Designing Parallel Data Processing for Enabling Large-Scale Sensor Applications

Milan Kabáč, Charles Consel, Nic Volanschi

▶ **To cite this version:**

Milan Kabáč, Charles Consel, Nic Volanschi. Designing Parallel Data Processing for Enabling Large-Scale Sensor Applications. Personal and Ubiquitous Computing, 2017, 21 (3), pp.457-473. 10.1007/s00779-017-1009-1 . hal-01470281

HAL Id: hal-01470281

https://inria.hal.science/hal-01470281

Submitted on 17 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Personal and Ubiquitous Computing manuscript No.
(will be inserted by the editor)

Designing Parallel Data Processing for Enabling Large-Scale Sensor Applications

Milan Kabáč · Charles Consel · Nic Volanschi

Received: date / Accepted: date

Abstract Masses of sensors are being deployed at the scale of cities to manage parking spaces, transportation infrastructures to monitor traffic, and campuses of buildings to reduce energy consumption. These large-scale infrastructures become a reality for citizens via applications that orchestrate sensors to deliver high-value, innovative services. These applications critically rely on the processing of large amounts of data to analyze situations, inform users, and control devices.

This paper proposes a design-driven approach to developing orchestrating applications for masses of sensors that integrates parallel processing of large amounts of data. Specifically, an application design exposes declarations that are used to generate a programming framework based on the MapReduce programming model.

We have developed a prototype of our approach, using Apache Hadoop. We applied it to a case study and obtained significant speedups by parallelizing computations over twelve nodes. In doing so, we demonstrate that our design-driven approach allows to abstract over implementation details, while exposing architectural properties used to generate high-performance code for processing large datasets. Furthermore, we show that this high-performance support enables new, personalized services in a smart city. Finally, we discuss the expressiveness of our design language, identify some limitations, and present language extensions.

Keywords data processing · programming frameworks · sensors · map-reduce · orchestration

1 Introduction

Modern smart cities and smart territories [1] rely on wide-area infrastructures, populating a variety of environments with functionality-rich sensors. These smart environments include wide-area transportation management [2,3] and large-scale smart parking systems [4,5]. The emergence of smart environments validates large-scale sensor infrastructures as robust platforms for delivering innovative services to citizens.

Nevertheless, the successful adoption of these infrastructures critically relies on the ability to develop services. Currently, software development in this domain lacks programming models and methodologies to address key domain-specific challenges. In particular, masses of sensors produce large amounts of data that require to be analyzed efficiently to timely deliver high-value services to citizens and operators of smart environments. When considering tens of thousands of measurements, possibly accumulated over a period of time, processing of such data volumes becomes a critical issue. The pressure on processing only increases when the added values of the services rely on real-time or near-real-time analyses. In fact, the data *volume* to be processed and the *velocity* requirements of the applications to be developed may necessitate parallel processing [6]. For example, as cars rush into a city in the morning, drivers should receive up-to-date information about space availability in parking lots and estimations about its future trends, even if this involves processing

Milan Kabáč
Inria Bordeaux, France
E-mail: milan.kabac@inria.fr

Charles Consel
Bordeaux Institute of Technology & Inria Bordeaux
France
E-mail: charles.consel@inria.fr

Nic Volanschi
Inria Bordeaux, France
E-mail: eugene.volanschi@inria.fr

massive amounts of data repeatedly. When efficiency is paramount, it is a key challenge to develop an orchestrating application that exploits properties about the sensors, optimizes the strategies to collect sensor measurements, and crunches large amounts of data.

Beyond allowing to harness large-scale sensor infrastructures, the scalability of data processing is becoming a key enabling factor for delivering personalized and/or community-aware services [7,8]. Indeed, in such applications, not only do large amounts of sensor data have to be handled, but they must also be combined with massive data, contributed by user communities. These computations must be performed repeatedly for each user and still be delivered in a timely manner. Support for efficient parallel processing can thus also pave the way to the next level of smart city services for citizens, in terms of added value.

Existing approaches dedicated to big data processing provide limited ways to combine data processing strategies with the application logic. Apache Pig [9] and Hive [10] require developers to describe data processing in SQL-like query languages with limited support for user-defined functions. Language libraries, such as FlumeJava [11] allow developers to implement data processing via high-level language abstractions. These approaches provide data flow expressions and a set of rich data types to implement data processing. Developers still need to decide when and where data processing occurs, as well as how intermediate computations are combined. In the case of large-scale orchestration, applications may have to analyze sensor data a number of times using different algorithms, or combine them. These needs put an additional burden on developers since they have to introduce boilerplate code to separate library-specific code from the main application logic, interconnect and coordinate computations, store intermediate results, *etc.*

This paper proposes a design-driven approach to developing orchestrating applications for masses of sensors that integrates parallel processing of large amounts of data. In doing so, we extend our previous work on a design language dedicated to orchestrating sensors, named DiaSwarm [12], which did not address high-performance data processing. Our new approach provides the developer with declarations expressing when and where data processing occurs. The application design then compiles into a programming framework, based on the MapReduce programming model. This framework supports and guides the programming of the orchestration logic, while abstracting over the parallel processing of sensed data.

This article is an expanded version of a conference paper [13]. It provides details about the implementation of our approach, an example of personalized service enabled by our approach, and a review of the current limitations of the approach together with some corresponding language extensions.

1.1 Our contributions

High-level parallel processing model.
Our approach allows the developer to program against a framework based on the *MapReduce* programming model [14,15]. In doing so, the developer uses a well-proven approach to processing large datasets, based on a parallel implementation. We illustrate our approach with a case study of a parking management system.

A generative programming approach.
The generated parallel-processing programming frameworks have a carefully structured data and control flow, which enables data processing to be implemented efficiently. Our compiler generates programming frameworks that rely on the MapReduce model, exposing structural parallelism of the implementation. This strategy allows to cope with large datasets collected from masses of sensors.

Implementation.
Our approach is implemented[1] and takes the form of a plugin for the Eclipse IDE[2]. The plugin comprises a code generator, which currently produces programming support for the Apache Hadoop platform[3].

Validation.
Our implementation is validated with an experiment that runs application computations over a large dataset of synthetic sensor readings. The experiment demonstrates that programming frameworks generated by our approach exhibit scalable behavior.

Enabling personalized services.
We further illustrate the practical applicability of our approach through an example of personalized service for citizens of a smart city.

Exploring the design space.
Finally, we assess the expressiveness of DiaSwarm by reviewing and reconsidering design choices. This results in language extensions.

2 Background & Case Study

In this section, we provide a brief introduction of the DiaSwarm language [12] dedicated to development of

[1] http://phoenix.inria.fr/software/diaswarm
[2] http://eclipse.org/
[3] http://hadoop.apache.org/

orchestrating applications. DiaSwarm is a declarative domain-specific design language, which follows the Sense/-Compute/Control (SCC) paradigm promoted by Taylor *et al.* [16]. DiaSwarm provides high-level, declarative constructs to allow developers to deal with sensors and actuators at design time, prior to programming the application. Application design is processed by a compiler, which generates support for the developer that takes the form of a programming framework [17]. The generated programming framework reflects application design and covers domain-specific functionalities, such as service discovery, data gathering, component interaction and data processing. These dimensions are fully administered by the framework to allow developers to concentrate on the application logic.

Application design takes the form of a directed acyclic graph (DAG) comprising devices (*i.e.,* sensors and actuators) and application components, namely, contexts and controllers. Context components receive data from sensors via device sources. They refine raw data into application values and may publish these values to controller components. Controllers determine the devices that need to be actuated, as well as the type of action that needs to be triggered.

