PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

GMAVIS: A DOMAIN-SPECIFIC LANGUAGE FOR LARGE-SCALE GEOSPATIAL DATA VISUALIZATION SUPPORTING MULTI-CORE PARALLELISM

CLEVERSON LOPES LEDUR

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Ph.D. Luiz Gustavo Leão Fernandes
Co-Advisor: Prof. Ph.D. Isabel Harb Manssour

Porto Alegre
2016

Dados Internacionais de Catalogação na Publicação (CIP)

L475g Ledur, Cleverson Lopes

Gmavis : a domain-specific language for large-scale geospatial data visualization supporting multi-core parallelism / Cleverson Lopes Ledur. – 2016.
143 p.

Diss. (Mestrado) – Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul.
Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes
Co-Orientador: Prof. Dr. Isabel Harb Manssour

1. Linguagem de Programação de Domínio Específico (Computadores). 2. Processamento Paralelo (Computadores). 3. Dados Geoespaciais. 4. Visualização de Informação. 5. Informática. I. Fernandes, Luiz Gustavo Leão. II. Manssour, Isabel Harb. III. Título.

CDD 23 ed. 005.133

Salete Maria Sartori CRB 10/1363
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

STATEMENT OF PRESENTATION OF THE DISSERTATION

Dissertation entitled "GMAVIS: a Domain-specific Language for Large-Scale Geospatial Data Visualization Supporting Multi-core Parallelism" presented by Cleverson Lopes Ledur as part of the requirements to achieve the degree of Master in Computer Science approved on March 30, 2016 by the Examination Committee:

Prof. Dr. Luiz Gustavo Leão Fernandes — PPGCC/PUCRS
Advisor

Isabel HManssour

Prof. Dr. Isabel Harb Manssour — PPGCC/PUCRS
Co-Advisor

Prof. Dr. Avelino Francisco Zorzo — PPGCC/PUCRS

Prof. Dr. Marco Danelutto — University of Pisa

Prof. Dr. Carla Maria Dal Sasso Freitas — UFRGS

Ratified on 02/06/2016, according to Minute No. 011 by the Coordinating Committee.

Prof. Dr. Luiz Gustavo Leão Fernandes
Graduate Program Coordinator

This thesis is dedicated to my parents for their endless love, support and encouragement, and to all the people who never stop believing in me.

“I found I could say things with color and shapes that I couldn’t say any other way, things I had no words for.”
(Georgia O’Keeffe)

ACKNOWLEDGMENTS

First, I would like to thank my advisors for guiding this research and helping during the hard times. Also, I thank my colleague Dalvan Griebler who provided insight and expertise that greatly assisted in the research. I would like to thank the research support of FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul) and CAPES (Coordenação de Aperfeiçoamento Pessoal de Nível Superior). Additionally, I would like to thank the financial support of FACIN (Faculdade de Informática) and PPGCC (Programa de Pós-Graduação em Ciência da Computação). Personally, I would like to thank my parents and my brother to the support and patience during the hard work. Also, I want to especially thank my aunts Letícia and Varlenes who gave me my first computer when I was a child and provided support in many moments of my life.

GMAVIS: UMA LINGUAGEM ESPECÍFICA DE DOMÍNIO PARA VISUALIZAÇÕES DE DADOS GEOESPACIAIS EM LARGA ESCALA COM SUPORTE A PARALELISMO EM ARQUITETURAS *MULTI-CORE*

RESUMO

A geração de dados tem aumentado exponencialmente nos últimos anos devido à popularização da tecnologia. Ao mesmo tempo, a visualização da informações permite a extração de conhecimentos e informações úteis através de representação de dados com elementos gráficos. Diferentes técnicas de visualização auxiliam na percepção de informações sobre os dados, tal como a identificação de padrões ou anomalias. Apesar dos benefícios, muitas vezes a geração de uma visualização pode ser uma tarefa difícil para os usuários com baixo conhecimento em programação de computadores. E torna-se mais difícil quando esses usuários precisam lidar com grandes arquivos de dados, uma vez que a maioria das ferramentas não oferece os recursos para abstrair o pré-processamento de dados. Considerando este contexto, neste trabalho é proposta e descrita a GMaVis, uma linguagem específica de domínio (DSL), que permite uma descrição de alto nível para a criação de visualizações usando dados geoespaciais através de um pré-processador de dados paralelo e um gerador de visualizações. GMaVis foi avaliada utilizando duas abordagens. Na primeira foi realizada uma análise de esforço de programação, através de um software para estimar o esforço de desenvolvimento com base no código. Esta avaliação demonstrou um alto ganho em produtividade quando comparado com o esforço de programação exigido com APIs ou bibliotecas que possuem a mesma finalidade. Na segunda abordagem foi realizada uma avaliação de desempenho no pré-processador de dados paralelo, que demonstrou um ganho de desempenho quando comparado com a versão sequencial.

Palavras-Chave: DSL, Visualização de Informações, Dados Geoespaciais, Processamento Paralelo de Dados.

GMAVIS: A DOMAIN-SPECIFIC LANGUAGE FOR LARGE-SCALE GEOSPATIAL DATA VISUALIZATION SUPPORTING MULTI-CORE PARALLELISM

ABSTRACT

Data generation has increased exponentially in recent years due to the popularization of technology. At the same time, information visualization enables the extraction of knowledge and useful information through data representation with graphic elements. Moreover, a set of visualization techniques may help in information perception, enabling finding patterns and anomalies in data. Even tought it provides many benefits, the information visualization creation is a hard task for users with a low knowledge in computer programming. It becomes more difficult when these users have to deal with big data files since most tools do not provide features to abstract data preprocessing. In order to bridge this gap, we proposed GMaVis. It is a Domain-Specific Language (DSL) that offers a high-level description language for creating geospatial data visualizations through a parallel data preprocessor and a high-level description language. GMaVis was evaluated using two approaches. First we performed a programming effort analysis, using an analytical software to estimate development effort based on the code. This evaluation demonstrates a high gain in productivity when compared with programming effort required by other tools and libraries with similar purposes. Also, a performance evaluation was conducted in the parallel module that performs data preprocessing, which demonstrated a performance gain when compared with the sequential version.

Keywords: DSL, Information Visualization, Geospatial Data, Parallel Data Processing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

API – Application Programming Interface

AST – Abstract Syntax Tree

BS – Binary Search

CPU – Central Processing Unit

CSS – Cascading Style Sheets

CSV – Comma-Separated Values

CUDA – Compute Unified Device Architecture

D3 – Data-Driven Documents

DSL – Domain-Specific Language

EAF – Effort Adjustment Factor

EBNF – Extended Backus–Naur Form

GPL – General-Purpose Language

GPU – Graphics Processing Unit

HTML – HyperText Markup Language

I/O – Input/Output

IDE – Integrated Development Environment

IS – Interpolation Search

LALR – Look-Ahead Left Right

LS – Linear Search

MATLAB – MATrix LABoratory

MPI – Message Passing Interface

POSIX – Portable Operating System Interface

RAM – Random Access Memory

TBB – Threading Building Blocks

WMS – Warehouse Management System

YACC – Yet Another Compiler Compiler

CONTENTS

1. INTRODUCTION

Data generation has increased exponentially in recent years. In 2002, about five exabytes were electronically transferred [LV15]. In 2007, the amount of digital data produced in a year surpassed the world's data storage capacity for the first time. Then in 2009, 800 exabytes of data were generated [MCB+11]. The International Data Corporation (IDC)[1] estimates that this volume may grow about 44 times by 2020, which implies a 40 percent rate of annual growing. Many fields such as social networks, government data, health care, and stock market keep producing this data and increase its production every year worldwide [ZH13].

Big data analysis provide interesting information that can help in decision-making. Currently, there are many techniques to perform big data analysis and help users to gain quicker insights. Among these techniques, it is possible to highlight: artificial and biological neural networks; models based on the principle of the organization; methods of predictive analysis; statistics; natural language processing; data mining; optimization and data visualization [ZH13]. All of these techniques can extract information/knowledge and produce predictions to help people in decision making. Furthermore, big data analysis can empower the scientific and general community to solve problems, create new products, identify potential innovation, and improve people's lives. Moreover, it is currently used in many areas like biology, health, finance, social networking and other fields that require its advantages.

Data visualization is an efficient and powerful technique used in big data analysis. It has an advantage over other techniques because the human brain can process visual elements faster than texts/values. The human perception system processes an image in parallel while texts and values are limited to a sequential reading process [WGK10]. Consequently, data visualization is very useful for big data analysis, since it facilitates and accelerates the human understanding of data/information. Information representation using graphical elements has evolved, and it has been used in several areas to increase human perception [Gha07].

1.1 Motivation

In a simplified way, the visualization pipeline, explained in Chapter 2, has three main stages. The first one is the *data pre-processing* phase that includes data modeling and selection. The second one is *data to visual mappings*, which covers the mapping of the data to visual structures. The last stage is the *view transformation*, which refers to the users interactions on the visualization [WGK10]. Current tools that enable the creation of geospatial data

[1] http://www.idc.com/

visualization do not provide abstraction of complexities for the first phase. Thus, although big data visualization offers many benefits, its production is still a challenge [ZCL13]. Users with a low-level knowledge of software development may have a hard time for creating an information visualization for a large amount of data, because it requires high cost and effort in programming to process and manipulate large volumes of data. Most current tools require at least some computer programming knowledge, even when using high-level interface libraries/tools (e.g., Google Charts, Leaflet, OpenLayers, Processing, etc.). Therefore, when domain users are dealing with big datasets or huge files, they need to implement or use a third software or tool to preprocess data before the visualization creation. These tasks take considerable time that users could use to focus on domain problems.

Since a visualization creation for a vast amount of data requires a high computational cost in processing, parallel programming can be explored to achieve better performance. Parallel programming enables applications to take advantage of parallel architectures by dividing the data or the processing to perform in several processor unities [RR13]. Thus, the main advantages of parallel programming are reducing execution time and improving performance by performing more tasks or processing more data in less time. Currently, there are many parallel computer architectures such as multi-core/GPU workstations, clusters, and grids. Today, even simple workstations have multiple processing units and this type of architecture is extremely accessible for any domain user. Hence, exploring these computer architectures through parallel programming is essential to reach better performance in computing.

Nevertheless, taking advantage of parallelism is a difficult task because it requires analysis of some factors. These factors may influence the process of software parallelization. For example, it is essential to choose a computer system architecture, analyze the software problem, select the parallel programming model that best fit the problem, learn about a parallel programming interface, and other tasks. Decisions must also be made during development that may change the final product, sometimes achieving unexpected results such as unsatisfactory performance. Parallel programming also requires another set of considerations, such as how the software will handle memory, data dependency, communication, input/output of data, synchronization, interface that better solve the problem, and how data/tasks will be split. Thus, parallel programming is also a hard task for domain users, since they have to deal with low-level programming and understand many questions in this area to extract the maximum performance of parallel computers.

1.2 Research Scenario

GMaVis and this work contributes to a framework built at GMAP[2] research group. It is the result of a set of researches and works developed in the research group that has as main objective to facilitate the creation and implementation of parallel applications. GMaVis was built on top of this framework, at the application level, generating SPar [Gri16] annotations and using its compiler to generate a parallel data preprocessor. Figure 1.1 illustrates this framework.

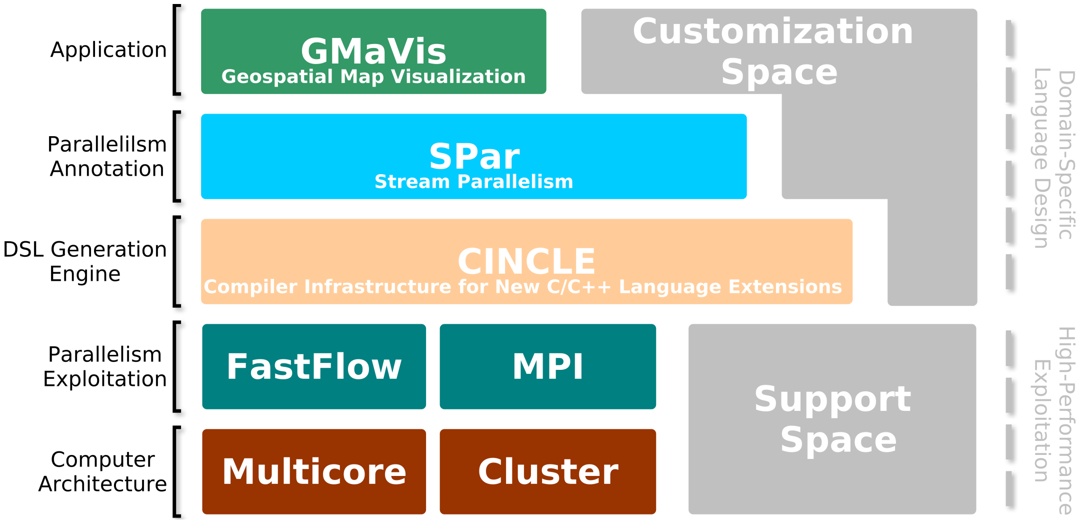


Figure 1.1 – Research scenario framework (Extracted from [Gri16]).

SPar is a domain-specific language that allows the parallelism of code using annotations and the implementation of parallel processing based on stream parallelism. SPar was proposed to address stream processing applications. It is a C++ embedded domain-specific language (DSL) for expressing stream parallelism by using standard C++11 attribute annotations. It introduces high-level parallel abstractions for developing stream based parallel programs as well as reducing sequential source code rewriting. Spar allows C++ developers to express stream parallelism by using the standard syntax grammar of the host language. It also enables minimal sequential code rewriting thus reducing the effort needed to program the parallel application. Additionally, SPar provides flexibility for annotating the C++ sequential code in different ways. In the DSL Generation Engine, Cincle, a compiler infrastructure for new C/C++ language extensions enables SPar compiler to generate code to FastFlow and MPI (Message Passing Interface) that takes advantage of different architecture systems.

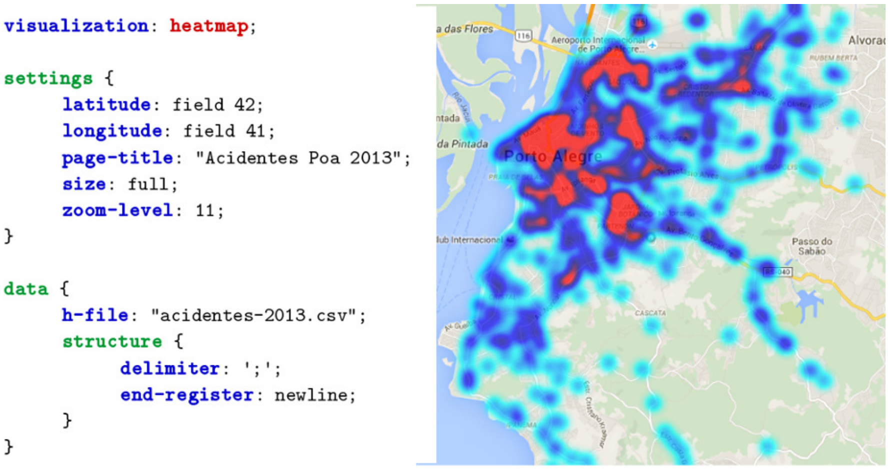
Therefore, SPar simplified the parallelization of the data preprocessor module by enabling GMaVis to compile the same code for both parallel or sequential execution by just modifying a compiler argument. GMaVis used the same code as the sequential version

[2] www.inf.pucrs.br/gmap

annotated in order to generate parallel version with SPar compiler. Also, it enabled GMaVis to easily abstract the parallel programming completely from users. Thus, domain users will not have to worry about creating parallel programming code to speed up data processing during visualization creation.

1.3 Objectives and Contributions

Considering this context, the main goal of this work is to present and describe GMaVis [LGMF15], an external domain-specific language (DSL) that facilitates the creation of visualization of geospatial information by using multi-core architectures to process data in parallel. Its compiler abstracts complexities from the whole visualization creation process, even in the *data pre-processing* phase. Also, it allows domain users with low-level knowledge in computer programming to create these visualizations through a high-level description language. These users can easily do it with a few lines of code, using simple declarations and blocks to express visualization details. Currently, GMaVis supports the creation of three types of geospatial visualization: markedmap, clusteredmap and heatmap. Figures 1.2 and 1.3 illustrate the GMaVis source code required to create a heatmap and a markedmap, respectively. These examples show that GMaVis has a short and simple grammar, enabling users to create a visualization with a few lines of code.



```
visualization: heatmap;

settings {
      latitude: field 42;
      longitude: field 41;
      page-title: "Acidentes Poa 2013";
      size: full;
      zoom-level: 11;
}

data {
      h-file: "acidentes-2013.csv";
      structure {
            delimiter: ';';
            end-register: newline;
      }
}
```

Figure 1.2 – Example of GMaVis code to generate a heatmap.

Furthermore, GMaVis provides declarations to create filters and classes. These declarations can be used to select data, remove registers, apply data classification, highlight specific values, and create legends with symbols and classification description. Also, it includes delimiter specification that enables the use of raw datasets as input. In this case, GMaVis parses the input data using the specified delimiters. Moreover, this DSL provides

```
visualization: markedmap;
settings {
  latitude: field 12;
  longitude: field 11;
  marker-text: "<img src=" field 15 " width=200>";
  page-title: "Photos of 2014 by Camera";
  size: full;
}
data {
  file: "BIGDATA_YAHOO/yfcc100m_dataset-0";
  structure {
    delimiter: tab;
    end-register: newline;
    date-format: "YYYY-MM-DD";
  }
  filter:  field 4 is greater than date "2014-02-01";
  classification {
    class ("Canon"): field 6 contains "Canon";
    class ("Sony"): field 6 contains "Sony";
    class ("Nikon"): field 6 contains "Nikon";
    class ("Panasonic"): field 6 contains "Panasonic";
    class ("Apple"): field 6 contains "Apple";
    class ("Kodak"): field 6 contains "Kodak";
  }
}
```

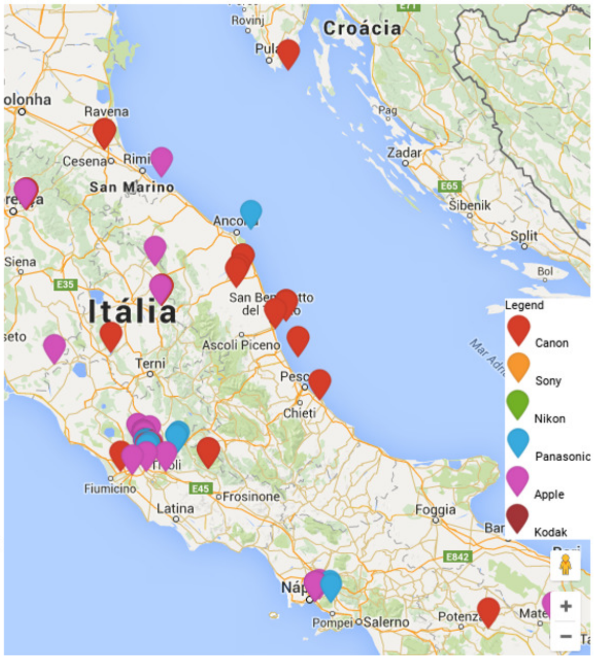


Figure 1.3 – Example of GMaVis code to generate a markedmap.

automatic data processing, requiring users to only inform the data structure and field's location for latitude and longitude.

A compiler was developed in order to enable the use of GMaVis. This compiler processes the DSL code together with input dataset to generate the geospatial data visualization automatically. Also, it allows users to use big and raw datasets. Big datasets are supported because this compiler can process huge data files even in low memory computers through a smart memory allocation and data processing. Also, raw data[3] can be used since GMaVis description language supports data structure specification. A parallel data preprocessor module is responsible for parsing, filtering, classifying, and generating an output with only useful information to generate the visualization. As a result, domain users do not need to handle data manually or use data processing tool. This compiler also generates this data preprocessor module automatically, using the information in the input code.

1.3.1 Research Questions and Hypotheses

The following research questions were defined for this research evaluation, based on the context and problem.

- **Q1** Is it possible to create a DSL (GMAVIS) to reduce the programming effort for creating geospatial visualization?

[3] In this case, raw data is considered as an output text of some processing with a structure with delimiter characters. This definition does not include audio, video or images.

- **Q2** Can the parallel code generated by this DSL speed up the data processing of raw geospatial data?

With these two research questions, some hypotheses were considered. H1 and H2 are related to Q1, and H3 and H4 were defined in respect to Q2.

- **H1** GMaVis requires less programming effort than visualization libraries (Google Maps API, Leaflet and OpenLayers).
- **H2** The inclusion of automatic data pre-processing in GMaVis reduces programming effort.
- **H3** GMaVis can generate parallel code annotations using SPar for speeding up performance.
- **H4** Code generation for SPar is simpler than using TBB (Thread Building Blocks) for speeding up performance.

This work has three main contributions described as follows:

- First, it offered a high-level interface that abstracts all raw data pre-processing, programming elements (*i.e.*, functions, variables, and expressions), visualization generation and parallel programming as a whole.
- Second, it presents an efficient internal data preprocessor module, which has enabled the processing of huge data files on computers with low memory specifications.
- Third, it presents a method for generating visualizations automatically through a simple description language, with total abstraction of parallel programming and low level programming. Also, this work includes the implementation of a compiler to receive both source code and data to generate the visualization automatically.

We evaluated the DSL using programming effort and performance approaches. First we wanted to know if GMaVis could increase code productivity and reduce the effort required to implement a geospatial data visualization from scratch. Also, we verified if it would be possible to increase performance with a parallel version of data preprocessor module. Results from programming effort analysis showed that GMaVis can reduce the effort for developing geospatial data visualization since it requires less lines of code than other libraries. Moreover, it does not need an extra effort to preprocess data before the visual mapping and display phases of visualization creation, abstracting complexities from the whole visualization creation process. Performance results exposes that GMaVis achieved less execution time and higher throughput rates to preprocess data using parallel processing. Also, a comparison of data preprocessors parallelized using SPar and TBB [Rei07] presented high throughput and less execution time when using SPar working with big datasets. In some applications, TBB performed in less time with small datasets.

1.4 Thesis Organization

The remainder of this work is organized in the following way. Chapter 2 presents some background concepts related with this research. Chapter 3 introduces and compares the most important related works. Chapter 4 details the proposed domain-specific language and Chapter 5 explains its implemented compiler. Chapter 6 presents the methodology and evaluation results. Finally, Chapter 7 presents the final considerations and future works.

2. BACKGROUND

This chapter presents some concepts related to this work. Also, it introduces some background required to understand the proposed DSL and the compiler creation process. It starts with an introduction to domain-specific language (DSL), its types, benefits and steps to design. Section 2.2 provides information about data visualization, explaining its importance and providing details about the visualization creation workflow. It also presents a brief explanation about Google Maps API [Sve10], Leaflet [Lea15], and OpenLayers [Ope15]. Section 2.3 introduces some parallel programming concepts and demonstrates important factors to consider when implementing a parallel program. Finally, Section 2.4 briefly defines concepts about compiler creation, and describes the tools used to develop the GMaVis compiler, such as Flex [Gao15b] and Bison [Gao15a].

2.1 Domain-Specific Languages

Historically, technology became popular because many complexities were abstracted. Initially, computers were programmed in low-level fashion with codes for specific hardware architectures. Only some specialists were able to perform the machine programming. With the appearance of higher-level programming languages, computer programming became more popular, enabling a significant technology evolution. Currently, complex applications are built because hardware supports it. However, these applications cannot be created using only a few lines of code and require a huge effort from developers. Looking for coding productivity and less effort in programming, the domain specific language (DSL) approach was established to abstract programming complexities for specific domain users [Fow10].

To better define a domain-specific language, consider that P is a set of all conceivable programs. Program p in P is a conceptual representation of a computation that is executed on a universal computer (or Turing machine). Language L_i represents a structure or notation for expressing or encoding the programs in P. Thus, a program P_i in P is encoded in L_i, and can be denoted as P_{L_i}. Also, in L there are several languages that can express a program p in different ways [VBD+13]. For example, a matrix multiplication program is a program P_i in P that can be represented or codified in several languages in L, such as C++, Java, Pascal, and many others. Thus a matrix multiplication program can be considered a P_{L_i}.

A domain, represented by D, is a set of programs with common characteristics or similar purposes. It can also be a subset of programs P_{L_i} written in a specific language L_i. Moreover, a domain can be considered a body of knowledge about the real world, which requires some form of software. Then, P_D would be a subset of programs in P that pro-

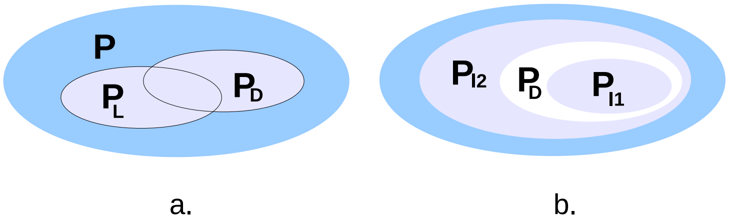


Figure 2.1 – DSL concepts (Adapted from [VBD+13]).

vides computations for D and overlaps with P_L, since multiple languages can express the P_D program as illustrated in Figure 2.1 (a) [VBD+13]. For example, we can consider data visualization as a domain. It is possible to find many programs being used in the real world written in languages that enable the visualization of data. These programs are used to address problems in the data visualization domain and can be written using some programming languages in a set of existing programming languages, but not all of them.

A Domain-Specific Language, represented by l_D for a domain D is specialized for encoding and expressing programs in P_D, as illustrated in Figure 2.1 (b). They are more efficient in representing a P_D program than other languages since they use domain abstractions and require less code to implement applications. With these abstractions, users do not have to spend their time with implementation details and irrelevant things to the problem in D [VBD+13]. Also, with DSLs, users do not need to express a problem using a GPL (general purpose language). GPLs do not focus on the domain problem and, depending on the expressiveness, can generate much more code to express simple domain elements [Fow10]. To illustrate, consider the last example about a data visualization domain. We can consider a programming language, which provides a way to create applications for data visualization domain with abstractions, without the requirement of code details or graphical elements in a low-level abstraction, as a domain-specific language.

2.1.1 DSL Types

DSLs are defined as programming languages targeted to solve a particular problem, using a common vocabulary within the domain. Unlike general purpose languages, a DSL may offer abstractions for enabling users to focus on domain problems. Thus, DSL users can focus just on the core aspects of a subject, ignoring unnecessary implementation details [vDKV00]. Currently, there are three types of DSLs [Fow10]:

- **Internal DSLs** - DSLs built and embedded with a general purpose language. These DSLs use the flexibility of a general purpose language to increase the expressiveness and offer more possibilities to the user.

- **External DSLs** - Address DSLs that have their own grammar, without any general purpose language as host. These DSLs usually have simple grammar and focus on offering high-level programming language interfaces for the user. On the other hand, users will need to learn a new language and may have limited expressiveness.
- **Workbench DSLs** - These are non-textual DSLs that use graphic representations for development through an interface development environment (IDE). With a workbench DSL, users can easily visualize the domain problems with visual elements such as spreadsheets, graphical models, and shapes. Also, some domain experts may feel more comfortable manipulating visual elements than source code [Gho10].

2.1.2 Advantages in Using DSLs

The use of DSLs to address domain problems provides some benefits [MHS05]. First, it allows better software specification, development and maintenance. Usually, the language interface is created based on the domain problem and has a grammar with a vocabulary closer to the domain, with terms and commonly used words. It facilitates the understanding of code even for non-developers. Also, DSL packaging abstract implementation details allow users to focus on important questions to solve their problems. Moreover, it enables efficiency, since a DSL reduces the required effort to write or modify a code, and the communication between developers and domain users is improved. Thus, it provides high consistency of applications with software specifications.

DSLs allow software to be expressed using a friendly language, at the same abstraction level of the domain problem. This enables domain experts to understand, verify, change and develop their software by themselves, without requiring experienced developers with specific knowledge in computer programming. Furthermore, this empower domain users and enables the creation of high quality applications because domain experts, who have a deeper knowledge about domain problems, can make improvements.

2.1.3 DSL's Design and Implementation

A DSL design and implementation requires the execution of five main steps. These steps are decision, analysis, design, implementation, and deployment, as illustrated in Figure 2.2. Initially, developers have to consider if a DSL will effectively help users to reduce development effort and/or costs. Furthermore, the decision phase includes an analysis of the availability to create a new DSL, its costs, and advantages. The analysis step is performed to understand the domain problem, its complexities, possible barriers, and how to improve and facilitate domain users to solve their problems through a DSL. Also, it determines which type of DSL the domain users expect. Moreover, to avoid costs and time-consumption, this step includes a study concerning existing DSLs, to see if it is possible to use an existing DSL instead of creating a new from scratch [MHS05].

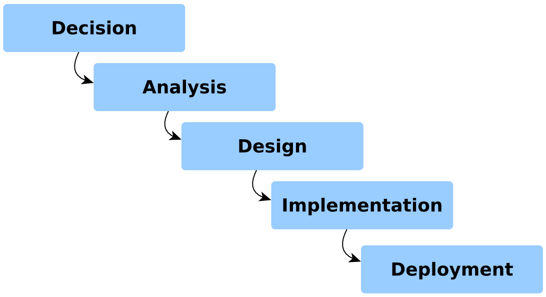


Figure 2.2 – DSL design steps (Adapted from [MHS05]).

The third step is the design, where the DSL creator will make decisions about implementation aspects that will influence the whole DSL cycle life. For example, it can start with the decision about creating an external or internal DSL. Creating an internal DSL enables the use of an existing familiar programming language for domain users, requiring less effort from users in the learning process. However, when users do not know about a GPL, implementing an internal DSL will require more time for users because they will have to learn both the GPL and the DSL. Also, in this step, the DSL developer will specify the vocabulary, grammar, transformation rules, DSL architecture, and additional libraries or tools. The next step is the implementation, which includes the creation of the DSL software that will enable users to generate applications. In this phase, the compiler or interpreter, and the whole DSL architecture will be implemented. Finally, the deployment phase includes the DSL implementation for users [MHS05].

2.2 Data Visualization and Geospatial Data

Data visualization is defined as "the communication of information using graphical representations" [WGK10]. People use images to express information since before any formalization in written language. Because the human perception system processes images in parallel, this achieves faster results in finding and extracting information. Sequential reading limits learning and information collecting from text or values to a slower process. Thus, visualization techniques are used in daily activities, such as the utilization of a map in an unknown region, a chart of the stock market, an analysis of the human population and in advertising. In all of these activities, visualizations can provide a representation of textual or verbal information. It can also be used as a complementary alternative to provide a quick and adequate understanding of information [WGK10].

Currently, many data visualization tools offer different possibilities to represent data. Usually, these visualization tools are classified by the supported amount or type of data, data representation and its techniques to generate and display. Visualization techniques used in the creation process depend on the kind of input data [Gha07]. Therefore, there are different data visualization techniques that are more appropriate for each data type. It is up to the visualization creator to choose the best visualization technique for the input data.

Often, the process of creating a data visualization follows a workflow, which, in a simplified way, has three main phases. Figure 2.3 demonstrates this workflow. The first phase is *data pre-processing*. It includes an analysis of the input, structure and the kind of data that is being handled. When this data comes from a raw dataset without previous processing, data preparation is required to select, filter, classify and remove undesired registers. The second phase is *data to visual mappings*, where the data elements is mapped into visual representations, according to the attributes previously selected. Finally, the visualization is generated in *view transformations*. It also includes setting the scene parameters and interactions between the user and the visualization, which allows them to modify details and use mechanisms to have a richer experience in data exploration [WGK10]. Users can change the final visualization by changing information in the three phases according to their programming and data manipulation knowledge.

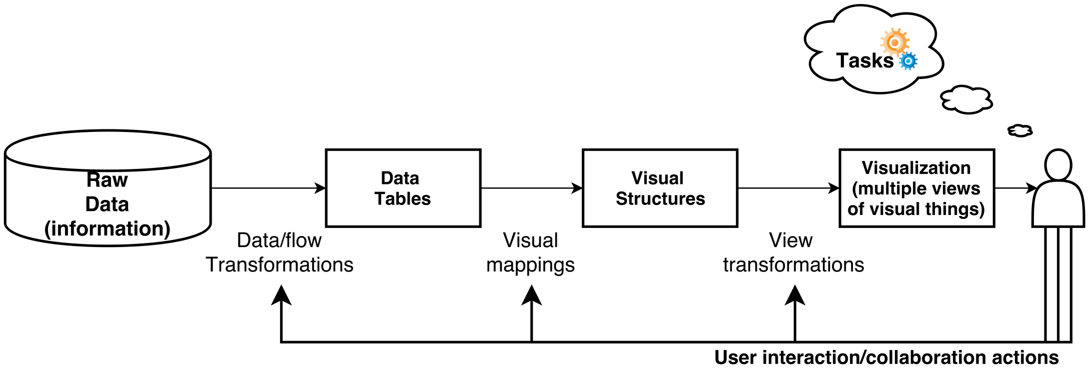

Tasks
Raw Data (information)
Data Tables
Visual Structures
Visualization (multiple views of visual things)
Data/flow Transformations
Visual mappings
View transformations
User interaction/collaboration actions

Figure 2.3 – Data visualization pipeline (Adapted from [CMS99]).

The process of visualization creation using a large dataset, however, is a hard task for users with a low-level of knowledge in computer programming. For example, to create a big data visualization, domain users have to deal with big data processing, analysis, and visualization algorithms/technologies. Also, depending on the visualization type, data mining algorithms are fundamental to select, classify, reduce or optimize data. If the visualization has some interactivity with users, it will need a parallel interface to process the graphic elements. For all these requirements, users need to look for algorithms, tools, and platforms and learn about them before implementation. Since these users are from science/industry and have limited knowledge of programming languages, this implies a great deal of effort and time to visualize an information from the datasets. This effort is even significant if raw data is input, which requires pre-processing.

This work focuses on geospatial data visualizations that use a special type of data which specifies the location of an object or phenomena in each register, usually through its *Latitude* and *Longitude*. Examples of geospatial data are global climate modeling, environmental records, economic and social measures and indicators, customer analysis, and crime data. The strategy used to represent this kind of data is to map the spatial attributes directly to a 2D map visualization [WGK10].

Some libraries allow users to create geospatial data visualizations, such as Google Maps API, OpenLayers, and Leaflet. When using these libraries, it is up to the user to pre-process the data in the correct format. Furthermore, for massive data, it is often required to create a software to process it automatically and provide an output file with the format used in the library. Figure 4.1(a) shows the workflow of traditional libraries to generate a visualization. The dotted line around *Generation of the Visualization* and *Library Format Data* demonstrate the supporting scope of these libraries without needing extra programming by the user. Because we used these libraries in this work, each one will be briefly explained bellow.

2.2.1 Google Maps API

Introduced by Google in 2005, this API improved the use of maps on the web, allowing users to drag and interact to find information. Basically, Google Maps API uses HTML, CSS, and JavaScript to display the map and its elements. This API use tiles, which are small pieces of images, to create a full map. Initially, tiles are loaded in the background with Ajax calls. Then these are inserted into <div>tags in an HTML page. When the user interacts with a created map with the Google Maps API, it sends coordinates and detailed information to a server, and this provides with new tiles that load on the map [Sve10].

To create a data visualization maps using the Google MAPs API, users must have some knowledge of JavaScript to create variables, objects and functions. For example, the steps to generate a simple map are the following. Initially, it requires the inclusion of the library in the HTML file through a URL. Also, a map object may be declared and associated with a previously created variable. This object receives some information, as initial position, zoom, layers and the *div* element which will receive the map in HTML code. Finally, users need to have data to insert in this visualization. For a marked map, each marker requires, at least, three lines of code to generate an object with latitude and longitude information. If a classification is desired, such as to change marker colors or sizes, users may insert a variable receiving another object with an icon declaration.

