Java2SDG: Stateful Big Data Processing for the Masses

Raul Castro Fernandez
Imperial College London
rc3011@doc.ic.ac.uk

Panagiotis Garefalakis
Imperial College London
pg1712@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@doc.ic.ac.uk

***Abstract*—Big data processing is no longer restricted to specially-trained engineers. Instead, domain experts, data scientists and data users all want to benefit from applying data mining and machine learning algorithms at scale. A considerable obstacle towards this "democratisation of big data" are programming models: current scalable big data processing platforms such as Spark, Naiad and Flink require users to learn custom functional or declarative programming models, which differ fundamentally from popular languages such as Java, Matlab, Python or C++. An open challenge is how to provide a big data programming model for users that are not familiar with functional programming, while maintaining performance, scalability and fault tolerance.**

We describe JAVA2SDG, a compiler that translates annotated Java programs to *stateful dataflow graphs* (SDGs) that can execute on a compute cluster in a data-parallel and fault-tolerant fashion. Compared to existing distributed dataflow models, a distinguishing feature of SDGs is that their computational tasks can access distributed mutable state, thus allowing SDGs to capture the semantics of stateful Java programs. As part of the demonstration, we provide examples of machine learning programs in Java, including collaborative filtering and logistic regression, and we explain how they are translated to SDGs and executed on a large set of machines.

I. INTRODUCTION

Today we witness the "democratisation of big data"—not just specialised developers want to process and analyse big data but also data scientists, domain experts and business analysts [2]. This is hindered, however, by the bespoke programming models that data-parallel processing platforms, such as Spark [16], Naiad [11] and Flink [6], require users to learn.

While early data-parallel platforms follow the MapReduce programming model [3], which is easy for users to master, they are limited in their expressiveness—e.g. they lacks efficient support for iterative algorithms. With users wanting to execute more complex data mining and machine learning algorithms, platforms need to become more expressive. Current platforms [16], [11], [6] have support for iteration and advanced functions, but they expose functional programming interfaces. Typical users, however, are familiar with *imperative* programming languages such as Java, Matlab, Python or C++, thus struggling with the collections of higher-order functions of current platforms. The programmability of big data platforms consequently becomes a major entry barrier for new users.

Using imperative languages for big data processing, however, introduces challenges: achieving high-throughput becomes difficult when programs have access to *mutable state*. Current big data platforms rely on stateless dataflow graphs that avoid all forms of state—hence their dependence on functional programming models. While this simplifies parallel computation and failure recovery [3], it is incompatible with an imperative programming model.

We argue that, to support an imperative programming model, big data platforms should instead embrace mutable state and offer it as a first-class abstraction. Based on this idea, we describe **stateful dataflow graphs (SDGs)** [5], a new model for data-parallel processing that separates large *mutable state* from the processed *data*. SDGs include primitives for maintaining distributed state efficiently: if tasks can operate on state entirely in parallel, the state is *partitioned* across nodes; if this is not possible, tasks are given local instances of *replicated* state for independent computation. Computation can include synchronisation points at which all partial state instances are accessed, and state instances can be reconciled at any point according to the semantics of the algorithm.

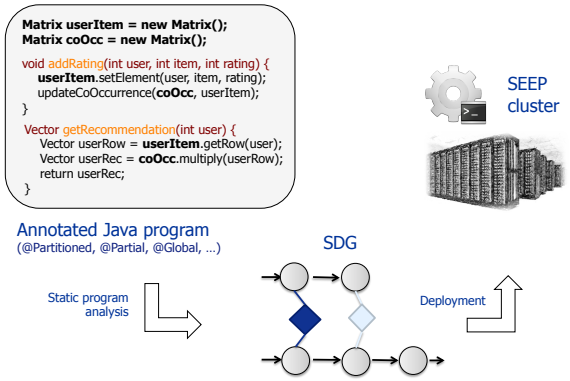

Fig. 1: Data-parallel execution of Java programs using JAVA2SDG

As shown in Figure 1, we demonstrate the features and operation of **JAVA2SDG** [5], a compiler that translates annotated Java programs to SDGs, enabling their execution on a cluster in a data-parallel and fault-tolerant fashion. The JAVA2SDG compiler infers the dataflow and the types of state accesses from the Java program and uses this information to generate an executable SDG. Using static program analysis, it extracts the computational tasks, state fields and variable-level dataflow from the Java program.

