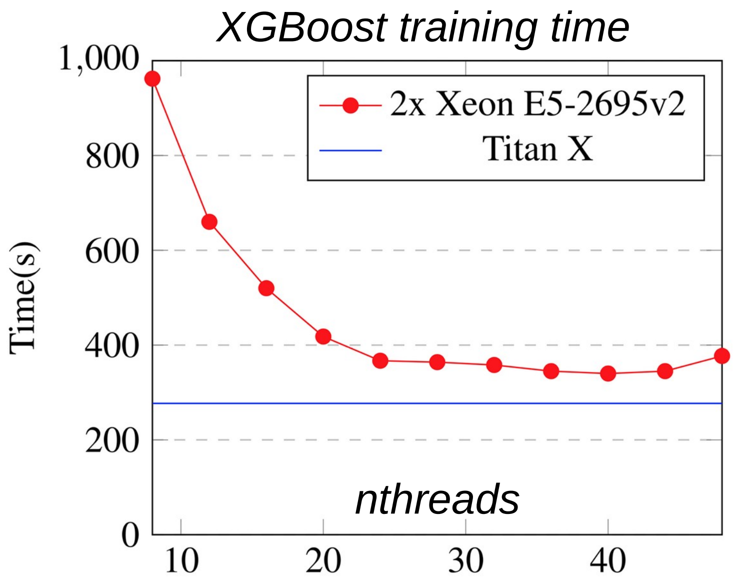
Distributed Machine Learning

Efficient DL’23, Episode II

Yandex
Research

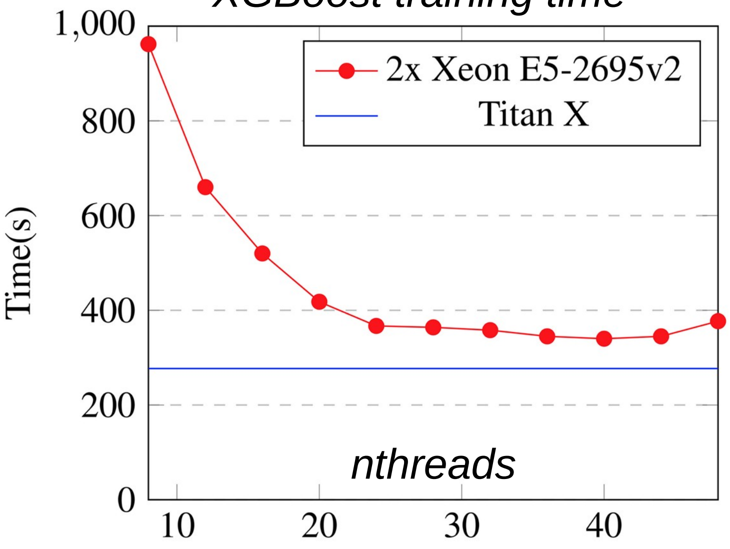
British Hedgehog
Preservation Society

Зачем это всё?

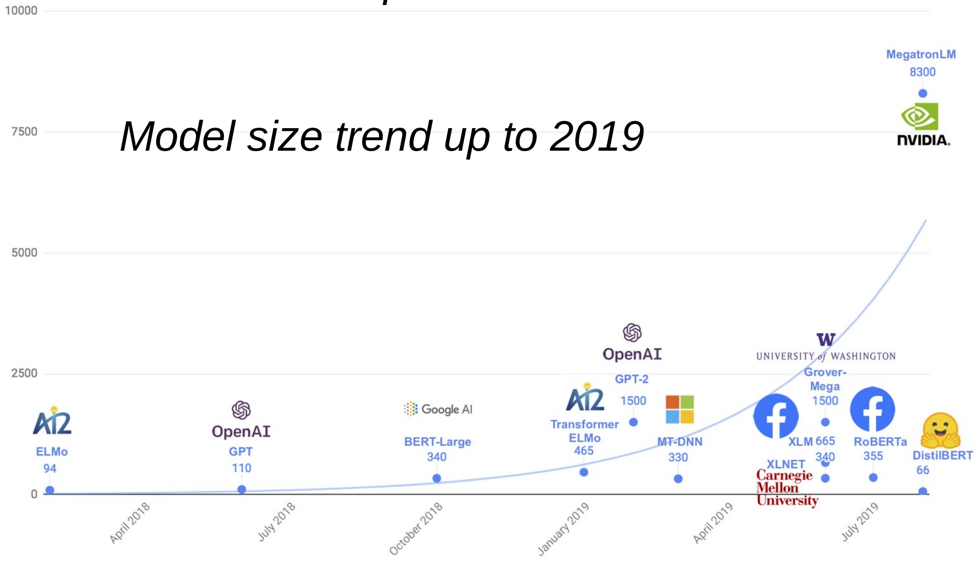


Зачем это всё?

XGBoost training time



10^6 x parameters vs time



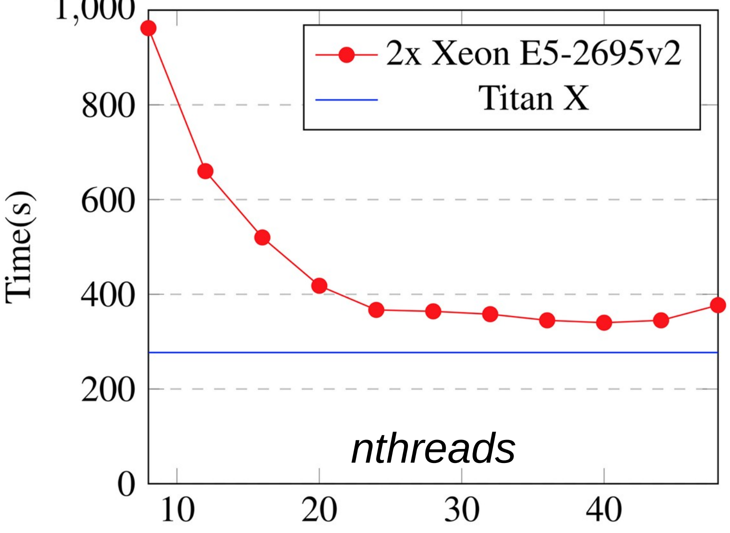
BERT-Large Training Times on GPUs

Time	System	Number of Nodes	Number of V100 GPUs
47 min	DGX SuperPOD	92 x DGX-2H	1,472
67 min	DGX SuperPOD	64 x DGX-2H	1,024
236 min	DGX SuperPOD	16 x DGX-2H	256

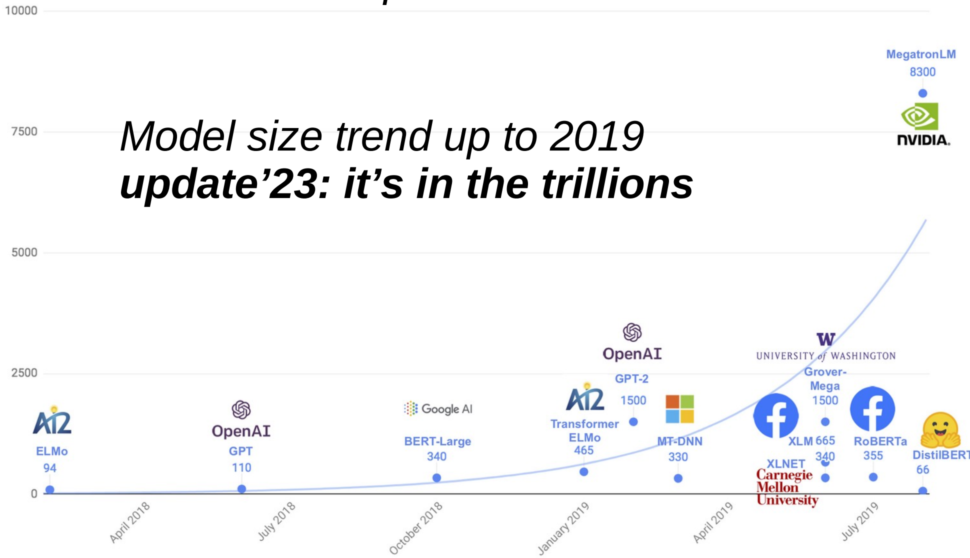
*(single V100 – **over 2 weeks**)*

Зачем это всё?

XGBoost training time



10^6 x parameters vs time



BERT-Large Training Times on GPUs

Time	System	Number of Nodes	Number of V100 GPUs
47 min	DGX SuperPOD	92 x DGX-2H	1,472
67 min	DGX SuperPOD	64 x DGX-2H	1,024
236 min	DGX SuperPOD	16 x DGX-2H	256

*(single V100 – **over 2 weeks**)*

Зачем мы тут?

Заставить много железяк вместе учить одну модель

Зачем мы тут?

Заставить много железяк вместе учить одну модель

понять общие подходы

закодить своими руками

на python / pytorch

TL;DR our plan

lectures 4,5,6

4) Distributed machine learning

Embeddings or log.regression with tons of training data

TL;DR our plan

lectures 4,5,6

4) Distributed machine learning

Embeddings or log.regression with tons of training data

5) Data-parallel deep learning

Train BERT-base on wikipedia in 20 minutes or less

TL;DR our plan

lectures 4,5,6

4) Distributed machine learning

Embeddings or log.regression with tons of training data

5) Data-parallel deep learning

Train BERT-base on wikipedia in 20 minutes or less

6) Model-parallel deep learning

Fine-tune and deploy models with 100B parameters

TL;DR our plan

lectures 4,5,6

4) Distributed machine learning

Embeddings or log.regression with tons of training data

5) Data-parallel deep learning

Train BERT-base on wikipedia in 20 minutes or less

6) Model-parallel deep learning

Fine-tune and deploy models with 100B parameters

like OPT-175B, BLOOM-176B, YALM, GLM, Galactica

TL;DR our plan

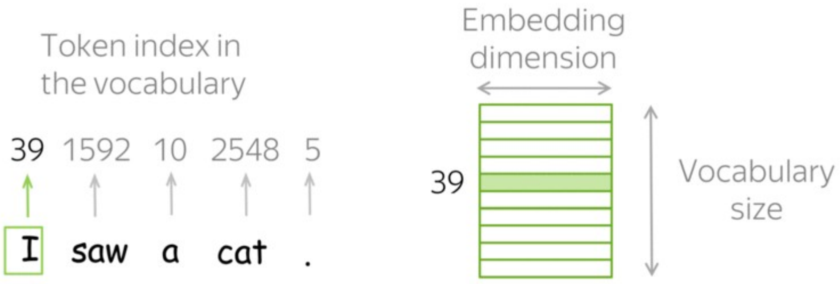
lectures 4,5,6

4) Distributed machine learning

Embeddings or log.regression with tons of training data

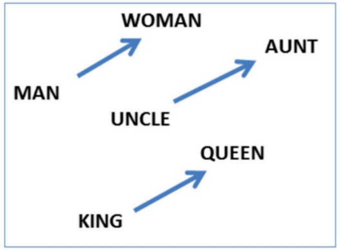
Today: learn the basics behind it all

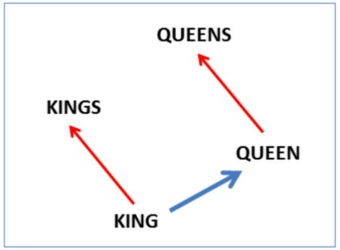
Example problem: word embeddings



semantic: v(**king**) – v(**man**) + v(**woman**) ≈ v(**queen**)

syntactic: v(**kings**) – v(**king**) + v(**queen**) ≈ v(**queens**)





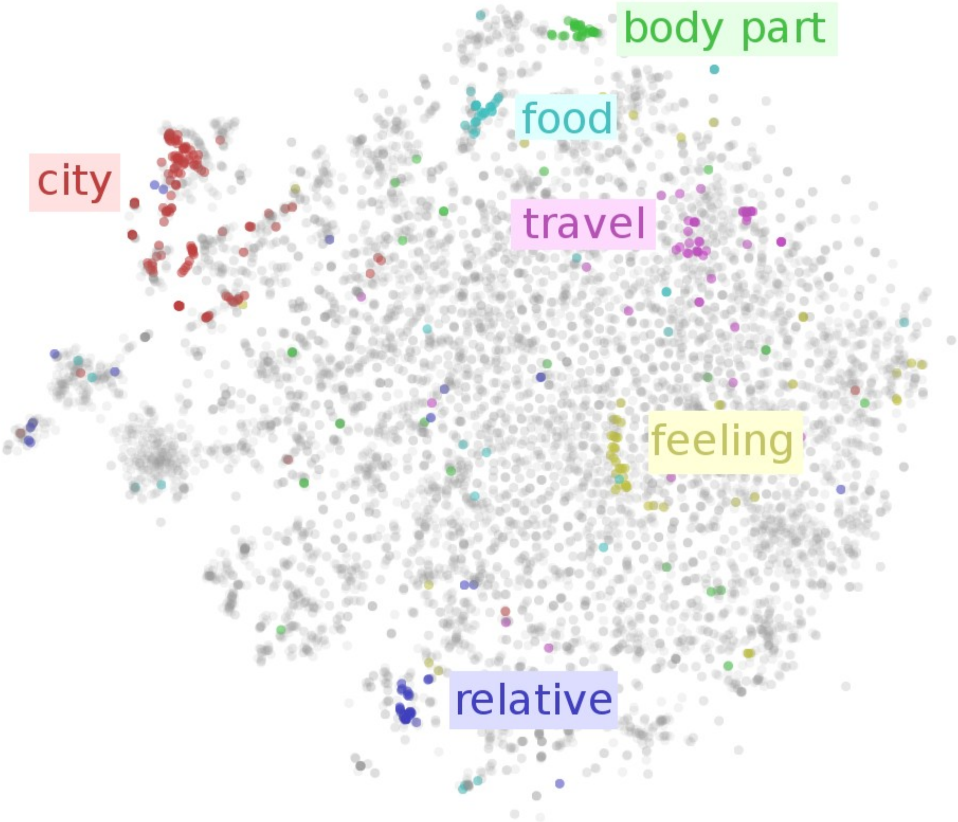
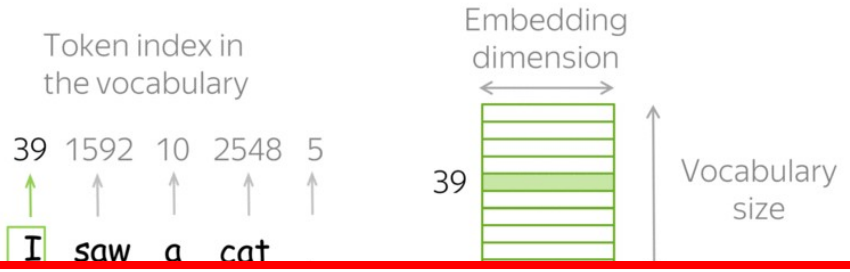
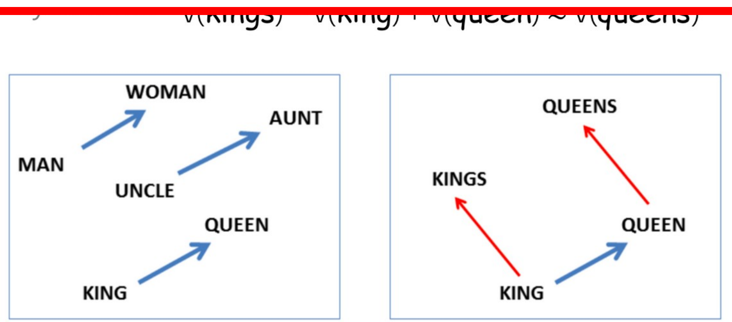


