Pontifícia Universidade Católica
do Rio de Janeiro

Rodrigo da Silva Ferreira

InterIMAGE Cloud Platform: The Architecture of a Distributed Platform for Automatic, Object-Based Image Interpretation

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-Graduação Engenharia Elétrica of the Departamento de Engenharia Elétrica, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor em Engenharia Elétrica.

Advisor: Prof. Raul Queiroz Feitosa
Co-Advisor: Profa. Cristiana Bentes

Rio de Janeiro
April 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Rodrigo da Silva Ferreira

InterIMAGE Cloud Platform: The Architecture of a Distributed Platform for Automatic, Object-Based Image Interpretation

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-Graduação em Engenharia Elétrica of the Departamento de Engenharia Elétrica do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requeriments for the degree of Doutor

Prof. Raul Queiroz Feitosa
Advisor
Departamento de Engenharia Elétrica – PUC-Rio

Profa. Cristiana Bentes
Co-Advisor
UERG

Profa. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Prof. Gilberto Ribeiro de Queiroz
INPE

Prof. Gilson Alexandre Ostwald Pedro da Costa
Departamento de Engenharia Elétrica – PUC-Rio

Prof. Thales Sehn Körting
INPE

Prof. José Eugenio Leal
Coordinator of the Centro Técnico
Científico da PUC-Rio

Rio de Janeiro, April 30th, 2015

Rodrigo da Silva Ferreira

The author graduated in Electrical Engineering at the Universidade do Estado do Rio de Janeiro (UERJ) in 2008, with specialization in Computer and Systems. Obtained the degree of Mestre in Electrical Engineering with emphasis on Signal Processing and Control, at the Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) in 2010. Since then has worked in the field of remote sensing image analysis.

Bibliographic data

Ferreira, Rodrigo da Silva

InterIMAGE cloud platform: the architecture of a distributed platform for automatic, object-based image interpretation / Rodrigo da Silva Ferreira; advisor: Raul Queiroz Feitosa ; co-advisor: Cristiana Bentes. – 2015.

159 f. : il. (color) ; 30 cm

Tese (doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Elétrica, 2015.

Inclui bibliografia

1. Engenharia Elétrica – Teses. 2. Sensoriamento remoto. 3. Análise de imagens baseada em objetos geográficos (GEOBIA). 4. Processamento distribuído. 5. Computação em nuvem. I. Feitosa, Raul Queiroz. II. Bentes, Cristiana. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Engenharia Elétrica. IV. Título.

CDD: 621.3

Para Mydiã e minha família.

Acknowledgements

Foremost, I am greatly thankful to my advisors, Prof. Raul Queiroz Feitosa and Prof. Cristiana Bentes, for all their support, suggestions and help during my PhD research.

I also thank my colleagues Gilson Costa, Dário Borges, Patrick Happ and Victor Quirita for their help in the discussions and implementation of ICP.

I would like also to show my gratitude to Prof. Paolo Gamba, who warmly welcomed me at the University of Pavia and made my stay there as rewarding as possible. I extend these thanks to the colleagues and friends that I made there; you made this experience much more pleasant.

I thank my colleagues from LVC for their companionship, friendship and help along the way.

I also gratefully acknowledge the financial support of CNPq and CAPES, without which this research would not be possible.

I thank my parents, Antônio and Silvana, for their love, guidance and help in all moments; and my brothers, Rafael and Rômulo, for their friendship and love.

I also thank my parents-in-law, Edmundo and Márcia, for all their support and love.

Lastly, I would like to give special thanks to my beloved wife Mydiã who has always been there for me; and to God, that is always by my side.

Abstract

Ferreira, Rodrigo da Silva; Feitosa, Raul Queiroz (Advisor); Bentes, Cristiana (Co-advisor). **InterIMAGE Cloud Platform: The Architecture of a Distributed Platform for Automatic, Object-Based Image Interpretation.** Rio de Janeiro, 2015, 159p. PhD Thesis – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

The general objective of this thesis was the development of a distributed computational architecture for the automatic, object-based interpretation of large volumes of remote sensing image data, focusing on data and processing distribution in a cloud computing environment. Two specific objectives were pursued: (i) the development of a novel distributed architecture for image analysis that is able to deal with vectors and rasters at the same time; and (ii) the design and implementation of an open-source, distributed platform for the interpretation of very large volumes of remote sensing data. In order to validate the new architecture, experiments were carried out using two classification models – land cover and land use – on a QuickBird image of an area of the São Paulo municipality. The classification models, proposed by Novack (Novack09), were recreated using the knowledge representation structures available in the new platform. In the executed experiments, the platform was able to process the whole land cover classification model on a 32,000x32,000-pixel image (~3.81 GB), with approximately 8 million image objects (~23.2 GB), in just one hour, using 32 machines in a commercial cloud computing service. Equally interesting results were obtained for the land use classification model. Another possibility of parallelism provided by the platform's knowledge representation structures was also evaluated.

Keywords

Remote Sensing; Geographic Object-Based Image Analysis (GEOBIA); Distributed Processing; Cloud Computing.

Resumo

Ferreira, Rodrigo da Silva; Feitosa, Raul Queiroz; Bentes, Cristiana. **Plataforma em Nuvem InterIMAGE: A Arquitetura de uma Plataforma Distribuída para a Interpretação Automática de Imagens Baseada em Objetos.** Rio de Janeiro, 2015, 159p. Tese de Doutorado – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

O objetivo genérico desta tese foi o desenvolvimento de uma arquitetura computacional distribuída para a interpretação automática, baseada em objetos, de grandes volumes de dados de imagem de sensoriamento remoto, com foco na distribuição de dados e processamento em um ambiente de computação em nuvem. Dois objetivos específicos foram perseguidos: (i) o desenvolvimento de uma nova arquitetura distribuída para análise de imagens que é capaz de lidar com vetores e imagens ao mesmo tempo; e (ii) a modelagem e implementação de uma plataforma distribuída para a interpretação de grandes volumes de dados de sensoriamento remoto. Para validar a nova arquitetura, foram realizados experimentos com dois modelos de classificação – um de cobertura da terra e outro de uso do solo – sobre uma imagem QuickBird de uma área do município de São Paulo. Os modelos de classificação, propostos por Novack (Novack09), foram recriados usando as estruturas de representação do conhecimento da nova plataforma. Nos experimentos executados, a plataforma foi capaz de processar todo o modelo de classificação de cobertura da terra para uma imagem de 32.000x32.000 pixels (~3,81 GB), com aproximadamente 8 milhões de objetos de imagem (~23,2 GB), em apenas 1 hora, utilizando 32 máquinas em um serviço de nuvem comercial. Resultados igualmente interessantes foram obtidos para o modelo de classificação de uso do solo. Outra possibilidade de paralelismo oferecida pelas estruturas de representação de conhecimento da plataforma também foi avaliada.

Palavras-chave

Sensoriamento Remoto; Análise de Imagens Baseada em Objetos Geográficos (GEOBIA); Processamento Distribuído; Computação em Nuvem.

Table of contents

List of figures

List of tables

1
Introduction

In the last decades, the availability of remote sensing image data has grown enormously. This is mainly due to the advent of new satellites and aircrafts, and the consequent drop in the costs of remotely sensed images, many of them made freely available. But not only the number of images has grown; the spatial, spectral and temporal resolutions have increased and created a new demand for scalable solutions to image interpretation problems (Vatsavai13).

NASA EOSDIS (Earth Observing System Data and Information System) project (EOSDIS15), for example, comprised in September 2013 about 10 PB of data, with an average growth of 8.5 TB per day. Besides NASA, space agencies like ESA (European Space Agency), JAXA (Japan Aerospace Exploration Agency), among others, produce a similar amount of remote sensing image data daily.

Private companies show the same scenario. Companies like Digital Globe, Planet Labs and Skybox Imaging (acquired by Google in 2014) are about to sum up one hundred satellites orbiting the globe. By 2018, Skybox Imaging expects to have a constellation of 24 satellites imaging the globe three or four times a day (IEEESpectrum13). The company also provides 90-second, high-definition videos at 30 frames per second of any spot on Earth. It is a huge market that was recently leveraged by the announcement of the alleviation of image resolution restrictions by the U.S. government (MarketWired14). With the new policy, Digital Globe, for example, will be able to sell WorldView-3 images at up to 0.25m of spatial resolution in the first trimester of 2015, expanding the company market opportunities in additional $400 million (Reuters14).

All these changes encourage the improvement of current solutions and the development of new solutions to problems such as agriculture health monitoring, natural disaster response, mining operations, carbon and maritime monitoring. However, to completely unveil these possibilities new methods and tools are demanded. These new methods and tools must be able to extract information from such high-resolution imagery, and analyze and interpret such amount of remote sensing data.

On the one hand, recent studies (Blaschke01; Blaschke14) argue that higher image resolutions entail a paradigm shift in image interpretation. Instead of pixel-based approaches, very-high-resolution images led scientists to build a new methodology, more suitable to the new characteristics of these images. In this approach, called GEOBIA (Geographic Object-Based Image Analysis), image analysis is based on image objects. It allows image analysts to use other information present in image objects like texture, shape and context that may likely improve the classification accuracy.

On the other hand, new tools must be able to handle huge volumes of data seamlessly. The characteristics of these data indicate that their storage and processing should be done on highly parallel systems, such as shared-nothing clusters (Olston08).

There are some software suites that follow the GEOBIA approach and offer a distributed solution for very large image data. One of them is a very successful commercial software called eCognition (eCognition15). This system was the first commercial software package to provide object-based techniques for image analysis and leveraged the new approach as it made it easier for researches (specially from environmental and biomedical sciences) to embed their interpretation models into the system.

Although eCognition is the most adopted software for this purpose in the world, there are some disadvantages. The first is the cost. A single license costs a few thousands of dollars. While this may not be a problem for some universities and researchers around the world, it hinders its use by scientists with fewer resources. The second is even more important: eCognition is a proprietary software. It is not possible to know how its algorithms were implemented. The third one is about extensibility. It is not possible to extend the system, so users have to wait for specific features to be implemented by the company. The last one is related to scalability. Its distributed solution is oriented towards a local cluster that is often expensive and hard to maintain.

In the last years, cloud computing solutions have become an appealing alternative to solve this kind of problem, mainly because of its low cost and scalability potential. As image datasets grow, a cloud system can be resized accordingly, still delivering acceptable costs and processing time.

However, programming for distributed environments such as cloud systems can be tiresome. Recent technologies, specifically the MapReduce model (Dean04) and its open-source version Hadoop MapReduce, made it a much easier task. By abstracting the details of parallelization, fault-tolerance, data distribution and load balancing, MapReduce allows users to focus on data processing.

Among all the freely available tools that follow the GEOBIA approach, InterIMAGE (Costa08) is the only system that provides most of the functionalities present in eCognition. It is an open-source platform for object-based image analysis that takes the GEOBIA ideals to a broader audience. Unlike eCognition, the system is open-source and extensible, allowing the inclusion of external operators that perform specific tasks like image segmentation, feature extraction and classification. Although InterIMAGE is a fully functional system, it has limitations regarding the size of the images and the number of image objects it can process.

Although eCognition and InterIMAGE provide a whole set of object-based image analysis tools, their support to very large datasets is quite limited. While InterIMAGE provides only some multi-core functionalities, eCognition's distributed solution lacks the scalability that only cloud systems can provide.

Many works have tackled the problems of spatial operations (Nishmura11; Lu12; Zhong12; Aji13; Eldawy13) and image processing (Liu12; Wang12; Lin13; Liu13) in cloud environments, but none has investigated object-based image analysis in such distributed systems. This is mainly due to the inherent complexity of dealing with raster and vector data at the same time. This work fills this gap by proposing a novel architecture that performs distributed object-based image analysis, by considering raster and vector data at the same time in the interpretation process. In order to do that, on-demand image handling and distributed MapReduce-based strategies are proposed.

Another aspect to object-based image analysis is related to knowledge representation. eCognition, InterIMAGE and their predecessors (Kummert98; Liedtke99; Bückner01) provide a semantic network to represent expert knowledge. Besides that, the execution of image interpretation operations is highly constrained by the hierarchical structure of the semantic network. Although the work proposed in this thesis also provides a semantic network for knowledge representation, its interpretation strategy is controlled by a graph structure that lends much more flexibility to the expert for representing his knowledge in the system. This structure allows image operators to be combined in different ways and provides two levels of parallelism in cloud environments.

Like InterIMAGE, this novel architecture also allows users to extend the system by adding new operators. Operators are defined in a high-level language that allows non-technical users to include new operators and combine existing operators to create new ones.

Ultimately, this thesis proposes a distributed platform for object-based image analysis that relies on the MapReduce model for parallel processing. The goal of the proposed platform *is to provide a costless platform that is able to process the enormous amount of image data available today following the GEOBIA approach*. The proposed architecture is inspired by the original InterIMAGE system and inherits some of its main features. However, it is designed to execute in a cloud computing environment where data and processing distribution allows it to handle very large datasets. The platform is named *InterIMAGE Cloud Platform (ICP).*

1.1. Objectives

The general objective of this research project is the proposal of a distributed computational architecture to support the object-based analysis of very large remote sensing data that provides robustness and performance through data partitioning and processing distribution in a cloud computing environment with maximum parallelism exploitation.

The present work concentrates on two main specific objectives. The first is the development of a novel distributed architecture that provides: data partitioning and distributed processing for image interpretation problems based on the MapReduce model; intuitive expert knowledge modelling; extensibility and flexibility. The second is the design and implementation of a distributed platform for object-based interpretation of very large remote sensing data.

To validate the proposed architecture, experiments were conducted aiming at the performance assessment of two interpretation models (land cover and land use) using a QuickBird image of an area of the São Paulo municipality.

1.2. Thesis contributions

This thesis presents a novel distributed architecture for object-based image analysis, which enables the analysis and interpretation of very large remote sensing data. The main contributions are listed below:

(1) The specification of a distributed image interpretation architecture that:

- is performant and robust to very large datasets by partitioning input data and distributing processing across potentially hundreds or thousands of machines.
- considers raster and vector data at the same time in the interpretation process.
- is extensible, allowing the inclusion of new operators.
- allows the recursive definition of operators which speeds up the construction of new interpretation models.
- provides a graph structure that allows the construction of flexible interpretation models and enables two levels of parallelism.

(2) Distributed strategies for:

- spectral feature computation.
- spatial conflict resolution.
- topological feature computation.
- recursive computation.
- hierarchical feature computation.

(3) The implementation of a platform that is able to process large volumes of data in a reasonable time and provides intuitive expert knowledge modelling.

1.3.
Thesis organization

The next chapter gives a general survey on the topics addressed in this thesis. The fundamentals of geographic object-based image analysis (GEOBIA), cloud computing, MapReduce, Hadoop and Pig are presented.

Chapter 3 presents a literature review and describes the state of the art on distributed systems for image and geographic data processing.

Chapter 4 presents the proposed architecture. The architecture components and strategies are discussed in detail.

Chapter 5 presents the software prototype and discusses in detail the experiments and the results obtained.

Chapter 6 presents the final conclusions along with some directions for future researches.

2
Theoretical foundations

This chapter presents the theoretical foundations for understanding the architecture proposed in chapter 4. The fundamentals of geographic object-based image analysis (GEOBIA), cloud computing, MapReduce, Hadoop and Pig are presented.

