CS 267
Lecture 16: Parallel Machine Learning, Part 1

(Supervised Learning)

Aydin Buluc

https://sites.google.com/lbl.gov/cs267-spr2021/

Slide acks: Amir Gholami, Forrest Iandola, Prabhat, and Yang You

Outline of the parallel ML lectures

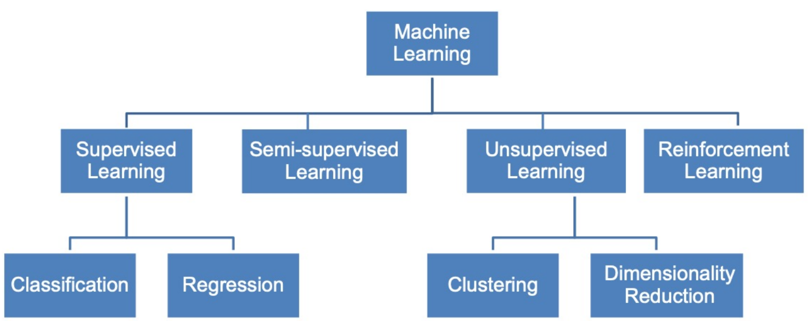
Today: Part 1, Intro and Supervised Learning
- Machine Learning & Parallelism Intro
- Neural Network Basics
- Deep Neural Network Training
- Support Vector Machines

Tuesday: Part 2, Unsupervised and Semi-Supervised Learning
- Non-Negative Matrix Factorization
- Spectral Clustering
- Graph Embedding and Graph Neural Networks

Machine Learning Classes and Tasks

The central question in Machine Learning:

"How can we build computer systems that automatically improve with experience, and what are the fundamental laws that govern all learning processes?"



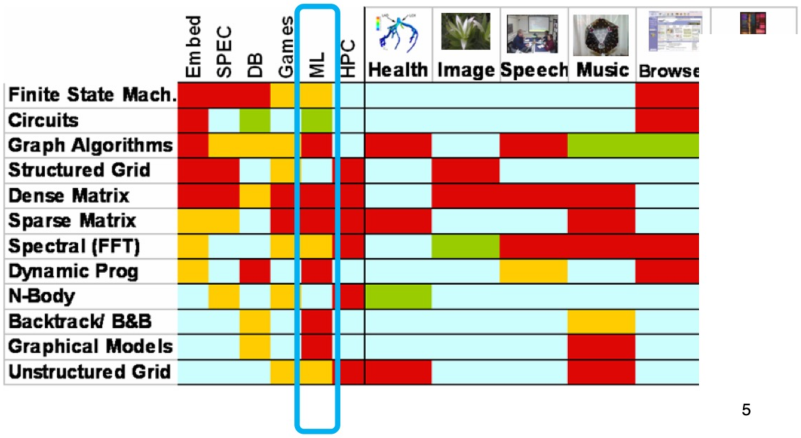
Machine Learning Sources of Confusion

Method vs. Task: A common confusion is between specific learning methods and learning tasks.
- Example #1: Principal Component Analysis is a method for dimensionality reduction
- Example #2: Support Vector Machines are methods used for supervised learning tasks.

Another confusion comes from ***optimization techniques*** vs. **learning methods**.
- Example #1: Sequential Minimal Optimization is an optimization technique to train Support Vector Machines
- Example #2: Stochastic Gradient Descent is a popular optimization technique to train Neural Networks.

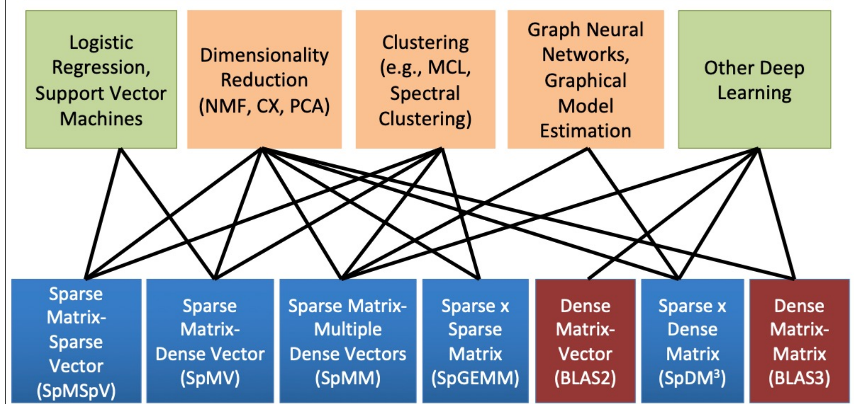
Motifs

The Motifs (formerly "Dwarfs") from "The Berkeley View" (Asanovic et al.)
Motifs form key computational patterns



Machine Learning relies heavily on Linear Algebra

Higher-level machine learning tasks



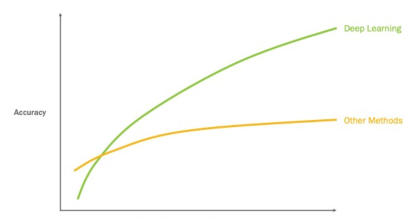
Graph/Sparse/Dense BLAS functions (in increasing arithmetic intensity)

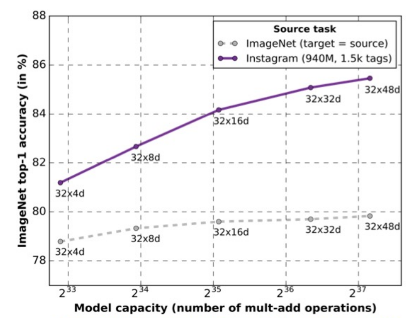
Parallelism in Machine Learning

Implicit Parallelization: Keep the overall algorithm structure (the sequence of operations) intact and parallelize the individual operations.
Example: parallelizing the BLAS operations in previous figure
+ Often achieves exactly the same accuracy (e.g., model parallelism in DNN training)
- Scalability can be limited if the critical path of the algorithm is long

Explicit Parallelization: Modify the algorithm to extract more parallelism, such as working on individual pieces whose results can later be combined
Examples: CA-SVM and data parallelism in DNNs
+ Significantly better scalability can be achieved
- No longer the same algorithmic properties (e.g. HogWild!).

