ParMAC: distributed optimisation of nested functions,

with application to learning binary autoencoders

Miguel Á. Carreira-Perpiñán and **Mehdi Alizadeh**

Electrical Engineering and Computer Science

University of California, Merced

http://eecs.ucmerced.edu

Outline

- Optimising nested (deep) systems using the method of auxiliary coordinates (MAC)
- Particular case: binary autoencoders
- Distributed optimisation with MAC: ParMAC

Nested systems: a ubiquitous model pattern in ML

Common in computer vision, speech processing, machine learning...

- Object recognition pipeline:

  image pixels $\rightarrow$ SIFT/HoG $\rightarrow$ $\begin{matrix}k\text{-means} \\ \text{sparse coding}\end{matrix}$ $\rightarrow$ pooling $\rightarrow$ classifier $\rightarrow$ object category

- Phone classification pipeline:

  waveform $\rightarrow$ MFCC/PLP $\rightarrow$ classifier $\rightarrow$ phoneme label

- Preprocessing for regression/classification/search/etc.:

  image pixels $\rightarrow$ PCA/LDA $\rightarrow$ classifier $\rightarrow$ output/label
  image pixels $\rightarrow$ projection $\rightarrow$ thresholding $\rightarrow$ search $\rightarrow$ nearest neighbour index

- Deep net: $\mathbf{x} \rightarrow \{\sigma(\mathbf{w}_i^T \mathbf{x} + a_i)\} \rightarrow \{\sigma(\mathbf{w}_j^T \{\sigma(\mathbf{w}_i^T \mathbf{x} + a_i)\}) + b_j\} \rightarrow \cdots \rightarrow \mathbf{y}$

Mathematically, they construct a (deeply) nested, parametric mapping from inputs to outputs:

$$\mathbf{F}(\mathbf{x}; \mathbf{W}) = \mathbf{F}_{K+1}(\ldots \mathbf{F}_2(\mathbf{F}_1(\mathbf{x}; \mathbf{W}_1); \mathbf{W}_2) \ldots; \mathbf{W}_{K+1})$$

Training nested systems: chain rule, backpropagation

- Apply the chain rule, layer by layer, to obtain a gradient wrt all the parameters.

  Ex.: $\frac{\partial}{\partial \mathbf{g}}(\mathbf{g}(\mathbf{F}(\cdot))) = \mathbf{g}'(\mathbf{F}(\cdot))$, $\frac{\partial}{\partial \mathbf{F}}(\mathbf{g}(\mathbf{F}(\cdot))) = \mathbf{g}'(\mathbf{F}(\cdot))\,\mathbf{F}'(\cdot)$.

  Then feed to nonlinear optimiser.

  Gradient descent, CG, L-BFGS, Levenberg-Marquardt, Newton, etc.

- Straightforward and widespread in practice.
- Disadvantages:
  - difficult to parallelise
  - requires differentiable layers in order to apply the chain rule
  - the gradient is cumbersome to compute, code and debug

    this may be avoided with automatic differentiation
  - requires nonlinear optimisation
  - vanishing gradients $\Rightarrow$ ill-conditioning $\Rightarrow$ slow progress even with second-order methods

    this gets worse the more layers we have.

The method of auxiliary coordinates (MAC)

M. Carreira-Perpiñán and W. Wang: *Distributed optimization of deeply nested systems*, AISTATS 2014, arXiv:1212.5921.

- A general strategy to train all the parameters of a nested system. Not an algorithm but a meta-algorithm (like EM).
- Allows layerwise training (simple submodels trainable by reusing existing algorithms) and introduces parallelism.

Basic idea (design pattern):

1. Turn the nested problem $E_{\text{nested}}(\mathbf{W})$ into an unnested, constrained problem by introducing new parameters to be optimised over (the auxiliary coordinates $\mathbf{Z}$) that decouple the layers.
2. Optimise the constrained problem with a penalty method.
   Augmented Lagrangian, quadratic penalty, etc.
3. Apply alternating optimisation to the penalised function:
   - Over $\mathbf{W}$: submodels step (reuse a single-layer training algorithm, typically)
   - Over $\mathbf{Z}$: coordinates step (needs to be solved specially for each case)

Example: binary autoencoder

M. Carreira-Perpiñán and R. Raziperchikolaei: *Hashing with binary autoencoders*, CVPR 2015.

Original (nested) objective function (reconstruction error), $\mathbf{W} = (\mathbf{h}, \mathbf{f})$:

$$E_{\text{nested}}(\mathbf{h}, \mathbf{f}) = \sum_{n=1}^{N} \|\mathbf{x}_n - \underbrace{\mathbf{f}(\mathbf{h}(\mathbf{x}_n))}_{\text{nesting}}\|^2 \qquad \begin{cases} \text{encoder } \mathbf{h}\text{: } \mathbb{R}^D \to \{0,1\}^L \\ \text{decoder } \mathbf{f}\text{: } \{0,1\}^L \to \mathbb{R}^D \end{cases} \text{(hash functions)}$$

Difficult optimisation because E_{nested} depends on the output of $\mathbf{h}$, which is binary, so E_{nested} is piecewise constant, nonsmooth and nonconvex. The problem is usually NP-complete. The chain rule does not apply.

Application: fast search in image databases via binary hashing.

We map a high-dimensional vector $\mathbf{x} \in \mathbb{R}^D$ (e.g. an image) to a low-dimensional binary vector $\mathbf{z} = \mathbf{h}(\mathbf{x}) \in \{0,1\}^L$ using a binary hash function $\mathbf{h}$, such that Hamming distances in the binary space approximate distances in the original space.

Example: binary autoencoder (cont.)

We apply the MAC design pattern:

1. MAC-constrained problem: auxiliary coordinates $\mathbf{Z} = (\mathbf{z}_1, \dots, \mathbf{z}_N)$:

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

2. MAC-penalised objective (e.g. quadratic-penalty function), as $\mu \to \infty$:

$$\min_{\mathbf{h},\mathbf{f},\mathbf{Z}} \sum_{n=1}^{N} \underbrace{\left(\|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2 \right)}_{\text{unnested, single-layer}} \quad \text{s.t.} \quad \begin{cases} \mathbf{z}_n \in \{0,1\}^L \\ n = 1, \dots, N. \end{cases}$$

3. Alternating optimisation over $\mathbf{W} = (\mathbf{h}, \mathbf{f})$ and $\mathbf{Z}$: ...

Example: binary autoencoder (cont.)

❸ Alternating optimisation over $\mathbf{W} = (\mathbf{h}, \mathbf{f})$ and $\mathbf{Z}$:

- $\mathbf{W}$ step (submodels): one submodel per BA unit:
  - $\mathbf{h}$: fit L binary classifiers to $\{(\mathbf{x}_n, \mathbf{z}_n)\}_{n=1}^N$
  - $\mathbf{f}$: fit D regression mappings to $\{(\mathbf{z}_n, \mathbf{x}_n)\}_{n=1}^N$

  } $L + D$ submodels in parallel

  solved using existing algorithms

- $\mathbf{Z}$ step (coordinates): for each point $\{\mathbf{z}_n\}_{n=1}^N$ solve a binary optimisation in L variables:

  } N coordinates in parallel

$$\min_{\mathbf{z}_n \in \{0,1\}^L} \|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2$$

which can be solved exactly by enumeration or approximately by coordinate descent.

Binary autoencoder: MAC algorithm pseudocode

input $\mathbf{X}_{D\times N} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, $L \in \mathbb{N}$

initialise $\mathbf{Z}_{L\times N} = (\mathbf{z}_1, \dots, \mathbf{z}_N) \in \{0,1\}^{LN}$

for $\mu = 0 < \mu_1 < \cdots < \mu_\infty$

  parfor $l = 1, \dots, L$ — $\mathbf{W}$ step: $\mathbf{h}$

    $h_l(\cdot) \leftarrow$ fit SVM to $(\mathbf{X}, \mathbf{Z}_{\cdot l})$

  parfor $d = 1, \dots, D$ — $\mathbf{W}$ step: $\mathbf{f}$

    $f_d(\cdot) \leftarrow$ least-squares fit to $(\mathbf{Z}, \mathbf{X}_{d\cdot})$

  parfor $n = 1, \dots, N$ — $\mathbf{Z}$ 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 $\mathbf{h}$, $\mathbf{Z} = \mathbf{h}(\mathbf{X})$

Repeatedly solve: classification ($\mathbf{h}$), regression ($\mathbf{f}$), binarisation ($\mathbf{Z}$).

