# COMP 414/514: Optimization – Algorithms, Complexity and Approximations

Lecture 11

# Overview

$$\min_{x} \quad f(x)$$

$$\text{s.t.} \quad x \in \mathcal{C}$$

- Different objective classes
- Different strategies within each problem
- Different approaches based on computational capabilities
- Different approaches based on constraints

And, always having in mind applications in machine learning, AI and signal processing

The focus of this lecture

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & x \in \mathcal{C} \end{aligned}$$

# The focus of this lecture

$$\min_{x} \quad f(x)$$

$$\cancel{\text{s.t.} \quad x \in \mathcal{C}}$$

Unconstrained optimization

# The focus of this lecture

$$\min_x \quad f(x) := \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

$$\cancel{\text{s.t.} \quad x \in \mathcal{C}}$$

Unconstrained optimization

The focus of this lecture

Huge!

$$\min_{x} \quad f(x) := \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

$$\cancel{\text{s.t.} \quad x \in \mathcal{C}}$$

Unconstrained optimization

# Overview

- In this lecture, we will:
  - Discuss how to **distribute optimization in large-scale settings**
  - Study **synchrony vs. asynchrony** in gradient descent
  - Provide some rough theoretical results on how asynchrony affects performance
  - Alternatives and state of the art

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non−linear classification, neural networks)

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non-linear classification, neural networks)
- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non−linear classification, neural networks)
- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

- In words: $i$) we select a training sample, $ii$) we compute the gradient, $iii$) we update the model

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non−linear classification, neural networks)

- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

- In words: $i$) we select a training sample, $ii$) we compute the gradient, $iii$) we update the model

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non-linear classification, neural networks)

- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

- In words: $i)$ we select a training sample, $ii)$ we compute the gradient, $iii)$ we update the model

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non−linear classification, neural networks)

- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

- In words: $i$) we select a training sample, $ii$) we compute the gradient, $iii$) we update the model

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non−linear classification, neural networks)

- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

- In words: $i$) we select a training sample, $ii$) we compute the gradient, $iii$) we update the model

- Properties: $i$) the current model $x_t$ is used for the computation of $\nabla f_{i_t}(\cdot)$

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non−linear classification, neural networks)
- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

- In words: $i)$ we select a training sample, $ii)$ we compute the gradient, $iii)$ we update the model
- Properties: $i)$ the current model $x_t$ is used for the computation of $\nabla f_{i_t}(\cdot)$
  $ii)$ when we update the model, the state of the system is as when we read $x_t$

# Recall: Stochastic gradient descent

- SGD is used **almost everywhere**: training classical ML tasks (linear prediction, linear classification), training modern ML tasks (non-linear classification, neural networks)

- In simple math, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)$$

- In words: $i)$ we select a training sample, $ii)$ we compute the gradient, $iii)$ we update the model

- Properties: $i)$ the current model $x_t$ is used for the computation of $\nabla f_{i_t}(\cdot)$
  $ii)$ when we update the model, the state of the system is as when we read $x_t$
  $iii)$ The whole process is sequential

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t) = x_{t-1} - \eta \left( \nabla f_{i_t}(x_t) + \nabla f_{i_{t-1}}(x_{t-1}) \right) = \cdots = x_0 - \eta \sum_j \nabla f_{i_j}(x_j)$$

# How to run SGD on multiple processing units

- SGD (as presented above) operates on a single machine
  (single CPU, single memory, single communication bus line)

(Very rough description)

# How to run SGD on multiple processing units

- SGD (as presented above) operates on a single machine
  (single CPU, single memory, single communication bus line)

(Very rough description)

- Can we identify where computation/communication happens in:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)\,?$$

# How to run SGD on multiple processing units

- SGD (as presented above) operates on a single machine (single CPU, single memory, single communication bus line)

(Very rough description)

- Can we identify where computation/communication happens in:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)\,?$$

- $i)$ model $x_t$ needs to be "transferred" where computation of $\nabla f_{i_t}(\cdot)$ happens
  $ii)$ data point $f_{i_t}(\cdot)$ needs to be transferred where $\nabla f_{i_t}(\cdot)$ happens
  $iii)$ the update $x_t - \eta \nabla f_{i_t}(x_t)$ overwrites (usually) the current model

# How to run SGD on multiple processing units

- SGD (as presented above) operates on a single machine
  (single CPU, single memory, single communication bus line)

(Very rough description)

- Can we identify where computation/communication happens in:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)\,?$$

- $i)$ model $x_t$ needs to be ``transferred`` where computation of $\nabla f_{i_t}(\cdot)$ happens
  $ii)$ data point $f_{i_t}(\cdot)$ needs to be transferred where $\nabla f_{i_t}(\cdot)$ happens
  $iii)$ the update $x_t - \eta \nabla f_{i_t}(x_t)$ overwrites (usually) the current model

- ``But we have GPUs!``: Limitation is its memory (model/data do not fit)
  Not easy to parallelize on GPU

# How to run SGD on multiple processing units

- SGD (as presented above) operates on a single machine
  (single CPU, single memory, single communication bus line)

(Very rough description)

- Can we identify where computation/communication happens in:

$$x_{t+1} = x_t - \eta \nabla f_{i_t}(x_t)\ ?$$

- $i)$ model $x_t$ needs to be "transferred" where computation of $\nabla f_{i_t}(\cdot)$ happens
  $ii)$ data point $f_{i_t}(\cdot)$ needs to be transferred where $\nabla f_{i_t}(\cdot)$ happens
  $iii)$ the update $x_t - \eta \nabla f_{i_t}(x_t)$ overwrites (usually) the current model

- "But we have GPUs!": Limitation is its memory (model/data do not fit)
  Not easy to parallelize on GPU

- How can we distribute this computation over multiple processing units?

# Interlude: what we mean by ``distributed``?

# Interlude: what we mean by "distributed"?

- Disclaimer: there are people in Rice CS with 1000x more expertise on these topics (see John Mellor-Crummey)

(Spoiler alert: a very rough description next)

# Interlude: what we mean by "distributed"?

- Disclaimer: there are people in Rice CS with 1000x more expertise on these topics (see John Mellor-Crummey)

(Spoiler alert: a very rough description next)

- **Single node distributed computing:**
  - $i$) Single machine, many cores (up to 100s)
    $ii$) Shared memory (all processors have access to it)
    $iii$) Communication to RAM is relatively cheap

# Interlude: what we mean by "distributed"?

- Disclaimer: there are people in Rice CS with 1000x more expertise on these topics (see John Mellor-Crummey)

(Spoiler alert: a very rough description next)

- **Single node distributed computing**:
    - *i*) Single machine, many cores (up to 100s)
      *ii*) Shared memory (all processors have access to it)
      *iii*) Communication to RAM is relatively cheap

- **Multi-node distributed computing**:
    - *i*) Many machines (up to 1000s), probably with many cores each
      *ii*) Shared-nothing architecture (each machine has its own CPU, storage)
      *iii*) Communication between nodes is much less cheap than single node

# Distributing gradient computations

- Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

# Distributing gradient computations

- Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

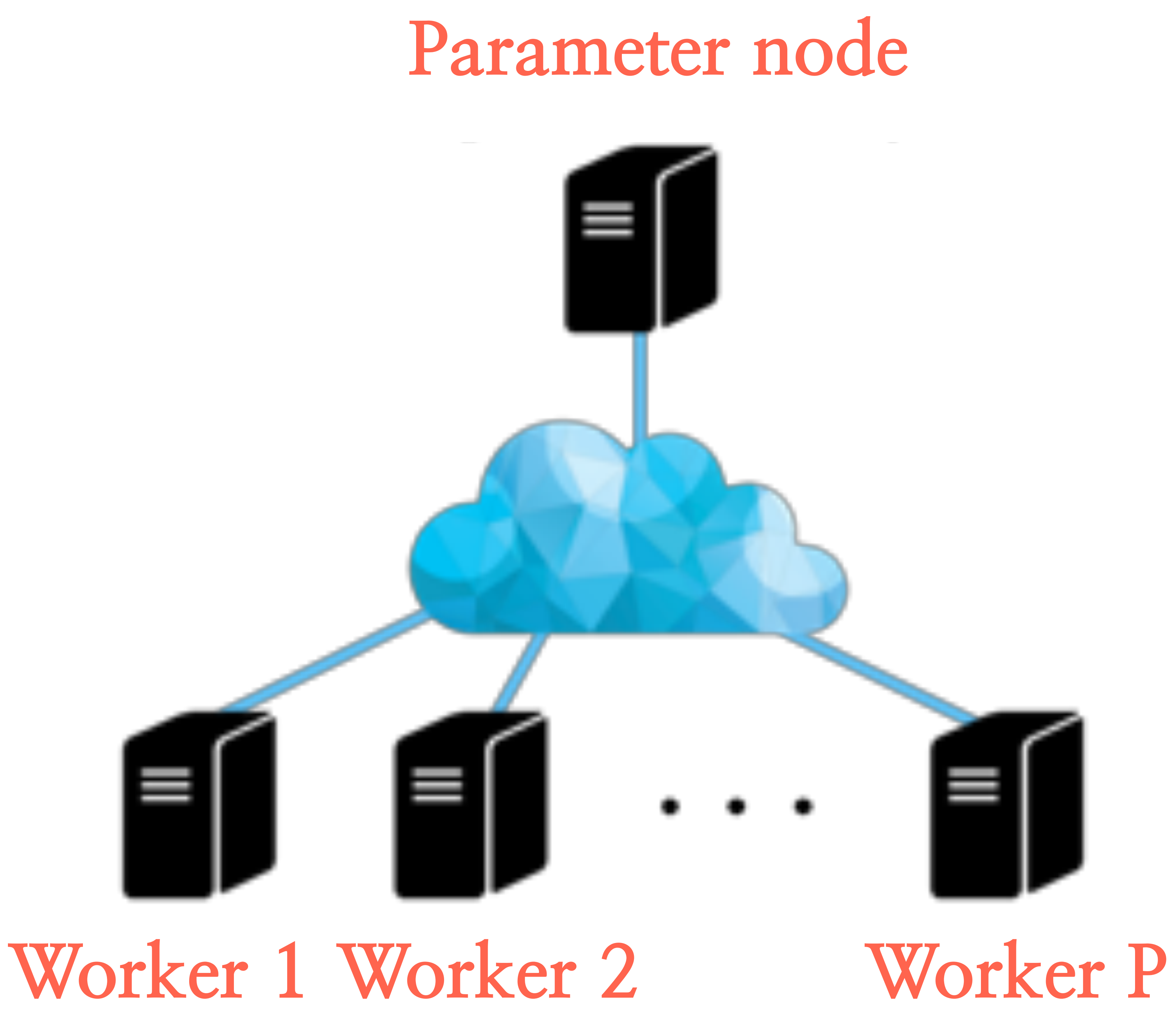
Clear use of dist. computing

# Distributing gradient computations

- Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

Clear use of dist. computing



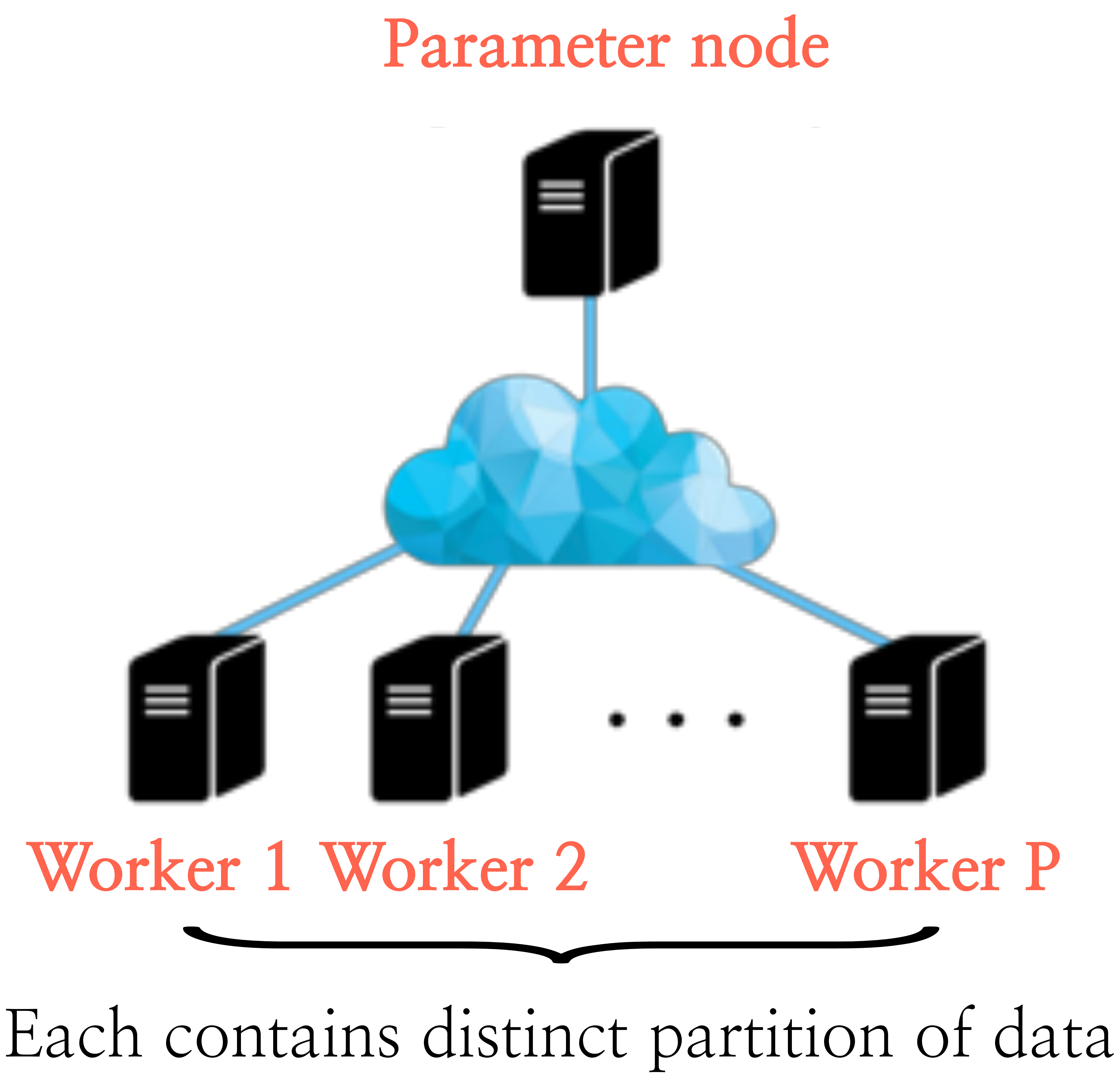

Each contains distinct partition of data

# Distributing gradient computations

– Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

Clear use of dist. computing



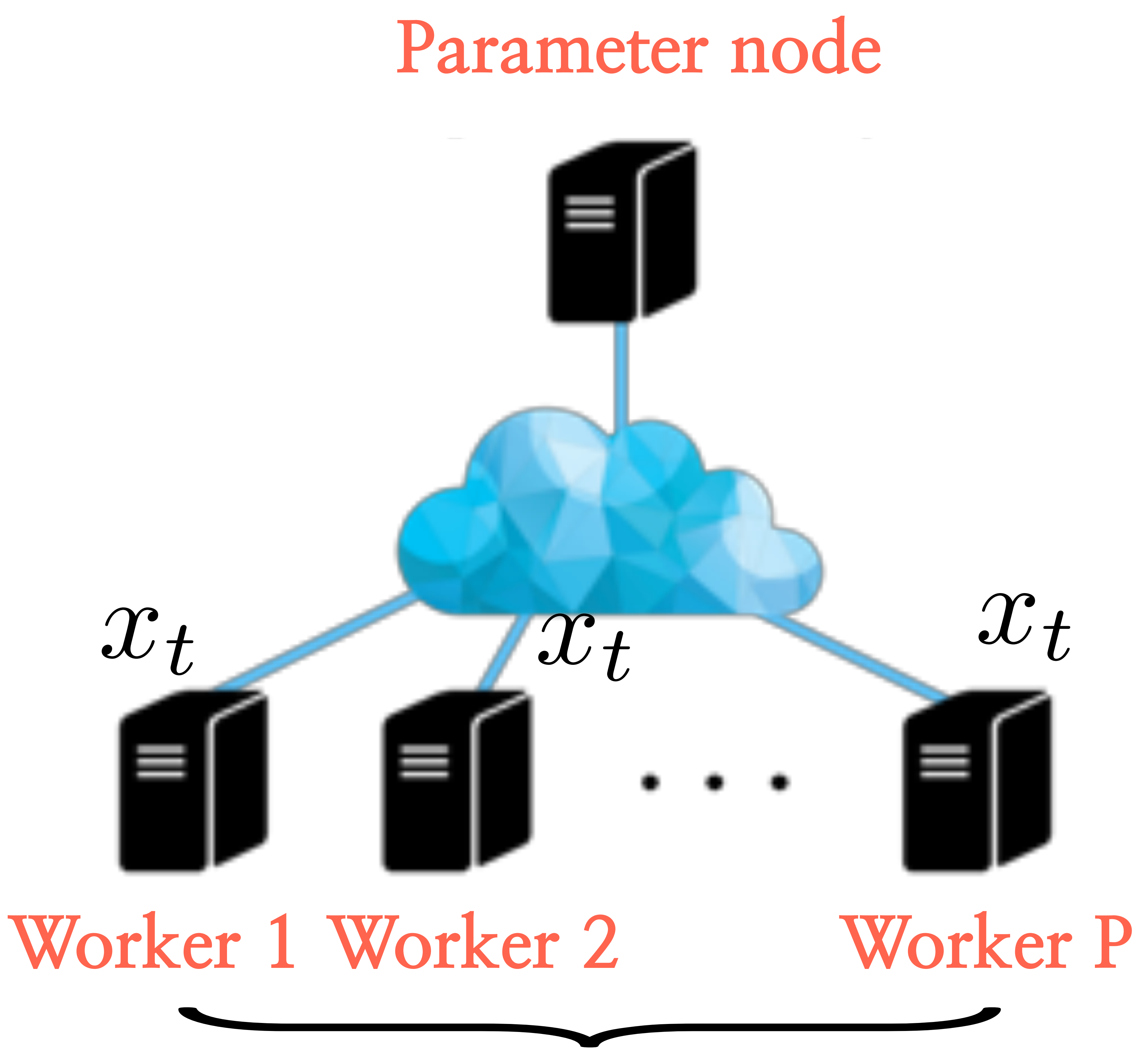

*i*) Parameter node keeps and distributes model $x_t$ at every cycle/iteration

# Distributing gradient computations

– Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

Clear use of dist. computing




Each contains distinct partition of data

$i)$ Parameter node keeps and distributes model $x_t$ at every cycle/iteration

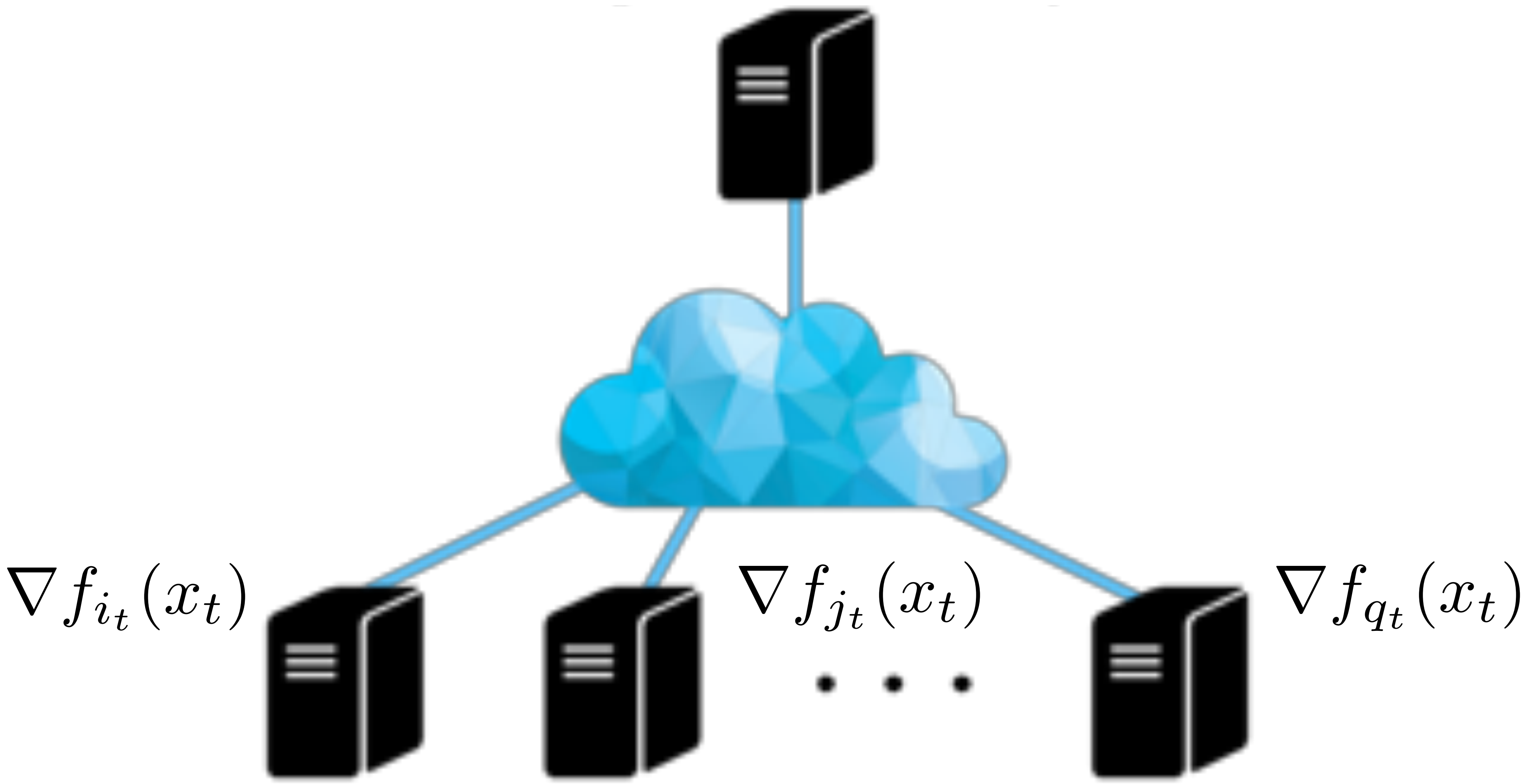
# Distributing gradient computations

– Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

Clear use of dist. computing

Parameter node




Worker 1 Worker 2 Worker P

Each contains distinct partition of data

$i)$ Parameter node keeps and distributes model $x_t$ at every cycle/iteration

$ii)$ Worker nodes compute part of the full gradient, based on the part of data they have

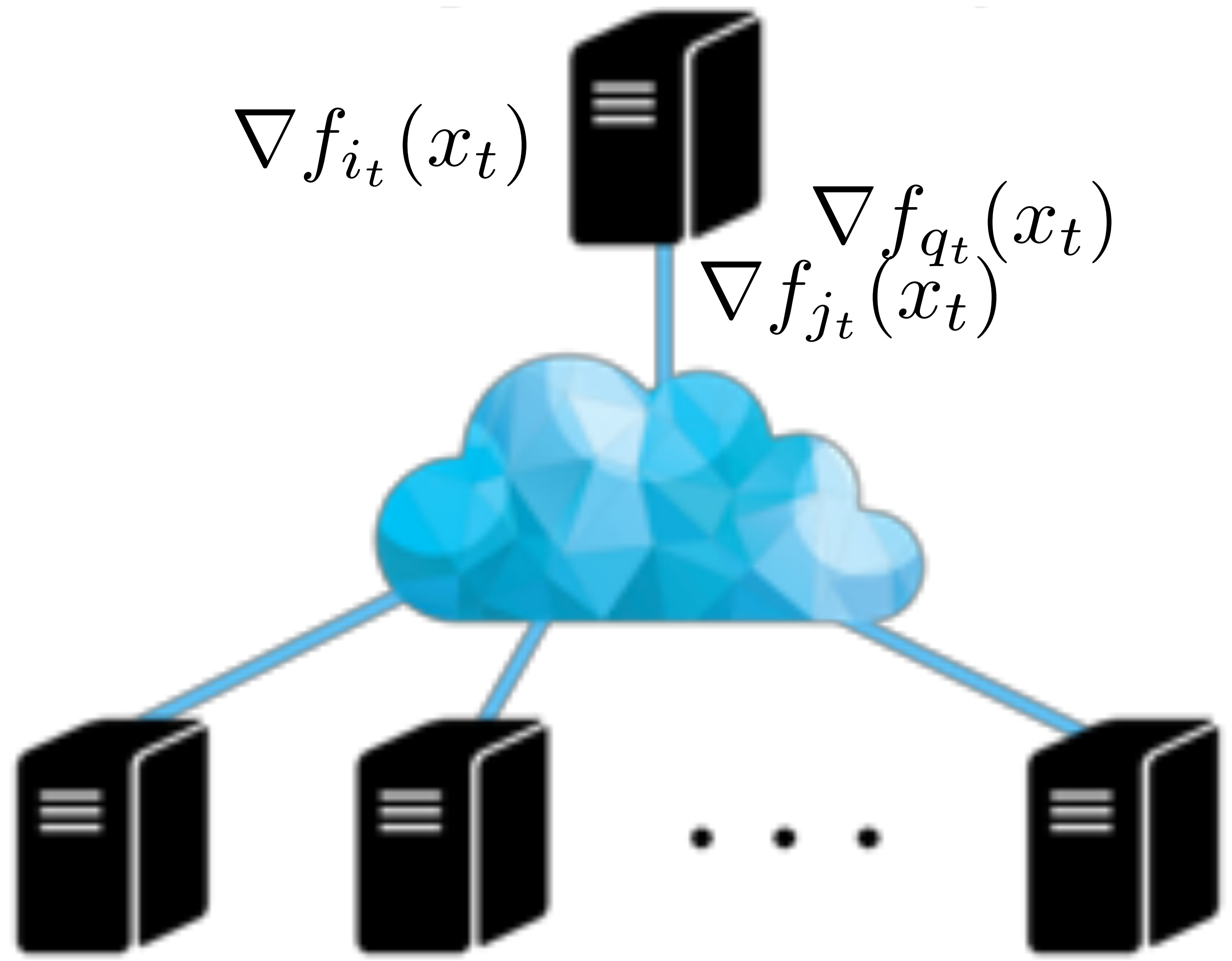
# Distributing gradient computations

– Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

Clear use of dist. computing

Parameter node




Worker 1 Worker 2 Worker P

Each contains distinct partition of data

$i)$ Parameter node keeps and distributes model $x_t$ at every cycle/iteration

$ii)$ Worker nodes compute part of the full gradient, based on the part of data they have

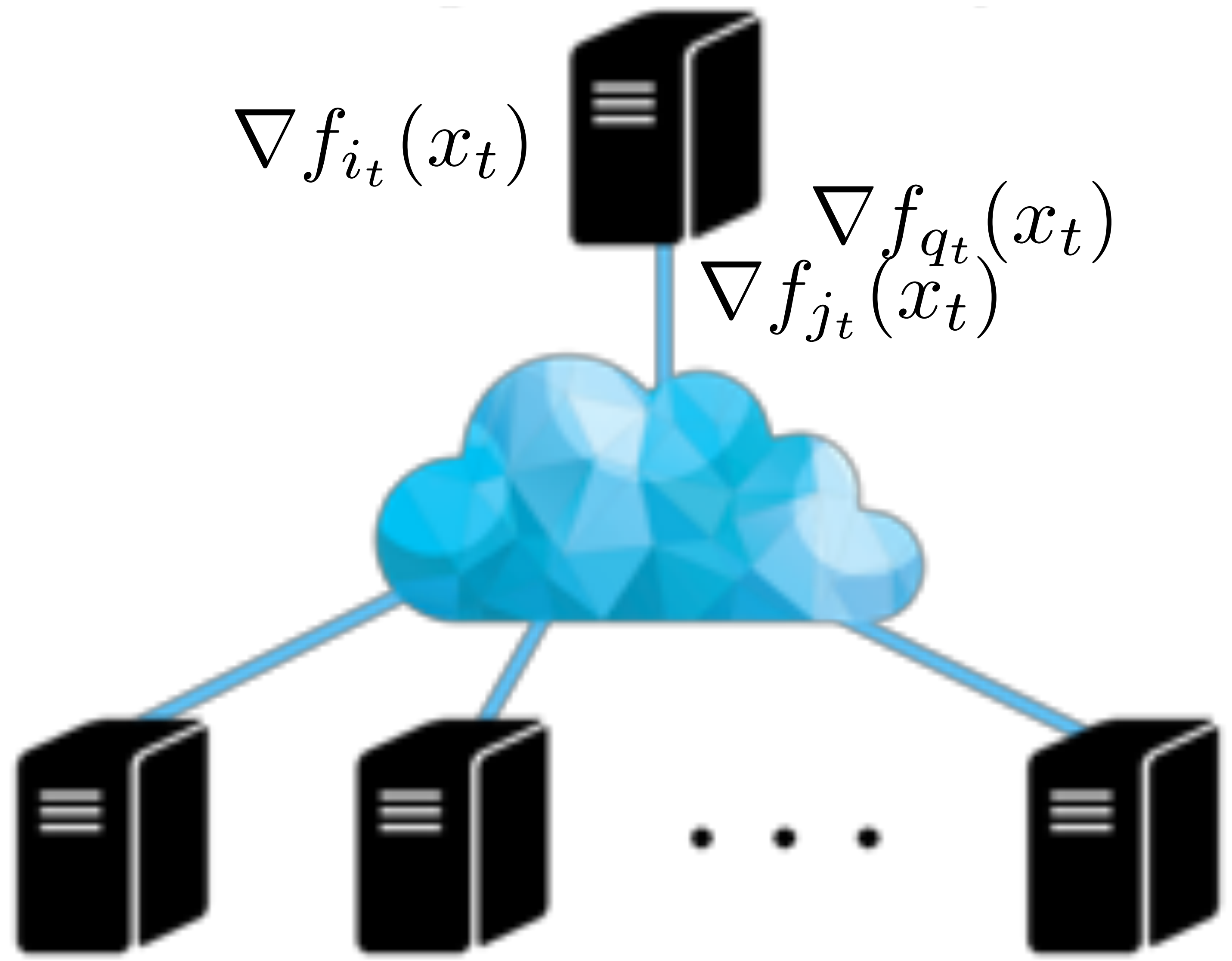
# Distributing gradient computations

– Consider the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

Clear use of dist. computing

Parameter node




Worker 1 Worker 2 Worker P

Each contains distinct partition of data

*i*) Parameter node keeps and distributes model $x_t$ at every cycle/iteration

*ii*) Worker nodes compute part of the full gradient, based on the part of data they have

*iii*) Parameter node waits for **all gradient parts** to be collected to do the gradient step

(..till the very last slow worker – active research: tackle stranglers)

# Distributing gradient computations

- "Things are looking good so far.. What's wrong with this scheme?"

