2017 IEEE 41st Annual Computer Software and Applications Conference

A High-Level DSL for Geospatial Visualizations with Multi-core Parallelism Support

Cleverson Ledur, Dalvan Griebler, Isabel Manssour, Luiz Gustavo Fernandes
PUCRS, Faculdade de Informática, Computer Science Graduate Program,
Fax: +555133203621, Av. Ipiranga, 6681, CEP: 90619-900 – Porto Alegre – Brazil
Email:{cleverson.ledur, dalvan.griebler}@acad.pucrs.br, {isabel.manssour, luiz.fernandes}@pucrs.br

***Abstract*—The amount of data generated worldwide associated with geolocalization has exponentially increased over the last decade due to social networks, population demographics, and the popularization of Global Positioning Systems. Several methods for geovisualization have already been developed, but many of them are focused on a specific application or require learning a variety of tools and programming languages. It becomes even more difficult when users have to manage a large amount of data because state-of-the-art alternatives require the use of third-party pre-processing tools. We present a novel Domain-Specific Language (DSL), which focuses on large data geovisualizations. Through a compiler, we support automatic visualization generations and data pre-processing. The system takes advantage of multi-core parallelism to speed-up data pre-processing abstractly. Our experiments were designated to highlight the programming effort and performance of our DSL. The results have shown a considerable programming effort reduction and efficient parallelism support with respect to the sequential version.**

I. Introduction

Recently, big data has become of global interest in the industries, academic institutions, and governments. In this context, geo-referenced data has exponentially increased. The McKinsey Global Institute presented that about 1 PByte of geospatial data was produced in 2009 and predicted a 20% annual growth. Also, 2.5 Exabytes of data is being generated every day, with a significant percentage of spatial information [1]. For example, Google produces about 25 PBytes of data each day, the majority containing geospatial information in pictures, videos, and maps [2].

Many visualization techniques are applied to geospatial information using Geographic Information Systems (GIS), programming libraries, and frameworks. Geospatial data visualization can be useful for many things such as learning about human behavior and avoiding traffic jams through mobile applications [3]. Among the techniques of geospatial data analysis, data visualization can help domain users to quickly gain insights [4]. The tools available presently do not provide the necessary abstractions in the pre-processing step of the visualization pipeline [5]. Thus, even though data visualization offers many benefits, its generation is still remains a challenge [6]. Domain users have difficulties dealing with a large amount of data, since it demands high costs and a great deal of programming effort to process and manipulate the raw data.

Pre-processing large amounts of data is a performance problem in the related works as well (Section II). Parallel computing may help to accelerate processing because it enables applications, with well implemented parallel programming techniques, to take advantage of parallel architectures [7]. Nevertheless, this is a difficult task, which requires specialized skills regarding tools, methodologies, and modeling [8].

Our goal is to improve the geospatial visualization creation experience by reducing the effort required for pre-processing data and visualization implementation. Therefore, we implemented a DSL to abstract complexities and let users focus on gaining quick insights in the geospatial data domain. Our DSL implements three modules to provide these benefits: a code analyzer, data pre-processor, and visualization generator. These modules abstract the complexities of all phases in the visualization creation. We proposed our high-level description language in [9], introducing our idea and the main problem. In this paper, we extended the language and implemented its multi-language code generation technique. Also, we provide two novel visualization types, a compiler, and support to automatic parallelism exploitation for multi-core systems.

The paper presents the following contributions:

- DSL-based approach implementation and its compiler for abstracting all geospatial visualization steps;
- An efficient insertion of parallelism annotations in the data pre-processor module that enables us to completely abstract parallelism aspects from the DSL's users;
- A set of experiments with different visualization types and real world data sets with respect to [9].

The paper is organized as follows. Section II presents related work. Section III introduces the DSL's Language. Section IV describes the DSL's compiler implementation. Section V presents the strategy used to support parallelism in multi-core systems. The experiments, results, and discussion are described in Section VI. Section VII concludes this paper.

II. Related Work

There are visualization frameworks and tools that provide different visualization techniques for geospatial data. For example, GeoSpy [10] is a Web platform for geospatial visualization that enables the creation of maps using less broadband than traditional APIs. Titan [11], which provides data ingestion, processing, and visualization, and ParaView [12], which allows scientists to visualize and analyze large data sets, are limited to volumetric and scientific visualization types that are offered by VTK.

Protovis [13] is an extensive graphical toolkit that offers a high-level abstraction for creating general visualization, but does not provide the creation of maps or data pre-processing

0730-3157/17 $31.00 © 2017 IEEE
DOI 10.1109/COMPSAC.2017.18

IEEE computer society

abstraction. SksOpen [14], e.g., is a system that enables efficient indexing, querying, and visualization of geospatial data. However, it differs from our DSL since it has a limited number of TerraFly visualizations, focusing on online data processing and does not working locally with big data sets.