For a better understanding of data visualization creation using this library, some pieces of code are presented as examples. Because of the number of lines occupied, we do not show the additional HTML and JavaScript code, focusing just on the Google Maps API code required to create the data visualization. The Listing 2.1 presents how to build a map with markers using Google Maps API.

```
 1 //HTML CODE (with API declaration and required elements)
 2
 3   var map = new google.maps.Map(document.getElementById('map'), {
 4     zoom: 6,
 5     center: new google.maps.LatLng(54, 12);
 6     mapTypeId: google.maps.MapTypeId.ROADMAP,
 7   });
 8
 9 //...Data...
10   var marker = new google.maps.Marker({    //This piece of code is replicated
11     position: new google.maps.LatLng(54, 12),  //for each additional marker inserted.
12     map: map
13   });
14 //...Data...
15
16
17 //HTML CODE (Body)
```

Listing 2.1 – Map creation with Google Maps API [Sve10].

When a developer wants to insert a new marker on the map, he must add the data and its code in lines 10 and 13 of the Listing 2.1. Note that, for each marker inserted in the map, the developer needs to replicate this piece of code, modifying the information about latitude and longitude. If this data visualization handles a large volume of data, then the final code will be large, considering the numbers of lines. Also, the Google Maps API does not provide a feature to process data automatically. It requires users to handle data manually or use an external tool to process data before it is inserted into JavaScript declarations and put in the HTML code. The previous example showed a code to create a simple map with one marker. However, if a user needs to create one with one hundred markers, each data element would have to be determined and inserted into the JavaScript code, replicating the *marker* variable. This task may increase the effort to create the map and, depending on the quantity of data, it would require the use of external tools or software development to perform this activity.

2.2.2 OpenLayers

OpenLayers is an open source JavaScript library that provides features for displaying map data in web browsers. It also provides an API for building web-based geographic applications. Furthermore, it offers a set of components, such as maps, layers, or controls.

OpenLayers offers access to a great number of data sources using many different data formats. Moreover, it implements many standards from Open Geospatial Consortium[1] [Ope15].

Similar to the Google Maps API, OpenLayers also requires the library declaration in the HTML file. The process for creating a simple map with OpenLayers is similar to that presented in the Google Maps API. First a declaration of an object to a map with details and information must be given to create the visualization. In OpenLayers, the creation of at least one layer is required. A layer is a feature in OpenLayers that allows for the implementation of different types of data visualization in a single file. For example, it enables the creation of a heatmap with markers using two layers. However, when creating a simple map with markers, it modifies the insertion of markers on the map. Here, each marker will not only be created but also associated with a layer.

```
1  //HTML CODE (with API declaration and required elements)
2    var map = new ol.Map({
3        target: 'map',
4        layers: [new ol.layer.Tile({source: new ol.source.MapQuest({layer: 'sat'})})],
5        view: new ol.View({
6          center: ol.proj.transform([54.41, 12.82], 'EPSG:4326', 'EPSG:3857'),
7          zoom: 4
8        })
9      });
10   var markers = new OpenLayers.Layer.Markers("Markers");
11   map.addLayer(markers);
12
13 //Data...
14   var lonLat = new OpenLayers.LonLat( −89.461412 , 30.912627) //This piece of code is replicated
15            .transform(                                     //for each additional marker inserted.
16              new OpenLayers.Projection("EPSG:4326"),
17              map.getProjectionObject()
18            );
19 //Data...
20
21   markers.addMarker(new OpenLayers.Marker(lonLat));
22 //HTML CODE (Body)
```

Listing 2.2 – Layer creation with OpenLayers library [Ope15].

Listing 2.2 presents an example code for creating a simple map with one marker. In the first lines, it has the map specification with details about centering, position, tiles and zoom. Then, a layer called *markers* is declared in line 10. This layer receives the markers in the following lines of code. As in the first example with the Google Maps API, here the code to insert a marker is also replicated. Thus, users will need to handle their data manually or use an external tool to process data before creating the visualization.

2.2.3 Leaflet

Leaflet is an open-source JavaScript library used to create interactive maps. Leaflet works by taking advantage of HTML5 and CSS3 [Lea15] to implement maps and display vi-

[1] http://www.opengeospatial.org

sual elements. Similar to the previous libraries, it allows for the creation of maps using geospatial data. Moreover, it provides and supports many features that enable implementation of maps with detailed specifications, since it offers tile layers, markers, popups, vector layers, polylines, polygons, circles, rectangles, circle markers, GeoJSON layers, image overlays, WMS layers and layer groups. Listing 2.3 demonstrates how a simple map with one marker is created using Leaflet library.

```
//HTML CODE (with API declaration and required elements)

  var map = L.map('map').setView([51.505, -0.09], 13);

  L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png', {
    attribution: '&copy; <a href="http://osm.org/copyright">OpenStreetMap</a> contributors'
  }).addTo(map);

//Data...
  L.marker([51.5, -0.09]).addTo(map)                  //This piece of code is replicated
    .bindPopup('A pretty CSS3 popup. <br> Easily customizable.') //for each additional marker inserted.
    .openPopup();
//Data...


//HTML CODE (Body)
```

Listing 2.3 – Map creation with Leaflet library [Lea15]

The marker and map creation process in Leaflet is similar to other libraries. Initially the HTML file is inserted into the library. Next a variable that instantiates a map object is created. Another variable then instantiates a marker object given a geographical point and this marker is added to the map. Finally, an option in this instantiation is declared, which allows the specification of a text to be shown when a marker is clicked. As in other libraries, the user must have knowledge of JavaScript programming to create a data visualization map. Also, the user will need to process and format data before inserting it into the code.

2.3 Parallel Programming

The performance of microprocessor has increased about 50% per year between 1986 and 2002 [Pac11]. This increase in performance of single core computers generated an expectation in users for the next generations of processors. However, the improvements in performance reduced about 20% after 2002 [Pac11], because single core processors and the monolithic design could not support higher clock frequencies without a high level of heat and power consumption. Therefore, most manufacturers started producing their processors with multiple processing units in a single integrated circuit. This enabled manufacturers to keep increasing the performance and offer fast machines for processing and solving problems with high computational cost, such as climate modeling, protein folding, drug discovery, energy research, and data analysis [Pac11].

Computer architectures with multiple processors have become popular, and currently it is easier to find a parallel computer. Many current smartphones have more than one processing unit and enable parallel processing. If we consider the use of graphical processors, the number increases. In recent years, parallel architectures have been used to solve large problems and process large amounts of data quickly, enabling progress in many areas. This is possible because a parallel computer has a set of processing units that work together to solve a large problem. We can classify these computers systems as shared and distributed memory architectures, as illustrated in Figure 2.3. Shared memory architectures are characterized by all processors units acessing a centralized memory. However, in distributed memory architecture, each processor unit has its own memory, and any communication between them is performed through messages using an interconnection network. In both architectures, it is possible to create parallel applications to be used efficiently, which requires the use of parallel programming during development [CGS97].

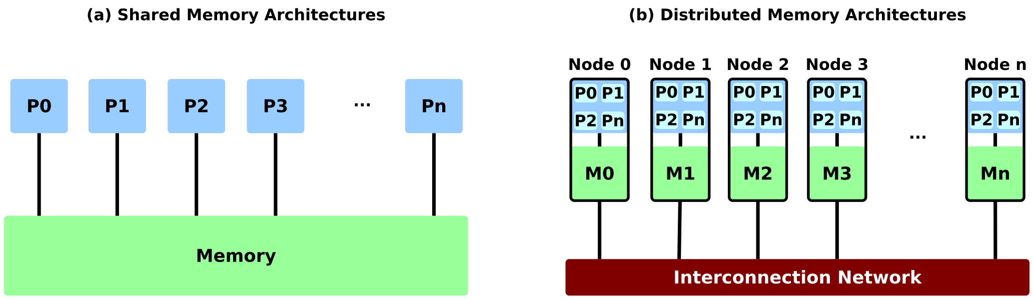


Figure 2.4 – Parallel architectures types (Adapted from [Bar15]).

Until 2002, most programs were written for single-core processing and could not take advantage of multiple cores in the same hardware. The popularization of parallel architectures required modifications in software programming [Pac11]. Developers needed to use parallel programming to create software that could process in parallel using multiple processor units. This also required the rewriting of single-core programs to be executed in parallel and use the computational resources efficiently. Currently, most computers have more than one processor, which requires the development of parallel applications to use all the available resources.

Parallel programming allows users to develop applications that use multiple processing units. This is possible by dividing data/tasks [Pac11] to run in several processor units. In parallel computing applications, tasks are broken into several, often many, similar subtasks to process independently. Before completion, each result is combined, providing the processing output. Even parallel programming offering fast processing and decreasing execution time, most programs are written for sequential processing, focusing on conventional single-core systems because the process to parallelize a software is difficult, and requires experienced developers.

2.3.1 Exploring Parallelism

Exploring parallelism requires changing a sequential code to execute in parallel. There are many ways to parallelize software, and they all depend on the hardware, how the problem will be solved, available software, interfaces for parallel programming and the strategy that will be used. To organize the parallelization process, some models were defined according to the parallelization strategy. A parallel programming model is an abstraction of how a system works. It specifies the programmer's view of a parallel application. Some factors can influence the decision of which model use, such as architectural design, the language, compiler, or the available libraries. There are different parallel programming models for the same architecture, and the choice of one depends on the application parallelization strategy.

We can classify parallel programs by the strategy used. This strategy is related to the hardware because different computer architectures require different implementations of parallel programs. For example, a single computer with a multi-core has a different parallel program implementation than a cluster of computers. Thus, parallel programming is intrinsically related to the hardware. Furthermore, developers that create parallel software must know the target computer architecture that will run the program. This is required because they must decide between the models that can produce better results and optimize the algorithm to extract good performance. The three possible classes for this division based on computer architecture are parallel programs that use shared memory, message passing or both, called hybrid programs [MSM04]. These models are presented bellow.

- **Shared Memory** The shared memory model shares a common address space in memory among processes. These processes can read and write in this memory asynchronously. There are mechanisms like locks or semaphores that resolve contentions, prevent race conditions, deadlocks and control this access to memory to avoid problems [MSM04]. The advantage of using this model is the facility to implement a parallel program because the developer does not need to worry about the communication between processes. If we do not consider hardware implementation, in shared memory model all the processes have equal access to the memory. Figure 2.3.1 illustrates this model where multiple processes access the shared memory for both reading and writing.

- **Message Passing** In this model, the processes have their own memory to local and private access. This model does not have a global memory for universal access. Thus, the multiple tasks can run on a physical machine or multiple machines connected by a network. This model uses messages for processes to communicate and exchange data through the network connection. Furthermore, it is complex to implement a par-

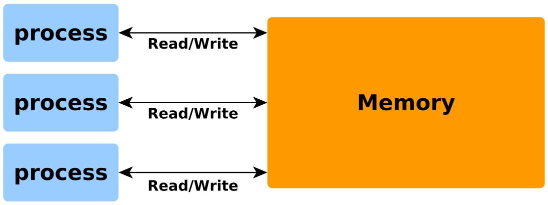


Figure 2.5 – Shared memory model.

allel program using this model because the developer must implement the communication among processes. For example, when a process sends a message with data, the receiver process must have an operation for receiving this message. Sending and receiving must be previously planned and implemented. Currently, some libraries enable the creation of parallel programs for this model, such as the well known and commonly used MPI [Pac11]. Figure 2.3.1 illustrates the message passing model with two nodes performing data communications among four processes through messages in a network.

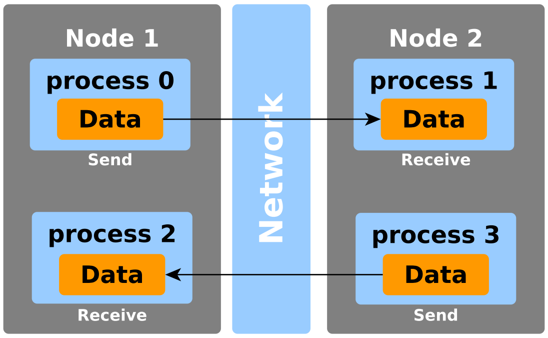


Figure 2.6 – Message passing model (Adapted from [Bar15]).

- **Hybrid** This model combines both shared and distributed memory models previously presented. It enables the creation of programs that take advantage of architectures such as clusters with multi-core or many-core computers. In the hybrid model, the processes have a shared memory inside the node that enables processes to communicate in the same node. Also, this processes can use message passing to communicate with other processes in the cluster. Moreover, it allows a process to perform intensive processing using local data and communicate with other processes to exchange this data, taking advantage of both grain and course granularity [MSM04]. Figure 2.3.1 demonstrates this model with two nodes. In this figure, processes are performing reading/writing operations on shared memory and send/receive operations to exchange information among nodes.

Also, it is possible to classify parallel programs according to the decomposition of the problem. Some problems can be split into processes in parallel by only dividing the

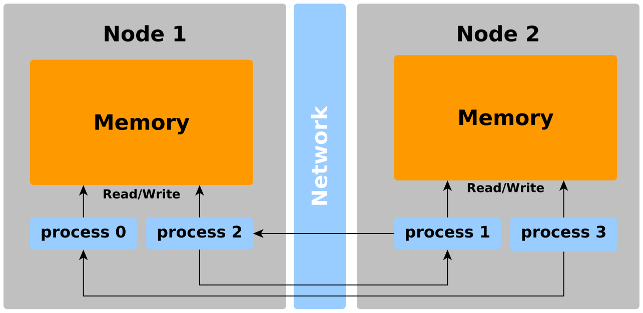


Figure 2.7 – Hybrid model (Adapted from [Bar15]).

data to run on multiple cores. On the other hand, other programs that have multiple tasks can process these in parallel processes. We can divide problem decomposition into data parallelism and task parallelism. Both models for parallel programming are explained as follows.

- **Data Parallelism:** Data parallelism model is characterized by treating the memory address space globally or performing operations on a data set in parallel. This model organizes data in a standard structure, for example, an array or cube, and tasks work in parallel on this data structure. During parallel processing, each task works on a different part of the same data structure. Usually, tasks perform the same operation on the data, for example, *to sum 3 with an array element*. However, it can perform different operations for each partition of data [Pac11].

- **Task Parallelism:** This model is most applicable for programs that can break a problem into parallel tasks. This problem can have distinct functions, loops interactions and costly parts of the code that can be performed in parallel. The objective of this model is to divide tasks among cores. If it is applied to programs with few tasks, the development becomes easier. However, if the problem has many tasks, it can become complicated to control the execution of each process, and how it writes or reads memory without interfering with the result of other tasks [Pac11].

- **Stream Parallelism:** Stream parallelism is a special kind of task decomposition that has as characteristic a sequence of tasks. This model can be imagined as an *Assembly line*, where data is received as input and a set of operations is performed. However, this processing is organized to receive multiple pieces of data and performs the operations with a time overlap, resulting in a parallel execution [Bar15].

2.3.2 Design of Parallel Programs

Parallelization is used to transform a sequential program into a parallel program to better use a parallel system. Parallelism has three main divisions: parallelism at the instruction level, data parallelism, and loop parallelism [RR13]. The systematic way to parallelize code consists of an initial computation decomposition. In this step, computation dependencies are determined in sequential code. Depending on the dependencies in a computation, it can be decomposed into subtasks. Thus, tasks are assigned to processes or threads, and these are mapped to physical processors or cores [RR13]. However, the problem may be studied to facilitate the planning and implementation of parallelism. The three typical steps that must be performed for software parallelization are:

- **Study problem or code**: It is important to know how the program works because, when parallelizing software, one must make choices that can change the final result. Moreover, the developer has to guarantee that the parallel program provides the same results as the sequential program for the expected input value. Thus, it can be easily achieved if the developer knows well the problem when developing [Bar15].
- **Look for parallelism opportunities:** The second step in parallelization is to look for parallelism opportunities. This is easy for experienced developers, but can be hard for novices. In this step, developers may use a strategy to verify the parts of the program that can be decomposed to process in parallel. The developer needs to find concurrency and decide how to take advantage of it. Also, the developer can break the program into tasks, dividing these among processors. Some decisions in this step can impact heavily on the development of the parallel version, since sometimes the program must be restructured or an entirely new algorithm must be created [Bar15].
- **Try to keep all cores busy:** When a program is parallelized, the objective is to divide the processing among the cores/processors to achieve the result in less execution time. If a program is not parallelized correctly, cores can become idle during the processing. A well developed parallelization keeps the cores busy, processing data or tasks to achieve results quickly. It is possible to measure the use of cores through the *efficiency* calculus [Bar15].

Synchronization

Determining the sequence processes perform their operations is a critical point to consider when creating a parallel program. When many processes execute in parallel, and access shared spaces in memory, it is important to analyze and examine the sequences of writing/reading. For example, if two processes read the same value stored in a memory space, and one of the processes changes this value after reading, the final result will be incorrect because the second process will read the modified value. Therefore, in this case, mechanisms must be implemented to control this sequence, making both processes read the value before the first process modifies it. However, applying this procedure can impact performance since processes stay idle waiting for a determined process to continue. Also, these mechanisms can serialize some parts of the code. Some procedures to apply control in synchronization such as lock/semaphore, synchronous communication operations, and barriers are described below [Pac11].

- **Locks/Semaphores:** A lock or semaphore is used to serialize and protect the access to determined global data or a specific piece of code. It works by setting a flag in a piece of data. So, it will be used only by a specific task. Then this unique task can safely access and modify the protected data. If another task tries to use this data or code, it will wait until the first task unsets the flag. This operation can affect the performance, but it is often required when tasks use common data or communicate using shared memory [Pac11].

- **Synchronous through communication:** This mechanism is used in parallel programs that communicate between processes. It works by performing blocking communications among processes to coordinate the processing in sequence. For example, a task that may perform an operation will receive a message from another process informing it that can proceed [Pac11].

- **Barriers:** This mechanism usually is applied to all the tasks in a parallel program. It works by creating a barrier in the code where all the tasks must be achieved to continue the processing. This enables the synchronization of threads and guarantees that all the processes execute a procedure at the same time [Pac11].

Data Dependencies

Data dependency is a concept to express when there is a dependence between the order of program operations that affects the results. It is common in shared memory models that have multiple processes accessing the same memory location. Usually, a data dependence is an inhibitor of parallelism since one operation needs to be executed after another, forcing a sequential execution. An example of data dependency can be seen in a program that needs the result of a previous computation to realize an operation. And its result will be the input of another operation [Pac11]. In shared memory architectures, data dependency can be handled by the use of synchronization mechanisms. In distributed memory, it is possible to use communication or barriers for implementing a parallel program with data dependency [Bar15].

Granularity

Granularity is a measure of the ratio of computation versus communication [Bar15]. There are two types of granularity: fine-grain parallelism and coarse-grain parallelism. Both are described bellow.

- **Fine-grain Parallelism:** This type of granularity has a small number of computations between the communications. This kind of granularity facilitates the implementation of load balancing. However, it is easy to have less performance due to the overhead generated by communication. Sometimes, with very fine granularity, it is possible that communication takes more time than processing [Bar15].
- **Coarse-grain Parallelism:** Represents processing with few communication events and a high number of computations. This kind of granularity has more opportunities to parallelize and for performance gain. However, it can become hard to handle load balancing [Bar15].

I/O

A problem that parallel programmers can face when trying to parallelize code is I/O. Usually, programs that have a lot of I/O operations can be hard to parallelize because loading or saving data processes can take more time than processing. For example, the reading of a file from the hard disk can take more time than processing it. Also, saving it can take even more time. In this case, the parallelization does not affect performance too much. Currently, some distributed file systems enable parallel I/O operations using distributed computers with replicated data. In shared memory architectures, it is possible to implement a stream processing model to process data in small chunks as it is loaded, doing it in a interpolated time [Bar15].

2.3.3 Parallel Interfaces

Several parallel programming interfaces allow and facilitate the development of parallel programs. If we limit this scope for shared-memory architectures, then it is possible to mention: POSIX Threads [NBF96], OpenMP [CJvdP07], FastFlow [ART10], TBB [Rei07], Cilk [Lei09], Intel Parallel Studio [INT15], SPar [Gri16] and many other libraries. These libraries can be used based on different problems. Table 2.1 describes the target architecture for each library and the supported parallel programming models. In this analysis, we only consider parallel interfaces that target shared memory, since this is the type of interface applied in this work.

Table 2.1 – Comparison of parallel programming interfaces.

Interface	System Architecture	Target Memory Architecture	Task Parallelism	Data Parallelism	Stream Parallelism	Annotation Based
Pthreads	Multi-core	Shared Memory	x	x	x	
OpenMP	Multi-core	Shared Memory	x	x		x
Fastflow	Multi-core	Shared/Distributed Memory	x	x	x	
SPar	Multi-core / Cluster	Shared/Distributed Memory	x	x	x	x
TBB	Multi-core	Shared Memory	x	x		
Cilk	Multi-core	Shared Memory	x	x		x
Intel Parallel Studio	Multi-core	Shared Memory	x	x		

2.4 Compilers Creation

A language can be formal or natural. A natural language is a language that human beings use to communicate, which uses a set of symbols and a set of rules that are combined to communicate information to a group that understands this. On the other hand, a formal language also uses symbols and a set of rules. However, it is designed for machines (programming languages, communication protocols, etc.). Both formal and natural language have a set of rules, called grammar [Kri09], that defines the structure and rules of this language.

A compiler is a computer program that reads a language written in a particular programming language and translates it into another language. For this operation, a set of steps is performed. Initially, a lexical analysis (or linear analysis) is performed to scan the input code and recognize each language element, creating tokens. After, a syntax analysis (or hierarchical analysis) group the tokens recognized according to a grammar specification to generate the output during rules matching. The third step in a compilation is the semantic analysis. It is a verification of the code that tries to find possible semantic errors. For example, if a declaration must receive an integer, and the user defines a float number, this phase will verify and, convert the value to an integer or report a semantic error [ASU06].

Some compilers also have an optimization phase to improve input code after code generation. This step usually recognizes possible improvements in the code and change it to simplify, improve or speed up the compilation product. Finally, the code generation is performed, which transforms the received code in another language, following transformation rules [ASU06]. This research used Flex and Bison to perform parsing in the GMaVis compiler. Both will be briefly explained next.

Flex

Flex (The Fast Lexical Analyzer) [Gao15b] is a lexical analyzer generator constructed in C/C++. A lexical analyzer can be defined as a program which recognizes patterns in text performing the lexical analysis. It is usually written and contains the patterns to be recognized in the form of regular expressions. Figure 2.8 illustrates the workflow in Flex. The input to Flex is a file containing tokens defined using regular expressions, called rules. Lex/Flex generates a C source file, lex.yy.c, which defines a routine yylex() as output. This file is compiled and linked with the -lfl library to produce an executable. When the executable runs, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code. Flex produces an entire scanner module that can be combined with other compiler modules to create a compiler.

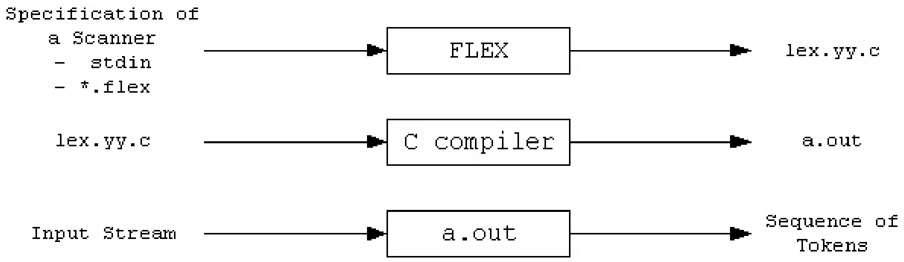


Figure 2.8 – Flex workflow (Extracted from [Gao15b]).

Flex file has a set of regular expressions for recognizing each element of the language. Its receives the source code and performs an operation for each element recognized, also described in the Flex file. This file is divided into three parts: scanner declarations, token definitions and actions, and subroutines. In the scanner declaration section, it is declared libraries, links for parsers and also definitions for specific lexical rules. The token definitions and actions section contains the regular expressions for matching language elements and generating tokens. This section also has actions that will execute when each regular expression is identified in the input source code. Finally, subroutines section may contain functions or declarations related to the actions of the second section.

Bison

Bison [Gao15a] is a general-purpose parser generator that converts a grammar description (Bison Grammar Files) for an LALR(1) context-free grammar into a C program to parse that grammar. The Bison parser is a bottom-up parser. Figure 2.9 illustrates the workflow of Bison. Bison receives a file containing the whole grammar and rules of a language. Then, it generates a C file that is compiled and enables parsing any input stream of a code. It tries to reduce the entire input to a single grouping whose symbol is the grammar's start symbol, by shifts and reductions. Bison is upward compatible with Yacc: all properly-written Yacc grammars may work with Bison with no change.

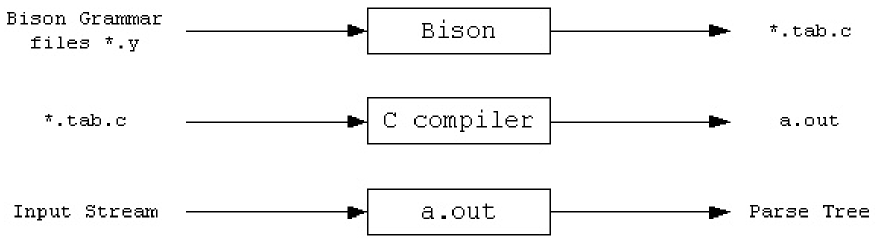


Figure 2.9 – Bison workflow (Extracted from [Gao15a]).

There are two basic parsing approaches: top-down and bottom-up. Intuitively, a top-down parser begins with the start symbol. It traces the leftmost derivation of the string by looking at the input string. When it finishes, a parse tree is generated top-down. A bottom-up parser generates the parse tree bottom-up by tracing the rightmost derivation in reverse by starting with the input string and working backward to the start symbol, for each given string.

3. RELATED WORK

This chapter presents some DSLs that were proposed to aid domain users with the creation of data visualization. Vivaldi [CCQ+er], ViSlang [RBGH14], Diderot [CKR+12], Shadie [HWCP15] and Superconductor [MTAB15] are the DSLs presented in Sections 3.1 to 3.5, and the last section contains a comparison between these DSLs and a discussion related to this work.

3.1 Vivaldi

In 2013, a domain-specific language called Vivaldi [CCQ+er] was proposed. This DSL provides volumetric processing and visualization rendering in parallel using distributed and heterogeneous architectures while providing a high-level language based on *Python*. With this DSL the user does not need to be concerned with the programming of many-core processors and GPU accelerators. It is because programming software to explore GPU clusters is hard since it requires hardware knowledge and low-level programming. Also, Vivaldi provides standard functions and mathematical operators for customized visualization and high-throughput image processing applications.

Vivaldi has been designed for ease of use while providing high-throughput performance supporting distributed heterogeneous architectures. It includes a flexible and high-level programming language, a compiler, and a runtime system. The grammar language is similar to Python, facilitating its use for domain experts who can focus on application objectives without needing to learn parallel languages such as MPI or CUDA. This DSL also abstracts the communication among processes required in parallel programming. A memory model provides a simple and unified view of memory without any data communication code required from the users. Data is replicated on different compute nodes while the runtime system performs automatic and transparent synchronization. Also, Vivaldi provides domain-specific functions and some mathematical operators used in visualization and scientific computing to allow users to write customized applications for volume rendering and processing.

The target computer architecture that Vivaldi uses is clusters of heterogeneous computing nodes (*i.e.*, CPUs and GPUs in each node) on distributed memory system. Vivaldi abstracts low-level architecture and programming details since it provides abstractions to manage computing resources and memory using a unified high-level programming environment. Thus, users do not need to have knowledge about CUDA or OpenCL to use GPU because Vivaldi generates parallel GPU code automatically. Also, Vivaldi abstracts

MPI communication between nodes and transparently handles memory through a memory manager in its runtime system. Figure 3.1 illustrates Vivaldi system overview.

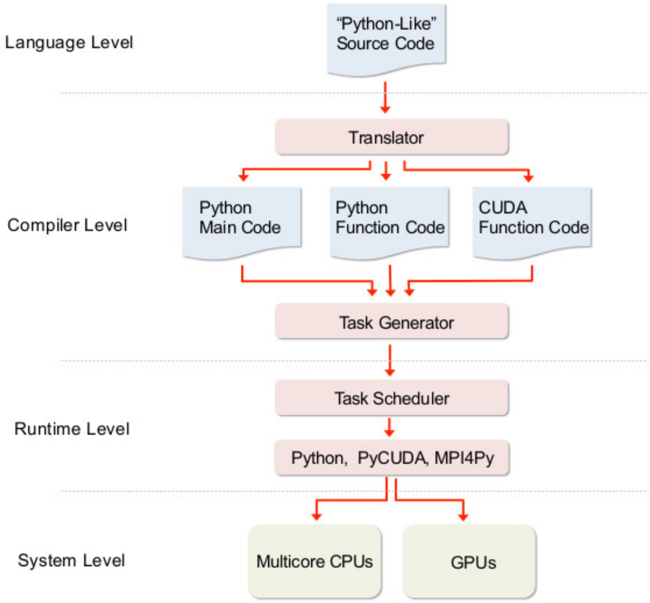
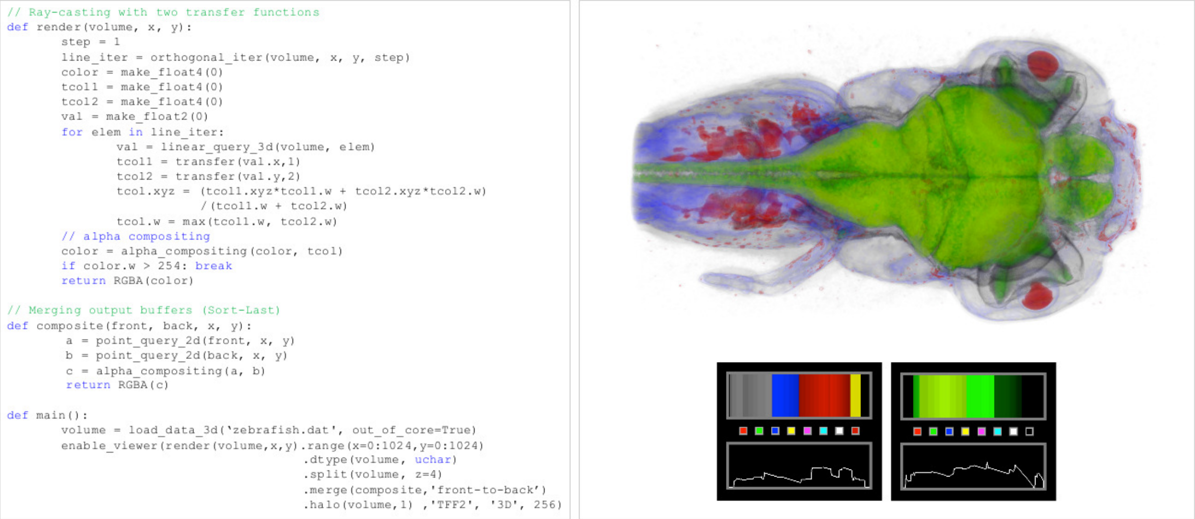


Figure 3.1 – Vivaldi system overview (Extracted from [CCQ⁺er]).

Vivaldi deals with each computing resource (*i.e.*, CPU, GPU) as an independent execution unit. Each execution unit is assigned an execution ID for processing tasks. For functions, users will define the tasks, input/output data, and task decomposition scheme. The structure of a Vivaldi program is a collection of functions where the main function is the driver that connects user-defined functions to build a computing pipeline. Therefore, each function call generates a set of tasks that are processed in parallel by available execution units. As demonstrated in Figure 3.1, the input source code is translated into different types of native source code. Then, it is saved inside Python main, Python functions, and CUDA functions. Tasks are generated based on the main function code. Finally, each task is scheduled to run on one of the execution units depending on the scheduling policy. Low-level data communication and function execution are abstracted and handled by Python, CUDA, and MPI runtime functions.

Figure 3.2 presents a Vivaldi code example and its output. The output visualization is a volume rendering of a juvenile zebrafish performed on a GPU cluster. It uses an input task generation scheme to split and to transfer functions. Brain and nerve systems are rendered in green while other regions are in gray, blue, and red.

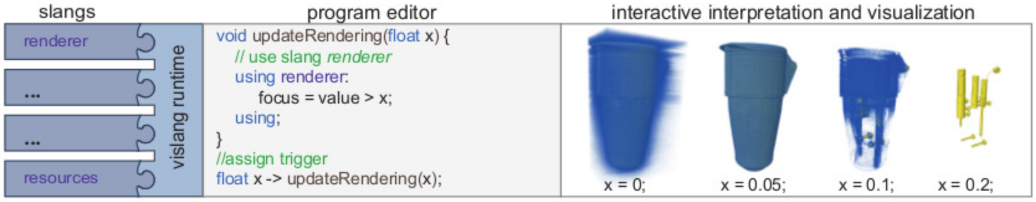


```
// Ray-casting with two transfer functions
def render(volume, x, y):
        step = 1
        line_iter = orthogonal_iter(volume, x, y, step)
        color = make_float4(0)
        tcol1 = make_float4(0)
        tcol2 = make_float4(0)
        val = make_float2(0)
        for elem in line_iter:
                val = linear_query_3d(volume, elem)
                tcol1 = transfer(val.x,1)
                tcol2 = transfer(val.y,2)
                tcol.xyz = (tcol1.xyz*tcol1.w + tcol2.xyz*tcol2.w)
                            /(tcol1.w + tcol2.w)
                tcol.w = max(tcol1.w, tcol2.w)
        // alpha compositing
        color = alpha_compositing(color, tcol)
        if color.w > 254: break
        return RGBA(color)

// Merging output buffers (Sort-Last)
def composite(front, back, x, y):
         a = point_query_2d(front, x, y)
         b = point_query_2d(back, x, y)
         c = alpha_compositing(a, b)
         return RGBA(c)

def main():
        volume = load_data_3d('zebrafish.dat', out_of_core=True)
        enable_viewer(render(volume,x,y).range(x=0:1024,y=0:1024)
                                        .dtype(volume, uchar)
                                        .split(volume, z=4)
                                        .merge(composite,'front-to-back')
                                        .halo(volume,1) ,'TFF2', '3D', 256)
```

Figure 3.2 – Vivaldi code and output (Extracted from [CCQ+er]).

3.2 ViSlang

ViSlang [RBGH14] is an interpreted DSL elaborated to offer the possibility of creating scientific visualizations for facilitating its creation by scientists and avoiding the difficulties of handling a low-level language. A great contribution offered by ViSlang is the possibility of extending the DSL using *slangs*. This feature allows the implementation of others DSLs to address other issues that are within the domain. Figure 3.3 illustrates the use of *slangs*.



```
void updateRendering(float x) {
  // use slang renderer
  using renderer:
     focus = value > x;
  using;
}
//assign trigger
float x -> updateRendering(x);
```

Figure 3.3 – Slangs in ViSlang (Extracted from [RBGH14]).

