TransMR: Data-Centric Programming Beyond Data Parallelism

Ananth Grama

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

With help from Giorgos Kollias, Naresh Rapolu, Karthik Kambatla, and Adnan Hassan

Thanks to NSF, DoE, the Center for Science of Information, Microsoft, and Intel.

Outline

- Motivating applications – PageRank, Graph Alignment
- Case study: single *mat-vec* in MapReduce
- Asynchrony and speculation: Optimizations across iterations
  - Asynchronous algorithms through Relaxed Synchronization
  - Speculative parallelism through `TransMR` (transactional *MapReduce*)
  - Locking techniques for efficient distribution transactions
- Future Work

Motivating Example: Functional PageRank (PR)

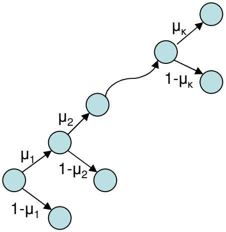
Computing PageRank (PR)

- PageRank as a *random surfer process*: Start surfing from a random node and keep following links with probability μ restarting with probability $1 - \mu$; the node for restarting will be selected based on a personalization vector v. The ranking value x_i of a node i is the probability of visiting this node during surfing.
- PR can also be cast in power series representation as $x = (1 - \mu) \sum_{j=0}^{k} \mu^j S^j v$; S encodes column-stochastic adjacencies.

Functional rankings

- A general method to assign ranking values to graph nodes as $x = \sum_{j=0}^{k} \zeta_j S^j v$. PR is a functional ranking, $\zeta_j = (1 - \mu)\mu^j$.
- Terms attenuated by outdegrees in S *and* damping coefficients ζ_j.

Functional Rankings Through Multidamping [Kollias, Gallopoulos, AG, TKDE'13]



Computing μ_j in multidamping

Simulate a functional ranking by random surfers following emanating links with probability μ_j at step j given by :

$\mu_j = 1 - \frac{1}{1+\frac{\rho_{k-j+1}}{1-\mu_{j-1}}}, j = 1, ..., k,$

where $\mu_0 = 0$ and $\rho_{k-j+1} = \frac{\zeta_{k-j+1}}{\zeta_{k-j}}$

Examples

LinearRank (LR) $x^{\mathrm{LR}} = \sum_{j=0}^{k} \frac{2(k+1-j)}{(k+1)(k+2)} S^j v \,:\; \mu_j = \frac{j}{j+2}, j = 1, ..., k.$

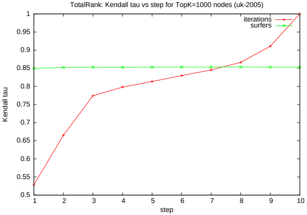
TotalRank (TR) $x^{\mathrm{TR}} = \sum_{j=0}^{\infty} \frac{1}{(j+1)(j+2)} S^j v \,:\; \mu_j = \frac{k-j+1}{k-j+2}, j = 1, ..., k.$

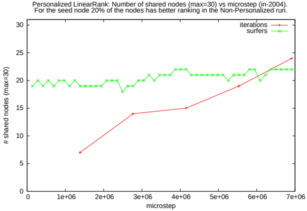
Multidamping and Computational Cost

Advantages of multidamping

- Reduced computational cost in *approximating* functional rankings using the Monte Carlo approach. A random surfer terminates with probability $1 - \mu_j$ at step j.
- Inherently parallel and synchronization free computation.

Multidamping Performance

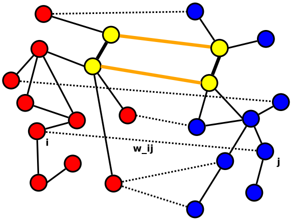




Approximate ranking: Run n surfers to completion for graph size n. How well does the computed ranking capture the "reference" ordering for `top-`k nodes, compared to standard iterations of equivalent computational cost/number of operations? *[Left]*

Approximate personalized ranking: Run $< n$ surfers to completion (each called a microstep, x-axis), but only from a selected node (personalized). How well can we capture the "reference" `top-`k nodes, i.e., how many of them are shared (y-axis), compared to the iterative approach of equivalent computational cost? *[Right]*

Motivating Example: Graph Matching



- **Node similarity:** Two nodes are similar if they are linked by other similar node pairs. By pairing similar nodes, the two graphs become *aligned*.

