ParMAC: distributed optimisation of nested functions using auxiliary coordinates

Miguel Á. Carreira-Perpiñán

Electrical Engineering and Computer Science

University of California, Merced

`http://eecs.ucmerced.edu`

work with **Mehdi Alizadeh** and **Ramin Raziperchikolaei**

Outline

- Optimising nested (deep) systems using the method of auxiliary coordinates (MAC)
  - Chain-rule gradient, greedy layerwise training and MAC
  - Design pattern
  - Convergence guarantees
- Some examples of nested models trainable with MAC
  - Binary autoencoders (application: binary hashing for fast image search)
- Distributed optimisation with MAC: ParMAC
  - Circular and parameter-server topology
  - Convergence guarantees
  - A theoretical model of the parallel speedup
  - Implementation of ParMAC for binary autoencoders in MPI
  - Experiments in a computer cluster

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. It depends on the model, objective function, how the auxiliary coordinates are introduced, the penalty function, how the steps are optimised... We can achieve steps that are closed-form, convex, nonconvex, binary, etc.

But the following always hold:

- $\mathbf{Z}$ step: N independent subproblems $\mathbf{z}_1, \dots, \mathbf{z}_N$, 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
  - parameter server: implements parallel SGD.

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

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.

ParMAC: the $\mathbf{W}$ step, circular topology

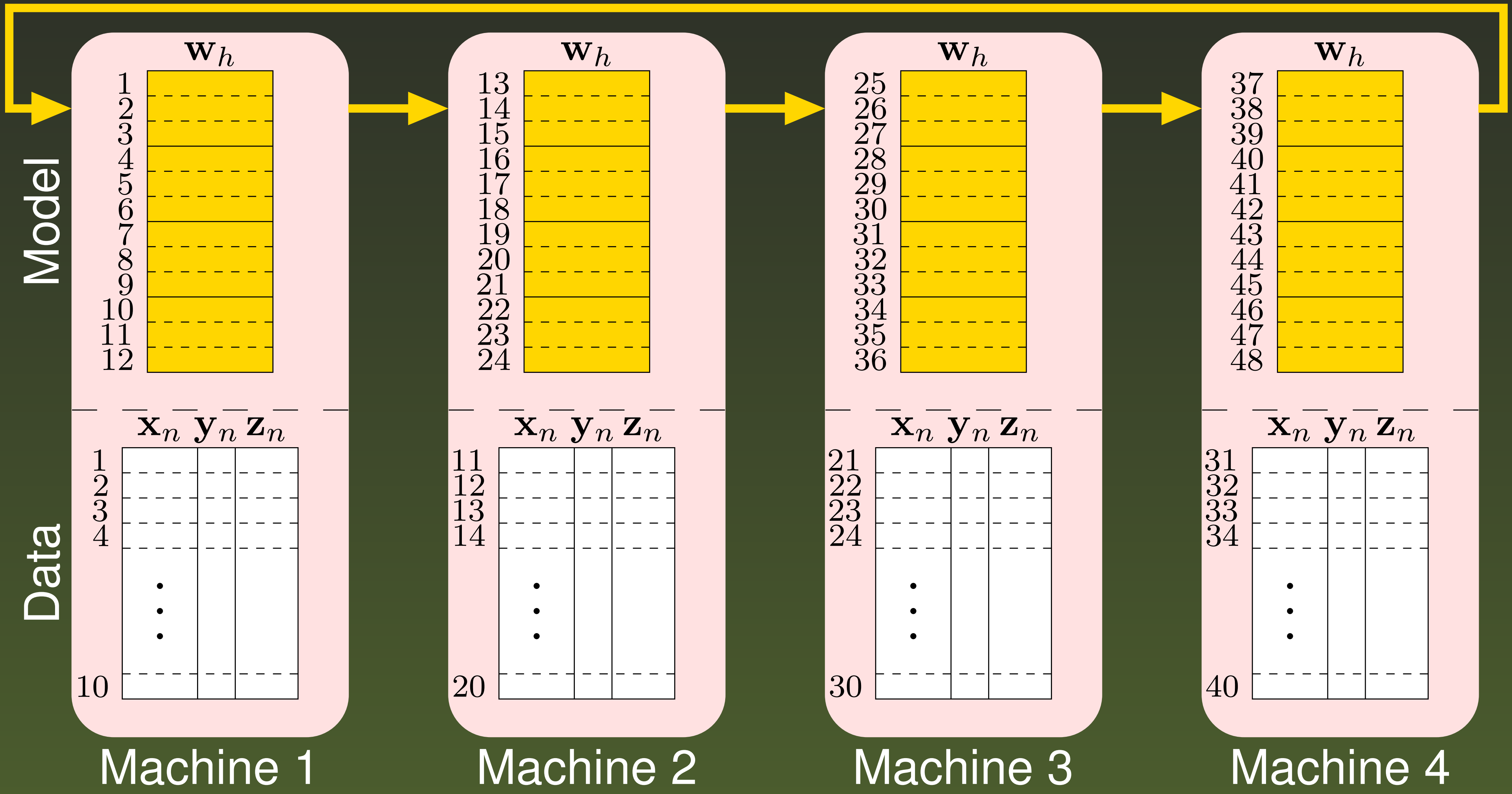
Synchronous version (easier to analyse but less practical):

- 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.

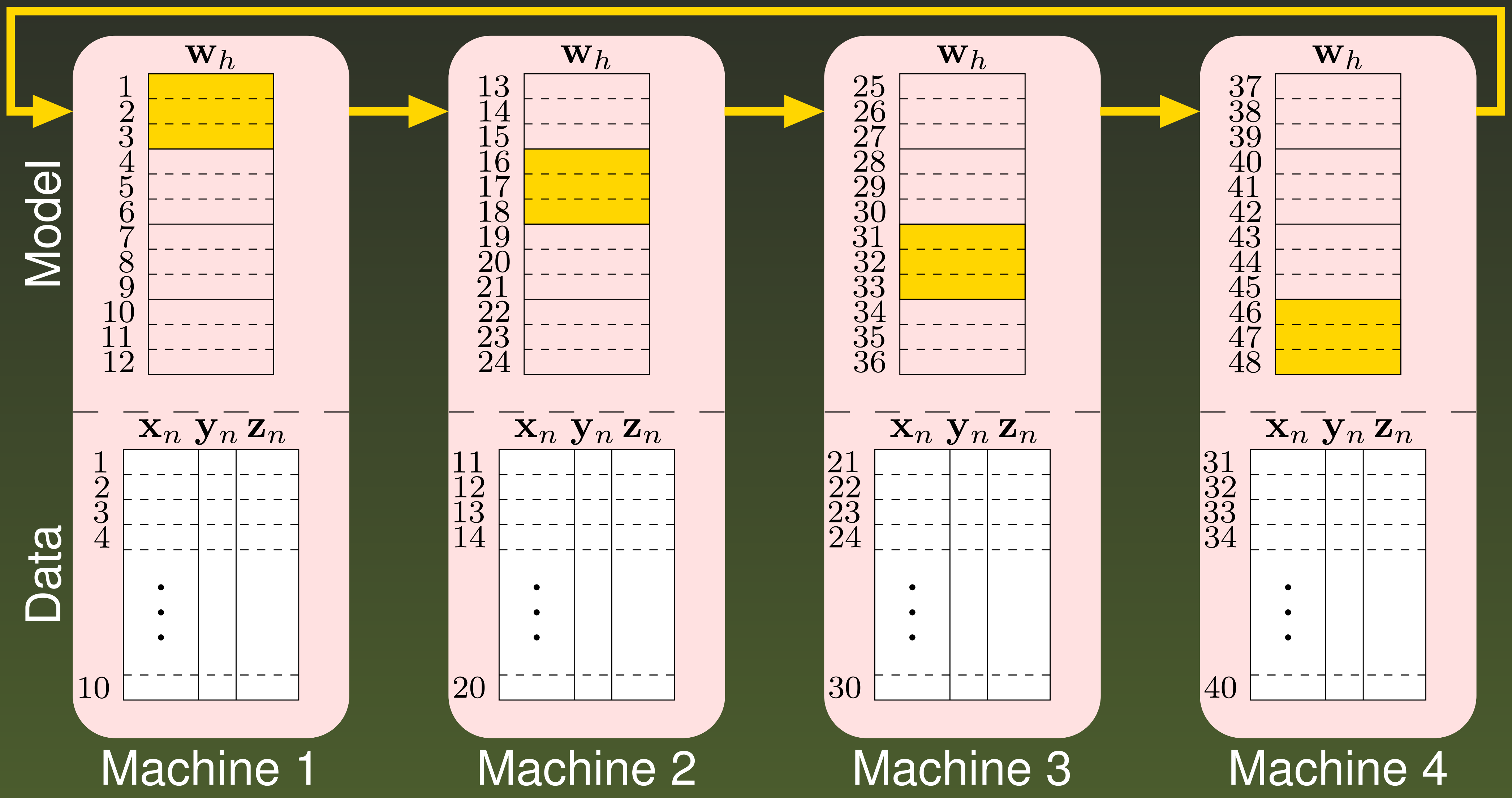
Asynchronous version. Each machine keeps a queue of submodels to be processed, and repeatedly performs the following operations: extract a submodel from the queue, process it (except in epoch $e+1$) and send it to the machine's successor (which will insert it in its queue).