Some DSLs have enabled high-level and easy specification for users. Vivaldi [15] provides volumetric processing and visualization rendering through functions and mathematical operators used in scientific visualization. Another DSL is ViSlang [16], which is developed to enable the creation of scientific visualizations. Its main contribution is allowing the DSL to be extended using *slangs*. Diderot [17] simplifies the implementation of parallel methods of biomedical image analysis and visualization. Shadie [18] is a GPU-based volume visualization DSL built around the concept of shaders: self-contained descriptions of the desired visualization written in a high-level Python-like language.

None of these DSLs (Vivaldi, ViSlang, Diderot, or Shadie) support the generation of geospatial data visualization. On the other hand, Superconductor [19] is a high-level language for creating visualizations that aims to interact with large amounts of data. It can offer geospatial data visualization and allows for the use of the D3 library. However, users have to build their visualizations from the scratch, specifying how data will be mapped to visual representations. Our DSL facilitates the creation of geospatial data visualization by providing a high-level description language that abstracts computer programming aspects such as variables, functions, data pre-processing, and parallel programming.

III. GMaVis in a Nutshell

With the goal of simplifying the creation of visualizations for large-scale geospatial data, we propose an external DSL that provides a high-level specification language called GMaVis[1]. It enables users to express filter, classification, data format, and visualization specifications. GMaVis has limited expressiveness to reduce complexity and automatize decisions and operations such as data pre-processing, visualization zoom, and starting point location. Thus, it is easy and quick for users to learn and use.

We used three real-world data sets to present the DSL grammar through examples. The first was the YFCC100M data set. It was provided by Yahoo Labs [20] and has about 54GB of data, divided into ten files. It is a public multimedia data set with 99.3 million images and 0.7 million videos, all from Flickr and under Creative Commons licensing. The second data set is about traffic accidents occurred in the city of Porto Alegre, Brazil. It contains information about accident types, vehicles involved, weather, date, time, severity, and location. Finally, we used a data set with data about all the airports in the world, with information about latitude and longitude, city, country, and airport code.

Our first example is Listing 1 and its output is illustrated in Figure 1(a). The first statement in line 1 is a **`visualization:`** declaration, where the visualization type chosen to appear on the visualization is `clusteredmap`. In addition, we also provide the implementation of `heatmap` and `markedmap`. In Line 2 of Listing 1, the block **`settings`** begin, which contains the declarations used to specify details of the visual aspects. Also, it receives the fields where important attributes are located, such as latitude and longitude. Lines 3 and 4 are being declare **`latitude`** and **`longitude`** to inform the field values. This specifies the position in the data set where this information can be found. Line 5 has a **`marker-text`** to be display as a text on the marker when the user clicks on it. The **`page-title`** declaration in Line 6 of Listing 1 informs the title which will be placed in the visualization. In line 7 a `size` declaration is used to set the visualization size it will occupy on the web page. This declaration can also be the `medium` and `small` values.

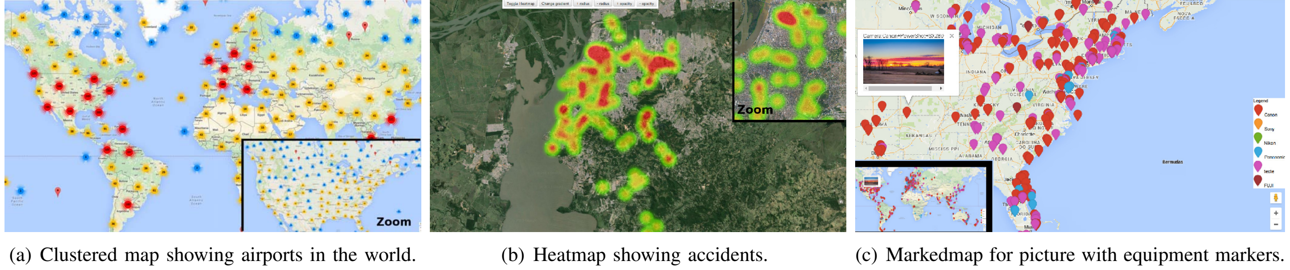
There is also a **`data`** block from Line 9 to 15. It contains declarations that specify the input files and filters. Users can also include sub-blocks for data structure and classification. In our example, an input file is declared in Line 10, receiving a string with the system path to the file. This declaration can repeat as many times as necessary to include the whole data set. Furthermore, a **`structure`** sub-block is declared in Line 11. It has a **`delimiter`** and an **`end-register`** declaration specifying that a comma separates the values in the input data set and that a newline character separates the registers. This declaration can receive any character or the defined keywords `tab`, `comma`, `semicolon` and `newline`.

```
1 visualization: clusteredmap;
2 settings {
3   latitude: field 7;
4   longitude: field 8;
5   marker-text: field 1 field 2;
6   page-title: "Airports in World";
7   size: full;
8 }
9 data {
10   file: "vis_codes/airports.data";
11   structure {
12     delimiter: ',';
13     end-register: newline;
14   }
15 }
```

Listing 1. GMaVis code for Clusteredmap (the result is in Figure 1(a)).

