Large-scale Graph Analysis in Data-Centric Models like MapReduce

Ananth Grama

Center for Science of Information and
Dept. of Computer Science,
Purdue University

Outline

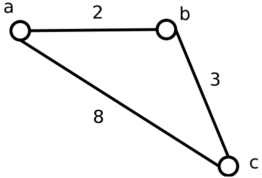
- Motivating Application(s)
  - Analyses of web-scale graph-structured datasets
  - Graph algorithms and (series of) matrix-vector products
  - *mat-vec* implementations in MapReduce
- Performance Considerations
  - Asynchronous algorithms through Relaxed Synchronization
  - Speculative parallelism through `TransMR` (transactional *MapReduce*)
  - Locking techniques for efficient distribution transactions
- Future Work

Large-scale graph analysis

- Dataset characteristics
  - Massive distributed graph-structured datasets
  - Graphs with billions of nodes, running into petabytes of storage (e.g., web graphs and social networks)
- Application characteristics
  - Most graph algorithms can be modeled as a series of matrix-vector products (*mat-vecs*)
    e.g., PageRank, Shortest-path problems, etc.
  - Each *mat-vec* requires distributed execution
  - Algorithmic efficiency achieved through **asynchrony** and **amorphous data-parallelism**
- **MapReduce** for scalable, distributed execution of each *mat-vec*

Single Source Shortest Path problem

- Input: Adjacency matrix – A; and the source vertex – u.
- Let x be the distance (from u) vector.



$$A = \begin{pmatrix} \infty & 2 & 8 \\ 2 & \infty & 3 \\ 8 & 3 & \infty \end{pmatrix} \qquad x = \begin{pmatrix} 0 \\ \infty \\ \infty \end{pmatrix}$$

Single Source Shortest Path can be computed as:

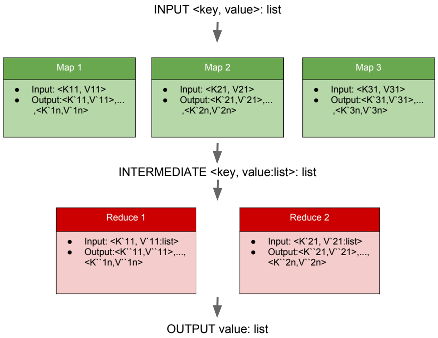
- Iterations of *mat-vecs* until the resultant vector converges
- In each *mat-vec*, each element a_{ij} is computed as:

$$a_{ij} = min_{\forall j}(a_{ij} + x_j)$$

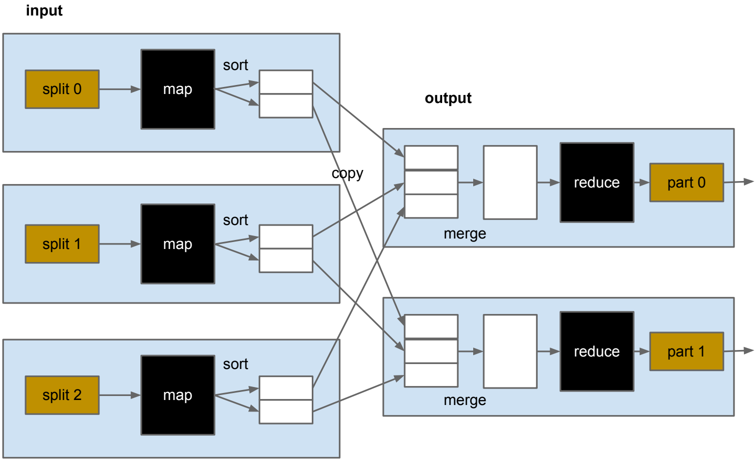
MapReduce

Executes user-defined functions on individual data-items distributed across several machines in parallel.

