuffling, load balancing, fault tolerance and streaming data processing. We develop ParMAC to learn binary autoencoders for fast image retrieval, and demonstrate near-linear speedups in a 128-processor cluster with a training set of 100 million high-dimensional points.

1 Introduction: big data, parallel processing and nested models

Serial computing has reached a plateau and parallel, distributed architectures are becoming widely available, from processors with a few cores to cloud computing with 1000s of processors. 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 parallelize well. The cost of communicating (through the memory or a network) greatly exceeds the cost of computing and will continue to do so for the foreseeable time (Fuller and Millett, 2011; Graham et al., 2004). Thus, good parallel algorithms must minimize communication and maximize computation per processor, while creating sufficiently many subproblems to benefit from as many processors as possible. The load (in runtime) on each processor should be approximately equal. Faults become more frequent as the number of processors increases, particularly if they are inexpensive processors. Processors 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 processor, so distributed-memory architectures are necessary. Sending data between processors 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 $y = f_{K+1}(\ldots f_2(f_1(x))\ldots)$, i.e., consisting of a nested mapping from the input $x$ to the output $y$. Such nested models involve multiple parameterized 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 optimization 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 processors is far costlier than computation. This paper proposes ParMAC, a parallel, distributed framework to learn nested models using MAC, implements it in 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 describe MAC in general and for BAs (section 3), introduce the ParMAC model and its MPI implementation (section 4), and then show experimental results (section 5).

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, 2015) 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 nodes), 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 minimum (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; Chilimbi et al., 2014). 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 processors 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 processors, 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., 2008; Vladymyrov and Carreira-Perpiñán, 2013).

ParMAC is specifically designed for nested models, which are typically nonconvex and include deep nets and many other models. 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 \( f(x; W) = f_{K+1}(\ldots f_2(f_1(x; W_1); W_2)\ldots; W_{K+1}) \) with parameters \( W \), 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 parallelization. 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 \( W \) step) and over the coordinates (which we call the \( Z \) step). The result is a coordination-minimisation (CM) algorithm: the minimisation (\( W \)) step updates the parameters by splitting the nested model into independent submodels and training them using existing algorithms, and the coordination (\( Z \)) 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 interprocess communication low; the use of stochastic optimisation in the \( W \) step; and the tradeoff between the different amount of parallelism in the \( W \) and \( Z \) steps. 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 \( h(x) \) that maps a real vector \( x \in \mathbb{R}^D \) onto a binary code vector with \( L < D \) bits, \( z \in \{0,1\}^L \), and a linear decoder \( f(z) \) which maps \( z \) back to \( \mathbb{R}^D \) in an effort to reconstruct \( x \). We will call \( h \) a binary hash function (see later). Let us write \( h(x) = s(Ax) \) (\( A \) includes a bias by having an extra dimension \( x_0 = 1 \) for each \( x \)) where \( A \in \mathbb{R}^{L \times (D+1)} \) 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 \( X = (x_1, \ldots, x_N) \), our objective function, which involves the nested model \( y = f(h(x)) \), is the usual least-squares reconstruction error:

\[
E_{BA}(h,f) = \sum_{n=1}^{N} \|x_n - f(h(x_n))\|^2. \tag{1}
\]

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

\[
\min_{h,f,Z} \sum_{n=1}^{N} \|x_n - f(z_n)\|^2 \quad \text{s.t.} \quad z_n = h(x_n), \; z_n \in \{0,1\}^L, \; n = 1, \ldots, N. \tag{2}
\]

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 \( \mu \), so the constraints are eventually satisfied:

\[
E_Q(h,f,Z;\mu) = \sum_{n=1}^{N} \left( \|x_n - f(z_n)\|^2 + \mu \|z_n - h(x_n)\|^2 \right) \quad \text{s.t.} \quad z_n \in \{0,1\}^L, \; n = 1, \ldots, N. \tag{3}
\]

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

- Over \( Z \) for fixed \((h,f)\), 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 \( n \)): \( \min_{z} \|x - f(z)\|^2 + \mu \|z - h(x)\|^2 \) s.t. \( z \in \{0,1\}^L \). 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 \( W = (h,f) \) for fixed \( Z \), we obtain \( L + D \) independent problems: for each of the \( L \) single-bit hash functions (which try to predict \( Z \) optimally from \( X \)), each solvable by fitting a linear SVM; and for each of the \( D \) linear decoders in \( f \), each a linear least-squares problem. With linear \( h \) and \( f \) this simply involves fitting \( L \) SVMs to \((X,Z)\) and \( D \) linear regressors to \((Z,X)\).

