大規模計算論
“High Performance Parallel Stochastic Gradient Descent in Shared Memory”

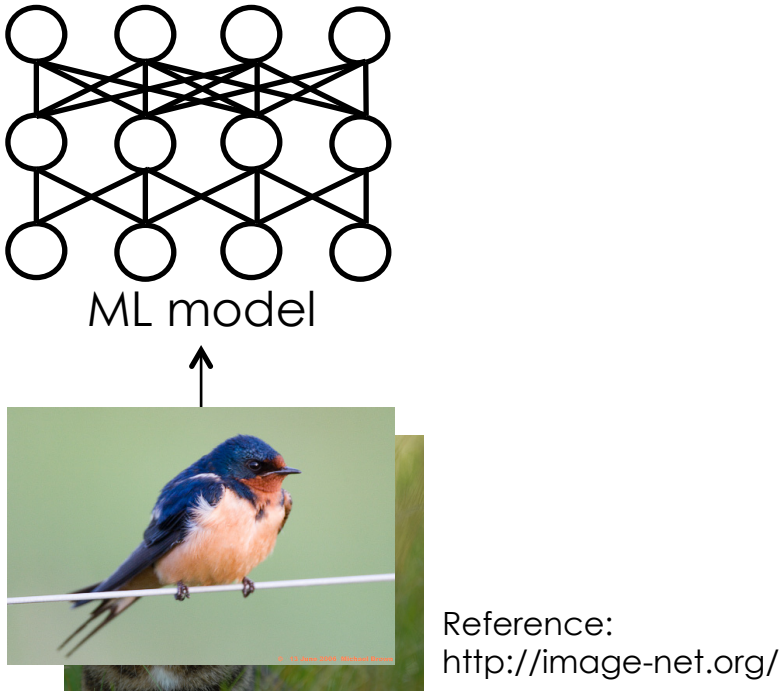
2016/10/04

東京工業大学 情報理工学院 修士1年

大山洋介

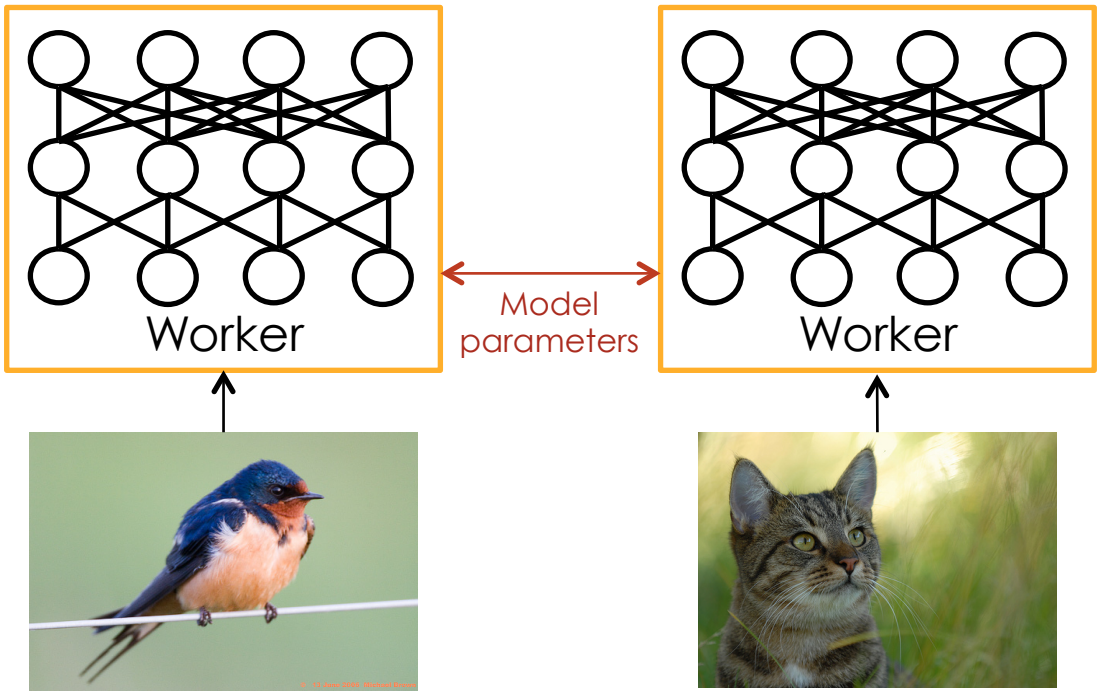
Data Parallel and Model Parallel

- Machine learning can be parallelized in several ways
  - Parallelizing data samples
  - Parallelizing ML model



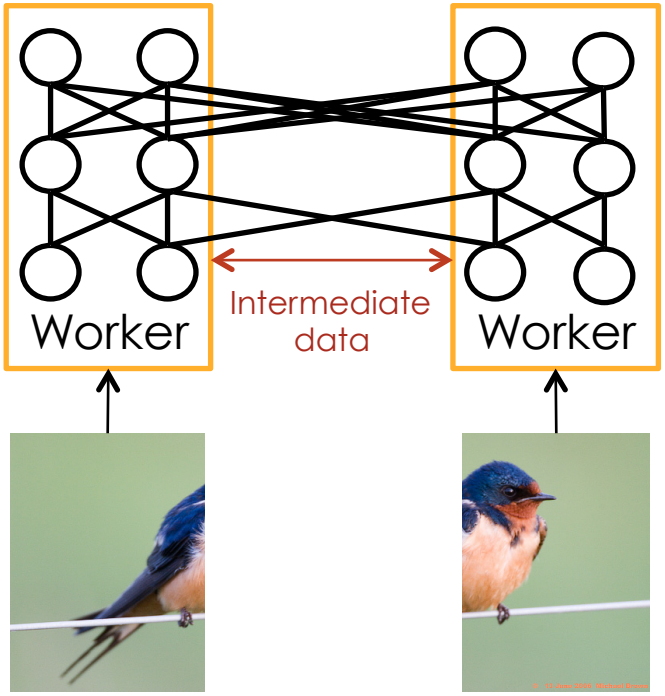
Data Parallel and Model Parallel

- **Data parallel**: Parallelizing data samples with multiple workers
  - Model parameters are synchronized among workers
  - Parameters can be managed by dedicated workers (parameter servers)



