Distributed Learning - Data Parallelization

Amir H. Payberah
payberah@kth.se
2020-10-12
https://fid3024.github.io
Replicate a whole model on every device.

Data Parallelization (1/4)

- Replicate a whole model on every device.
- Train all replicas simultaneously, using a different mini-batch for each.

Data Parallelization (2/4)

- \( k \) devices

\[ \begin{align*}
J_i(w) &= \left| \beta_i \right| \sum_{x \in \beta_i} l(x, w), \quad \forall i = 1, 2, \ldots, k \\
G_i(w, \beta_i) &= \left| \beta_i \right| \sum_{x \in \beta_i} \nabla l(w, x)
\end{align*} \]

[Update]

[Load data]

Data Parallelization (2/4)

- **k** devices

- \( J_i(w) = \frac{1}{|\beta_i|} \sum_{x \in \beta_i} l(x, w), \forall i = 1, 2, \cdots, k \)

Data Parallelization (2/4)

- **k** devices

\[ J_i(w) = \frac{1}{|\beta_i|} \sum_{x \in \beta_i} l(x, w), \forall i = 1, 2, \cdots, k \]

- \[ G_i(w, \beta_i) = \frac{1}{|\beta_i|} \sum_{x \in \beta_i} \nabla l(w, x) \]

- **k** devices
- $J_i(w) = \frac{1}{|\beta_i|} \sum_{x \in \beta_i} l(x, w), \forall i = 1, 2, \ldots, k$
- $G_i(w, \beta_i) = \frac{1}{|\beta_i|} \sum_{x \in \beta_i} \nabla l(w, x)$
- $G_i(w, \beta_i)$: the local estimate of the gradient of the loss function $\nabla J_i(w)$.

Data Parallelization (3/4)

- Compute the gradients aggregation (e.g., mean of the gradients).
- \( F(G_1, \cdots, G_k) = \frac{1}{k} \sum_{i=1}^{k} G_i(w, \beta_i) \)

- **Update the model.**
- \( \mathbf{w} := \mathbf{w} - \eta F(G_1, \cdots, G_k) \)

Data Parallelization Design Issues

- The aggregation algorithm
- Communication synchronization and frequency
- Communication compression
- Parallelism of computations and communications
The Aggregation Algorithm
How to aggregate gradients (compute the mean of the gradients)?
The Aggregation Algorithm

- How to aggregate gradients (compute the mean of the gradients)?

- Centralized - parameter server
How to aggregate gradients (compute the mean of the gradients)?

- Centralized - parameter server
- Decentralized - all-reduce
The Aggregation Algorithm

- **How to aggregate gradients** (compute the mean of the gradients)?
- Centralized - parameter server
- Decentralized - all-reduce
- Decentralized - gossip
Aggregation - Centralized - Parameter Server

- Store the model parameters outside of the workers.
Aggregation - Centralized - Parameter Server

- Store the model parameters outside of the workers.
- Workers periodically report their computed parameters or parameter updates to a (set of) parameter server(s) (PSs).

Mirror all the model parameters across all workers (no PS).
Mirror all the model parameters across all workers (no PS).

Workers exchange parameter updates directly via an allreduce operation.

Aggregation - Distributed - Gossip

- No PS, and no global model.

Aggregation - Distributed - Gossip

- No PS, and no global model.
- Every worker communicates updates with their neighbors.

Aggregation - Distributed - Gossip

- No PS, and no global model.
- Every worker communicates updates with their neighbors.
- The consistency of parameters across all workers only at the end of the algorithm.

Reduce and AllReduce (1/2)

- **Reduce**: reducing a set of numbers into a smaller set of numbers via a function.

  E.g., \( \text{sum}([1, 2, 3, 4, 5]) = 15 \)

  Reduce takes an array of input elements on each process and returns an array of output elements to the root process.

  [https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce]
Reduce and AllReduce (1/2)

- **Reduce**: reducing a *set of numbers* into a *smaller set of numbers* via a function.
- E.g., \( \text{sum([1, 2, 3, 4, 5])} = 15 \)
- **Reduce**: reducing a set of numbers into a smaller set of numbers via a function.
- E.g., \( \text{sum}([1, 2, 3, 4, 5]) = 15 \)
- Reduce takes an array of input elements on each process and returns an array of output elements to the root process.
Reduce and AllReduce (1/2)

- **Reduce**: reducing a set of numbers into a smaller set of numbers via a function.
- E.g., \( \text{sum}([1, 2, 3, 4, 5]) = 15 \)
- Reduce takes an array of input elements on each process and returns an array of output elements to the root process.

[https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce]
AllReduce stores reduced results across all processes rather than the root process.
- **AllReduce** stores **reduced results** across **all processes** rather than the root process.