# Distributing gradient computations

- ``Things are looking good so far.. What's wrong with this scheme?``
- ``Well, it might be the case that we don't have all data at once``

**Online learning**: 1. Data samples arrive one-at-a-time, as we optimize

2. (For some reason), we don't have access to all data

# Distributing gradient computations

- ``Things are looking good so far.. What's wrong with this scheme?``
- ``Well, it might be the case that we don't have all data at once``

  **Online learning**: 1. Data samples arrive one-at-a-time, as we optimize

  2. (For some reason), we don't have access to all data

- ``But, there are cases where we have finite & fixed data – see neural networks``

# Distributing gradient computations

- ``Things are looking good so far.. What's wrong with this scheme?``
- ``Well, it might be the case that we don't have all data at once``

**Online learning** : 1. Data samples arrive one-at-a-time, as we optimize

2. (For some reason), we don't have access to all data

- ``But, there are cases where we have finite & fixed data – see neural networks``
- ``Well, the problem here is that full gradient descent does not perform well``

**Generalization vs. training error** :

1. If we care about only the training error, full GD could work well
2. In ML tasks, we often care about the generalization error, i.e., the performance of the model on unseen data

# Training vs. generalization error and GD

- Gradient descent converges to the "first-seen" stationary point; SGD explores a bit the landscape before converging

# Training vs. generalization error and GD

- Gradient descent converges to the "first-seen" stationary point; SGD explores a bit the landscape before converging

- Gradient descent overfits the landscape of training data; however the performance deteriorates on unseen data (different landscape)

## Whiteboard

# Training vs. generalization error and GD

- Gradient descent converges to the "first-seen" stationary point; SGD explores a bit the landscape before converging

- Gradient descent overfits the landscape of training data; however the performance deteriorates on unseen data (different landscape)

Whiteboard

(This relates to the question "large vs. small batch training")

# Distributing gradient computations

- Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

# Distributing gradient computations

- Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

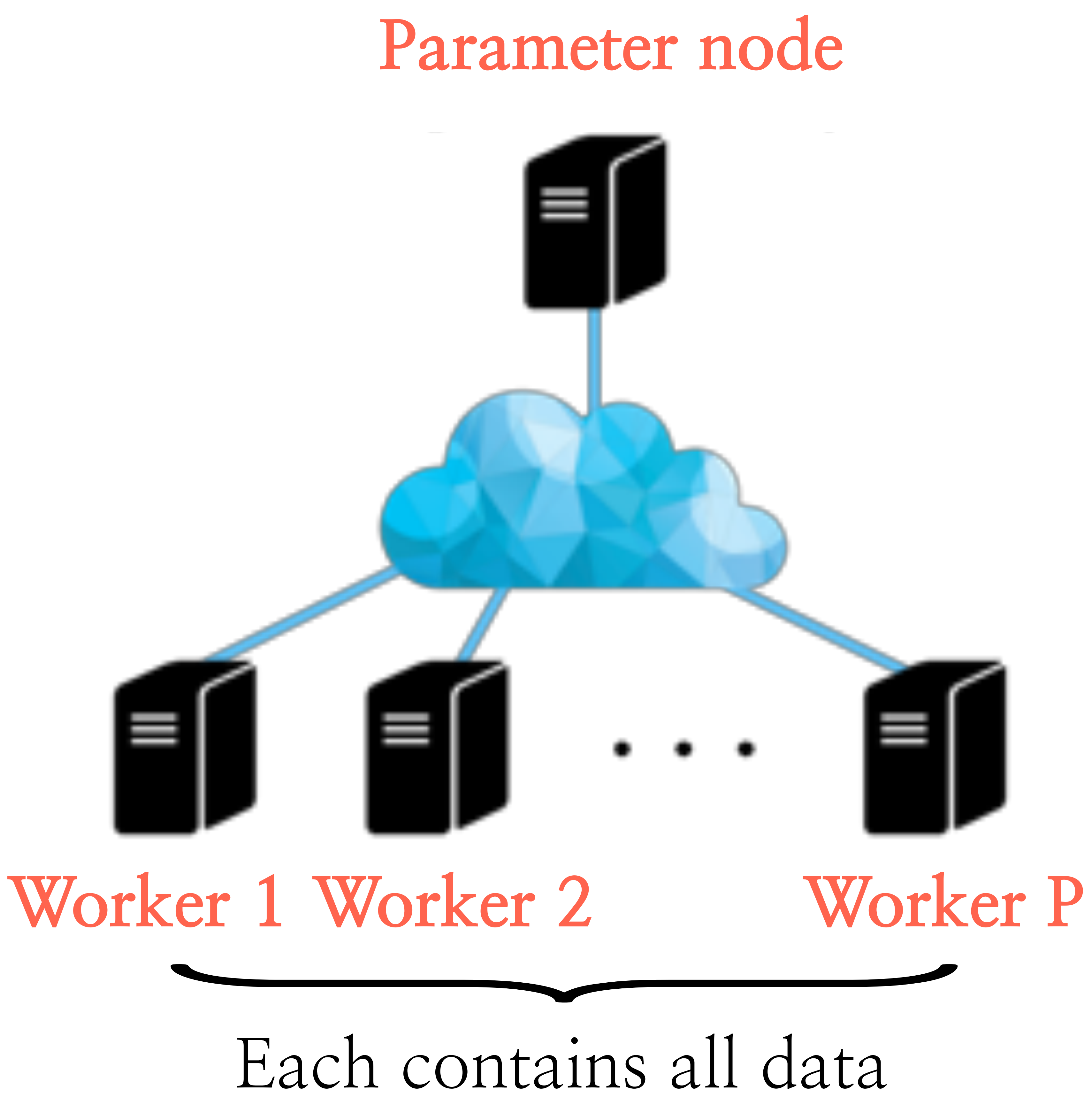
Parallelism in coordinates

# Distributing gradient computations

- Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates



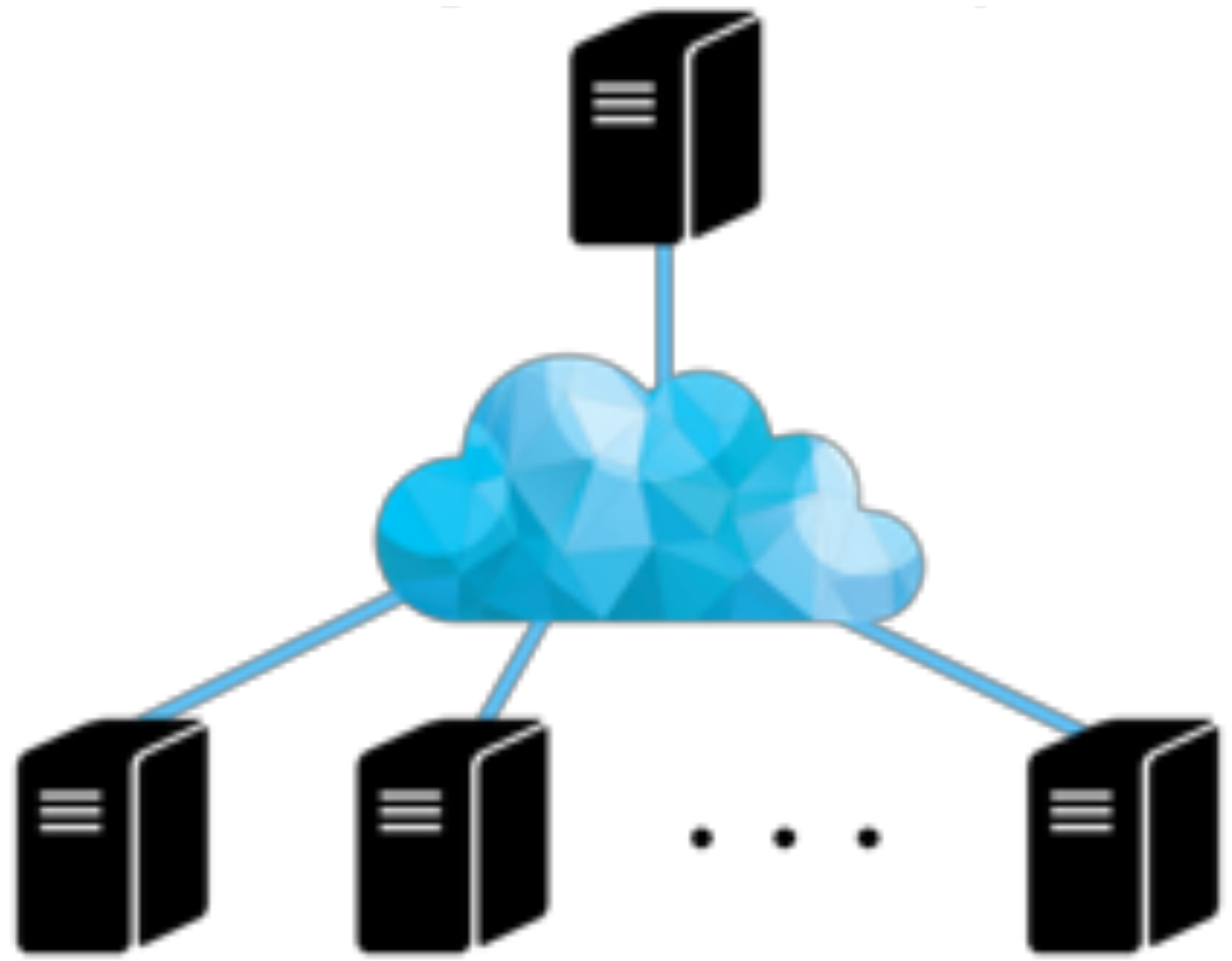
# Distributing gradient computations

– Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates

Parameter node



Worker 1 Worker 2 Worker P

Each contains all data

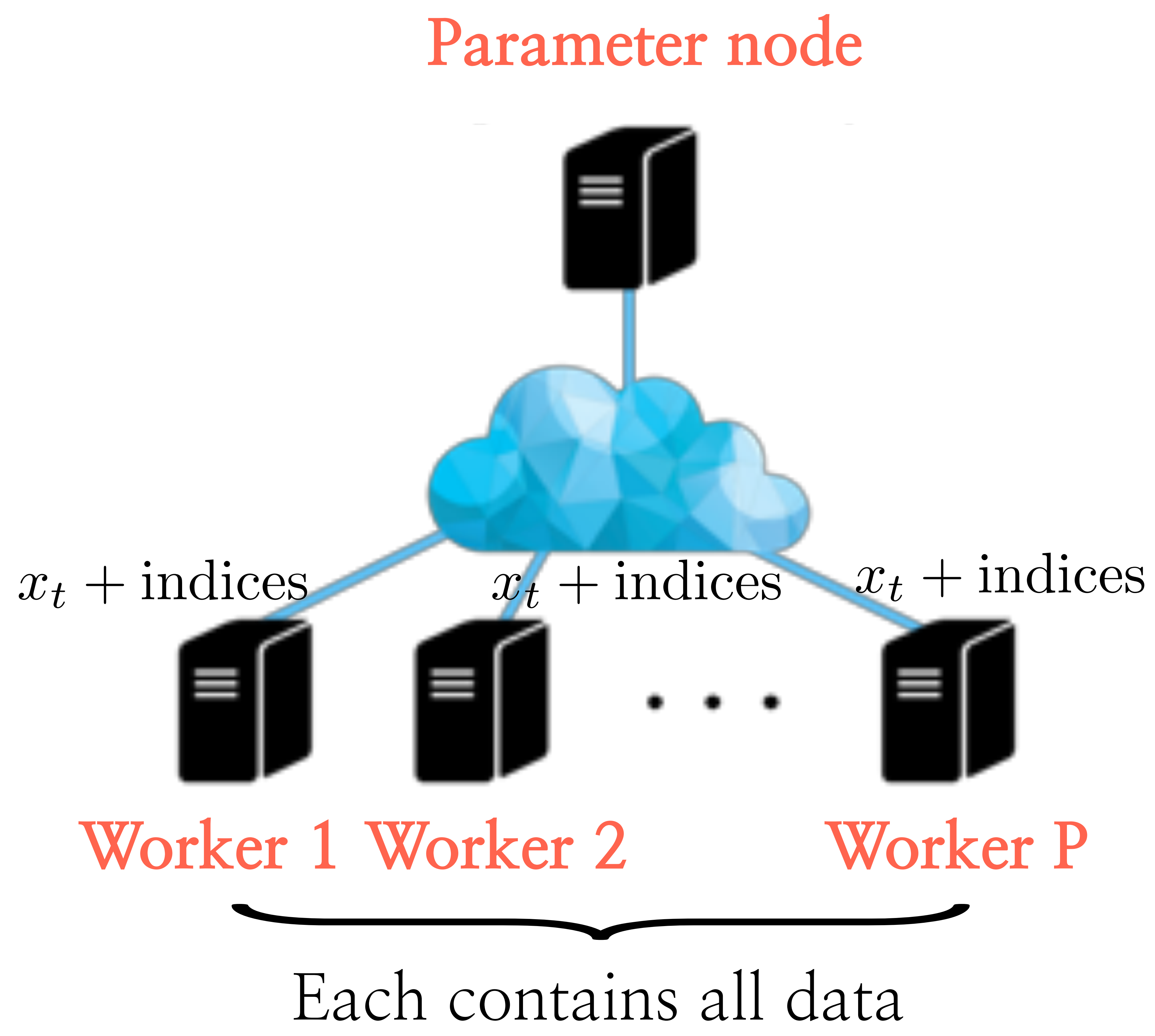
$i)$ Parameter node keeps and distributes model $x_t$ at every cycle/iteration

# Distributing gradient computations

– Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates




$i)$ Parameter node keeps and distributes model $x_t$ at every cycle/iteration

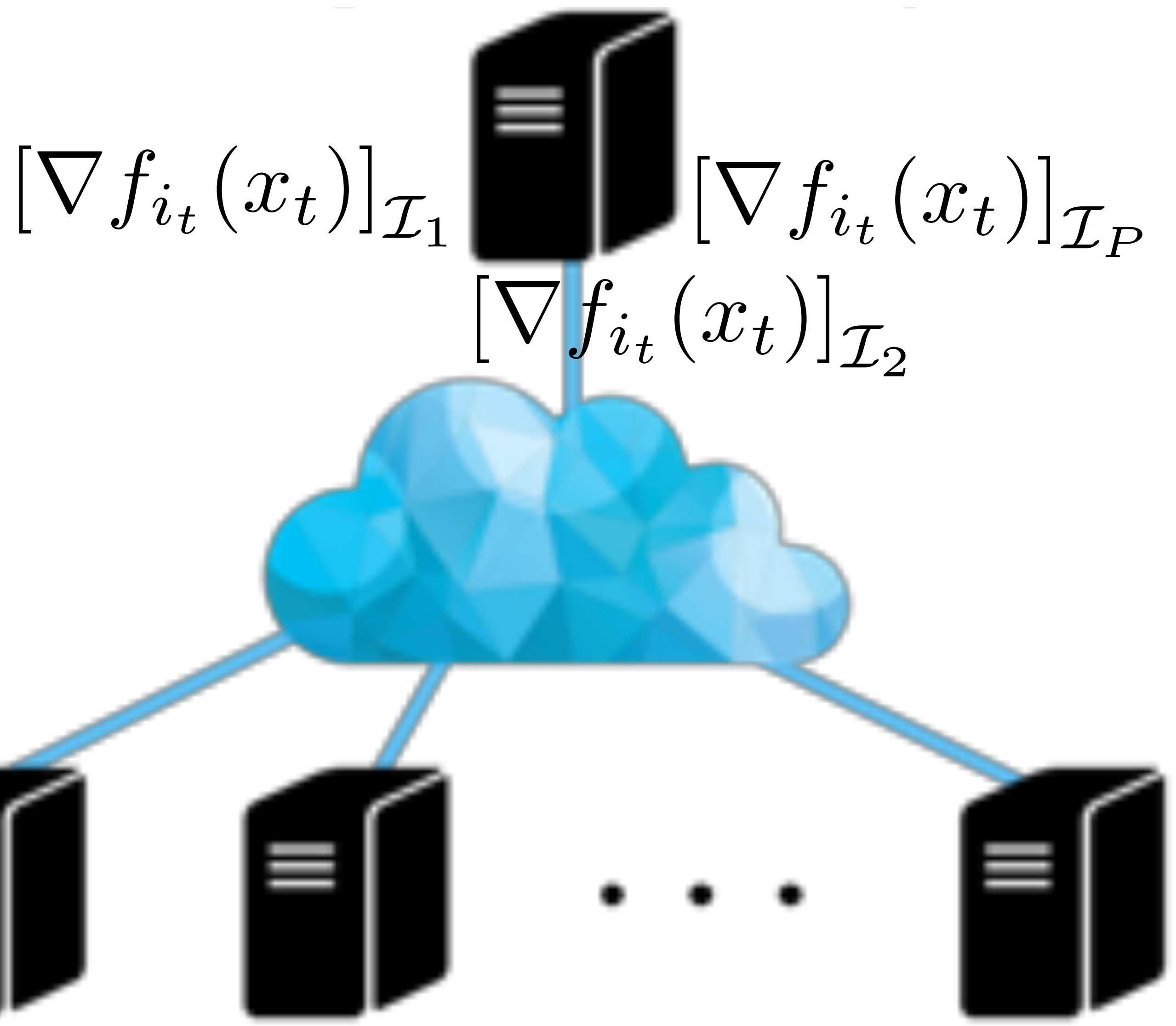
# Distributing gradient computations

– Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates

Parameter node




Worker 1 Worker 2 Worker P

Each contains all data

*i)* Parameter node keeps and distributes model $x_t$ at every cycle/iteration

*ii)* Worker nodes compute part of (stochastic) gradient, based on the coordinates they are asked by the parameter node

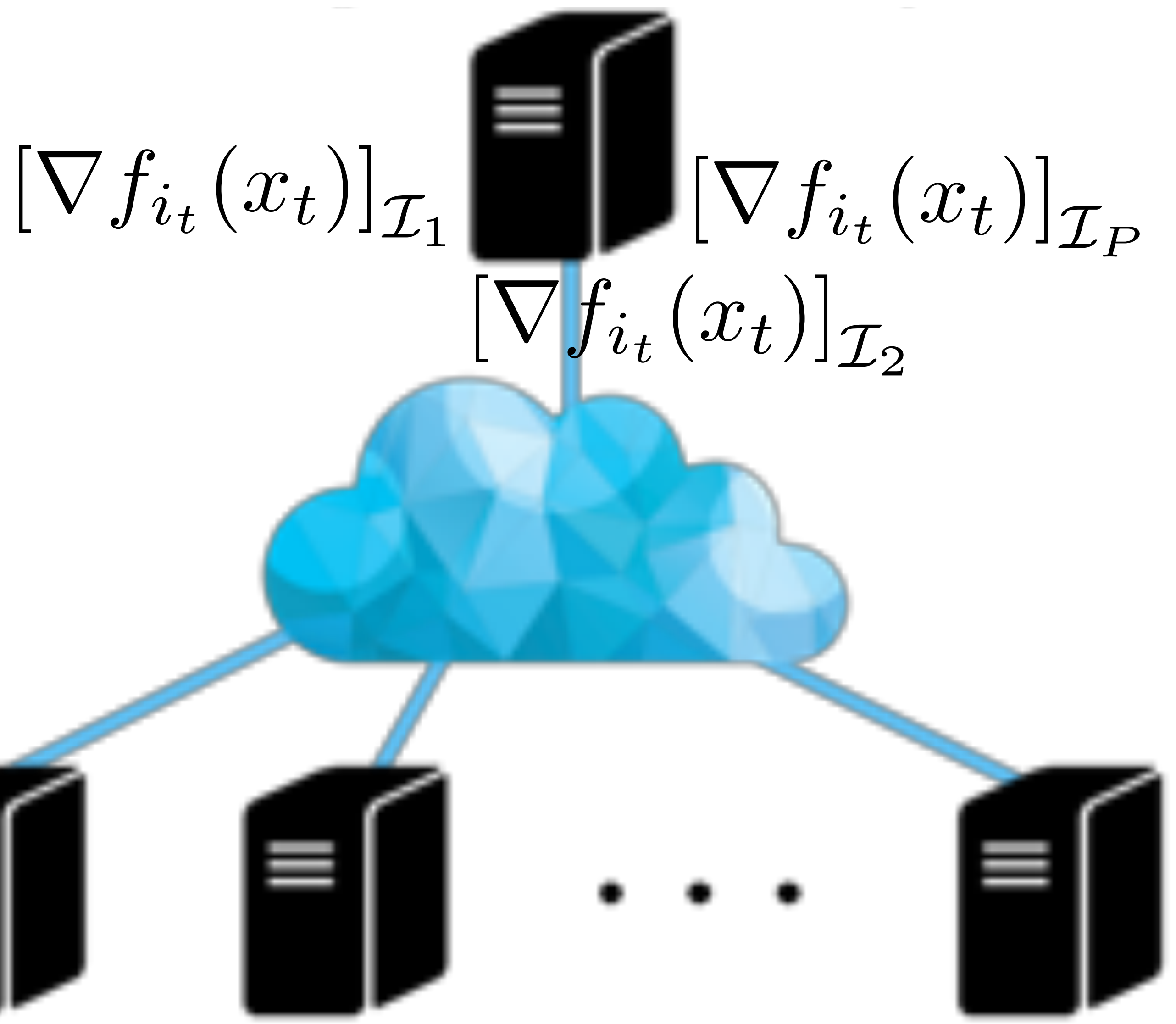
# Distributing gradient computations

- Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates

Parameter node




Worker 1 Worker 2 Worker P

Each contains all data

*i*) Parameter node keeps and distributes model $x_t$ at every cycle/iteration

*ii*) Worker nodes compute part of (stochastic) gradient, based on the coordinates they are asked by the parameter node

*iii*) Parameter node waits for **all gradient parts** to be collected to do the gradient step

(..till the very last slow worker)

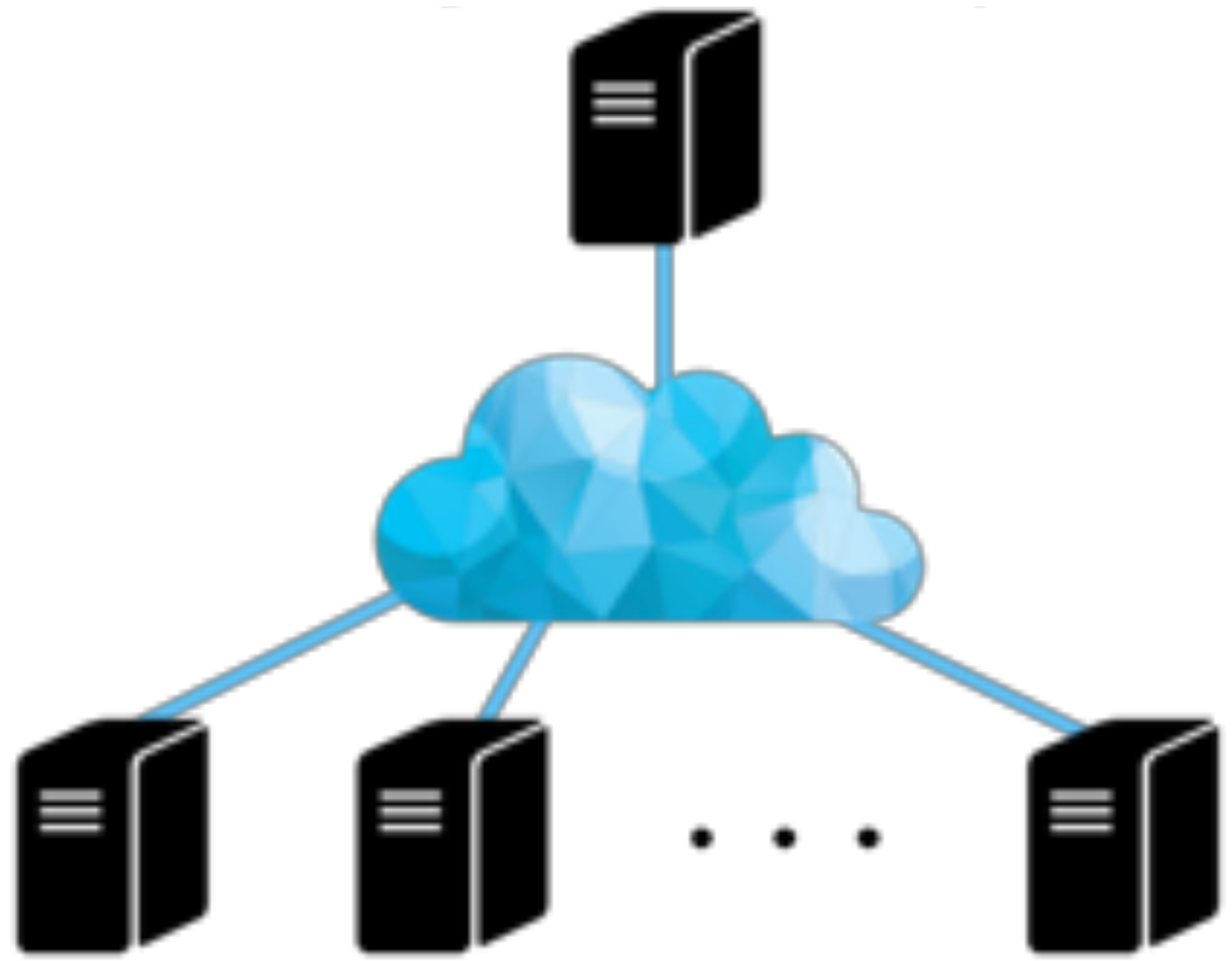
# Distributing gradient computations

– Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates

Parameter node




Each contains all data

$i)$ Relates to coordinate descent algorithms

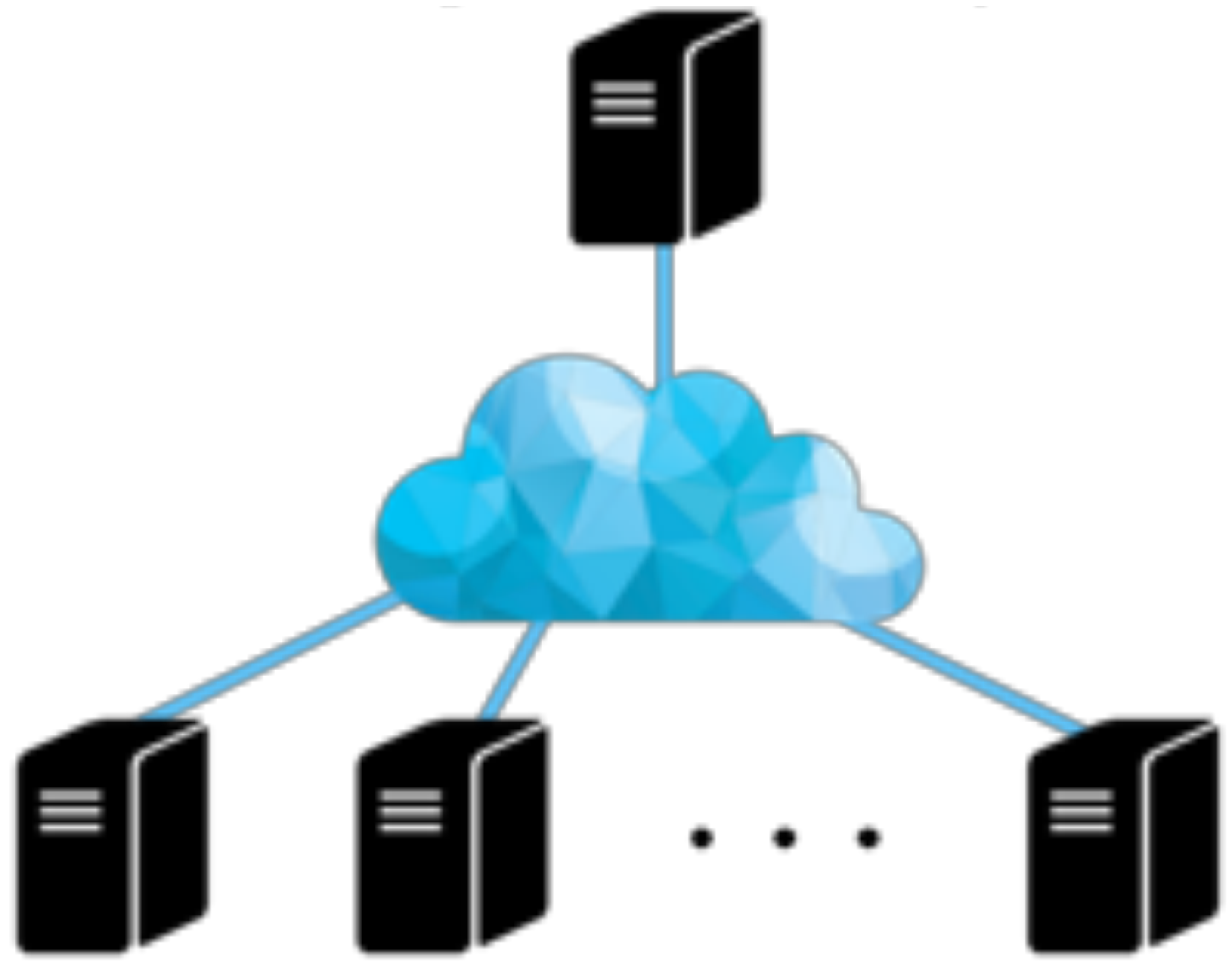
# Distributing gradient computations

– Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates

Parameter node



Worker 1 Worker 2 Worker P

Each contains all data

$i)$ Relates to coordinate descent algorithms

$ii)$ Could be part of a large−scale implementation, where part of the model is too large to be computed in a centralized fashion