- Simple programming model — *map* and *reduce* functions
- Scalable, distributed execution of these functions on massive amounts of data on commodity hardware



MapReduce: DataFlow



Shotest Path (Iterative *mat-vec*) in MapReduce

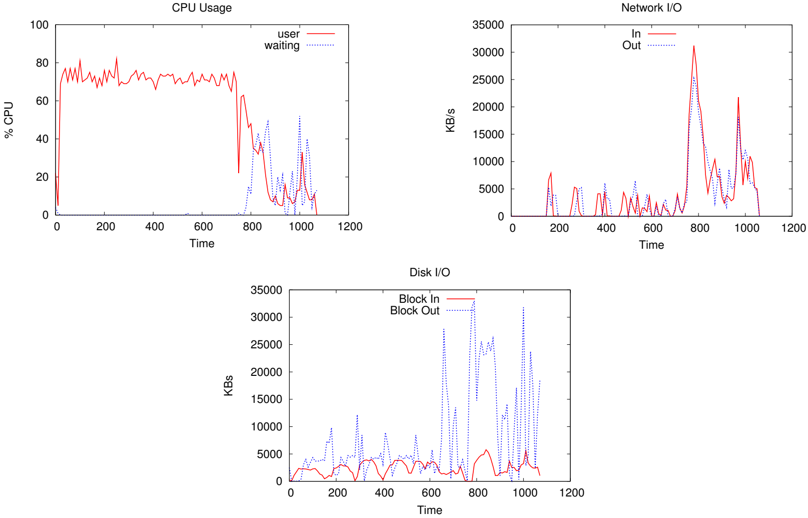
```
map(node, adjList) {
  for each arc in adjList {
    output(arc.dst, node.distance + arc.weight)
  }
}
```

```
reduce(node, newDistList) {
  node.distance = min(newDistList)
}
```

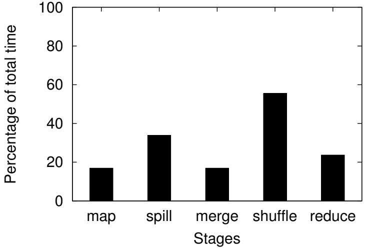
```
while(not converged) {
  runMapReduceJob(map, reduce, Ax)
}
```

- We call this the **Naive mat-vec** — *map* takes an adjacency list as input
- Pegasus (CMU) implementation reads one edge per *map*

Naive *mat-vec*: Resource Utilizaton



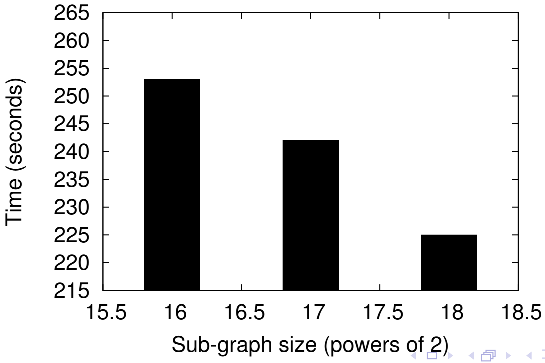
Optimizing the *mat-vec*



- Stages overlap; hence, sum of percentages $>$ 100
- I/O time $>$ Computation time
- For performance:
  - Batch read data
  - Each *map* processes more data (as much as can fit in memory)

Partitioned *mat-vec*

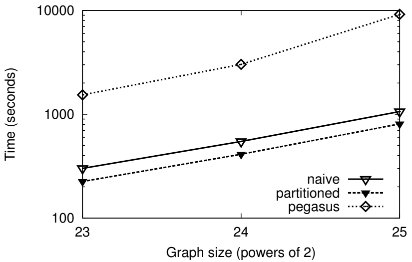
- Each *map* operates on a *graph partition* — a bunch of adjacency lists
- Partition size is constrained by the heap size
- On our setup, maximum partition size was 2^{19} nodes