[https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce]
AllReduce Example

Initial state

Worker A
17 11 1 9
Worker B
5 13 23 14
Worker C
3 6 10 8
Worker D
12 7 2 12

After AllReduce operation

Worker A
37 37 36 43
Worker B
37 37 36 43
Worker C
37 37 36 43
Worker D
37 37 36 43

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
AllReduce Implementation

- All-to-all allreduce
- Master-worker allreduce
- Tree allreduce
- Round-robin allreduce
- Butterfly allreduce
- Ring allreduce
AllReduce Implementation - All-to-All AllReduce

- Send the array of data to each other.
- Apply the reduction operation on each process.

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
AllReduce Implementation - All-to-All AllReduce

- Send the array of data to each other.
- Apply the reduction operation on each process.
- Too many unnecessary messages.

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
Selecting one process as a master, gather all arrays into the master.

Perform reduction operations locally in the master.

Distribute the result to the other processes.
- Selecting **one process** as a **master**, gather all arrays into the master.
- Perform **reduction operations** locally in the **master**.
- **Distribute the result** to the **other processes**.
- The master becomes a **bottleneck (not scalable)**.

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
AllReduce Implementation - Other implementations

- Some try to minimize bandwidth.
- Some try to minimize latency.

[Zhao H. et al., arXiv:1312.3020, 2013]
The Ring-Allreduce has two phases:
1. First, the share-reduce phase
2. Then, the share-only phase
In the share-reduce phase, each process $p$ sends data to the process $(p+1) \% m$

- $m$ is the number of processes, and $\%$ is the modulo operator.
In the *share-reduce* phase, each process $p$ sends data to the process $(p+1)\% m$
- $m$ is the number of processes, and $\%$ is the modulo operator.

The array of data on each process is divided to $m$ chunks ($m=4$ here).
In the share-reduce phase, each process $p$ sends data to the process $(p+1) \mod m$
  • $m$ is the number of processes, and $\%$ is the modulo operator.

The array of data on each process is divided to $m$ chunks ($m=4$ here).

Each one of these chunks will be indexed by $i$ going forward.
In the first share-reduce step, process A sends $a_0$ to process B.
In the first share-reduce step, process A sends $a_0$ to process B.

Process B sends $b_1$ to process C, etc.
When each process receives the data from the previous process, it applies the reduce operator (e.g., sum)

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
When each process **receives the data** from the previous process, it applies the **reduce operator** (e.g., sum)

- The reduce operator should be **associative** and **commutative**.

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
When each process receives the data from the previous process, it applies the reduce operator (e.g., sum)

- The reduce operator should be associative and commutative.

It then proceeds to send it to the next process in the ring.
The **share-reduce** phase finishes when each process holds the **complete reduction** of chunk $i$. 

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
The **share-reduce** phase finishes when each process holds the **complete reduction** of chunk \(i\).

At this point each **process** holds a part of the **end result**.

\[ r_i = a_i + b_i + c_i + d_i \]

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
The **share-only** step is the same process of sharing the data in a ring-like fashion **without** applying the reduce operation.

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da]
The **share-only** step is the same process of sharing the data in a ring-like fashion **without** applying the reduce operation.

This consolidates the **result of each chunk** in every process.

$$r_i = a_i + b_i + c_i + d_i$$

[https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da](https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da)
Master-Worker AllReduce vs. Ring-AllReduce

- $N$: number of elements, $m$: number of processes

- **Master-Worker AllReduce**
  - First each process sends $N$ elements to the master: $N \times (m - 1)$ messages.
  - Then the master sends the results back to the process: another $N \times (m - 1)$ messages.
  - Total network traffic is $2(N \times (m - 1))$, which is proportional to $m$.

- **Ring-AllReduce**
  - In the share-reduce step each process sends $N^m$ elements, and it does it $m - 1$ times: $N^m \times (m - 1)$ messages.
  - On the share-only step, each process sends the result for the chunk it calculated: another $N^m \times (m - 1)$ messages.
  - Total network traffic is $2(N^m \times (m - 1))$. 


Master-Worker AllReduce vs. Ring-AllReduce

- **N**: number of elements, **m**: number of processes

**Master-Worker AllReduce**

- First each process sends \( N \) elements to the master: \( N \times (m - 1) \) messages.
- Then the master sends the results back to the process: another \( N \times (m - 1) \) messages.
- Total network traffic is \( 2 \times (N \times (m - 1)) \), which is proportional to \( m \).
Master-Worker AllReduce vs. Ring-AllReduce

- \(N\): number of elements, \(m\): number of processes

- **Master-Worker AllReduce**
  - First each **process** sends \(N\) elements to the **master**: \(N \times (m - 1)\) messages.
Master-Worker AllReduce vs. Ring-AllReduce

- **N**: number of elements, **m**: number of processes

- **Master-Worker AllReduce**
  - First each process sends \( N \) elements to the master: \( N \times (m - 1) \) messages.
  - Then the master sends the results back to the process: another \( N \times (m - 1) \) messages.
Master-Worker AllReduce vs. Ring-AllReduce