Image source: Lena’s blog, Ruder’s blog

Example problem: word embeddings



This is an example problem, don't focus on NLP too much
computationally similar: large-scale LogReg, SVD, GBDT



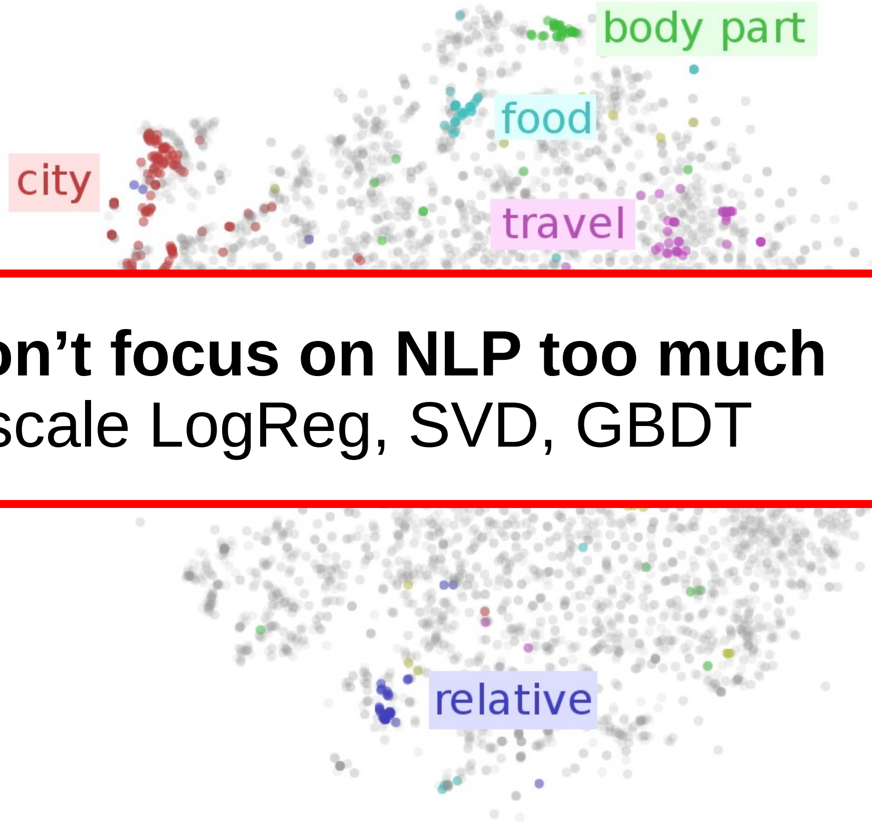
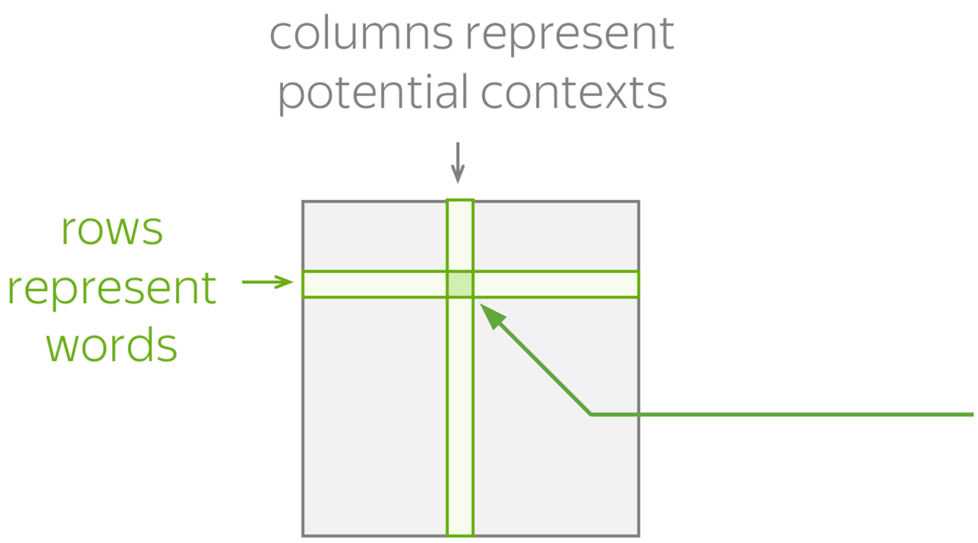


Image source: Lena's blog, Ruder's blog

Co-occurence matrix

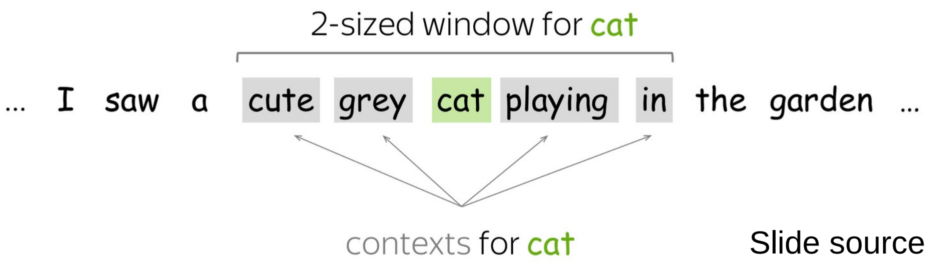


Context:

- surrounding words in a L-sized window

Matrix element:

- N(w, c) – number of times word w appears in context c



Note: *in our case, N is symmetric!*

Slide source: Lena's blog

GloVe

context vector

word vector

bias terms (also learned)

$$L=\sum_{i\neq j} w\left(N(i,j)\right)\cdot\left(\langle \vec{v}_i, \vec{v}_j \rangle + b_i + b_j - \log N(i,j)\right)^2$$

Slide source: Lena’s blog

GloVe

context vector

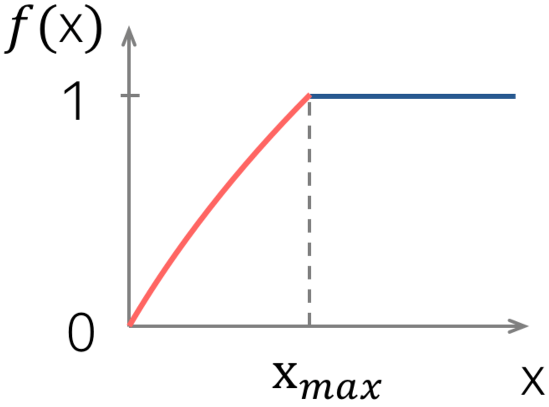
word vector

bias terms (also learned)

$$L = \sum_{i \neq j} w(N(i,j)) \cdot \left(\langle \vec{v}_i, \vec{v}_j \rangle + b_i + b_j - \log N(i,j) \right)^2$$

Weighting function to:

- penalize rare events
- not to over-weight frequent events



$$\begin{cases} (x/x_{max})^{\alpha} & \text{if } x < x_{max}, \\ 1 & \text{otherwise.} \end{cases}$$

$\alpha = 0.75, x_{max} = 100$

Slide source: Lena's blog

GloVe

context vector

word vector

bias terms (also learned)

$$L=\sum_{i\neq j} w\left(N\left(i,j\right)\right)\cdot\left(\langle \vec{v}_i, \vec{v}_j\rangle+b_i+b_j-\log N\left(i,j\right)\right)^2$$

Learn more: *lena-voita.github.io/nlp_course/word_embeddings.html*

So how do we train ‘em?

Training Step

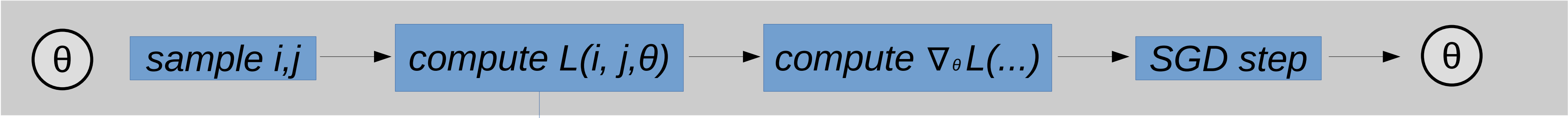
θ → *sample i,j* → *compute L(i, j,θ)* → *compute $\nabla_\theta L(\ldots)$* → *SGD step* → θ

current params
θ: {v, b}

$$L=\sum_{i\neq j} w\left(N\left(i,j\right)\right)\cdot\left(\left\langle \vec{v}_i,\vec{v}_j\right\rangle+b_i+b_j-\log N\left(i,j\right)\right)^2$$

updated params

Training Step

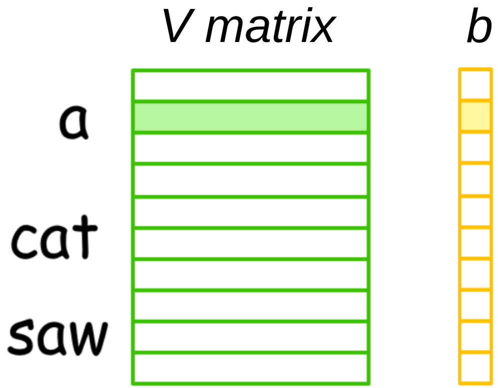


current params θ: {v, b}

updated params

$$L=\sum_{i\neq j} w\left(N\left(i,j\right)\right)\cdot\left(\langle\vec{v}_i,\vec{v}_j\rangle+b_i+b_j-\log N\left(i,j\right)\right)^2$$

Trainable parameters:



Training Step

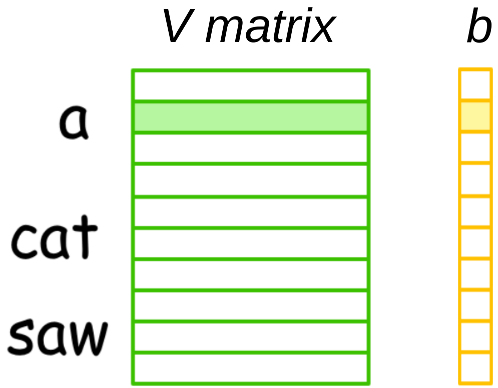
θ → sample i,j → compute L(i, j,θ) → compute $\nabla_\theta L(\ldots)$ → SGD step → θ

current params θ: {v, b}

updated params

$$L=\sum_{i\neq j} w(N(i,j))\cdot(\langle \vec{v}_i,\vec{v}_j\rangle+b_i+b_j-\log N(i,j))^2$$

Trainable parameters:



How do we go faster with 8 CPU cores?

Training Step

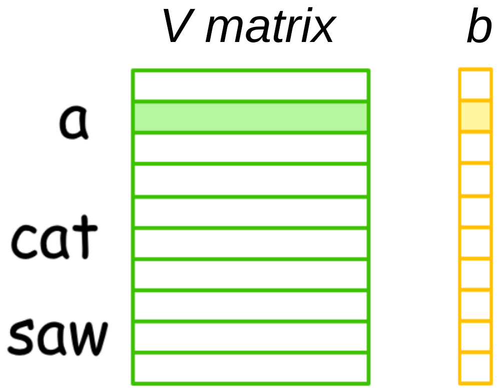
θ → sample i,j → compute L(i, j,θ) → compute $\nabla_\theta L(\ldots)$ → SGD step → θ

current params θ: {v, b}

updated params

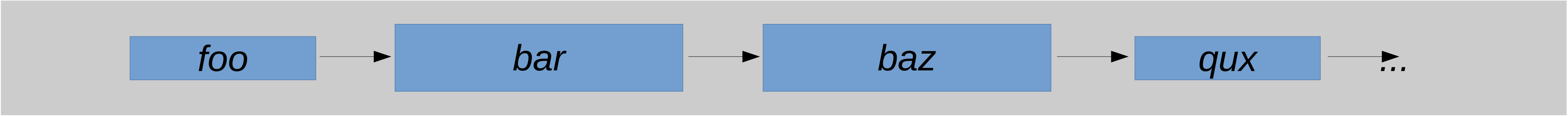
$$L=\sum_{i\neq j} w(N(i,j))\cdot\left(\langle \vec{v}_i,\vec{v}_j\rangle+b_i+b_j-\log N(i,j)\right)^2$$

Trainable parameters:



[let's formalize your ideas]

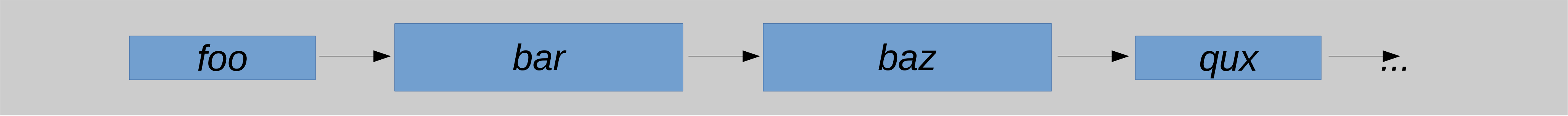
Rules: Process



Process:

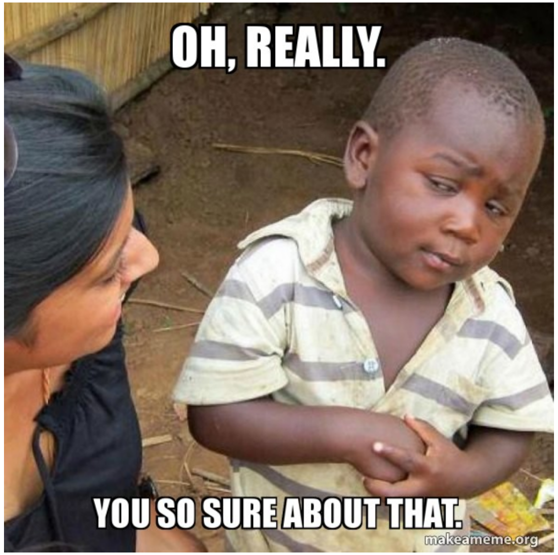
- Runs some code
- Has some memory
- No one else can access your memory

Rules: Process

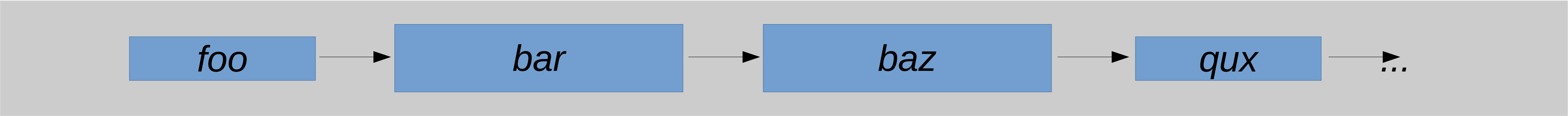


Process:

- Runs some code
- Has some memory
- No one else can access your memory



Rules: Process

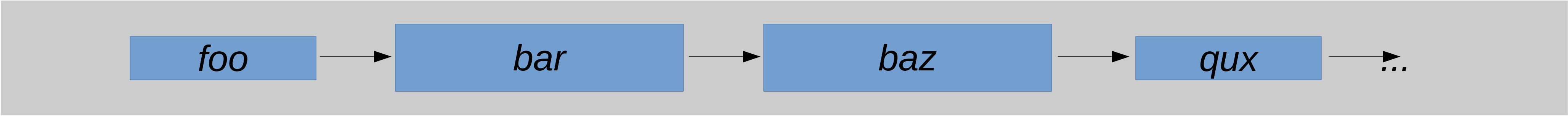


Process:

- Runs some code
- Has some memory
- No one else can access your memory*

* – not if you use shared memory

Rules: Process

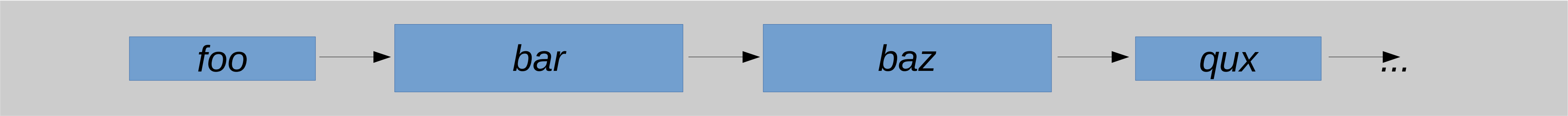


Process:

- Runs some code
- Has some memory
- No one else can access your memory*†

* – not if you use shared memory
† – superuser can still do that (os-dependent)

Rules: Process



Process:

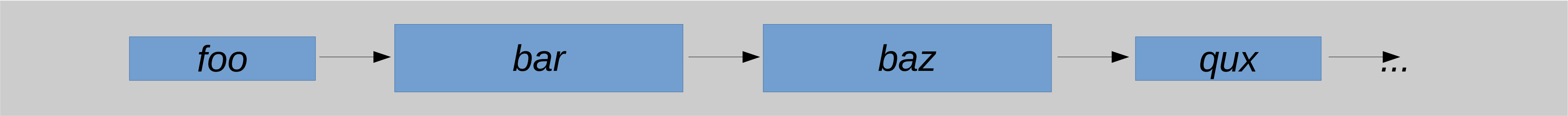
- Runs some code
- Has some memory
- No one else can access your memory*†‡

* – not if you use shared memory

† – superuser can still do that (os-dependent)

‡ – attacker can do that through spectre/meltdown/etc

Rules: Process

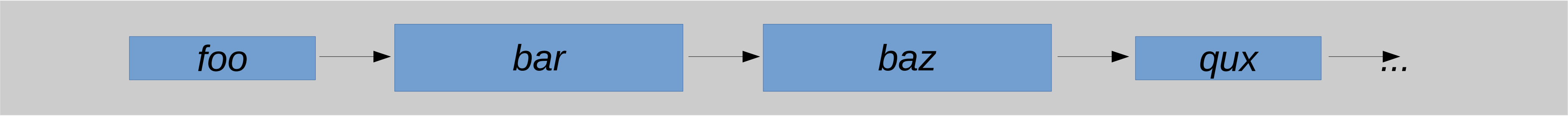


Process:

- Runs some code
- Has some memory
- No one else **should** access your memory*†‡

*†‡ – not relevant for this course

Rules: Process



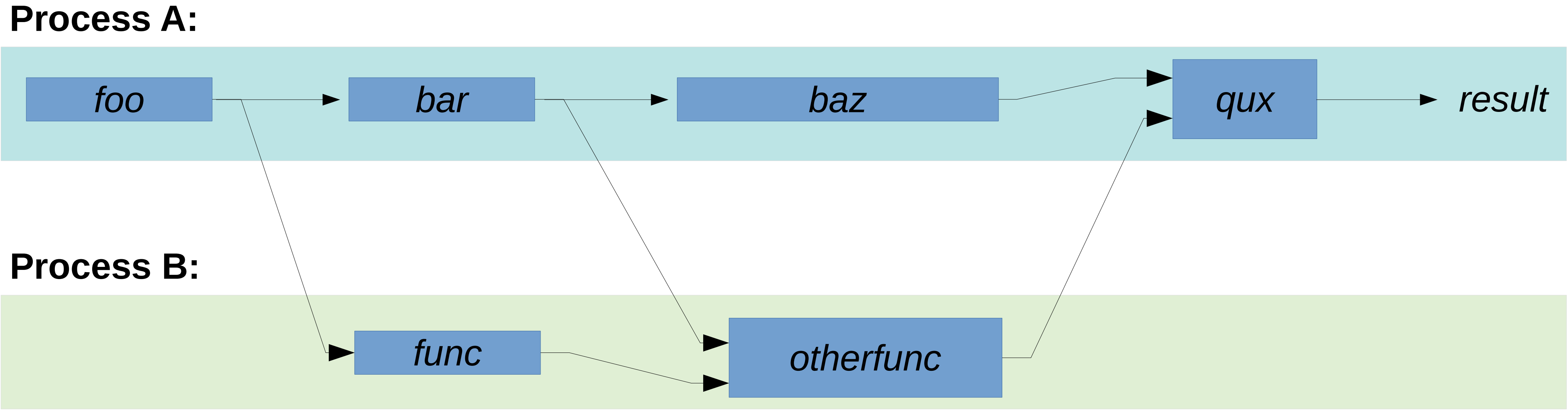
Process:

- Runs some code
- Has some memory
- No one else **should** access your memory*†‡

*†‡ – not relevant for this course

Q: How do we make processes work together?

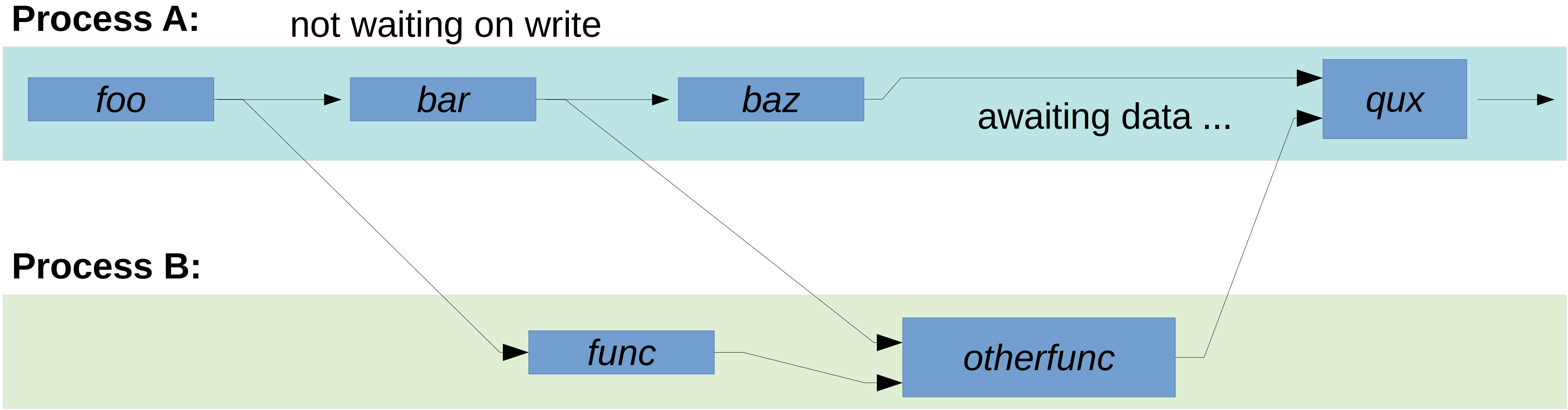
Rules: Channel / Pipe



Channel (pipe):

- Communication in O(message size)
- Asynchronous read/write

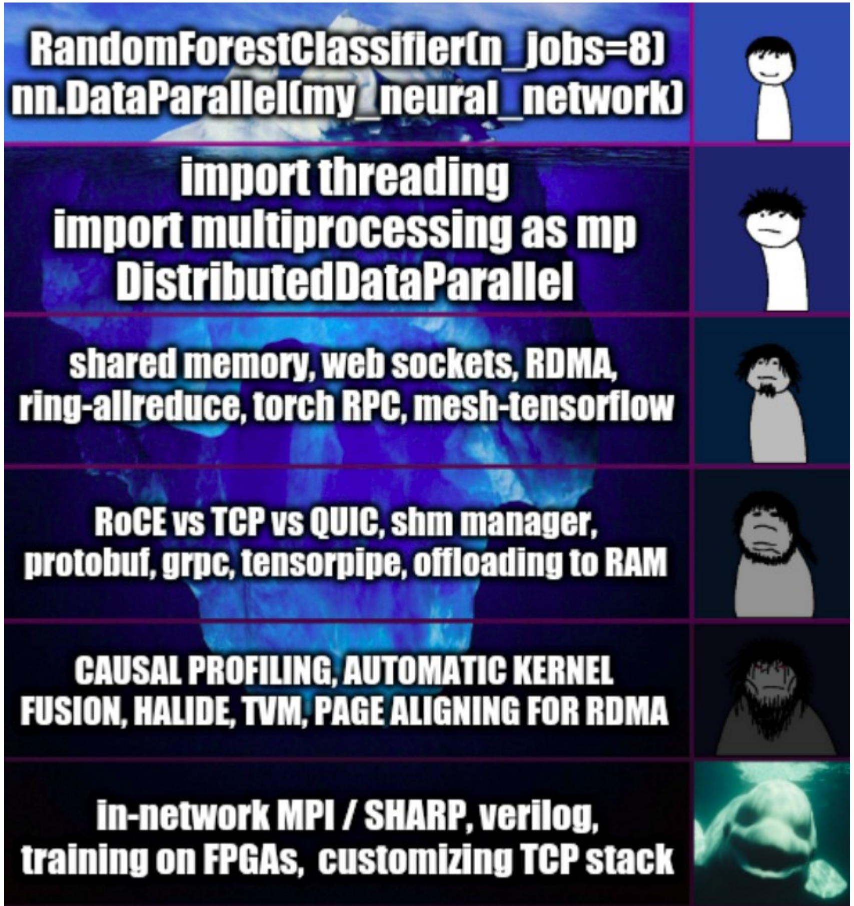
MP Rules



Channel (pipe):

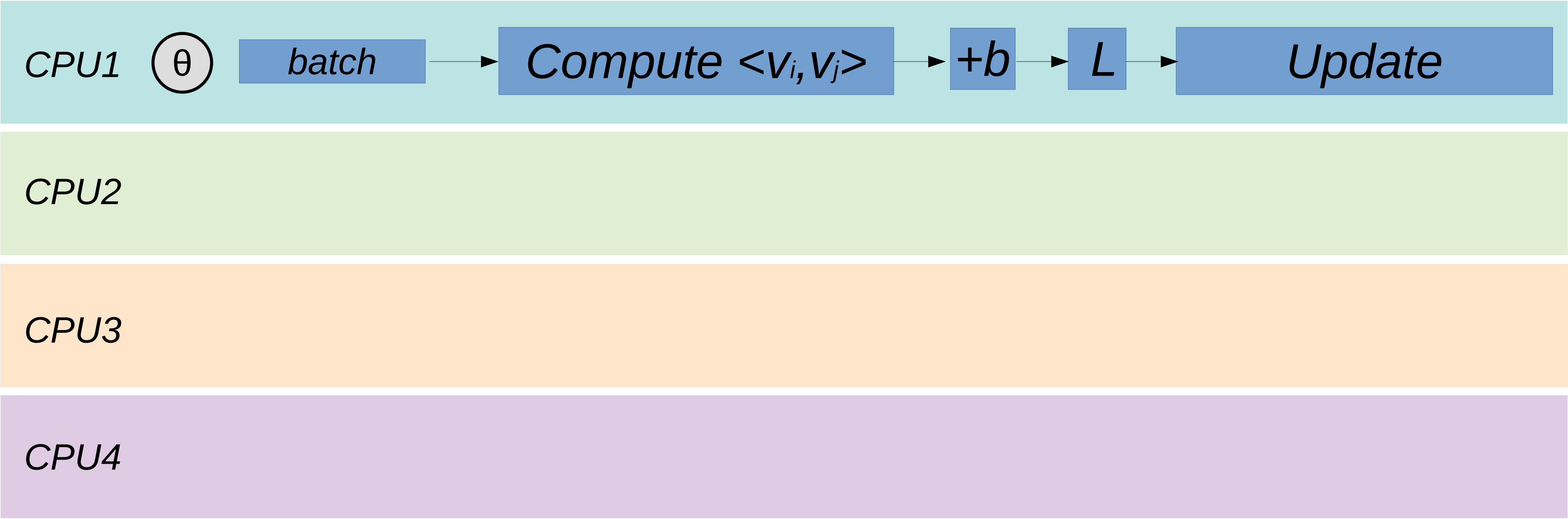
- Communication in O(message size)
- **Asynchronous** read/write

Details are (not) important



Operation parallelism

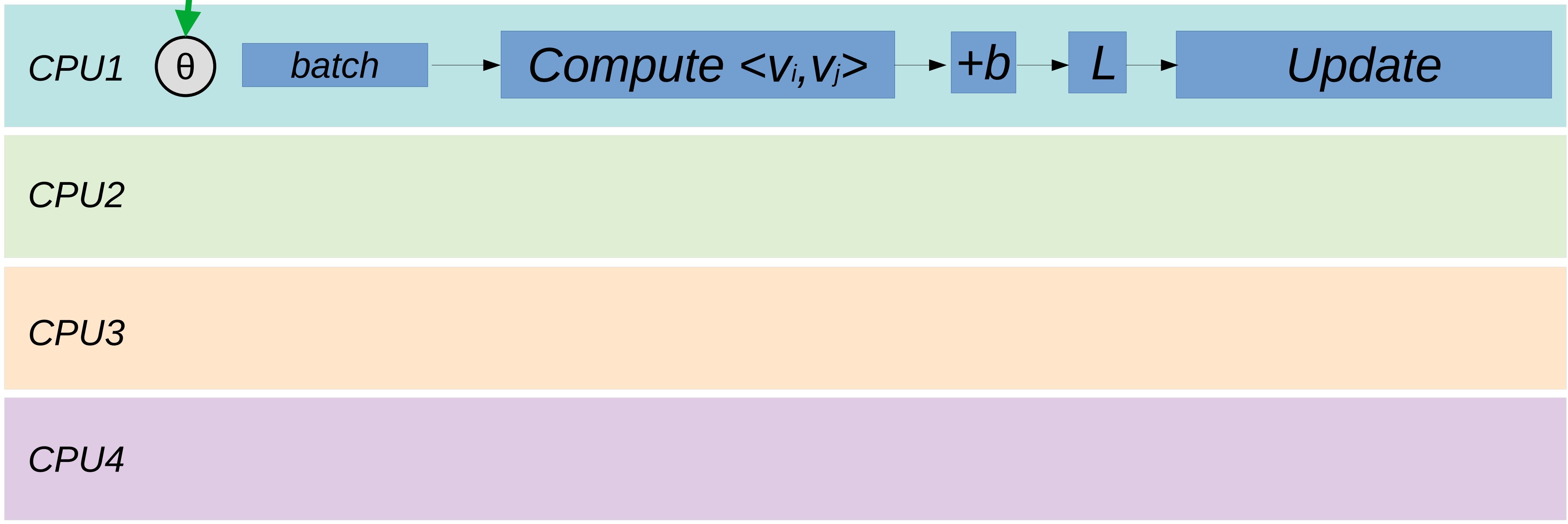
run algorithm in parallel without changing the math



