An Exploration of Optimization Algorithms for High Performance Tensor Completion

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

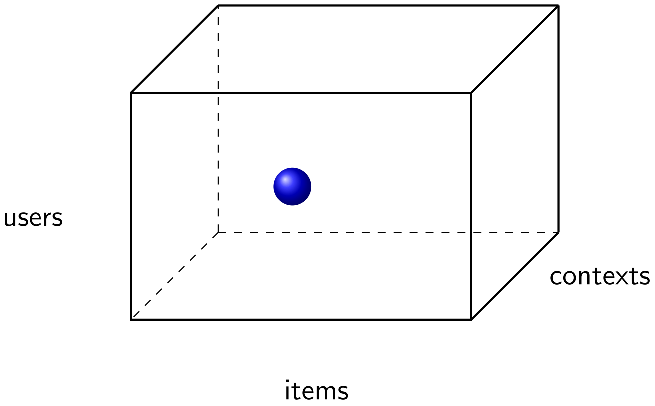
[1]Department of Computer Science & Engineering, University of Minnesota
[2]Parallel Computing Lab, Intel Corporation
*shaden@cs.umn.edu

Outline

Table of Contents

Tensor introduction

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

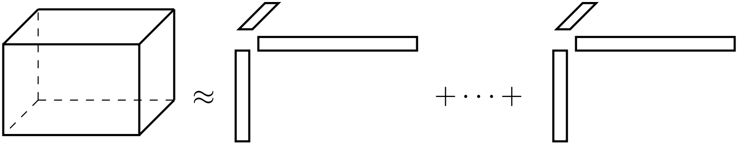


Tensor completion

- Many tensors are sparse due to missing or unknown data.
  - Missing values are *not* treated as zero.
- Assumption: the underlying data is low rank.
- Tensor completion estimates a low rank model to recover missing entries.
  - Applications: precision healthcare, product recommendation, cybersecurity, and others.

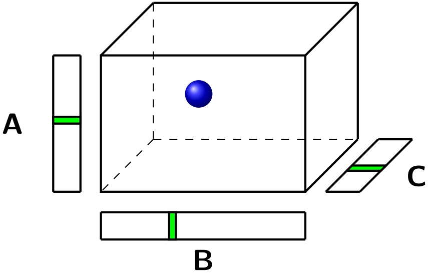
Tensor completion

- Many tensors are sparse due to missing or unknown data.
  - Missing values are *not* treated as zero.
- Assumption: the underlying data is low rank.
- Tensor completion estimates a low rank model to recover missing entries.
  - Applications: precision healthcare, product recommendation, cybersecurity, and others.
- The *canonical polyadic decomposition* (CPD) models a tensor as the summation of rank-1 tensors.



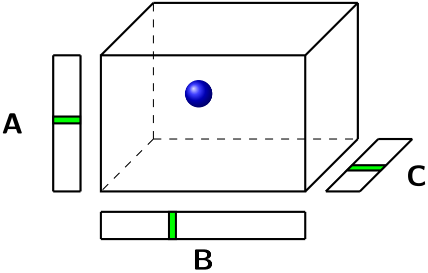
Tensor completion with the CPD

$\mathcal{R}(i,j,k)$ 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:

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

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

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. Number of operations.
  2. Convergence rate.
  3. Computational intensity.
  4. Parallelism.

Experimental setup

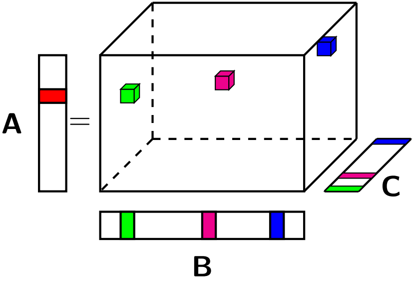
- Source code was implemented as part of SPLATT with MPI+OpenMP.
- Experiments are on the Cori supercomputer at NERSC.
  - Nodes have two sixteen-core Intel processors (Haswell).
- Experiments show a rank-10 factorization of the Yahoo Music (KDD cup) tensor.
  - 210 million *user-song-month* ratings.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