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



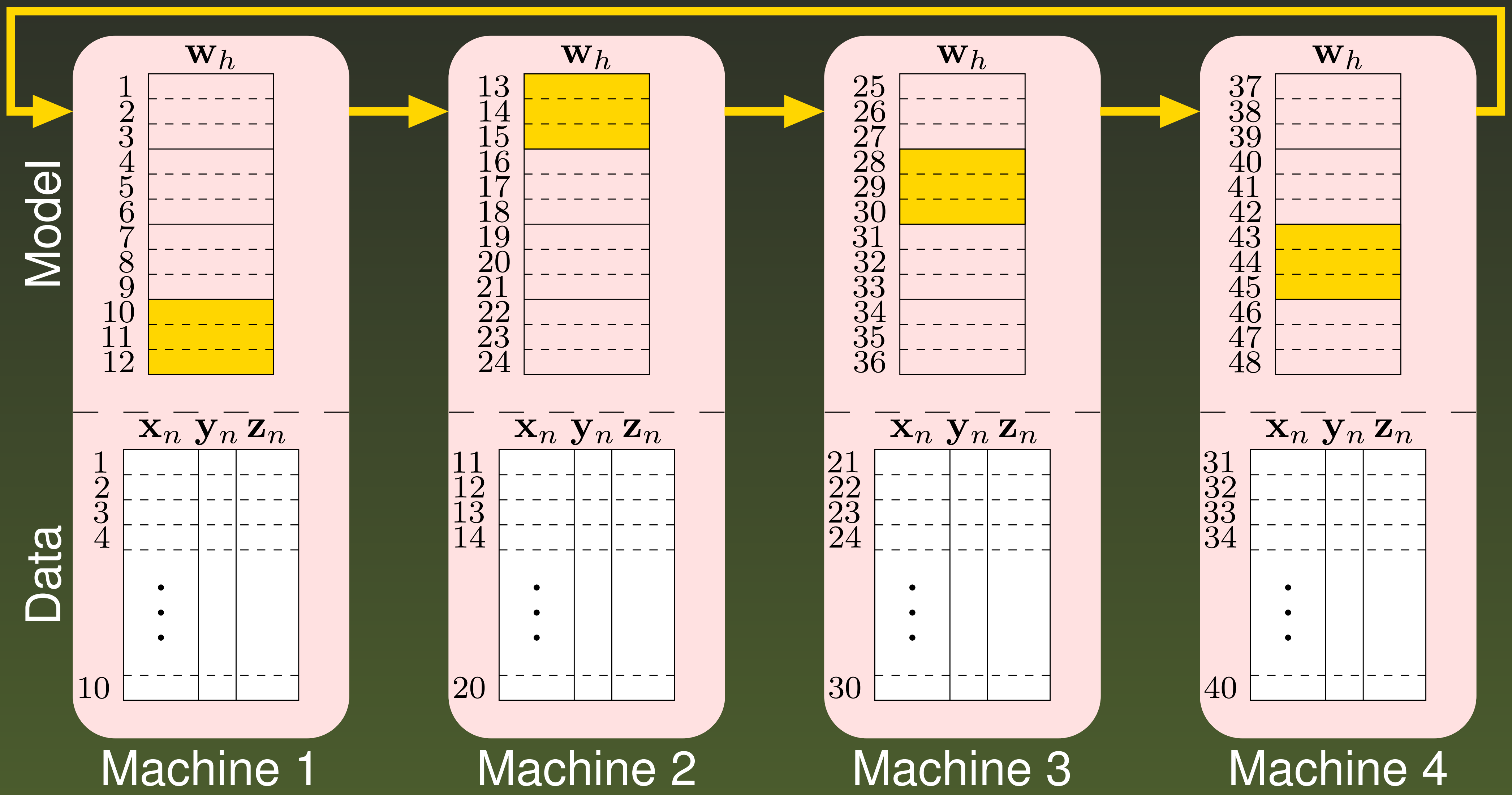
$P = 4$ nodes, $M = 12$ submodels $\mathbf{w}_h$ and $N = 40$ data points $(\mathbf{x}_n, \mathbf{y}_n)$. Submodels h, $h + M$, $h + 2M$ and $h + 3M$ are copies of submodel h, but only one of them is the most currently updated. At the end of the $\mathbf{W}$ step all copies are identical.

ParMAC: the W step, circular topology (cont.)



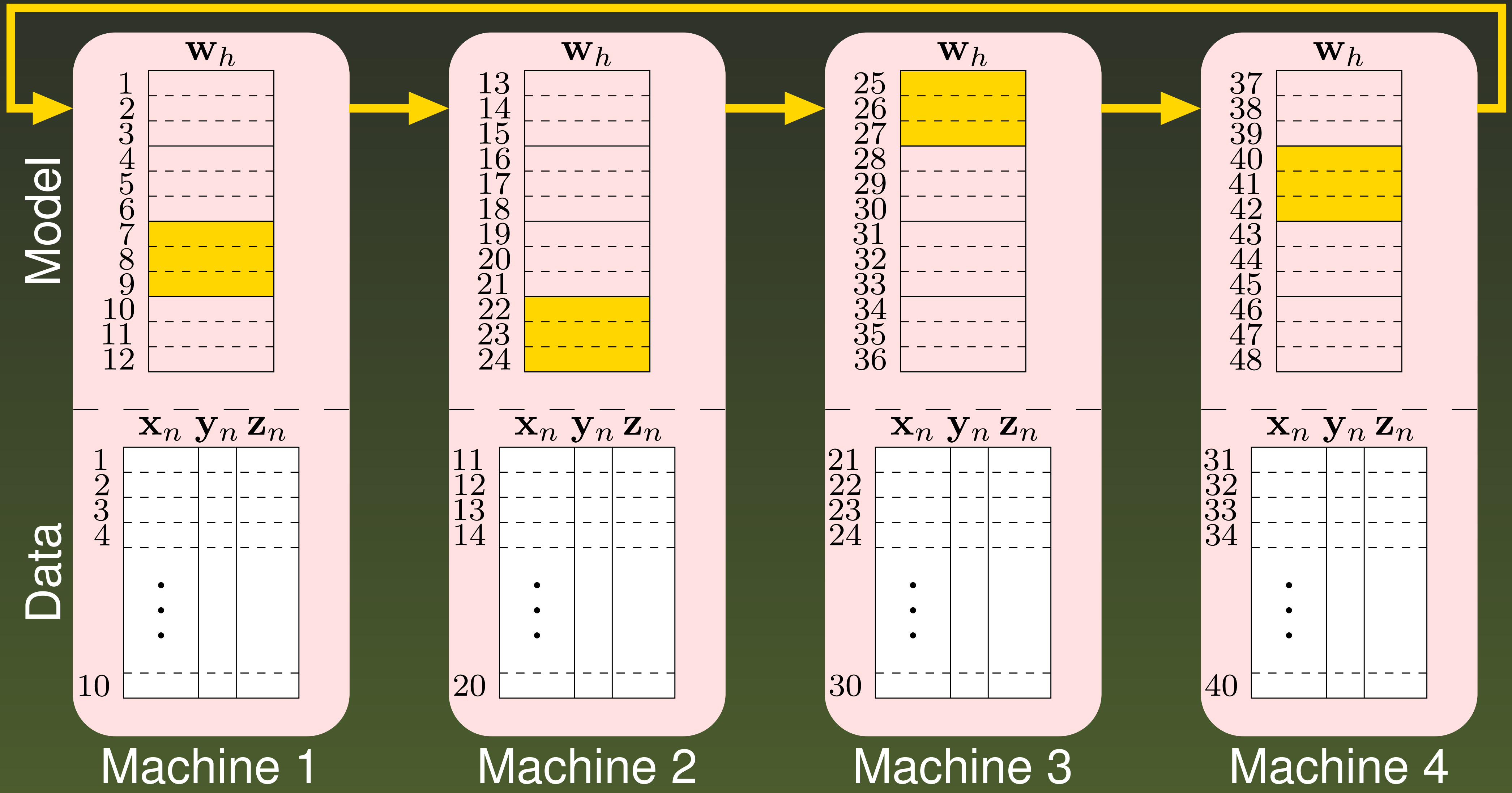
Computation + communication epoch, tick 0

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



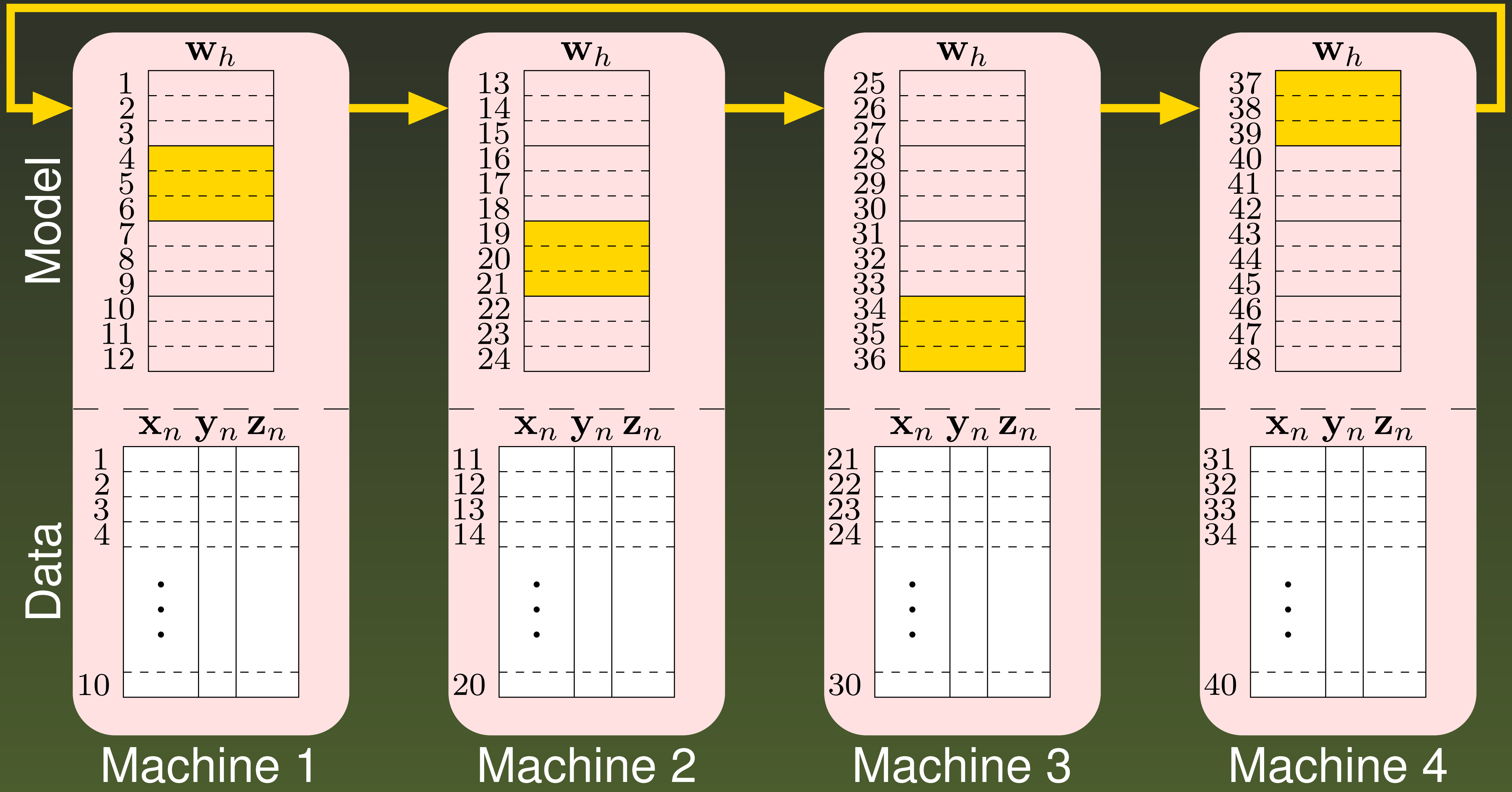
Computation + communication epoch, tick 1