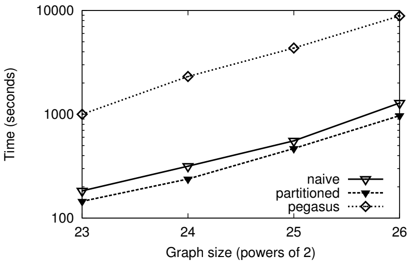
Pegasus vs Naive vs Partitioned – 1

- Partition-size $= 2^{18}$

4 Amazon EC2 nodes

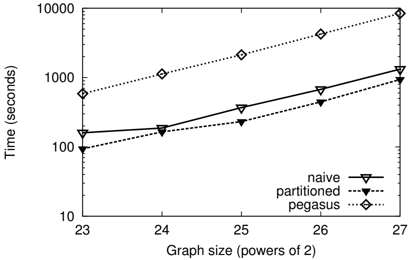


8 Amazon EC2 nodes

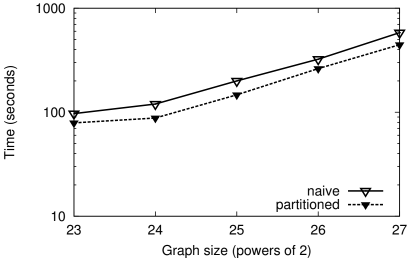


Pegasus vs Naive vs Partitioned – 2

16 Amazon EC2 nodes



32 Amazon EC2 nodes



Optimizations across iterations

Algorithmic optimizations:

- **Asynchronous algorithms**: Algorithms that allow asynchrony — ordering of updates doesn't affect the correctness of the algorithm
  e.g., Single Source Shortest Path, PageRank, Network Alignment
- **Amorphous data-parallelism**: Algorithms where concurrent computations can have potential conflicts, the conflicts are rare and can only be detected at runtime
  e.g., Boruvka's MST, Single Source Shortest Path

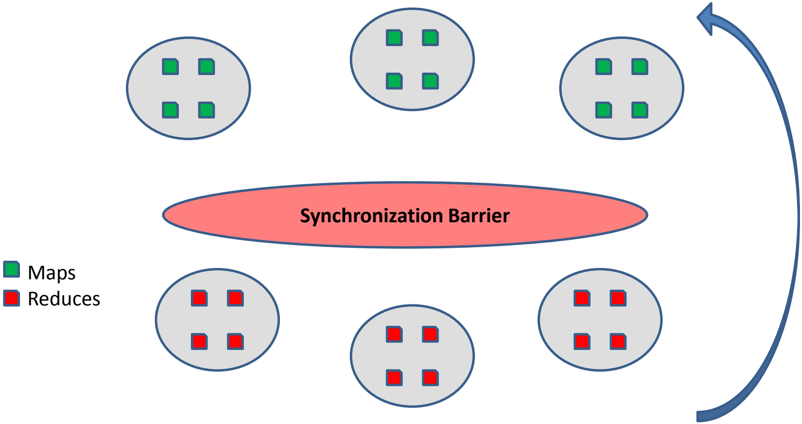
Asynchronous (Iterative) Algorithms

- Improve performance in parallel environments.
  - Infrequent synchronization reduces communication
  - Examples
    - Graph algorithms, Numerical methods, Classification, etc.
- More pronounced gains in distributed environments
  - Higher communication and data-movement costs
  - Read once, process multiple times allows more computation for the same I/O cost

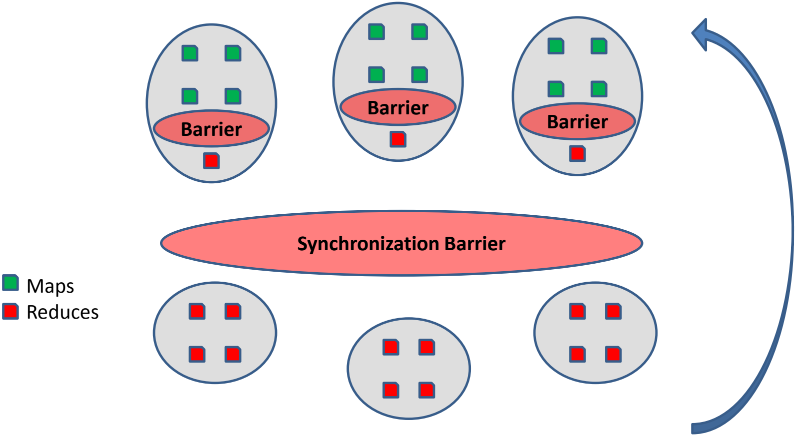
Proposal: Relaxed Synchronization

- Goal: Synchronize once every few iterations
- Approach: Two levels of MapReduce
  - Global MapReduce: The regular MapReduce
    - Requires global synchronization
  - Local MapReduce: MapReduce within a global map
    - Each global map runs a few iterations of local MapReduce
    - Partial synchronization of data of a single global map task
  - Input data partitioning for fewer dependencies across partitions