Data Parallel and Model Parallel

- **Model parallel**: Parallelizing ML model with multiple workers
  - Intermediate data might be communicated among workers



Selected Papers
STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning

- J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. **STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning**. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, pages 5:1?5:16, New York, NY, USA, 2016. ACM.
  - Model parallelism solves these problems that data parallelism doesn't
    - Naïve concurrent updates violate dependency across parameters
    - Parameters converge at different rates
  - The authors propose **Scheduled Model Parallelism (SchMP)** and its framework **STRADS**
    - SchMP LDA topic modeling and Lasso achieved 10x and 5x faster convergence than recent baselines

Selected Papers

STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning

- The user implements `schedule()`, `update()` and `aggregate()`
  - `schedule()` select parameters to update
    - Approximate graph partitioning algorithm can be implemented to solve the uneven convergence rate problem
  - `update()` compute intermediate result to update the model
  - `aggregate()` collect the intermediate result and update the model

Algorithm 2 SchMP Dynamic, Prioritized Lasso

$\mathbf{X}, \mathbf{y}$: input data
$\{\mathbf{X}\}^p, \{\mathbf{y}\}^p$: rows/samples of $\mathbf{X}, \mathbf{y}$ stored at worker p
β: model parameters (regression coefficients)
λ: ℓ_1 regularization penalty
τ: $\mathcal{G}$ edges whose weight is below τ are ignored

Function `schedule`$(\beta, \mathbf{X})$:
  Pick $L > P$ params in β with probability $\propto (\Delta\beta_a)^2$
  Build dependency graph $\mathcal{G}$ over L chosen params:
    edge weight of (β_a, β_b) = correlation$(\mathbf{x}^a, \mathbf{x}^b)$
  $[\beta_{\mathcal{G}_1}, \ldots, \beta_{\mathcal{G}_K}]$ = findIndepNodeSet$(\mathcal{G}, \tau)$
  For $p = 1..P$:
    $\mathbf{S}_p = [\beta_{\mathcal{G}_1}, \ldots, \beta_{\mathcal{G}_K}]$
  Return $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$

Function `update`$(p, \mathbf{S}_p, \{\mathbf{X}\}^p, \{\mathbf{y}\}^p, \beta)$:
  For each param β_a in $\mathbf{S}_p$, each row i in $\{\mathbf{X}\}^p$:
    $R_p[a]$ += $x_a^i y^i - \sum_{b \neq a} x_a^i x_b^i \beta_b$
  Return R_p

Function `aggregate`$([R_1, \ldots, R_P], \mathbf{S}_1, \beta)$:
  For each parameter β_a in $\mathbf{S}_1$:
    temp = $\sum_{p=1}^{P} R_p[a]$
    $\beta_a = \mathcal{S}(\text{temp}, \lambda)$

Selected Papers

GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server

- H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. **GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server**. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, pages 4:1?4:16, New York, NY, USA, 2016. ACM.
  - Training DNN on large number of GPUs is insufficient due to data movement overhead, GPU stalls, and limited GPU memory
  - The authors proposed **GeePS**, a parameter server implementation for distributed deep learning
    - GeePS manages the location of DNN parameters and local data(such as Input data and intermediate data) in background
    - GeePS achieved 13x speedup on 16 GPUs
    - GeePS achieved higher throughput on four GPUs than 108 CPU-only machines

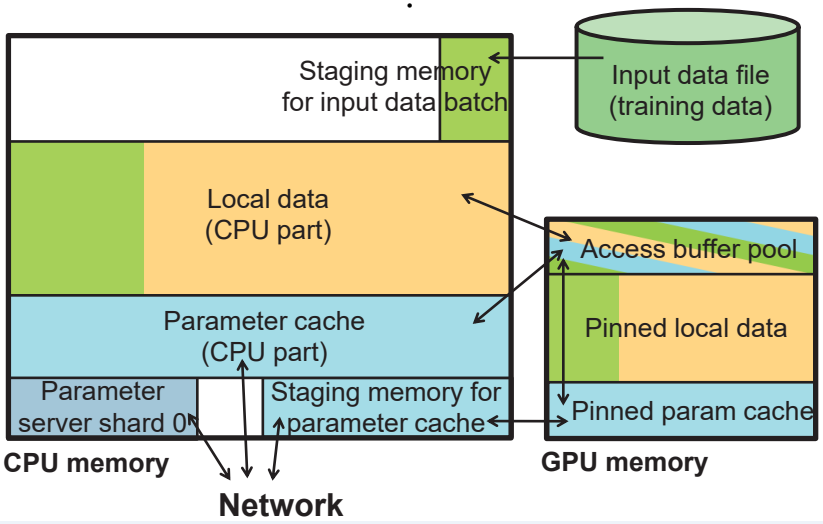
Selected Papers

GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server

1. GeePS collect access information of buffers on GPU memory
   - Since training DNN is iterative, these access pattern is static
2. A GeePS thread performs CPU-GPU data movement in background, based on the collected access information
   - If all data don't fit in GPU memory, GeePS utilize buffer pool to swap buffers between CPU and GPU dynamically

function TRAINMINIBATCH($trainData$, $virtual$)
  # Forward pass
  for $i = 0 \sim (L-1)$ **do**
    $paramDataPtr \leftarrow$
     $geeps.Read(paramDataKeys_i, virtual)$
    $localDataPtr \leftarrow$
     $geeps.LocalAccess(localDataKeys_i, virtual)$
    if not $virtual$ **then**
      Setup $layer_i$ with data pointers
      Forward computation of $layer_i$
    end if
    $geeps.PostRead(paramDataPtr)$
    $geeps.PostLocalAccess(localDataPtr)$
  end for

