Scaling with the Parameter Server
Variations on a Theme

Alexander Smola
Google Research & CMU
alex.smola.org

Thanks

Amr
Ahmed

Nino
Shervashidze

Joey
Gonzalez

Shravan
Narayanamurthy

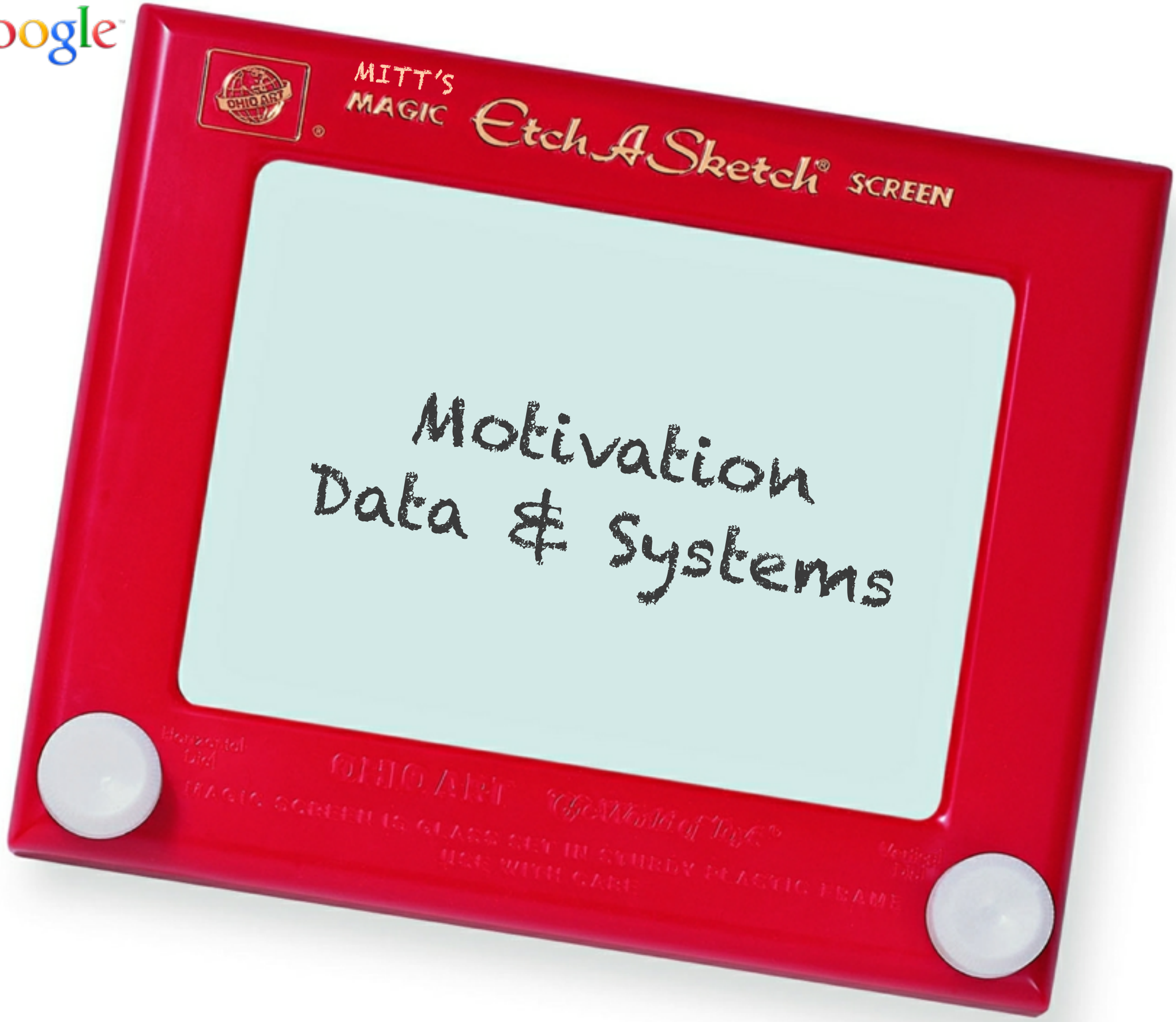
Sergiy
Matyusevich

Markus
Weimer

Practical Distributed Inference

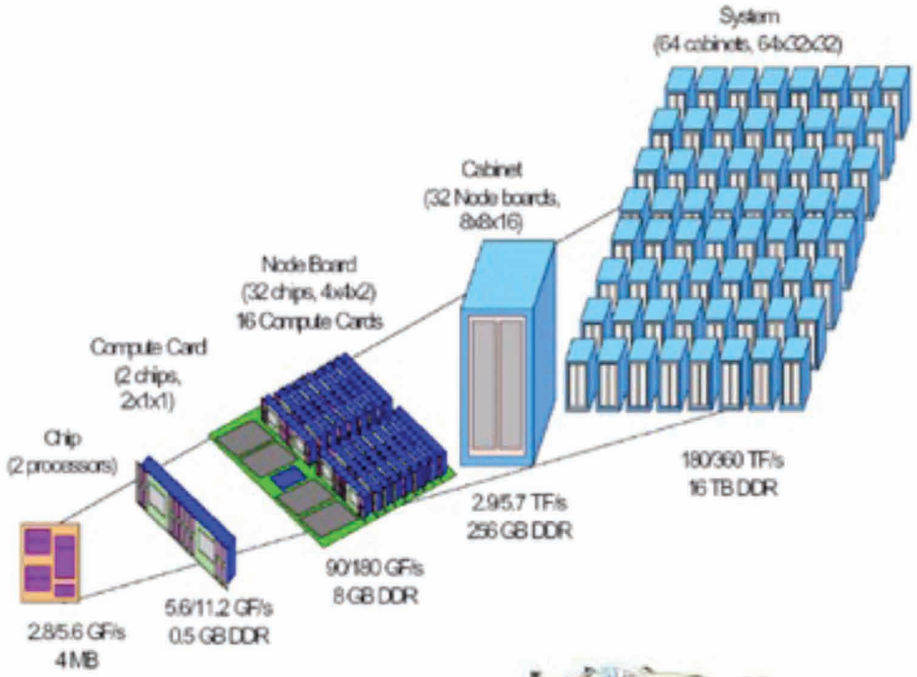
- Multicore
  - asynchronous optimization with shared state
- Multiple machines
  - exact synchronization (Yahoo LDA)
  - approximate synchronization
  - dual decomposition

Google



Commodity Hardware

- High Performance Computing
  Very reliable, custom built, expensive



- Consumer hardware
  Cheap, efficient, easy to replicate,
  Not very reliable, deal with it!

Google

The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//people/jeff/stanford-295-talk.pdf

Slide courtesy of Jeff Dean

Scaling problems

- Data (lower bounds)
  - >10 Billion documents (webpages, e-mails, advertisements, tweets)
  - >100 Million users on Google, Facebook, Twitter, Yahoo, Hotmail
  - >1 Million days of video on YouTube
  - >10 Billion images on Facebook
- Processing capability for single machine 1TB/hour
  But we have much more data
- Parameter space for models is big for a single machine (but not too much)
  Personalize content for many millions of users
- Need to process data on **many cores** and **many machines simultaneously**

Some Problems

- Good old-fashioned supervised learning
  (classification, regression, tagging, entity extraction, ...)
- Graph factorization
  (latent variable estimation, social recommendation, discovery)
- Structure inference
  (clustering, topics, hierarchies, DAGs, whatever else your NP Bayes friends have)
- Example use case - combine information from generic webpages, databases, human generated data, semistructured tables into knowledge about entities.

Some Problems

- Good old-fashioned supervised learning
  (classification, regression, tagging, entity extraction, ...)
- Graph factorization
  (latent variable estimation, social recommendation, discovery)
- Structure inference
  (clustering, topics, hierarchies, DAGs, whatever else your NP Bayes friends have)
- Example use case – combine information from generic webpages, databases, human generated data, semistructured tables into knowledge about entities.

How do we solve it at scale?

Some Problems

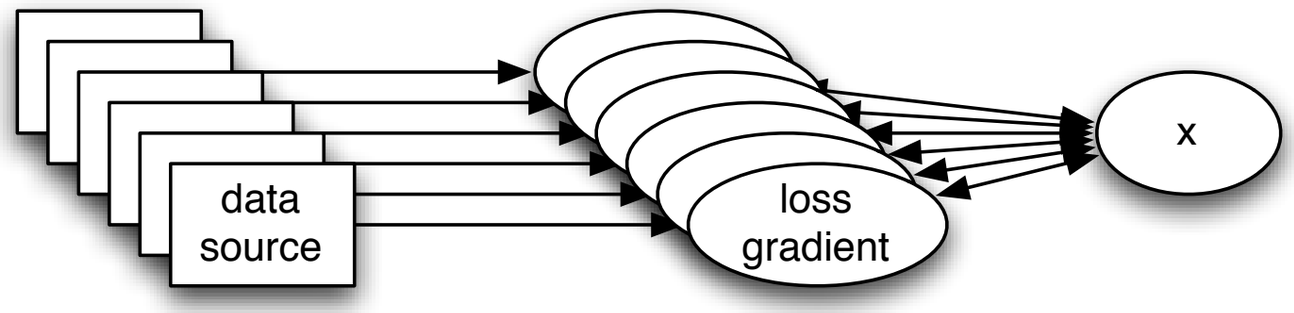
- Good old-fashioned supervised learning
  (classification, regression, tagging, entity extraction, ...)
- Graph factorization
  (latent variable estimation, social recommendation, discov...)
- Structure inference
  (clustering, topics, hierarchies, DAGs, whatever else your NP Bayes friends have)
- Example use case – combine information from generic webpages, databases, human generated data, semistructured tables into knowledge about entities.

How do we solve it at scale?

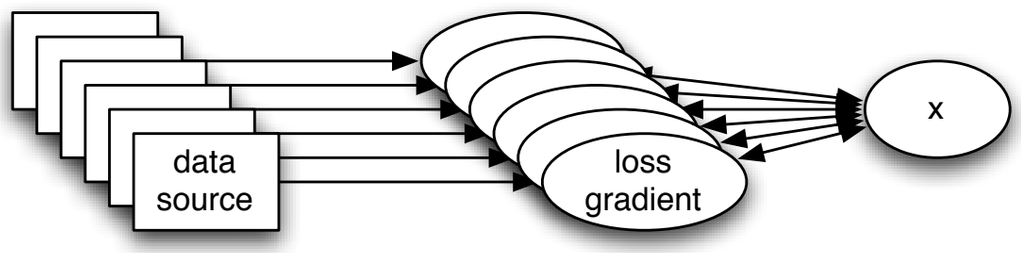

MITT'S
MAGIC
Etch A Sketch
SCREEN
Multicore
parallelism

Multicore Parallelism

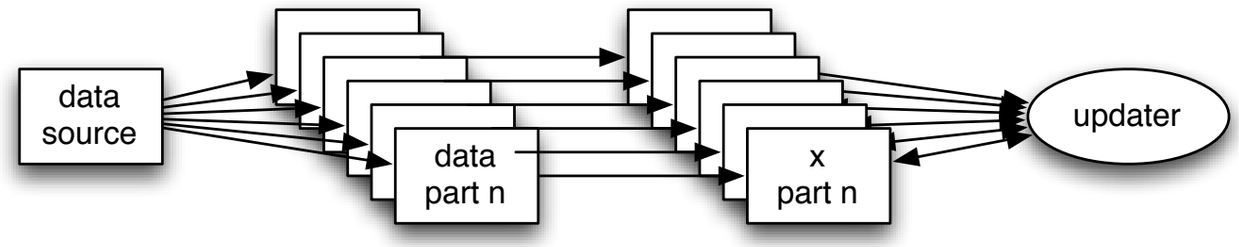
- Many processor cores
  - Decompose into separate tasks
  - Good Java/C++ tool support
- Shared memory
  - Exact estimates – requires locking of neighbors (see e.g. Graphlab)
    Good if problem can be decomposed cleanly (e.g. Gibbs sampling in large model)
  - Exact updates but delayed incorporation – requires locking of state
    Good if delayed update is of little consequence (e.g. Yahoo LDA, Yahoo online)
  - Hogwild updates – no locking whatsoever – requires atomic state
    Good if collision probability is low



Stochastic Gradient Descent



data parallel



parameter parallel

- Delayed updates
  (round robin for data parallelism, aggregation tree for parameter parallelism)
- Online template

$$\underset{w}{\text{minimize}} \sum_i f_i(w)$$

Input: scalar $\sigma > 0$ and delay τ
for $t = \tau + 1$ **to** $T + \tau$ **do**
  Obtain f_t and incur loss $f_t(w_t)$
  Compute $g_t := \nabla f_t(w_t)$ and set $\eta_t = \frac{1}{\sigma(t-\tau)}$
  Update $w_{t+1} = w_t - \eta_t g_{t-\tau}$
end for

Guarantees

- **Worst case guarantee** (Zinkevich, Langford, Smola, 2010)
  SGD with delay τ on τ processors is no worse than sequential SGD