Operation parallelism

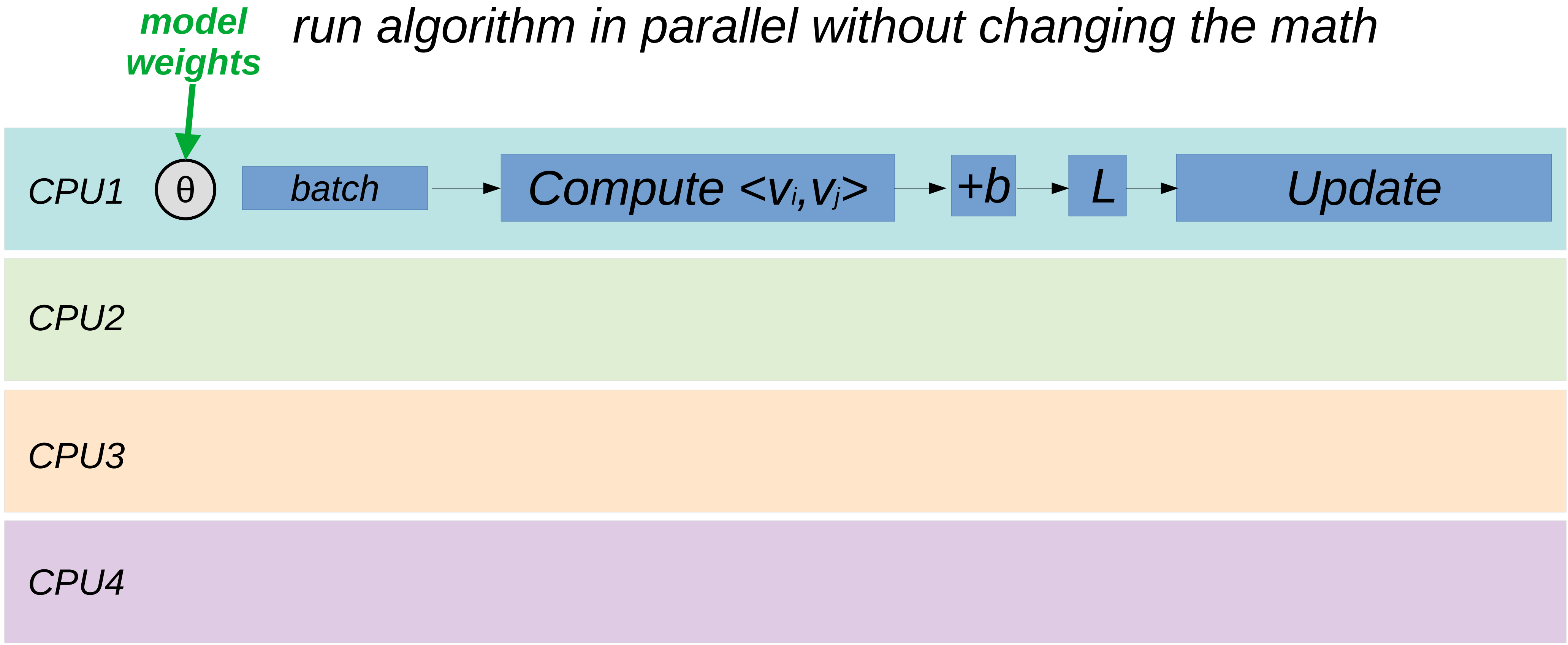
run algorithm in parallel without changing the math

model weights



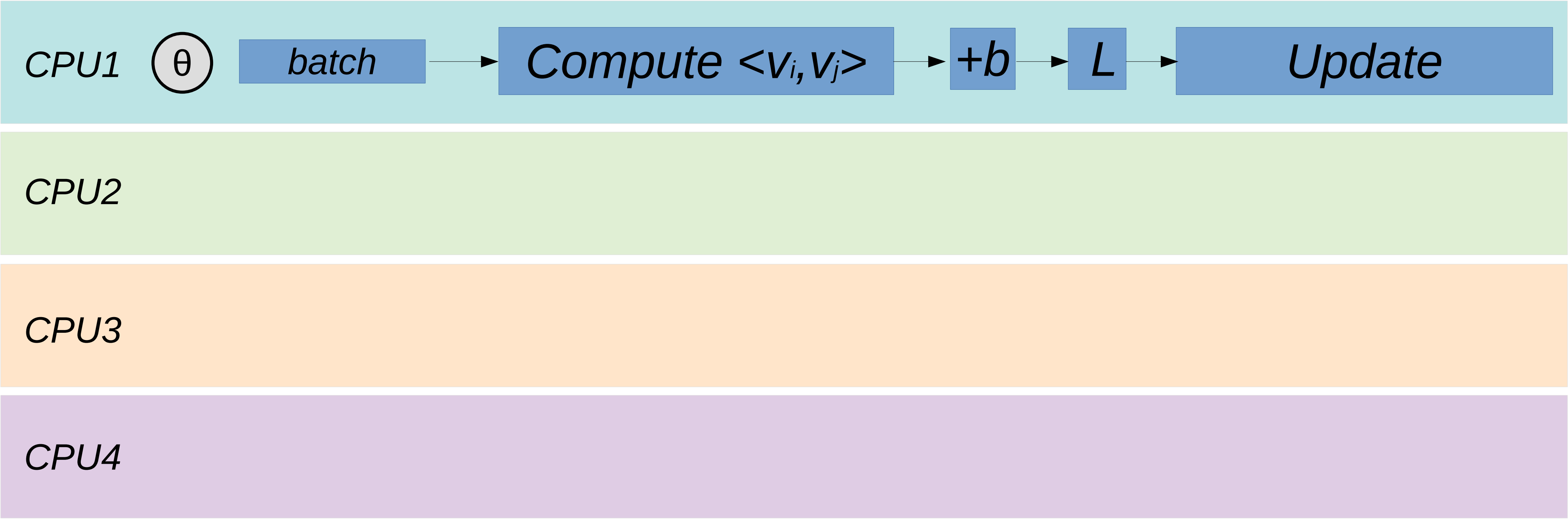
Operation parallelism

run algorithm in parallel without changing the math



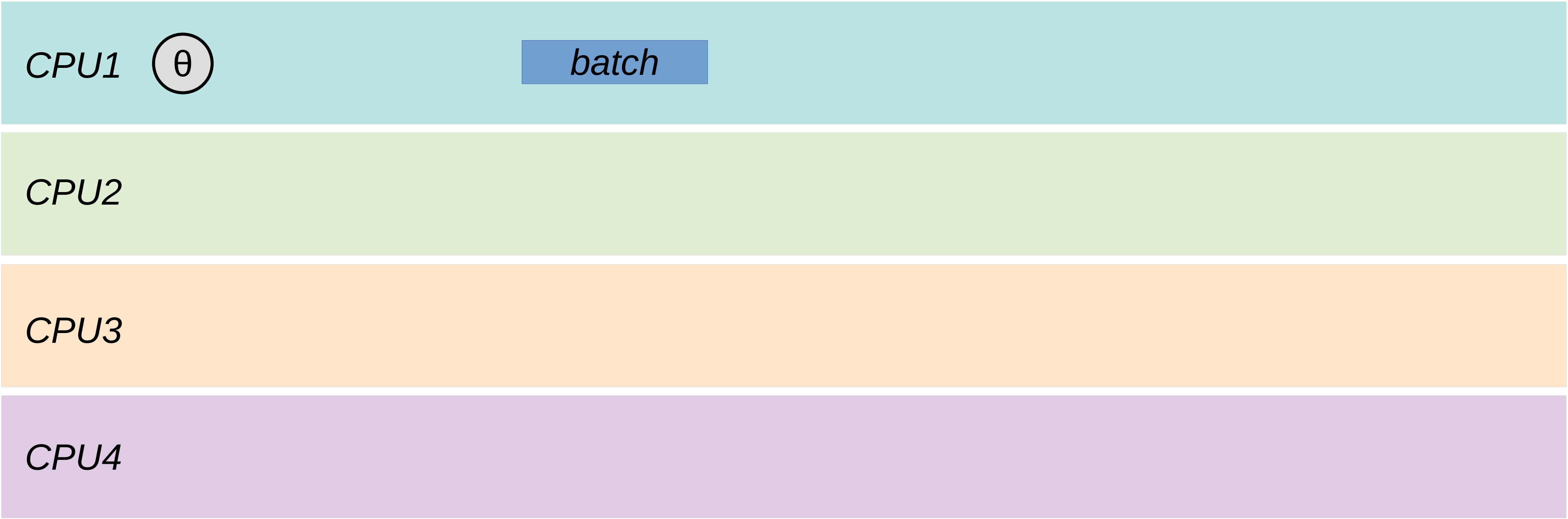
“Data parallelism”

*Each process runs **full** model on **some** samples*



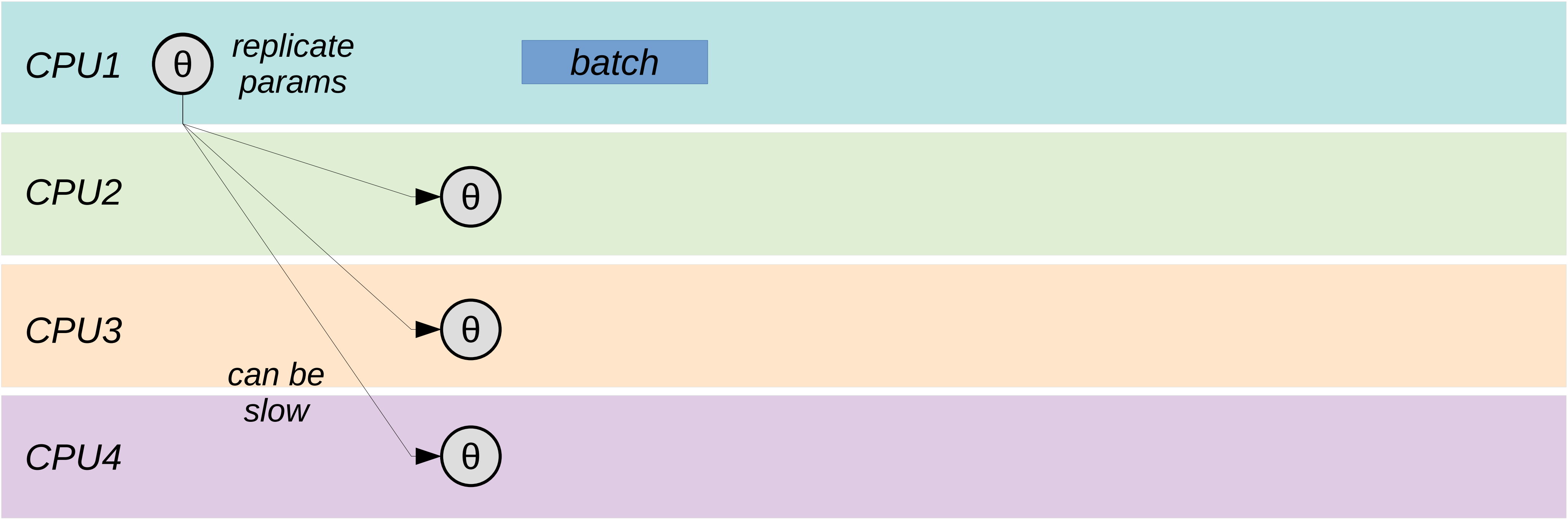
“Data parallelism”

*Each process runs **full** model on **some** samples*



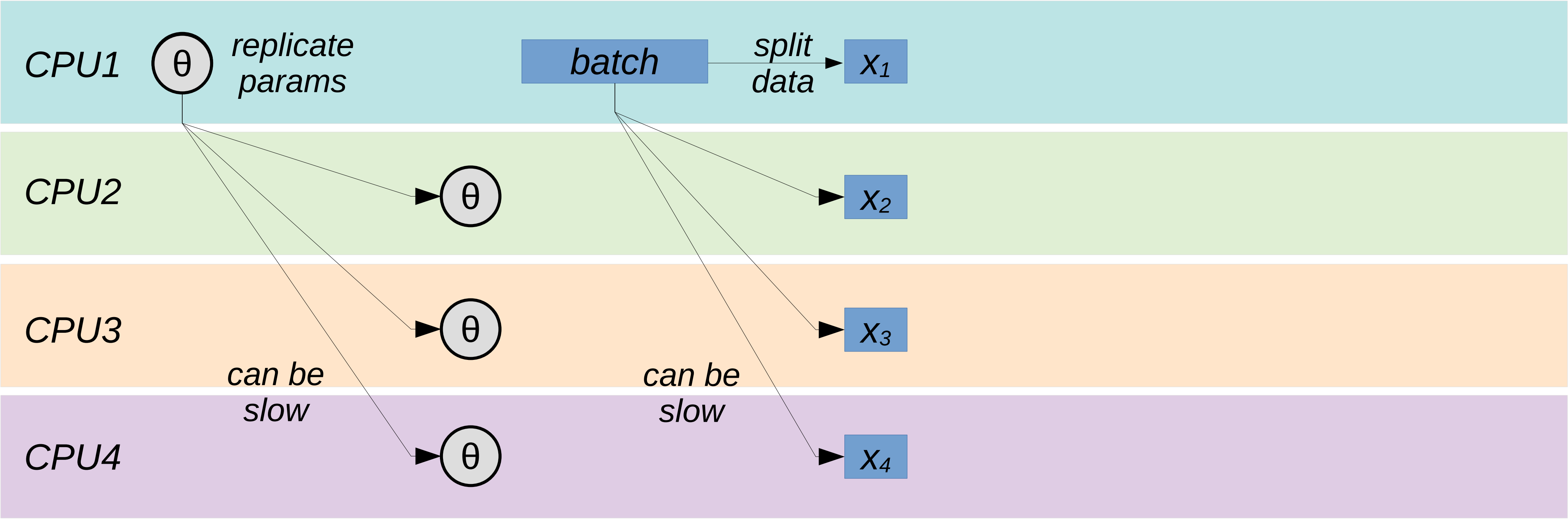
“Data parallelism”

