HPC Formulations of Optimization Algorithms for Tensor Completion

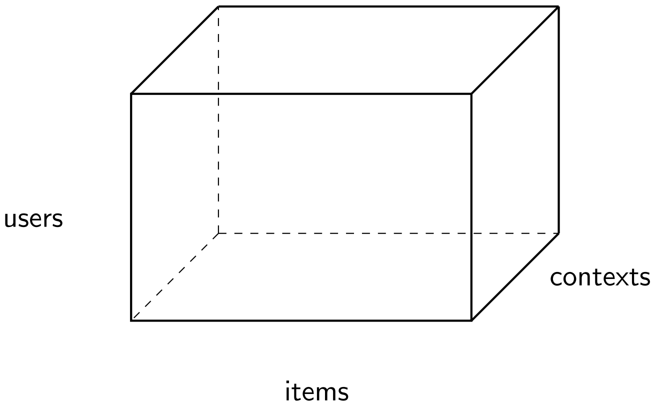
Shaden Smith[1,2], Jongsoo Park[3], and George Karypis[1]

[1]University of Minnesota
[2]Intel Parallel Computing Lab
[3]Facebook

shaden.smith@intel.com

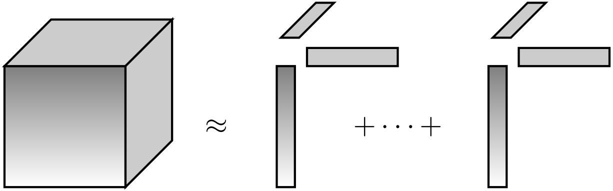
Tensor introduction

- Tensors are the generalization of matrices to $\geq 3D$.
- Tensors have N dimensions (or *modes*).
  - We will use dimensions $I \times J \times K$ in this talk.



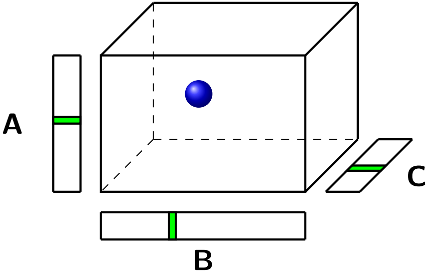
Tensor completion

- Many tensors are sparse due to missing or unknown data.
  - Missing values are *not* treated as zero.
- Tensor completion estimates a low rank model to recover missing entries.
  - Applications: recommender systems, social network analysis, ...



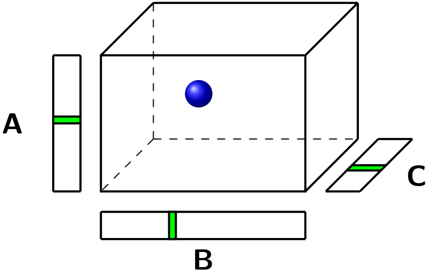
Tensor completion with the CPD

$\mathcal{R}(i, j, k)$ is written as the inner product of $\mathbf{A}(i, :)$, $\mathbf{B}(j, :)$, and $\mathbf{C}(k, :)$.



Tensor completion with the CPD

$\mathcal{R}(i, j, k)$ is written as the inner product of $\mathbf{A}(i,:)$, $\mathbf{B}(j,:)$, and $\mathbf{C}(k,:)$.



We arrive at a non-convex optimization problem:

$$\underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\text{minimize}} \qquad \underbrace{\mathcal{L}(\mathcal{R}, \mathbf{A}, \mathbf{B}, \mathbf{C})}_{\text{Loss}} + \underbrace{\lambda \left(||\mathbf{A}||_F^2 + ||\mathbf{B}||_F^2 + ||\mathbf{C}||_F^2 \right)}_{\text{Regularization}}$$

$$\mathcal{L}(\mathcal{R}, \mathbf{A}, \mathbf{B}, \mathbf{C}) = \frac{1}{2} \sum_{\text{nnz}(\mathcal{R})} \left(\mathcal{R}(i, j, k) - \sum_{f=1}^{F} \mathbf{A}(i, f)\mathbf{B}(j, f)\mathbf{C}(k, f) \right)^2$$

Challenges

Optimization algorithms

- Algorithms for *matrix* completion are relatively mature.
    - How do their tensor adaptations perform on HPC systems?
- Several properties to consider when comparing algorithms:
    1. Convergence rate.
    2. Number of operations and computational intensity.
    3. Memory footprint.
    4. Parallelism!