MAC in general

MAC applies very generally:

- More complex nesting, e.g. K layers:
  $\mathbf{F}(\mathbf{x}; \mathbf{W}) = \mathbf{F}_{K+1}(\dots \mathbf{F}_2(\mathbf{F}_1(\mathbf{x}; \mathbf{W}_1); \mathbf{W}_2) \dots; \mathbf{W}_{K+1})$.
- Various loss functions, full/sparse layer connectivity, constraints...

The resulting optimisation algorithm depends on:

- the type of layers $\mathbf{F}_k$ used
  linear, logistic regression, SVM, RBF network, decision tree...
- how/where we introduce the auxiliary coordinates
  at no/some/all layers; before/after the nonlinearity; penalty function
- how we optimise each step
  choice of method; inexact steps, warm start, caching factorisations, stochastic updates...

Among other advantages, it allows one to handle nondifferentiable problems and introduce parallelism, by breaking the original, nested problem into many small, unnested subproblems.

Distributed optimisation with MAC: ParMAC

Large dataset stored over P machines (N/P points/machine).

Distributing the computation necessary for faster training or dataset may not fit in a single machine.

Although MAC has inherent parallelism, in the distributed setting the machines must communicate.
How to reduce the communication? What speedups can we expect?

In MAC, the specific algorithm varies from problem to problem, but the following always hold:

- $\mathbf{Z}$ step: N independent subproblems $\mathbf{z}_1, \ldots, \mathbf{z}_N$, one per data point.
  For objective functions of the form $\sum_n L(\mathbf{x}_n; \mathbf{W})$.
  Each $\mathbf{z}_n$ depends on all or part of the current model.
- $\mathbf{W}$ step: M independent submodels. Examples:
  - Binary autoencoder: $M =$ # hash functions and linear decoders.
  - Deep net: $M =$ # hidden units.

  Each submodel depends on all the data and coordinates.

Distributed optimisation with MAC: ParMAC (cont.)

Computation vs communication:

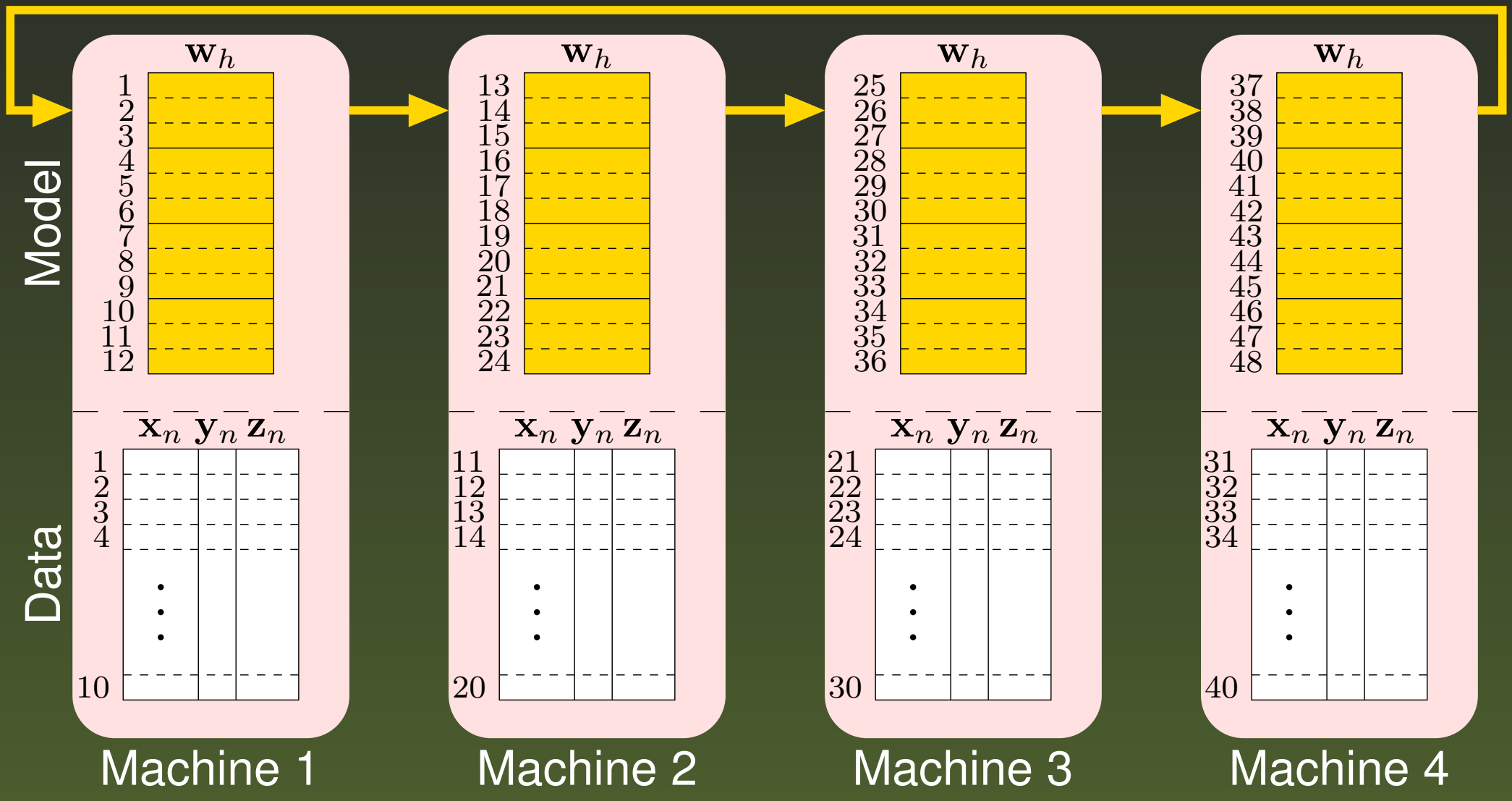
- The cost of communicating (through the memory hierarchy or a network) greatly exceeds the cost of computing in time & energy.
- It is essential to limit the amount of communication between machines so it does not obliterate the benefit of parallelism.

Basic ideas in ParMAC:

- Never communicate training data $(\mathbf{X}, \mathbf{Y})$ or coordinates $\mathbf{Z}$.
- Each machine keeps a disjoint portion of $(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ corresponding to its subset of points.
- Only model parameters are communicated, during the $\mathbf{W}$ step:
  - circular topology: implements SGD (implicitly)

How does this affect the $\mathbf{W}$ and $\mathbf{Z}$ steps?

ParMAC: setup



$P = 4$ nodes, $M = 12$ submodels $\mathbf{w}_h$ and $N = 40$ data points $(\mathbf{x}_n, \mathbf{y}_n)$. 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.

ParMAC: the $\mathbf{Z}$ step

The $\mathbf{Z}$ step behaves just as in ordinary MAC:

- Before the $\mathbf{Z}$ step starts, each machine contains all the (just updated) submodels, as well as its portion of the data and auxiliary coordinates.
- Each machine processes its auxiliary coordinates independently of all other machines.
- No communication occurs.
- At the end of the $\mathbf{Z}$ step, each machine contains its portion of the data and (just updated) auxiliary coordinates, as well as all the submodels.

This step has extremely high parallelism: N independent subproblems, one per data point, no communication. It is never a bottleneck in the parallel speedup.

ParMAC: the $\mathbf{W}$ step

This step has high parallelism: M independent subproblems, one per submodel, but with communication. It is the speedup bottleneck.

