Templating Shuffles

Qizhen Zhang¹,²,³, Jiacheng Wu⁴, Ang Chen⁵, Vincent Liu¹, Boon Thau Loo¹

¹University of Pennsylvania, ²Microsoft Research, ³University of Toronto,
⁴University of Washington, ⁵Rice University
General Structure of Large-scale Analytics

• **Compute**
  - Distributed workers process local shards of data independently

• **Combine (optional)**
  - Preliminary results are locally processed before data exchange

• **Shuffle**
  - Resharding and transmitting data for the next phase of processing
Shuffle as Critical Component

• Encompasses CPU, bandwidth, and latency overhead
  • Compression, serialization, message processing, transmission, etc.

• A rich history of tuning shuffle for efficient data analytics
  • DBMSs, MapReduce, and graph processing

• A primary bottleneck in emerging cloud platforms
  • Serverless and disaggregated memory/storage
Challenges in Optimizing Shuffle

• Dependent on **workloads**
  • Disk activities, combinable, aggregable, data redundancy...

• Dependent on **data center architecture**
  • CPU performance, network bandwidth, communication locality across machines...

• Must adapt to **changes**
  • Data center topology updates caused by network failures
  • Disaggregated compute, memory, and storage
  • Next-generation data center network designs are increasingly complicated
TeShu: A **Templated Shuffle Layer**

- **Adapts** to workload and data center infrastructure **changes**
- **Easily supports** existing data analytics systems and enables future ones
- **Users implement** shuffle primitives as **templates** with unknown characteristics of workloads and infrastructure as parameters
- **Data analytics systems** instantiate **templates** by populating parameters
Outline

• Motivation & TeShu Vision

• TeShu Design

• Expressiveness

• Evaluation

• Future Directions
Applications invoke shuffle API and instantiate templates to plans

Shuffle Manager stores and serves templates

A plan can sample messages to measure the efficiency of a particular shuffle strategy for adaptiveness
Shuffle Templates

- Python-like programs with parameters below

<table>
<thead>
<tr>
<th>Template parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SEND(dst, msg)</code></td>
<td>Send <code>msg</code> to <code>dst</code></td>
</tr>
<tr>
<td><code>RECV(src)</code></td>
<td>Return data received from <code>src</code></td>
</tr>
<tr>
<td><code>FETCH(src)</code></td>
<td>Return data fetched from <code>src</code></td>
</tr>
<tr>
<td><code>PART(msgs, srcs, partFunc)</code></td>
<td>Partition <code>msgs</code> into <code>dsts</code> according to <code>partFunc</code></td>
</tr>
<tr>
<td><code>COMB(msgs, combFunc)</code></td>
<td>Combine <code>msgs</code> according to <code>combFunc</code></td>
</tr>
<tr>
<td><code>SAMP(msgs, rate, partFunc)</code></td>
<td>Sample <code>msgs</code> based on <code>rate</code> and <code>partFunc</code></td>
</tr>
</tbody>
</table>

Basic communication (supporting both pull and push)
Populated by framework-native communication libraries

Partitioning, combing, and sampling
Populated by shuffle arguments and our sampling approach

- Example: vanilla shuffling (pull mode)

**Sender template:**
```
PART(bufs, dsts, partFunc)
```

**Receiver template:**
```
for s in srcs:
    bufs[n] = FETCH(n)
```
Shuffle API

• Shuffles are instances of concurrent communication between a fixed set of sources and destinations

  IDs of worker, template, and shuffle call
  Sources, destinations, and data buffers

• shuffle(wId, templateId, shuffleId, srcs, dsts, bufs, partFunc, combFunc)

  Functions for data partitioning and combining (optional)

• Partition function maps a piece of data to a destination worker
• Combine function merges two pieces of data into one
• These arguments are used to populate template parameters
Shuffle Management

- Involves Shuffle Manager and the application
  - System operators implement shuffle templates that are stored in Shuffle Manager
  - Application invokes shuffle API, instantiates templates into plans, and caches them
Outline

• Motivation & TeShu Vision

• TeShu Design

• Expressiveness

• Evaluation

• Future Directions
Implementing Existing Shuffle Algorithms

- TeShu can express vanilla shuffling and existing shuffle optimizations in a few lines of code

<table>
<thead>
<tr>
<th>Shuffle algorithm</th>
<th>Pattern</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vanilla shuffling</td>
<td>Push/pull</td>
<td>5</td>
</tr>
<tr>
<td>Coordinated shuffling [CIDR ‘13]</td>
<td>Pull</td>
<td>9</td>
</tr>
<tr>
<td>Bruck shuffling [IJHPCA ‘05]</td>
<td>Push</td>
<td>11</td>
</tr>
<tr>
<td>Two-level exchange [SIGMOD ‘20]</td>
<td>Push</td>
<td>18</td>
</tr>
</tbody>
</table>
Adaptive Shuffling

- Sampling-enabled data center network-aware shuffle optimization

A local shuffle at each level
- Pros: reduced traffic at next level by combining (benefit oversubscribed networks)
- Cons: additional shuffling and combining overhead
- Need to compare the benefit of traffic reduction and the overhead
  Measure through data sampling
Data Sampling