Experimental setup

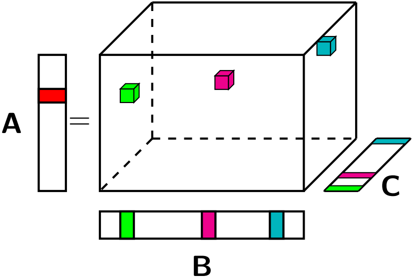
- Source code was implemented as part of SPLATT with MPI+OpenMP.
- Experiments are on the Cori supercomputer at NERSC.
  - Nodes have two sixteen-core Intel Xeon CPUs (Haswell).
- Experiments show a rank-10 factorization of the Yahoo Music (KDD cup) tensor.
  - 210 million *user-song-month* ratings.
  - More datasets and ranks in the paper.
- Root-mean-squared error (RMSE) on a test set measures solution quality:

$$\text{RMSE} = \sqrt{\frac{2 \cdot \mathcal{L}(\boldsymbol{\mathcal{R}}, \mathbf{A}, \mathbf{B}, \mathbf{C})}{\text{nnz}(\boldsymbol{\mathcal{R}})}}$$

Alternating least squares (ALS)

- Each row of **A** is a linear least squares problem.
- $\mathbf{H}_i$ is an $|\mathcal{R}(i,:,:)| \times F$ matrix:
  - $\mathcal{R}(i,j,k) \rightarrow \mathbf{B}(j,:) * \mathbf{C}(k,:)$ (elementwise multiplication).

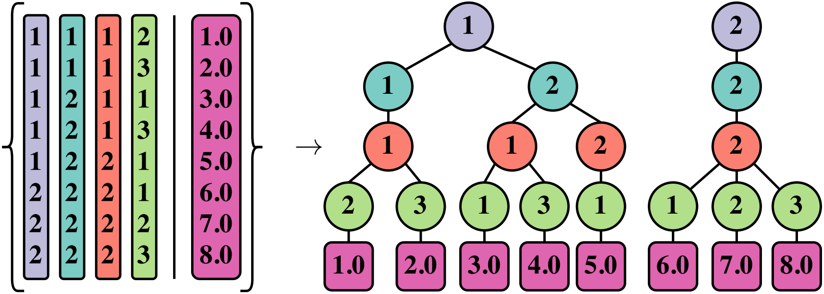
$$\mathbf{A}(i,:) \leftarrow \underbrace{\left(\mathbf{H}_i^T \mathbf{H}_i + \lambda \mathbf{I}\right)^{-1}}_{\text{normal eq.}} \underbrace{\left(\mathbf{H}_i^T \operatorname{vec}(\mathcal{R}(i,:,:))\right)}_{\text{MTTKRP}}$$



1	1	1	2	1.0
1	1	1	3	2.0
1	2	1	1	3.0
1	2	1	3	4.0
1	2	2	1	5.0
2	2	2	1	6.0
2	2	2	2	7.0
2	2	2	3	8.0

Compressed sparse fiber (CSF) [IPDPS'15; IA3'15]

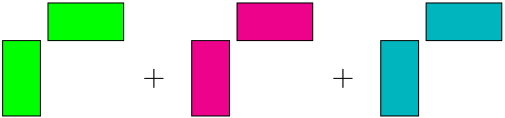
- Modes are recursively compressed.
- Paths from roots to leaves encode non-zeros.



BLAS-3 formulation

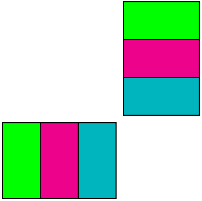
- Element-wise computation is an outer product formulation.
  - $\mathcal{O}(F^2)$ work with $\mathcal{O}(F^2)$ data per non-zero.

