High Performance Computing
8^{th} lecture

Kazuki Osawa : 16M30444

25 Oct. 2016

Selected paper

Exploiting iterative-ness for parallel ML computations

Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky,
Qirong Ho, Gregory R. Ganger, Phillip B. Gibbons*, Garth A. Gibson, Eric P. Xing
Carnegie Mellon University, *Intel Labs

- SoCC'14 3-5 Nov. 2014, Seattle, Washington, USA.
- ACM 978-1-4503-3252-1.

http://dl.acm.org/citation.cfm?doid=2670979.2670984

The contributions of this paper

1. Identify **iterative-ness** in ML applications

2. Specializations for **exploiting iterative-ness**

3. Concept of “**virtual iteration**”

Abstract

Machine learning applications

- Optimization problem
  - Find the “optimal” parameter values
  - The chosen model match the input data
- Many ML applications use iterative algorithms
- Same pattern of access to parameters
- Can and should be exploited

Parameter server approach

- Share model parameters among worker threads
- Exploiting the repeating pattern
  - **Reduce dynamic** cache and server structures
  - Use **static** pre-serialized structures
  - Inform **prefetch** and **partitioning** decisions
  - Data placement **avoiding contention and slow accesses**

Experiments

- 3 target ML applications
  - Collaborative Filtering (CF)
  - Topic Model (TM)
  - PageRank (PR)
- Exploitation reduce per-iteration time by 33-98%
- Robust to variation in the patterns

Outline

1. Introduction
2. Iterative ML and systems
3. Exploiting iterative-ness for performance
4. Implementation
5. Evaluation
6. Conclusion

Outline

1. Introduction
2. Iterative ML and systems
3. Exploiting iterative-ness for performance
4. Implementation
5. Evaluation
6. Conclusion

ML approaches

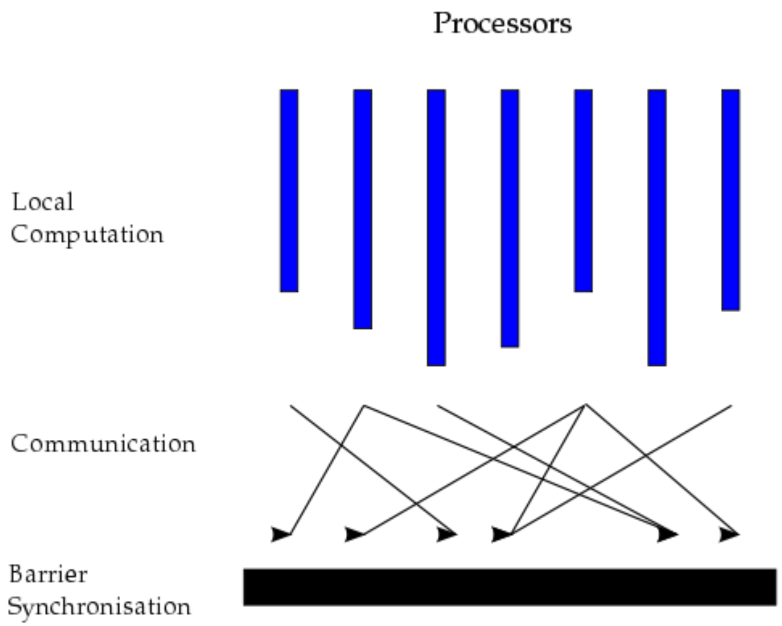
- Determine model parameter best fit input data
- Algorithm iterates over the input data
- Refine current best estimate of parameter values

Parallelizing ML computations

- Partition input data among worker threads
- Worker threads across cores and machines
- Share only parameter values
- Maintain distributed values by parameter server
- Synchronize each iteration with a barrier
- BSP (: Bulk Synchronous Parallel) style

Bulk Synchronous Parallel Model

- A number of components
- A router deliver messages between 2 components
- Facilities for synchronizing components



Knowable repeating patterns

- Each thread processes
  - Its portion of the input data
  - In the same order in each iteration
- Same subset of parameters are read & updated
- Each iteration involves the same pattern

Exploiting patterns

- Within a machine
  - State can be placed in memory NUMA zone
  - Closest to the core on which it runs
  - Reduce lock contention
  - Synchronize only when required
- Cross-machine overheads
  - Partitioning
  - Prefetching
- Static structure for servers’ and workers’ state

Outline

1. Introduction
2. Iterative ML and systems
3. Exploiting iterative-ness for performance
4. Implementation
5. Evaluation
6. Conclusion

Iterative fitting of model parameters

- Major subset of ML approaches
- Process a set of input data
- Identify mathematical model that fits data
- Minimize an objective function that describes error

Parallel computation model

- “Big Data” required for detail model
- Partition input data among the worker thread
- iterative ML based on BSP

Parameter server architecture

- All state shared among worker threads
- Kept in key-value store
- Worker threads process assigned input data
  - READ
  - UPDATE
  - CLOCK

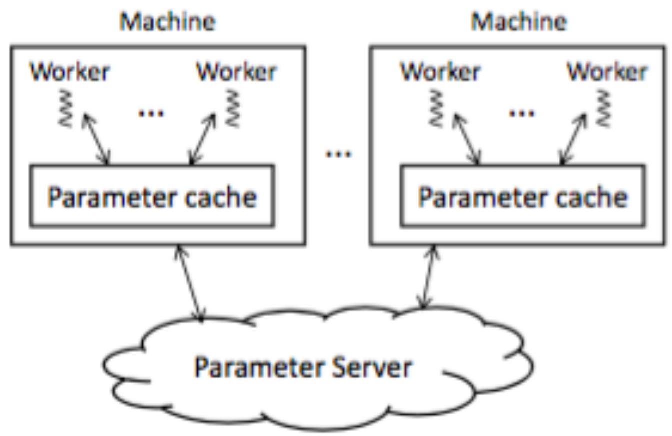


Figure 1. Parallel ML with parameter server.