2.1.
Geographic Object-Based Image Analysis (GEOBIA)

Since the early 1970's, most of the image processing methods have been based on the classification of individual pixels. This approach has been the dominant paradigm in remote sensing for many years and, in fact, there are a fair number of well-stablished techniques to classify images by pixel (Strahler86; Weng09).

This approach is reasonable as long as the objects of interest are smaller or similar in size to the image's spatial resolution (L-resolution) (Hay01; Blaschke04). Once the target objects get larger than the image's spatial resolution (H-resolution), another approach has proved more adequate (Fisher97; Blaschke01; Burnett03). Instead of focusing on the statistical analysis of single pixels, this new approach is concerned with the spatial patterns within the target objects and the derived features such as texture, shape and context that may likely improve the classification accuracy. This new approach came to be known as object-based image analysis (OBIA).

One of the reasons that made this approach come into existence was the advent, in the last decades, of new sensors with higher spatial resolutions. High resolution images increased the *within-class* spectral variability and uncovered the limitations of the pixel-based approach (Hay96; Wang09). Another factor that leveraged the object-based approach was the release of eCognition (eCognition15), in 2000, the first commercial software conceived for the delineation and analysis of *image objects* from remote sensing imagery.

Hay and Castilla (Hay08) argue that although OBIA is also used in other fields like biology, medicine and astronomy, when it comes to remote sensing, there are earth surface related concepts that should be reflected in the name of the discipline. In this sense, they proposed the name “Geographic Object-Based Image Analysis” (GEOBIA). They define GEOBIA as “a sub-discipline of Geographic Information Science (GIScience) devoted to developing automated methods to partition remote sensing imagery into meaningful image objects, and assessing their characteristics through spatial, spectral and temporal scales, so as to generate new geographic information in GIS-ready format”.

In this sense, GEOBIA represents a bridge between the raster domain of remote sensing and the vector domain of GIS while the generation of polygons (classified image objects) is the link between these two worlds. Thus, at its fundamental level GEOBIA requires image segmentation.

Image objects are discrete regions of a digital image that are internally coherent and different from its surroundings (Castilla08; Haralick85). This is usually the outcome of a segmentation algorithm and is not uncommon to equate image segments to image objects. However, the authors argue that, as the image segmentation procedure may produce under- and over-segmented results according to a human observer, image objects could be redefined as the *segments derived from a good segmentation algorithm*.

The authors also define *meaningful image objects* as the image objects that represent *geographic objects*, which are bounded geographic regions that can be identified for a period of time as the referent of a geographic term. Examples of geographic terms would be a ‘glacier’, a ‘mineral extraction site’ or anything that can be represented in a map.

In a recent paper (Blaschke14), Blaschke et al. presented this change from a pixel-based to an object-based approach as a paradigm shift. This claim is based on the growing interest on the field, as the number of papers, conferences, journals and books dedicated to the topic has increased significantly in the last years (Blaschke14). The authors also argue that besides shape, texture and context information, there are more advantages in using GEOBIA. Among these advantages are the possibility to adapt many of Object-Oriented (OO) concepts and methods to GEOBIA; to work with multi-scales and object hierarchies; and to include ontologies and semantics into the interpretation process.

The authors conclude that GEOBIA is not just a collection of segmentation, analysis and classification methods. In their opinion, it is an evolving paradigm with specific tools, software, methods, rules and language; and that it is increasingly being used for studies that conceptualize and formalize knowledge that represents location-based reality.

The next section presents an open-source system that follows the GEOBIA paradigm. The architecture proposed in this work is based on this system.

2.1.1. InterIMAGE

InterIMAGE is an open-source platform for automatic, knowledge-based image interpretation developed in collaboration between the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and the Brazilian National Space Research Institute (INPE). The system is based on GeoAIDA (Bückner01), a system developed in Germany from which it inherits its basic design, knowledge structures and control mechanisms (Costa08).

The interpretation strategy in InterIMAGE is based on a *semantic network* that represents hierarchically the semantic concepts (classes) expected to be found in an image. The nodes represent real world concepts and the arcs represent the relations between the nodes. These relations are strictly hierarchical in which each node has exactly one parent node. To each operator it is possible to associate a top-down and a bottom-up operator.

The interpretation process takes place in two steps: a top-down (TD) and a bottom-up (BU) step. The TD step starts from the uppermost nodes and goes down until it reaches the leaf-nodes. It represents a model-driven processing where a net of object hypotheses is created in conformity with the semantic network. These hypotheses correspond to geographic regions in the image. In the TD step, when the interpretation control reaches a semantic node, it fires the associated *top-down or holistic operator*. Holistic operators are executable programs specialized on the detection of specific semantic concepts. In order to do that, these operators use image processing operations like segmentation, feature extraction and classification. Nodes that do not have a holistic operator associated to them perform a structural classification based on their sub-nodes.

The geographic regions detected by a holistic operator associated to a node are passed as masks to its child nodes, which will then execute their own holistic operators. The system allows the definition of different segmentation parameters and classification rules for each semantic concept. This characteristic enables the system to perform powerful classifications but creates the need for a spatial resolution operation in the BU step, as the same geographic region can be associated to more than one class.

When the leaf-nodes are reached, the BU step is executed going from the leaf-nodes to the top. In this step, the system confirms or discards hypotheses generated in the TD step, resolves eventual spatial conflicts between the hypotheses and updates the shapes of the image objects if necessary. The resolution of spatial conflicts can be done in the semantic network level, by just assigning different weights for each semantic concept. This way, the image object that belongs to the class with the higher weight wins. Another option is to use the so called *decision rules*. By using this mechanism, the user can define the weights of each image object directly or through the use of fuzzy sets. The system provides an intuitive graphical interface for the creation of such rules.

Figure 1 depicts InterIMAGE’s interpretation process. The user must provide the knowledge model (semantic network) and input data in order to execute an interpretation. For the semantic node, the user must define the holistic operators and BU rules associated to each node. The input data can be any type of vector and raster data.

The box in the middle shows the interpretation strategy. An instance net as output from the TD step and its transformation in instance net during the BU step. The output is a multi-scale thematic map and a symbolic description of the scene.

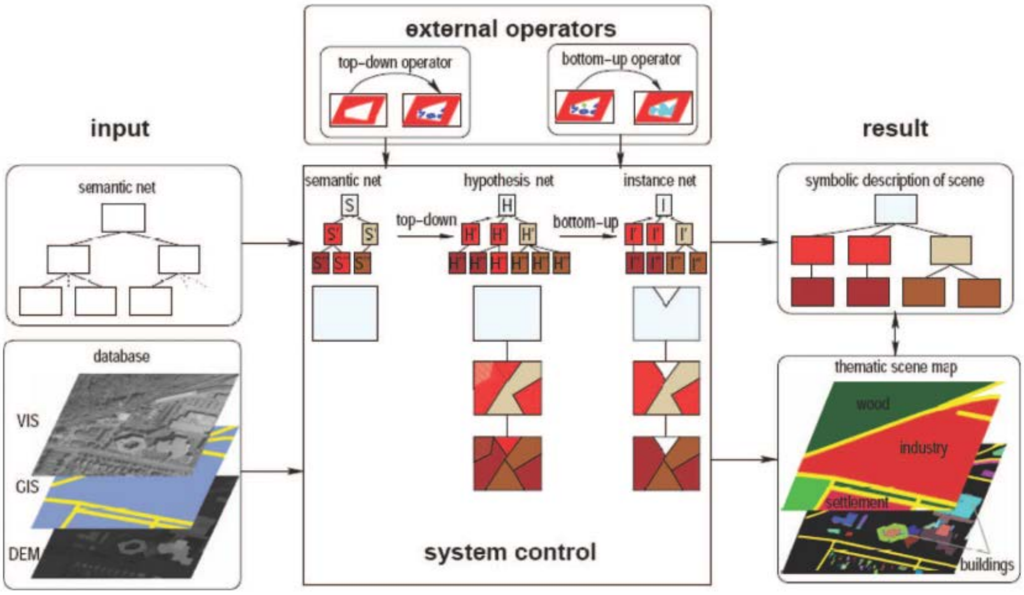


Figure 1: interpretation process in InterIMAGE.

Source: Adapted from Pahl (2008).

Although InterIMAGE is a fully functional system, it has limitations regarding the size of the images and the number of image objects it can process. Of course, there is no direct relation between the image size and the number of objects; it depends on the segmentation algorithm used and its input parameters. However, recent studies (Novack09) have shown that in some cases even images as small as 2000x2000 pixels could not be processed.

Recently, the segmentation program was improved in order to segment larger images. This solved only part of the problem, as these larger images tend to generate a higher number of image objects that exceeds the limit of objects the system’s core can process (this problem can be solved up to a certain limit by increasing the available RAM memory).

This was one of the main motivations for the distributed architecture proposed in this work that works on local clusters but is mainly oriented towards cloud environments. The next sections introduce cloud computing and some technologies that compose the architecture discussed in chapter 4.

2.2. Cloud computing

Cloud computing refers to applications and services that run on a distributed network using virtualized resources and accessed by common Internet protocols and networking standards (Sosinsky11). The term cloud computing brings with it two essential concepts: *virtualization* and *abstraction*.

Virtualization is achieved by pooling and sharing resources. A centralized infrastructure provides processing and storage functionalities on demand. Costs are assessed on a per-use basis and resources can scale easily.

Abstraction refers to the ability to abstract the details of system implementation from users and developers. In cloud computing, physical systems are not specified, data location is unknown and system administration is outsourced.

Cloud computing is usually divided in two sets of models: *deployment* and *service* models. The first one is related to the purpose of the cloud and its location. In this context, a cloud can be *public*, *private*, *hybrid* or a *community* cloud. While in a public cloud the infrastructure is available for public use, in a private cloud it is available for the exclusive use of a company. By contrast, a hybrid cloud combines multiple clouds (private or public). A community cloud is one built to serve a common purpose. It may be used by various organizations that share the same values, policies and so on.

Regarding service models, there are several models described in the literature, being these three the most accepted: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS).

In the IaaS model, providers offer computers (physical or virtual) and other resources. These resources are pooled on demand from a data center. The user is responsible for the maintenance of operating systems and applications. One example of this model is Amazon Elastic Compute Cloud (EC2).

In the PaaS model, besides hardware infrastructure, providers deliver a computing platform including operating system, programming language execution environment, databases and web servers. Application developers use the system to develop and run their software solutions without having to manage the underlying hardware and software layers. Examples of this model are Google AppEngine, Windows Azure Platform and Amazon Elastic MapReduce (EMR).

Finally, in the SaaS model, providers offer applications along with the infrastructure and platform they run on. This model is also known as “on-demand software” and is usually priced on a per-use basis or under a subscription fee. One example of this model is GoogleApps.

According to Mell and Grance (Mell11), there are some characteristics that cloud computing systems must offer:

- **On-demand self-service**: a client must be able to provision resources without having to contact vendor personnel.
- **Broad network access**: as it relies heavily on network communication, a cloud system must provide fast internet access.
- **Resource pooling**: a cloud service provider dynamically allocates physical and virtual resources as needed.
- **Rapid elasticity**: a cloud computing system must be able to quickly scale up and out. This scaling can be manual or automatic.
- **Measured service**: to charge a client on a per-use basis, a cloud service provider must measure every resource like the amount of storage, network I/O and processing power used.

The models and the essential characteristics of cloud computing are shown in Figure 2.

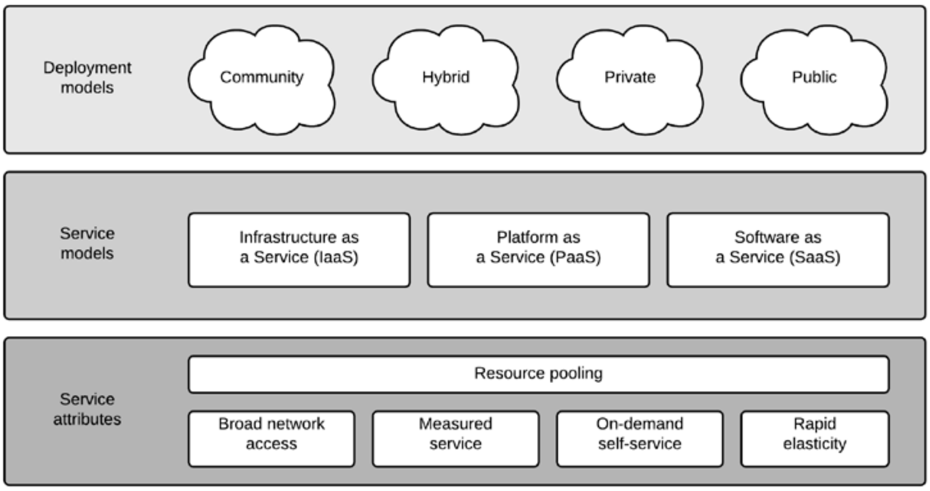


Figure 2: cloud computing definitions.

Source: Adapted from Sosinsky (2011).

These characteristics bring together a range of advantages in using cloud computing systems. The first one is related to cost. Cloud computing systems offer lower costs by operating at a higher efficiency. Besides that, a client company can reduce its costs by outsourcing IT management and using only the resources it needs at a given time.

By offering load balancing and failover mechanisms cloud computing systems are also highly reliable, often much more reliable than a domestic solution. These systems are also easier to use. Companies have to care much less about hardware requirements and software licenses. Its centralized structure makes it easier to upgrade softwares and always provide users with the latest versions.

There are also some disadvantages. While for small companies cloud systems may be the best solution, larger organizations can afford the IT staff necessary to develop custom software solutions specially designed for their needs. Such specialized solutions are likely going to outperform a generic system.

Another disadvantage is related to network latency and communication overhead. Applications that depend heavily on data transfer or that are performance sensitive can suffer significantly. For these applications cloud computing may not be the best model.

Finally, the major concerns about cloud computing are privacy and security. It is quite difficult to assure data secrecy when sending data over a network or storing it in a cloud service beyond your control. It is not possible to predict what cloud providers will do in face of government actions and, besides that, each country has its own policies and legislations. It is the client's responsibility to comply with the applicable jurisdictions.

The next section presents a platform that became ubiquitous when it comes to distributed storage and processing in the cloud: Hadoop.

2.3. Apache Hadoop

Hadoop is an infrastructure for large-scale distributed batch processing. It was designed to process large volumes of data by efficiently distributing work across hundreds or thousands of commodity computers connected to a network (HadoopTutorial15).

Although Hadoop is best known for MapReduce and its distributed file system (HDFS), the term is also used for a family of related projects such as Pig (to be seen later in this chapter), Hive (Hive15), HBase (HBase15) and ZooKeeper (ZooKeeper15), to name a few.

Hadoop was created at Yahoo! and has its origins in Apache Nutch, a part of the Apache Lucene project. In 2006, it became an independent subproject called Hadoop and in 2008 it was made a top-level project at Apache (Hadoop15).

The next sections present Hadoop's two main components: MapReduce and HDFS.

2.3.1. Hadoop Distributed File System (HDFS)

HDFS is the storage component of Hadoop. It is a distributed file system optimized for high throughput and works best when reading and writing large files (gigabytes and larger) (Holmes12). The system is based on Google File System (GFS) (Ghemawat03) and follows its design goals of *scalability*, *reliability* and *availability*.

To achieve scalability, the system overcomes the storage limits of an individual machine by dividing the data in subsets that are stored on different machines in a cluster. The main advantage of this approach is that it is possible to increase the system's storage capacity by just adding more machines (or storage devices) to the cluster.