- **N**: number of elements, **m**: number of processes

**Master-Worker AllReduce**
- First each **process** sends **N** elements to the **master**: \( N \times (m - 1) \) messages.
- Then the **master** sends the results back to the **process**: another \( N \times (m - 1) \) messages.
- Total network traffic is \( 2(N \times (m - 1)) \), which is proportional to \( m \).
Master-Worker AllReduce vs. Ring-AllReduce

- \( N \): number of elements, \( m \): number of processes

- **Master-Worker AllReduce**
  - First each **process** sends \( N \) elements to the **master**: \( N \times (m - 1) \) messages.
  - Then the **master** sends the results back to the **process**: another \( N \times (m - 1) \) messages.
  - Total network traffic is \( 2(N \times (m - 1)) \), which is **proportional** to \( m \).

- **Ring-AllReduce**
Master-Worker AllReduce vs. Ring-AllReduce

- **N**: number of elements, **m**: number of processes

- **Master-Worker AllReduce**
  - First each **process** sends **N** elements to the **master**: \(N \times (m - 1)\) messages.
  - Then the **master** sends the results back to the **process**: another \(N \times (m - 1)\) messages.
  - Total network traffic is \(2(N \times (m - 1))\), which is proportional to \(m\).

- **Ring-AllReduce**
  - In the **share-reduce** step each **process** sends \(\frac{N}{m}\) elements, and it does it \(m - 1\) times: \(\frac{N}{m} \times (m - 1)\) messages.
Master-Worker AllReduce vs. Ring-AllReduce

- **N**: number of elements, **m**: number of processes

**Master-Worker AllReduce**
- First each **process** sends **N** elements to the **master**: \(N \times (m - 1)\) messages.
- Then the **master** sends the results back to the **process**: another \(N \times (m - 1)\) messages.
- Total network traffic is \(2(N \times (m - 1))\), which is proportional to **m**.

**Ring-AllReduce**
- In the **share-reduce** step each **process** sends \(\frac{N}{m}\) elements, and it does it \(m - 1\) times: \(\frac{N}{m} \times (m - 1)\) messages.
- On the **share-only** step, each **process** sends the result for the chunk it calculated: another \(\frac{N}{m} \times (m - 1)\) messages.
Master-Worker AllReduce vs. Ring-AllReduce

- **N**: number of elements, **m**: number of processes

- **Master-Worker AllReduce**
  - First each **process** sends **N** elements to the **master**: \(N \times (m - 1)\) messages.
  - Then the **master** sends the results back to the **process**: another \(N \times (m - 1)\) messages.
  - Total network traffic is \(2(N \times (m - 1))\), which is proportional to \(m\).

- **Ring-AllReduce**
  - In the **share-reduce** step each **process** sends \(\frac{N}{m}\) elements, and it does it \(m - 1\) times: \(\frac{N}{m} \times (m - 1)\) messages.
  - On the **share-only** step, each **process** sends the result for the chunk it calculated: another \(\frac{N}{m} \times (m - 1)\) messages.
  - Total network traffic is \(2\left(\frac{N}{m} \times (m - 1)\right)\).
Communication Synchronization and Frequency
Synchronization

- When to synchronize the parameters among the parallel workers?
Synchronizing the model replicas in data-parallel training requires communication

- between workers, in allreduce
- between workers and parameter servers, in the centralized architecture
Synchronizing the model replicas in data-parallel training requires communication
  • between workers, in allreduce
  • between workers and parameter servers, in the centralized architecture

The communication synchronization decides how frequently all local models are synchronized with others.
It will influence:

- The communication \textit{traffic}
- The \textit{performance}
- The \textit{convergence} of model training
Communication Synchronization (2/2)

- It will influence:
  - The communication traffic
  - The performance
  - The convergence of model training

- There is a trade-off between the communication traffic and the convergence.
Reducing Synchronization Overhead

Two directions for improvement:
Reducing Synchronization Overhead

Two directions for improvement:

1. To relax the synchronization among all workers.
Reducing Synchronization Overhead

- Two directions for improvement:

1. To relax the synchronization among all workers.

2. The frequency of communication can be reduced by more computation in one iteration.
Communication Synchronization Models

- Synchronous
- Stale-synchronous
- Asynchronous
- Local SGD
After each iteration, the workers synchronize their parameter updates.
After each iteration, the workers synchronize their parameter updates.

Every worker must wait for all workers to finish the transmission of all parameters in the current iteration, before the next training.

After each iteration, the workers synchronize their parameter updates.

Every worker must wait for all workers to finish the transmission of all parameters in the current iteration, before the next training.

Stragglers can influence the overall system throughput.

After each iteration, the workers synchronize their parameter updates.