Modern machine learning is high-performance computing





Source: Advancing state-of-the-art image recognition with deep learning on hashtags

Slide source: Misha Smelyanskiy (Facebook AI co-design director)

Outline of the parallel ML lectures

Today: Part 1, Intro and Supervised Learning

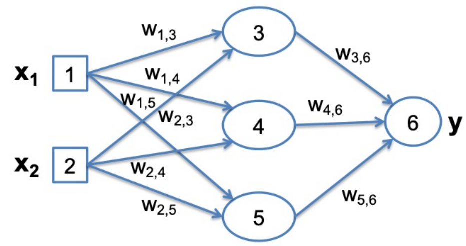
- Machine Learning & Parallelism Intro
- Neural Network Basics
- Deep Neural Network Training
- Support Vector Machines

Tuesday: Part 2, Unsupervised and Semi-Supervised Learning

- Non-Negative Matrix Factorization
- Spectral Clustering
- Graph Embedding and Graph Neural Networks

Training Neural Networks

- Training is to **adjust the weights (W)** in the connections of the neural network, in order to change the function it represents.

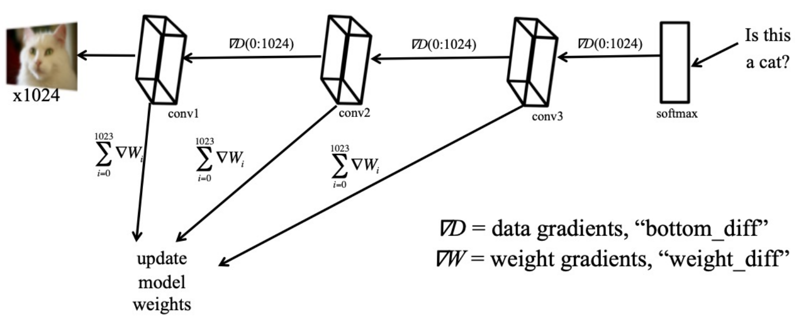


Only parameters are weights for simplicity (i.e. ignore bias parameters)

W: the matrix of weights

A "shallow" neural network with only one hidden layer (nodes 3,4,5), two inputs and one output.

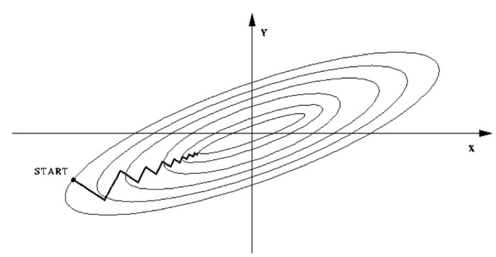
Deep Neural Network Training

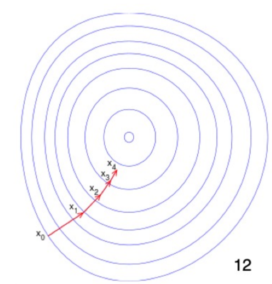


Gradient Descent

$$W^{t+1} \leftarrow W^{t} - \alpha \cdot \nabla_W f(W^{t}, x)$$

- Also called the steepest descent algorithm
- In order to minimize a function, move towards the opposite direction of the gradient at a rate of α.
- α is the step size (also called the learning rate)
- Used as the ***optimization backend*** of many other machine learning methods (example: NMF)





Stochastic Gradient Descent (SGD)

$$\text{Assume } f(W^t,x) = \frac{1}{n}\sum_{i=1}^{n} f_i(W^t,x)$$

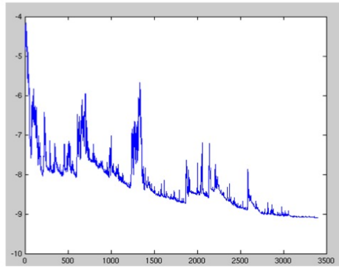
$$W^{t+1} \leftarrow W^t - \alpha \cdot \nabla_W f_i(W^t,x)$$

Pure SGD: compute gradient using 1 sample

$$W^{t+1} \leftarrow W^t - \alpha \cdot \frac{1}{b}\sum_{i=k+1}^{k+b} \nabla_W f_i(W^t,x)$$

Mini-batch: compute gradient using b samples

f is not going down for every iteration

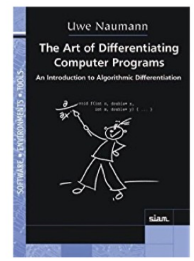


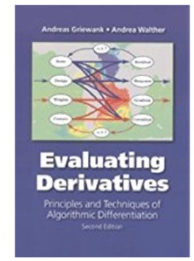
- Actually the name is a misnomer, *this is not a "descent" method*
- But we will stick to it anyway to avoid confusion.
- Performance and parallelism requires batch training
- Larger batch sizes hurt convergence as they get trapped easily
- SGD escapes sharp local minima due to its "noisy" gradients

Training Neural Networks

- Training is performed using an optimization algorithm like SGD
- **SGD needs derivatives.**
- The algorithm ***to compute derivatives*** on a neural network is called ***back-propagation***.
- The back-propagation algorithm is ***not a training algorithm***
- **Idea:** Repeated application of the chain rule from calculus

Back-propagation is just a special case of the ***reverse mode automatic/algorithmic differentiation***



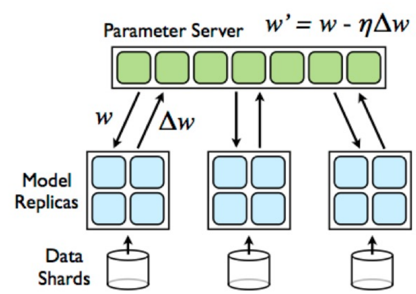


Parallelization Opportunities

1. **Data parallelism**
   Distribute* the input (sets of images, text, audio, etc.)
   a) **Batch parallelism**
      - Distribute each full sample to a different processor
      - *When people mention data parallelism in literature, this is what they mean 99% of the time*
   b) **Domain parallelism**
      - Subdivide samples and distribute parts to processors.
2. **Model parallelism:**
   Distribute the neural network (NN), i.e. its weights
3. **Pipeline parallelism:**
   Inter-batch parallelism, pipelined through NN layers

*: "Distribute" = giving parts to processors (in contrast to replicating)