$\mathbf{H}_i =$



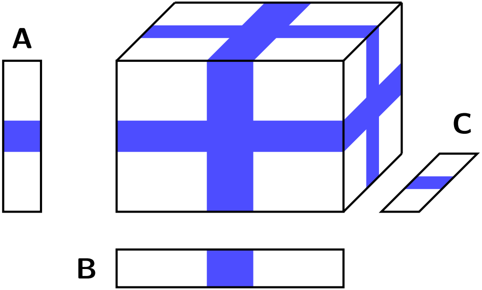
- Instead, collect $(\mathbf{B}(j,:) * \mathbf{C}(k,:))$ into a matrix $\mathbf{Z}$.

$\mathbf{H}_i =$



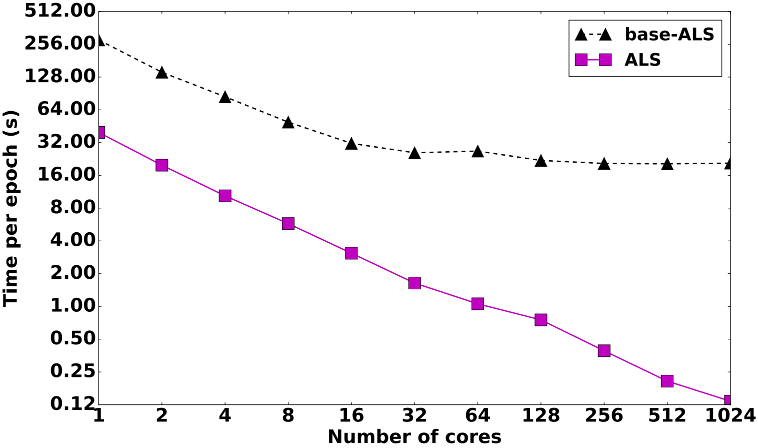
Distributed ALS [Choi & Vishwanathan '14; Shin & Kang '14]

- **Challenge:** unlike the traditional CPD, we have asymmetric communication.
  - Aggregating the partial $\mathbf{H}_i$ matrices is $O(IF^2)$.
- We use a *coarse-grained* decomposition.
- Only the updated rows need to be communicated, taking $O(IF)$.



ALS evaluation

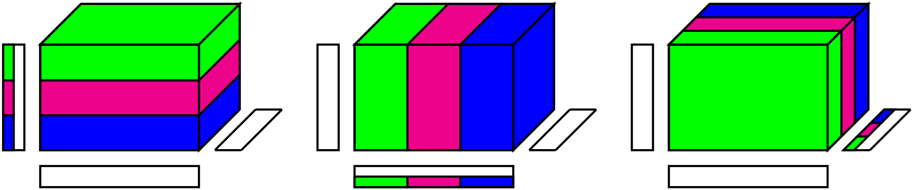
295× relative speedup and 153× speedup over base-ALS.



base-ALS is a pure-MPI implementation in C++ [Karlsson et al. '15]. **ALS** is our MPI+OpenMP implementation with one MPI rank per node.

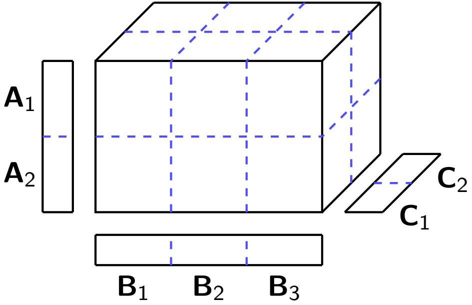
Coordinate descent (CCD++)

- Select a variable and update while holding all others constant.
- Rank-1 factors are updated in sequence.



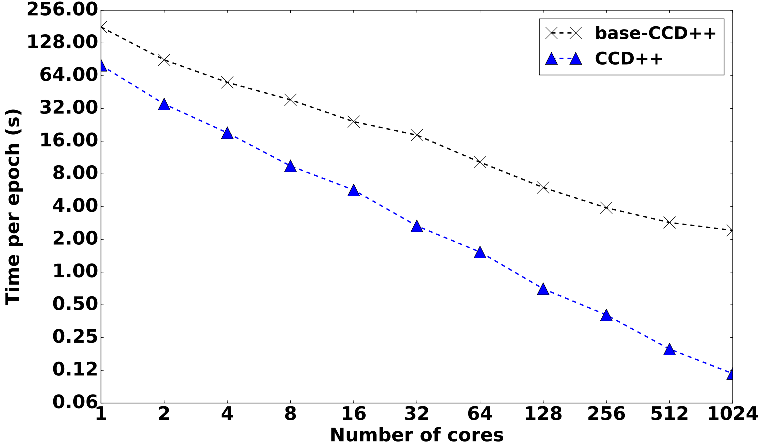
Distributed CCD++

- CCD++ has a communication volume matching traditional CPD, so we can leverage the work there.
- Medium- and fine-grained decompositions are scalable to large machines.