Shortest Path using traditional MapReduce



Shortest Path with Relaxed Synchronization



Relaxed Synchronization: PL Semantics

$$l, \sigma \implies \sigma(l) \quad \text{(LOCAL-LOOKUP)} \qquad l, \lambda \implies \lambda(l) \quad \text{(GLOBAL-LOOKUP)}$$

$$Apply(\mathbf{I}, < e, f_m, f_r, l >) \implies_g \mathbf{I}\; e\; f_m\; f_r\; l \qquad \text{(APPLY-ITER)}$$

$$\frac{\mathbf{while}(cond_g)\; \mathbf{G}\; cond_l\; f_m\; f_r\; l_g, \lambda \implies_g l_g{}', \lambda'}{\mathbf{I}\; cond_g\; f_m\; f_r\; l_g, \lambda \implies_g l_g{}', \lambda'} \qquad \text{(ITER-MAPRED)}$$

$$\frac{\begin{array}{c} ch\; l_g, \lambda, \sigma \equiv \bar{l}_l, \lambda, \sigma' \\ \mathbf{while} cond\; \mathbf{map}\; (\mathbf{L}\; f_m\; f_r)\; \bar{l}_l, \sigma \implies_l \bar{l}_l{}', \sigma' \; agg\; \bar{l}_l{}', \sigma, \lambda \equiv l_g{}', \sigma, \lambda' \; \mathbf{fold}\; f_r\; l_g{}', \lambda \implies_g l_g{}'', \lambda' \end{array}}{\mathbf{G}\; cond\; f_m\; f_r\; l_g, \lambda, \sigma \implies_g l_g{}'', \lambda', \sigma'} \qquad \text{(MAPRED-GLOBAL)}$$

$$\frac{\mathbf{map}\; f_m\; l_l, \sigma \implies_l l_l{}'', \sigma'' \qquad \mathbf{fold}\; f_r\; l_l{}'', \sigma \implies_l l_l{}', \sigma'}{\mathbf{L}\; f_m\; f_r\; l_l, \sigma \implies_l l_l{}', \sigma'} \qquad \text{(MAPRED-LOCAL)}$$

$$ch\; l_g \equiv \bar{l}_l \equiv \{l_l \mid l_l \subset l_g \;\&\; \cap_{l_l,\, l_l \epsilon \bar{l}_l}\, l_l = \phi\}; \forall l_i, \sigma_i[l_i \mapsto \lambda(l)] \qquad \text{(CHUNKIFY)}$$

$$agg\; \bar{l}_l \equiv l_g \equiv \cup_{l_l \epsilon \bar{l}_l}\, l_l; \forall l_i, \lambda[l \mapsto l_i] \qquad \text{(AGGREGATE)}$$

Realizing the semantics

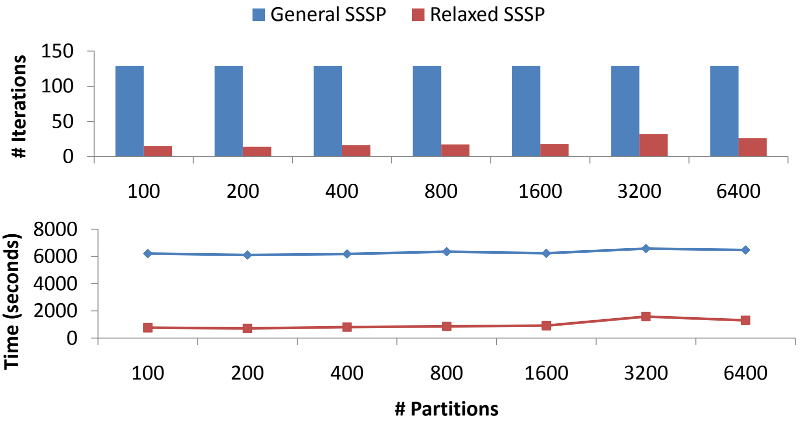
- Code *gmap, greduce, lmap, lreduce*
  - lmap, lreduce use EmitLocalIntermediate() and Emit Local()
  - Synchronized hashtables for local storage

```
gmap(xs : X list) {

  while(no-local-convergence-intimated) {
    for each element x in xs {
      lmap(x); // emits lkey, lval
    }

    lreduce(); // operates on the output of lmap functions
  }

  for each value in lreduce-output{
    EmitIntermediate(key, value);
  }
}
```

