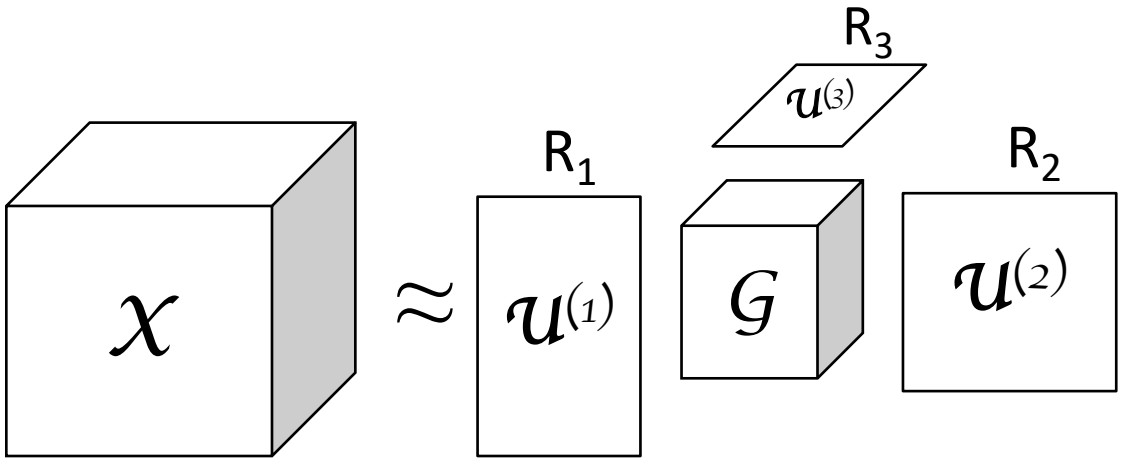
UNIVERSITY OF OREGON

Optimizing Tensor Decomposition on HPC Systems – Challenges and Approaches

Jee Whan Choi
Dept. of Computer and Information Science
University of Oregon

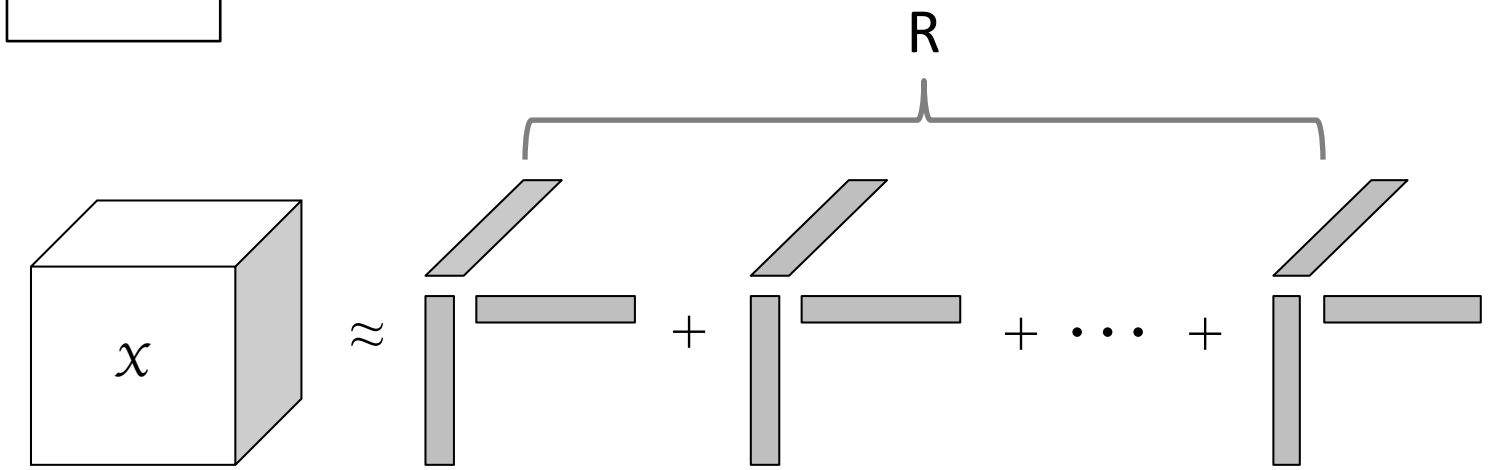
IPDPS HIPS 2019
May 20th 2019, Rio de Janeiro, Brazil

Two popular tensor decomposition algorithms



Tucker

Canonical Polyadic (CP)

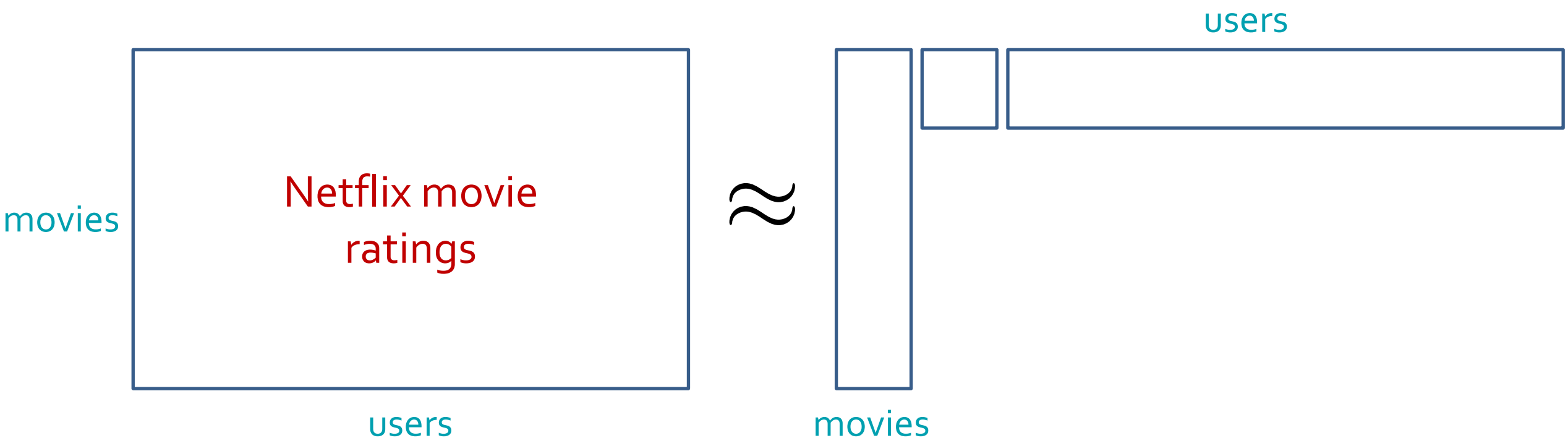


Why tensor decomposition (TD)?

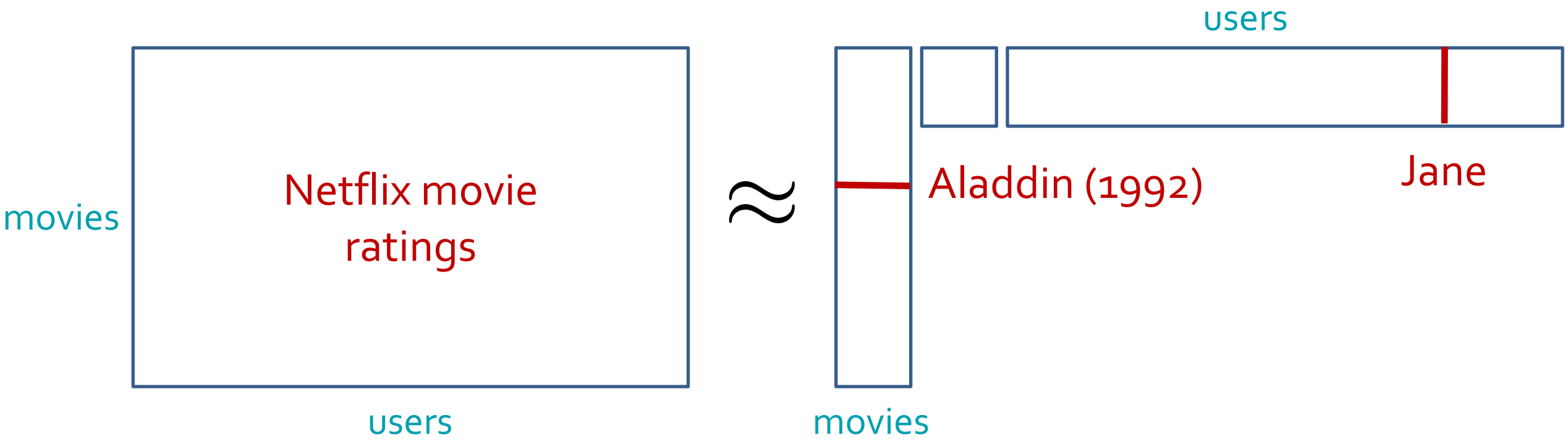
- Natural way of representing data with multi-way relationship
  - Social networks, healthcare records, product reviews, and more
- Latent variable modeling
  - Hidden Markov Model
  - Mixture of Gaussian
  - Latent Dirichlet Allocation (LDA)