- Let $\tilde{A}$ and $\tilde{B}$ be the normalized adjacency matrices of the graphs (normalized by columns), H_{ij} be the independently known similarity scores (preferences matrix) of nodes $i \in V_B$ and $j \in V_A$, and μ be the fractional contribution of topological similarity.

- To compute X, IsoRank iterates:

$$X \leftarrow \mu \tilde{B} X \tilde{A}^T + (1 - \mu) H$$

Network Similarity Decomposition (NSD) [Kollias, Mohammadi, AG, TKDE'12]

Network Similarity Decomposition (NSD)

- In n steps of we reach $X^{(n)} = (1-\mu)\sum_{k=0}^{n-1} \mu^k \tilde{B}^k H (\tilde{A}^T)^k + \mu^n \tilde{B}^n H (\tilde{A}^T)^n$
- Assume that $H = uv^T$ (1 component). Two phases for X:
  1. $u^{(k)} = \tilde{B}^k u$ and $v^{(k)} = \tilde{A}^k v$ *(preprocess/compute iterates)*
  2. $X^{(n)} = (1-\mu)\sum_{k=0}^{n-1} \mu^k u^{(k)} {v^{(k)}}^T + \mu^n u^{(n)} {v^{(n)}}^T$ *(construct X)*

  This idea extends to s components, $H \sim \sum_{i=1}^{s} w_i z_i^T$.
- NSD computes matrix-vector iterates and builds X as a sum of outer products; these are much cheaper than triple matrix products.

We can then apply Primal-Dual or Greedy Matching (1/2 approximation) to extract the actual node pairs.

NSD: Performance [Kollias, Madan, Mohammadi, AG, BMC RN'12]

Species	Nodes	Edges
celeg (worm)	2805	4572
dmela (fly)	7518	25830
ecoli (bacterium)	1821	6849
hpylo (bacterium)	706	1414
hsapi (human)	9633	36386
mmusc (mouse)	290	254
scere (yeast)	5499	31898

Species pair	NSD (secs)	PDM (secs)	GM (secs)	IsoRank (secs)
celeg-dmela	**3.15**	152.12	7.29	783.48
celeg-hsapi	**3.28**	163.05	9.54	1209.28
celeg-scere	**1.97**	127.70	4.16	949.58
dmela-ecoli	**1.86**	86.80	4.78	807.93
dmela-hsapi	**8.61**	590.16	28.10	7840.00
dmela-scere	**4.79**	182.91	12.97	4905.00
ecoli-hsapi	**2.41**	79.23	4.76	2029.56
ecoli-scere	**1.49**	69.88	2.60	1264.24
hsapi-scere	**6.09**	181.17	15.56	6714.00

- We compute similarity matrices X for various pairs of species using Protein-Protein Interaction (PPI) networks. $\mu = 0.80$, uniform initial conditions (outer product of suitably normalized 1's for each pair), 20 iterations, one component.
- We then extract node matches using PDM and GM.
- *Three orders of magnitude speedup* from NSD-based approaches compared to IsoRank.

NSD: Parallelization [KKG JPDC'13, Submitted, KMSAG ParCo'13 Submitted]

Parallelization: NSD has been ported to parallel and distributed platforms.