Figure 1(b) illustrates a heatmap that shows traffic accidents in the city of Porto Alegre, Brazil. To generate this visualization from Listing 1, we added in the **`settings`** block a **`zoom-level`** declaration, defining where the visualization starts using a greater zoom for the center point. This is a great advantage with respect to Google Maps API, because GMaVis automatically gives the initial central point visualization (latitude and longitude). Moreover, as this visualization aims to show only accidents where people were killed or injured, a different file has to be provided (in line 10 of Listing 1) as well as a filter must be specified after the **`structure`** block to get only the right data. This specification receives logical operations to apply over each data element. When true, the data element is inserted in the output visualization. The logical operators enable users to create logical operations, used in both filters and classification. For our example, we used operator `is greater than` twice. GMaVis also has `or` and `and` operators to merge the two logical operations.

[1]https://gmap.pucrs.br/gmavis/



(a) Clustered map showing airports in the world. (b) Heatmap showing accidents. (c) Markedmap for picture with equipment markers.

Fig. 1. Geopeatial visualization generations with GMaVis.

The `markedmap` visualization is illustrated in Figure 1(c) and created through the code in Listing 2. This visualization uses Yahoo Flickr data.We used **`marker-text`** declaration by combining strings and fields values to insert the picture into the box of each marker. As field 15 contains the URL of the image, we inserted the `image` keyword to generate the source link that will show the picture when users select a marker, and Figure 1(c) shows how it looks. This visualization has a **`date-format`** declaration in Line 14, inside the **`structure`** sub-block. This declaration is always required when a filter or class uses fields with a date. Currently, many data sets use different formats of data around the world. GMaVis overcomes this by enabling users to specify the format that is being used in the input data. Therefore, users can specify using the international format described in the ISO 8601. In **`data`** block, note that a **`classification`** sub-block has a set of **`class`** declarations, with a string inside parenthesis (that specifies the class description) and receives a logical operation as a value. It will classify input data and inform the visualization generator to use colored icons and a legend to display markers.

```
1  visualization: markedmap;
2  settings {
3    latitude: field 12;
4    longitude: field 11;
5    marker-text: "Camera:" field 6 image(field 15);
6    page-title: "Photos by Camera Brand";
7    size: full;
8  }
9  data {
10   file: "yfcc100m_dataset_all";
11   structure {
12     delimiter: tab;
13     end-register: newline;
14     date-format: "YYYY-MM-DD";
15   }
16   filter: field 4 is between date "2014-01-01" and
       date "2014-02-01";
17   classification {
18     class("Canon"): field 6 contains "Canon";
19     class("Sony"): field 6 contains "Sony";
20     class("Nikon"): field 6 contains "Nikon";
21     class("Panasonic"): field 6 contains "
       Panasonic";
22     class("Apple"): field 6 contains "Apple";
23     class("FUJI"): field 6 contains "FUJI";
24   }
25 }
```

Listing 2. GMaVis code for Markedmap (the result is in Figure 1(c)).

IV. Compiler Implementation

Our compiler recognizes the DSL source code and generates geospatial data visualizations application code. We created it by using Flex and Bison tools in C/C++ language. First of all, the visualization source code is parsed. Then, the DSL's parser generates an Abstract Syntax Tree (AST) where it saves relevant information such as input file paths, logical operations, delimiters, visualization title, and size. The DSL code generation is performed in two phases. First, it generates the data pre-processor by creating a C++ file and compiling this, generating an executable. The data pre-processor runs to transform raw input data into the filtered and classified visualization data. The second phase generates the final visualization using HTML/Javascript and Google Maps API. Therefore, the data visualization generator receives stored information and generates the map visualization based on the AST. The next sections will present the data pre-processor and visualization generator modules.

A. Data Pre-processor

The data pre-processor module is responsible for transforming input data, applying filtering and classification operations. This module enabled our DSL to abstract the first phase of visualization creation pipeline [5], preventing users from the task of manually dealing with large data sets. It works by receiving the input data, processing and saving an output file with structured and formatted data. The main modules are $Read$, $Process$ and $Write$. They are defined along with other operations in Table I.

TABLE I
The data pre-processor operations and its definitions.