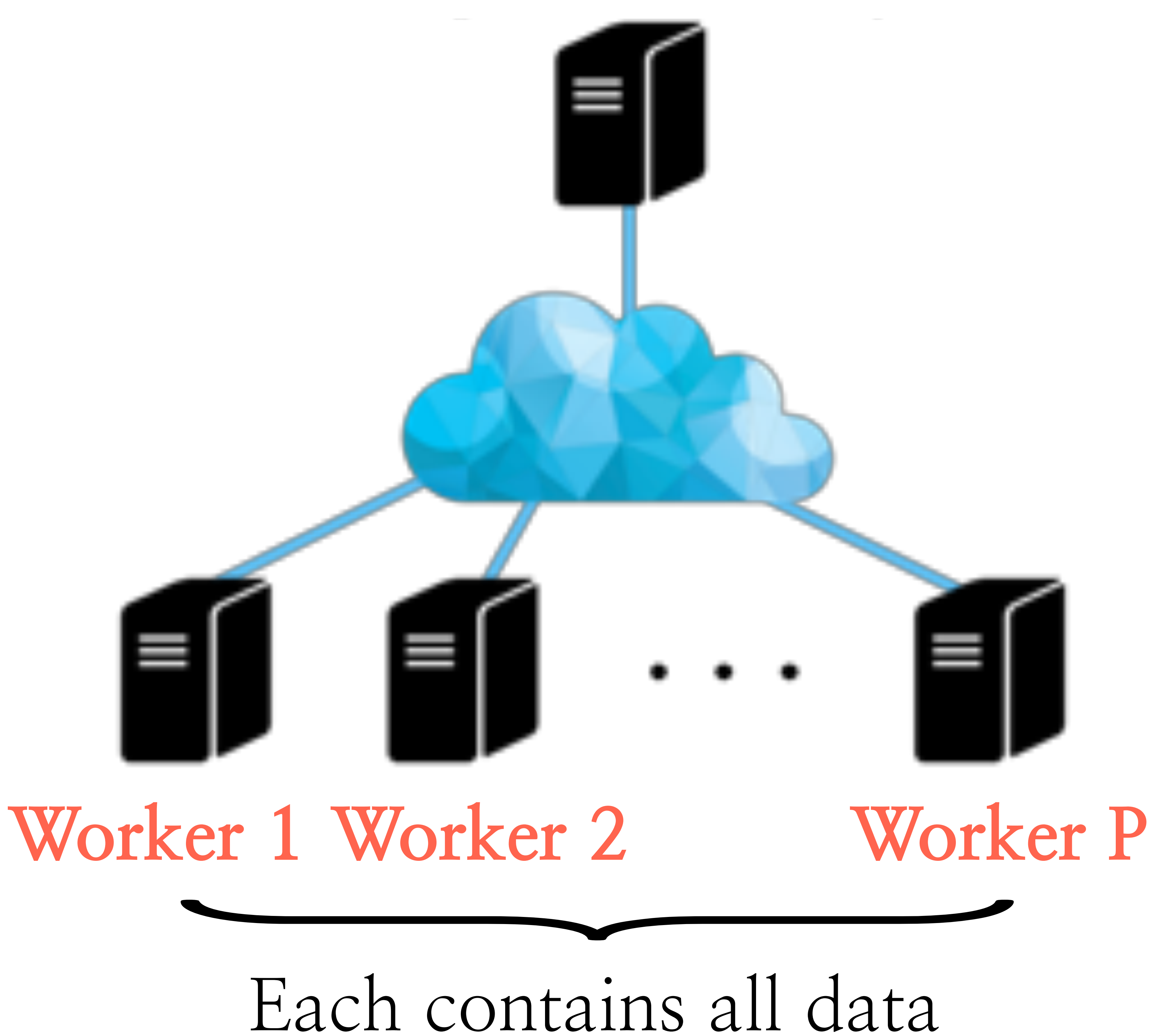
# Distributing gradient computations

- Consider the case where even $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node

$$x_{t+1} = x_t - \eta \nabla f_i(x_t)$$

Parallelism in coordinates

Parameter node




*i*) Relates to coordinate descent algorithms

*ii*) Could be part of a large-scale implementation, where part of the model is too large to be computed in a centralized fashion

*iii*) Could be an overkill to only compute updates for a subset of entries

# Distributing gradient computations

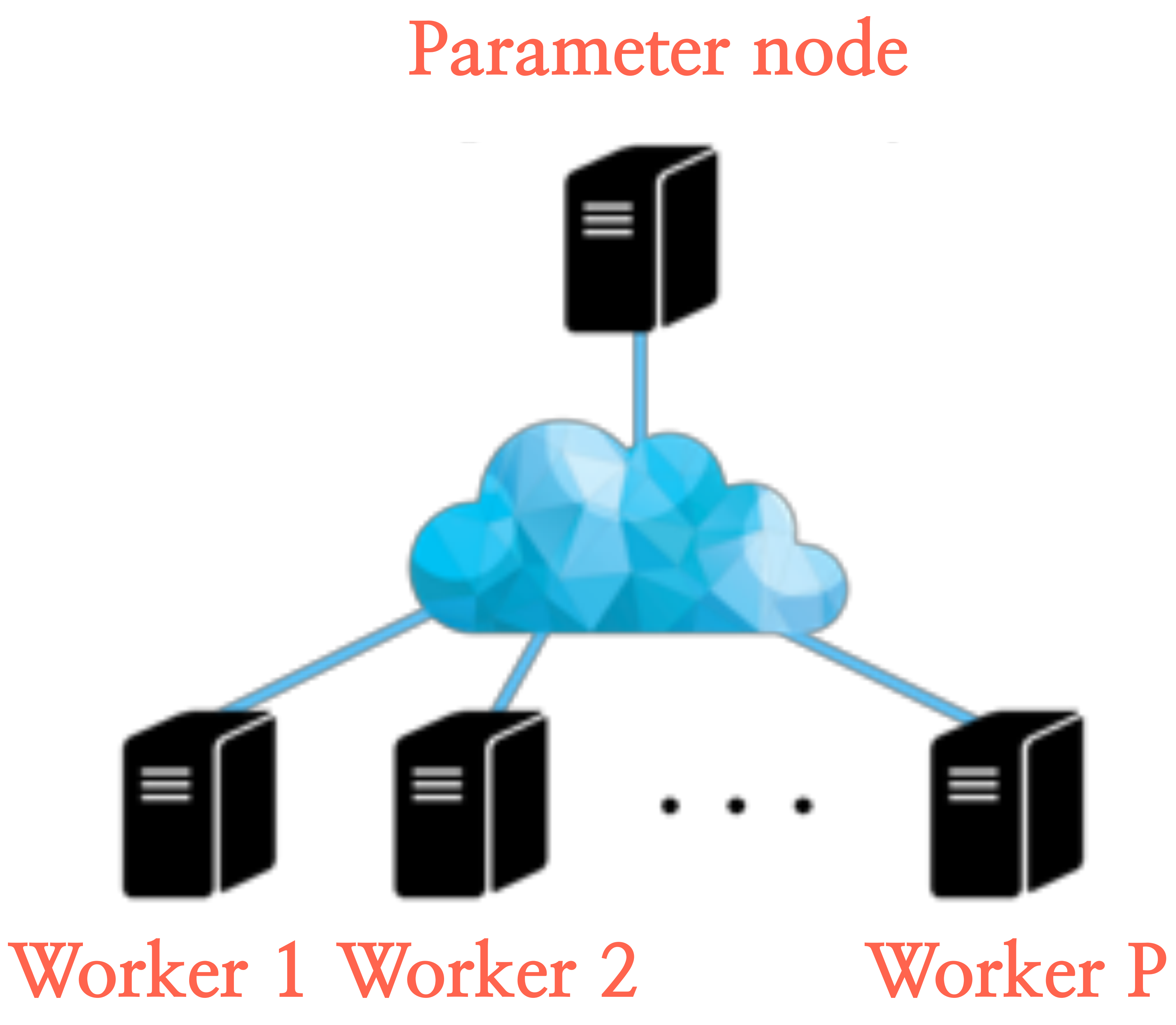
- What about the setting in–between? **Mini–batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

# Distributing gradient computations

- What about the setting in-between? **Mini-batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

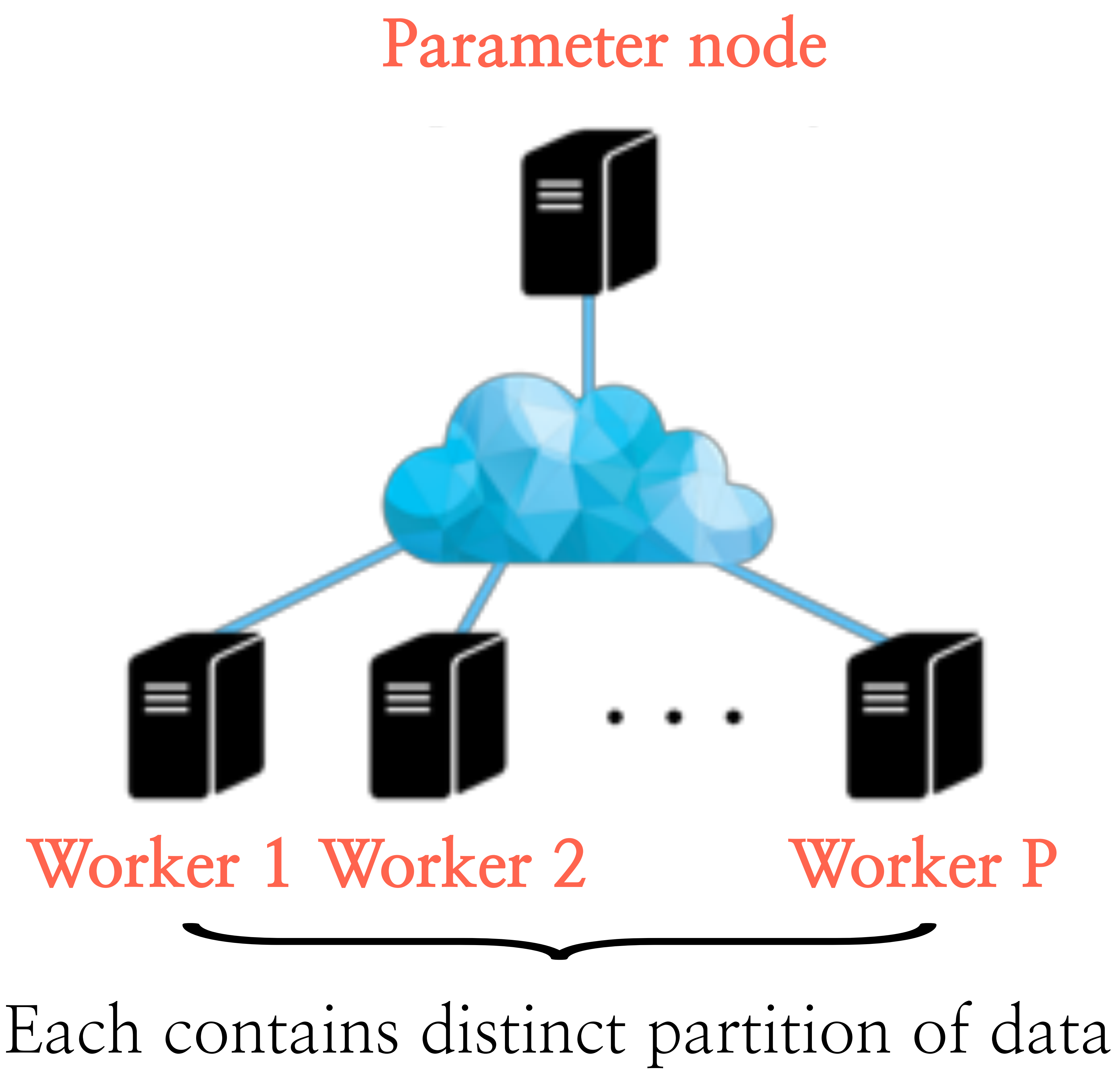



Each contains distinct partition of data

# Distributing gradient computations

– What about the setting in–between? **Mini–batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

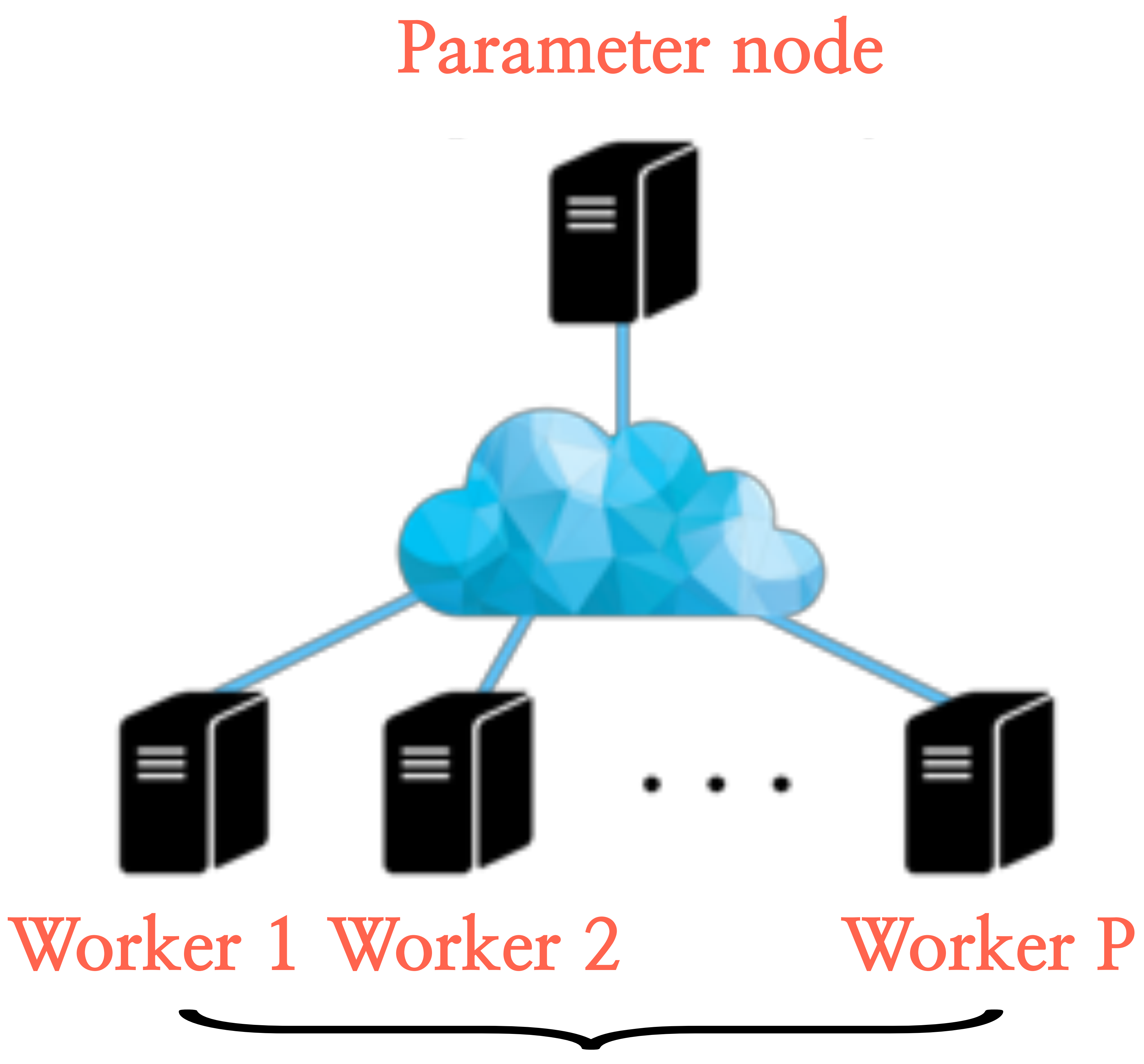



$i)$ Parameter node keeps and distributes model $x_t$ at every cycle/iteration

# Distributing gradient computations

– What about the setting in−between? **Mini−batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$




Each contains distinct partition of data

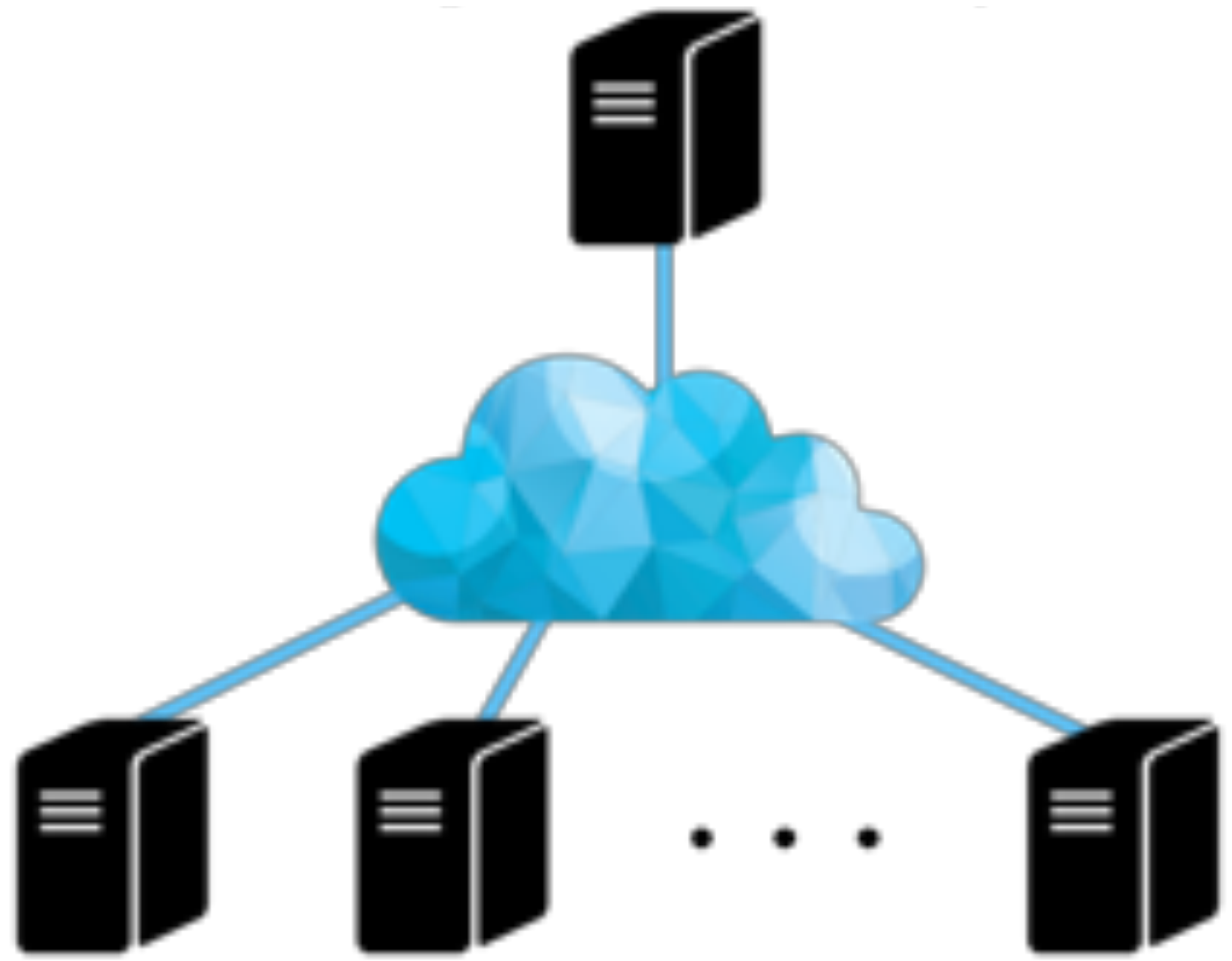
$i)$ Parameter node keeps and distributes model $x_t$ at every cycle/iteration

$ii)$ Worker nodes compute part of **mini−batch** gradient

# Distributing gradient computations

– What about the setting in−between? **Mini−batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$




Each contains distinct partition of data

*i*) Parameter node keeps and distributes model $x_t$ at every cycle/iteration

*ii*) Worker nodes compute part of **mini−batch** gradient

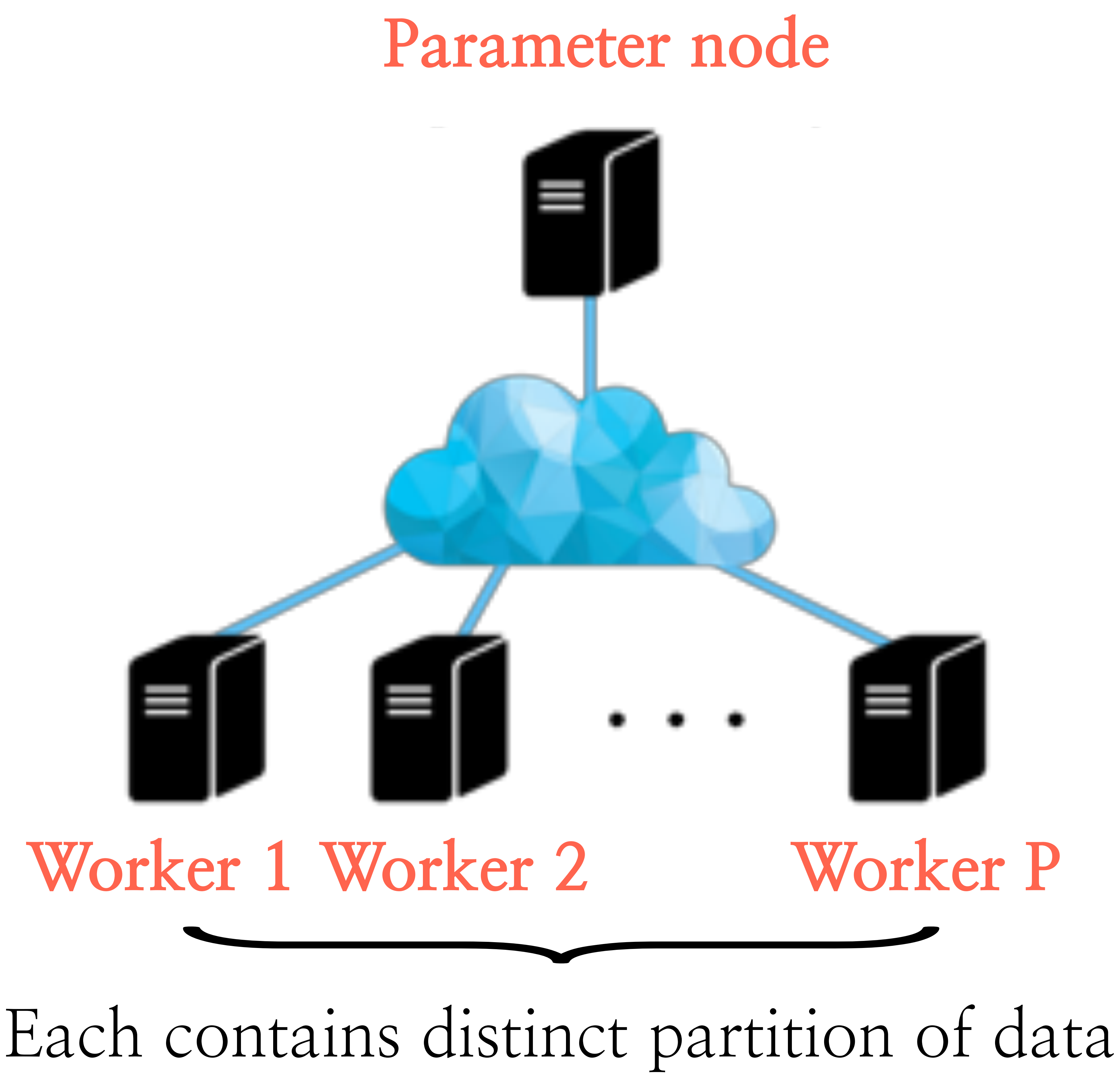
*iii*) Parameter node waits for **all gradient parts** to be collected to do the mini−batch step

(..till the very last slow worker)

# Distributing gradient computations

– What about the setting in–between? **Mini–batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$



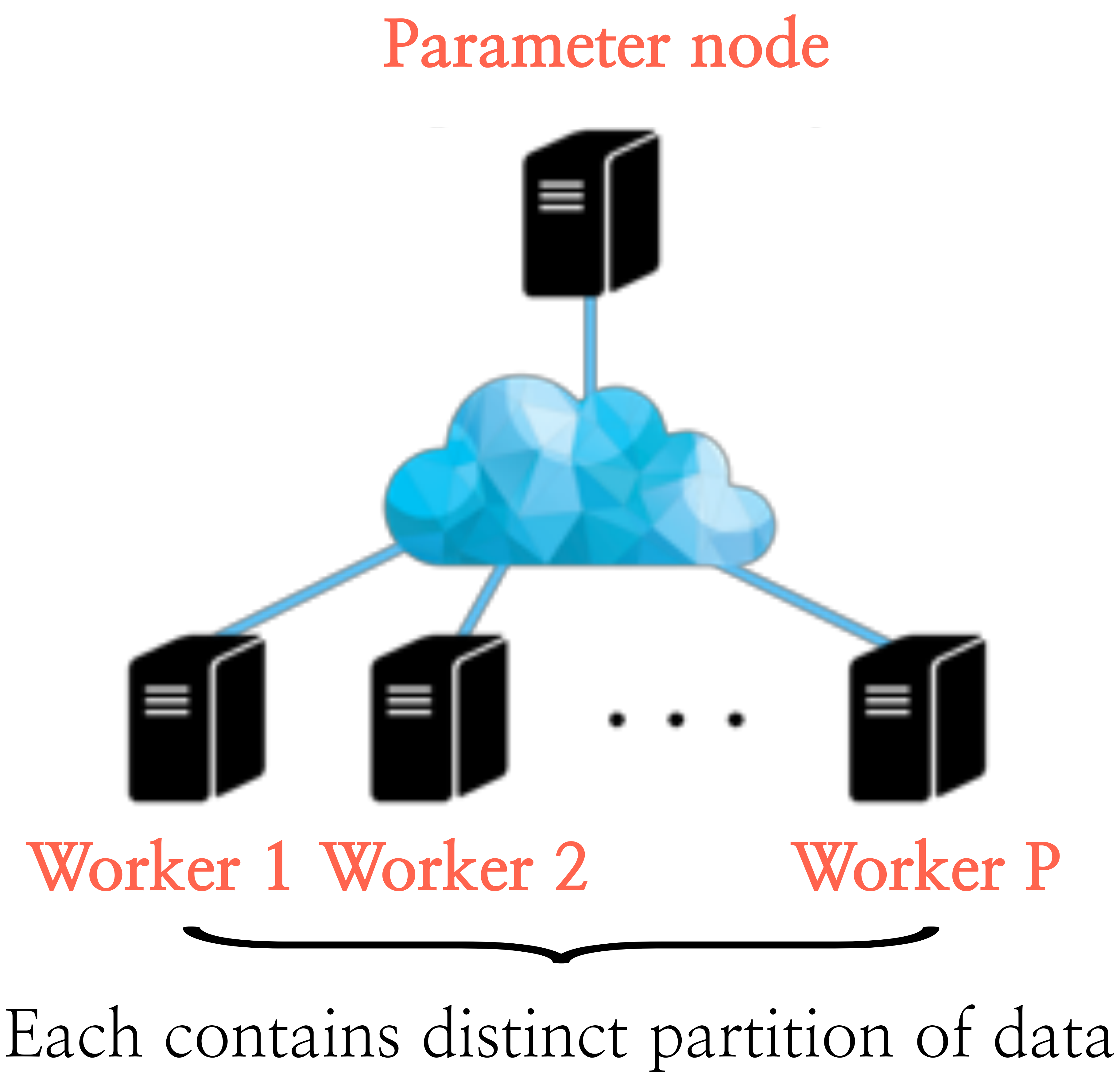

$i)$ Still requires synchronization; each worker has less work to do

(Discussion about large batch training)