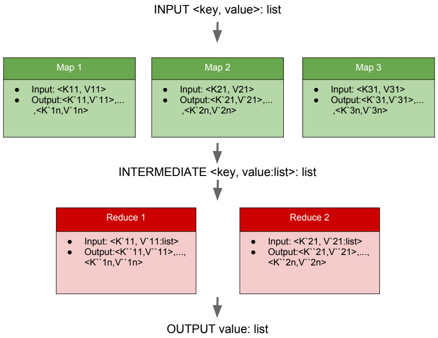
- We have aligned up to million-node graph instances using over 3K cores.
- We process graph pairs of over a billion nodes and twenty billion edges each (!), on MapReduce-based distributed platforms.

More on this in the rest of the talk.

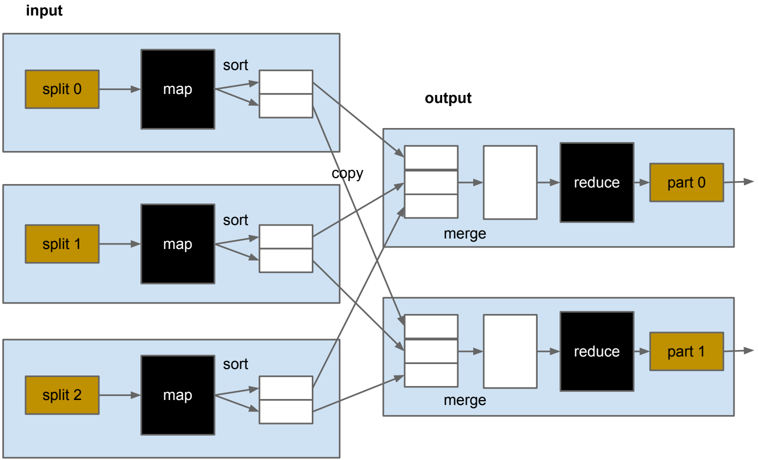
MapReduce: Basics

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

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



MapReduce: DataFlow



Example: Shotest Path (*mat-vec*) in MapReduce

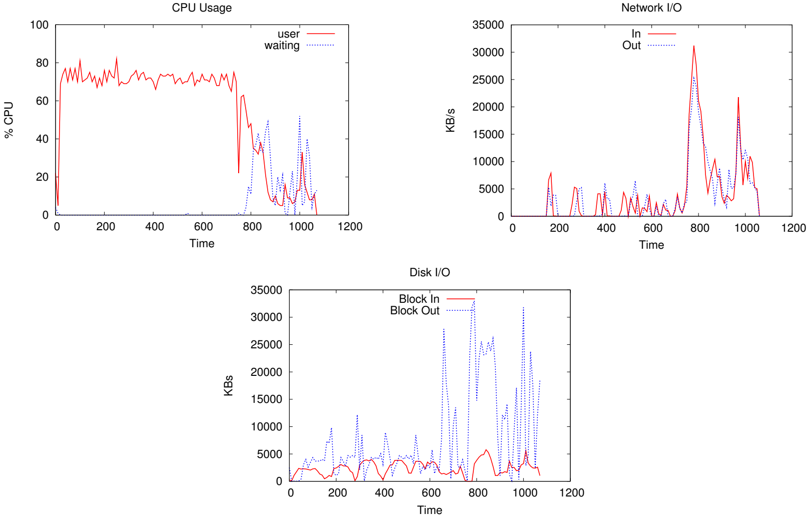
```
map(node, adjList) {
  for each arc in adjList {
    output(arc.dst, node.distance + arc.weight)
  }
}
```

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

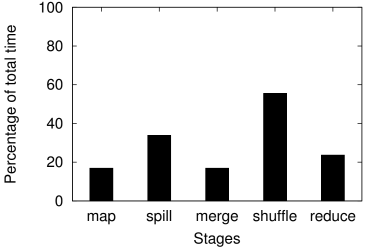
```
while(not converged) {
  runMapReduceJob(map, reduce, Ax)
}
```