As part of the demonstration, we show how JAVA2SDG enables the execution of several Java programs at scale: collaborative filtering, logistic regression and a key/value store. We use SEEP [4], a stateful data-parallel processing system, as the execution engine for SDGs. We explain the translation of Java code to SDGs through its different stages with the help of graphical representations generated by JAVA2SDG. The audience can explore existing Java programs and modify them to understand an imperative big data processing model.

Next, we describe the Java programs that will be demonstrated in detail. §III explains stateful dataflows graphs and their generation. §IV introduces the use of the programming model from a user perspective. In §V, we give an overview of the proposed demonstration. The paper finishes with a related work discussion (§VI) and conclusions (§VII).

Algorithm 1: Logistic regression

```
1  final int NUM_FEATURES = 1000;
2  final int ITER = 15;
3  final double DELTA = 0.01;
4
5  @Partial double[] weights = new double[NUM_FEATURES];
6
7  void train(List<double[]> samples, List<Double> labels) {
8    for (int i = 0; i < ITER; i++) {
9      for (double[] sample : samples) {
10       double predicted = classify(sample, weights);
11       double[] gradient = new double[NUM_FEATURES];
12       for (int j = 0; j < NUM_FEATURES; j++) {
13         double f = sample[j];
14         gradient[j] = (f * (labels.get(j) - predicted) * DELTA);
15       }
16       weights = add(weights, gradient);
17     }
18     double[] gWeights = mergeWeights(@Global weights);
19     @Global weights = gWeights;
20   }
21 }
22 double test(double[] sample) {
23   double label = classify(sample, weights);
24   return label;
25 }
26 double[] mergeWeights(@Collection List<double[]> weights) {
27   double [] gWeights = new double[NUM_FEATURES];
28   for (double[] pw : weights)
29     for (int i = 0; i < NUM_FEATURES; i++)
30       gWeights[i] = gWeights[i] + pw[i];
31   int numSamples = weights.size();
32   for (int i = 0; i < NUM_FEATURES; i++)
33     gWeights[i] = gWeights[i] / numSamples;
34   return gWeights;
35 }
36 double classify(double[] sample, double[] weights) {
37   double z = 0;
38   for (double w : weights)
39     z += multiply(w, sample);
40   return 1 / (1 + Math.exp(-z));
41 }
```

II. Programming Model

Our imperative programming model allows users to implement algorithms with mutable state in Java. In this section, we describe Java programs that can be translated by JAVA2SDG and executed on a cluster running the SEEP platform. The annotations (starting with '@') will be explained in §IV.

Logistic regression. Algorithm 1 is a complete implementation of logistic regression, an iterative machine learning algorithm widely used to perform supervised classification tasks. It represents the learned model as a vector of weights (line 5). The `train` method receives a new data item and uses it to train the model (line 7); the `test` method uses the trained model to classify a yet unknown item (line 22). We use this program to train a model that classifies machine- and human-generated web requests in an online fashion.

Collaborative filtering. We also demonstrate the implementation of a collaborative filtering algorithm (see Algorithm 2), typically used to provide recommendations. The algorithm uses matrices to represent user preferences about items and the relationships between items. It implements two functions, `addRating()` and `getRecommendation()`, that update the matrices and provide online recommendations, respectively. We use this algorithm to provide online film recommendations to users based on data available from Netflix.

Distributed key/value store. To show the generality of our programming model, we also implement a distributed in-memory key/value store in Java (see Algorithm 3). The key/value store maintains a map of counters that record the number of visits of users to a web site. We provide methods to both update and retrieve the number of visits per user. The program differs from a single-node implementation in only one annotation, but, after compilation by JAVA2SDG, it can execute on a cluster of machines.