Example applications

- Collaborative Filtering
  - Used in recommender systems
  - Discover latent interactions between two entities
- Topic Model
  - Unsupervised method
  - Discovering hidden semantic structures
- PageRank
  - Assign weighted score to every vertex in a graph
  - Score measures its importance in the graph

Outline

1. Introduction
2. Iterative ML and systems
3. Exploiting iterative-ness for performance
4. Implementation
5. Evaluation
6. Conclusion

Obtaining per-iteration access sequence

- Two options
  - Explicit reporting of the sequence
  - Explicit reporting of the iteration boundaries
- Report access sequence once
- Report at beginning

```
// Original
init_params()
ps.clock()
do {
  do_iteration()
  ps.clock()
} while (not stop)
```

```
// Gather in first iter
init_params()
ps.clock()
do {
  if (first iteration)
    ps.start_gather(real)
  do_iteration()
  if (first iteration)
    ps.finish_gather()
  ps.clock()
} while (not stop)
```

```
// Gather in virtual iter
ps.start_gather(virtual)
do_iteration()
ps.finish_gather()
init_params()
ps.clock()
do {
  do_iteration()
  ps.clock()
} while (not stop)
```

Virtual iteration

- Each application thread reports operations for an iteration (READ, UPDATE, CLOCK)
- No real values are involved
  - Very fast
- Require not so much coding effort

  ⇒Virtual iteration
- Require too much coding effort

  ⇒Explicit identification of iteration boundary

Identification of iteration boundaries

- Identify the start & end of an iteration
- Remove the need for pattern recognition
- Allow the parameter server to transition to more efficient operation after 1st iteration
- Involve some overheads
    - Initialization & 1st iteration are not iterative-ness specialized

Exploiting access information

- Data placement across machines
- Data placement inside a machine
- Static per-thread caches
- Efficient data structures
- Prefetching

Data placement across machines

- If parameters are co-located with computation that use them
  - Communication demands & latency can be reduced
- Accessing of each input data
  - Involve only a subset of the parameters
- Accessing of parameters by different workers
  - With different frequencies

Data placement inside a machine

- Modern multi-core machines
  - Multiple sockets
  - Multiple memory NUMA zones
- Memory access speed depending on “distance”
- Knowledge of access sequences
  - Co-locate worker threads & data
  - They access frequently to the same NUMA memory zone

Static per-thread caches

- Per-worker-thread caching
  - Contention between worker threads
  - Access to remote NUMA memory zone
- Employing a static cache policy
  - The best set of entries to be cached
  - Never evicts them

Efficient data structures

- Knowledge of access pattern
  - Knowledge of full set of entries
- More efficient, less general data structure
- Reducing marshaling overhead by eliminating the need to extract and marshal each value one-by-one

Prefetching

- Each worker thread must use updated value after each CLOCK (BSP)
- Prefetching can help mask the high latency
- Knowing access pattern maximize the potential value of prefetching
- Constructing large batch prefetch requests once and using them each iteration is more efficient

Outline

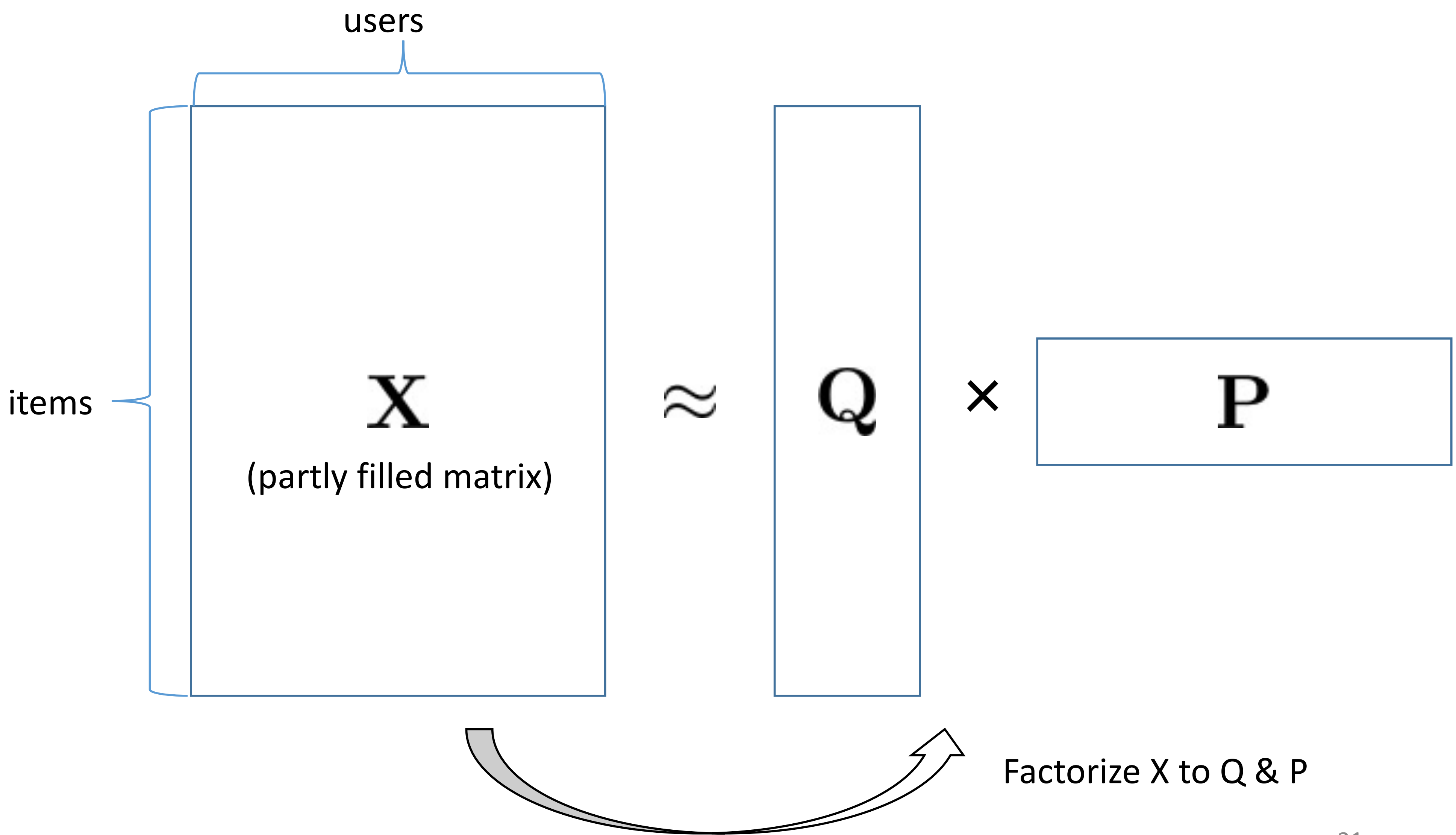
1. Introduction
2. Iterative ML and systems
3. Exploiting iterative-ness for performance
4. Implementation
5. Evaluation
6. Conclusion