Definition	Description
$F = \{\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n\}$	F is a set of input files to be processed and α represents a single file from a partitioned data set.
$Split(\alpha)$	It splits a data set file of F into N chunks.
$D = \{d_1, d_2, d_3, ..., d_n\}$	D is a set of chunks of a single file. We can say that D is the result of a $Split(\alpha)$ function.
$Process(D)$	It processes a single file D of F
$Read(d)$	It opens and reads a data chunk d of a α in F.
$Filter(d)$	It filters a given data chunk d in D,producing a set of registries to create the visualization.
$Classify(...)$	It classifies the results of $Filter(...)$.
$Write(...)$	It saves the results of $\sum_{i=1}^{n} Process(F)$, where F represents a set of files (α) in an output file to be used in the visualization generation.

The DSL's compiler uses details from the source code to generate this data pre-processor by using C++ programming language. We chose C++ because it enables us to create an

application using a vast set of parallel programming interfaces and enabling low-level improvements in memory management and disk reading. The compiler starts by generating a file called `data_preprocessor.cpp`. All code generated is performed sequentially, including libraries, constants, and data types. Moreover, relevant DSL source code information is transformed and written in the file.

Figure 2 shows an example of the transformation flow performed when parsing a filter sentence. Initially, Flex reads the source code file and generates tokens. Each token is illustrated by a circle shape in Figure 2. Then, Bison combines these tokens according to the DSL's grammar to build the AST. For each node in the tree, Bison performs one or more of the previously described actions. For example, during the construction of the non-terminal node `Logical Operation`, the compiler verifies the semantic value of the received expression by checking the `Integer Literal` node value to generate the data pre-processor function call. In the final step, the compiler uses the information obtained to generate data pre-processor code (illustrated inside the rectangular shapes).

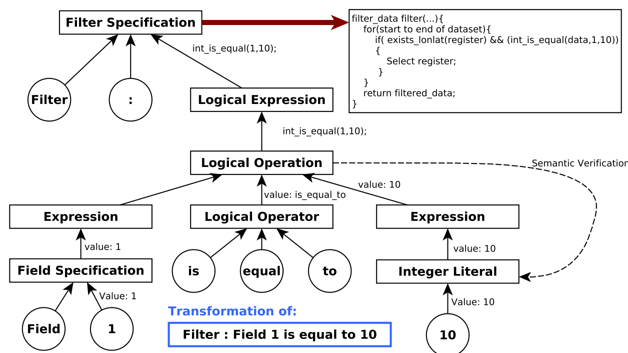


Fig. 2. Representation of the AST parsing.

Finally, we call the GCC compiler to create the executable of our data pre-processor. After compilation, the DSLs' compiler calls the data pre-processor and waits for it to perform data transformations and output the preprocessed data. This output is loaded into the visualization generator, which will be explained in the next section.

B. Visualization Generator

This module receives the output data from the data pre-processor and uses stored information from the source code. The DSL's parser provides information about details specified in the setting block, such as size, title, marker text, and visualization to be created. We specially subdivided the data visualization generation module into four main operations. The **preprocessed data loading and parsing** consists of parsing and loading the output file that contains the pre-processed data. **Data consistency check** performs a consistency check on all the latitudes and longitudes to verify if there is a value using non-numeric/pointers characters. **Central point calculus** gets the maximum and minimum positions for both latitude and longitude, using the middle point among them to give the central point. **Code generation** creates the HTML file, using the input data and all need information for the visualization.

The visualization generator creates the HTML file, printing HTML, CSS, Javascript, and Google Maps API code based on the specifications of the DSL's source code. Our DSL does not generate the visual elements from scratch, since we take advantage of Google Maps' API, thus, simplifying the compiler's work. After the generation, the user is informed where the file is located. Only then the visualization can be displayed using a web browser. Users can interact with the visualization using the mouse to zoom in/out and click on markers to display text boxes. Also, heatmap visualization enables users to increase and decrease the radius as well as change the visualization colors. Finally, users can do everything that Google Maps' API enables by default.

V. Supporting Parallelism in Multi-core Systems

Preliminary performance results considering execution time of three different data workloads have demonstrated that the data pre-processor module has a high computational cost when working with big files [9], [21]. To speed up performance and enable efficient use of multi-core architectures, we provided a parallel version of the data pre-processor module in this paper. It works in a streaming fashion, performing like a pipeline computation. We opted for SPar [22], since it provides a suitable interface for stream processing and a better coding productivity than the state-of-the-art tools, requiring only introducing the code annotations properly. Although SPar provides a simple and high-level interface, programmers have to correctly annotate the source code to achieve good speed-up and correct results. The way to completely abstract parallelism exploitation from the end-users is to introduce SPar annotations during the DSL compilation time.

After performance tracing, we determined that the best performance alternative was to introduce annotations in the process function (Listing 3). It has an explicit pipeline where each operation has data dependency on an earlier step. In this case, it reads each chunk from the disk, filters, classifies, and finally, writes the data in the output file.

```
1 function process(args...){
2   Open(in_file);
3   [[spar::ToStream, spar::Input(in_file)]]
4   while(!in_file.eof()){
5     d_size = Split(in_file);
6     d = Read(in_file, d_size);
7     [[spar::Stage, spar::Input(d), spar::Output(c)
      ]]{
8       f = Filter(d);
9       c = Classify(f);
10    }
11    [[spar::Stage, spar::Input(c)]]{
12      Open(out_file);
13      Write(c, out_file);
14      Close(out_file);
15    }
16  }
17  Close(in_file);
18 }
```

Listing 3. Representation of the process function with SPar.