# Distributing gradient computations

- What about the setting in–between? **Mini–batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$




*i*) Still requires synchronization; each worker has less work to do

*ii*) Introduces a tradeoff between statistical efficiency, computations efficiency (in terms of convergence) and communication efficiency

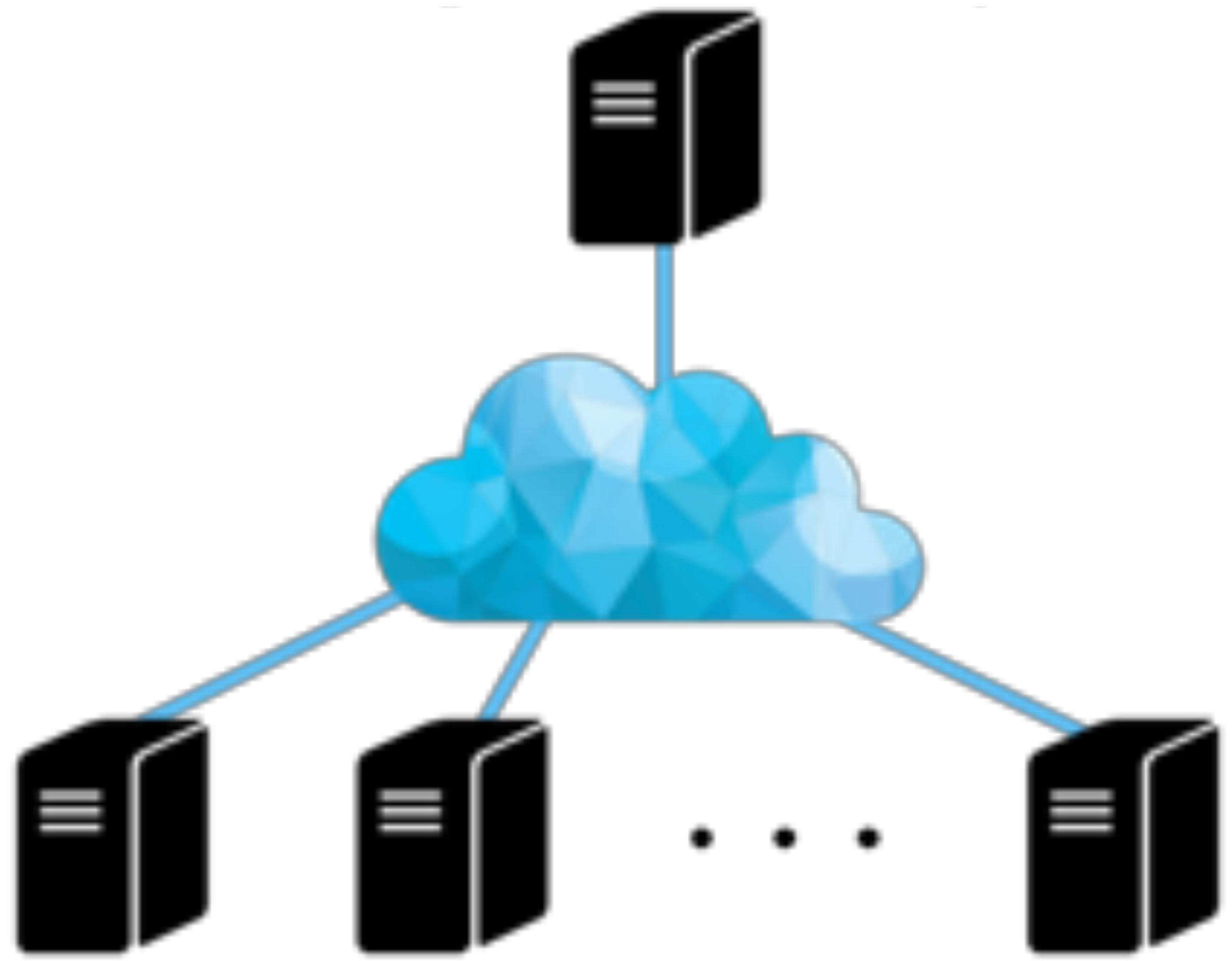
(Discussion about large batch training)

# Distributing gradient computations

- What about the setting in−between? **Mini−batch SGD**

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

Parameter node



Worker 1 Worker 2 Worker P

Each contains distinct partition of data

*i*) Still requires synchronization; each worker has less work to do

*ii*) Introduces a tradeoff between statistical efficiency, computations efficiency (in terms of convergence) and communication efficiency

*iii*) Usually computing $\nabla f_{i_t}(x_t)$ is cheap per node

(Discussion about large batch training)

# Distributing gradient computations

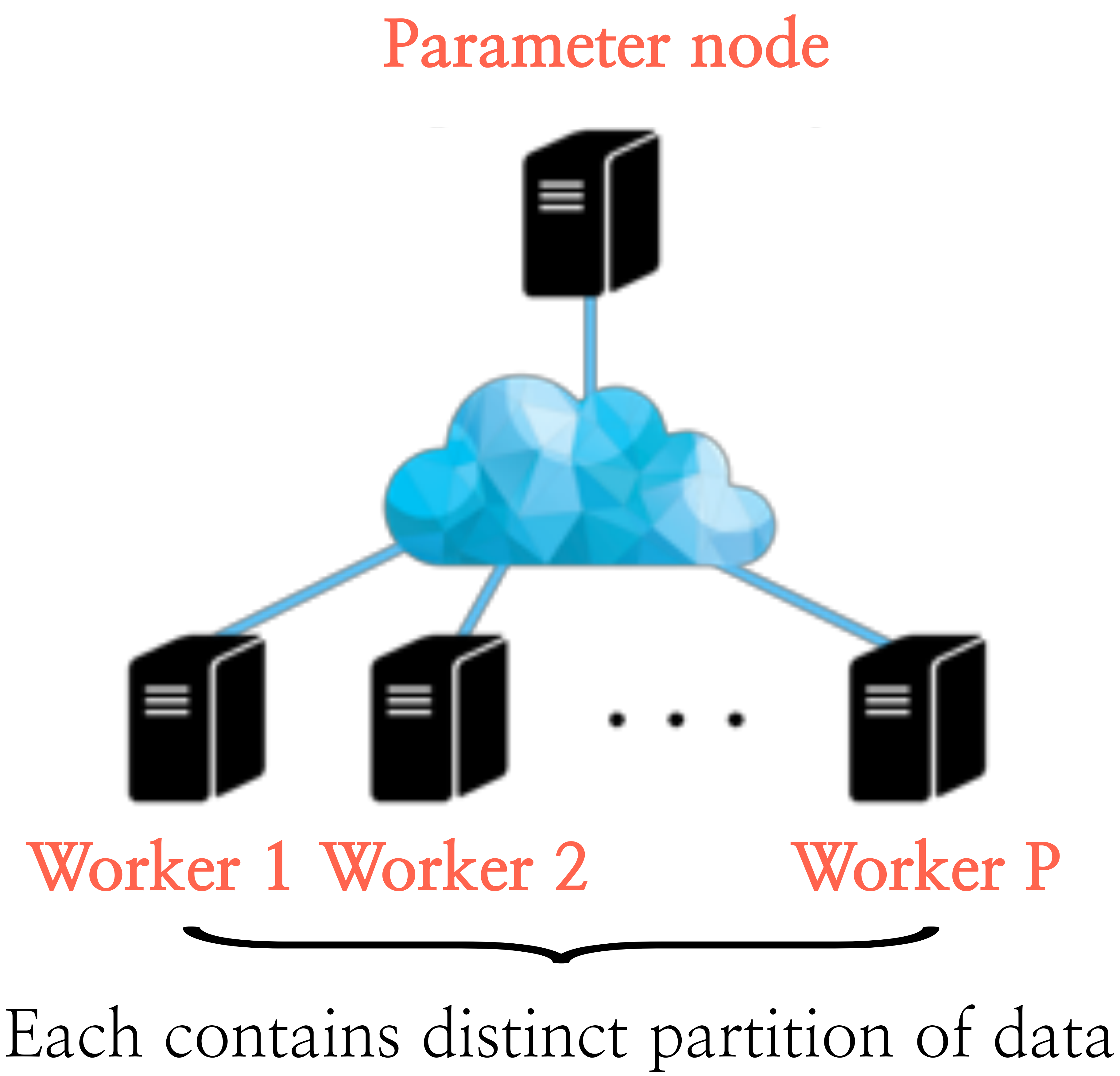
- What if we run **mini-batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \qquad \text{(on each worker node)}$$

# Distributing gradient computations

– What if we run **mini–batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \qquad \text{(on each worker node)}$$



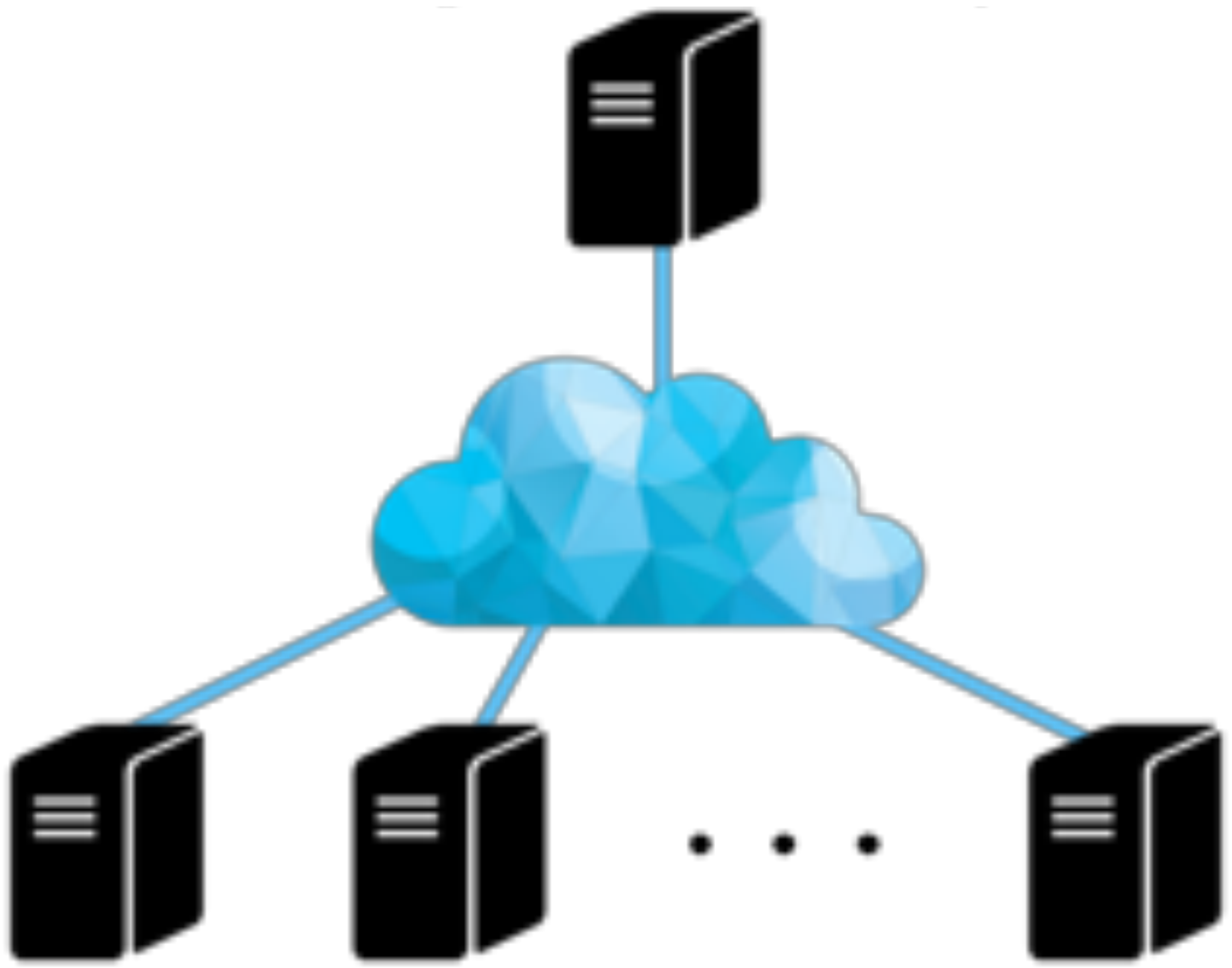
# Distributing gradient computations

– What if we run **mini-batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

(on each worker node)

Parameter node



Worker 1 Worker 2 Worker P

Each contains distinct partition of data

$i)$ Parameter node does.. nothing until the end

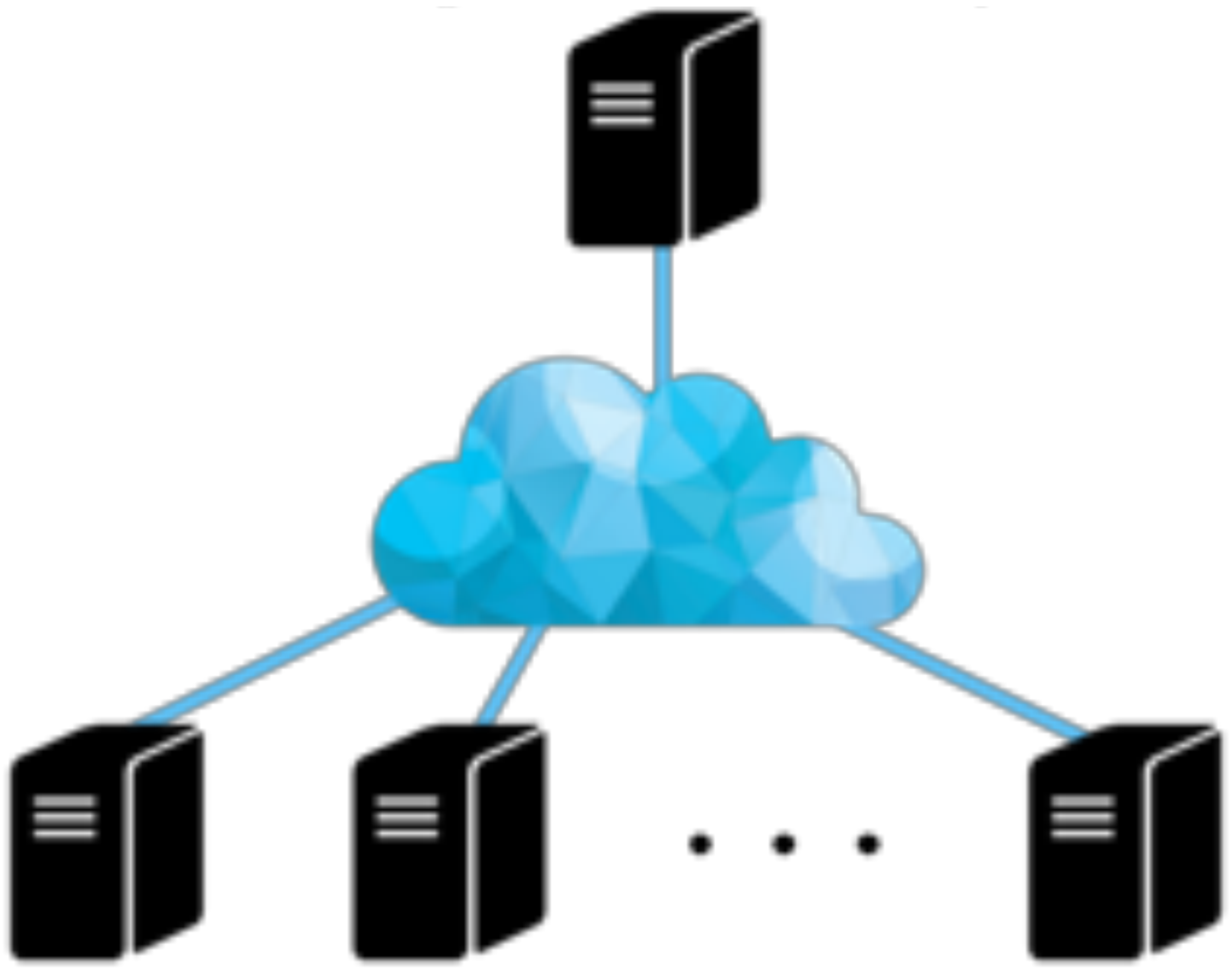
# Distributing gradient computations

– What if we run **mini–batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

(on each worker node)

Parameter node



Worker 1 Worker 2 Worker P

$\underbrace{\hspace{10em}}$

Each contains distinct partition of data

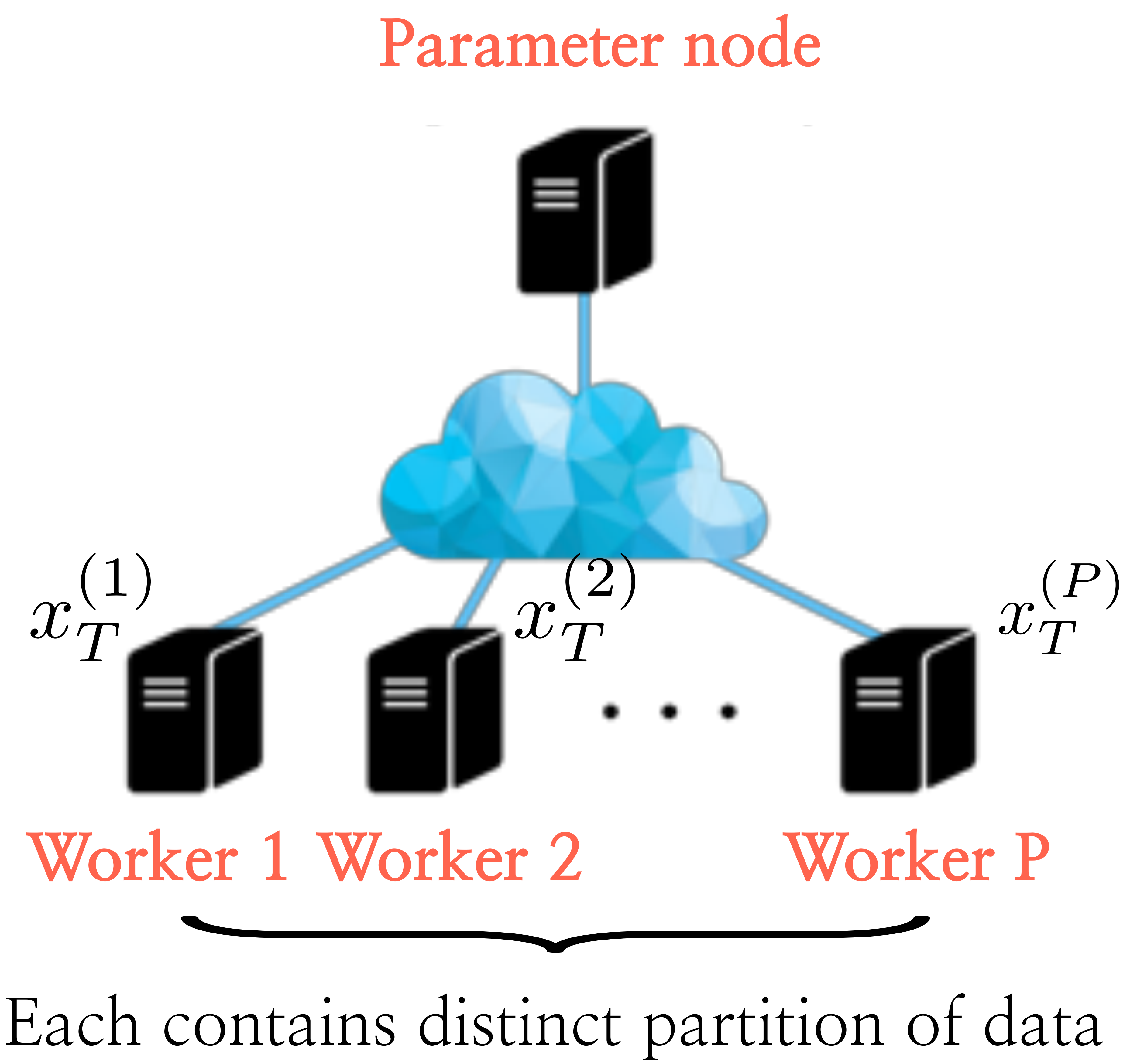
*i*) Parameter node does.. nothing until the end

*ii*) Worker nodes do **mini–batch SGD** as if there is no distributed computation

# Distributing gradient computations

- What if we run **mini–batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$ (on each worker node)



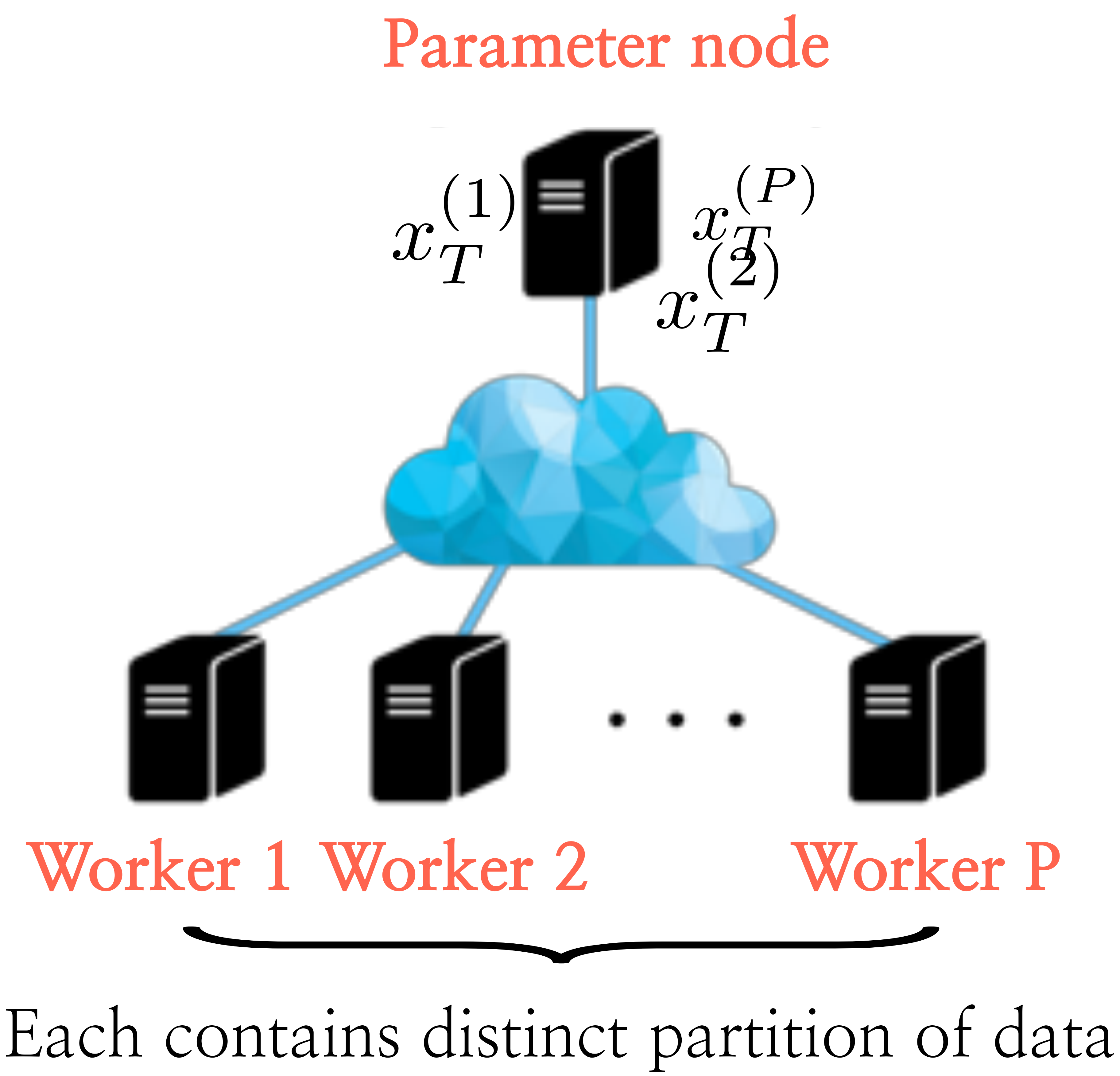

*i)* Parameter node does.. nothing until the end

*ii)* Worker nodes do **mini–batch SGD** as if there is no distributed computation

# Distributing gradient computations

– What if we run **mini–batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \qquad \text{(on each worker node)}$$




$i)$ Parameter node does.. nothing until the end

$ii)$ Worker nodes do **mini–batch SGD** as if there is no distributed computation

$iii)$ Parameter node waits for **all the models** to be collected, and be averaged

(..till the very last slow worker)

# Distributing gradient computations

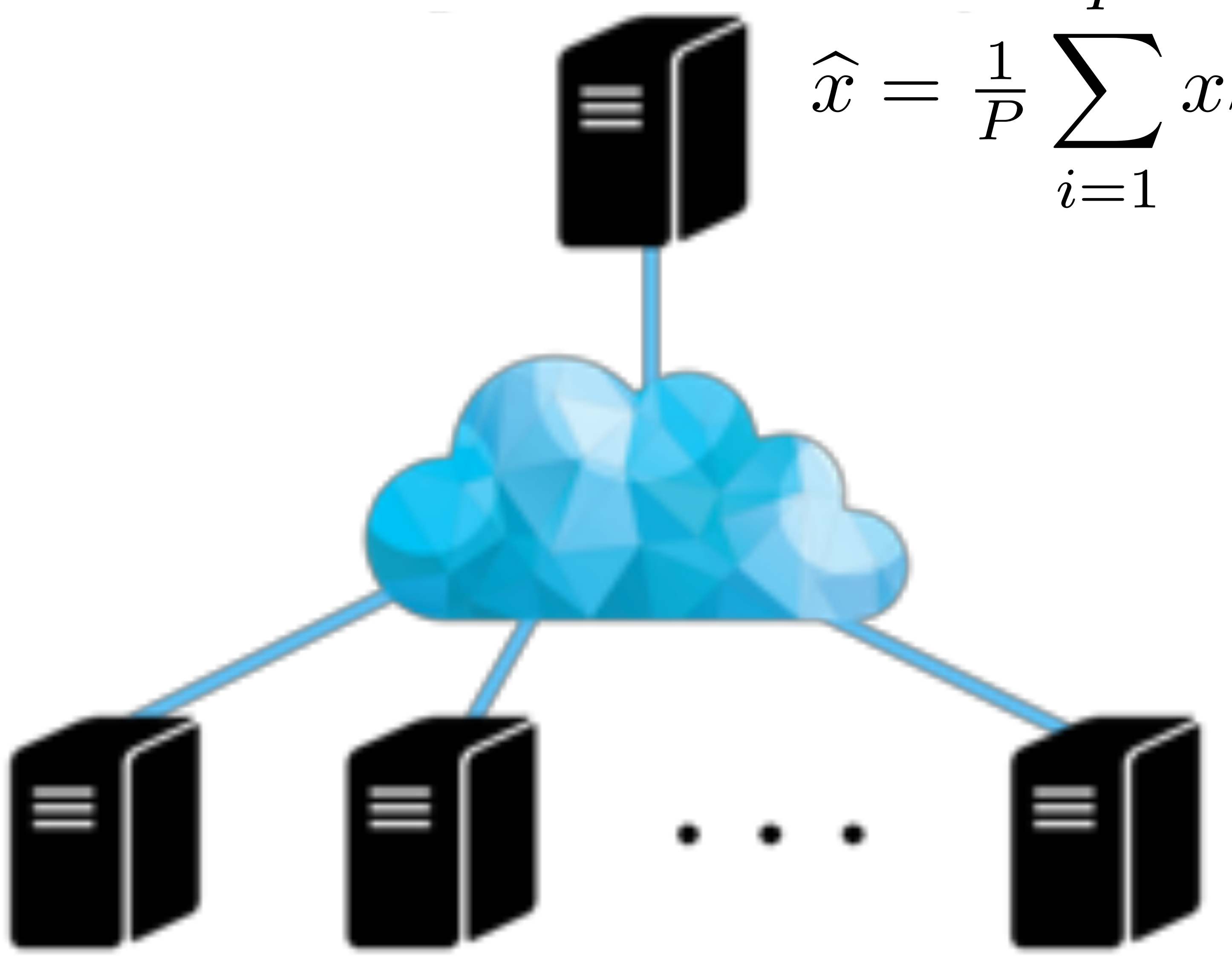
– What if we run **mini–batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