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



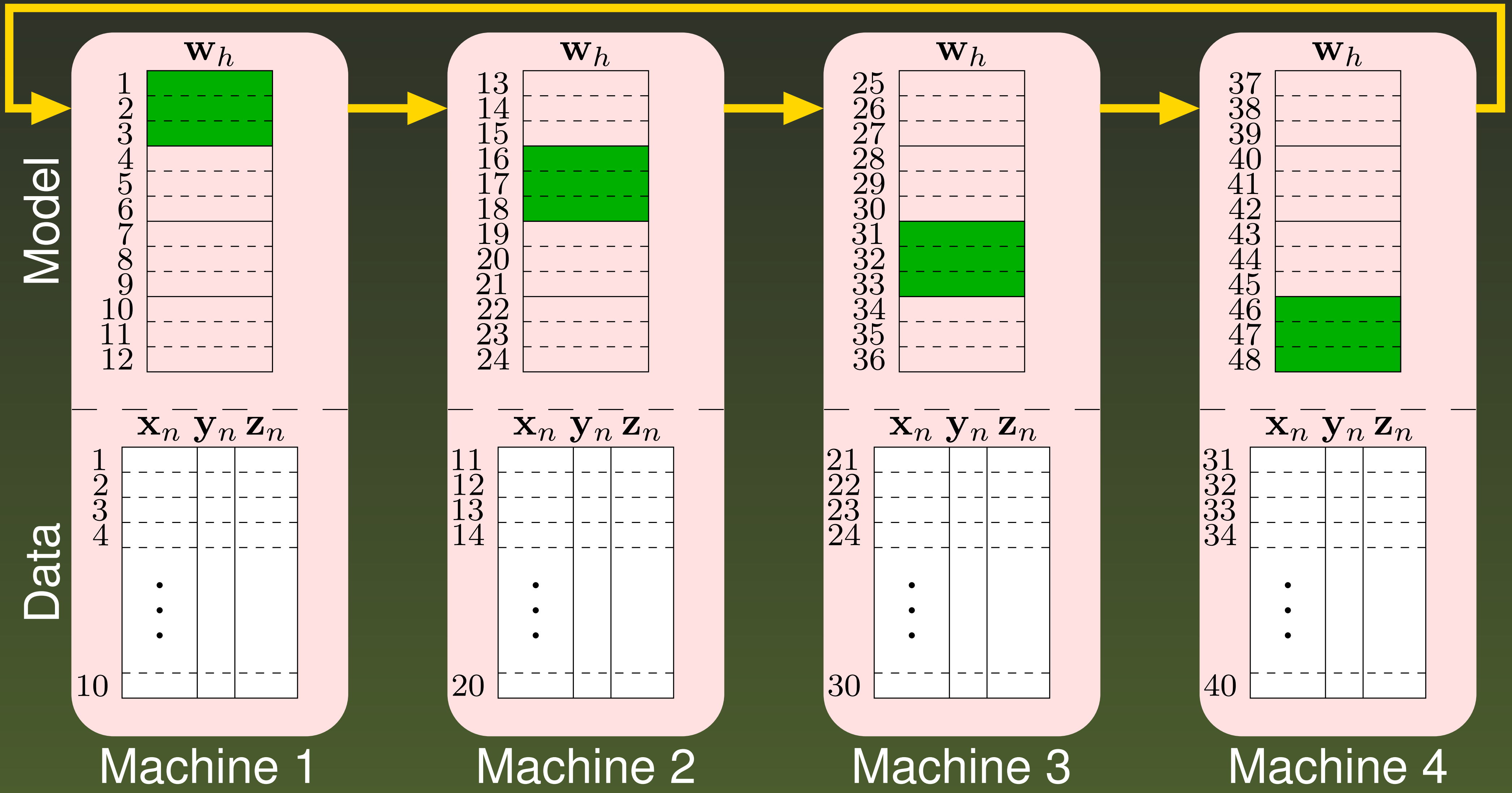
Computation + communication epoch, tick 2

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



Computation + communication epoch, tick 3

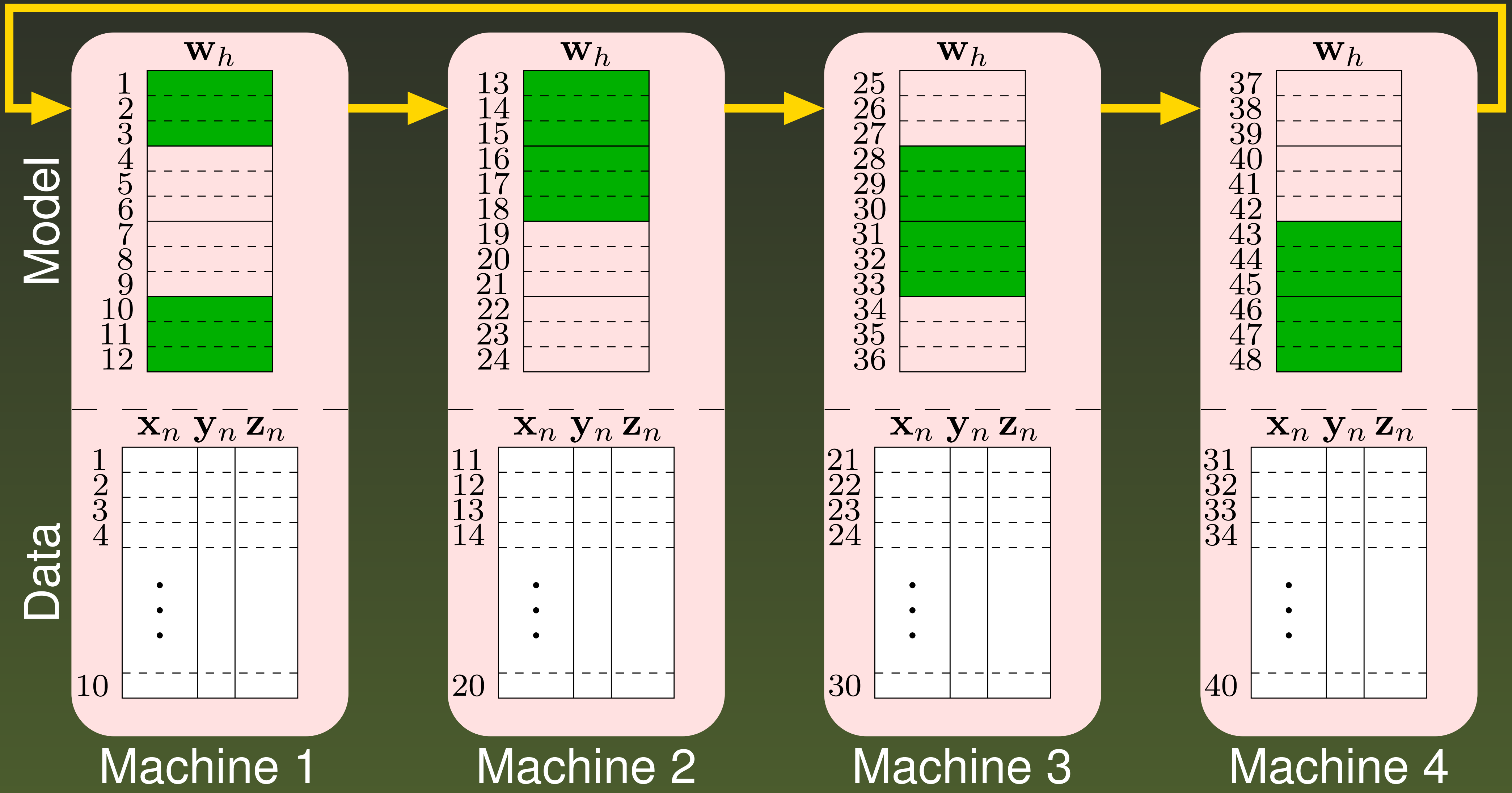
ParMAC: the $\mathbf{W}$ step, circular topology (cont.)