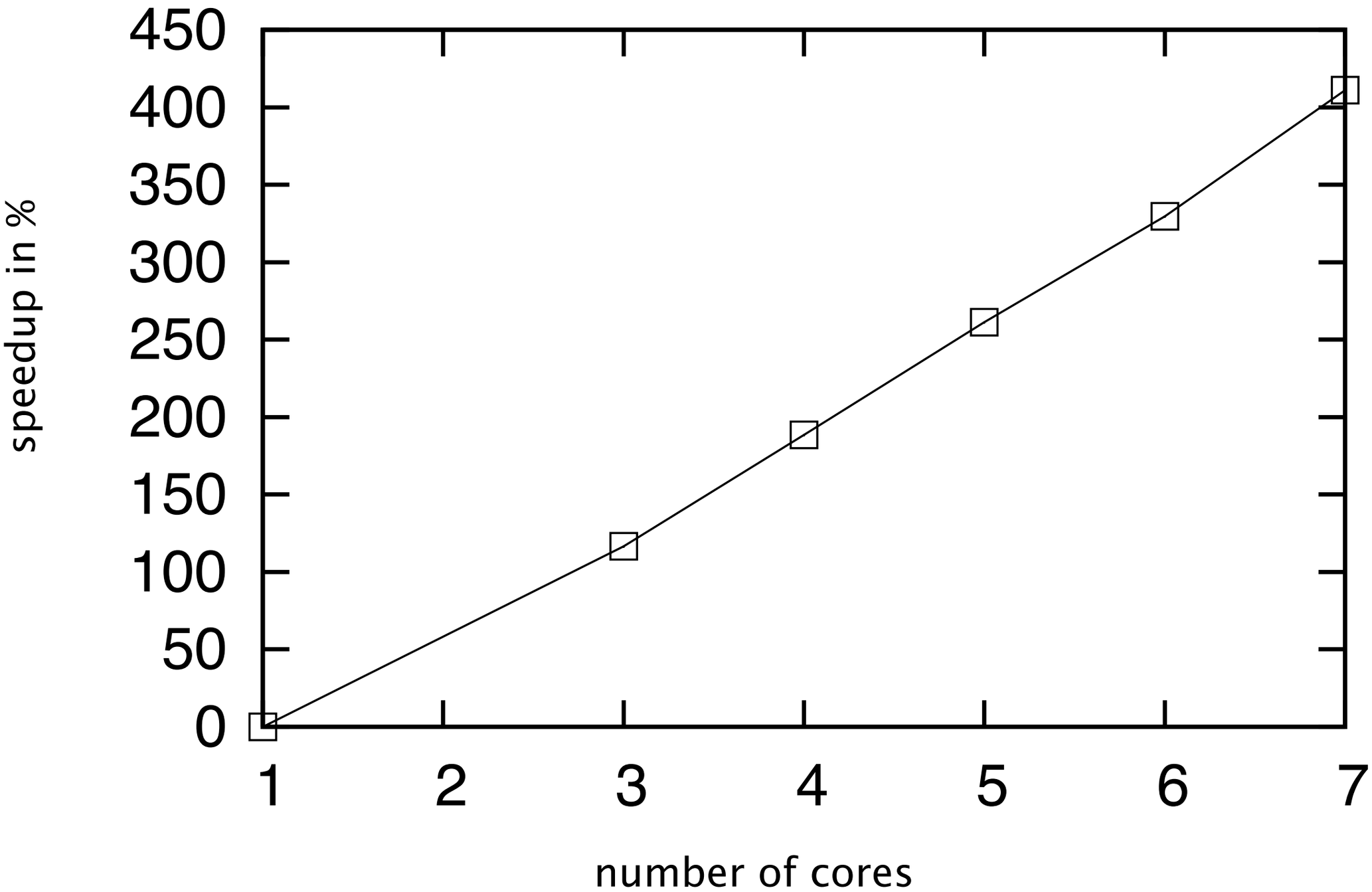
$$\mathbf{E}[f_i(w)] \leq 4RL\sqrt{\tau T}$$

- **Lower bound is tight**
  Proof: send same instance τ times
- **Better bounds with iid data**
  - Penalty is covariance in features
  - Vanishing penalty for smooth f(w)

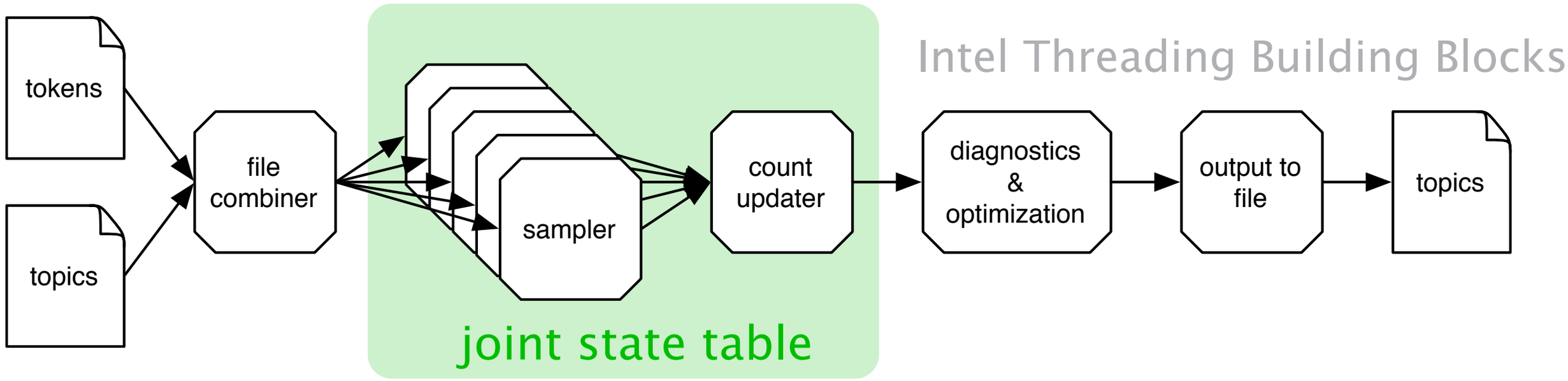
$$\mathbf{E}[R[X]] \leq \left[28.3R^2H + \frac{2}{3}RL + \frac{4}{3}R^2H\log T\right]\tau^2 + \frac{8}{3}RL\sqrt{T}.$$

- **Works even (better) if we don't lock between updates**
  (Recht, Re, Wright, 2011) Hogwild

Speedup on TREC



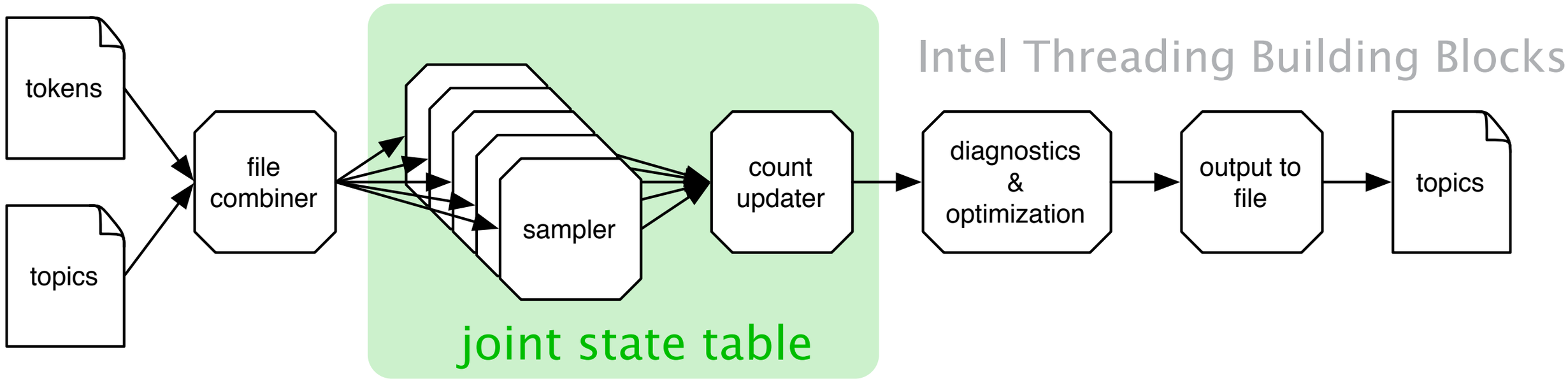
LDA Multicore Inference



- Decouple multithreaded sampling and updating (almost) avoids stalling for locks in the sampler
- Joint state table
  - much less memory required
  - samplers synchronized (10 docs vs. millions delay)
- Hyperparameter update via stochastic gradient descent
- No need to keep documents in memory (streaming)

Smola and Narayanamurthy, 2010

LDA Multicore Inference



- Sequential collapsed Gibbs sampler, separate state table
  Mallet (Mimno et al. 2008) – slow mixing, high memory load, many iterations
- Sequential collapsed Gibbs sampler (parallel)
  Yahoo LDA (Smola and Narayanamurthy, 2010) – fast mixing, many iterations
- Sequential stochastic gradient descent (variational, single logical thread)
  VW LDA (Hoffman et al, 2011) – fast convergence, few iterations, dense
- Sequential stochastic sampling gradient descent (only partly variational)
  Hoffman, Mimno, Blei, 2012 – fast convergence, quite sparse, single logical thread

Smola and Narayanamurthy, 2010

General strategy

- Shared state space
- Delayed updates from cores
- Proof technique is usually to show that the problem hasn't changed too much during the delay (in terms of interactions).

- More work
  - Macready, Siapas and Kauffman, 1995
    Criticality and Parallelism in Combinatorial Optimization
  - Low, Gonzalez, Kyrola, Bickson, Guestrin and Hellerstein, 2010
    Shotgun for l1

This was easy ...

what if we need many machines?

This was easy ...

what if we need many machines?

This was easy ...

what if we need many machines?


MITT'S
MAGIC
Etch A Sketch
SCREEN
Parameter Server
30,000 ft view

Why (not) MapReduce?

- Map(key, value)
  process instances on a subset of the data / emit aggregate statistics
- Reduce(key, value)
  aggregate for all the dataset – update parameters
- This is a parameter exchange mechanism (simply repeat MapReduce)
  good if you can make your algorithm fit (e.g. distributed convex online solvers)
- Can be slow to propagate updates between machines & slow convergence
  (e.g. a really bad idea in clustering – each machine proposes different clustering)
  Hadoop MapReduce loses the state between mapper iterations

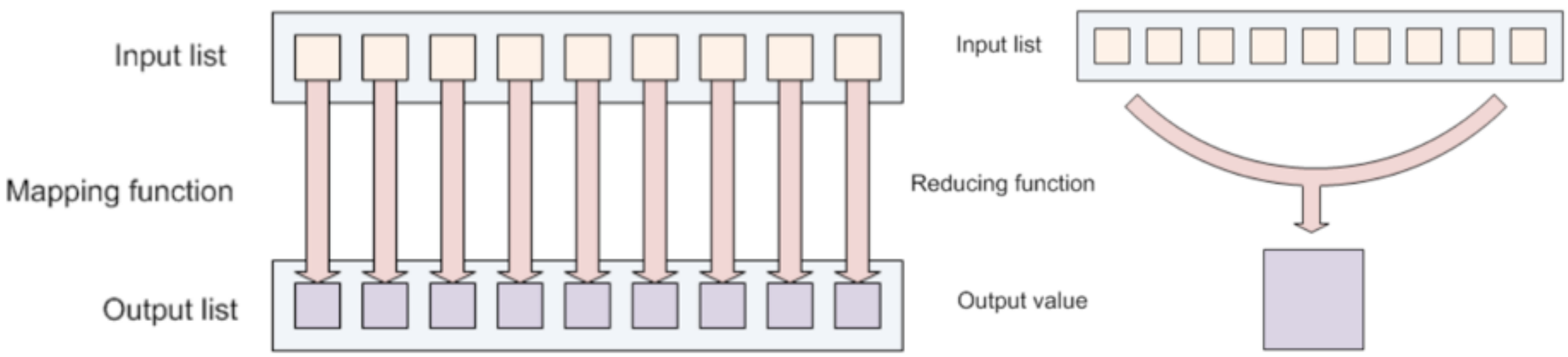
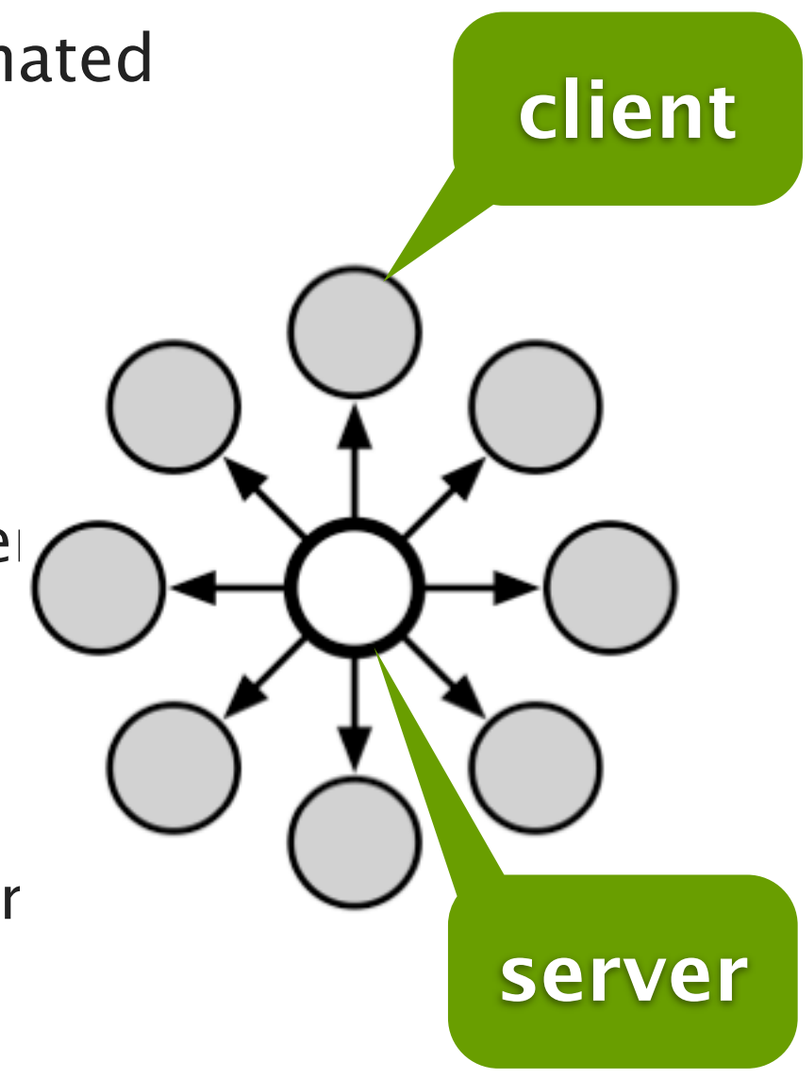