(on each worker node)

Parameter node

$$\widehat{x} = \frac{1}{P} \sum_{i=1}^{P} x_T^{(i)}$$



Worker 1 Worker 2 $\cdots$ Worker P

Each contains distinct partition of data

$i)$ Parameter node does.. nothing until the end

$ii)$ Worker nodes do **mini–batch SGD** as if there is no distributed computation

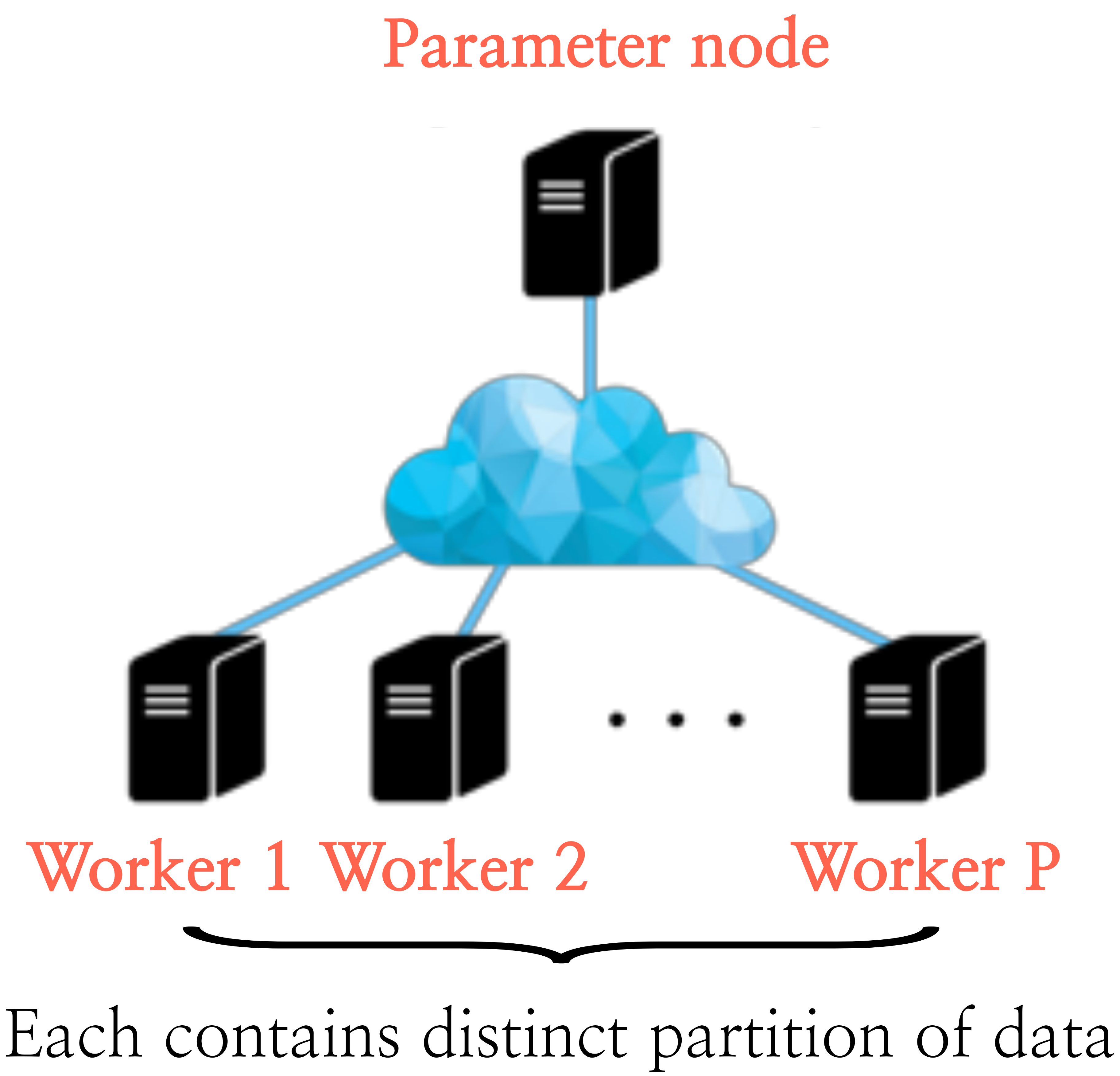
$iii)$ Parameter node waits for **all the models** to be collected, and be averaged

(..till the very last slow worker)

# Distributing gradient computations

– What if we run **mini–batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$



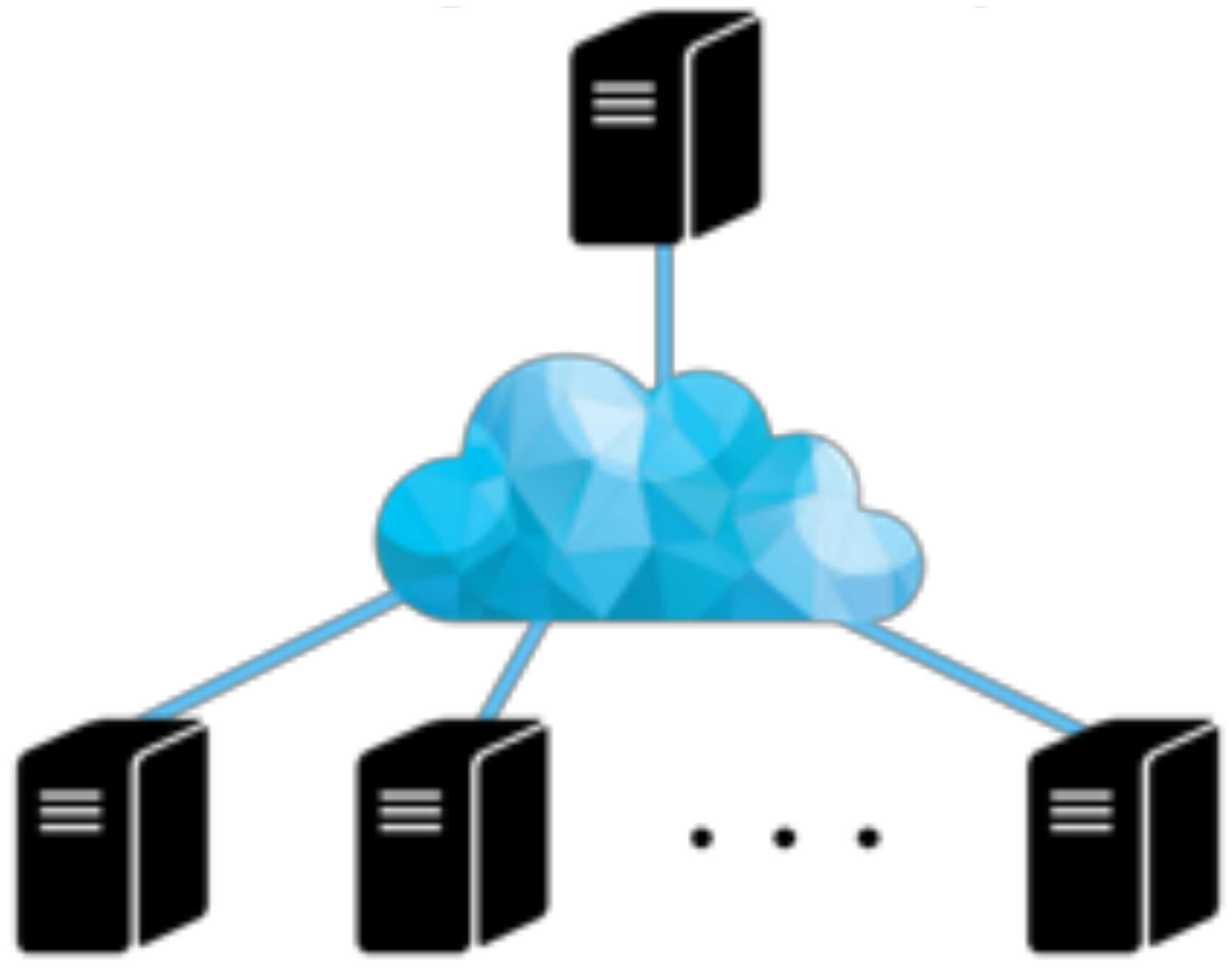

$i)$ Minimal communication: every node works on its own, and sends the model at the end of its execution

# Distributing gradient computations

– What if we run **mini–batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

Parameter node



Worker 1 Worker 2 Worker P

$\underbrace{\hspace{6cm}}$

Each contains distinct partition of data

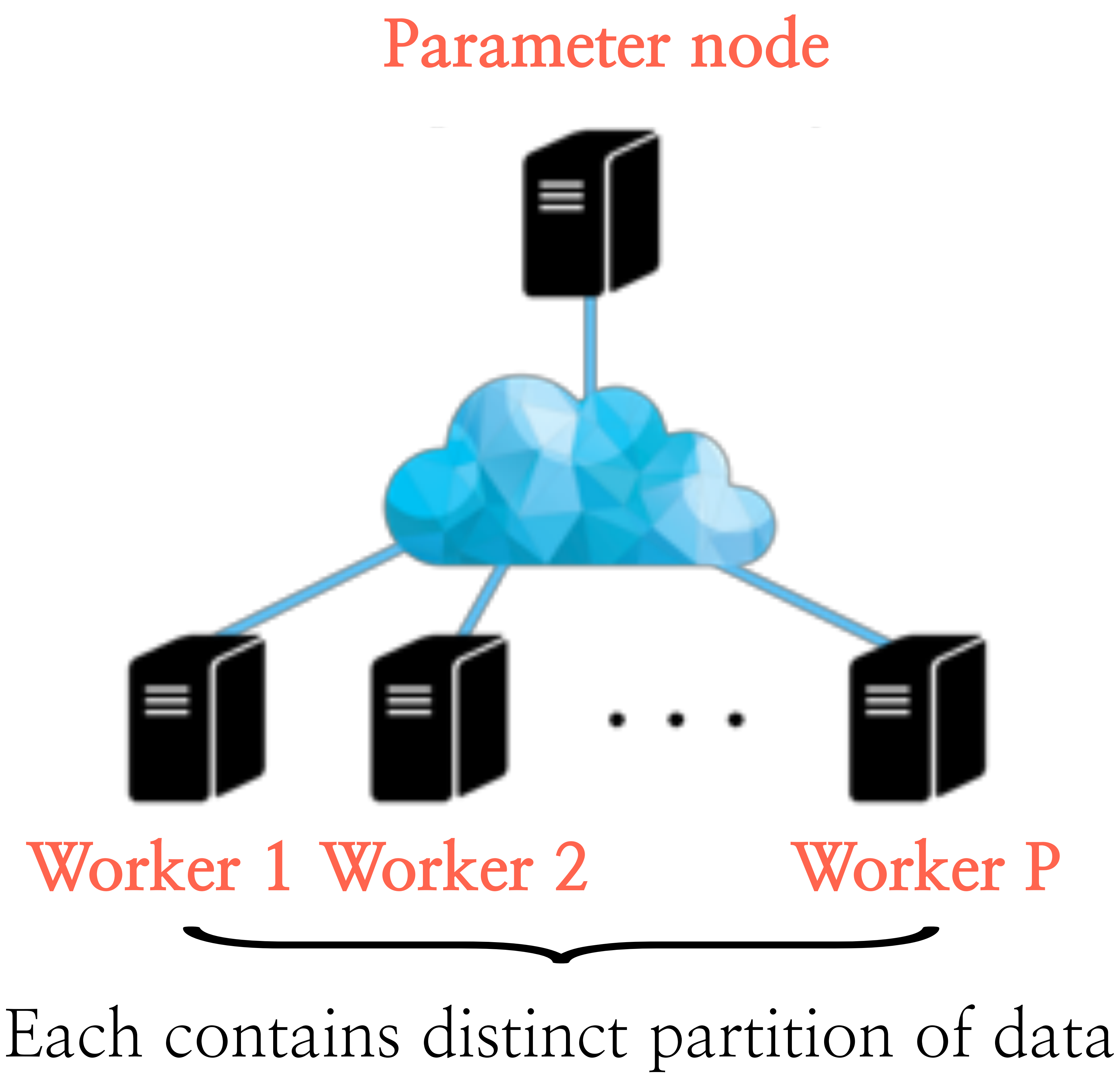
$i)$ Minimal communication: every node works on its own, and sends the model at the end of its execution

$ii)$ The model was designed for convex problems – the idea is that each subproblem has a solution close to the global one – thus averaging does not hurt

# Distributing gradient computations

– What if we run **mini-batch SGD** in parallel and combine at the end:

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$




$i)$ Minimal communication: every node works on its own, and sends the model at the end of its execution

$ii)$ The model was designed for convex problems – the idea is that each subproblem has a solution close to the global one – thus averaging does not hurt

$iii)$ Final decision is prediction averaging – similar ideas hold for random forests

# Using distributed computing in a different way

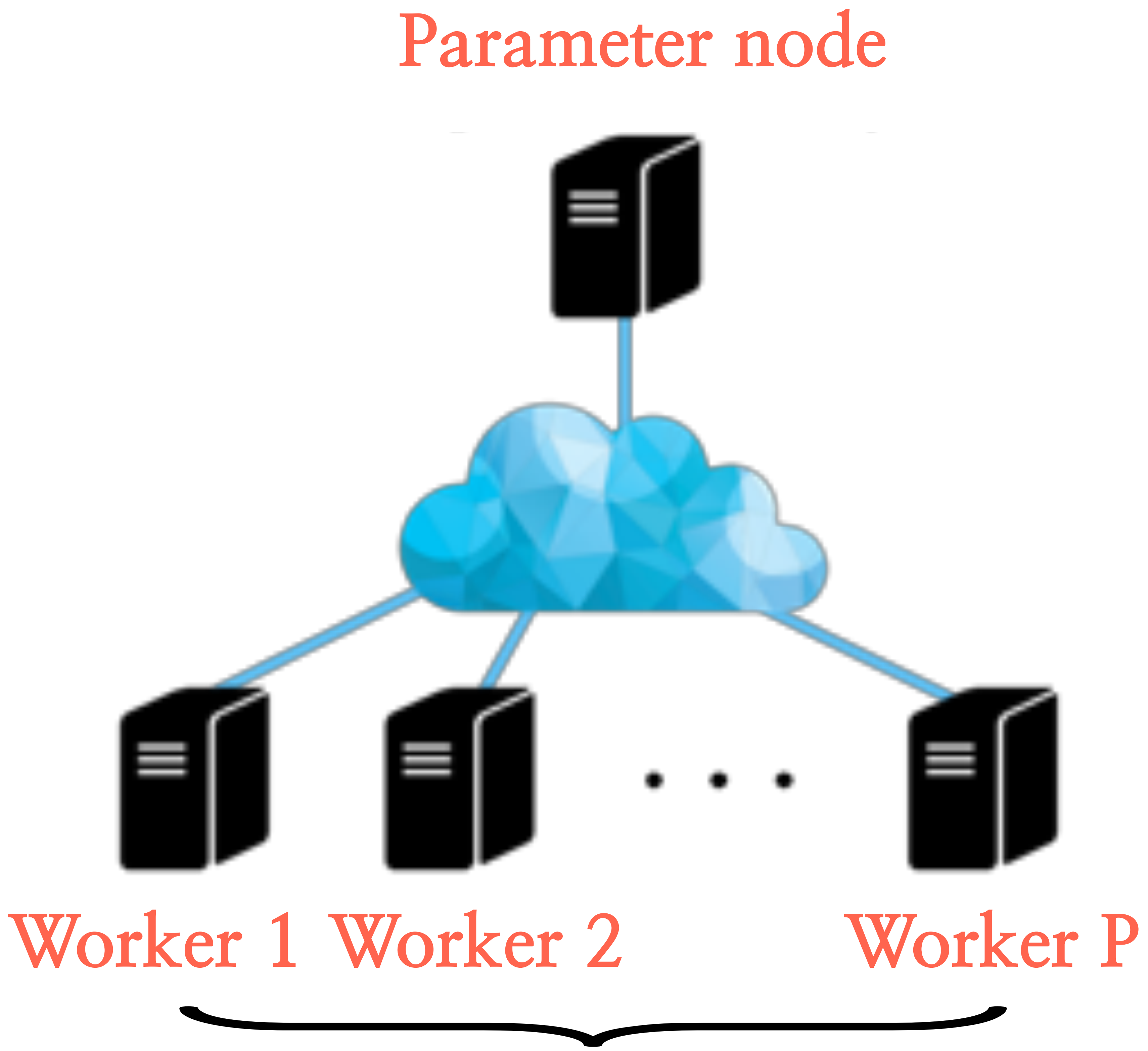
- Run code in parallel as a way for hyperparameter optimization

$$x_{t+1} = x_t - \eta_1 \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \qquad \bullet \quad \bullet \quad \bullet \qquad x_{t+1} = x_t - \eta_q \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

# Using distributed computing in a different way

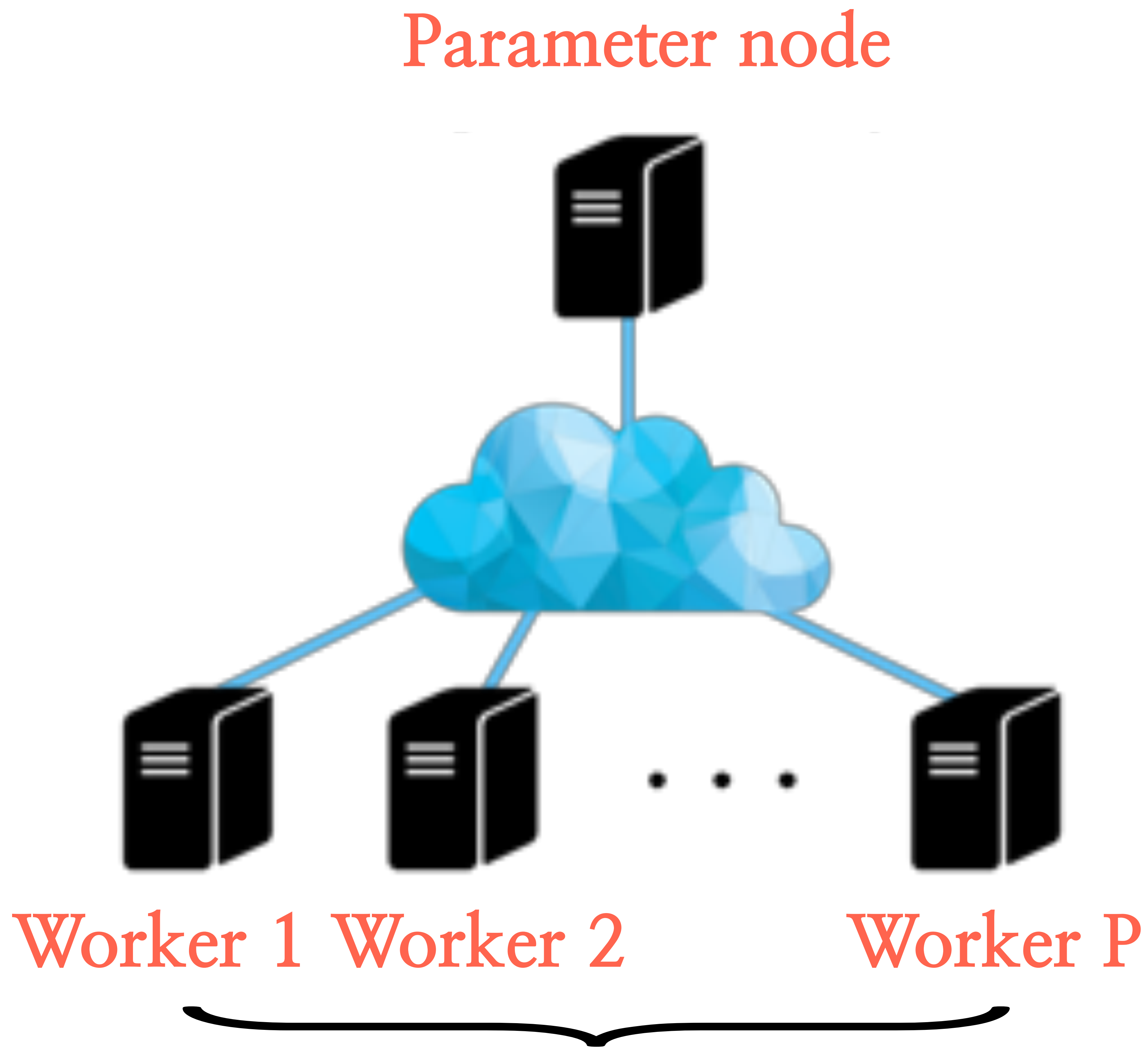
– Run code in parallel as a way for hyperparameter optimization

$$x_{t+1} = x_t - \eta_1 \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \quad \cdots \quad x_{t+1} = x_t - \eta_q \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$




Each contains distinct partition of data

• • •



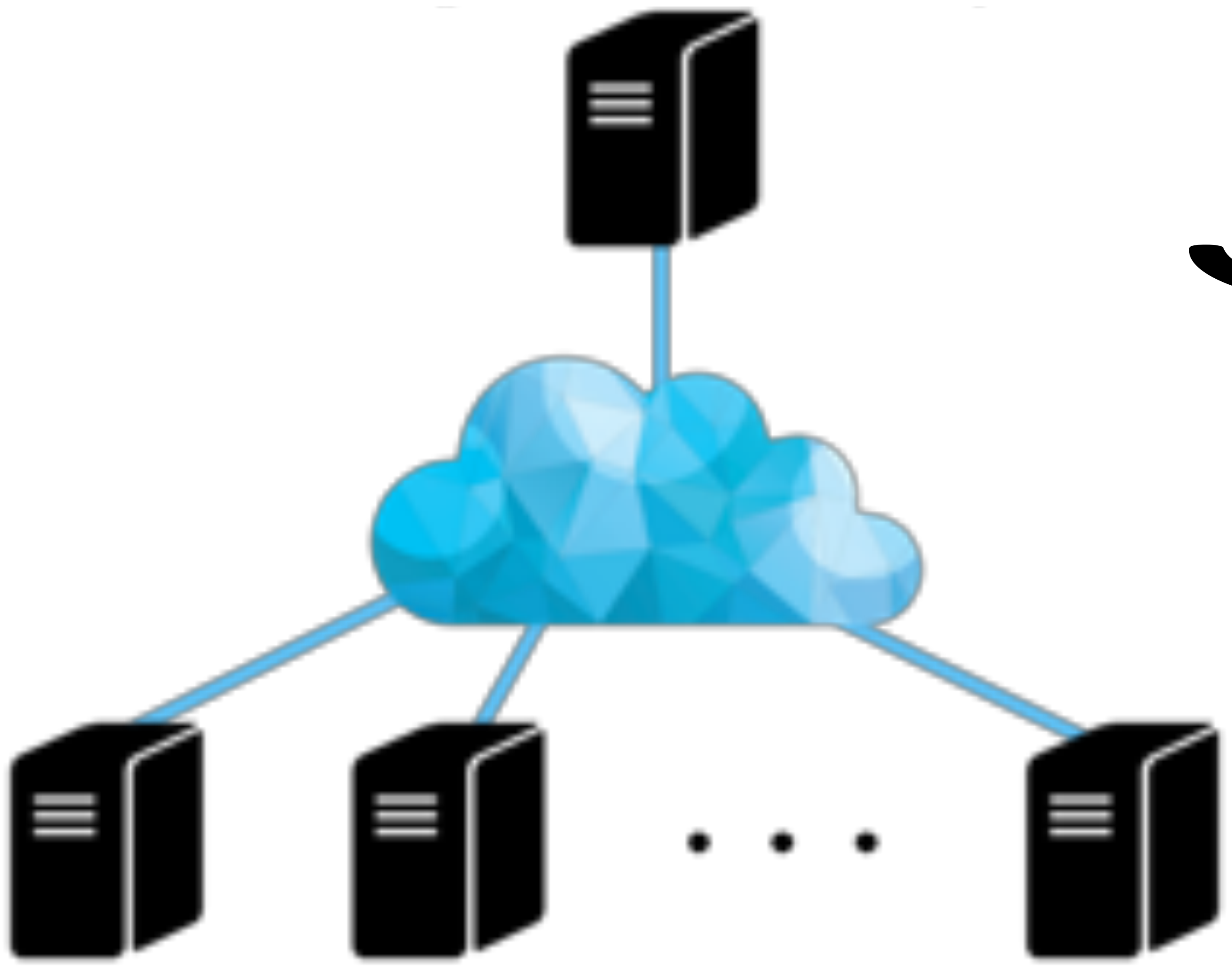

Each contains distinct partition of data

# Using distributed computing in a different way

- Run code in parallel as a way for hyperparameter optimization

$$x_{t+1} = x_t - \eta_1 \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \quad \cdots \quad x_{t+1} = x_t - \eta_q \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

Parameter node



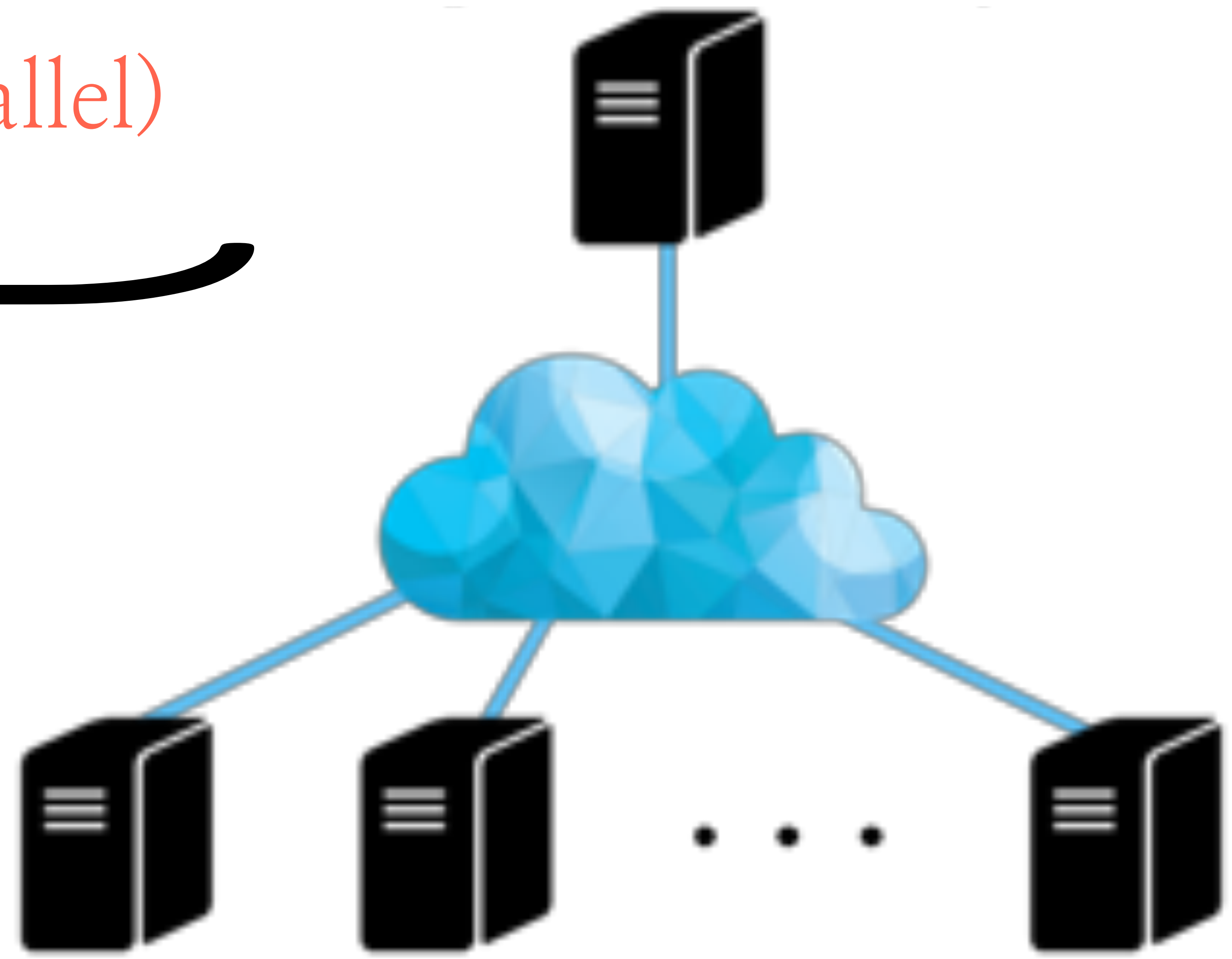
Worker 1 Worker 2 Worker P

Each contains distinct partition of data

$\underbrace{\hphantom{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}_{}$ (q different settings, ran in parallel)