Batch Parallelism #1



Parameter server is some sort of master process

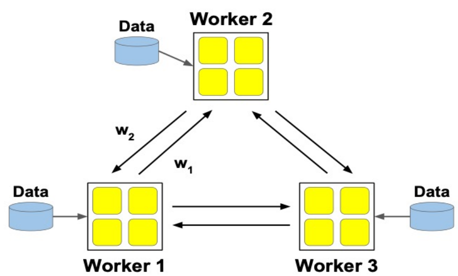
- The fetching and updating of gradients in the parameter server can be done either ***synchronously*** or ***asynchronously***.
- Both has pros and cons. Over-synchronization hurts performance where asynchrony is not-reproducible and might hurt convergence

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

Batch Parallelism #2

Options to avoid the parameter server bottleneck

1. **For synchronous SGD:** Perform all-reduce over the network to update gradients (good old MPI_Allreduce)
2. **For asynchronous SGD:** Peer-to-peer gossiping



Peter Jin, Forrest Iandola, Kurt Keutzer, "How to scale distributed deep learning?" NIPS ML Sys 2016

Model Parallelism

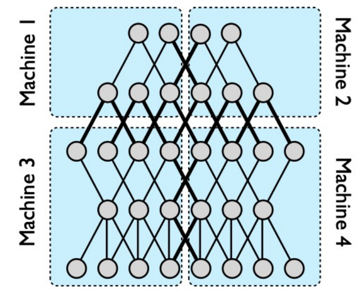
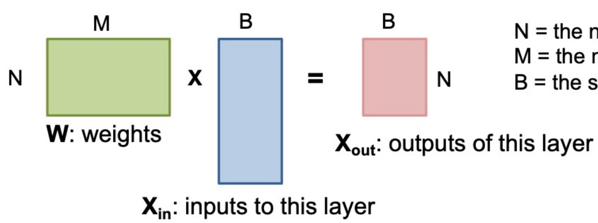


Figure shows both *inter-layer model parallelism* (a.k.a. *pipeline parallelism*) and *intra-layer model parallelism*

Interpretation #1: Partition your neural network into processors
Interpretation #2: Perform your matrix operations in parallel

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

Batch-parallel SGD training of NNs as matrix operations

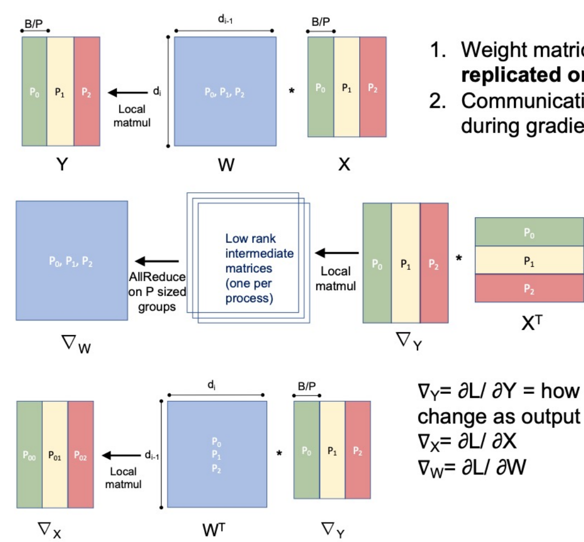


N = the number of outputs
M = the number of inputs
B = the size of the minibatch

The impact to parallelism:

- **W** is replicated to processor, so it doesn't change
- X_{in} and X_{out} gets skinnier if we only use data parallelism, i.e. distributing **b=B/p** mini-batches per processor
- GEMM performance suffers as *matrix dimensions get smaller* and *more skewed*
- **Result:** Batch parallelism can hurt single-node performance

Batch Parallel SGD training of NNs as matrix operations



1. Weight matrices (W) are **replicated on each process**
2. Communication only happens during gradient reductions

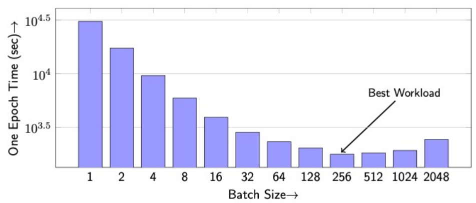
$\nabla_Y = \partial L / \partial Y$ = how did the lost function change as output activations change?
$\nabla_X = \partial L / \partial X$
$\nabla_W = \partial L / \partial W$

Batch Parallel Strong Scaling

- **Per-iteration communication cost of batch parallelism is independent of the batch size:**
  - **larger batch → less communication per epoch (full pass over the data set)**

$$T_{comm}(batch) = 2\sum_{i=0}^{L}\left(\alpha \log(P) + \beta \frac{P-1}{P}|W_i|\right)$$

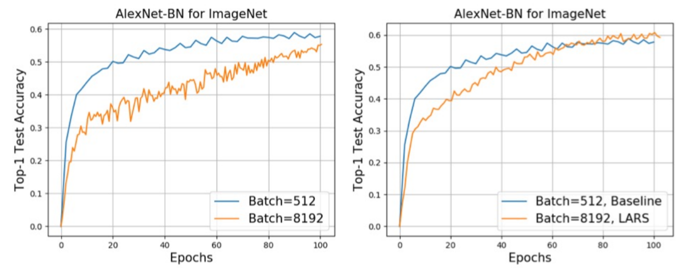
- **But processor utilization goes down significantly for P>>1**
  - **Result: Batch parallel has poor strong scaling**



One epoch training time of AlexNet computed on a single KNL

Problems with Batch Parallelism

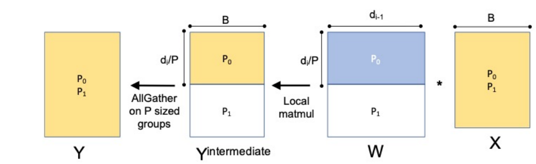
- **Can not train large models** due to single node memory limitations
- Batch parallel **scaling is limited to batch size B**
  - Larger Batch -> higher strong scaling efficiency
- But vanilla **SGD loses efficiency for large batch sizes**
- Use LARS (Layer-wise Adaptive Rate Scaling) instead

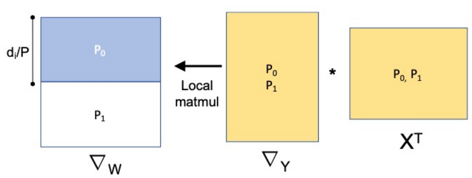


(a) Training without LARS (b) Training with LARS

Ginsburg, Boris, Igor Gitman, and Yang You. "Large Batch Training of Convolutional Networks with Layer-wise Adaptive Rate Scaling." (2018).