*Each process runs **full** model on **some** samples*



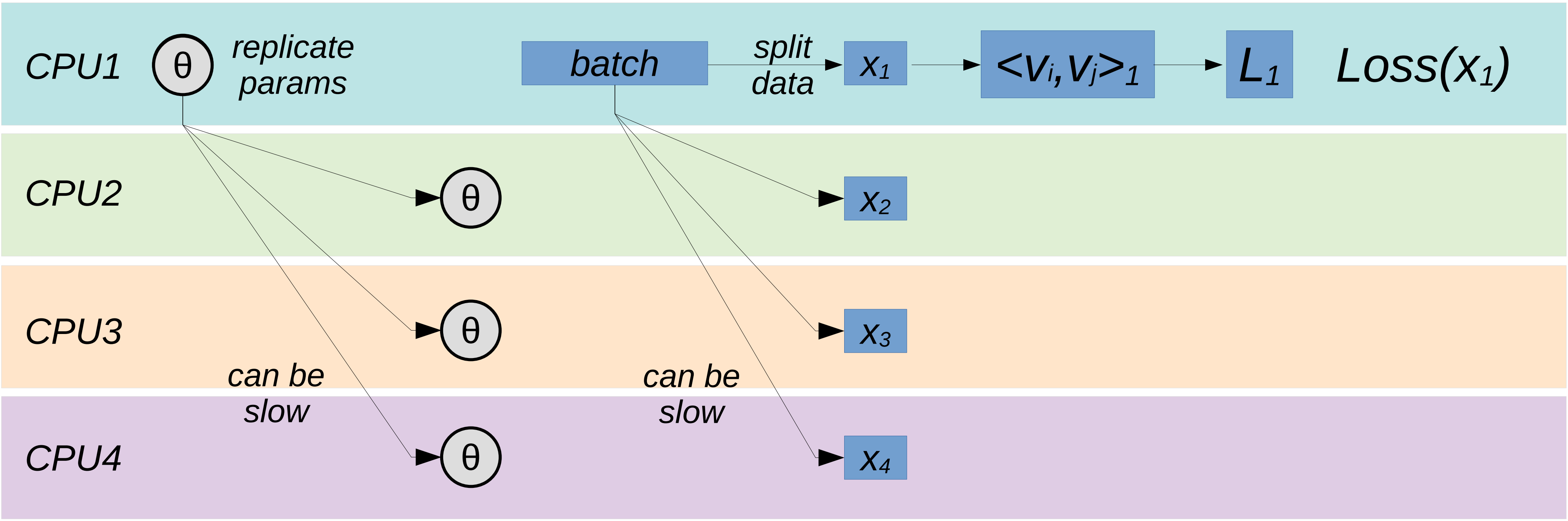
“Data parallelism”

*Each process runs **full** model on **some** samples*



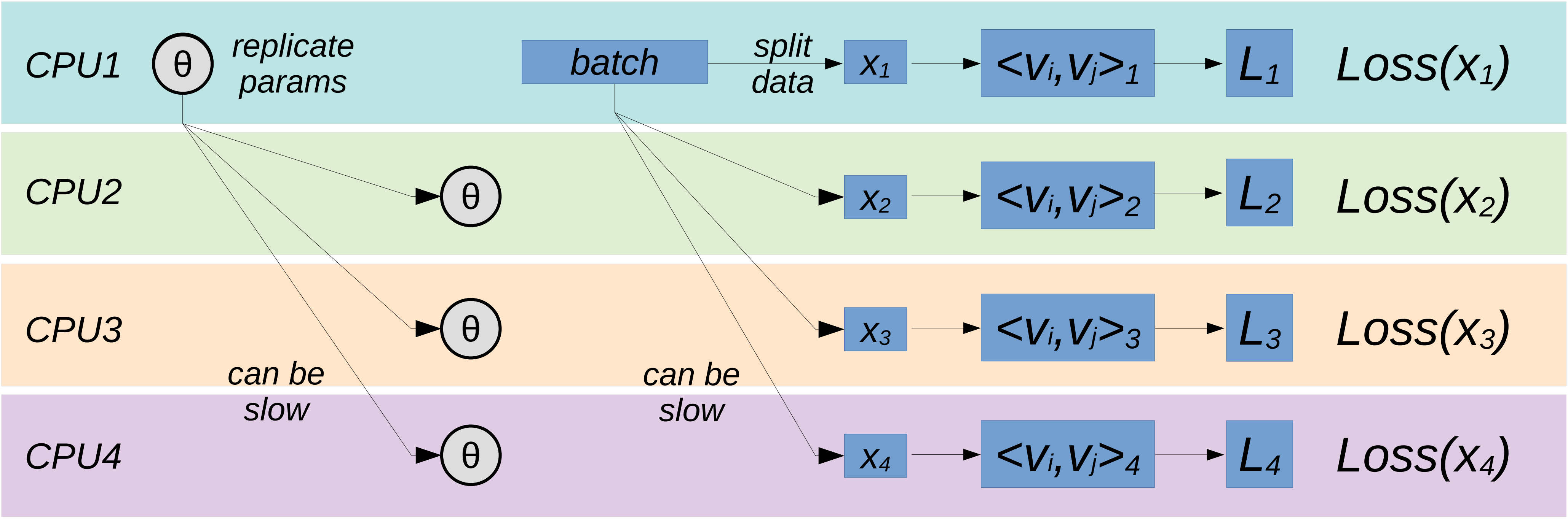
“Data parallelism”

*Each process runs **full** model on **some** samples*



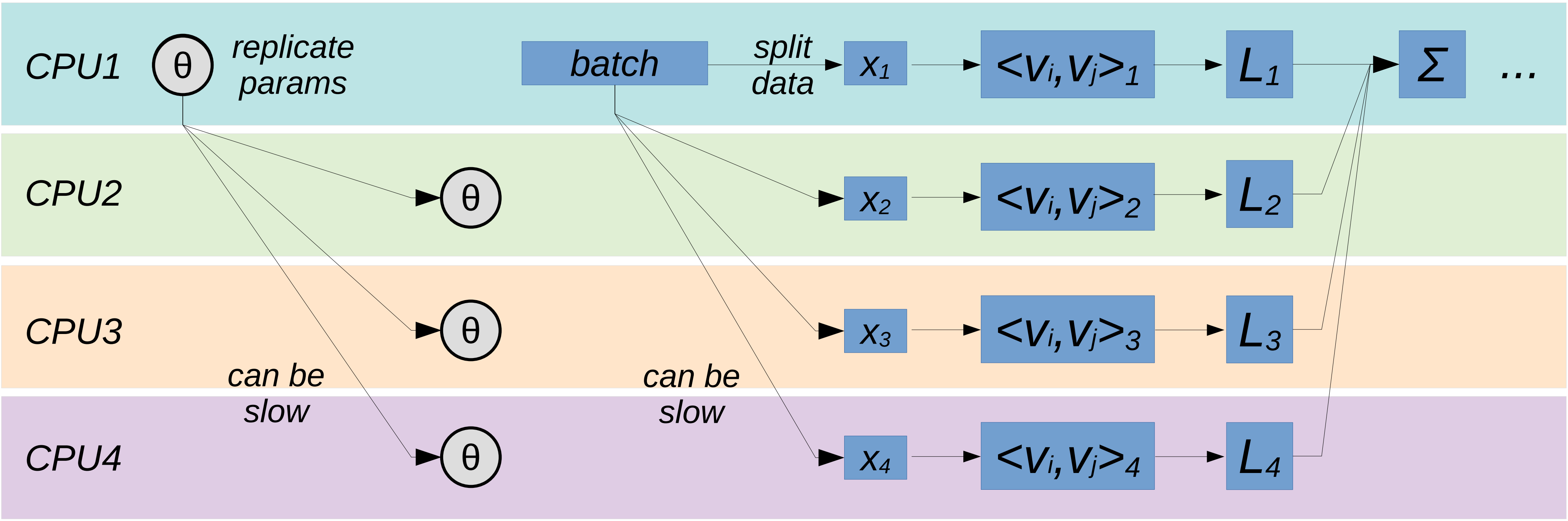
“Data parallelism”

*Each process runs **full** model on **some** samples*



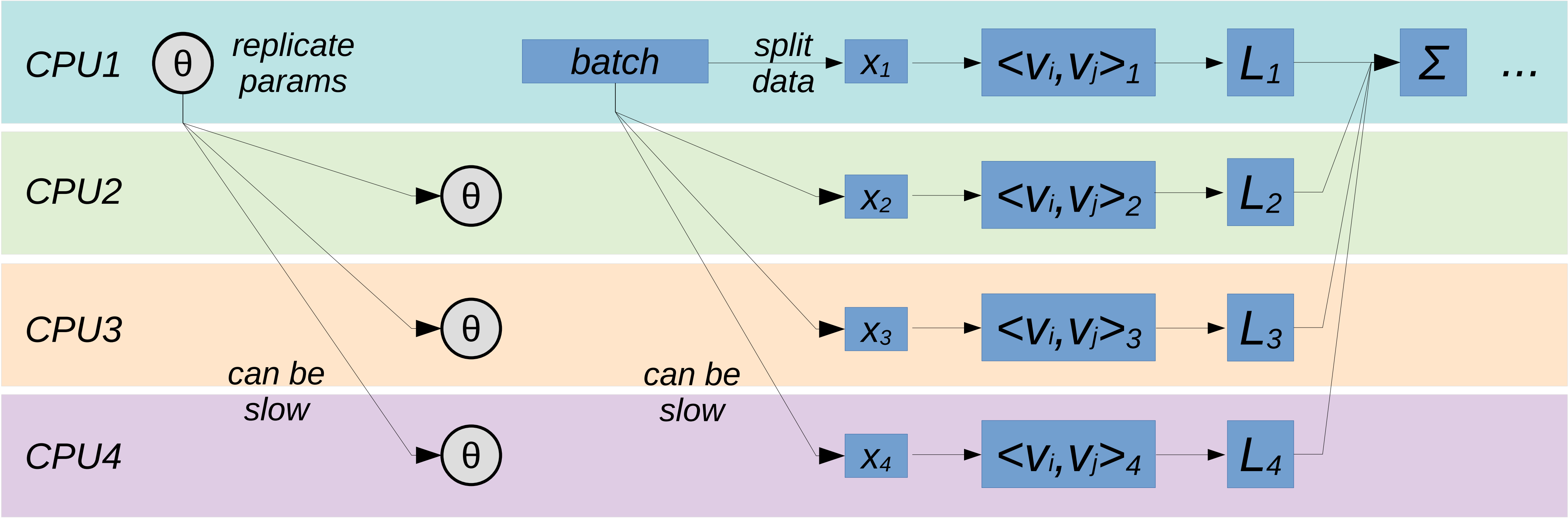
“Data parallelism”

*Each process runs **full** model on **some** samples*



“Data parallelism”

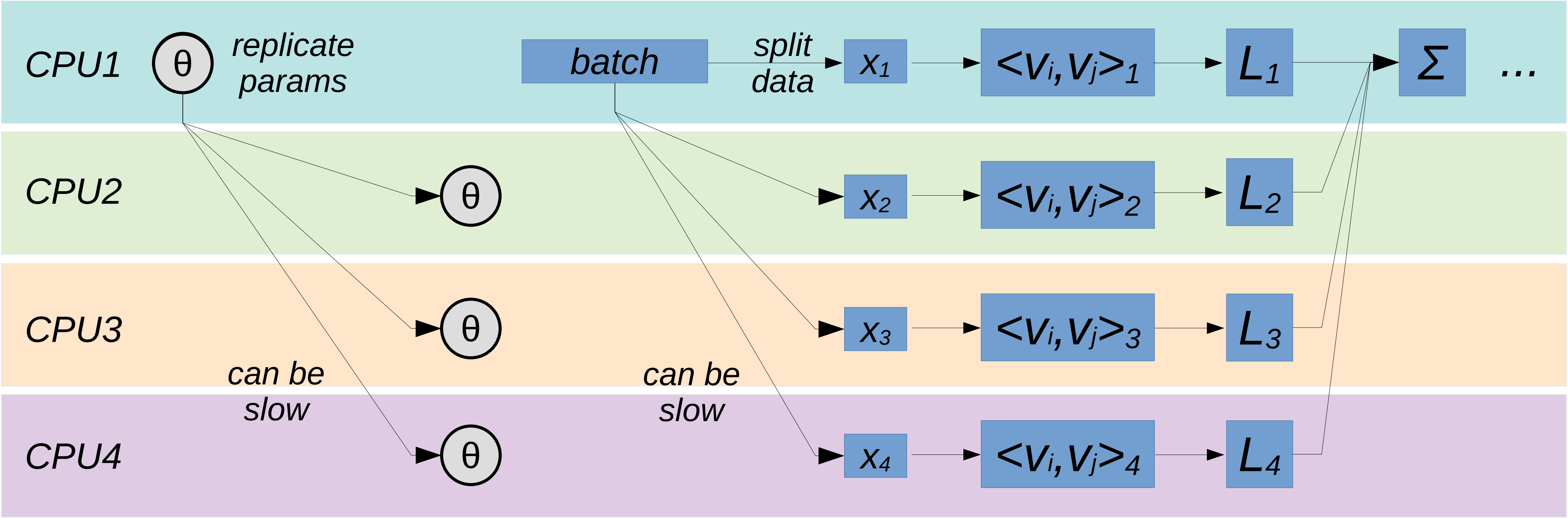
Q: Is it guaranteed to be faster?



“Data parallelism”

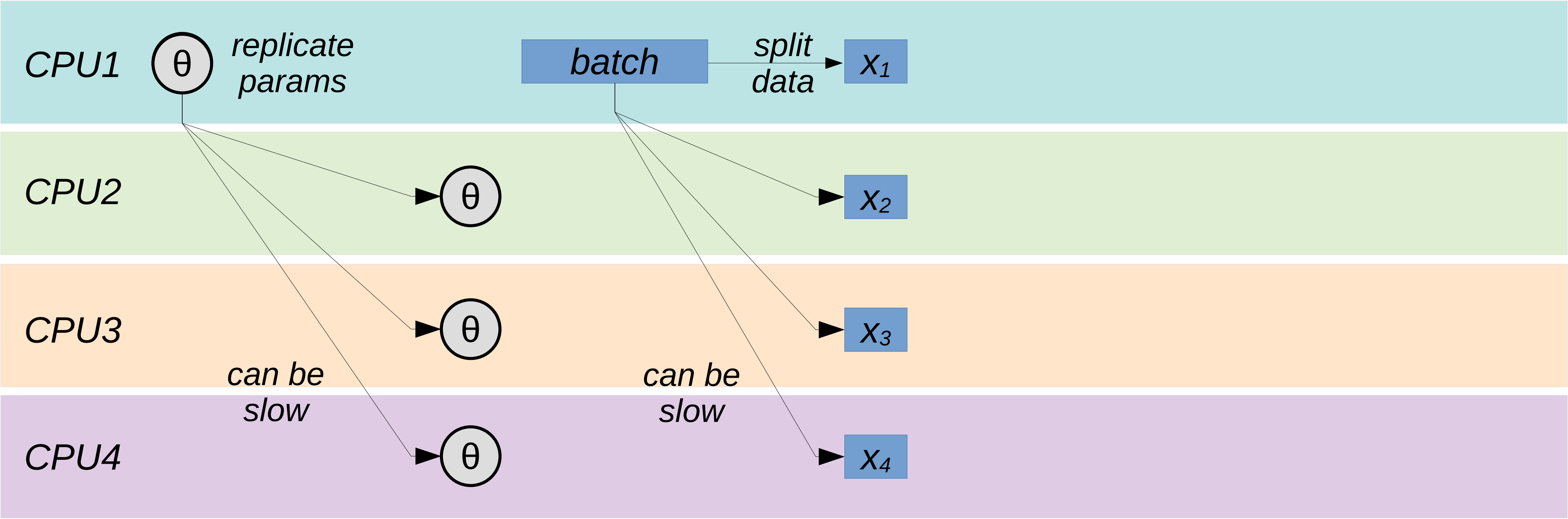
Q: Is it guaranteed to be faster?