⋮



Selected Papers

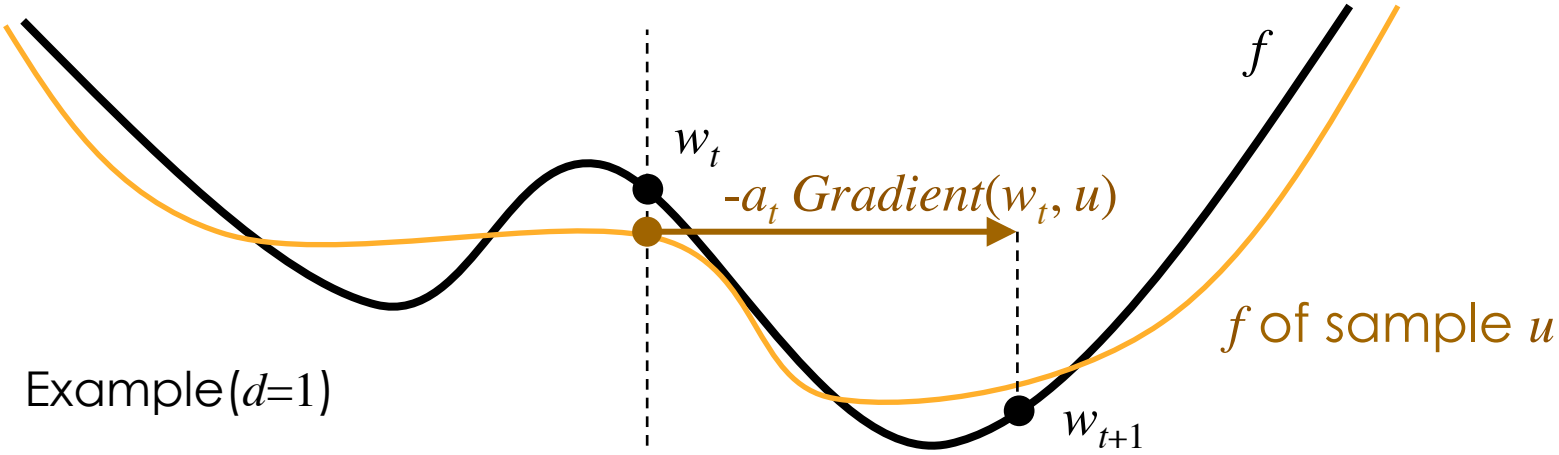
- S. Sallinen, N. Satish, M. Smelyanskiy, S. Sury, C. Re. **High Performance Parallel Stochastic Gradient Descent in Shared Memory**. IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2016.
  - **Stochastic Gradient Descent (SGD)** is a popular optimization method used to train machine learning models
  - Existing parallel SGD implementations may reduce hardware efficiency and/or statistical efficiency as scale
  - The authors proposed a new, scalable, communication-avoiding, many-core friendly implementation of SGD, **HogBatch**
    - HogBatch is a combination of **Hogwild** and **mini-batching**

Introduction
Stochastic Gradient Descent (SGD)

- Stochastic Gradient Descent (SGD)
  - A popular optimization method to train machine learning models

$$w_{t+1} = w_t - a_t \; Gradient(w_t, u)$$

  - $w_t \in \mathbb{R}^d$: d-features model
  - $a_t > 0$: learning rate
  - $U = (u_1 u_2 \cdots u_n)^{\mathrm{T}} \in \mathbb{R}^{n \times d}$: n-samples d-features dataset
  - $Gradient(w_t, u)$: gradient of objective function f of sample u at w_t



Introduction
Stochastic Gradient Descent (SGD)

- SGD requires less computational cost to update than other traditional approaches

  - Machine learning problems typically do not require updating with very high accuracy
  - cf. Interior-Point Method, Newton Method

- SGD is inherently sequential with dependency across iterations

  - Some variants exposes extra parallelism, which come at the loss of **statistical** and **hardware** efficiency
    - Statistical: the number of iterations to converge is increased
    - Hardware: the amount of inter-core communication and cache miss is increased

Introduction
Stochastic Gradient Descent (SGD)

- SGD can be modified in a variety of ways
  1. Data access and parallelization strategy
     - **Hogwild**
     - **Mini-batching**
  2. Objective function (loss function)
     - Linear, logistic, hinge loss, least squares, …
  3. How to compute the gradient and fix learning rates
     - ADAGRAD tunes learning rate automatically
     - Stochastic Average Gradient (SAG) uses an average gradient to do model updates

- **This paper focuses on 1., the fundamental algorithm that affect hardware efficiency**

Parallelizing SGD

- Parallelizing across…
  - **Features**: Since the problem is typically sparse, there are small amounts of parallelism
  - **Non-zero features**: Since elements of w is written randomly in parallel, significant inter-core traffic happen to maintain cache coherence
  - **Samples**: Updates is computed with **stale** w, which may degrade statistical efficiency
    - **Staleness**: the number of updates to the global model that happen between
      - "the time the model is read by a thread" to
      - "the time the model update is written back by the thread"
      - Sequential SGD always provides zero staleness
    - **Hogwild**, **Mini-Batching** and **HogBatch**

Parallelizing SGD
Mini-Batching

- S samples (batch) are combined to do one model update

$$w_{t+S} = w_t - a_t \sum_{b=u}^{u+S} Gradient(w_t, b)$$

  - The batch can be divided across threads
    1. Each thread update its private gradient vector for the part of the batch independently
    2. Threads update the global model synchronously
  - Mini-batching breaks the sequential semantics of SGD
    - Each gradient of sample $u+i$ uses w_t instead of w_{t+i}
    - This affects the statistical efficiency