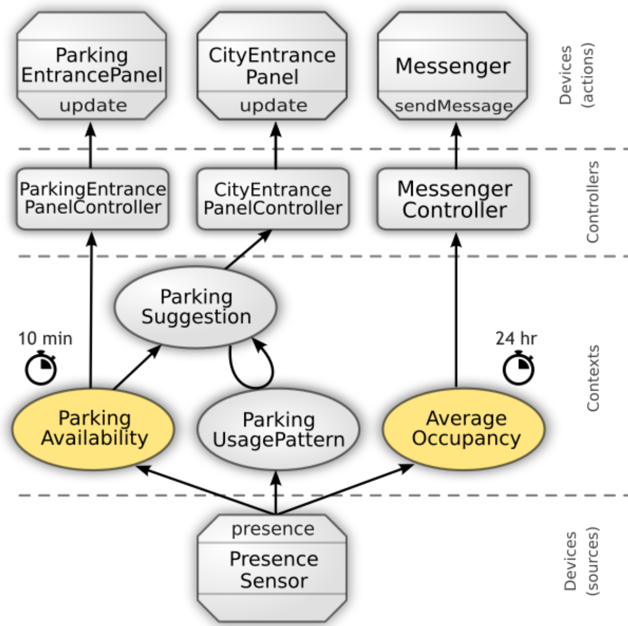

Fig. 1: The graphical view of the parking management application.

2.1 Case study

We illustrate the salient features of DiaSwarm with a smart city application, which monitors the occupancy of parking lots to guide cars to available parking spaces. The application collects data from presence sensors, which are buried under the ground and determine availability of parking spaces via magnetic field variations. The application provides drivers with the number of available parking spaces for each parking lot in the city. This information is displayed on screens at the entrance of parking lots. The application also suggests parking lots to drivers entering the city to optimize the flow of traffic. Finally, the application determines the average occupancy level of each parking lot in 24 hours. The occupancy level is provided to parking managers via messages.

Fig. 1 presents a graphical view of the parking management application in SCC. The `PresenceSensor` device produces values via the `presence` source to the subscribed context components, namely `ParkingAvailability`, `ParkingUsagePattern`, and `AverageOccupancy`. The `ParkingAvailability` context computes the number of available parking spaces in parking lots and publishes these values at regular intervals to the `ParkingEntrancePanel` controller, which in turn triggers the `update` action to refresh the number of available parking spaces on entrance screens. Parking suggestions for drivers are computed by the `ParkingSuggestion` context, which is invoked every time the `ParkingAvailability` context publishes a value. In this case, the computation carried out by `ParkingSuggestion` context requires also data from the `ParkingUsagePattern` context. The resulting suggestions are published to the `CityEntrancePanelController`, which refreshes these suggestions on entrance panels. The average occupancy level functionality is designed in a similar fashion with the exception of providing computations over a 24-hour period (*i.e.,* `AverageOccupancy` context).

2.2 Preliminaries

Let us now briefly present the salient features of DiaSwarm declarations through fragments of the design of our case study, displayed in Fig. 2. Note that we omit details on controller components and actuators. The complete design for the parking management application and further information on DiaSwarm can be found on our website.[4]

Service discovery. DiaSwarm service discovery is part of the design phase. The language provides application-specific high-level constructs for discovering objects in the large. The `grouped by` clause allows sensor data to be presented to applications through subsets of interest. In the case of the, `ParkingAvailability` context,

[4] `http://phoenix.inria.fr/software/diaswarm`

parking spaces are gathered together in parking lots, as shown in line 3. Similarly, in line 10, the `AverageOccupancy` context groups presence values by parking lots and computes average occupancy over 24 hours.

Data gathering. DiaSwarm provides three data delivery models, inspired by the domain of wireless sensor networks [18], namely periodic, event-driven and query driven. Data delivery declarations are called *interaction contracts*. Examples are listed in lines 2 and 9, where both `ParkingAvailability` and `AverageOccupancy` contexts require presence measurements to be provided every 10 minutes. Thus, according to these interaction contracts both context components will be activated every 10 minutes with presence values. Furthermore, the event-driven model provides data to context components upon an event of interest (*e.g.,* intrusion). The query-driven model allows a context to request data from devices and other contexts.

Programming frameworks. To enforce domain-specific functionalities (*e.g.,* service discovery) during programming, Java programming frameworks are produced by a compiler from DiaSwarm designs. These frameworks provide an abstract class for each component, which in turn requires developers to implement components by subclassing every abstract class.

```
1  context ParkingAvailability as Availability[] {
2    when periodic presence from PresenceSensor <10 min>

3    grouped by parkingLot
4    with map as Boolean reduce as Integer
5    always publish;
6  }

8  context AverageOccupancy as ParkingOccupancy[] {
9    when periodic presence from PresenceSensor <10 min>

10   grouped by parkingLot every <24 hr>
11   with map as Presence reduce as Integer
12   always publish;
13 }

15 device PresenceSensor {
16   attribute parkingLot as ParkingLotEnum;
17   source presence as Boolean;
18 }
19
20 structure Presence {
21   presence as Boolean;
22   time as String;
23 }
```

Fig. 2: Excerpt of the parking management application design in DiaSwarm.

2.3 Data processing

Although high level, the DiaSwarm declarations suggest data processing models. Specifically, an application is reactive and consists of chains of component activations. A chain is executed when its initial activation condition holds, which is always related to a sensor, and depends on its delivery model: a sensor publishes data spontaneously or is sampled periodically. The execution of a chain ends if one or more actuators are invoked or a component does not publish any value. Additionally, when a component declaration groups values (*e.g.,* `grouped by parkingLot`), it will process a sequence of values, indexed by the grouping attribute (*i.e.,* `parkingLot`). For example, in the `ParkingAvailability` component, the processing will receive values from all the presence sensors, indexed by parking lot identifiers (*i.e.,* `ParkingLotEnum`). Additionally, this construct allows values to be accumulated over a period of time, as illustrated by the `AverageOccupancy` context (line 8). The declaration in line 10 allows presence values, not only to be grouped by `parkingLot`, but also to be accumulated over a 24-hour period (keyword `every`).

3 Exposing Parallelism

The large amount of data collected from sensors calls for efficient processing strategies. We now examine how an application design influences the way data are processed. This study allows us to propose extensions to DiaSwarm and novel treatments of declarations to generate efficient parallel processing of large-scale datasets.

Our aim is to put in synergy design and programming by leveraging design declarations to expose parallelism and allow efficient processing strategies to be implemented. An ideal case study is the `grouped by` directive because it partitions a large set of gathered data and exposes a processing strategy that matches the MapReduce programming model. Indeed, this programming model is dedicated to processing large datasets in a massively parallel manner [14, 15]. It requires processing to be split into two phases: Map and Reduce. Following our approach, data processing needs to be reflected in the design phase. This is done by extending the `grouped by` directive with an optional clause that specifies what types of values are produced by both the Map and Reduce phases. This is illustrated in Fig. 2, where the `ParkingAvailability` declaration includes a MapReduce clause that declares the Map phase to produce `Boolean` values and the Reduce phase to produce `Integer` values.