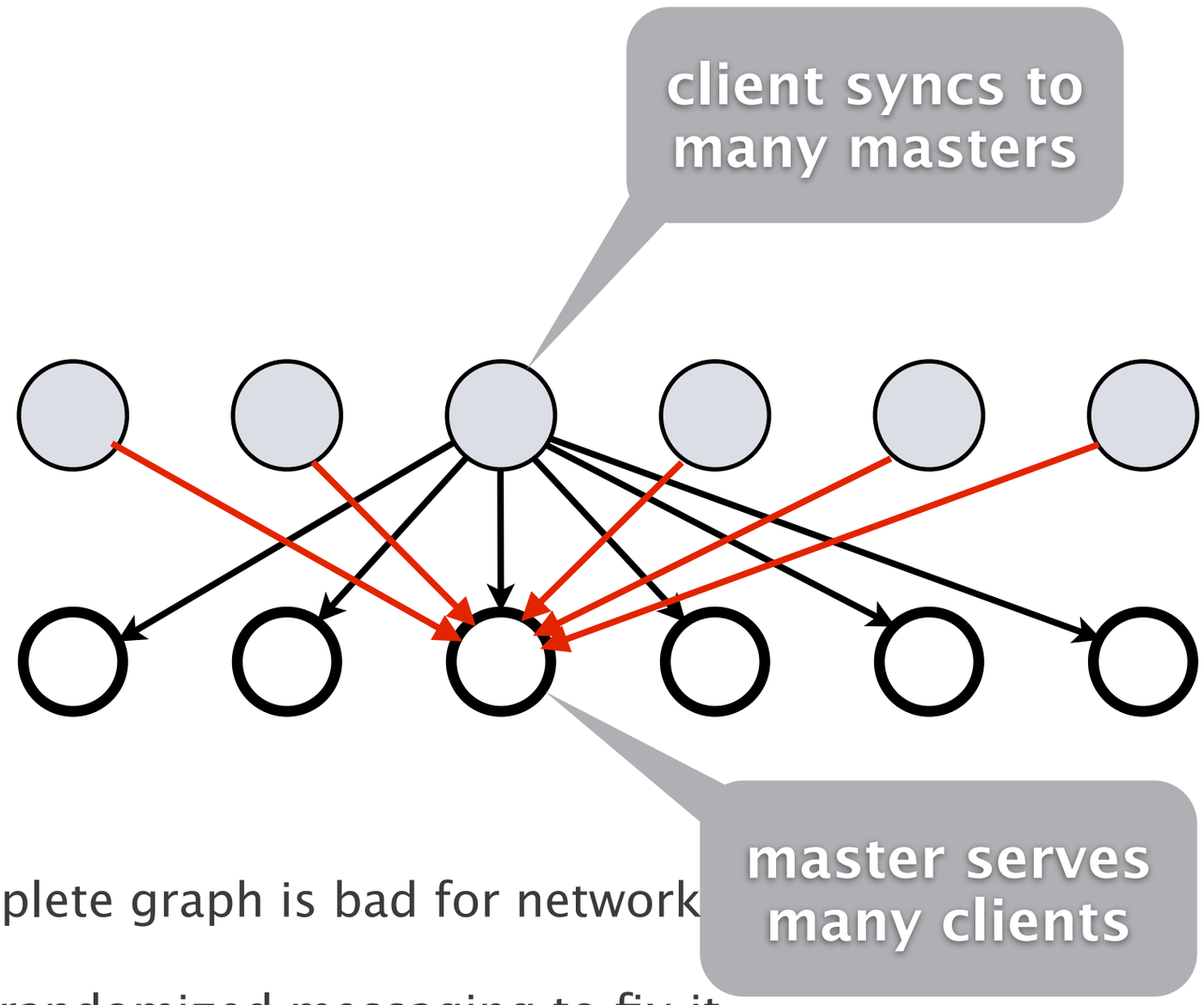
diagram from Ramakrishnan, Sakrejda, Canon, DoE 2011

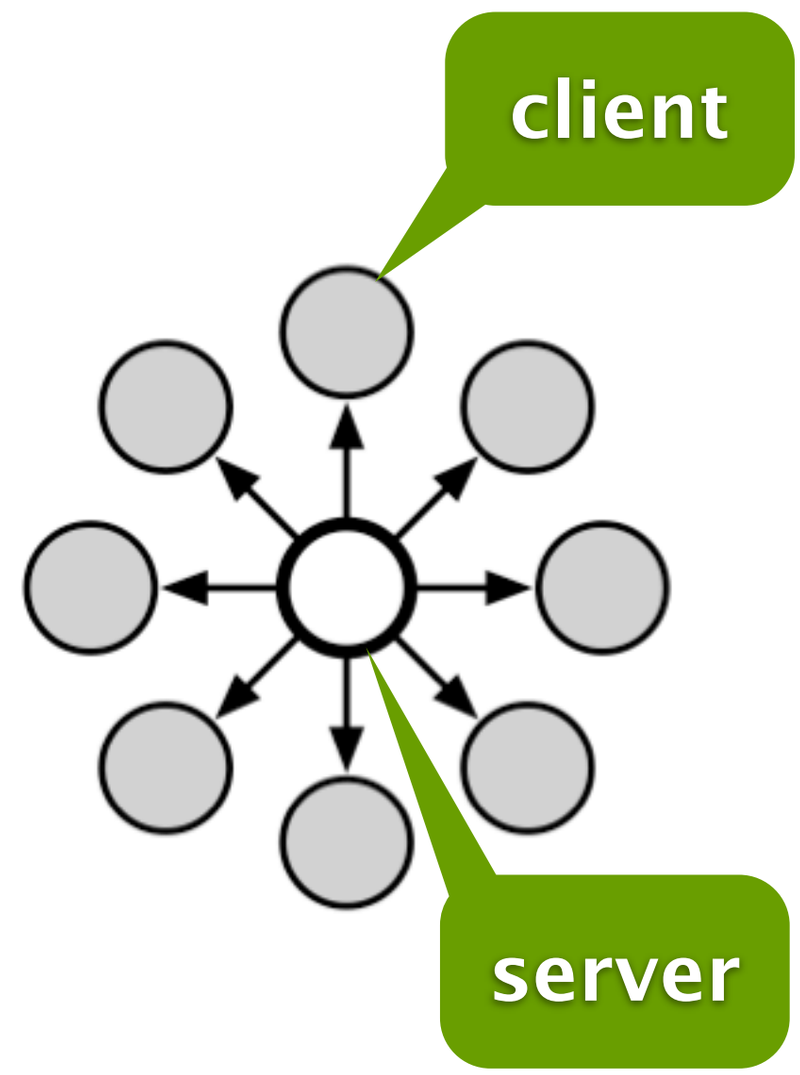
General parallel algorithm template

- Clients have local copy of parameters to be estimated
- P2P is infeasible since $O(n^2)$ connections
  (see Asuncion et al. for amazing tour de force)
- Synchronize* with parameter server
  - **Reconciliation protocol**
    average parameters, lock variables, turnstile counter
  - **Synchronization schedule**
    asynchronous, synchronous, episodic
  - **Load distribution algorithm**
    single server, uniform distribution, fault tolerance, r



General parallel algorithm template





complete graph is bad for network

use randomized messaging to fix it

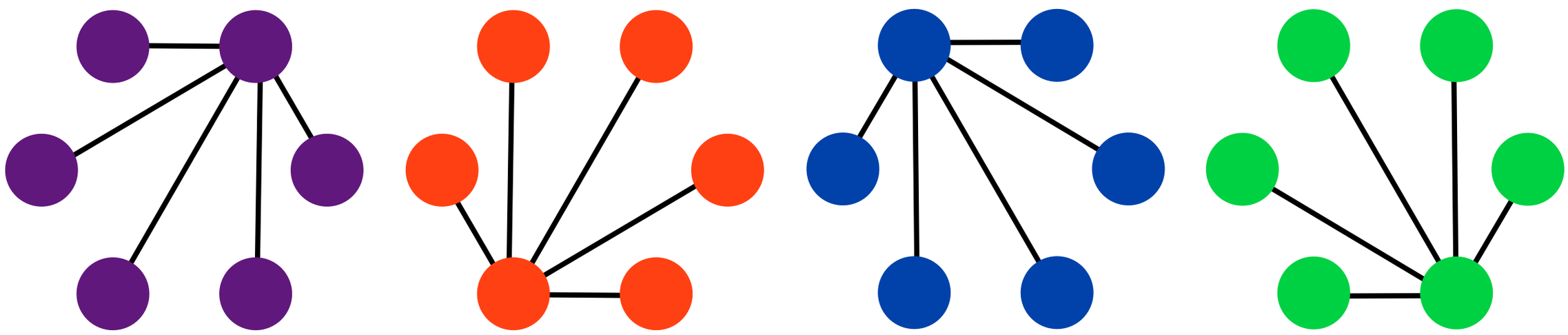
Desiderata

- Variable and load distribution
  - Large number of objects (a priori unknown)
  - Large pool of machines (often faulty)
  - Assign objects to machines such that
    - Object goes to the same machine (if possible)
    - Machines can be added/fail dynamically
  - Consistent hashing (elements, sets, proportional)
- Symmetric, dynamically scalable, fault tolerant
  - for large scale inferences
  - for real time data sketches

Random Caching Trees

- Cache / synchronize an object
- Uneven load distribution
- Must not generate hotspot

- For given key, pick random order of machines
- Map order onto tree / star via BFS ordering



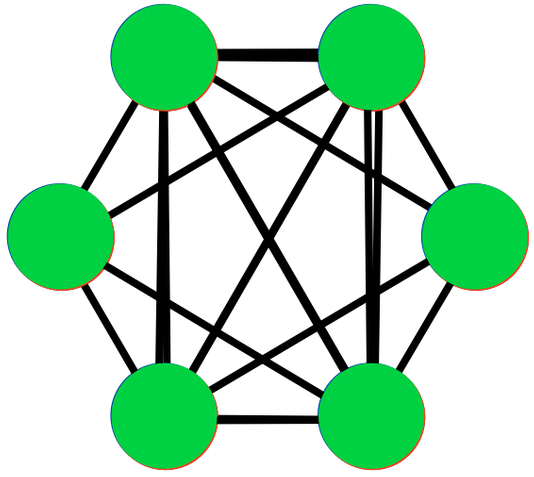
Karger et al. 1999, Ahmed et al. 2011

Random Caching Trees

- Cache / synchronize an object
- Uneven load distribution
- Must not generate hotspot

(Karger et al. 1999 – 'Akamai' pape

- For given key, pick random order of machines
- Map order onto tree / star via BFS ordering



Karger et al. 1999, Ahmed et al. 2011

Argmin Hash

- Consistent hashing

$$m(\text{key}) = \operatorname*{argmin}_{m \in \mathcal{M}} h(\text{key}, m)$$

  - Uniform distribution over machine pool M
  - Fully determined by hash function h. No need to ask master
  - If we add/remove machine m' all but O(1/m) keys remain

$$\Pr\{m(\text{key}) = m'\} = \frac{1}{m}$$

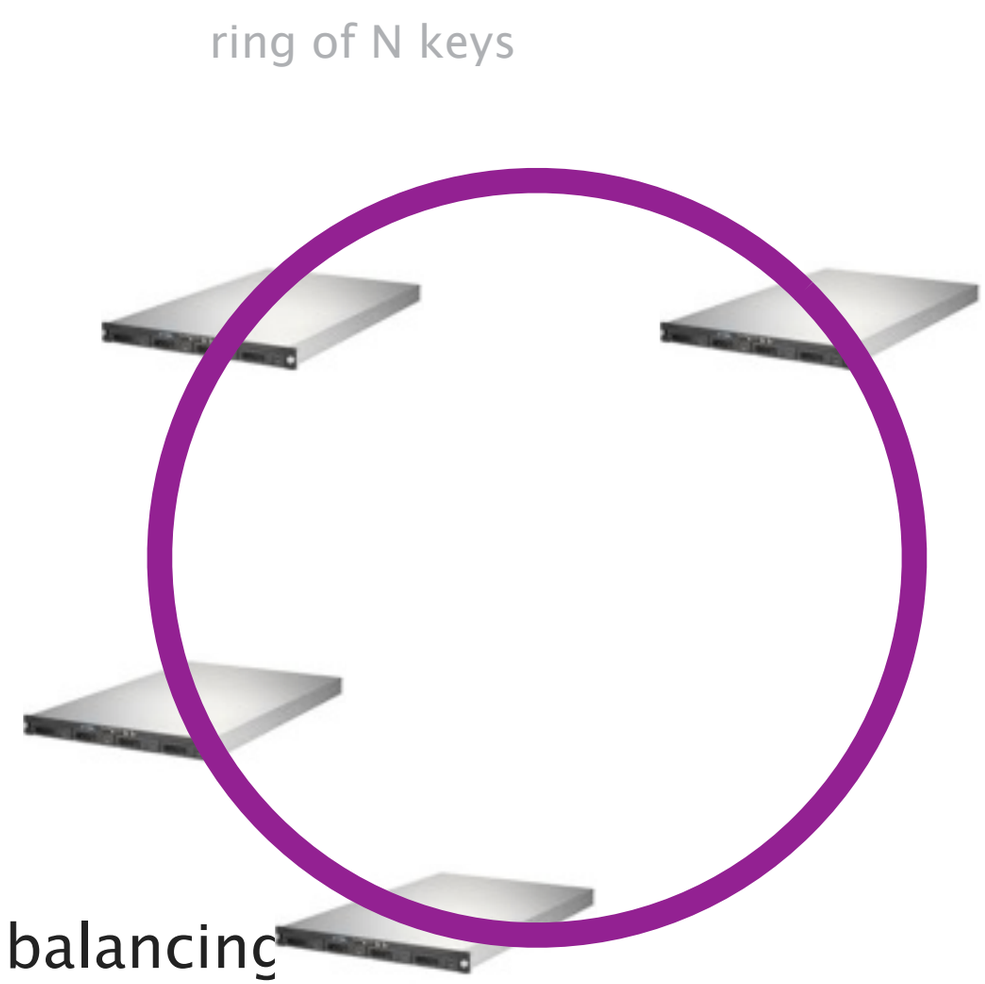
- Consistent hashing with k replications

$$m(\text{key}, k) = k \operatorname*{smallest}_{m \in \mathcal{M}} h(\text{key}, m)$$