The user must choose a schedule for the penalty parameter \( \mu \) (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 \( \mu \) (Carreira-Perpiñán and Raziperchikolaei, 2015). This occurs whenever \( Z \) does not change compared to the previous \( Z \) step, which gives a practical stopping criterion. Also, in order to generalise well to unseen data, we stop iterating for a \( \mu \) value not when we (sufficiently) optimize \( E_Q(h,f,Z;\mu) \), 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 \( Z \), and besides is faster. We also have to initialize \( Z \). 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.
Each \( z_n \) acts as a running example, but the ideas apply beyond deep nets. Consider a regression problem of mapping \( x \) to \( y \) (for a BA, \( x = y \)). 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 \( x \) to \( y \) (both high-dimensional) with a deep net \( f(x) \) given a dataset of \( N \) pairs \((x_n, y_n)\). We minimise the least-squares error (other loss functions are possible):

\[
E(W) = \frac{1}{2} \sum_{n=1}^{N} \|y_n - f(x_n; W)\|^2 \quad f(x; W) = f_{K+1}(\ldots f_2(f_1(x; W_1); W_2)\ldots; W_{K+1})
\]

where each layer function has the form \( f_k(x; W_k) = \sigma(W_k x) \), 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:

\[
\frac{1}{2} \sum_{n=1}^{N} \|y_n - f_{K+1}(z_{K,n}; W_{K+1})\|^2 \quad \text{s.t.} \quad \begin{cases} 
z_{K,n} = f_K(z_{K-1,n}; W_K) \\
z_{1,n} = f_1(x_n; W_1) \end{cases} \quad n = 1, \ldots, N.
\]

Each \( z_{k,n} \) can be seen as the coordinates of \( x_n \) in an intermediate feature space, or as the hidden unit activations for \( x_n \). Intuitively, by eliminating \( Z \) 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:

\[
E_Q(W,Z; \mu) = \frac{1}{2} \sum_{n=1}^{N} \|y_n - f_{K+1}(z_{K,n}; W_{K+1})\|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} \|z_{k,n} - f_k(z_{k-1,n}; W_k)\|^2
\]

over \((W,Z)\) and drive \( \mu \to \infty \). This defines a continuous path \((W^*(\mu), Z^*(\mu))\) 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 \( f \) and unfolding...
it over layers. Every squared term involves only a shallow mapping; all variables \((W, Z)\) are equally scaled, which improves the conditioning of the problem; and the derivatives required are simpler: we require no backpropagated gradients over \(W\), and sometimes no gradients at all. We now apply alternating optimisation of the quadratic-penalty objective over \(Z\) and \(W\):

**W step (submodels)** Minimising over \(W\) for fixed \(Z\) 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 \(Z\) for fixed \(W\) separates over the coordinates \(z_n\) for each data point \(n = 1, \ldots, N\) and can be solved using the derivatives wrt \(z\) of the single-layer functions \(f_1, \ldots, f_{K+1}\) (omitting the subindex \(n\)): \(\min_z \|y - f_{K+1}(z_K)\|^2 + \mu \sum_{k=1}^K \|z_k - f_k(z_{k-1})\|^2\).

Thus, the \(W\) step results in many independent, single-layer single-unit submodels that can be trained with existing algorithms, without extra programming cost. The \(Z\) 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 \(W\) step) which are coordinated through the auxiliary variables (the \(Z\) 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 \(Z\) step, the \(N\) subproblems for \(z_1, \ldots, z_N\) are independent, one per data point. Each \(z_n\) step depends on all or part of the current model. (2) In the \(W\) 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 \((x_n, y_n, z_n)\)). 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.