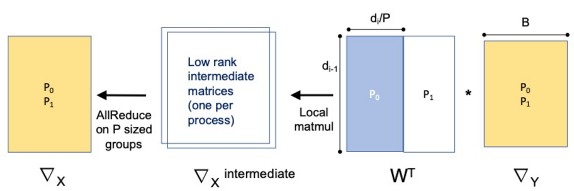
Model Parallel SGD training of NNs as matrix operations





intra-layer model parallelism:

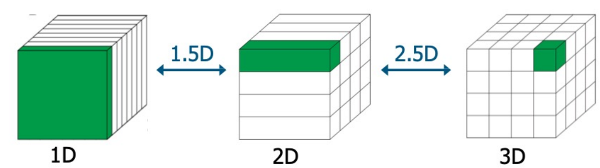
1. Same data passes through all processes, but this is limited to the mini-batch size
2. Two communication steps



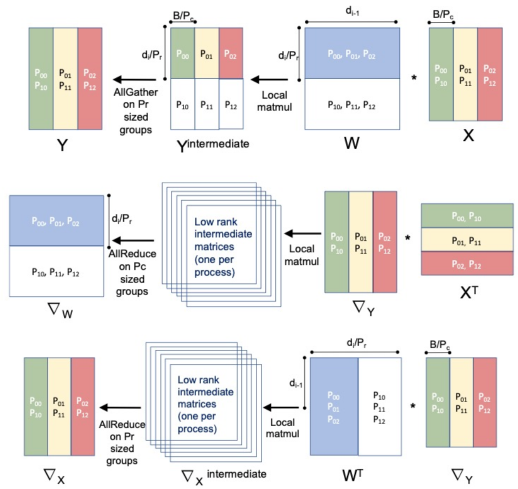
Combinations of various parallelism opportunities

- There are several different ways to combine DNN training parallelism opportunities.
  - It helps to think in terms of matrices again.
- We will exploit ***communication-avoiding matrix algorithms***; which trade off some storage (judicious replication) at the expense of reduced communication.
  - Deep Learning community is already OK with data or model replication in many cases

A succinct classification of parallel matrix multiplication algorithms



Batch & Model Parallel SGD training of NNs as matrix operations

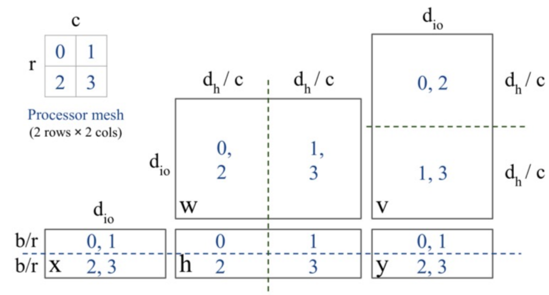


Processes are 2D indexed:
P= P_r x P_c

Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, Aydın Buluç. "Integrated Model, Batch, and Domain Parallelism in Training Neural Networks." ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'18)

How to implement hybrid parallelism

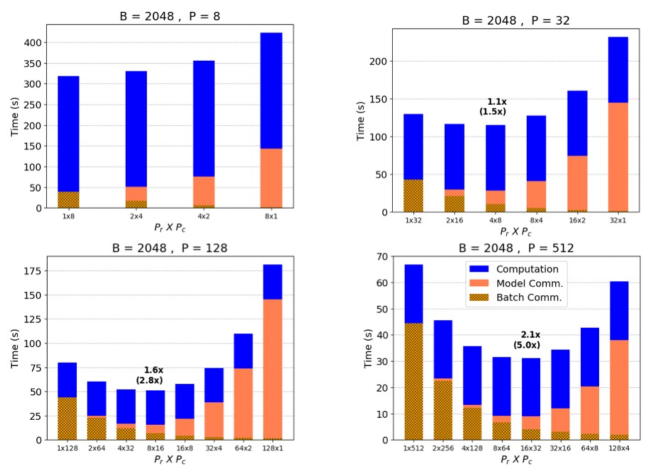
1. Do it yourself manually on PyTorch matrices, if you understand how your preferred hybrid parallelism maps to matrices
2. There are also tools to simplify, such as Mesh-Tensorflow



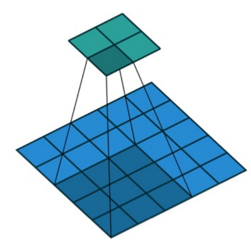
Koanantakool P, et al. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. IPDPS, 2016
Shazeer N, et al. Mesh-TensorFlow: Deep Learning for Supercomputers. NeurIPS. 2018

Integrated Batch + Model Scaling

° **For large processes integrated could provide up to 2x speedup**



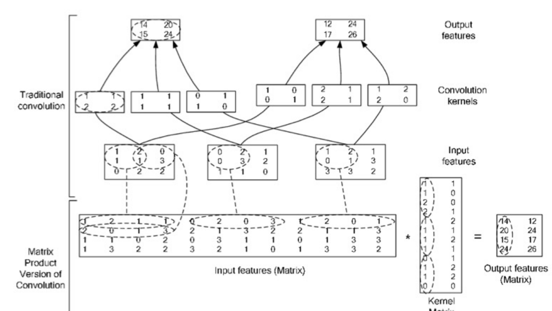
What about convolutional neural networks (CNNs)?



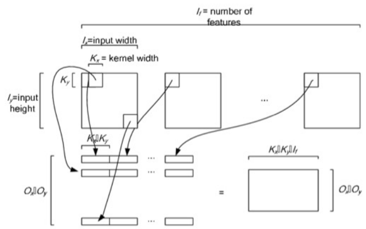
Left: Vincent Dumoulin, Francesco Visin - [A guide to convolution arithmetic for deep learning]

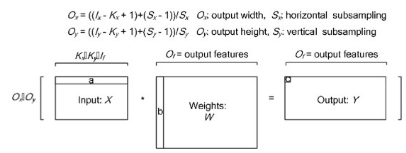
Bottom: Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing.

You can convert direct convolutions to matrix multiplies with overhead proportional to the ratio of overlap (function of stride length) to the filter size

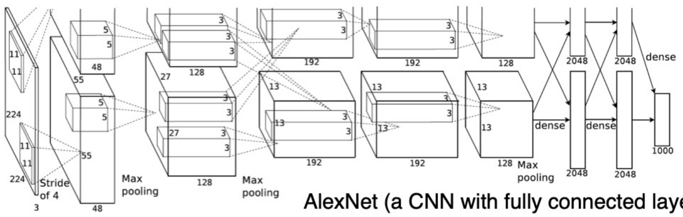


What about convolutional neural networks (CNNs)?





