Stateful Distributed Dataflow Graphs: Imperative Big Data Programming for the Masses

Peter Pietzuch
prp@doc.ic.ac.uk

Large-Scale Distributed Systems Group
Department of Computing, Imperial College London
http://lsds.doc.ic.ac.uk

EIT Digital Summer School on Cloud and Big Data 2015 – Stockholm, Sweden
Growth of Big Data Analytics

Big Data Analytics: gaining value from data
- Web analytics, fraud detection, system management, networking monitoring, business dashboard, ...

Need to enable more users to perform data analytics
TIOBE Programming Community Index

Source: www.tiobe.com

Programming Language Popularity
Distributed dataflow frameworks tend to favour functional, declarative programming models

- MapReduce, SQL, PIG, DryadLINQ, Spark, ...
- Facilitates consistency and fault tolerance issues

Domain experts tend to write imperative programs

- Java, Matlab, C++, R, Python, Fortran, ...
Example: Recommender Systems

Recommendations based on past user behaviour through collaborative filtering (cf. Netflix, Amazon, ...):

Distributed dataflow graph

(eg MapReduce, Hadoop, Spark, Dryad, Naiad, ...)

Exploits data-parallelism on cluster of machines
Collaborative Filtering in Java

Matrix `userItem = new Matrix();`
Matrix `coOcc = new Matrix();`

```java
void addRating(int user, int item, int rating) {
    userItem.setElement(user, item, rating);
    updateCoOccurrence(coOcc, userItem);
}
```

```java
Vector getRec(int user) {
    Vector userRow = userItem.getRow(user);
    Vector userRec = coOcc.multiply(userRow);
    return userRec;
}
```

**User-Item matrix (UI)**

<table>
<thead>
<tr>
<th></th>
<th>Item-A</th>
<th>Item-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-A</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>User-B</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

**Co-Occurrence matrix (CO)**

<table>
<thead>
<tr>
<th></th>
<th>Item-A</th>
<th>Item-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item-A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Item-B</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Update with new ratings

Multiply for recommendation

Update with new ratings

User-B

1 2

Crossing out User-B with Item-B

User-Item matrix (UI)

Co-Occurrence matrix (CO)
Collaborative Filtering in Spark (Java)

```java
// Build the recommendation model using ALS
int rank = 10;
int numIterations = 20;
MatrixFactorizationModel model = ALS.train(JavaRDD.toRDD(ratings), rank, numIterations, 0.01);

// Evaluate the model on rating data
JavaRDD<Tuple2<Object, Object>> userProducts = ratings.map(
    new Function<Rating, Tuple2<Object, Object>>() {
        public Tuple2<Object, Object> call(Rating r) {
            return new Tuple2<Object, Object>(r.user(), r.product());
        }
    });
JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = JavaPairRDD.fromJavaRDD(
    model.predict(JavaRDD.toRDD(userProducts)).toJavaRDD().map(
        new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
            public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
                return new Tuple2<Tuple2<Integer, Integer>, Double>(
                    new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
            }
        }));
JavaRDD<Tuple2<Double, Double>> ratesAndPreds =
    JavaPairRDD.fromJavaRDD(ratings.map(
        new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
            public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
                return new Tuple2<Tuple2<Integer, Integer>, Double>(
                    new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
            }
        })).join(predictions).values();```
Collaborative Filtering in Spark (Scala)

```scala
// Build the recommendation model using ALS
val rank = 10
val numIterations = 20
val model = ALS.train(ratings, rank, numIterations, 0.01)