No, sending data may take longer than computing



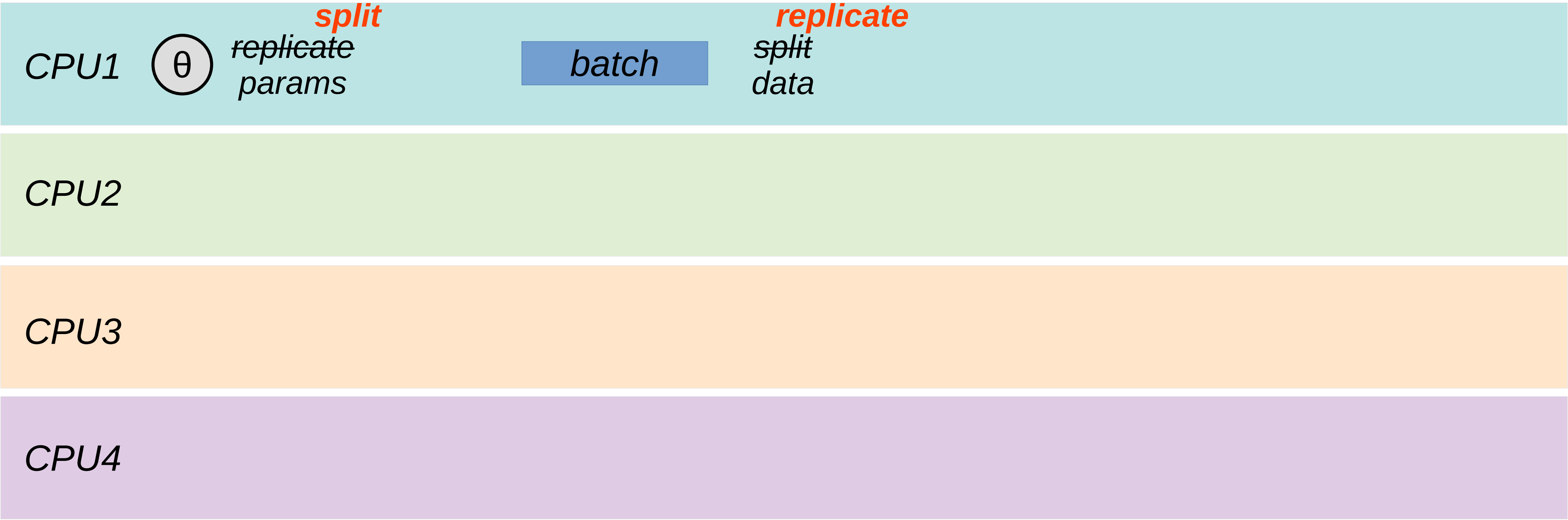
“Data parallelism”

Q: any other way to compute <vi, vj> in parallel?



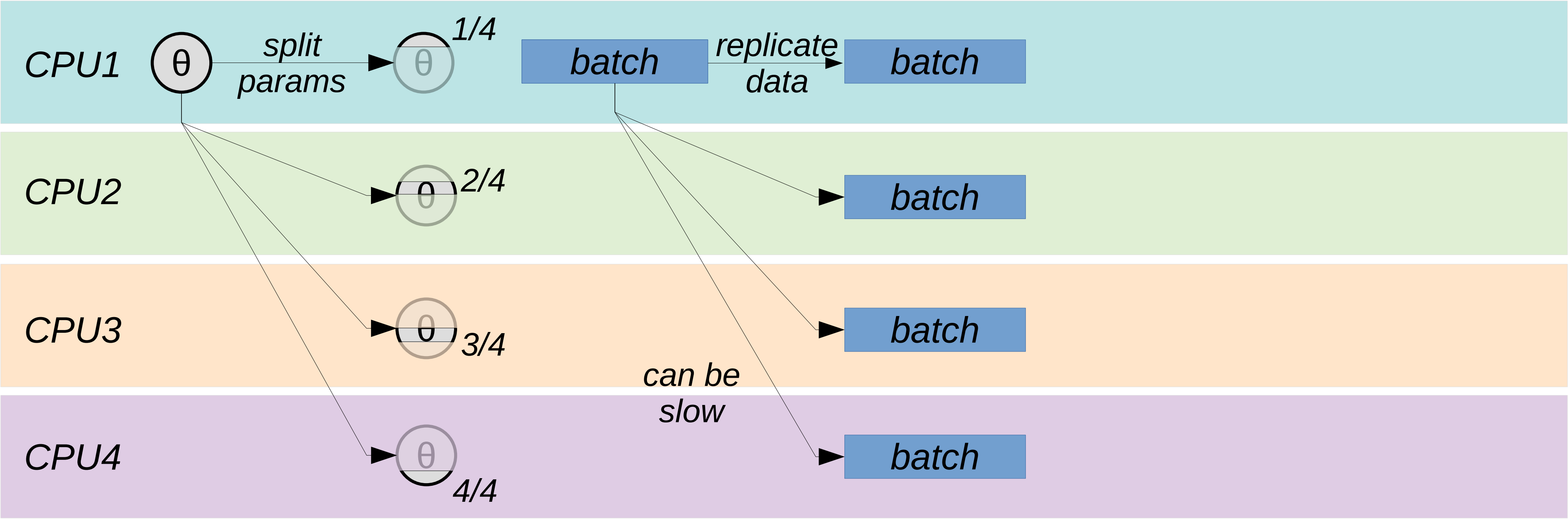
“Model parallelism”

*Each process runs **partial** model on **all** samples*



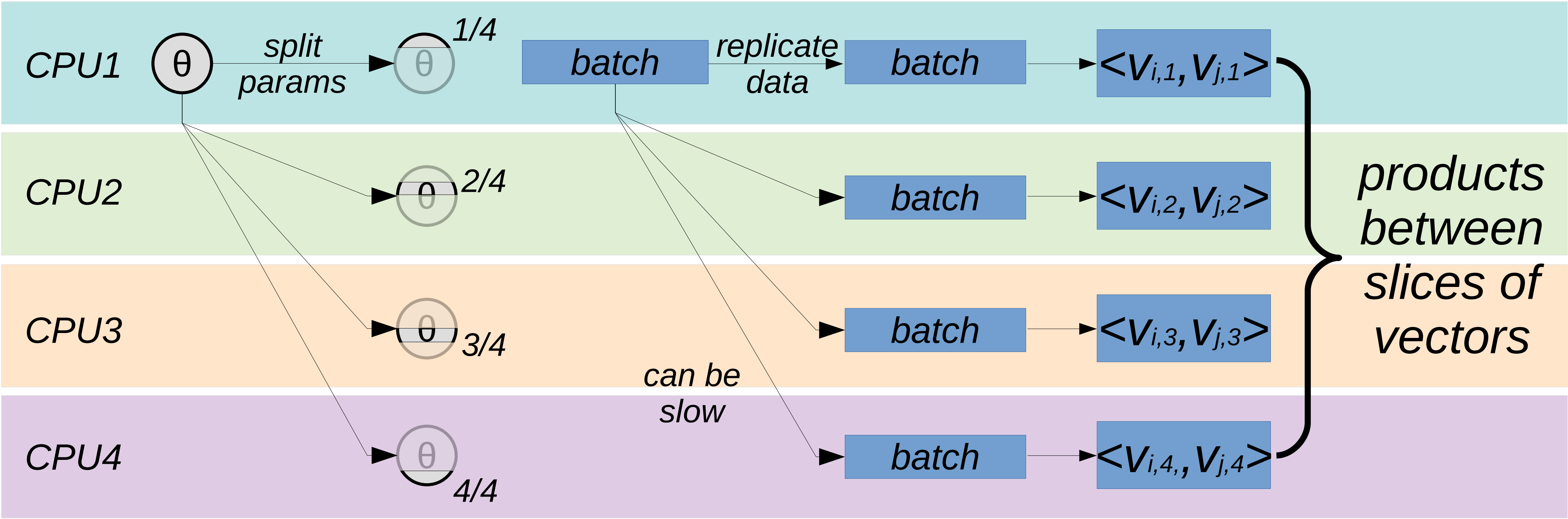
“Model parallelism”

*Each process runs **partial** model on **all** samples*



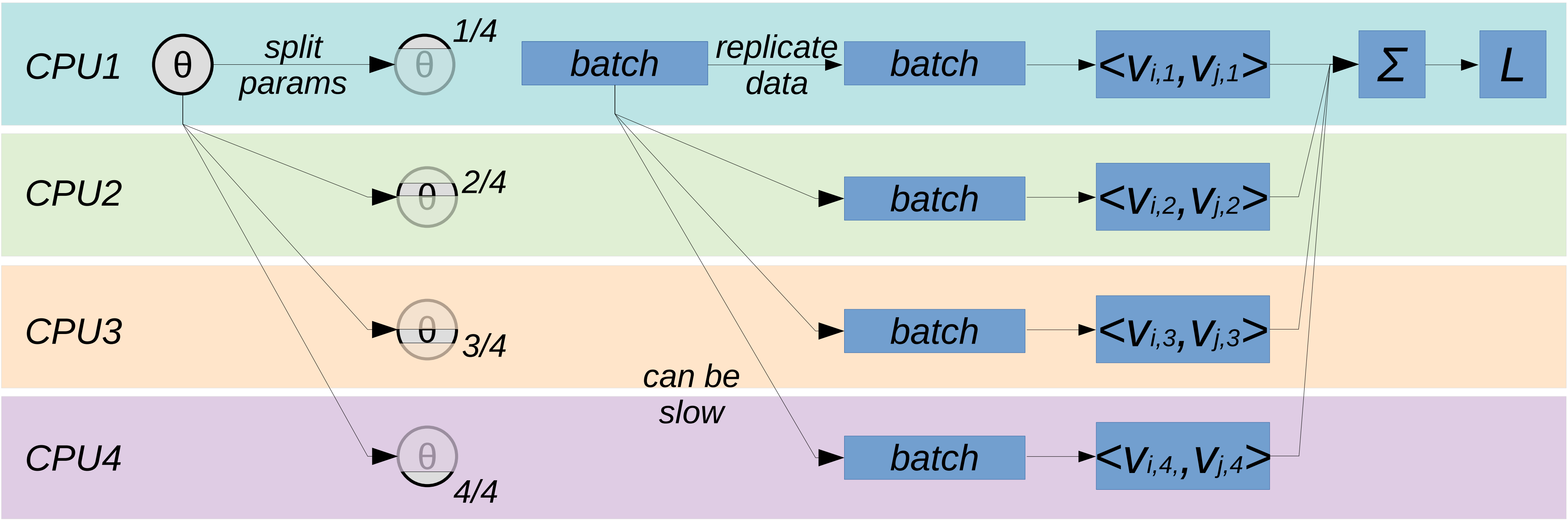
“Model parallelism”

*Each process runs **partial** model on **all** samples*



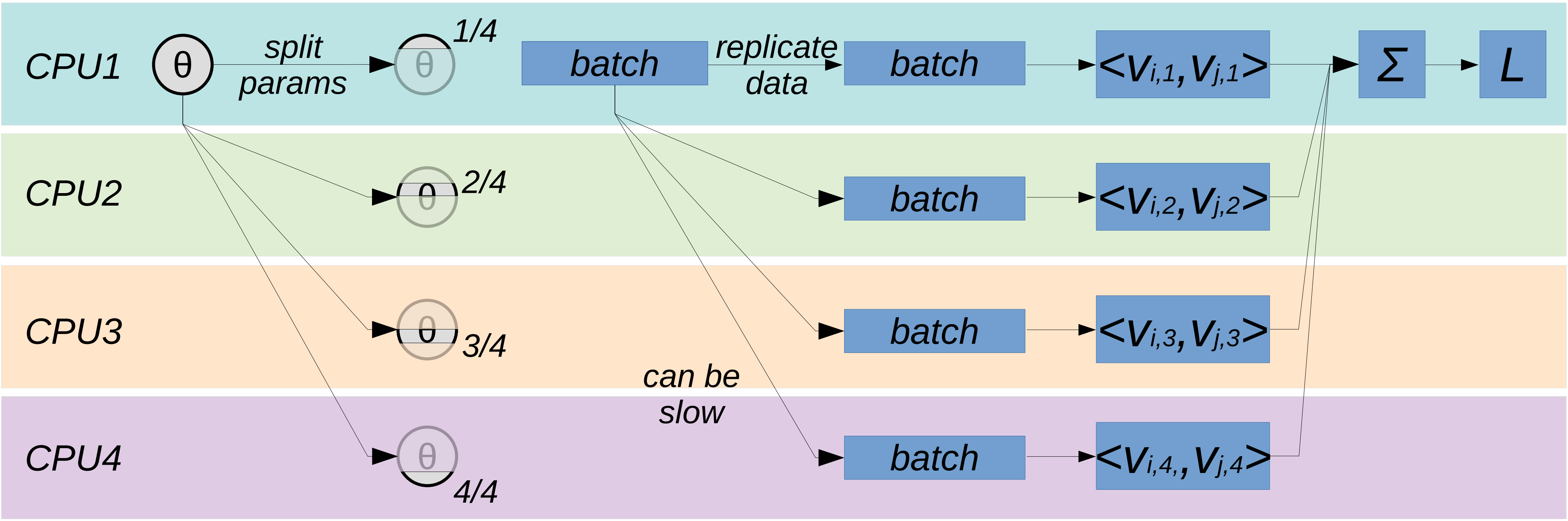
“Model parallelism”

*Each process runs **partial** model on **all** samples*



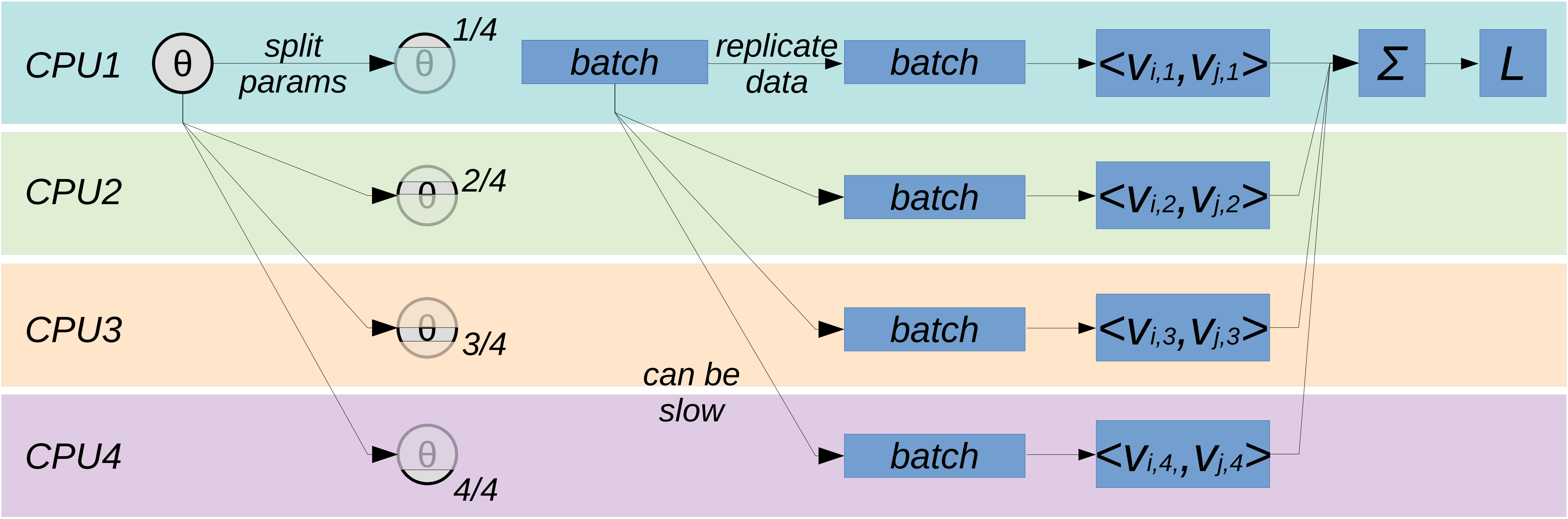
"Model parallelism"

***Q:** do we have to send params from P1 each time?*



"Model parallelism"

Optimization: each process can update it's own params locally instead of receiving them from P1



Summary: operation parallelism

Data-parallel: *one process applies all model on* ***partial data***

Model-parallel: *one process applies* ***partial model*** *on all data*

Which one is better..
for word2vec?
In general?