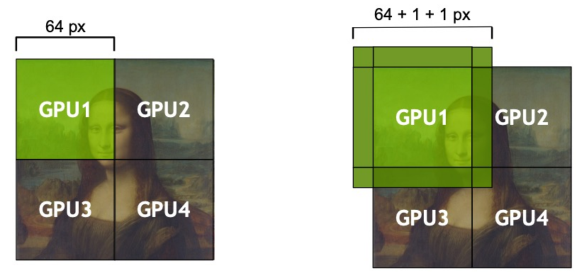
Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing.



AlexNet (a CNN with fully connected layers)

Domain Parallel

- The general idea is the same as *halo regions* or *ghost zones* used to parallelize stencil codes in HPC
  - Before a convolution, exchange local receptive field boundary data

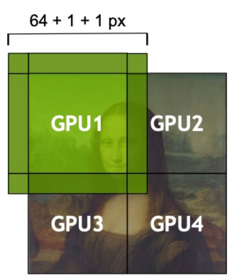


Peter Jin, Boris Ginsburg, and Kurt Keutzer. "Spatially Parallel Convolutions" ICLR Workshop Track, 2018

Communication Complexity of Domain Parallel

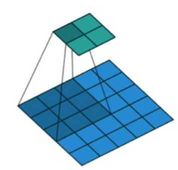
- Additional communication for halo exchange during forward and backwards pass
  - Negligible cost for early layers for which activation size is large (i.e. convolutional)

$$T_{comm}(domain) = \sum_{i=0}^{L}(\alpha + \beta B X_W^i X_C^i k_h^i/2) + \sum_{i=0}^{L}(\alpha + \beta B Y_W^i Y_C^i k_w^i/2) + 2\sum_{i=0}^{L}\left(\alpha \log(P) + \beta \frac{P-1}{P}|W_i|\right)$$



Domain Parallel Scaling

- Domain parallel scaling on V100 GPUs
  - B=32, C=64, K=3, D=1, R=1



Resolution	GPUs	Fwd. wall-clock	Bwd. wall-clock
128 × 128	1	2.56 ms (1.0×)	6.63 ms (1.0×)
	2	1.52 ms (1.7×)	3.50 ms (1.9×)
	4	**1.23 ms (2.1×)**	**2.33 ms (2.8×)**
256 × 256	1	10.02 ms (1.0×)	26.81 ms (1.0×)
	2	5.34 ms (1.9×)	11.79 ms (2.3×)
	4	**3.11 ms (3.2×)**	**6.96 ms (3.9×)**
512 × 512	1	45.15 ms (1.0×)	126.11 ms (1.0×)
	2	20.18 ms (2.2×)	60.15 ms (2.1×)
	4	**10.65 ms (4.2×)**	**26.76 ms (4.7×)**

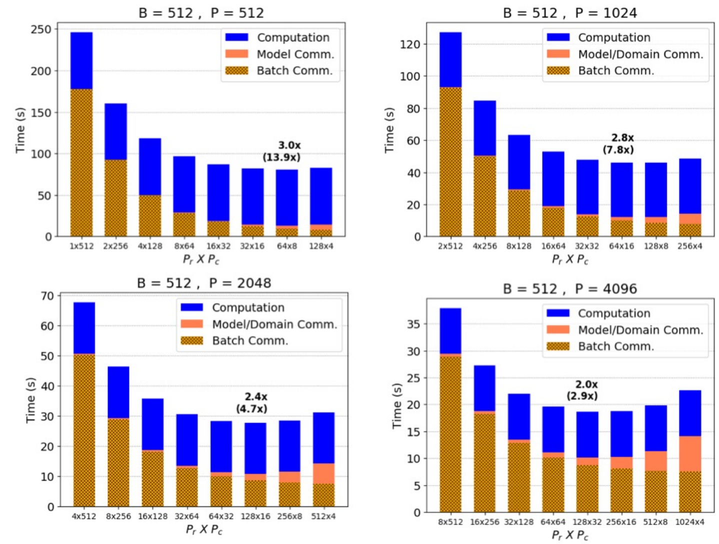
Peter Jin, Boris Ginsburg, and Kurt Keutzer. "Spatially Parallel Convolutions" ICLR Workshop Track, 2018
Figure: Dumoulin, V., Visin, F.. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285*, 2016.

Integrated Batch, Domain, and Model Parallel

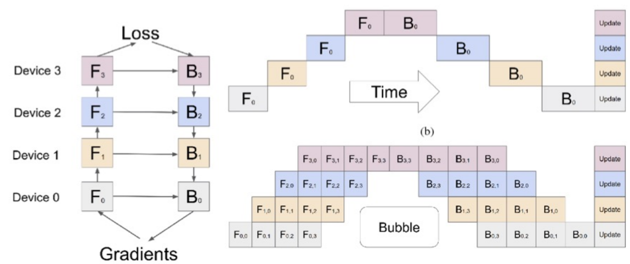
- **Batch + Model** for layers with small activation size and large parameters (fully-connected layers and transformer networks)
- **Batch + Domain** for layers with large activation sizes (convolutional layers)

$$
\begin{aligned}
T_{comm} = &\sum_{i\in L_M}\left(\alpha\log(P_r) + \beta\frac{B}{P_c}\frac{P_r-1}{P_r}d_i\right) + \quad 2\sum_{i\in L_M}\left(\alpha\log(P_r) + \beta\frac{B}{P_c}\frac{P_r-1}{P_r}d_{i-1}\right)\\
&+2\sum_{i\in L_M}\left(\alpha\log(P_c) + \beta\frac{P_c-1}{P_c}\frac{|W_i|}{P_r}\right) + \quad \sum_{i\in L_D}\left(\alpha + \beta\frac{B}{P_c}X_W^i X_C^i k_h^i/2\right)\\
&+\sum_{i\in L_D}\left(\alpha + \beta\frac{B}{P_c}X_W^{i+1}X_C^{i+1}k_w^i/2\right) + \quad 2\sum_{i\in L_D}\left(\alpha\log(P) + \beta\frac{P-1}{P}|W_i|\right)
\end{aligned}
$$