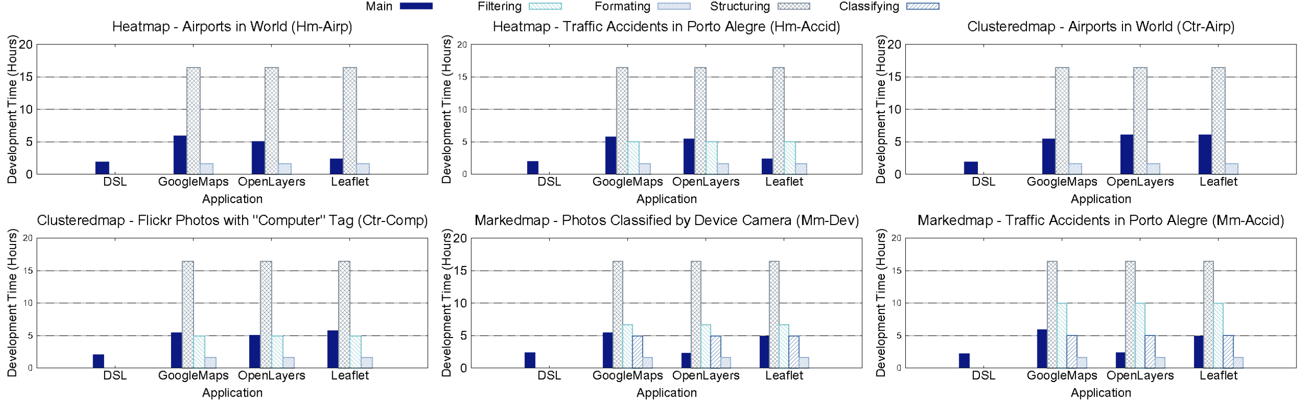


Fig. 3. A comparison of the programming effort results.

Listing 3 illustrates a high-level coding representation of this function, which already has the SPar annotations generated by the GMaVis compiler. As we can observe, SPar required no modifications in the original code. Note that the `process` function behaves in a sequence of stages. The first annotation is `ToStream` (Line 3) which defines the beginning of the stream region and performs initial operations between Lines 4 and 6, including reading data (we consider this the first stage). It consumes files and a set of other variables required for calculating and controlling the processing of the real source code. In Line 7, we inserted a `Stage` annotation which consumes file chunks to be processed. This stage produces a 'string' of data that was filtered and classified as output. The last stage (annotated in Line 11) consumes this 'string' and writes it in the output file.

The GMaVis compiler is able to generate this SPar annotation when the user includes the `--parallel` parameter in the command line as an argument. This compilation flag allows our compiler to known whether to compile the data pre-processor module code with GCC or SPar compiler. The SPar compiler performs the source-to-source code transformation enabling stream parallelism. SPar overlaps processing by using a pipeline strategy where Processor Units (PUs) perform parallel operations, and there is a system thread per PU. More than one operation is executed at the same time.

As the first `Stage` annotation in Listing 3 may process stream elements independently, we can replicate it to achieve a greater degree of parallelism. GMaVis generates `Replicate()` in the attribute list of this stage when a value is specified in the command line for `--parallel <number>`. Unfortunatly, the last stage cannot be replicated because it performs a state-full operation. SPar is able to write results in order to guarantee correct visualizations, although the previous stage works with simultaneous replicas. We can observer that there is a non-linear pipeline behaving through SPar, where all communication and threading models are completely abstracted by their annotations. Therefore, the replicated stage increases the parallelism, performing the same operations (filter and classify) over different stream items/chunks (d_i) in parallel, while still achieving parallelism where the pipeline overlaps.

We enabled parallelism support for the data pre-processor module with minimal changes in the GMaVis compiler, and increasing the code generated by only 9%. As SPar is fully compliant with C++14, there were no major difficulties with language compatibility. The most challenging task was to identify where and how to insert annotations efficiently and deal with stage replication for achieving good performance.

VI. Experiments and Discussion

Experiments were performed to evaluate and highlight the programmability and performance of GMaVis. We used the COCOMO model [23] to estimate the development time, which is an indication for the programming effort. This model takes into account the entire development cycle for generating a visualization, including the initial process of planning, coding, testing, documenting, and deploying it for users. COCOMO named it as cost drivers to express the development environment such as, developers' knowledge, tools, team size, product complexity, etc. They were calibrated once for all the libraries using the following settings: RELY and CPLX were set as very-low; AEXP was set as low; PCAP, LEXP, MODP, and TOOL were set as very high. The remaining cost drivers were set as nominal.