The DiaSwarm compiler generates a programming framework that requires the developer to provide an im-

```java
1 public class ParkingAvailability extends AbstractParkingAvailability
2                         implements MapReduce<ParkingLotEnum, Boolean, ParkingLotEnum, Boolean, ParkingLotEnum,
                              Integer> {
3     @Override
4     public void map(ParkingLotEnum parkingLot, Boolean presence, MapCollector<ParkingLotEnum, Boolean> collector
          ) {
5         if(!presence)
6             collector.emitMap(parkingLot, true);
7     }

9     @Override
10    public void reduce(ParkingLotEnum parkingLot,
11                       List<Boolean> values, ReduceCollector<ParkingLotEnum, Integer> collector) {
12        int sum = 0;
13        for (int i = 0; i < values.size(); i++) {
14            sum++;
15        }

17        collector.emitReduce(parkingLot, sum);
18    }

20    @Override
21    protected List<Availability> onPeriodicPresence(Map<ParkingLotEnum, Integer> presenceByParkingLot) {
22        List<Availability> availabilityList = new ArrayList<Availability>();

24        for(Entry<ParkingLotEnum, Integer> parkingLot : presenceByParkingLot.entrySet()) {
25            Availability availability = new Availability(parkingLot.getKey(), parkingLot.getValue());
26            availabilityList.add(availability);
27        }

29        return availabilityList;
30    }
31 }
```

Fig. 3: An implementation of the ParkingAvailability context with MapReduce.

plementation for both the Map and Reduce phases of the data processing. As shown in Fig. 3, this is done by implementing `map` and `reduce` methods declared in the generated MapReduce interface. In conformance with the MapReduce model, the Map function is passed a key and a value, which correspond to the parking lot identifier (*i.e.,* the attribute of the `grouped by` directive) and an availability status, provided by the corresponding sensor. The `emitMap` method is invoked to produce each key/value pair result of the Map phase. The framework-generated code groups the results of the Map phase into a list that is then passed to the Reduce phase. This phase sums up the set of values associated with a given intermediate key and, subsequently, emits the availability of a parking lot (`emitReduce`). The data resulting from the MapReduce computation are presented to the developer in the form of a map (line 21). The `onPeriodicPresence` method (line 21 to 30) wraps data resulting from the MapReduce process into the `availabilityList` sequence (line 26), which is returned to subscribed components (*i.e.,* `ParkingEntrancePanelController`, `ParkingSuggestion`).

Although our example involves simple processing, in practice, our design-driven generative approach reduces programming efforts by automatically generating application-specific MapReduce programming frameworks. Furthermore, the generated code keeps the development process straightforward since it prevents specificities of the MapReduce implementation (job scheduling/-configuration/execution, distributed file system, APIs, *etc.*) to percolate into the application logic.

4 Generating a Programming Framework

Our design-driven development approach facilitates the processing of large datasets collected from sensor infrastructures by providing the developer with a customized framework, following the MapReduce programming model. In this section, we show how generative programming is used to produce support for combining an orchestrating application with an actual implementation of MapReduce, namely Hadoop.

Apache Hadoop is an open source implementation of the MapReduce paradigm, which has gained increasing attention over the last years and is currently being used

```
public class ParkingAvailabilityJob extends Configured implements Tool {

    public static class ParkingAvailabilityMap extends MapReduceBase
                                                    implements Mapper<LongWritable, Text, Text, BooleanWritable> {
        @Override
        public void map(LongWritable key, Text value, OutputCollector<Text, BooleanWritable> output, Reporter
             reporter) {
            jobLauncher.doMap(key, value, output);
        }
    }

    public static class ParkingAvailabilityReduce extends MapReduceBase
                                                    implements Reducer<Text, BooleanWritable, Text, IntWritable> {
        @Override
        public void reduce(Text key, Iterator<BooleanWritable> values, OutputCollector<Text, IntWritable> output
             , Reporter reporter) {
            jobLauncher.doReduce(key, values, output);
        }
    }

    @Override
    public int run(String[] args) {
        JobConf conf = new JobConf(getConf(), ParkingAvailabilityJob.class);
        conf.setInputFormat(TextInputFormat.class);
        // Remaining configuration
    }
}
```

Fig. 4: An example of the generated Hadoop MapReduce program for the ParkingAvailability context.

by a number of companies, including IBM, LinkedIn, Facebook and Google [19]. In our approach, our compiler generates a MapReduce program that relies on the Hadoop framework. Furthermore, this MapReduce program defines default configuration parameters that enable a job to be executed in Hadoop.

In the next three subsections, we explain how Hadoop jobs are automatically generated, and how they are integrated in the control and data flow of the generated framework. The subsequent section discusses the possible integration of big data processing backends other than Hadoop, for supporting continuous stream processing in addition to batch processing.

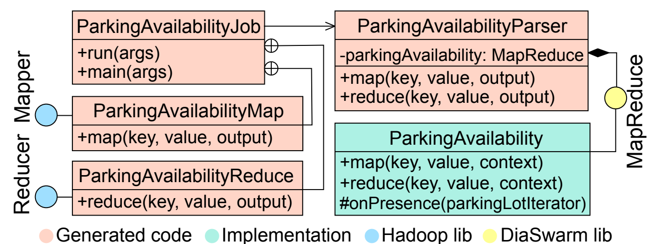

Fig. 5: The generated support for integrating Apache Hadoop.

4.1 Setting up Hadoop jobs

Let us describe first how Hadoop jobs are automatically generated, by examining the code generated for the `ParkingAvailability` context, shown in Fig. 5. The `ParkingAvailabilityJob` class defines a Hadoop MapReduce program, which comprises the definition of both the `map` and `reduce` methods along with code related to the job configuration and execution. Both the Map function and the Reduce function are implemented by overriding the `map` and `reduce` methods of the respective `Mapper` and `Reducer` interfaces. Typically, when using the Hadoop MapReduce library, the definition of the `map` and `reduce` methods resides in the MapReduce program. In this case, however, the implementation of these operations has already been provided by the developer in the `ParkingAvailability` class. The MapReduce program invokes the user-defined `map` and `reduce` methods via the `ParkingAvailabilityParser` class, which keeps an instance of the `ParkingAvailability` context. `ParkingAvailabilityParser` interprets input data of the MapReduce program as corresponding DiaSwarm types and invokes the required `map`/`reduce` method. Consequently, results from the user-defined `map`/`reduce` method are translated to the MapReduce program and submitted via its output collector.

Fig. 4 shows the `ParkingAvailabilityJob` class, which defines the MapReduce program for the `ParkingAvailability` context. The compiler generates a minimal MapReduce program for every context declared as Map-

Reduce at design time. The type of input data for a generated MapReduce program is defined by the input format, which defaults to `TextInputFormat` (line 22). In our approach, sensor data is stored in the JSON format. In our case study, each presence status delivered to the application is converted to JSON and occupies precisely one line in the resulting dataset. Furthermore, each presence entry is defined by the timestamp of the event, device attributes (*i.e.,* id, parking lot) and the presence source. `TextInputFormat` fits such usage since it splits the input dataset to provide the Map function with one line of text (*i.e.,* one JSON entry) at a time. In a MapReduce program, any key or value type implements the `Writable` interface, which allows Hadoop to serialize objects for transmission over the network [20]. To facilitate the development of MapReduce programs, Hadoop already provides `Writable` wrapper classes for the majority of Java primitives (*e.g.,* `boolean` → `BooleanWritable`). In addition, developers may provide custom datatypes by defining classes implementing the `Writable` interface. At this stage, design declarations are of great importance since they allow the compiler to interpret key and value types of the resulting MapReduce program. For instance, as shown in Fig. 2, the `ParkingAvailability` context declares the output value type of the Map function as `Boolean` (line 4). As a result, the compiler matches the `Boolean` data type with the corresponding `BooleanWritable` wrapper class (Fig. 4, line 6). Moreover, an enumeration is interpreted as a string and matched with the `Text` wrapper class (Fig. 4, line 6). Finally, design declarations using complex data types result in the generation of a custom wrapper class, which implements the `Writable` interface and reflects the entire structure of the datatype.

4.2 Managing control flow

The control flow of the generated framework depends upon the declared interaction contracts between sensors (devices) and the application logic (contexts and controllers). In particular, the contexts include those declared as MapReduce jobs, which are automatically generated as shown above. Depending on the interaction contracts specified, the following scheduling strategies are chosen:

- If an "every" clause is specified with a period T, the corresponding context is invoked with this periodicity.
- Otherwise, if "when periodic" is specified with a period T, the context is invoked with this periodicity.
- Otherwise, if "when provided" is specified, the context is invoked any time the sensor produces a value.
- Otherwise, "when required" remains the only possible option. In this case, the context is invoked only when explicitly requested by a higher-level context declaring a "get" on the current context.