Tensor decomposition is analogous to SVD

Tensor decomposition is analogous to SVD

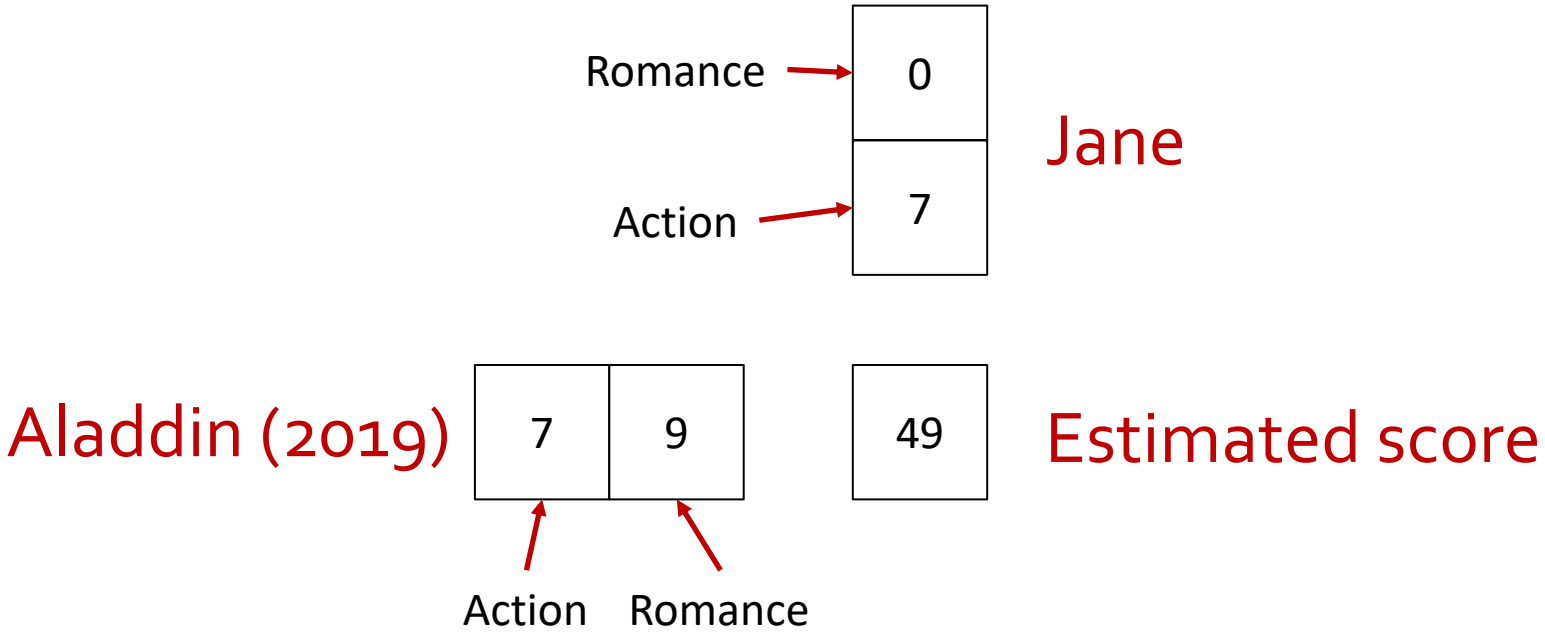


Tensor decomposition is analogous to SVD



Estimating the “score” is as simple as taking the dot product

- Let’s say movies only have two “latent” properties – action and romance



Why tensor decomposition?

- Pros
  - Matrix factorization is **not** unique whereas tensor decomposition is unique (given some conditions)
  - Retains the multi-way relationship that is typically lost when formulated as a matrix problem
- Cons
  - Determining the rank is NP-hard
  - Thinking in higher dimensions is difficult

Applications of tensor decomposition

- Signal processing
  - Signal separation, code division
- Data analysis
  - Phenotyping (electronic health record), network analysis, data compression
- Machine learning
  - Latent variable model (natural language processing, topic modeling, recommender systems, etc.)
  - Neural network compression

My prior work has covered both dense and sparse data for CP and Tucker

- Sparse tensors
  - Blocking sparse tensors on shared- and distributed-memory systems for CPD
  - Workload balancing for sparse Tucker on distributed-memory systems
- Dense tensors
  - Using GPUs to accelerate dense Tucker algorithms
  - Workload balancing for dense Tucker on distributed-memory systems

In 2015, HPC research in sparse TD focused on flops

- Naïve kernel
  - Regular: **3 * m * R** flops (2mR for initial product + scale, mR for accumulation)
- CSF
  - **2R(m + P)** flops, P is # of non-empty fibers
  - typically p <<< m
- DFacTo
  - Formulates kernel as SpMV
  - Each column is computed independently via 2 SpMV
  - **2R(m + P)** flops
- GigaTensor
  - MapReduce
  - Increased parallelism, but more flops
  - **5mR** flops

m = # of non-zeros
P = # of non-empty fibers
R = rank

Does this make sense for ***sparse*** data?

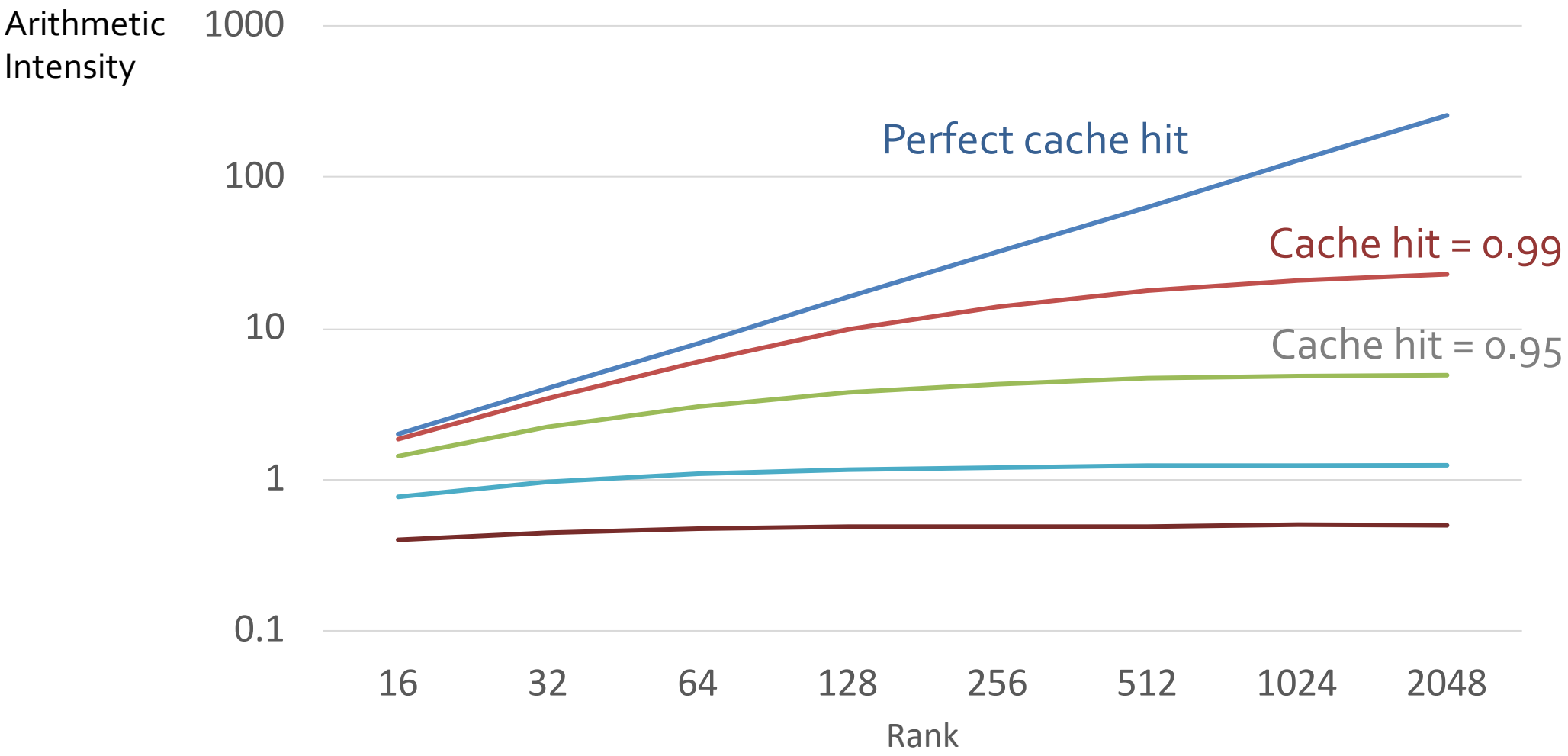
- Sparse computations are generally memory bandwidth-bound
- So, why was CSF, DFacto giving better performance?

Roofline model applied to CSF MTTKRP

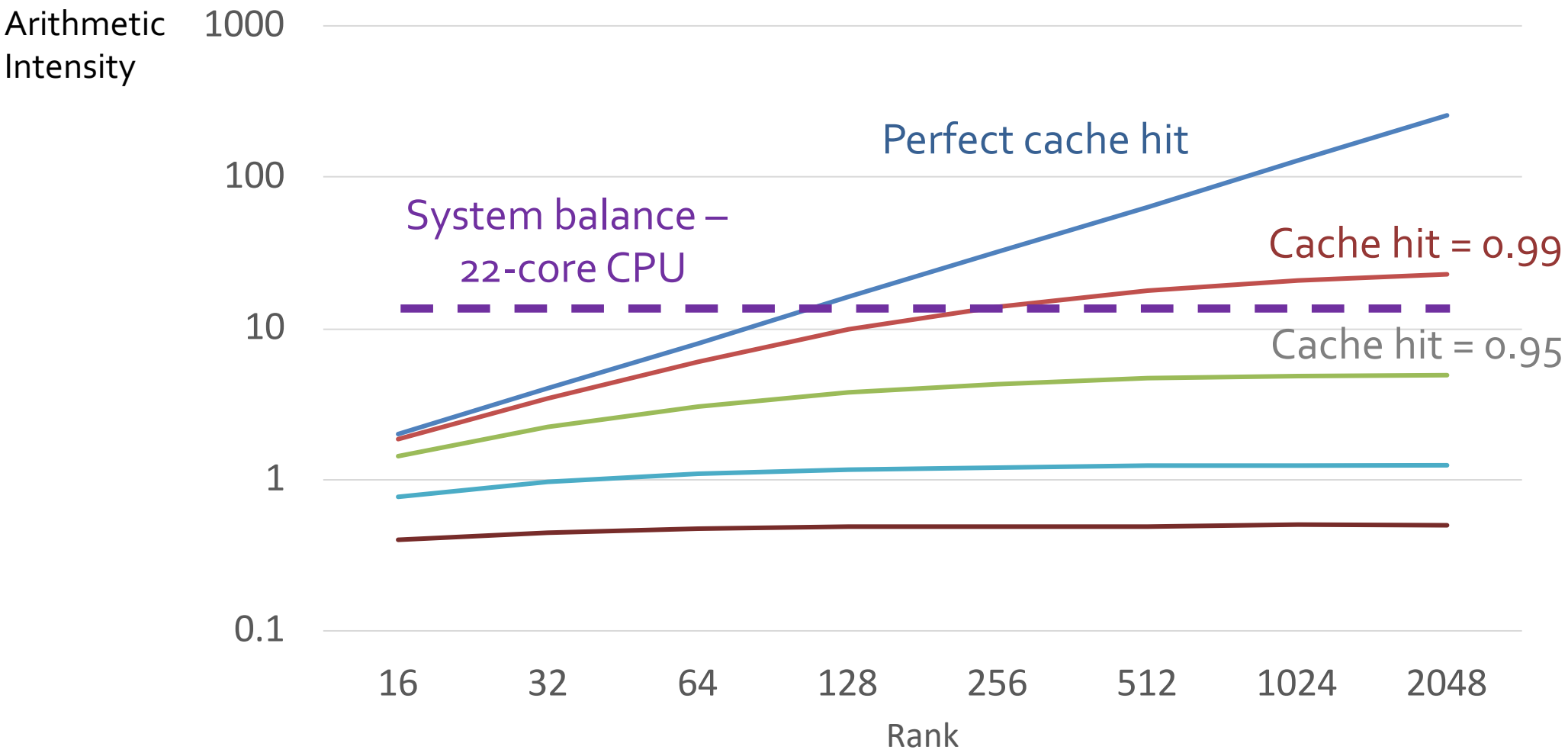
- Let's calculate the # of flops and # of bytes and compare
  - Flops: W = **2R(m + P)**
  - Data: Q = **2m** (value + mode-2 index) + **2P** (mode-3 index + mode-3 pointer)