Evaluation — Relaxed Synchronization

- 3 applications
  - Single Source Shortest Path (MST, transitive closure, etc.)
  - PageRank (mat-vec: eigen value and linear system solvers)
- Test bed
  - 8 Amazon EC2 Large Instances
    - 64-bit compute units with 15 GB RAM, 4x 420 GB storage
    - Hadoop 0.20.1; 4 GB heap space per slave

Single Source Shortest Path

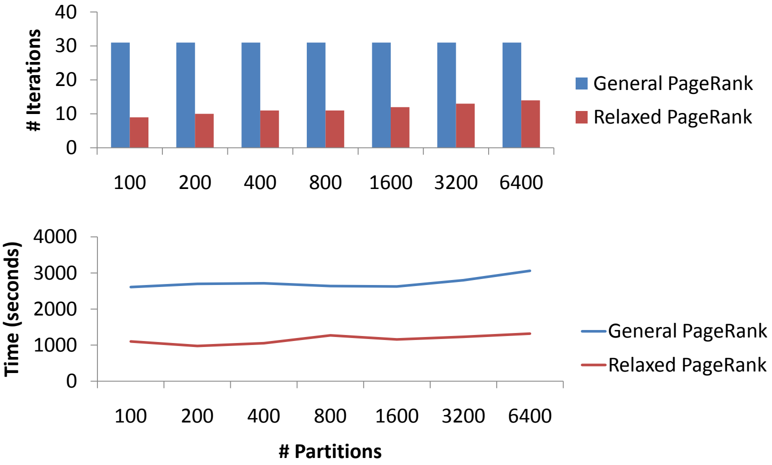


PageRank

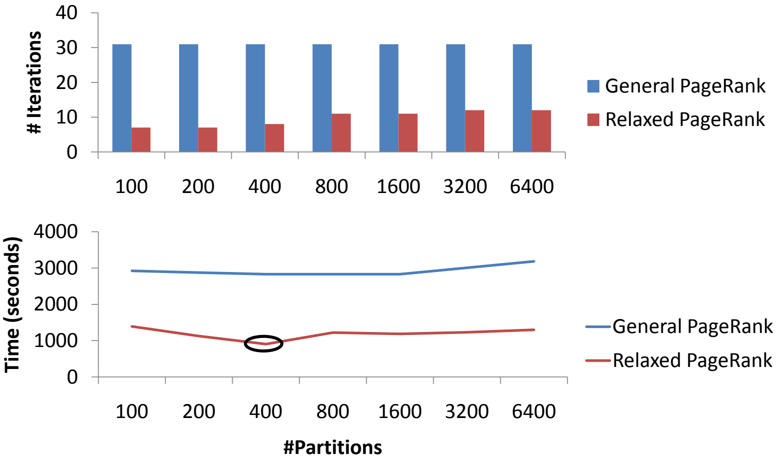
- Input: Partitioned using METIS (less than 10 seconds)
- Damping factor = 0.85