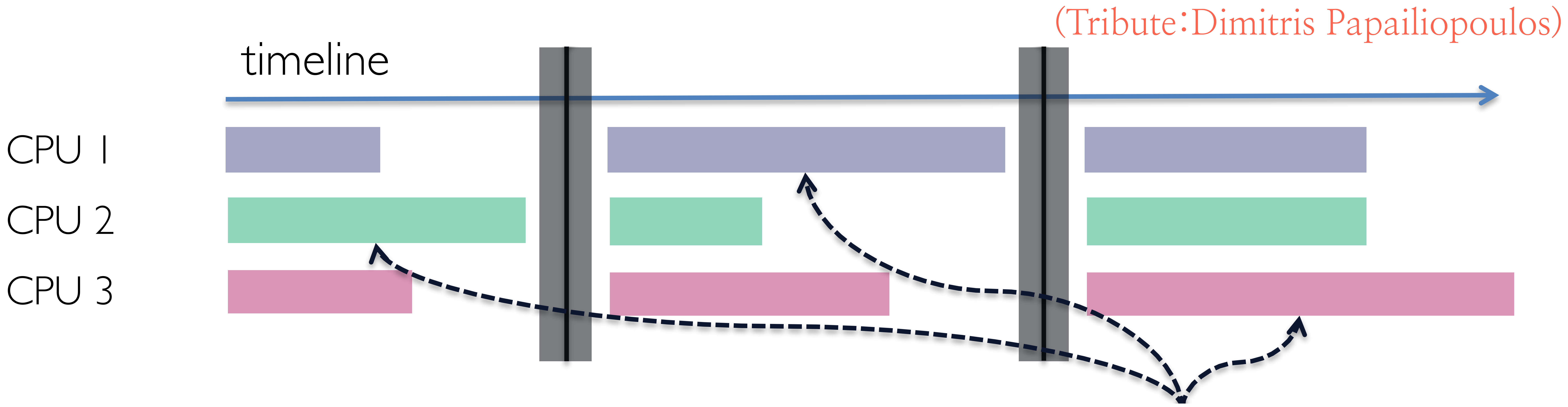
$\cdots$

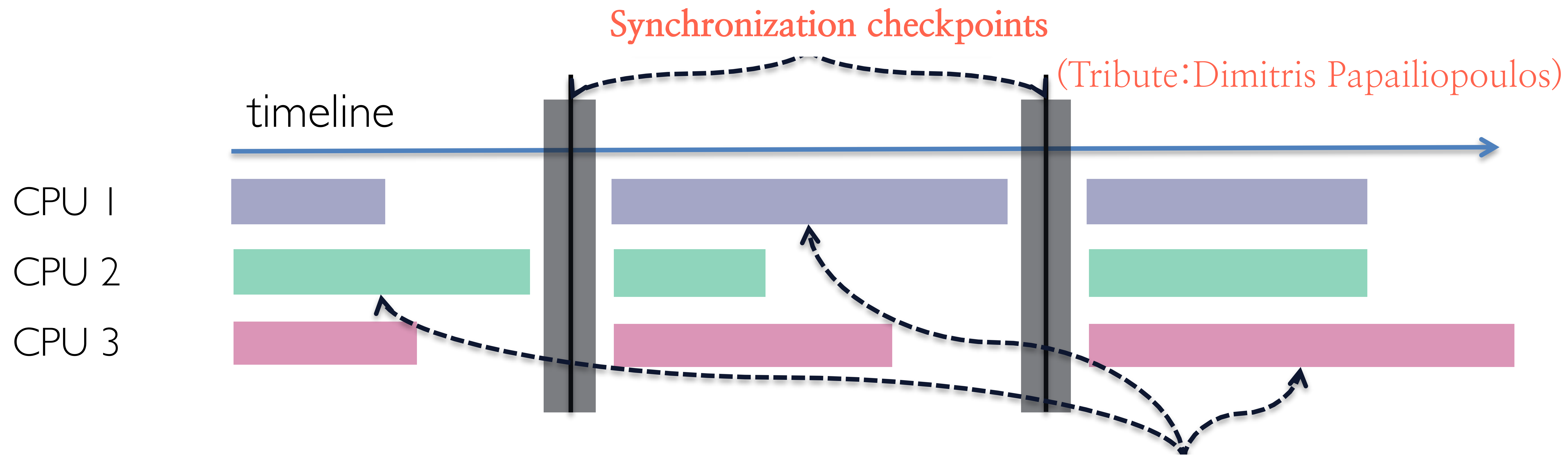
Parameter node



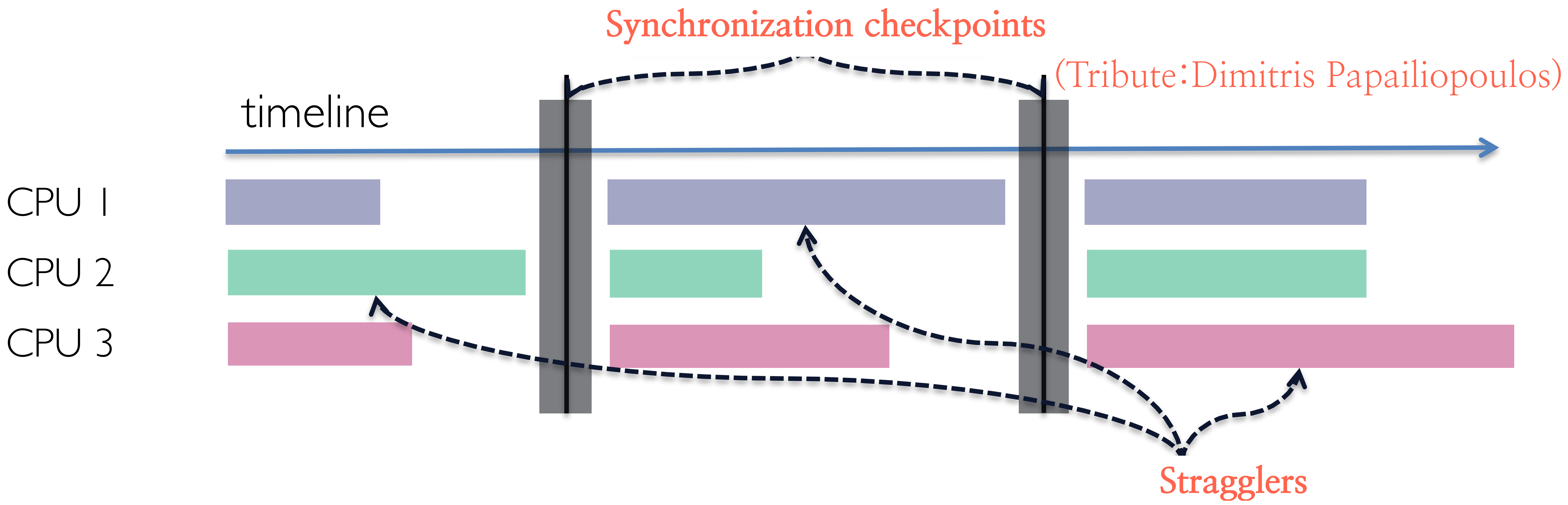
Worker 1 Worker 2 Worker P

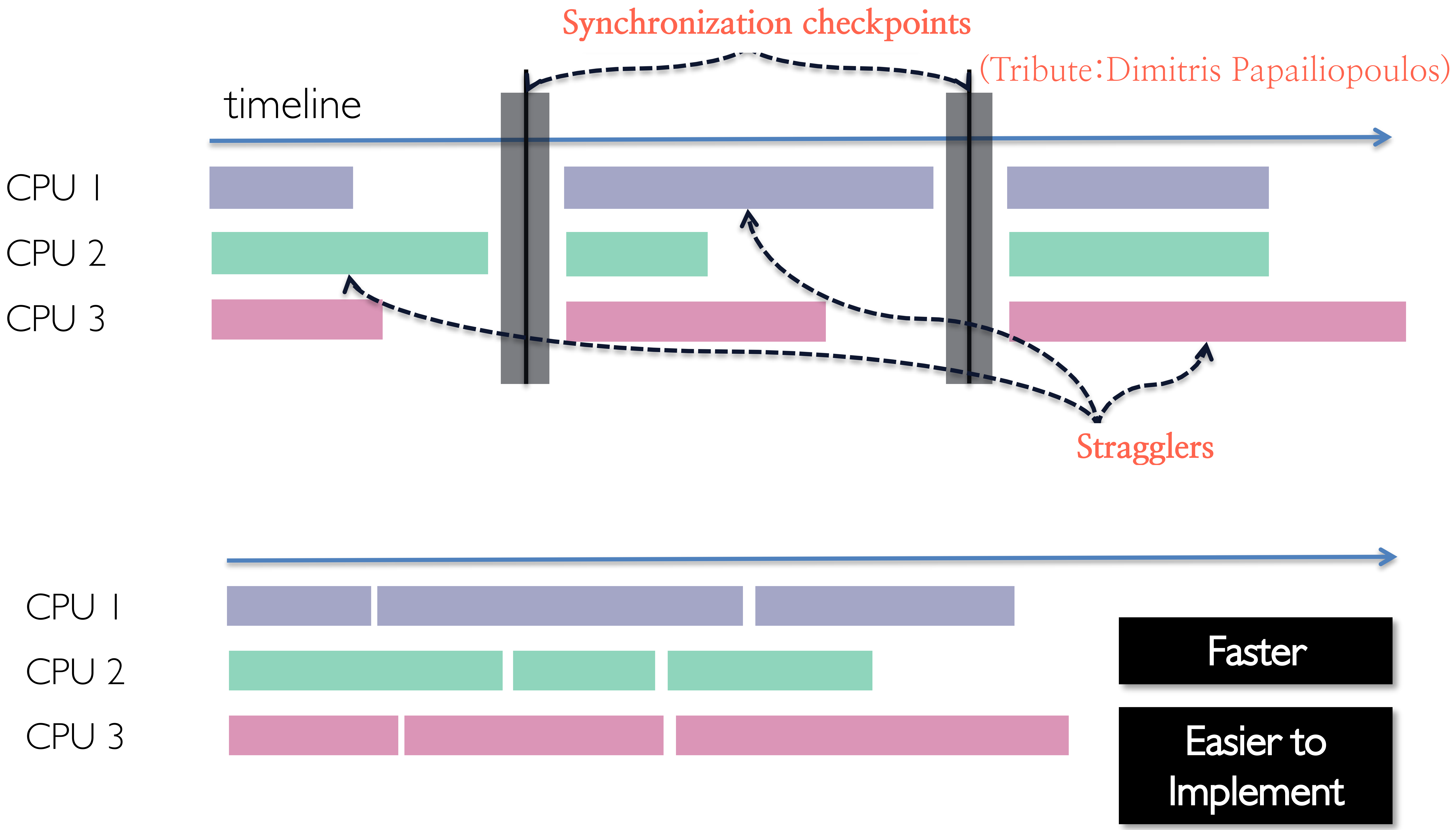
Each contains distinct partition of data


(Tribute:Dimitris Papailiopoulos)
timeline
CPU 1
CPU 2
CPU 3



Synchronization checkpoints
(Tribute:Dimitris Papailiopoulos)
timeline
CPU 1
CPU 2
CPU 3


Synchronization checkpoints
(Tribute：Dimitris Papailiopoulos)
timeline
CPU 1
CPU 2
CPU 3
Stragglers



Synchronization checkpoints
(Tribute:Dimitris Papailiopoulos)
timeline
CPU 1
CPU 2
CPU 3
Stragglers
CPU 1
CPU 2
CPU 3
Faster
Easier to Implement

# Limitations of such distributed computing

# Limitations of such distributed computing

- **Synchronization**：
  - Must wait for the slowest worker to synchronize all workers, and keep all nodes aware of each other's updates to the model.

# Limitations of such distributed computing

- **Synchronization**:
  - Must wait for the slowest worker to synchronize all workers, and keep all nodes aware of each other's updates to the model.

- **Synchronization is often quite expensive**:
  - Consider the following setting: we have P workers, and P−1 of them have already sent their updates to the parameter server. The whole system has to wait for the last worker to complete and send his part, in order to proceed.

# Limitations of such distributed computing

- **Synchronization**:
  - Must wait for the slowest worker to synchronize all workers, and keep all nodes aware of each other's updates to the model.
- **Synchronization is often quite expensive**:
  - Consider the following setting: we have P workers, and P−1 of them have already sent their updates to the parameter server. The whole system has to wait for the last worker to complete and send his part, in order to proceed.
- **Alternatives or we have to bear with this situation?**

# Asynchronous distributed computing

# Asynchronous distributed computing

- **Multicore systems can host large-scale problems**:
  - Instead of using clusters of processing nodes, one can use a single inexpensive work station that can host problems that, after preprocessing, involve a few terabytes of data

# Asynchronous distributed computing

- **Multicore systems can host large-scale problems:**
  - Instead of using clusters of processing nodes, one can use a single inexpensive work station that can host problems that, after preprocessing, involve a few terabytes of data

- **Advantages of multicore systems:**
  - Low latency + high throughput shared main memory
  - High bandwidth of multiple disks
  - Fast multithread processors

# Asynchronous distributed computing

- **Multicore systems can host large-scale problems**:
  - Instead of using clusters of processing nodes, one can use a single inexpensive work station that can host problems that, after preprocessing, involve a few terabytes of data

- **Advantages of multicore systems**:
  - Low latency + high throughput shared main memory
  - High bandwidth of multiple disks
  - Fast multithread processors

- Main **bottleneck**:
  - Synchronization (locking) amongst processors

# Asynchrony in SGD

– Run SGD in parallel without locks!

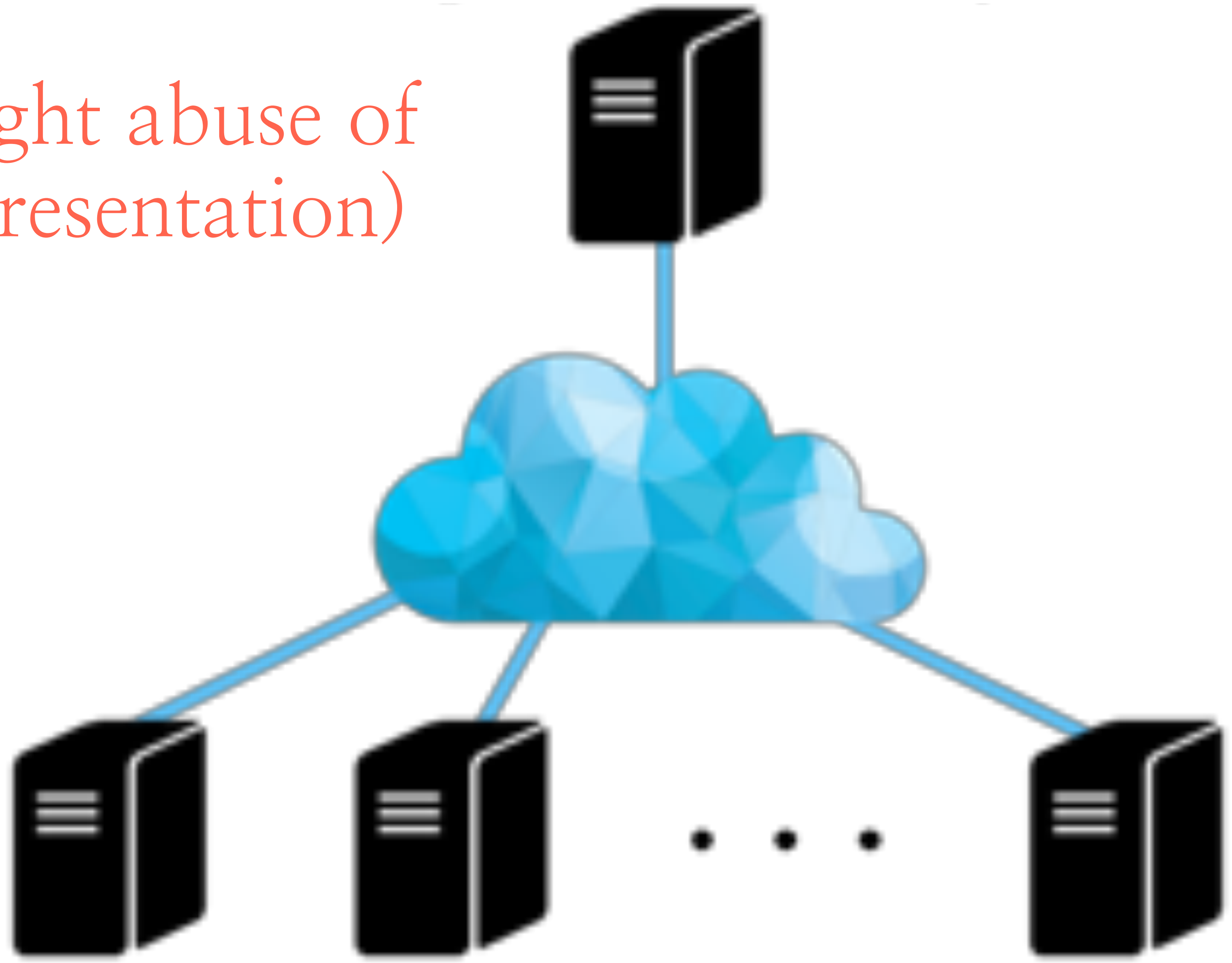
# Asynchrony in SGD

– **Run SGD in parallel without locks!**

– All threads have access to shared memory

Shared memory

(Slight abuse of representation)

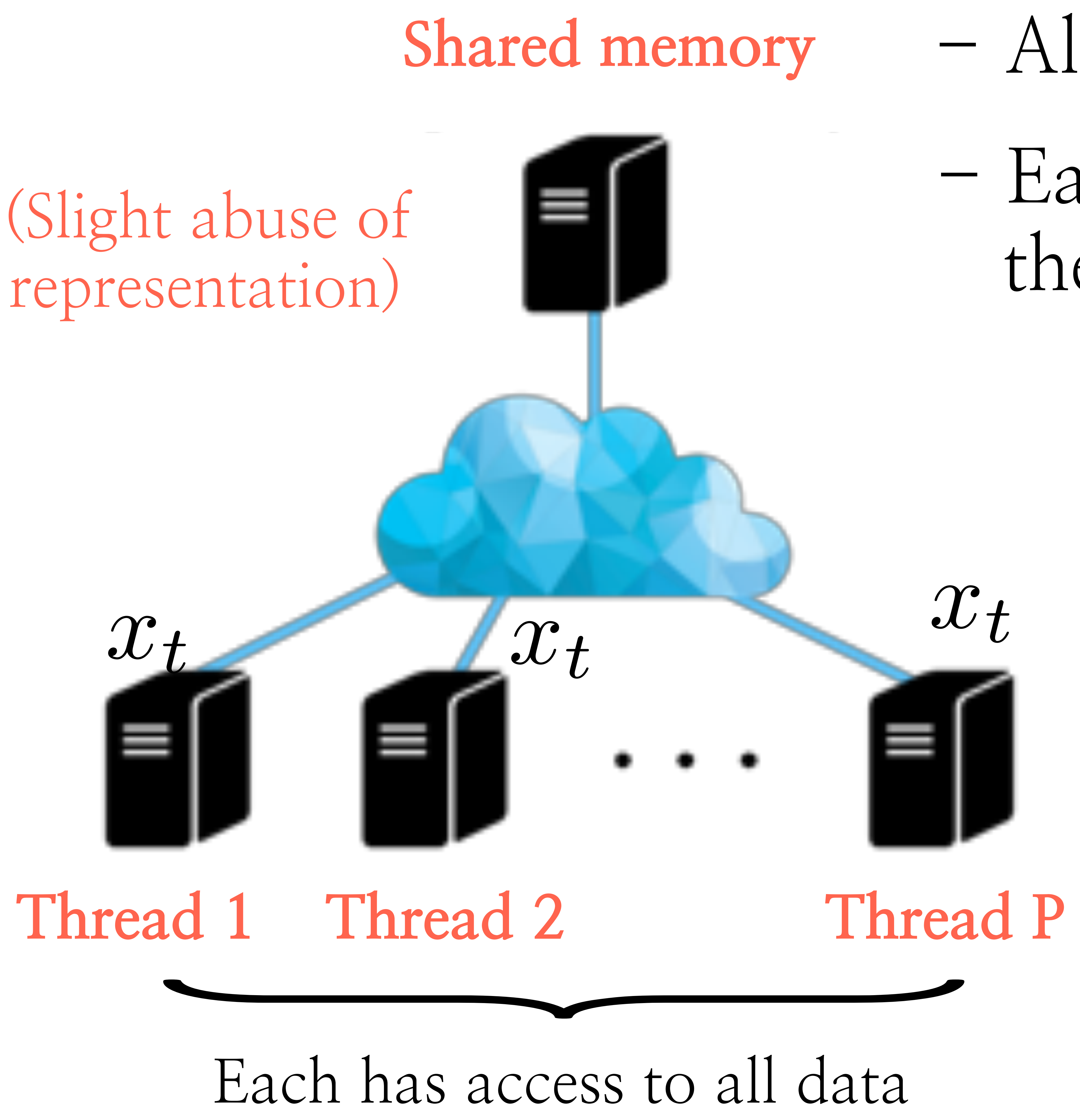


Thread 1 Thread 2 Thread P

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{Each has access to all data}}$$

# Asynchrony in SGD

– **Run SGD in parallel without locks!**

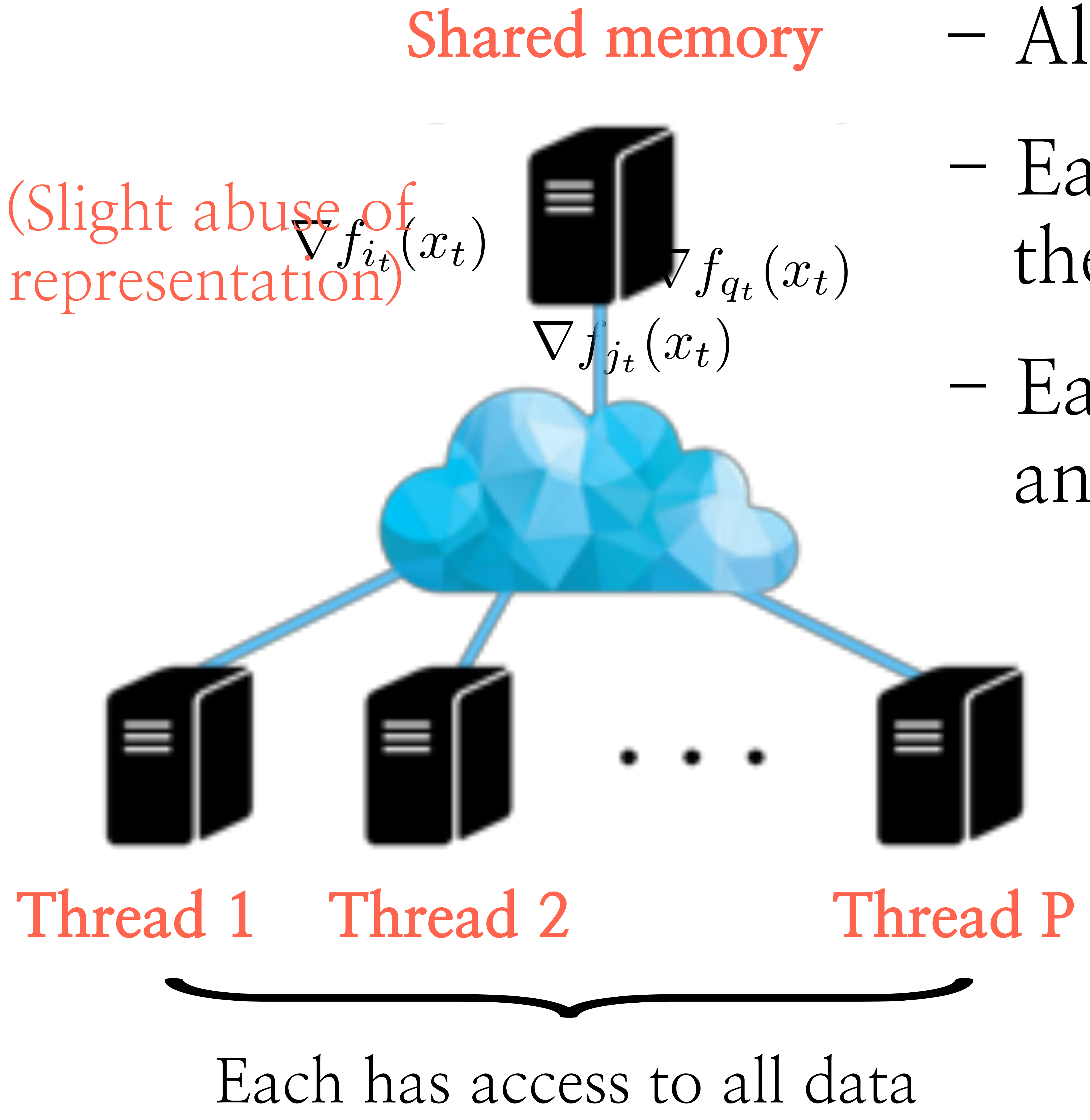



– All threads have access to shared memory

– Each thread can independently ask for the current model in memory

# Asynchrony in SGD

– **Run SGD in parallel without locks!**




– All threads have access to shared memory

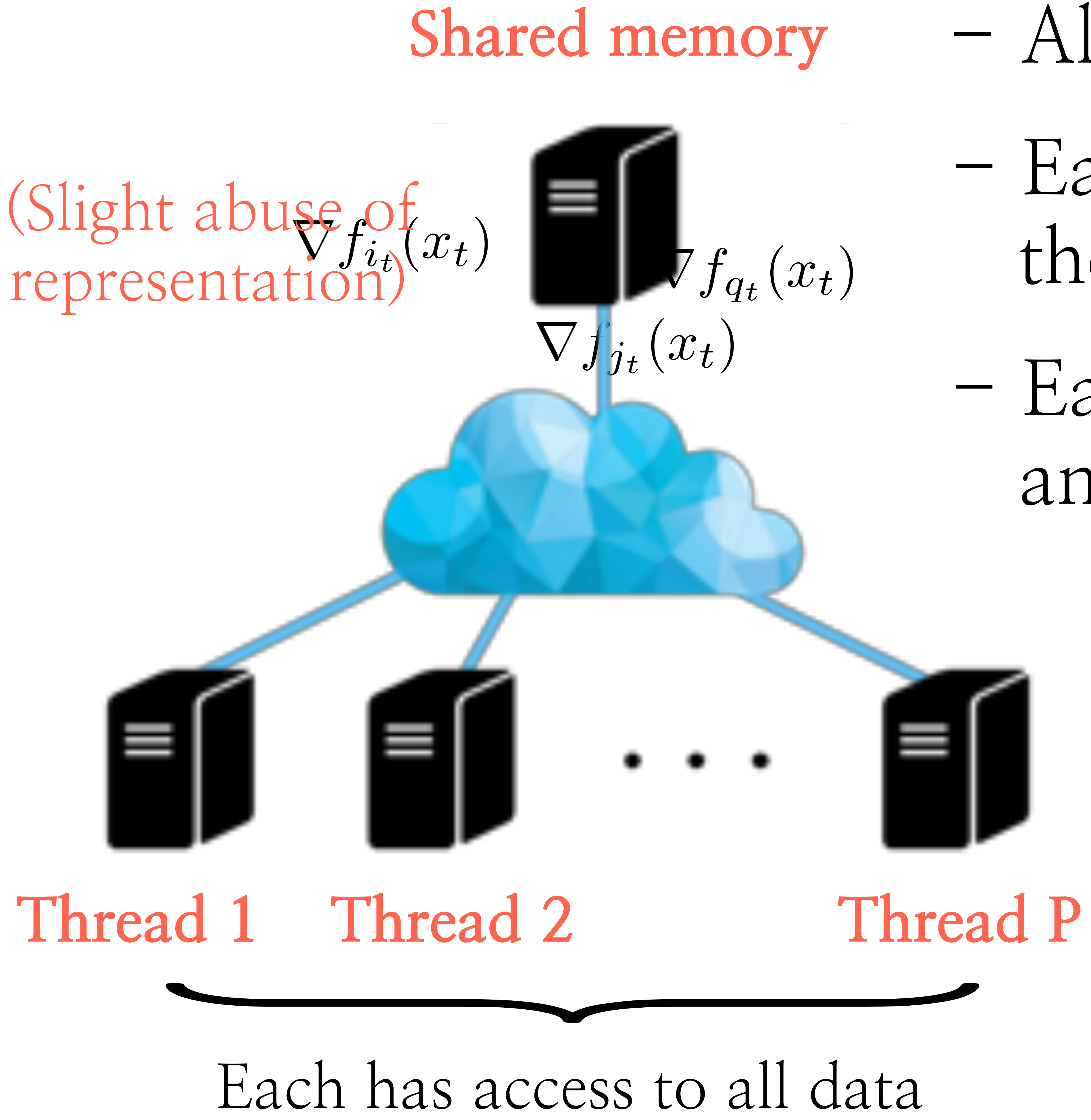
– Each thread can independently ask for the current model in memory