CCD++ distributed-memory evaluation

685× relative speedup and 21× speedup over base-CCD++.

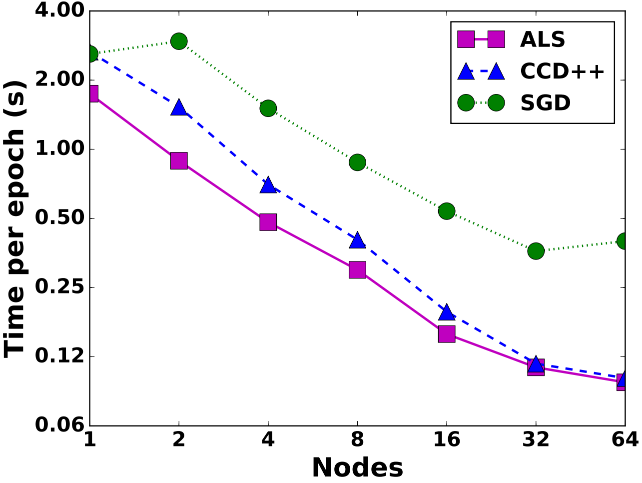


base-CCD++ is a pure-MPI implementation in C++ [Karlsson et al. '15].

CCD++ is our MPI+OpenMP implementation with two MPI ranks per node.

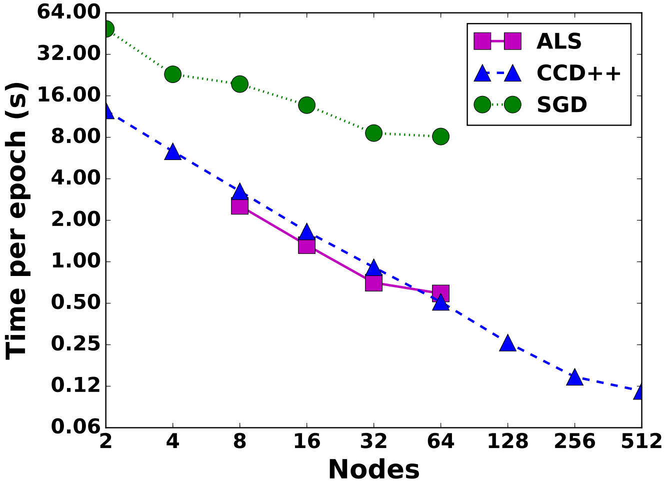
Strong scaling

- SGD exhibits initial slowdown as strata teams are populated.
- All methods scale to (past) 1024 cores.



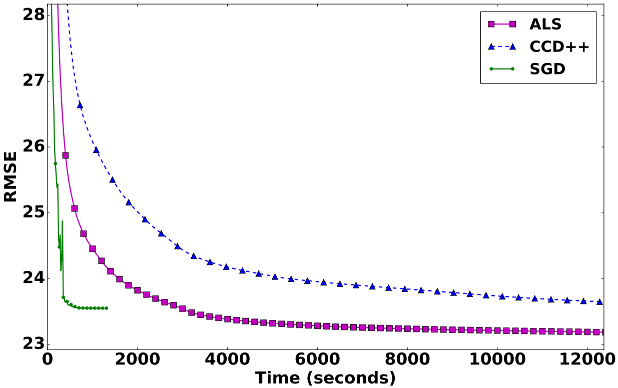
Patents strong scaling

Patents is a $46 \times 240K \times 240K$ tensor with 2.9B non-zeros.