	GraphA	**GraphB**
Nodes	280,000	100,000
Edges	3 million	3 million

PageRank Performance: GraphA



PageRank Performance: GraphB



Beyond Data-Parallelism

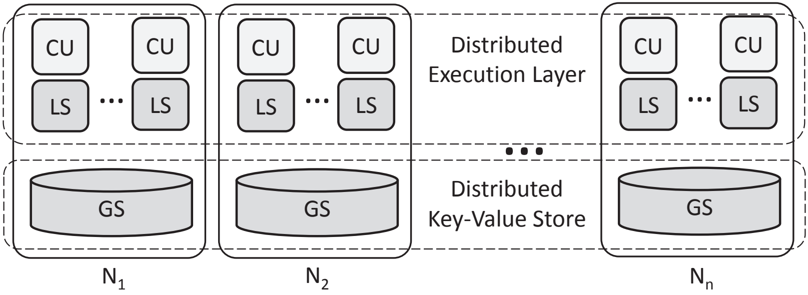
- Amorphous data-parallelism
    - Most of the data can be operated on in parallel.
    - Some of them conflict, which can only be detected at runtime.
        - "The Tao of Parallelism", Pingali et.al., PLDI'11
        - The Galois System
- Online algorithms/ Pipelined workflows
    - MapReduce Online [Condie'10] is an approach needing heavy checkpointing.
- Software Transactional Memory (STM)

TransMR: Transactional MapReduce

- Goal: Exploit amorphous data-parallelism
- Support data-sharing across concurrent computations to detect and resolve conflicts at runtime
- Solution:
  - Use distributed key-value stores as shared address space across computations
  - Address inconsistencies arising due to the disparate fault-tolerance mechanisms
  - Transactional execution of *map* and *reduce* functions

TransMR: System Architecture

- Distributed key-value store provides a shared-memory abstraction to the distributed execution-layer.



Semantics of the API

- Data-centric function scope – Map/Reduce/Merge etc, – termed as a Computation Unit (CU), is executed as a transaction.
- Optimistic reads and write-buffering. Local Store (LS) forms the write-buffer of a CU.
  - Put (K, V): Write to LS which is later atomically committed to GS.
  - Get (K, V): Return from LS, if already present; otherwise, fetch from GS and store in LS.
  - Other Op: Any thread local operation.
- The output of a CU is always committed to the GS before being visible to other CUs of the same or different type.
  - Eliminates the costly shuffle phase of MapReduce.

Design Principles

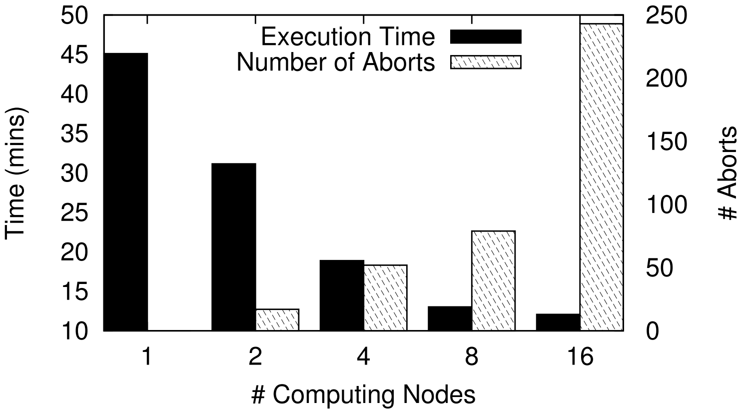
- Optimistic concurrency control over pessimistic locking
  - Locks are acquired at the end of the transaction. Write-buffer and read-set is validated against those of concurrent Trx assuring serializability.
  - Client is potentially executing on the slowest node in the system; in this case, pessimistic locking hinders parallel transaction execution.
- Consistency (C) and Tolerance to Network Partitions (P) over Availability (A) in CAP Theorem for Distributed transactions.
  - Application correctness mandates strict consistency of execution. Relaxed consistency models are application-specific optimizations.
  - Intermittent non-availability is not too costly for batch-processing applications, where client is fault-prone in itself.

Evaluation

- We show performance gains on two applications, which are hitherto implemented sequentially without transactional support; both exhibit Optimistic data-parallelism.
- Boruka's MST
  - Each iteration is coded as a Map function with input as a node. Reduce is an identity function. Conflicting maps are serialized while others are executed in parallel.
  - After n iterations of coalescing, we get the MST of an n node graph.
  - A graph of 100 thousand nodes, with average degree of 50, generated based on the forest-fire model.

Boruvka's MST

- Speedup of 3.73 on 16 nodes, with less than 0.5 % re-executions due to aborts.