HDFS follows traditional filesystem design. Files are stored as blocks or chunks and the system has some metadata to keep track of the mapping between filenames and blocks, directory tree structure, permissions, etc. One difference is that the blocks are much larger than general purpose filesystems. While these filesystems use 4 KB or 8 KB block sizes, HDFS uses 64 MB by default, but it is not uncommon to see larger blocks of 128 MB, 256 MB or even 1 GB. By using larger blocks, HDFS minimizes disk seek operations, resulting in high throughput (Sammer12).

For reliability, the system uses the concept of *replication*. Data blocks are replicated on multiple machines in the cluster, three by default. When the number of copies of a block falls below the *replication factor*, the system automatically makes a new copy of that block. The replication mechanism also helps the system to achieve high availability, as the system can tolerate machine failures and recover from them automatically.

An HDFS cluster has two types of nodes: one *namenode* (master) and a number of *datanodes* (workers) (White12). As shown in Figure 3, the HDFS client communicates with the namenode to retrieve information about the filesystem metadata, and with datanodes to read and write files. The namenode keeps in memory the filesystem metadata, such as which datanodes manage the blocks for each file, and provides management and control services. The datanodes store, retrieve data blocks and report to the namenode periodically with the list of blocks they are storing.

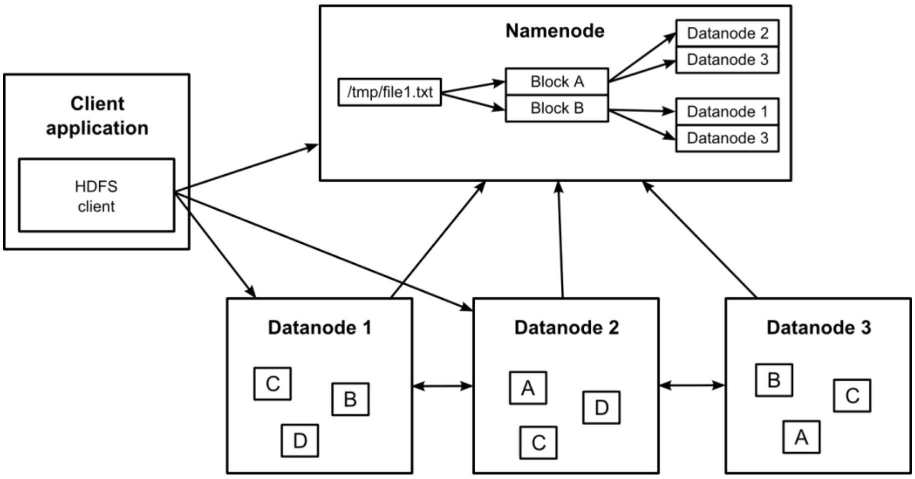


Figure 3: HDFS architecture.

Source: Adapted from Holmes (2012).

As there is only one namenode, it represents a single point of failure in the system (Venner09). If the machine running the namenode is no longer available the filesystem cannot be used and all the files in the filesystem will be lost. Hadoop provides two mechanisms to make the namenode resilient to failures. The first way is to back up the filesystem metadata periodically; another way is to run a *secondary namenode*. This node does not act as a namenode, as the name suggests, but it merges periodically the filesystem metadata with the edit log and keeps a copy of the merged data that can be used if the namenode fails.

2.3.2. MapReduce model

The MapReduce model, first proposed by Google (Dean04), is a method for solving petascale problems in large clusters of commodity machines (Venner09). It was largely inspired by the *map* and *reduce* primitives present in Lisp and other functional languages.

The model hides the inherent complexity of distributed systems and allows the parallelization and distribution of large-scale computations. With this abstraction it is possible to focus on the data processing and to rely on MapReduce to manage the details of parallelization, fault-tolerance, data distribution and load balancing.

In their seminal work, Dean and Ghemawat (Dean04) claimed that their implementation of MapReduce was already being used by Google to process many terabytes of data on thousands of machines on a daily basis. The framework was highly scalable and drew the attention of both the academy and the industry. It did not take long for others to come up with open-source implementations of their model, being Hadoop's the most widely adopted. This is the MapReduce implementation used in ICP.

Sammer (2012) highlights the main features of the model. Firstly, it is *simple to develop*. The developer does not have to worry about socket or thread programming. By using simple functional programming concepts, it is possible to build robust applications to deal with extremely large datasets. Secondly, MapReduce is *highly scalable*. Tasks are independent and can be executed in parallel on separate machines. MapReduce is a shared-nothing system and applications can readily take advantage of recently added machines. Thirdly, it *automatically parallelizes and distributes work*. While developers focus on the map and reduce functions, MapReduce stores the input data in a distributed filesystem and takes advantage of its characteristics to parallelize data processing. Finally, MapReduce is *fault tolerant*. In an environment with a large number of machines, failure is not an exception, but it is likely to happen. The framework deals with such a scenario by rescheduling failed tasks and keeping track of problematic nodes.

MapReduce is composed by three steps: *map, shuffle and sort*, and *reduce*. Firstly, the framework takes as *input* a set of key/value pairs, which represents a logical record from the input data source. In the case of a file it could be a line, or if the input source is a table in a database, it could be a row. The *map* function is applied to an input key/value pair and produces zero or more *intermediate* key/value pairs. The *shuffle and sort* are responsible for determining the reducer that should receive a key/value pair (partitioning) and ensuring that, for a given reducer, all its input keys are sorted. The *reduce* function gets an intermediate key and the set of values associated to that key and combines or aggregates those values in order to produce a smaller set of values, typically just one output value.

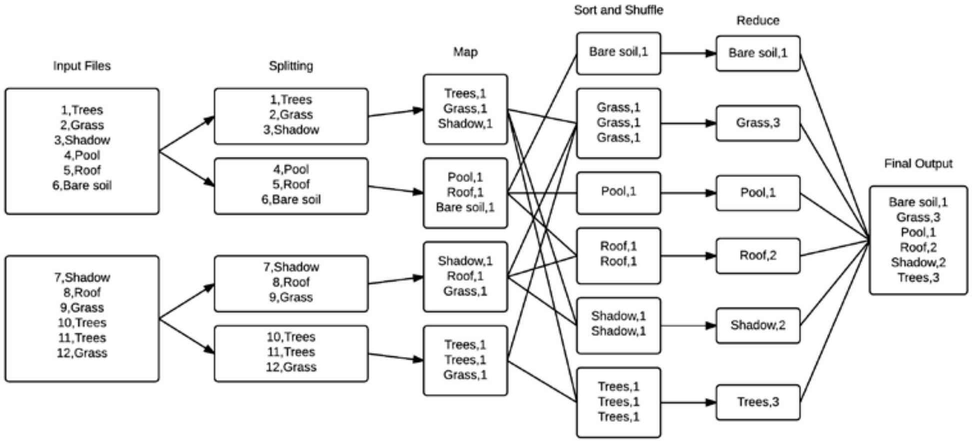


Figure 4: MapReduce example.

As an example, let us consider a simple application that counts the number of image objects in each land cover class (Figure 4). Let us assume that we have two input files with 12 records containing object id and land cover class. Firstly, the map phase splits the input data according to a given split size. In this example, each split equates to three lines. For each split, the map phase splits each record and outputs the class (intermediate key) and a 1 to indicate the class has been seen one time (intermediate value). Once the map phase is complete, the sort and shuffle phase sorts all records by key, collects the records with the same key and sends them to the same reducer. The reduce phase sums up the number of times each class was seen and outputs these values together with the class as output.

The next section presents some details of Hadoop's MapReduce implementation.

2.3.3. Hadoop MapReduce

Hadoop MapReduce is the computation component of Hadoop that implements the MapReduce model. Hadoop MapReduce is aware of HDFS and can use the namenode during the scheduling of tasks to decide the location of the map and reduce tasks, following the "moving computation to data" philosophy. This avoids network overhead as workers do not need to copy data over the network to access it (Sammer12).

This ability is close to the notion of *data locality*. In many high-performance computing (HPC) systems data is stored on a large shared centralized storage system. During the execution of a job, workers retrieve the data from the central storage system, process it, and write the result back to the storage system. For large datasets, when there is a large number of workers retrieving the same data at the same time, performance suffers.

Instead of a central storage system, MapReduce uses a distributed file system where each node is at the same time a storage node and a compute node. The framework pushes the computation to the machines where blocks can be read locally. This characteristic benefits from HDFS's replication feature. Replication not only increases data availability but also increases the chance to assign a task to a machine that is available to perform a computation.

As shown in Figure 5, Hadoop MapReduce architecture is similar to the master-slave model in HDFS (Holmes12). There are two major processes in Hadoop MapReduce: the *jobtracker* and the *tasktracker*. The jobtracker is the master process. It accepts job submissions from clients, schedules tasks execution on worker nodes and provides other administrative functions. There is one jobtracker per MapReduce cluster. Clients and tasktrackers communicate with the jobtracker via remote procedure calls (RPC). Tasktrackers use regular heartbeats to inform the jobtracker about their progress, just like the relationship between datanodes and the namenode in HDFS.

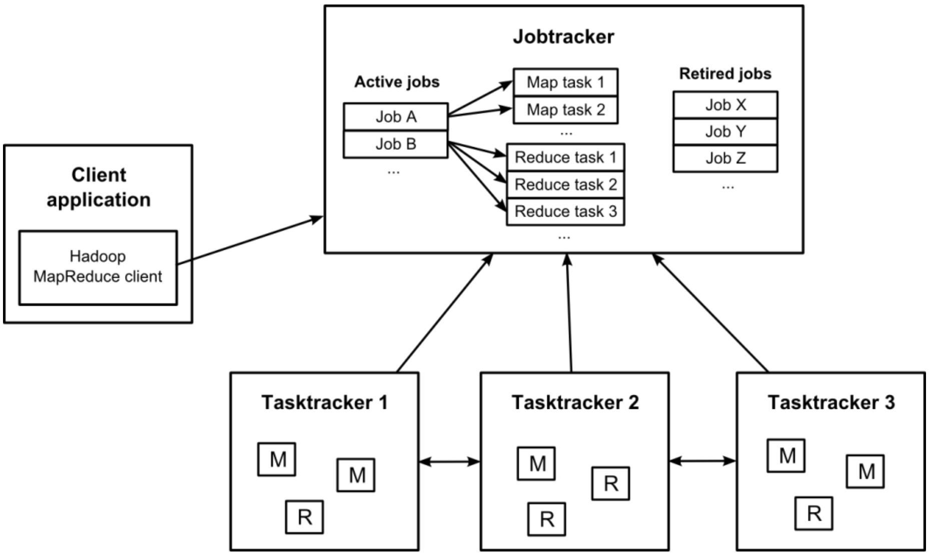


Figure 5: Hadoop MapReduce architecture.

Source: Adapted from Holmes (2012).

The tasktracker accepts task assignments from the jobtracker, instantiates the user code, executes the tasks locally and reports progress back to the jobtracker periodically. There is always a single tasktracker on each worker node. Tasktrackers and HDFS’s datanodes run on the same machines providing compute and storage capabilities, respectively. Each tasktracker has a number of map and reduce *task slots*, that represent the available resources in the worker machine that a task can be assigned to. Setting the number of task slots depends on the number of cores (physical and virtual), disk, memory and whether or not the job is CPU intensive. Similarly, finding the right balance between the number of map and reduce tasks is job dependent.

2.3.3.1. Fault tolerance

To achieve fault tolerance, MapReduce treats failures as common and inevitable. A cluster with tens, hundreds or thousands of machines is very likely to experience failures at a significant rate (Sammer12), regardless if the problem is in the machines, disks, network or even in the data.

To cope with task failures, Hadoop MapReduce provides the concept of *task attempts*. A tasktracker has a limit of four task attempts by default, below this limit the failed tasks are rescheduled to run. If this limit is exceeded, the tasktracker goes to the job-level blacklist, which prevents tasks from the same job from being assigned to that specific tasktracker. If tasks from multiple jobs fail on a specific tasktracker, that tasktracker is added to the global blacklist for 24 hours.

If the tasktracker of a specific worker does not send a heartbeat for a configurable period, the tasktracker is considered dead, along with the tasks it was assigned, and the tasks are rescheduled to another tasktracker.

Like the namenode in HDFS, the jobtracker is a single point of failure in Hadoop MapReduce. If it fails (or the machine on which it runs) the whole job fails.

2.4.
Apache Pig

Pig is a framework for executing data flows in parallel on Hadoop (Gates11). It includes a language, *Pig Latin*, for expressing these data flows; and an engine that compiles Pig Latin scripts into a series of one or more MapReduce jobs that are then executed in the cluster.

Pig started out as a research project at Yahoo! Research in the early 2000’s. With the wide adoption of the framework in the following years, Pig became a top-level Apache project in 2010 (Pig15).

The MapReduce programming model is appealing to programmers because it simplifies the development of parallel applications. There are only two high-level declarative primitives (map and reduce) that must be defined and the rest of the code (the map and reduce functions) can be written without worrying about parallelism.

However, the MapReduce model has its limitations. Its one-input, two-stage dataflow is very rigid. To perform tasks with a different dataflow, joins or n-stage flows for example, inelegant workarounds have to be conceived (Olston08). Even for simple operations like projection and filtering, custom code must be provided. These factors lead to code that is difficult to reuse and to maintain. Additionally, the system cannot perform optimizations due to the opaque nature of the map and reduce functions.

The goal of Pig's development team is to make Pig Latin the native language of parallel data-processing environments like Hadoop (Gates11). Pig provides several advantages over using MapReduce directly. It provides processing operations such as group by, order by, filter and projection that otherwise would have to be coded directly in MapReduce. This gets even worse for more complicated operations like join. Pig provides some complex, non-trivial implementations of these operations. For example, Pig has join and order by operators that take into account the uneven distribution of records per key in the reduce phase and rebalance the reducers. Algorithms like this may take months to be written in MapReduce.

Pig Latin scripts are easier to write and to maintain than MapReduce's Java code. This means that writing the same code in Pig takes less time and fewer lines of code than in MapReduce. On the one hand, there may be some specific algorithms that are harder to implement in Pig. With MapReduce, developers have more control and, given enough time, they will likely outperform a generic system like Pig. On the other hand, Pig framework analyzes the whole data flow in the attempt of optimizing it before it is compiled in MapReduce jobs. This usually leads to better performance than programming MapReduce directly.

Its creators claimed that Pig was designed to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of MapReduce (Olston08). By offering this high-level framework Pig provides another key advantage. As MapReduce evolves, the code is likely to change from version to version, making old codes hard to maintain or even incompatible with newer versions. Pig hides away MapReduce's implementation details from users making these transitions between versions much smoother.

As an example, let us consider a simple SQL statement that finds the average area of large objects for each sufficiently large class:

```
SELECT class, AVG(area) FROM objects WHERE area > 200 GROUP BY
class HAVING COUNT(*) > 10^3
```

An equivalent Pig Latin script would be:

```
large_objects = FILTER objects BY area > 200;
groups = GROUP large_objects BY class;
big_groups = FILTER groups BY COUNT(large_objects) > 10^3
output = FOREACH big_groups GENERATE class,
AVG(large_objects.area);
```

A Pig Latin program is a sequence of steps, much like in a programming language. Each step carries out a single high-level data transformation such as filtering, grouping, and aggregation, like in SQL. These high-level primitives make low-level MapReduce manipulations unnecessary.

