TransMR: Data Centric Programming Beyond Data Parallelism

Naresh Rapolu
Karthik Kambatla
Prof. Suresh Jagannathan
Prof. Ananth Grama

Limitations of Data-Centric Programming Models

- Data-centric programming models (MapReduce, Dryad etc.) are limited to data-parallelism in any phase.
  - Two map operators cannot communicate with each other.
  - This is mainly due to the deterministic-replay based fault-tolerance model: Replay should not violate application semantics.
  - Consider presence of side-effects: Writing to persistent storage or network based communication.

INPUT: The quick brown fox jumps over a lazy dog.

Execution 1:

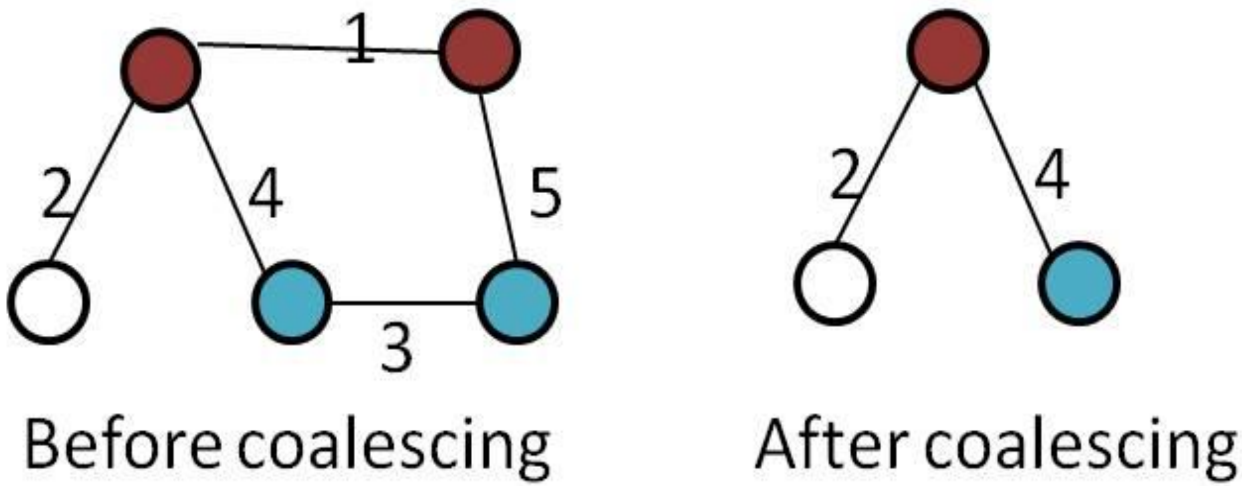
The	Quick	Brown	Fox
1	1	1	1

Execution 2:

The	Quick	Brown	Fox	Jumps	Over	A	Lazy	Dog
2	2	2	2	1	1	1	1	1

Need for side-effects

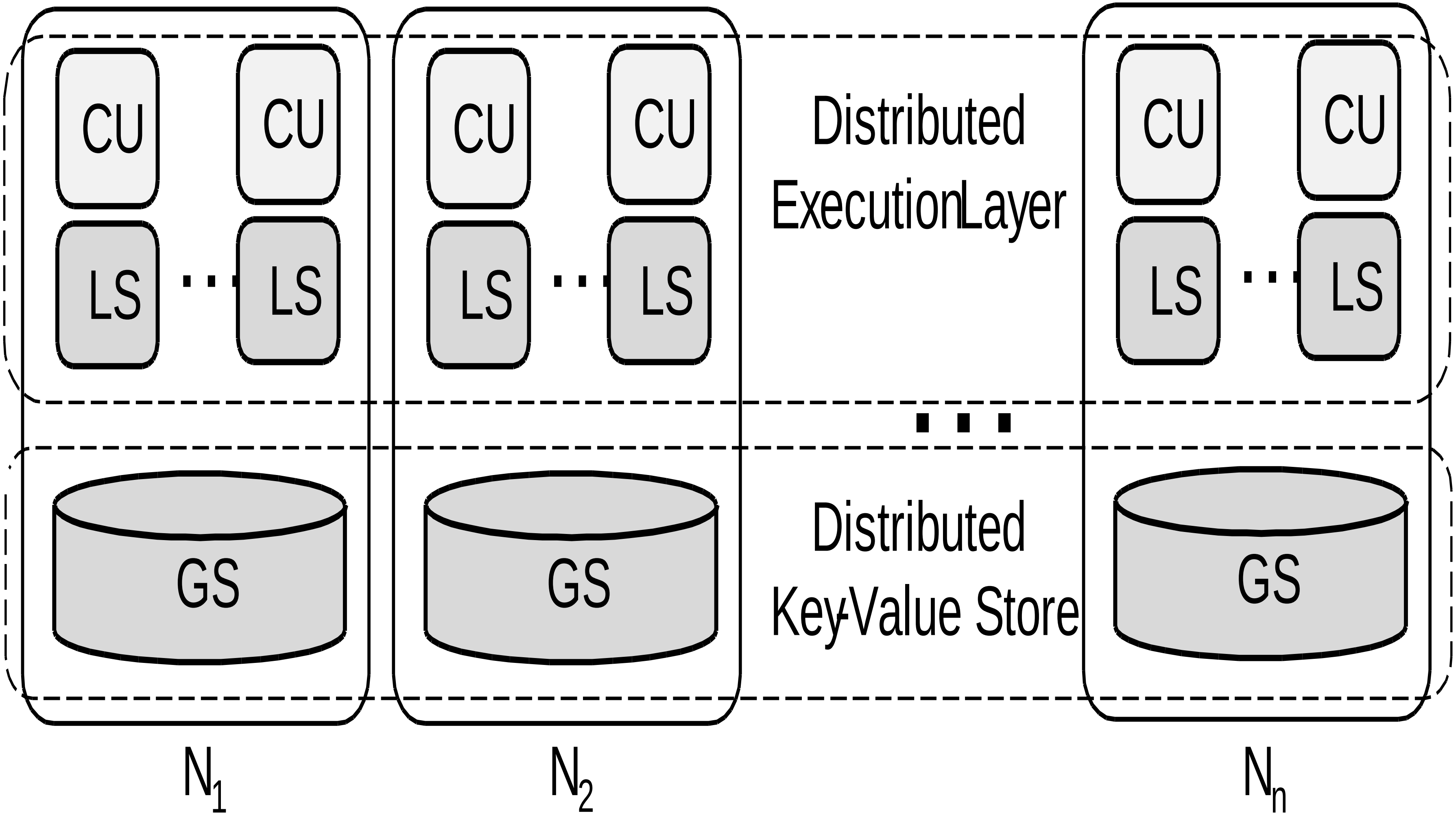
- Side-effects lead to communication/ data-sharing across computations.
- Boruvka's algorithm to find MST
  - Each iteration coalesces a node with its closes neighbor. Iterations which do not cause conflicts can be executed in parallel.



Beyond Data Parallelism

- Amorphous Data Parallelism
  - Most of the data can be operated on in parallel.
  - Some of them conflict and can only be detected dynamically 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) Benchmark applications
  - STAMP, STMBench etc.

System Architecture



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

Semantics of TransMR (Transactional MapReduce)

$$
\begin{aligned}
LocalStore &:= \{\Sigma_1, ..., \Sigma_m\} && (1)\\
GlobalStore &:= \{\Gamma\} && (2)\\
\sigma \in \Sigma &= L \rightarrow Z && (3)\\
\gamma \in \Gamma &= L \rightarrow Z && (4)\\
Fn &:= \{f_m, f_r\} && (5)\\
f \in Fn &:= Atomic\{Op^*\} && (6)\\
Op &:= Get\ k | Put\ (k, v) | Other && (7)\\
b \in Boolean &:= \{True, False\} && (8)\\
k, v \in Values &:= \{b, UnObservable\} && (9)\\
l &:= [v_1, ..., v_n] && (10)
\end{aligned}
$$

(a) Syntax

$$l, \sigma \implies \sigma(l) \qquad \text{(LOCAL)}$$

$$l, \gamma \implies \gamma(l) \qquad \text{(GLOBAL)}$$

$$\frac{\textbf{map}\ f_m\ \bar{l}, \gamma \implies \bar{l}'', \gamma'' \qquad \textbf{fold}\ f_r\ \bar{l}'', \gamma'' \implies \bar{l}', \gamma'}{\textbf{TMR}\ f_m\ f_r\ \bar{l}, \gamma \implies \bar{l}', \gamma'} \qquad \text{(TMR)}$$