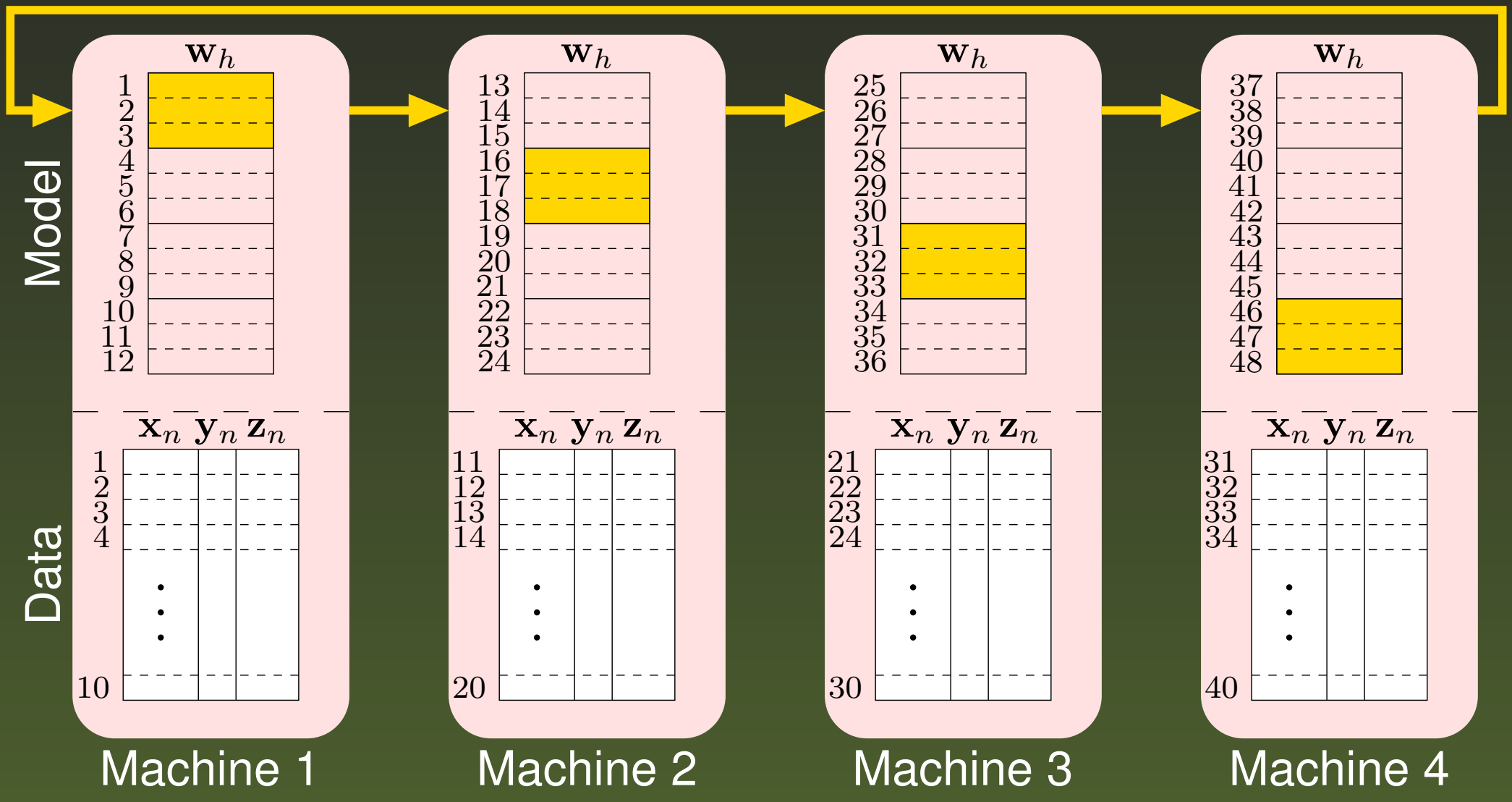
Synchronous version (easier to analyse but less practical):

- Before the $\mathbf{W}$ step starts, each machine contains its portion of the data and (just updated) auxiliary coordinates and all the submodels.
- At each clock tick, in parallel for the P machines, each machine:
  - updates a different portion M/P of the submodels,
  - then sends the submodels updated to its successor.

  After P ticks, each submodel has visited all P machines and completed one "epoch".
- This is repeated e times (for e epochs).
- A final communication-only epoch ensures each machine contains the entire updated model.

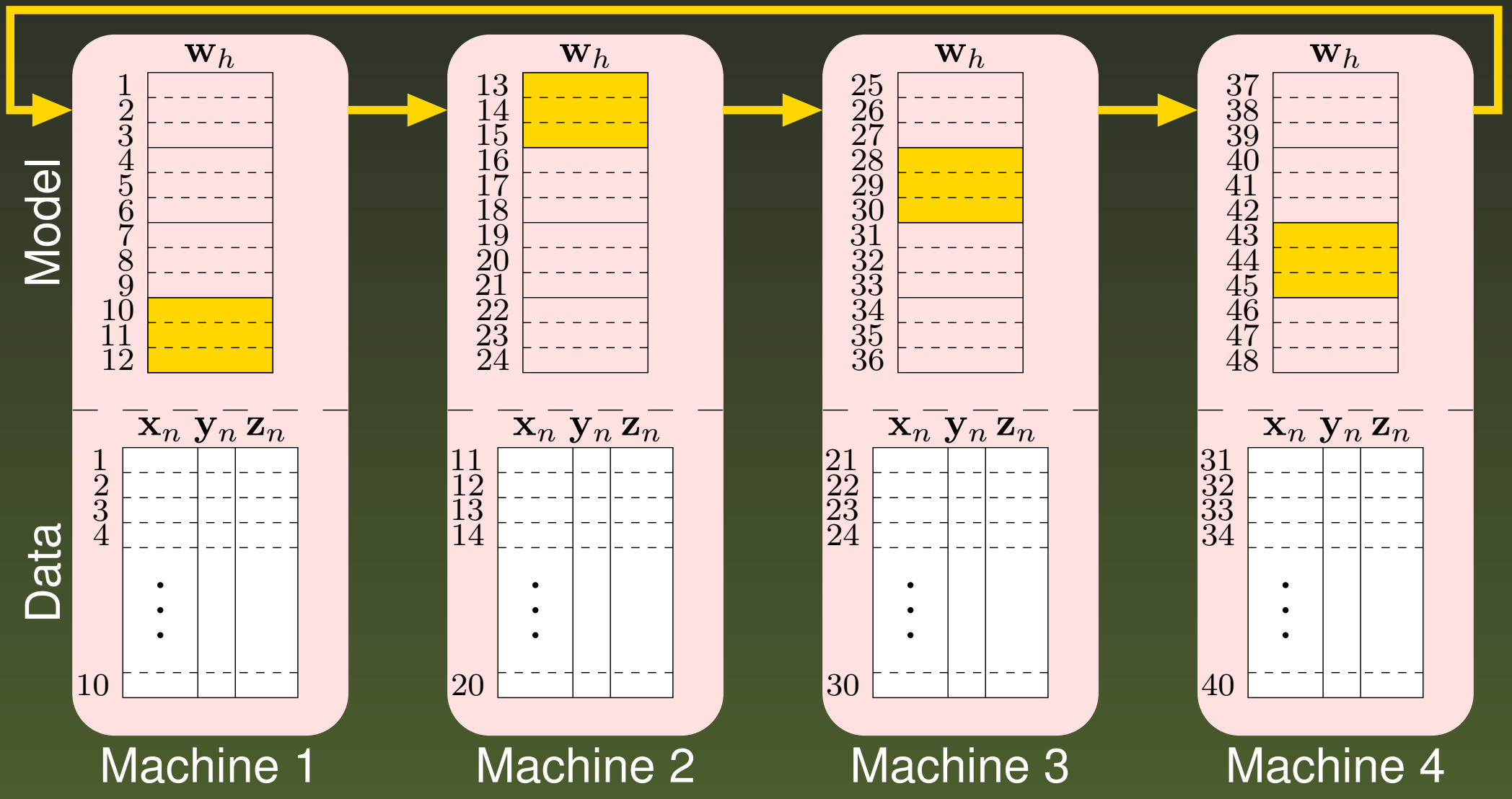
Asynchronous version. Each machine keeps a queue of submodels to be processed, and repeatedly performs the following operations: extract a submodel from the queue, process it (except in epoch $e+1$) and send it to the machine's successor (which will insert it in its queue).