Convergence @ 1 core

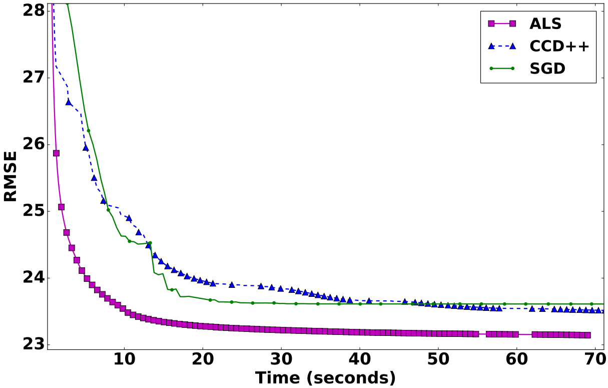
SGD rapidly converges to a high quality solution.



Convergence is detected if the RMSE does not improve after 20 epochs.

Convergence @ 1024 cores

- ALS now has the lowest time-to-solution.
- CCD++ and SGD exhibit similar convergence rates.



Convergence is detected if the RMSE does not improve after 20 epochs.

Wrapping Up

- Careful attention to sparsity and data structures can give over $10\times$ speedups.
- There is no "best" algorithm – it depends on your hardware architecture and problem.
  - SGD: best in a serial setting.
  - ALS: best in a multi-core setting or with a few nodes, but has a large memory footprint.
  - CCD++: best on large-scale systems, but requires high memory-bandwidth.

`http://cs.umn.edu/~splatt/`

Backup Slides

Stochastic gradient descent (SGD)

- Randomly select entry $\mathcal{R}(i,j,k)$ and update **A**, **B**, and **C**.
  - $\mathcal{O}(F)$ work per non-zero.

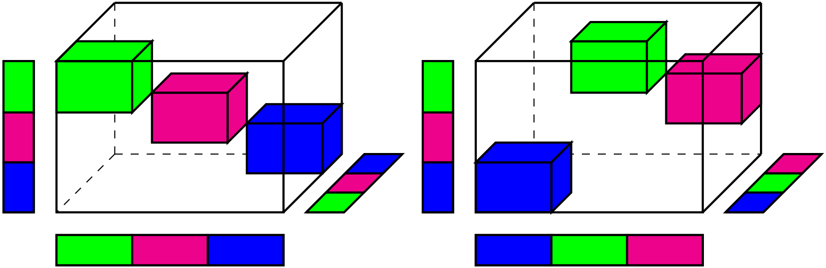
$$\delta_{ijk} \leftarrow \mathcal{R}(i,j,k) - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f)$$

$$\mathbf{A}(i,:) \leftarrow \mathbf{A}(i,:) + \eta\left[\delta_{ijk}\left(\mathbf{B}(j,:) * \mathbf{C}(k,:)\right) - \lambda\mathbf{A}(i,:)\right]$$
$$\mathbf{B}(j,:) \leftarrow \mathbf{B}(j,:) + \eta\left[\delta_{ijk}\left(\mathbf{A}(i,:) * \mathbf{C}(k,:)\right) - \lambda\mathbf{B}(j,:)\right]$$
$$\mathbf{C}(k,:) \leftarrow \mathbf{C}(k,:) + \eta\left[\delta_{ijk}\left(\mathbf{A}(i,:) * \mathbf{B}(j,:)\right) - \lambda\mathbf{C}(k,:)\right]$$

η is the step size; typically $\mathcal{O}(10^{-3})$.

Stratified SGD

- *Strata* identify independent blocks of non-zeros.
- Each stratum is processed in parallel.

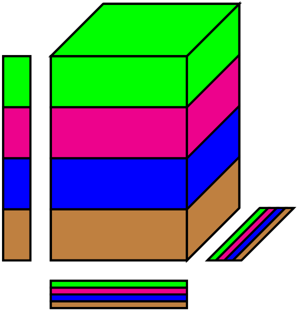


Limitations of stratified SGD:

- There is only as much parallelism as the smallest dimension.
- Sparsely populated strata are communication bound.

Asynchronous SGD (ASGD)

- Processes overlap updates and exchange to avoid divergence.
  - Local solutions are combined via a weighted sum.
- Go Hogwild! on shared-memory systems.

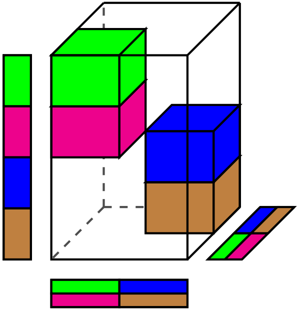


Limitations of ASGD:

- Convergence suffers unless updates are frequently exchanged.

