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.
```

<table>
<thead>
<tr>
<th>Execution 1:</th>
<th>The</th>
<th>Quick</th>
<th>Brown</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution 2:</th>
<th>The</th>
<th>Quick</th>
<th>Brown</th>
<th>Fox</th>
<th>Jumps</th>
<th>Over</th>
<th>A</th>
<th>Lazy</th>
<th>Dog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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 closest 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)

(a) Syntax

LocalStore := \{\Sigma_1, ..., \Sigma_m\} \quad (1)

GlobalStore := \{\Gamma\} \quad (2)

\sigma \in \Sigma = L \rightarrow Z \quad (3)

\gamma \in \Gamma = L \rightarrow Z \quad (4)

F_n := \{f_m, f_r\} \quad (5)

f \in F_n := \text{Atomic}\{Op^*\} \quad (6)

Op := \text{Get } k | \text{Put } (k, v) | \text{Other} \quad (7)

b \in \text{Boolean} := \{\text{True}, \text{False}\} \quad (8)

k, v \in \text{Values} := \{b, \text{UnObservable}\} \quad (9)

l := [v_1, ..., v_n] \quad (10)

(b) Semantics

\frac{l, \sigma \rightarrow \sigma(l)}{(\text{LOCAL})}

\frac{l, \gamma \rightarrow \gamma(l)}{(\text{GLOBAL})}

\text{map } f_m \bar{l}, \gamma \rightarrow \bar{l}'', \gamma'' \quad \text{fold } f_r \bar{l}'', \gamma'' \rightarrow \bar{l}', \gamma' \quad (\text{TMR})

\text{if } (k \notin \text{domain}(\sigma)) \text{ then } \sigma' = \sigma[k \mapsto \gamma(k)]

\text{else } \sigma' = \sigma

k, \sigma' \rightarrow v \quad (\text{GET})

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

\frac{\text{Other}, \sigma, \gamma \rightarrow \text{UnObservable}, \sigma, \gamma} {(\text{OTHER})}

\frac{\text{Op}_1, \sigma, \gamma \rightarrow v_1, \sigma'_1, \gamma}{...}

\frac{\text{Op}_n, \sigma_{n-1}, \gamma \rightarrow v_n, \sigma'_n, \gamma}{\forall k_i \in \text{domain}(\sigma) \quad m = |\sigma|,}

\gamma' = \gamma[k_1 \mapsto \sigma(k_1), ..., k_i \mapsto \sigma(k_i), ... k_m \mapsto \sigma(k_m)]

\frac{\text{Atomic}(\text{Op}_1, \text{Op}_2, ..., \text{Op}_n), \gamma \rightarrow v_n, \gamma'}{(\text{FN})}
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?