// Evaluate the model on rating data
val usersProducts = ratings.map {
  case Rating(user, product, rate) => (user, product)
}
val predictions =
  model.predict(usersProducts).map {
    case Rating(user, product, rate) => ((user, product), rate)
  }
val ratesAndPreds = ratings.map {
  case Rating(user, product, rate) => ((user, product), rate)
}.join(predictions)
```

All data immutable

No fine-grained model updates
Stateless MapReduce Model

Data model: (key, value) pairs

Two processing functions:
- \( \text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2) \)
- \( \text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3) \)

Benefits:
- Simple programming model
- Transparent parallelisation
- Fault-tolerant processing
Our goals:

Imperative **Java programming model** for big data apps

High throughput through **data-parallel execution** on cluster

**Fault tolerance** against node failures

<table>
<thead>
<tr>
<th>System</th>
<th>Mutable State</th>
<th>Large State</th>
<th>Low Latency</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>No</td>
<td>n/a</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Spark</td>
<td>No</td>
<td>n/a</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Storm</td>
<td>No</td>
<td>n/a</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Naiad</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Stateful Dataflow Graphs (SDGs)

1. Annotated Java program (@Partitioned, @Partial, @Global, ...)
   - Program.java
   - Static program analysis

2. Data-parallel Stateful Dataflow Graph (SDG)

3. SEEP distributed dataflow framework
   - Dynamic scale out & checkpoint-based fault tolerance
   - Cluster

4. Experimental evaluation results
Tasks process data

State Elements (SEs) represent state

Tasks have access to arbitrary state

State elements (SEs) represent in-memory data structures
- SEs are mutable
- Tasks have local access to SEs
- SEs can be shared between tasks
Challenges with Large State

**Mutable state** leads to concise algorithms but complicates **scaling** and **fault tolerance**