ViSlang is a system that decreases the cost of developing new DSLs since it integrates multiple DSLs in one solution to support their flexibility. Figure 3.3 provides an overview of ViSlang design. ViSlang has a library and an execution environment with support for an extension mechanism. By integrating it into a visualization system, the runtime can execute commands and change the behavior of the visualization. This runtime performs as a unified programming interface. Internally, DSLs that extend ViSlang are named slangs. It works in the following mode. User input is received and executed by an interpreter. If a command starts with the keyword *using*, it will be transmitted to the corresponding slang. Figure 3.3 demonstrates an example code that makes use of the slang renderer. It offers a DSL that allows properties of data mapping to visual elements. This example gives an overview of the ViSlang runtime system where multiple DSLs are combined using common data processing and visualization modules. ViSlang currently focuses on processing and visualization of static volumes. This is not a restriction of the system design but the current implementation. However, one of the major design goals of ViSlang is its extensibility and ability to support other data structures.

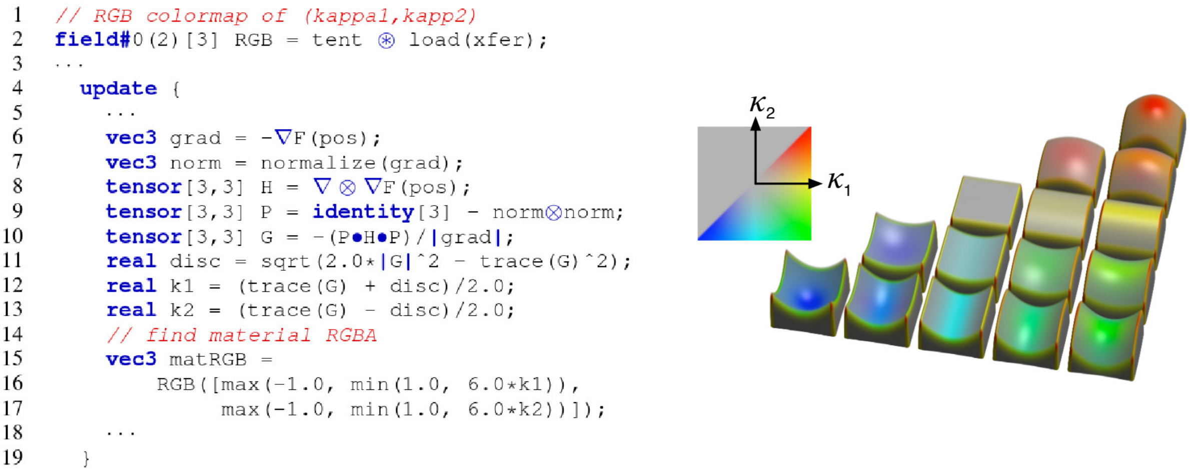
3.3 Diderot

Diderot [CKR+12, KCS+16] is a DSL that simplifies portable implementation of parallel methods of biomedical image analysis and visualization. Image analysis extracts quantitative or geometric descriptions of the image structure to characterize specific properties of the underlying organ or tissue. Visualization combines measurements of local image data properties with elements of computer graphics to qualitatively depict structures via rendered images. Diderot supports a high-level model of computation that supports continuous tensor fields.

Diderot permits programmers to express algorithms directly regarding tensors, tensor fields, and tensor field operations, using the same mathematical notation that is used in traditional vector and tensor calculus. Diderot objective is to be useful for prototyping image analysis and visualization methods in contexts where a meaningful evaluation of the methods requires its application to real image data. Besides, the real data volumes are of a size that requires efficient parallel computation. With its support for high-level mathematical notation, Diderot is also useful in educational contexts where the conceptual transparency of the implementation is of primary importance.

Figure 3.4 presents an example of a simple, direct volume rendering code. This example code also demonstrates a field to implement a color assignment function (*i.e.*, the RGB field). This example shows this using bilinear interpolation, which is provided by the kernel. The output image from this code also includes an image of the bivariate colormap function.

The DSL compiler has approximately 19,000 lines of code that are organized in three phases: the front-end, optimization and reducing, and code generation. Different target languages perform code generation: sequential C code with vector extensions, parallel C code, OpenCL, and CUDA.



```
// RGB colormap of (kappa1,kapp2)
field#0(2)[3] RGB = tent ⊛ load(xfer);
...
  update {
    ...
    vec3 grad = -∇F(pos);
    vec3 norm = normalize(grad);
    tensor[3,3] H = ∇ ⊗ ∇F(pos);
    tensor[3,3] P = identity[3] - norm⊗norm;
    tensor[3,3] G = -(P•H•P)/|grad|;
    real disc = sqrt(2.0*|G|^2 - trace(G)^2);
    real k1 = (trace(G) + disc)/2.0;
    real k2 = (trace(G) - disc)/2.0;
    // find material RGBA
    vec3 matRGB =
        RGB([max(-1.0, min(1.0, 6.0*k1)),
             max(-1.0, min(1.0, 6.0*k2))]);
    ...
  }
```

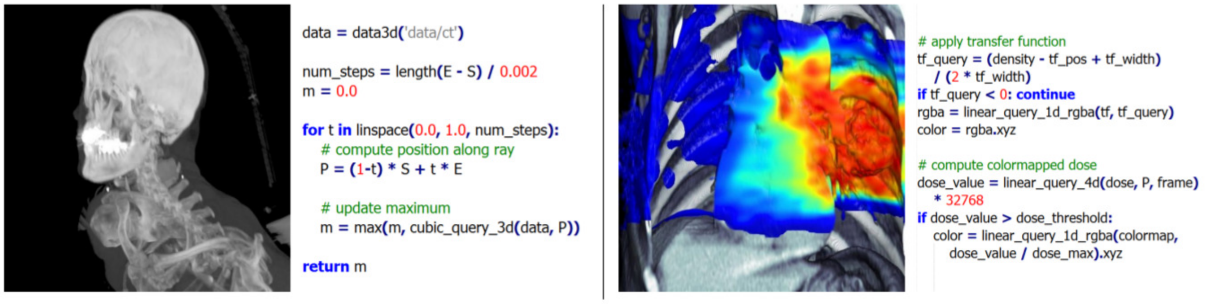
Figure 3.4 – Diderot code example using tensors (Extracted from [CKR+12]).

3.4 Shadie

Shadie [HWCP15] is a GPU-based volume visualization framework built around the concept of shaders: self-contained descriptions of the desired visualization written in a high-level Python-like language that can be written by non-experts, often in 10-20 lines of code. Type inference and source-to-source translation to efficient CUDA code allow for interactive framerates. The concepts of rays, transfer functions, lighting computations, or cut-planes are unknown to the core renderer; instead, these are purely features of the shaders. Furthermore, traditional volume rendering can be easily combined with ray-traced implicit surfaces or maximum-intensity projections within the same image. Any multidimensional datasets can combine in arbitrary ways to produce the final visualization.

The main objective was to provide a DSL for medical proposes and help professionals with little programming knowledge for GPUs to render images using the computing power of parallel architectures for performance gain through the utilization of a friendly tool. A Shadie program can be thought of as a sequence of statements that, given a ray segment, computes the color of the pixel. Also, a Shadie program usually begins with some parameter definitions, where the parameters are either datasets (1D, 2D, 3D or 4D) or floating point constants (scalar or vector), which are bound to the GUI and modifiable on the fly [HWCP15]. Figure 3.5 presents two example codes and their output visualizations.

For the development of language, some goals were considered. First, the system must be able to combine multiple volumetric datasets, commonly available for a single patient. Therefore, it supports 4D data sets and also time variations. Second, the target audience is scientific domain users, such as doctors and physicists who are knowledgeable in math and programming languages such as MATLAB and Python, but not low-level program-



```
data = data3d('data/ct')

num_steps = length(E - S) / 0.002
m = 0.0

for t in linspace(0.0, 1.0, num_steps):
   # compute position along ray
   P = (1-t) * S + t * E

   # update maximum
   m = max(m, cubic_query_3d(data, P))

return m
```

```
# apply transfer function
tf_query = (density - tf_pos + tf_width)
   / (2 * tf_width)
if tf_query < 0: continue
rgba = linear_query_1d_rgba(tf, tf_query)
color = rgba.xyz

# compute colormapped dose
dose_value = linear_query_4d(dose, P, frame)
   * 32768
if dose_value > dose_threshold:
   color = linear_query_1d_rgba(colormap,
      dose_value / dose_max).xyz
```

Figure 3.5 – Shadie example (Extracted from [HWCP15]).

ming. Hence, it tries to use an interface that is similar to the existing syntax in high-level languages to facilitate the use of DSL for this users. Also, a goal was to get satisfactory performance through the exclusion of issues could influence it. Also, the output visualization is kept in a single file, allowing it to be easily manipulated. Shadie also offers hardware abstraction since the system get the maximum advantage of the GPU while it is transparent to the user [HWCP15].

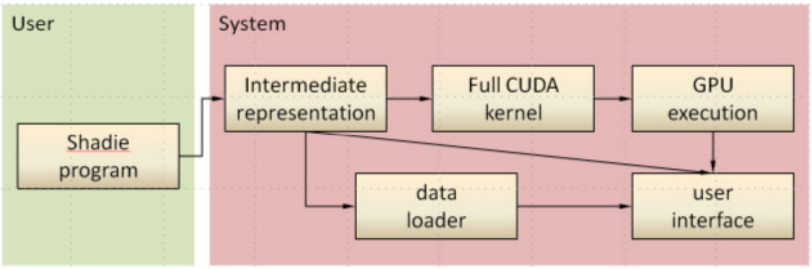


Figure 3.6 – Shadie compilation workflow (Extracted from [HWCP15]).

Finally, Shadie presents a simple workflow, as shown in Figure 3.6. There is an intermediate representation generated based on the Shadie program. It generates both CUDA kernels to execute in GPU and the data loader, providing the final visualization in the user interface.

3.5 Superconductor

Superconductor [MTAB15] is a high-level language for creating visualizations that aim to interact with large amounts of data. It seeks to provide three design axes through the use of different programming languages. Other DSLs, to take advantage of the wide availability of data, have to choose between scale and interactivity. In some cases it requires using large monitors to show visualization that uses a large quantity of data, often confusing users with many visual elements. In this context, Superconductor is presented to address three design axes providing scale, interactivity and productivity as described in the following goals.

- Scale: Some visualizations need to support a huge number of data points. For example, Matlab and Circos are used for static visualizations of large data sets.
- Interactivity: Interactions should be within 100ms and animation may achieve 30fps. For example, JavaScript and its libraries such as D3 [2] are used for animating data and orchestrating interactions with users.
- Productivity: Programming must be performed using a high-level interface or language like JavaScript. Users can personalize visualizations using these interfaces.

Most languages used to create visualizations support one or two of the goals mentioned, but superconductor seeks to address all the three. Figure 3.7 presents the three dimensions explored by Superconductor.

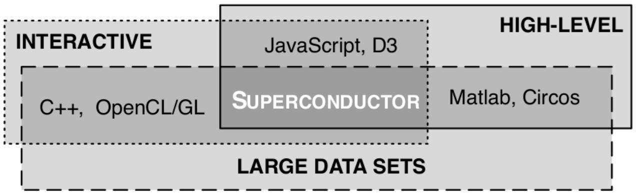


Figure 3.7 – Superconductor: design axes for visualization languages (Extracted from [MTAB15]).

As explained, Superconductor uses C++ and OpenCL/GL programming languages to provide interactivity and also works with large volumes of data. In addition to providing high-level abstractions, it uses Javascript and D3 for interactive visualizations. Matlab and Circuses provide both a high-level interface and work with large volumes of data. However, they only generate static visualizations. Superconductor combines the benefits of these languages to meet these three objectives.

This DSL has an architecture that contains three stages of execution as illustrated in Figure 3.8. Initially, the input data is filtered and configured according to the visualization. Then, selectors are defined in the high-level programming interface to associate colors, sizes, and shapes for each data type. Then, the size and position of each display element is adjusted. Finally, a rendering module receives the data and uses a rendering API to perform this in GPU.

Superconductor [MTAB15] provides a set of generic big data visualization features. It does not focus on offering a particular visualization type. Also, it allows the user to personalize the visualization by changing visual elements as illustrated in Figure 3.9. Superconductor is more robust than our work. However, users have to build their visualizations from

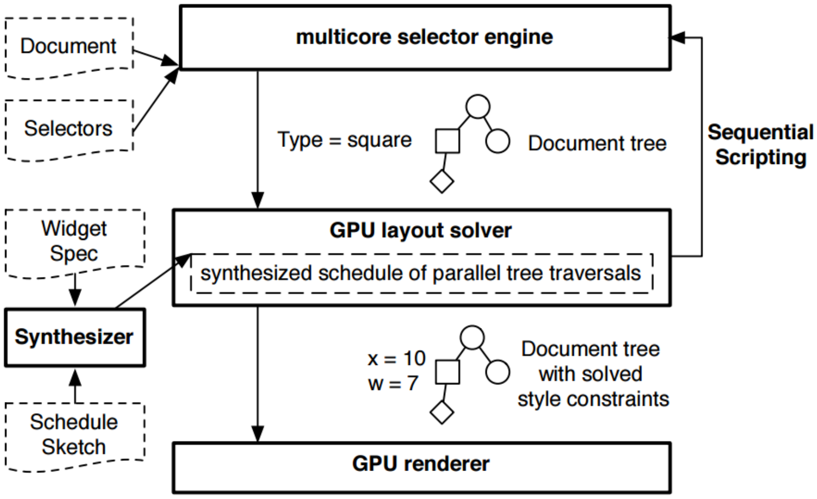


Figure 3.8 – Superconductor architecture (Extracted from [MTAB15]).

the scratch, thinking about how graphic elements will appear and specifying how data will be mapped for visual elements. In our DSL, the goal is to abstract programming language interaction using a user specification approach through a description language.

```
class Root
  children c : [ Node ]                 //layout tree schema
  input width = 300;                    //default overridable by selectors
  var height = width;
  var render = paintSquare(0,0,votes[last],votes[last])
  var votes[−1] = 0
  votes[i] = votes[i − 1] + c[i].votes
  ...
class Leaf : Node
  input count = 0, min = 0.0, max = 1.0
  var votes = turnout ∈ (min,max) ? count : 0
  ...
schedSketch: parPreorder _ ; parPostorder _ ; _ ; _ ; _
```

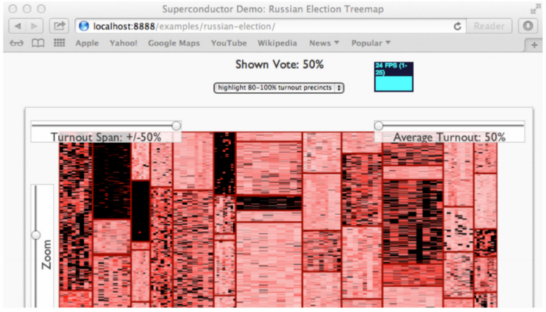


Figure 3.9 – Code generation in Superconductor (Extracted from [MTAB15]).

3.6 Synthesis

Previous sections presented some DSLs that were proposed to facilitate the creation of data visualization for specific domain users. In this section a comparison of these DSLs is done, considering characteristics as development abstractions, supported visualization techniques, and parallel programming aspects. We also inserted GMaVis in this comparison to highlight the differences among previous work and the proposed DSL.

Initially, we compared and evidence that each DSL is made to provide visualizations used in the target domain. Table 3.1 presents the domain that each DSL focus on offering visualizations. Vivaldi, ViSlang, Diderot, and Shadie focus on the generation of volumetric visualizations and do not provide geospatial data visualization maps creation. Superconductor [MTAB15] offers a set of generic big data visualization features. It is not designed for a specific visualization type. Also, it allows the user to personalize the visualization by changing visual elements. Compared to our work, Superconductor is more robust. However, users have to build their visualizations from scratch interacting with low-level programming, while we abstract programming language interaction using a user specification approach. Thus, analyzing the domain and the supported visualizations, these DSLs have different approaches from what we propose in GMaVis.

Table 3.1 – Domain of each DSL.

DSL	**Domain**
Vivaldi [CCQ+er]	Scientific/Volume Visualization
ViSlang [RBGH14]	Scientific/Volume Visualization
Diderot [CKR+12]	Image Analysis and Medical Visualization
Shadie [HWCP15]	Medical Visualization
Superconductor [MTAB15]	General Interactive Visualization
GMaVis [LGMF15]	Geospatial Data Visualization Maps

A common goal in these DSLs is offering high-level interfaces for abstracting programming complexities. Most of these complexities are about low-level specifications, rendering, and parallel processing. Moreover, if we analyze complexity abstractions considering the phases of visualization creation described in Section 2.2 we can highlight that the DSLs abstract only complexities from *visual mappings* and *view transformation* phases. Table 3.6 presents the phases that each DSL abstracts.

Table 3.2 – Complexities abstraction in each visualization creation phase.

DSL	**Data Pre-processing**	**Data to Visual Mappings**	**View Transformations**
Vivaldi [CCQ+er]		X	X
ViSlang [RBGH14]		X	X
Diderot [CKR+12]		X	X
Shadie [HWCP15]		X	X
Superconductor [MTAB15]		X	X
GMaVis [LGMF15]	X	X	X

Through this comparison, it is possible to verify that only GMaVis abstracts complexities from *data pre-processing* phase. This phase includes a great deal of work required to create a visualization, which involves handling data to select, classify and format it. If these data are generated using different formats than those expected by the visualization tool/library, users will have additional work to format. Some DSLs were built as internal DSL, which means that users can use the general purpose language to deal with data before inserting it in the mechanisms offered by the DSL. However, users will need to create code to deal with this data. This phase becomes even harder if users are dealing with huge data files. For example, when a domain user needs to generate a visualization of data stored in a file with 30 GBytes. If the computer used have only 4 GBytes of memory, this user will have to split this file to read and process. Furthermore, if the file has 500GB or 1TB, it can become very slow, often requiring parallel processing to speed it up. Therefore, it implies more effort for development.

With data production increasing exponentially, it is easy to achieve Gigabytes or Terabytes of data. Thus, GMaVis offers complexity abstraction in the *data pre-processing* phase, through the specification of filters and classification of data. Also, parallel programming is completely abstracted from users. Moreover, it abstracts complexities in *data to visual mappings* and *view transformations*, since users will simply specify a few details about the data visualization map using a descriptive language.

The DSLs presented in this work apply parallel processing to speed up processing. Some DSLs apply parallel programming in the rendering of volumes and for generating visualizations. Others use this in interaction, through the use of GPU processing. In our case, GMaVis provides parallel processing in the *data pre-processing* phase, for filtering, classifying and selecting data. Table 3.3 compares the application of parallel programming among the DSLs.

Table 3.3 – Parallel processing in each visualization creation phase.

DSL	**Data Pre-processing**	**Data to Visual Mappings**	**View Transformations**
Vivaldi [CCQ+er]		X	X
ViSlang [RBGH14]		X	X
Diderot [CKR+12]		X	X
Shadie [HWCP15]		X	X
Superconductor [MTAB15]		X	X
GMaVis [LGMF15]	X		

We can highlight that almost DSLs apply parallel processing in *data to visual mappings* and *view transformations* phases. Also, these DSLs do not apply parallel processing in *data pre-processing* phase. However, GMaVis applies parallel processing in the *data pre-processing* phase but does not apply in the *data to visual mappings* and *view transformations* phases. It is because GMaVis visualizations do not require high processing to create and display visualization. The costlier processing is the *data pre-processing* phase, which includes loading, parsing, classifying, filtering, selecting and formatting data. Also, Vivaldi, ViSlang, Diderot, Shadie, and Superconductor provide visualizations in 3D and volumetric images that require high processing to enable both interaction and visualization creation.

4. GMAVIS: A DSL FOR GEOSPATIAL DATA VISUALIZATION MAPS

During background studies, we did not found a tool that provides data processing and easy ways for domain users to create a visualization. For these users, it became a hard task since some of them may have a low-level knowledge in computer programming [WWA15, ZCL13]. For example, to create a data visualization, users have to deal with data processing, analysis, and visualization algorithms/technologies. Some domain users do not have this knowledge about computer programming and may have a hard time generating a simple visualization. Also, users that need to generate a simple map with markers may know at least HTML and JavaScript. Additionally, this user will need to learn how to use a library, like Google Maps API or OpenLayers. If this user is handling a huge dataset, this task becomes harder since a third software or tool to process, select and filter this data will be required.

Therefore, we decided to develop GMaVis, an external DSL that provides an automatic generation of data visualizations for large-scale geospatial data. This DSL provides a high-level interface through a description language that can receive additional visualizations in future. Since it abstracts complexities from computer programming issues, such as functions, variables, methods and web development issues, it is more accessible for users with low-level knowledge of computer programming to create data visualizations and analyzes data.

The first section of this chapter introduces GMaVis, presenting its main advantages, characteristics, context, environment, and explaining how it interacts with users. Also, this section briefly presents the compiler modules. Section 4.2 then formalizes GMaVis' grammar and language, explaining the possible declarations, blocks and values. Finally, Sections 4.3 and 4.4 briefly describe the function and objectives of GMaVis' internal modules.

4.1 The Proposed DSL

To facilitate the creation of visualizations for large-scale geospatial data, we propose GMaVis, an external DSL to provide a high-level specification language [LGMF15]. The goal is to be as close as possible to the domain vocabulary, supporting a suitable and friendly language syntax. It has a description language with a high level of abstraction that enables users to express filter, classification, data format, and visualization details specification. This language has limited expressiveness which makes easy for the user to learn this.

This DSL abstracts many programming and data processing aspects. Instead of creating a visualization using general purpose languages that require some computer programming knowledge, users can specify the desired visualization and how the data must be processed in only a few lines of code. The DSL performs all data parsing, processing, and classification, generating the visualization. Moreover, this DSL abstracts the transformation phase, as explained in Chapter 3 and as shown in Figure 4.1 where it is possible to see the scope that traditional visualization tools abstract from users. Figure 4.1(a) demonstrates traditional visualization tools that simply abstract the visualization generation phase from users. Data pre-processing is not included in the scope since features are not provided to improve users experience in this aspect. The proposed DSL, illustrated in Figure 4.1(b) abstracts the whole pipeline for creating a visualization because it offers data pre-processing mechanisms. Therefore, users do not have to preprocess data manually or use external tools.

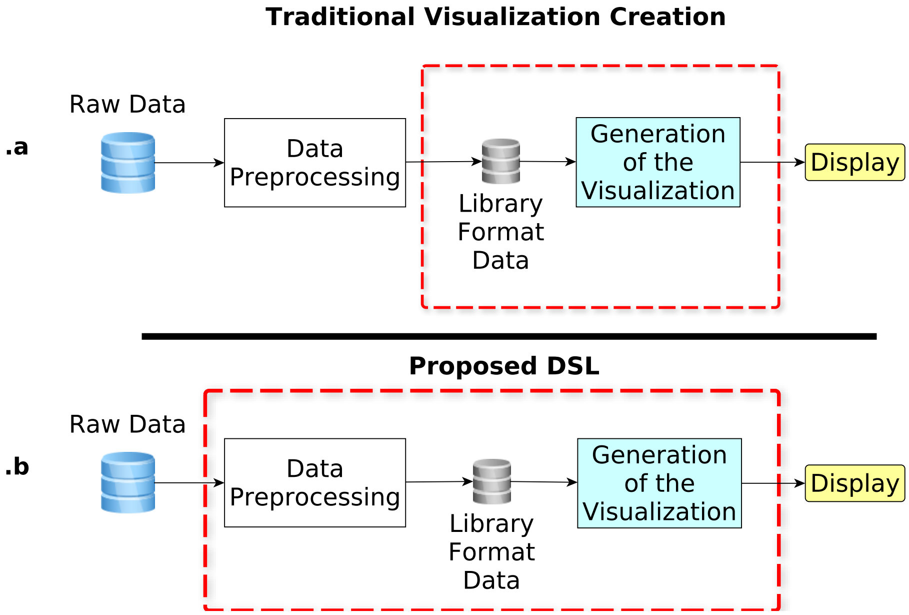


Figure 4.1 – Data visualization creation comparison.

Users of this DSL instead of other tools and libraries as Google Maps API, Leaflet and OpenLayers will have some advantages. First they will not have to know programming aspects like functions, variables, methods and any other web development issue. Second, the user will have a data processing that empowers data filtering, cleaning, and classification automatically. This DSL provides the same operations when working with huge files, because it offers an optimized file loading in memory that allows one to open files bigger than the RAM memory available in the equipment. The third advantage is that this DSL is not linked with a host language, and its interface is extremely close to the user domain. Furthermore, the parallel programming in this DSL is completly abstracted. Users will have a parallel data preprocessor without being required to implement any parallel code. This is possible because we use another DSL to generate parallel code.

GMaVis can be used through a compiler which receives GMaVis source code and datasets, process it and generates the visualization. GMaVis compiler performs code generation in two phases to process data before the visualization creation. The compiler and the whole code generation process is explained in Chapter 5.

This DSL is a description language where users specify the visualization, filters, classification, and data format. This code is received by the compiler that parses it and depending on the declarations, generates the visualization. Some internal processes are performed during processing of GMaVis code. First a data preprocessor is generated which has all of the code to parse and process the input data, filter, classify and generate an output with only the data that will be used in the visualization. Also, it creates the visualization generator to receive the output with selected data and generate the data visualization file. Thus, we internally divided our DSL into three internal parts as following described. Figure 4.2 illustrates each part.

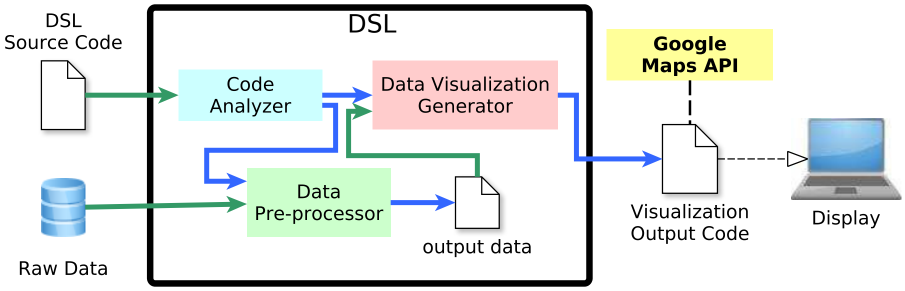


Figure 4.2 – DSL enviroment.

A Code Analyzer was built inside this DSL compiler. It is the module that receives the DSL source code as input to do the lexical analysis, parsing, semantic analysis, and code generation. These steps are performed by a compiler constructed in C/C++ using Flex and Bison. Two modules are generated in this phase, one is called `data preprocessor` and another is `data visualization generator`. The data preprocessor module enables this DSL to open and parse files bigger than the memory available, and process them for filtering and classifying their data. Then, this module stores an output file with only data used in the data visualization. On the other hand, data visualization generator generates the HTML file with the visualization code in Java-script and Google Maps API library using the output data from data preprocessor and information parsed in the compiler. Also, we can implement other types of visualization in the future, using other libraries. This architecture enables the creation of a visualization using a small source code and data as input. In the following sections, each part of this architecture will be detailed.

4.2 Interface

This DSL contributes by providing a high-level interface for processing, filtering, classifying data and creating data visualization maps. It enables the creation of a data visualization using only a few lines of code. We created GMaVis with a description language for simplifying the implementation of a new data visualization. It has its own grammar, thus, we can classify it as an external DSL. This decision was made because an external language simplifies the use since we have the freedom to create an interface similar to natural language. Being an external DSL, it only enables the creation of code to specify a supported data visualizations, since it has a limited expressiveness. However, it is easy to learn and use GMaVis, since it has a short grammar, close to the domain vocabulary. Therefore, the user will not need to worry about other characteristics of host languages, such as functions, variables, and low-level programming, as in internal DSLs or general purpose languages.

This DSL language consists of blocks, declarations, properties, and values. A block is defined as a keyword followed by an '{' character, a list of declarations and a '}' character. A declaration has a property and value. All GMaVis properties are formed by keywords and a colon *(':')* character. Finally, values can be literals (strings, characters, integer or float numbers), logical expressions, fields specifications, or keywords. Each declaration accepts some specific values that are checked during compilation.

GMaVis has three main elements specified globally in the source code. The first element consists of a visualization declaration which specifies the type of data visualization to be created. The second element is a *settings block* with declarations about visualization details such as latitude/longitude fields, size, and title. Finally, a data block with declarations used in data processing such as filters, classification, data structure and delimiters must be declared.

The goal is to boost this DSL with other types of data visualization. It was built using generalizations for adding new visualization techniques in the future. The specification of the code starts with a visualization declaration informing the visualization type name as a value. With this, we can implement other types of data visualization in the future, and users can change the type of visualization easily by just changing a few lines of code.

The EBNF for this grammar can be found in appendix B. Also, details about the parsing performed and compilation is explained in Chapter 5.

4.2.1 Keywords

This DSL has some reserved words that are recognized in the lexical analysis and give meaning to the grammar. They must be used in the right place to effectively express the code and describe the visualization. The following tables describe each keyword of GMaVis. Table 4.1 specifies the block names that are used before the (’’) characters in a block declaration. The following keywords, in Table 4.2, are properties, which are used in declarations followed by a colon (’:’). Table 4.3 describes some keywords used in value specification. Table 4.4 describes the keywords used in operators. These are used inside logical operations and become the value of a declaration. Finally, Table 4.5 illustrates the values to express the a type of visualization used in a visualization declaration.

Table 4.1 – GMaVis block reserved words.

Keyword	**Description**
data	Contains all the data declarations.
classification	Used to declare classification rules.
structure	Names a block with data structure declarations.
visualization-settings	Determines details for the data visualization.

Table 4.2 – GMaVis properties reserved words.

Keyword	**Description**
class, filter	Specifies a logic to select/classify data.
date-format	Specifies the date format used in input file.
delimiter, end-register	Used to inform data delimiters.
file	Receives the location of files.
latitude, longitude	Specifies the fields containing geo-positioning.
marker-text	Considered for specifying the text of markers.
page-title	Receives the visualization title information.
size	Used for specifying the visualization size.
visualization	Receives the visualization type name.

Table 4.3 – GMaVis value reserved words.

Keyword	**Description**
field	Represents a field when used with properties.
full, medium, small	Values to express sizes.

Table 4.4 – GMaVis operator reserved words.

Keyword	**Description**
and, or	Logical operators to join values ($\wedge$ and $\vee$).
contains	Used to verify the existence of an object inside of another ($\in$).
different, equal	Used to apply equality operations ($\neq$ and =).
greater, less	Express a logical operation of size ($>$ and $<$).
is, than	Determines the relationship between two objects.

Table 4.5 – GMaVis visualization type reserved words.

Keyword	Description
markedmap	Used to create a markedmap visualization.
heatmap	Specifies that a heatmap must be created.
clusteredmap	When this value is informed it expresses the wish to create a clusteredmap.

4.2.2 Values

In GMaVis, most of the information expressed to create the visualization, filters, classes, inform fields and the data structure is given with values. These values are used together with properties to form a declaration. It is possible to use literals (integer, float, characters and strings), logical expressions and field specifications as values. However, users must provide them with the appropriate property to generate a declaration. Some declarations do not accept determined types of values, for example, a marker-text property cannot receive a logical expression. Alternatively, a filter property cannot receive a literal float.

Literals

GMaVis contains the option of using fixed constant values such as literals. Other values are formed by keywords and these literals are therefore crucial for this DSL. These literals can be used alone in some declarations, but also are used to express a field specification or logical operation. The accepted literals in this DSL and its regular expression for recognition are described bellow.

Literal Integer This type of literal comprehends integer numbers. It is recognized in the lexical analysis using the following regular expression.

$$[0-9]+ \tag{4.1}$$

Literal Float Float literals are related to float numbers. The following regular expression is used to recognize this type of value in the source code.

$$[0-9]+\backslash.[0-9]+ \tag{4.2}$$

Literal Character A literal character comprehends single characters inside of single quotes. It is recognized with the regular expression described bellow.

$$'.' \tag{4.3}$$

Literal String This represents a combined set of characters, escape sequences, ponctuation and any characters. It is recognized with the following regular expression.

$$\backslash''(\backslash\,\backslash\,.|[\,\hat{}\,\backslash\backslash''])^{*}\backslash'' \tag{4.4}$$

This DSL does not require the use of other literal types, such as booleans, to express the description of supported visualization types. However, it is possible to implement other literal types in the future if a novel visualization type were to require this.

Logical Expressions

Logical expressions were required in this DSL for specifying filters and classes of data. Logical expressions have one or more logical operations. These logical operations can be combined using OR or AND operators. GMaVis has eight logical operations implemented that can be combined to express a logical expression in a filter or class declaration. Figure 4.3 illustrates the possible logical operations in GMaVis using *x* for fields positions, and *y* and *z* to express literal values. This also demonstrates the possibility of combining operators through a link with AND and OR operators.

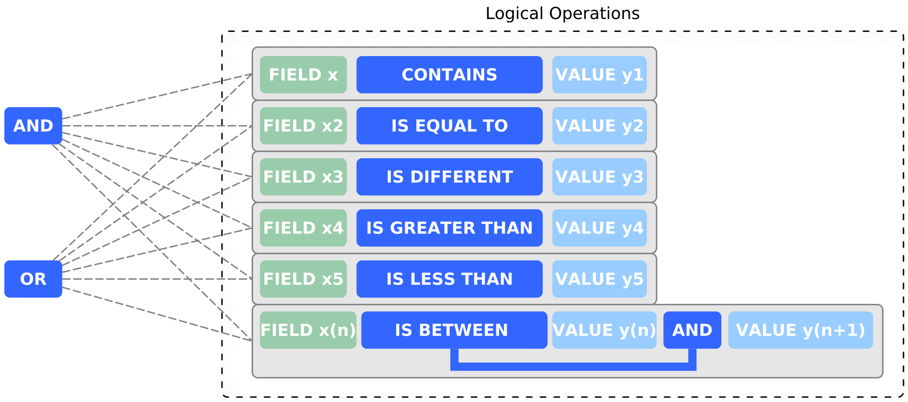


Figure 4.3 – Logical operators to filter and classify.

The structure of a logical operation has a field specification followed by a logical operator. Most operators receives a string value or a field representation. Moreover, the operator `is between` has a structure that includes an AND operator to express that the field value must be among the values specified before and after it.

The main objective when using these logical expressions is to express a condition to be applied to data registers. If the logical expression is determined to be true when applied to a data register, it will be used. Otherwise, it will not. Therefore, logical expressions can be used in filtering to select registers or in classes to input registers into a class.

Field Specification

The specification of fields enables the data preprocessor to locate data inside the dataset for manipulation. Considering that most raw datasets have a structure with delimiter characters to separate data in an organized way it is possible to specify each field using literal integers. Figure 4.4 illustrates the identification of a field number in a simple dataset.

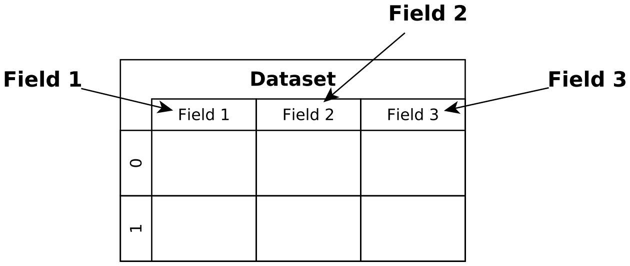


Figure 4.4 – Field value in dataset.

In the grammar, a field specification has a `field` keyword followed by a literal integer. These specifications are used in many declarations, such as marker-text, latitude and longitude, filters, and classes. These fields are recognized during data pre-processing through the data split using the delimiters specified in user source code.

Size Specification

Size specifications are values that use the keywords `small`, `medium` and `full` to express a size. They are used in `size` declaration to specify the size for a visualization in a HTML page. For example, if a `full` value is specified for the visualization, this will occupy the whole page. Otherwise, other values will adjust the visualization for other sizes. If a user with more experience intends to modify the HTML page to insert some text or even insert the visualization in a website, they can use `small` or `medium` values to occupy 50% or 80% of the page. Moreover, if the user is not satisfied with the size and prefers to adjust it for another size, the width and height values can be changed in the generated HTML file.