Parallelizing SGD
Mini-Batching

Period to process all samples in the dataset

Algorithm 1: Mini-Batch SGD pseudocode for one datapass

```
1  for (st = 0; st < num_samples/SIZE; st += SIZE) {
2    #pragma omp parallel for schedule(dynamic)
3    for (index = st; index < SIZE; index++) {
4      // Sparse vector operation.
5      g_tid[TID] += a * Gradient(model, index);
6    } // (implicit thread barrier)
7
8    #pragma omp parallel for schedule(static)
9    for (f = 0; f < num_features; f++) {
10     for (t = 0; t < NUM_THREADS; t++)
11       model[f] = model[f] - g_tid[t][f];
12   } // (implicit thread barrier)
13 }
```

private vector

Thread ID

global vector

Dense vector operation

Parallelizing SGD
Mini-Batching

- Pros
  - **One update per batch size**: Inter-core traffic is reduced
  - **Thread independent tasks**: Irregular access of the sparse vector operation is totally private
- Cons
  - **Reduction**: All threads need to reduce their private gradients to do update
  - **Thread synchronization**: Threads have to synchronize before/after the reduction
  - **Updates are stale**: The updates within the batch become increasingly stale

Parallelizing SGD
Hogwild

- Each thread perform their own asynchronous model updates
  - Although data race conditions may occur, Hogwild works well for very sparse datasets
    - In sparse datasets, many samples has non-zero elements on mostly different indices

Algorithm 2: Hogwild SGD pseudocode for one datapass

```
1  #pragma omp parallel for schedule(dynamic)
2  for (index = 0; index < num_samples; index++) {
3    // Sparse vector operation.
4    model = model - a * Gradient(model, index);
5  }
```

Parallelizing SGD
Hogwild

- Pros
  - **Thread asynchronicity**: Threads do not have to synchronize
  - **Minimum staleness**: Threads compute gradient with the current model visible to the thread at that time
- Cons
  - **Race conditions**: It is quite possible that parts of the update can be lost due to race condition if the problem is not so sparse
  - **Inter-core communication**: High cross-core traffic occurs to keep cache coherence
    - In the authors' experiments, core-to-core communication alone could consume up to 60% of the execution cycles

Hogwild + Mini-Batching: HogBatching

- Each thread process one batch (Mini-Batching), and perform asynchronous model update (Hogwild)
  - In HogBatching, write to `model` is dense
    - `g_tid[TID]` is more dense than each gradient after the aggregation
    - Although the write invalidate cache lines, many new values are written per one invalidation

Algorithm 3: HogBatching SGD pseudocode for one datapass

```
1   #pragma omp parallel for schedule(dynamic)
2   for (st = 0; st < num_samples/SIZE; st += SIZE) {
3     for (index = st; index < SIZE; index++) {
4       // Sparse vector operation.
5       g_tid[TID] += a * Gradient(model, index);
6     }
7
8     for (f = 0; f < num_features; f++)
9       model[f] = model[f] - g_tid[TID][f];
10  }
```

Hogwild + Mini-Batching: HogBatching

- Pros
  - **Thread asynchronicity**: As in Hogwild, threads do not have to synchronize
  - **Thread independent tasks**: As in Mini-Batching, threads has their own independent subset of samples to process
  - **Reduced staleness**: Staleness may be less than Mini-Batching, since other threads may update the global model in the middle of batch processing
- Cons
  - **Race conditions**
  - **Inter-core communication**
  - However, these issues are drastically reduced than Hogwild because threads have to write the global model per batch

Hogwild + Mini-Batching: HogBatching

	Serial	Mini-Batch	Hogwild	HogBatch
Parallelism	×	✓	✓	✓
#update/#sample	× 1	✓ less than 1	× 1	✓ less than 1
Inter-core communication	-	✓	×	△
Staleness	✓ 0	×	✓	△
Model update	Sparse	Dense	Sparse	Dense

Hardware

Statistical

Staleness Properties

- For Hogwild and HogBatch, the max-stale of the last sample of a batch increases as other threads update the model asynchronously
  - It is assumed that one model update takes the same amount of time
- For Mini-Batch and HogBatch, the stale of the last sample is always more than the batch size

TABLE1: Staleness Analysis of the last sample of a batch
T: #thread, S: Mini-Batch size, HS: HogBatch size

Method	Min-Stale (For final update in batch)	Max-Stale	Example: T=8, S=1024, HS=(S/T) [min, max]
Hogwild	0	(T-1)	[0, 7]
Mini-Batch	S	S	[1024, 1024]
HogBatch	HS	(T*HS)	[128, 1024]

Staleness Properties
Improving Staleness

- For Mini-Batch and HogBatch, staleness can be reduced if each thread use `model-g_tid[TID]` instead of `model` to compute gradient
  - Intuitively each thread updates its local model
  - This causes a significant improvement on statistical efficiency
    - In their experiments, time to convergence is improved up to 30%, especially for denser problems

TABLE II: Improved Staleness Analysis

Method	Min-Stale (For final update in batch)	Max-Stale	Example: T=8, S=1024, HS=(S/T) [min, max]
Hogwild	0	(T-1)	[0, 7]
Mini-Batch	(S) - (S/T)	(S) - (S/T)	[896, 896]
HogBatch	(HS) - (HS) = 0	(T*HS) - (HS)	[0, 896]

Implementation Options with HogBatching

- HS of HogBatching can be larger than S/T of Mini-Batching
  - It is because minimum staleness for HogBatching improves statistical efficiency
  - Experiments showed that the optimal HS can lie between S/T to S
- Model updates (`g_tid[TID]`) can be treated as a sparse format in either of two ways
  - Holding a bitmap of non-zero indices, and performing a bit scan to get the indices
  - Using a map data structure of the indices
  - Experiments showed that these strategies was only useful for extremely sparse problems

Implementation Options with HogBatching

- SMT-aware hierarchical parallelism further improve performance
  - Two threads on a core share a private gradient vector (`g_group`)
    - This reduces cache pressure of the core

Algorithm 4: Many-Core HogBatch SGD pseudocode, for one datapass

```
1   GROUP_START = get_group_start(TID);
2   // Group parallel asynchronous
3   for (st = GROUP_START; st < GROUP_COUNT; st++) {
4     WORK_START = get_worker_start(st, TID);
5     // Worker parallel asynchronous
6     for (id = WORK_START; id < WORK_COUNT; id++) {
7       // Sparse vector operation.
8       g_group += a * Gradient((model - g_group), id);
9     }
10    sync();  // Sync threads in the group.
11
12    // All threads in the group perform the update.
13    for (f = TID; f < num_features; f += T_PER_GROUP)
14      model[f] = model[f] - g_group[f];
15  }
```