Writing a Pig Latin program is similar to specifying a query execution plan (i.e. a dataflow graph) which makes it easier for programmers to understand and control how the data will be processed. For programmers, this method is more appealing than writing an SQL query. An SQL query is usually oriented to answer just one question. When users need to perform several data operations things get more complicated. One option is to create separate queries and store the partial results in temporary tables. Another option is to write subqueries that have to be nested in order to process the data in the right order. Both options are not so appealing. With Pig, there is no need to worry about temporary tables or subqueries. It was conceived to process long chains of data operations.

The main reason for these differences is that SQL was designed for a RDBMS environment. In these environments, data is stored in tables, normalized and with proper constraints. That is not the case with Pig. It was designed to work in the Hadoop environment, where data is not normalized and there are no tables.

Thus, Pig Latin use cases are usually extract transform load (ETL) data pipelines, research on raw data and iterative processing, but the main one is *data pipelines* (Gates11). Furthermore, Pig is suitable for the *batch processing* of data. It is the appropriate tool to sequentially process gigabytes or terabytes of data, but when random lookups are needed or the workload is much smaller, NoSQL (Fowler12) databases may be a better solution.

2.4.1. Pig Latin

Pig Latin is a dataflow language (Gates11). This means it differs from most programming languages like Java and C++, which are control flow languages. In these languages the dataflow is just a side effect. In Pig Latin, the user defines a sequence of steps where each step represents a single, high-level data transformation.

Although Pig Latin programs provide an explicit sequence of operations, they do not have to be executed in that order. High-level primitives like `GROUP` and `FILTER` allow some optimizations. Let us consider a Pig Latin script that is interested in image objects that have a high rectangular fit score but have a small area:

```
rectangular_objects = FILTER objects BY rectangularFit(geometry) >
0.8;
small_objects = FILTER rectangular_objects BY area(geometry) < 5.0;
```

Although the script suggests that the filter that uses `rectangularFit` will be executed before the filter that uses `area`, this might not be the best decision. If the function `rectangularFit` is an expensive user defined function, it would be more efficient to first filter the objects by `area` and then invoke `rectangularFit` on the resulting objects. If these filters were implemented directly in a map or reduce function, this optimization would be impossible.

Another feature of the language is the ability to operate over plain input files without any schema information. The user has only to provide Pig with a function that parses the content of the file into Pig Latin's data model. There is no need for a time-consuming data import process. Similarly, the output can be written in any format according to a user defined function that converts Pig Latin's data model into a byte sequence. This allows the framework to readily interoperate with other applications.

The language is built for the processing of web-scale data, thus Pig Latin includes only a small set of primitives that can be easily parallelized. Operations that do not lend themselves to efficient parallel execution were intentionally excluded but can still be implemented via UDFs (user defined functions). In this case, users are responsible for the efficiency of their programs.

To facilitate the debugging process over such large scale datasets, Pig Latin also provides a novel interactive debugging environment that illustrates the output of each step of the user's program. The sample data is carefully chosen to resemble the real data as much as possible.

2.4.1.1. User Defined Functions (UDFs)

One of Pig's main features is the possibility to extend the framework. All aspects in Pig Latin, including grouping, filtering, joining and per-tuple processing can be customized through the use of UDFs. Users can combine the default operators with their own or others' functions. Currently, these UDFs can be written in six languages: Java, Jython, Python, JavaScript, Ruby and Groovy.

The most extensive support is provided for the Java language. With functions written in Java it is possible to customize all parts of the processing including load/store functions, column transformation and aggregation (Pig15). Java functions are also more efficient because they are implemented in the same language as Pig and because they can implement additional interfaces like *Algebraic* and *Accumulator*, that can speed up the processing significantly. The former does that by utilizing Hadoop's combiner, which reduces the skew in the reduce tasks and the amount of data sent over the network between the map and reduce tasks. The latter allows Pig's UDFs to work on subsets of the data and avoids records to be spilled to disk. Support for the other languages is limited.

Pig comes with a large number of built-in UDFs. Besides these UDFs, there is also *PiggyBank* (PiggyBank15), a collection of user-contributed UDFs that is shipped along with Pig.

There are four types of UDFs: *evaluation, filter, load* and *store* functions. Evaluation functions can process and return single elements of data or collections of data. Filter functions are a special case of evaluation functions that can only return Boolean values. Load and store functions are the ones concerned with data input and output.

2.4.2. Pig’s engine

Pig runs on Hadoop and uses its distributed file system (HDFS) and its processing framework, MapReduce. One of Pig’s advantages is that there is no need to worry about the map, shuffle and reduce phases. Pig’s engine manages the decomposition of the commands present in a Pig Latin script into the appropriate MapReduce phases.

The engine analyzes a Pig Latin script and understands the data flow described by the user. In this step, the engine can do an early check for errors and perform some optimizations before compiling the Pig Latin script into MapReduce jobs. Figure 6 shows how a sequence of Pig Latin commands would be compiled into MapReduce jobs.

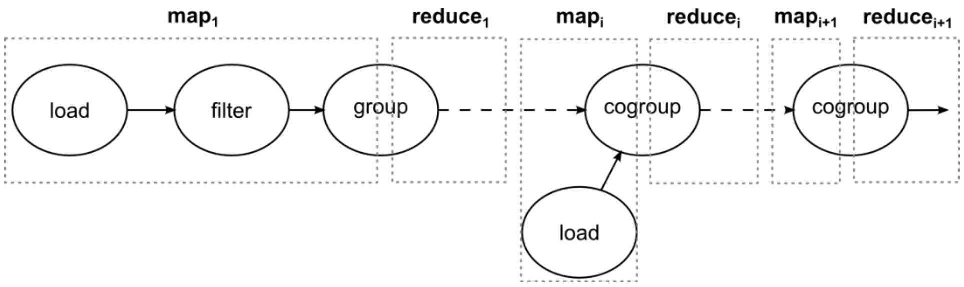


Figure 6: MapReduce compilation of Pig Latin.

Source: Adapted from Olston (2008).

Although Pig currently uses Hadoop as its execution platform, its parser and logical plan constructor are independent of the execution platform. Only the compilation of the logical plan into a physical plan depends on the specific execution platform (Olston08). The latest versions provide a pluggable execution engine feature which will allow Pig to run on non-MapReduce engines in the future.

3
Related work

This work, to the best of our knowledge, is the first to provide a complete MapReduce-based architecture for geographic object-based image analysis. Similar efforts have been made for medical images (Aji12), where the authors proposed a MapReduce-based query system for microscopy images. This system implemented a spatial query system providing basic spatial operations like join and nearest neighbor. This architecture included a query engine called RESQUE (Real-time Spatial Query Engine), a spatial SQL-to-MapReduce translator based on *YSmart* (Lee11) and Hadoop as the execution engine.

In 2013, a more generic version of this system was proposed (Aji13). Now with a new name – Hadoop-GIS – the system was meant to not only work with medical images but any spatial data, including remote sensing data. However, the main objective was still similar: to provide a MapReduce-based spatial query system with join, containment and aggregation queries. This work improved the original system by providing boundary object handling and skew-aware data partitioning.

In the same year, another study proposed a spatial extension to the Hadoop framework, called SpatialHadoop (Eldawy13), which would enable the efficient processing of spatial operations. The extension provided a two-layered spatial index over Hadoop with implementations of Grid file (Nievergelt84) and R-tree (Guttman84). They are used as a global index that partitions data across cluster nodes and local indexes data organize data inside each node, respectively. SpatialHadoop achieved up to 260x speedup for a dataset of 128GB in comparison to Hadoop. The spatial operations tested were polygon union, skyline, convex hull, closest pair and farthest pair.

Although the results are interesting, there is an important issue with this approach related to technology dependency. The spatial extension is too closely related to Hadoop's source code. That means that any major changes in Hadoop (that are likely to happen) would imply in major changes or even incompatibility of the extension. By using on-demand, R-tree-based spatial indexes, Hadoop-GIS (Aji13) achieved almost linear speedups in comparison to the commercial system DBMS-X but at a safer distance from Hadoop's source code. The index creation overhead, as shown in the experiments, only accounted for a small fraction of the overall query response.

Several similar systems were proposed to perform spatial operations. Parallel Secondo (Lu12) is a parallel spatial DBMS which uses Hadoop as a distributed task scheduler, while spatial DBMS instances running on cluster nodes take care of storage and query processing. *MD*-HBase (Nishmura11) extends HBase to support multidimensional indexes which allow efficient range and *k*NN queries. VegaGiStore (Zhong12) is a spatial query system that relies on a geography-aware approach and a two-tier distributed spatial index.

All these works implement a sort of spatial query system based on Hadoop that emulate common operations of conventional SDBs (spatial databases) (Güting94). SDBs perform well in relatively small datasets (Shekhar03) but their capabilities can hardly meet the performance requirements of queries over big spatial data. Another emerging solution is the KVS (key-value stores) systems such as BigTable (Chang08), HBase (HBase15) and Cassandra (Lakshman10). They are proved to be interesting scalable alternatives to store big semi-structured data.

Although these approaches are interesting for query processing they are not suitable for data processing. In ICP, data is short-living, as partial results are combined in order to produce new results in a sort of pipeline or data flow. Thus, MapReduce batch processing framework is more adequate for this type of processing, while databases have their place providing real-time interaction (queries) between the user interface and the final results of these pipelines.

In (Zhou98), a partitioning based approach for parallelizing spatial joins is discussed. This work uses a multiple-assignment, single-join approach with the PBSM (Partition Based Spatial-Merge Join) algorithm (Patel96). It also provides a strategy to rebalance the tasks to achieve better parallelization.

In ICP, we use a *dynamic assignment* approach, where *multiple assignment* (Lo96) and *multiple matching* (Zhou98) approaches are used depending on whether data replication is needed. Following the approaches of (Zhong12), (Aji13) and (Eldawy13), our architecture uses a two-tier distributed spatial index based on a global tile grid and on on-demand R-tree-based spatial indexes. The system relies on MapReduce for load balancing. Although there are some similarities between Hadoop-GIS (Aji13) and ICP, there are two major differences. Firstly, Hadoop-GIS is not concerned with image handling and spectral feature computation: the features are provided along with the image objects as input. Secondly, Hadoop-GIS is interested in providing a spatial query system and not a framework for image analysis. Although image segmentation is not in the scope of this thesis, the final objective of ICP is to provide a full-fledged object-based image analysis platform comprising image segmentation, feature extraction and classification.

Programming MapReduce directly can be difficult for non-technical users. Several high-level languages were developed to simplify the interaction with Hadoop MapReduce including Pig Latin (Olston08), HiveQL (Thusoo09) and Y-Smart (Lee11). These languages allow users to describe their programs in terms of primitive operations like filter, sort and join. While HiveQL and Y-Smart are SQL-to-MapReduce translators, Pig Latin presents a different approach. It focuses on data processing which is more suitable for programmers.

Eldawy and Mokbel (Eldawy14) propose, as a continuation of their previous work (Eldawy13), a spatial extension to the Pig framework. Relying on the ESRI Geometry API (ESRI15), the authors introduce spatial operations like *contains*, *overlaps* and *intersection*. Aji et al. (Aji13) also proposes a spatial extension to HiveQL with spatial constructs, spatial query translation and execution. Their extension is based on their own spatial query engine called RESQUE.

ICP relies on the Pig framework – which provides a high-level data flow language (Pig Latin) and an efficient MapReduce compiler – and extends this framework with spatial operations and spectral feature computation. Spatial operations are powered by the JTS library (JTS15) and image operations rely on Java Advanced Imaging library (JAI15) and ImgLib2 (Pietzsch12).

Regarding image processing systems, there are also a number of works. Lin et al. (Lin13) proposes a cloud-based framework for massive remote sensing image storage and processing using Mahout (Mahout15) and MapReduce. A pixel-based classification is performed using the K-means implementation provided by the Mahout library. Liu et al. (Liu12) introduces a MapReduce approach for processing large-scale remote sensing images. Their approach uses a multiresolution tile pyramid and a tile-based processing to achieve parallelism in MapReduce. The experiments show interesting results for SURF (Bay06) and Sobel edge detection algorithms. Liu et al. (Liu13) proposes a distributed system that relies on HBase (HBase15) for image data storage and MapReduce for image data processing. Wang et al. (Wang12) presents a large-scale multimedia data mining approach using the MapReduce framework. Some methods like point detection and clustering are investigated for images and event detection and near-duplicate retrieval for videos. All these works have in common that they are focused on image processing and point- or pixel-based image analysis.

In fact, it seems that much has been done on distributed systems for spatial operations and raster processing, but little or nothing has been done, in this context, for object-based image analysis. This is interesting, as object-based image analysis places itself as a bridge between the raster and the vector domains.

The architecture proposed in this work fills this gap by providing a platform that enables image segmentation (although it is out of the scope of the present thesis), feature extraction and image object classification in a distributed environment, considering raster data and vector data in the interpretation process. Thus, ICP is a platform based on Hadoop and Pig frameworks that provides a distributed architecture to perform object-based image analysis.

There is another aspect to ICP that is related to knowledge representation. One of the main goals of the platform is to provide an intuitive, flexible and powerful knowledge representation structure. Image interpretation has been regarded, traditionally, as a pattern recognition problem. Among the pattern recognition categories presented in (Jain00), statistical (Webb02), machine learning (Li00; Mciver01; Zhong08; Chi05) and structural methods (Sagerer97; Liedtke99; Bückner01; Schiewe01; Centeno03) are particularly important for the analysis of remote sensing data (Costa10).

While *statistical* and *machine learning* methods present a high demand for training samples, *structural methods* use expert’s knowledge to reduce significantly this requirement. This main characteristic makes such methods more suitable for remote sensing applications. Structural methods involve complex patterns in which knowledge is built recursively from simpler patterns until primitive patterns are reached. In knowledge-based or cognitive systems, the classes of objects that are expected to be found in an image are defined based on an explicit representation of the expert’s knowledge about their spectral, morphological and topological characteristics (Costa10).

There are other advantages of utilizing explicit knowledge representation (Crevier97). Firstly, knowledge can be added to a knowledge base, without modifying preexisting rules. Secondly, it can be more easily validated and favors interactive problem solution. Finally, explicit knowledge leverages collaboration and knowledge exchange between those working on similar problems (Costa10).

Systems like ERNEST (Kummert98), AIDA (Liedtke99), GeoAIDA (Bückner01) and InterIMAGE (Costa08) use a semantic network to represent knowledge. Besides declarative knowledge, this structure is closely related to the image interpretation strategy (procedural knowledge), where *data-driven* and *model-driven* operations are performed according to the hierarchical structure of the semantic network.

Like these systems, ICP provides a semantic network for the hierarchical modelling of target classes. However, the procedural knowledge is detached from this hierarchical structure and represented in a more generic graph structure. A graph lends more flexibility for the modelling of the expert’s knowledge allowing different combinations of operations.

4
InterIMAGE Cloud Platform (ICP)

The distributed architecture for object-based image analysis proposed in this work is inspired by the InterIMAGE system (Costa08). InterIMAGE is a knowledge-based platform that follows the GEOBIA approach. It is an extensible, open-source system for the automatic, knowledge-based interpretation of remote sensing data. InterIMAGE has the following features: (i) it is free, unlike other softwares that perform object-based image analysis and can cost a few thousands of dollars; (ii) it is extensible; the software provides mechanisms that allows users/developers to extend the system by adding external operators to perform specific tasks; (iii) it is open-source; it means it is possible to check the source code and understand exactly what the system does behind the scenes.