Computation + communication epoch, tick $4 = P$

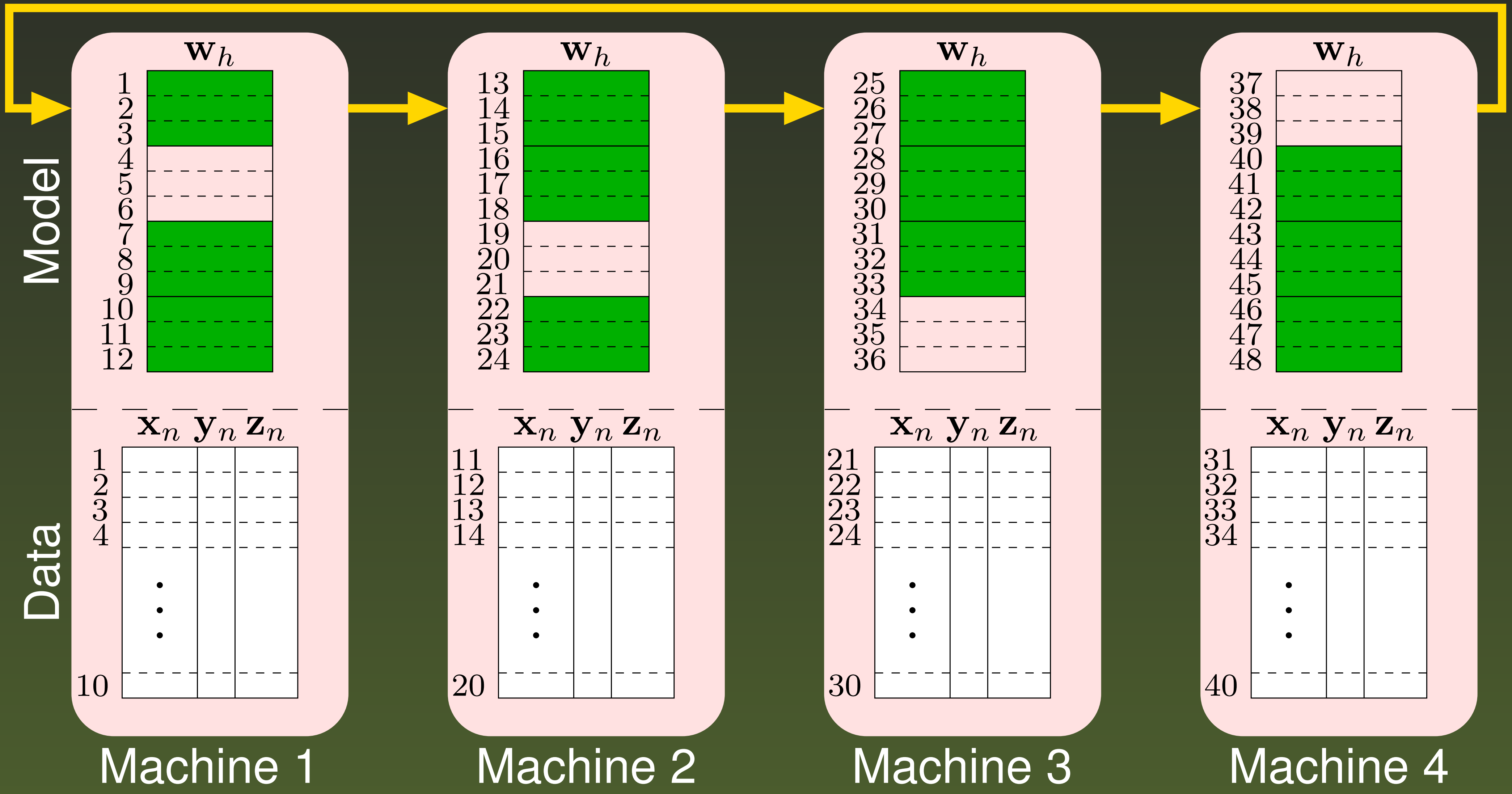
Communication-only (final) epoch, tick 0

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



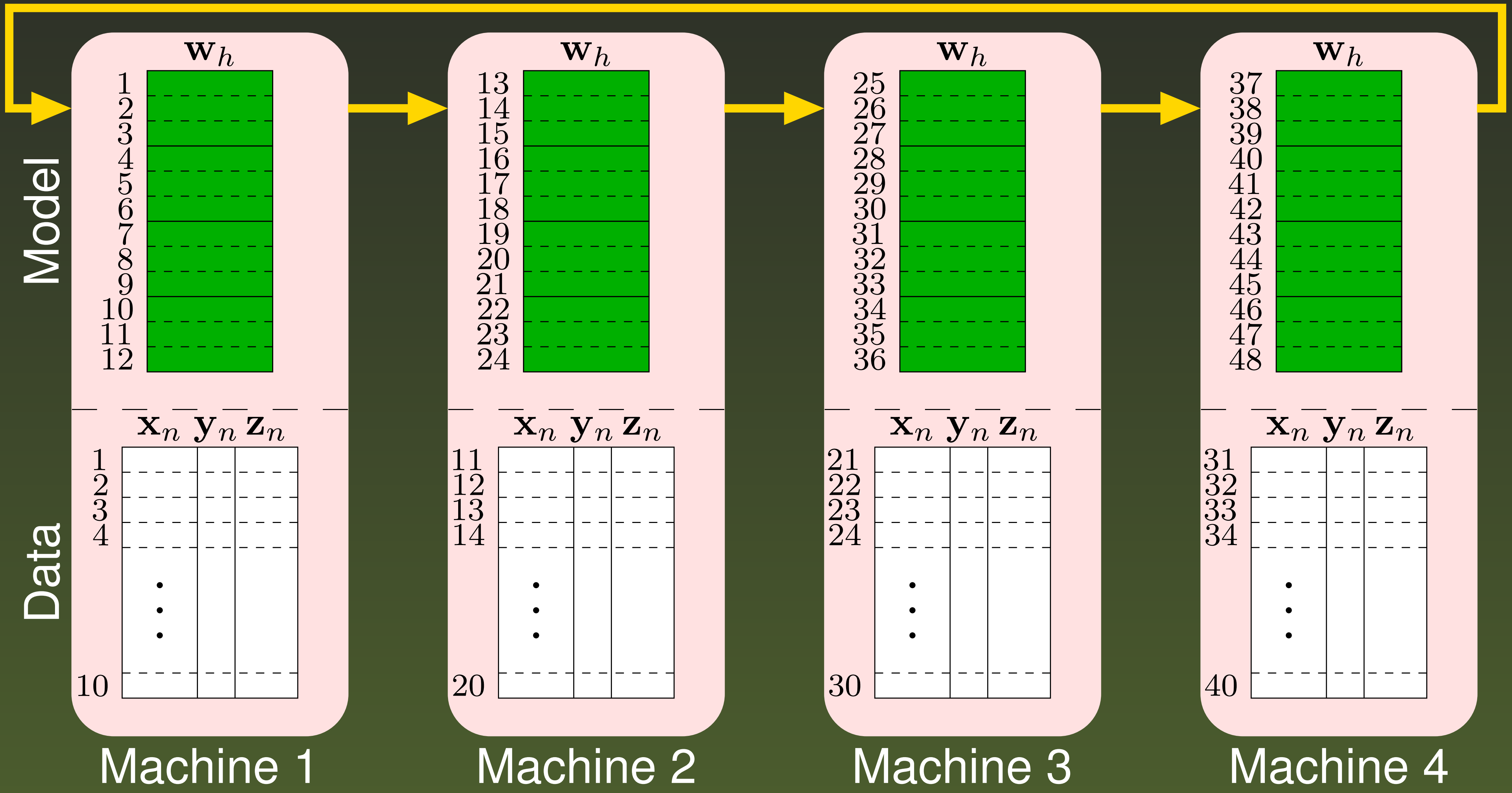
Communication-only (final) epoch, tick 1

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



Communication-only (final) epoch, tick 2

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



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

ParMAC: the $\mathbf{W}$ step, circular topology (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).