Shared by threads in a core

Experimental Analysis
Experimental Setup

- Hardware
  - Intel Xeon E5-2697 v3 Haswell @ 2.6 GHz
    - 14 cores (28 threads including SMT)
  - 64 GB RAM
  - Red Hat Enterprise Linux Server release 6.5
- Software
  - Intel C++ Compiler 15.0.2, parallelized by OpenMP
  - All values are in single precision format
  - Logistic regression loss is used for SGD algorithm

$$\sum_{u \in U} \log(1 + \exp(-y_u p_u))$$

Label of sample u $\in \{-1, 1\}$

Dot product of model w and sample u

Experimental Analysis
Experimental Setup

- Datasets
  - Seven binary-labeled datasets with varying feature size and sparsity patterns

TABLE III: Experiment Dataset

NNZ/ (Examples*Features)

Sparse ↑ Dense

Dataset Name	Examples	Features	NNZ	Sparse%	NNZ/Row	Avg/Row
news20.binary	19,996	1,355,191	9,097,916	0.034	1 to 16,423	454.987
RCV1-v2	781,265	276,544	60,534,218	0.028	4 to 1,585	77.482
RCV1-v1-test	677,399	47,236	49,556,258	0.155	4 to 1,224	73.157
real-sim	72,309	20,958	3,709,083	0.245	1 to 3,484	51.295
w8a	64,700	300	753,862	3.884	1 to 114	11.652
connect4	67,557	126	2,837,394	33.333	42 to 42	42.000
covtype	581,012	54	6,940,438	22.121	9 to 12	11.945

First 5 samples of news20.binary

```
-1  1:0.016563  2:0.016563  3:0.016563   4:0.016563 …
-1  1:0.013067  2:0.013067  3:0.013067   5:0.013067 …
-1 40:0.028421 54:0.028421 75:0.028421  81:0.028421 …
-1 40:0.048057 57:0.048057 75:0.048057  97:0.048057 …
-1 40:0.084515 75:0.084515 97:0.084515 103:0.084515 …
```

Experimental Analysis
Experimental Setup

- Reporting
  - Training time until it achieves a *closeness of "optimal" loss* are reported
    - $(2\text{-}|current|/|optimal|)*100\%$
    - Unless otherwise specified, it is 99.5%
    - Time to I/O is ignored
  - The "optimal" loss are computed with L-BFGS, a second order optimization method
- Parameters
  - Learning rate and batch size are sweeped and only the best result is presented
  - The learning rate per iteration is adjusted as $alpha/\text{sqrt}(\#iteration)$
- Regularization
  - L2 regularization with the Lambda value $1/\#samples$ are applied for all methods

Experimental Analysis
Results of our Evaluation

- In most datasets, HogBatch is the best solution in terms of time-to-convergence
  - Hogwild is the best alternative in sparser datasets
  - Hogwild beats HogBatch in news20.binary
    - Extremely less write conflict/false sharing due to high sparsity
    - HogBatch has global model update overhead in dense format

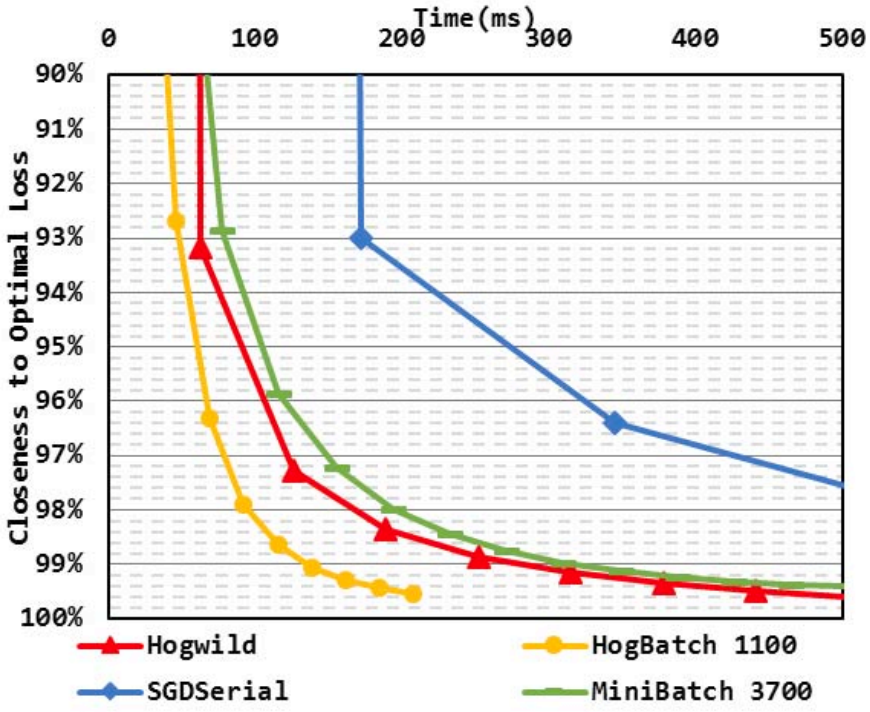
TABLE IV: Speedup (as time to 99.5% convergence) of HogBatch over best alternative solution out of Serial, Mini-Batching, Hogwild on a 14 core system

Sparse ↑ Dense

Dataset	Sparse%	Features	Best Alt	vs Best Alt
news20.binary	0.034	1,355,191	Hogwild	0.86x
RCV1-v2	0.028	276,544	Hogwild	1.87x
RCV1-test	0.155	47,236	Hogwild	2.43x
real-sim	0.245	20,958	Hogwild	3.85x
w8a	3.884	300	Hogwild	8.97x
connect4	33.333	126	Mini-Batch	5.81x
covtype	22.121	54	Serial	20.16x