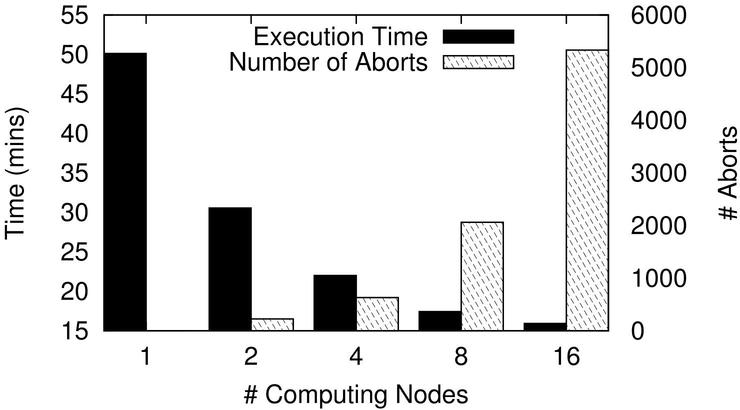


Maximum flow using Push-Relabel algorithm

- Each Map function executes a Push or a Relabel operation on the input node, depending on the constraints on its neighbors.
- Push operation increases the flow to a neighboring node and changes their "Excess".
- Relabel operation increases the height of the input node if it is the lowest among its neighbors.
- Conflicting Maps – operating on neighboring nodes – get serialized due to their transactional nature.
- Only sequential implementation possible without support for runtime conflict detection.

Maximum flow using Push-Relabel algorithm

- Speedup of 4.5 is observed on 16 nodes with 4% re-executions on a windown of 40 iterations.



TransMR: Conclusions

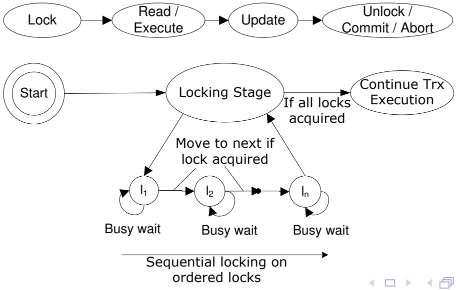
- TransMR programming model enables data-sharing in data-centric programming models for enhanced applicability.
- Similar to other data-centric programming models, the programmer only specifies operation on the individual data-element without concerning about its interaction with other operations.
- Prototype implementation shows that many important applications can be expressed in this model while extracting significant performance gains through increased parallelism.

Distributed Transactions on Key-Value Stores

- Transactions are costly in a large scale distributed settings
  - two-phase locking (concurrency control)
  - two-phase commit (atomicity)
- Careful examination of the protocols and optimizations crucial to performance of TransMR-like systems
- These optimizations also useful for general purpose transactions on databases using key-value store as the underlying storage

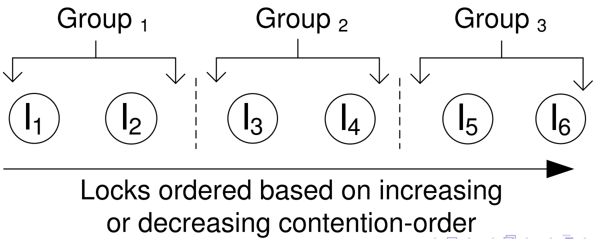
Lock management in distributed transactions

- Lock management the major bottleneck affecting the latency of distributed transactions.
- Consider Strong Strict two phase locking (SS2PL) – waiting case: The lock-acquiring stage is the only sequential stage. The other stages can be parallelized to finish in a single round-trip.
- Holds true even in optimistic-concurrency techniques where the locks are acquired at the end.



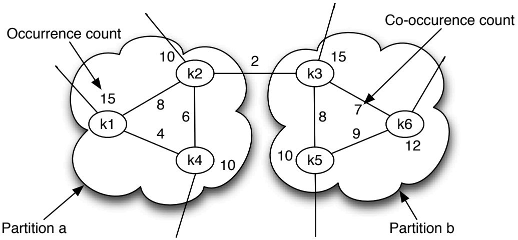
Workload aware lock management

- **Contention based Lock-ordering**: Order the locks so as to decrease the total amount of waiting time.
- For the waiting case, the lock with the least contention should be acquired first. This increases pipelining while decreasing lock-holding times.
- Contention order is a runtime characteristic, and is updated consistently. All clients should adhere to the same order to avoid deadlocks.