All performance experiments were run using different replica numbers, where 15 repetitions were performed to get the arithmetic mean. Our machine had two Intel Xeon E5645 - 2.4GHz (Hyper-Threading) processors and 24GB of RAM memory. Also, the operating system was Ubuntu 14.04.4 LTS (GNU/Linux 3.13.0-86-generic x86_64) and the data pre-processor module was compiled with GCC-5.3.0 using -O3 optimization flag. Moreover, six visualizations were created using the data sets presented in Subsection III. For each type of visualization provided in the DSL, two visualizations were created with the following characteristics: **[Ctr-Airp]** Clusteredmap that shows the Airports in World; **[Ctr-Comp]** Clusteredmap that presents markers with Flickr Photos tagged with the word "Computer"; **[Hm-Airp]** Heatmap that shows the frequency of Airports in World; **[Hm-Accid]** Heatmap that demonstrates the frequency of traffic accidents in Porto Alegre, Brazil; **[Mm-Dev]** Markedmap with markers that represent photos classified by the device camera used to take this; **[Mm-Accid]** Markedmap with markers that represent traffic accidents in Porto Alegre, Brazil.

A. Programming Effort

Figure 3 presents the development time results obtained through the COCOMO model. In these results, we used five metrics: **Main**, **Structuring**, **Filtering**, **Formatting** and **Classifying**. The **Main** code refers to the HTML/Javascript/library code of the visualization. The **Structuring**, **Filtering**, **Formatting** and **Classifying** refers to the required code to implement automatic data processing. Some visualization compared in this evaluation did not required filtering and classifying. We divided the data pre-processor module code into these four operations to demonstrate this factor. Also, our DSL presents only **Main** results because it abstracts all the four data pre-processing steps, requiring users to implement only the DSL code.

First, we must highlight that the development time results can be considered very high for just coding. However, this measurement considers the whole development software cycle. It estimates the total development time, including planning, coding, testing, documenting, and deploying the software. Therefore, the first two graphs of Figure 3 (Hm-Airp and Hm-Accid) concern Heatmap visualizations. They are not implementing a classification and only Hm-Airp does not require filtering due to the data set format. We can observe that our DSL obtained better results than Google Maps API, Leaflet, and OpenLayers, because it presents a lower estimated development time. Also, we can see that Leaflet requires less code to implement a map visualization than Google Maps API and OpenLayers, since it abstracts HTML code to specify visualization details like position and size.

Two applications for clustered map (Ctr-Airp and Ctr-Comp) were measured. The effort for both the main and data pre-processing were similar to Heatmap when compared to the related solutions. In contrast, our DSL obtained a lower development time estimation since it required fewer lines of code and did not require additional implementations to process data. Moreover, we can observe that Leaflet had a similar result compared with OpenLayer and Google Maps API. This result is due to the implementation, which does not abstract the parameters like in Heatmap. Also, the clustered map visualization is not a native visualization in Leaflet, requiring the use of a plugin. Also, the application Ctr-Airp presented less development time than Ctr-Comp, because filtering was again not necessary in the data set.

The last results present the coding development time estimation for two markedmap visualizations (Mm-Dev and Mm-Accid). Unlike the previous visualizations, they use filtering and classification, requiring more code because they include the creation of mechanisms or the use of external software. Both visualizations presented less development time in the `main` code (which contains only the HTML/Javascript/library code) for GMaVis. In this example, OpenLayers presented the second lowest main code development time, since it required less code to implement this kind of visualization. These visualizations demonstrated the great advantage of GMaVis.

Moreover, even not considering the fact that GMaVis avoids the implementation or use of external software to process data, it improves the developing time on geospatial visualization maps. These results are achieved because GMaVis does not require the implementation of a set of programming elements, such as functions, variables, methods, or any additional code. Also, it requires fewer lines of code to implement most visualizations analyzed, compared to other libraries, which can be verified in Table II.

TABLE II
SOURCE LINES OF CODE FOR EACH APPLICATION.

Application	Our DSL	Google Maps	OpenLayers	Leaflet
Ctr-Airp	15	34	46	46
Ctr-Comp	17	34	28	39
Hm-Airp	15	42	29	24
Hm-Accid	17	41	34	24
Mm-Dev	25	34	22	27
Mm-Accid	21	43	24	27

B. Performance Results

In this section, we present a performance evaluation to demonstrate that our parallelism support implementation was targeted. We show the parallel processing gains with respect to the sequential version of our data pre-processor in Table III. First, taking into account that some applications presented better execution times than others because they did not perform *filtering* and *classifying* operations, requiring only reading, parsing, and writing to the output file, thereby reducing the processing (middle) stage presented in Section IV-A.