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 \((X, Y)\), the auxiliary coordinates \(Z\), 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 \((X, Y, Z)\) corresponding to a subset of the points. Only model parameters are communicated, during the \(W\) step, following a circular topology, which implicitly implements a stochastic optimisation. The model parameters are the hash functions \(h\) and the decoder \(f\) for BAs, and the weight vector \(w_h\) 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 nodes, each with their own memory and CPU (possibly multicore), which are connected through a network. The nodes are connected in a circular (ring) unidirectional topology, i.e., node 1 \(\rightarrow\) node 2 \(\rightarrow\) \(\cdots\) \(\rightarrow\) node \(P\) \(\rightarrow\) node 1, where “node \(p\) \(\rightarrow\) node \(q\)” means node \(p\) can send data directly to node \(q\) (and we say node \(q\) is the successor of node \(p\)). Call \(D = \{(x_n, y_n, z_n): n \in \{1, \ldots, N\}\}\) the entire dataset and corresponding coordinates. Each node \(p\) will store a subset \(D_p = \{(x_n, y_n, z_n): n \in I_p\}\) such that the subsets are disjoint and their union is the entire data, i.e., the index sets satisfy \(I_p \cap I_q = \emptyset\) if \(p \neq q\) and \(\bigcup_{p=1}^P I_p = \{1, \ldots, N\}\). Before the \(Z\) step starts, each node will contain all the (just updated) submodels. This means that in the \(Z\) step each node \(p\) processes its auxiliary coordinates \(\{z_n: n \in I_p\}\) independently of all other nodes, i.e., no communication occurs.

The \(W\) step is more subtle. At the beginning of the \(W\) step, each node 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 node, 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.
Figure 2: ParMAC model with $P = 4$ nodes, $M = 12$ submodels and $N = 40$ data points. "$w_h$" 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 $W$ step all copies are identical.

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 node takes to process its portion $M/P$ of submodels. At each clock tick, the $P$ nodes update each a different portion $M/P$ of the submodels. For example, in fig. 2, at clock tick 1 node 1 updates submodels 1–3 using its data $D_{I_1}$ (where $I_1 = \{1, \ldots, 10\}$); node 2 updates submodels 4–6; node 3 updates submodels 7–9; and node 4 updates submodels 10–12. This happens in parallel. Then each node sends the submodels updated to its successor, also in parallel. In the next tick, each node updates the submodels it just received, i.e., node 1 updates 10–12, node 2 updates submodels 1–3, node 3 updates submodels 4–6; and node 4 updates submodels 7–9 (and each node always uses its data portion, which never changes). This is repeated until each submodel has visited each node and thus has been updated with the entire dataset $D$. 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 node 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 nodes. Finally, before starting with the $Z$ step, each node must contain the entire (just updated) model. We achieve this by running a final round of communication without computation, i.e., each node sends its just updated submodels to its successor. Thus, after one clock tick, node $p$ sends $M/P$ final submodels to node $p + 1$ and receives $M/P$ submodels from node $p - 1$. After $P - 1$ clock ticks, each node has received the remaining $M(1–1/P)$ submodels that were finished by other nodes, hence each node contains a (redundant) copy of all the current submodels.

Since each submodel is updated as soon as it visits a node, rather than computing the exact gradient once it has visited all nodes 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 node’s data portion into minibatches, which should be typically the case in practice). From this point of view, we can view the $W$ step as doing SGD on each submodel in parallel by having each submodel visit the minibatches in each node.

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 asynchronously in parallel 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 $z_n$ are optimised independently, in parallel over machines (since each machine contains $x_n$, $y_n$, $z_n$, and all the model parameters). No communication occurs at all.

**Expected speedup** We can estimate the computation and communication times, and expected parallel speedup, in the ParMAC model assuming $P \leq N, M$, i.e., we have no more nodes than data points (almost certain in practice) or submodels (true for some models). Call $t_W$ and $t_Z$ the CPU times to process one data point with the entire model (this is the aggregate of all submodels processing one data point) in the $W$ and $Z$ steps, respectively, and $t_e$ the time to send the entire model from one node to another. Then the runtime of the $W$ step with $e$ epochs is $et_W N + (e + 1)t_e$ (CPU time to process all points within one node, plus the time to send the entire model $e + 1$ times), and the runtime of the $Z$ step is $t_Z N$ (CPU time to process all points within one node, no communication). Hence, the runtime per ParMAC iteration ($W+Z$ step) with $P$ processors is $T(P) = (et_W + t_Z) N + (e + 1)t_e$ for $P > 1$ and $T(1) = (et_W + t_Z) N$. We obtain a parallel speedup

$$S(P) = \frac{T(1)}{T(P)} = N/\left(\frac{N}{P} + \frac{(e + 1)t_e}{et_W + t_Z}\right) \approx P$$

(7)

if $N \gg P$. In practice $t_e$ is probably quite larger than $t_W$ and $t_Z$ (though this depends on what the optimizations are in the $W$ and $Z$ steps), but its effect will be negligible in large datasets compared to the $N/P$ term. Indeed, our experiments show a near-linear scaling with up to 128 processors even with relatively small datasets as long as we have sufficiently many submodels ($M \geq P$).