Visualization Type Specification

GMaVis offers the creation of three different visualization types: *markedmap*, *clusteredmap*, and *heatmap*. To specify in the source code which type of visualization must be created, a visualization type specification is used. Table 4.5 describe the three possible keywords allowed to express a visualization type.

4.2.3 Declarations

Figure 4.5 demonstrates the structure of a declaration using an example of visualization specification. Its structure is formed by a property keyword (described in table 4.2, followed by a colon (':') character and a value (described in Subsection 4.2.2). Finally, a semicolon is used to indicate the end of the declaration. Most declarations of GMaVis are used inside blocks. The only declaration that excepts this rule is the visualization specification since it is created in the global scope of the code.

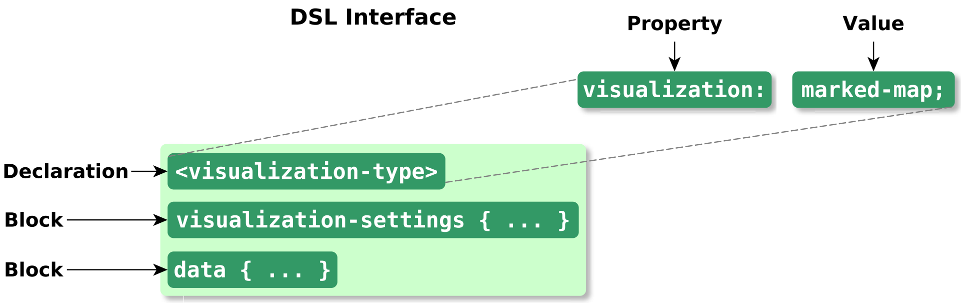


Figure 4.5 – GMaVis interface elements.

GMaVis has a set of declarations used to express filters, classes, fields for specific proposes, visualization details, data structure and location, among other information. These declarations are described as follows. A semicolon is required at the end of each declaration. It will not be mentioned in the next declaration structure descriptions, but it must always be inserted at the end of a declaration.

Filter Declaration. User can construct filters using logical expressions. A filter is declared using a *filter* property and a logical expression as a parameter. Bellow the structure of a filter declaration, using a logical expression as a specification is described.

```
filter: <logical-expression> ;
```

Since a filter declaration uses logical operators to combine one or more information about fields, it can be used to select data and display only information that is useful in the visualization. For this, we use some logical operators, that when true use the register for the data visualization adding this to the output.

Class Declaration. This DSL also allows the creation of classes to select a set of registers from a dataset and differentiate them from others in the visualization. To do so, the sub-block classifications were created to allow the user to specify logical expressions for the data that will be classified. A class declaration structure has a keyword `class` followed by a colon (':') character and a logical expression, equal to the one used in filters. This logical expression will be applied in all the dataset registers. When it is found true, the register will be inserted in the class.

```
class : <logical-expression> ;
```

Users can also inform an alias for each class. In this case, a legend using the classes names will be generated.

Latitude/Longitude Declaration Geospatial data visualizations require an important type of data that represents position. GMaVis has two declarations to receive this information through of a value with field specification. Both latitude and longitude declarations are required to generate the visualization. Without this information, it is not possible to insert data in a map visualization. The structure of both is the same, except for the keyword. It is a keyword, `latitude` or `longitude`, a colon (':') character followed by a field specification value.

```
latitude : <field-specification> ;
```

```
longitude : <field-specification> ;
```

These declarations enable the data preprocessor to insert latitude and longitude information directly in the output code for selected data.

File declaration File declarations are used to express the path to a file with data. It is possible to have an unlimited quantity of files in a source code since a dataset can be split into a set of files with data. This declaration structure begins with a keyword `file`, followed by a colon (':') character and a literal string. In this literal string, users may inform the path to the file that will be processed. The structure is described below.

```
file : <literal-string> ;
```

Delimiter/End-register Declaration Users can specify the characters used as delimiter in the data fields through the `delimiter` declaration. They can also inform the character used to mark the end of a register. For example, in a CSV (Comma Separated Values) file, the delimiter will be the comma (',') and the end register will be a new line character.

```
delimiter : <literal-character> ;
```

```
end-register : <literal-character> ;
```

The structure of these declarations are similar, instead of the keyword. The delimiter declaration has an initial `delimiter` keyword followed by a colon (':') character and a literal character. The end register declaration has the same structure, but it is used as a `end-register` keyword.

Marker-text Declaration This declaration is used for visualization types that have markers. It enables users to express strings, fields or a combination of both to be used inside a textbox that appears when users select a marker.

```
marker-text : <literal-string> (or/and) <field-specification> ;
```

Its structure has a `marker-text` keyword followed by a colon (':') character and a literal string, a field specification, or both combined.

Page-title Declaration Page title declaration is used to insert a title in the HTML visualization file. Its structure has a `page-title` keyword followed by a colon (':') character and a literal string.

```
page-title : <literal-string> ;
```

Size Declaration This declaration is used to declare the required size of a visualization. It receives three possible values for size-specification. The size declaration structure initially has a `size` keyword followed by a colon (':') and a size specification. Table 4.3 describes the three possible sizes for size specification.

```
size : <size-specification> ;
```

Date Format Declaration GMaVis supports filtering and classifying based on date values. Since datasets usually have different formats of data, several formats were implemented in this DSL. A date format declaration is required to specify for the compiler how the data field must be parsed. The format is received inside a literal string. Its structure has a `date-format` keyword followed by a colon (':') keyword and the literal string. This structure is illustrated below.

```
date-format : <literal-string> ;
```

Inside this literal string, users must inform the format using dd, DD, MM, mm, YY, YYYY for all date information. For example, users can declare a date format such as `"DD-MM-YYYY"` for parsing data with the following format `"01-01-2016"`. In this case, the compiler knows that between the numbers there is a character. Other cases can have more then one character, having a string or even a description like `"month 01 -`

`day 01 - year 2016"`. This case would require a string like `"month MM - day DD - year YYYY"`. Thus, this compiler enables the parsing of the basic and most commonly used date formats, even if inside a description[1].

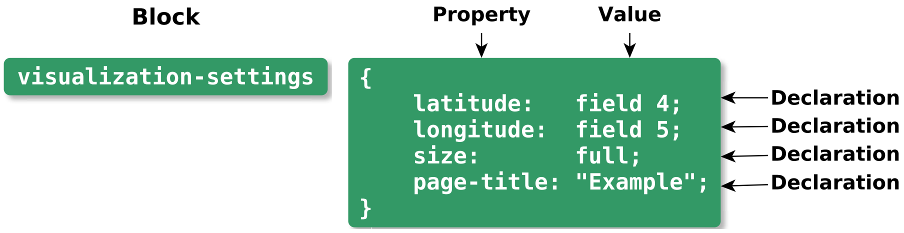
Visualization Declaration One of the most important declarations in GMaVis is the visualization declaration. This specifies the type of visualization that will be created.

`visualization :` *<visualization-type-specification>* `;`

It is declared in the global scope of the code, and its structure has a `visualization` keyword, followed by a colon (':') and a visualization type specification. Current visualization types that GMaVis support are described in Table 4.5.

4.2.4 Blocks

GMaVis organizes declarations in blocks. Each block contains declarations that are formed by property and value. GMaVis has four different blocks: data block, classification block, structure block and settings block. The structure of a block is a keyword followed by a left brace character, a declaration sequence and finally a right brace character. Figure 4.6 uses a settings block to illustrate the structure



```
visualization-settings
{
    latitude:   field 4;
    longitude:  field 5;
    size:       full;
    page-title: "Example";
}
```

Figure 4.6 – GMaVis settings block example.

In global scope, GMaVis receives two blocks, one for data and another for setting some details about the visualization. Also, two or more subblocks inside of a data block should be used to express classification and structure. Each block and accepted declarations are described bellow.

Block Data Data block has declarations related to data processing. It contains declarations with information about input data and how it will be loaded and processed by the data preprocessor. This block is divided in four elements: *file declarations*, a *structure block*, a *filter declaration*, and a *classification block*.

[1] Currently, GMaVis does not include or consider processing date formats with a text description for months or numbers, such as January, February, March, etc. It is planed for future work.

Block settings Block settings contain declarations related to visualization details. They receive some declarations as *latitude* and *longitude*, `page title`, `size`, `marker-text`, and `zoom level`.

Block structure The structure sub-block contains information about the file type, delimiter, and end register character. This information is crucial for the data preprocessor to recognize the limits of each register.

Block classification The classification sub-block allows the creation of markers with different colors for each class. This block receives the declaration of classes, which are transformed in data pre-processing in functions to select data and to be inserted into a particular class.

This grammar could be constructed without blocks in order to have less code. Yet It was used to better organize declarations. Since this grammar may increase in the future, it is interesting to use blocks to group declarations with similar characteristics.

4.3 Data Preprocessor

The data preprocessor is a module that is generated to process the input data. GMaVis compiler uses details in the source code to generate this data preprocessor in the C/C++ language. It is compiled and executed to receive the input files, parse them, process and save an output with only data that will be used in the visualization. The processing includes both filtering and classifying data registers.

We divided the Data Preprocessor into five operations: partitioning, structuring, filtering, classifying and output saving as presented in Figure 4.7. Each operation is described as follows.

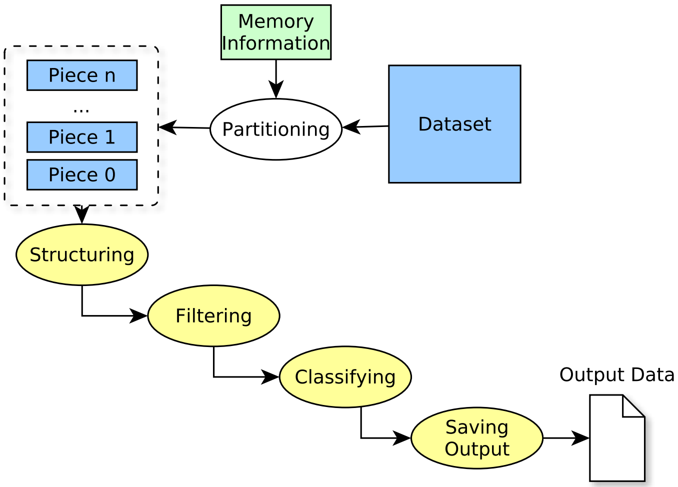


Figure 4.7 – Data preprocessor workflow.

Partitioning Partitioning is the first function used in this workflow. The partitioning phase enables the use of large datasets in low memory architectures, because it separates dataset into pieces to process it. Therefore, a computer with low memory can read and process a big file. For example, a system with 8GBytes of memory can process a file with 50Gbytes, if it is divided into many chunks.

Structuring This is the first step in data processing, which structures the data using the information about delimiters and end register characters given in the source code. This step is required to find information and process the file.

Filtering The second step in processing is to apply the filter defined in the source code. This applies the logical expression in all the registers, and if it is true, marks it to be inserted in the subsequent processing steps.

Classifying Classification is performed in the selected registers in the filtering process. This process verifies each register using the logical expressions defined in each class declaration. When true, the register is inserted in the class.

Saving Output The last step is to save the results in an output to use in visualization generation. This file uses the same delimiter and end register to separate fields and registers since modifications can break the file consistency when reading again. This may happen because delimiters and end registers are not used inside of the data. If it is modified, there is a possibility of having the same character inside a data value, and the parser will recognize it as a delimiter.

4.3.1 Data Processing Algorithm

Processing a huge quantity of data can be costly. Thus, it is important that the software that performs this operation uses smart search and selection of registers. Also, memory usage is a critical factor in this process. If an algorithm requires data replication to perform operations, it will require more cycles of processing pieces of data to finish the execution.

Considering this context and with the objective of implementing and optimizing the algorithm for processing data, we did a study on search algorithms [CL11] to choose an optimal one to perform the register selection. This study is based on the complexity time of each algorithm expressed using the Big O notation [Knu76, DSR11, Hil04]. Three search algorithms were analyzed: Linear Search, Binary Tree, and Interpolation Search.

For the comparison, we used the time and space complexity. As these algorithms would be used to process huge quantities of data, we considered the worst case to compare them. Also, space complexity was considered to compare these algorithms because the memory space used is an important issue to consider. Obviously, on the actual multi-core computer architectures, this may present a memory problem. Thus, for this evaluation we sought an algorithm that presents a minimum $\mathcal{O}(n)$ for worst time-space, maintaining the same quantity of data.

Table 4.6 presents the time complexity of each algorithm. Also, it exposes the limitations in respect to sorted and unsorted data. We highlight that the linear search algorithm has the largest complexity time because this verifies all the elements of the array during the search. On the other hand, this algorithm does not require a sorted array. Moreover, binary search and interpolation search algorithms present lower complexity times, they requires a sorted array to find one or a set of registers.

Table 4.6 – Time complexity of search algorithms.

Algorithm Name	**Time Complexity**	**Limitation**
Linear Seach	$\mathcal{O}(n)$	No Limitation
Binary Search	$\mathcal{O}(n \log n)$	Sorted Dataset
Interpolation Search	$\mathcal{O}(\log \log n)$	Sorted Dataset

Through this analysis, we can highlight that, for ordered arrays, an interpolation search algorithm is the best alternative because this has a lower time complexity. Otherwise, for disordered arrays, the linear search is more convenient because this does not need the use of a sort algorithm to organize registers. To certify if it would be possible to combine an interpolation or binary search algorithm with a sorting algorithm to obtain better results in unsorted arrays, in Table 4.7 we analyze these algorithms combining them with some sorting algorithms. Considering LS as Linear search, BS as a Binary Search and IS as an Interpolation Search. In this comparison, we used the sum of both search and sort algorithms. Therefore, considering that when combining two $\mathcal{O}$ values the greater is the result, we used the greater value to compare each combination with the Linear Search algorithm.

Table 4.7 – Search algorithm comparison considering sorting.

Algorithms	**Time Complexity (Worst)**	**Time Complexity (Total)**	**Space Complexity (Total)**
LS	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
BS+Merge Sort	$\mathcal{O}(\log n)$+$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
BS+Quicksort	$\mathcal{O}(\log n) + \mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
BS+Heapsort	$\mathcal{O}(\log n)$+$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$
BS+Selection Sort	$\mathcal{O}(\log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
BS+Bubble Sort	$\mathcal{O}(\log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
BS+Gnome Sort	$\mathcal{O}(\log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
BS+Insertion Sort	$\mathcal{O}(\log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
BS+Radix Sort	$\mathcal{O}(\log n)$+$\mathcal{O}(kn)$	$\mathcal{O}(kn)$	$\mathcal{O}(k + n)$
IS+Merge Sort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
IS+Quicksort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
IS+Heapsort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$
IS+Selection Sort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
IS+Bubble Sort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
IS+Gnome Sort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
IS+Insertion Sort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
IS+Radix Sort	$\mathcal{O}(\log \log n)$+$\mathcal{O}(kn)$	$\mathcal{O}(kn)$	$\mathcal{O}(k + n)$

As verified in Table 4.7, the time complexity of binary and interpolation search algorithms are increased with sorting operations. The implementation of a binary or interpolation search is more complex than a linear search algorithm since it needs the development of a sorting algorithm. As analyzed, most these algorithms will be performed using unsorted datasets if applied in our data preprocessor. Considering the information presented, we created the processing functions based on a linear search algorithm, performing filters and classifications in the whole dataset.

4.4 Visualization Generator

The last step in generating the visualization in GMaVis is performed in the visualization generator. This module receives the output data from the data preprocessor and the information from the source code. Then, the visualization generator produces data visualization code in HTML according to the specifications declared in the DSL source code. The output file of this module is an HTML file with visualization.

The visualization generator receives information from the code analyzer about details specified in the setting block. This is used in code generation to print HTML and Javascript code in the file, combined with data preprocessor output data. Finally, it creates an output file containing the required code to display the visualization map with the Google Maps API call, information about the title, sizes, markers, legends, and the data to be presented.

Our DSL does not generate the visual elements of the data visualization from scratch. To facilitate code generation, libraries such as Google Maps API are used in the HTML to generated code. This library was chosen to convert data to visual mappings, because it requires a short code for a simple code generation and provides a high-level interface for manipulating map details and inserting data. It also allows the implementation of other types of visualizations in the future without changing either the DSL architecture or output of the data preprocessor.

The HTML file with the visualization is generated by simply printing strings with common code directly in a file. However, there are pieces of code that are modified depending the visualization, data, specification details or other issues that require special attention. The type of code required to be generated in that specific part is verified through conditions statements. More details about the visualization generator, code generation, its internal functions, iteration with other modules, generation of HTML file and other internal details are explained in Chapter 5.

4.5 Use Cases

For better understanding of GMaVis grammar some use cases that demonstrates six applications (2 for each visualization type) using the GMaVis features will be presented in this section. This starts with Subsection 4.5.1 that exposes the three datasets employed in this applications. Subsection 4.5.2 presents their descriptions, codes and output images. These applications also were used in the GMaVis evaluations presented in Chapter 6.

4.5.1 Datasets

In these use cases, real-world data was used. Among the six developed applications, each dataset is used for two applications. Thus, we used three datasets for these experiments. These datasets are described below.

YFCC100M In the first dataset, we used YFCC100M [TSF+15], provided by Yahoo Labs. The YFCC100M dataset has about 54GB of data, divided into ten files. This is a public multimedia data set with 99.3 million images and 0.7 million videos, all from Flickr and under Creative Commons licensing.

Traffic Accidents We also used a dataset from DataPoa[2] with traffic accident data in the city of Porto Alegre,Brazil that contains information about the type of accidents, vehicles, date and time, level and location. It has 39 fields, including latitude and longitude, and about 20.937 registers of accidents between 01-01-2013 and 12-31-2013.

Airports A dataset obtained in OpenFlights[3] that contains all the airports in the world was used, with information about latitude and longitude, city, country and airport code. It has 8107 registers with 12 fields with information about the airports.

4.5.2 Sample Applications

In order to demonstrate GMaVis grammar and evaluate the DSL/Compiler, Six visualization applications were created using the datasets presented in subsection 4.5.1. Also, two applications were created for each type of visualization provided in GMaVis (MarkedMap, Heatmap, and Clusteredmap). Each application code and visualization is presented as follows.

Airports in world (clustered map)

In this visualization, a marker is displayed for each airport in the world. It used clustered map visualization type, which joins markers with less proximity. Thus, it is possible to identify countries with greater number of airports by clusters with red color. Figure 4.8 presents the generated visualization. Listing 4.1 describes the code to create this visualization using GMaVis.

[2] Available in: http://www.datapoa.com.br/dataset/acidentes-de-transito

[3] http://openflights.org/data.html

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

Listing 4.1 – Airports in world (clustered map) code.

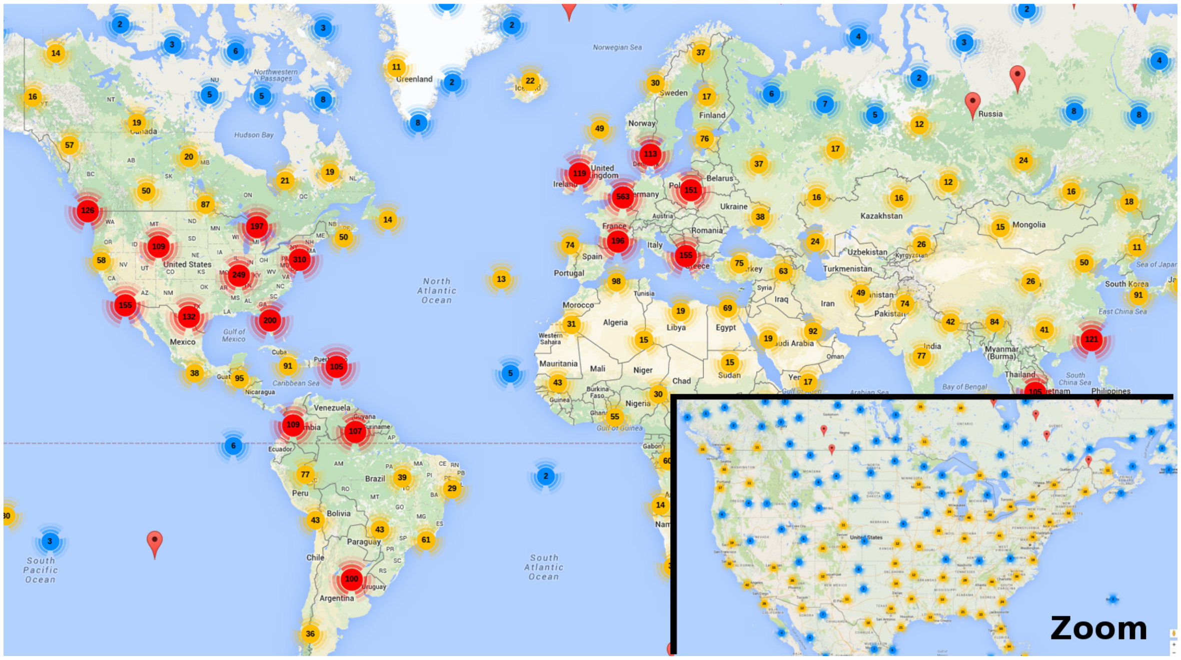


Figure 4.8 – Airports in world (clustered map).

Flickr Photos with "Computer" Word as Tag (clustered map)

This visualization also used clustered map to display markers that represent a photo taken and uploaded to Flickr. This visualization used a filter to select just pictures that have a tag that contains the word `computer`. This visualization displayed more occurrences of pictures with this tag in the United States and England. Figure 4.9 presents the generated visualization. Listing 4.2 describes the code for creating this visualization using GMaVis.

```
visualization: clusteredmap;
settings {
  latitude: field 12;
  longitude: field 11;
  marker-text: field 9 "and" field 8;
  page-title: "Flicker photos with computer tag in 2014";
  size: full;
}
data {
  file: "yfcc100m_dataset-0";
  structure {
    delimiter: tab;
    end-register: newline;
    date-format: "YYYY-MM-DD";
  }
  filter: field 8 contains "computer" or field 9 contains "computer";
}
```

Listing 4.2 – Flickr photos with "Computer" word as tag (Clustered Map) code.

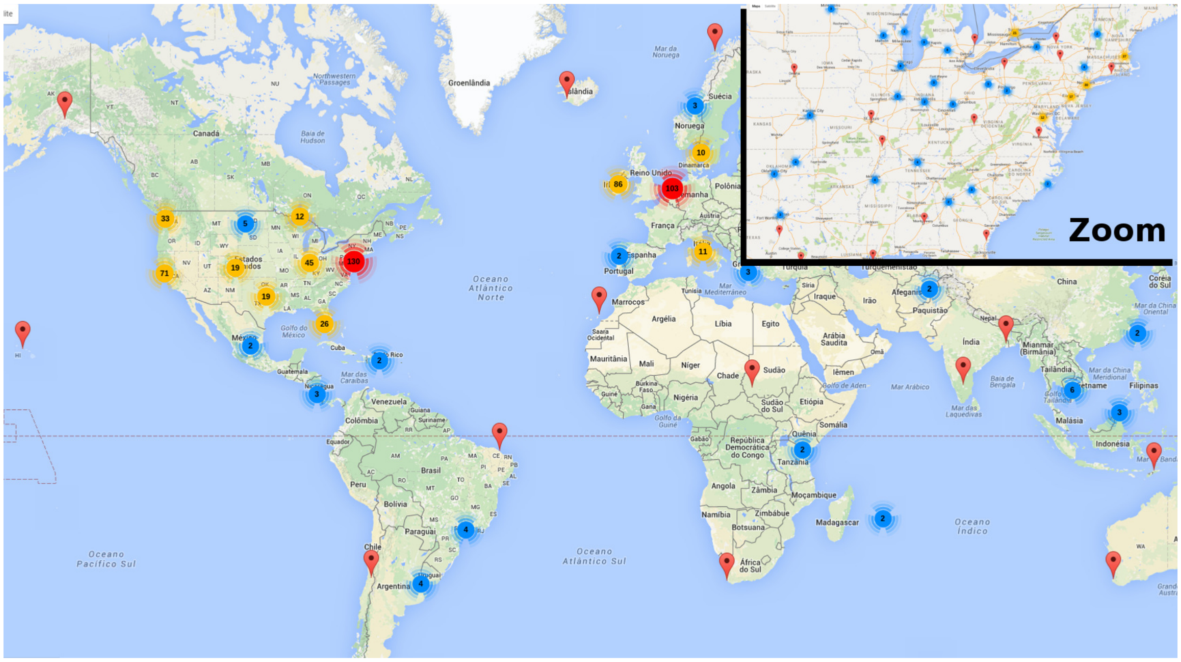


Figure 4.9 – Flickr photos with "Computer" word as tag (clustered map).

Airports in world (heatmap).

In this visualization, a marker is displayed for each airport in the world. Here a heatmap visualization was used. This visualization highlight places that have higher data frequency. In this case, it is possible to verify a greater number of airports in Europe. Figure 4.10 presents the generated visualization. Listing 4.3 describes the code for creating this visualization using GMaVis.

```
visualization: heatmap;
settings {
  latitude: field 7;
  longitude: field 8;
  marker-text: field 1 field 2 field 3 field 4;
  page-title: "Airports in World";
  size: full;
}
data {
  file: "vis_codes/airports.dat";
  structure {
    delimiter: ',';
    end-register: newline;
  }
}
```

Listing 4.3 – Airports in world (heatmap) code.

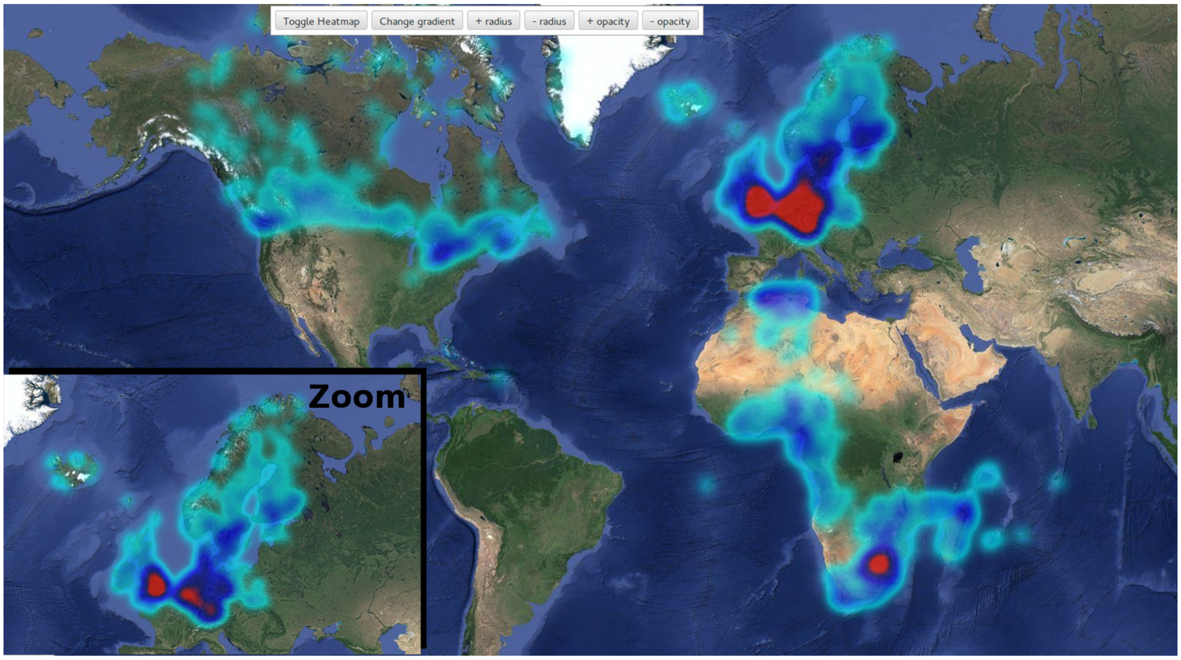


Figure 4.10 – Airports in world (heatmap).

Traffic Accidents in Porto Alegre, Brazil (heatmap).

This visualization used 2013 traffic accident data to display areas with greater frequency of accidents. In this visualization, areas with the color red have more accidents than those in yellow or green. Figure 4.11 presents the visualization generated. Listing 4.4 describes the code for creating this visualization using GMaVis.

```
visualization: heatmap;
settings {
  latitude: field 42;
  longitude: field 41;
  page-title: "Acidentes Poa 2014";
  size: full;
  zoom-level: 11;
}
data {
  h-file: "acidentes-2013.csv";
  structure {
    delimiter: ';';
    end-register: newline;
    date-format: "YYYY-MM-DD";
  }

  filter: field 13 is greater than 0 or field 14 is greater than 0;
}
```

Listing 4.4 – Traffic accidents in Porto Alegre, Brazil (heatmap) code.

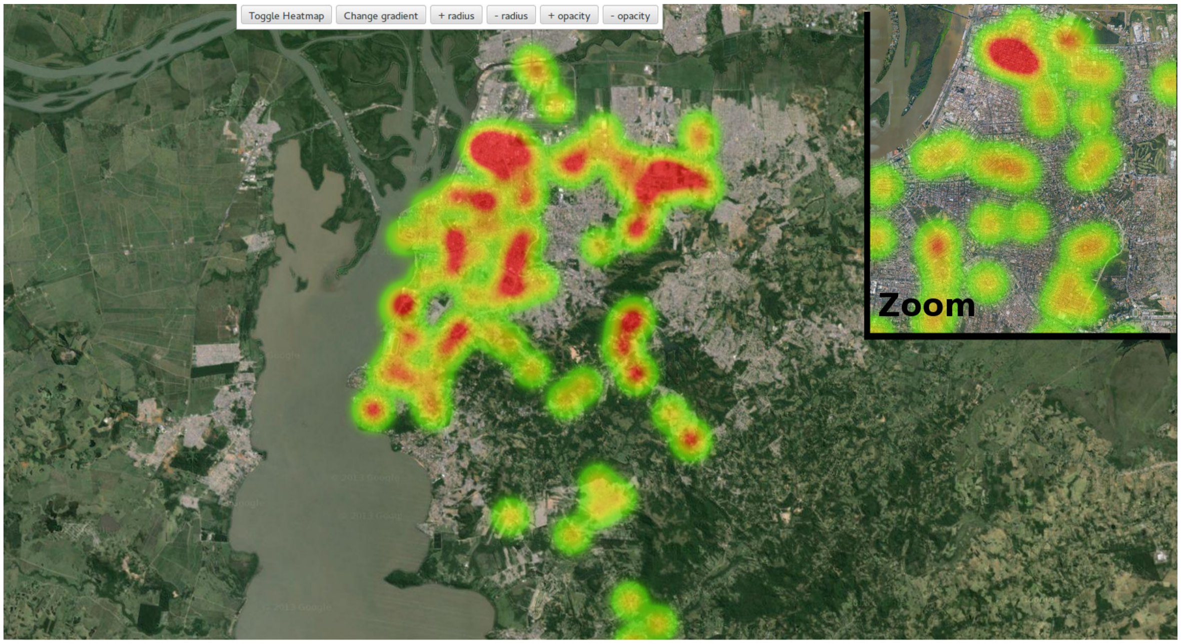


Figure 4.11 – Traffic accidents in Porto Alegre, Brazil (heatmap).

Traffic Accidents in Porto Alegre, Brazil with Classification (marked map)

A marked map was used in this visualization to display a marker for each traffic accident that happened during a month in 2013. This visualization classified data based on the number of deaths. Here red markers represent traffic accidents with deaths and orange markers accidents without deaths. Figure 4.12 presents the generated visualization. Listing 4.5 describes the code for creating this visualization using GMaVis.

```
visualization: markedmap;
settings {
  latitude: field 42;
  longitude: field 41;
  page-title: "Acidentes Poa 2014";
  size: full;
  zoom-level: 11;
}
data {
  h-file: "/home/cleversonledur/Downloads/acidentes-2013.csv";
  structure {
    delimiter: ';';
    end-register: newline;
    date-format: "YYYYMMDD";
  }

  filter: field 9 is greater than date "20131201" and field 9 is less than date "20131230";

  classification{
    class ("Fatal"): field 13 is greater than 0 or field 14 is greater than 0;
    class ("N-Fatal"): field 12 is greater than 0 or field 11 is greater than 0;
  }
}
```

Listing 4.5 – Traffic accidents in Porto Alegre, Brazil with classification (marked map) code.

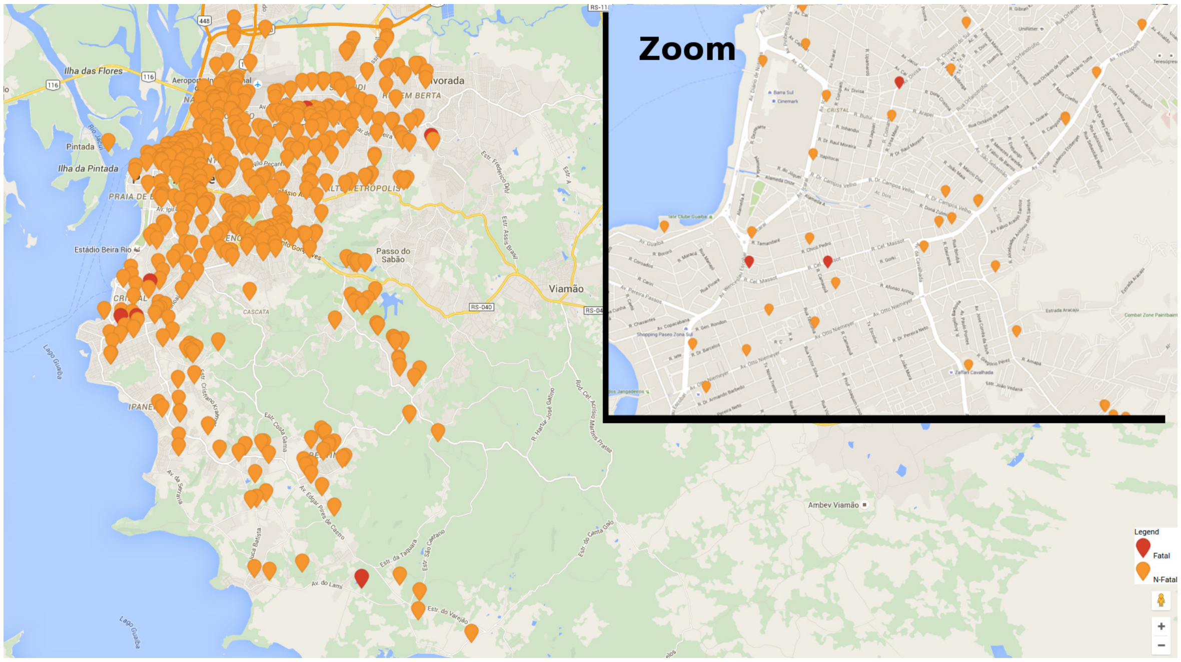


Figure 4.12 – Traffic accidents in Porto Alegre, Brazil (classified marked map).