These characteristics make InterIMAGE and other free and open-source softwares (FOSS) a much more interesting tool for scientists than commercial alternatives. It did not take long for this system to start drawing the attention of the academy. This can be seen by the number of academic works that investigate the possibilities of the system (Novack10; Sousa11; Camargo12). However, as a system which's core was developed in the beginning of the 2000's (Bückner01), when the number of satellites and image resolutions available today were not a reality, the system was not conceived for the interpretation of such large datasets. This came to be one of its main limitations.

In order to stay up to the new sub-meter resolutions and to the huge amount of image data produced today, InterIMAGE needed to be redesigned. The architecture that will be presented in this chapter – named *InterIMAGE Cloud Platform (ICP)* – is a total redesign of InterIMAGE and provides a distributed platform for cloud environments based on Hadoop and Pig. By utilizing the MapReduce programming model, this new platform is able to interpret very large remote sensing image datasets. The next sections present this novel architecture in detail.

4.1. Architecture overview

ICP’s architecture inherits some features from InterIMAGE. There is a *semantic network* where the user can hierarchically define the semantic concepts that they expect to find during the execution of an interpretation model; and a *control strategy* that executes operators that perform specific tasks. Although the general idea is similar, the new architecture is different in many ways.

In InterIMAGE, besides representing the class hierarchy, the semantic network also defines the interpretation logic, i.e. the operators’ execution sequence in an interpretation model. In ICP, the semantic network represents only declarative knowledge, while procedural knowledge is defined in a different component called *operator graph*. This new component allows the construction of more powerful and flexible models in comparison to the original InterIMAGE. This flexibility makes it easier for the interpretation models created in ICP to adhere to the user’s knowledge.

Another difference is that in InterIMAGE there is limited data partitioning and processing parallelization, i.e. at some point of the interpretation the whole dataset is processed at once. This limits the input size the system can handle to the size it can fit in the main memory. From the user’s perspective, that is undoubtedly where the two systems differ the most. In ICP, data storage and processing is inherently distributed, which allows the system to process large volumes of data in parallel.

Although InterIMAGE allows the use of some fuzzy logic in its *decision rule*s, it does not take full advantage of this approach. It keeps a copy of an object hypothesis for each class it has been assigned, even when the geographic regions are the same (e.g. the image segments were yielded by the same segmentation algorithm). In ICP, the same image object can belong to more than one class with different membership values, very much like a fuzzy system would do. This not only avoids data replication but also simplifies the *spatial conflict resolution* procedure.

Although it is relatively simple, in InterIMAGE, to create a new operator and add it to the system, this procedure still requires good programming skills. In

ICP, operators are created in Pig Latin which makes it easier for non-technical users to add new functionalities to the system.

These characteristics show that ICP does not only provide a distributed platform for parallel execution, it also offers a more flexible and powerful architecture that facilitates the construction of interpretation models. Figure 7 illustrates the proposed architecture.

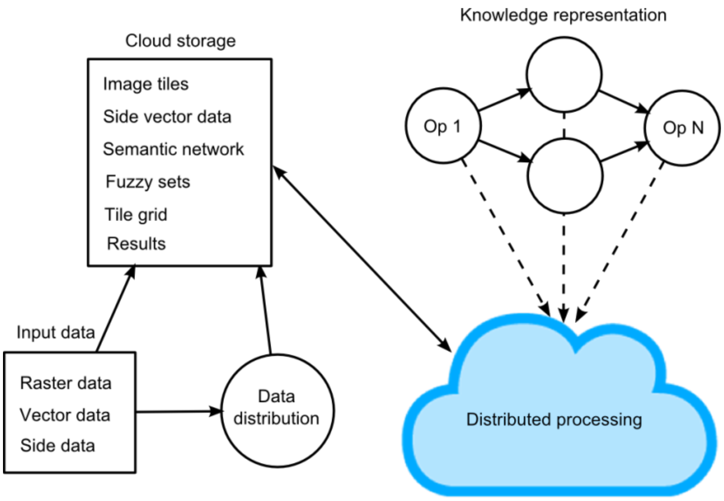


Figure 7: ICP’s architecture.

ICP’s architecture (Figure 7) is composed by three main components: *knowledge representation*, *data distribution* and *distributed processing*. Next sections discuss each of these components.

4.2. Knowledge representation

As stated before, knowledge representation in ICP is obtained through two different components: *semantic network* and *operator graph*. The former is responsible for declarative knowledge while the latter is responsible for procedural knowledge.

4.2.1. Semantic network

The semantic network represents hierarchically the semantic concepts or classes that an expert expects to find during an interpretation. It is a tree-like structure where nodes represent concepts and the edges represent relations between these concepts (Figure 8).

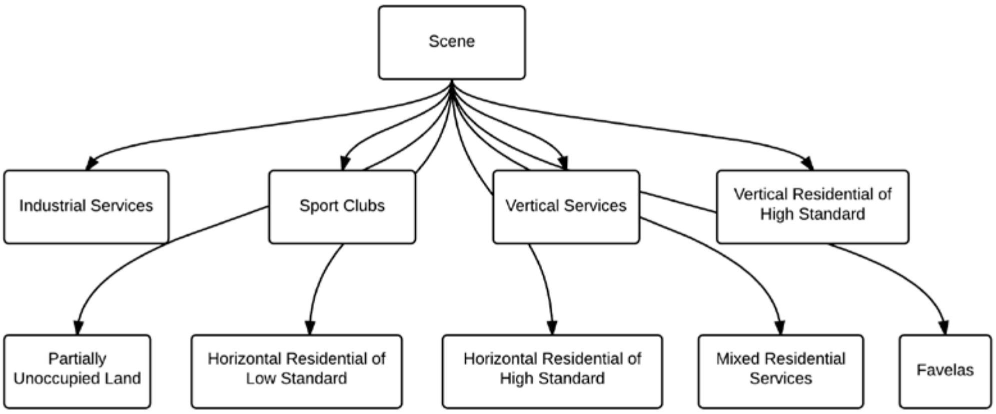

Scene
Industrial Services
Sport Clubs
Vertical Services
Vertical Residential of High Standard
Partially Unoccupied Land
Horizontal Residential of Low Standard
Horizontal Residential of High Standard
Mixed Residential Services
Favelas

Figure 8: Example of a semantic network.

4.2.2. Operator graph

The operator graph (Figure 9) is responsible for the *control strategy*. It is a directed graph that defines the operators and their relations in an interpretation model. The nodes represent operators that are mostly related to the generation and validation of object hypotheses associated to the classes defined in the semantic network; and the edges represent the graph’s structure, indicating the operators’ execution sequence.

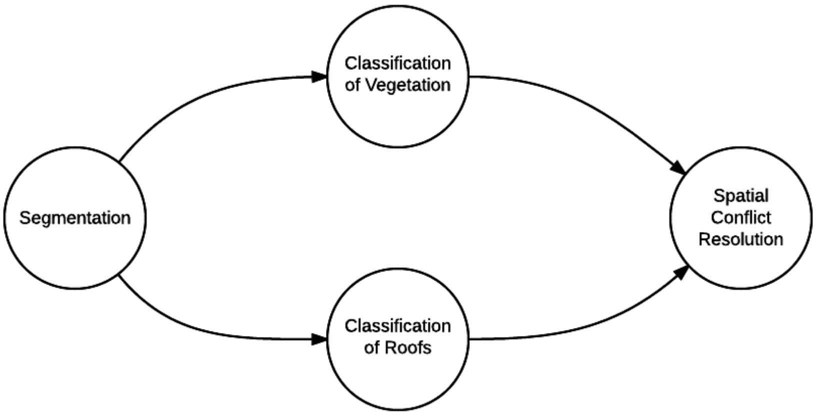


Figure 9: Example of an operator graph.

A common workflow in ICP is composed by three steps: *object generation, hypothesis generation* and *hypothesis validation*. Firstly, the segmentation process generates image objects. Secondly, classification operators take these image objects as input and classify them. It is possible that in this step the same original image object or image objects obtained from different segmentations and that present some overlap are classified as different classes. That is why from this point on they are called object hypotheses. In order to help the system to decide which hypotheses will win in the end, along with the class, classification operators also produce a membership value that represent the confidence they have that an object belong to a specific class. Finally, a hypothesis validation step takes place where spatial conflicts are resolved.

In InterIMAGE, there are two types of operators: top-down and bottom-up. The former is used to find regions in the image associated to specific semantic concepts and to build a net of object hypotheses. The latter is used to validate or discard these hypotheses and resolve eventual spatial conflicts generating the final net of object instances. Each semantic node may have a top-down and a bottom-up operator associated to it.

In ICP, there is no concept of top-down and bottom-up steps. As stated before, the interpretation is no longer based on the semantic network's hierarchy. The semantic network represents only declarative knowledge while the procedural knowledge is defined in the operator graph.

A graph structure lends much more flexibility to the construction of an interpretation model. While in InterIMAGE the semantic nodes can have only one parent node, in ICP a graph node may be preceded and succeeded by any number of nodes. In this sense, in the new architecture, graph nodes (operators) may perform object generation, hypothesis generation or object validation.

In InterIMAGE, holistic (top-down) operators can be written in (almost) any programming language. Besides these operators, the system also provides another mechanism: *decision rules*. Decision rules allow the user to manipulate object hypotheses by computing features, combining and filtering them. Internally, these rules are programmed in a specific language but the latest versions provide an intuitive user interface for that. These characteristics create two levels of interaction with the system. In some sense, decision rules are custom operators that users can easily define and operators are executable programs that only programmers can create. Another consequence of this architecture is that it is not possible to access external operators from a decision rule.

In ICP, operators and decision rules are treated the same way; both of them are written in Pig Latin (from this point on, the term operator will be used to refer to both of them). This not only simplifies the system, but also allows operators to be called from a decision rule and vice-versa. In fact, ICP's architecture allows operators to be created upon other existing operators, making it easier for users to profit from what others have done in the platform. For example, an operator that classifies vegetation can be included in a decision rule that classifies land cover classes. This also is valid for the operator graph itself. A whole complex land cover classification model can be included into a higher level classification model by just including one operator. This recursive use of knowledge allows the creation of very powerful interpretation models.

In ICP, operators are defined by a Pig Latin template script and its input parameters. An operator can use Pig Latin commands and UDFs. As an example, here is the code for a basic operator that imports an external segmentation (some details were left out for the sake of readability):

```
--Loads image objects
load = LOAD '$INPUT_PATH' USING
org.apache.pig.builtin.JsonLoader('geometry, properties');
```

```
--Filters out invalid geometries
selection = FILTER load BY II_IsValid(geometry, properties, '');

--Computes tile information
projection = FOREACH selection GENERATE geometry,
II_ToProps(II_CalculateTiles(geometry,
properties#'tile'),'tile',properties) AS properties;

--Sets objects' class
projection = FOREACH projection GENERATE geometry,
II_ToProps('$CLASS','class',properties) AS properties;

--Sets objects' membership value
projection = FOREACH projection GENERATE geometry,
II_ToProps($RELIABILITY,'membership',properties) AS properties;

--Stores resulting objects
STORE projection INTO '$OUTPUT_PATH' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

The commands in `red` are Pig Latin commands. The functions in `blue` are UDFs and the terms in **`bold black`** are the input parameters. This operator basically reads a given input segmentation file, filters out invalid polygons, computes the tile labels, sets the class and membership properties and stores the result.

ICP is also extensible. It is possible to extend the system's functionalities by adding new operators and UDFs. UDFs are operators' building blocks and, depending on the objective of one operator, developers can use existing UDFs or create new ones to achieve their goal. Almost all system's UDFs are written in Java but they can also be written in other languages as shown in section 2.4.1.1.

4.2.2.1.
Graph execution

When an interpretation model is executed, the nodes (operators) in the operator graph are visited in an iterative process (Figure 10).

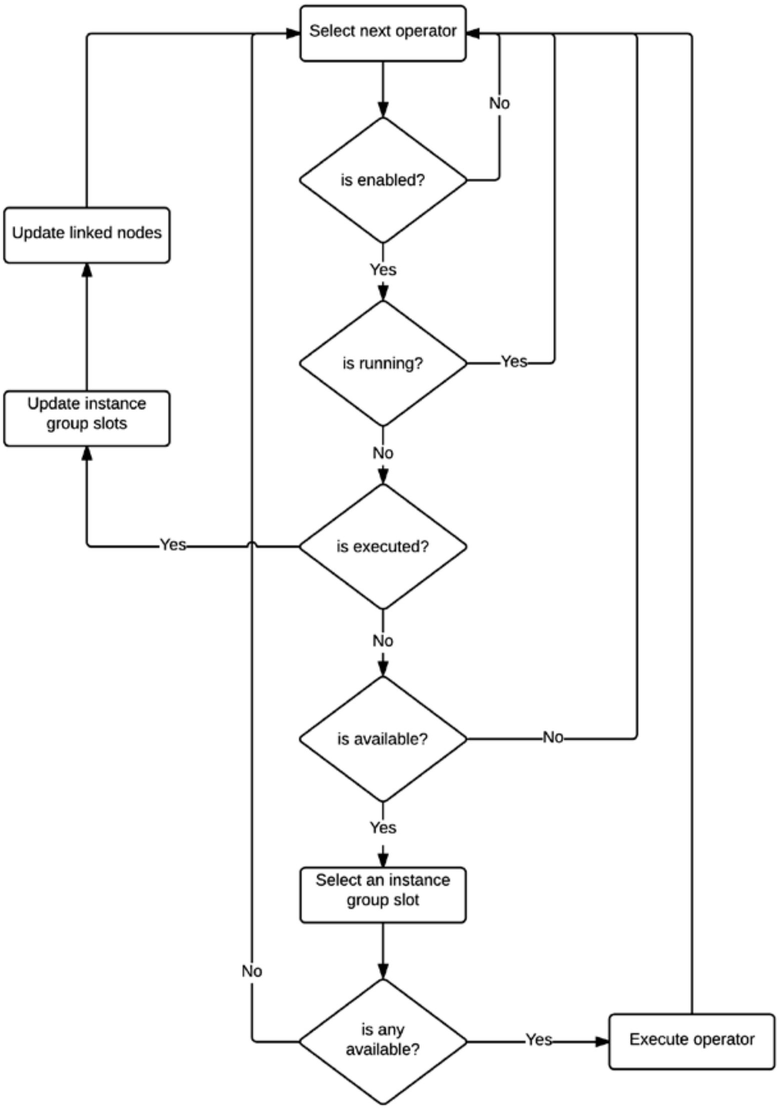


Figure 10: flowchart of the graph execution.

Before the execution, the graph has at its disposal a number of available instance group slots. They represent separate Hadoop clusters in the cloud. When an operator is selected, the graph controller verifies if the operator is *enabled* or *running*. Disabled and running nodes are skipped. Then, the controller verifies if a node that was running *finished* execution. If so, the instance group slot occupied by this operator is released and the operators that depend on it are notified.

If the operator was not running, the controller checks if it is *available*. It means verifying if all previous nodes the operator depends on were executed. If it is available, the controller assigns it to an available instance group slot. If there are no instance group slots available, the node is skipped and will be considered again in the next loop.

When an operator is selected for execution in an instance group slot, its Pig Latin script is parsed by the system’s core and the input parameters are replaced. Once the final script is obtained, it is sent to the cloud for execution.

4.2.2.2. Two-level parallelism

ICP achieves two levels of parallelism. Once an operator is executed in the cloud, ICP parallelizes its operations depending on the number of available processing units (logical cores). This is the *operator-level parallelism*.

Since in cloud environments the costs drop dramatically, it is not uncommon to have cloud systems with hundreds or even thousands of machines. With this compute power, it is possible to divide the cloud infrastructure in different instance groups and run one operator in each group at the same time, achieving *graph-level parallelism*.

It is noteworthy that this form of parallelism is constrained by the operators and graph characteristics. Even with more compute power, if an operator depends on the output of another operator it will not be possible to run them in parallel. This aspect will be clearly exemplified in the experimental analysis reported in the next chapter.

4.2.2.3. Fault tolerance

As the results of each operator are stored in an auxiliary cloud storage service, it allows users to easily recover from errors. If an operator fails because one of its MapReduce jobs has failed or for any other particular reason, users do not have to re-execute the graph from the beginning, it is possible to just re-execute the failed operator and continue the interpretation from that point on.

This characteristic may also save significant time for users when they are still building their interpretation models. They can re-execute only the operator they are working on without having to execute the whole graph again. The fact that partial results are stored also may help users to build their interpretation models as they can compare the results of each operator and understand step by step what is happening in the interpretation. Although the architecture provides this mechanism, the development of a graphical user interface is still necessary in order to allow users to fully benefit from it.

4.3. Data distribution

As stated before, ICP deals with raster and vector data at the same time. This section presents data representation and data partitioning strategies, giving special focus to how raster data are handled in the platform.

4.3.1. Data representation

In ICP, images are stored in tiles. Each image tile is represented by two files. The first one is a TIF file, that is a subset of the image itself, including all bands; and a META file that holds information about the image tile. Here is the META file description:

```
Line 1: number of bands
Line 2: image width
Line 3: image height
Line 4: geographic west
Line 5: geographic north
Line 6: geographic east
Line 7: geographic south
```

The vector data is stored in a way that it can be processed by Pig. Pig supports several data formats and the one chosen for ICP was *JSON (JavaScript Object Notation).* JSON is a widely-adopted, human-readable text format. It is an open standard that is usually used as an alternative to XML. Here is the description of the vector data file:

```
{“geometry”:“WKT format”,“properties”:{“key1”:“value1”,…}}
```

Each line in a file represents a polygon or image object. The vector object is composed by two fields: *geometry* and *properties*. The *geometry* is expressed in another widely used format called *Well-Known Text (WKT)*. It is a text markup language for representing vector geometry objects. The format was proposed by the *Open Geospatial Consortium (OGC)* (OGC15) and is supported in almost all spatial databases. With WKT it is possible to represent nearly any 2D geometry.

The *properties* field holds the properties of a vector object. An object has some *basic properties* like *tile label*, *crs* (coordinate reference system), *iiuuid* (InterIMAGE Universally Unique Identifier), *class*, and *membership*, but the number of properties may grow as new features are computed during the execution of an interpretation.

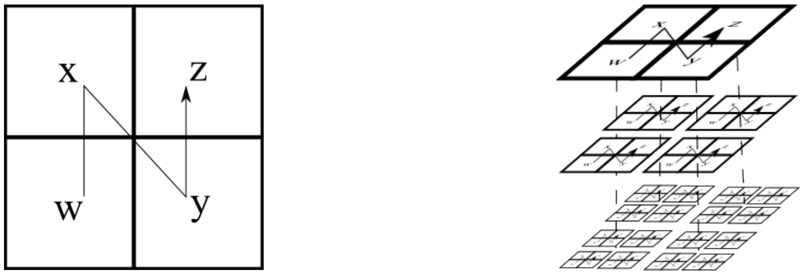
Side data like auxiliary vector data, semantic network and fuzzy sets are represented in a serialized object file.

4.3.2. Data partitioning

ICP achieves computational parallelism by partitioning the input data with respect to a geographic tile grid. The platform allows the interpretation of more than one image at the same time. The images may come from different sensors and have different pixel resolutions. When it is the case, the system considers the maximum resolution in order to partition the images.

As input images may be spatially disjoint, the geographic tile grid is built considering a given Coordinate Reference System (CRS) and is independent of the boundaries of the images. The tile size is defined in pixels and defaults to 512 as suggested by Sample and Ioup (Sample10).

In order to label the geographic tiles a *space-filling curve* (Sagan94) was used, more specifically a *z-order* curve (Figure 11 (a)). It maps a multidimensional space to one dimension and preserves locality. The resulting ordering can be regarded as the order one would get from a depth-first traversal of a quadtree (Figure 11 (b)).



(a) (b)

Figure 11: z-order (a) and quadtree representation (b).

If we name each quadrant, say ***w-x-y-z*** (Figure 11 (a)), the lower-left tile in the quadtree shown in Figure 11 (b) would have the following label: ***www***. This procedure is also known as *locality preserving hashing* (Indyk97), where the relative distance between input values is preserved in the relative distance between output hash values. Another interesting property of these hash values is that by just adding or dropping a letter it is possible to easily walk through the tree, reaching different hierarchical levels. When no hierarchical (recursive) operation is needed, the system also provides a basic tile label with numeric values.

As an example, let us consider Figure 12. It shows an image and the 81 geographic tiles that were computed for it based on the reference system EPSG:32723 - WGS 84 / UTM zone 23S. All the tile labels start with ***xwwxx***, which means that the 5th hierarchical level contains all the tiles that intersect the image. After that, each tile label has yet 10 more levels until the leaves of the quadtree are reached. Figure 12 also shows the tile label and number for the tiles in each corner of the image. For the tile in the lower-left corner, the image presents the last four levels of the quadtree in different colors.

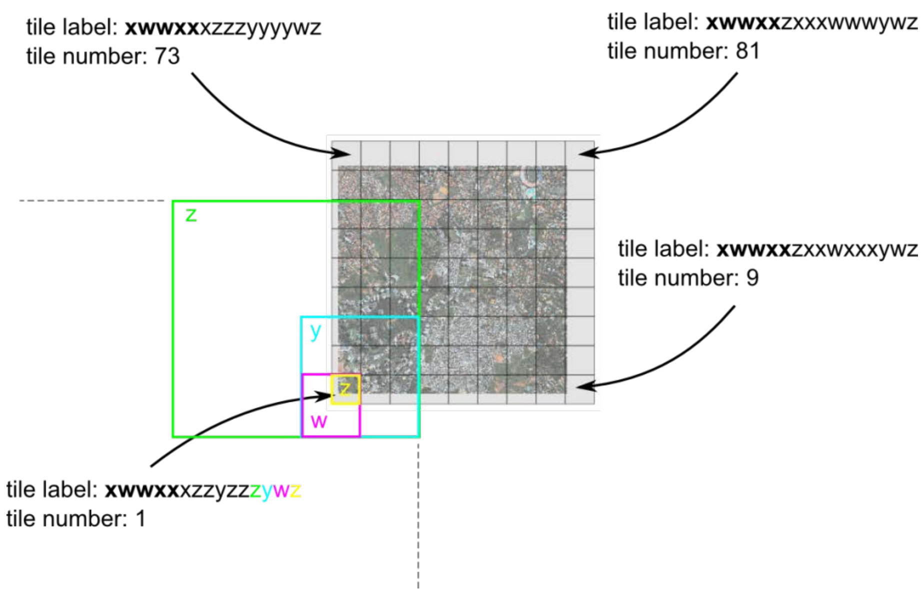

tile label: **xwwxx**xzzzyyyywz
tile number: 73
tile label: **xwwxx**zxxxwwwywz
tile number: 81
z
tile label: **xwwxx**zxxwxxxywz
tile number: 9
y
z
w
tile label: **xwwxx**xzzyzzzywz
tile number: 1

Figure 12: geographic tile representation.

Image tiles are not stored on HDFS; they are stored in an auxiliary cloud storage service and retrieved only when spectral information are needed. This mechanism will be explained in details in the next section.

Regarding image objects, the system may partition them using two different methods, depending on the operation. As an image object may intersect more than one tile, in the cases where replication is needed ICP uses *multiple assignment* (Figure 13 (a)), assigning an object to all the tiles it intersects; when there is no replication the platform uses *multiple matching* (Figure 13 (b)), where an object is assigned to a single tile. If the object intersects more than one tile, it is assigned to the tile with the lowest label, called *original tile*.

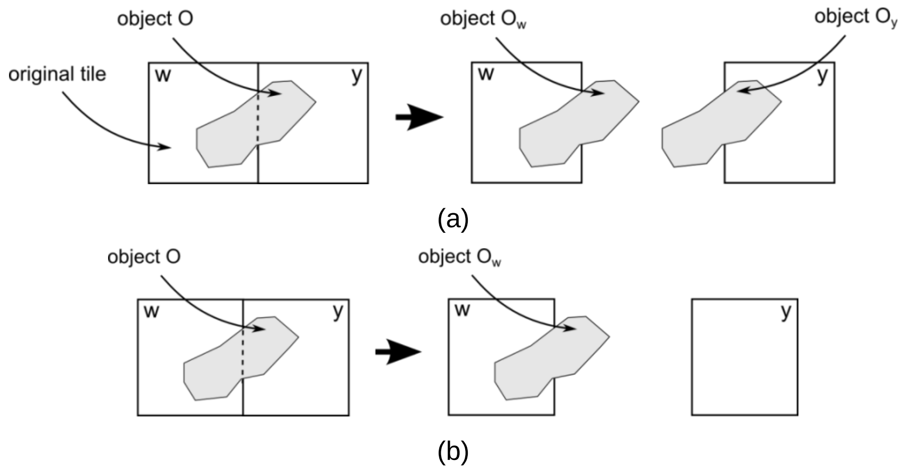


Figure 13: multiple assignment (a) and multiple matching (b).

Image objects are stored on HDFS. They are distributed in the cluster and replicated according to MapReduce’s replication factor.

4.3.3. Raster data

As stated before, ICP works with raster and vector data at the same time. One of the challenges in ICP was the definition of how to deal with both data types in MapReduce. While the systems presented in Chapter 3 which emulate spatial databases use geographic polygons as MapReduce’s main input, the other systems which are focused on image processing use the images themselves.

In fact, it is not straightforward to represent image data in MapReduce since *MapReduce is tailored for text data*. This is an important consideration as all MapReduce features were developed having text data in mind, including compression algorithms. However, these systems were quite successful at proposing new data representation techniques to overcome this limitation.

In ICP, there were three options. The first option (1) was to consider image object data (vector data) as MapReduce’s main input and load image data separately. The second option (2) was to do the opposite: consider image data as MapReduce’s main input and load vector data separately, and finally, the third one (3) was to consider both, image and vector data, as MapReduce’s main input at the same time.

A first consideration is that the granularity of vector data is much higher than image data. As an example, an image of 32,000x32,000 pixels that was used in our experiments has 3.8 GB on disk. During the interpretation, the number of image objects that were created reached 8 million, what represents around 23.2 GB of data on disk. A volume of data that is more suitable for MapReduce.

A second consideration is that *image data are not prevalent in an interpretation*. If we think about the whole chain of operations that make up an interpretation model in ICP, the computation of spectral features represents a small part of it. Once image data are considered and the spectral features are computed the rest of the interpretation is interested only in the computed spectral features, not in the image itself. As an example, the whole land use interpretation model executed in our experiments does not use image data whatsoever.

The first consideration makes option 2 to not seem so appealing, as a large set of vector data would have to be loaded from an auxiliary storage service. The second consideration also indicates that option 3 is not interesting as image data would have to be loaded, distributed and replicated on HDFS even when their processing represent a small fraction of the whole interpretation. Another drawback of option 3 is to have two different file types in MapReduce, one tailored for vector data and one for image data. This would make the system more complex.

For these reasons, ICP implements option 1. This approach considers image data as *auxiliary data*. It means they are not part of MapReduce main input data, but are loaded from an auxiliary storage service. In this approach, MapReduce input is only composed by image object data and image data are loaded on demand. One drawback of this approach is that image data have to be loaded from another storage service in the cloud and this may incur in performance loss. However, the performance in these cases depends on where image data are stored. In our experiments, for example, image tiles were stored on the same cloud service, sharing the same network as the processing machines, making these latencies imperceptible.

In order to minimize network communication, a strategy was created. Before the data of a specific image tile are considered within an operator, the image objects are grouped by tile. Therefore, for all the objects that intersect that image tile, it is loaded just once. Another strategy is to compute all spectral features that will be used in an operator at once. This way, the same image tile does not have to be loaded several times. The operation type for this approach is presented in section 4.5.2.1.

4.4. Distributed processing

This section presents the distributed environment ICP is designed for and explains how the distributed processing takes place.

4.4.1. Distributed environment

Hadoop works with clusters and cloud environments. In fact, local clusters are a particular case of cloud environments where the machines are usually homogeneous and geographically close to each other.

One important aspect of local clusters is that they are *expensive*. The cost of setting up a local cluster is not only related to the cost of the computers themselves, but also to electric power and maintenance costs. Another consideration is that if more compute power is needed, a solution is to buy more machines, but this also involves a demand for more *space* that is not always available.

For these reasons, ICP was designed for cloud environments (section 2.2). These systems usually offer much cheaper prices and charge users in a per-use basis, what enables their use by students and scientists without the resources to set up a local cluster.

As stated before, since clusters are a particular case of cloud environments, the platform also works with clusters. In fact, the platform works even in a single desktop as long as Hadoop and Pig are properly installed and configured.

4.4.2. Cloud processing

ICP’s cloud processing relies on Hadoop and Pig frameworks for MapReduce processing and data flow execution. As mentioned earlier, before execution, input data is partitioned (4.3.2) and sent to the cloud. After that, ICP analyzes the operator graph and executes each operator, by sending its Pig Latin script to the cloud for execution (4.2.2.1).

Once the operator’s Pig Latin script is sent to the cloud, the Pig framework analyzes the script and compiles it into MapReduce jobs that will be executed in a distributed fashion according to the number of available machines (*K*) (Figure 14). In our experience the overhead of using the Pig framework is negligible as the compilation time takes only a few seconds.

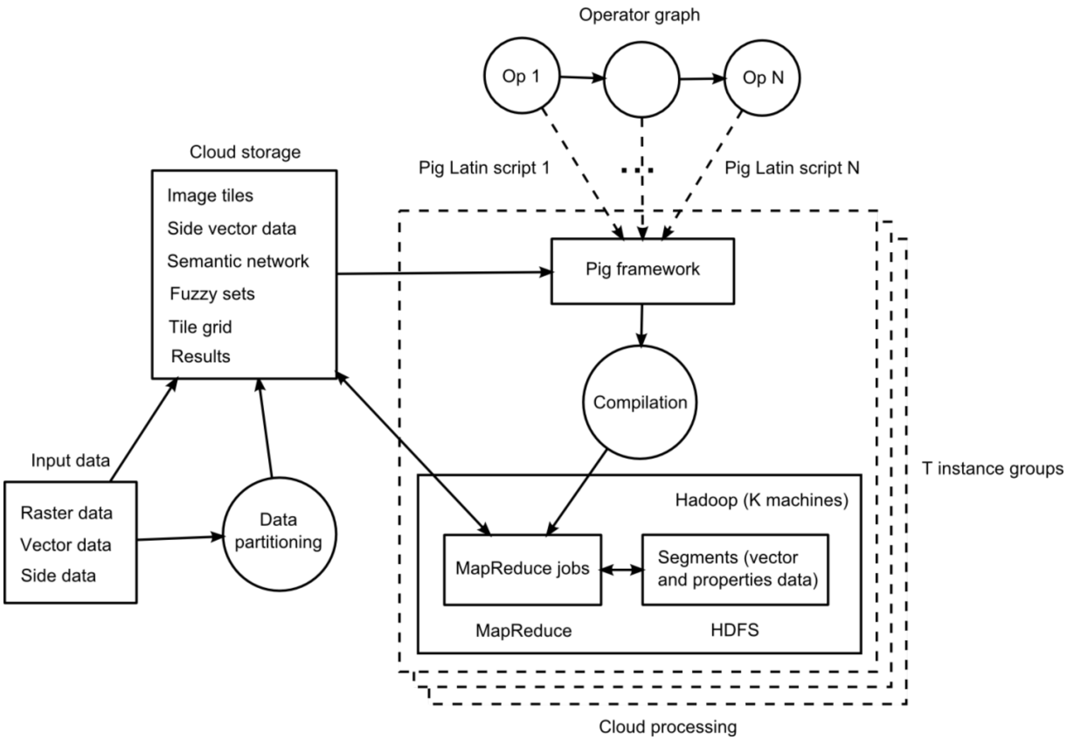


Figure 14: Cloud processing in detail.

The framework reads the Pig configuration parameters that are set in the script and is also responsible for loading the required libraries and shipping them to the machines that will execute the operator. Pig’s parser also optimizes the script in order to produce less complex MapReduce jobs.