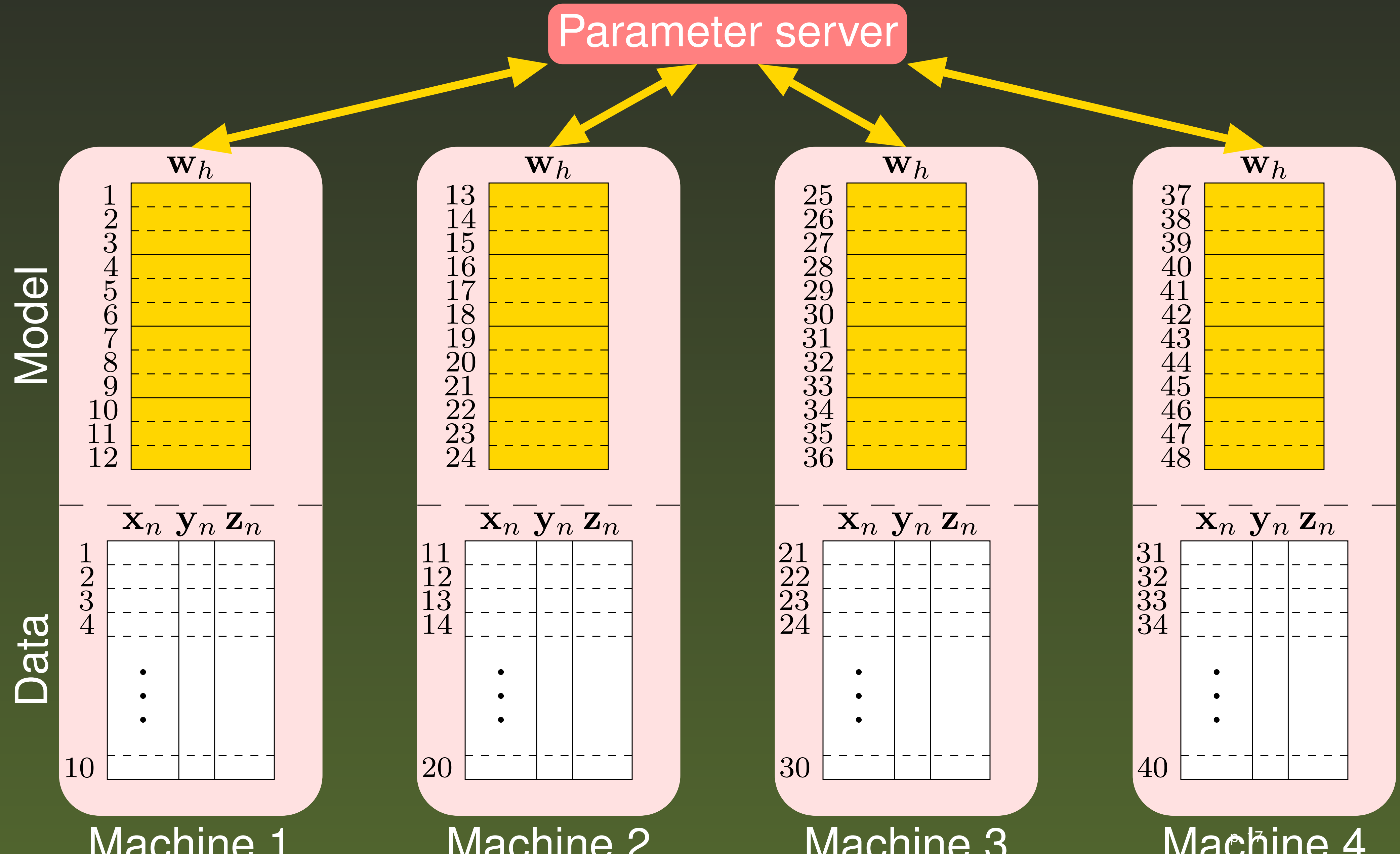
ParMAC, circular topology: extensions

- Data shuffling:
  - Visit data points within each machine at random.
  - Can also randomise the circular topology at each epoch.
- Load balancing:
  - $\mathbf{W}$, $\mathbf{Z}$ steps: work proportional to the number of data points N.
  - If the processing power of machine p is proportional to $\alpha_p > 0$, allocate to it $N\alpha_p/(\alpha_1 + \cdots + \alpha_P)$ data points.
- Streaming: new data added or old data discarded:
  - within a machine, by adding or removing data points to it
  - by adding or removing a machine to the topology

    needs support from the parallel processing library.
- Fault tolerance: similar to removing a machine, but unintended:
  - discard the machine, reconnect circular topology
  - if in $\mathbf{W}$ step, revert submodels lost in the faulty machine to its predecessor's copy of those submodels

ParMAC: the W step, parameter server topology

We do parallel SGD on each submodel independently.



Parallel SGD with a parameter server

- P workers each with N/P points.
- Each worker runs SGD on its own model for a while.
- Workers send their model to the server regularly. The server averages the models (in parameter space) and broadcasts them back to the workers. This can be done synchronously or asynchronously.

Convergence with convex problems:

- For a small enough step size η and some technical conditions.
- Parallelisation stagnates past a certain P.

  Zinkevich et al 2010: expected distance to minimum $\leq \frac{a\eta}{\sqrt{P}} + \frac{b\eta}{P} + c\eta \xrightarrow[P\to\infty]{} c$.
- In practice, the speedups over serial SGD are generally modest.

Convergence with nonconvex problems:

- It does not converge in general. Averaging can fail with local optima.
- With some care, it can work in practice, but this is not easy.

  Average models that are close in parameter space and thus associated with the same optimum.

ParMAC: convergence

MAC theorem (assuming differentiable functions, for quadratic penalty):

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

The key requirement is to make the gradient of the MAC-penalised function smaller than a set tolerance τ_k, i.e., stay close enough to the path $(\mathbf{W}^*(\mu), \mathbf{Z}^*(\mu))$ that converges (as $\mu \to \infty$) to a local solution.

In MAC, this is achieved by iterating sufficiently the $\mathbf{W}$ and $\mathbf{Z}$ step optimisations with suitable algorithms.

How can this be achieved in ParMAC's $\mathbf{W}$ step?

ParMAC: convergence (cont.)

Circular topology:

- Under standard conditions for SGD, even if nonconvex submodels.

  Robbins-Monro schedule on the learning rate.
- Tighter conditions when the subproblems in the $\mathbf{W}$ step are convex.

  Bound the distance to the minimum in objective function value.

⇒ Convergence of ParMAC to a local stationary point guaranteed by the same theorem as MAC, *for convex and nonconvex submodels*, with an added SGD-type condition for the $\mathbf{W}$ step.

Parameter-server topology:

- Convex $\mathbf{W}$ step: guaranteed by using a parallel SGD condition.
- Nonconvex $\mathbf{W}$ step: no guarantees.

⇒ Convergence of ParMAC to a local stationary point guaranteed by the same theorem as MAC, *but only for convex submodels*, with an added parallel-SGD-type condition for the $\mathbf{W}$ step.

Distributed EM / k-means

ParMAC (with circular or parameter-server topology) applies just as well to the EM algorithm. Ex: for a mixture of M Gaussians:

- E step: set the posterior probabilites for each point, $\mathbf{p}_1, \dots, \mathbf{p}_N$. Like the $\mathbf{Z}$ step (closed-form solution).
- M step: set the M proportions, means & covariances (submodels) independently from each other as an average over the data. Like the $\mathbf{W}$ step (closed-form form solution in terms of sufficient statistics).

Since the M step is closed-form:

- A single epoch suffices to complete the M step.
- The distributed EM behaves identically to the serial EM, and has the same convergence guarantees.

Circular vs parameter-server topology in the $\mathbf{W}$ step

Run time $\begin{cases} \text{Circular:} & \left(t_r^{\mathbf{W}} \frac{N}{P} + 2t_c^{\mathbf{W}}\right) \lceil \frac{M}{P} \rceil P \\ \text{Parameter-server:} & \left(t_r^{\mathbf{W}} \frac{N}{P} + 2t_c^{\mathbf{W}} \frac{P}{C}\right) M \end{cases}$ where:

- $t_r^{\mathbf{W}}$: computation time per submodel and data point
- $t_c^{\mathbf{W}}$: machine-machine communication time per submodel
- $C \in [1, P]$: # workers the parameter server can communicate with in parallel. System-dependent; can also use several parameter servers.

So the computation time (minibatch updates within each machine) is the same, but the parameter-server topology has more communication, and some additional disadvantages:

- parallel SGD converges more slowly than true SGD
- difficult to apply if the $\mathbf{W}$ step is nonconvex
- needs an extra machine to act as parameter server.

Both topologies differ in how they employ the available parallelism:

- circular: update different, independent submodels
- parameter-server: update replicas of the same submodels

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 $T(P)$ using P machines.
- Speedup $S(P) = T(1)/T(P) =$ (serial runtime) / (parallel runtime).

Consider a ParMAC algorithm:

- on a dataset with N training points, distributed over P identical machines (each with N/P points)
- with M independent submodels of the same size in the $\mathbf{W}$ step
- using a circular topology in the $\mathbf{W}$ step

  parameter server: same model if using $t_c^{\mathbf{W}} \frac{P}{C}$
- operating synchronously.

We ignore small overheads (setup and termination) and estimate the total runtime as proportional to the number of iterations.

Theoretical model of the parallel speedup (cont.)

Parameters of the theoretical model of the speedup:

- $P \geq 1$: number of machines.
- $N \geq 1$: number of training points. Assume $N > P$ is divisible by P.
- $M \geq 1$: number of submodels in the $\mathbf{W}$ step. May be $<$, $=$, $> P$.
- $e \geq 1$: number of epochs in the $\mathbf{W}$ step.
- $t_r^{\mathbf{W}} > 0$: computation time per submodel & data point in the $\mathbf{W}$ step.

  Time to process (within the current epoch) one data point by a submodel, i.e., the time do an SGD update to a weight vector, per data point.
- $t_c^{\mathbf{W}} > 0$: communication time per submodel in the $\mathbf{W}$ step.

  Time to send one submodel from one machine to another, including overheads (buffering, partitioning into messages or waiting time). We assume communication does not overlap with computation.
- $t_r^{\mathbf{Z}} > 0$: computation time per data point in the $\mathbf{Z}$ step.

  Time to finish one data point entirely, using whatever optimisation algorithm performs the $\mathbf{Z}$ step.

We assume $t_r^{\mathbf{W}}$, $t_c^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}$ are constant and equal for every submodel and data point. Not true in reality…

Theoretical model of the parallel speedup (cont.)

Runtime in each step:

- $\mathbf{Z}$ step: $T^{\mathbf{Z}}(P) = M\frac{N}{P}t_r^{\mathbf{Z}}$.

  The time for any one machine to process its N/P points on all M submodels.

- $\mathbf{W}$ step: $T^{\mathbf{W}}(P) = \lceil M/P \rceil \left(t_r^{\mathbf{W}}\frac{N}{P} + t_c^{\mathbf{W}}\right) Pe + \lceil M/P \rceil t_c^{\mathbf{W}} P$.

  Make M divisible by P by adding fictitious submodels that do work but are discarded at the end.
  Runtime in each tick (there are Pe ticks):

  - $\lceil M/P \rceil \frac{N}{P} t_r^{\mathbf{W}}$: time for any machine to process its $\frac{N}{P}$ points on its portion $\lceil \frac{M}{P} \rceil$ of submodels.
  - $\lceil M/P \rceil t_c^{\mathbf{W}}$: time for any machine to send its portion of submodels.