ParMAC: the W step (cont.)



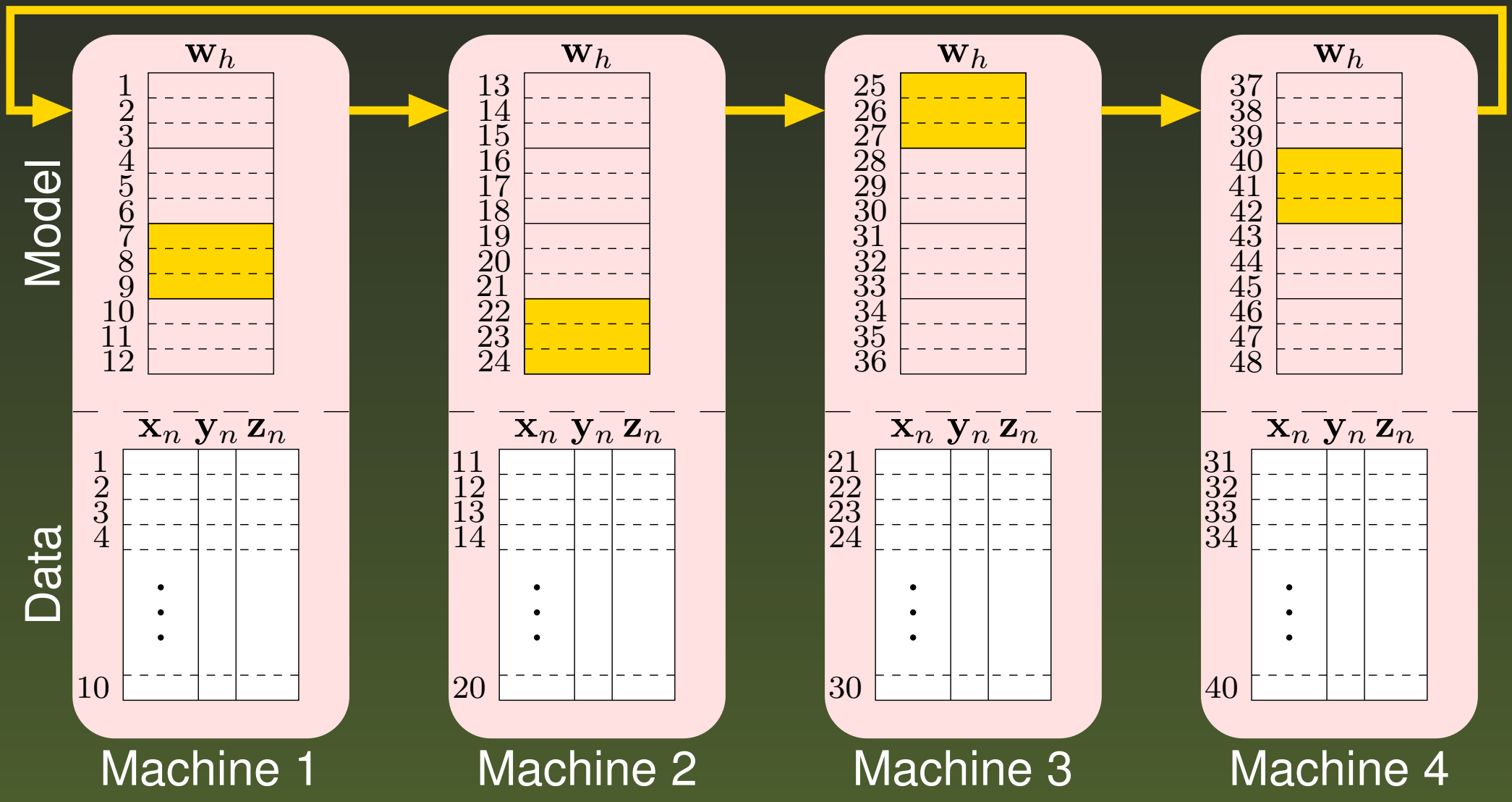
Computation + communication epoch, tick 0

ParMAC: the W step (cont.)



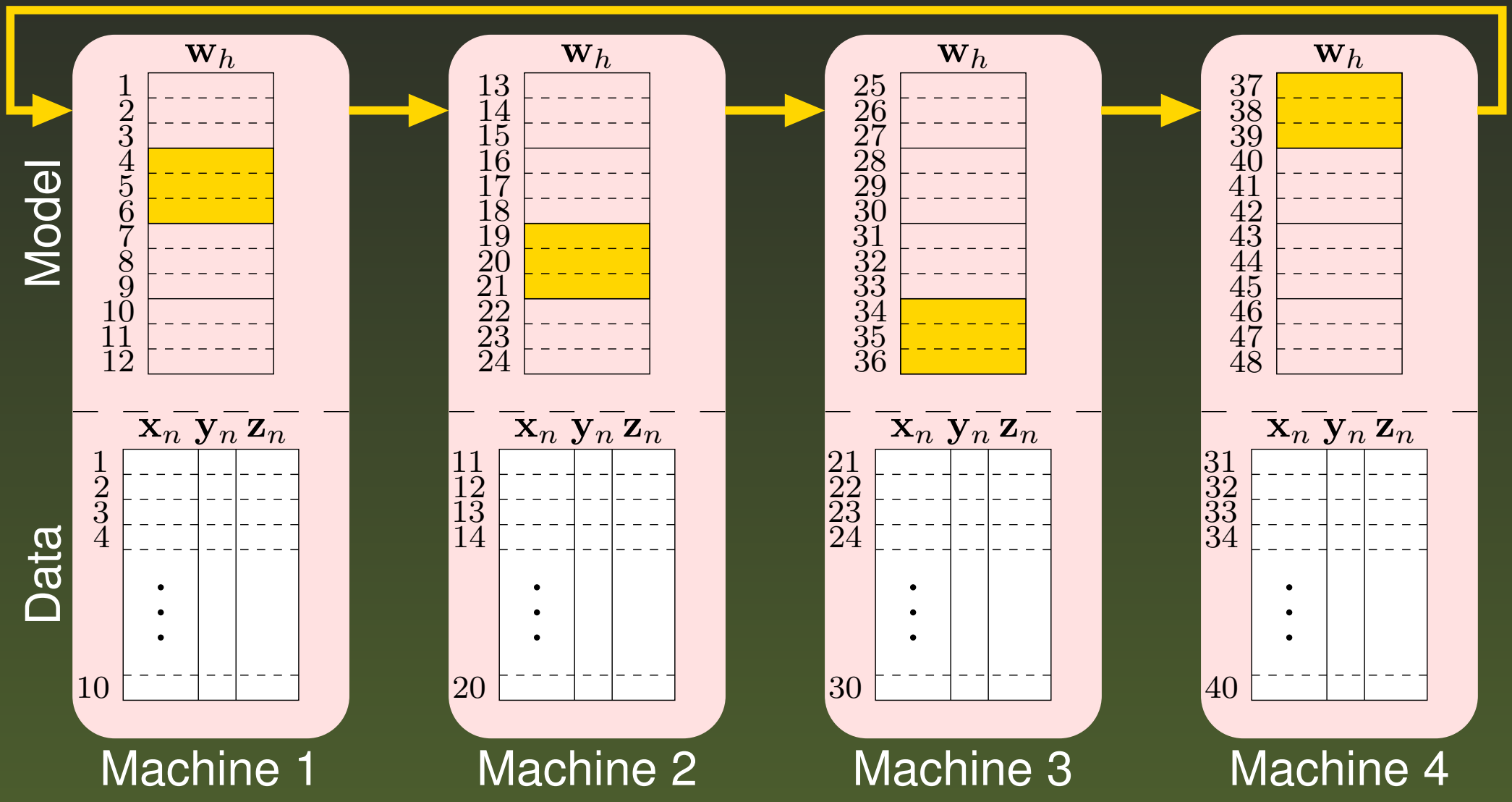
Computation + communication epoch, tick 1

ParMAC: the W step (cont.)