- 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



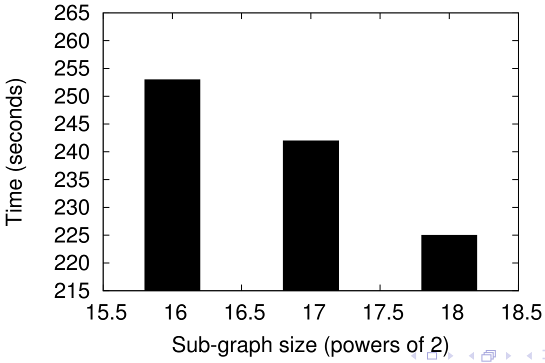
Optimizing the *mat-vec* further



- 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)

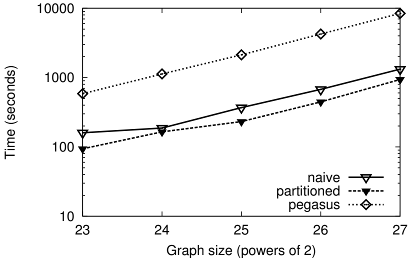
Partitioned *mat-vec*

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

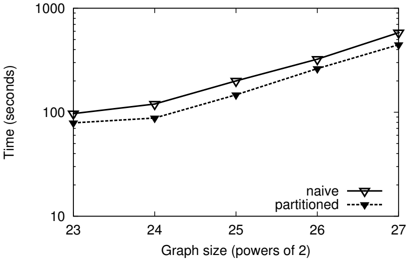


Performance: Pegasus vs Naive vs Partitioned

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., PageRank, Alignments
- **Speculative 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

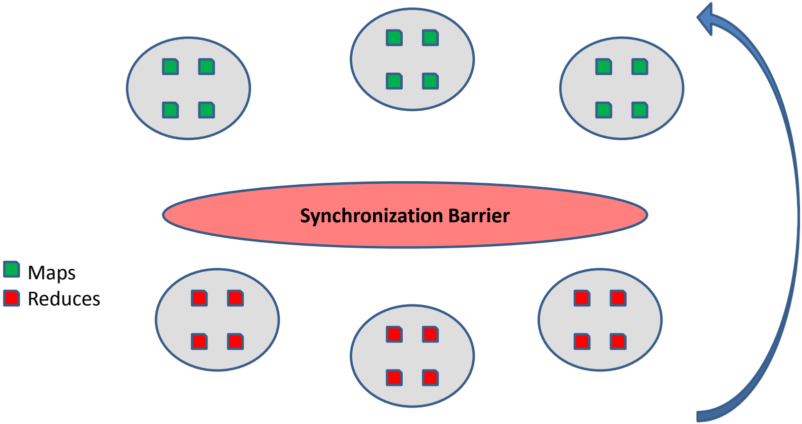
Asynchronous Iterations

- Improve performance in parallel environments.
  - Infrequent synchronization reduces communication
  - Examples
    - Graph algorithms, Numerical methods, ML kernels, 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