TABLE III
RESULTS OF DATA PRE-PROCESSOR IN SIX APPLICATIONS.

App.	Input File Size (MBytes)	Seq. Time (sec.)	Best Time with Parallelism (sec.)	N. of Repli.
Ctr-Airp	Large - 17275.57	424.86	155.59 (3.66 x faster)	4
	Medium - 2879.26	62.48	19.55 (3.13x faster)	4
	Small - 479.87	10.99	4.66 (4.24x faster)	4
Ctr-Comp	Large - 17192.59	293.41	99.34 (3.06x faster)	3
	Medium - 2865.43	23.47	7.01 (3.39x faster)	4
	Small - 477.57	4.14	1.69 (2.99x faster)	4
Hm-Accid	Large - 17272.83	358.85	118.41 (4.12x faster)	3
	Medium - 2878.8	62.48	16.89 (3.30x faster)	4
	Small - 479.8	10.99	3.30 (2.70x faster)	4
Hm-Airp	Large - 17275.57	428.45	158.12 (3.00x faster)	4
	Medium - 2879.26	62.48	18.52 (3.69x faster)	4
	Small - 479.87	10.99	4.38 (2.96x faster)	4
Mm-Accid	Large - 17272.83	428.45	118.71 (2.77x faster)	3
	Medium - 2878.8	62.48	19.14 (3.06x faster)	4
	Small - 479.8	10.99	4.01 (3.65x faster)	4
Mm-Dev	Large - 17192.59	293.63	98.96 (3.37x faster)	3
	Medium - 2865.43	23.03	7.07 (3.07x faster)	4
	Small - 477.57	4.13	1.66 (4.05x faster)	4

Our DSL's parallel data pre-processor is faster than the sequential version even in small data sets. In the best case scenario, it increases performance for the Hm-Airp application 3.69 times, as can be seen in Table III. Mostly because the Airport data set has 12 columns with a greater number of registers than the other data sets that have about 40 columns. Thus, the replicated stages will have more work to do over the data chunk (stream item).

With the results of large input data size, we discovered that the disk becomes a bottlenecked when the data pre-processor module needs to read chunks frequently. Using a fixed chunk size of 100MBytes, about 172 times the read routine will be called in the large data set while 28 for medium and 4 in

small. Also, this chuck size takes a considerable time to read. Consequently, replicating the processing stage many times does not mean an increase in the parallelism efficiency because replicated stages need to wait for new chunks. We identified that the chuck size plays an important role in the data pre-processor module, where smaller chunk sizes keep the pipeline stages always busy, mainly on large data sets. Therefore, we concluded that changing the chunk size dynamically may improve performance in some cases.

Concerning the medium input data size, we needed four replicas to achieve the best execution times. Even though less concurrent disk operations were performed, it needed more replicates than the large workload due to the small number of stream items. We achieved similar scalability with the small workload as the number of replicates increased.

Our results demonstrated that parallelism exploitation is not trivial in the data pre-processor module. In addition to varying chunk sizes, other approaches could improve performance by reducing the disk bottleneck. One is to optimize hardware by deploying RAID as well as flash disk storage. Another way is to implement a distributed file system such as HDFS (Hadoop Distributed File System). The underlying performance is not fully delivered by the SPar DSL. The experiments have shown that it depends on the application constraints. Unfortunately, the multi-core systems still do not have many alternatives to solve disk bottlenecks, which imposes scalability restrictions.

VII. Conclusions and Future Works

This paper contributed by providing a new DSL that abstracts complexities in the whole visualization pipeline, including features of the data transformation step and does not require users to handle data manually. The experiment results showed that our DSL reduced the effort and SLOCs to implement the three supported data visualizations (clusteredmap, heatmap, and markedmap). Compared to related work, we simplified the implementation of geo-visualizations, specially for large data sets. By using standard tools like Bison and Flex, we were able to generate code for multiple programming languages at the back-end (C++ for data pre-processor module as well as HTML, JavaScript and CSS for visualization generator). Thus, we have contributed with a new compiler that automatically parsers the DSL and generates the application code. Moreover, we completely abstracted the parallelism aspects and made it work for multi-core systems. This capability extension on our DSL enabled users to generate faster visualizations.

For future works, we are expanding the provided visualization techniques to offer the most commonly used geospatial visualizations for big data analysis. We are considering the use of other libraries, such as D3 to allow other interactions with the visualization. Such improvements will add new features to create filters and interactivity elements in the final visualization. Furthermore, we plan to enable the distributed file system support for data processing.

Acknowledgement

The authors wish to gratefully acknowledge the research support of FAPERGS, CAPES, and the financial support of FACIN (Faculty of Informatics) and PPGCC from PUCRS.

References

[1] J.-G. Lee and M. Kang, “Geospatial Big Data: Challenges and Opportunities,” *Big Data Research*, vol. 2, no. 2, pp. 74–81, 2015.