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

Table of Contents

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



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 $\mathbf{A}(i,:)$

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

BLAS-3 formulation

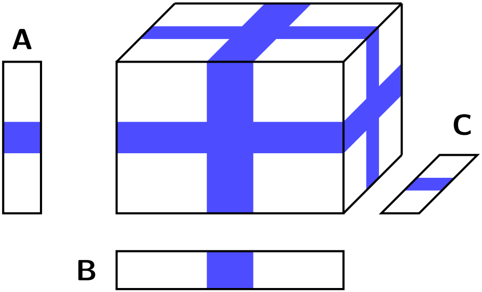
- Element-wise computation is an outer product formulation.
  - $\mathcal{O}(F^2)$ work with $\mathcal{O}(F^2)$ data per non-zero.
- Instead, append $(\mathbf{B}(j,:) * \mathbf{C}(k,:))$ to a matrix $\mathbf{Z}$.
  - When $\mathbf{Z}$ is full, do a rank-k update: $\mathbf{N}_i \leftarrow \mathbf{N}_i + \mathbf{Z}^T\mathbf{Z}$.

Algorithm 2 ALS: updating $\mathbf{A}(i,:)$

```
1: N_i ← 0^{F×F}, q_i ← 0^{F×1}, Z ← 0
2: for (i, j, k) ∈ R(i, :, :) do
3:     Append (x ← B(j, :) * C(k, :)) to Z
4:     q_i ← q_i + R(i, j, k) x^T
5: end for
6: N_i ← N_i + Z^T Z
7: A(i, :) ← (N_i + λI)^{-1} q_i
```

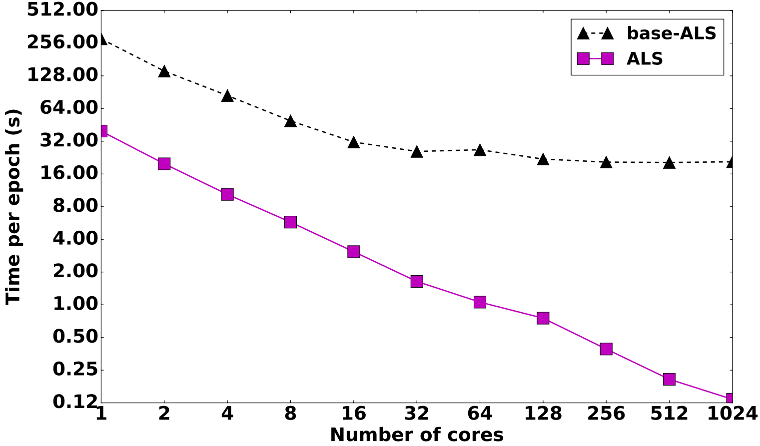
Parallel ALS

- We impose a 1D partition on each of the factors.
- Non-zeros are then distributed according to the row partitionings.
- Only the updated rows need to be communicated.
- If mode is short, cooperatively form rows and aggregate the normal equations.



ALS evaluation

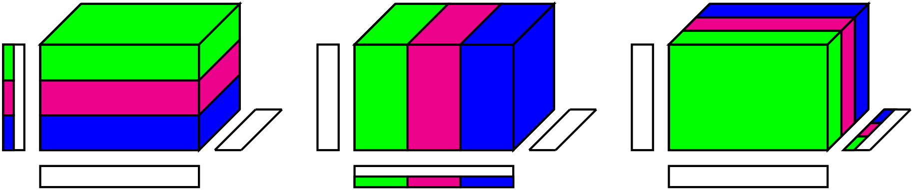
295× relative speedup and 153× speedup over base-ALS.



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

Coordinate descent (CCD++)

- Select an variable and update while holding all others constant.
  - Can be done exactly because problem is quadratic.
- Rank-1 factors are updated in sequence.



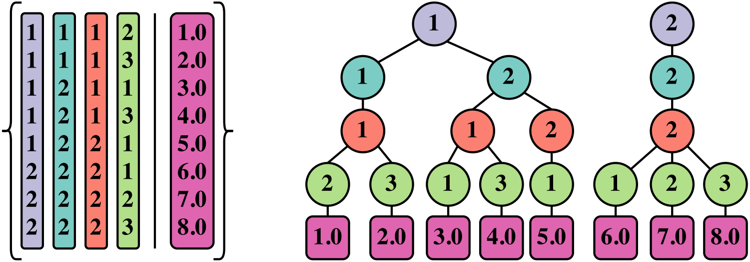
CCD++ formulation

- $\mathcal{O}(F)$ work per non-zero.
- Each epoch requires $3F$ passes over the tensor.
  - Heavily dependent on memory bandwidth.