Constrained k-way graph partitioning

- Graph partitioning algorithm to split the locking into k non-overlapping partitions, minimizing the sum of weights on cut-edges, while approximately balancing the total weight (sum of node-weights) of individual partitions.
- The result of the partitioning algorithm is the load-balanced-partitioning of locks among k storage nodes.



Evaluation

- A cluster of 20 machines was used for all evaluations. Each machine had a Quad-core xeon processor with 8 GB of RAM. HBase is the underlying key-value store.
- The YCSB benchmark was extended with the atomic multi-put operation. A client transaction involves an atomic Read-Modify-Write operation on a set of keys.
- The keys for the atomic operation are generated using a Zipfian generator with variable zipfian parameter. Each transaction updates 15 keys out of 50K keys.

Two phase locking: waiting version

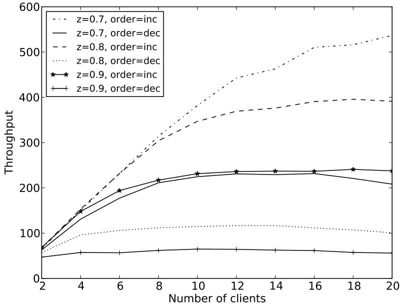


Figure: Performance of Lock Ordering

- Ordering of keys in increasing-order of their contention significantly better than the decreasing order.
- The increasing-order reduced lock holding time for highest contended locks reducing waiting time for other transactions.

Two phase locking: waiting version

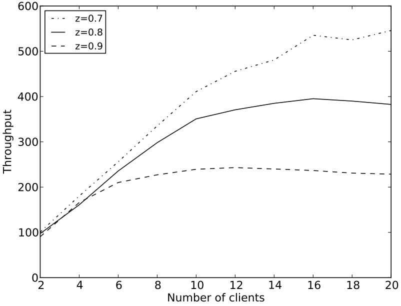


Figure: Performance of Lock Partitioning

- Partitioning is done using Metis and partitions are placed at separate nodes. Lock-partitioning improves the throughput by reducing the number of network-roundtrips needed for sequential locking by the client.

Optimistic concurrency control – no-waiting version

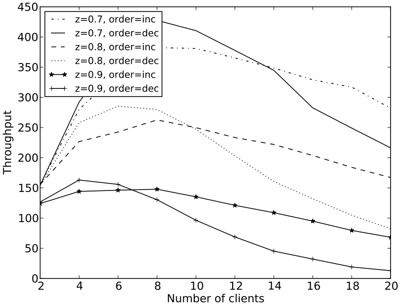


Figure: Performance of Lock Ordering

- Smaller improvement for OCC mainly due to the shorter duration of locking.
- At similar contention levels, the throughput of optimistic concurrency control is significantly lower.

Optimistic concurrency control – no-waiting version

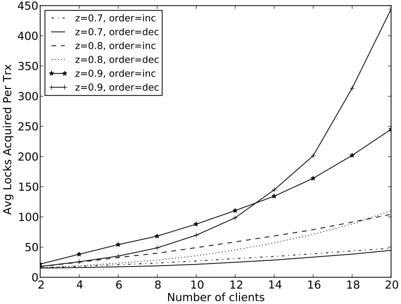


Figure: Lock wastage due to restarts

- Optimistic techniques not suitable at high contention levels as the time spent in reading and local updating gets wasted due to conflicts during commit.

Lock Optimization: Conclusions

- The waiting version of SS2PL with increasing-contention-order and partitioning outperforms the other protocols significantly.
- locks is an important step towards increasing performance.
- Understanding the workload - even simple statistics on contention - is enough to achieve significant gains (up to 200%).
- Lock-partitioning through graph clustering and partitioning techniques can be performed dynamically to achieve performance gains.