    + **(1-α)Rm** (mode-2 factor) + **(1-α)RP** (mode-3 factor)
- Arithmetic Intensity
  - Ratio of work to communication **I = W/Q**
  - I = W / (Q * 8 Bytes) = **R / (8 + 4R(1-α))**

Arithmetic intensity vs. system balance (on the latest CPU)



Arithmetic intensity vs. system balance (on the latest CPU)



Pressure point analysis (PPA) reveals the bottlenecks

- Pressure point analysis
  - Probe potential bottlenecks by creating and eliminating instructions/data access
  - If we suspect that # of registers is the bottleneck, try increasing/decreasing their usage to see if the exec. time changes.
  - Source code & assembly instrumentation – e.g., inline assembly to prevent dead code elimination (DCE)

Kenneth Czechowski, **Performance Analysis Using the Pressure Point Analysis**, PhD dissertation

We start from the baseline implementation

Time	Pressure point
2.6s	Baseline (2R(m + P) flops)

Using COO instead of CSF only increases exec. time by < 2%

Time	Pressure point
2.6s	Baseline (2R(m + P) flops)
2.64s	Move flops to inner loop (3 * m * R flops)

Increasing flops only changes time by < 2%

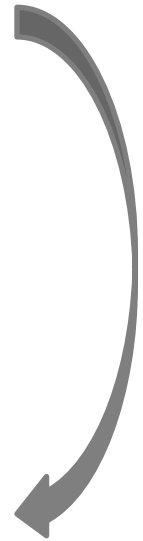
Removing access to C (accessed once per fiber): exec. time down by 7%

Time	Pressure point
2.6s	Baseline (2R(m + P) flops)
2.64s	Move flops to inner loop (3 * m * R flops)
2.43s	Access to C removed

Removing per-fiber access to matrix C has a bigger impact than increasing flops

Suspicion confirmed: memory access to B is the bottleneck

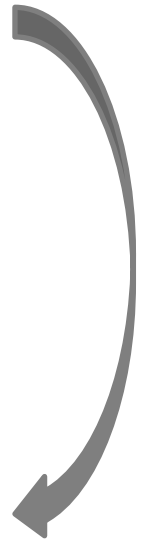
Time	Pressure point
2.6s	Baseline (2R(m + P) flops)
2.64s	Move flops to inner loop (3 * m * R flops)
2.43s	Access to C removed
1.81s	Access to B limited to L1 cache



Limiting our suspect has a **huge** impact

Completely removing it give us an extra 6% - why?

Time	Pressure point
2.6s	Baseline (2R(m + P) flops)
2.64s	Move flops to inner loop (3 * m * R flops)
2.43s	Access to C removed
1.81s	Access to B limited to L1 cache
1.63s	Access to B removed completely



Eliminating it completely gives us an extra 6% boost

Conclusions from our empirical analysis

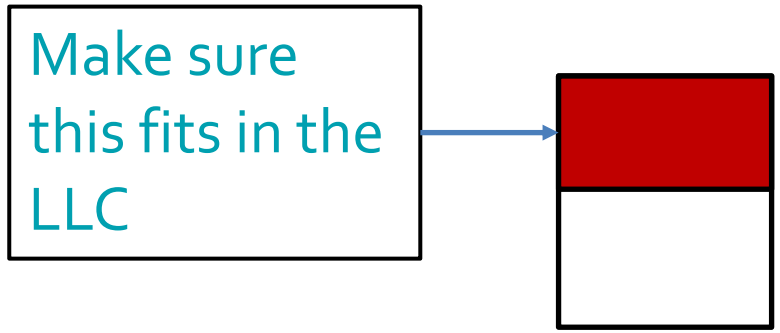
- Flops aren't the issue
- Bottlenecks
  1. Data access to factor matrix B (and not the tensor e.g., SpMV)
  2. Load instructions (why previous attempt at cache blocking was not successful)

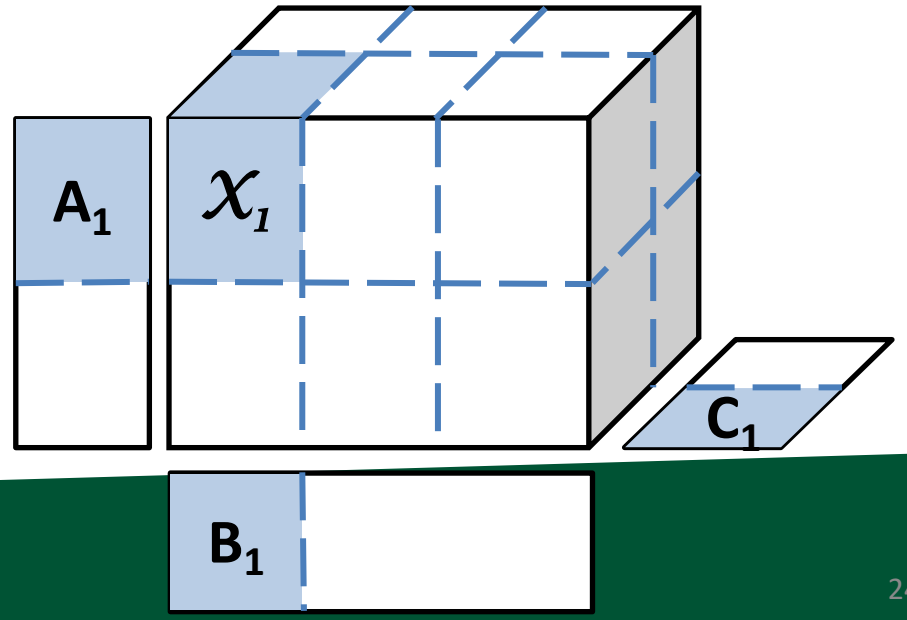
Conclusions from our empirical analysis

- Flops aren't the issue
- Bottlenecks
  1. Data access to factor matrix B (and not the tensor) → cache blocking
  2. Load instructions (why previous attempt at cache blocking was not successful) → register blocking

We use n-D blocking (intuitive) and rank blocking (less intuitive)

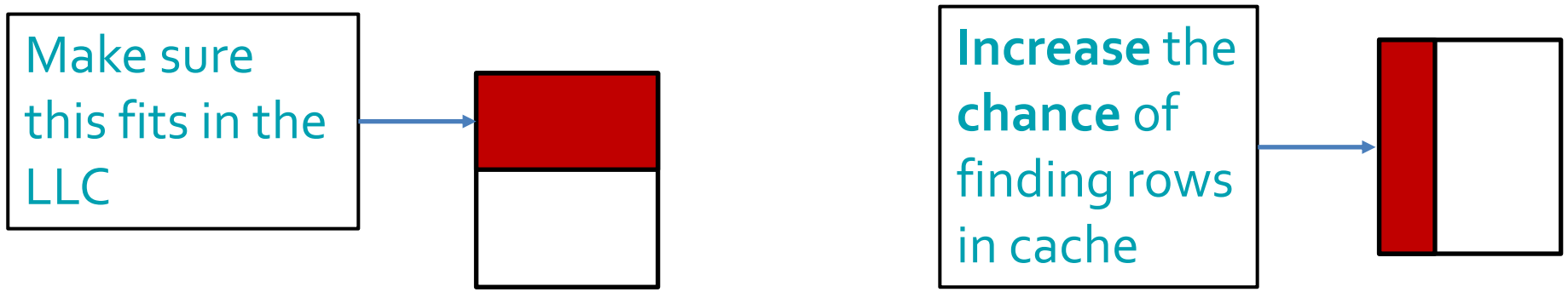
- Multi-dimensional blocking
  - **3D blocking – maximize re-use of both matrix B and C**
  - **Multiple access to the factor matrices**
- Rank blocking





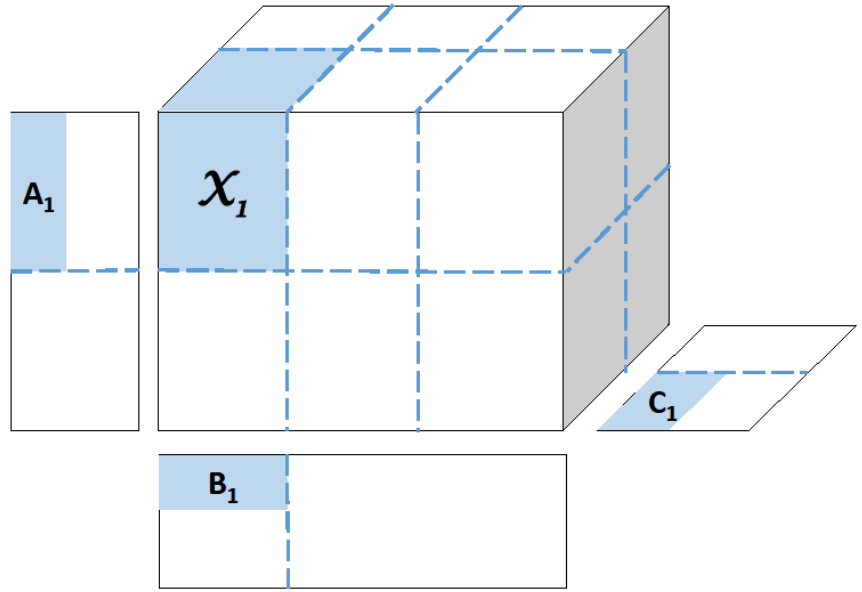
We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
  - Multiple access to the factor matrices
- Rank blocking
  - **Agnostic to tensor sparsity**
  - **Very little change to the code**
  - **Requires tensor replication**