When the jobs are executed, Hadoop MapReduce reads the operator’s input data from an auxiliary cloud storage service. Since one operator can be compiled into one or more MapReduce jobs, all temporary files are stored and read from HDFS. When all the jobs that compose the operator finish executing, the final result is stored back in the auxiliary cloud storage service.

If more than one instance group is available in the cloud, the execution is slightly different. If there are *T* instance groups available (Figure 7), instead of waiting for one operator to finish to execute the next one, ICP can execute up to *T* operators in parallel. This can speed up the processing depending on the operators that are being executed and the graph structure.

Although the compilation of the Pig Latin script in MapReduce jobs is automatic, both frameworks – Hadoop and Pig – provide dozens of configuration parameters that allow users to finely tune the behavior of both systems.

4.5. ICP's operations

It is possible to think of an operator as a set of high-level operations. Each operation is represented by a set of Pig Latin commands and UDFs that work together to perform a specific task. In this section, ICP's operation types are presented.

4.5.1. Spatial-blind operation

This is the distributed operation type that does not rely on spatial locality. In this operation type, the objects are processed in a distributed fashion according to their original distribution on HDFS. The computation is performed and the results are stored on the output objects' properties field. Examples of this operation type are morphological feature computation, spatial filtering and classification. These three examples are described below and a list with all spatial-blind operations implemented in ICP can be found in Appendix C.

4.5.1.1. Morphological features

Morphological features need only the geometry of an image object to be computed. Once the geometry is passed to the function, features like area and centroid can be computed in a distributed fashion.

4.5.1.2. Spatial filtering

In order to perform spatial operations efficiently, ICP relies on two structures, a global tile grid and on-demand R-tree-based spatial indexes. ICP relies on the JTS library for index and geometric operations (JTS15). Since data is partitioned in tiles, spatial operations can be processed in a *filter-and-refine* fashion (Figure 15). Although there is a spatial aspect to these operations, they are spatial-blind because they do not care about the location of the image object in the cloud.

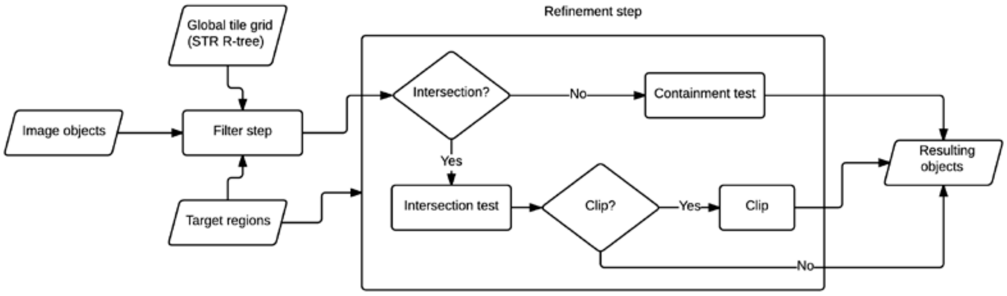


Figure 15: *filter-and-refine* workflow.

In the filter step, image objects that do not intersect the target regions can be quickly filtered out. The global tile grid is queried for each target region and a list with the intersecting tiles is created. After that, image objects that are not assigned to one of these tiles are discarded. Target regions are geographic regions defined by the user that represent the areas they want to process.

In the refinement step, the objects that passed the first coarse filter are checked against the precise geometric predicate, which is more computationally expensive. Only the objects that pass the second test are kept.

ICP provides two types of filters: *intersection* and *containment*. The first verifies which image objects *intersect* the given target regions while the second verifies the image objects that are *within* the target regions. After an intersection filter, ICP also provides a UDF that can *clip* the image objects with respect to the given target regions.

When spatial indexes are needed, R-trees are created with the Sort-Tile-Recursive (STR) algorithm (Rigaux02) which maximize space utilization and fairly minimize overlaps between nodes in comparison to basic R-Trees.

The general algorithm for this operation is the following:

1. Load tile grid.
2. Load target regions.
3. Compute a list with the tiles that intersect the target regions.
4. Filter objects based on the tile list.
5. Filter refinement (optional: clipping).

Load tile grid. The tile grid is loaded from an auxiliary cloud storage service. A spatial index is created.

Load target regions. The vector objects that represent the target regions are loaded from an auxiliary cloud storage service. A spatial index is created.

Compute a list with the tiles that intersect the target regions. The global tile grid index is queried for each target region. A list is created with all the tiles they intersect.

Filter objects based on the tile list. This step represents a coarse filter where the objects that are not assigned to the tiles in the list are filtered out.

Filter refinement. The remaining objects are filtered by checking the precise geometric predicate - *intersection* or *containment* – in relation to the target regions. The target regions' spatial index is used for fast lookup. In this step, the objects may also be clipped.

4.5.1.3. Classification

Classification functions like *Bayesian Classifier*, *Random Forest*, *Decision Tree* and *SVM* are also examples of this operation type. Their general algorithm is presented below:

1. Load sample data.
2. Train classifier.
3. Classify objects.

Load sample data. Sample data is loaded from an auxiliary cloud storage service.

Train classifier. The classifier is trained with the given sample data.

Classify objects. Objects are classified using the trained classifier.

4.5.2. Spatial-aware operation with replication

This operation type relies on spatial locality. Objects are grouped according to the geographic tile they are assigned to and boundary objects are replicated taking into account all the geographic tiles they intersect (*multiple assignment*). This is the general algorithm:

1. Intersecting tiles computation.
2. Object replication.
3. Object grouping by tile.
4. Partial results computation.
5. Object grouping by original tile.
6. Partial results combination.

Intersecting tiles computation. In this step, each object gets a list with all the tiles they intersect. The tile with the lowest label is assigned as *original tile*.

Object replication. Here, the objects are replicated according to each tile they intersect. The copies of the same object have the same *iiuuid*. All copies, except the original object, have a special field called *iirep*, which allows the system to remove them when they are no longer necessary.

Object grouping by tile. In this step, all image objects (including replicated ones) are grouped by tile.

Partial results computation. In this step, the corresponding image tile is loaded. The computation is performed considering only the intersection between the object and the geographic tile. Partial results are stored in the output objects.

Object grouping by original tile. In this step, all objects are grouped by their *original tile*.

Partial results reduction. In the final step, the partial results of the objects with the same *iiuuid* are combined and all replicated objects are discarded.

Examples of this operation type are the computation of *spectral and topological features*, and *spatial conflict resolution*. These three operations are explained in detail below. Since the general operation algorithms are similar, the discussions will focus on the *partial results computation* and *partial results reduction* steps.

4.5.2.1.
Spectral Features

In this operation type, after the image objects are grouped by tile, a UDF named *Partial Spectral Features* is called (Figure 16). This UDF receives the spectral features to be computed as an input parameter. The function loads the corresponding image tile from an auxiliary cloud storage service. Once the image tile is loaded, the UDF computes the partial values for all the required spectral features, for each image object. If the image object intersects other tiles, only the intersection with the current tile is considered. When all the partial values are computed, a new field called *spectral features* is added to the object's property field, where the partial values are written.

When the objects are regrouped by *original tile*, another UDF is called: *Final Spectral Features* (Figure 16). In this UDF, all objects with the same *iiuuid* are grouped. These objects represent the copies of the same object that have been created in the replication step. Once the copies are gathered together, partial values are combined and the final feature value is stored in the object's property field of the original object. After that, the original object is written in the output and the copies are deleted.

Up to now, the implemented spectral features are *mean*, *maximum pixel value*, *minimum pixel value*, *band ratio*, *brightness*, *band mean arithmetic*, *amplitude value* and *standard deviation* (feature definitions can be found in Appendix E).

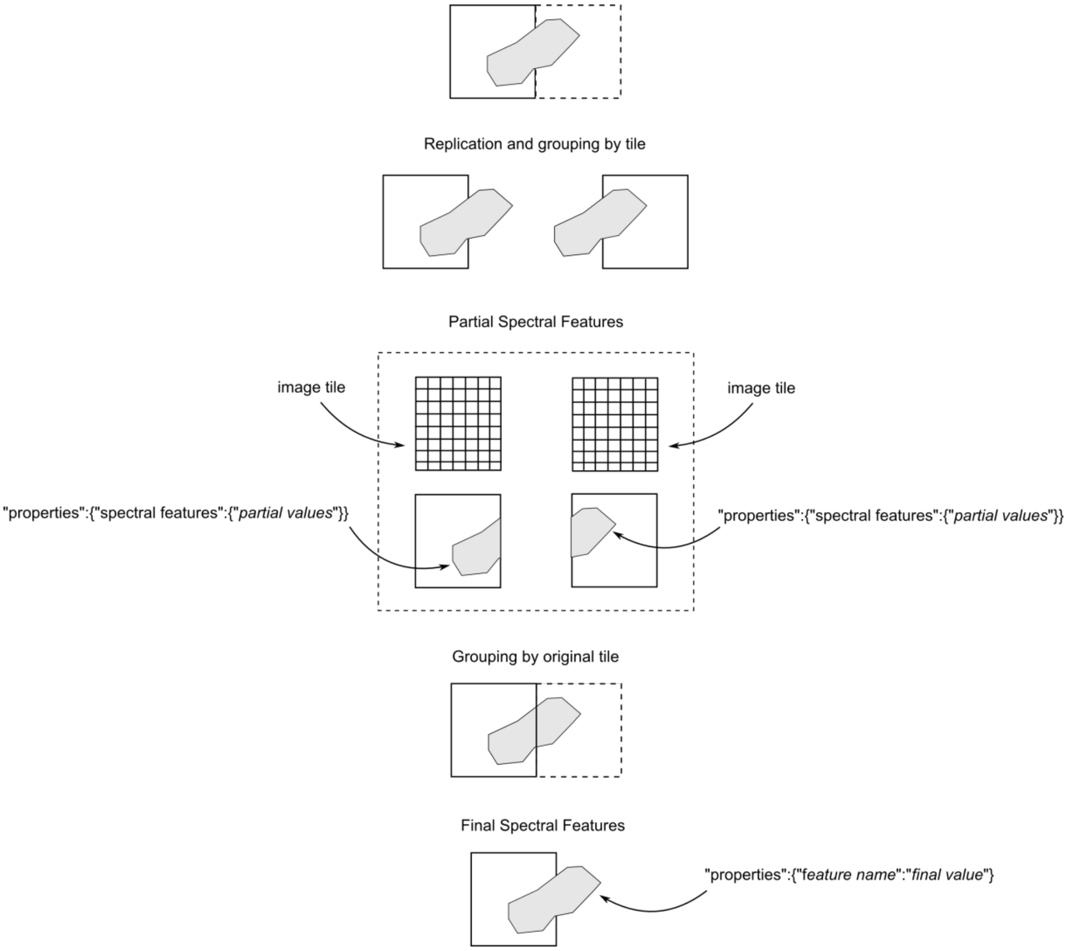

Replication and grouping by tile
Partial Spectral Features
image tile
image tile
"properties":{"spectral features":{"partial values"}}
"properties":{"spectral features":{"partial values"}}
Grouping by original tile
Final Spectral Features
"properties":{"feature name":"final value"}

Figure 16: spectral features operation.

4.5.2.2. Spatial Resolve

Like InterIMAGE, ICP allows different segmentation algorithms (or different segmentation parameters) to be considered in the same interpretation model. This characteristic leads to spatial conflicts when, for the same geographic region, there is more than one object hypothesis.

Because of its architecture, even when the segmentations are the same, InterIMAGE creates a copy of the object hypothesis for each class it was assigned to. For example, if there is a specific segmentation for vegetation classes – and there are two classes, trees and grass – the same image object is duplicated, one for each class. The object copies, later on, undergo a procedure called *spatial conflict resolution*.

In ICP, there are three approaches for spatial conflict resolution. In the first one, called *fuzzy spatial resolution* (Figure 17), when the same image object has to be classified in two different classes, a special property is created for that object called *classification*. This property, in fact, keeps a list of class names and membership values. Thus, it is possible to have the same image object classified as grass with membership 0.4 and classified as trees with membership 0.8. This greatly avoids data replication and simplifies the classification process. In this case, when the final classification is calculated, the system basically takes the class with the highest membership value. This is an example of a spatial-blind operation (section 4.5.1).

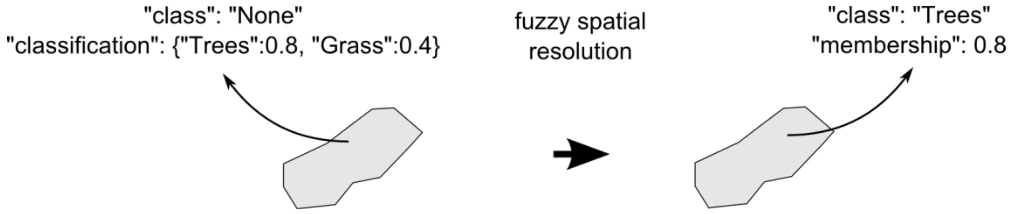


Figure 17: fuzzy spatial resolution.

When segments produced by the same segmentation are processed in different operators, it is not possible to apply this approach, since the system creates one object hypothesis for each operator. In this case, ICP applies a *simple spatial resolution* (Figure 18). As hypotheses that cover the same geographic region have the same *iiuuid*, the system groups the hypotheses by *iiuuid* and takes the hypothesis with the higher membership. The other copies are discarded. This approach is simpler than the one implemented in InterIMAGE, that uses in this case, the same approach used for segments produced by different segmentations. This is an example of a spatial-aware operation without replication (section 4.5.3).