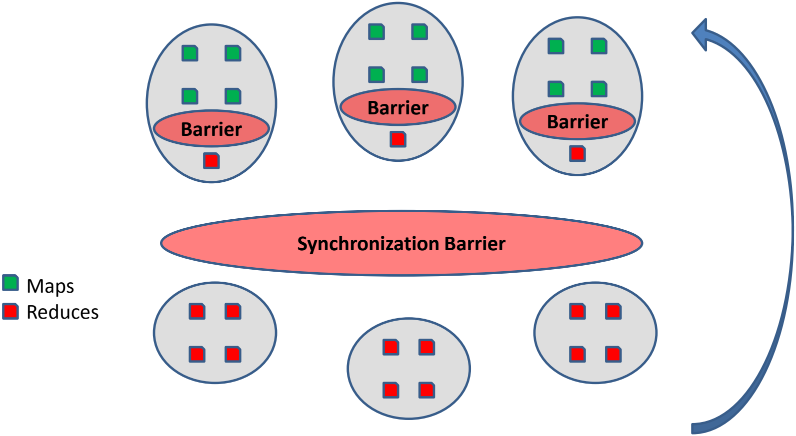
Relaxed Synchronization

- 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

PageRanks Using Traditional MapReduce



PageRank with Relaxed Synchronization



Realizing Relaxed Synchronization Semantics

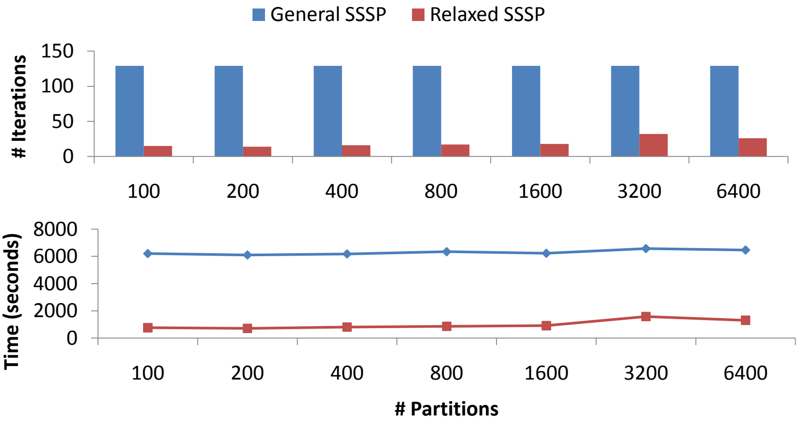
- Code *gmap, greduce, lmap, lreduce*
  - lmap, lreduce use EmitLocalIntermediate() and Emit Local()
  - Synchronized hashtables for local storage

```
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

- Sample applications
  - Single Source Shortest Path (MST, transitive closure, etc.)
  - PageRank (mat-vec: eigen value and linear system solvers)
- Experimental Testbed
  - 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

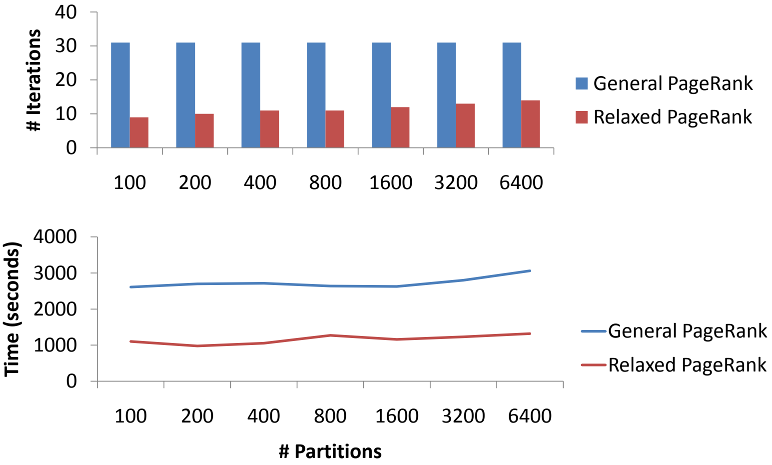


PageRank

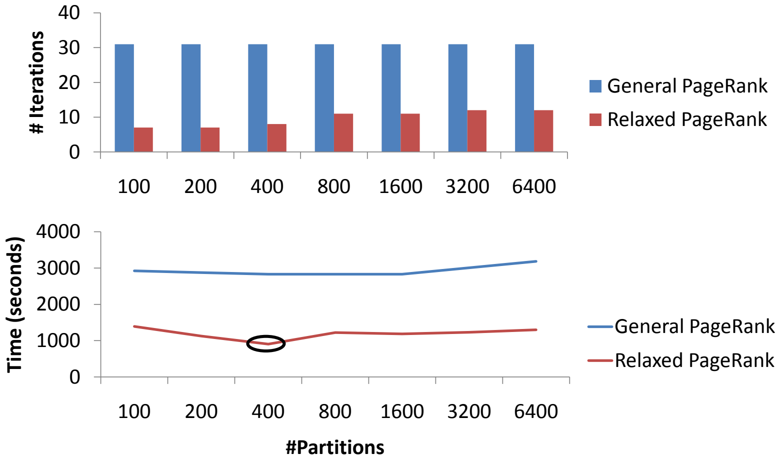
- Input: Partitioned using METIS (less than 10 seconds)
- Damping factor = 0.85

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

PageRank Performance: GraphA



PageRank Performance: GraphB



Beyond Data-Parallelism: Speculation

Speculative parallelism

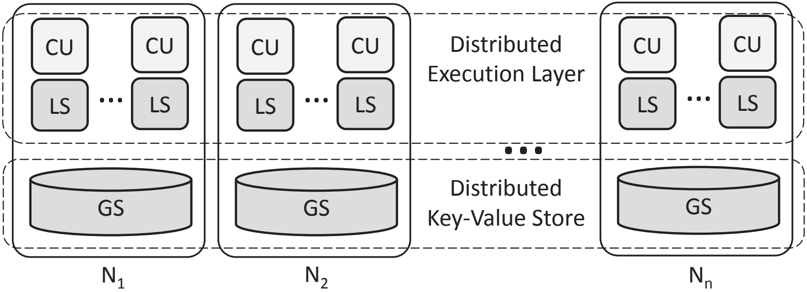