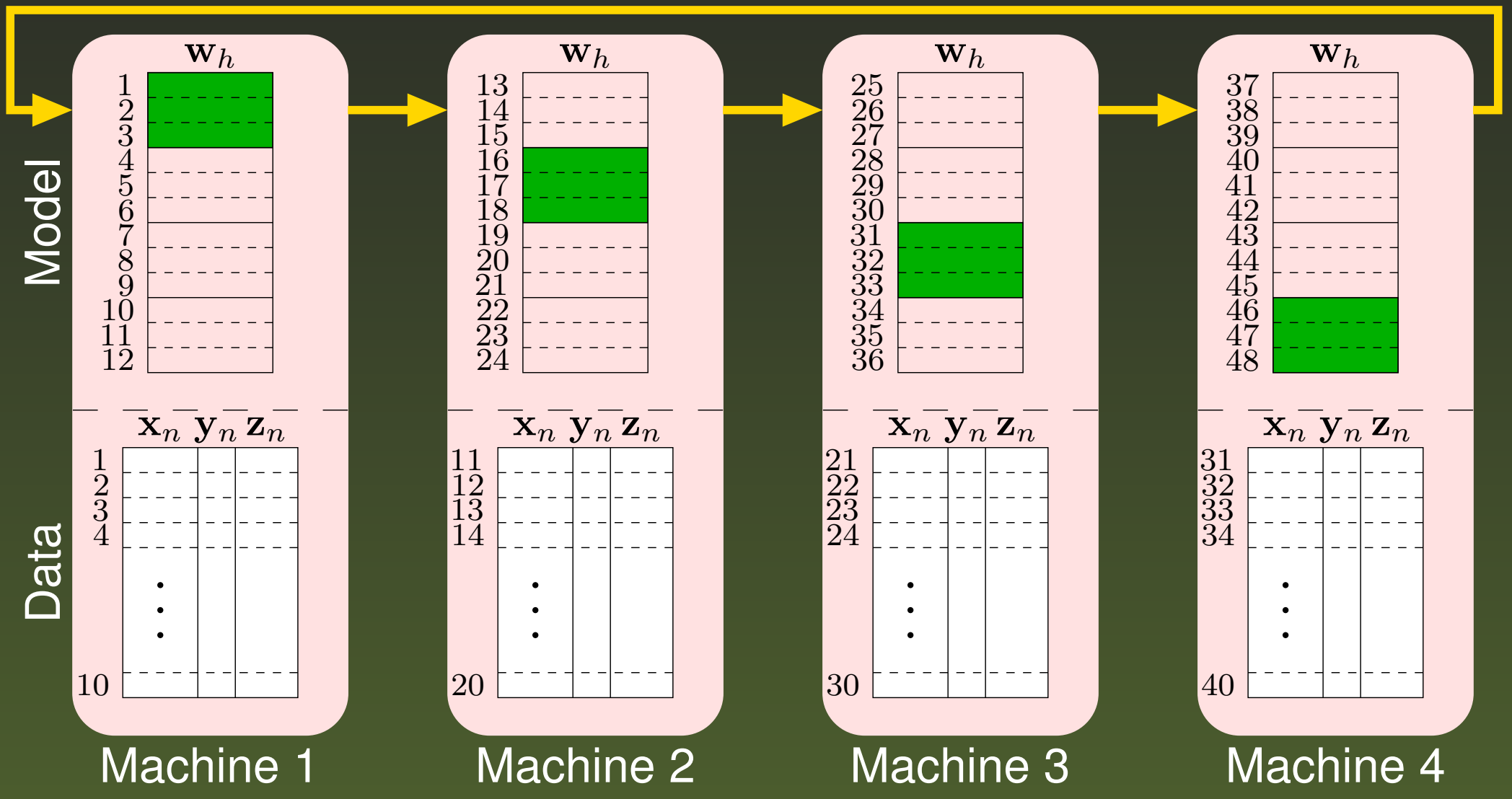
Computation + communication epoch, tick 2

ParMAC: the W step (cont.)



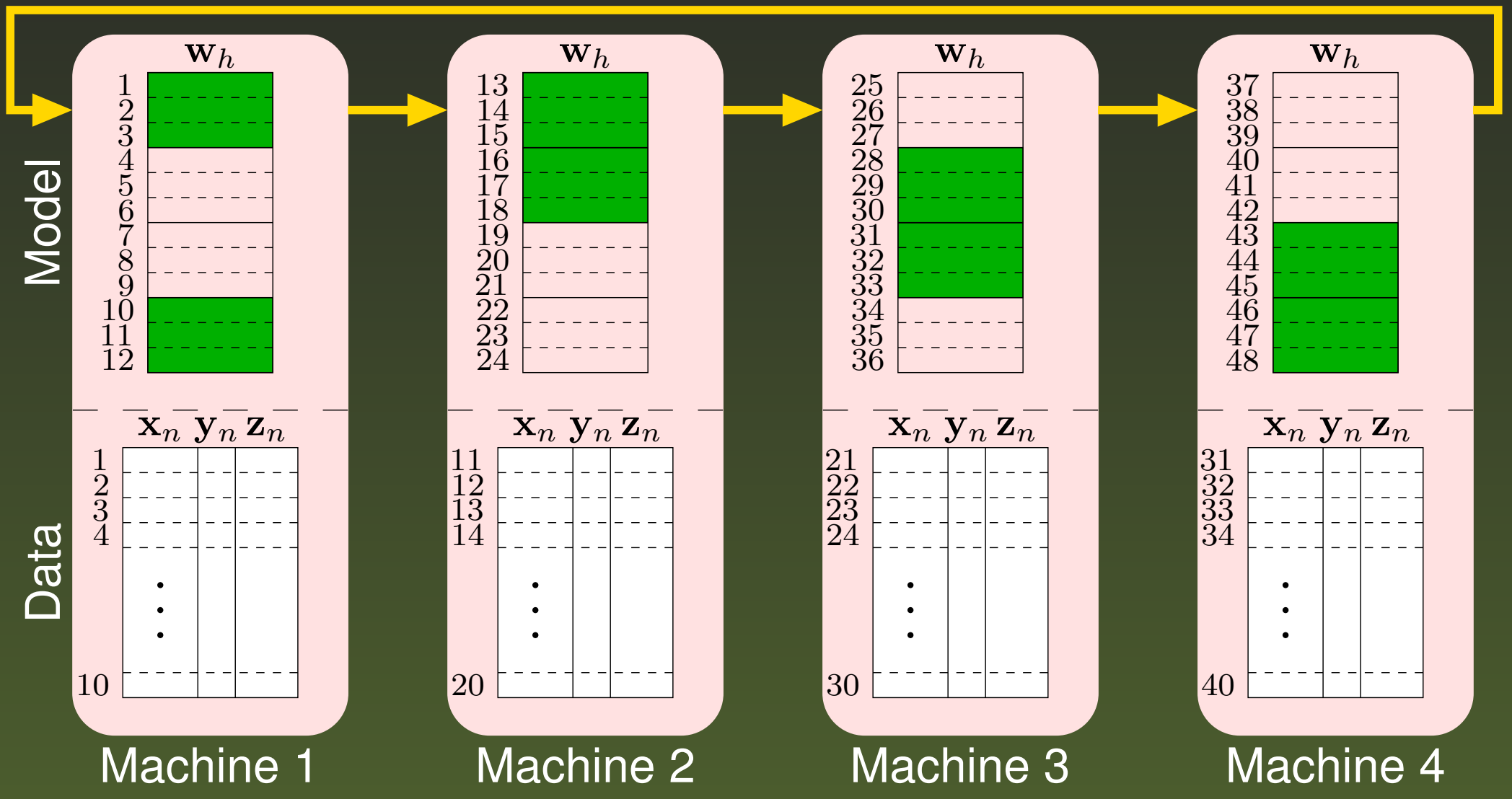
Computation + communication epoch, tick 3

ParMAC: the W step (cont.)