### 4.1 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. With distributed systems, this can sometimes be a problem and require data movement across processors. Shuffling is easy in ParMAC. Within a processor, we can simply access the local data (minibatches) in random order at each epoch. Across processors, we can simply reorganise the circular topology randomly (while still circular) at the beginning of each new epoch (by resetting the successor’s address of each node).

**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 nodes 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 node. If the processing power of node $p$ is proportional to $\alpha_p > 0$ (where $\alpha_p$ could represent the clock frequency of node $p$, say), then we allocate to node $p$ a subset of the $N$ points proportional to $\alpha_p$, i.e., node $p$ gets $N\alpha_p/(\alpha_1 + \cdots + \alpha_P)$. This is done once and for all at loading time.

**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 node collects more data than it can store. The circular topology allows us to add or remove nodes 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 node (e.g. as this node collects new data), and likewise old data are discarded within a node. And (2) new data are added by adding a new node to the topology, and old data are discarded by removing an existing node from the topology. Both forms are easily achieved in ParMAC. The first form, within-node, is trivial: a node can always add or remove data without any change to the system, because the data for each node is private and never interacts with other nodes 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 $\{(x_n, y_n, z_n)\}$ from that node. Adding data means inserting $\{(x_n, y_n)\}$ in that node and, if necessary, creating within that node coordinate values $\{z_n\}$ (e.g. by applying the nested model to $x_n$). We never upload or send any $z$ values over the network.

The second form, creating a new node or removing an existing one, is barely more complicated. Imagine we currently have $P$ nodes. We can add a new node, with its own preloaded data $\{(x_n, y_n)\}$, as follows. Adding it to the circular topology simply requires connecting it between any two nodes (done by setting the address of their successor): before we have node $p \rightarrow p + 1$, afterwards we have node $p \rightarrow p + 1 \rightarrow p + 2$. 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 node is simply sending along a copy of the final submodels. In the $Z$ step, we proceed as usual, but with $P + 1$ nodes. Removing a node is easier. To remove node $p$, we do so in the $Z$ step, reconnecting node $p - 1 \rightarrow p + 1$ and returning node $p$ to the cluster. That is all. In the subsequent $W$ step, all nodes contain the full model, and the submodels will visit the data in each node, thus not visiting the data in the removed node.

**Fault tolerance** This can be handled similarly to streaming when discarding a node, except that the fault can occur at any time and is not intended. Imagine a fault occurs at node $p$ and we need to remove it. If it happens during the $Z$ step, all we need to do is discard the faulty node and reconnect the circular topology. If it happens during the $W$ step, we also discard and reconnect. Some submodels will have already been updated on node $p$'s data and some will not, but this is not a problem. All we have to do is make sure that each updated submodel is updated on all other nodes. This can be achieved by tagging each submodel with a list of the nodes it has not yet visited. At the beginning of the $W$ step the list of each submodel contains $\{1, \ldots, P\}$, i.e, all nodes. When this list is empty, for a submodel, then that submodel is finished and needs no further updates. In fact, this mechanism allows us to run ParMAC in asynchronous mode. Submodels need not be updated in sync as in the imaginary clock argument we gave earlier. All a node $p$ needs to do upon receiving a submodel is check its list: if $p$ is not in the list, then the submodel has already visited node $p$ and been updated with its data, so node $p$ simply sends it along to its successor without updating it again. If $p$ is in the list, then node $p$ updates the submodel, removes $p$ from its list, and sends it along to its successor.

### 4.2 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. 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 the most widely used model 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 5.1). Our ParMAC C++ code compiles and runs with both implementations. We used the highest compiler optimization level, specifically we ran mpicc -O3 -lgs1 -lgs1chias. 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 3 shows the main steps of the ParMAC algorithm for the BA. All the functions starting with MPI are API calls from the MPI library. As with all MPI programs, we start the code by initialising the MPI environment and end by finalising it. To receive data we use the asynchronous nonblocking MPI receive function MPI_Recv, followed by a wait to make sure we receive the submodel before starting to train it. 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. MPI_Bsend blocks until the buffer is copied to the MPI internal memory; after that, the MPI server takes care of sending the data appropriately. The benefit of using this version of send is that the programmer does not need to manage the buffer, so the code is simpler. Appendix A briefly describes important MPI functions and their arguments.