Summary: operation parallelism

Data-parallel: *one process applies all model on* ***partial data***

best for smaller model, more computations

Model-parallel: *one process applies* ***partial model*** *on all data*

best for larger model, fewer computations

Which one is better..

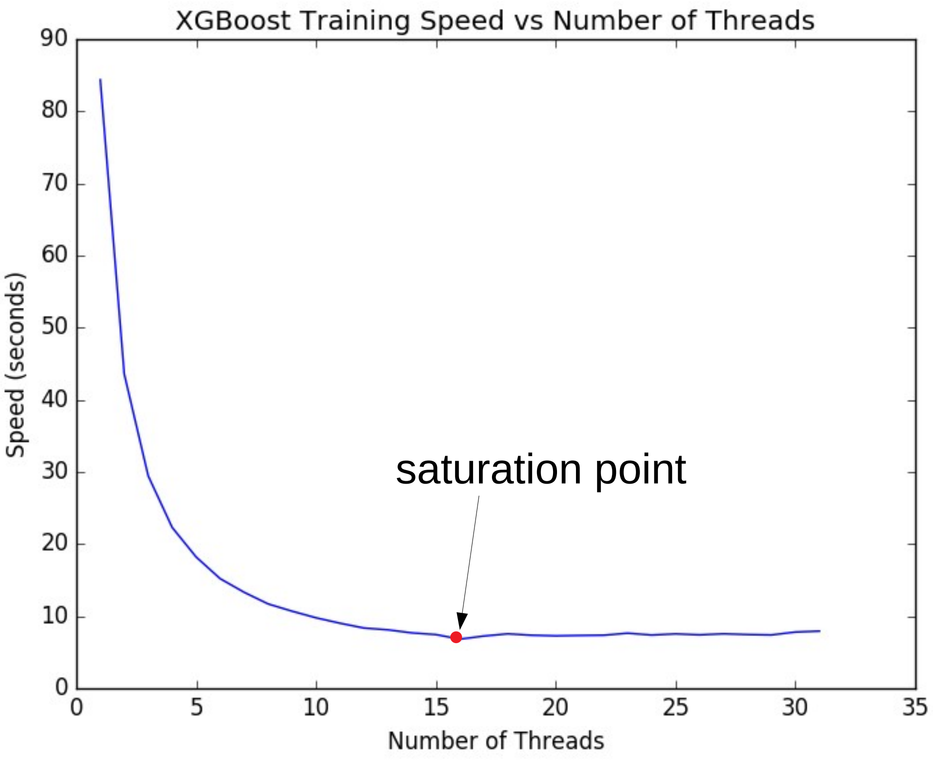
for word2vec?

In general?

Summary: operation parallelism

Если time() < 19:00, тут можно поговорить про бустинг.
и-или сделать перерыв

Operation parallelism

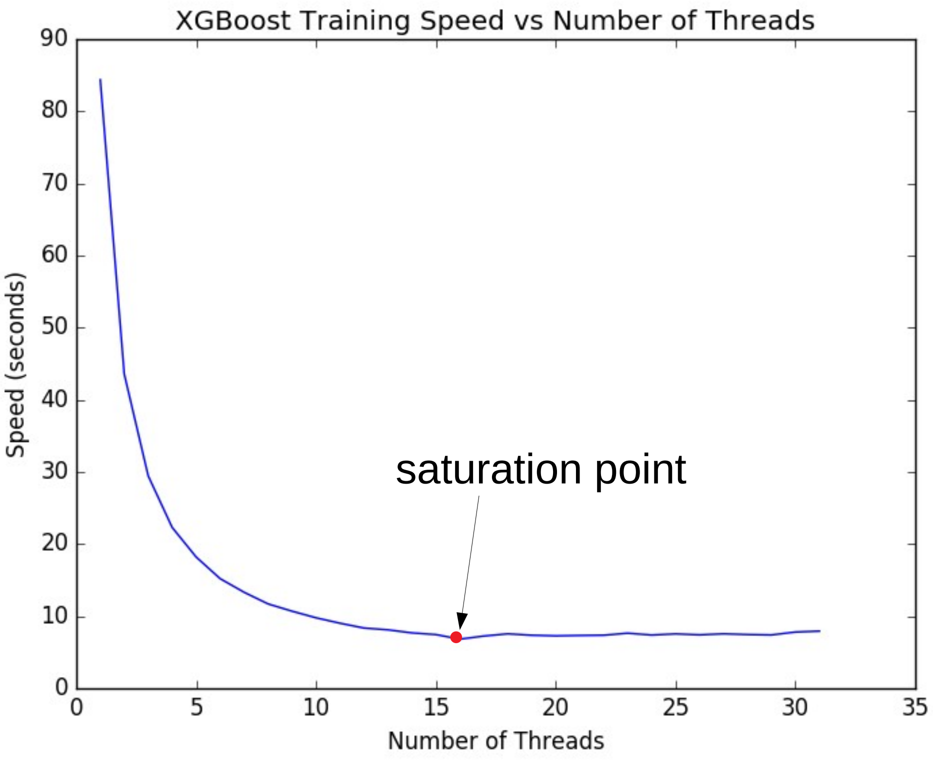


More processes = more overhead

- *waiting for each other*
- *sending data over the network*
- *performance fluctuations*

Eventually adding more threads will no longer boost performance

Operation parallelism



More processes = more overhead

- *waiting for each other*
- *sending data over the network*
- *performance fluctuations*

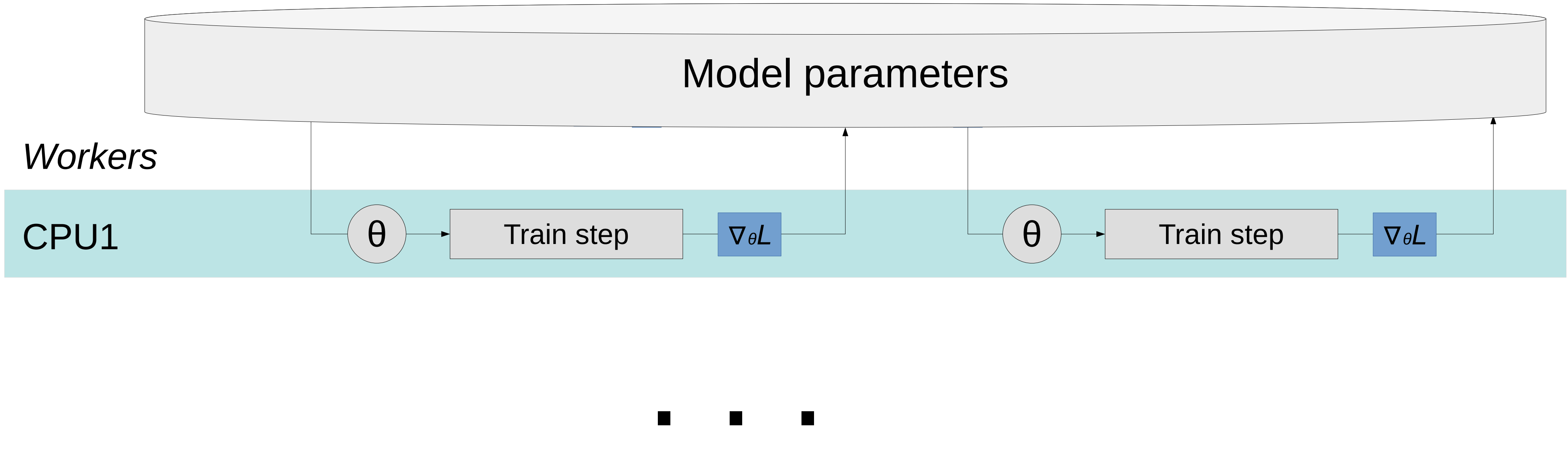
Eventually adding more threads will no longer boost performance

How do we push this point further?

Parameter Server

Paper: Smola et al. (2010)

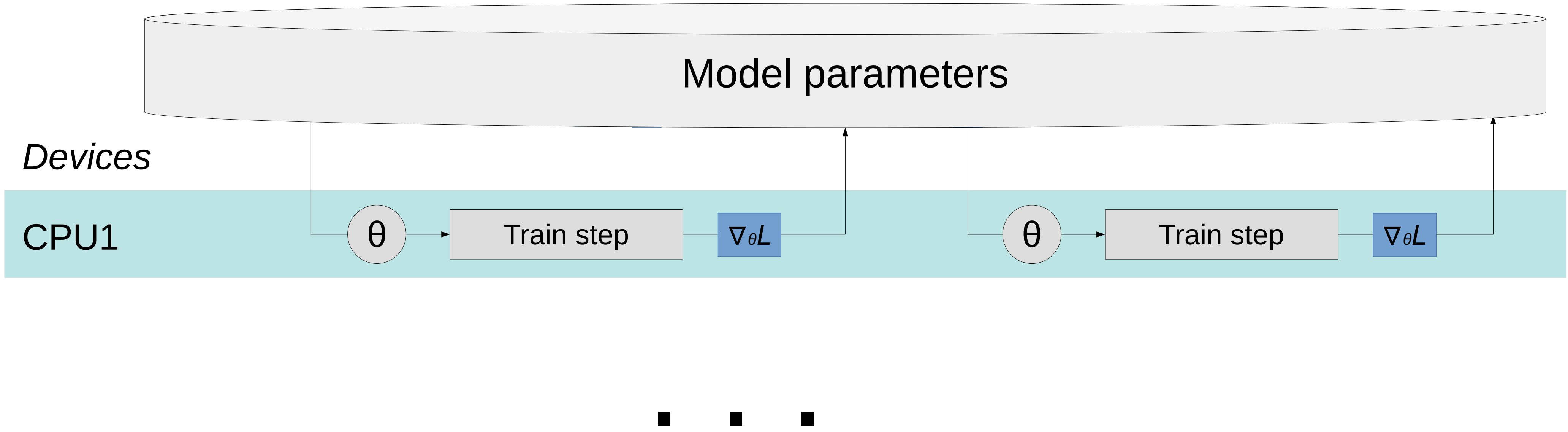
Make a dedicated process for parameters & optimizer



Asynchronous training

HOGWILD! arxiv.org/abs/1106.5730

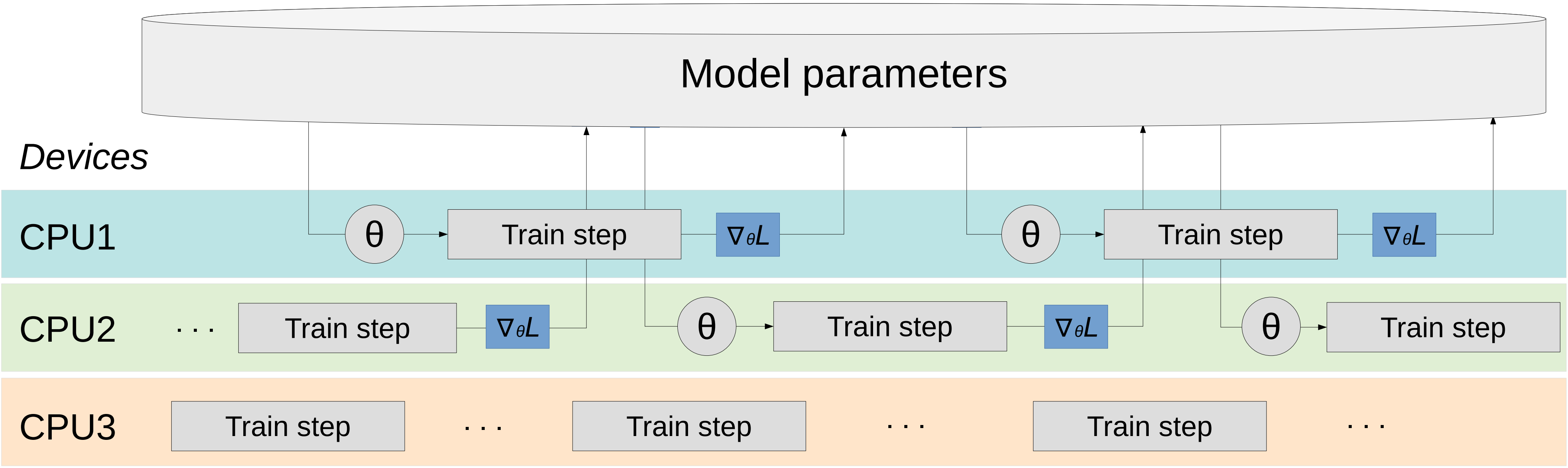
Idea: remove synchronization step alltogether, use parameter server