– Each thread computes an update (=gradient) and then updates shared memory

(the order that updates are sent is random)

# Asynchrony in SGD

– **Run SGD in parallel without locks!**




– All threads have access to shared memory

– Each thread can independently ask for the current model in memory

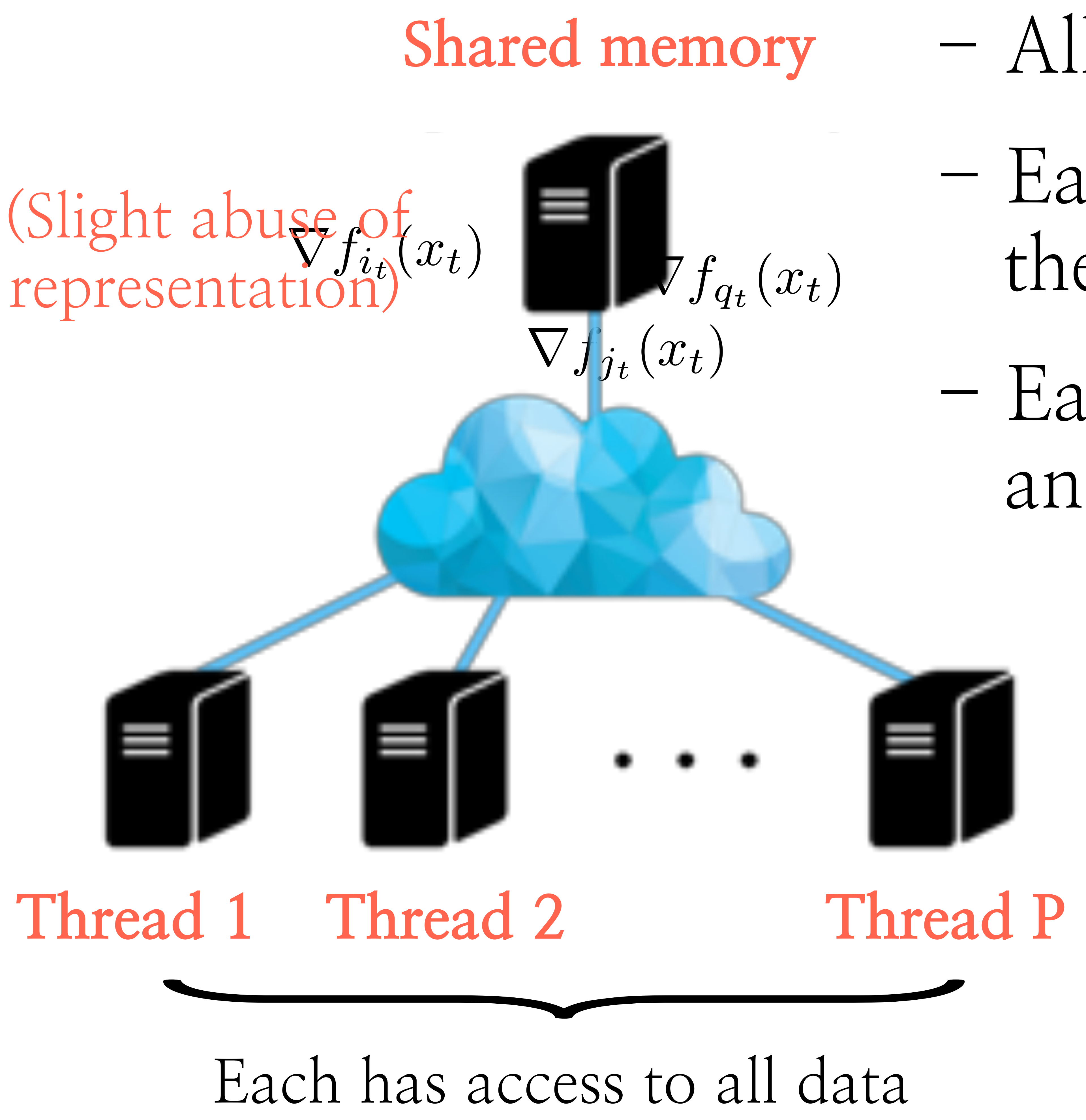
– Each thread computes an update (=gradient) and then updates shared memory

(the order that updates are sent is random)

– The controller in shared memory updates the model in a first–in–first–served fashion

# Asynchrony in SGD

– **Run SGD in parallel without locks!**




– All threads have access to shared memory

– Each thread can independently ask for the current model in memory

– Each thread computes an update (=gradient) and then updates shared memory

(the order that updates are sent is random)

– The controller in shared memory updates the model in a first-in-first-served fashion

– Assuming all threads have collected $x_t$

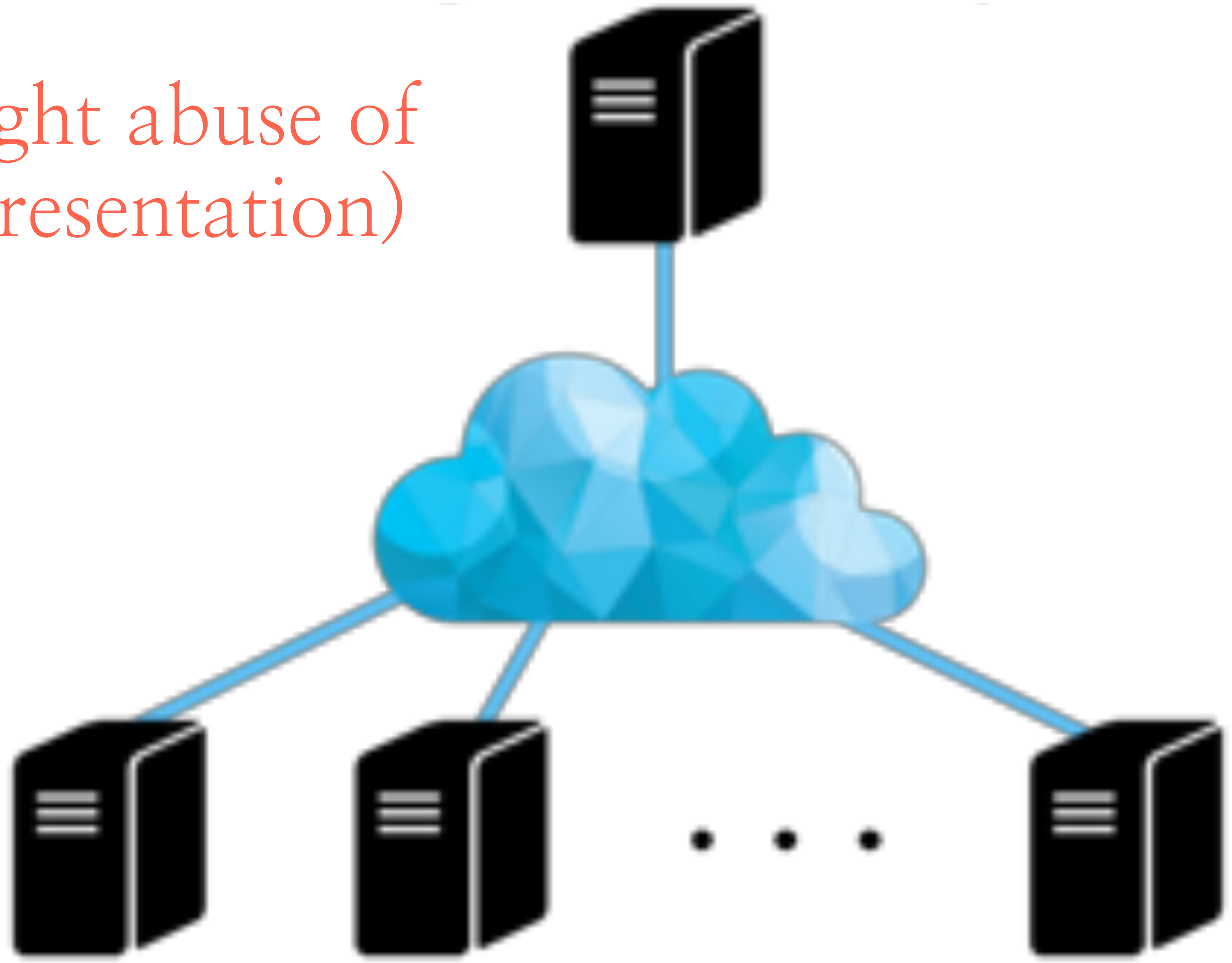
$$x_{t+1} = x_t - \eta\left(\nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + \cdots + \nabla f_{q_t}(x_t)\right)$$

# Asynchrony in SGD

– **Run SGD in parallel without locks!**

Shared memory

(Slight abuse of representation)



Thread 1 Thread 2 Thread P

Each has access to all data

$$x_{t+1} = x_t - \eta\left(\nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + \cdots + \nabla f_{q_t}(x_t)\right)$$

# Asynchrony in SGD

Google

Microsoft

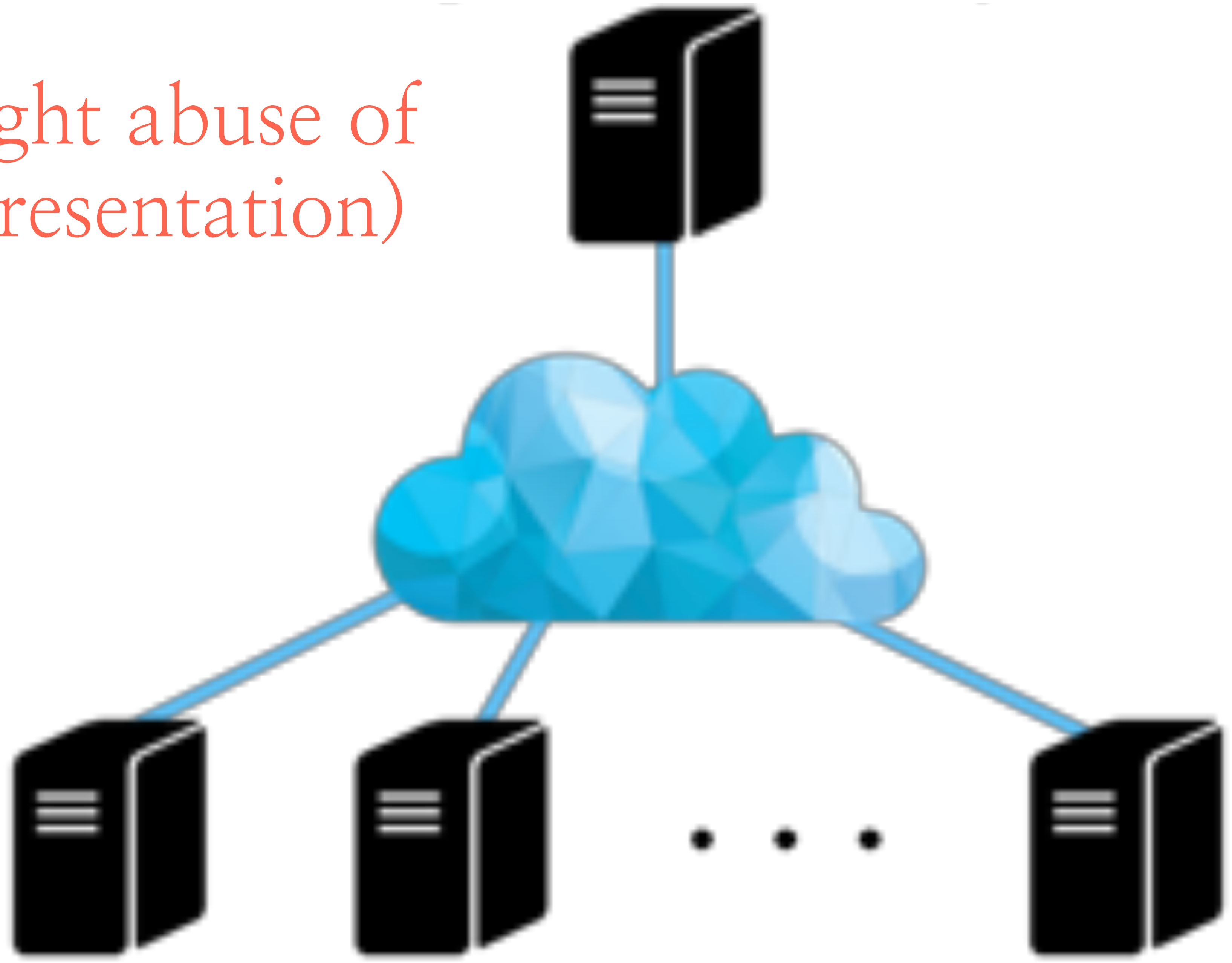
– **Run SGD in parallel without locks!**

Shared memory

$$x_{t+1} = x_t - \eta \left( \nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + \cdots + \nabla f_{q_t}(x_t) \right)$$

(..and this might be an straightforward case)

(Slight abuse of representation)

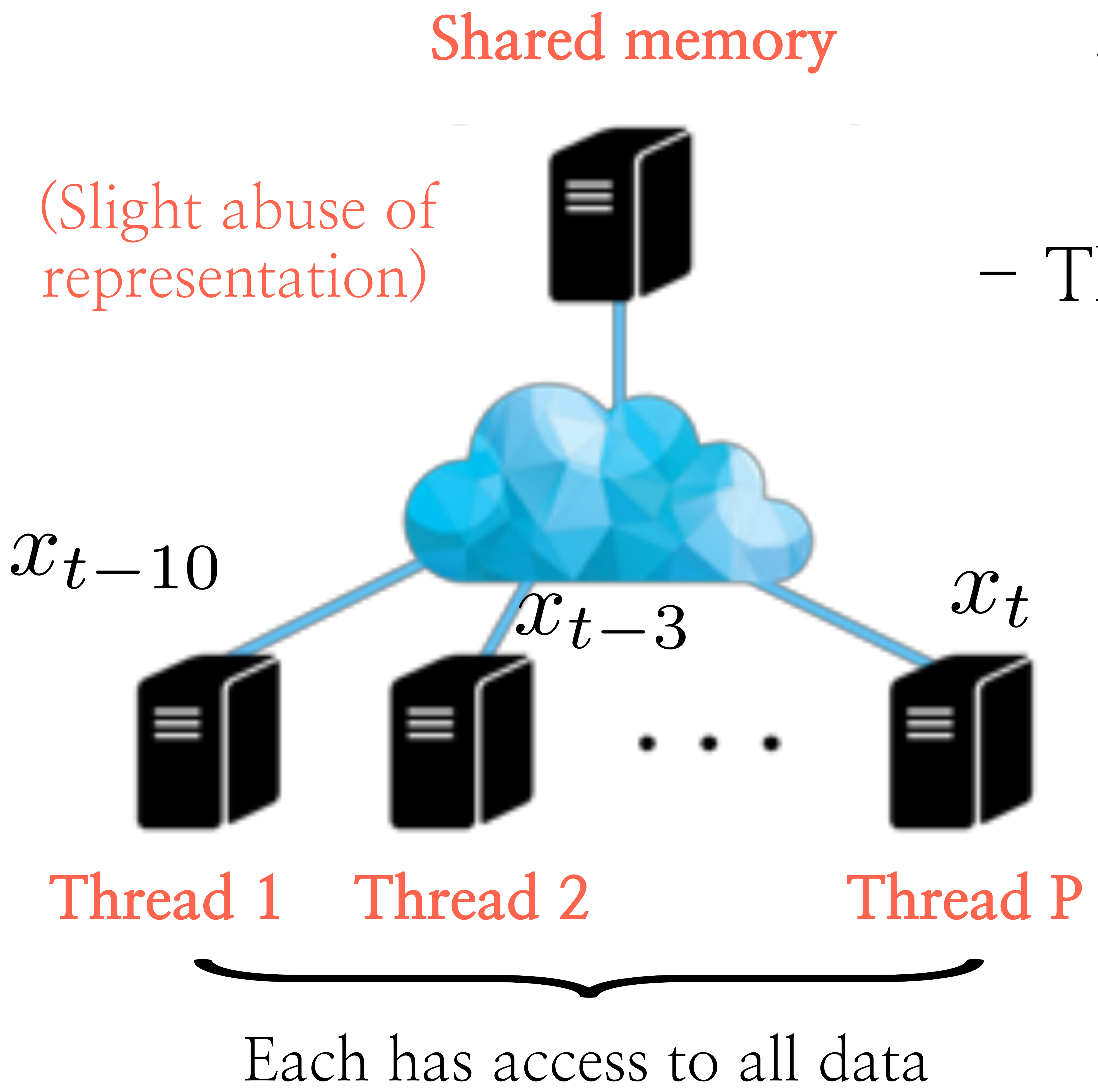


Thread 1 Thread 2 Thread P

Each has access to all data

# Asynchrony in SGD

– **Run SGD in parallel without locks!**




$$x_{t+1} = x_t - \eta \left(\nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + \cdots + \nabla f_{q_t}(x_t)\right)$$

(..and this might be an straightforward case)

– Threads might process an older model version

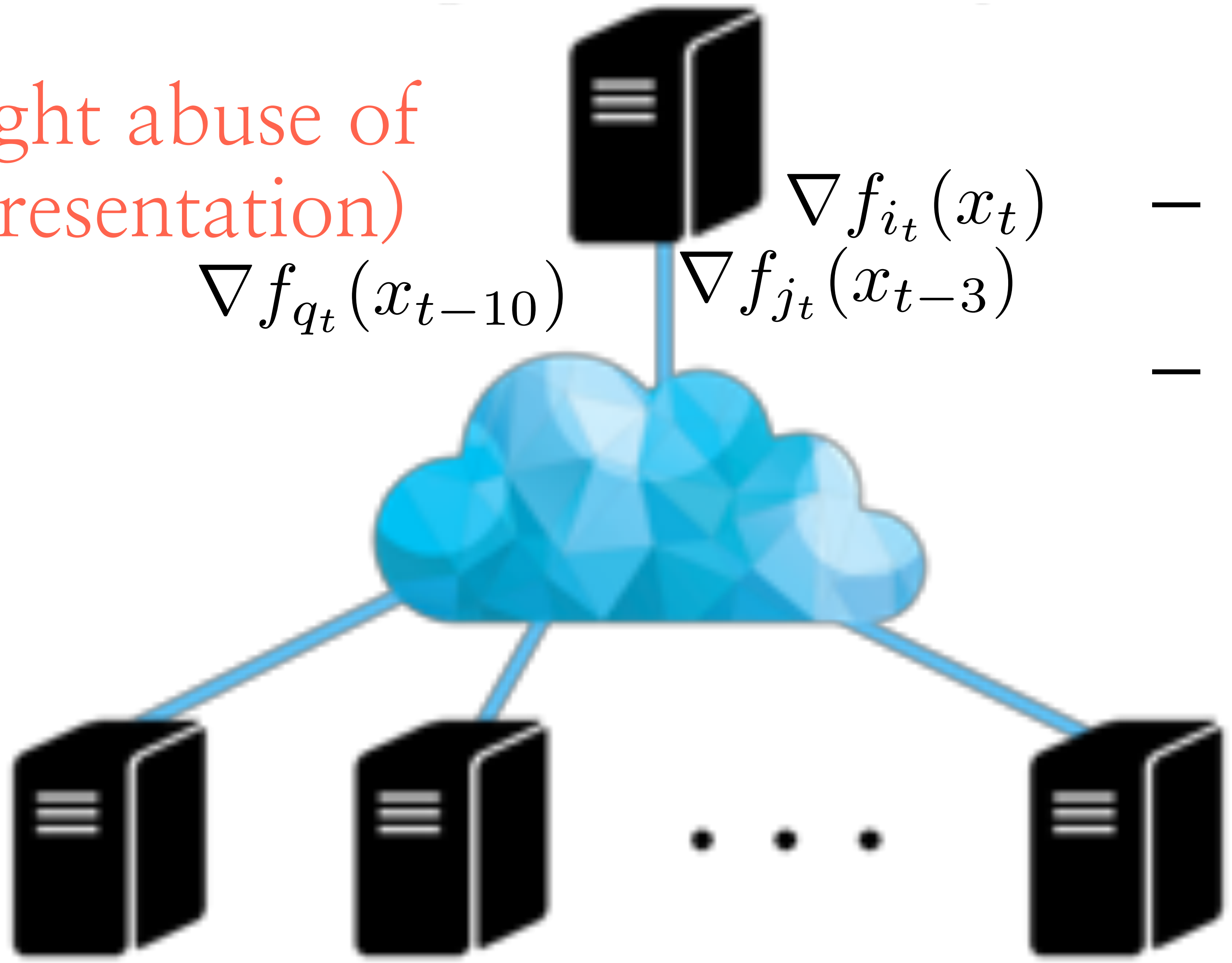
# Asynchrony in SGD

- Run SGD in parallel without locks!

Shared memory

$$x_{t+1} = x_t - \eta \left( \nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + \cdots + \nabla f_{q_t}(x_t) \right)$$

(Slight abuse of representation)




Thread 1 Thread 2 Thread P

$\underbrace{\hphantom{Thread 1 \quad Thread 2 \quad \cdots \quad Thread P}}_{}$

Each has access to all data

(..and this might be an straightforward case)

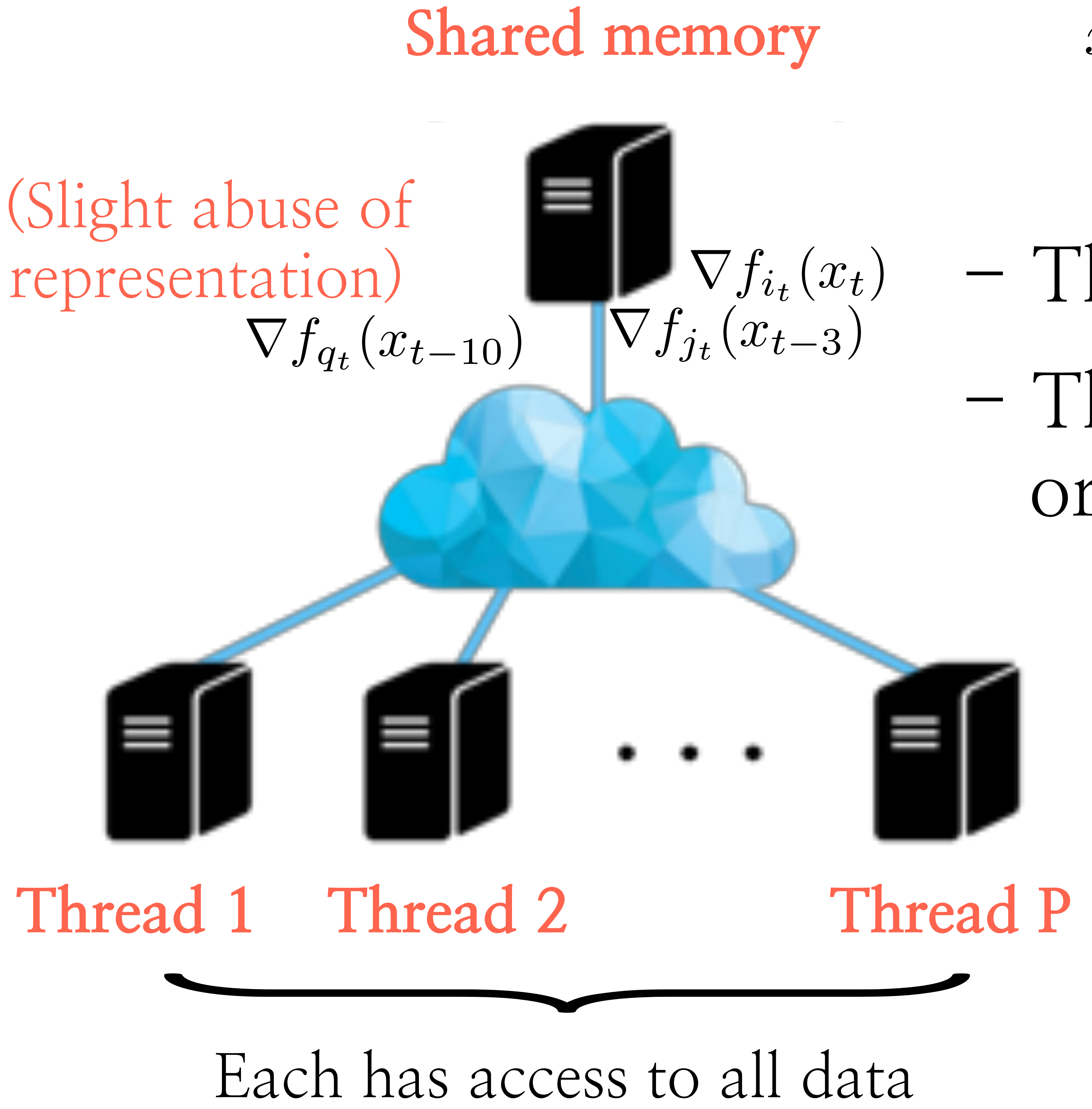
- Threads might process an older model version
- Threads might complete the ``job`` in an arbitrary order.

# Asynchrony in SGD

– **Run SGD in parallel without locks!**

$$x_{t+1} = x_t - \eta\left(\nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + \cdots + \nabla f_{q_t}(x_t)\right)$$

(..and this might be an straightforward case)




– Threads might process an older model version

– Threads might complete the "job" in an arbitrary order.

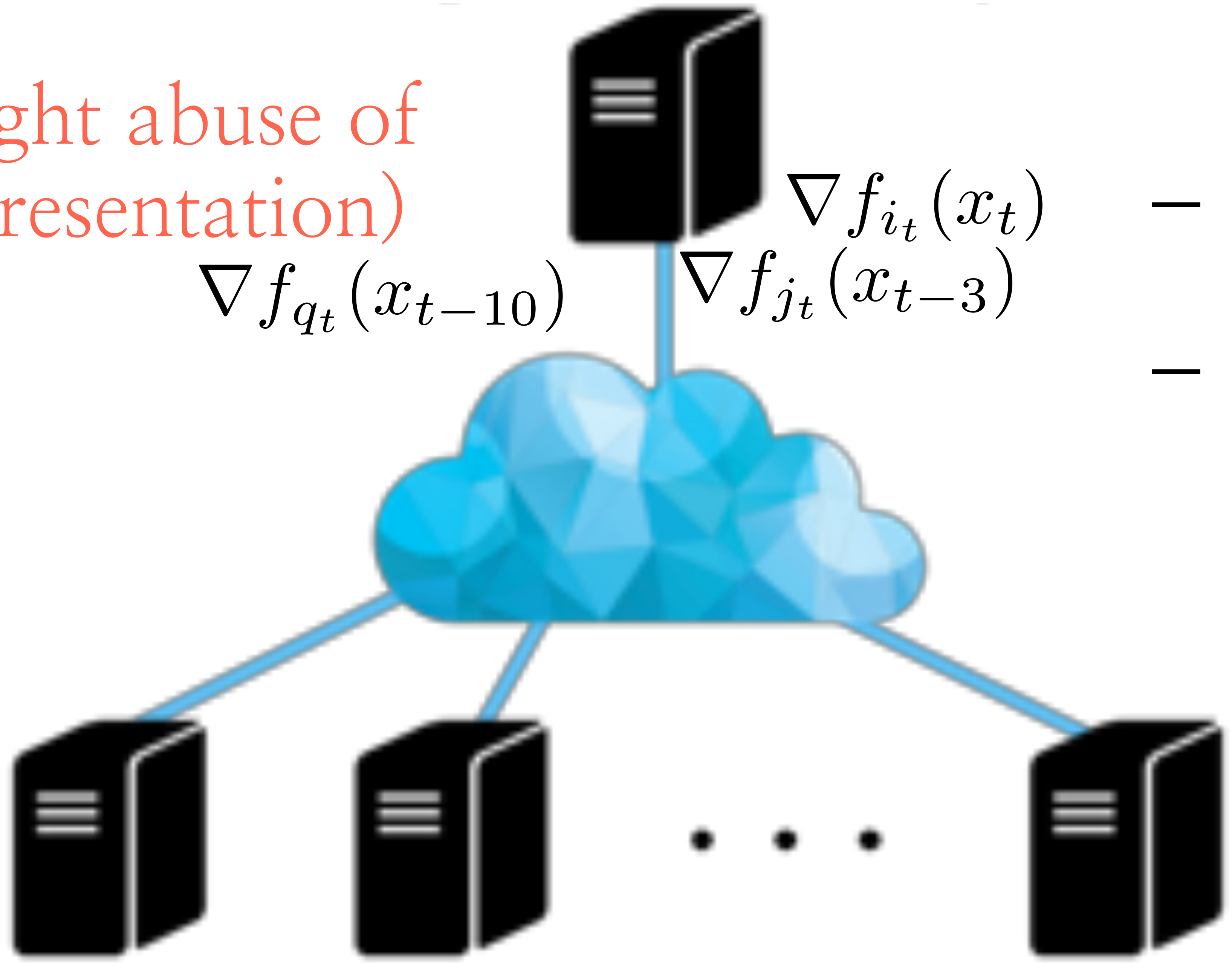
– And it can get more complex:

# Asynchrony in SGD

- **Run SGD in parallel without locks!**

Shared memory

(Slight abuse of representation)




Each has access to all data

$$x_{t+1} = x_t - \eta\left(\nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + \cdots + \nabla f_{q_t}(x_t)\right)$$

(..and this might be an straightforward case)

- Threads might process an older model version
- Threads might complete the "job" in an arbitrary order.