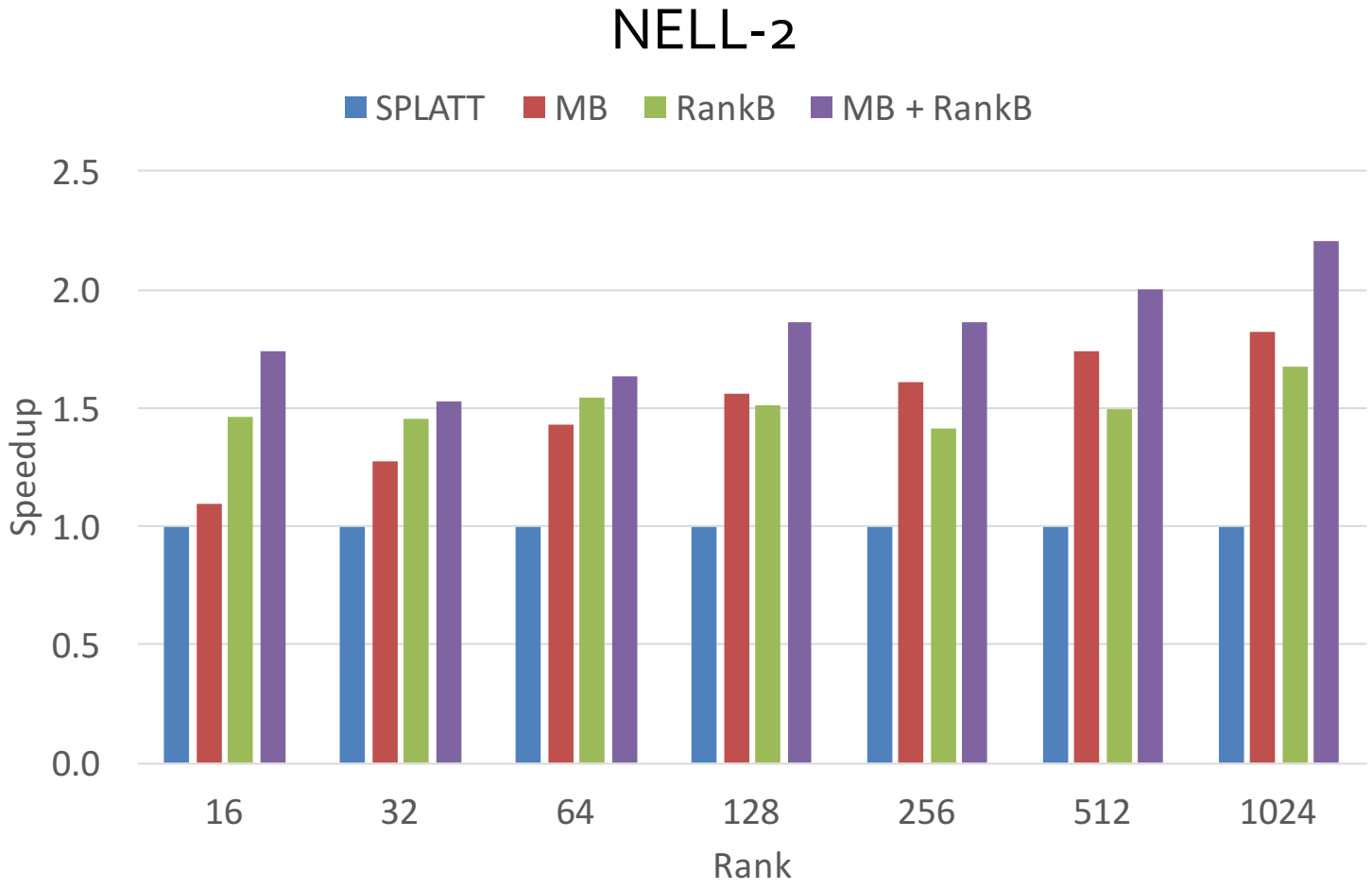
We can combine n-D blocking with rank blocking

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
  - Multiple access to the factor matrices
- Rank blocking
  - Agnostic to tensor sparsity
  - Very little change to the code
  - Requires tensor replication
- Multi-dimensional + rank blocking
  - **Partial replication**
  - **"Best of both worlds" re-use**
  - **Even more repeated accesses to tensor/factor**



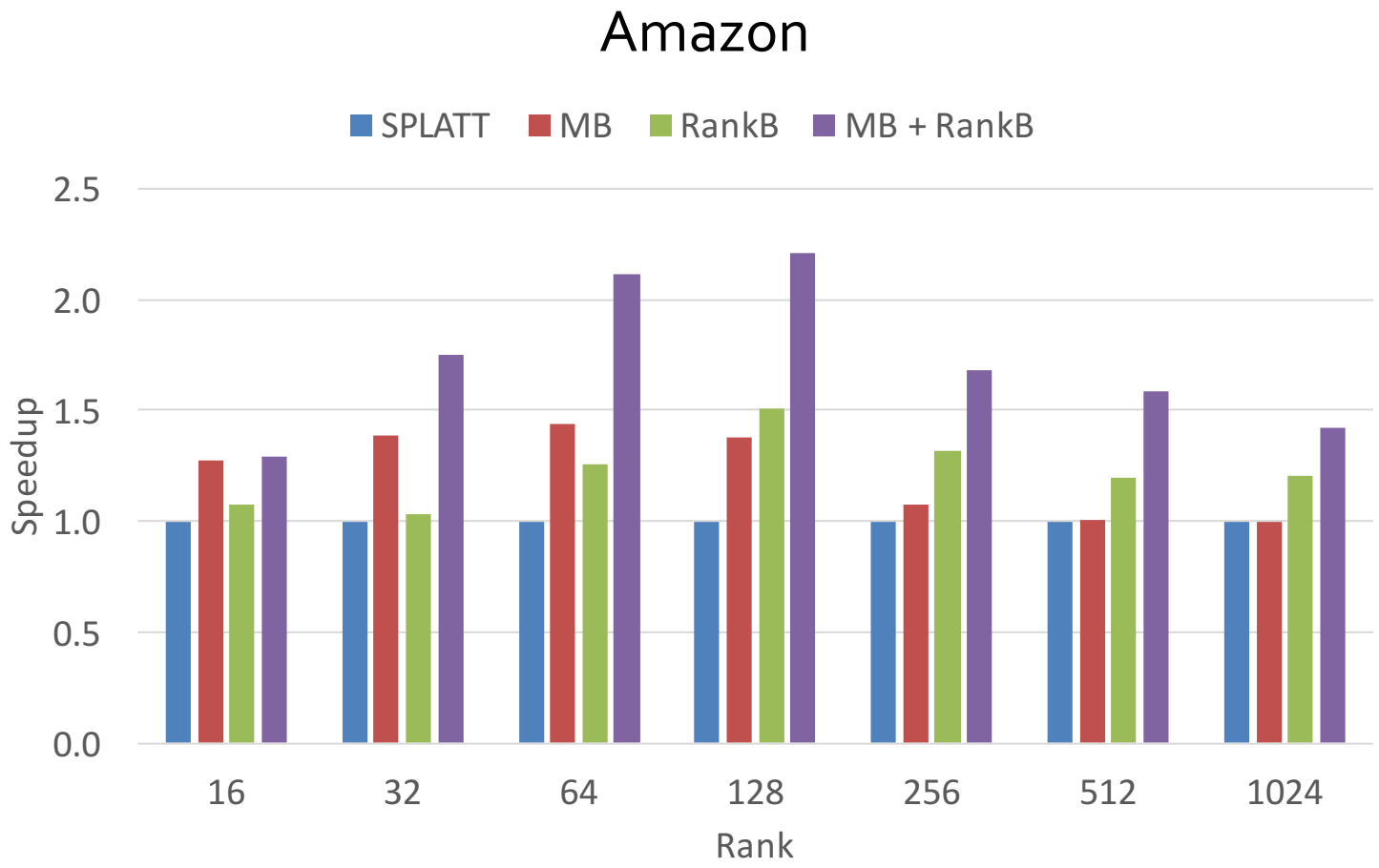
For small tensors, blocking becomes more effective at higher rank sizes

- With small dimension sizes, there is already good cache re-use without explicit blocking
- Only when rank size is large enough, do we see significant benefit from blocking



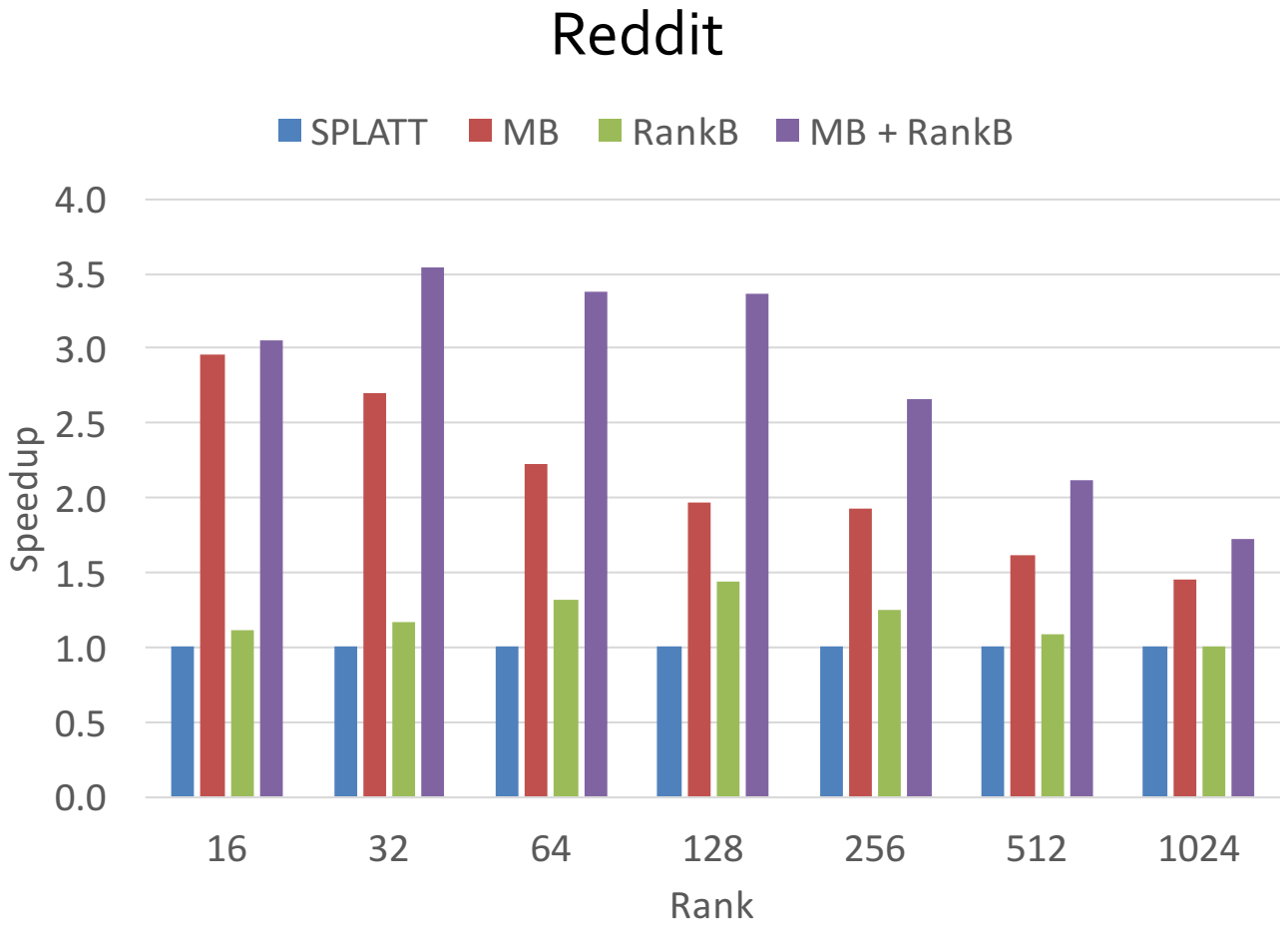
For large tensors, blocking becomes less effective at higher ranks