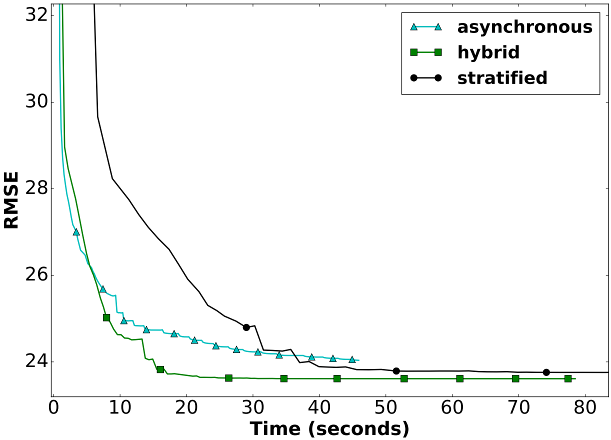
Hybrid stratified/asynchronous SGD

- Limit the number of strata to reduce communication.
- Assign multiple processes to the same stratum (called a *team*).
- Each process performs updates on its own local factors.
- At the end of a strata, updates are exchanged among the team.



Effects of stratification on SGD @ 1024 cores

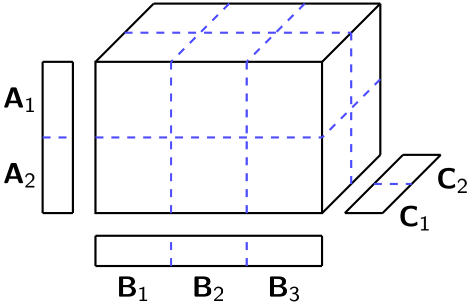
Hybrid stratification combines the speed of ASGD with the stability of stratification.



Hybrid uses sixteen teams of four MPI processes.

Parallel CCD++

- Shared-memory: each entry of $\mathbf{A}(:, f)$ is computed in parallel.
- Distributing non-zeros with a 3D grid limits communication to the grid layers.
  - Distributing non-zeros requires α_i and β_i to be aggregated.
  - Communication volume is $\mathcal{O}(IF)$ per process.
- For short modes, use a grid dimension of 1 and fully replicate the factor.



Alternating least squares (ALS)

- Normal equations $\mathbf{N}_i = \mathbf{H}_i^T \mathbf{H}_i$ are formed one non-zero at a time.
- $\mathbf{H}_i^T \operatorname{vec}(\mathcal{R}(i,:,:))$ is similarly accumulated into a vector $\mathbf{q}_i$.

Algorithm 1 ALS: updating $\mathbf{A}(i,:)$

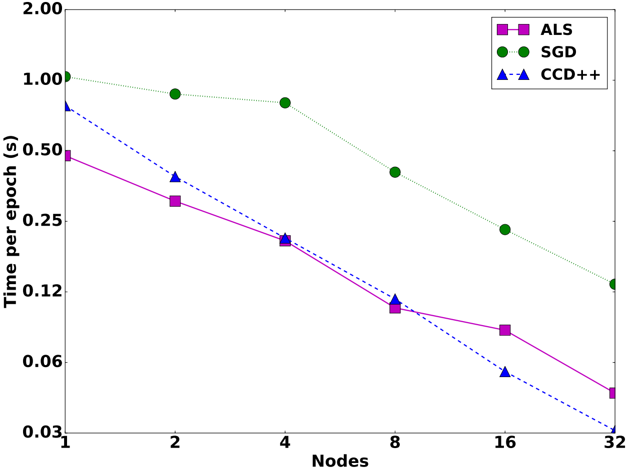
$\mathbf{N}_i \leftarrow \mathbf{0}^{F \times F}$, $\mathbf{q}_i \leftarrow \mathbf{0}^{F \times 1}$
for $(i,j,k) \in \mathcal{R}(i,:,:)$ **do**
    $\mathbf{x} \leftarrow \mathbf{B}(j,:) * \mathbf{C}(k,:)$
    $\mathbf{N}_i \leftarrow \mathbf{N}_i + \mathbf{x}^T \mathbf{x}$
    $\mathbf{q}_i \leftarrow \mathbf{q}_i + \mathcal{R}(i,j,k)\mathbf{x}^T$
end for
$\mathbf{A}(i,:) \leftarrow (\mathbf{N}_i + \lambda \mathbf{I})^{-1} \mathbf{q}_i$