  $\lceil M/P \rceil t_c^{\mathbf{W}} P$: time of the final round of communication.

Total runtime $T^{\mathbf{W}}(P) + T^{\mathbf{Z}}(P)$ per ParMAC iteration with P machines:

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

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

Theoretical model of the parallel speedup (cont.)

Parallel speedup:

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

by defining the following ratios of computation vs communication:

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

- independent of training set size, # submodels, # machines
- depend on the actual computation in the $\mathbf{W}$ and $\mathbf{Z}$ step (optimisation); and on the performance of the distributed system

  computation power of each machine, communication speed over the network or shared memory, efficiency of the parallel processing library that handles the communication between machines.
- In practice $\rho, \rho_1, \rho_2 \ll 1$ (communication much slower than computation).

  Their actual values can vary considerably depending on the problem.

Theoretical model of the parallel speedup (cont.)

Analysis of the speedup model: various situations possible depending on the parameters. The most practical one is the

large dataset case: $P \ll \rho_2 N, \; M < \rho_1 N.$

Then the speedup is:

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

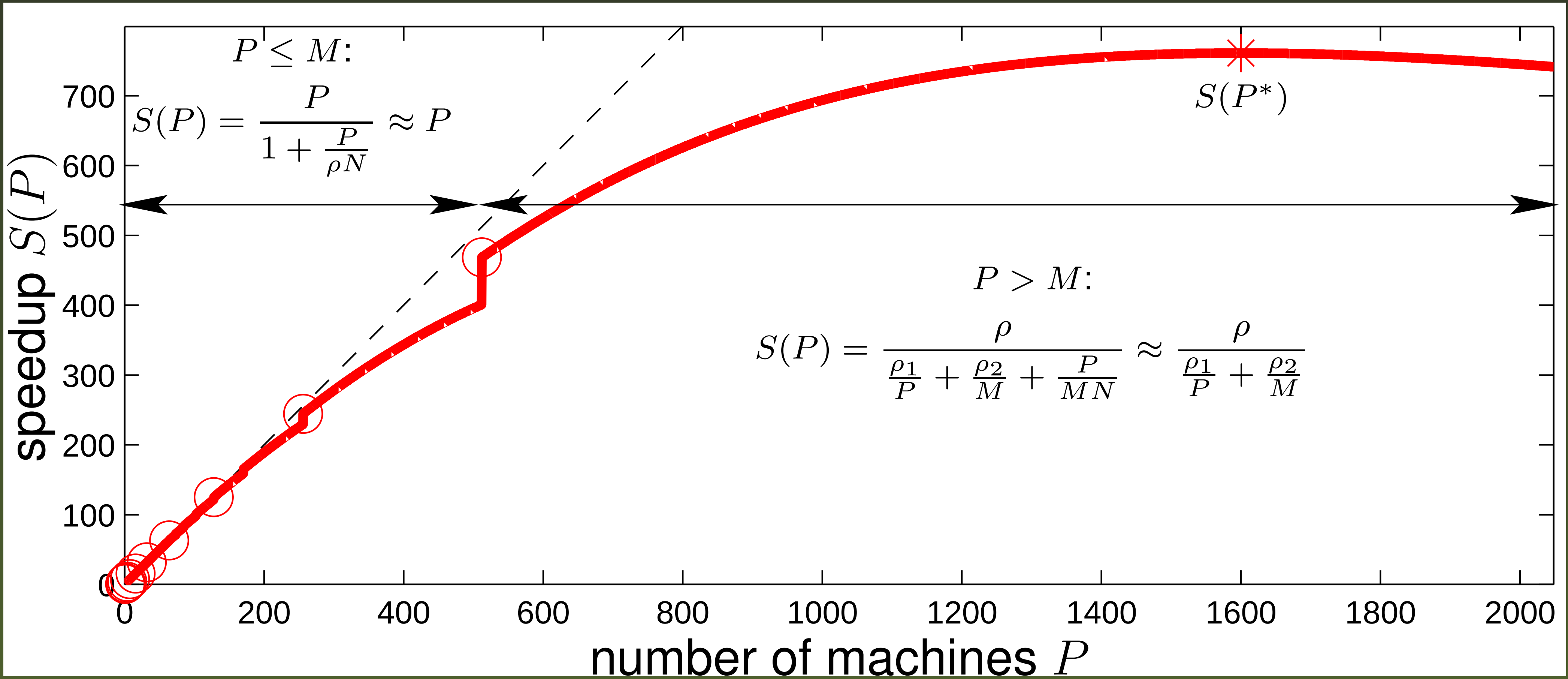
and the maximum speedup is achieved for $P > M$:

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

Theoretical model of the parallel speedup (cont.)

Typical theoretical speedup curve for realistic parameter settings.

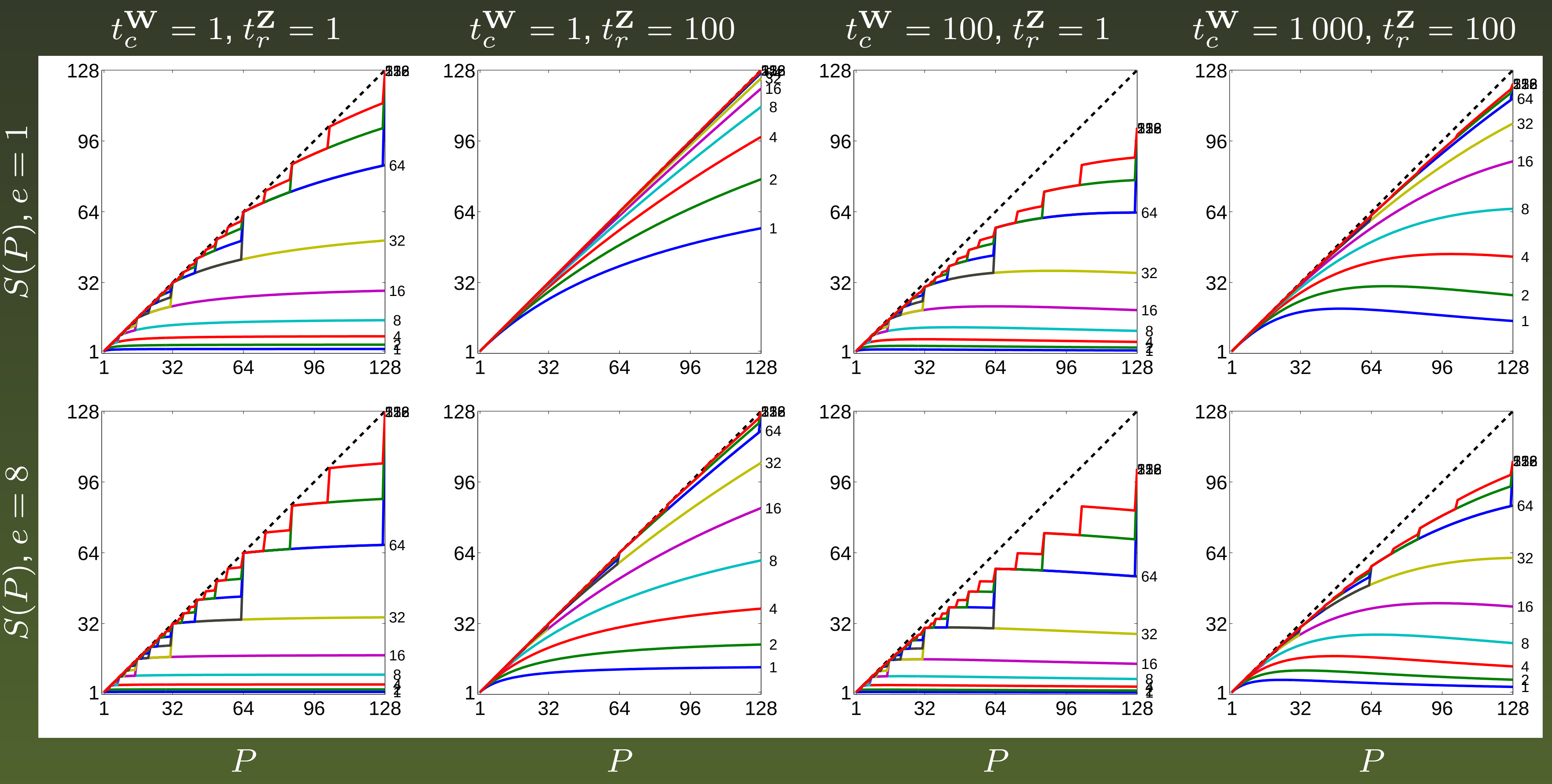
$N = 1$M data points, $M = 512$ submodels, $e = 1$ epoch in the $\mathbf{W}$ step, $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} = 5$, $t_c^{\mathbf{W}} = 10^3$ (so $\rho_1 = 0.0025$, $\rho_2 = 0.0005$ and $\rho = 0.003$)



Theoretical model of the parallel speedup (cont.)

Theoretical speedup curves for other parameter settings.

$N = 50\text{K}$, $M \in \{2^0, \dots, 2^9\}$, $e \in \{1, 8\}$, $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} \in \{1, 100\}$, $t_c^{\mathbf{W}} \in \{1, 100, 1\,000\}$



Theoretical model of the parallel speedup (cont.)

Practical considerations:

- The speedup is unchanged by trading off dataset size (N) and computation/communication times ($t_r^{\mathbf{W}}$, $t_r^{\mathbf{Z}}$, $t_c^{\mathbf{W}}$) in various ways.