Flickr Photos Took in 2014 Classified by Camera Brand Used

This visualization shows photo taken and uploaded to Flickr during 2014. The dataset used in this visualization has a field with information about the device used to take the photo. Then a classification was performed to display different color markers related to the camera brand. Figure 4.13 presents the visualization generated. Listing 4.6 describes the code for creating this visualization using GMaVis.

```
visualization: clusteredmap;
settings {
  latitude: field 12;
  longitude: field 11;
  marker-text: "Camera:" field 6 "<br><img src=" field 15 " width=200>";
  page-title: "Photos of 2014 Classified by Camera Brand";
  size: full;
}
data {
  file: "/home/cleversonledur/Documents/BIGDATA_YAHOO/yfcc100m_dataset-0";
  structure {
    delimiter: tab;
    end-register: newline;
    date-format: "YYYY-MM-DD";
  }
  filter:  field 4 is between date "2014-01-01" and date "2014-02-01";
  classification {
    class ("Canon"): field 6 contains "Canon";
    class ("Sony"): field 6 contains "Sony";
    class ("Nikon"): field 6 contains "Nikon";
    class ("Panasonic"): field 6 contains "Panasonic";
    class ("teste"): field 6 contains "Apple";
    class ("FUJI"): field 6 contains "FUJI";
  }
}
```

Listing 4.6 – Flickr photos taken in 2014 classified by used camera brand code.

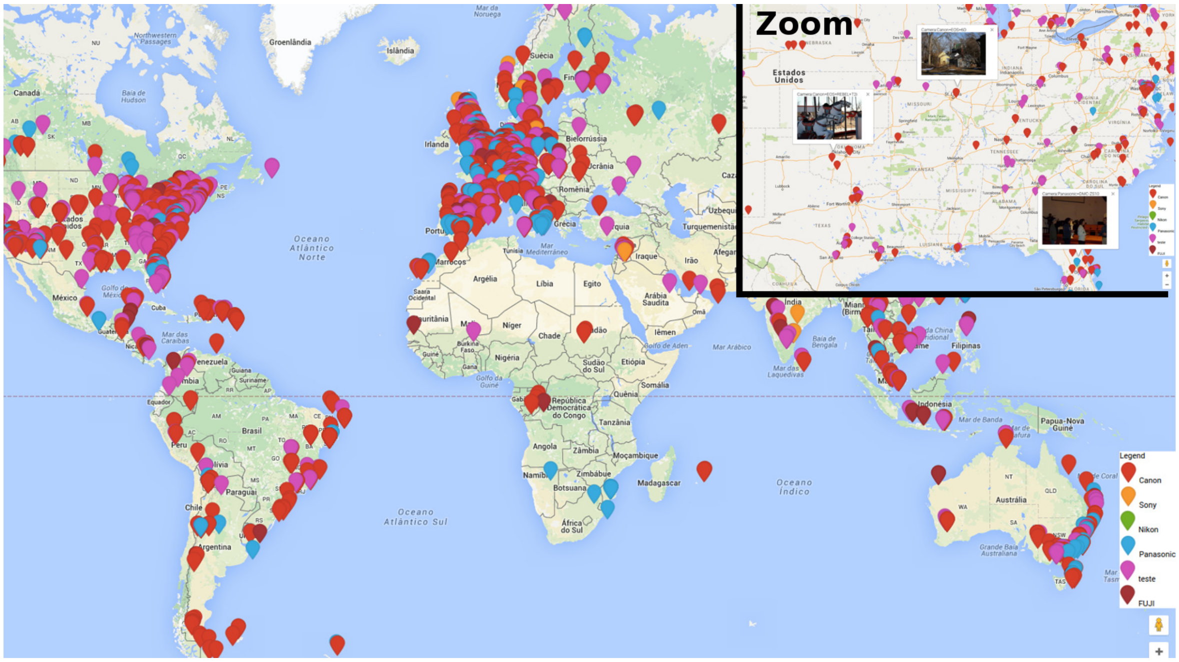


Figure 4.13 – Flickr photos taken in 2014 classified by used camera brand.

5. GMAVIS' COMPILER

A compiler was constructed to recognize the DSL source code and to generate geospatial data visualizations using GMaVis. This compiler was created using Flex and Bison to parse and generate code. These tools were briefly described in Chapter 2. It was developed in C/C++ language to provide better memory control when using dynamic memory allocation. Figure 5.1 illustrates this compiler structure.

This compiler receives the source code and performs lexical and syntax analysis, generating the tokens and combining them according to the specified grammar rules. As each rule is recognized in Bison, different actions are performed such as: saving information, calling functions to process values, concatenation of strings and flagging. This enables information storing to generate the data preprocessor and visualization generator. Also, a set of flags enable a semantic analysis to verify required fields and declared values. We did not create an optimization process in this compiler because it has a short and simple grammar.

The next subsections will explain the most important compilation phases. Section 5.1 will describe the lexical analysis performed in Flex. Section 5.2 will present details about grammar recognition in syntax analysis in Bison. Semantic analysis process will be introduced in Section 5.4. Finally, code generation for visualization generation and also for parallel and sequential data preprocessor will be explained in Section 5.5.

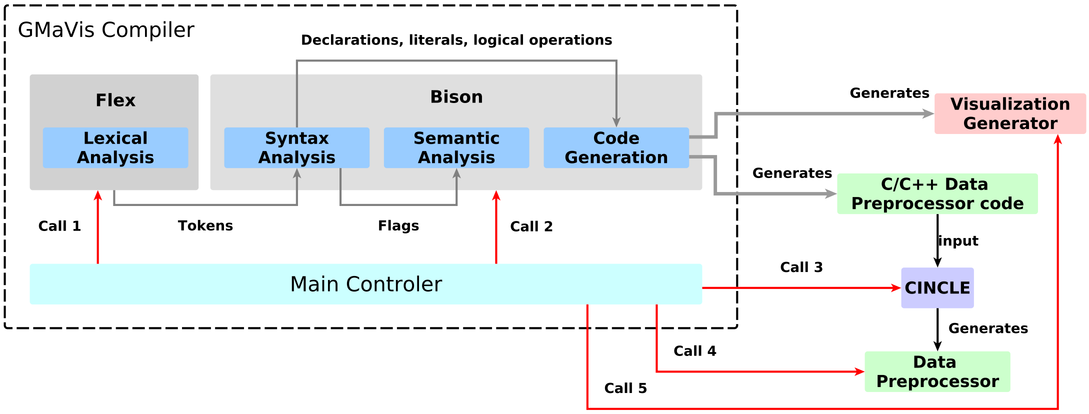


Figure 5.1 – GMaVis compiler structure.

5.1 Token Recognition

The first phase in this DSL compiler process is the lexical analysis, where tokens, keywords, literals, operators and punctuation are recognized. For token recognition, we used Flex. As explained in Chapter 2, flex uses regular expressions to match tokens. Thus, a set of regular expressions was created to recognize each token required in this DSL. Initially, a file with the required format of Flex was created. Then, a set of regular expressions were created for each token recognized in the source code. This enabled the creation of an initial scanner that reads the input source code and generates tokens. These tokens are used to verify if the source code follows the grammar and for code generation.

In GMaVis, some keywords were defined to express declarations or specific information. For example, the keyword ̈filter ̈is used in a filter declaration, which works combined with a logical expression. When the lexer reads the word `filter`, it generates a token for this since there is a regular expression in our scanner that recognizes this. The same operation is performed for other keywords, punctuation and literals. The following regular expressions were used to recognize the entire DSL.

The regular expressions were inserted in a flex file with a .l extension. This file is split into three parts: definitions, rules, and user code. Initially, we inserted some variables used to control the line number and token location in the file. This enabled the creation of message errors with the exact token location in the input source code. Also, this first part allows the insertion of the bison file declaration and C/C++ libraries to be used in the compiler. In the second section of GMaVis flex file, we inserted an important option `%option caseinsensitive` to disable the case-sensitive recognition in this compiler. Since the main objective was to facilitate the use of this DSL, the GMaVis compiler does not differentiate upper-case from lower-case letters. Thus, if the user creates a code with the work ̈Filter ̈or ̈filter ̈, the scanner will recognize it as the same token. Also, in this section, it was inserted the regular expressions to recognize the tokens. When the scanner recognizes a particular regular expression in the input source code, it can generate an action. In this specific case, the scanner creates a token and returns it to Bison to use in grammar recognition. GMaVis do not need to recognize spaces and escape characters. Therefore, when the scanner recognizes one of these no action is performed. The third section in a flex file was not used in this compiler since all the control is performed in the bison file. Next section describes this file.

5.2 Grammar Recognition

An important part of this compiler is the Bison file with the .y extension. This contains all the grammar expressed in a YACC format as well as, some functions such as main, the tokens declarations, token type declarations, and the actions for each grammar rule. The general concept is that when Flex recognizes the tokens, bison stores them in a stack and performs a shift or a reduce operation using the grammar rules, as explained in Chapter 2. Then a reduce in some grammar rules is performed, which results in an action. For example, this action can be to generate code, store important information, recover from an error or report some information to users.

In GMaVis compiler, different actions are performed for each grammar rule. Also, GMaVis grammar is very simple, so an AST (Abstract Syntax Tree) was not required. Because it is a description language, we instead used flags to verify semantic errors, as explained in Section 5.4. Therefore, the actions performed in each grammar rule were reduced. Also, creating an AST would require more execution time and memory, therefore not doing so enabled a faster and low memory consumption compiler. Thus, only four different actions are performed when a rule is matched: to set flags, store source code information, generate logical expression code, and report important information during parsing. Each action is explained in the next section.

5.2.1 Flag Setting

Flags were used to enable a code verification about which fields were declared, type of information used, literals and data formats specified in input source code. These flags are logical variables, which receive a true or false value. A flag is set initially as false and when it receives a true value when set. In this compiler the following variables were used as flags.

date_format_required: It is true when a field that receives a value uses a literal date. Through this flag, it is possible to verify required functions and special dealing with date formats in code generation. Also, we can check in the semantic analysis if a format for data was given. If not, a semantic error is reported.

delimiter_special *and* end_register_special: These flags, when true, inform that the user specified a special character as a delimiter and did not use the `tab` or `newline` reserved word of GMaVis. In this case, the compiler may perform an operation of removing quotes from the string returned by the scanner analysis to print this delimiter in the generation of code.

do_marker_text *and* first_marker_field: These flag are set when the field `marker-text` is recognized in the input source code. This flag is used in code generation for both data preprocessor and visualization generator since a markedmap visualization without a text for each marker is differently generated of when using this feature.

do_classification: This flag is set when the compiler recognizes the declaration of a classification block. It is required since a visualization that uses classification is generated with different icons and information and visualizations without classification are simpler. Also, in data pre-processing, different actions are performed when a classification is required.

date_format_set: This flag informs a declaration of a `data-format` in input source code. It is used in the semantic analysis to verify if the data format is declared when a declaration receives a literal date as input. For example, if a user declared a filter that uses a field with a date, but no format for this date is declared, the compiler must generate a semantic error.

delimiter_set *and* end_register_set: These flags are set to inform if users declared the `delimiter` or `end-register` for the input file. These are used in the semantic analysis to generate a semantic error for the user.

location_set: This flag was created for the field `location` declaration. It may be used in the future for new visualizations. Currently, this field is only reserved for future use.

value_set: This flag was created for the field `value` declaration. It may be used in the future for new visualizations. Currently, this field is only reserved for future use.

zoom_set: This flag informs that a `zoom` declaration was specified. It is used in code generation during the visualization generation phase. If this tag is false, a default zoom value is applied.

page_title_set: This flag informs if the declaration `page-title` was created in the input source code. It is used in the visualization generation to choose between inserting a string value from an information variable or generating a default title.

latitude_set *and* longitude_set: For verifying if the required field `latitude` was declared, we use this flag in semantic analysis. If it is not set, the compiler reports a semantic error message.

size_set: The field `size` informs the size to generate the visualization. If this flag is false, the compiler generates the visualization using a full-size visualization as default. It is only used in code generation.

Furthermore, the presented flags are used to improve and facilitate code recognition, semantic analysis, and code generation. Most flags are used in many phases, and they are set during the parsing. These flags have enabled a simpler compiler construction without the creation of an AST and functions to browse this. Of course, it is possible since GMaVis have a short grammar with a description language and is not implemented inside another general purpose language.

5.2.2 Storing Information

To enable the code generation after the parsing phase, a great deal of information needs to be stored. Since GMaVis have a short grammar, it is possible to store the required information in variables without creating an AST. This information is stored as the compiler executes the grammar recognition, through the actions in each Bison grammar rule. The following information is stored in variables for using in the code generation.

page_title: This is a string that receives the content of the declaration `page-title`. It is used in the code generation to generate the title code in HTML.

required_field1: This is a string with the generated code to be inserted in data preprocessor, which specifies the number of the field for required_field1. In the implemented visualization types, the required code is the `latitude` declaration content.

required_field2: (string) This is a string with the generated code to be inserted in data preprocessor, which specifies the number of the field for required_field2. In the implemented visualization types, the required code is the longitude declaration content.

output_fields: This string is created by a set of concatenations from logical fields specifications and strings that are declared in the `marker-text` declaration content. It is used in the code generation to create the output string applied inside a C++ if statement.

visualization_size: This string stores the content used in declaration `size`, which can be *full*, *medium*, or *small*. It is used in the visualization generation to create the size values (width and height) for the map `div` in HTML.

date_format_type: This string receives a string concatenated during the grammar recognition. This string contains the format given in the `data-format` declaration and is used in the operations of filters with data. It is passed to the data preprocessor during code generation.

delimiter: This is a string that stores the information received in the `delimiter` declaration inside the structure block. This variable is used in the generation of data preprocessor and visualization generation to read the dataset file and parse it.

end_register: This is a string that stores the information received in the `end-register` declaration inside the structure block. This variable is used in the generation of data preprocessor and visualization generation to read the dataset file and parse it.

filter: The filter variable stores a string that is concatenated during the grammar recognition in bison. This filter is generated and inserted in the data preprocessor. It is used inside the if statement to select data in the filtering process.

zoom_level: It is a string that stores the content of a declaration `zoom-level`. It is used in visualization generation to adjust the initial zoom.

vis_type: This variable controls the visualization that will be generated. The compiler converts the visualization name in the input source code to an integer. This variable is used both in the data preprocessor and visualization generator.

format_type: This variable controls the kind of file that will be received as input.

files_to_process: This is a vector of strings that have a list with files declared in the `file` declaration.

classification_classes: This is a vector of strings with logical expressions created during grammar recognition. This information is used in the code generation of data preprocessor to specify the operation that may be performed to classify each data element.

classification_alias: This vector of strings is used in combination with the classification_classes to store the alias for the class correspondent. This field is used in the visualization generation to create the legend with the classes.

5.2.3 Logical Expressions Generation

Inside `data` block, GMaVis expects logical expressions in filter declarations and class specifications. These logical expressions have logical operators that can receive different values types. Each logical operator combined with an acceptable value type has a function in the data preprocessor that performs the logical operation in the input data and returns a boolean value. During compilation, expressions are converted to these functions. To generate the data preprocessor with the same expressions and logic that is received in the source code, a string with function calls related to the logical operator is concatenated. Table 5.1 presents these functions related to the logical operator and input data type. This string is generated during grammar recognition using expressions' actions to concatenate the data preprocessor code.

Figure 5.2 illustrates this operation through an example. In this example, a logical expression that uses a logical operator `is equal` is applied in `field 2` to verify the values that are equal to the value `1000`. When the compiler recognizes the whole logical operation, receiving the field specification, logical operator, and the value, an action of concatenating the function call to `int_is_equal` with the respective values is applied. This function call is concatenated to the string variable `filter` or `classification_classes[n]` (described in Section 5.2.2, in the next grammar reduce. Also, as the input value modifies the function call, before this action the value type is verified to generate the correct function call.

Table 5.1 – Data preprocessor functions for each logical operator.

GmaVis Logical Operator	Input Data Type	Generated Function in Data Preprocessor
Is equal to	string	string_is_equal(*counter,data,i,field,value*)
	date	date_is_equal(*counter,data,i,field,date2*)
	integer	int_is_equal(*counter,data,i,field,value*)
	float	float_is_equal(*counter,data,i,field,value*)
Is different than	string	string_is_different(*counter,data,i,field,value*)
	date	date_is_different(*counter,data,i,field,date2*)
	integer	int_is_different(*counter,data,i,field,value*)
	float	float_is_different(*counter,data,i,field,value*)
Is greater than	date	date_is_greater(*counter,data,i,field,date2*)
	integer	int_is_greater(*counter,data,i,field,value*)
	float	float_is_greater(*counter,data,i,field,value*)
Is less than	date	date_is_less(*counter,data,i,field,date2*)
	integer	int_is_less(*counter,data,i,field,value*)
	float	float_is_less(*counter,data,i,field,value*)
Is between and	date	date_is_between(*counter,data,i,field,date2,date3*)
	integer	int_is_between(*counter,data,i,field,value1,value2*)
	float	float_is_between(*counter,data,i,field,value1,value2*)
contains	string	string_contains(*counter,data,i,field,value*)

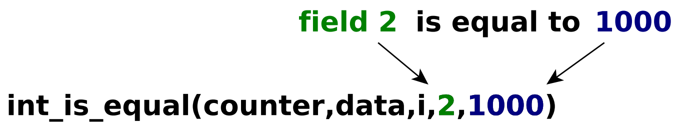


Figure 5.2 – Example of logical expression transformation.

It is also possible to combine logical operations using `or` and `and` operators. These operators are respectively converted to `||` and `&&`. This conversion is required since this logical operation is inserted inside an if C++ statement. C++ accepts the operators `or` and `and` as alternative operators. However it is not common to use.

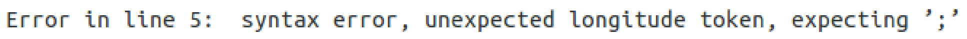
5.3 Syntax Analysis

Since GMaVis grammar is very short, it does not have ambiguities and complex syntax. Thus, all the syntax analysis is performed by Flex and Bison during the grammar recognition. Both Flex and Bison report errors when an unexpected token is received. Thus, an analyzer did not have to be implemented to verify syntax problems. Moreover, message errors that Flex and Bison presents can be personalized with an internal function called `yyerror` to print the token with the error message. GMaVis compiler also prints the token name that is expected in the statement or declaration. For example, Listing 5.1 presents a piece of code that reproduces an error. Here, the semicolon was removed from the end of a `latitude` declaration in line 4. Thus, it may reproduce an error in line 5 when it finds the keyword `longitude`.

```
1 visualization: markedmap;
2
3 settings {
4   latitude: field 12
5   longitude: field 11;
6   ...
7 }
8 ...
```

Listing 5.1 – Syntax error example.

When compiling a code with this syntax error, it will reproduce the error message illustrated in Figure 5.3. This error message informs the error line in the source code, the information of a syntax error, the unexpected token received and which token should be received to match with the grammar rule in Bison.



```
Error in line 5:  syntax error, unexpected longitude token, expecting ';'
```

Figure 5.3 – Error example for a syntax error.

Bison default configurations do not report an error informing the unexpected and expected token in the syntax error message. A configuration in the Bison file (.y) had to be set to accept report errors with more details. Thus, it was declared `%define parse.error verbose`. It forces Bison to present errors with detailed information.

5.4 Semantic Analysis

Semantic analysis is performed through the flags, as explained in Section 5.2. Since this is a description language with a simple grammar, the construction of an AST both to generate code or implement the code analysis was not required. It is possible to verify if the inserted code is correct and have the required information to generate the visualization through flags that receive boolean values during grammar recognition.

The semantic verification is simple. It verifies if required declarations for specific cases are correct. For example, the following algorithm describes one of the verifications performed. In this example, it is verified if the fields for `longitude` and `latitude` were declared when the visualization is a `markedmap`. If this analysis finds a false value, the algorithm returns an error to the user.

The previous example shows only two verifications performed in the semantic analysis. However, there are more 15 verifications that when false returns the following errors messages.

```
if visualization is a markedmap then
    if latitude was not declared then
        print Error: A required field is not declared.
        print For a marked map visualization you must declare the field for Latitude.
    end
    if longitude was not declared then
        print Error: A required field is not declared.
        print For a marked map visualization you must declare the field for Longitude.
    end
end
```

Algorithm 5.1 – Semantic analysis of latitude declaration for markedmap.

1. Error: A required field is not declared. For a marked map visualization, you must declare the field for Latitude.
2. Error: A required field is not declared. For a marked map visualization, you must declare the field for Longitude.
3. Error: Location declaration ignored. It is not required in this visualization type.
4. Error: Value declaration ignored. It is not required in this visualization type.
5. Error: A required field is not declared. For a heatmap visualization, you must declare the field for Latitude.
6. Error: A required field is not declared. For a heatmap visualization, you must declare the field for Longitude.
7. Error: Location declaration ignored. It is not required in this visualization type.
8. Error: Value declaration ignored. It is not required in this visualization type.
9. Error: A required field is not declared. For a clustered map visualization you must declare the field for Latitude.
10. Error: A required field is not declared. For a clustered map visualization you must declare the field for Longitude.
11. Error: Location declaration ignored. It is not required in this visualization type.
12. Error: Value declaration ignored. It is not required in this visualization type.
13. Error: You must declare the data format inside data block.
14. Error: End register character of the file has not been declared.
15. Error: Delimiter character of the file has not been declared.

5.5 Code Generation

Code generation is performed in two steps in this compiler. First the data preprocessor is generated to process input data. This first generation uses the information received in `block` data to open input files and apply filter and classification. Also, information about file structure is used to parse the file. The second code generation creates the visualization. It receives the output file with the formatted data from data preprocessor and creates the visualization file in HTML inserting both code and data. Also, a parallel version of data preprocessor was implemented using SPar. We inserted annotations in the sequential code to compile this parallel version using the SPar compiler, called Cincle.

Information stored in variables and flags, as described in Section 5.2.2 enables this code generation. Since the information required to generate are stored in variables, it is easy to insert them in the correct place in the generation. Thus, this code generation becomes very simple. The major difficulty is to handle uncommon pieces of code when generating different visualization types. This is difficult because input data may or may not be classified, have markers, have a legend for classes and filtering that can modify the final visualization code.

The next subsections will explain how code generation is performed in this compiler. Subsection 5.5.1 will describe the code generation for the data preprocessor. Subsection 5.5.2 details the generation realized for the parallel version of data preprocessor using SPar. Finally, Section 5.5.3 explains the code generation for visualization generation.

5.5.1 Data Preprocessor Generator

The data preprocessor is generated to perform operations such as filtering, classifying, data selection and formatting. These operations are realized through functions that parse, process and save the output data. The generation of a data preprocessor consists of printing in a file with the name `data_preprocessor.cpp` the code and call the G++ compiler. After the parallel version is implemented, it is not required to call G++, even if the code generated is sequential since the SPar compiler also enables the generation of sequential code. This is explained in detail in Subsection 5.5.2. Thus, we basically follow the workflow illustrated in Figure 5.4 where the data preprocessor generator receives the variables with information about files to be processed, data structure, date formats, filter, and classes from the code analyzer. This information is generated with the code for data preprocessor.

The use of flags in this phase is very important. This generation also requires information about the flags about declarations to know how the data preprocessor will be implemented. For example, if the data visualization does not require classification, it is not

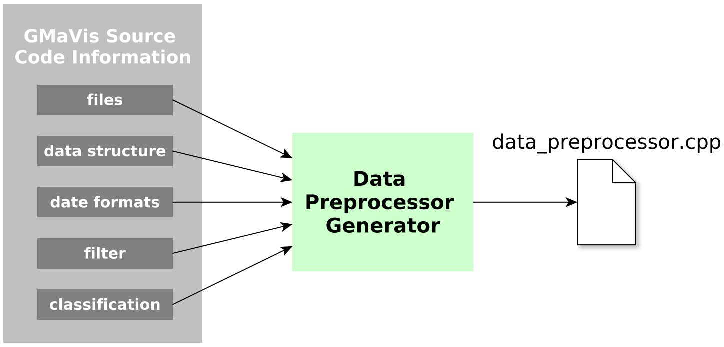


Figure 5.4 – Data preprocessor generation.

necessary to include functions and variables that are used to perform a classification in data preprocessor. The same applies for filtering and text in markers. Furthermore, flags are used in the creation of functions to generate output since it needs to know how much information and what fields must be printed.

This code generation requires special attention to the following steps since depending on the input GMaVis code the data preprocessor is modified.

Input Files Input files are inserted in the main data pre-processing code, which sends a function called `process()`. The list of files received during the grammar recognition is used in this phase of code generation. The path to the file is written in the main code enabling the data preprocessor to read and process each one.

Data Structure Data structure information is used to parse the input file. The delimiter and the end register character are informed in the DSL code. This information is now stored in a string. This string is copied to the data_preprocessor variables inside `process()` function.

Date Format Dealing with date requires some additional processing that is not in other value types. In GMaVis the users specify a date using a string and must insert the format of this date also, since data preprocessor cannot recognize this automatically. Users specify the format of this data through the `date format` declaration. In code generation this information is used to generate the logical operators that deal with the date: date_is_equal(...), date_is_different(...), date_is_greater(...), date_is_less(...) and date_is_between(...).Also, to inform data value, the user may specify the keyword date before the string.

Filters Filters generation is performed using the information stored in the variable filters, which receives a logical expression as explained in Section 5.2.3. These filters are generated inside an if in the `filter` function.

Classification Classification generation inside data preprocessor is similar to the filters. The main difference here is that we can have many classes. Each class is written inside the function `classification` in the data preprocessor.

Output Saving Function This phase in data pre-processing requires special attention for visualizations that have markers with text and classification. The addition of these two pieces of information in the output file results in more columns, and consequently, a different code generated. Markers' text information are stored in a variable as explained in Section 5.2.2. It requires additional processing for concatenating strings and field contents specified in the declaration `marker-text`. Classification fields are simple to handle in the output since they do not require any concatenation, just a print of a new column at the end of the file.

5.5.2 Data Preprocessor in Parallel

Preliminary results [LGMF15] through execution time using three data loads concluded that the data preprocessor module has a high computational cost when working with big files. To speed up the data pre-processing and enable efficient use of multi-core architectures that have multiple cores, we implemented a parallel version that takes advantage of parallel architectures with shared memory. Furthermore, the GMaVis compiler was required to generate this data preprocessor in parallel, abstracting complexities of visualization creation and parallelism from users. Users of GMaVis do not have contact with parallel code since it is totally abstracted.

To implement a parallel data preprocessor, we need to understand how a single process runs when in a sequential version. Basically Figure 5.5 illustrates a dataset that is partitioned in *N* pieces. These pieces are processed sequentially through the three stages of data processing. The first represents the loading of the file to memory and verification for data consistency, represented by `STR` in the figure. The second stage is the data parsing, application of filters, and classification. In this figure, this stage is illustrated using `PRO`. Finally, a stage receives a string with the output and writes this down in a file. This stage is represented by `OUT`. Thus, each piece of data flows through each stage until the output data is achieved. In the sequential version, a single process performs each operation for each time unit, resulting in a higher execution time as the input increases.

As verified, the data preprocessor has an explicit pipeline. It is complex to parallelize with some interfaces based on loop/iteration decomposition or parallel tasks, such as OpenMP. However, it is possible to parallelize this problem using interfaces that support problem decomposition using a stream parallelism model, as described in Chapter 2. Figure 5.6 illustrates how it is possible to parallelize a data preprocessor using a pipeline model that overlaps time through performing the three stages in different processors. Thus, we

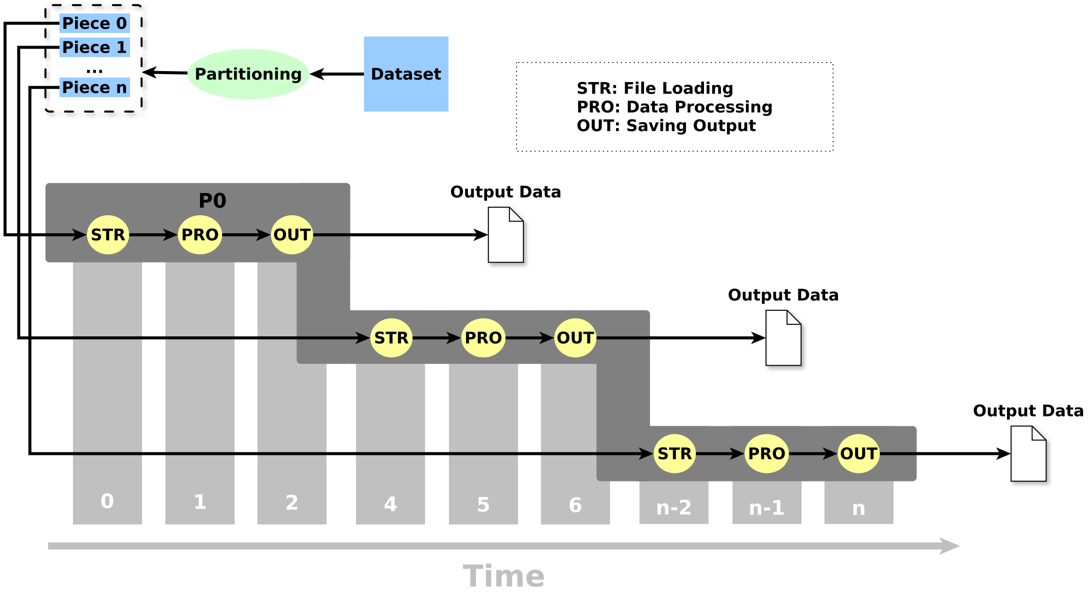


Figure 5.5 – Example of sequential processing in data preprocessor.

considered using three parallel programming interfaces to parallelize the data preprocessor and, generate it using this interface. TBB, Fastflow, and SPar enabled the parallelization of this software.

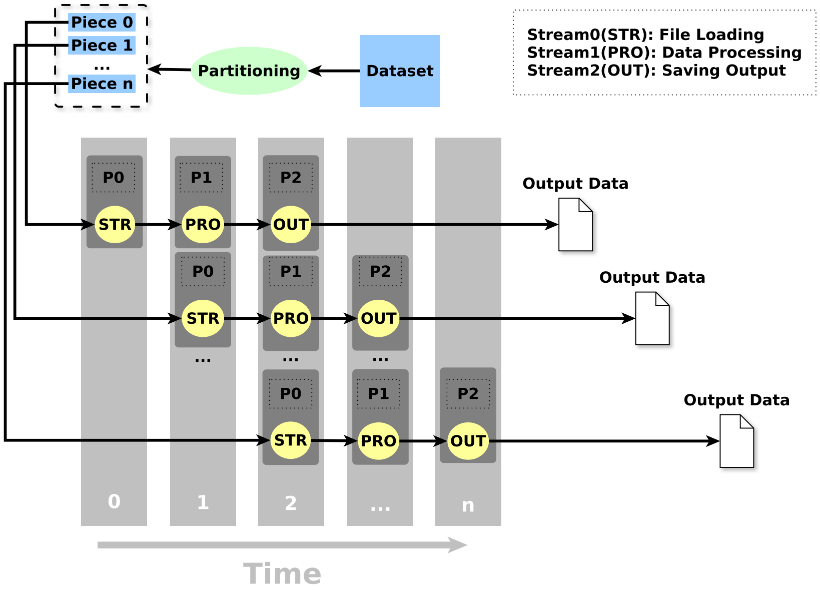
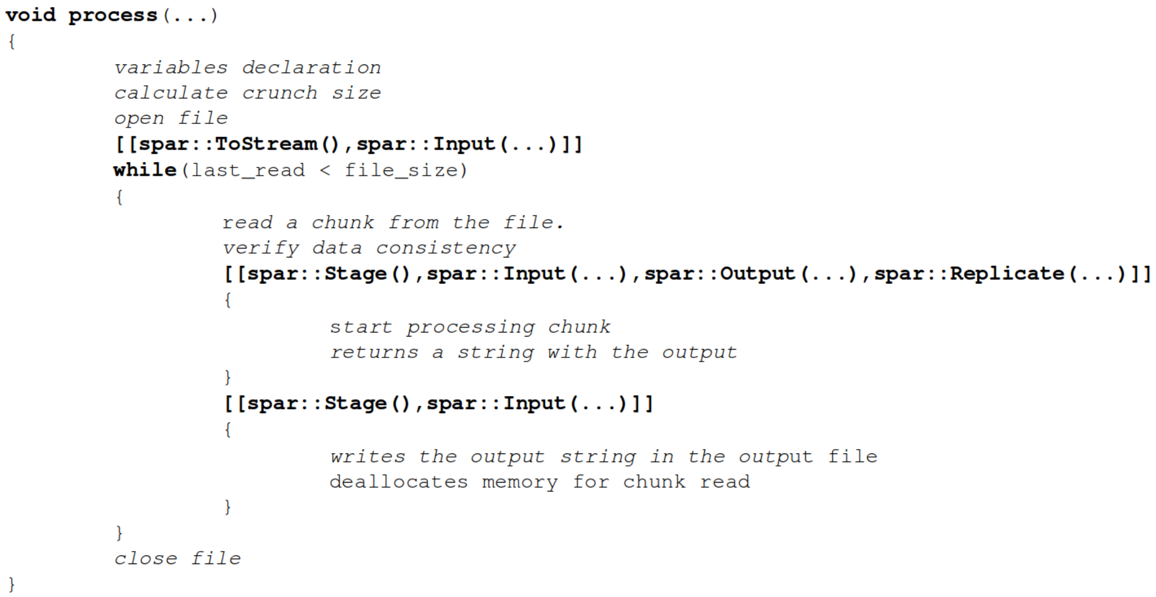


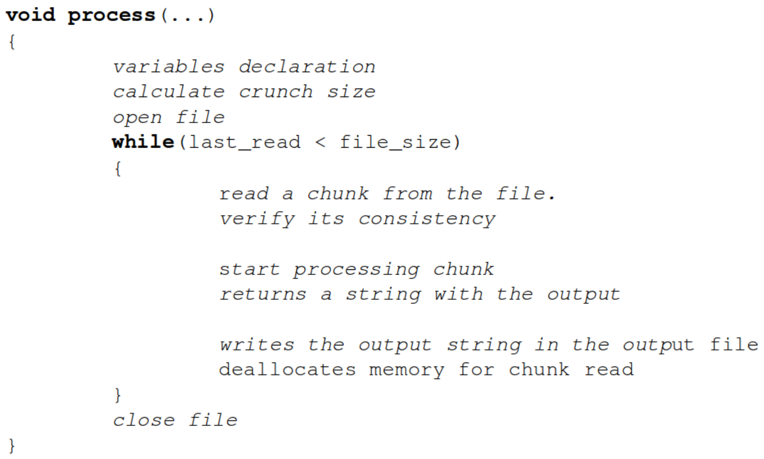
Figure 5.6 – Parallel preprocessor using pipeline strategy.

Looking to speed up the pre-processing generation, we used SPar to generate a parallel version of data preprocessor. As previously mentioned, we considered implementing this using TBB and Fastflow. SPar was chosen because it facilitates code generation in the GMaVis compiler. SPar uses annotations to generate parallel code. Thus, the code that we implemented to parallelize the data preprocessor using SPar is almost the same as is used in the sequential version. To parallelize this software, just a few lines of code had to be implemented in the sequential version. Figure 5.7 illustrates the function `process`, which has the three stages. Here we inserted the three lines of code that SPar requires to parallelize the code for the strategy illustrated in Figure 5.6. Figure 5.8 illustrates the same sequential.



```
void process(...)
{
        variables declaration
        calculate crunch size
        open file
        [[spar::ToStream(),spar::Input(...)]]
        while(last_read < file_size)
        {
                read a chunk from the file.
                verify data consistency
                [[spar::Stage(),spar::Input(...),spar::Output(...),spar::Replicate(...)]]
                {
                        start processing chunk
                        returns a string with the output
                }
                [[spar::Stage(),spar::Input(...)]]
                {
                        writes the output string in the output file
                        deallocates memory for chunk read
                }
        }
        close file
}
```

Figure 5.7 – Illustration of process function code with SPar annotations.



```
void process(...)
{
        variables declaration
        calculate crunch size
        open file
        while(last_read < file_size)
        {
                read a chunk from the file.
                verify its consistency

                start processing chunk
                returns a string with the output

                writes the output string in the output file
                deallocates memory for chunk read
        }
        close file
}
```

Figure 5.8 – Illustration of sequential process function code.