$$\frac{\begin{array}{c}\textbf{if}\ (k \notin domain(\sigma))\ \textbf{then}\ \sigma' = \sigma[k \mapsto \gamma(k)] \\ \textbf{else}\ \sigma' = \sigma \\ k, \sigma' \implies v\end{array}}{Get\ k, \sigma, \gamma \implies v, \sigma', \gamma} \qquad \text{(GET)}$$

$$\frac{\sigma' = \sigma[k \mapsto v]}{Put\ (k, v), \sigma, \gamma \implies True, \sigma', \gamma} \qquad \text{(PUT)}$$

$$\frac{}{Other, \sigma, \gamma \implies UnObservable, \sigma, \gamma} \qquad \text{(OTHER)}$$

$$\frac{\begin{array}{c}Op_1, \sigma, \gamma \implies v_1, \sigma_1', \gamma \\ Op_2, \sigma_1, \gamma \implies v_2, \sigma_2', \gamma \\ \dots \\ Op_n, \sigma_{n-1}, \gamma \implies v_n, \sigma_n', \gamma \\ \forall k_i \in domain(\sigma) \qquad m = |\sigma|, \\ \gamma' = \gamma[k_1 \mapsto \sigma(k_1), ..., k_i \mapsto \sigma(k_i), ... k_m \mapsto \sigma(k_m)]\end{array}}{Atomic(\ Op_1,\ Op_2, ...,\ Op_n), \gamma \implies v_n, \gamma'} \qquad \text{(FN)}$$

(b) Semantics

Semantics Overview

- 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 CU's of the same or different type.
  - Eliminates the costly shuffle phase of MapReduce.

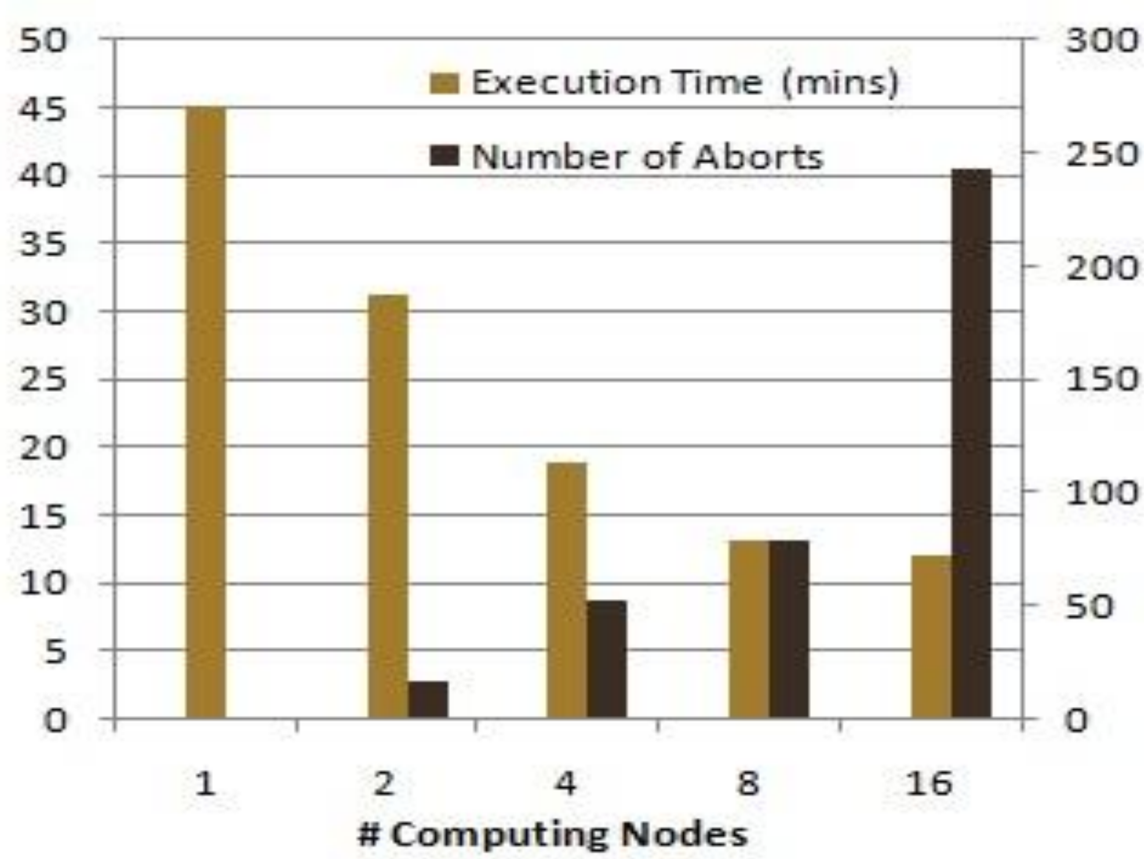
Design Principles

- Optimistic concurrency control over pessimistic locking.
  - No locks are acquired. 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
  - Presence of Data dependencies.
  - Both exhibit Optimistic data-parallelism.
- Boruvka’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.