  - If we add/remove a machine only O(k/m) need reassigning (also self repair)
- Cost to assign is O(m). This can be expensive for 1000 servers

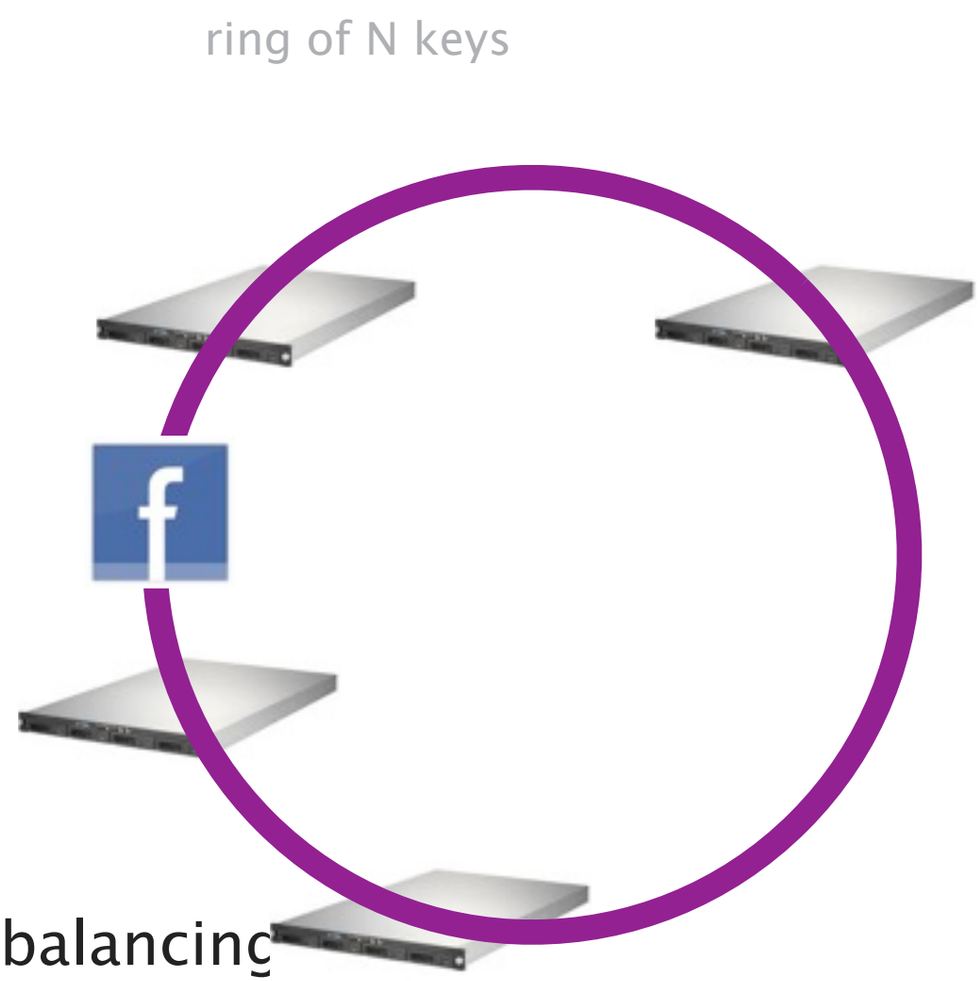
Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor (however, big problem for neighbor)
- Uneven load distribution (load depends on segment size)
- Insert machine more than once to fix this (do not use messy Cassandra-style manual balancing
- For k term replication, simply pick the k leftmost machines (skip duplicates)



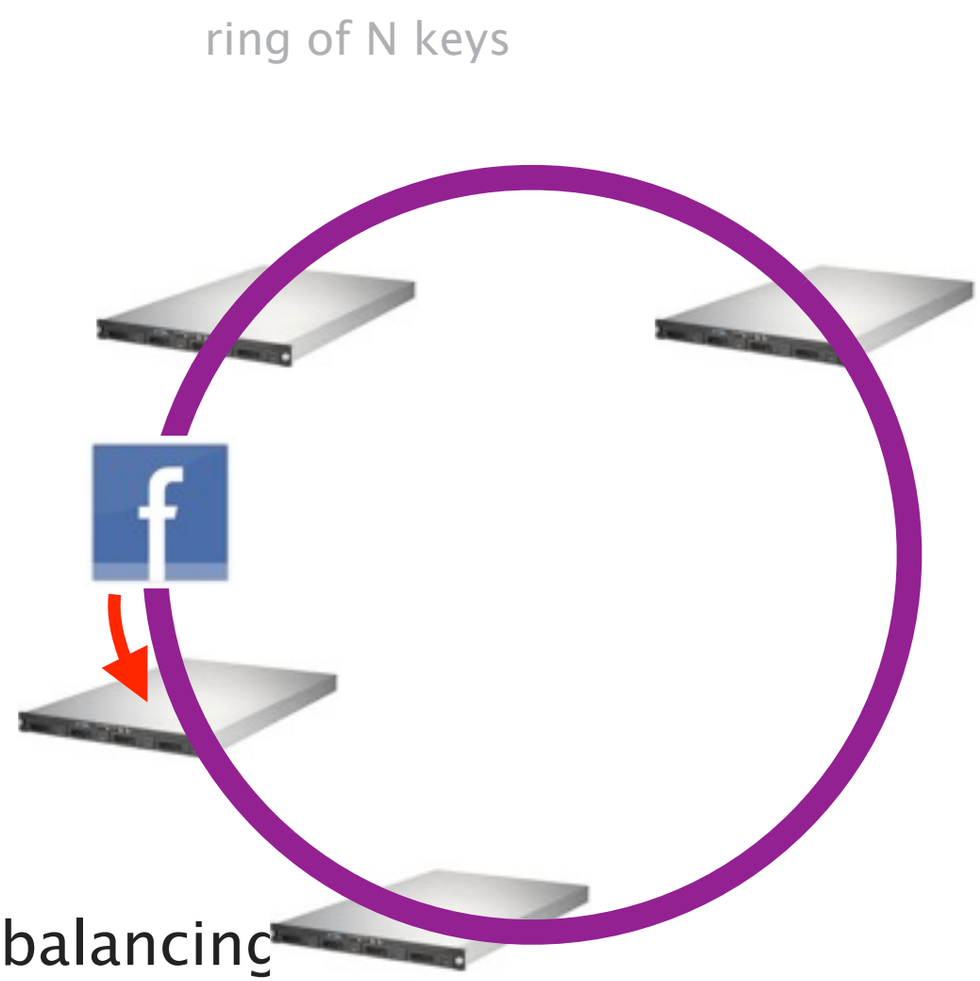
Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor (however, big problem for neighbor)
- Uneven load distribution (load depends on segment size)
- Insert machine more than once to fix this (do not use messy Cassandra-style manual balancing
- For k term replication, simply pick the k leftmost machines (skip duplicates)



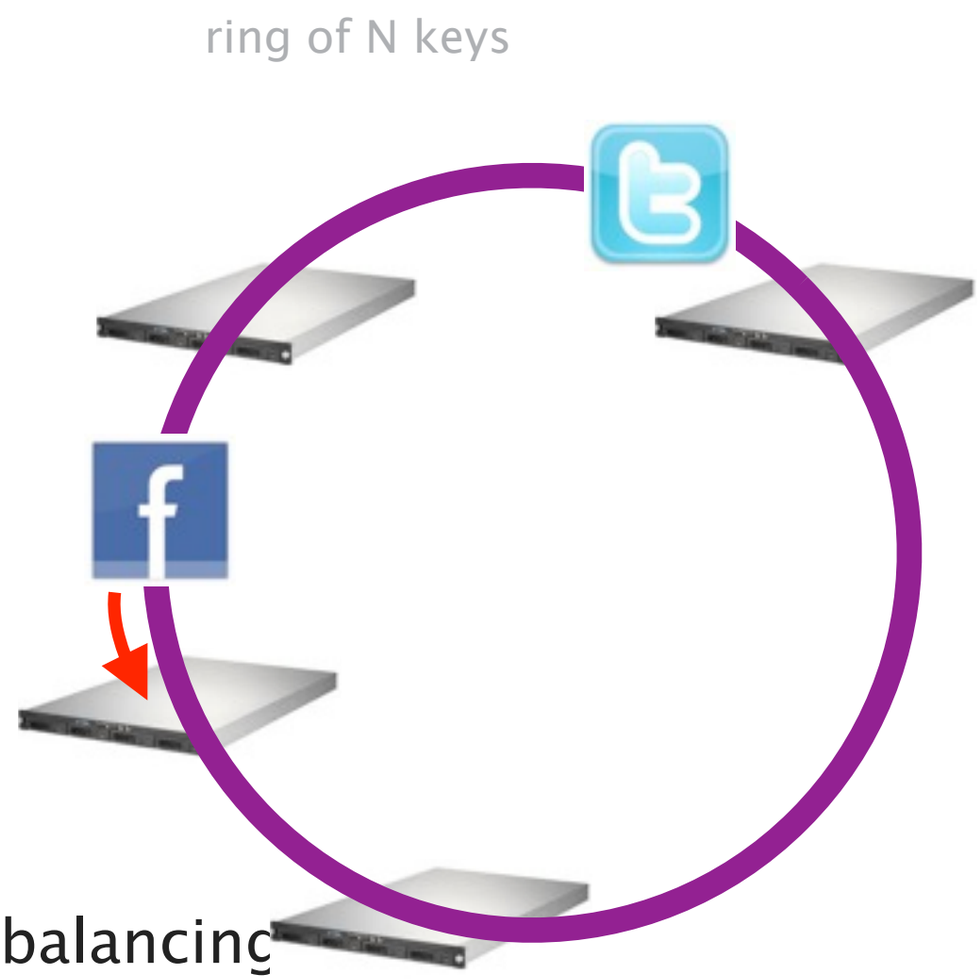
Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor
  (however, big problem for neighbor)
- Uneven load distribution
  (load depends on segment size)
- Insert machine more than once to fix this
  (do not use messy Cassandra-style manual balancing
- For k term replication, simply pick the k leftmost machines (skip duplicates)



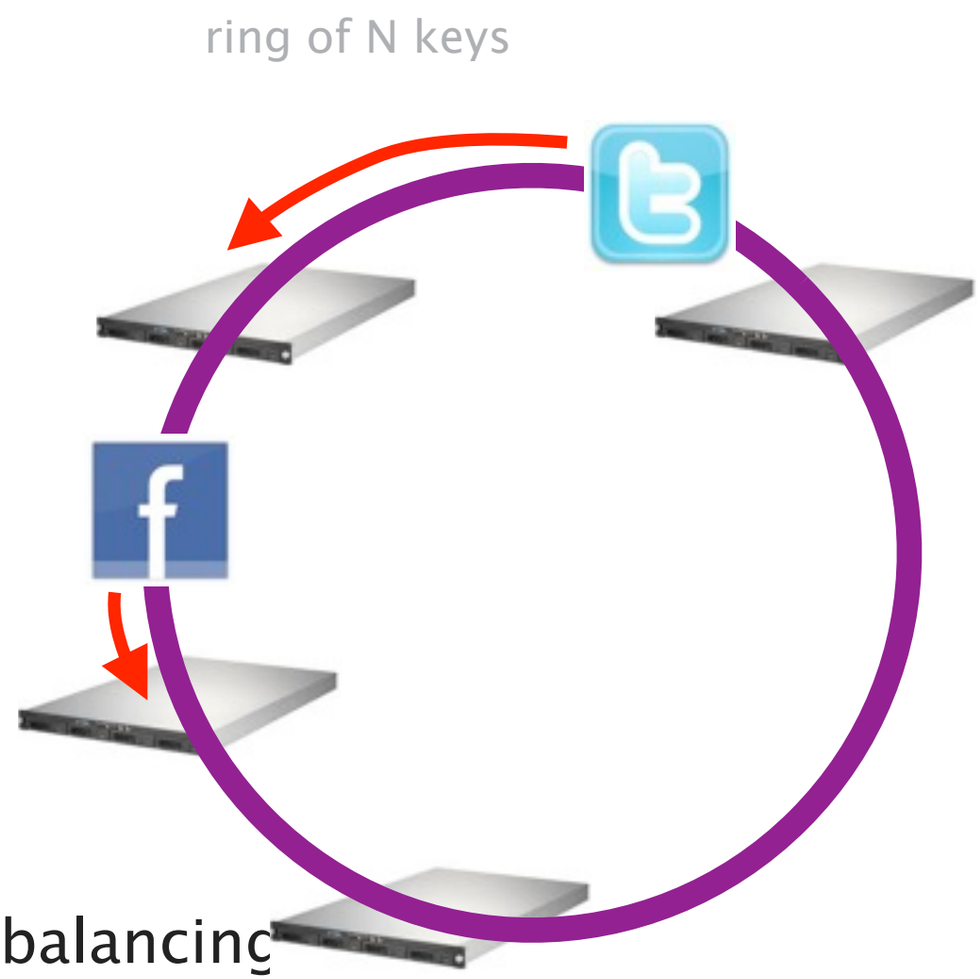
Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor
  (however, big problem for neighbor)
- Uneven load distribution
  (load depends on segment size)
- Insert machine more than once to fix this
  (do not use messy Cassandra-style manual balancing
- For k term replication, simply pick the k leftmost machines (skip duplicates)