Every worker must wait for all workers to finish the transmission of all parameters in the current iteration, before the next training.

Stragglers can influence the overall system throughput.

High communication cost that limits the system scalability.

Alleviate the straggler problem without losing synchronization.

Alleviate the straggler problem without losing synchronization.

The faster workers to do more updates than the slower workers to reduce the waiting time of the faster workers.

- **Alleviate the straggler** problem without losing synchronization.

- The **faster workers** to do **more updates** than the **slower workers** to reduce the waiting time of the faster workers.

- **Staleness bounded barrier** to limit the **iteration gap** between the fastest worker and the slowest worker.

For a maximum staleness bound $s$, the update formula of worker $i$ at iteration $t + 1$:

$$w_{i,t+1} := w_0 - \eta \left( \sum_{k=1}^{t} \sum_{j=1}^{n} G_{j,k} + \sum_{k=t-s}^{t} G_{i,k} + \sum_{(j,k) \in S_{i,t+1}} G_{j,k} \right)$$
For a maximum staleness bound $s$, the update formula of worker $i$ at iteration $t + 1$:

$$w_{i,t+1} := w_0 - \eta \left( \sum_{k=1}^{t} \sum_{j=1}^{n} G_{j,k} + \sum_{k=t-s}^{t} G_{i,k} + \sum_{(j,k) \in S_{i,t+1}} G_{j,k} \right)$$

The update has three parts:

For a maximum staleness bound $s$, the update formula of worker $i$ at iteration $t+1$:

$$w_{i, t+1} := w_0 - \eta \left( \sum_{k=1}^{t} \sum_{j=1}^{n} G_{j,k} + \sum_{k=t-s}^{t} G_{i,k} + \sum_{(j,k) \in S_{i, t+1}} G_{j,k} \right)$$

The update has three parts:

1. **Guaranteed pre-window updates** from clock 1 to $t$ over all workers.

For a maximum staleness bound $s$, the update formula of worker $i$ at iteration $t + 1$:

$$ w_{i,t+1} := w_0 - \eta \left( \sum_{k=1}^{t} \sum_{j=1}^{n} G_{j,k} + \sum_{k=t-s}^{t} G_{i,k} + \sum_{(j,k) \in S_{i,t+1}} G_{j,k} \right) $$

The update has three parts:

1. **Guaranteed pre-window updates** from clock 1 to $t$ over all workers.
2. **Guaranteed read-my-writes in-window updates** made by the querying worker $i$.

For a maximum staleness bound $s$, the update formula of worker $i$ at iteration $t + 1$:

$$w_{i,t+1} := w_0 - \eta \left( \sum_{k=1}^{t} \sum_{j=1}^{n} G_{j,k} + \sum_{k=t-s}^{t} G_{i,k} + \sum_{(j,k) \in S_{i,t+1}} G_{j,k} \right)$$

The update has three parts:

1. Guaranteed pre-window updates from clock 1 to $t$ over all workers.
2. Guaranteed read-my-writes in-window updates made by the querying worker $i$.
3. Best-effort in-window updates. $S_{i,t+1}$ is some subset of the updates from other workers during period $[t - s]$. 

It completely eliminates the synchronization.

It completely **eliminates the synchronization**.

Each work **transmits its gradients** to the PS **after it calculates the gradients**.

Communication Synchronization - Asynchronous (1/2)

- It completely **eliminates the synchronization**.

- Each work **transmits its gradients** to the PS **after it calculates the gradients**.

- The PS updates the global model **without waiting** for the other workers.

\[ w_{t+1} := w_t - \eta \sum_{i=1}^{n} g_{i,t-\tau_{k,i}} \]
\[ \mathbf{w}_{t+1} := \mathbf{w}_t - \eta \sum_{i=1}^{n} G_{i,t-\tau_{k,i}} \]

- \( \tau_{k,i} \) is the time delay between the moment when worker \( i \) calculates the gradient at the current iteration.

Communication Synchronization - Local SGD

- All workers run several iterations, and then averages all local models into the newest global model.

Communication Synchronization - Local SGD

- All workers run several iterations, and then averages all local models into the newest global model.

- If $I_T$ represents the synchronization timestamps, then:

$$w_{i,t+1} = \begin{cases} 
  w_{i,t} - \eta G_{i,t} & \text{if } t + 1 \notin I_T \\
  w_{i,t} - \eta \frac{1}{n} \sum_{i=1}^{n} G_{i,t} & \text{if } t + 1 \in I_T 
\end{cases}$$

Communication Compression
Communication Compression

- Reduce the communication traffic with little impact on the model convergence.
Communication Compression

- Reduce the communication traffic with little impact on the model convergence.
- Compress the exchanged gradients or models before transmitting across the network.
Communication Compression

- Reduce the communication traffic with little impact on the model convergence.
- Compress the exchanged gradients or models before transmitting across the network.
- Quantization
Communication Compression

- Reduce the communication traffic with little impact on the model convergence.
- Compress the exchanged gradients or models before transmitting across the network.
- Quantization
- Sparsification
Communication Compression - Quantization

- Using lower bits to represent the data.