Collaborative Filtering

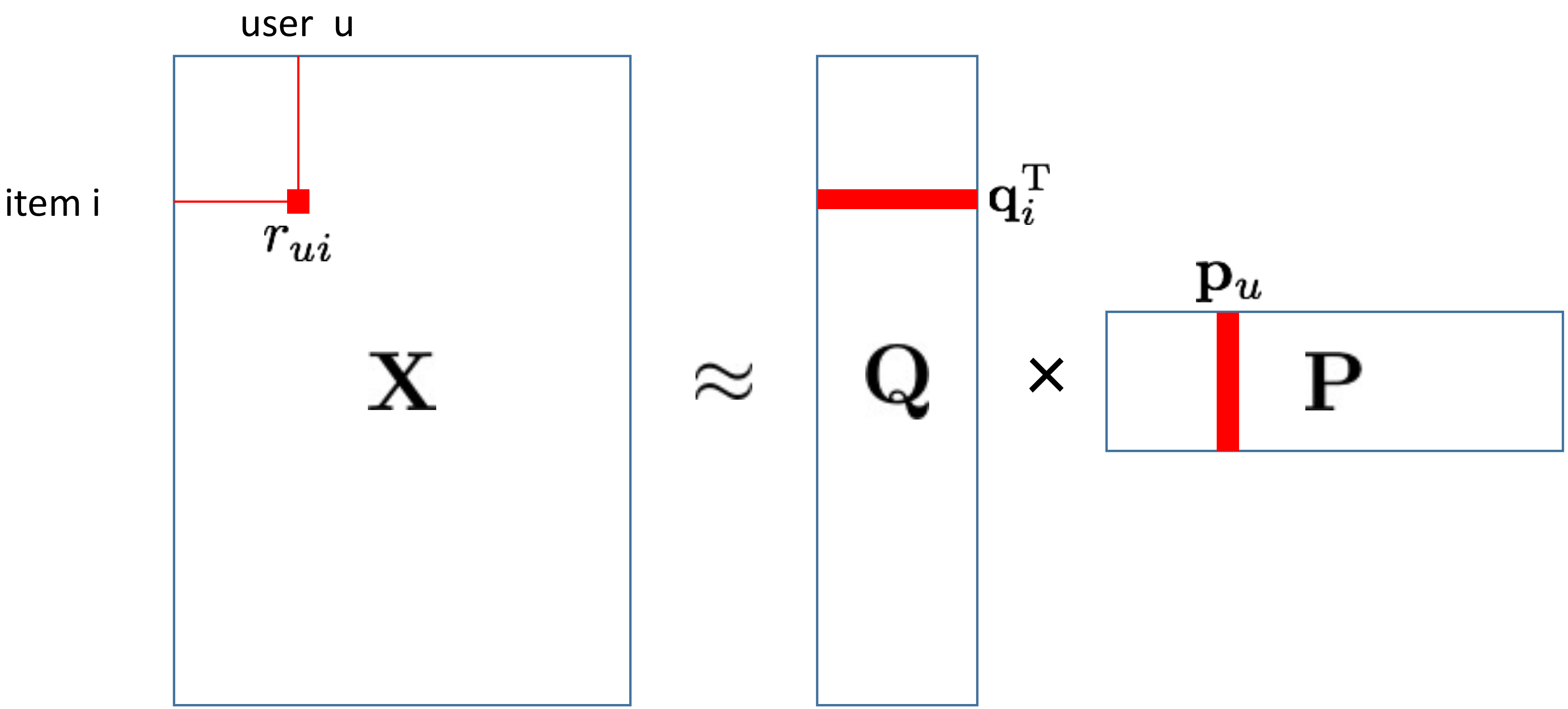
- Commonly used in recommender systems
- e.g. recommending movies to users on Netflix

	user1	user2	user3	user4	user5
item1	5		3	2	
item2		4	4	2	1
item3	3	5	3	4	3
item4	3			1	
item5	1	2	3	?	3

Collaborative Filtering



Collaborative Filtering



r_{ui} : User u's rating of item i

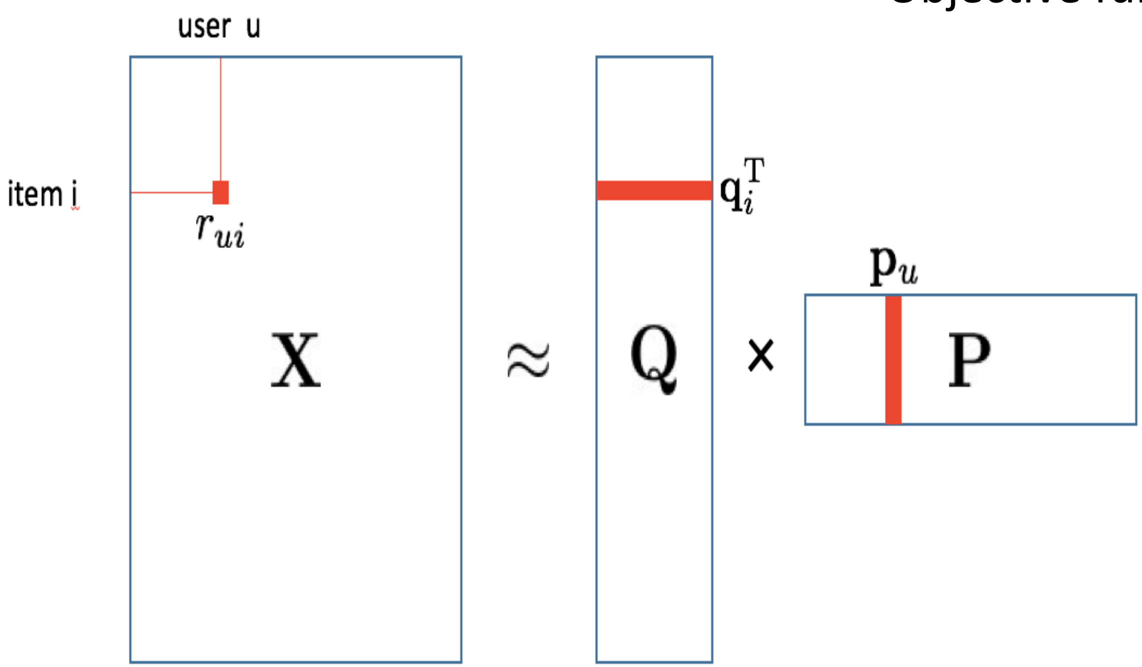
$\hat{r}_{ui} = q_i^T p_u$: Estimate

SGD for CF

Optimize parameters

$$\min_{q^*,p^*} \sum_{(u,i)\in K} (r_{ui} - q_i^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

Objective function



Update parameters

- $q_i \leftarrow q_i + \gamma \cdot (e_{ui} \cdot p_u - \lambda \cdot q_i)$
- $p_u \leftarrow p_u + \gamma \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u)$