In our case study, the `ParkingAvailability` context declares that data must be gathered from presence sensors in a 10-minute time window, according to a periodic delivery model (Fig. 2, line 2). Data processing takes place when the time window elapses; that is, every 10 minutes, for our case study. At runtime, this job is executed with respect to the gathered sensed data and produces a result. The orchestrating application recovers the result, which is passed to the context via its callback method (*e.g.,* `onPeriodicPresence` for `ParkingAvailability`).

4.3 Managing the data flow

Managing the data flow in the generated framework involves (1) supplying data from sensors in suitable data structures, according to context declarations (*e.g.,* "group by" clauses), and (2) buffering data if needed, to interface between sensor delivery models and the scheduling strategies defined above.

The various clauses in context interaction contracts are processed as follows by the compiler:

- If "when required" is specified in a contract, there are no sensor values involved. Rather, such a contract declares that the value produced by the context is kept available for subsequent requests from higher-level contexts, or by controllers. There is no buffering of older values produced by the context: only the last value produced is available to client components. In such a contract, it is not possible to specify a "grouped by" clause, nor "every" or "map ... reduce".
- If "when provided" is specified, the context will receive all the values produced by the sensor or lower-level context; no value is lost.
- If "when periodic ... $<T>$" is specified, the context will receive values sampled with a periodicity of T from the sensor or lower-level context.
- If "grouped by a" is specified, data must come from a sensor device, and a must be one of the device attributes. In this case, the sensor values are indexed by the value of attribute a. If a "map ... reduce" clause is also specified, key-value pairs $<k, v>$ are supplied to the map phase, where k is the value of attribute a for the sensor that produced value v.

If no "map ... reduce" clause is specified, the context receives pairs <k, list(v)>, where the v values were produced by all the sensors whose attribute a is equal to k.

- If "every <T>" is specified, data must come from a sensor device. In this case, values from the sensors (gathered as specified by its event-driven or periodic delivery model) are accumulated during a period T and then passed together in a single context invocation.

In our case study, the `AverageOccupancy` context receives values sampled from all presence sensors every 10 minutes, indexed by parking lot, and accumulated over periods of 24 hours. Data processing takes place every 24 hours, and invokes a MapReduce job to efficiently cope with the size of the batched data.

4.4 Other data processing methods

Nowadays, the field of Big Data is attracting much attention from research and industry. The tool-development efforts devoted to dealing with rapidly emerging sources of big data result in an abundance of open-source projects [21]. Apache Hadoop is a widely-used tool to deal with large-scale datasets because it provides a reliable and scalable solution, maintained by a large community of developers. Hadoop is a batch-processing tool, typically used to analyze log files of large-scale systems, collected over a long period of time. The order of magnitude of these systems may range from hundreds of gigabytes to terabytes and, possibly petabytes. Apache Spark [22] is an alternative large-scale, data processing tool, which is gaining popularity due to its promise to outperform Hadoop by 10x [23]. Spark is an in-memory, data processing framework, which builds upon fault tolerant abstractions, manipulated using a rich set of operators, called Resilient Distributed Datasets (RDDs) [24]. In contrast with batch-processing tools, Apache Storm [25] primarily targets the processing of unbounded streams of data. Storm is an example of a CEP [26] system, where the data flow through a network of transformation entities. An application topology forms a directed acyclic graph, where stream sources (spouts) flow data to sinks (bolts); it implements a single transformation on the provided stream. In the context of large-scale orchestration, the power of batch-processing tools can be leveraged to analyze long-term datasets for trends in the usage of the city's infrastructure (*e.g.,* parking lots) and to identify structural degradation (*e.g.,* buildings, bridges). Stream processing tools, on the other hand, are best-suited to deal with high-frequency sensor readings, which typically involve tracking applications (*e.g.,* vehicle position, parking place availability). In the future, we intend to extend the parallel data-processing compiler to integrate both Spark and Storm, allowing developers to choose the right tool for their project.

5 Experimental Evaluation

To assess our approach, we have conducted a series of tests to examine the overall behavior of the MapReduce programming model for processing large amounts of sensor data. To do so, we developed a prototype of the parking management system, with Hadoop as the target platform, and analyzed the scalability of our approach using various datasets. In addition, we evaluated the design of the application and observed how specific design choices may impact the overall performance of an orchestrating application.

5.1 Experimental setup

The experimentation focuses on the average parking occupancy feature of our case study. The `AverageOccupancy` context processes sensor data synthesized for a 24-hour period, calculates the average occupancy of a parking lot, and notifies the parking manager via a `Messenger` device.

Machines. The experiment was carried out on a cluster of 12 nodes running within a private Eucalyptus [27] cloud. Each node in the cloud corresponds to a m2.xlarge type virtual machine instance with 2 CPUs, 2GB of RAM and 10GB of disk space. Every instance ran the DataStax Enterprise 4.6.1 [28] image, which is a big data platform leveraging tools such as Apache Hadoop and Apache Spark.

Datasets. We generated synthetic datasets to simulate a city's sensor infrastructure for the parking management system. Each dataset contains sensor data, indicating parking space occupancy, which is emitted every 10 minutes over 24 hours (*i.e.,* 144 measurements per sensor). We generated datasets for different sensor infrastructures, ranging from 10 000 to 200 000 sensors per dataset, thus testing the MapReduce program with datasets including up to 28 800 000 input records. The values for presence sensors are generated randomly, not according to any particular distribution. We did not attempt to simulate a realistic occupation of parking spaces, since the computation time in our prototype application is independent from the distributions of occupation times, and we focus here on evaluating just the scalability of the generated framework.

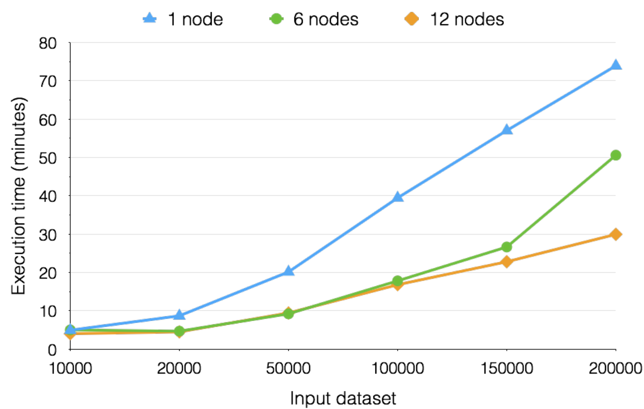

Fig. 6: Performance comparison between different cluster setups.

5.2 Experimental results

Scalability. Fig. 6 shows the performance of our parking management program. We compare its execution time with respect to 3 cluster setups – one, six and twelve nodes – and an increasing input dataset size. As can be expected, the execution time of the one-node setup increases the fastest, compared to the six and twelve node setups. The six and twelve node setups perform at par for the smallest dataset sizes (from 10 000 to 50 000 sensors) because their computing power is under-used. As the size of the datasets increases, the performance of these two setups gradually separate, showing better performance for the twelve-node setup. These preliminary results show that our compiler generates MapReduce implementations that attain expected scalability. Furthermore, these results demonstrate that declarations at the design level can benefit performance by driving compilation strategies, such as parallelization in our case study. This is achieved by introducing high-level insights (MapReduce constructs) in DiaSwarm.