It is possible to verify that SPar requires few modifications in the sequential code to parallelize. If we use TBB or FastFlow to parallelize this function, it would require implementing additional code to create this same parallel version. Therefore, using SPar, we had a faster and easier parallel code generator implementation. Moreover, SPar's compiler also enables us to compile both sequential and parallel code with the same annotated code, just changing a command line parameter. Hence, the same code illustrated in Figure 5.7 enables the creation of both executables since it keeps the same semantic implementation. If we need a sequential executable, the parameter `-cincle` must be removed from the compiler command line. This makes the G++ compiler ignore the annotations of SPar and generates a sequential executable.

5.5.3 Data Visualization Generator

The data visualization generator uses some information to create the visualization. First it receives information from the parser, using flags and stored strings to create an HTML file with both code and data. It also receives the file containing the output data previously processed in data preprocessor as input. It opens the file with the output data, and inserts each element, code or information in a specific location. Information received from the compiler is described as follows.

Auto Focus One piece of information required in Google Maps API is the latitude and longitude to focus the first visualization load. Usually, the user informs this, however, when dealing with many points it is hard to verify this manually. Thus, the data visualization generator calculates this based on the average latitude and longitude of generated markers, using the following equation. Consider n as the total number of registers to be shown in the visualization. It calculates the average latitude and longitude registers to create the centering position values.

$$\text{latitude-focus} = \frac{\sum_{i=1}^{n} latitude_i}{n} \tag{5.1}$$

$$\text{longitude-focus} = \frac{\sum_{i=1}^{n} longitude_i}{n} \tag{5.2}$$

Visualization Type Depending on the visualization, a different code is generated. This information is received and is the first verification before starting the generation.

Data Structure It is used to read the file. The data preprocessor generates the output with the same delimiters of the original data. Then, this phase of generation requires this information to read the output.

Classification It is required to know if there is a classification applied in the output data. This will require implementing different icons for markers and, if declared in the source code, a legend.

Title This is used to create a page with the same title informed in the source code.

Size It is used to create the visualization with the size declared. It can be small, medium or full.

Marker Text Information about marker text is required to create values to show when markers are clicked.

Figure 5.9 illustrates the generation process in the data visualization generator. An important aspect is that there are pieces of code that are common in any visualization. These pieces of code are always generated directly, without any verification in the declarations or data. However, there are specific pieces of code that are related to the received source code, and these specific pieces of codes have to specially verify the visualization type, data, classifications and other details. In the data visualizations implemented, there is a specific place where data is inserted. Also, different visualizations implement different formats for inserting this data like arrays or variables for each marker. Because there is not a common way to insert this, the data visualization generator formats the data for each visualization.

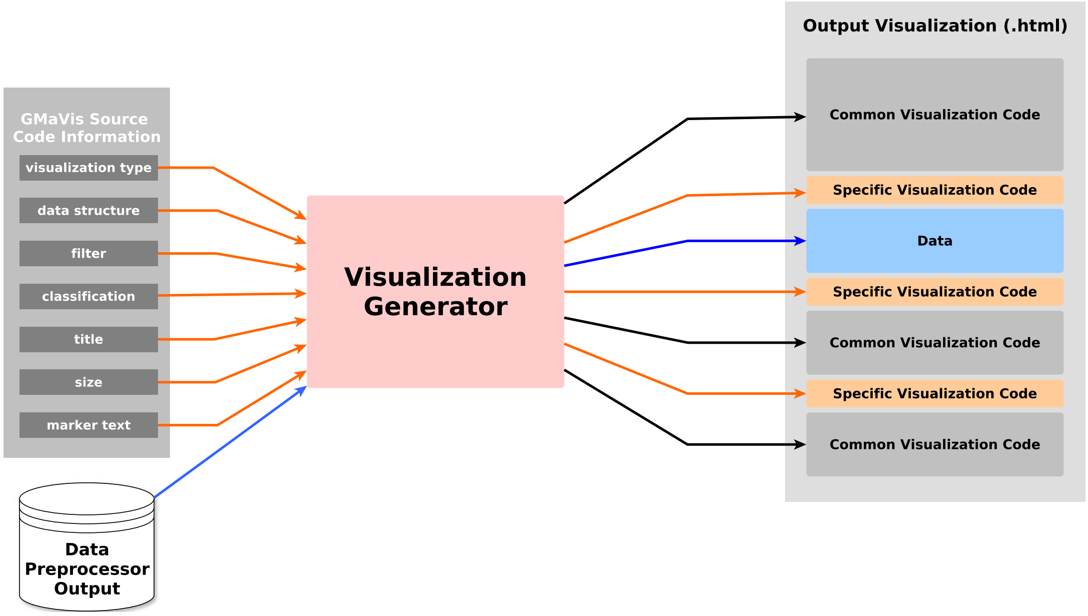

GMaVis Source Code Information
visualization type
data structure
filter
classification
title
size
marker text
Data Preprocessor Output
Visualization Generator
Output Visualization (.html)
Common Visualization Code
Specific Visualization Code
Data
Specific Visualization Code
Common Visualization Code
Specific Visualization Code
Common Visualization Code

Figure 5.9 – Data visualization generation.

The generated visualization uses Google Maps API to display the map and visual elements. This API was chosen because it offers a set of visualizations with an easy implementation, facilitating code generation. Also, Leaflet and OpenLayers, were analyzed, but they have complex libraries and do not offer significant gains for GMaVis. Even using Google Maps API, it is possible to implement visualizations from others libraries since it is created in a similar way. To do so, one must understand the library and how the information can be inserted.

6. EVALUATION AND DISCUSSION

This chapter presents the methodology used and results obtained with the GMaVis evaluation. Section 6.1 presents the methodology used to evaluate this DSL. As previously explained, we divided this evaluation into two parts. First, we analyzed the programming effort and code productivity with GMaVis and other map implementation tools. Second, we conducted a performance analysis to evaluate GMaVis' compiler and its code generation. Thus, Section 6.2 presents the results for the first experiments. Section 6.3 explains the performance analysis results.

6.1 Methodology

In order to validade GMaVis productivity and performance of parallel processing, some research questions and hipotheses were defined. Subsection 1.3.1 describes these research questions and some hypotheses. To validate GMaVis and answer the research questions, two different evaluations were performed. The first evaluation, described in Subsection 6.1.2, is an analysis of the programming effort to validate the effectiveness of facilitating visualization creation with GMaVis. This evaluation compares the effort to create a visualization using GMaVis and traditional tools, through COCOMO [Boe81] and SLOCCount [Whe16] estimation results. The second evaluation, explained in Subsection 6.1.3, is about performance. It evaluates the parallel data preprocessor to verify if it achieves better performance compared to the sequential version. Since TBB [Rei07] was also considered to be used to parallelize data preprocessor module because it offers stream parallelism support, we compared its aplication to SPar. This comparison verifies if a parallelized TBB version presents better results than the automatic generated SPar version of data preprocessor.

6.1.1 Applications and Input Datasets

Both evaluations (programming effort and performance) used the applications presented in the Subsection 4.5.2. These applications use different features provided by GMaVis, such as filters, classification, file structure specification, and logical operators. Moreover, these applications were implemented using Google Maps API, Leaflet, and OpenLayers in order to compare the effort required to create a visualization using GMaVis to different libraries with the same purpose. Also, the same datasets described in Subsection 4.5.1 were used in these evaluations, but, for performance evaluation, these datasets were replicated to reach a similar size among them. Table 6.1 describes the dataset sizes in Mbytes. Three groups of datasets were defined: (i) big with approxematelly 17 GBytes, (ii) medium with about 2.8 GBytes and (iii) small with at least 470 MBytes.

Table 6.1 – Dataset sizes.

Dataset Name	Size(MBytes)
accidents-big	17272.83
accidents-medium	2878.80
accidents-small	479.80
airports-big	17275.57
airports-medium	2879.26
airports-small	479.87
flickr-big	17192.59
flickr-medium	2865.43
flickr-small	477.57

6.1.2 Development Effort Evaluation

We evaluated GMaVis programming effort using a comparative analysis of the effort required to create geospatial data visualization using GMaVis, Google Maps API, Leaflet and OpenLayers. For this analysis, six geospatial data visualizations, presented in Subsection 4.5.2, were created. Each visualization type has two different applications that use different datasets. Also, a data preprocessor in C/C++ for each visualization was created in order to select, filter and classify data for Google Maps API, Leaflet, and OpenLayers. This is not the only way users can preprocess data, since external tools or databases can be used. However, this data preprocessor software is just for estimation and demonstrates that users must put in extra effort to preprocess data. Measuring the effort required in the *data pre-processing* phase was required because the measured libraries do not provide features for pre-processing data. On the other hand, GMaVis has built-in declarations, and does not require the implementation or use of a third type of software to preprocess data unlike other libraries.

Then, all the applications were measured using the COCOMO model [Boe81]. It is a usability model for measuring code and estimation metrics for development time and effort based on the physical source lines of code. We compare the entire development cycle for generating a visualization, including the initial process of planning, coding, testing, documenting and deploying it for users. The code required to implement each visualization was analyzed. Initially, this evaluation considered the code required to implement the visualization which includes HTML and JavaScript code. Also, an analysis of developed data pre-processing required to create the visualization using Google Maps API, Leaflet and OpenLayers was performed.

In order to apply COCOMO model, we used SLOCCount[3] tool. It is a software measurement tool, which counts the physical source lines of code (SLOC), ignoring empty lines and comments. It also estimates development time, cost and effort based on the original Basic COCOMO[1] model. The suite supports a wide range of both old and modern programming languages (e.g., C++, Javascript, HTML, and CSS), which are naturally inferred by SLOCCount and thus used for measurement [Whe16]. SLOCCount suite [Whe16] was used by a set of other researches (e.g., [GAF14, LGMF15, AGLF15a, AGLF15b, VSJ$^+$14, SET$^+$09, LRCS14, HFB05, RGBMA06, KEH$^+$09, VSJ$^+$14, STM10]) to measure software complexity.

SLOCCount counts physical lines of code that are also called "non-blank, non-comment lines" in the source code. Furthermore, physical SLOC is defined as a line ending in a newline or end-of-file marker, and that contains, at least, one non-whitespace non-comment character. Comment delimiters (characters other than newlines starting and ending a comment) are considered comment characters. Data lines only including whitespace (e.g., lines with only tabs and spaces in multiline strings) are not included in the count [Whe16].

After counting physical lines, SLOCcount performs an estimation of effort and schedule required to develop the input source code. To express effort, SLOCcount uses person-years and person-months units. Moreover, to express schedule, it uses the total years and months as units. When users use SLOCCount in default mode, the "Basic COCOMO" model is set. This model includes the effort estimation about the whole cycle to develop an application, such as design, code, test, and documentation time (both user, administrator and development documentation). This is important to consider when verifying the results since it produces high-cost estimates [Whe16].

The basic COCOMO model is usually enough to estimate development effort, but it can produce limited accuracy. When using the basic COCOMO, users do not take some important factors into consideration. In order to improve accuracy users can use the "Intermediate COCOMO" and "Detailed COCOMO" models that have some factors to perform a more detailed estimation. These enable more accurate results. Thus, if the sloccount user has information about the domain, developers, development environment and other details, it can be used to improve the model's accuracy. It is possible to change the `-effort` and `-schedule` factor parameters in the SLOCCount command line. Another important question to determine when using SLOCCount with "detailed COCOMO" is the application's mode, which can be "Organic", "embedded", or "semidetached". "Organic" is the default configuration used in Sloccount because most applications are in this category [Whe16]. Table 6.2 shows the difference between the software models available in COCOMO model.

In the COCOMO model, it is possible to use some corrective factors to determine effort. These factors are called cost drivers and enable a detailed analysis of effort according to the application, developers, environment and other issues that affect the development

[1] http://www.dwheeler.com/sloccount/sloccount.html#cocomo

Table 6.2 – Software category in COCOMO model (Adapted from [Whe16]).

Category	Description
Organic	Relatively small software teams develop software in a highly familiar, in-house environment. It has a stable development environment, minimal need for innovative algorithms, and requirements can be relaxed to avoid extensive rework.
Semidetached	This is an intermediate step between organic and embedded. This is characterized by reduced flexibility in the requirements.
Embedded	The project must operate within tight (hard-to-meet) constraints, and requirements and interface specifications are often non-negotiable. The software will be embedded in a complex environment that the software must deal with as-is.

cycle. Each cost driver has a value that is used to calculate the effort through the product of all factors. The result is the final effort factor. Table 6.3 presents the cost drivers. The chosen values for this analysis have highlighted cells.

Table 6.3 – COCOMO cost drivers (Adapted from [Whe16]).

Cost Drivers		Ratings					
ID	Driver Name	Very Low	Low	Nominal	High	Very High	Extra High
RELY	Required software reliability	0.75	0.88	1.00	1.15	1.40	-
DATA	Database size	-	0.94	1.00	1.08	1.16	-
CPLX	Product complexity	0.70	0.85	1	1.15	1.3	1.65
TIME	Execution time constraint	-	-	1.00	1.11	1.30	1.66
STOR	Main storage constraint	-	-	1.00	1.06	1.21	1.56
VIRT	Virtual machine volatility	-	0.87	1.00	1.15	1.30	-
TURN	Computer turnaround time	-	0.87	1.00	1.07	1.15	-
ACAP	Analyst capability	1.46	1.19	1.00	0.86	0.71	-
AEXP	Applications experience	1.29	1.13	1.00	0.91	0.82	-
PCAP	Programmer capability	1.42	1.17	1.00	0.86	0.70	-
VEXP	Virtual machine experience	1.21	1.10	1.00	0.90	-	-
LEXP	Programming language experience	1.14	1.07	1.00	0.95	-	-
MODP	Use of "modern" programming practices	1.24	1.1	1.00	0.91	0.82	-
TOOL	Use of software tools	1.24	1.1	1.00	0.91	0.83	-
SCED	Required development schedule	1.23	1.08	1.00	1.04	1.10	-

Some cost drivers were set for this experiment to achieve more accurate results. These driver costs were used to measure all the libraries and GMaVis. Some cost drivers were not used since they do not make sense and do not apply to this case. For example, `virtual machine volatile` was not considered in this analysis since we do not use or have virtual machines in the environment. Thus, in this case, we set this as a nominal value (1.00). Each value considered in this analysis is explained bellow.

Required software reliability This measure represents required software reliability. For example, if a software failure would risk human life, then it may be set to *very high*.

Product complexity This represents the complexity of the final product. For this measure, it is important to consider the following areas that can generate complexities in the software: control, computation, device-dependent, or data management operations.

Applications experience It expresses the knowledge and experience of a developer or team about applications that support development, such as IDEs, programming libraries, frameworks and other tools.

Programmer capability Similar to the previous item, this measure considers the overall knowledge of a team or developer. This includes not only applications, but also analysis, communication, team cooperation, and interaction.

Programming language experience It is related to the level of knowledge and experience about the project programming language and software tool. Developers with more experience in the programming language can create software with less effort than a developer with low-level knowledge.

Use of "modern" programming practices This represents the use of modern programming practices. It includes code reuse, top-down incremental development, structured design notation, top-down requirements analysis and design, design and code walkthroughs or inspections, structured code, use of libraries and other methods that improve and facilitate development.

Use of software tools This measure is related to the use of software that facilitates the development such as IDEs, programming libraries, frameworks and other tools.

To generate effort based on the attributes, each attribute is rated using one of six values from "very low" to "extremely high". Then, each value implies in an effort multiplier that results in an effort adjustment factor (EAF) [Boe81]. Thus, EAF can be used to improve the accuracy of estimation in SLOCCount. Moreover, SLOCCount performs the following calculus automatically, it will be presented the intermediate COCOMO model as follows:

(Effort)

$$E \;=\; a_i \;(KLOC)^{b_i} \; x \; EAF \tag{6.1}$$

(Development time)

$$D = c_i(Effort\ Applied)^{d_i} \tag{6.2}$$

(People required)

$$P = Effort/Development\ Time \tag{6.3}$$

Where the given a_i, b_i, c_i, and d_i are based on the values in table 6.4.

Table 6.4 – Effort estimation in COCOMO model variables (Extracted from [Whe16]).

Software Model	**a_i**	**b_i**	**c_i**	**d_i**
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Based on this table and in the chosen values, the following calculus was used to generate the effort adjustment factor(EAF).

$$EAF = 0.75 \times 0.70 \times 1.13 \times 0.70 \times 0.95 \times 0.82 \times 0.83 = 0.2685 \qquad (6.4)$$

In this analysis, we used the organic software model. Thus, in order to calculate the effort to sloccount, the following calculus multiplying the EAF by the Organic software model value of 2.4 was performed.

$$Effort = 2.4 \times 0.2685 = 0.6444 \qquad (6.5)$$

Considering that the exponent for effort in an organic model is 1.05, the following command were used to call SLOCCount and measure the effort estimation.

```
sloccount -effort 0.6444 1.05 <source_project>
```

Finally, this command line was called inside the project folder of each visualization. This was also used for data preprocessor analysis. This command results in data with information about the total physical source lines of code, estimated development effort in person-years and person-months, estimated schedule in years and months, estimated average number of developers, and total estimated cost to develop.

6.1.3 Performance Evaluation

Initially, we performed a preliminary evaluation of the execution time when processing huge quantities of data. For these tests, we replicated the YFFCC100M data set to achieve 100GB of data. We did a measurement of 10GB, 50GB, and 100GB to verify the time it takes for processing. With these results it was possible to conclude that it would be interesting to implement a parallel version of data processing, enabling it to generate a visualization using a huge dataset, for example 1 TBytes, in a reasonable amount of time.

With the parallel version of the data preprocessor, we performed a comparison between the sequential and parallel version with different input sizes to measure the throughput of applied parallelism. Speedup and efficiency calculus were considered in this analysis. However, execution time and throughput were enough to express the results and performs the comparisons required in this context. In this evaluation, each algorithm was executed ten times for each input size and number of processors, both in a parallel and sequential version. These tests were performed using a multi-core computer Blade Dell PowerEdge M610 with the following specification.

- Two processors Intel Xeon Six-Core E5645 2.4GHz Hyper-Threading
- 24 GBytes of memory.
- Software:
  - Operational System: Ubuntu 14.04 LTS
  - C/C++ Compiler: G++ 5.2
  - SPar Compiler: CINCLE

In order to verify if the parallel generation to SPar is efficient, a parallel version of TBB was implemented by a third person. Both TBB and SPar code were compared using the same method for sequential and parallel version.

Considering three dataset sizes presented in Subsection 4.5.1, six applications and ten executions were performed. Thus, a total of 180 executions for each sequential and parallel version. Parallel versions were executed both using SPar and TBB, using 3 to 24 threads[2]. Thus, there were 180 executions for sequential version, 4320 for SPar and 4320 for TBB parallel version, totaling 8820 executions. Execution time were measured using the C++ library std::chrono[3]. This allowed a timestamp to be created before and after a specific process, calculating the time it took to perform. Thus, the execution time of the whole pipeline was collected, including file loading, processing, and output saving.

With the execution time and the input size, it was possible to calculate the throughput. Throughput is the average output of a determined production process per unit time [Rus08]. It defines a work in process (WIP) and a determined unity of time, also called cycle time (CT). Little's Law equation enables the calculation of throughput since it is defined as the relation between WIP, cycle time and throughput. In order to measure data preprocessor throughput this equation was adapted [Gus11] to fit this purpose. Little's Law equation is presented as follows.

$$TH = \frac{WIP}{CT} \tag{6.6}$$

Applying this equation to this work, consider t_i as the execution time of a single execution i. As mentioned previously, each execution configuration was performed ten times.

[2] It starts in 3 because there are three stages in the pipeline (loading, processing and saving output). Thus, parallel versions cannot have less than three threads.

[3] http://www.cplusplus.com/reference/chrono/

Thus, the following equation was used to calculate an average of ten executions of a data preprocessor generated for a *d* dataset. We also calculated the standart deviation to verify values consistency.

$$\bar{t}_d = \frac{\sum_{i=1}^{10} t_{id}}{10} \tag{6.7}$$

With the average execution time in seconds, for a specific *d* dataset processed in data preprocessor, it is possible to calculate the throughput using the dataset size s_d in bits using the equation below.

$$TH_d = \frac{s_d}{\bar{t}_d} \tag{6.8}$$

Finally, the resulting TH_d provides the throughput value when processing a determined quantity of data for each time unit. In this analysis, we used a measure in Mbytes/seconds.

6.2 Code Productivity Results

To verify if GMaVis offers more code productivity and reduces the effort required to implement a visualization, we constructed six applications using the three visualization types offered in GMaVis. Each visualization was created using GMaVis, Leaflet, Google Maps API, and OpenLayers and is explained in detail in Section 4.5. GMaVis does not require users to implement software or handle data manually to perform data pre-processing. In these results, we measure the necessary code to automatically implement data processing, according to each visualization type. Since some applications do not require filtering or classifying, we split the code and measured each part. It is possible to verify, in all the graphics, that the `structuring` phase takes much more time. It requires more time to develop due to the higher number of lines to implement. Before analyzing the results, it is important to highlight that at first, the development time results can seem very high. However, this measure considers the whole cycle to develop software. It estimates the time to plan, code, test, document and deploy the software.

The graph in Figure 6.1 shows the results for Heatmap applications. It is important to highlight that this type of visualization does not implement classification. This visualization type only displays a layer with different colors to express the frequency of elements in the location. It is possible to verify that GMaVis obtained better results than Google Maps API, Leaflet, and OpenLayers since it presents a lower estimation time to develop. Also, it is possible to verify that Leaflet requires less code to implement a map visualization than Google

Maps API and OpenLayers. The first application does not implement a filter. This reduces the required code in data processing for Google Maps API, OpenLayers, and Leaflet.

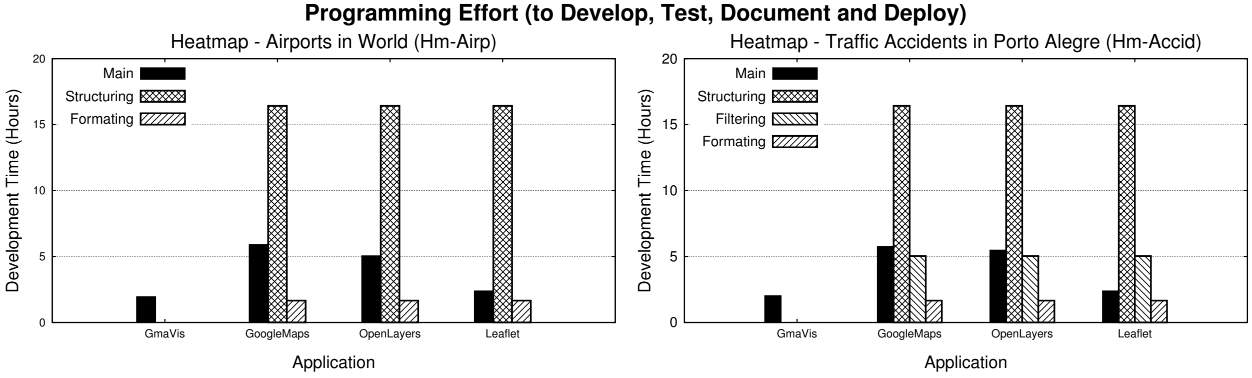


Figure 6.1 – Code productivity results for heatmap applications.

Two applications for clustered map were measured. Figure 6.2 present the results for their development time estimation. Similar to the first graph, this shows the effort for both the main and data pre-processing. Here GMaVis obtained a lower time estimation to develop since it required fewer lines of code and did not require additional implementations to process data. Moreover, it is possible to verify that Leaflet kept a similar result compared with OpenLayers and Google Maps API. This is because this implementation is for clusteredmaps and the code required to create it is different from the first heatmap application. Application `Airports in World` presented less development time than `Flickr Photos with "Computer" tag"` since it does not use filtering.

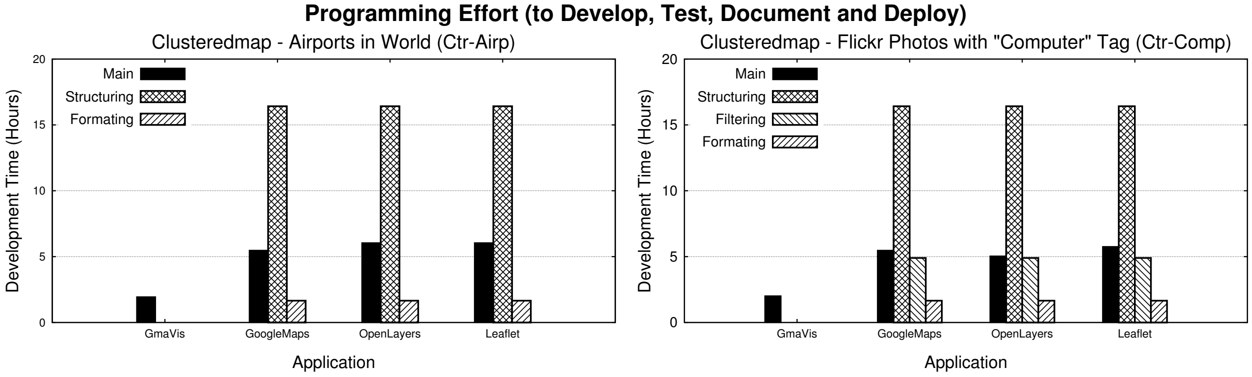


Figure 6.2 – Code productivity results for clustered map applications.

Even with preliminary results that show less development time for markedmap applications, two data visualizations were created to confirm this. Figure 6.3 shows programming effort estimation in development time for these visualizations. Different from the previous results, these use both filtering and classification, requiring more time to develop since it includes the creation of mechanisms or use of external software. If a user uses a C++ code, it would require the time expressed in the graph. Both visualizations present less development time in `main` code for GMaVis. In this example, OpenLayers presented the second less development time in main code, since it required less code to implement this kind of visualization.

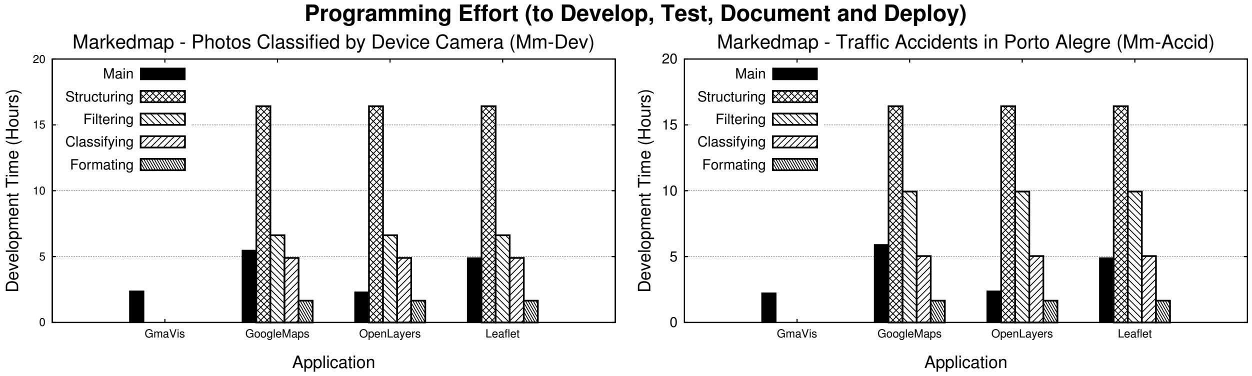


Figure 6.3 – Code productivity results for marked map applications.

SLOCcount also estimates the cost required to implement a software. Figure 6.4 and Table 6.5 presents the results in dollars for developing each visualization considering just the main code. In this graph, the cost required to perform the pre-processing was not inserted or considered. The following abbreviations were used in this graph.

Ctr-Airp Clusteredmap - Airports in World

Ctr-Comp Clusteredmap - Flickr Photos with "Computer" Tag

Hm-Airp Heatmap - Airports in World

Hm-Accid Heatmap - Traffic Accidents in Porto Alegre

Mm-Dev Markedmap - Photos Classified by Device Camera

Mm-Accid Markedmap - Traffic Accidents in Porto Alegre

Table 6.5 – SLOCCount estimation about cost to develop each application.

Application	**GmaVis**	**Google Maps**	**OpenLayers**	**Leaflet**
Ctr-Airp	$33,00	$77,00	$105,00	$105,00
Ctr-Comp	$37,00	$77,00	$63,00	$89,00
Hm-Airp	$33,00	$96,00	$65,00	$53,00
Hm-Accid	$37,00	$93,00	$77,00	$53,00
Mm-Dev	$56,00	$77,00	$49,00	$60,00
Mm-Accid	$46,00	$98,00	$53,00	$60,00

Here, unlike `Mm-Dev` application, GMaVis requires less development cost. This demonstrates that even not considering the fact that GMaVis avoids the implementation or use of external software to process data, it can improve productivity for developing the

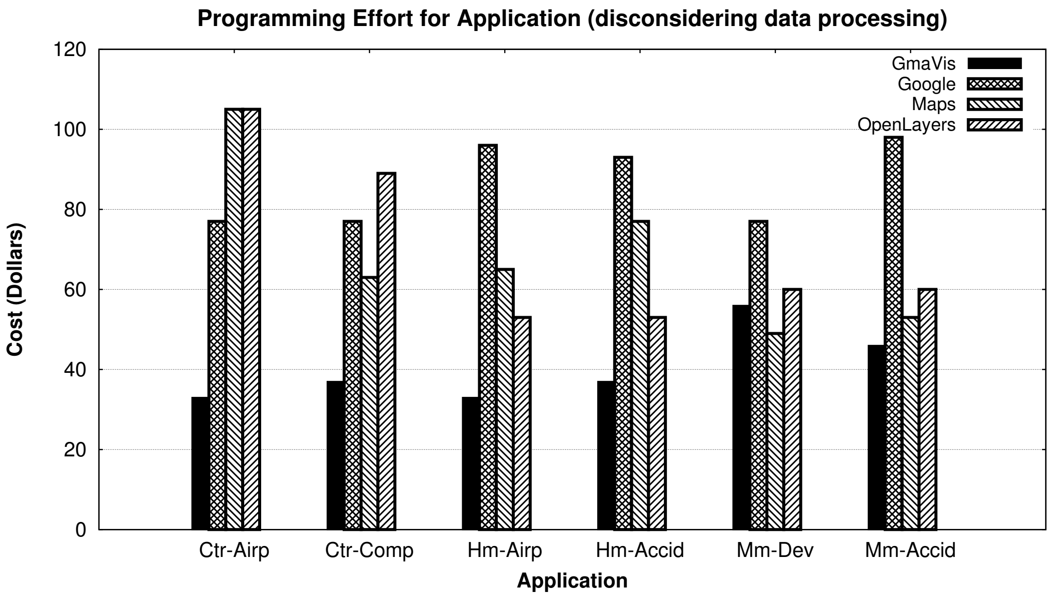


Figure 6.4 – Cost estimation results for each application.

provided geospatial visualization maps. This can be achieved since GMaVis does not require the implementation of a set of programming elements, such as functions, variables, methods or any additional code. Also, it requires less lines of code to implement most analyzed visualizations, when compared to other libraries. Table 6.6 shows the SLOC results for each visualization.

Table 6.6 – SLOC (Physical Lines of Code) for each application.

Application	GmaVis	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

With the results presented, it is possible to answer the first research question (**Q1**). It asks if GMAVIS can reduce the programming effort for creating a geospatial visualization. Two hypotheses were formulated. It is possible to confirm hypothesis 1 (**H1**) with the results presented that demonstrate a significant reduction in development time and cost to develop the provided visualization types using GMaVis compared to other visualization libraries. Also, hyphothesis 2 (**H2**) is proven when a data preprocessor code is measured and inserted for each application. With this estimation, it is possible to conclude that GMaVis also reduces the effort required to develop when the abstraction of *data pre-processing* phase is provided.

6.3 Performance Results

We measured the completion time of the data preprocessor during this research [LGMF15]. The goal was to verify how long it takes to process a huge quantity of data and generate an output. We could not compare this with traditional libraries once the data pre-processing is done using different approaches, or in some cases, manually. Table 6.7 presents the results for data *pre-processing* and *data to visual mappings* phases. *Data pre-processing* encompasses the total execution time to open, structure, filter, classify the input data, and save the output file. *Data to visual mappings* phase includes the total execution time to generate the visualization and display it to the user. We varied the input data using 10GB, 50GB, and 100GB of data.

Table 6.7 – Completion times (seconds) (Extracted from [LGMF15]).

Size	Data Pre-processing	Data to Visual Mappings - Google Maps API
10GB	110.4948 (Std. 0.9763)	2.910 (Std. 1.6084)
50GB	544.0506 (Std. 9.4225)	3.2738 (Std. 2.0663)
100GB	1098.9284 (Std. 19.0383)	3.8536 (Std. 2.7584)

If we consider the complexity time of the linear search, which is $\mathcal{O}(n)$, we can verify a similar completion time in the data pre-processing as it grows in a linear way according to the input size. Consequently, it is possible to observe an increasing order of approximately 11 times ($11.0494 \times 10GB$, $10.8810 \times 50GB$, and $10.9892 \times 100GB$). We can conclude that as the input data size modifies, the execution time to process the data will be related to the completion time and input size, keeping the time complexity in a linear order.

These execution time results demonstrated that as the input data size is modified, the times in *data to visual mappings* phase have a minimal difference. However, *data pre-processing* process expresses a huge difference between the different input sizes. As a consequence, it is important to consider that data pre-processing process has a higher computational cost when the input size increases. We also observed that automatic data pre-processing optimizes the visualization implementation. If users develop using traditional libraries, handling a huge quantity of data manually can take much more time.

Thus, the data preprocessor was parallelized to speed up the visualization creation. Moreover, we recognized that this parallelization should be generated in GMaVis compiler since the data preprocessor is entirely generated in the compilation of GMaVis source code. This required a study to verify if GMaVis compiler could generate this parallel code to a parallel programming interface. Three interfaces, Fastflow, TBB, and SPar were considered. Other interfaces were not considered since the application type requires the interface to support stream parallelism. Thus, OpenMP could not be used to parallelize this application.

SPar was chosen to parallelize because it facilitates the creation of parallel code. While other interfaces require changes in sequential code to introduce parallelism, SPar only requires the insertion of simple annotations in the sequential code specifying input, output, and replications of stages. To compare the increase of code require to implement the parallel version of the data preprocessor with SPar and TBB, we used the generated code for the data preprocessor without SPar annotations, generated code with SPar annotations and TBB implementation of data preprocessors manually created. Table 6.8 demonstrates the sequential and parallel lines of code for the data processor that each application uses.

Table 6.8 – Lines of code to parallelize data preprocessor using SPar and TBB.

Application	**Sequential**	**SPAR**		**TBB**	
	SLOC	SLOC	***Code Increase*** (%)	SLOC	***Code Increase*** (%)
CM_CP	210	216	2,857	274	30,476
CM_AIR	199	205	3,015	263	32,160
HM_AC	210	216	2,857	274	30,476
HM_AIR	199	205	3,015	263	32,160
MM_DEV	224	230	2,678	288	28,571
MM_ACID	216	222	2,777	280	29,629

It is possible to verify that SPar requires fewer lines of code to implement the parallel version than TBB. This is important in code generation because using SPar hard modifications did not have to be performed in the compiler. A few lines of code had to be added before some statements to inform the beginning of a stream and stage. If we use TBB to parallelize, it would be required to generate functions and change the code. Figure 6.5 highlights the results in this table, presenting the increase percentage of lines of code compared with the sequential version for both SPar and TBB interfaces.