CCD++ formulation

- $\mathcal{O}(F)$ work per non-zero.
- Each epoch requires NF passes over the tensor.
  - Heavily dependent on memory bandwidth.

$$
\begin{aligned}
\delta_{ijk} &\leftarrow \boldsymbol{\mathcal{R}}(i,j,k) - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \\
\alpha_i &\leftarrow \sum_{\boldsymbol{\mathcal{R}}(i,:,:)} \delta_{ijk} \left(\mathbf{B}(j,f)\mathbf{C}(k,f)\right) \\
\beta_i &\leftarrow \sum_{\boldsymbol{\mathcal{R}}(i,:,:)} \left(\mathbf{B}(j,f)\mathbf{C}(k,f)\right)^2 \\
\mathbf{A}(i,f) &\leftarrow \frac{\alpha_i}{\beta_i + \lambda}
\end{aligned}
$$

Netflix strong scaling



Communication volume on Yahoo!

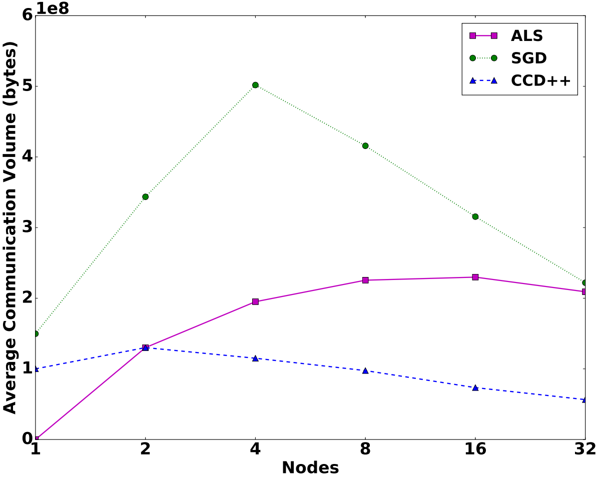
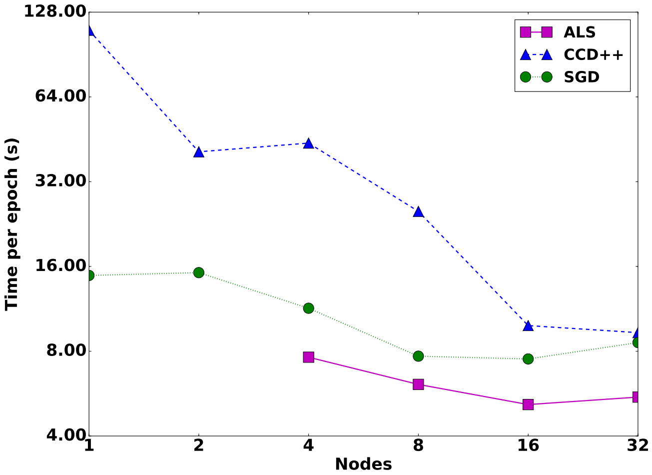


Figure: Average communication volume per node on the Yahoo! dataset. CCD++ and SGD use two MPI ranks per node and ALS uses one.

Amazon strong scaling



Scaling factorization rank on 1024 cores

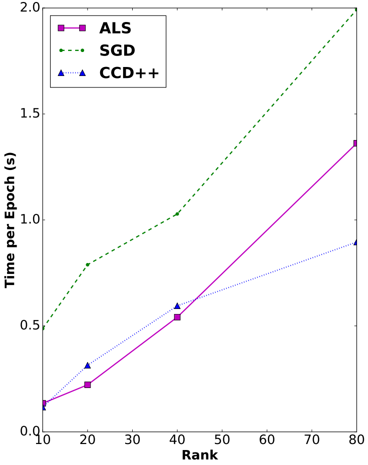


Figure: Effects of increasing factorization rank on the Yahoo! dataset.