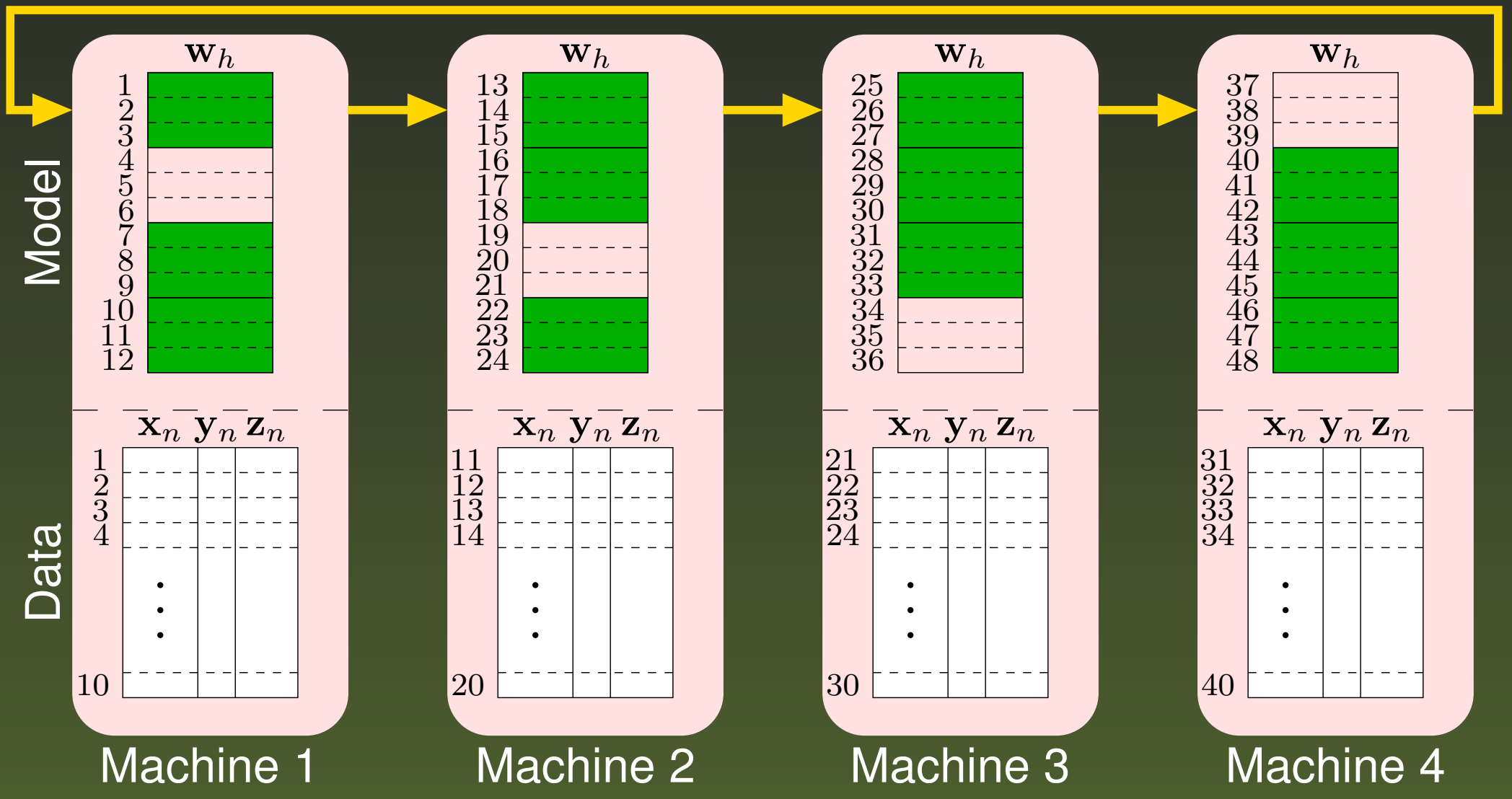
Computation + communication epoch, tick 4 $= P$
Communication-only (final) epoch, tick 0

ParMAC: the W step (cont.)



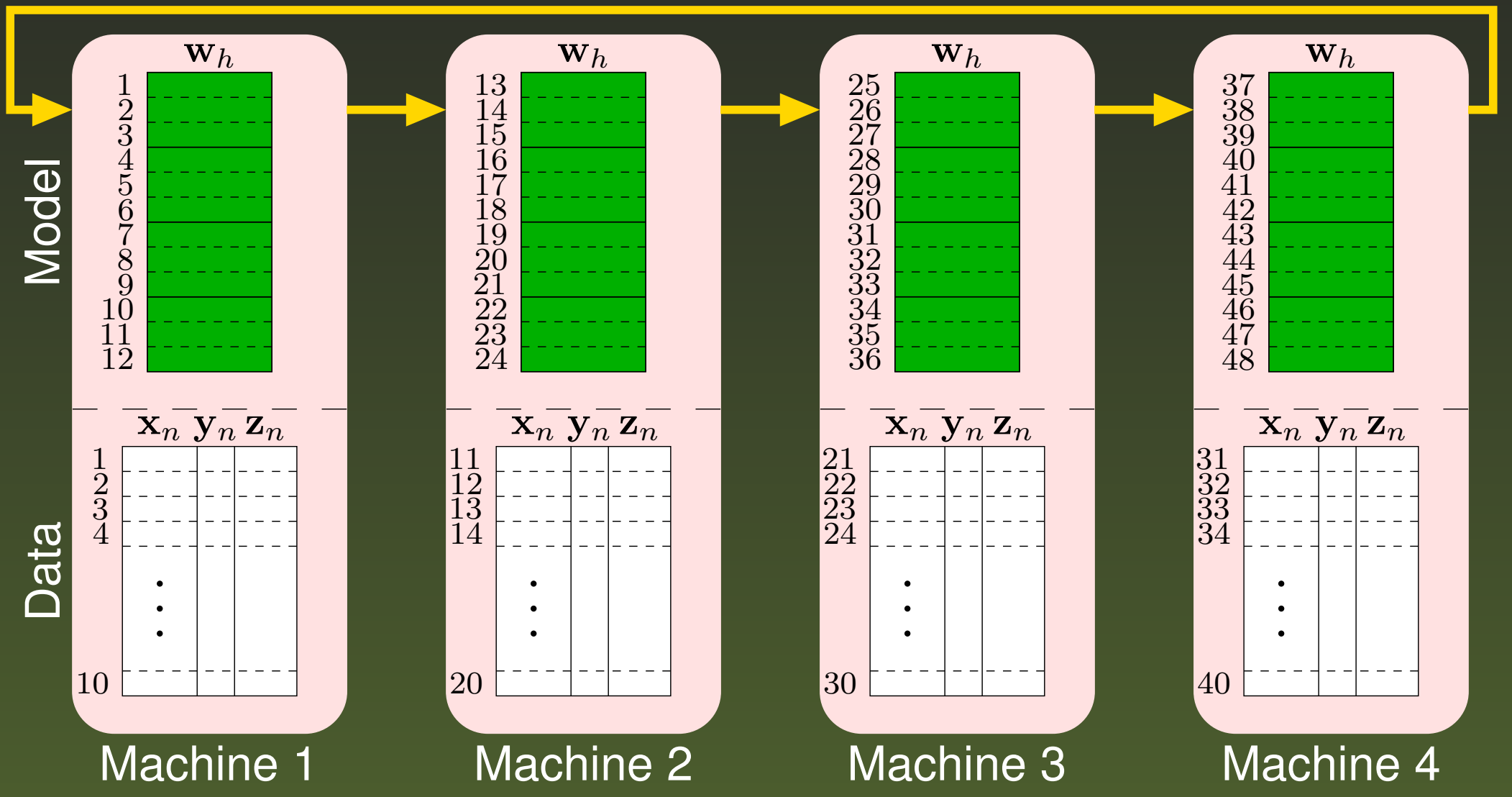
Communication-only (final) epoch, tick 1

ParMAC: the W step (cont.)



Communication-only (final) epoch, tick 2

ParMAC: the W step (cont.)



Communication-only (final) epoch, tick 3 $= P - 1$

ParMAC: the $\mathbf{W}$ step (cont.)