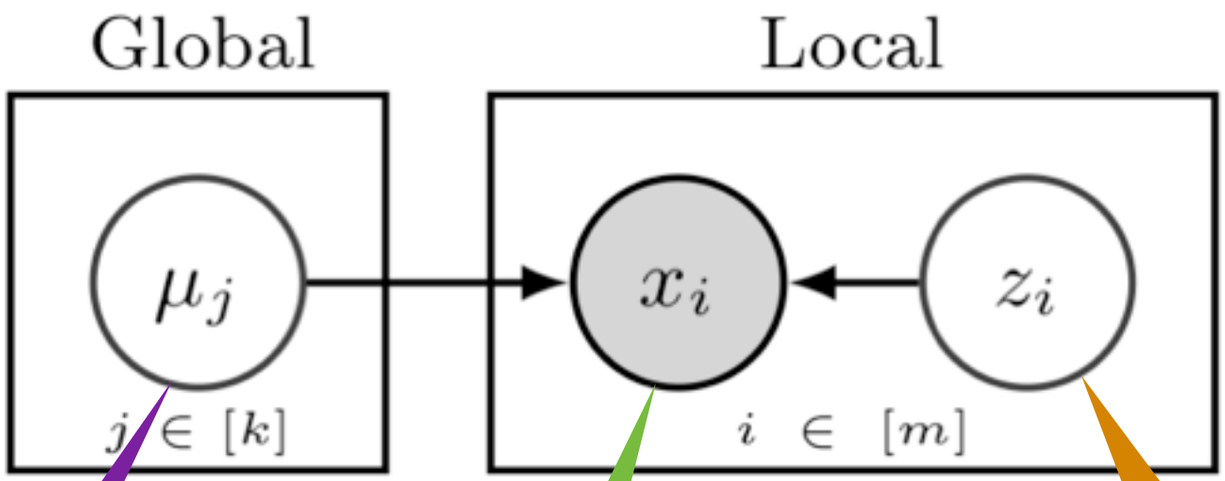
Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor
  (however, big problem for neighbor)
- Uneven load distribution
  (load depends on segment size)
- Insert machine more than once to fix this
  (do not use messy Cassandra-style manual balancing
- For k term replication, simply pick the k leftmost machines (skip duplicates)




MITT'S
MAGIC
Etch A Sketch
SCREEN
Exact
Synchronization

Motivation – Latent Variable Models

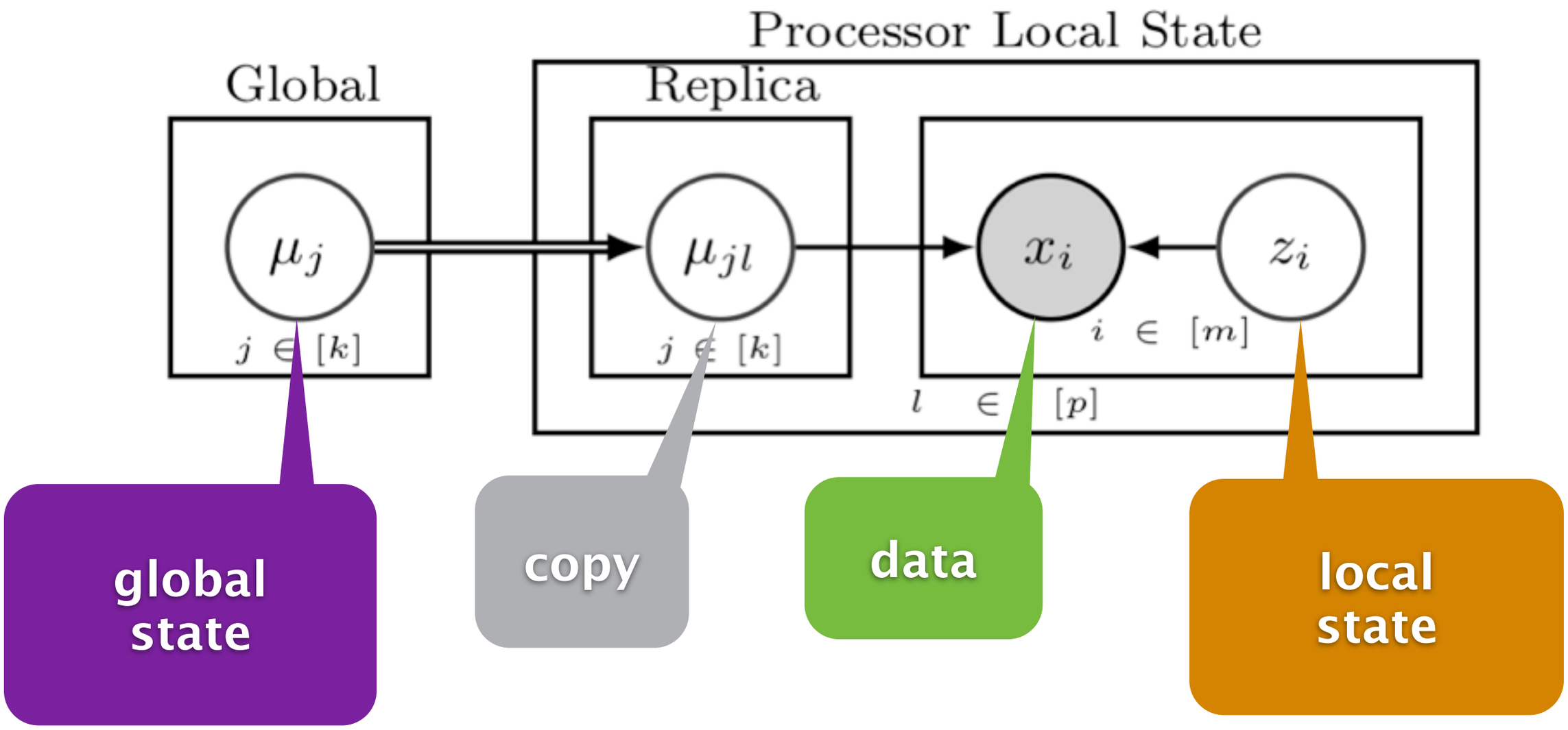


global state

data

local state

Distribution



Preserving the polytope

Abelian group

- Delayed count updates

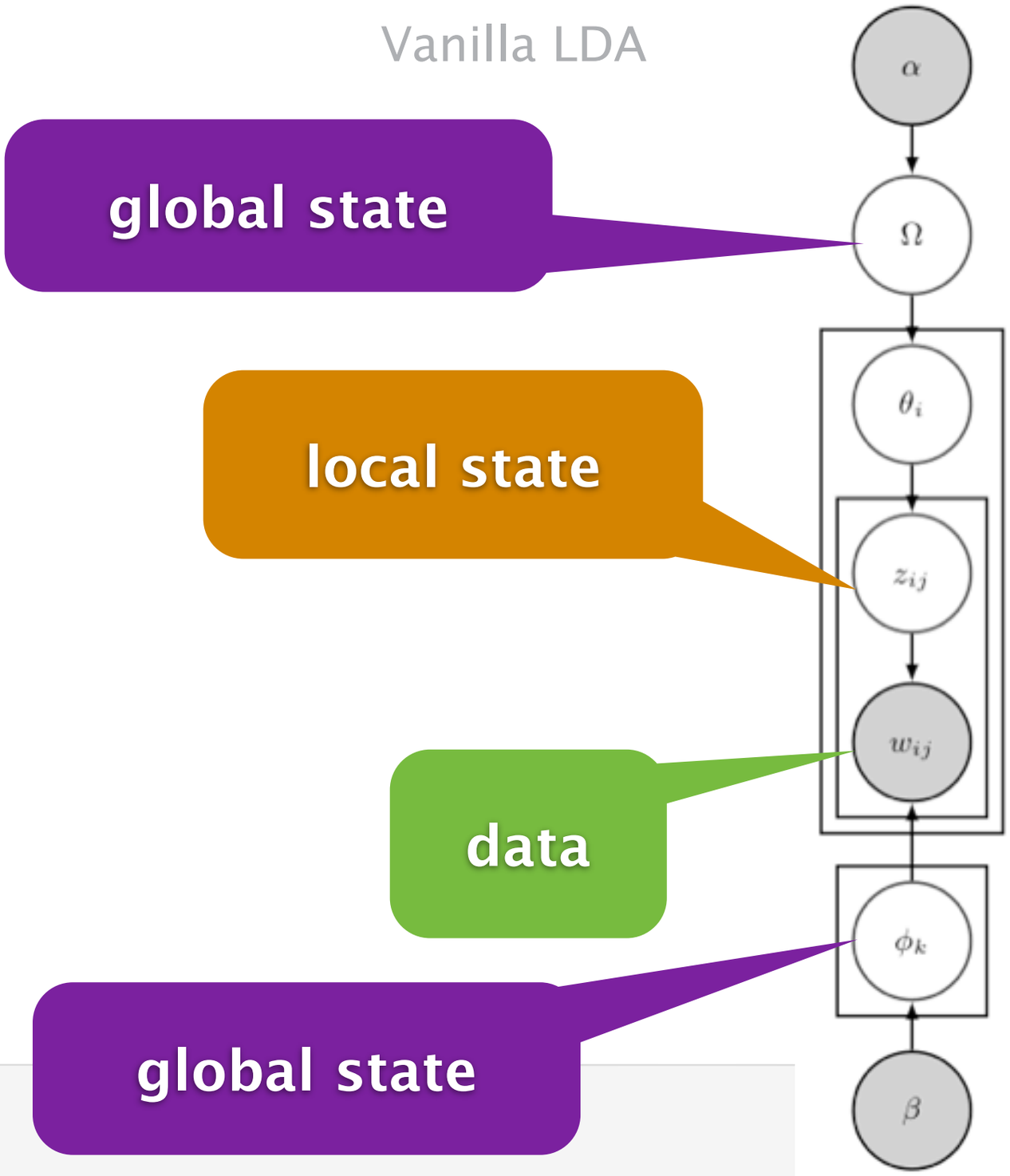
$$p(X|\mu_0, m_0) = \int d\theta p(\theta|\mu_0, m_0) \prod_{i=1}^{m} p(x_i|\theta) = p\left(\sum_{i=1}^{m} \phi(x_i)|\mu_0, m_0\right)$$

  - Collapsed representation for exponential families (bad things can happen otherwise – negative counts, indefinite covariances)
  - Need to keep track of aggregate state of random variables
- Exchangeable random process
  - See also Church by Mansinghka, Tenenbaum, Roy etc.
  - Need to maintain statistic of the aggregate

Delays are OK. Approximation is not!