Algorithm 2: Online collaborative filtering

```
1  @Partitioned Matrix userItem = new Matrix();
2  @Partial Matrix coOcc = new Matrix();
3
4  void addRating(int user, int item, int rating) {
5    userItem.setElement(user, item, rating);
6    Vector userRow = userItem.getRow(user);
7    for (int i = 0; i < userRow.size(); i++)
8      if (userRow.get(i) > 0) {
9        int count = coOcc.getElement(item, i);
10       coOcc.setElement(item, i, count + 1);
11       coOcc.setElement(i, item, count + 1);
12     }
13 }
14 Vector getRecommendation(int user) {
15   Vector userRow = userItem.getRow(user);
16   @Partial Vector userRec = @Global coOcc.multiply(userRow);
17   Vector rec = merge(@Global userRec);
18   return rec;
19 }
20 Vector merge(@Collection Vector[] allUserRec) {
21   Vector rec = new Vector(allUserRec[0].size());
22   for (Vector cur : allUserRec)
23     for (int i = 0; i < allUserRec[0].size(); i++)
24       rec.set(i, cur.get(i) + rec.get(i));
25   return rec;
26 }
```

Algorithm 3: Distributed key/value store

```
1  @Partitioned Map<Integer, Integer> store = new HashMap<>();
2
3  public void incrUserVisits(int uid) {
4    int visits = 0;
5    if (store.containsKey(uid))
6      visits = store.get(uid);
7    store.put(uid, visits + 1);
8  }
9  public int getUserVisits(int uid) {
10   int visits = store.get(uid);
11   return visits;
12 }
```

We highlight two aspects of our programming model. First, it allows the concise representation of stateful algorithms, as opposed to stateless dataflow models that force users to rewrite algorithms without side-effects. Second, it permits users to combine high-throughput and latency-sensitive functions in the same program, which thus merging the computation and serving of results.

III. Stateful Dataflow Graphs

Stateful dataflow graphs (SDGs) are designed to facilitate the translation of stateful imperative programs to a dataflow representation that performs parallel, iterative computation.

A. Overview

An SDG specifies the computation as a dataflow graph that separates computation and state. As shown in the example in Figure 2, which depicts the SDG for Algorithm 2, an SDG has two types of vertices: (i) *task elements* (TEs) are computational tasks that transform input to output data; and (ii) *state elements* (SEs) represent the computational state accessed and used by TEs. *Dataflows* carry data between TEs. To achieve fault tolerance, SDGs rely on an asynchronous checkpointing mechanism with log recovery [5].

Task elements (TEs) are arbitrary functions that transform input dataflows into output dataflows. A TE may run multiple instances on distributed nodes to process data in parallel. For example, the SDG in Figure 2 has 5 TEs, which form the dataflow graph of the collaborative filtering algorithm.

State elements (SEs) encapsulate the state used by TEs for computation. Access to SEs from TEs is always local, and SEs are implemented using arbitrary in-memory data structures such as maps, lists or matrices. SEs can be distributed across nodes if (i) the state is too large to fit in the memory of a single

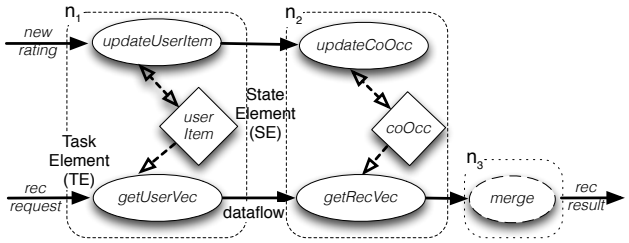

Fig. 2: Example of stateful dataflow graph with task elements, state elements and dataflows

node; or (ii) the TE that accesses the state is itself distributed for parallel processing.

SEs can be distributed according to one of two strategies: *partitioned* and *partial*. Since the distribution strategy affects the semantics of the program, the correct strategy must be chosen by the user using source code annotations, as described in §IV-A.

Partitioned state. A *partitioned SE* can be split into disjoint sets. For example, a map with integer keys ranging from $[0:n]$ can be split into two disjoint subsets according to a partitioning key k, namely $[0:k]$ and $[k{+}1:n]$.

As a consequence, a TE accessing a partitioned SE must access the correct partition. Partitioned SEs yield better performance because access by TEs to different partitions can occur in parallel without the need for synchronisation, but not all algorithms use state access patterns that can be partitioned.

Partial State. If the state cannot be partitioned, a *partial SE* creates multiple replicas of the state on different nodes. Each TE reads and writes its local SE replica independently, permitting it to diverge from the other replicas. Each SE replica therefore represents incomplete (partial) information about the SE, e.g. it may not include all updates performed on the SE.

When accessing a partial SE, the default access is to the single local replica (e.g. as shown in Algorithm 1, line 16). An explicit source code annotation (see §IV-A) permits access to *all* replicas of the partial state.