4.3 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 4.1.** Consider the constrained problem (5) and its quadratic-penalty function $E_Q(W, Z; \mu)$ of (6). Given a positive increasing sequence $(\mu_k) \to \infty$, a nonnegative sequence $(\tau_k) \to 0$, and a starting point $(W^0, Z^0)$, suppose the QP method finds an approximate minimiser $(W_k^*, Z_k^*)$ of $E_Q(W_k, Z_k; \mu_k)$ that satisfies $\|\nabla_{W, Z}E_Q(W_k, Z_k; \mu_k)\| \leq \tau_k$ for $k = 1, 2, \ldots$. Then, $\lim_{k \to \infty} (W_k^*, Z_k^*) = (W^*, Z^*)$, which is a KKT point for the problem (5), and its Lagrange multiplier vector has elements $\lambda_n^* = \lim_{k \to \infty} -\mu_k (Z_n^* - F(Z_n^*, W_k; x_n))$, $n = 1, \ldots, N$.

This theorem applies to the general case of $K$ differentiable layers. 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 $\mu_k$ 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 Z step remains unchanged with respect to MAC (the fact that the optimisation is distributed is irrelevant since the $N$ subproblems $z_1, \ldots, z_N$ are independent). The 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 wrt each submodel (since they are independent subproblems in the W step) below an arbitrary tolerance. This can also be guaranteed under standard conditions. In general, we can use convergence conditions from stochastic optimisation (Beveniste 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 $\eta_t$ 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). We can give much tighter conditions on the convergence and the convergence rate when the subproblems in the 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, 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 $O(1/t^\alpha)$ or $O(1/t^\beta)$ where the coefficients $\alpha > 0$, $0 < \beta < 1$ and the constant factors in the $O$-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 W step. This convergence guarantee is independent of the number of models and subproblems (since they are independent in the W step) and the number of processors $P$ (since
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(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(submodeltypes) {
        case 'SVM': HtrainSGD();
        case 'linlayer': FtrainSGD();
      }
    } else 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_lookup_table();
      // pick the successor process from the lookup table
      loadsubmodel(sendbuffer);  // load the submodel from its struct into the 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 3: Binary autoencoder ParMAC algorithm (fragment), showing important MPI calls.
Table 1: Detailed hardware specification of the two machines used in our experiments.

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

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 $W$ step exactly (as MAC assumes). First, each processor $p = 1, \ldots, P$ computes the exact sum of per-point gradients for each model (by summing over its data portion), in parallel. Then, we aggregate these $P$ partial gradients into one exact gradient, for each model. This could be done via a parameter server, or by having each processor acts as the parameter server for one model, 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 known. 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 $W$ 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 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.

5 Experiments

5.1 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). Each node contains 16 Intel Xeon E5-2670 processors, 64GB RAM (4GB/processor) 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](http://idi.ucsd.edu/computing).

**Shared-memory** This is a 72-processor machine (32 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 processors of the 72.

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 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 $32 \times 32$ 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 (Jegou 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 $E_{BA} (h, f)$ which we want to minimise, eq. (1). (2) The quadratic-penalty function $E_Q (h, f, Z; \mu)$ which we actually minimise for each value of $\mu$, 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 $(K, k) = (1000, 100)$ for CIFAR, $(100, 100)$ for SIFT-10K and $(10000, 10000)$ 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 $Z$ 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 processor).

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), with all the default settings. The SGD step size is tuned automatically in each iteration by examining the first 1,000 datapoints.

We use a multiplicative \mu schedule $\mu_i = \mu_0 a^i$ where the initial value $\mu_0$ 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 ($i = 0, \ldots, 25$). 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.

5.2 Effect of stochastic steps in the W step  Figures 4 to 6 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 processors $P$ on the learning curves (errors $E_Q$ and $E_{BA}$, precision). As the number of epochs increases, the 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 redundancy 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 ($L = 16$) compared to the number of processors (up to $P = 128$), so the W step has less parallelism.

Fig. 6 shows the positive effect of data shuffling in the W step. To shuffle the minibatches, the successor of a processor 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.

5.3 Speedup  The fundamental advantage of ParMAC and distributed optimisation in general is the ability to train on datasets that do not fit in single-processor machines, and the reduction in runtime because of parallel process-
Figure 4: SIFT-10K dataset. **Left two columns:** single processor \((P = 1)\) and different number of epochs \(e\) in the \(W\) step. **Right two columns:** fixed number of epochs (either 1 or 8) but different number of processors \(P\).