Optimization through design. Beyond significantly improving the execution time of an orchestrating application, Hadoop opens up further optimization opportunities at the design level. For instance, in our case study, the `AverageOccupancy` context processes a dataset of presence values to produce the average occupancy of each parking lot for the last 24 hours. A closer look at the application design reveals that the computation provided by the `AverageOccupancy` context could be achieved by leveraging the computation of the `ParkingAvailability` context. The computed availability of parking spaces could thus be provided to the `AverageOccupancy` context at regular intervals, defined by the data delivery contract (*i.e.*, <10 min>)

```
1 context AverageOccupancy as ParkingOccupancy[] {
2   when provided ParkingAvailability // replaces line
        9, Fig. 2
3   grouped every <24 hr> // replaces line 10, Fig. 2
4   always publish;
5 }
```

Fig. 7: The ParkingAvailability context factorizing the computation performed by AverageOccupancy.

of the `ParkingAvailability` context. As a result, the `AverageOccupancy` context would use the provided data to calculate an average over the period of 24 hours.

The suggested design adjustments are depicted in Fig. 7. As can be noticed, the design of the application remains straightforward. More importantly, this design prevents sensor readings from being processed multiple times: the `AverageOccupancy` context factorizes the computations performed by the `ParkingAvailability` context. This caching strategy reduces the total time and resources the application requires for data processing. In fact, as shown in Fig. 7, the computation performed by the `AverageOccupancy` context no longer involves processing of a large dataset on a cluster (hence the MapReduce clause is omitted).

This major optimization also has a direct impact on application upkeep costs, since nowadays companies delegate processing of large datasets to cloud computing platforms (*e.g.*, Amazon Web Services) with a time-of-use pricing model.

6 Enabling New Services

The experimental results reported in the previous sections show that the integration of the MapReduce programming model in DiaSwarm and its Hadoop backend fulfil the promise of handling large amounts of data coming from massive sensor infrastructures. This section demonstrates on a concrete application scenario how the scalable data processing of our approach enables the development of new services involving personalization and community awareness. The example application is a community-aware extension of our parking management application.

The `ParkingManager` application described in Section 2 includes a context called `ParkingSuggestion` for displaying parking suggestions on city entrance panels. These suggestions are generic in that they are visible to all the drivers entering the city and contain a selection of parking lots having the best availability at a given moment. However, the added value of parking suggestions can be considerably enhanced by providing personalized suggestions to each driver entering the city

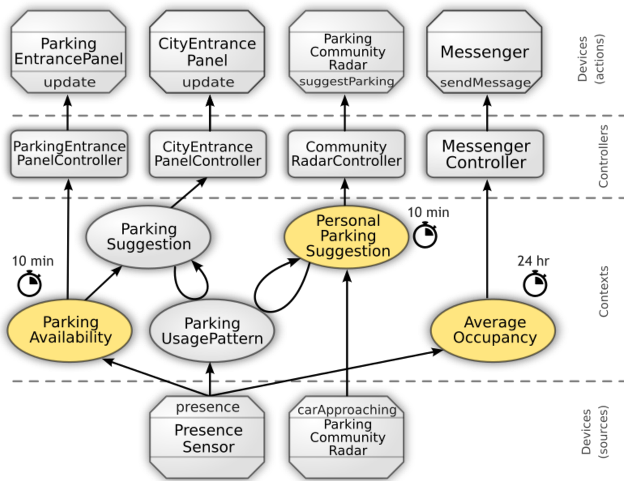

Fig. 8: The graphical view of the community-aware parking management application.

```
1  structure Expectation { carID as String;
       estArrivalTime as String; }
2  action SuggestParking {
3    suggestParking(carID as String, parkingLot as
         ParkingLotEnum);
4  }
5  device ParkingCommunityRadar {
6    attribute parkingLog as String;
7    source carApproaching as Expectation;
8    action SuggestParking;
9  }
10 context PersonalParkingSuggestion as Suggestion[] {
11   when provided carApproaching from
         ParkingCommunityRadar
12     grouped by parkingLot every <10 min>
13     with map as Expectation reduce as Demand
14   get ParkingUsagePattern
15   maybe publish;
16 }
```

Fig. 9: The `PersonalParkingSuggestion` context.

based on their intended destination and estimated time of arrival. This important improvement can be done by integrating the `ParkingManager` application with a community-based navigation system such as Waze[5] or Google Maps[6].

These community services can be integrated in DiaSwarm applications in the form of a software sensor device called `ParkingCommunityRadar`, defined in Figure 9. This device produces data of type `Expectation` when a trip is (re)computed in a community-based GPS navigation application used by a driver, or when the estimated time of arrival for an ongoing trip changes significantly. An instance of this software sensor is associated to each parking lot, as indicated by its `parkingLot` attribute. For privacy reasons, the destination reported to the `ParkingCommunityRadar` is not the ultimate destination of the driver (*e.g.,* precise GPS coordinates), but rather the intended parking lot for leaving the car. The device also features a `SuggestParking` action which allows the application to send a personal parking suggestion to a specific car driver. Drivers are identified based on the user identifier in the navigation application, or based on the license plate number, for instance.

Using this software sensor, personalized suggestions can be sent to drivers by adding the `PersonalParking-Suggestion` context to the `ParkingManager` application, as shown in Figure 8. The new context is declared in Figure 9.

The `PersonalParkingSuggestion` context receives data from all the ongoing trips and sends personal parking suggestions only when the intended parking lot is predicted to be full at the estimated time of arrival. In this case, an alternative nearby parking is suggested for which better availability is predicted. The availability predictions reuse the ParkingUsagePattern context described in Section 2, which contains a predictive model based on the parking usage history. We assume that this model provides predictions at a granularity of 10 minutes, but coarser-grained models could be accommodated using interpolation.

The interaction contract of this context specifies that destinations are collected in an event-driven model, but processed only every 10 minutes. This allows to check the future availability of the parking lots (according to the predictive model) by cumulating parking requests for the near future, from ongoing trips. Thus, the destinations accumulated over the last 10 minutes are first partitioned, according to the estimated time of arrival in future intervals of 10 minutes. For each such interval, the predicted availability of each parking lot is compared to the number of estimated arrivals; if the result shows a shortage of available parking spaces, cars in exces are sent an alternative parking suggestion. The cars to be redirected are the ones with the latest estimated time of arrival. Indeed, assuming that the estimated times and the predicted availabilities are accurate, the redirected cars are precisely those that will find a saturated parking lot. The other cars are not notified in any way, saving the attentional resources of the drivers.

According to the declarations in the context, a MapReduce job is scheduled every 10 minutes to cope with a large potential number of ongoing trips, while ensuring timely notifications for possible redirections. The map phase rounds estimated arrival times to the closest 10-minute interval. The reduce phase computes the cumu-

[5] https://www.waze.com
[6] https://www.google.com/maps/about/

lated demand on a parking lot for each time slice. Based on the MapReduce result, the context compares the demand with the predicted availability for each parking lot to produce a (possibly empty) list of parking redirection suggestions. The output of the context is published to the `CommunityRadarController` component, which sends the corresponding notifications to the concerned drivers via the `ParkingCommunityRadar` devices of the overloaded parking lots.

This enhanced version of the parking management application provides a personalized service to drivers, preventing parking lots to be overloaded. Detection of potential parking overload is based on community data of ongoing trips, provided by drivers accepting to communicate their destinations in exchange for more reliable guidance to find available parking spaces. However, this feature requires efficient computation support for large data, ensured in our approach by MapReduce design annotations.

7 Extensions

The presentation of our approach and its evaluation show that we offer a convenient and efficient way to express large-scale computations over massive sensor infrastructures, at the scale of the Internet of Things. Furthermore, this processing power enables more personalized services to be delivered to users, without compromising service responsiveness. This section takes a step back to address the current limitations of DiaSwarm in terms of expressiveness. We review the underlying language design rationale, and introduce language extensions to lift these limitations. Some of these extensions are implemented and released in our available prototype.[7]

7.1 Specifying intermediate keys

As shown in Section 3, the MapReduce programming model is exposed at a high level of abstraction to developers. Indeed, they only have to declare that a context such as `ParkingAvailability` is to be implemented in the MapReduce model and provide the types of values produced by the map and reduce phases (as in Figure 2). Then, the developers specify the map and reduce computations (as in Figure 3), but are abstracted away from implementation issues, specific to a MapReduce backend engine, such as implementing and configuring Hadoop jobs.