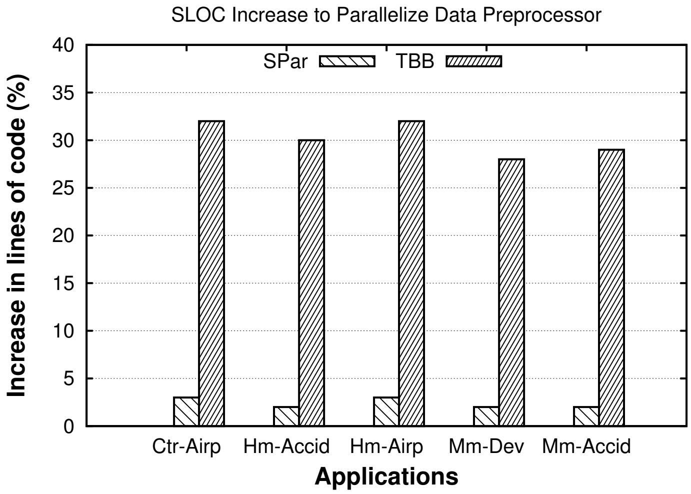


Figure 6.5 – Comparison of SPar and TBB for parallelize data preprocessor.

Since SPar is in a beta version, under development, it was expected that performance results would not achieve the same provided by the TBB manually created data preprocessors. However, the following results demonstrate that SPar can provide better results in some cases. Also, the following graphic demonstrates average results of execution time and throughput from 10 executions in each configuration. It was used three data sizes with applications in the sequential, parallel with SPar, and parallel with TBB versions.

The objective was to verify if GMaVis increases performance using parallel processing in the data preprocessor. Also, we analyzed if SPar parallel code generated in GMaVis is adequate and provides the same level of performance when compared with a version manually parallelized by an experienced parallel developer.

The first application is the clusteredmap named `Airports in world`. Figure 6.6 demonstrates the results for execution time and throughput using the three dataset sizes and replications in parallel processing from 1 to 12. Replication 0 signifies the sequential execution. This results demonstrates better execution time for SPar with a big dataset. Throughput values for SPar were higher than TBB for the big dataset, as expected when verifying the execution time. Thus, when using SPar to process this dataset with the filters applied in the used application, the amount of data processed each second is larger than that processed in the sequential or TBB version.

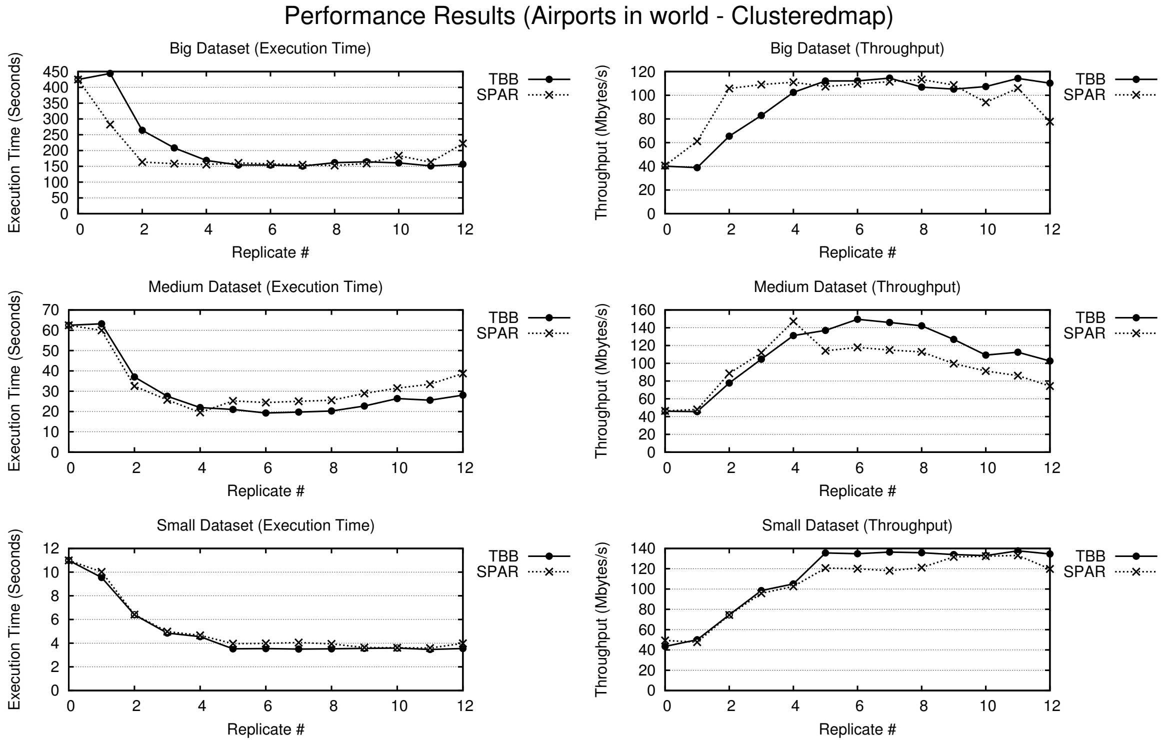


Figure 6.6 – Airports in world (clustered map).

However, for medium and small dataset sizes the results are not the same. In the medium dataset, SPar demonstrates better performance from 1 to 4 replicates, while TBB has better performance after five replicates. In small dataset sizes, results of execution time are close for all the replication configurations. In throughput results, the differences in both results are shown. It is possible to verify that TBB has better performance after a high number of replications than SPar. This may be because the increase in replications increase access to shared memory. TBB provides highly concurrent container classes that enable a fine-grained locking and lock-free techniques, overcoming this problem.

Figure 6.7 presents the results for the clusteredmap `Flickr Photos with "Computer" Word as Tag` data preprocessor. The same configuration of previous results was used in these executions and results. Since this application uses different filters to select data, it is expected that the results will be different. In this case, we can verify that there is a performance gain in both interfaces. In the big dataset, TBB has a higher execution time than SPar version, thus resulting in a lower throughput rate for TBB. However, in medium and small dataset sizes the behavior is different, presenting almost an equal execution time for both SPar and TBB until four replicates, and a difference to 12 replications.

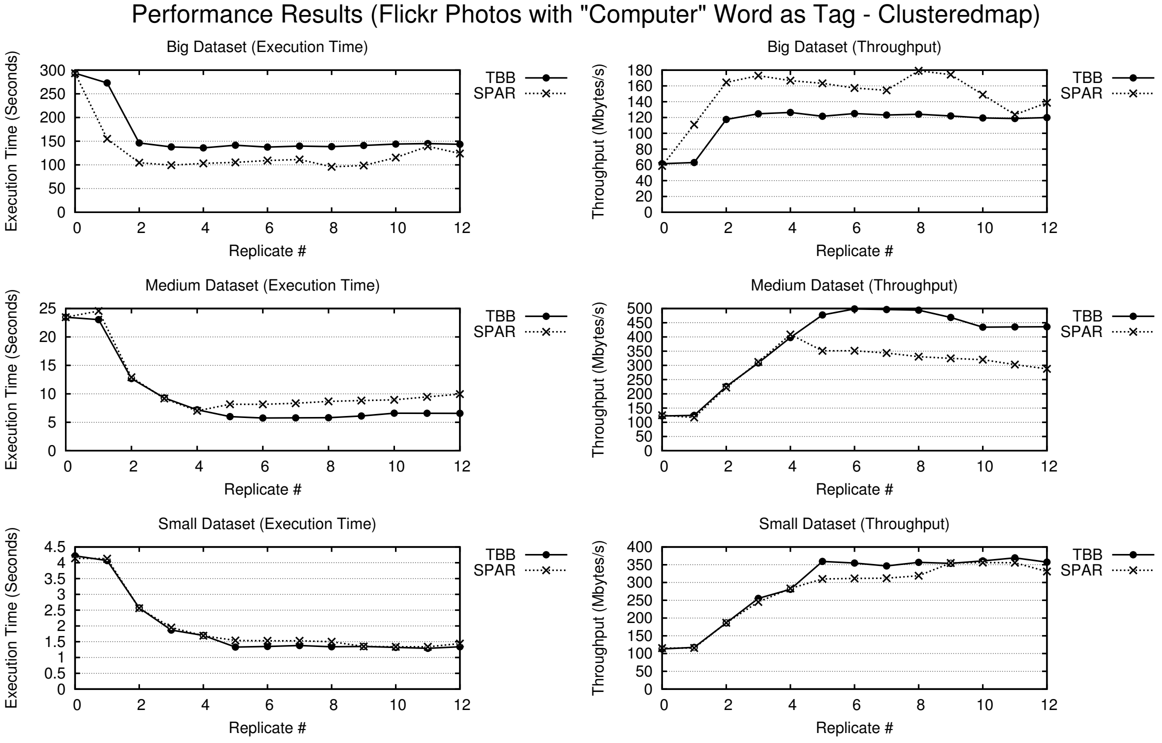


Figure 6.7 – Flickr photos with "computer" word as tag (clustered map).

Throughput results are similar compared with previous applications in three dataset sizes. There is the same throughput rate until four replicates and a higher difference appears after this. Also, it is possible to verify that after a number of replications, the throughput does not change too much. This happens because this application has a serialization processing in the beginning and the end of processing. These steps are parallelized using temporal overlap as explained in Chapter 5. However, increasing replication improves performance only in the middle stage, responsible for structuring data, applying filters and classification. If this step achieves a small execution time compared with the first and last stages (input/output of data), the increase in replication will not result in significant increases in performance when analyzing the whole execution time.

The results demonstrated in Figure 6.8 are about the heatmap `Airports in the world` data preprocessor. This presents similar behaviors as seen in the previous applications. However, with a big dataset size, Spar had execution times closer to those in the TBB version between four and nine replicates. This occurred because this visualization did not implement any filtering or classifying, performing just data structuring and formatting processes in the processing stage. Thus, even increasing the number of replicates, it presented an execution time very close to that of TBB because most of the processing is loading and saving data to the disk. The same happens in medium and small dataset sizes, where execution time is almost equal for both interfaces.

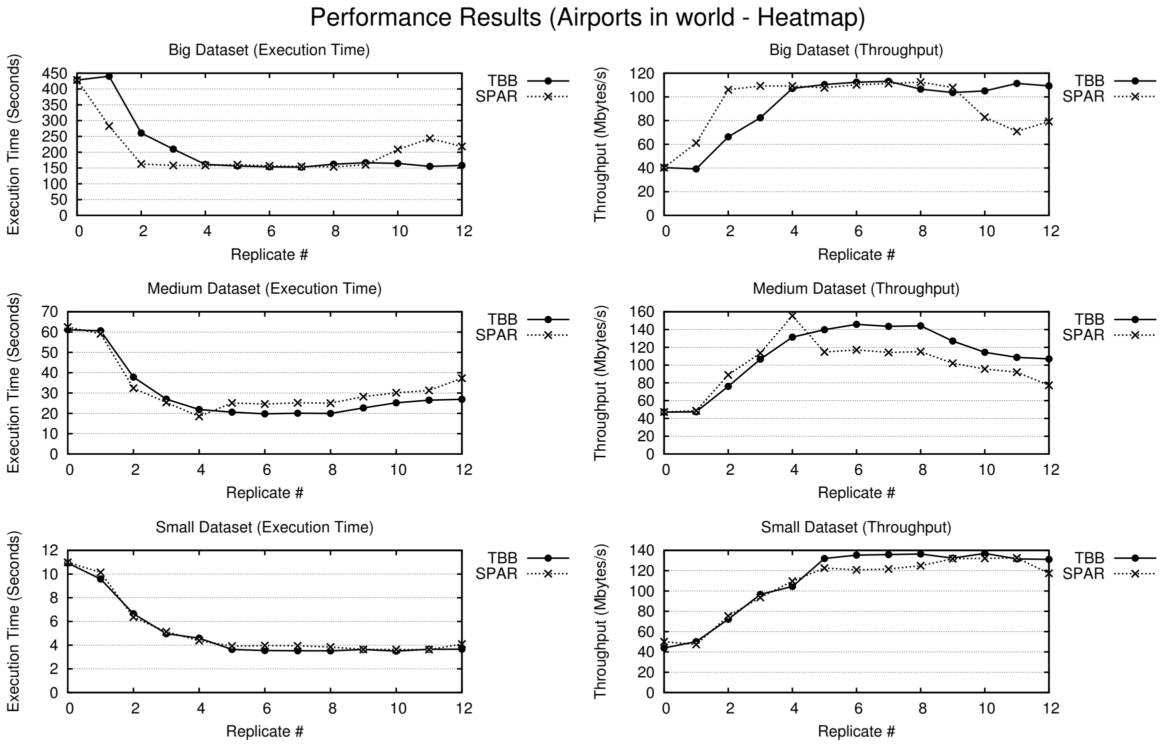


Figure 6.8 – Airports in world (heatmap).

Though GMaVis has been shown to have a performance gain using parallel programming, it is possible to verify that this does not continue as the number of processor units change. This behavior was expected due to the sequential processing in the initial and final stages. Figure 6.9 presents results for the Heatmap `Traffic Accidents in Porto Alegre, Brazil` , where this also happens. However, in medium data size this graph presents an increase in execution time after eight replicates. This visualization applies a filter in data. It is possible that increasing replication in this step resulted in an overhead to join data and save it at the last stage. Small data size did not have this same behavior since the data was small and its joining was faster. In a big dataset size, this also seems to occur. However, the execution time to load and save a file is too high and this becomes less evident.

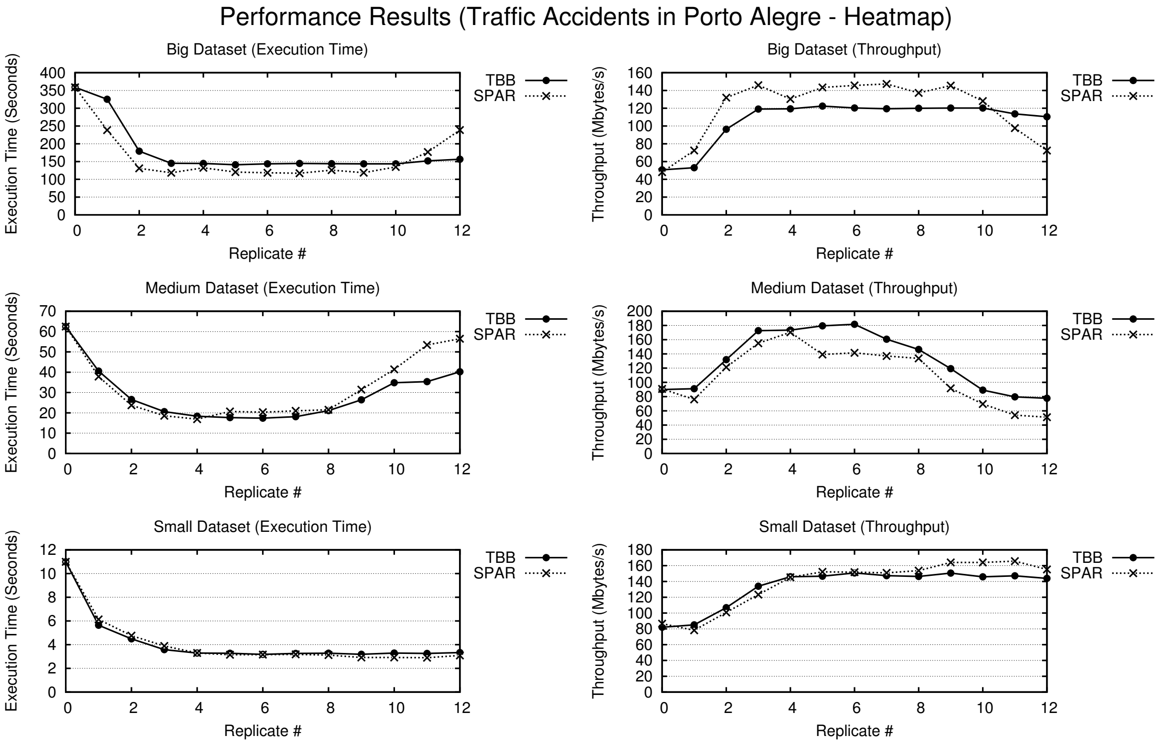


Figure 6.9 – Traffic accidents in Porto Alegre, Brazil (heatmap).

Throughput graphs present this behavior through a decrease in throughput rate for both SPar and TBB versions. With a big dataset size, the throughput for SPar decreases more than TBB, based on the relationship with execution time. Also, medium dataset size shows a curve where initially there is an increase of throughput and after six replicate both SPar and TBB start to decrease rates. Nonetheless, this parallel data preprocessor still improves performance for a limited number of replicates.

Figure 6.10 shows the results of data preprocessor used in the markedmap application for `Traffic Accidents in Porto Alegre, Brazil`. This visualization has classification, resulting in a costly processing stage. With a big dataset size, both TBB and SPar demonstrate similar results, with SPar providing less execution time than TBB. The execution time is the same as in the parallel version even as replication increases after the fourth replicate. The big dataset size graph presents a higher throughput rate for SPar. However, both TBB and SPar achieve similar results with 12 replicates. The same behavior is observed in the last result for the medium dataset size for this application. Since it uses the same dataset of about 42 fields, it is possible to conclude that the number of registers for each piece of data processed has a smaller number of registers then in other datasets. Thus, processing a small number of registers takes little time compared to load/save data, and it generates a bottleneck in the output saving process. With the small dataset size, this is not observed due to the small amount of data, which is processed quickly in both loading and saving data.

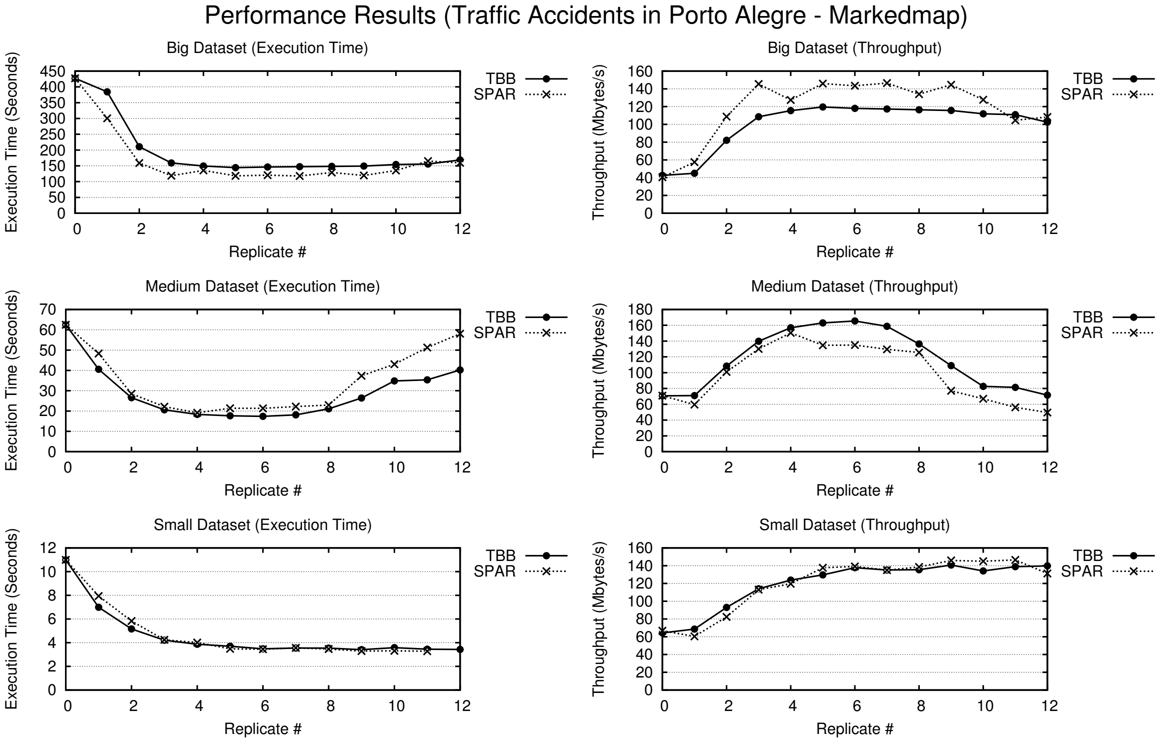


Figure 6.10 – Traffic accidents in Porto Alegre, Brazil with classification (marked map).

The throughput of both small and medium dataset sizes presented favorable results for TBB. With a medium dataset size, SPar had an execution time close to TBB, but after four replicates, it started to decrease. TBB throughput also decreased but presented higher rates. With a small dataset size, both interfaces have a similar rates, keeping the throughput approximately 140 Mbytes/s.

Figure 6.11 shows the last performed executions of the data preprocessor of `Flickr Photos Taken in 2014 Classified by Used Camera Brand`. This application used a marked map visualization type, applying filtering and classifying. Its results for execution time are similar to the first analyzed application. With a big dataset size, SPar presented a lower execution than TBB. It also had throughput higher than TBB. Medium dataset size initially presented execution time values to SPar and after five replicates this changes, but there is a small increase rate as replication occurs. This is highlighted in throughput results, where rates start to decrease after replicate 5 for SPar and 6 for TBB. Finally, small dataset size presents better results when compared with sequential time, there is not significant increase in execution time after four replications.

Considering that GMaVis data preprocessor requires sequential reading and saving of data directly to the disk, results presented good execution times and speed up for the parallel version of the data preprocessor. This answers research question two (**Q2**) by

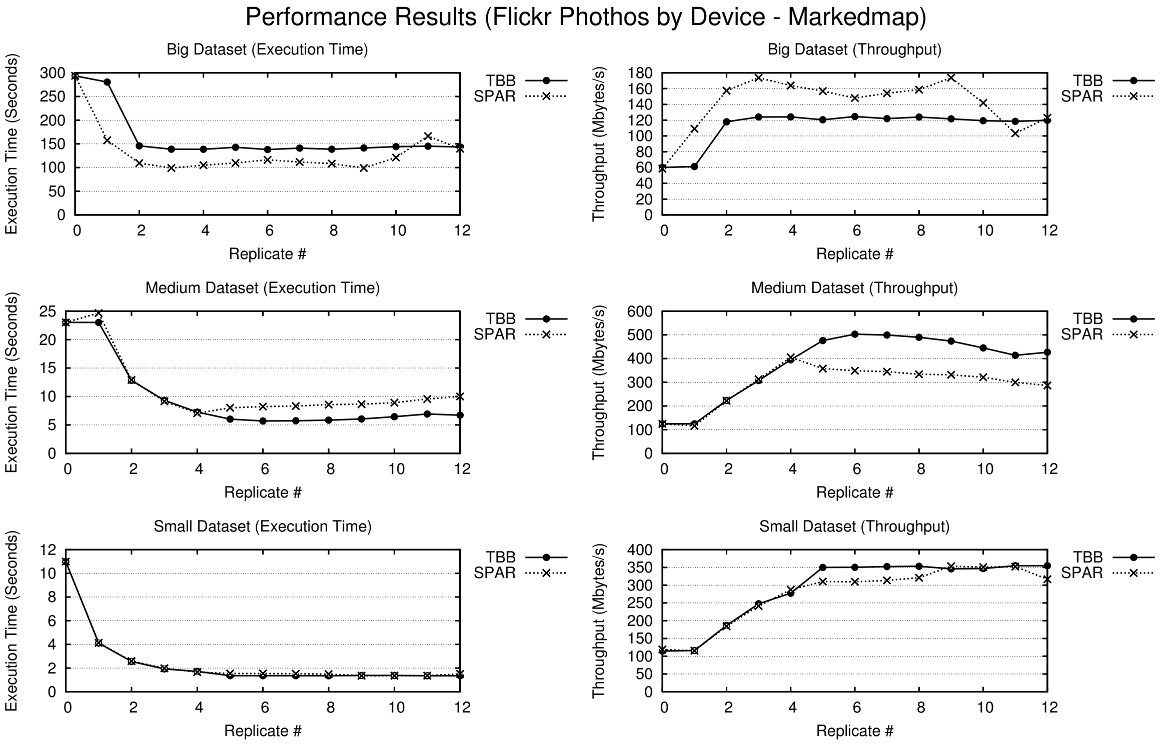


Figure 6.11 – Flickr photos classified by camera brand (marked map).

proving the third hypothesis (**H3**), since GMaVis can generate SPar annotations and improve performance using parallel processing. These results highlight that GMaVis' data preprocessor did not present a parallel version that scales as more processor unities are addressed to execute because this type of application presents these bottlenecks. A way to avoid this kind of bottleneck is to use multiple disks to perform loading and storing in parallel, and this is possible using distributed file systems. Since we have focused on multi-core architectures in this first implementation, it was not possible to parallelize for distributed memory architectures. However, SPar goals are to offer automatic parallelization to distributed memory architectures in the future, and it will be possible to generate sequential, shared-memory or distributed memory parallel code using the same annotations. All of this can be done without modifying the code generation in the GMaVis compiler.

The described execution time and throughput results presented a gain in performance when compared to the sequential data preprocessor version. It enabled quicker creation of data visualization, mainly when dealing with big datasets where the speed is much more evident for users. The application of a stream parallelism model was possible here, a temporal overlap of stages to process loading/writing data in parallel could be done. Also, the comparison between SPar and TBB further enforces that even more that SPar was the correct choice to generate parallel code, because it not only reduced the required code to be generated in GMaVis, but also presented good results when compared to a code created by an experienced developer in parallel programming. The comparison between TBB and

SPar also proves hypothesis four (**H4**), showing that both generation and execution for SPar improves and facilitates the code generation and processing in GMaVis.

7. CONCLUSIONS

In this work, we proposed GMaVis, a DSL (domain-specific language) for geospatial data visualization creation to offer features for users to create data visualizations with only a few lines of code by using a high-level description language to specify the visualization. We also provides a compiler that receives GMaVis source code and data as input to generate the data visualization automatically. Both GMaVis and the compiler were described in this work, including grammar, compilation phases, internal processing, and parallel code generation.

This DSL used SPar to parallelize the data preprocessor since it provides an easy way to implement parallel code both in a parallel or sequential way by just changing a compiler argument. It enabled the generation of the same code to be used in both architectures, whereas other interfaces would have required the implementation of two different code generations. Furthermore, this DSL generates parallel code to run on multi-core systems. It aims to generate parallel code to run in distributed memory systems, however, shared memory systems are more accessible for domain users than a cluster with multiple computers. Therefore, to create this DSL and enable its first use, we initially focused on multi-core systems.

Also, two research questions were formulated in order to evaluate this DSL in respect to code usability and performance. In order to answer these questions, two experiments for analyzing programming effort and data processing execution time were performed. Furthermore, both research questions, methodology and results are briefly described bellow.

Q1 - ***Is it possible to create a DSL (GMAVIS) to reduce the programming effort for creating geospatial visualization?***

To answer **Q1**, two evaluations of GMaVis were performed and analyzed. First the programming effort estimation data produced by SLOCCount was used to compare the effort required for creating a information visualization when using GMaVis to other libraries such as Google Maps API, OpenLayers, and Leaflet. Results demonstrated that even when providing the abstraction of complexities in the *data pre-processing* phase and totally parallel programming abstraction, GMaVis could reduce the programming effort and cost required to implement the three supported data visualizations (clusteredmap, heatmap, and markedmap). Furthermore, it measured an application that could perform data processing for the compared libraries, to estimate the effort required if users were to implement it. Thus, it is possible to confirm that GMaVis can reduce not only the programming effort but also the cost of development for creating geospatial data visualizations (*i.e.*, markedmaps, clusteredmaps and heatmaps).

Q2 - ***Can the parallel code generated by this DSL speed up the data processing of raw geospatial data?***

In respect to **Q2**, an evaluation in a parallel data preprocessor that was generated in the GMaVis compiler using SPar. This evaluation used execution time and throughput of six applications parallelized with SPar using the GMaVis compiler and manually with TBB, using three data loads and many configurations of parallel processing replication. The objective was to verify if it has better performance in parallel execution and to analyze if GMaVis generates the correct annotations to SPar in order to have good performance. Thus, we compared GMaVis generated code with SPar annotations to the TBB manually parallelized version of the data preprocessor. Results demonstrated that both versions increased performance through decreasing the execution time when compared to the sequential version. Also, both SPar and TBB offered an increase in throughput for the data preprocessor, enabling the processing of more data in less time compared to the sequential version.

7.1 Contributions

This work contributes by providing a domain-specific language that abstracts complexities in the whole visualization creation workflow, since it includes the *data pre-processing* phase features and does not require users to handle data manually. Also, the proposed DSL abstracts all parallelism. Users do not need to think about computer architecture for parallel processing, interfaces, communication, synchronization or any other parallel programming complexities. They instead focus on the description language to express the visualization using their data.

Another contribution of this work is to enable users to process huge files in multi-core systems when implementing a visualization. This makes it easier for users that have to create a visualization of datasets that have large files, bigger than available memory. Through a study of different selected algorithms, it was possible to implement an efficient data search in unsorted data and use an effective memory management by splitting data.

Also, estimations prove that this DSL enables users to create data visualization with less development effort, time and cost. This is possible because GMaVis has a simple grammar with few declarations to specify only required information to generate the visualization. Furthermore, this grammar is recognized by a built-in compiler. This compiler transforms GMaVis' source code and data into the final visualization, creating all of the visualization steps automatically and performing data processing in parallel through the use of SPar. Moreover, the parallel code generated in GMaVis to SPar demonstrated execution time and throughput results close to those manually implemented in a TBB version of a parallel data preprocessor. This shows that GMaVis also can efficiently generate the annotations for SPar during data preprocessor generation.

7.2 Identified Limitations

During this DSL creation and evaluation, some limitations were identified. The first limitation is due to the system architecture used to parallelize the code. Shared memory architectures have a bottleneck when reading data from disk, which requires sequential processing. This step in data processing made it difficult to parallelize and achieve good performance results compared to the sequential version. This is because the data must be read from the disk, processed and then written again. Both the initial and final step is performed sequentially. However, the parallelism to shared memory architectures were chosen because currently most computers have this architecture and it is more accessible than a cluster or grid. Thus, the first version of GMaVis was implemented targeting these types of architectures. In future work, the plan is to improve this DSL to perform in distributed memory architectures.

Since GMaVis is an external DSL that has a high-level description language, it offers low expressiveness. This is because to offer a higher level of abstraction, it is required to keep some features hidden from users that allow changing some visualization details, such as colors, icons, and specific sizes. However, GMaVis generates the visualization with a readable code, and it enables experienced users to change the final visualization if needed.

This DSL has focused in geospatial data visualizations. However, data analysis may require the use of different visualization types/techniques to achieve its objective. Currently, GMaVis only offers three visualization types. If users need to implement another type of visualization, it will not be possible with this DSL. However, the insertion and support of other visualization types in is planned for the future, using geospatial data.

The data preprocessor in GMaVis only supports non-hierarchical data files like CSV or tab separated files. Hierarchical data files such as JSON or XML were not supported yet. However, the creation of a pre-parser for these files are planned for future work. Also, it is possible to use or convert a hierarchical into a non-hierarchical data file.

7.3 Future Work

For future work, this high-level interface will be expanded to offer the most commonly used visualization types for big data analysis and abstract complexities throughout the whole visualization creation workflow. Thus, many domain users may use this DSL. This will support a set of visualization types which will facilitate users to create and modify their visualization by simply changing a few lines of source code. These visualizations may be inserted as modules in the DSL's compiler. Also, it is possible to use other libraries or generate them from scratch, and use parallel processing in *data to visual mappings* phase for offering 3D visualizations using GPU or multi-core processing.

Also, data declarations can be extended in this interface to offer smart classifications and data selection through the use of machine learning and data mining algorithms. The main challenge here is to apply data classification in pieces of data, since GMaVis data preprocessor splits data to process. Moreover, extending this language grammar is possible by adding more logical operators and filtering options, data values and enabling users to specify data directly from the web.

Furthermore, it is possible to improve this DSL through the use of distributed memory architectures for better performance and execution time in data processing. This also includes the possibility of using the MapReduce model, since it can provide distributed data processing and has been used in other projects [MR14, Gup15, RS14, LLA14] with similar objectives. After this implementation, it will be possible to create a smart scheduler to verify the best target architecture (distributed or shared memory) to process input data in parallel. This choice will be made based on variables such as, input size, filters, classification and other issues that may influence execution time.

REFERENCES

[AGLF15a] Adornes, D.; Griebler, D.; Ledur, C.; Fernandes, L. G. “A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures”. In: Proceedings of the 27th International Conference on Software Engineering & Knowledge Engineering, 2015, pp. 6.

[AGLF15b] Adornes, D.; Griebler, D.; Ledur, C.; Fernandes, L. G. “Coding Productivity in MapReduce Applications for Distributed and Shared Memory Architectures”, *International Journal of Software Engineering and Knowledge Engineering*, 2015, pp. 4.

[ART10] Aldinucci, M.; Ruggieri, S.; Torquati, M. “Porting Decision Tree Algorithms to Multicore Using FastFlow”. In: Proceedings of the Machine Learning and Knowledge Discovery in Databases, 2010, pp. 7–23.

[ASU06] Aho, A. V.; Sethi, R.; Ullman, J. D. “Compilers: Principles, Techniques, and Tools (Second Edition)”. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, 1000p.

[Bar15] Barney, B. “Introduction to Parallel Computing”. Source: https://computing.llnl.gov/tutorials/parallel_comp/, 2014 (accessed October 11, 2015).

[Boe81] Boehm, B. W. “Software Engineering Economics”. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981, 767p.

[CCQ$^+$er] Choi, H.; Choi, W.; Quan, T.; Hildebrand, D. G.; Pfister, H.; Jeong, W.-K. “Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems”, *IEEE Transactions on Visualization and Computer Graphics*, vol. 20–12, 2014 December, pp. 2407–2416.

[CGS97] Culler, D. E.; Gupta, A.; Singh, J. P. “Parallel Computer Architecture: A Hardware/Software Approach”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, 1056p.

[CJvdP07] Chapman, B.; Jost, G.; van der Pas, R. “Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)”. London, England: The MIT Press, 2007, 23p.

[CKR$^+$12] Chiw, C.; Kindlmann, G.; Reppy, J.; Samuels, L.; Seltzer, N. “Diderot: A Parallel DSL for Image Analysis and Visualization”. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2012, pp. 111–120.

[CL11] Cormen, T. H.; Leiserson, C. E. “Introduction to Algorithms”. London, England: The MIT Press, 2011, 1312p.

[CMS99] Card, S. K.; Mackinlay, J. D.; Shneiderman, B. “Readings in Information Visualization: Using Vision to Think”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, 712p.

[DSR11] Devi, S. G.; Selvam, K.; Rajagopalan, S. P. “An Abstract to Calculate Big O Factors of Time and Space Complexity of Machine Code”. In: Proceedings of the International Conference on Sustainable Energy and Intelligent Systems, 2011, pp. 844–847.

[Fow10] Fowler, M. “Domain-Specific Languages”. Massachusetts, USA: Pearson Education, 2010, 640p.

[GAF14] Griebler, D.; Adornes, D.; Fernandes, L. G. “Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures”. In: Proceedings of the 26th International Conference on Software Engineering & Knowledge Engineering, 2014, pp. 25–30.

[Gao15a] Gao, L. “Bison Tutorial”. Source: http://alumni.cs.ucr.edu/~lgao/teaching/bison.html, January 2015 (accessed November 09, 2015).

[Gao15b] Gao, L. “FLEX Tutorial”. Source: http://alumni.cs.ucr.edu/~lgao/teaching/flex.html, January 2015 (accessed November 26, 2015).