Experimental Analysis
Results of our Evaluation

- In RCV1 (0.155% sparsity),
  - **Hogwild** showed similar convergence behavior to **Serial** in terms of loss-per-pass
  - **Mini-Batching** is worse than **Hogwild** in terms of loss-per-pass
    - In time-per-pass it is near parity because it is twice faster
  - **HogBatching** took the advantage of both

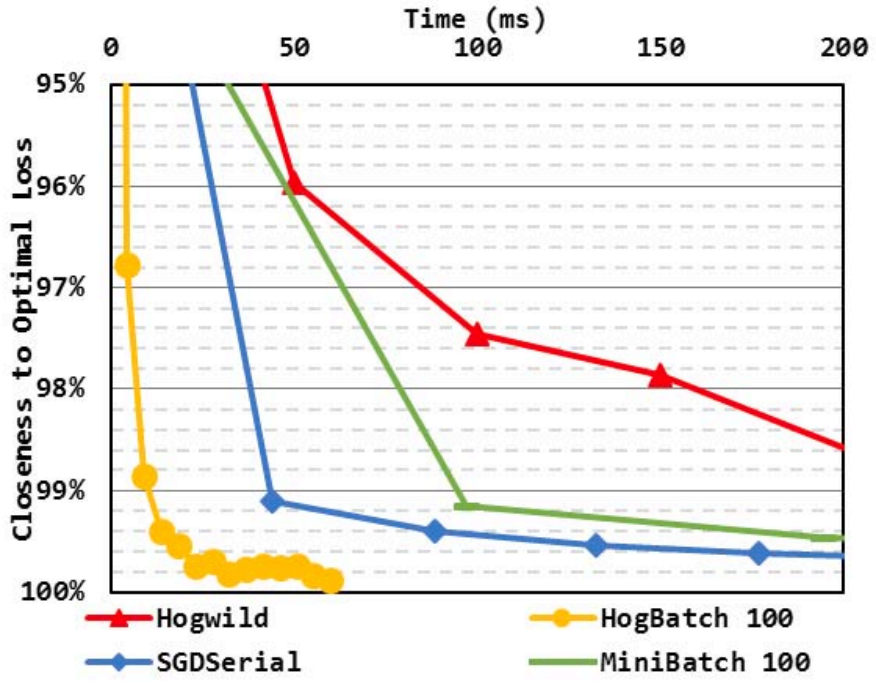


(a) RCV1-v1-test

Fig. 1: Closeness to the optimal solution over time. Each point represents a dataset pass.

Experimental Analysis
Results of our Evaluation

- In covtype (22.121% sparsity),
  - **Hogwild** is slower than **Serial**, due to low hardware efficiency
  - **Mini-Batching** is slower than **Serial**, due to low statistical efficiency
  - **HogBatch** scaled near-linearly

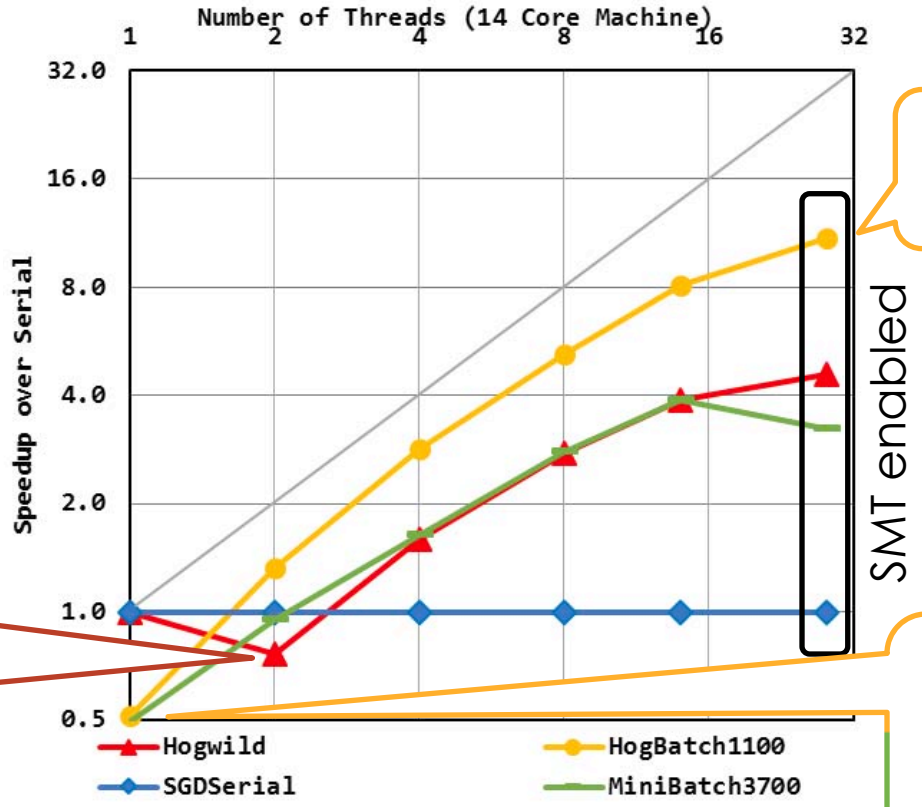


(b) covtype

Fig. 1: Closeness to the optimal solution over time. Each point represents a dataset pass.