Integrated Batch, Domain, and Model Parallel



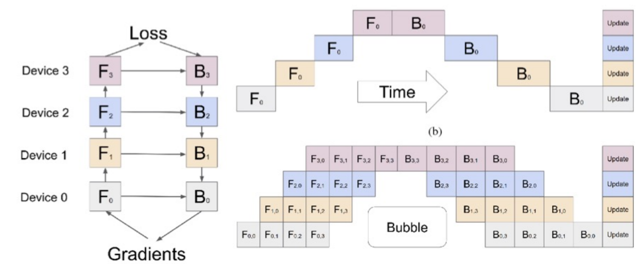
Pipeline Parallelism



- Pipeline parallelism divides the input mini-batch into **smaller micro-batches**, enabling different GPUs to work on different micro-batches simultaneously. Gradients are applied synchronously at the end.
- Without micro-batching, pipelining would not expose any real parallelism (i.e., layers would merely be processed by different GPUs)

- Petrowski A, Dreyfus G, Girault C. Performance analysis of a pipelined backpropagation parallel algorithm. IEEE Transactions on Neural Networks. 1993 Nov;4(6):970-81 (original idea)
- Huang Y, Cheng Y, Chen D, Lee H, Ngiam J, Le QV, Chen Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. NeurIPS, 2019 (figure source)

Pipeline Parallelism



- Pipeline parallelism is a mix of (inter-layer) **model parallelism,** as *it parallelizes across the inter-layer NN model structure*, and **batch parallelism,** as *it needs micro-batches of data for filling the pipeline*.
- **Pipeline bubbles:** start of the forward propagation on a minibatch requires the backprop of the previous minibatch to complete
- Contribution of pipeline parallelism to the total available parallelism is **a multiplicative factor that is bounded by the NN depth**. Without a deep network, all micro-batching would achieve is batch parallelism.

Pipeline parallelism: synchronization of the weight matrix

In the backpropagation, you need to compute gradients using the same weight matrix W you used during forward propagation:

1. Either each process can store its own W: $W_1, W_2, \dots, W_p$, effectively increasing memory footprint to match batch parallelism
2. Or we do periodic flushes so we can use one synchronized W.

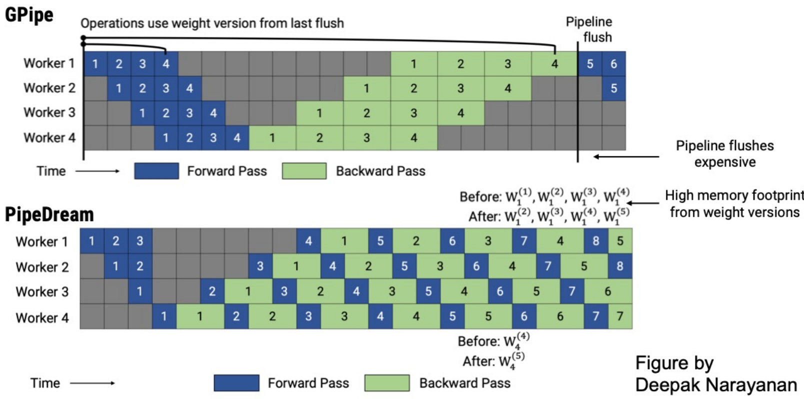


Figure by Deepak Narayanan

Summary of distributed deep learning

- Large batch size training often lead to suboptimal learning, but this can be mitigated with better learning algorithms such as LARS
- Integrated parallelism uses communication-avoiding algorithms to extend scaling beyond batch size
- Integrated parallelism optimally combines model and data (batch and domain) parallelism and often performs better than each extreme.
- It is often better [1] to use (inter-layer) **model parallelism** for fully connected layers (large parameters), and **batch parallelism** for convolutional layers (large activations)
- Pipeline parallelism can also be combined (in principle) with intra-layer model parallelism.

[1] Krizhevsky, Alex. "One weird trick for parallelizing convolutional neural networks." *arXiv preprint arXiv:1404.5997* (2014).

Outline of the parallel ML lectures

Today: Part 1, Intro and Supervised Learning

- Machine Learning & Parallelism Intro
- Neural Network Basics
- Deep Neural Network Training
- Support Vector Machines

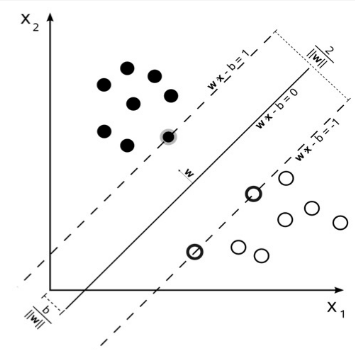
Tuesday: Part 2, Unsupervised and Semi-Supervised Learning

- Non-Negative Matrix Factorization
- Spectral Clustering
- Graph Embedding and Graph Neural Networks

Support Vector Machine (the linear case)

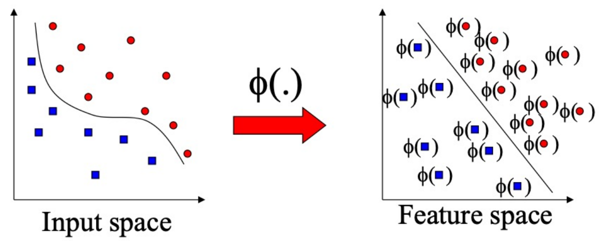
$$\max_{w,b} \quad \frac{1}{\|w\|}$$

$$\text{s.t} \quad y_i(w^T x_i + b) \geq 1, \quad \forall i$$



- Only the classification constraints on the support vectors are active
- Naively, leads to a giant quadratic constrained optimization (QP) problem
- Special algorithms, such as Sequential Minimal Optimization (SMO), decompose this giant QP to smaller (in fact ***minimal***) QP sub-problems.

Kernel Support Vector Machine (the non-linear case)



Slide: Eric Xing

Note: feature space is of higher dimension than the input space in practice

- Computation in the feature space can be costly because it is high dimensional
  - The feature space is typically infinite-dimensional!
- ***The kernel trick*** comes to rescue

Kernel Methods

- Kernel methods replace the dot-product similarity of linear models with an arbitrary Kernel function that computes the similarity between x and y

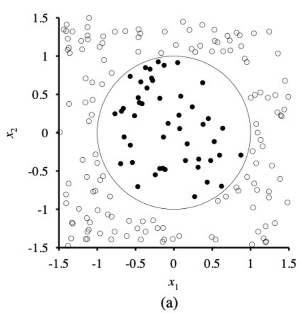
$$K(x, y) : X \times X \to \mathbb{R}$$

- Kernels need to be symmetric and positive semi-definite.
- Gaussian Kernel (i.e., Radial Basis Function) is the de-facto kernel in ML.