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

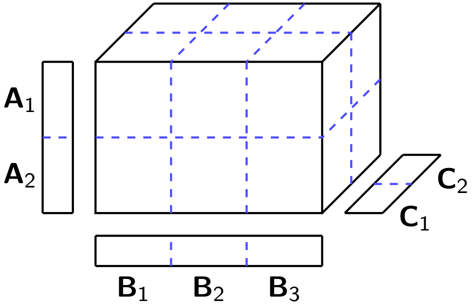
Compressed sparse fiber (CSF)

- CSF is a generalization of the CSR structure for matrices.
- Paths from roots to leaves encode non-zeros.
- CSF reduces the memory bandwidth of the tensor and also structures accesses to the factors.



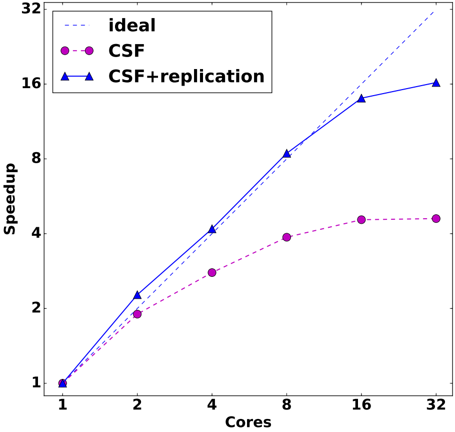
Parallel CCD++

- 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.



CCD++ shared-memory evaluation

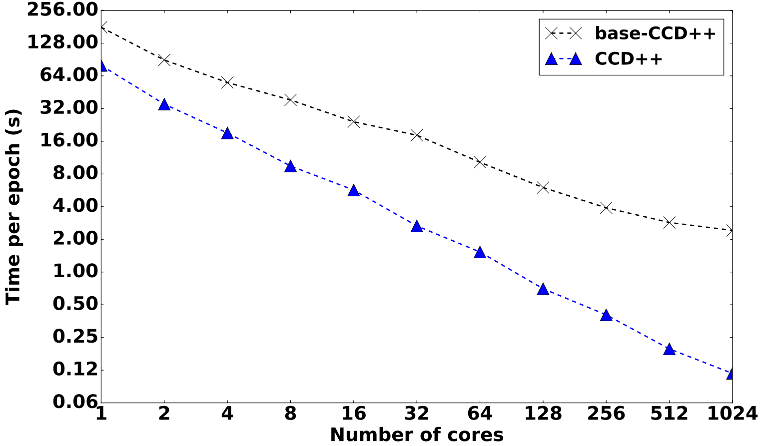
Replicating the short mode improves speedup from 4× to 16×.



CSF uses a CSF representation. **CSF+replication** uses a CSF representation and replicates α and β for load balance.

CCD++ distributed-memory evaluation

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



base-CCD++ is a pure-MPI implementation in C++ [Karlsson '15]. **CCD++** is our MPI+OpenMP implementation with two MPI ranks per node.

Stochastic gradient descent (SGD)

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

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

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

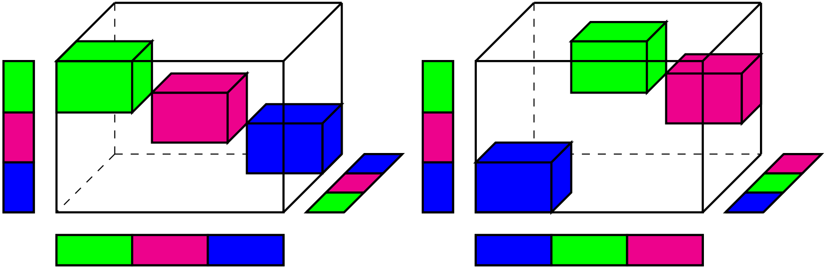
$$\mathbf{B}(j,:) \leftarrow \mathbf{B}(j,:) + \eta \left[\delta_{ijk} \left(\mathbf{A}(i,:) * \mathbf{C}(k,:)\right) - \lambda \mathbf{B}(j,:)\right]$$

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

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

Stratified SGD [Beutel '14]

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

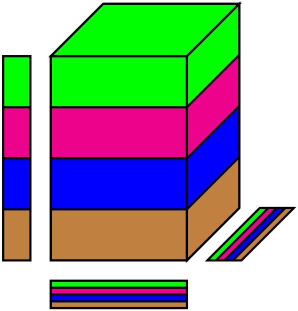


Limitations of stratified SGD:

- 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.