Experimental Analysis
Scaling with Cares

- In RCV1 (0.155% sparsity, 47,236 features)



11x than Serial
>2x than Hogwild

Inter-core communication is more severe than parallelism

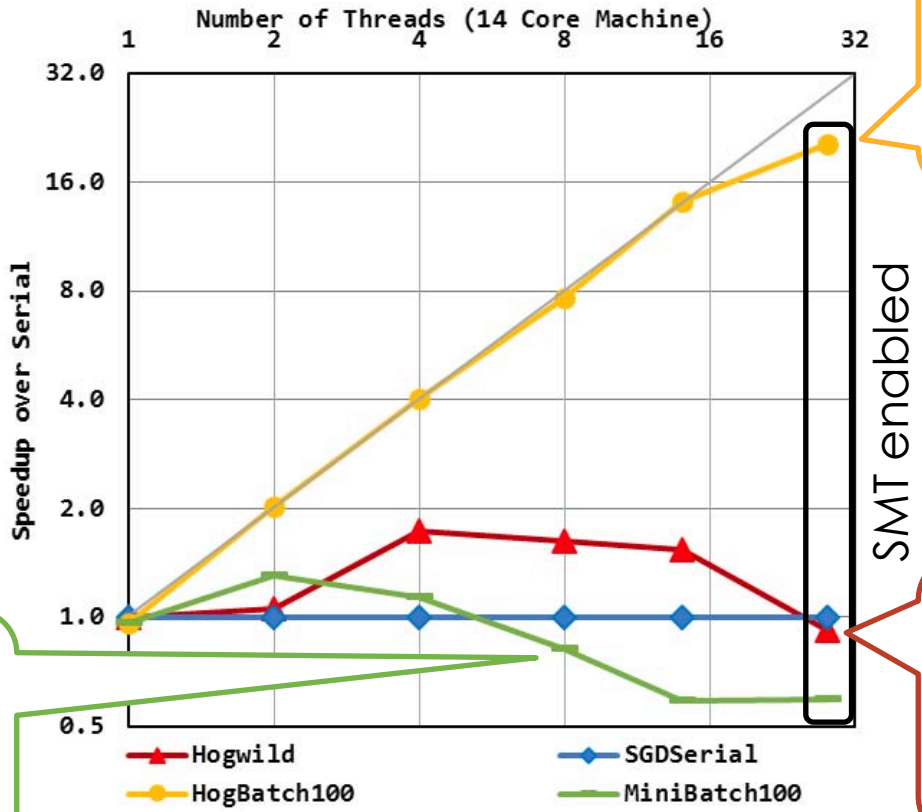
Overhead due to gradient aggregation on large #features

(a) RCV1-v1-test

Fig. 2: Speedup over serial, in time-to-convergence

Experimental Analysis
Scaling with Cares

- In covtype (22.121% sparsity, 54 features)



Super-linear scaling (20x) due to cache improvement

Batch size=100 is too small to parallelize; larger batch leads slower convergence

Inter-core communications and write conflict hurts performance

(b) convtype

Fig. 2: Speedup over serial, in time-to-convergence

Experimental Analysis
Scaling with Frequency

- Hogwild prefers frequency to #core
  - The frequency of core interconnect is governed by core frequency
- Mini-Batch and HogBatch prefers #core as well as frequency
  - This characteristic matches recent many-core trend
  - HogBatch is more scalable than Mini-Batch in terms of #core

		Cores
	SGDSerial	1
Frequency	1.3	4.499
	2.6	2.281

		Cores					
	Hogwild	1	2	4	8	14	14 + HT
Frequency	1.3	5.299	6.751	3.296	1.757	1.243	1.038
	2.6	2.595	2.975	1.436	0.829	0.590	0.500

		Cores					
	MiniBatch	1	2	4	8	14	14 + HT
Frequency	1.3	8.250	4.400	2.586	1.443	1.031	1.336
	2.6	4.608	2.403	1.399	0.820	0.586	0.707

		Cores					
	HogBatch	1	2	4	8	14	14 + HT
Frequency	1.3	8.137	3.257	1.577	0.798	0.533	0.370
	2.6	4.45	1.721	0.800	0.439	0.281	0.208

Fig. 3: Time (in seconds) to 99.5% optimum loss, across core count and variable frequency[GHz]
Using RCV1-v1-test

Experimental Analysis
Future Scalability

- The authors simulated HogBatch on large #core system with an execution-driven simulator *Spiner* [9]
  - Single-threaded 2-wide in-order core at 1.8 GHz
  - 2-dimentional mesh with 2 cores per mesh stop
    - 2 cycle hop latency
    - Link bandwidth of 64 bytes/s and MESIF coherence protocol
      - MESIF: MESI + Forwarding(=Shared willing to reply read request)

Typo of 64 GB/s ?

- On RCV-v1 dataset,
  - 64 cores achieved 53x scalability
  - 128 cores achieved 90x scalability
  - Loss in convergence per pass compared to serial was ~25%
    - In the 14 core machine, it was ~10%

Multi Model Regression

- Training multiple models simultaneously are useful when each sample has multiple labels
  - Labels and models are matrices, not vectors
  - Model-dimension can be parallelized, as well as sample-dimension
- Batching methods is no longer cache friendly
  - Multiple models and labels are dense and may not fit the cache
  - **Hogwild would be the best approach for multiple models training**

Algorithm 5: Multi-Model Hogwild SGD pseudocode for one datapass

```
1  #pragma omp parallel for schedule(dynamic)
2  for (index = 0; index < num_samples; index++) {
3    #pragma simd
4    for (m = 0; m < NUM_MODELS; m++) {
5      // Sparse indices update of model[m]
6      model[m] -= a * Gradient(model[m], index);
7  } }
```

Multi Model Regression

- Model matrix should be allocated as row-major format
  - Same indices for each model are stored consecutively, followed by optional padding
    - Padding prohibits one cache line from holding parts of several indecies
  - This format allows to execute SIMD operations
    - Since Hogwild updates models sparsely, it access the same indices of all models at once

$$W = (w_1 w_2 \cdots w_k) = \begin{pmatrix} w_1^{(1)} & w_2^{(1)} & \cdots & w_k^{(1)} \\ w_1^{(2)} & w_2^{(2)} & \cdots & w_k^{(2)} \\ & & \ddots & \\ \cdots & \cdots & & \cdots \\ w_1^{(d)} & w_2^{(d)} & \cdots & w_k^{(d)} \end{pmatrix}$$