$$e_{ui} \overset{def}{=} r_{ui} - q_i^T p_u$$

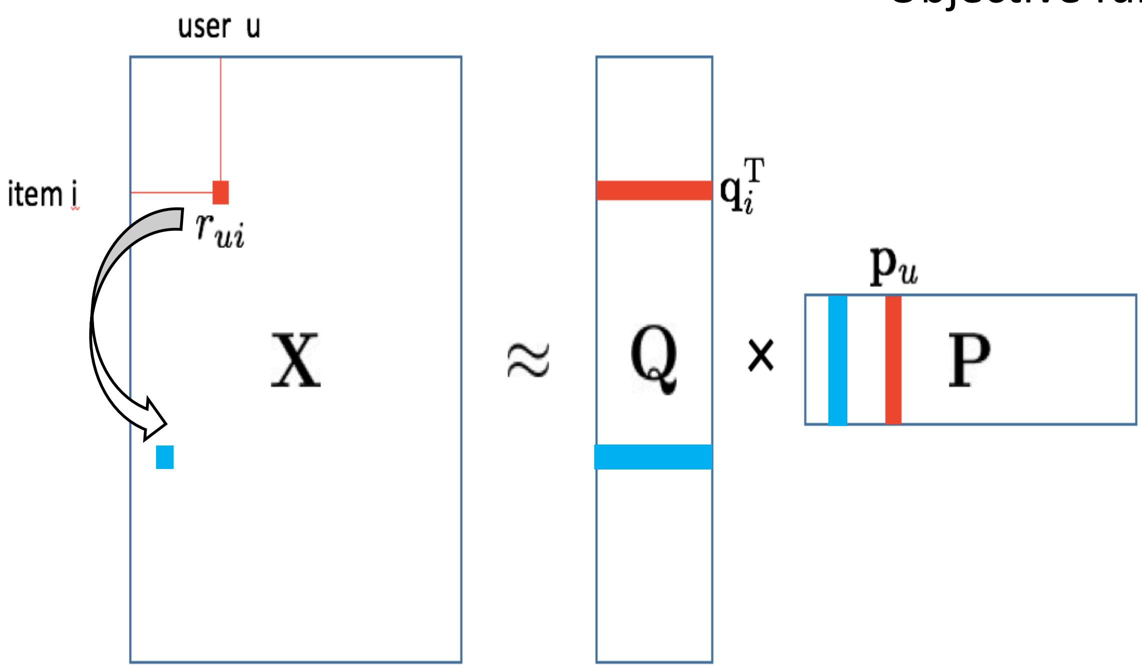
γ : *learning rate*

SGD for CF

Optimize parameters

$$\min_{q^*,p^*} \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

Objective function



Select next input data randomly

Update parameters

- $q_i \leftarrow q_i + \gamma \cdot (e_{ui} \cdot p_u - \lambda \cdot q_i)$
- $p_u \leftarrow p_u + \gamma \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u)$

$$e_{ui} \overset{def}{=} r_{ui} - q_i^T p_u$$

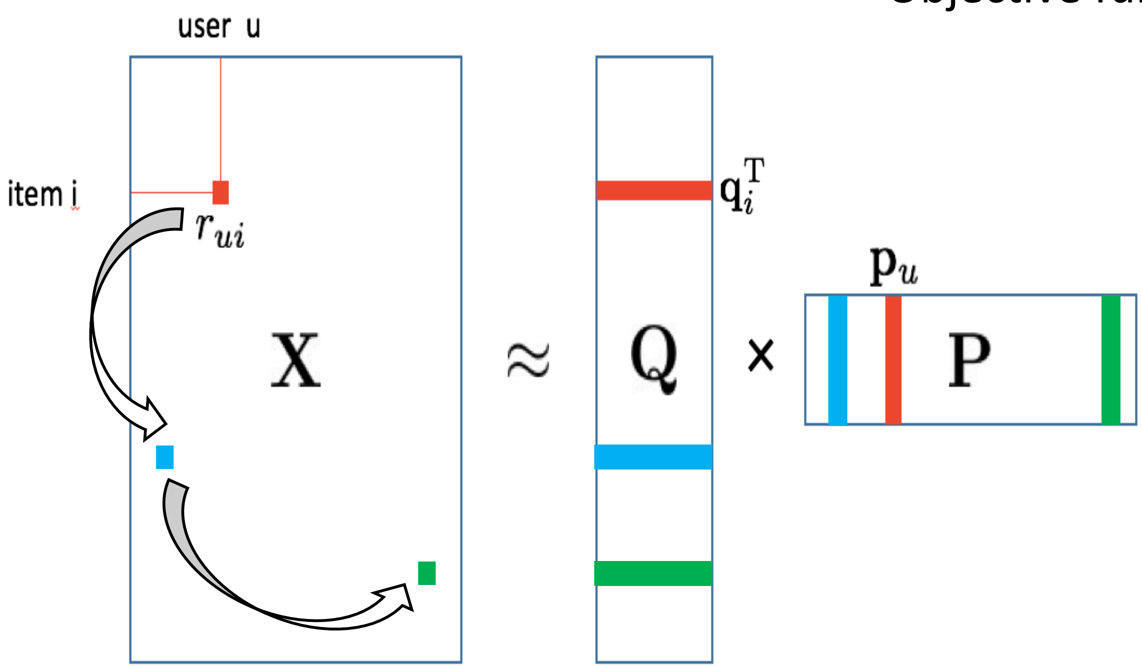
γ : *learning rate*

SGD for CF

Optimize parameters

$$\min_{q^*,p^*} \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

Objective function



Select next input data randomly

Update parameters

- $q_i \leftarrow q_i + \gamma \cdot (e_{ui} \cdot p_u - \lambda \cdot q_i)$
- $p_u \leftarrow p_u + \gamma \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u)$