- Most of the data can be operated on in parallel.
- Some executions conflict. These can only be detected at runtime. [Pingali et.al., PLDI'11]
- Online algorithms/ Pipelined workflows
  - MapReduce Online [Condie'10] is an approach needing havy checkpointing.
- Software Transactional Memory (STM)

TransMR: Transactional MapReduce

- Goal: Exploit speculative 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

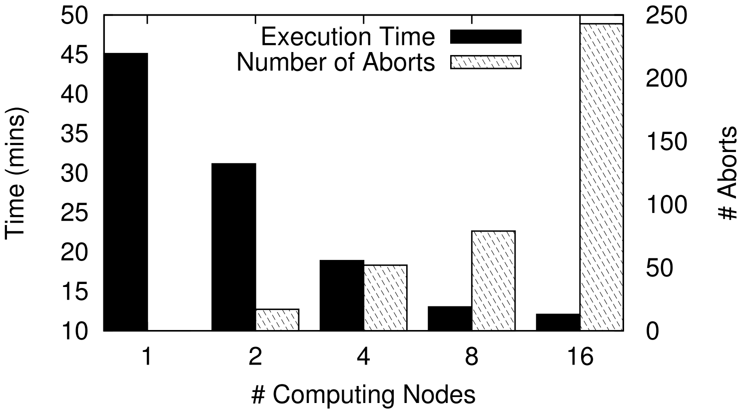
- Optimistic concurrency control over pessimistic locking
  - Locks are acquired at the end of the transaction. Write-buffer and read-set is validated against those of concurrent Trx assuring serializability.
  - Client is potentially executing on the slowest node in the system; in this case, pessimistic 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.

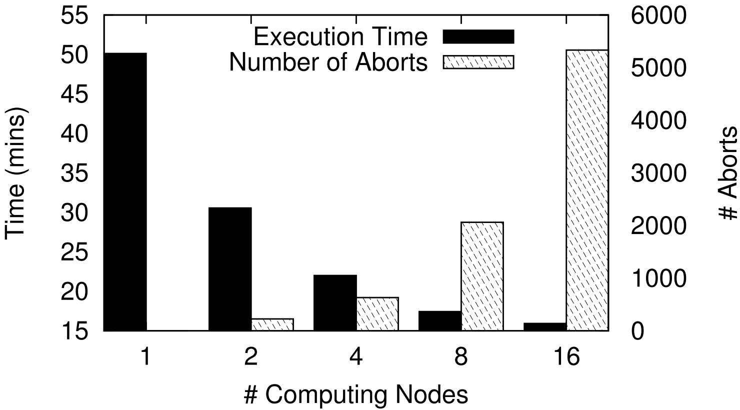


Maximum Flow Using Push-Relabel Algorithm

- 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: Intermediate Lessons

- 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.

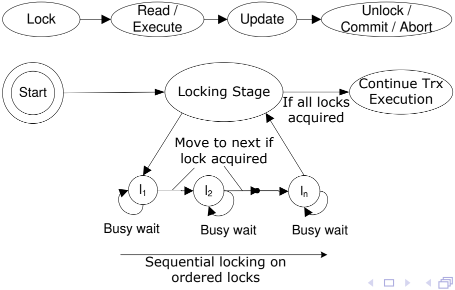
BUT: What about the locking operations!