A special *merge task* in the SDG introduces a global synchronisation point by collecting data from all partial SE replicas and combines it in an application-specific manner. As a result, frequent merging of partial SEs affects performance—partial SEs should only be used if the semantics of the algorithm does not permit state partitioning.

B. Translation Process

An SDG is generated from an (annotated) Java program by the JAVA2SDG compiler. The compiler operates in two stages, *dataflow construction* and *code generation*. The dataflow construction stage translates the Java program using static analysis. The code generation stage performs bytecode rewriting and synthesis of executable code for the task elements in the SDG based on the original Java program. The generated code can be executed by SEEP [4], a data-parallel processing platform that supports SDGs (see Figure 1).

Dataflow construction. The first stage extracts the required TEs and SEs from the program and creates the SDG topology. Java attributes used by methods are translated to SEs. JAVA2SDG has SE implementations for common Java classes such as `Map`, `List` and `Set`; new types of SEs can be supported as long as the user provides implementation for checkpointing, partitioning and recovery of state [5].

Statements in the Java program are translated to TEs by splitting them into groups such that each group only accesses a single SE. This strategy results in two desirable properties: (i) it facilitates data parallelism by associating a TE with only one SE; and (ii) it introduces pipeline parallelism by creating a sequence of TEs.

The topology of the SDG is determined based on the *control flow* of the Java program. The dataflows between TEs are derived from the program using live variable analysis at the boundaries between TEs. Live variables between TE boundaries must be exchanged over the network as dataflows.

Code generation. In the second stage, JAVA2SDG generates new code and rewrites existing one to execute the SDG in the cluster. First it synthesises new code that: (i) creates wrappers around SEs so that the system can manage them for fault tolerance; (ii) embeds the TE code in a template that allows the platform to deploy and schedule TEs; and (iii) acts as the driver program for setting up the correct topology of the SDG as part of the deployment.

Second, it rewrites the Java bytecode of TE implementations so that the content of variables at TE boundaries is serialised and transmitted over the network between nodes.

IV. WRITING JAVA2SDG PROGRAMS

This section explains how users write annotated Java programs for JAVA2SDG. Users provide (i) the methods that process the input data and generate output and (ii) a deployment configuration that specifies the input data and the output to be used. When writing the methods that process data, this section explains how to use source code annotations that disambiguate the handling of distributed state.

A. Source Code Annotations

When the user provides an implementation of logistic regression, as shown in Algorithm 1, they want to execute the `train` method in a distributed fashion to achieve high processing throughput with a large volume of training data. As a result, the field `weights` (line 5), which contains the model to be trained, should be distributed.

Note that we do not attempt to be completely transparent for developers or to address the general problem of automatic code parallelisation [7]. Instead, the JAVA2SDG compiler requires information on how to distribute the model, and how the distributed model should be accessed by the `train` and `test` methods. Next we explain how this information is provided by the user through source code annotations:

@Partitioned and @Partial. If the state of an algorithm can be partitioned across nodes, the field should be annotated with `@Partitioned`. In the example of the logistic regression algorithm, however, the `train` method requires access to *all* `weights`, which means that `weights` cannot be partitioned. Instead, the user adds the `@Partial` annotation to the definition of `weights`, which results in multiple distributed replicas of `weights` being created, representing it as a partial SE in the SDG. These replicas can be updated independently by separate instances of the `train` method executing on different nodes.

@Global. When the `train` method accesses the partial `weights`, the user must choose if the access refers to only *one*

of the replicas or *all* of them. By default, access is only to a single local replica of `weights`, e.g. as shown in line 10.

After each iteration of the training stage, however, the replicas of `weights`, which were trained independently, must be merged to reach convergence of the model. Therefore, the access to `weights` is annotated with the `@Global` annotation to denote that the statement refers to all replicas (line 18).

@Collection. The reconciliation of the partial `weights` is performed by the `mergeWeights` method (line 26). To indicate that this method receives all replicas of `weight` as input, its input parameter has the `@Collection` annotation, which turns the parameter into an array with one entry per replica. This permits `mergeWeights` to compute a new global value of `weights`.

V. Demonstration Overview

Next we describe what users see as part of this demonstration, which includes three algorithms: (i) a collaborative filtering implementation; (ii) a logistic regression implementation; and (iii) a distributed key/value store.