$$e_{ui} \overset{def}{=} r_{ui} - q_i^T p_u$$

γ : *learning rate*

Parameter server for CF

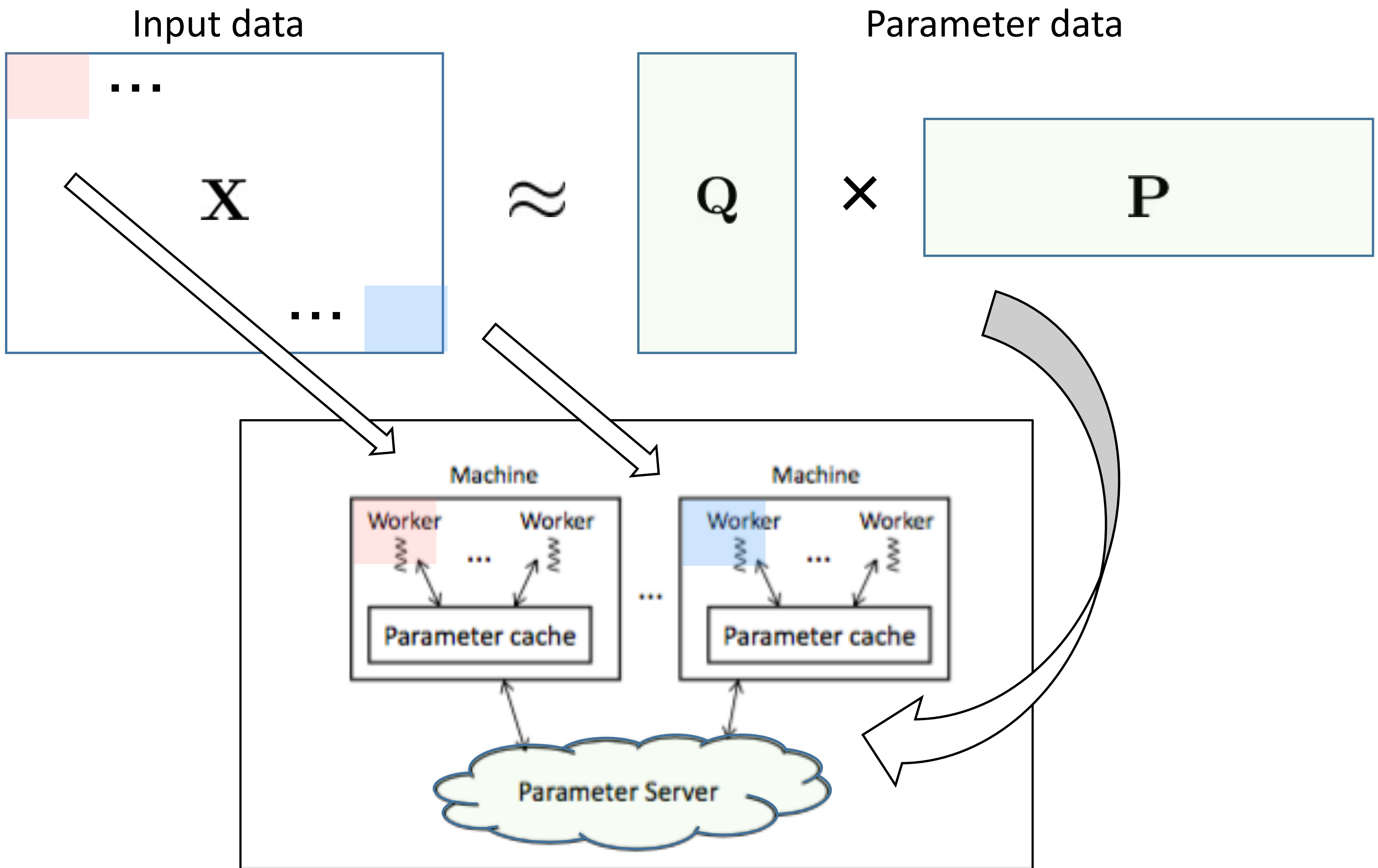
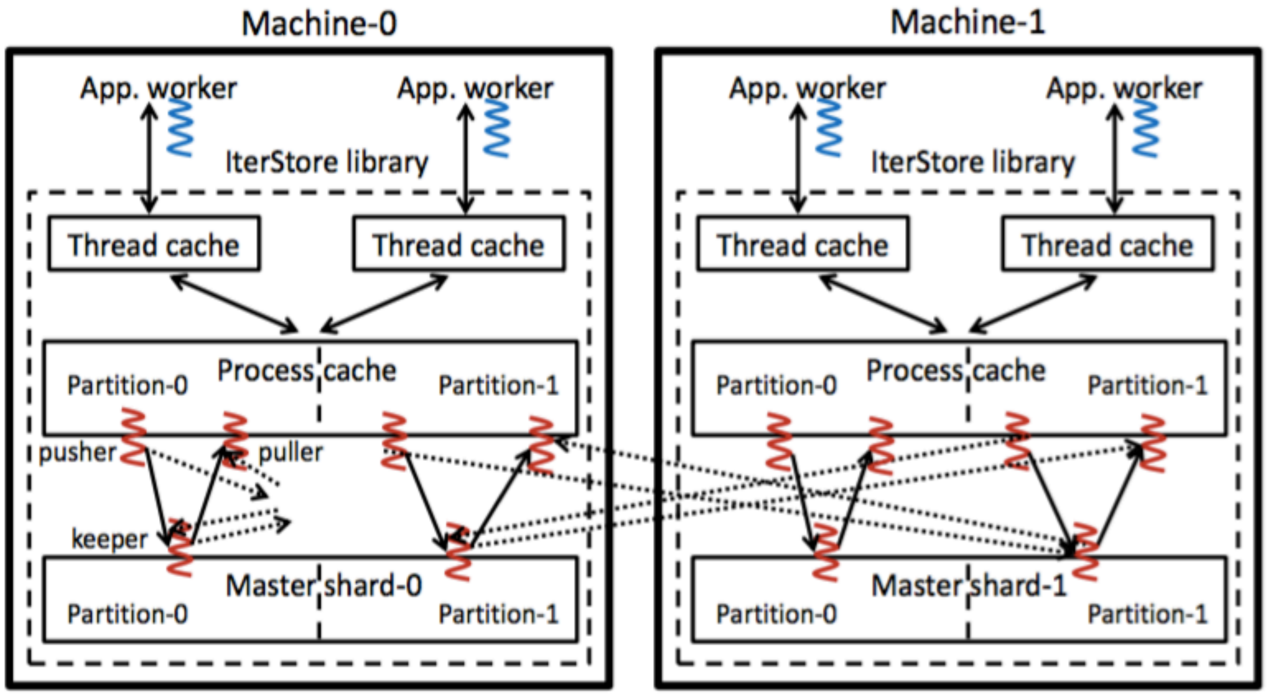


Figure 1. Parallel ML with parameter server.

IterStore (parameter server)



- Shard of the master version of data in its master store (not duplicated)
- App. threads access the process cache
- IterStore follows BSP model
- Master stores are devided into M partitions
- N IterStore machines manage the parameter data

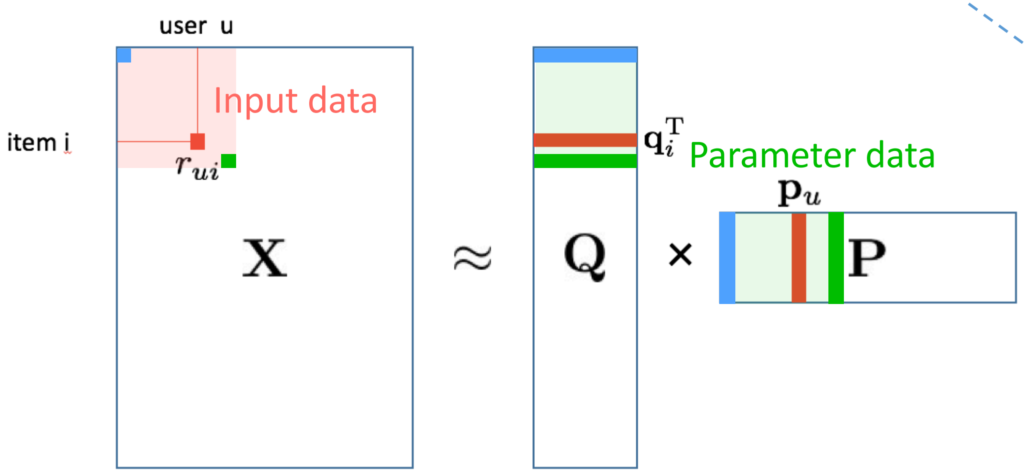
Parameter data in IterStore (CF)