- And it can get more complex:

  "Threads can read a model state that only stayed in memory for a short time and between other memory writes"

# Asynchrony in SGD

- Does it work?

# Asynchrony in SGD

(..a bit more involved set up)

– Does it work?

## Large Scale Distributed Deep Networks

**Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen,**
**Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato,**
**Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng**
{jeff, gcorrado}@google.com
Google Inc., Mountain View, CA

# Asynchrony in SGD

(..a bit more involved set up)

– Does it work?

"..asynchronous SGD, rarely applied to nonconvex problems, works very well for training deep networks, particularly when combined with Adagrad adaptive learning rates.."

## Large Scale Distributed Deep Networks

**Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen,**
**Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato,**
**Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng**
`{jeff, gcorrado}@google.com`
Google Inc., Mountain View, CA

# Asynchrony in SGD

(..a bit more involved set up)

– Does it work?

"..asynchronous SGD, rarely applied to nonconvex problems, works very well for training deep networks, particularly when combined with Adagrad adaptive learning rates.."

"..There is little theoretical grounding for the safety of these operations for nonconvex problems, but in practice we found relaxing consistency requirements to be remarkably effective..."

**Large Scale Distributed Deep Networks**

**Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng**
{jeff, gcorrado}@google.com
Google Inc., Mountain View, CA

# Asynchrony in SGD

(..a bit more involved set up)

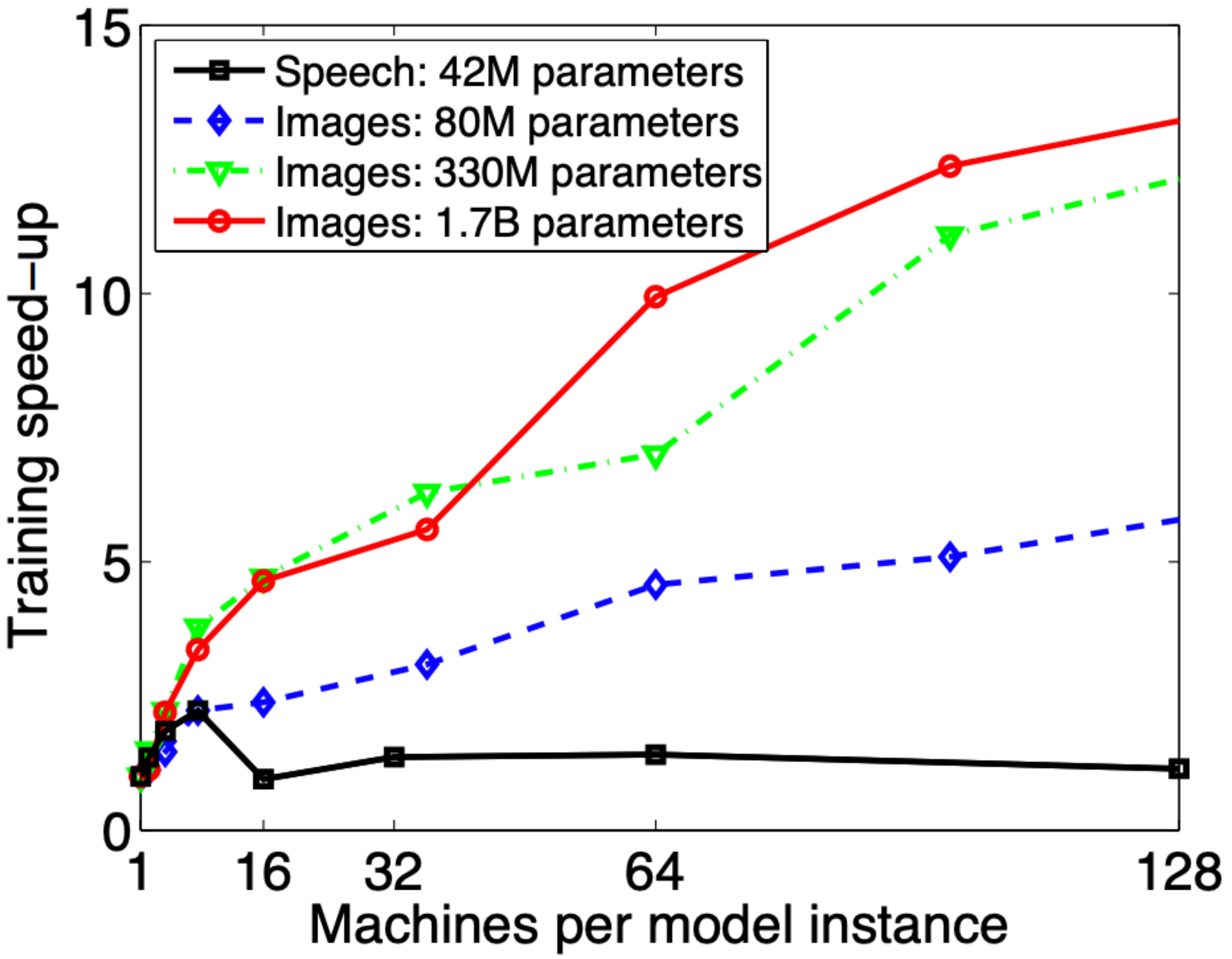
– Does it work?

## Large Scale Distributed Deep Networks

**Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen,**
**Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato,**
**Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng**
{jeff, gcorrado}@google.com
Google Inc., Mountain View, CA

"..asynchronous SGD, rarely applied to nonconvex problems, works very well for training deep networks, particularly when combined with Adagrad adaptive learning rates.."

"..There is little theoretical grounding for the safety of the operations for nonconvex problems, but in practice we found relaxing consistency requirements to be remarkably effective..."




– Comm. bottleneck

– We can increase batch size – but, we deal with worse generali–nation error

# Asynchrony in SGD

- Can we prove anything about asynchrony in SGD?

HOGWILD!: "..an update scheme that allows processors accesss shared memory with the possibility of overwriting each other's work

# Asynchrony in SGD

– **Can we prove anything about asynchrony in SGD?**

HOGWILD!: "..an update scheme that allows processors accesss shared memory with the possibility of overwriting each other's work

– Setting: $\min_x f(x) := \sum_{e \in E} f_e(x_e)$

where: $x \in \mathbb{R}^n$ $E$ is a collection of items, say samples

$e \subset [n]$ (each element e is a collection of indices in [n] but also an index from a set of samples E)

# Asynchrony in SGD

- **Can we prove anything about asynchrony in SGD?**

HOGWILD!: "..an update scheme that allows processors accesss shared memory with the possibility of overwriting each other's work

- Setting: $\min_x f(x) := \sum_{e \in E} f_e(x_e)$

where: $x \in \mathbb{R}^n$ $E$ is a collection of items, say samples

$e \subset [n]$ (each element e is a collection of indices in [n] but also an index from a set of samples E)

- Slight abuse of notation:

$f_e(\cdot)$: denotes a component of sum of functions, indexed by sample $e$

$x_e$ : corresponds to sub-vector, indexed by an index set $e$ (connected to sample $e$ )

# Asynchrony in SGD

– Key observation: $n$ & $|E|$ are large, while individual $f_e(\cdot)$ act on a small number of components of $x \in \mathbb{R}^n$

# Asynchrony in SGD

- Key observation: $n$ & $|E|$ are large, while individual $f_e(\cdot)$ act on a small number of components of $x \in \mathbb{R}^n$

- Example: Sparse SVM

  Given data $E = \{(z_1, y_1), \ldots, (z_{|E|}, y_{|E|}\}$ where $y_i$ labels and $z_i \in \mathbb{R}^n$ are features, we solve:

$$\min_x \sum_{\alpha \in E} \max\left(1 - y_\alpha \cdot x^\top z_\alpha,\ 0\right) + \lambda \|x\|_2^2$$

# Asynchrony in SGD

- Key observation: $n$ & $|E|$ are large, while individual $f_e(\cdot)$ act on a small number of components of $x \in \mathbb{R}^n$

- Example: Sparse SVM

  Given data $E = \{(z_1, y_1), \dots, (z_{|E|}, y_{|E|}\}$ where $y_i$ labels and $z_i \in \mathbb{R}^n$ are features, we solve:

$$\min_x \sum_{\alpha \in E} \max\left(1 - y_\alpha \cdot x^\top z_\alpha,\ 0\right) + \lambda \|x\|_2^2$$

- Observe that, if $z_\alpha$ is very sparse (which happens in reality often), then

$$x^\top z_\alpha = x_\alpha^\top z_\alpha \longleftarrow$$

Main objective depends on a subset of entries

# Asynchrony in SGD

- Some quantities:

  $\Omega$ : maximum number of features involved over all samples

  $\Delta$ : maximum frequency of features that can appear in samples

  $\rho$ : approaches 1 if features are very common across examples

  (..and thus we expect often collisions)

# Asynchrony in SGD

- Some quantities:

  $\Omega$ : maximum number of features involved over all samples

  $\Delta$ : maximum frequency of features that can appear in samples

  $\rho$ : approaches 1 if features are very common across examples

  (..and thus we expect often collisions)

- Configuration:

  $p$ : number of processors

  Each processor can read model $x$ and contribute an update to $x$

# Asynchrony in SGD

**Algorithm 1** HOGWILD! update for individual processors

1: **loop**
2: Sample $e$ uniformly at random from $E$
3: Read current state $x_e$ and evaluate $G_e(x)$
4: **for** $v \in e$ **do** $x_v \leftarrow x_v - \gamma b_v^T G_e(x)$ (coordinate-wise)
5: **end loop**

- Notation:

$G_e(x) \in \mathbb{R}^n$: gradient with non-zeros indexed by $e$, and scaled such that

$$\mathbb{E}\left[G_e(x_e)\right] = \nabla f(x)$$

Observe that $[G_e(x_e)]_{e^c} = 0$

# Asynchrony in SGD

**Algorithm 1** HOGWILD! update for individual processors

1: **loop**
2: Sample $e$ uniformly at random from $E$
3: Read current state $x_e$ and evaluate $G_e(x)$
4: **for** $v \in e$ **do** $x_v \leftarrow x_v - \gamma b_v^T G_e(x)$ ← (coordinate−wise)
5: **end loop**

– In words:

1. Each processor samples $e$ uniformly at random
2. Each processor computes the gradient $f_e$ at $x_e$
3. Each processor applies update on each coordinate in $e$

# Asynchrony in SGD

- Asynchrony: $x_j$ denotes the variable after j updates. Generally updated with stale gradients

  $x_{k(j)}$ denotes the state of the variable when was read

Whiteboard

# Asynchrony in SGD

- Asynchrony: $x_j$ denotes the variable after j updates. Generally updated with stale gradients

  $x_{k(j)}$ denotes the state of the variable when was read

Whiteboard

No Demo (no resources)

# Asynchrony in SGD

- Properties of asynchronous HOGWILD! algorithm:

# Asynchrony in SGD

- **Properties of asynchronous HOGWILD! algorithm:**
  - When the data access is **sparse** (i.e., SGD modifies a portion of the variables per step), memory overwrites could be rare

# Asynchrony in SGD

- **Properties of asynchronous HOGWILD! algorithm:**
  - When the data access is **sparse** (i.e., SGD modifies a portion of the variables per step), memory overwrites could be rare

  - This further indicates that **asynchrony introduces barely any error in the computations**

# Asynchrony in SGD

- **Properties of asynchronous HOGWILD! algorithm:**
  - When the data access is **sparse** (i.e., SGD modifies a portion of the variables per step), memory overwrites could be rare
  - This further indicates that **asynchrony introduces barely any error in the computations**
  - The authors show (theoretically and experimentally) a near-linear speedup, with the number of processors used

# Asynchrony in SGD

- **Properties of asynchronous HOGWILD! algorithm:**
  - When the data access is **sparse** (i.e., SGD modifies a portion of the variables per step), memory overwrites could be rare
  - This further indicates that **asynchrony introduces barely any error in the computations**
  - The authors show (theoretically and experimentally) a near-linear speedup, with the number of processors used
  - In practice, lock-free SGD exceeds even theoretical guarantees

# Alternatives to avoid asynchrony

(in other words, how we can decrease communication burden?)

- Standard SGD: each entry of the gradient is represented as a **float** number

  $O(32 \cdot p)$ bits : the size of each gradient sent over network

# Alternatives to avoid asynchrony

(in other words, how we can decrease communication burden?)

- Standard SGD: each entry of the gradient is represented as a **float** number

  $O(32 \cdot p)$ bits : the size of each gradient sent over network

- Quantized SGD: each entry of the gradient is quantized to some levels

  $O(\ell \cdot p)$ bits : where $\ell \ll 32$ is the levels of quantization

# Alternatives to avoid asynchrony

(in other words, how we can decrease communication burden?)

– Standard SGD: each entry of the gradient is represented as a **float** number

$O(32 \cdot p)$ bits : the size of each gradient sent over network

– Quantized SGD: each entry of the gradient is quantized to some levels

$O(\ell \cdot p)$ bits : where $\ell \ll 32$ is the levels of quantization

**QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding**

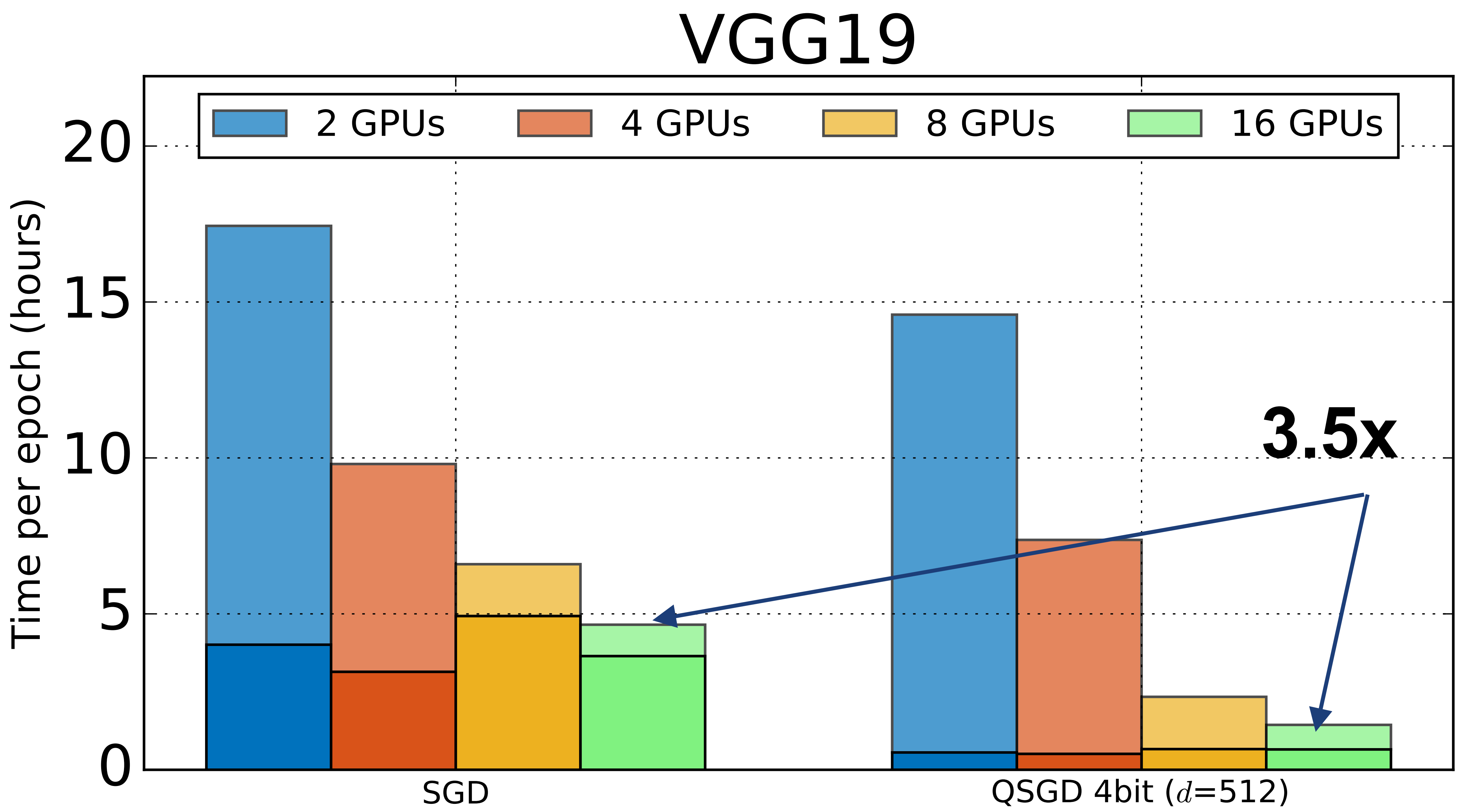

**Dan Alistarh**
IST Austria & ETH Zurich
dan.alistarh@ist.ac.at

**Demjan Grubic**
ETH Zurich & Google
demjangrubic@gmail.com

**Jerry Z. Li**
MIT
jerryzli@mit.edu

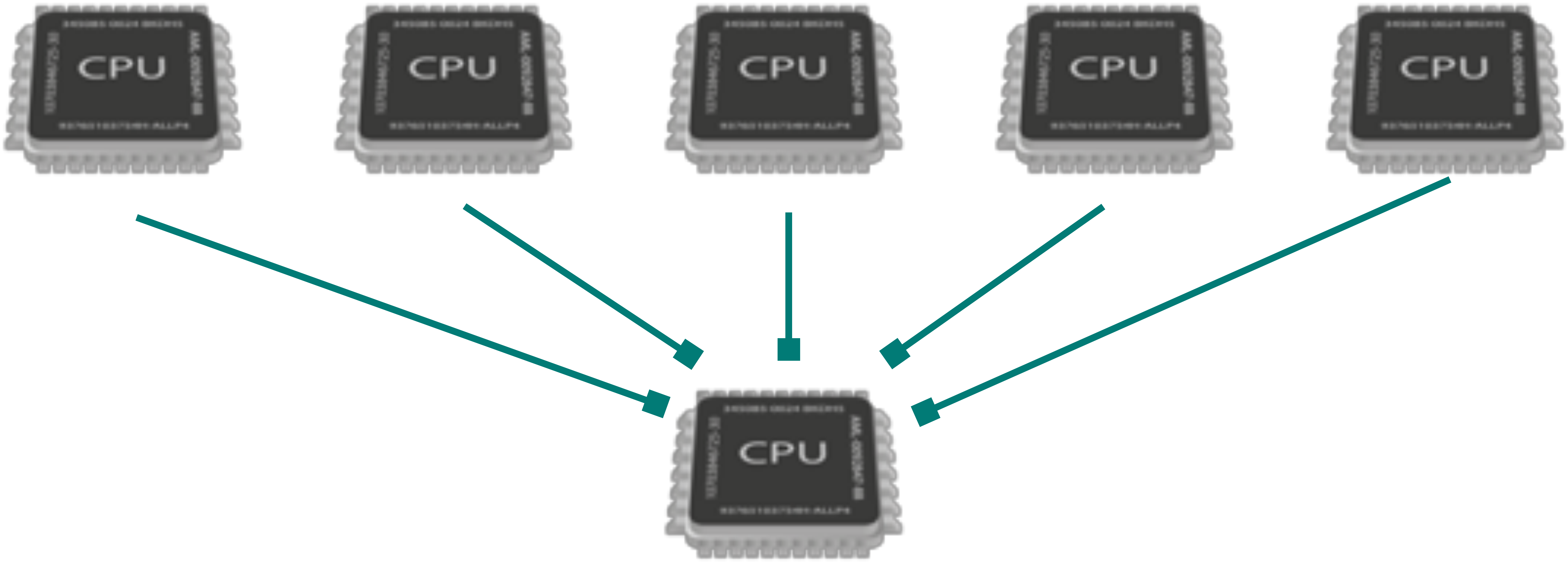
**Ryota Tomioka**
Microsoft Research
ryoto@microsoft.com

**Milan Vojnovic**
London School of Economics
M.Vojnovic@lse.ac.uk

# Alternatives to avoid asynchrony

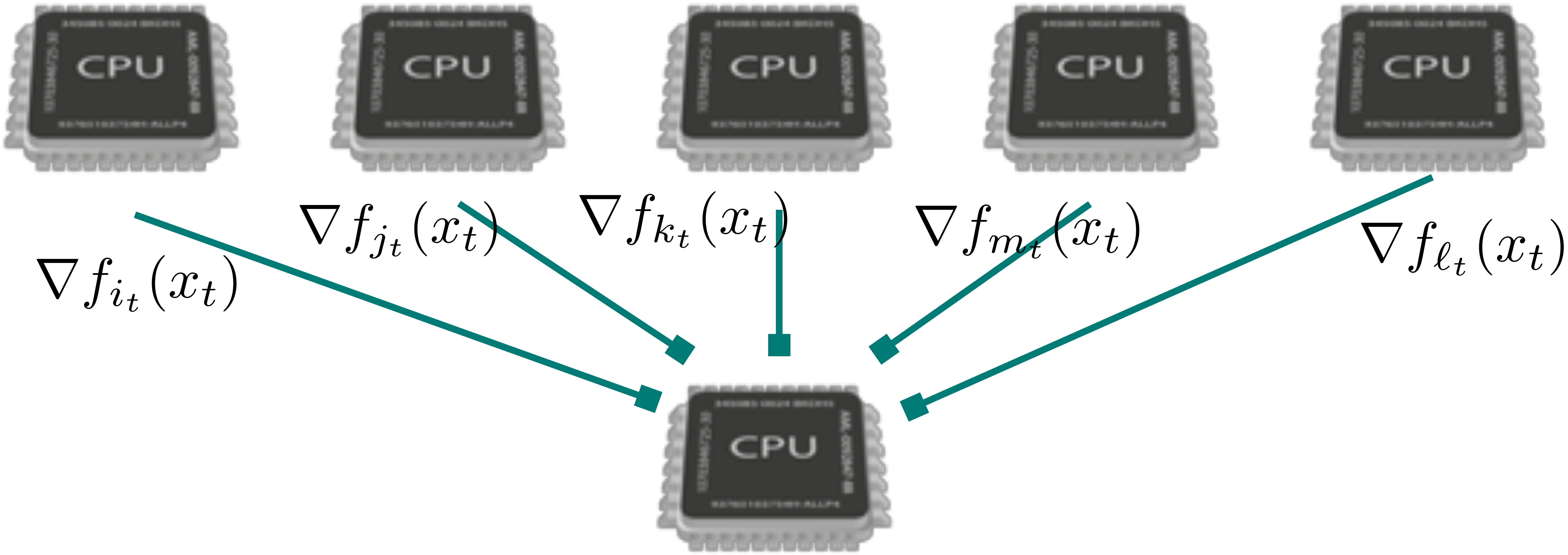
(in other words, can we make synchronization not be a big problem?)




- Right learner can slow down the performance of synchronized SGD

# Alternatives to avoid asynchrony

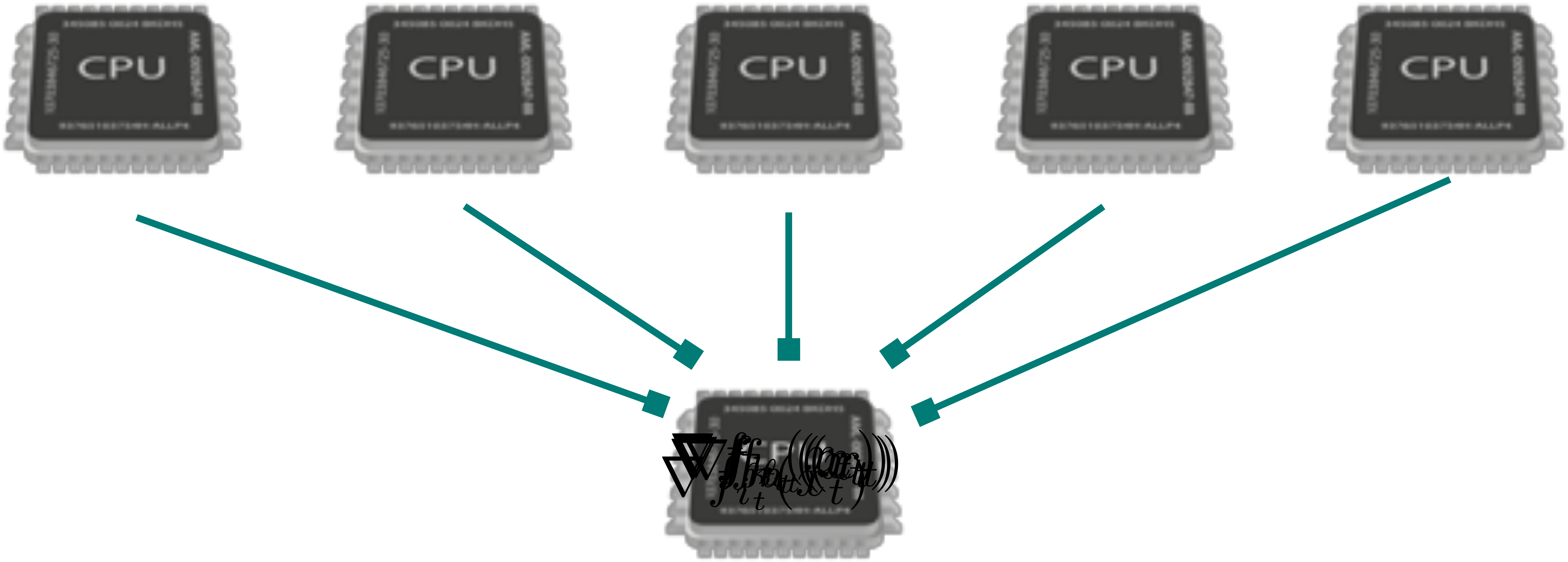
(in other words, can we make synchronization not be a big problem?)




– Right learner can slow down the performance of synchronized SGD

# Alternatives to avoid asynchrony

(in other words, can we make synchronization not be a big problem?)




– Right learner can slow down the performance of synchronized SGD

# Alternatives to avoid asynchrony

(in other words, can we make synchronization not be a big problem?)

## Revisiting Distributed Synchronous SGD

**Jianmin Chen**[*], **Xinghao Pan**[*†], **Rajat Monga, Samy Bengio**
Google Brain
Mountain View, CA, USA
{jmchen,xinghao,rajatmonga,bengio}@google.com

**Rafal Jozefowicz**
OpenAI
San Francisco, CA, USA
rafal@openai.com

### Abstract

Distributed training of deep learning models on large-scale training data is typically conducted with *asynchronous* stochastic optimization to maximize the rate of updates, at the cost of additional noise introduced from asynchrony. In contrast, the *synchronous* approach is often thought to be impractical due to idle time wasted on waiting for straggling workers. We revisit these conventional beliefs in this paper, and examine the weaknesses of both approaches. We demonstrate that a third approach, synchronous optimization with backup workers, can avoid asynchronous noise while mitigating for the worst stragglers. Our approach is empirically validated and shown to converge *faster* and to *better* test accuracies.

# Alternatives to avoid asynchrony

(in other words, can we make synchronization not be a big problem?)

# Alternatives to avoid asynchrony

(in other words, can we make synchronization not be a big problem?)

- **Sparsification of gradients**: instead of quantizing all entries, keep the most important ones

# Alternatives to avoid asynchrony

(in other words, can we make synchronization not be a big problem?)

- **Sparsification of gradients:** instead of quantizing all entries, keep the most important ones
- **Large batch training:** give more ``work`` to workers by increasing the batch size. However it needs careful parameter tuning to make it work

# Alternatives to avoid asynchrony

(in other words, can we make synchronization not be a big problem?)

- **Sparsification of gradients:** instead of quantizing all entries, keep the most important ones

- **Large batch training:** give more ``work`` to workers by increasing the batch size. However it needs careful parameter tuning to make it work

- **Variants of HOGWILD! that minimize communication conflicts:** some computation is performed to distribute examples to different cores so that examples do not ``conflict``.

# Conclusion

- Distributed computing is at the heart of developments in modern ML
- There are different ways to exploit distributed computing: **hyper parameter optimization, coordinate descent, mini-batch synchronous** SGD, **asynchronous** SGD
- Which configuration to use depends on the problem and the resources at hand
- These topics are highly attractive (research-wise): they define the notion of systems + machine learning (look for SysML conference)