[2] R. R. Vatsavai, A. Ganguly, V. Chandola, A. Stefanidis, S. Klasky, and S. Shekhar, “Spatiotemporal Data Mining in the Era of Big Spatial Data: Algorithms and Applications,” in *Inter. Workshop on Analytics for Big Geospatial Data*. Redondo Beach, USA: ACM, Nov. 2012, pp. 1–10.

[3] D. C. Cugler, D. Oliver, M. R. Evans, S. Shekhar, and C. B. Medeiros, “Spatial Big Data: Platforms, Analytics, and Science,” *GeoJournal*, 2013.

[4] M.-J. Kraak and F. Ormeling, *Cartography: Visualization of Spatial Data*. Harlow: Guilford Press, 2011.

[5] K. Moreland, “A Survey of Visualization Pipelines,” *IEEE Trans. on Vis. and Comp. Grap.*, vol. 19, no. 3, pp. 367–378, 2013.

[6] J. Zhang, Y. Chen, and T. Li, “Opportunities of Innovation under Challenges of Big Data,” in *Proceedings of the International Conference on Fuzzy Systems and Knowledge Discovery*, ser. FSKD’13. Shenyang, China: IEEE, July 2013, pp. 669–673.

[7] T. Rauber and G. Runger, *Parallel programming: For Multicore and Cluster Systems*. Bayreuth, Germany: Springer Berlin Heidelberg, 2013.

[8] P. E. McKenney, “Is Parallel Programming Hard, And, If So, What Can You Do About It?” http://www.rdrop.com/p̃aulmck/, Feb 2010.

[9] C. Ledur, D. Griebler, I. Manssuor, and L. G. Fernandes, “Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets,” in *ACS/IEEE Inter. Conference on Computer Systems and Applications*. Marrakech, Marrocos: IEEE, November 2015, p. 8.

[10] N. Sharakhov, N. Polys, and P. Sforza, “GeoSpy: A Web3D Platform for Geospatial Visualization,” in *Proceedings of the 1st ACM SIGSPATIAL International Workshop on MapInteraction*, ser. MapInteract ’13. New York, NY, USA: ACM, December 2013, pp. 30–35.

[11] B. Wylie and J. Baumes, “A Unified Toolkit for Information and Scientific Visualization,” *Visual Data Analytics*, vol. 7243, 2009.

[12] J. Ahrens, B. Geveci, and C. Law, “ParaView: An End-User Tool for Large-Data Visualization,” in *Visualization Handbook*. Burlington, Canada: Elsevier, 2005, pp. 717–731.

[13] M. Bostock and J. Heer, “Protovis: A Graphical Toolkit for Visualization,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1121–1128, Nov. 2009.

[14] Y. Lu, M. Zhang, S. Witherspoon, Y. Yesha, Y. Yesha, and N. Rishe, “SksOpen: Efficient Indexing, Querying, and Visualization of Geospatial Big Data,” in *Inter. Conference onMachine Learning and Applications*, vol. 2. Miami, Florida, USA: IEEE, Dec 2013, pp. 495–500.

[15] H. Choi, W. Choi, T. Quan, D. G. Hildebrand, H. Pfister, and W.-K. Jeong, “Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems,” *IEEE Trans. on Vis. and Comp. Grap.*, vol. 20, no. 12, pp. 2407–2416, 2014.

[16] P. Rautek, S. Bruckner, M. Groller, and M. Hadwiger, “ViSlang: A System for Interpreted Domain-Specific Languages for Scientific Visualization,” *IEEE Trans. on Vis. and Comp. Grap.*, vol. 20, no. 12, pp. 2388–2396, 2014.

[17] G. Kindlmann, C. Chiw, N. Seltzer, L. Samuels, and J. Reppy, “Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis,” *IEEE Trans. on Vis. and Comp. Grap.*, vol. 22, no. 1, pp. 867–876, 2016.

[18] M. Hasan, J. Wolfgang, G. Chen, and H. Pfister, “Shadie: A Domain-Specific Language for Volume Visualization,” http://miloshasan.net/Shadie/shadie.pdf, 2010.

[19] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodk, “Superconductor: A Language for Big Data Visualization,” http://goo.gl/y5vFth, February 2013.

[20] B. Thomee, D. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L. Li, “The New Data in Multimedia Research,” *Communication of the ACM*, vol. 59, no. 2, pp. 64–73, January 2016.

[21] C. Ledur, “GMaVis: A Domain-Specific Language for Large-Scale Geospatial Data Visualization Supporting Multi-core Parallelism,” Master’s thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, March 2016.

[22] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, “SPar: A DSL for High-Level and Productive Stream Parallelism,” *Parallel Processing Letters*, vol. 27, no. 01, p. 20, March 2017.

[23] B. W. Boehm, *Software Engineering Economics*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.