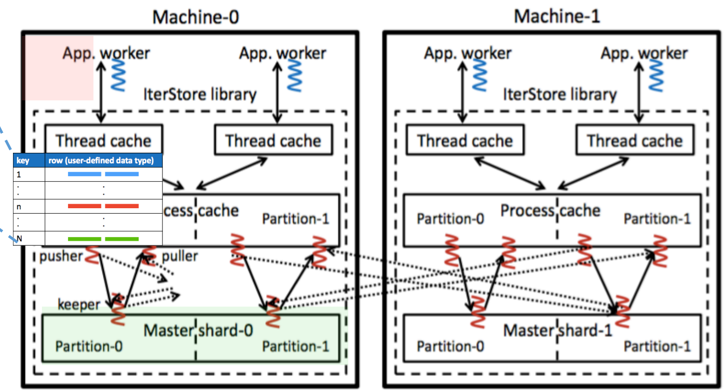
Manage data as a collection of rows indexed by keys.

key	row (user-defined data type)
1	
:	:
k	
:	:
K	

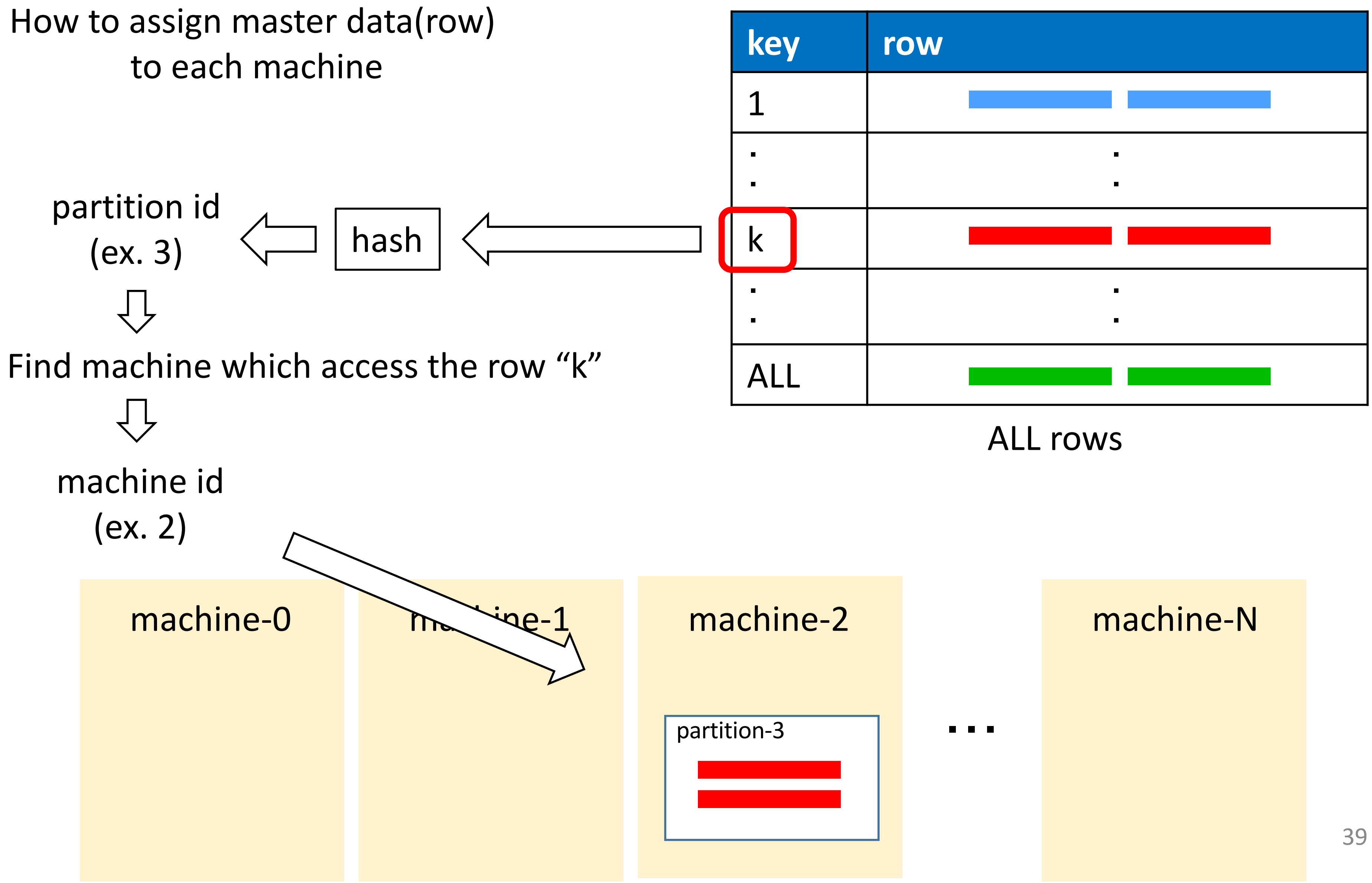
rows in Machine-0

- Fixed size
- Serializable
- Defined with an associative aggregation operation

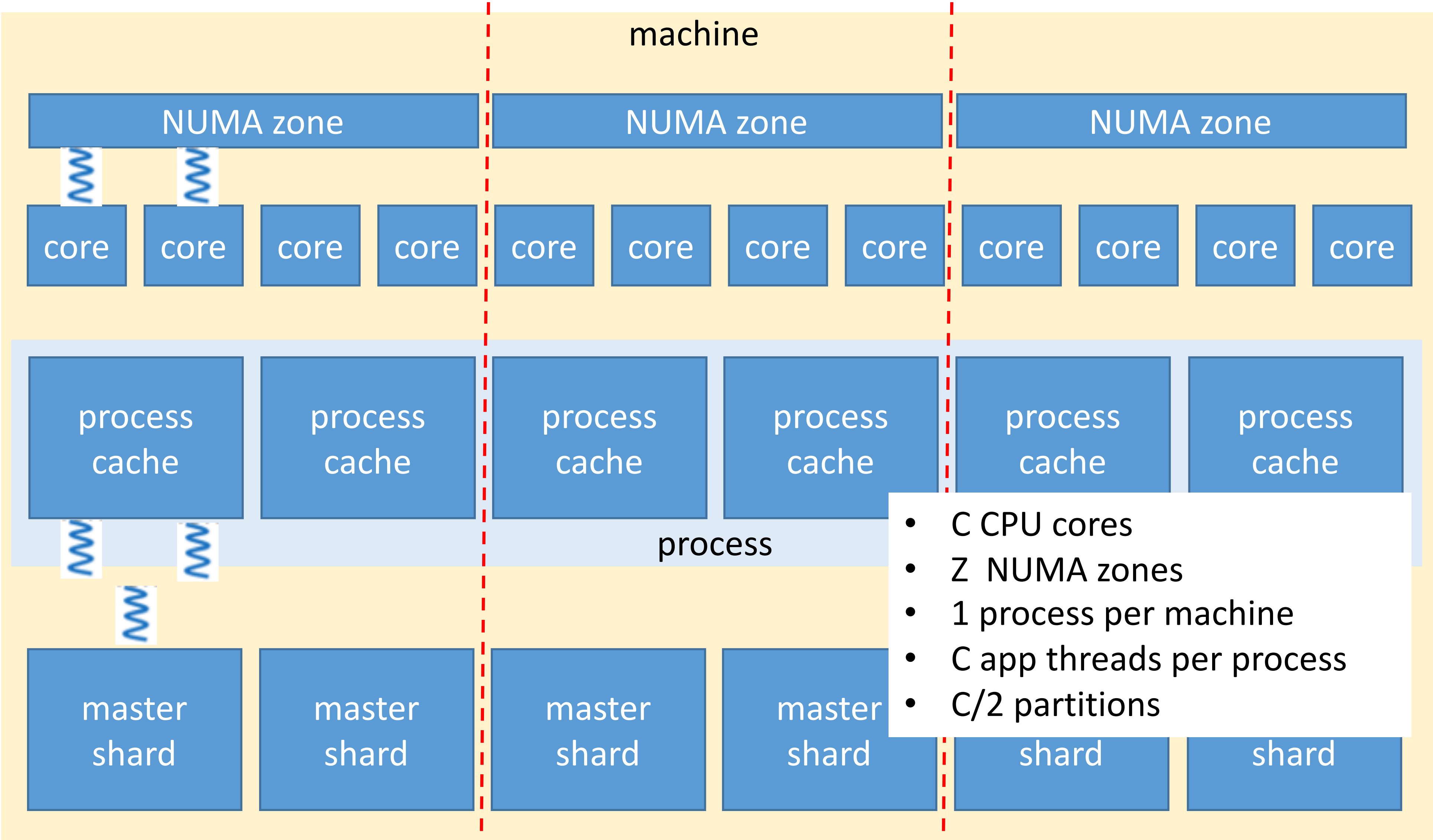




Data placement across machine



Data placement inside a machine



- C CPU cores
- Z NUMA zones
- 1 process per machine
- C app threads per process
- C/2 partitions

Contention and locality-aware thread caches

$$
\begin{aligned}
\mathrm{CP}_i^{(j)} &= 1 - \prod_{i' \neq i} (1 - \mathrm{AP}_{i'}^{(j)}) \\
&= 1 - \prod_{i' \neq i} (1 - \mathrm{AF}_{i'}^{(j)}) \\
&\leq 1 - (1 - \max_{i' \neq i} \mathrm{AF}_{i'}^{(j)})^{n_j - 1} \\
&\leq 1 - (1 - \max_{i' \neq i} \mathrm{AFT}_{i'}^{(j)})^{n_j - 1} \\
&= 1 - (1 - \frac{\mathrm{CPB}}{n_j - 1})^{n_j - 1} \\
&\approx (n_j - 1) \times \frac{\mathrm{CPB}}{n_j - 1} \\
&= \mathrm{CPB}
\end{aligned}
$$

key	row (user-defined data type)
1	
. .	. .
j	
. .	. .
K	

rows in a Machine

$AP_i^{(j)}$	access probability	thread-i accesses row-j
$AF_i^{(j)}$	access frequency	
$AFT_i^{(j)}$	AF threshold	
$CP_i^{(j)}$	contention probability	
CPB	CP bound	

To reduce all CP to below CPB,

set $AFT_i^{(j)} = \dfrac{CPB}{n_j - 1}$

and if $AF_i^{(j)} > AFT_i^{(j)}$

make thread-i cache row-j in its thread cache