k models

Multi Model Regression

- Until 32 models, Hogwild get benefits the SIMD-friendly layout

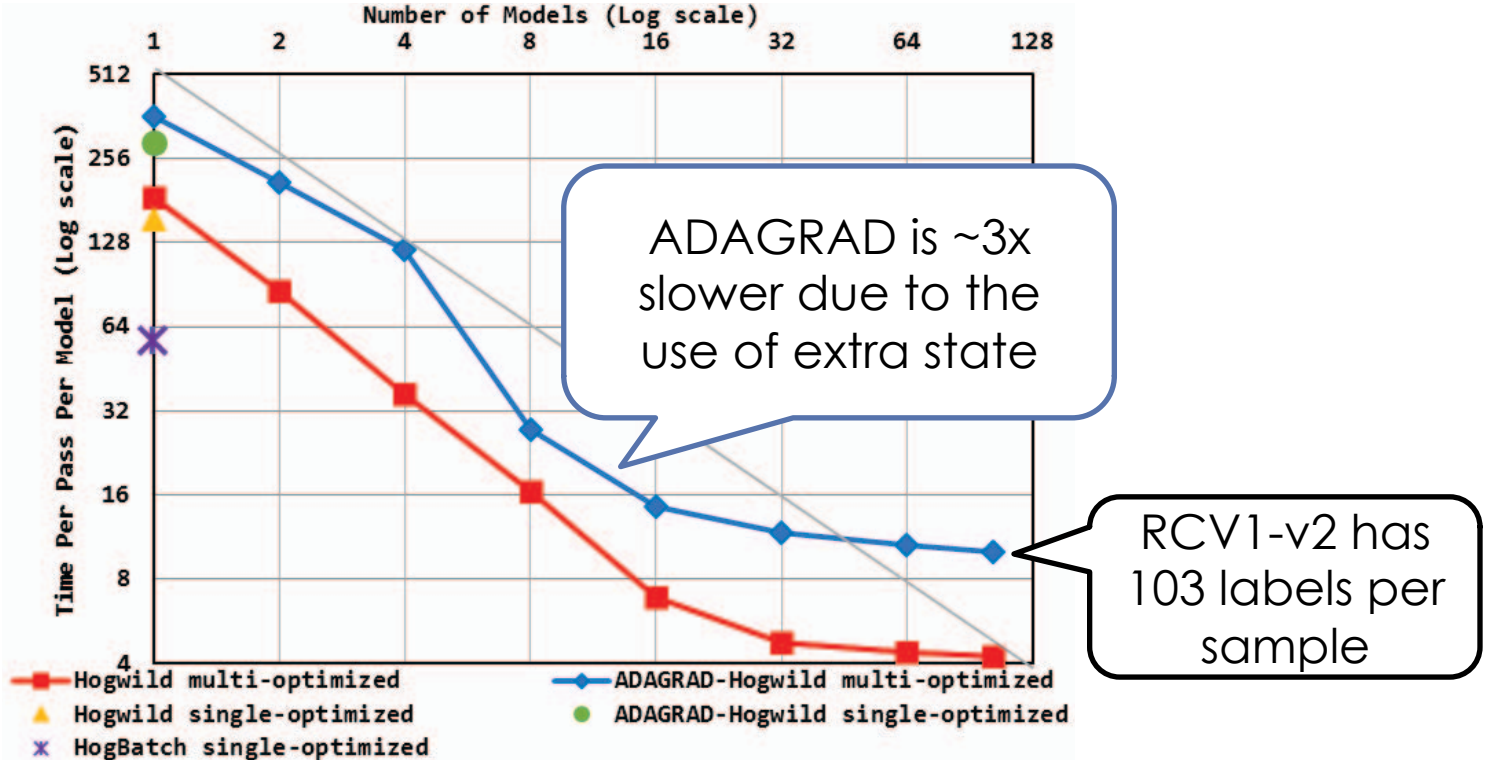


Fig. 4: Time per pass Per Model, scaling with number of models
Using RCV1-v2

Comparison to State-of-the-Art

- The authored compared their implementation with *BidMach* [2], a general purpose machine learning framework
  - Intel Xeon E5-2680 Sandy Bridge at 2.7 GHz
  - NVIDIA Titan X
  - Same parameters and ADAGRAD are used for comparison
- For single model performance, HogBatch is ~100x better than BidMach's CPU, and significantly faster than its GPU

TABLE V: Single model comparison using RCV1-v1-test dataset

Implementation	Hardware	Time/Pass (ms)
BidMach	TITAN X	723
BidMach	Sandy Bridge	14,190
CPU optimized (Mini-Batch)	Sandy Bridge	289
CPU optimized (Hogwild)	Sandy Bridge	253
CPU optimized (HogBatching)	Sandy Bridge	147
CPU optimized (HogBatching)	Haswell	111

BidMach is not so optimized for single model and CPU, as the developers mention

Comparison to State-of-the-Art

- For multiple models,
  - Hogwild on Sandy Bridge is on par with the GPU
  - Hogwild on Haswell slightly beats the GPU

TABLE VI: Multi model comparison using RCV1-v1-test dataset

	Implementation	Hardware	Models	Time / Pass (ms)
	BidMach	TITAN X	103	2,170
	BidMach	Sandy Bridge	103	120,720
Hogwild	CPU optimized	Sandy Bridge	103	2,010
Hogwild	CPU optimized	Haswell	103	1,283
Hogwild	CPU optimized	2x Haswell	103	724

51(52) models per CPU

Related Work

- Hogwild [1] by F. Niu et al. provided a strong foundation of the paper
- M. Zinkevich et al. offered parallel SGD that splits the workload and each machine perform SGD on a subset of data before averaging [13]
- C. De Sa et al. described an analysis of Hogwild-like method Buckwild [16]
- Ce Zhang and C. Ré et al. presented DimmWitted, characterizing state of the art SGD on NUMA [17]

Conclusions

- The authors presented HogBatch, which achieves superior hardware/statistical efficiency and up to near linear scalability
  - HogBatch is friendly towards future many-core platforms
- As future work, the authors intend to explore the use of HogBatch on other ML problems
  - Stochastic coordinate descent algorithms
  - Collaborative filtering problems
  - Non-convex problems