Communication Compression - Quantization

- Using lower bits to represent the data.
- The gradients are of low precision.

Reduction the number of elements that are transmitted at each iteration.

Communication Compression - Sparsification

- Reducing the **number of elements** that are transmitted at each iteration.
- Only **significant gradients** are required to **update the model parameter** to **guarantee the convergence** of the training.

Communication Compression - Sparsification

- **Reducing** the **number of elements** that are transmitted at each iteration.
- Only **significant gradients** are required to **update the model parameter** to guarantee the convergence of the training.
- E.g., the **zero-valued** elements are no need to transmit.

Parallelism of Computations and Communications
- The *layer-wise structure* of deep models makes it possible to *parallels* the communication and computing tasks.
The layer-wise structure of deep models makes it possible to *parallels* the communication and computing tasks.

Optimizing the order of computation and communication such that the communication cost can be minimized.
Parallelism of Computations and Communications (2/3)

- Wait-free backward propagation (WFBP)
- Merged-gradient WFBP (MG-WFBP)

Parallelism of Computations and Communications (3/3)

Wait-free backward propagation (WFBP)
Merged-gradient WFBP (MG-WFBP)

[shi et al., MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms, 2018]
Parallelism of Computations and Communications (3/3)

Wait-free backward propagation (WFBP)
Parallelism of Computations and Communications (3/3)

Wait-free backward propagation (WFBP)

Merged-gradient WFBP (MG-WFBP)

[shi et al., MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms, 2018]
TicTac: Accelerating Distributed Deep Learning with Communication Scheduling
Computation vs. Communication

- The iteration time in deep learning systems depends on the time taken by
  1. Computation
  2. Communication
  3. The overlap between the two

[shi et al., MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms, 2018]
Computation vs. Communication

- The *iteration time* in deep learning systems depends on the time taken by
  1. Computation
  2. Communication
  3. The overlap between the two

- When workers receive the parameters from the PS at the beginning of each iteration, all parameters are not used simultaneously.

[shi et al., MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms, 2018]
Computation vs. Communication

- The iteration time in deep learning systems depends on the time taken by
  1. Computation
  2. Communication
  3. The overlap between the two

- When workers receive the parameters from the PS at the beginning of each iteration, all parameters are not used simultaneously.

- Identifying the best schedule of parameter transfers is critical for reducing the blocking on computation.

[shi et al., MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms, 2018]
Good vs. Bad Execution Order

(a) Toy Computational Graph

(b) Good Execution Order

(c) Bad Execution Order

[Hashemi et al., TicTac: Accelerating Distributed Deep Learning with Communication Scheduling, 2019]
High GPU Utilization

- High GPU utilization can be achieved in two ways:
  1. When total communication time is less than or equal to the computation time.
  2. With efficient overlap of communication and computation.

Techniques improve GPU utilization:
- Increasing computation time
- Decreasing communication time
- Better interleaving of computation and communication
High GPU utilization can be achieved in two ways:

1. When total communication time is less than or equal to the computation time.
High GPU Utilization

- High GPU utilization can be achieved in two ways:
  1. When total communication time is less than or equal to the computation time.
  2. With efficient overlap of communication and computation.
High GPU Utilization

- **High GPU utilization** can be achieved in two ways:
  1. When total **communication time** is less than or equal to the **computation time**.
  2. With efficient **overlap** of **communication** and **computation**.

- Techniques **improve GPU utilization**: 
High GPU Utilization

- High GPU utilization can be achieved in two ways:
  1. When total communication time is less than or equal to the computation time.
  2. With efficient overlap of communication and computation.

- Techniques improve GPU utilization:
  - Increasing computation time
High GPU Utilization

- High GPU utilization can be achieved in two ways:
  1. When total communication time is less than or equal to the computation time.
  2. With efficient overlap of communication and computation.

- Techniques improve GPU utilization:
  - Increasing computation time
  - Decreasing communication time
High GPU utilization can be achieved in two ways:

1. When total communication time is less than or equal to the computation time.
2. With efficient overlap of communication and computation.