Example – User Profiling

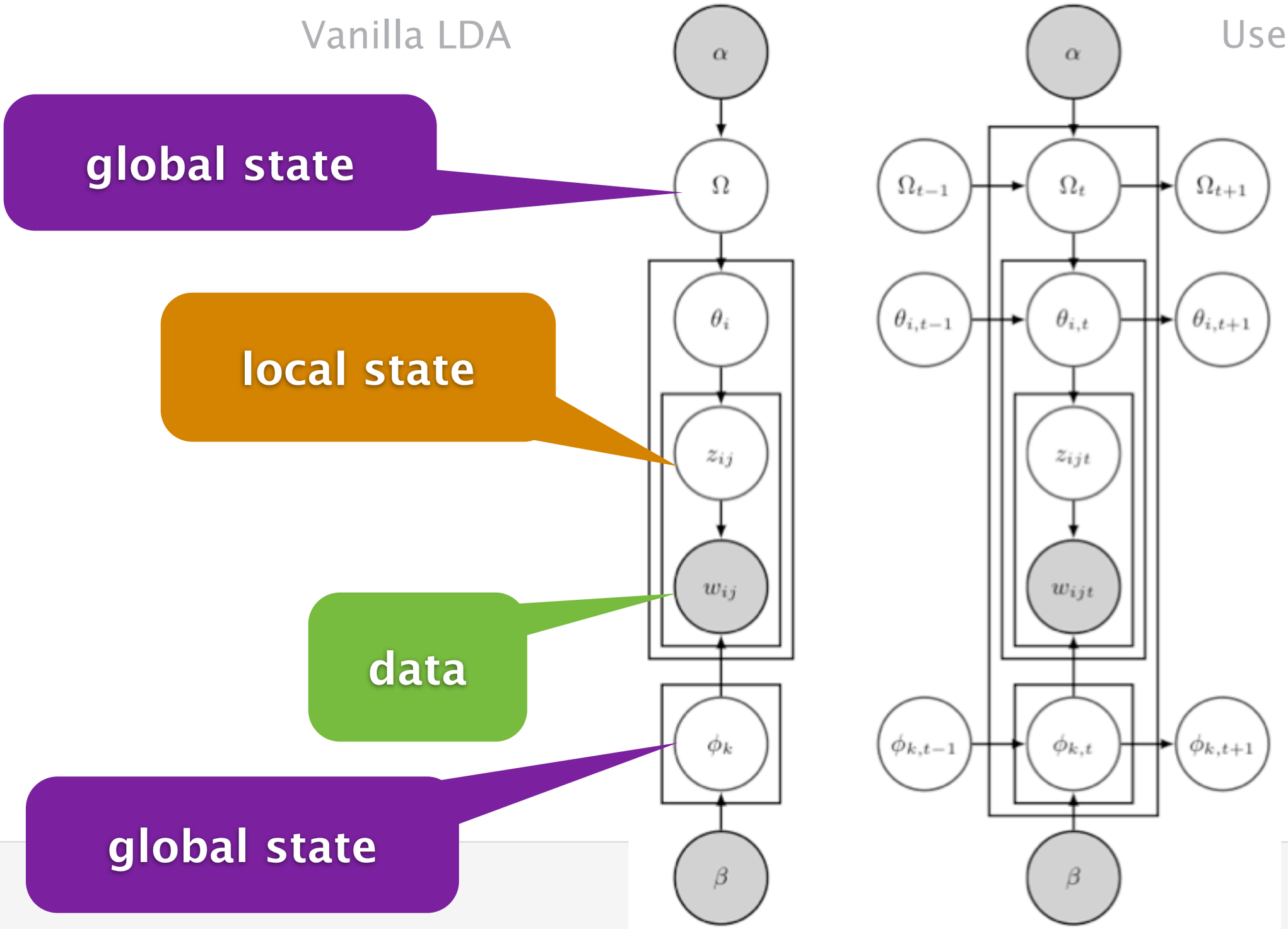
Vanilla LDA



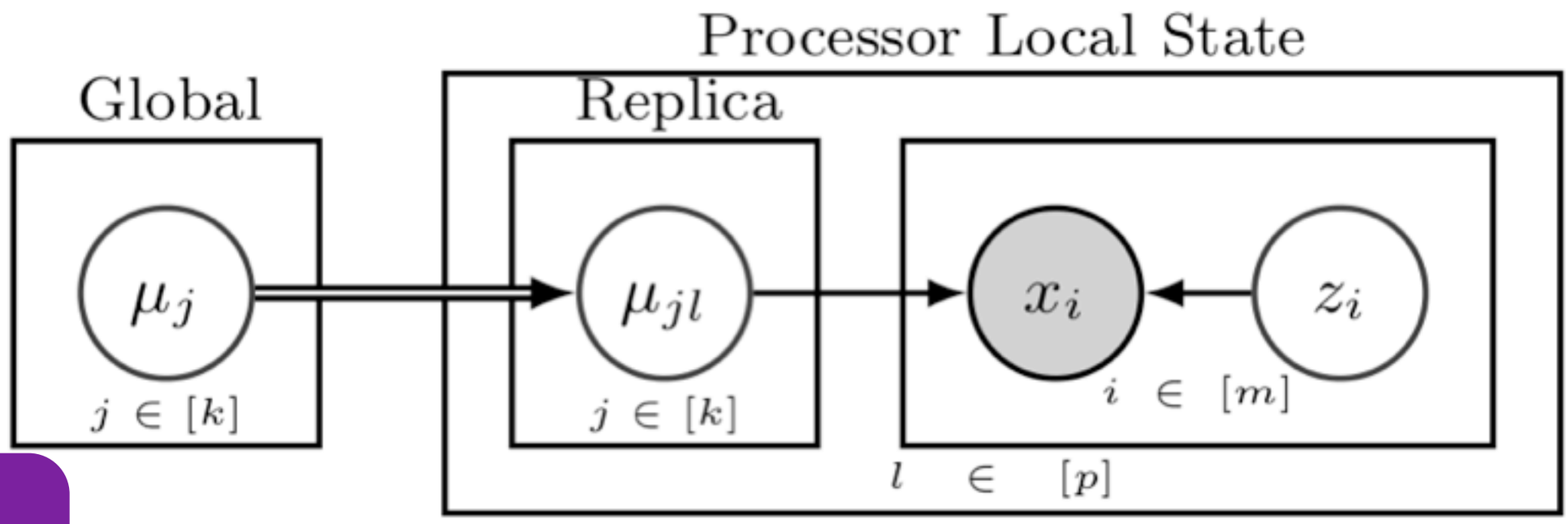
Example – User Profiling

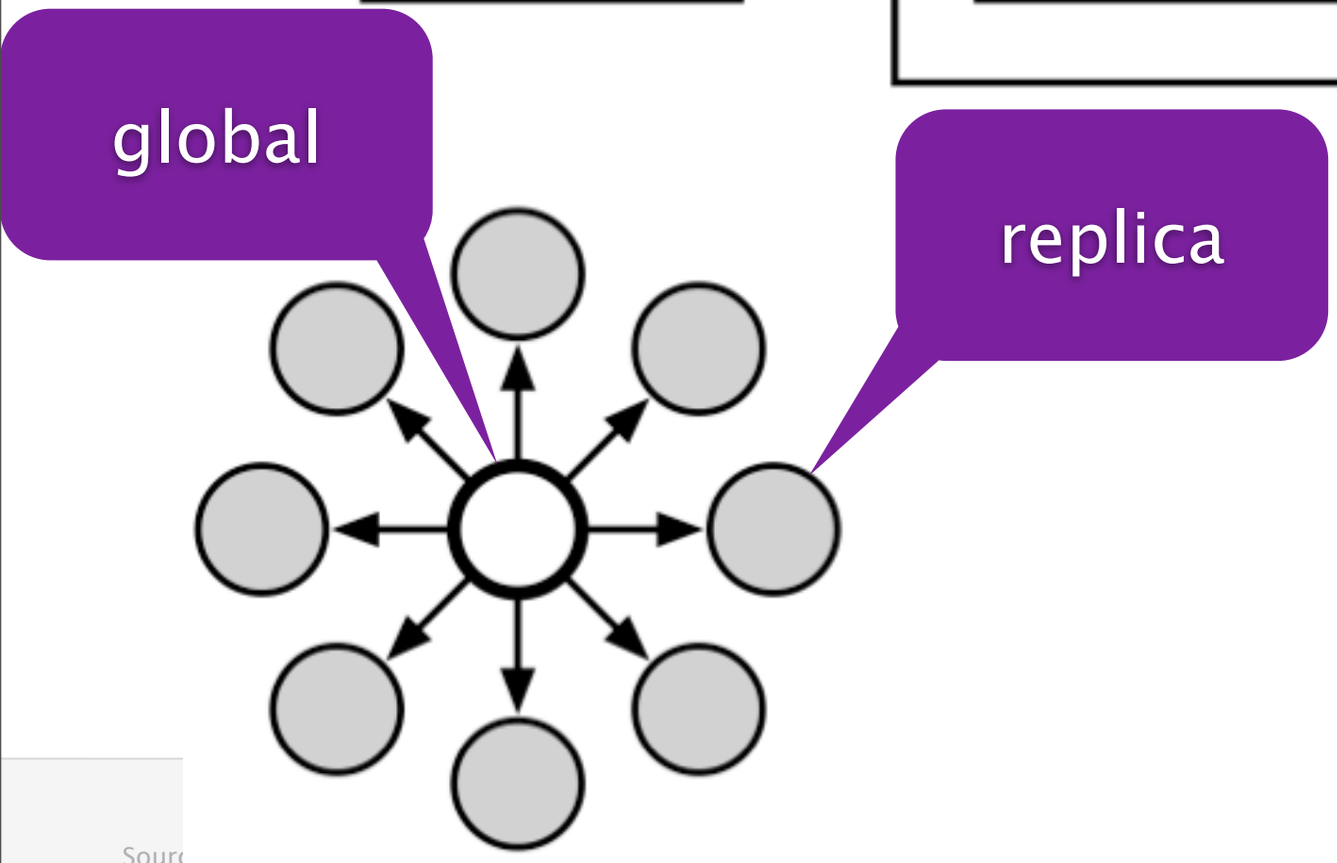
Vanilla LDA

User profiling

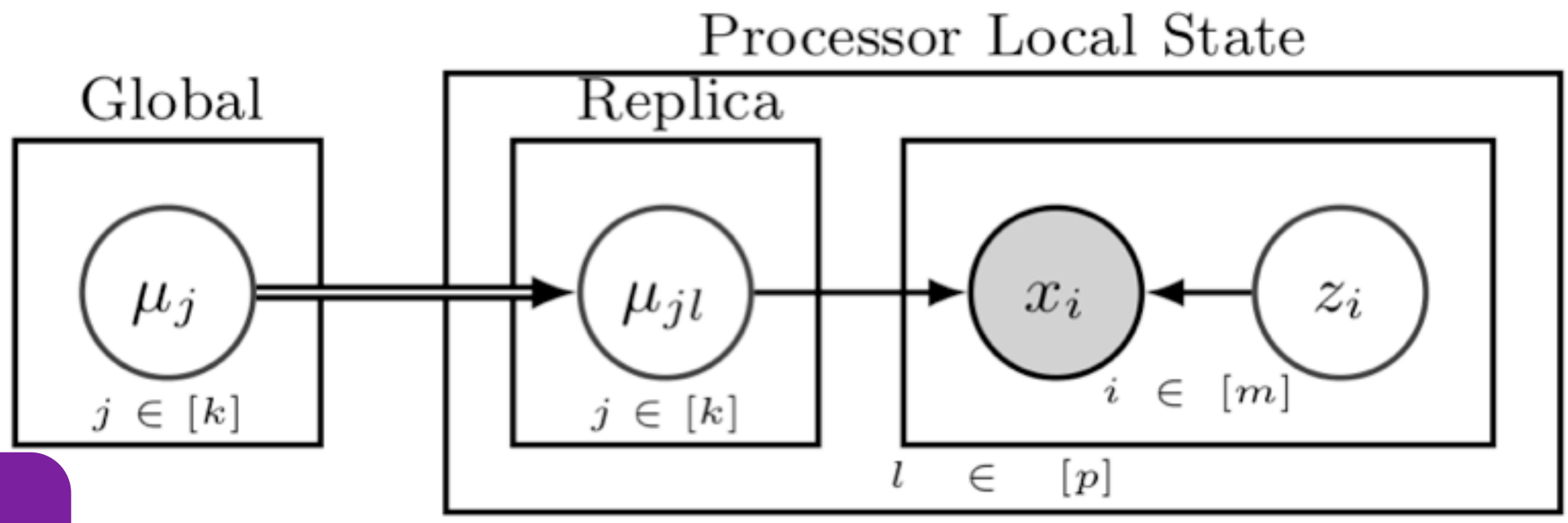


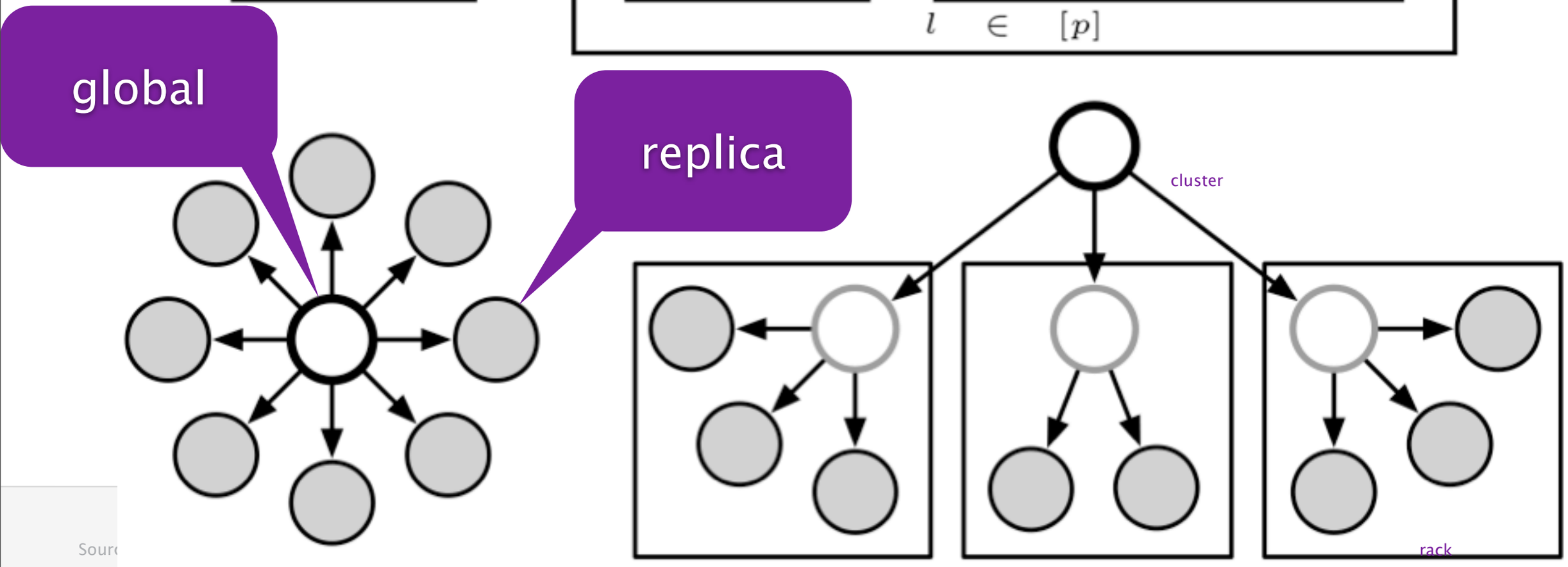
Distribution





Distribution





Synchronization

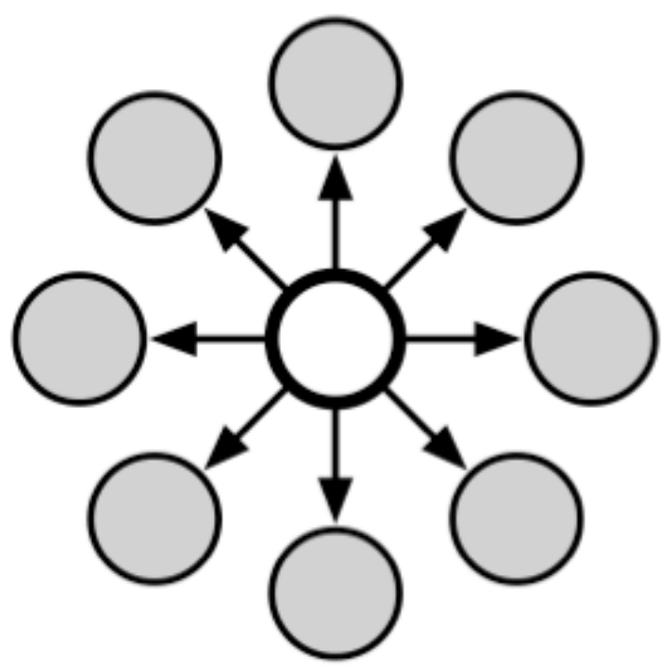
- Child updates local state
  - Start with common state
  - Child stores old and new state
  - Parent keeps global state
- Transmit differences asynchronously
  - Inverse element for difference
  - Abelian group for commutativity (sum, log-sum, cyclic group, exponential families