Asynchronous training

HOGWILD! arxiv.org/abs/1106.5730

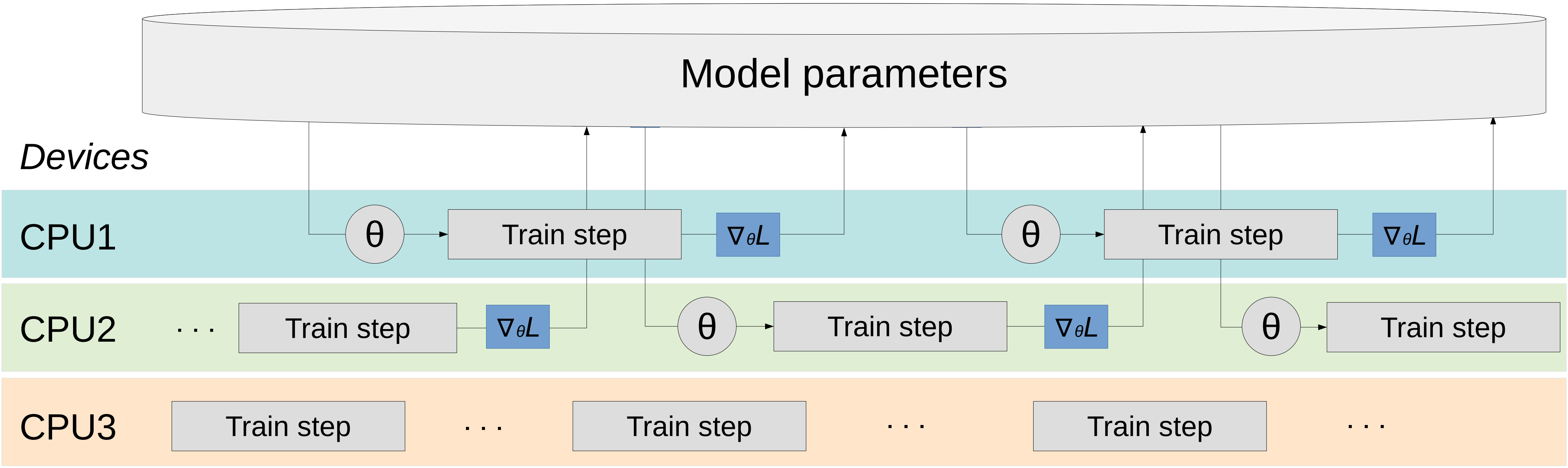
Idea: remove synchronization step alltogether, use parameter server



Asynchronous training

HOGWILD! arxiv.org/abs/1106.5730

Idea: remove synchronization step alltogether, use parameter server

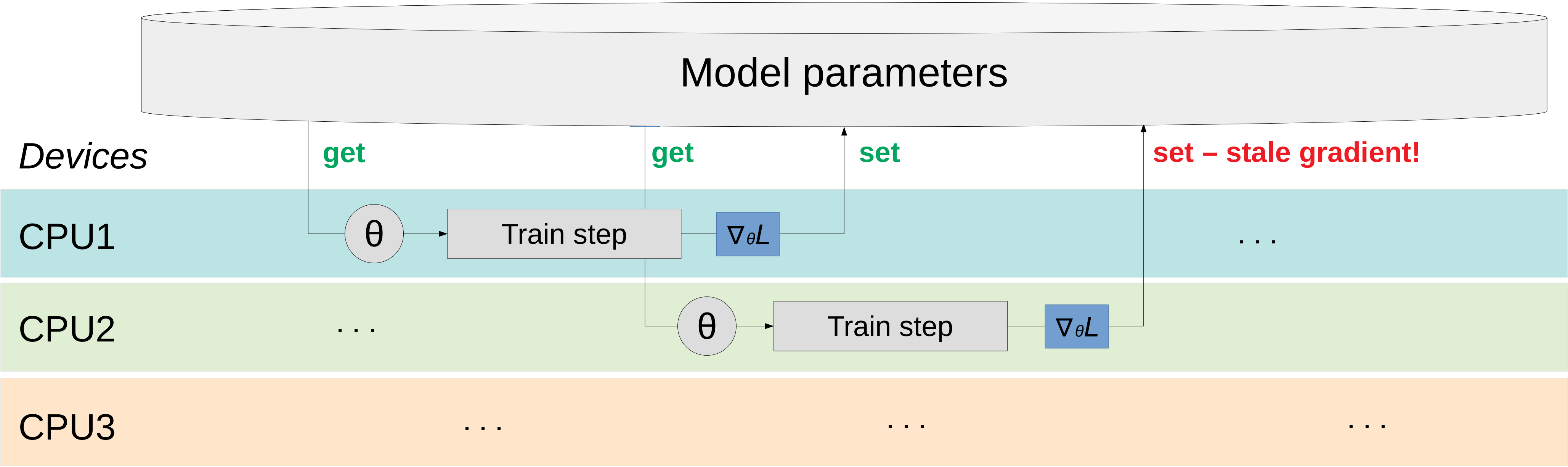


Q: have we lost anything by going asynchronous?

Asynchronous training

HOGWILD! arxiv.org/abs/1106.5730

Idea: remove synchronization step alltogether, use parameter server



Staleness-aware SGD

Paper: arxiv.org/abs/1511.05950 & others

Updates accumulated: $c = \lfloor (\lambda / n) \rfloor$

Average gradient: $$g_i = \frac{1}{c} \sum_{l=1}^{c} \alpha(\tau_{i,l}) \Delta\theta_l, \ l \in \{1, 2, \ldots, \lambda\}$$

New parameters: $\theta_{i+1} = \theta_i - g_i,$

Staleness-aware SGD

Paper: arxiv.org/abs/1511.05950 & others

n = total workers
λ = "accumulation factor"

Updates accumulated: $c = \lfloor (\lambda / n) \rfloor$

Average gradient: $g_i = \frac{1}{c} \sum_{l=1}^{c} \alpha(\tau_{i,l}) \Delta\theta_l, \; l \in \{1, 2, \ldots, \lambda\}$

New parameters: $\theta_{i+1} = \theta_i - g_i,$

Staleness-aware SGD

Paper: arxiv.org/abs/1511.05950 & others

Updates accumulated: $c = \lfloor (\lambda / n) \rfloor$

Average gradient: $$g_i = \frac{1}{c} \sum_{l=1}^{c} \underline{\alpha(\tau_{i,l})} \Delta\theta_l, \; l \in \{1, 2, \ldots, \lambda\}$$

staleness-dependent
"learning rate"

New parameters: $\theta_{i+1} = \theta_i - g_i,$

Staleness-aware SGD

Paper: arxiv.org/abs/1511.05950 & others

Updates accumulated: $c = \lfloor (\lambda / n) \rfloor$

Average gradient: $g_i = \frac{1}{c} \sum_{l=1}^{c} \alpha(\tau_{i,l}) \Delta\theta_l, \; l \in \{1, 2, \ldots, \lambda\}$

New parameters: $\theta_{i+1} = \theta_i - g_i,$

$$\alpha_{i,l} = \frac{\alpha_0}{\tau_{i,l}}$$

base learning rate

staleness (≥1)

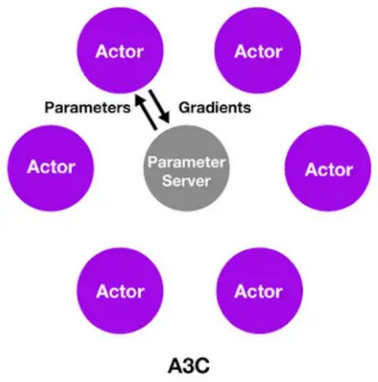
aka number of skipped updates

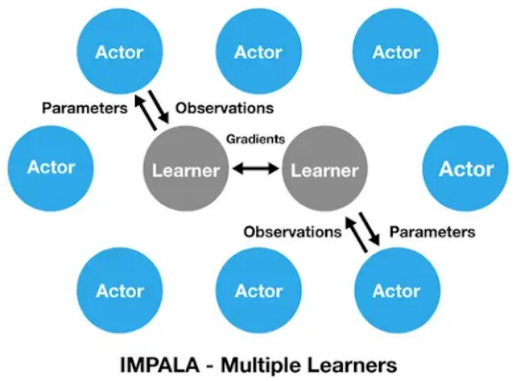
Parameter Server Applications

Conventional ML: *e.g. (Logistic Regression, CNN classifiers)*

Paper (sharded PS): https://www.cs.cmu.edu/~muli/file/ps.pdf
Another paper (optimizaton tricks): parameter_server_nips14.pdf
PyTorch tutorial (hogwild), TF tutorial (parameter server)

Reinforcement learning:





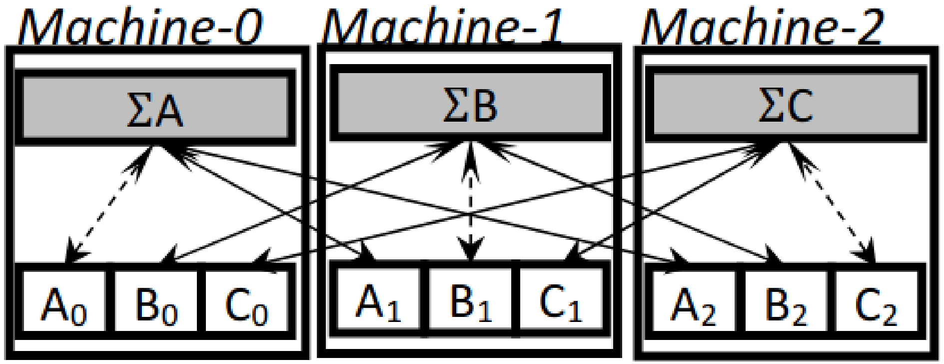
Async. RL: arxiv.org/abs/1602.01783
IMPALA: arxiv.org/abs/1802.01561
SEED RL: arxiv.org/abs/1910.06591

More:
(english) https://youtu.be/kOy49NqZeqI
(russian) https://youtu.be/wswbMkT55mI

Modern Parameter Server Systems

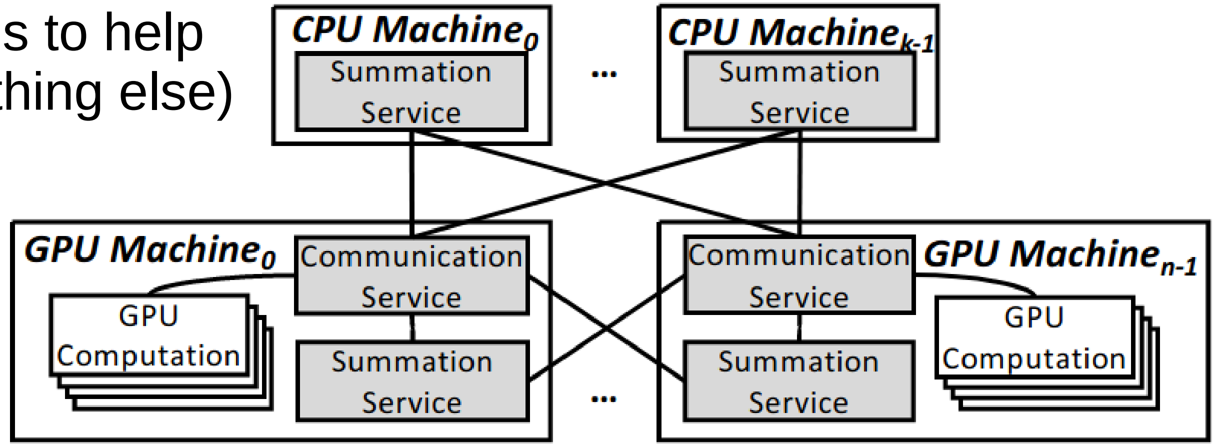
Read more: https://www.usenix.org/system/files/osdi20-jiang.pdf

Baseline: sharded PS on every node
Each server runs a subset of weights



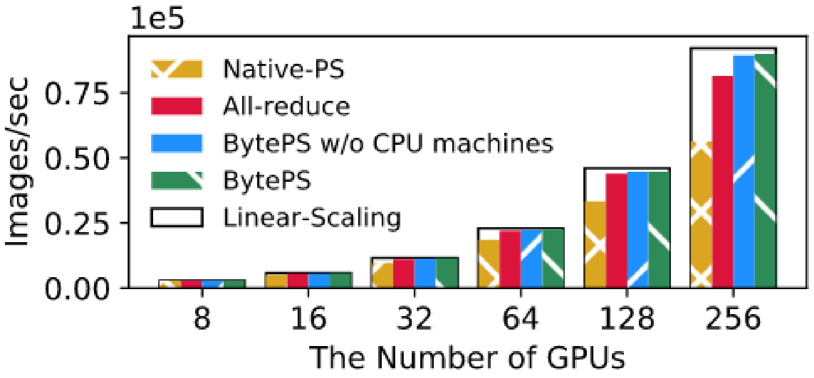
BytePS: add non-GPU nodes to help averaging gradients (and nothing else)

CPU nodes are ~10x cheaper to deploy / rent

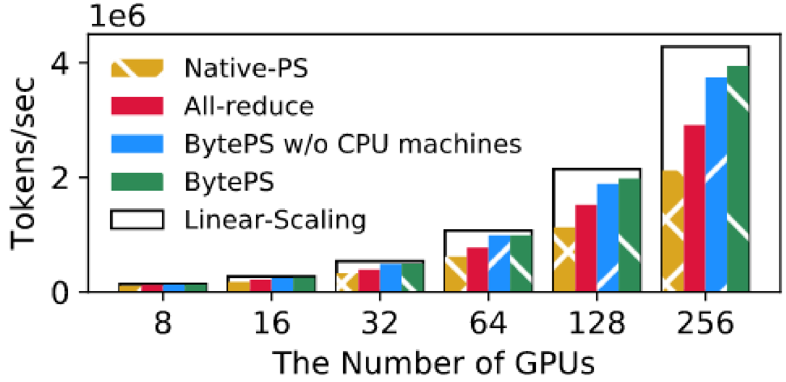


Modern Parameter Server Systems

Read more: https://www.usenix.org/system/files/osdi20-jiang.pdf

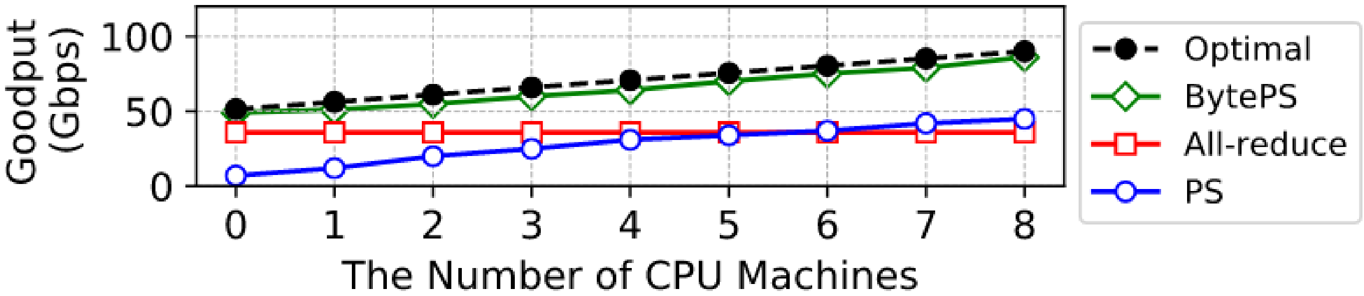


(a) TensorFlow, ResNet-50, batch=256 images



(b) MXNet, BERT-Large, batch=8192 tokens

The effect of auxiliary CPU nodes




"That's all Folks!"
Isberg©