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

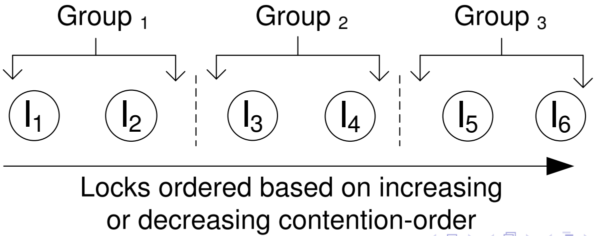
Lock Management in Distributed Transactions

- 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.



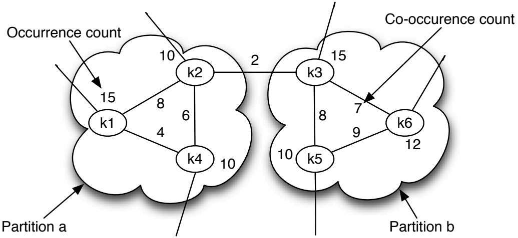
Workload Aware Lock Management

- **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

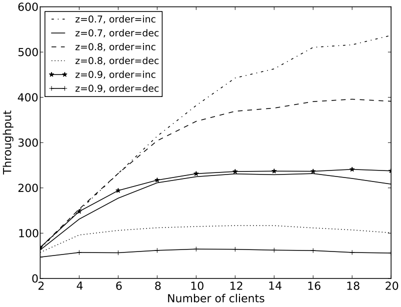


Figure: Performance of Lock Partitioning

- 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

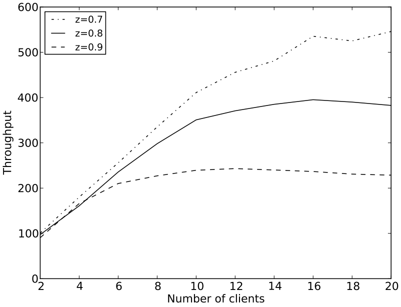


Figure: Performance of Lock Partitioning

- 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

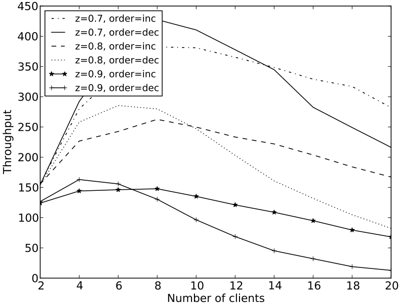


Figure: Performance of Lock Ordering

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

Optimistic Concurrency Control – No-waiting Version

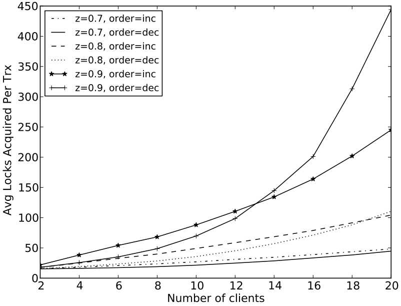


Figure: Lock wastage due to restarts

- 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.
- Restarts due to conflicts constitute a major overhead in distributed transactions. Reducing restarts by busy-waiting for 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.