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$$

- We could pre-compute the Kernel (Gram) Matrix for all pairs of samples, but this is too expensive.

The Kernel Trick



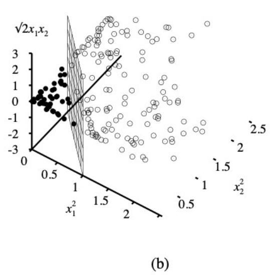


Figure source: Russell & Norvig

The circular decision boundary in 2D (a) becomes a linear boundary in 3D (b) using the following transformation: $\phi(x_1, x_2) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$

If we define the kernel function as $\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$

Then no need to carry out ϕ(.) explicitly as $\Phi(x) \cdot \Phi(y) = (x \cdot y)^2$

$$(x_1^2, x_2^2, \sqrt{2}x_1x_2) \cdot (y_1^2, y_2^2, \sqrt{2}y_1y_2) = (x_1y_1 + x_2y_2)^2$$

Major Bottleneck of Kernel SVM

- Input dataset: n-by-d matrix (n >> d)
  - X1, X2, …, Xn. Xi is a vector has d features
- Generate a n-by-n Kernel (Gram) matrix at runtime
  - K[i][j] = exp(-r||Xi - Xj||^2), r is positive number
- O(d*n^2) operations and O(n^2) memory are huge!
  - a small input generates a large Kernel matrix
  - 357MB input (52K-by-90) = 2000GB Kernel matrix
- Solution: SMO (sequential minimal optimization)
  - using iterative method, avoiding Kernel matrix
  - using ***two rows of Kernel matrix*** each iteration
  - key kernel for sparse inputs: ***sparse matrix times sparse vector*** (as in the BFS case from graph lecture)

Sequential Minimal Optimization

$$\max_{\alpha} \; W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \boxed{\kappa(\mathbf{x}_i^T \mathbf{x}_j)}$$ The kernel function

$$\text{s.t.} \quad 0 \le \alpha_i \le C, \quad i = 1, \dots, m$$

$$\boxed{\sum_{i=1}^{m} \alpha_i y_i = 0.}$$ The equality constraint

- The smallest possible optimization problem involves “two” Lagrange multipliers at a time
- Because just changing on multiplier would violate the equality constraint

Platt, John. “Fast training of support vector machines using sequential minimal optimization." Advances in kernel methods. 1999.

Sequential Minimal Optimization

$$\max_{\alpha} \; W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \boxed{\kappa(\mathbf{x}_i^T \mathbf{x}_j)}$$ The kernel function

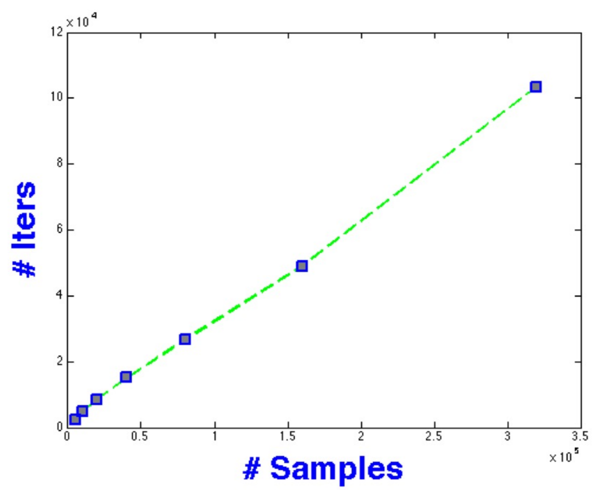
$$\text{s.t.} \quad 0 \le \alpha_i \le C, \quad i = 1, \dots, m$$

$$\boxed{\sum_{i=1}^{m} \alpha_i y_i = 0.}$$ The equality constraint

Repeat until convergence:

1. Select some pair α_i and α_j to update next (using a heuristic that tries to pick the two that will allow us to make the biggest progress towards the global maximum).
2. Re-optimize W(α) with respect to α_i and α_j, while holding all the other α_k 's fixed.

#Iterations = O(#samples), bad Weak Scaling!



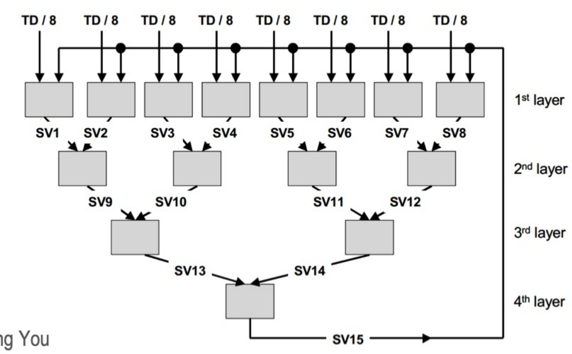
Slide: Yang You

Cascade SVM (NIPS’04)

Data is partitioned *evenly* and processed by multiple SVMs

Remove the non support vectors layer-by-layer

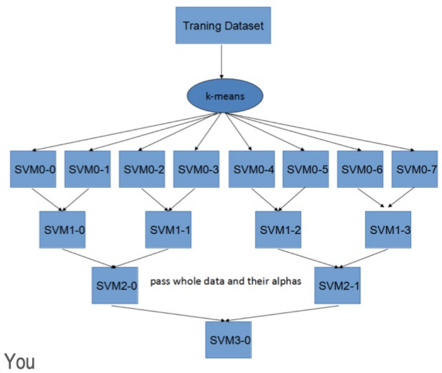
- data is the support vectors (SV) of previous layer (major reduction)
- pass parameters α_i of SVs to next layer for a better initiation (warm start)



Slide: Yang You

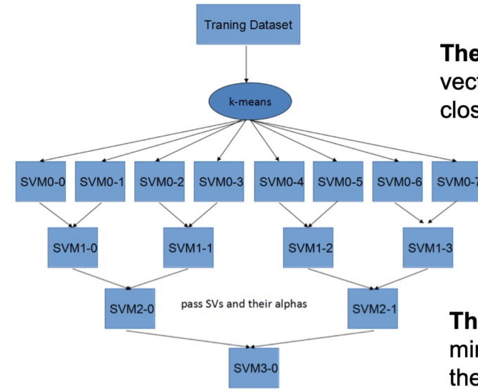
Divide-and-Conquer SVM: DC-SVM (ICML'14)