Techniques improve GPU utilization:

- Increasing computation time
- Decreasing communication time
- Better interleaving of computation and communication
Overlap Coefficient (1/2)

- **Overlap coefficient:** \( \alpha = \frac{N + C - T}{\min(N, C)} \)

\( N \): the communication time

\( C \): the computation time

\( N + C \) is the iteration time when there is no overlap
Overlap Coefficient (1/2)

- **Overlap coefficient**: \( \alpha = \frac{N + C - T}{\min(N, C)} \)
- **T**: the actual iteration time
Overlap Coefficient (1/2)

- **Overlap coefficient**: \( \alpha = \frac{N+C-T}{\min(N,C)} \)
- **T**: the actual iteration time
- **N**: the communication time
Overlap Coefficient (1/2)

- **Overlap coefficient**: \( \alpha = \frac{N + C - T}{\min(N, C)} \)
- **T**: the actual iteration time
- **N**: the communication time
- **C**: the computation time

\( N + C \) is the iteration time when there is no overlap.
Overlap Coefficient (1/2)

- **Overlap coefficient:** $\alpha = \frac{N+C-T}{\min(N,C)}$
- $T$: the actual iteration time
- $N$: the communication time
- $C$: the computation time
- $N+C$ is the iteration time when there is no overlap
Overlap Coefficient (2/2)

- **Overlap coefficient:** \( \alpha = \frac{N+C-T}{\min(N,C)} \)

- The maximum overlap possible is given by \( \min(N,C) \), which is achieved when the smaller quantity completely overlaps with the large quantity.

- **GPU utilization:** 
  \[ U = \frac{C}{T} = \frac{C}{N+C-\alpha \min(N,C)} = \frac{1}{1+\rho-\alpha \min(\rho,1)} \] 

- \( \rho = \frac{N}{C} \): the communication/computation ratio
Overlap Coefficient (2/2)

- Overlap coefficient: \( \alpha = \frac{N+C-T}{\min(N,C)} \)

- The maximum overlap possible is given by \( \min(N,C) \), which is achieved when the smaller quantity completely overlaps with the large quantity.
Overlap Coefficient (2/2)

- **Overlap coefficient:** \( \alpha = \frac{N+C-T}{\min(N,C)} \)

- The maximum overlap possible is given by \( \min(N, C) \), which is achieved when the smaller quantity completely overlaps with the large quantity.

- **GPU utilization:** \( U = \frac{C}{T} = \frac{C}{N+C-\alpha \min(N,C)} = \frac{1}{1+\rho-\alpha \min(\rho,1)} \)

- \( \rho = \frac{N}{C} \): the communication/computation ratio
Scheduling Algorithm

- Prioritize transfers that speed up the critical path in the DAG, by reducing blocking on computation caused by parameter transfers.
Scheduling Algorithm

- Prioritize transfers that speed up the critical path in the DAG, by reducing blocking on computation caused by parameter transfers.

- **TIC**: Timing-Independent Communication scheduling
Scheduling Algorithm

- Prioritize transfers that speed up the **critical path** in the DAG, by **reducing blocking on computation** caused by **parameter transfers**.

- **TIC**: Timing-Independent Communication scheduling

- **TAC**: Timing-Aware Communication scheduling
Scheduling Algorithm

- Prioritize transfers that speed up the **critical path** in the DAG, by **reducing blocking on computation** caused by **parameter transfers**.

- **TIC**: Timing-Independent Communication scheduling
  - Prioritize those transfers that **reduce** blocking on **network transfers**.

- **TAC**: Timing-Aware Communication scheduling
Scheduling Algorithm

- Prioritize transfers that speed up the critical path in the DAG, by reducing blocking on computation caused by parameter transfers.

- **TIC**: Timing-Independent Communication scheduling
  - Prioritize those transfers that reduces blocking on network transfers.

- **TAC**: Timing-Aware Communication scheduling
  - Prioritize those transfers that reduces the blocking of computation.
Prioritize those transfers that reduces blocking on network transfers.
Prioritize those transfers that reduces blocking on network transfers.

Prioritize based only on vertex dependencies in the DAG.
Prioritize those transfers that reduces blocking on network transfers.

Prioritize based only on vertex dependencies in the DAG.

Higher priorities are given to transfers that are least blocking on computation.
Prioritize those transfers that reduces blocking on network transfers.

Prioritize based only on vertex dependencies in the DAG.

Higher priorities are given to transfers that are least blocking on computation.

Ignore the ops time, and use the number of communication ops instead.
Prioritize those transfers that reduces blocking on network transfers.

Prioritize based only on vertex dependencies in the DAG.

Higher priorities are given to transfers that are least blocking on computation.

Ignore the ops time, and use the number of communication ops instead.

E.g., $\text{op}_1.M = \text{Time}(\text{recv}_1)$ and $\text{op}_2.M = \text{Time}(\text{recv}_1) + \text{Time}(\text{recv}_2)$. 
Prioritize those transfers that reduces the blocking of computation.
Prioritize those transfers that reduces the blocking of computation.

Prioritize transfers that maximize $\alpha$ by using information on:

1. Vertex dependencies among ops specified by the computational DAG.
2. Execution time of each op estimated with time oracle.
Prioritize those transfers that reduces the blocking of computation.

Prioritize transfers that maximize $\alpha$ by using information on:

1. Vertex dependencies among ops specified by the computational DAG.
Prioritize those transfers that reduces the blocking of computation.