```
Matrix userItem = new Matrix();
Matrix coOcc = new Matrix();
```

**State will not fit into single node**

**Challenge:** Handling of distributed state?
State Elements support two abstractions for distributed mutable state:

Partitioned SEs:
Tasks access *partitioned* state by key

Partial SEs:
Tasks can access *replicated* state
Partitioned SE split into disjoint partitions

Key space: [0-N] ➔ [0-k] ➔ [(k+1)-N]

User-Item matrix (UI)

<table>
<thead>
<tr>
<th></th>
<th>Item-A</th>
<th>Item-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-A</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>User-B</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Access by key ➔ hash(msg.id) ➔ Dataflow routed according to hash function

State partitioned according to partitioning key
Partial SEs are replicated (when partitioning is impossible)
- Tasks have local access

Access to partial SEs either **local** or **global**

**Local** access: Data sent to one

**Global** access: Data sent to all
State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values

Requires application-specific **merge logic**
- Merge task reconciles state and updates partial SEs
State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values

Multiple partial values

Merge logic
State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values

Barrier collects partial state
SDG for Collaborative Filtering
SDG for Logistic Regression

Requires support for iteration
Stateful Dataflow Graphs (SDGs)

Annotated Java program (@Partitioned, @Partial, @Global, ...)

Program.java

Static program analysis

Data-parallel Stateful Dataflow Graph (SDG)

SEEP distributed dataflow framework

Dynamic scale out & checkpoint-based fault tolerance

Cluster
Partitioned State Annotation

@Partition field annotation indicates partitioned state

@Partitioned Matrix userItem = new Matrix();
Matrix coOcc = new Matrix();

void addRating(int user, int item, int rating) {
    userItem.setElement(user, item, rating);
    updateCoOccurrence(coOcc, userItem);
}

Vector getRec(int user) {
    Vector userRow = userItem.getRow(user);
    Vector userRec = coOcc.multiply(userRow);
    return userRec;
}
Partial State and Global Annotations

@Partitioned Matrix userItem = new Matrix();
@Partial Matrix coOcc = new Matrix();

void addRating(int user, int item, int rating) {
    userItem.setElement(user, item, rating);
    updateCoOccurrence(@Global coOcc, userItem);
}

@Partial field annotation indicates partial state

@Global annotates variable to indicate access to all partial instances
@Partitioned Matrix userItem = new Matrix();
@Partial Matrix coOcc = new Matrix();

Vector getRec(int user) {
    Vector userRow = userItem.getRow(user);
    @Partial Vector puRec = @Global coOcc.multiply(userRow);
    Vector userRec = merge(puRec);
    return userRec;
}

Vector merge(@Collection Vector[] v){
    /*...*/
}

@Collection annotation indicates merge logic
Java2SDG: Translation Process

Annotated Program.java → Extract TEs, SEs and accesses → Live variable analysis

Extract **state** and **state access patterns** through static code analysis

SOOT Framework

Annotated Program.java

TE and SE access code assembly → SEEP runnable

Javassist

Generation of **runnable code** using TE and SE connections
Stateful Dataflow Graphs (SDGs)

Annotated Java program (@Partitioned, @Partial, @Global, ...)

Program.java

Static program analysis

Data-parallel Stateful Dataflow Graph (SDG)

SEEP distributed dataflow framework

Cluster

Dynamic scale out & checkpoint-based fault tolerance
Scale Out and Fault Tolerance for SDGs

High/bursty input rates $\rightarrow$ Exploit data-parallelism

Large scale deployment $\rightarrow$ Handle node failures

Partitioning of state

Loss of state after node failure
Framework has state management primitives to:
- Backup and recover state elements
- Partition state elements

Integrated mechanism for scale out and failure recovery
- Node recovery and scale out with state support

Expose state as external entity to be managed by the distributed dataflow framework
What is State?

Processing state

<table>
<thead>
<tr>
<th>Item 1</th>
<th>Item 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>User A</td>
<td>2</td>
</tr>
<tr>
<td>User B</td>
<td>4</td>
</tr>
</tbody>
</table>

Buffer state

- Data ts1
- Data ts2
- Data ts3
- Data ts4
State Management Primitives

- Makes state available to framework
- Attaches \textit{last processed data timestamp}

- Moves copy of state from one node to another

- Splits state to scale out tasks
State Primitive: Checkpointing

**Challenge: Efficient checkpointing of large state in Java?**
- No updates allowed while state is being checkpointed
- Checkpointing state should not impact data processing path

**Asynchronous, lock-free checkpointing**
1. Freeze mutable state for checkpointing
2. Dirty state supports updates concurrently
3. Reconcile dirty state
State Primitives: Backup and Restore
State Primitives: Partition

Processing state modeled as \((\text{key}, \text{value})\) dictionary

State partitioned according to key \(k\)
- Same key used to partition streams
Two cases:
- Node B fails ➔ Recover
- Node B becomes bottleneck ➔ Scale out
Recovering Failed Nodes

Periodically, stateful tasks checkpoint and back up state to designated upstream backup node. Use backed up state to recover quickly.

State restored and unprocessed data replayed from buffer.
Scaling Out Tasks

Finally, upstream node replays unprocessed data to update checkpointed state.

For scale out, backup node already has state elements to be parallelised.
Distributed M-to-N Backup/Recovery

Challenge: Fast recovery?
- Backups large and cannot be stored in memory
- Large writes to disk through network have high cost

M to N distributed backup and parallel recovery
- Partition state and backup to multiple nodes
- Recover state to multiple nodes
Stateful Dataflow Graphs (SDGs)

Annotated Java program (@Partitioned, @Partial, @Global, ...)

Program.java

Static program analysis

Data-parallel Stateful Dataflow Graph (SDG)

SEEP distributed dataflow framework

Dynamic scale out & checkpoint-based fault tolerance

Cluster

Experimental evaluation results
100 GB training dataset for classification
Deployed on Amazon EC2 ("m1.xlarge" VMs with 4 vCPUs and 16 GB RAM)

SDGs have comparable throughput to Spark despite mutable state
Mutable State Access: Collaborative Filtering

Collaborative filtering, while changing read/write ratio (add/getRating)
Private cluster (4-core 3.4 GHz Intel Xeon servers with 8 GB RAM)

SDGs serve fresh results over large mutable state
Elasticity: Linear Road Benchmark

**Linear Road Benchmark** [VLDB’04]
- Network of toll roads of size $L$
- Input rate increases over time
- SLA: results $< 5$ secs

Deployed on Amazon EC2 (c1 & m1 xlarge instances)

Scales to $L=350$ with 60 VMs

$L=512$ highest reported result in literature [VLDB’12]

SDGs can scale dynamically based on workload
Large State Size: Key/Value Store

Increase state size in distributed key/value store

SDGs can support online services with mutable state
Summary

Programming models for Big Data matter
- Logic increasingly pushed into bespoke APIs
- Existing models do not support fine-grained mutable state

Stateful Dataflow Graphs support mutable state
- Automatic translation of annotated Java programs to SDGs
- SDGs introduce new challenges in terms of parallelism and failure recovery
- Automatic state partitioning and checkpoint-based recovery

SEEP available on GitHub: https://github.com/lsds/Seep/

Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch, "Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management", SIGMOD’13

Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch, "Making State Explicit for Imperative Big Data Processing", USENIX ATC’14

Thank you! Any Questions?

Peter Pietzuch
<prp@doc.ic.ac.uk>
http://lsds.doc.ic.ac.uk