  Scaling by $\alpha > 0$:

$N, t_r^{\mathbf{W}}, t_r^{\mathbf{Z}}$	$\rightarrow$	$\alpha N, \frac{1}{\alpha} t_r^{\mathbf{W}}, \frac{1}{\alpha} t_r^{\mathbf{Z}}$	*larger dataset, faster computation*
$N, t_c^{\mathbf{W}}$	$\rightarrow$	$\alpha N, \alpha t_c^{\mathbf{W}}$	*larger dataset, slower communication*
$t_r^{\mathbf{W}}, t_r^{\mathbf{Z}}, t_c^{\mathbf{W}}$	$\rightarrow$	$\alpha t_r^{\mathbf{W}}, \alpha t_r^{\mathbf{Z}}, \alpha t_c^{\mathbf{W}}$	*faster computation, faster communication.*

- Pick M divisible by P if $P < M$ (more efficient parallelism, no machines ever idle).
- Machines of different power: data load proportional to power.
- Models of different sizes: group models to equalise them.

  Binary autoencoder with L encoders and D decoders: group the D decoders into L groups of D/L decoders each → total $M = 2L$ same-size submodels.

- Other considerations: cost of machines, etc.

Parameter-server topology: replace $t_c^{\mathbf{W}}$ with $t_c^{\mathbf{W}} \frac{P}{C}$ in $S(P)$.

ParMAC: implementation for binary autoencoders

- We implemented the circular topology.
- C/C++ using the GSL and BLAS libraries for numerical operations.
- Message Passing Interface (MPI) for interprocess communication:
  - Different processes cannot directly access each other's memory space. Data is transferred by sending messages from one process to another, or collectively among multiple processes.
  - Widely used for high-performance parallel computing.
  - Several implementations available, such as MPICH or OpenMPI.
- Basic code structure:
  - Initialise/finalise MPI environment at start/end of code.
  - To receive data: synchronous blocking `MPI_Recv`.

    Process calling `MPI_Recv` blocks until data arrives.
  - To send data: buffered blocking `MPI_Bsend`.

    Allocate memory and attach it to the system. Process calling `MPI_Bsend` blocks until the buffer is copied to the MPI internal memory. After that, the MPI library takes care of sending the data.

ParMAC: implementation for binary autoencoders (cont.

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

for (iter=1 to length($\mu$)) {  // ——————————————————

  // begin $\mathbf{W}$-step
  visitedsubmodels = 0;
  // each process visits all the submodels, epochs + 1 times
  while (visitedsubmodels <= totalsubmodelcount*epochs) {
    // stepcounter indicates how far trained each submodel is
    if (stepcounter > 0) {  // not 1st submodel? wait to receive
      // MPI_Recv blocks until requested data is available
      MPI_Recv(receivebuffer, commbuffsize,
        MPI_CHAR, MPI_ANY_SOURCE, MODEL_MSG_TAG,
        MPI_COMM_WORLD, &recvStatus);
      savesubmodel(receivebuffer);
    }
```

```
      if (stepcounter < epochs*mpisize) {    // not in last round
        switch(submodeltype)    // train submodel according to type
        case 'SVM': HtrainSGD();
        case 'linlayer':  FtrainSGD();
      }
      if (stepcounter < (ringepochs+1)*mpisize) {
        // pick the successor process from the lookup table
        successor = next_in_lookuptable();
        loadsubmodel(sendbuffer);
        MPI_Bsend(sendbuffer, taskbufsize*sizeof(double),
          MPI_CHAR, successor, MODEL_MSG_TAG, MPI_COMM_WORLD);
      }
      visitedsubmodels++;
    }
    // end $\mathbf{W}$-step

    // begin $\mathbf{Z}$-step
    updateZ();                          // optimise auxiliary coordinates
    // end $\mathbf{Z}$-step

  }    // ——————————————————————————

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

Experiments

ParMAC with circular topology, e communication epochs (no within-machine epochs).

Computing systems:

- Distributed-memory: 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.
- Shared-memory: 72-processor machine with 256GB RAM at UC Merced. Processors communicate through shared memory. We used up to $P = 64$ proc.

Image retrieval datasets:

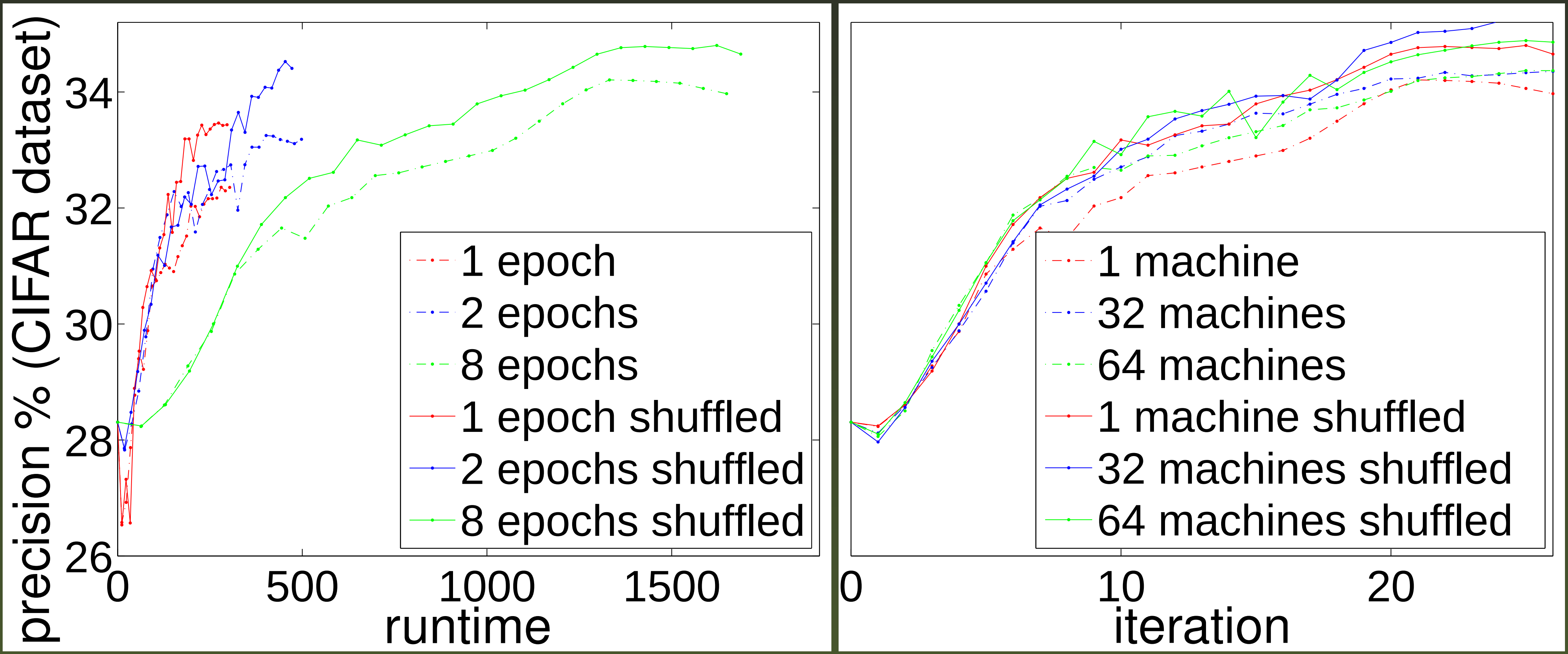
- CIFAR: $D = 320$ GIST features; $N =$ 50k.
- SIFT: $D = 128$ SIFT features; $N =$ 10k, 1M (SIFT-1M), 100M (SIFT-1B).

  SIFT-1B: largest(?) public benchmark for nearest-neighbour search with known ground-truth.

Binary autoencoder:

- Encoder: linear or RBF network (L SVMs). 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: effect of stochastic steps in the $\mathbf{W}$ step