- Difference between DC-SVM and Cascade
  - DC-SVM passes all data layer-by-layer
  - DC-SVM uses kernel k-means to partition the dataset



Combine DC-SVM with Cascade: DC-Filter

- Only pass support vectors layer-by-layer (reduce workload)
- Use kernel k-means to divide the dataset



Theorem #1: the set of support vectors from the subproblems is close to that of the whole problem

Theorem #2: Kernel kmeans minimizes the difference between the solution of subproblems and of the whole problem

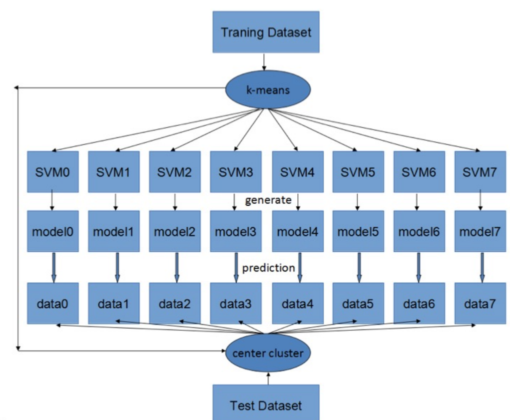
Bottleneck of Cascade, DC-SVM, and DC-Filter

- Occupy P machines, but bottom level uses 1 machine
- Lower levels cost more time than top level

level 1^{st}	6000	6000	6000	6000	6000	6000	6000	6000
time: 5.49s	4.87	4.92	4.90	4.68	5.12	5.10	5.49	4.71
iter: 6168	5648	5712	5666	5415	5936	5904	6168	5453
SVs: 5532	746	715	717	718	686	707	721	699
level 2^{nd}	1461		1435		1393		1420	
time: 1.58s	1.58		1.50		1.35		1.45	
iter: 7485	7485		7211		6713		7035	
SVs: 5050	1292		1263		1256		1239	
level 3^{rd}	2555				2495			
time: 3.34s	3.34				3.30			
iter: 9081	8975				9081			
SVs: 4699	2388				2311			
level 4^{th}	4699							
time: 9.69s	9.69							
iter: 14052	14052							
SVs: 4475	4475							

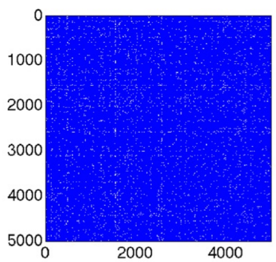
CP-SVM: Cluster Partition SVM (1-level divide-and-conquer)

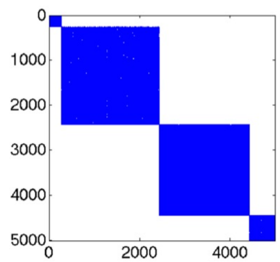
- Divide: K-means partitions data into P parts; P models
- Conquer: Euclidean distance to select best model



Why CP-SVM works?

- When ||Xi - Xj||^2 is large, exp(-r||Xi - Xj||^2) is zero
- K-means maximize different groups' Euclidean distance
- These two matrices have similar F-norm
- Analysis assumes the Gaussian kernel: For a given sample, only the support vectors close to it can have an effect on the classification





Slide: Yang You

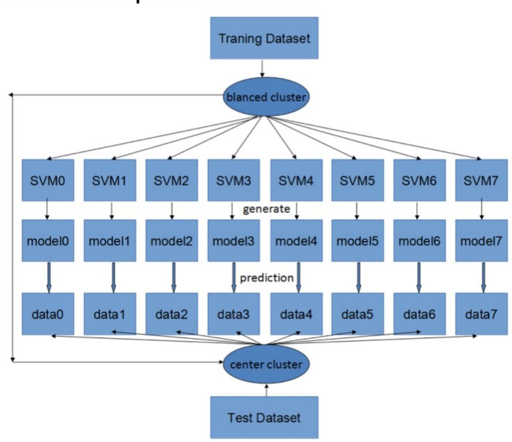
CP-SVM's bottleneck: load imbalance (because of K-means)

- **Machine 5, time: 0.75s, #iter: 1522, #samples: 4800**
- **Machine 1, time: 0.79s, #iter: 1384, #samples: 4800**
- **Machine 7, time: 0.94s, #iter: 1748, #samples: 4800**
- **Machine 6, time: 1.14s, #iter: 2337, #samples: 4800**
- **Machine 2, time: 1.14s, #iter: 2339, #samples: 4800**
- **Machine 4, time: 1.31s, #iter: 2856, #samples: 4800**
- **Machine 3, time: 5.48s, #iter: 6723, #samples: 9600**
- **Machine 3, time: 6.48s, #iter: 7915, #samples: 9600**

Slide: Yang You

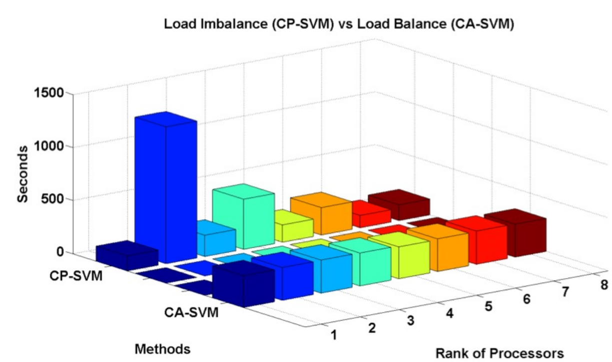
CA-SVM: Communication Avoiding SVM

- Design a balanced clustering to replace K-means
- Still approximating the distance separation property of k-means as much as possible.



Load Balance Comparison

- **Test dataset is epsilon with 128k samples on 8 machines**



Slide: Yang You

0.2 accuracy loss for 6.6x speedup

- Overall comparison for IJCNN dataset
- All methods use the same number of machines

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008, 23.8)
Cascade	95.5%	37,789	13.5s (0.007, 13.5)
DC-SVM	98.3%	31,238	59.8s (0.04, 59.7)
DC-Filter	95.8%	17,339	8.4s (0.04, 8.3)
CP-SVM	98.7%	7,915	6.5s (0.04, 6.4)
CA-SVM	98.5%	7,450	3.6s (0.005, 3.55)

You, Yang, et al. "CA-SVM: Communication-Avoiding Support Vector Machines on Distributed Systems", IPDPS, 2015