• Selects a subset of shuffle data from each worker based on sampling rate

• **Baseline: random sampling**, which is inaccurate with real data when sampling rate is kept low

• **Partition-aware sampling**: samples data according to the destinations to evaluate reduction rate when combiner is applied
  • Divides destination space into S buckets (S is calculated by sampling rate)
  • Allocates each piece of data into one of the buckets based on its destination
  • Selects Bucket j for sampling (j is randomly selected and consistent across workers)
Outline

• Motivation & TeShu Vision

• TeShu Design

• Expressiveness

• Evaluation

• Future Directions
Evaluation Setup

- Cluster: 2 racks of 10 servers, each with 16 cores @2.6GHz, 128 GB memory, and 10 Gbps NIC

- Network: oversubscription of inter-rack network varies (10:1, 4:1, 1:1)

- Software: Pregel+ for graph analytics

- Queries: PageRank and single source shortest path

- Datasets: UK-Web (3.7 billion edges) and Friendster (3.6 billion edges)
Sampling Performance

• Duplication estimation on shuffled data with typical workloads

<table>
<thead>
<tr>
<th>Sampling rate</th>
<th>Ground truth</th>
<th>Part.-aware sampling</th>
<th>Random sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>0.1833</td>
<td>0.1833</td>
<td>0.1986</td>
</tr>
<tr>
<td>0.1</td>
<td>0.1833</td>
<td>0.1833</td>
<td>0.7241</td>
</tr>
<tr>
<td>0.01</td>
<td>0.1833</td>
<td>0.1832</td>
<td>0.9622</td>
</tr>
<tr>
<td>0.001</td>
<td>0.1833</td>
<td>0.1829</td>
<td>0.9965</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.1833</td>
<td>0.1838</td>
<td>0.9997</td>
</tr>
</tbody>
</table>

• Sampling overhead (execution time)
Adaptive Shuffling Performance

- Compared to vanilla shuffling, adaptive shuffling speeds up queries from $3.9 \times$ to $14.7 \times$ by eliminating most of the communication cost.
- Across network oversubscription scenarios, adaptive shuffling always identifies the optimal shuffling strategy.

<table>
<thead>
<tr>
<th>Oversubscription Ratio</th>
<th>PR-UK</th>
<th>PR-FR</th>
<th>SSSP-UK</th>
<th>SSSP-FR</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:1</td>
<td>14.7×</td>
<td>9.4×</td>
<td>6.1×</td>
<td>7.1×</td>
</tr>
<tr>
<td>Execution Speedup</td>
<td>87.7%</td>
<td>89.8%</td>
<td>84.6%</td>
<td>84.3%</td>
</tr>
<tr>
<td>Shuffle Decision</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:1</td>
<td>9.4×</td>
<td>5.6×</td>
<td>5.2×</td>
<td>6.2×</td>
</tr>
<tr>
<td>Execution Speedup</td>
<td>85.5%</td>
<td>85.9%</td>
<td>81.6%</td>
<td>79.3%</td>
</tr>
<tr>
<td>Shuffle Decision</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:1</td>
<td>7.7×</td>
<td>3.9×</td>
<td>4.8×</td>
<td>4.8×</td>
</tr>
<tr>
<td>Execution Speedup</td>
<td>80.7%</td>
<td>76.4%</td>
<td>75%</td>
<td>66.8%</td>
</tr>
<tr>
<td>Communication Saving</td>
<td>$S,G$</td>
<td>$S,G$</td>
<td>$S,G$</td>
<td>$S,G$</td>
</tr>
<tr>
<td>Shuffle Decision</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

• Motivation & TeShu Vision

• TeShu Design

• Expressiveness

• Evaluation

• Future Directions
Future Directions

• **Co-scheduling shuffles** to achieve shorter flow completion times and thus improve application performance

• **Handling failures and stragglers** with shuffle records

• **Integrating with in-network techniques** to apply combining and sampling in the network to further improve performance

• **Templating shuffles for future data centers**, e.g., data movement between disaggregated resource components
Summary

• Tuning shuffle for large-scale data analytics is necessary but challenging and requires adaptiveness and portability

• TeShu provides a simple and expressive shuffle abstraction and offers it as a general layer to benefit various data analytics systems

• Shuffle templates and efficient sampling in TeShu enable portable and adaptive shuffle optimizations

• More aspects to be investigated and opportunities to be explored
Backup Slides
Related Work

• Riffle [EuroSys ‘18]
  • External shuffle service in Spark developed by Facebook
  • Merges small files to reduce random disk I/O

• Magnet [VLDB ‘20]
  • Optimized shuffle service in Spark developed by LinkedIn
  • Pushes shuffle data from mappers to reducers to pre-merge intermediate results before the reduce stage

• Remote Shuffle Service (RSS)
  • Spark shuffle service with dedicated remote shuffle servers developed by Uber
  • Separates shuffle data from mappers to improve reliability and scalability

• Exoshuffle
  • Shuffle layer for MapReduce in Ray that easily supports state-of-the-art shuffle optimizations and enables pipelined shuffles
More Related Work

• Optimizing shuffle for graph processing

• Optimizing shuffle for machine learning

• Optimizing the exchange operator in DBMSs