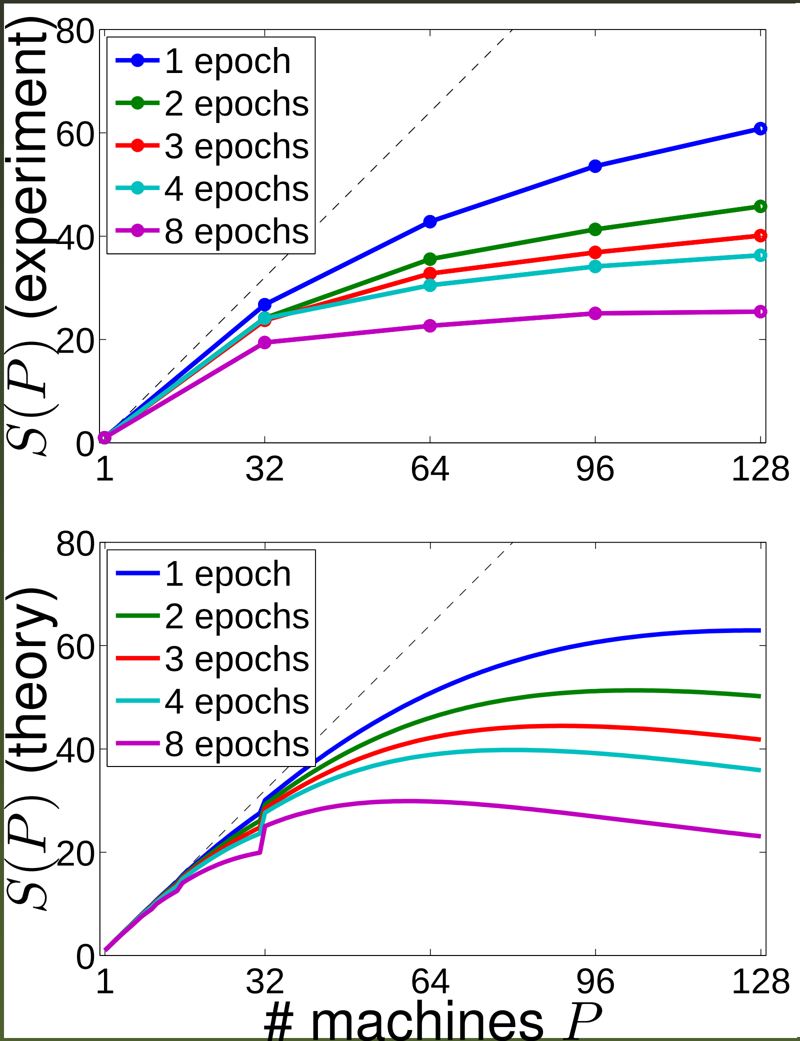
$P = 1, e \in \{1, 2, 8\}$

$e = 8, P \in \{1, 32, 64\}$

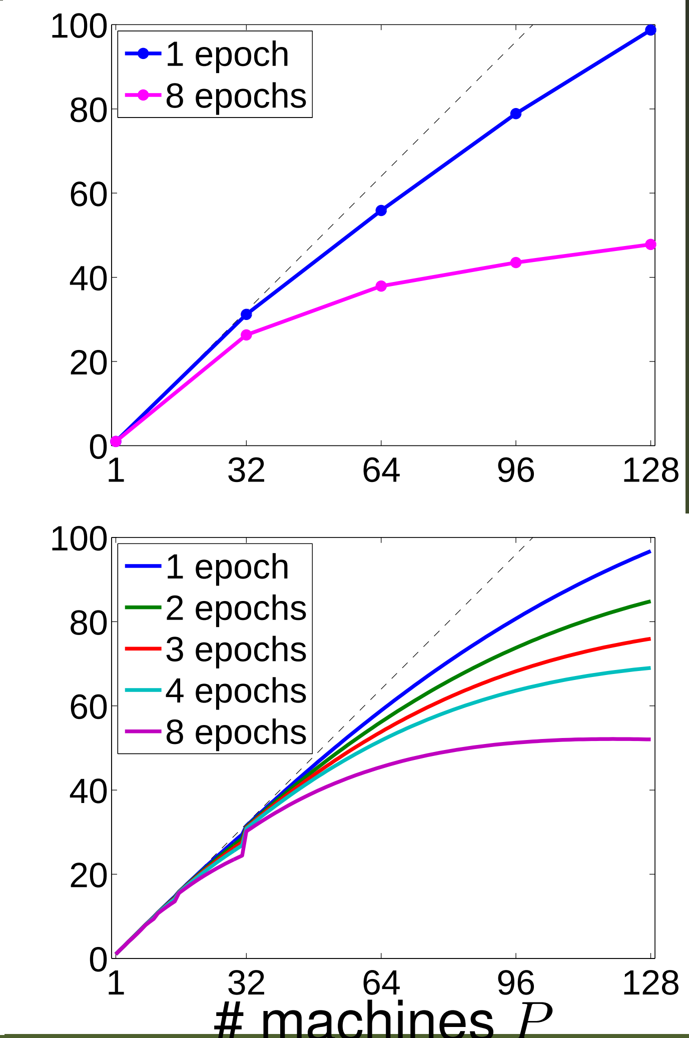
- More epochs → more exact $\mathbf{W}$ step.
  Little degradation with $e = 1$ epoch even though the dataset is small.
- Shuffling generally reduces the error and increases the precision with no increase in runtime.

Experiments: speedup $S(P)$

CIFAR $N = 50\text{K}$, $M = 32$, $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} = 200$, $t_c^{\mathbf{W}} = 10^4$

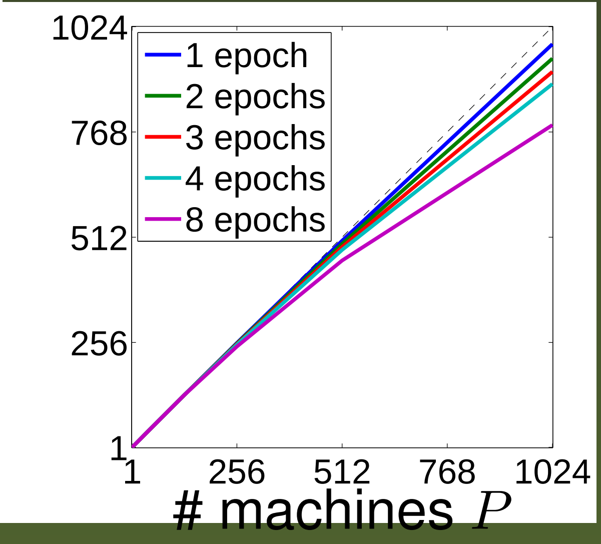


SIFT-1M $N = 10\text{M}$, $M = 32$, $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} = 40$, $t_c^{\mathbf{W}} = 10^4$



SIFT-1B $N{=}100\text{M}, M{=}128$ $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} = 40$, $t_c^{\mathbf{W}} = 10^4$

too long to run



SIFT-1M on $P = 128$ machines with $e = 8$: runtime 12', speedup $100\times$.

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.

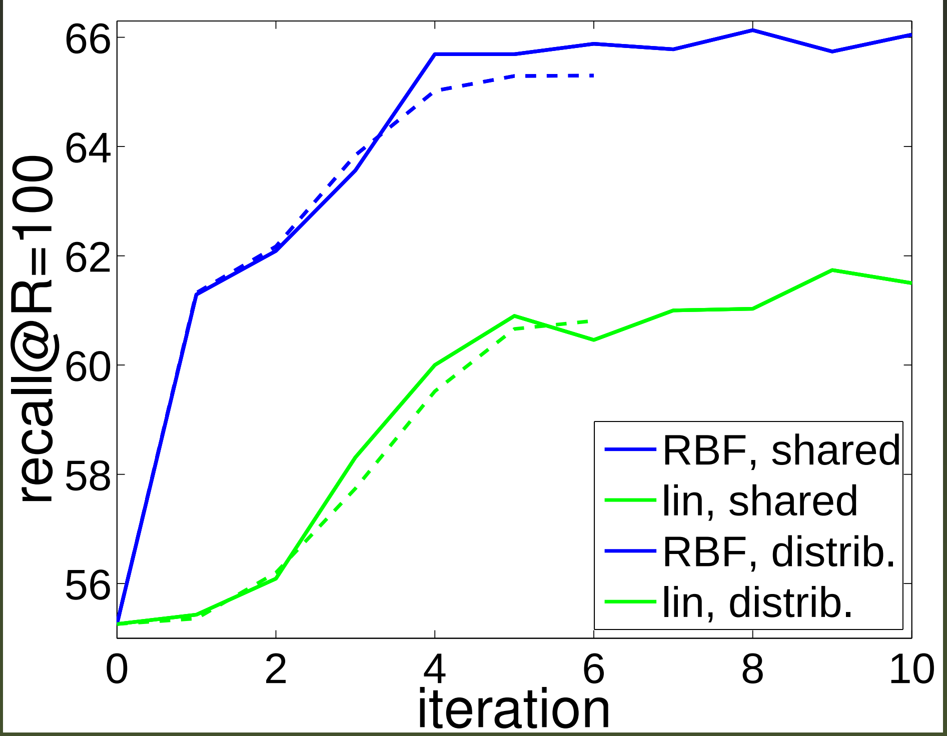
  Note small N and small M for CIFAR, so worse speedup than for SIFT-1M.
- SIFT-1B theoretical speedup nearly perfect in whole range of P.

Experiments: large-scale dataset, SIFT-1B

Distributed system: $P = 128$ processors.

Shared-memory system: $P = 64$ proc.

- Single-machine runtime: months.
- $N = 100$M training points, $D = 128$ (linear SVMs) or $2\,000$ (kernel SVMs), $L = 64$ bits (# hash functions).
- $e = 2$ epochs with shuffling.
- # MAC iterations: shared-memory 10, distributed 6.
- Learning curves essentially identical over distributed & shared-memory.
  The total runtime in the shared-memory system is shorter because it has faster processors and faster interprocessor communication.
- Speedup predicted to be nearly perfect for $P = 128$.



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

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
     - in a parameter-server topology, effectively doing parallel SGD
- Convergence to a stationary point of the objective function.
- Practical parallel speedup:
  - nearly perfect when # submodels $\gtrsim$ # machines
  - eventually saturates as we continue to increase the # machines.
- Data shuffling, load balancing, streaming & fault tolerance.

References and code

http://faculty.ucmerced.edu/mcarreira-perpinan/MAC

Original reference for MAC:

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

Tutorials, reviews:

- C-P: *A tutorial on nested system optimisation using the method of auxiliary coordinates*.
- C-P: *A gallery of proximal operators arising in the method of auxiliary coordinates*.

Extensions or related work:

- C-P & Alizadeh: *ParMAC: distributed optimisation of nested functions, with application to learning binary autoencoders*. arXiv:1605.09114.
- C-P & Raziperchikolaei: *Optimizing affinity-based binary hashing using auxiliary coordinates*. arXiv:1501.05352.
- C-P & Vladymyrov: *A fast, universal algorithm to learn parametric nonlinear embeddings*. NIPS 2015.
- C-P & Raziperchikolaei: *Hashing with binary autoencoders*. CVPR 2015, arXiv:1501.00756.
- Wang and C-P: *The role of dimensionality reduction in classification*. AAAI 2014, arXiv:1405.6444.
- Wang and C-P: *Nonlinear low-dimensional regression using auxiliary coordinates*. AISTATS 2012.
- C-P and Lu: *Parametric dimensionality reduction by unsupervised regression*. CVPR 2010.
- C-P and Lu: *Dimensionality reduction by unsupervised regression*. CVPR 2008.

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