local to global

$$\delta \leftarrow x - x^{\text{old}}$$

$$x^{\text{old}} \leftarrow x$$

$$x^{\text{global}} \leftarrow x^{\text{global}} + \delta$$



global to local

$$x \leftarrow x + (x^{\text{global}} - x^{\text{old}})$$

$$x^{\text{old}} \leftarrow x^{\text{global}}$$

Synchronization

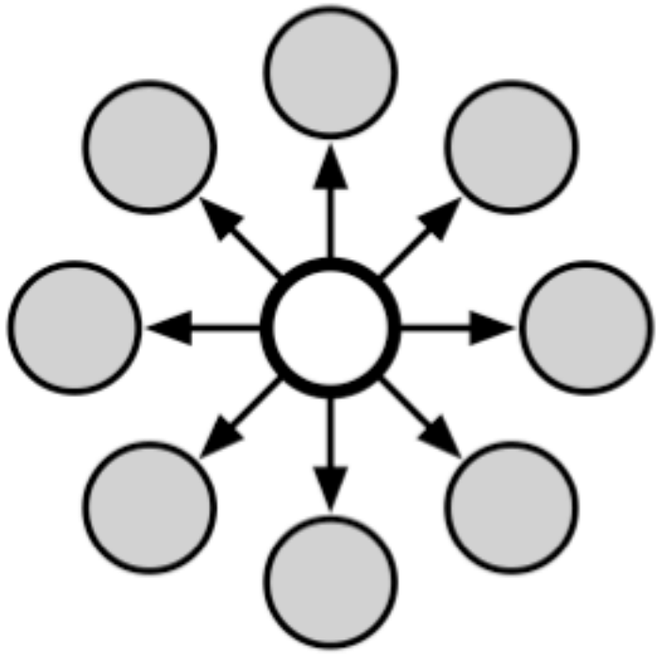
- Naive approach (dumb master)
  - Global is only (key,value) storage
  - Local node needs to lock/read/write/unlock master
  - Needs a 4 TCP/IP roundtrips – latency bound
- Better solution (smart master)
  - Client sends message to master / in queue / master incorporates it
  - Master sends message to client / in queue / client incorporates it
  - Bandwidth bound (>10x speedup in practice)

local to global

$$\delta \leftarrow x - x^{\text{old}}$$

$$x^{\text{old}} \leftarrow x$$

$$x^{\text{global}} \leftarrow x^{\text{global}} + \delta$$

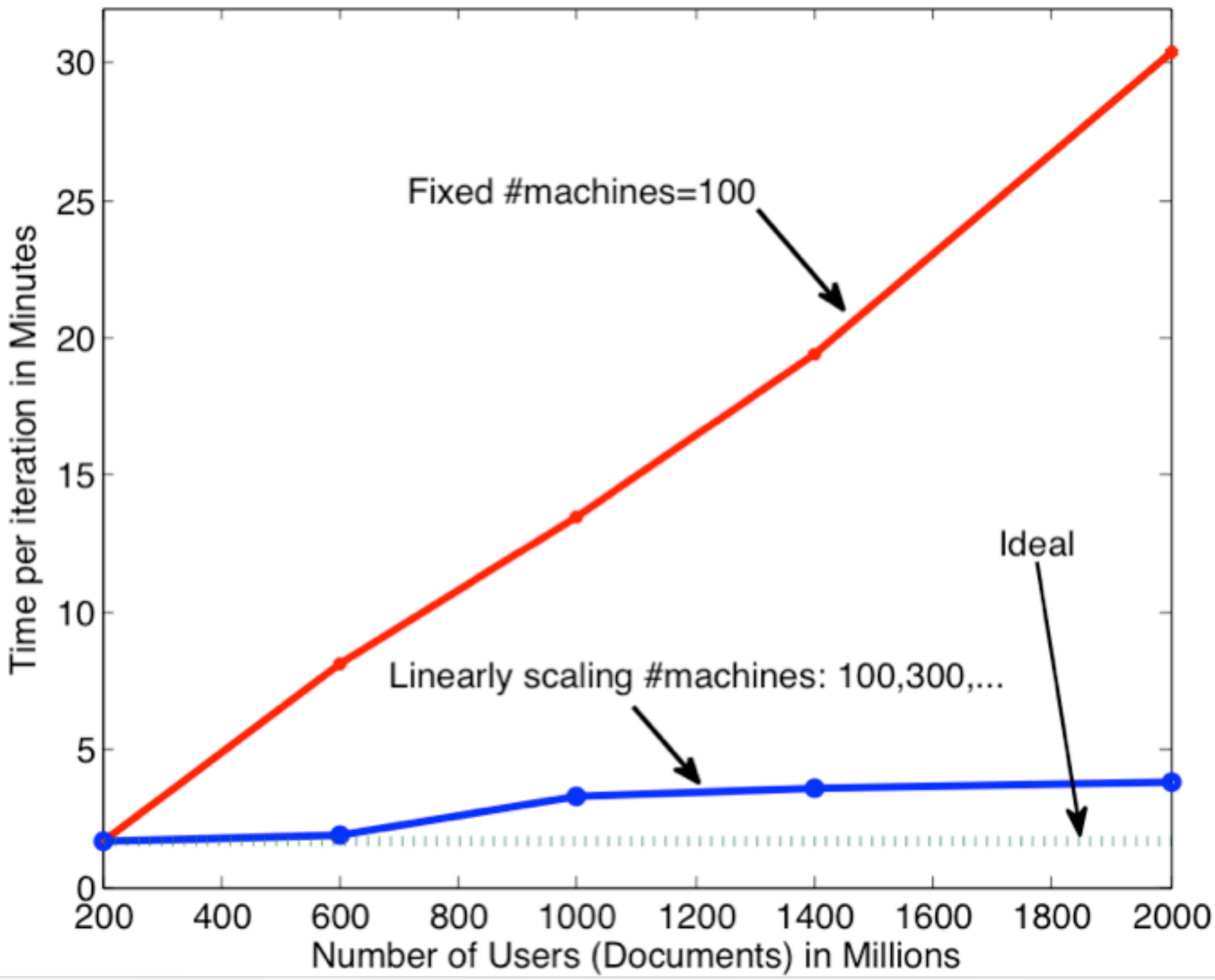


global to local

$$x \leftarrow x + (x^{\text{global}} - x^{\text{old}})$$

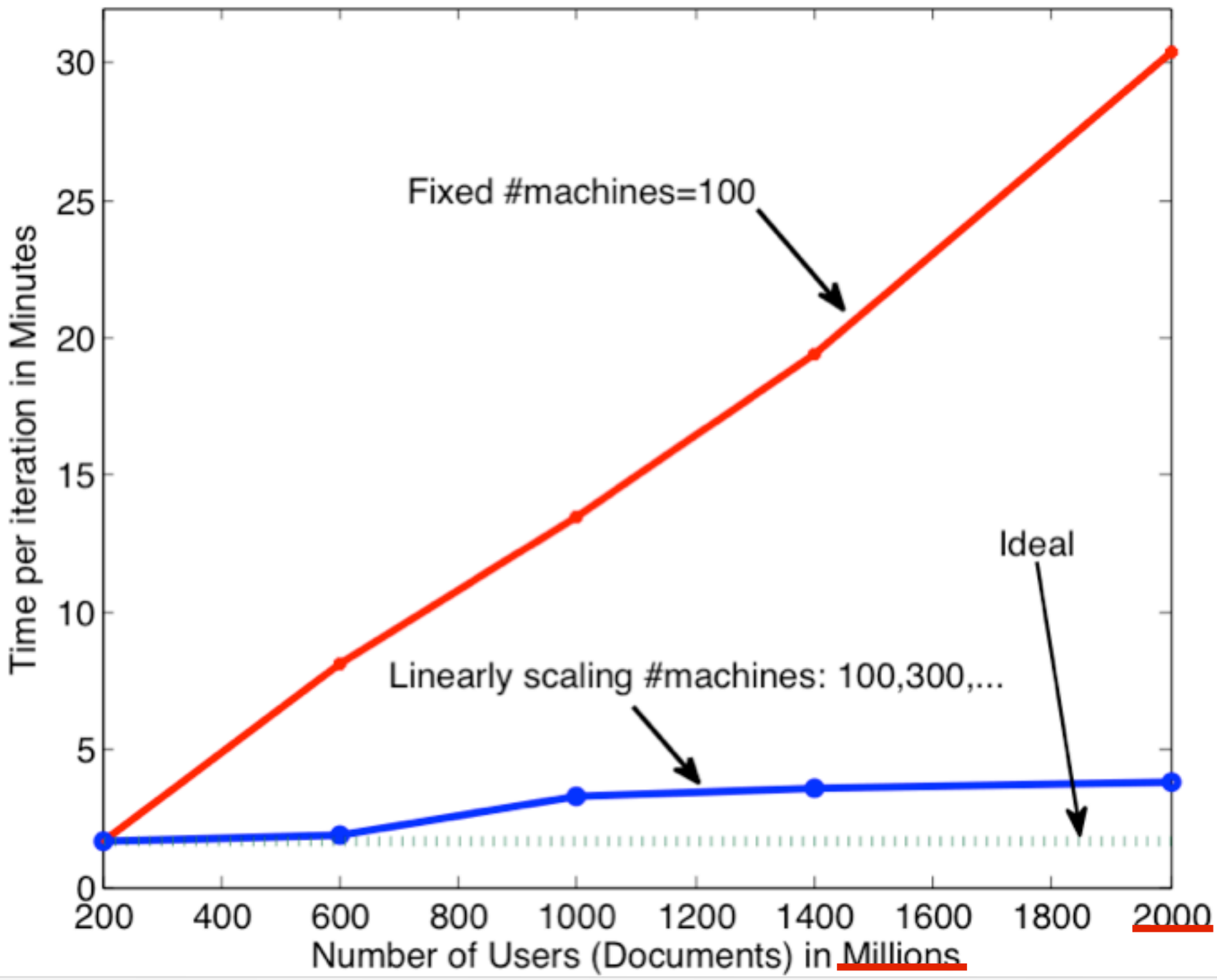
$$x^{\text{old}} \leftarrow x^{\text{global}}$$

Weak scaling (more data = more machines)



Source: place source info here

Weak scaling (more data = more machines)



Source: place source info here

Exact Synchronization in a Nutshell

- Each machine computes local updates
- Inference relative to local (stale) version of the model
- Send local changes to global
- Receive global changes at local client

- Only send / receive aggregate changes
- Easy change relative to single machine implementation
- Not fault tolerant (need to restart system if single machine fails)
- Delays may destroy convergence properties


MITT'S
MAGIC
Etch A Sketch
SCREEN
Approximate
Synchronization
&
Dual
Decomposition

Motivation – Distributed Optimization

- Distributed optimization problem

$$f(x) = \sum_i f_i(x)$$

- Decompose over p processors
- Make progress on subproblems $f_i(x_i)$ per processor
- Exchange updates with parameter server (difference & state)
- Retrieve related state from parameter server and update locally (do not assume that the x_i yield an orthogonal decomposition)
- Difference to exact parameter server: stochastic gradient descent updates

Properties

- Fault tolerant
  - Restart server(s) from last backup state
  - No need to restart entire system when individual machines fail
- Works well for deep belief networks
  - See Google Brain project
    Paper by Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng
- Fails to converge for graph factorization (tried and tested ...)
  - Initial convergence but parameters diverge subsequently
  - Caused by delay in parameter updates
    (local updates overcompensate changes to parameter values)

Dual Decomposition to the rescue

- Optimization problem

$$\underset{x}{\text{minimize}} \sum_i f_i(x)$$

$$\text{or equivalently} \quad \underset{x_i, z}{\text{minimize}} \sum_i f_i(x_i) \text{ subject to } x_i = z$$

- Lagrangian relaxation

$$L(x_i, z, \lambda) = \sum_i f_i(x_i) + \lambda \sum_i \|x_i - z\|^2$$

update x, z and Lagrange multipliers. **Include equality constraint if needed.**

- This explicitly deals with different values in local state and global consensus.

Synchronous Variant (MapReduce

- Lagrangian relaxation

$$L(x_i, z, \lambda) = \sum_i f_i(x_i) + \lambda \sum_i \|x_i - z\|^2$$

- Local step (Map step)
  Solve local minimization problems on each machine

$$\underset{x_i}{\text{minimize}}\, f_i(x_i) + \lambda \|x_i - z\|^2$$

- Global step (Reduce step + intermediate)
  - Aggregate local solutions and average to compute new value of z
  - Update Lagrange multiplier
  - Rebroadcast to local clients

Asynchronous Variant

- Lagrangian relaxation

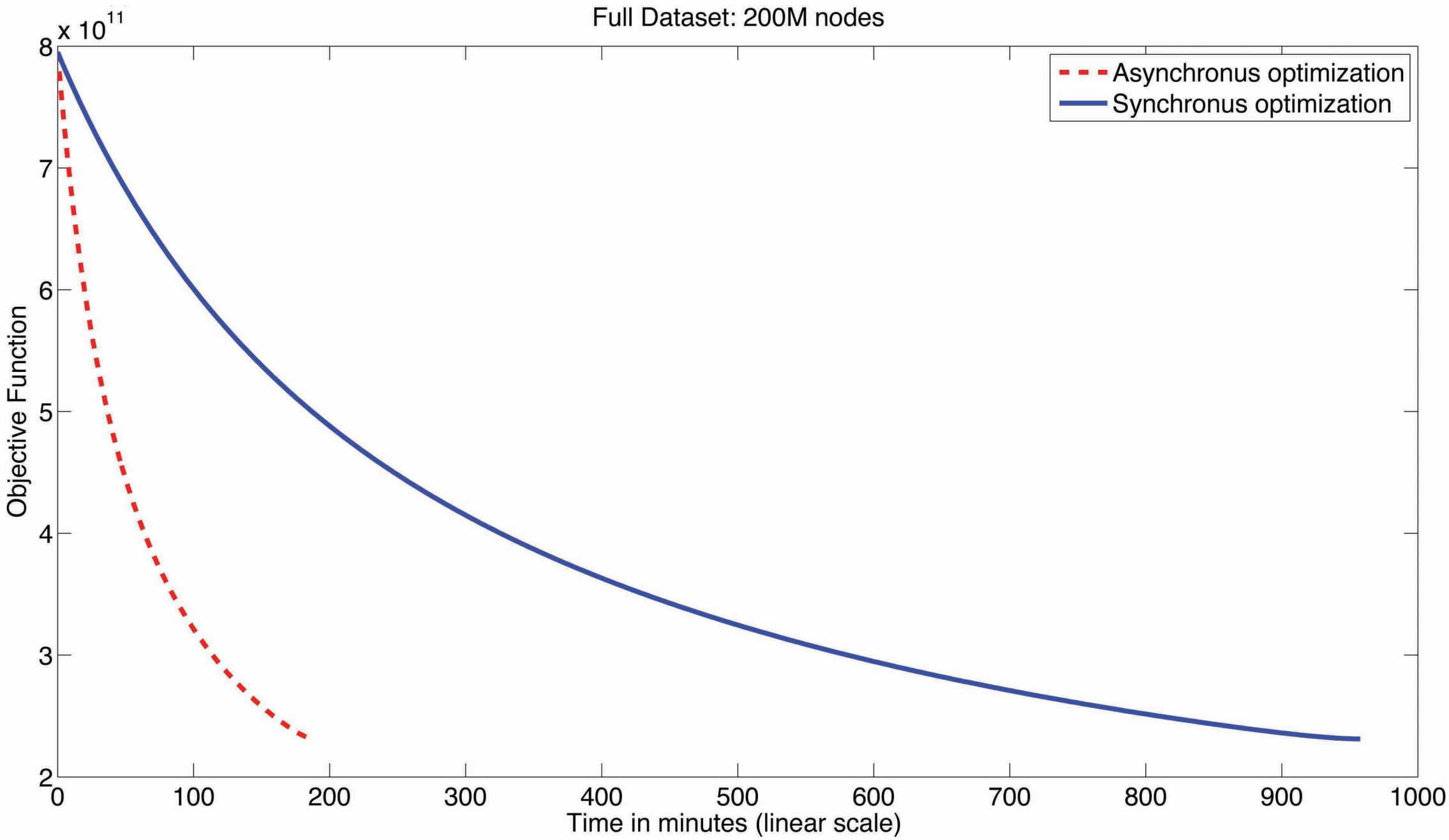
$$L(x_i, z, \lambda) = \sum_i f_i(x_i) + \lambda \sum_i \|x_i - z\|^2$$