Outline

1. Introduction
2. Iterative ML and systems
3. Exploiting iterative-ness for performance
4. Implementation
5. Evaluation
6. Conclusion

Experimental setup

- Hardware Information
  - 8-node cluster of 64-core machines
  - Each node has four 2-die 2.1 GHz 16 core AMD Opteron 6272 packages, with a total of 128GB of RAM and eight memory NUMA zones
  - The nodes run Ubuntu 12.04 and are connected via an Infiniband network interface

Application benchmarks

- CF
  - Netflix dataset
  - 480k-by-18k sparse matrix with 100m known elements
- TM
  - Nytimes dataset
  - 100m tokens in 300k documents
  - Vocabulary size of 100k
  - Generate 1000 topics
- PR
  - Twitter-graph dataset
  - 40m nodes and 1.5b edges

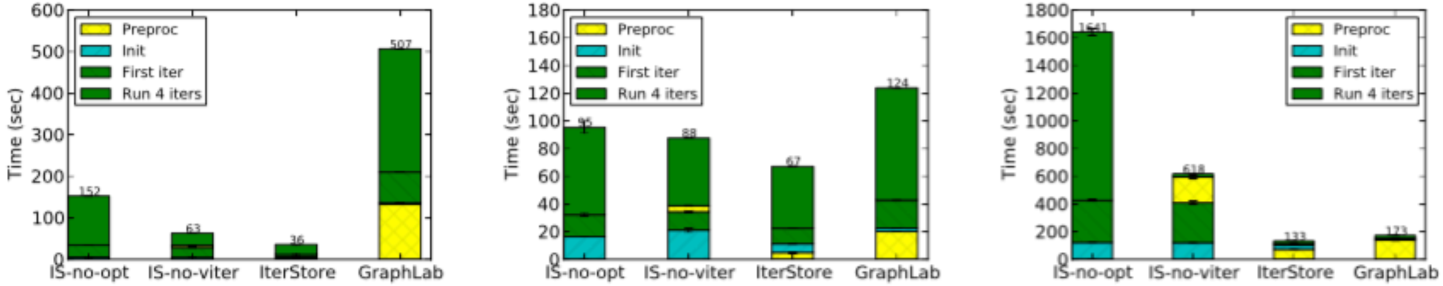
App.	# of rows	Row size (bytes)	Ave. node degree
CF	500k	8k	200
TM	400k	8k	250
PR	40m	8	75

Table 1. Features of benchmarks.

IterStore setup

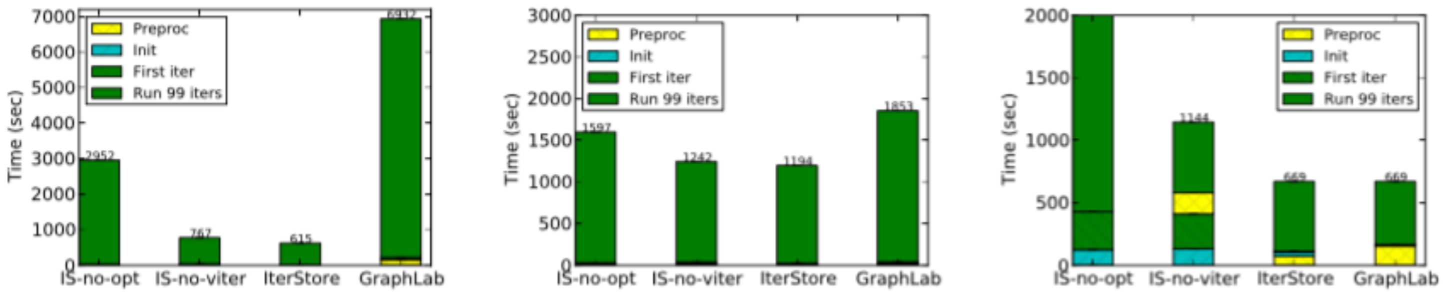
- One application process on each machine
- Each machine creates 64 computation threads
- Each machine is linked to one instance of IterStore library with 32 partitions
- Assume each machine has enough memory to not need replacement in its process cache

Overall performance



(a) Collaborative Filtering (b) Topic Modeling (c) PageRank

Figure 4. Performance comparison, running 5 iterations.