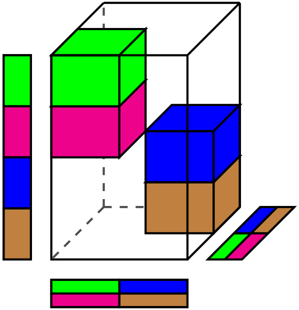


Limitations of ASGD:

- Convergence suffers unless updates are frequently exchanged.

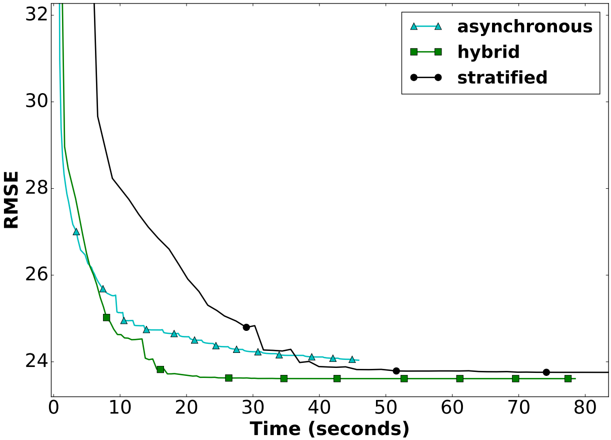
Hybrid stratified/asynchronous SGD

- 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.

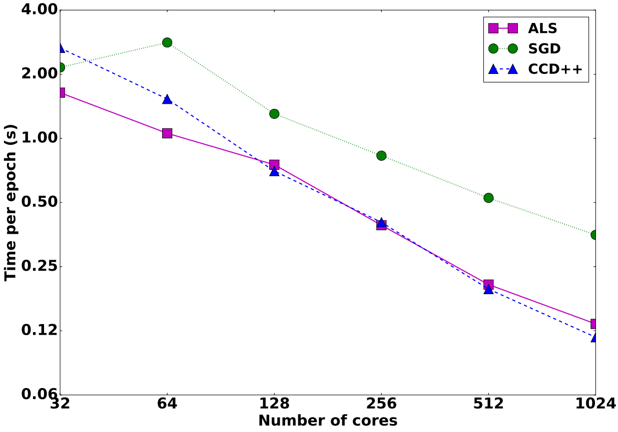


Hybrid uses sixteen teams of four MPI processes.

Table of Contents

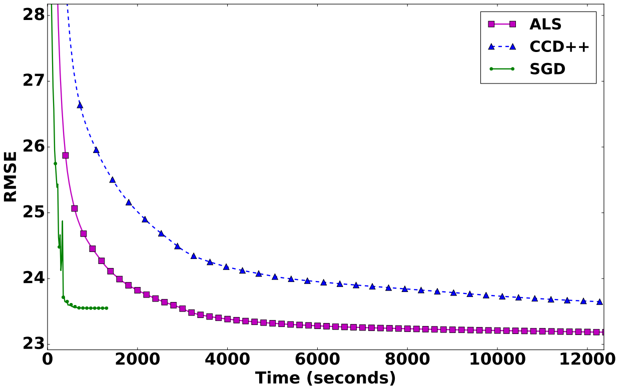
Strong scaling

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



Convergence @ 1 core

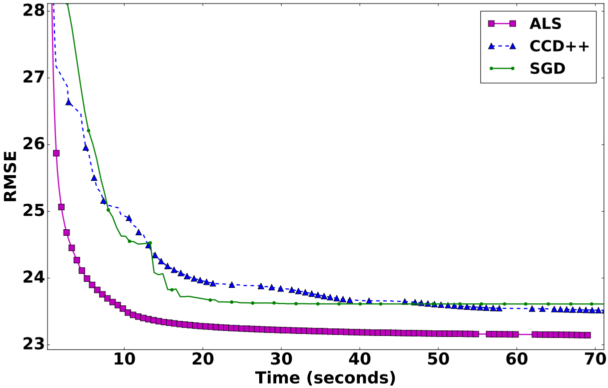
SGD rapidly converges to a high quality solution.



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.