- With large dimension sizes **and** large ranks, data sets are so big large number of blocks are required, and the **overhead** of blocking outweighs the benefit



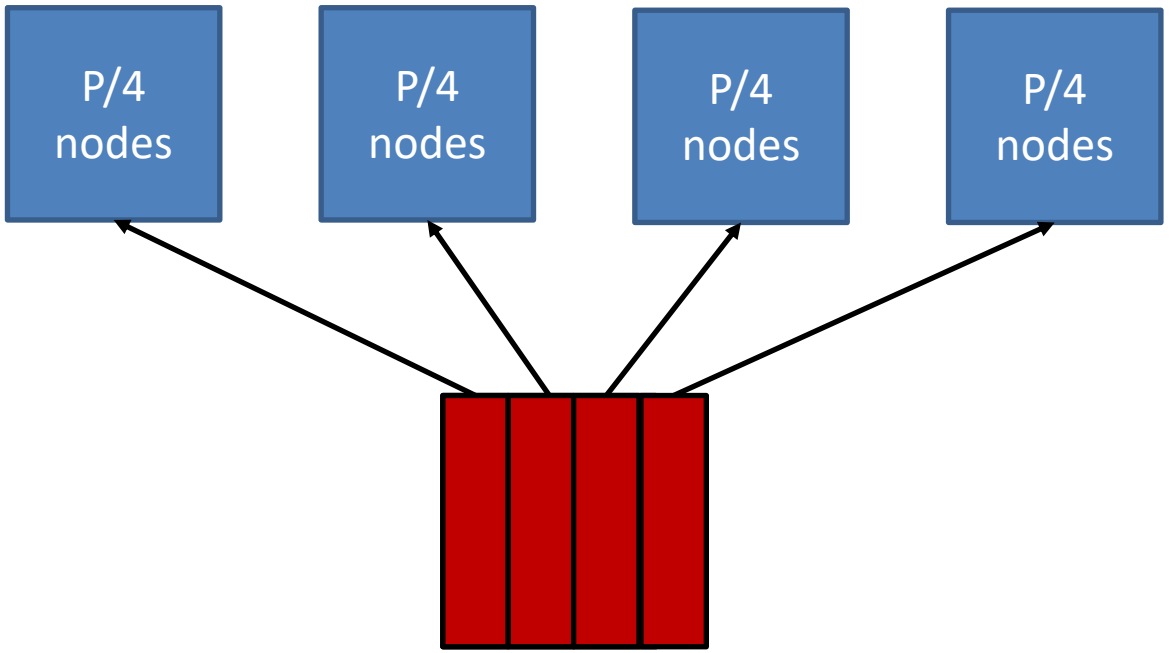
More potential benefit from blocking with real data sets

- Real data sets have clustering patterns which lead to higher speedups from blocking
- Combining rank blocking with n-D blocking yields the highest speedup

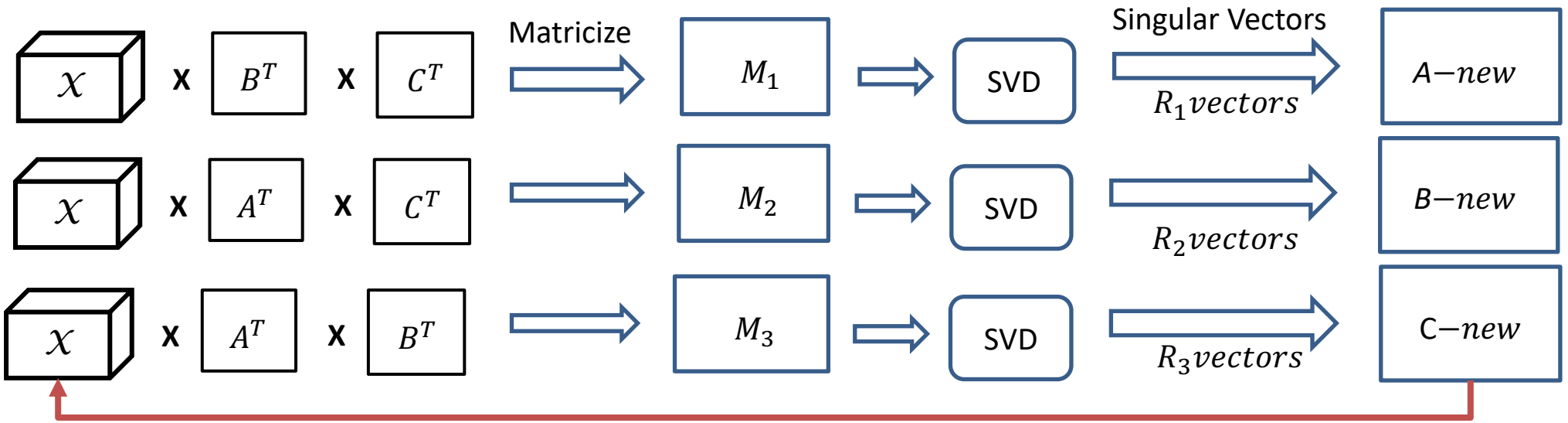


Rank blocking on distributed systems

- Scalability problems with traditional partitioning
  - Fewer non-zero per node ⟶ lower efficiency & higher comm. cost ⟶ poor scalability
- Rank blocking
  - No comm. between processor sets
  - Tensor replication



Higher Order Orthogonal Iteration (HOOI) Tucker algorithm



Alternating least squares (ALS)

Sparse HOOI – key kernels

- TTM
  - Computation only - all schemes have same computational load (i.e., FLOPs)
  - Load balance
- SVD
  - Both computation and communication
  - Both computational load and communication volume are determined by load balance
- Factor Matrix Transfer (FMT)
  - Communication only
  - At the end of each HOOI invocation, factor matrix rows need to be communicated among processors for the next invocation
  - Communication volume

Prior schemes for tensor distribution compared

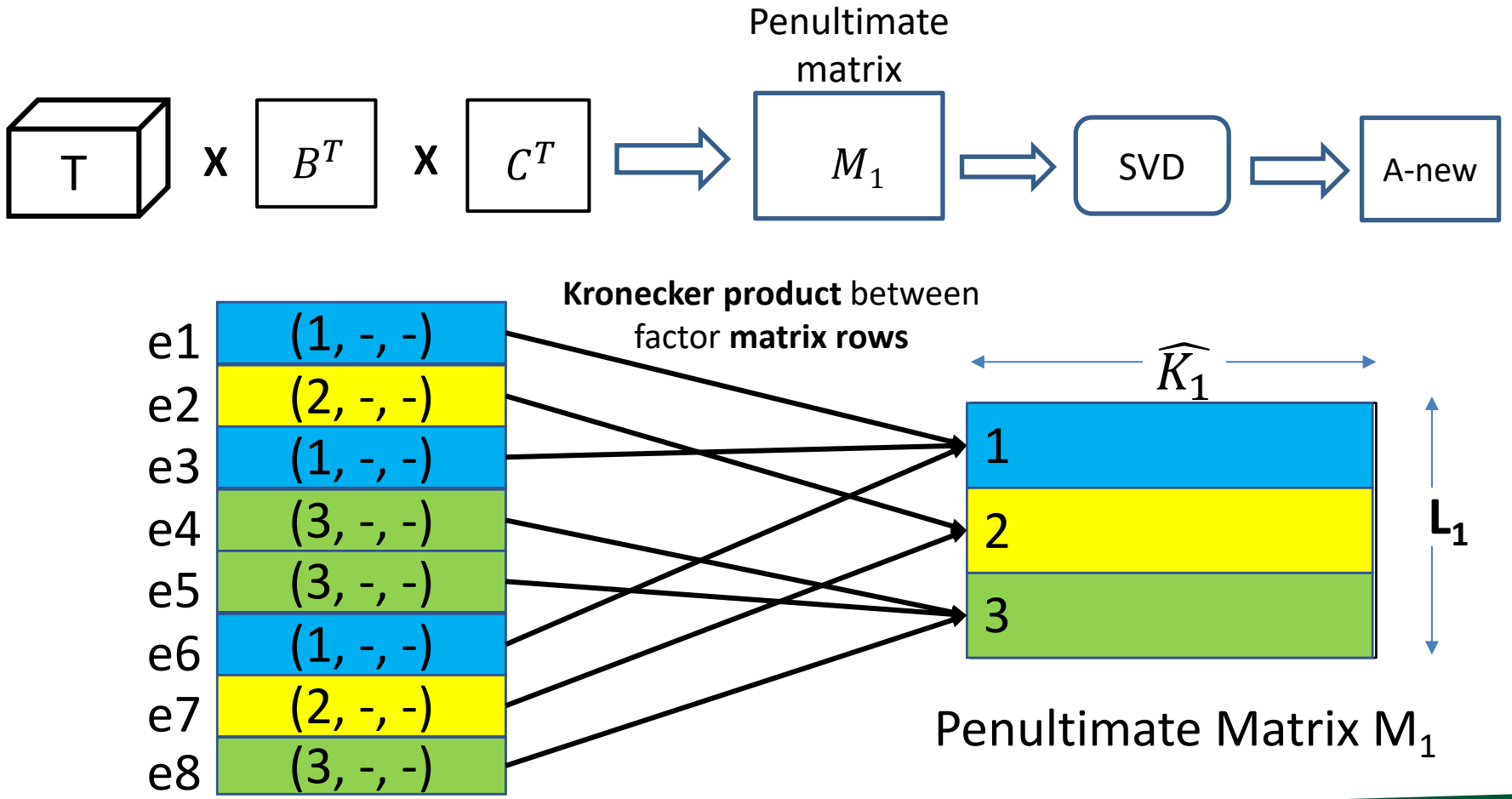
- Coarse - Coarse grained schemes [KU'16]
  - Allocate entire "slices" to processors
- Medium - Medium grained scheme [SK '16]
  - Grid based partitioning – similar to block partitioning of matrices
- Fine - Fine grained scheme [KU'16]
  - Allocate individual elements using hypergraph partitioning methods

	TTM	SVD	FMT	Dist. Time
Coarse	Inefficient	Efficient	Inefficient	Fast
Medium	Efficient	Inefficient	Efficient	Fast
Fine	Efficient	Inefficient	Efficient	Slow