(a) Collaborative Filtering (b) Topic Modeling (c) PageRank

Figure 5. Performance comparison, running 100 iterations. The “IS-no-opt” bar in the PageRank figure is cut off at 2000 sec, because it’s well over an order of magnitude worse than the other three.

Optimization effectiveness break down

(a) Turn on one of the optimizations.

(b) Turn off one of the optimizations.

(c) Preprocessing time break down.

Figure 6. Optimization effectiveness break down.

Contention and locality-aware caching

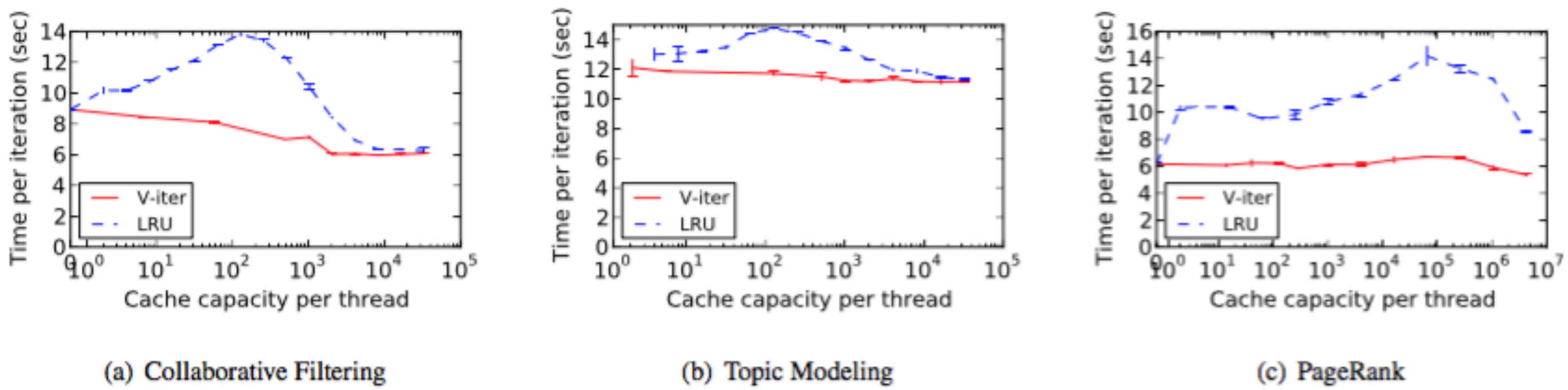


Figure 7. Comparing IterStore's static cache to LRU, varying the cache size (log scale).

IterStore's static thread-caching policy vs LRU(Least Recently Used) policy

Pipelined prefetching

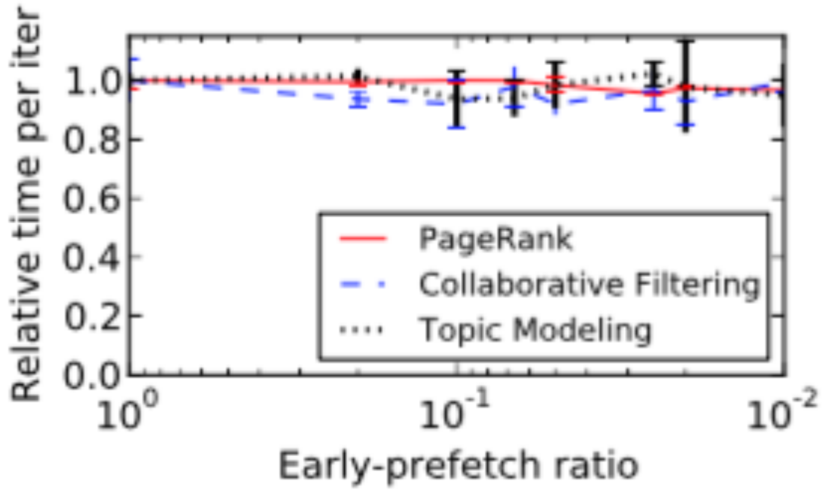
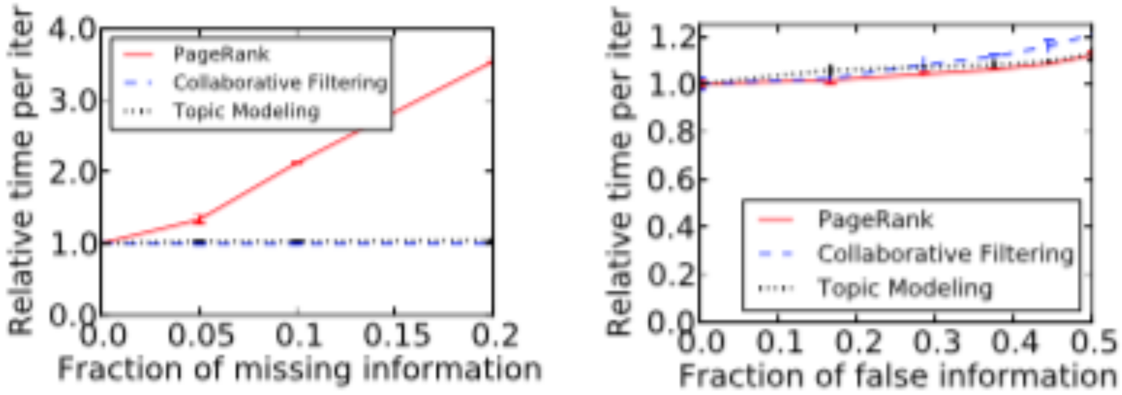


Figure 8. Pipelined prefetching.

Inaccurate information



(a) Missing Information. (b) False Information.

Figure 9. Influence of inaccurate information.

Comparison w/ single thread baselines

App.	Single-threaded	IterStore (512 threads)	Speedup
CF	374.7 sec	6.02 sec	62x
TM	1105 sec	11.2 sec	99x
PR	41 sec	5.13 sec	8x

Outline

1. Introduction
2. Iterative ML and systems
3. Exploiting iterative-ness for performance
4. Implementation
5. Evaluation
6. Conclusion

Conclusion

- Many ML applications make the same pattern of read and update accesses each iteration.
- The pattern can be exploited in parallel ML computations.
- Parameter server can specialize
  - Data structures
  - Data placement
  - Caching
  - Prefetching policies
- Experiments show the exploitation of iterative-ness reduce per-iteration execution times by 33-98%