However, this high level of abstraction currently hides not only implementation details of the backend, but also some of the power of the MapReduce model itself. Specifically, the MapReduce model allows to use two different sets of keys for indexing data in the map and reduce phases, as shown in Figure 10. Currently, the DiaSwarm language allows to specify only a single key set — an attribute of sensor devices—, to group the incoming data and its processing, along the map and reduce phases. This allows to use natural partitionings of sensors, according to attributes such as their location. We have chosen this strategy as it fits the major common case of large-scale sensing, while imposing minimal effort on developers. Nevertheless, more complex MapReduce computations could be specified if DiaSwarm allowed to specify a different key for the intermediate values, so as to exploit parallelism along another partitioning dimension.

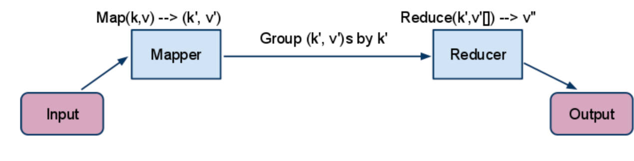

Fig. 10: The general MapReduce programming model.

For instance, an application for computing a real-time histogram of the target temperatures in all the homes in a city equipped with a HVAC (Heating, Ventilation and Air-Conditioning) system could group the readings geographically by street blocks for the map phase, but would use the discrete temperature value (an integral number of Celsius or Fahrenheit degrees) as a key in the reduce phase to compute the frequency of each value.

To cope with this limitation of DiaSwarm we extended the language syntax to allow specifying the intermediate key of the reduce phase. These keys could be used in the generated programming framework with very small changes of the Hadoop backend.

7.2 Parallelizing higher-level contexts

DiaSwarm allows designing MapReduce contexts taking the input from many instances of a sensor device, such as the `ParkingAvailability` context in Figure 2 taking its input from a large set of `PresenceSensor` device instances. Currently, it is not possible to specify a map/reduce clause in DiaSwarm for a higher-level context, that is, one which takes input from another context, rather than from a sensor device. Indeed, declaring a MapReduce context requires grouping incoming data by a sensor device attribute, which does not exist when data originates from a lower-level context. This choice

[7] http://phoenix.inria.fr/software/diaswarm

has been taken during the design of the DiaSwarm language for two reasons:

- contexts directly connected to a device produce refined information that is usually much more concise than the large raw dataset captured by sensors (this semantics is also implied by the "reduce" phase of a MapReduce context) — as a consequence, higher-level context tend to operate on smaller datasets;
- the number of actuators in an application is typically much smaller than the number of sensors, eliminating another potential need for computing large datasets in higher-level contexts (such a large dataset would be necessary if a massive amount of actuator instances would have to be actuated differently by the application).

However, in some complex applications, the first assumption may be violated, as some lower-level contexts may also produce large datasets – perhaps not computed by a MapReduce, but rather accumulating some form of big data that cannot be summarized without loosing its predictive value. Furthermore, the second assumption may not be fulfilled in applications involving a large number of actuators – this is indeed a natural trend in personalized applications. When either situation arises, it could make it worth to also parallelize some of the higher-level contexts in an application.

For instance, if the parking suggestions of the ParkingManager application in Figure 1 would have to be shown, not only at city entrances, but also on many public displays all over the city, these suggestions would have to be localized with respect to nearby parking lots and to their predicted usage patterns. In this situation, it would be a stringent requirement to parallelize the `ParkingSuggestion` context along parking lots or public displays, for example.

This current limitation could be removed by allowing to group the output of array-valued contexts using for instance a field of the indexed data structure (in our case, the parking lot field in the array entries produced by the `ParkingAvailability` context). Also, allowing to specify a map/reduce clause on a controller component could also open up many possibilities. This latter feature would require to allow grouping actuator devices along one of their attributes, similar to how sensors are currently handled in DiaSwarm.

7.3 Grouping by a computed attribute

The fact that sensor readings can only be grouped by a sensor device attribute also implies other limitations in the practical applicability of current DiaSwarm language. More precisely, grouping values by a device attribute works well when this attribute is of a large enumerated type, which is the case in many smart city applications, such as the list of parking lots in a city, or offices in a building, *etc.* However, other cases may not fit with this model. For instance, a particular component of a device attribute (such as the city code in the licence plate number of a car) could be more suited to cluster computations on sensor readings. Also, when mobile sensor are used, with their current location expressed as GPS coordinates, it makes more sense to cluster the readings with respect to regions delimited as GPS intervals, instead of precise GPS positions.

This limitation could be removed by allowing in a grouped-by construct, not only a device attribute name, but also a general expression involving a device attribute, such as `grouped by department(plateNo)` or `grouped by region(gpsLocation)`.

```
1  device CoSensor {
2    attribute parkingLot as ParkingLotEnum;
3    source pollution as Float;
4  }
5  context AverageCarPolution as Float[] {
6    when periodic presence from PresenceSensor <10 min>
7      grouped by parkingLot every <24 hr>
8      with map as Presence reduce as Integer
9      get pollution from CoSensor
10   always publish;
11 }
```

Fig. 11: Current syntax of DiaSwarm for grouping multi-sensor measurements.

7.4 Grouping heterogeneous sensor readings

In DiaSwarm, the MapReduce computation declared by a context can only process uniform sensor data, that is, originating from all the different instances of a sensor device. The context may of course get values from other "secondary" sensor devices, but these values are only taken into account after the map and reduce phases, for computing the final value produced by the context. For instance, Figure 11 shows a context computing the average pollution produced by a car in a parking lot, dividing the pollution sensed with a CO sensor at the end of the day by the average number of cars present during that day (see line 9). The pollution value in this case is not available to the MapReduce computation, but will only be passed as an extra argument to the `onPeriodicPresence` method in Figure 3.

Some applications could benefit from performing parallel computations on heterogeneous sensor data. For

instance, a more accurate way to compute the average pollution in a parking lot would be to average all the instant pollution values. This would require sensing the CO level at the same time as when occupancy is computed (*i.e.,* every 10 minutes), accumulating these simultaneous values over the computation period (24 hours), compute all the instant pollution values in the map phase, and compute their average in the reduce phase.

Removing this limitation would involve extending the syntax of DiaSwarm to allow specifying the "get" clause before the "grouped by" clause, thus expressing that these sensor values should be grouped together. It would also require that the generated programming framework samples the value of each secondary sensor each time a primary sensor is sampled, so as to accumulate complete snapshots of heterogeneous sensor readings.

7.5 Optional map and reduce clauses

When a context is declared as being computed with a MapReduce operator, the developer always has to specify a map and a reduce user-defined methods. However, in many common cases, either the map or the reduce are the identity function. This is not really a limitation of the current DiaSwarm language, because it does not forbid to implement such applications. Rather, some unneeded effort can be avoided by making both the map and reduce clauses optional, with the effect of automatically generating identity transformation. To do so, we extended DiaSwarm to allow either phase to be omitted in the declaration.

8 Discussion

This section discusses the applicability of our approach to other scenarios, and some of its current limitations.