Since each submodel is updated as soon as it visits a machine, rather than computing the exact gradient once it has visited all machines and then take a step, the $\mathbf{W}$ step is really carrying out stochastic steps for each submodel in parallel.

- With e epochs, the entire model parameters are communicated $e+1$ times within the $\mathbf{W}$ step.

Practically, two rounds of communication are likely sufficient:

- Have a submodel do e consecutive passes within each machine's data, then communicate it.

  This achieves two rounds of communication with less shuffling, but likely small impact on convergence rate.

- With large datasets we probably need few epochs anyway, particularly with shallow models (logistic regression, linear SVM).

  Even less than one epoch, i.e., a model need not see the whole data to achieve a good enough error.

- Epochs accumulate over MAC iterations (which repeat the $\mathbf{W}$ step).

ParMAC: other issues

See paper:

- Convergence

  Essentially, same guarantees as for MAC.

- Extensions:
  - Data shuffling
  - Load balancing
  - Streaming data
  - Fault tolerance
- Circular vs parameter-server topologies

  Circular better than parameter-server.

Theoretical model of the parallel speedup

Useful to estimate the optimal number of machines P to use with a given problem, or to explore the effect on the speedup of different parameter settings (e.g. the number of submodels M).

Runtime $T(P)$: P machines, N data points, M submodels, e epochs

$$\text{Speedup } S(P) = \frac{T(1)}{T(P)} = \frac{\text{serial runtime}}{\text{parallel runtime}} = \frac{\rho \frac{1}{\lceil M/P \rceil} MP}{\frac{1}{N}P^2 + \rho_2 P + \rho_1 \frac{1}{\lceil M/P \rceil} M}$$

by defining the following ratios of computation vs communication time:

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

These depend on the actual computation in the $\mathbf{W}$ and $\mathbf{Z}$ step, and on the communication performance of the distributed system.

Different speedups are possible in different situations (values of ρ_1, ρ_2, P, N, M, e).

Theoretical model of the parallel speedup (cont.)

In practice, $\rho \ll 1$ because the communication time dominates the computation time in current architectures.

With a large dataset (N large wrt P, M) the speedup simplifies to:

$$S(P) \approx \begin{cases} P, & P \leq M \text{ (perfect speedup)} \\ \rho / \left(\frac{\rho_1}{P} + \frac{\rho_2}{M} \right), & P > M \text{ (weighted harmonic mean)} \end{cases}$$

and the maximum speedup S^* (over P) is achieved for $P = P^* > M$:

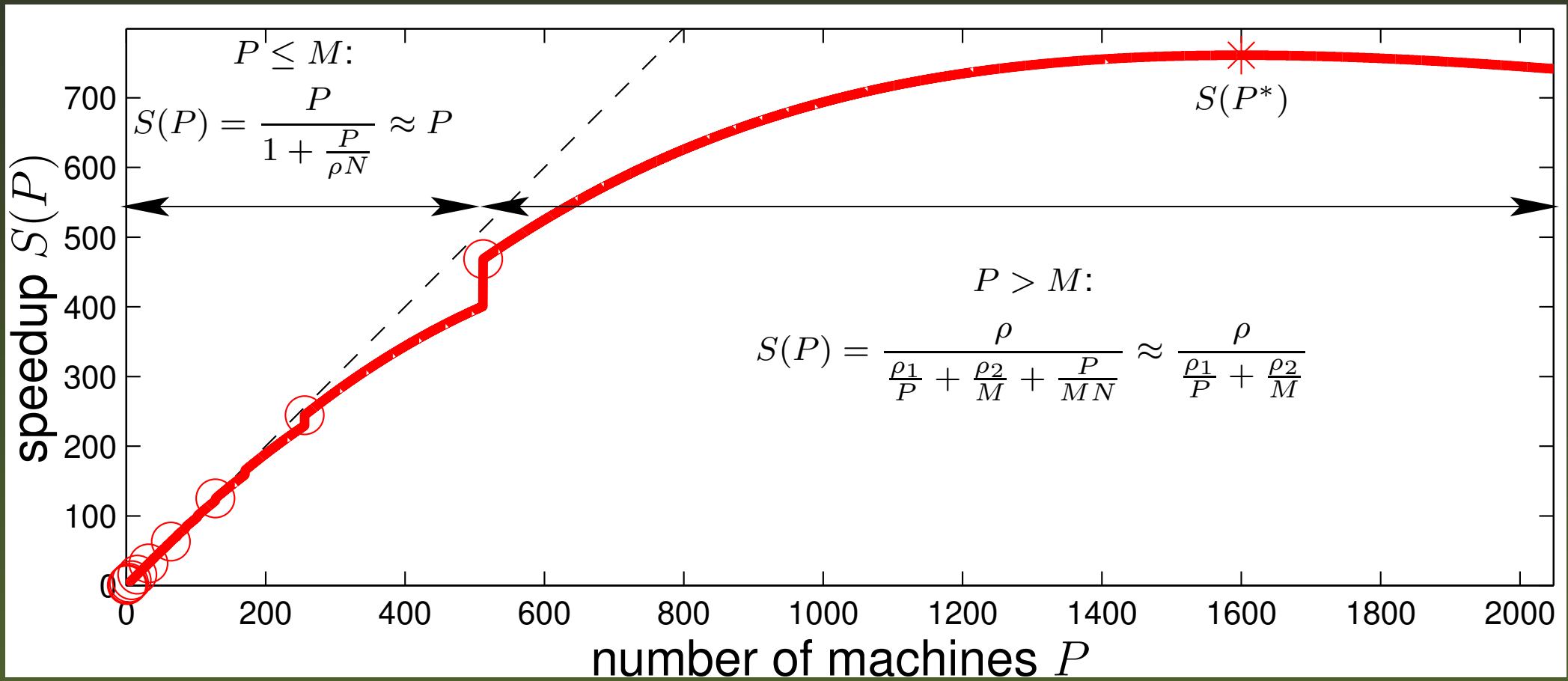
$$S^* = \frac{\rho M}{\rho_2 + 2\sqrt{\rho_1 M / N}} > M \quad \text{achieved at} \quad P^* = \sqrt{\rho_1 M N} > M.$$

So we expect very high parallel speedups with large datasets if $M \lesssim P$.

Theoretical model of the parallel speedup (cont.)

Typical theoretical speedup curve for realistic parameter settings.

$N = 1$M data points, $M = 512$ submodels, $e = 1$ epoch in the $\mathbf{W}$ step, $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} = 5$, $t_c^{\mathbf{W}} = 10^3$ (so $\rho_1 = 0.0025$, $\rho_2 = 0.0005$ and $\rho = 0.003$)



ParMAC: experiments with binary autoencoders

ParMAC implementation (e across-machine communication epochs):

- C/C++ using the GSL and BLAS libraries for numerical operations.
- Message Passing Interface (MPI) for interprocess communication.

Distributed-memory system: UCSD Triton Shared Computing Cluster (TSCC). Each node contains 2 8-core Intel Xeon E5-2670 processors (16 cores in total), 64GB DRAM (4GB/core) and a 500GB hard drive. Nodes connected through a 10GbE network. We used up to $P = 128$ processors.

Image retrieval datasets:

- CIFAR: $D = 320$ GIST features; $N = 50$k.
- SIFT: D=128 SIFT features; N=10k (SIFT-10K), 1M (SIFT-1M), 100M (SIFT-1B).
  SIFT-1B: largest(?) public benchmark for nearest-neighbour search with known ground-truth.