[Gha07] Ghanbari, M. “Visualization Overview”. In: Proceedings of the Southeastern Symposium on System Theory, 2007, pp. 115–119.

[Gho10] Ghosh, D. “DSLs in Action”. Greenwich, CT, USA: Manning Publications Co., 2010, 376p.

[Gri16] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Ph.D. Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, 200p.

[Gup15] Gupta, A. “Big Data Analysis Using Computational Intelligence and Hadoop: A Study”. In: Proceedings of the International Conference on Computing for Sustainable Global Development, 2015, pp. 1397–1401.

[Gus11] Gustafson, J. L. “Little’s Law”. In: *Encyclopedia of Parallel Computing*, Padua, D. (Editor), Springer US, 2011, pp. 1038–1041.

[HFB05] Hertz, M.; Feng, Y.; Berger, E. D. “Garbage Collection Without Paging”. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005, pp. 143–153.

[Hil04] Hill, E. “First Course: Data Structures and Algorithms Using Java”. iUniverse, 2004, 107p.

[HWCP15] Hasan, M.; Wolfgang, J.; Chen, G.; Pfister, H. “Shadie: A Domain-Specific Language for Volume Visualization”. Source: http://miloshasan.net/Shadie/shadie.pdf, 2010 (accessed December 15, 2015).

[INT15] INTEL. “Intel Software Network (Official page of Intel)”. Source: http://software.intel.com/en-us/articles/intel-parallel-studio-home>, June 2015 (accessed December 02, 2015).

[KCS+16] Kindlmann, G.; Chiw, C.; Seltzer, N.; Samuels, L.; Reppy, J. “Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis”, *IEEE Transactions on Visualization and Computer Graphics*, vol. 22–1, January 2016, pp. 867–876.

[KEH+09] Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; Winwood, S. “SeL4: Formal Verification of an OS Kernel”. In: Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles, 2009, pp. 207–220.

[Knu76] Knuth, D. E. “Big Omicron and Big Omega and Big Theta”, *SIGACT News*, vol. 8, April 1976, pp. 18–24.

[Kri09] Krithivasan, K. “Introduction to Formal Languages, Automata Theory and Computation”. India: Pearson Education, 2009, 436p.

[Lea15] Leaflet. “Leaflet Documentation”. Source: http://leafletjs.com/reference.html, June 2015 (accessed May 2, 2015).

[Lei09] Leiserson, C. E. “The Cilk++ Concurrency Platform”. In: Proceedings of the Annual Design Automation Conference, 2009, pp. 522–527.

[LGMF15] Ledur, C.; Griebler, D.; Manssuor, I.; Fernandes, L. G. “Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets”. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, 2015, pp. 8.

[LLA14] Liu, J.; Liu, F.; Ansari, N. “Monitoring and Analyzing Big Traffic Data of a Large-Scale Cellular Network with Hadoop”, *IEEE Network*, vol. 28–4, July 2014, pp. 32–39.

[LRCS14] Li, K.; Reichenbach, C.; Csallner, C.; Smaragdakis, Y. “Residual Investigation: Predictive and Precise Bug Detection”, *ACM Transactions on Software Engineering and Methodology*, vol. 24–2, December 2014, pp. 7:1–7:32.

[LV15] Lyman, P.; Varian, H. “How Much Information?” Source: http://groups.ischool.berkeley.edu/archive/how-much-info-2003/, October 2004 (accessed October 16, 2015).

[MCB+11] Manyika, J.; Chui, M.; Brown, B.; Bughin, J.; Dobbs, R.; Roxburgh, C.; Byers, A. H. “Big Data: The Next Frontier for Innovation, Competition, and Productivity”, Technical Report, McKinsey Global Institute, 2011, 156p.

[MHS05] Mernik, M.; Heering, J.; Sloane, A. M. “When and How to Develop Domain-Specific Languages”, *ACM Computing Surveys (CSUR)*, vol. 37–4, December 2005, pp. 316–344.

[MR14] Manikandan, S. G.; Ravi, S. “Big Data Analysis Using Apache Hadoop”. In: Proceedings of the International Conference on IT Convergence and Security, 2014, pp. 1–4.

[MSM04] Mattson, T.; Sanders, B.; Massingill, B. “Patterns for Parallel Programming”. Boston, EUA: Addison-Wesley Professional, 2004, 384p.

[MTAB15] Meyerovich, L. A.; Torok, M. E.; Atkinson, E.; Bodík, R. “Superconductor: A Language for Big Data Visualization”. Source: https://engineering.purdue.edu/~milind/lashc13/meyerovich-superconductor.pdf, February 2013 (accessed December 17, 2015).

[NBF96] Nichols, B.; Buttlar, D.; Farrell, J. P. “Pthreads Programming”. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 1996, 286p.

[Ope15] OpenLayers. “OpenLayers Documentation”. Source: http://openlayers.org/en/v3.6.0/doc/tutorials/introduction.html, June 2015 (accessed August 1, 2015).

[Pac11] Pacheco, P. “An Introduction to Parallel Programming”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, 370p.

[RBGH14] Rautek, P.; Bruckner, S.; Groller, M.; Hadwiger, M. “ViSlang: A System for Interpreted Domain-Specific Languages for Scientific Visualization”, *IEEE Transactions on Visualization and Computer Graphics*, vol. 20–12, December 2014, pp. 2388–2396.

[Rei07] Reinders, J. “Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism”. Sebastopol, USA: O’Reilly and Associates, 2007, 336p.

[RGBMA06] Robles, G.; Gonzalez-Barahona, J. M.; Michlmayr, M.; Amor, J. J. “Mining Large Software Compilations over Time: Another Perspective of Software

Evolution". In: Proceedings of the International Workshop on Mining Software Repositories, 2006, pp. 3–9.

[RR13] Rauber, T.; Runger, G. "Parallel programming: For Multicore and Cluster Systems". Bayreuth, Germany: Springer Berlin Heidelberg, 2013, 516p.

[RS14] Ramya, A.; Sivasankar, E. "Distributed Pattern Matching and Document Analysis in Big Data Using Hadoop MapReduce Model". In: Proceedings of the International Conference on Parallel, Distributed and Grid Computing, 2014, pp. 312–317.

[Rus08] Rust, K. "Using Little's Law to Estimate Cycle Time and Cost". In: Proceedings of the 40th Conference on Winter Simulation, 2008, pp. 2223–2228.

[SET+09] Shinagawa, T.; Eiraku, H.; Tanimoto, K.; Omote, K.; Hasegawa, S.; Horie, T.; Hirano, M.; Kourai, K.; Oyama, Y.; Kawai, E.; Kono, K.; Chiba, S.; Shinjo, Y.; Kato, K. "BitVisor: A Thin Hypervisor for Enforcing I/O Device Security". In: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2009, pp. 121–130.

[STM10] Siefers, J.; Tan, G.; Morrisett, G. "Robusta: Taming the Native Beast of the JVM". In: Proceedings of the ACM Conference on Computer and Communications Security, 2010, pp. 201–211.

[Sve10] Svennerberg, G. "Beginning Google Maps API 3". Berkely, CA, USA: Apress, 2010, 328p.

[TSF+15] Thomee, B.; Shamma, D. A.; Friedland, G.; Elizalde, B.; Ni, K.; Poland, D.; Borth, D.; Li, L. "The New Data and New Challenges in Multimedia Research", *CoRR arXiv eprint*, vol. abs/1503.01817, 2015.

[VBD+13] Voelter, M.; Benz, S.; Dietrich, C.; Engelmann, B.; Helander, M.; Kats, L. C. L.; Visser, E.; Wachsmuth, G. "DSL Engineering - Designing, Implementing and Using Domain-Specific Languages". dslbook.org, 2013, 558p.

[vDKV00] van Deursen, A.; Klint, P.; Visser, J. "Domain-Specific Languages: An Annotated Bibliography", *ACM SIGPLAN Notices*, vol. 35–6, June 2000, pp. 26–36.

[VSJ+14] Vazou, N.; Seidel, E. L.; Jhala, R.; Vytiniotis, D.; Peyton-Jones, S. "Refinement Types for Haskell". In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, 2014, pp. 269–282.

[WGK10] Ward, M. O.; Grinstein, G.; Keim, D. "Interactive Data Visualization: Foundations, Techniques, and Applications". Massachusetts, USA: CRC Press, 2010, 513p.

[Whe16] Wheeler, D. A. “SLOCCount Documentation”. Source: http://www.dwheeler.com/sloccount/, October 2015 (accessed January 11, 2016).

[WWA15] Wang, L.; Wang, G.; Alexander, C. A. “Big Data and Visualization: Methods, Challenges and Technology Progress”, *Digital Technologies*, vol. 1–1, 2015, pp. 33–38.

[ZCL13] Zhang, J.; Chen, Y.; Li, T. “Opportunities of Innovation under Challenges of Big Data”. In: Proceedings of the International Conference on Fuzzy Systems and Knowledge Discovery, 2013, pp. 669–673.

[ZH13] Zhang, J.; Huang, M. L. “5Ws Model for Big Data Analysis and Visualization”. In: Proceedings of the International Conference on Computational Science and Engineering, 2013, pp. 1021–1028.

APPENDIX A – PERFORMANCE RESULTS (RAW)

Table A.1 – MM_DEV - performance results

Big Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	293.63	3.80	293.63	4.15	58.55	60.10
1	157.60	4.65	280.56	8.38	109.09	61.28
2	109.28	12.91	145.73	4.04	157.33	117.97
3	98.96	1.55	138.61	6.13	173.74	124.03
4	104.86	17.65	138.51	7.18	163.96	124.12
5	109.64	11.57	142.71	5.41	156.81	120.47
6	116.16	4.27	137.98	3.62	148.01	124.60
7	111.57	0.47	140.99	2.06	154.09	121.94
8	108.37	21.18	138.67	15.64	158.65	123.98
9	98.97	0.83	141.28	1.71	173.71	121.69
10	121.22	40.70	144.09	0.72	141.83	119.31
11	166.41	56.95	145.09	0.52	103.31	118.49
12	139.70	67.77	143.46	1.27	123.07	119.84
Medium Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	23.03	0.44	23.03	0.39	124.52	124.43
1	24.67	0.61	23.02	0.22	116.15	124.48
2	12.91	0.18	12.83	0.33	222.03	223.34
3	9.15	0.18	9.33	0.22	313.09	307.26
4	7.07	0.13	7.26	0.20	405.23	394.93
5	8.01	2.57	6.02	0.16	357.76	476.10
6	8.22	0.19	5.70	0.14	348.56	502.91
7	8.31	0.12	5.74	0.17	344.75	499.33
8	8.58	0.15	5.85	0.21	333.80	489.83
9	8.64	0.30	6.05	0.24	331.69	473.96
10	8.92	0.14	6.44	0.30	321.15	444.90
11	9.56	0.08	6.92	0.59	299.76	413.81
12	10.01	0.19	6.72	0.41	286.34	426.71
Small Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	10.99	0.05	10.99	0.13	119.01	114.60
1	4.13	0.01	4.11	0.18	115.60	116.13
2	2.59	0.01	2.56	0.14	184.49	186.75
3	1.98	0.04	1.93	0.03	241.48	247.66
4	1.66	0.12	1.72	0.04	287.19	277.32
5	1.54	0.03	1.36	0.04	310.20	349.94
6	1.54	0.03	1.36	0.07	309.53	350.43
7	1.52	0.03	1.36	0.03	313.34	352.30
8	1.49	0.09	1.35	0.04	320.79	353.33
9	1.35	0.00	1.38	0.05	353.45	346.09
10	1.36	0.05	1.38	0.08	350.84	346.85
11	1.36	0.00	1.35	0.05	352.03	354.65
12	1.51	0.03	1.35	0.04	316.66	354.77

Table A.2 – MM_ACID - performance results

Big Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	427.16	1.47	427.16	20.11	40.44	42.60
1	300.39	4.28	384.45	15.87	57.50	44.93
2	158.85	2.69	210.48	3.90	108.74	82.06
3	118.71	1.77	159.03	2.76	145.51	108.61
4	135.47	8.28	149.56	1.76	127.50	115.49
5	118.35	7.50	144.45	2.45	145.95	119.57
6	120.38	11.32	146.36	1.89	143.49	118.01
7	117.88	1.20	147.20	2.89	146.53	117.34
8	128.91	6.39	148.34	3.35	133.99	116.44
9	119.48	0.55	149.35	6.98	144.57	115.65
10	135.21	103.48	154.28	10.06	127.75	111.95
11	165.05	164.75	155.69	19.17	104.65	110.94
12	159.51	102.94	168.61	12.56	108.29	102.44
Medium Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	62.48	0.70	40.70	0.41	70.90	70.73
1	48.31	0.34	40.58	1.12	59.59	70.93
2	28.52	2.57	26.57	2.50	100.93	108.36
3	22.13	1.06	20.59	2.02	130.07	139.84
4	19.14	0.64	18.35	1.07	150.44	156.90
5	21.37	1.02	17.66	1.21	134.72	163.04
6	21.33	0.43	17.40	0.64	134.98	165.41
7	22.22	1.21	18.14	0.50	129.56	158.67
8	22.93	1.72	21.11	0.94	125.56	136.39
9	37.34	5.32	26.43	4.84	77.09	108.93
10	43.03	7.75	34.84	5.89	66.91	82.64
11	51.26	6.03	35.38	6.40	56.16	81.37
12	58.06	4.13	40.20	3.48	49.58	71.61
Small Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	10.99	0.40	7.46	0.39	66.78	64.33
1	7.94	0.07	6.99	0.02	60.45	68.61
2	5.83	0.33	5.16	0.49	82.32	93.07
3	4.24	0.38	4.21	0.45	113.06	114.01
4	4.01	0.09	3.87	0.29	119.58	123.84
5	3.48	0.05	3.70	0.19	137.71	129.68
6	3.45	0.07	3.48	0.23	139.16	137.83
7	3.56	0.03	3.55	0.20	134.96	135.15
8	3.46	0.11	3.54	0.33	138.61	135.51
9	3.29	0.04	3.41	0.21	145.90	140.74
10	3.31	0.05	3.58	0.16	145.02	134.10
11	3.28	0.06	3.45	0.15	146.50	138.96
12	3.65	0.06	3.43	0.23	131.27	139.80

Table A.3 – HM_AIR - performance results

<table>
<tr><th colspan="7">Big Dataset</th></tr>
<tr><th>Rep.</th><th>Avarage Exec. Time SPar (sec.)</th><th>Std.dev. SPar</th><th>Avarage Exec. Time TBB (sec.)</th><th>Std.dev. TBB</th><th>Throughput SPar (Mbytes/s)</th><th>Throughput TBB(Mbytes/s)</th></tr>
<tr><td>0</td><td>428.45</td><td>17.26</td><td>428.45</td><td>11.08</td><td>40.32</td><td>40.30</td></tr>
<tr><td>1</td><td>282.56</td><td>7.25</td><td>440.46</td><td>7.87</td><td>61.14</td><td>39.22</td></tr>
<tr><td>2</td><td>162.90</td><td>2.31</td><td>260.62</td><td>8.23</td><td>106.05</td><td>66.28</td></tr>
<tr><td>3</td><td>158.15</td><td>1.52</td><td>209.67</td><td>6.86</td><td>109.24</td><td>82.39</td></tr>
<tr><td>4</td><td>158.12</td><td>5.26</td><td>161.27</td><td>13.22</td><td>109.25</td><td>107.12</td></tr>
<tr><td>5</td><td>160.46</td><td>6.41</td><td>156.42</td><td>3.31</td><td>107.66</td><td>110.44</td></tr>
<tr><td>6</td><td>156.73</td><td>8.06</td><td>153.68</td><td>9.16</td><td>110.22</td><td>112.41</td></tr>
<tr><td>7</td><td>155.28</td><td>2.98</td><td>152.70</td><td>7.43</td><td>111.25</td><td>113.13</td></tr>
<tr><td>8</td><td>153.65</td><td>6.59</td><td>162.08</td><td>11.78</td><td>112.44</td><td>106.59</td></tr>
<tr><td>9</td><td>160.05</td><td>3.99</td><td>166.60</td><td>3.93</td><td>107.94</td><td>103.69</td></tr>
<tr><td>10</td><td>208.54</td><td>188.08</td><td>164.45</td><td>4.07</td><td>82.84</td><td>105.05</td></tr>
<tr><td>11</td><td>243.69</td><td>132.73</td><td>155.12</td><td>7.96</td><td>70.89</td><td>111.37</td></tr>
<tr><td>12</td><td>218.06</td><td>152.97</td><td>158.06</td><td>7.48</td><td>79.23</td><td>109.29</td></tr>
<tr><th colspan="7">Medium Dataset</th></tr>
<tr><th>Rep.</th><th>Avarage Exec. Time SPar (sec.)</th><th>Std.dev. SPar</th><th>Avarage Exec. Time TBB (sec.)</th><th>Std.dev. TBB</th><th>Throughput SPar (Mbytes/s)</th><th>Throughput TBB(Mbytes/s)</th></tr>
<tr><td>0</td><td>62.48</td><td>1.74</td><td>61.12</td><td>3.78</td><td>47.28</td><td>47.11</td></tr>
<tr><td>1</td><td>59.15</td><td>0.69</td><td>60.65</td><td>3.66</td><td>48.67</td><td>47.47</td></tr>
<tr><td>2</td><td>32.37</td><td>0.46</td><td>37.83</td><td>1.80</td><td>88.96</td><td>76.10</td></tr>
<tr><td>3</td><td>25.37</td><td>0.83</td><td>27.00</td><td>1.10</td><td>113.51</td><td>106.65</td></tr>
<tr><td>4</td><td>18.52</td><td>0.45</td><td>21.90</td><td>0.60</td><td>155.47</td><td>131.45</td></tr>
<tr><td>5</td><td>25.10</td><td>3.43</td><td>20.59</td><td>0.46</td><td>114.73</td><td>139.83</td></tr>
<tr><td>6</td><td>24.59</td><td>1.84</td><td>19.75</td><td>0.39</td><td>117.11</td><td>145.81</td></tr>
<tr><td>7</td><td>25.21</td><td>0.99</td><td>20.06</td><td>0.83</td><td>114.20</td><td>143.57</td></tr>
<tr><td>8</td><td>25.02</td><td>0.67</td><td>19.97</td><td>0.63</td><td>115.07</td><td>144.19</td></tr>
<tr><td>9</td><td>28.21</td><td>2.65</td><td>22.66</td><td>1.32</td><td>102.05</td><td>127.06</td></tr>
<tr><td>10</td><td>30.11</td><td>3.28</td><td>25.18</td><td>2.24</td><td>95.64</td><td>114.35</td></tr>
<tr><td>11</td><td>31.29</td><td>1.63</td><td>26.48</td><td>2.59</td><td>92.02</td><td>108.74</td></tr>
<tr><td>12</td><td>37.28</td><td>2.16</td><td>26.91</td><td>1.07</td><td>77.24</td><td>106.99</td></tr>
<tr><th colspan="7">Small Dataset</th></tr>
<tr><th>Rep.</th><th>Avarage Exec. Time SPar (sec.)</th><th>Std.dev. SPar</th><th>Avarage Exec. Time TBB (sec.)</th><th>Std.dev. TBB</th><th>Throughput SPar (Mbytes/s)</th><th>Throughput TBB(Mbytes/s)</th></tr>
<tr><td>0</td><td>10.99</td><td>0.21</td><td>10.93</td><td>0.96</td><td>50.06</td><td>43.90</td></tr>
<tr><td>1</td><td>10.14</td><td>0.29</td><td>9.58</td><td>0.33</td><td>47.34</td><td>50.09</td></tr>
<tr><td>2</td><td>6.37</td><td>0.28</td><td>6.65</td><td>0.32</td><td>75.38</td><td>72.16</td></tr>
<tr><td>3</td><td>5.13</td><td>0.06</td><td>4.97</td><td>0.26</td><td>93.62</td><td>96.62</td></tr>
<tr><td>4</td><td>4.38</td><td>0.13</td><td>4.59</td><td>0.08</td><td>109.64</td><td>104.44</td></tr>
<tr><td>5</td><td>3.92</td><td>0.22</td><td>3.64</td><td>0.11</td><td>122.53</td><td>131.97</td></tr>
<tr><td>6</td><td>3.97</td><td>0.07</td><td>3.55</td><td>0.17</td><td>120.81</td><td>135.33</td></tr>
<tr><td>7</td><td>3.95</td><td>0.05</td><td>3.53</td><td>0.16</td><td>121.63</td><td>135.89</td></tr>
<tr><td>8</td><td>3.84</td><td>0.15</td><td>3.52</td><td>0.21</td><td>124.89</td><td>136.44</td></tr>
<tr><td>9</td><td>3.64</td><td>0.04</td><td>3.63</td><td>0.19</td><td>131.70</td><td>132.36</td></tr>
<tr><td>10</td><td>3.63</td><td>0.05</td><td>3.50</td><td>0.16</td><td>132.10</td><td>137.01</td></tr>
<tr><td>11</td><td>3.62</td><td>0.05</td><td>3.65</td><td>0.21</td><td>132.56</td><td>131.60</td></tr>
<tr><td>12</td><td>4.09</td><td>0.06</td><td>3.66</td><td>0.15</td><td>117.37</td><td>131.10</td></tr>
</table>

Table A.4 – HM_AC - performance results

Big Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	358.85	1.71	358.85	16.69	48.13	50.67
1	238.42	4.35	325.27	7.85	72.45	53.10
2	130.90	5.55	179.20	3.81	131.96	96.39
3	118.41	6.85	145.02	3.14	145.88	119.10
4	132.47	8.37	144.71	3.82	130.39	119.36
5	120.45	11.97	141.07	3.27	143.40	122.44
6	118.55	2.99	143.58	2.65	145.70	120.30
7	117.24	1.47	144.63	0.42	147.33	119.42
8	125.85	8.39	144.00	0.34	137.25	119.94
9	118.71	1.22	143.66	0.21	145.50	120.23
10	134.84	118.68	143.62	4.32	128.10	120.26
11	176.93	96.78	151.95	13.98	97.62	113.67
12	238.72	221.01	156.42	12.38	72.36	110.42
Medium Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	62.48	7.98	62.48	7.98	90.76	89.89
1	37.86	0.18	31.61	0.29	76.04	91.06
2	23.73	1.78	21.81	1.69	121.30	131.97
3	18.58	1.34	16.67	1.16	154.96	172.71
4	16.89	0.56	16.60	0.85	170.42	173.45
5	20.68	1.28	16.04	0.72	139.21	179.48
6	20.33	0.67	15.85	0.83	141.60	181.67
7	21.00	0.65	17.92	1.11	137.12	160.67
8	21.54	0.67	19.68	1.52	133.66	146.25
9	31.42	4.62	24.14	2.32	91.62	119.28
10	41.39	7.82	32.30	4.78	69.56	89.12
11	53.39	6.68	36.16	5.49	53.92	79.61
12	56.48	7.35	37.09	8.48	50.97	77.61
Small Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	10.99	1.29	5.85	1.51	86.56	82.01
1	6.13	0.05	5.64	0.17	78.23	85.07
2	4.76	0.06	4.49	0.20	100.82	106.86
3	3.89	0.11	3.58	0.27	123.30	133.98
4	3.30	0.07	3.29	0.16	145.27	145.92
5	3.15	0.49	3.27	0.22	152.18	146.66
6	3.16	0.04	3.18	0.18	151.74	150.82
7	3.18	0.04	3.26	0.16	150.87	147.30
8	3.12	0.13	3.28	0.21	153.72	146.40
9	2.92	0.04	3.19	0.24	164.07	150.61
10	2.92	0.02	3.29	0.22	164.09	145.89
11	2.90	0.04	3.26	0.29	165.72	147.17
12	3.09	0.03	3.33	0.18	155.26	143.94

Table A.5 – CM_CP - performance results

Big Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	293.41	5.56	293.41	11.99	58.60	61.50
1	154.86	3.52	272.94	7.15	111.02	62.99
2	104.53	8.80	146.28	3.99	164.47	117.53
3	99.34	2.41	137.80	5.97	173.06	124.76
4	103.12	12.88	135.97	4.56	166.72	126.44
5	105.30	14.46	141.44	5.47	163.27	121.55
6	109.24	5.05	137.47	4.08	157.38	125.06
7	111.37	1.76	139.60	1.32	154.38	123.15
8	95.95	12.29	138.48	0.39	179.18	124.15
9	98.65	0.93	140.94	0.81	174.28	121.98
10	115.37	14.69	143.98	0.76	149.02	119.41
11	139.11	50.30	144.84	3.16	123.59	118.70
12	123.96	12.31	143.36	1.29	138.70	119.92
Medium Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	23.47	4.83	23.47	7.53	124.92	122.08
1	24.56	0.52	23.05	0.43	116.69	124.32
2	12.91	0.18	12.68	0.34	221.95	225.95
3	9.18	0.27	9.28	0.15	312.02	308.64
4	7.01	0.13	7.20	0.16	408.89	397.86
5	8.16	4.98	6.00	0.13	351.12	477.44
6	8.15	0.23	5.75	0.22	351.47	498.60
7	8.34	0.12	5.77	0.23	343.62	496.32
8	8.67	0.16	5.80	0.18	330.60	494.17
9	8.82	0.38	6.11	0.20	324.81	468.63
10	8.95	0.26	6.59	0.29	320.10	434.51
11	9.46	0.09	6.58	0.47	302.89	435.37
12	9.96	0.24	6.57	0.36	287.81	435.94
Small Dataset						
Rep.	**Avarage Exec. Time SPar (sec.)**	**Std.dev. SPar**	**Avarage Exec. Time TBB (sec.)**	**Std.dev. TBB**	**Throughput SPar (Mbytes/s)**	**Throughput TBB(Mbytes/s)**
0	4.14	1.21	4.22	1.37	115.46	113.05
1	4.13	0.05	4.07	0.16	115.69	117.31
2	2.56	0.15	2.57	0.10	186.74	186.16
3	1.95	0.03	1.87	0.08	245.17	255.35
4	1.69	0.11	1.70	0.09	283.29	280.82
5	1.54	0.98	1.33	0.06	309.88	359.33
6	1.53	0.03	1.35	0.05	311.51	354.78
7	1.53	0.02	1.38	0.04	312.22	346.62
8	1.50	0.08	1.34	0.05	319.18	356.71
9	1.35	0.02	1.35	0.06	354.94	353.95
10	1.34	0.02	1.32	0.06	356.13	360.80
11	1.34	0.01	1.29	0.06	355.54	369.35
12	1.44	0.04	1.34	0.08	330.91	357.27

Table A.6 – CM_AIR - performance results

Big Dataset

Rep.	Avarage Exec. Time SPar (sec.)	Std.dev. SPar	Avarage Exec. Time TBB (sec.)	Std.dev. TBB	Throughput SPar (Mbytes/s)	Throughput TBB(Mbytes/s)
0	424.86	18.10	424.86	10.87	40.66	40.15
1	282.24	6.42	444.06	6.59	61.21	38.90
2	163.47	17.23	264.11	9.33	105.68	65.41
3	158.43	4.37	208.18	4.01	109.04	82.98
4	155.59	7.00	168.64	11.33	111.03	102.44
5	160.89	6.10	154.14	3.35	107.37	112.08
6	157.62	8.23	154.03	6.45	109.60	112.15
7	154.95	3.55	150.84	8.93	111.49	114.52
8	152.50	5.24	161.63	9.88	113.28	106.88
9	158.96	1.27	164.28	2.88	108.68	105.15
10	183.73	107.19	160.90	2.79	94.03	107.36
11	163.07	154.61	151.22	10.25	35.52	114.24
12	222.32	107.47	156.74	7.60	77.71	110.21

Medium Dataset

Rep.	Avarage Exec. Time SPar (sec.)	Std.dev. SPar	Avarage Exec. Time TBB (sec.)	Std.dev. TBB	Throughput SPar (Mbytes/s)	Throughput TBB(Mbytes/s)
0	62.48	10.85	62.48	10.70	46.39	46.08
1	59.93	1.19	63.18	3.36	48.05	45.58
2	32.50	0.57	37.02	2.90	88.59	77.77
3	25.73	2.17	27.53	1.33	111.89	104.58
4	19.55	0.94	21.95	0.71	147.27	131.16
5	25.25	4.53	21.02	0.40	114.03	136.99
6	24.43	2.17	19.26	0.70	117.88	149.51
7	25.05	0.71	19.73	0.57	114.95	145.92
8	25.53	1.01	20.26	0.96	112.78	142.14
9	28.90	2.57	22.69	1.04	99.62	126.91
10	31.52	3.00	26.37	2.88	91.35	109.21
11	33.44	2.22	25.59	1.73	86.11	112.50
12	38.72	4.17	28.07	2.45	74.36	102.59

Small Dataset

Rep.	Avarage Exec. Time SPar (sec.)	Std.dev. SPar	Avarage Exec. Time TBB (sec.)	Std.dev. TBB	Throughput SPar (Mbytes/s)	Throughput TBB(Mbytes/s)
0	10.99	1.81	10.99	2.96	49.35	43.35
1	10.02	0.31	9.55	0.17	47.68	50.00
2	6.41	0.29	6.40	0.30	74.46	74.61
3	4.98	0.30	4.85	0.23	95.81	98.46
4	4.66	0.48	4.55	0.22	102.56	105.06
5	3.96	0.71	3.52	0.20	120.68	135.53
6	3.98	0.10	3.54	0.20	119.97	134.82
7	4.05	0.06	3.50	0.19	118.01	136.42
8	3.94	0.21	3.52	0.13	121.09	135.81
9	3.63	0.29	3.56	0.19	131.65	133.99
10	3.61	0.05	3.59	0.13	132.34	132.97
11	3.59	0.24	3.47	0.13	133.13	137.60
12	3.99	0.09	3.55	0.20	119.81	134.57

APPENDIX B – GMAVIS EBNF GRAMMAR

```
⟨program⟩               ::= ⟨vis-decl⟩ ⟨settings⟩ ⟨data⟩

⟨vis-decl⟩              ::= ’visualization’ ’:’ ⟨vis-type⟩ ’;’

⟨vis-type⟩              ::= ’markedmap’
                         |  ’clusteredmap’
                         |  ’heatmap’

⟨settings⟩              ::= ’settings’ ’{’ ⟨settings-decl⟩+ ’}’

⟨settings-decl⟩         ::= ⟨location-name-decl⟩
                         |  ⟨value-decl⟩
                         |  ⟨zoom-level-decl⟩
                         |  ⟨page-title-decl⟩
                         |  ⟨size-decl⟩
                         |  ⟨required-fields-decl⟩
                         |  ⟨marker-text-decl⟩

⟨required-fields-decl⟩  ::= ⟨latitude-decl⟩ ⟨longitude-decl⟩
                         |  ⟨longitude-decl⟩ ⟨latitude-decl⟩

⟨longitude-decl⟩        ::= ’longitude’ ’:’ ⟨field-spec⟩ ’;’

⟨latitude-decl⟩         ::= ’latitude’ ’:’ ⟨field-spec⟩ ’;’

⟨page-title-decl⟩       ::= ’page’ ’-’ ’title’ ’:’ ⟨string-literal⟩ ’;’

⟨marker-text-decl⟩      ::= ’marker’ ’-’ ’text’ ’:’  ⟨marker-text-spec⟩+  ’;’

⟨size-decl⟩             ::= ’size’ ’:’ ⟨size-type⟩ ’;’

⟨location-name-decl⟩    ::= ’location-name’ ’:’ ⟨field-spec⟩ ’;’

⟨value-decl⟩            ::= ’value’ ’:’ ⟨field-spec⟩ ’;’

⟨zoom-level-decl⟩       ::= ’zoom-level’ ’:’ ⟨integer-literal⟩ ’;’

⟨size-type⟩             ::= ’full’
                         |  ’small’
                         |  ’medium’

⟨data⟩                  ::= ’data’ ’{’ ⟨data-block-decl⟩+ ’}’
```

```
⟨filter-decl⟩              ::= ’filter’ ’:’ ⟨logical-expression⟩ ’;’

⟨classification-block⟩     ::= ’classification’ ’{’ ⟨classification-decl⟩+ ’}’

⟨file-decl⟩                ::= ’file’ ’:’ ⟨string-literal⟩ ’;’
                            |  ’h-file’ ’:’ ⟨string-literal⟩ ’;’

⟨structure-block⟩          ::= ’structure’ ’{’ ⟨structure-decl⟩+ ’}’

⟨data-block-decl⟩          ::= ⟨structure-block⟩
                            |  ⟨classification-block⟩
                            |  ⟨filter-decl⟩
                            |  ⟨file-decl⟩+

⟨integer-literal⟩          ::= ’integer literal’
                            |  ’-’ ’integer literal’

⟨field-spec⟩               ::= ’field’ ⟨integer-literal⟩

⟨expression⟩               ::= ⟨field-spec⟩ ’is’ ’equal’ ’to’ ⟨data-value⟩
                            |  ⟨field-spec⟩ ’is’ ’different’ ’than’ ⟨data-value⟩
                            |  ⟨field-spec⟩ ’is’ ’greater’ ’than’ ⟨data-value⟩
                            |  ⟨field-spec⟩ ’is’ ’less’ ’than’ ⟨data-value⟩
                            |  ⟨field-spec⟩ ’is’ ’between’ ⟨data-value⟩ ’and’ ⟨data-value⟩
                            |  ⟨field-spec⟩ ’contains’ ⟨data-value⟩

⟨logical-expression⟩       ::= expression
                            |  ’class’ ’(’ string-literal ’)’ ’:’ ⟨logical-expression⟩ ’;’
                            |  ’(’ ⟨logical-expression⟩ ’)’
                            |  ⟨logical-expression⟩ ⟨connection-operator⟩ ⟨logical-expression⟩

⟨classification-decl⟩      ::= ⟨class-decl⟩

⟨string-literal⟩           ::= ’string literal’

⟨format-decl⟩              ::= ’format’ ’:’ ⟨format-type⟩ ’;’

⟨structure-decl⟩           ::= ⟨end-register-decl⟩
                            |  ⟨date-format-decl⟩
                            |  ⟨delimiter-decl⟩
                            |  ⟨format-decl⟩

⟨marker-text-spec⟩         ::= ⟨field-spec⟩
                            |  ⟨string-literal⟩
```

```
⟨connection-operator⟩  ::=  ’and’
                        |   ’or’

⟨date-format-decl⟩     ::=  ’date’ ’-’ ’format’ ’:’ ⟨string-literal⟩ ’;’

⟨char-literal⟩         ::=  ’character literal’

⟨float-literal⟩        ::=  ’float literal’
                        |   ’-’ ’float literal’

⟨data-value⟩           ::=  ⟨date-value⟩
                        |   ⟨integer-literal⟩
                        |   ⟨string-literal⟩
                        |   ⟨float-literal⟩
                        |   ⟨char-literal⟩

⟨class-decl⟩           ::=  ’class’ ’:’ ⟨logical-expression⟩ ’;’
                        |   ’class’ ’(’ ⟨string-literal⟩ ’)’ ’:’ ⟨logical-expression⟩ ’;’

⟨format-type⟩          ::=  ’csv’

⟨delimiter-type⟩       ::=  ’tab’
                        |   ’newline’
                        |   ⟨string-literal⟩

⟨date-value⟩           ::=  ’date’ ⟨string-literal⟩

⟨delimiter-decl⟩       ::=  ’delimiter’ ’:’ ⟨delimiter-type⟩ ’;’

⟨end-register-type⟩    ::=  ’tab’
                        |   ’newline’
                        |   ⟨string-literal⟩

⟨end-register-decl⟩    ::=  ’end’ ’-’ ’register’ ’:’ ⟨end-register-type⟩ ’;’
```