- Local step (continous)
  Solve local minimization problems (e.g. via SGD) and send updates to server

$$\underset{x_i}{\text{minimize}}\, f_i(x_i) + \lambda \|x_i - z\|^2$$

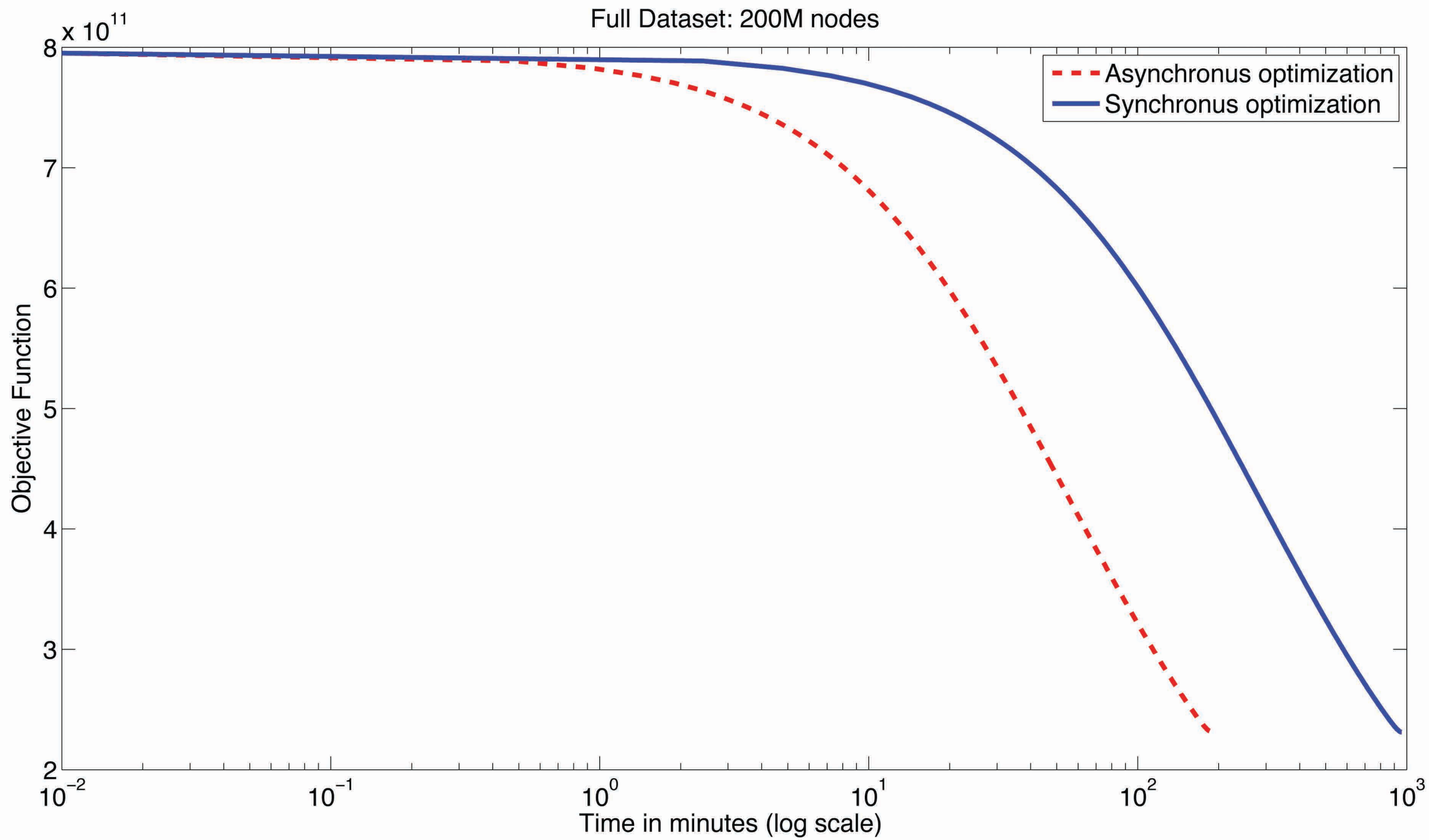
- Global step (continous)
  - Aggregate local solutions asynchronously from clients
  - Update Lagrange multiplier
  - Rebroadcast global state to local clients

Convergence (synchronous vs. asynchronous)

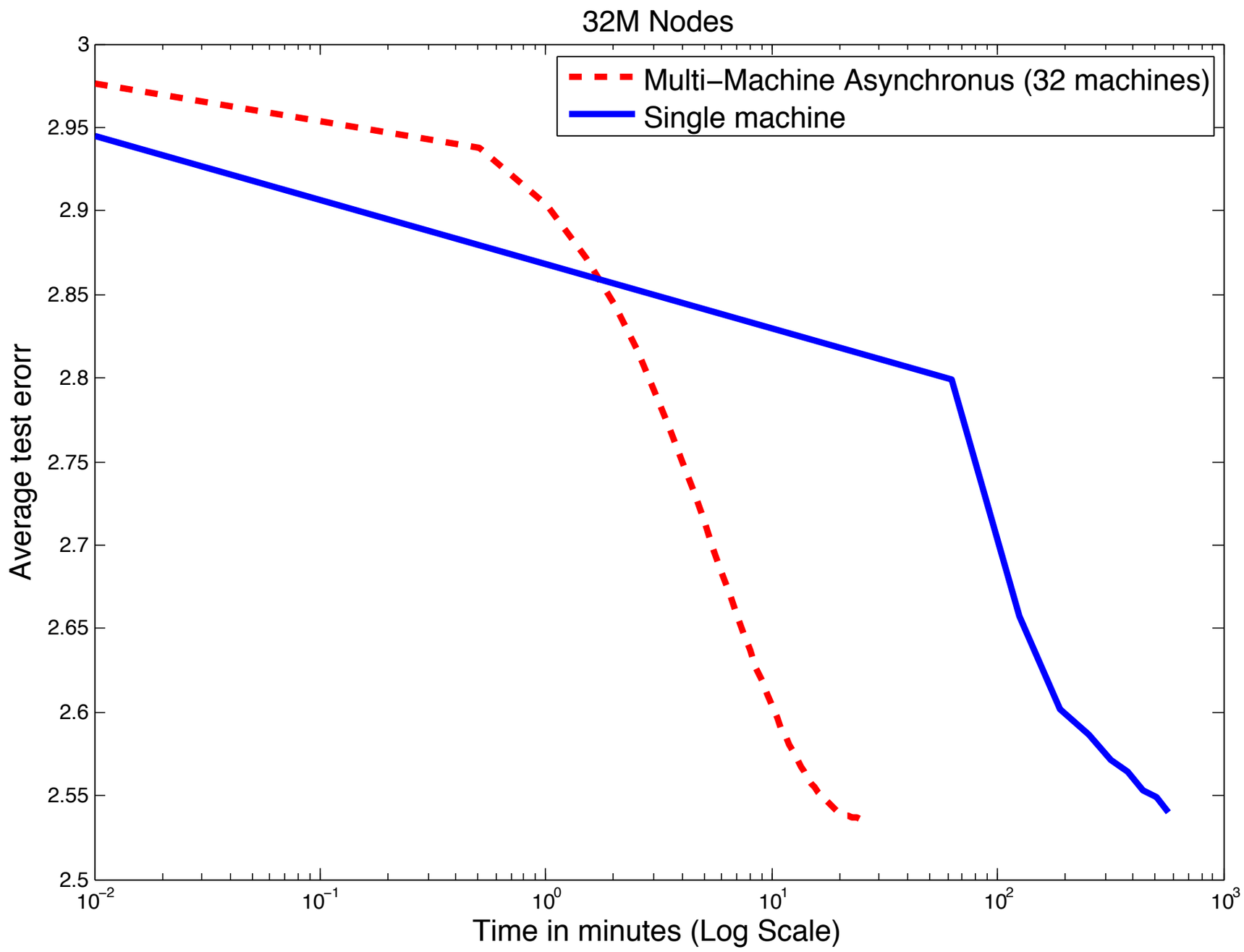


Source: place source info here

Convergence (synchronous vs. asynchronous)

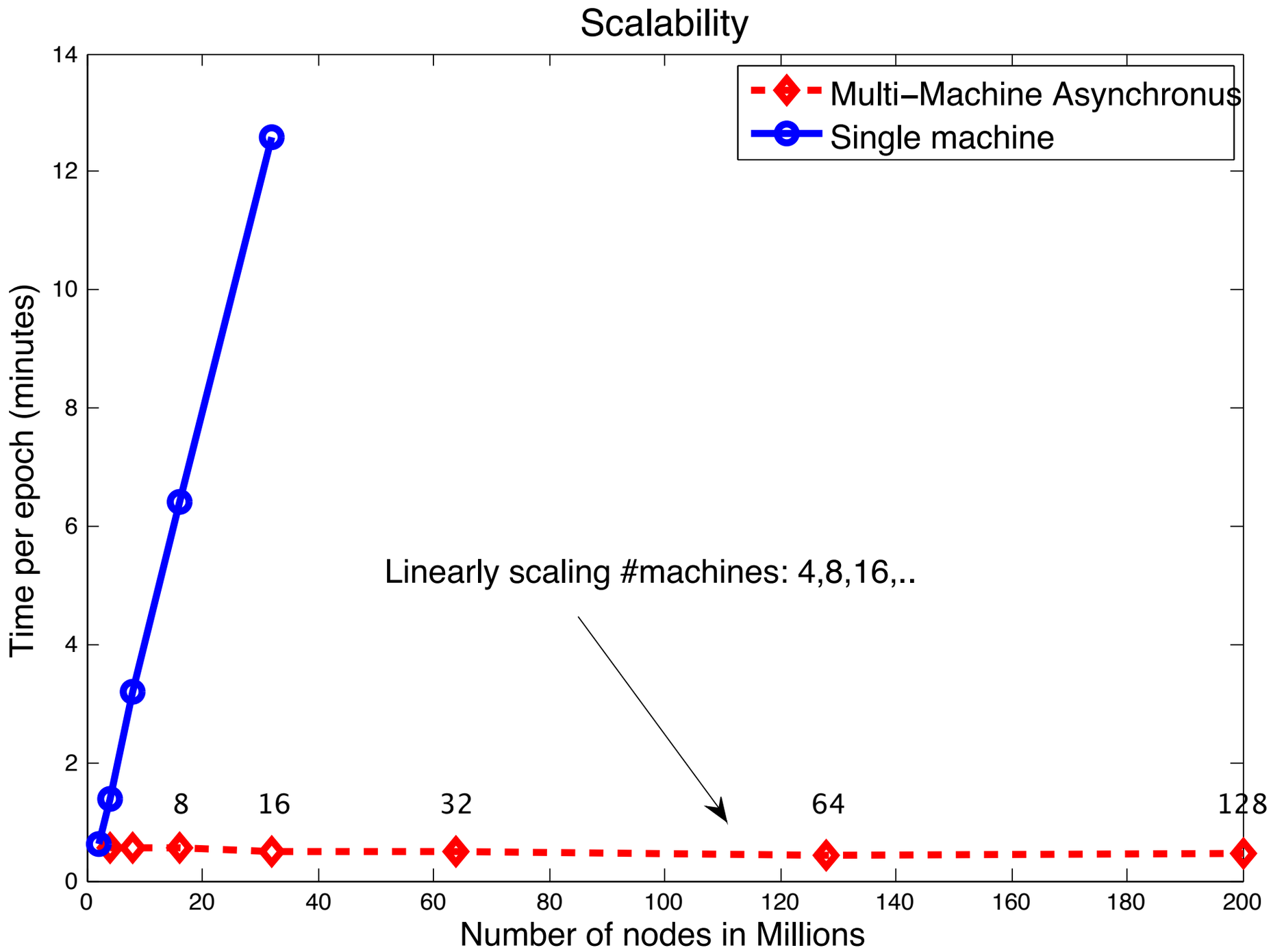


Acceleration (single CPU vs. 32 machines)



Source: place source info here

Weak scaling (more data = more machines)



Even more parameter server variants

- Graphlab (PowerGraph decomposition and updates)
- Facebook parameter server for EP updates
- Google brain project
- Graph factorization

... your algorithm here ...

- From January 2013 on at CMU
  Open source version (ping me if you want to contribute)

Google


MITT'S
MAGIC Etch A Sketch SCREEN
• Multicore
• stochastic gradient descent
(Hogwild, Slow learners are fast, ...)
• Multiple machines
• exact synchronization
• approximate synchronization &
dual decomposition