Binary autoencoder:

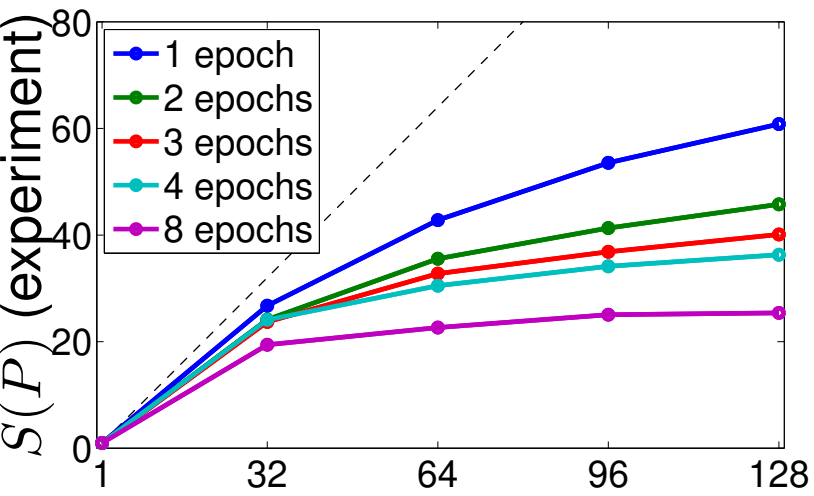
- Encoder: linear/RBF network (L SVMs), $L = \begin{cases} 16 \text{ (CIFAR, SIFT-1M)} \\ 64 \text{ (SIFT-1B)}. \end{cases}$
  Decoder: linear (D linear mappings).
- $\mathbf{W}$ step: SGD code from `http://leon.bottou.org/projects/sgd`.
- $\mathbf{Z}$ step: alternating optimisation over the bits of $\mathbf{z}_n \in \{0,1\}^L$.

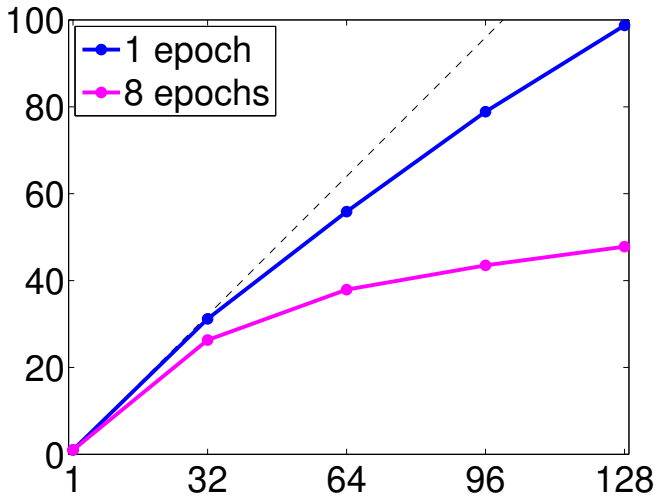
Experiments: speedup $S(P)$

CIFAR $N = 50$K, $M = 32$

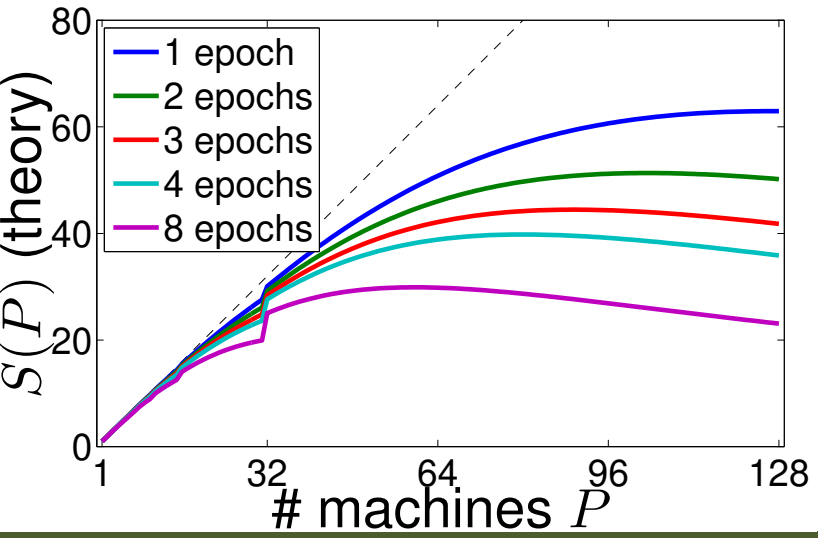
SIFT-1M $N = 10$M, $M = 32$

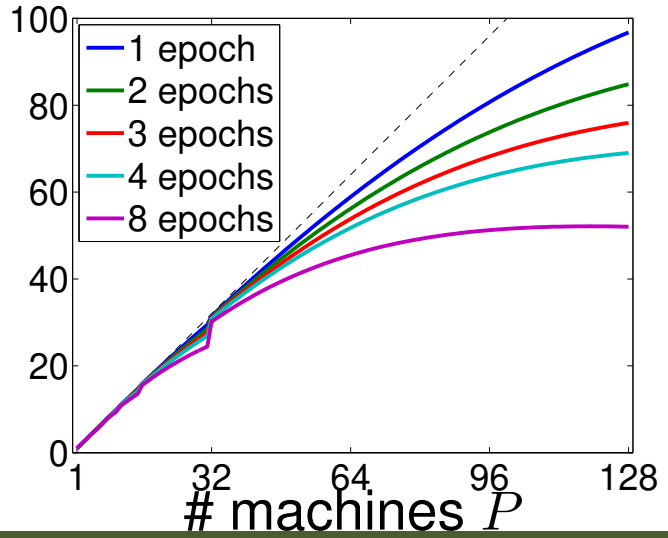
SIFT-1B $N=100$M, $M=128$

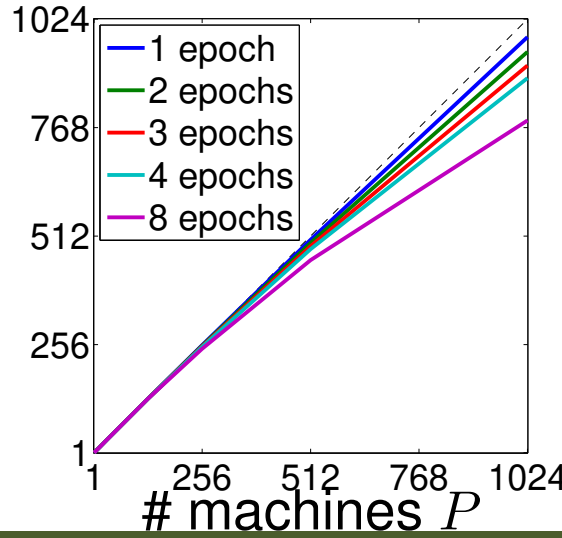




too long to run
in a single machine
(would take months)







Runtime on $P = 128$ machines:
- SIFT-1M ($e = 1$): 12', speedup 100×
- SIFT-1B ($e = 2$): 29h, speedup 128× (estimated).

Experiments: speedup $S(P)$ (cont.)

As a function of the # machines P for fixed problem size (dataset and model) (strong scaling), in the distributed memory system. In general:

- Theoretical speedups match experimental ones reasonably well.
  For BAs: $M = 2L$ effective submodels of the same size.
- The speedup is nearly perfect for $P \leq M$ and holds very well for $P > M$ up to the maximum # machines we used ($P = 128$ in the distributed system). Eventually it does decrease, of course.

Specifically for each dataset:

- CIFAR, SIFT-1M: the speedups flatten as the # epochs e (hence communication) increases, because for this experiment the bottleneck is the $\mathbf{W}$ step, whose parallelisation ability (# concurrent processes) is limited by M, which is small.
  Note small N and small M for CIFAR, so worse speedup than for SIFT-1M.
- SIFT-1B: theoretical speedup nearly perfect in whole range of P.

Conclusion: ParMAC

A distributed model for MAC for training nested, nonconvex models:

- MAC creates parallelism by introducing auxiliary coordinates for each data point to decouple nested terms in the objective function.
- ParMAC translates this parallelism to a distributed system by using:
  1. Data parallelism: each machine keeps a portion of the original data and its corresponding auxiliary coordinates.
  2. Model parallelism: independent submodels visit every machine in a circular topology, effectively doing SGD.
- Parallel speedup with large datasets:
  - nearly perfect when # submodels $\gtrsim$ # machines
  - eventually saturates as we continue to increase the # machines.
- Data shuffling, load balancing, streaming & fault tolerance.

Supported by NSF awards CAREER IIS–0754089, IIS–1423515 and a Google Faculty Research Award.