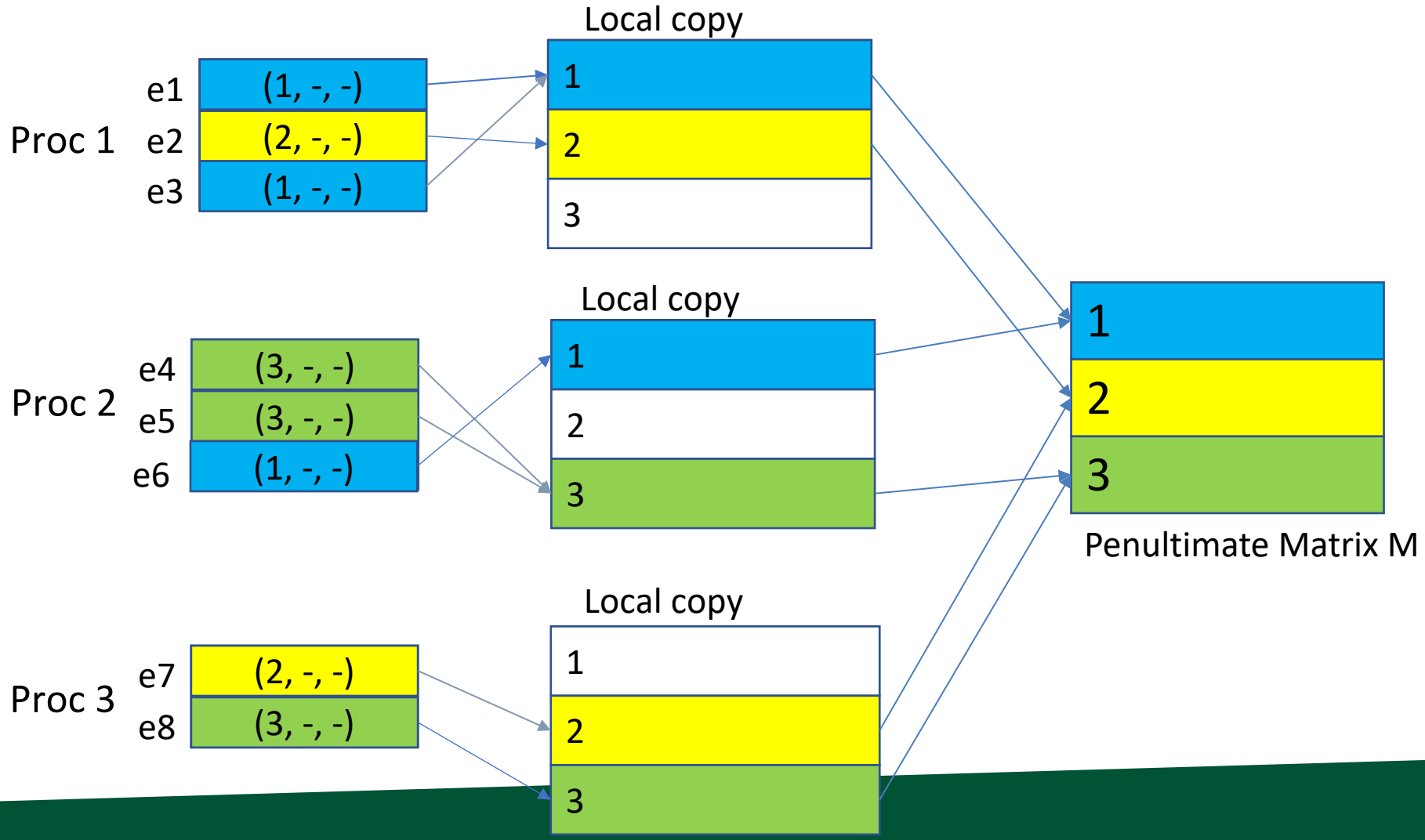
Our scheme – lite distribution – achieves better workload distribution

- Lite
  - Near optimal on TTM and SVD (both computation and communication)
  - Lightweight (i.e., fast distribution time)
  - Not optimal on FMT (but this is cheap)
  - Performance gain up to 3×

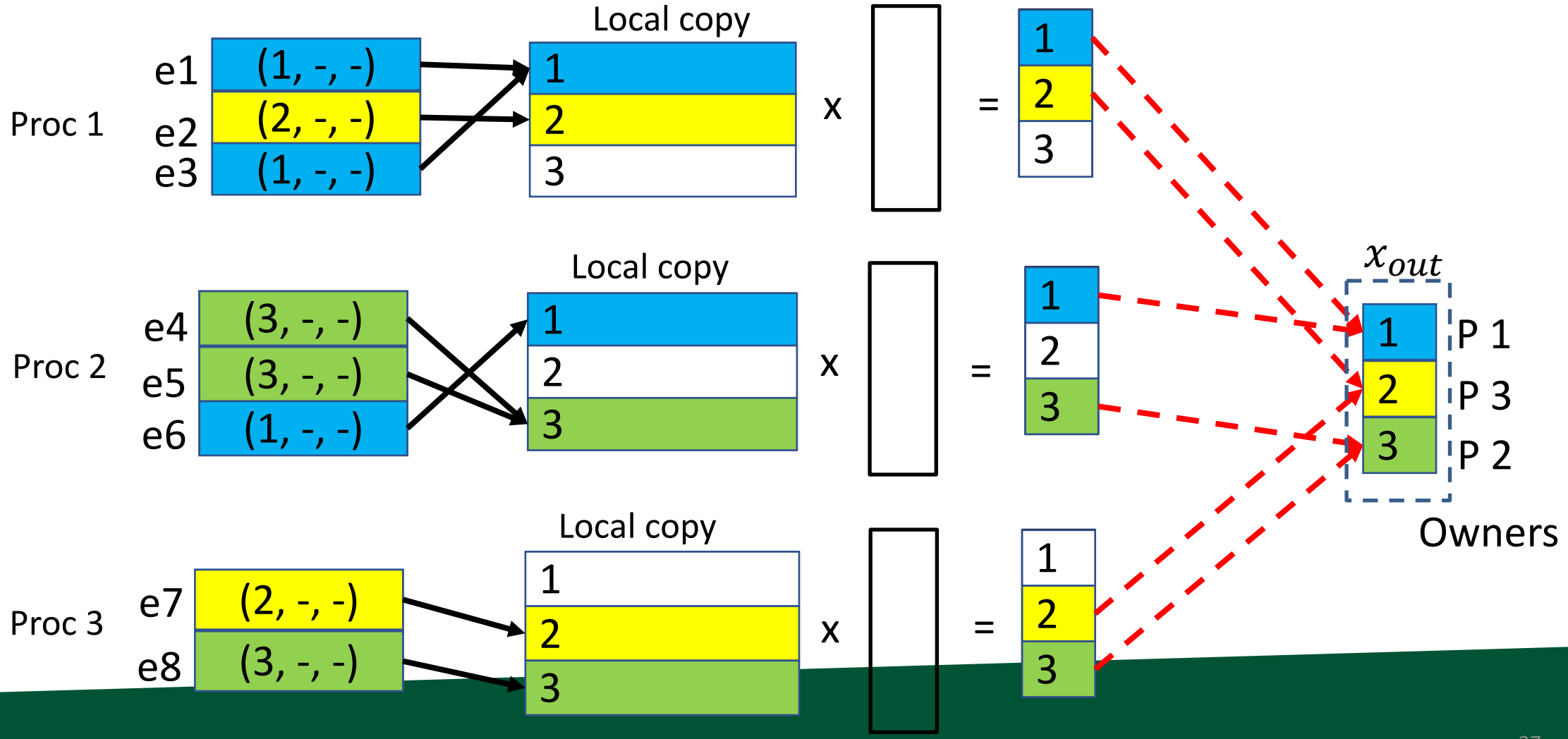
Example – sequential sparse TTM for mode 1



Example – distributed sparse TTM for mode 1



Example – SVD via the Lanczos method



Performance metrics (along each mode)

TTM

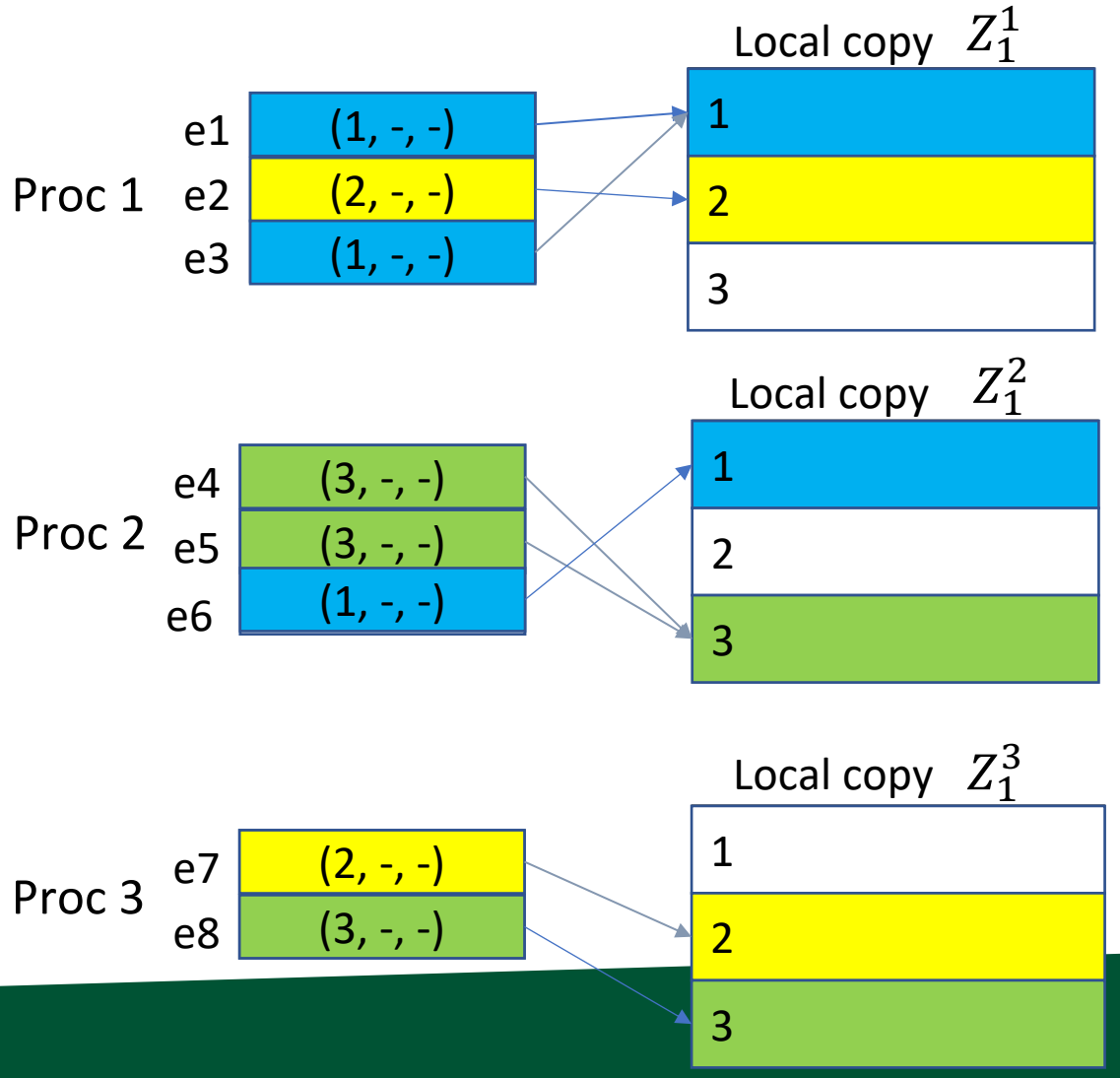
- TTM-LImb (Load Imbalance)
  - Max number of elements assigned to the processors
  - Optimal value – E / P