8.1 Applicability

The ParkingManager application used as our case study allowed us to illustrate the approach and its use on a concrete scenario. We deliberately chose an example involving straightforward computations for clarity reasons, enabling to show (1) concise and easy to understand snippets of the generated framework (Fig. 4) and (2) placeholders to be filled by developers (Fig. 3). Of course, computing parking availability by summing up unused spaces is not a compelling application for parallel processing. The extension to a personalized service discussed in Section 6 gave a more accurate view of realistic data-intensive applications. Real personalized services related to parkings which have been proposed include dynamic pricing schemes, where prices are continuously computed and updated with respect to sensing data streamed from the infrastructure, in order to optimize parking usage [29]. The same kind of dynamic pricing has been used for other scarce resources in smart cities such as shared mobility systems (pools of shared electric vehicles or bikes) in order to optimize their geographical redistribution [30]. These are concrete examples of real-time computations based on massive sensor deployments that could benefit from our domain-specific parallelization approach.

More data-intensive smart city services needing parallel computing will gradually appear following massive deployments of sensors and actuators, for such domains as traffic, weather, or energy consumption monitoring.

8.2 Limitations

One limitation of our approach is that it does not address the physical placement of computations on network nodes. In our current implementation, all computations are performed in the cloud, after data are gathered from all the sensors necessary to a given context (as specified in a "when provided/periodic" clause). The possibilities available in many sensor networks for in-network processing or distributed computing are not currently exploited. In principle, the graph of application components might be used to automatically push in the network some contexts that are "close" to sensors. Another approach would consist of extending DiaSwarm to allow explicit hints to be formulated in a declarative way. Optimizations of the underlying sensor network, such as optimized routing, could be triggered at different phases, *e.g.,* at deployment time or during runtime. We have proposed a vision including such ideas elsewhere [31], but they are not currently implemented in our prototype.

9 Related Work

In this section, we examine existing approaches that address the development of applications orchestrating sensors. We consider approaches from domains where orchestration of sensors is a common concern. Furthermore, we highlight the differences between our approach and large-scale data processing support.

Internet of Things (IoT). Patel *et al.* propose a multi-stage, model-driven approach, dedicated to the development of IoT applications [32]. This approach provides support at different stages of the development

process. At design time, the approach offers a set of customizable modeling languages for the specification of an application. The approach is complemented by code generation and task-mapping techniques for the deployment of node-level code onto devices. Even though this approach is aimed to facilitate the development process through guidance, Patel *et al.* do not provide details regarding the size of sensed data that are gathered and processed. They do not discuss what support is generated to facilitate the programming process. This approach does not address how masses of sensors are handled, nor does it present performance measurements to assess how it scales up for large datasets.

Pervasive computing. The domain of pervasive computing offers a number of approaches targeting the development of orchestrating applications. PervML [33] is a model-driven development approach that provides a conceptual framework for context-aware applications. The various aspects of a pervasive computing application are modeled by different types of UML diagrams. Dey *et al.* propose the Context Toolkit [34] that provides the programmer with building blocks to mediate between the contextual aspects of the environment and the application. Olympus goes beyond middleware in providing a programming framework dedicated to the development of pervasive computing systems [35]. Because it is based on a domain-specific framework, Olympus raises the level of abstraction and facilitates the development of applications. DiaSuite takes these approaches further by introducing a design language dedicated to the Sense/Compute/Control paradigm [36,37]. A design is used to generate a dedicated programming framework that guides, restricts, and supports the implementation phase.

All the above-mentioned approaches have been designed for the orchestration of objects in the small (*i.e.,* offices, buildings, *etc.*). They do not address challenges arising with large-scale infrastructures and do not provide strategies to tackle data-intensive processing.

Wireless sensor networks (WSN). Gupta *et al.* propose sMapReduce [38], a programming pattern inspired by the MapReduce programming model for mapping application behavior onto a sensor network and enabling complex data aggregation. sMapReduce divides the network-level user program into sMap and Reduce functions; this strategy respectively associates a behavior to sensor nodes and executes data aggregation over the network. Compared to our approach, sMapReduce remains lower-level since it provides network-level programming abstractions and introduces the network topology in computations.

Often, programming applications for WSNs is done at a low level, requiring the developer to have extensive knowledge about the underlying layers (network, hardware, OS). Mottola and Picco [39] surveyed a number of programming approaches for WSNs aimed to facilitate the programming of layers underlying applications; these approaches target sensor nodes, communication operations, routing strategies, *etc.* These works are complementary to ours in that they provide high-level abstractions that can be used by our compiler to target frameworks for WSNs. However, they do not provide support dedicated to dealing with large datasets produced from massive-scale sensor infrastructures.

Large-scale data processing. Apache Pig [9] and Apache Hive [10] are widely used as high-level platforms for analyzing large-scale datasets. These platforms provide SQL-like declarative query languages (*i.e.,* PigLatin & HiveQL) to express data analysis programs. These tools are well-suited for offline data analysis, but require some effort for running scripts from application code (*e.g.,* setting up a connection with a JDBC server). Sawzall [40] used by Google is a high-level scripting language for automating analyses on large data sets on top of the MapReduce execution model. Sawzall is not publicly available but is reported to improve the programming significantly, compared to C++ programming of MapReduce. High-level language libraries, such as FlumeJava [11], provide high-level abstractions dedicated to parallel processing; they provide support for user-defined functions, compared to SQL-like approaches.

Compared to the above-mentioned supports, our approach integrates, at the design level, two domain-specific fundamental dimensions: large-scale orchestration of sensors and large-scale data processing. The integrated nature of our approach allows developers to easily combine results from various computations. The design-driven nature of our approach is supported by high-level declarations, exposing such domain-specific information as service discovery and data delivery. Declarations are analyzed to determine data and control flow information, which in turn, is used to generate efficient, parallel-data processing frameworks.

10 Conclusion

We have proposed a design-driven approach to developing orchestrating applications for masses of sensors that integrates parallel processing of large amounts of sensed data. Our new approach provides the developer with design declarations expressing when and where data processing occurs. A compiler takes an application design

as input and produces a programming framework based on the MapReduce programming model. The generated framework supports and guides the programming of the orchestration logic, while abstracting over the parallel processing of sensed data.

We have demonstrated that our approach creates synergy between design and programming, allowing seamless introduction of high-performance computing strategies, as illustrated by the MapReduce programming model. We illustrated our approach with a case study of a parking management system. This case study was used to conduct an experiment on Apache Hadoop, demonstrating how our design-driven approach can be leveraged to parallelize the processing of large datasets and obtain significant speedups.

In the future, we intend to support the processing of unbounded streams of data, typical of sensors. Our declarative approach will allow us to design orchestrating applications that mix the processing of both large datasets and unbounded data streams, allowing us to abstract away these aspects.