Table of Contents

Wrapping Up

- Some of the principles in sparse matrix computations are useful here, but tensors bring many new challenges!
- Careful attention to sparsity and data structures can give over $10\times$ speedups.
- ALS has a high convergence rate and performs well on modern architectures due to its high compute intensity.
- CCD++ may be best for very large scale systems or ranks, however.

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

Questions?

Backup Slides

Communication volume on Yahoo!

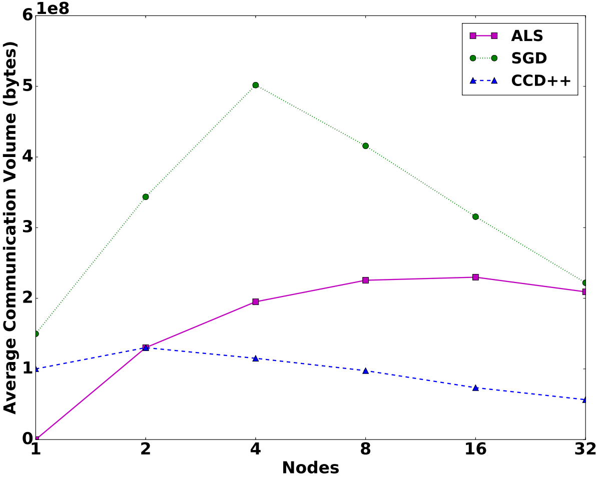
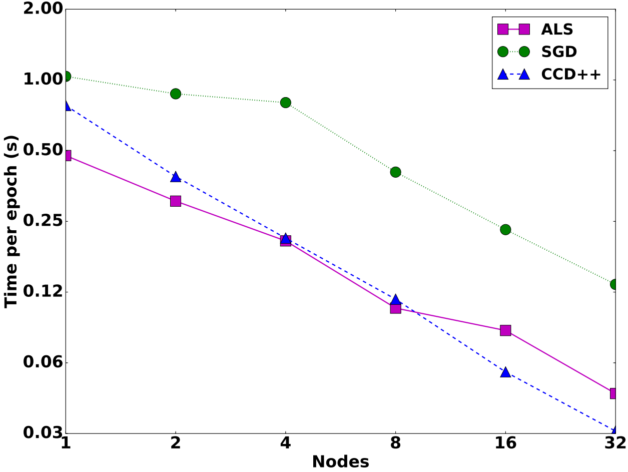
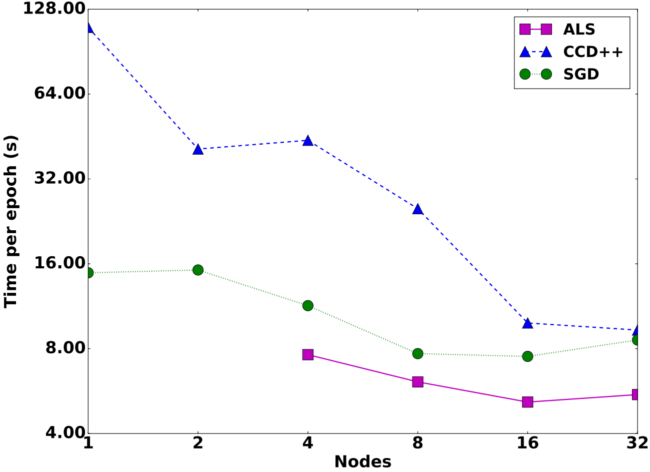


Figure: Average communication volume per node on the Yahoo! dataset. CCD++ and SGD use two MPI ranks per node and ALS uses one.

Netflix strong scaling



Amazon strong scaling



Scaling factorization rank on 1024 cores

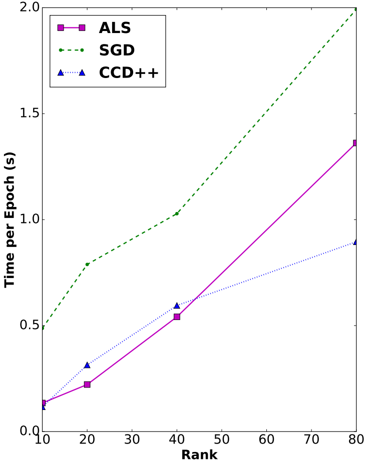


Figure: Effects of increasing factorization rank on the Yahoo! dataset.