Translation of algorithms. To demonstrate the details of the translation process performed by JAVA2SDG, we modify the compiler to produce a graphical representation of the SDG in addition to the runnable program. The SDG visualisation contains elements that give more details about the translation process, including (i) the results of the live variable analysis after the generation of each TE in the SDG; (ii) the grouping of Java statements into separate TEs based on their state accesses; and (iii) the type of access method used for each SE, i.e. local, global or partitioned.

Execution of algorithms. As part of the demonstration, we select the programs, modify them, explore their SDGs and execute them on a remote compute cluster. The execution uses the SEEP platform [4], [5], our implementation of a data-parallel processing system for SDGs.

VI. Related Work

Programming data-parallel platforms. A large class of data processing platforms support a functional or declarative programming model. MapReduce [3] only has two higher-order functions; next generation systems such as DryadLINQ [15] or Spark facilitate the automatic distribution of computation due to their use of stateless functions. While a functional or declarative model such as SQL, eases distribution and fault tolerance, imperative models such as R and Python remain more widely used among data scientists [8].

Storm [14] and SEEP [4] expose a low-level dataflow programming model: algorithms are defined as a dataflow pipeline, which is not a natural way to express applications, and is hard to debug. Naiad [11] supports functional and declarative programming models on top of its dataflow model, but requires the adoption of new primitives.

Imperative programming models. CIEL [10] uses imperative constructs such as the spawning of new tasks and futures, but this exposes the low-level execution of the dynamic dataflow graph to developers. Piccolo [12] is inspired by imperative distributed shared memory models, but users must choose the task granularity manually, i.e. there is no dataflow abstraction.

Pydron [9] introduces a one-sided communication model for stateless Python programs. In contrast, SDGs permit the use of explicit state, which facilitates the concise representation of stateful algorithms, and allows generic dataflows (not just master-client communication) to execute in parallel on clusters.

Program parallelisation. Extracting parallel dataflows from imperative programs is a hard problem [7]. Our approach is similar to that of Beck et al. [1], in which a dataflow graph is generated compositionally from the execution graph. Jade [13] is another approach for program parallelisation that relies on users explicitly deciding on the granularity of tasks and how these access share objects, similar to our partial annotations. JAVA2SDG does not require users to specify the granularity of each task. We find that our strategy for doing this automatically performs well for many practical big data applications.

VII. Conclusion

We want to enable a broader set of users to write programs that can execute efficiently on big data. We demonstrate **JAVA2SDG**, a compiler that translates annotated Java programs into SDGs, a stateful distributed dataflow model for executing programs in a data-parallel fashion on a compute cluster. We demonstrate how big data algorithms can be written naturally using this programming model and show how JAVA2SDG translates them to SDGs. We believe that this programming model is a first step towards the democratisation of big data.

References

[1] M. Beck and K. Pingali. From Control Flow to Dataflow. In *ICPP*, 1990.

[2] The Beckman Report on Database Research. http://beckman.cs.wisc.edu/beckman-report2013.pdf.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *CACM*, 2008.

[4] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD*, 2013.

[5] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*, 2014.

[6] Apache Flink 0.10.1. http://flink.incubator.apache.org, 2016.

[7] W. M. Johnston, J. Hanna, et al. Advances in Dataflow Programming Languages. In *ACM Computing Surveys*, 2004.

[8] KDnuggets Annual Software Poll. RapidMiner and R vie for the First Place. http://goo.gl/OLikb, 2014.

[9] S. C. Muller, G. Alonso, et al. Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud. In *OSDI*, 2014.

[10] D. Murray, M. Schwarzkopf, et al. CIEL: A Universal Exec. Engine for Distributed Data-Flow Comp. In *NSDI*, 2011.

[11] D. G. Murray, F. McSherry, et al. Naiad: A Timely Dataflow System. In *SOSP*, 2013.

[12] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI*, 2010.

[13] M. Rinard and M. Lam. The Design, Implementation and Evaluation of Jade. In *ACM TOPLAS*, 1998.

[14] Twitter Storm 0.10.0. github.com/apache/storm, 2016.

[15] Y. Yu, M. Isard, et al. DryadLINQ: a System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language. In *OSDI*, 2008.

[16] M. Zaharia, M. Chowdhury, et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.