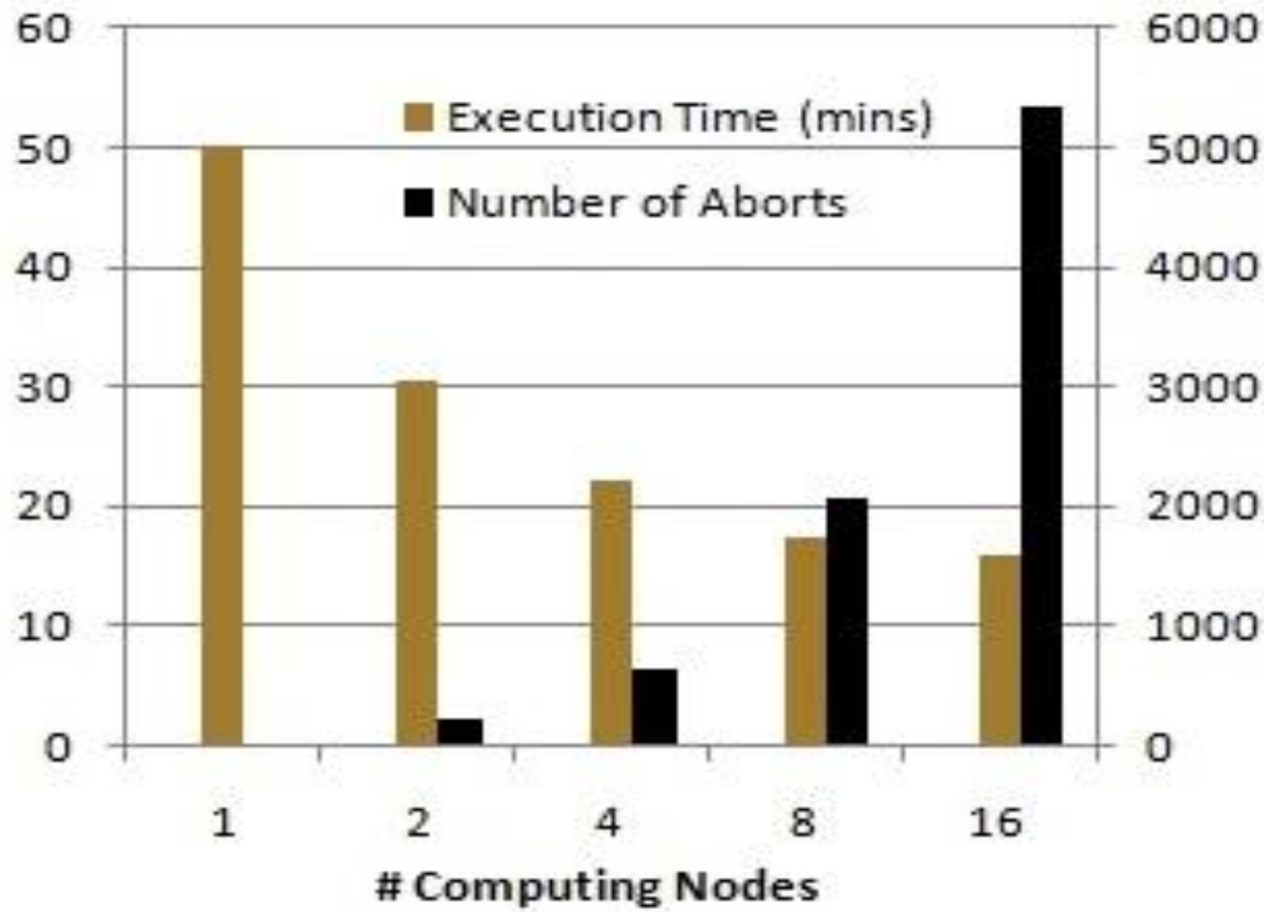
Boruvka's MST



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.



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

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.

Thank You!

Questions ?