SVD

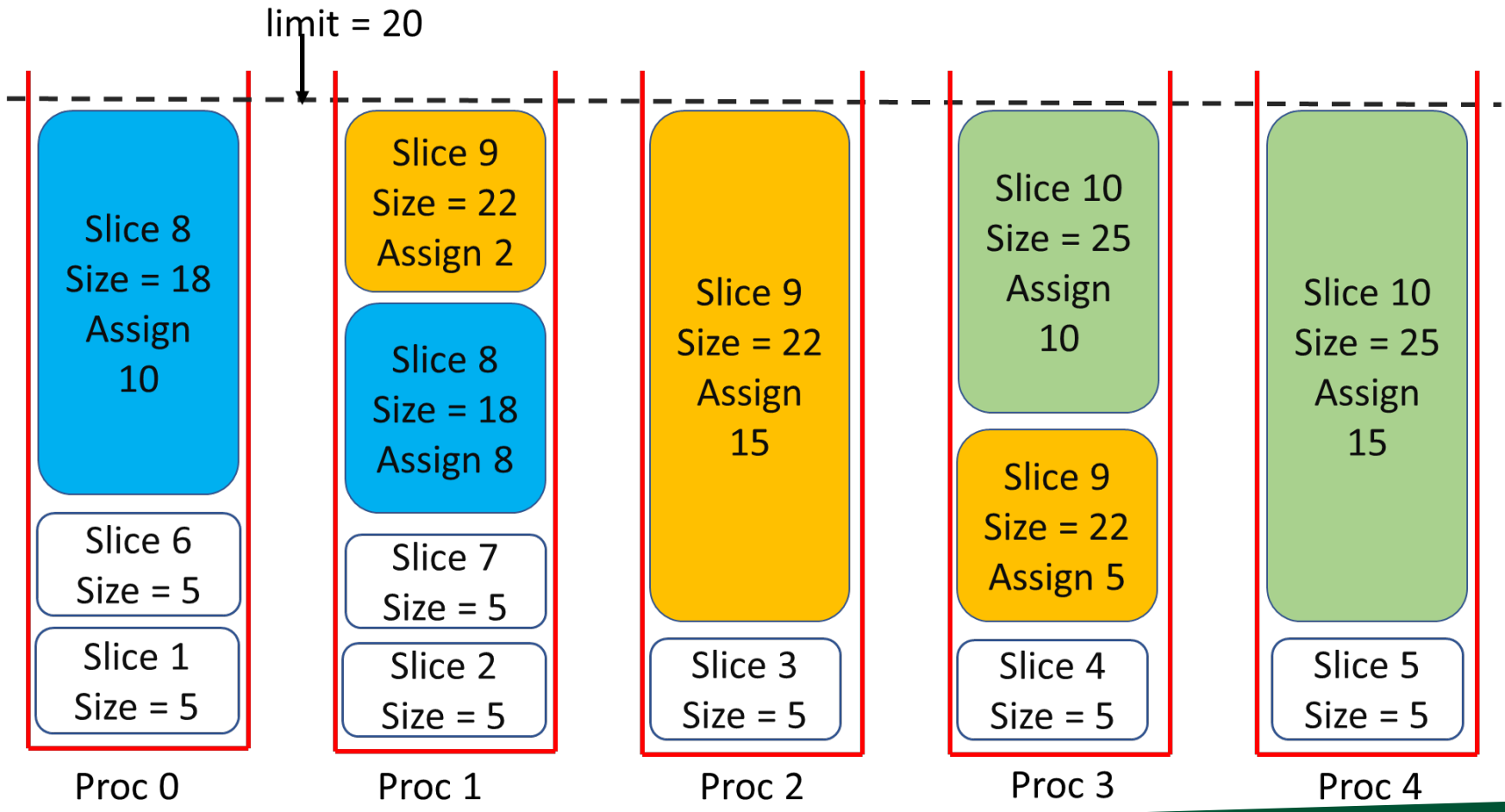
- SVD-Redundancy
  - Total number of times slices are “shared”
  - Measures computational load & comm. volume
  - Optimal value = L (length along the mode, no sharing)
- SVD-LImb:
  - Max number of slices shared by the processors
  - Optimal value = L / P

Factor Matrix Transfer

- Communication volume at each iteration



Lite distribution scheme is simple



How does our scheme fare?

- TTM-LImb <= E / P (optimal)
- SVD-Redundancy <= L + P (optimal = L)
- SVD-LImb <= L/P + 2 (optimal = L/P)

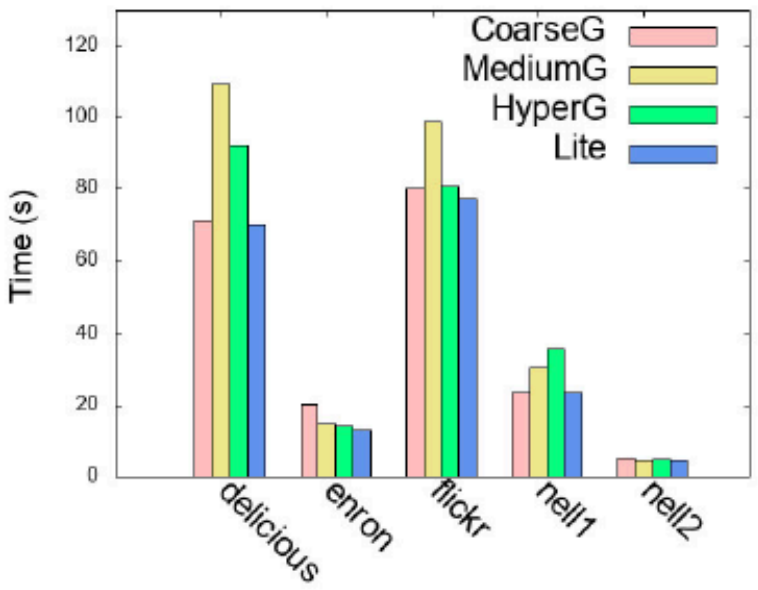
- Achieve near optimal
  - TTM computational load
  - SVD computational load, load balance
  - SVD communication volume
- Only issue is high factor matrix transfer volume
  - Computation dominates

Experimental evaluation

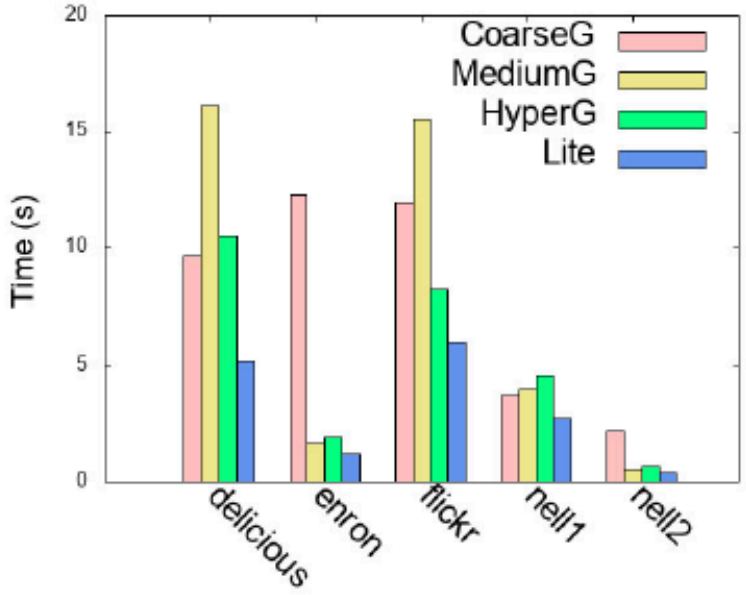
- R92 cluster – 2 to 32 nodes.
- 16 MPI ranks per node, each mapped to a core. (32 - 512 MPI ranks)
- Dataset : FROSTT repository (frostt.io)

Tensor	L_1	L_2	L_3	L_4	nnz
delicious	532K	17.2M	2.4M	1.4K	140M
enron	6K	5K	244K	1K	54M
flickr	319K	28M	1.6M	731	112M
nell1	2.9M	2.1M	25.4M	-	143M
nell2	12K	9K	28K	-	77M
amazon	4.8M	1.7M	1.8 M	-	1.7B
patents	46	239 K	239	-	3.5B
reddit	8.2M	176K	8.1M	-	4.6B