References

1. S. Garcia-Ayllon and J. L. Miralles, "New strategies to improve governance in territorial management: Evolving from smart cities to smart territories," *Procedia Engineering*, vol. 118, pp. 3 – 11, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877705815020512
2. Y. Mizuno and N. Odake, "Current Status of Smart Systems and Case Studies of Privacy Protection Platform for Smart City in Japan," in *2015 Portland International Conference on Management of Engineering and Technology (PICMET)*, Aug 2015, pp. 612–624.
3. M. Naphade, G. Banavar, C. Harrison, J. Paraszczak, and R. Morris, "Smarter Cities and Their Innovation Challenges," *Computer*, vol. 44, no. 6, pp. 32–39, June 2011.
4. Libelium, "Smart City project in Santander to monitor Parking Free Slots," March 2016. [Online]. Available: http://www.libelium.com/smart_santander_parking_smart_city.
5. Worldsensing SL, "Worldsensing and SIGFOX announce the world's largest Intelligent Parking deployment with Micronet, the SIGFOX Network Operator for Russia," March 2016. [Online]. Available: http://www.worldsensing.com/news-press/press-release-worldsensing-and-sigfox-announce-the-worlds-largest-intelligent-parking-deployment-with-micronet-the-sigfox-network-operator-for-russia.html.
6. K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey," *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, December 2012.
7. C. Patsakis, R. Venanzi, P. Bellavista, A. Solanas, and M. Bouroche, "Personalized medical services using smart cities' infrastructures," in *Proceeding of the 2014 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*. IEEE, 2014, pp. 665–669.
8. R. Seeliger, C. Krauss, A. Wilson, M. Zwicklbauer, and S. Arbanowski, "Towards personalized smart city guide services in future internet environments," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15 Companion. New York, NY, USA: ACM, 2015, pp. 563–568. [Online]. Available: http://doi.acm.org/10.1145/2740908.2743905
9. The Apache Software Foundation, "Apache Pig," March 2016. [Online]. Available: https://pig.apache.org.
10. The Apache Software Foundation , "Apache Hive," March 2016. [Online]. Available: https://hive.apache.org.
11. C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, Efficient Data-Parallel Pipelines," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, 2010, pp. 363–375.
12. M. Kabáč and C. Consel, "Orchestrating Masses of Sensors: A Design-Driven Development Approach," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. ACM, 2015, pp. 117–120.
13. M. Kabáč and C. Consel, "Designing Parallel Data Processing for Large-Scale Sensor Orchestration," in *13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC 2016)*, Toulouse, France, Jul. 2016, best Paper Award. [Online]. Available: https://hal.inria.fr/hal-01319730
14. R. Lämmel, "Google's MapReduce programming model - Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, Oct. 2008.
15. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
16. R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.
17. M. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks," *Commun. ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997.
18. S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, "A Taxonomy of Wireless Micro-Sensor Network Models," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 2, pp. 28–36, Apr. 2002.
19. The Apache Software Foundation, "Hadoop Wiki PoweredBy," March 2016. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy.
20. T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.
21. The Apache Software Foundation, "Projects Directory," March 2016. [Online]. Available: https://projects.apache.org/projects.html?category#big-data.
22. The Apache Software Foundation , "Apache Spark," March 2016. [Online]. Available: http://spark.apache.org.
23. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, 2010, pp. 10–10.
24. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 2012, pp. 2–2.
25. The Apache Software Foundation, "Apache Storm," March 2016. [Online]. Available: http://storm.apache.org.
26. G. Cugola and A. Margara, "Processing Flows of Information: From Data Stream to Complex Event Processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

27. Hewlett-Packard, "HP Helion Eucalyptus," March 2016. [Online]. Available: http://www.eucalyptus.com.
28. DataStax, "DataStax Enterprise," March 2016. [Online]. Available: http://www.datastax.com.
29. D. Mackowski, Y. Bai, and Y. Ouyang, "Parking Space Management via Dynamic Performance-Based Pricing," *Transportation Research Procedia*, vol. 7, pp. 170–191, 2015.
30. J. Pfrommera, J. Warrington, G. Schildbach, and M. Morari, "Dynamic Vehicle Redistribution and Online Price Incentives in Shared Mobility Systems," *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*, vol. 15, no. 4, AUGUST 2014.
31. M. Kabáč, C. Consel, and N. Volanschi, "Leveraging Declarations over the Lifecycle of Large-Scale Sensor Applications," in *13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC 2016)*, Toulouse, France, Jul. 2016. [Online]. Available: https://hal.inria.fr/hal-01319731
32. P. Patel, A. Pathak, D. Cassou, and V. Issarny, "Enabling High-Level Application Development in the Internet of Things," in *S-CUBE'13: 4th International Conference on Sensor Systems and Software*, Jun. 2013.
33. E. Serral, P. Valderas, and V. Pelechano, "Towards the Model Driven Development of Context-aware Pervasive Systems," *Pervasive Mob. Comput.*, vol. 6, no. 2, pp. 254–280, Apr. 2010.
34. A. K. Dey, G. D. Abowd, and D. Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Hum.-Comput. Interact.*, vol. 16, no. 2, pp. 97–166, Dec. 2001.
35. A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas, "Olympus: A High-Level Programming Model for Pervasive Computing Environments," in *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PERCOM '05)*. IEEE Computer Society, March 2005, pp. 7–16.
36. D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, 2011, pp. 431–440.
37. B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Balland, and C. Consel, "DiaSuite: a Tool Suite To Develop Sense/Compute/Control Applications," *Science of Computer Programming*, vol. 79, pp. 39–51, Jan. 2014.
38. V. Gupta, E. Tovar, L. M. Pinho, J. Kim, K. Lakshmanan, and R. Rajkumar, "sMapReduce: A Programming Pattern for Wireless Sensor Networks," in *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications (SESENA '11)*. ACM, 2011, pp. 37–42.
39. L. Mottola and G. P. Picco, "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, 2011.
40. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, Oct. 2005.

A DiaSwarm grammar

```
DomainModel =
               (IncludeSpec
               | ActionDef
               | AbstractElement)*;

AbstractElement =
               StructDef
               | EnumDef
               | DeviceDef
               | ContextDef
               | ControllerDef;

IncludeSpec = 'include' STRING;

StructDef = 'structure' ID '{' (StructFieldDef ';')+
     '}';

StructFieldDef = ID as DataTypeRef;

EnumDef = 'enumeration' ID '{' ID (',' ID)* '}';

VariableDef = ID as DataTypeRef;

DataTypeRef =
               (StructDef
               | EnumDef
               | PrimitiveTypeRef) (ListTag)?;

ListTag = '[]';

PrimitiveTypeRef =
                 'Integer'
               | 'Boolean'
               | 'String'
               | 'Float'
               | 'Binary';

DeviceDef = 'device' ID ('extends' DeviceDef)?
'{' (AttributeDef | SourceDef | ActionImpl)* '}';

AttributeDef = 'attribute' ID 'as' DataTypeRef ';' ;

SourceDef = 'source' ID 'as' DataTypeRef
('indexed' 'by' VariableDef (',' VariableDef)*)? ';'
     ;

ActionImpl = 'action' ActionDef ';' ;

ActionDef = 'action' ID '{' (OrderDef)+ '}';

OrderDef = ID '('(VariableDef(',' VariableDef)*)?')'
     ';' ;

ContextDef = 'context' ID 'as' DataTypeRef
('indexed' 'by' VariableDef (',' VariableDef)*)?
       '{' (InteractionContractDef)* '}';

InteractionContractDef =
       'when'
         ('required' (PullInteractionContract)?
       | ('provided' PushInteractionContract
            BehaviorPublication)
       | ('periodic' PushInteractionContract
            BehaviorPublication)) ';';

PullInteractionContract = ('get'
               (SourceRef | ContextDef))?
               (',' (SourceRef | ContextDef))*;

PushInteractionContract = DataRequirement
               (PullInteractionContract)?;

DataRequirement = DataSource
    ('grouped' 'by' AttributeDef
    ('every' Periodicity)?)?
    ('with'
      (('map' 'as' (DataTypeRef | '<'DataTypeRef ','
                                  DataTypeRef '>'))?
    &
      ('reduce' 'as' DataTypeRef)?))?;

DataSource = SourceRef | ContextDef (Periodicity)?;

SourceRef = SourceDef 'from' DeviceDef;

Periodicity = '<' INT TimeUnit '>';

BehaviorPublication = 'always' 'publish'
                       | 'no' 'publish'
                       | 'maybe' 'publish';

TimeUnit = 'hr' | 'min' | 's';

ControllerDef = 'controller' ID
               '{' (ControllerBehaviorDef)* '}' ;

ControllerBehaviorDef = 'when' 'provided' ContextDef
               ('get' ContextDef (',' ContextDef)*)?
               'do' ActionRef (',' ActionRef)* ';' ;

ActionRef = ActionDef 'on' DeviceDef;

ID = '^'?('a'..'z'|'A'..'Z'|'_')
         ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

INT = ('0'..'9')+;

STRING =
  '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u
      '|'"'|"'"|'\\' */
    | !('\\'|'"') )* '"' |
  "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u
      '|'"'|"'"|'\\' */
    | !('\\'|"'") )* "'";
```