Prioritize transfers that maximize $\alpha$ by using information on:

1. Vertex dependencies among ops specified by the computational DAG.
2. Execution time of each op estimated with time oracle.
Prioritize those transfers that reduces the blocking of computation.

Prioritize transfers that maximize $\alpha$ by using information on:

1. Vertex dependencies among ops specified by the computational DAG.
2. Execution time of each op estimated with time oracle.

To achieve this goal, the algorithm focuses on two cases:

1. Any communication and computation overlapping?
2. If no, choose one which eliminates the computation block sooner.
CARAMEL: Accelerating Decentralized Distributed Deep Learning with Computation Scheduling
Decentralized aggregation (no PS)
Decentralized aggregation (no PS)

Improve efficiency of decentralized DNN training
CARAMEL

- Decentralized aggregation (no PS)
- Improve efficiency of decentralized DNN training
- In terms of iteration time and GPU utilization
CARAMEL

- Decentralized aggregation (no PS)
- Improve efficiency of decentralized DNN training
- In terms of iteration time and GPU utilization
- CARAMEL achieves this goal through:
CARAMEL

- Decentralized aggregation (no PS)
- Improve efficiency of decentralized DNN training
- In terms of iteration time and GPU utilization
- CARAMEL achieves this goal through:
  1. Computation scheduling that expands the feasible window of transfer for each parameter (transfer boundaries)
CARAMEL

- Decentralized aggregation (no PS)
- Improve efficiency of decentralized DNN training
- In terms of iteration time and GPU utilization
- CARAMEL achieves this goal through:
  1. Computation scheduling that expands the feasible window of transfer for each parameter (transfer boundaries)
  2. Network optimizations that smoothen the load
In decentralized aggregation, all workers should have the parameter available for aggregation before the transfer can be initiated.
In decentralized aggregation, all workers should have the parameter available for aggregation before the transfer can be initiated.

- There are multiple feasible orders for executing operations in a DAG.
In decentralized aggregation, all workers should have the parameter available for aggregation before the transfer can be initiated.

There are multiple feasible orders for executing operations in a DAG.

The parameters may become available at different workers in varying orders.
In decentralized aggregation, all workers should have the parameter available for aggregation before the transfer can be initiated.

There are multiple feasible orders for executing operations in a DAG.

The parameters may become available at different workers in varying orders.

The transfer boundaries of a parameter represent the window when a parameter can be aggregated without blocking computation.
Computation Scheduling (1/2)

- In decentralized aggregation, all workers should have the parameter available for aggregation before the transfer can be initiated.

- There are multiple feasible orders for executing operations in a DAG.

- The parameters may become available at different workers in varying orders.

- The transfer boundaries of a parameter represent the window when a parameter can be aggregated without blocking computation.

- The start boundary is determined by the completion of the computation operation that updates the parameter.
In decentralized aggregation, all workers should have the parameter available for aggregation before the transfer can be initiated.

There are multiple feasible orders for executing operations in a DAG.

The parameters may become available at different workers in varying orders.

The transfer boundaries of a parameter represent the window when a parameter can be aggregated without blocking computation.

The start boundary is determined by the completion of the computation operation that updates the parameter.

The end boundary is the computation operation that reads the parameter.
CARAMEL expands these boundaries through scheduling optimizations of the computation DAG, where it

- Moves the start boundaries earlier.
- Pushes the end boundary by postponing the execution of some computation operations to the forward pass of the next iteration.

[Hashemi et al., CARAMEL: Accelerating Decentralized Distributed Deep Learning with Model-Aware Scheduling, 2020]
CARAMEL expands these boundaries through scheduling optimizations of the computation DAG, where it

1. Moves the start boundaries earlier.

[Hashemi et al., CARAMEL: Accelerating Decentralized Distributed Deep Learning with Model-Aware Scheduling, 2020]
CARAMEL expands these boundaries through scheduling optimizations of the computation DAG, where it

1. Moves the start boundaries earlier.
2. Pushes the end boundary by postponing the execution of some computation operations to the forward pass of next iteration.

[Hashemi et al., CARAMEL: Accelerating Decentralized Distributed Deep Learning with Model-Aware Scheduling, 2020]
Optimizations for *smoothening the network load* include:
Network Optimization

- Optimizations for *smoothening the network load* include:
  1. Batching of *small parameters* to reduce the network overhead.
Network Optimization

- Optimizations for *smoothening the network load* include:
  1. **Batching** of small parameters to reduce the network overhead.
  2. Adaptive **splitting and pipelining of parameters** to *accelerate aggregation* of large data.
Defining the Environment

- **T**: the actual iteration time
- **N**: the communication time
- **C**: the computation time

Overlap coefficient:
\[ \alpha = N + C - T \min(N, C) \]