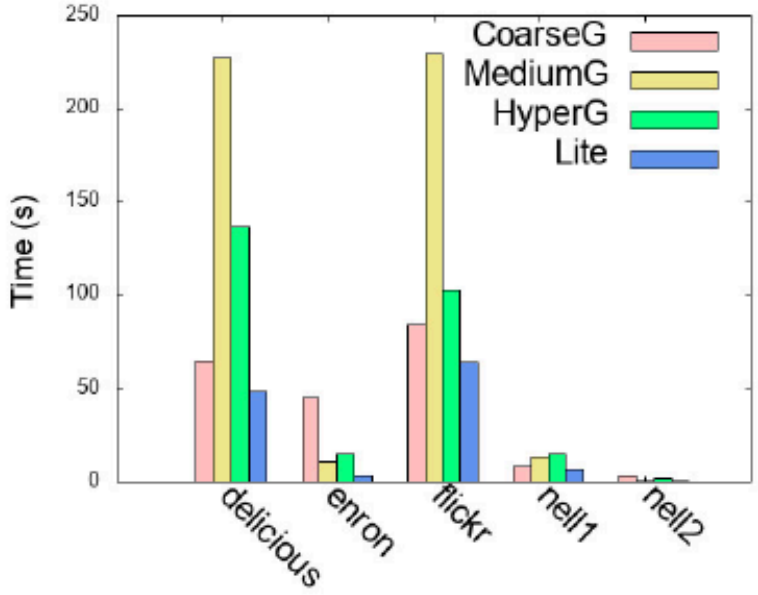
Execution time



(a) $K = 10$, ranks = 32



(b) $K = 10$, ranks = 512

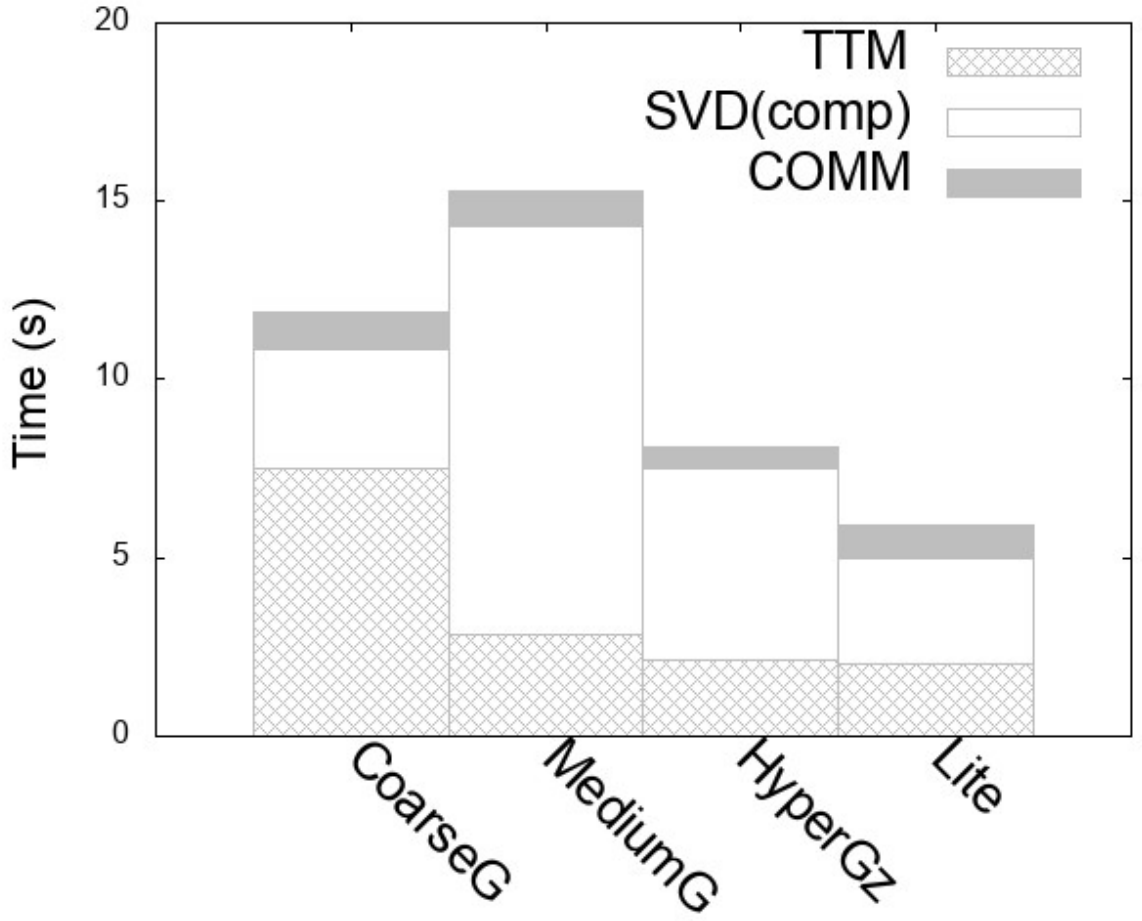


(c) $K = 20$, ranks = 512

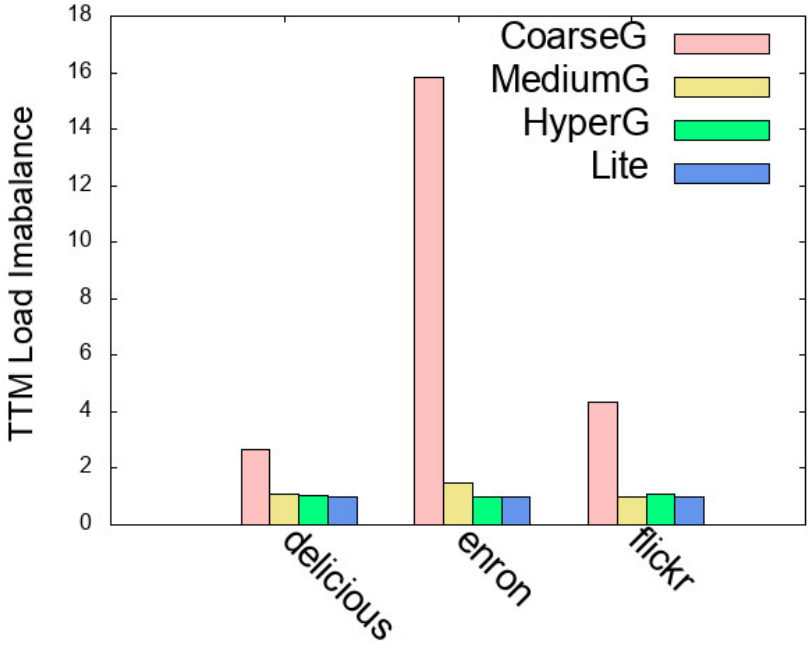
Speedup

Coarse – **12×** Medium – **4.5×** Hyper – **4. 1×** Best Prior – **3×**

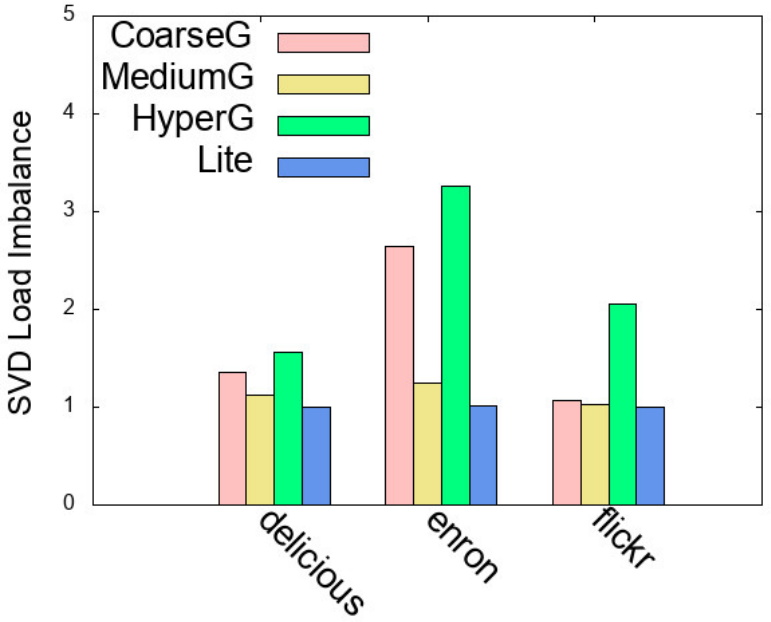
Breakdown – Flickr @ rank = 512 & K = 10



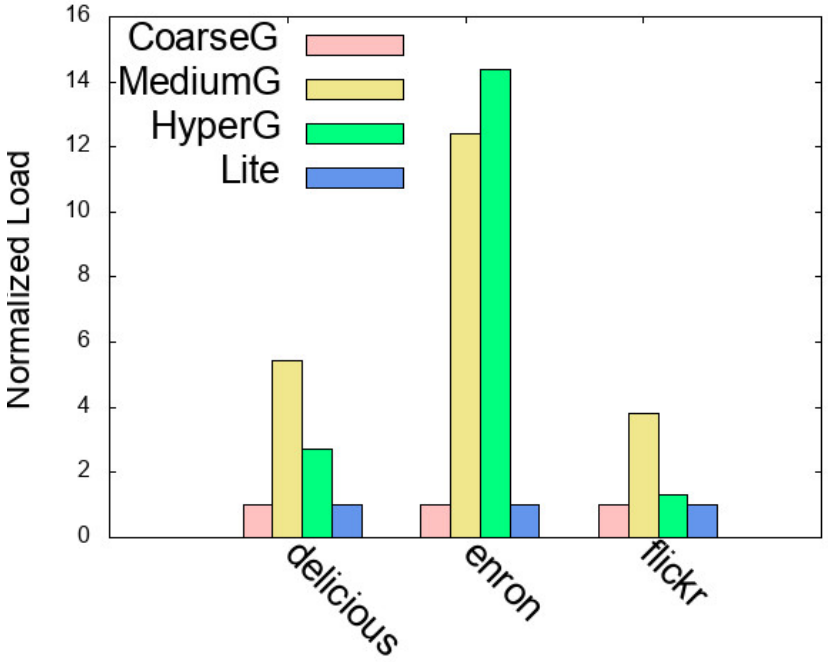
Comparison of the Performance Metrics



TTM Load Imbalance
(TTM-Limb)



SVD Load Imbalance
(SVD-Limb)



SVD Load
(SVD-Redundancy)

Strong Scaling Results (32 – 512 ranks)

Speedup	CG	MG	HG	Lite
delicious	7.4	6.8	8.8	13.4
enron	1.7	9.0	7.4	11.1
flickr	6.7	6.4	9.8	12.9
nell1	6.4	7.6	7.9	8.6
nell2	2.4	8.4	7.5	12.2
amazon	1.8	11.0	x	13.5
patents	2.7	14.5	x	15.5
reddit	1.8	14.2	x	14.6

Tensor Distribution Time

Time (s)	CG	MG	HG	Lite	HOOI
delicious	6.8	9.3	345	3.9	5.2
enron	0.1	0.08	125	0.1	1.1
flickr	10.9	14.0	203	5.5	6.0
nell1	10.5	13.9	356	6.2	2.7
nell2	0.07	0.05	91	0.07	0.3
amazon	2.9	5.5	x	2.5	8.7
patents	3.2	0.9	x	2.0	14.2
reddit	7.8	11.6	x	5.7	21.6

Challenges and possible solutions

- Data locality (both shared and distributed) is important in performance of TD algorithms
  - However, due to the diversity of the kernels, there is no single solution
  - High dimensionality makes everything more difficult
- Ideally
  - Finding the **right** programming model/abstraction for capturing data distribution (shared and distributed) and its impact on performance
  - Domain-specific language/compiler to overcome tensor-specific bottlenecks
  - Optimized libraries

Future work

- Future work
  - Efficient data structures for sparse tensors
  - Modeling the sparsity
  - Near-memory processing architectures for tensor computation
  - Energy efficiency (on mobile devices)

Reference

- [1] **Jee W. Choi**, Xing Liu, Shaden Smith, Tyler Simon, *Blocking Optimization Techniques for Sparse Tensor Computation*. 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'18).
- [2] Venkatesan T. Chakaravarthy, **Jee W. Choi**, Douglas J. Joseph, Prakash Murali, Yogish Sabharwal, S. Shivmaran, Dheeraj Sreedhar, *On Optimizing Distributed Tucker Decomposition for Sparse Tensors*. The 32nd ACM International Conference on Supercomputing (ICS'18).
- [3] **Jee W. Choi**, Xing Liu, Venkatesan T. Chakaravarthy, *High-performance Dense Tucker Decomposition on GPU Clusters*. The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18).
- [4] Venkatesan T. Chakaravarthy, **Jee W. Choi**, Xing Liu, Douglas J. Joseph, Prakash Murali, Yogish Sabharwal, Dheeraj Sreedhar, *On Optimizing Distributed Tucker Decomposition for Dense Tensors*. 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS'17).