Even though these datasets and especially the number of independent submodels \((L = 16\) hash functions and \(D\) linear decoders\) are relatively small, we achieve near-linear speedups up to the maximum number of processors 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 \(W\) step, whose parallelisation ability (i.e., the number of concurrent processes) is limited by \(L\) (the \(Z\) 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 \(W\) step. The runtime for SIFT-1M on \(P = 128\) processors 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 \((\text{Le et al., 2012}; \text{Dean et al., 2012})\), although admittedly the deep nets trained there were considerably larger than our BAs.

### 5.4 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)

---

1In “strong scaling”, the total problem size is fixed and the problem size on each processor is inversely proportional to the number of processors \(P\). In “weak scaling”, the problem size on each processor is fixed, so the total problem size is proportional to \(P\). High speedups are easier to obtain in weak scaling than in strong scaling \((\text{Goedecke and Hoisie, 2001})\).
for each query to its $k$ nearest vectors in the base set). The training set contains $N = 100M$ vectors, each consisting of 128 SIFT features.

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 $\sigma$ 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.
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).
We run 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 8–9:

<table>
<thead>
<tr>
<th>Hash function (encoder)</th>
<th>Recall@R=100</th>
<th>Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear SVM</td>
<td>61.5%</td>
<td>29.30</td>
</tr>
<tr>
<td>kernel SVM</td>
<td>66.1%</td>
<td>32.19</td>
</tr>
</tbody>
</table>

The learning curves (fig. 8) 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. 9). Note the $E_{BA}$ does not decrease monotonically. This is because MAC optimises
instead the penalised function $E_Q$, in an effort the minimise $E_{BA}$ as $\mu$ increases.

Based on our previous results, the small number of epochs and the larger number of submodels in the W
step, we expect near-linear speedups. We cannot compute the actual speedup because the single-processor
runtime is enormous. Using a scaled-down model and training set, we estimated that training in one processor
(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. 10.
5.5 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 \((1 \times 16, \text{pure shared-memory})\) to 16 nodes each with 1 processor \((16 \times 1, \text{pure distributed})\), and intermediate configurations such as 2 nodes each with 8 processors \((2 \times 8)\). We used the RBF model from the SIFT-1B experiment with all settings as before and trained it on a subset of 20K points for a single iteration. Figure 10 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.

6 Discussion

Developing parallel, distributed optimisation algorithms for nonconvex problems in machine learning is challenging, as shown by recent efforts by large teams of researchers (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). A circular topology is sufficient to achieve a low communication between processors. There is no close coupling 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 processors and consist of a \( Z \) step with no interprocessor
Figure 8: SIFT-1B dataset, using the shared-memory and distributed system (solid and dashed lines, resp.).

Figure 9: 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 10: 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.
communication, and a \( W \) step where submodels visit processors 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 processor only the model portion that it needs, or by allocating cores within a multicore node accordingly. Also, the \( W \) and \( Z \) step optimisations can make use of further parallelisation by GPUs or by distributed convex optimisation algorithms. Many more refinements can (and should) be done in an industrial implementation, such as storing and communicating 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., 2015b,a). In this paper, we have tried to keep implementation issues to a minimum and focus on the parallelisation speedups of ParMAC, rather than trying to achieve the very best performance for a particular dataset and model.

ParMAC is very efficient in communication: no data or coordinates are ever sent, only the entire model \( \epsilon + 1 \) times per iteration. Using one epoch (which is sufficient in large datasets) this is near optimal if we note the following. If the data cannot be communicated, then at every iteration each submodel must visit each node (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 that is necessary to be able to optimize a global model on the entire dataset with \( P \) nodes, because each node must be able to communicate with some other node. It has \( P \) edges and is truly distributed, with each node 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 nodes. This is a bipartite graph with bidirectional edges between servers and workers, having \( SW \) edges, which is quite larger than the number of nodes \( P = S + W \). 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, heterogeneous 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, 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.

7 Conclusion

We have proposed ParMAC, a distributed implementation of the method of auxiliary coordinates for training nested, nonconvex models in general, and demonstrated it 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 processor 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 processor 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 processors to a minimum within each iteration. When the number of submodels is comparable or larger than the number of processors, which is often the case, this exhibits near-linear speedups. ParMAC also makes it easy to account for data shuffling, load balancing, streaming and fault tolerance.

A Important MPI functions

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

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.

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 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.

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.

Acknowledgments

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.

References