GPU utilization:
\[ U = \frac{C}{T} = \frac{C}{N} + \frac{C}{\alpha} \min(N, C) \]
\[ \rho = \frac{N}{C} \]: the communication/computation ratio

\[ \rho = \frac{1}{1 + \rho - \alpha \min(\rho, 1)} \]
Defining the Environment

- $T$: the actual iteration time
- $N$: the communication time
- $C$: the computation time
- Overlap coefficient: $\alpha = \frac{N+C-T}{\min(N,C)}$
Defining the Environment

- **T**: the actual iteration time
- **N**: the communication time
- **C**: the computation time
- **Overlap coefficient**: $\alpha = \frac{N+C-T}{\min(N,C)}$
- **GPU utilization**: $U = \frac{C}{T} = \frac{C}{N+C-\alpha \min(N,C)} = \frac{1}{1+\rho-\alpha \min(\rho,1)}$
- $\rho = \frac{N}{C}$: the communication/computation ratio
CARAMEL Algorithm

- Dataflow DAG Optimizer
- Network Transfer Scheduler
- Parameter Batcher
- Adaptive Depth Enforcer
Stage 1: Determining the best order.

- Increasing the overlap coefficient $\alpha$ by prioritizing the computations that activate the communication operations as early as possible.

Stage 2: Enforcing the best order.

- Iteratively activate parameters in the best order chosen in the previous stage.
- Ensureing that at each given time, only ops needed for the target parameter update can be executed.
Stage 1: Determining the best order.

Stage 2: Enforcing the best order.
Stage 1: Determining the best order.
- Increasing the overlap coefficient $\alpha$ by prioritizing the computations that activates the communication operations as early as possible.

Stage 2: Enforcing the best order.
Stage 1: Determining the best order.
- Increasing the overlap coefficient $\alpha$ by prioritizing the computations that activates the communication operations as early as possible.

Stage 2: Enforcing the best order.
- Iteratively activate parameters in the best order chosen in the previous stage.
Stage 1: Determining the best order.
- Increasing the overlap coefficient $\alpha$ by prioritizing the computations that activates the communication operations as early as possible.

Stage 2: Enforcing the best order.
- Iteratively activate parameters in the best order chosen in the previous stage.
- Ensureing that at each given time, only ops needed for the target parameter update can be executed.
Network Transfer Scheduler

- **Increasing** the overlap coefficient $\alpha$ by **scheduling parameter transfers** efficiently.
- Transfers are scheduled in both **backward pass** and **forward pass**.

(a) Parameter Server  
(b) Decentralized aggregation

[Hashemi et al., CARAMEL: Accelerating Decentralized Distributed Deep Learning with Model-Aware Scheduling, 2020]
Small parameters incur large overhead.
Parameter Batcher

- Small parameters incur large overhead.

- Combining small parameters into groups.
Parameter Batcher

- Small parameters incur large overhead.
- Combining small parameters into groups.
- Parameters larger than a certain threshold are transferred without batching.
Adaptive Depth Enforcer

- Two stages in decentralized algorithms: **transferring** and **aggregating** data across nodes.
Two stages in decentralized algorithms: transferring and aggregating data across nodes.

- In each step, data is transferred on the network, and is sent to application to be reduced, before the result is sent again over the network.
Two stages in decentralized algorithms: transferring and aggregating data across nodes.

- In each step, data is transferred on the network, and is sent to application to be reduced, before the result is sent again over the network.

This process reduces the network utilization since the network is not utilized during the reduction at the application layer.
Adaptive Depth Enforcer

- Two stages in decentralized algorithms: transferring and aggregating data across nodes.
  - In each step, data is transferred on the network, and is sent to application to be reduced, before the result is sent again over the network.
- This process reduces the network utilization since the network is not utilized during the reduction at the application layer.
- Chunk (break) the data in to a few pieces, and transfer each chunk independently in parallel.
Two stages in decentralized algorithms: transferring and aggregating data across nodes.

- In each step, data is transferred on the network, and is sent to application to be reduced, before the result is sent again over the network.

This process reduces the network utilization since the network is not utilized during the reduction at the application layer.

- Chunk (break) the data in to a few pieces, and transfer each chunk independently in parallel.

- While one chunk is being reduced on the CPU, another chunk can be sent over the network: this enables pipelining of network transfer and application-level processing across various chunks.
Summary
Summary

- Data-parallel
- The aggregation algorithm
- Communication synchronization
- Communication compression
- Parallelism of computations and communications
- TicTac
- Caramel
Reference

- Tang et al., Communication-Efficient Distributed Deep Learning: A Comprehensive Survey, 2020
- Hashemi et al., TicTac: Accelerating Distributed Deep Learning with Communication Scheduling, 2019
- Hashemi et al., CARAMEL: Accelerating Decentralized Distributed Deep Learning with Model-Aware Scheduling, 2020
Questions?