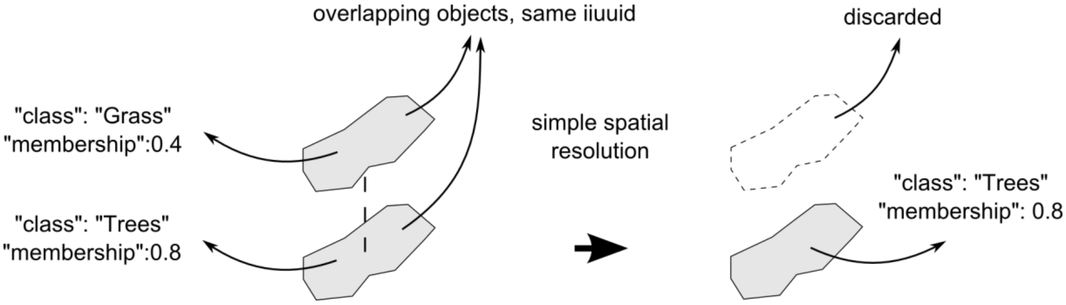


Figure 18: simple spatial resolution.

When there is a spatial conflict between objects originated in different segmentations, ICP follows the same approach as InterIMAGE, called *spatial conflict resolution* (Figure 19). In this approach, image objects are ordered ascendingly by their membership values and inserted in a list. After that, the objects in this list are *rasterized* and written on an auxiliary label image (the rasterization method selects the pixels of which centers lie within the image object). The procedure continues until all objects are written. In the end, the image objects with higher membership values remain in the label image while the others are overwritten during the process. After that, the system applies a *vectorization* on the label image and collects the winning objects and writes them in the output.

During a spatial conflict resolution, some objects may be completely discarded. The remaining objects may keep their original shape, have their areas reduced or even divided in two or more objects. It is usual to reset all objects' non-basic properties after this process as previously computed spectral and morphological features may have changed. This is a spatial-aware operation *with replication* and its distributed strategy will be explained below.

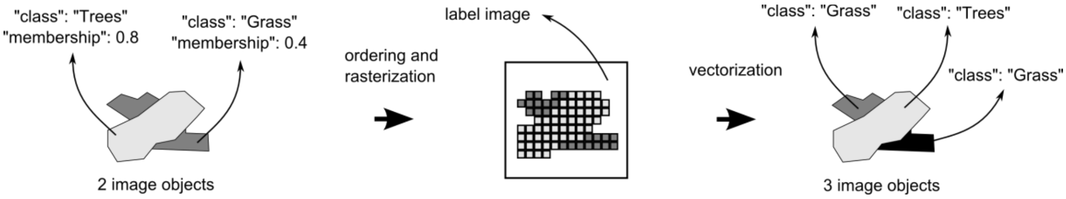


Figure 19: spatial conflict resolution.

When the objects belonging to the same tile are grouped, this operation type calls a UDF named *Partial Spatial Resolve* (Figure 20). As seen before, this UDF creates a list with all the objects in ascending order of membership value and iterates through this list rasterizing these objects and writing them on an auxiliary label image. This process continues until all objects are written. In the end, the objects with higher membership values remain in the label image while the others are overwritten during the process. Again, if an object intersects more than one tile, only the intersection with the current tile is considered. Finally, the function applies a vectorization on the label image and builds the image objects again.

When the objects are regrouped by their original tile, another UDF is called: *Final Spatial Resolve* (Figure 20). This UDF groups the objects with the same *iiuuid*'s and merges them creating a new object. If there are disjoint parts, the function creates new objects for them. In the end, all the objects created in this function receive new *iiuuid*'s and have their properties reset.

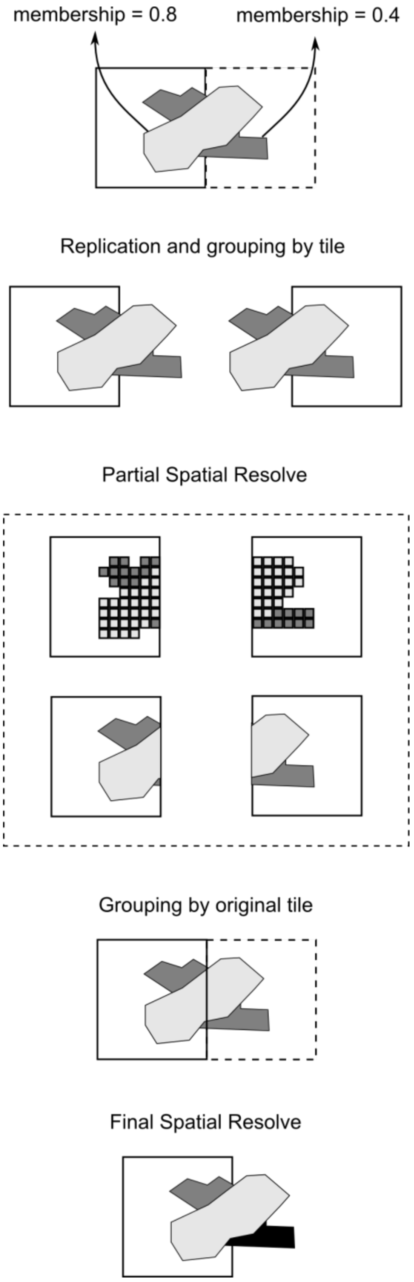


Figure 20: spatial resolve operation.

4.5.3. Spatial-aware operation without replication

This operation type relies on spatial locality. In this case, objects are grouped according to the geographic tile they are assigned to. No replication is needed (*multiple matching*). This is the general algorithm:

1. Object grouping by original tile.
2. Operation computation.

Object grouping by *original tile*. In this step, all objects are grouped by *original tile*.

Operation computation. The computation is done and the results are stored on the output objects' properties map.

One example of this operation type is the function *Simple Spatial Resolution* (section 4.5.2.2). This function resolves spatial conflicts between object hypotheses that cover the same geographic region, i.e., objects generated by the same segmentation algorithm. In the distributed strategy, image objects are grouped by *original tile*. The objects with the same *iiuuid* are compared and the winner hypothesis is the one with the highest membership value. The other object hypotheses for the same *iiuuid* are discarded.

4.5.3.1. Topological Features

In the partial results computation step, this operation type calls a UDF called *Partial Topological Features* (Figure 21). This UDF loads the global tile grid and creates a STR R-tree-based spatial index. After that, the function also creates a spatial index on the input image objects.

For each image object (target), the function verifies the intersecting objects (neighboring objects). Then, it computes a list with all the intersecting tiles of each target-neighbor object pair. The partial topological features are only computed if both objects have the same original label (lowest tile label). This precludes a topological feature from being computed more than once for objects that intersect more than one tile. When all the partial values are computed, a new field called *topological features* is added to the object's property field, where the partial values are written.

In the partial results reduction step, another UDF is called: *Final Topological Features* (Figure 21). In this UDF, all objects with the same *iiuuid* are grouped. These objects represent the copies of the same object that have been created in the replication step. Once the copies are grouped together, partial values are combined and the final feature value is stored in the object's property field of the original object. After that, the original object is written in the output and the copies are deleted.

The implemented topological features are: *number of*, *border to*, *relative border to*, *area of* and *relative area of* (feature definitions can be found in Appendix E).

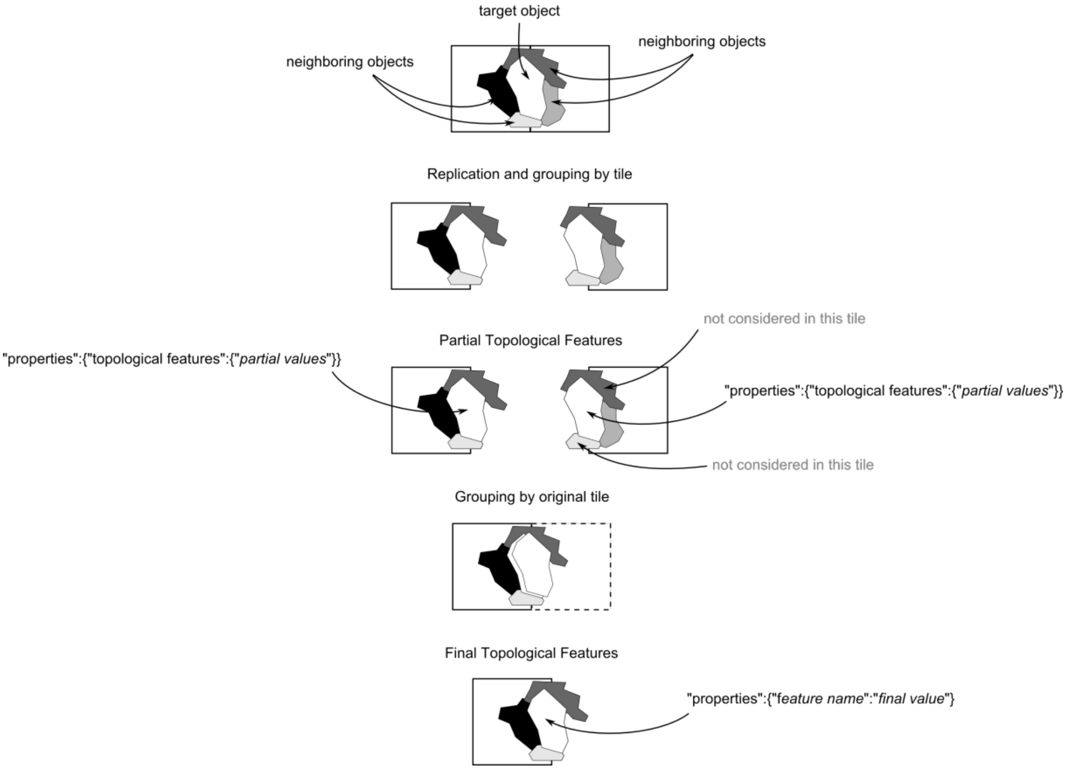

target object
neighboring objects
neighboring objects
Replication and grouping by tile
not considered in this tile
Partial Topological Features
"properties":{"topological features":{"*partial values*"}}
"properties":{"topological features":{"*partial values*"}}
not considered in this tile
Grouping by original tile
Final Topological Features
"properties":{"*feature name*":"*final value*"}

Figure 21: topological features operation.

4.5.4. Recursive operation

In this operation type, a recursive procedure takes place. Considering the hash values that represent the geographic tiles (as seen in section 4.3.2), it is possible to go up in the tile hierarchy by just dropping a letter from the *tile label*.

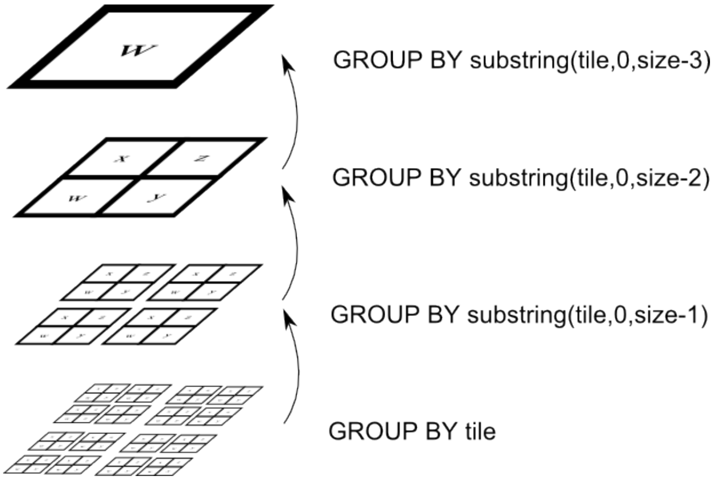

W
x
z
w
y
GROUP BY substring(tile,0,size-3)
GROUP BY substring(tile,0,size-2)
GROUP BY substring(tile,0,size-1)
GROUP BY tile

Figure 22: recursive operation.

The operation (Figure 22) starts from the lowest level of the quad-tree and performs the same computation on every level until it reaches the last level. The last level is not necessarily the top level of the quadtree, but the level that is large enough to geographically contain the input image(s) (section 4.3.2).

For example, if an object has the following tile label, ***wxyz***, it would be grouped in the first step with all the other objects assigned to the same tile. In the second step, this object would be grouped with all objects belonging to the super-tile ***wxy***. In the last two steps, it would be grouped with the objects that belong to super-tiles ***wx*** and ***w***, respectively. In the last step, all objects would be grouped in the same machine and the processing would stop.

This approach has limitations as the processing reaches the highest levels of the quadtree. In these levels, parallelism is hampered because there are fewer tiles to work with. Another issue in the highest levels is related to memory problems, as many objects may be grouped in the same machine.

There are two approaches to mitigate these limitations. In the first approach the computation should progressively reduce the number of objects in each step.

One example of this approach is an operation that merges neighboring objects that belong to the same class.

This operation aims at merging the image objects of the same class (or classes) that are connected. The classes to be merged are passed as an input parameter. In the first step, the objects are grouped according to the lowest level of the quadtree. The operation recursively considers the neighboring objects of each object and merges the ones that belong to the same class. When the step is finished, the number of objects is reduced since many objects were merged. The operation, then, regroups the image objects according to the upper level of the quadtree and repeats the procedure. The process continues until the last level is reached.

If the first approach still presents memory problems or the nature of the computation does not allow the reduction of the number of objects in each step, the computation should be such that part of the objects (that will no longer be needed) can be saved to an auxiliary cloud storage service in each step. This approach would process progressively fewer image objects at each step. Another option could be the combination of both approaches.

4.5.5. Hierarchical features

This operation type refers to a different kind of hierarchy. While the previous operation type refers to a recursive computation that is performed on each level of the geographic tile quad-tree representation, this one refers to a spatial computation that is carried out on objects that have a hierarchical relation. It is a special case of a spatial-aware operation that does not rely on geographic tile information.

In this operation type, an object that is in a lower position in the object hierarchy carry information about its parent object, i.e. the object that contains that object. An example of when this kind of hierarchical relation is created is when the spatial filtering with clipping is executed (section 4.5.1.2). This operation, besides clipping the objects according to a number of target regions, also stores on each object the *iiuuid* of the target region object (parent object) that contains it. This operation type relies on this sort of information to compute hierarchical features. The general algorithm is shown below:

1. Object co-grouping by the parent object’s *iiuuid*.
2. Operation computation.

Operation co-grouping by the parent object’s *iiuuid*. In this step, the parent object and its child objects are grouped by the parent object’s *iiuuid*.

Operation computation. Once the parent object and its child image objects are grouped, hierarchical features can be computed.

In this operation, after the parent and child objects are grouped, hierarchical features are computed for the parent object based on its child objects. For example, this operation can compute the number of child objects that belong to a specific class. This operation can also compute aggregated features based on the properties of the child objects. An example would be the computation of the mean rectangularity or the total area of child objects that belong to a certain class.

There is an issue related to this operation that will be better exemplified in the results of the experiments (section 5.5.1). If the parent object is very large, this operation can incur in memory issues, since a potential high number of objects will be grouped in the same machine.

Up to now, the implemented aggregation functions are *count*, *max*, *min*, *mean* and *sum*. These functions can be applied to any feature child objects have in common (feature definitions can be found in Appendix G).

5
Results and experimental analysis

This chapter presents the software prototype and the experiments performed to validate the proposed architecture.

5.1.
Software prototype

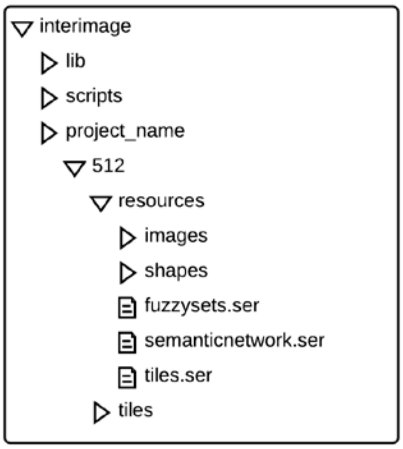
To implement the architecture proposed in this work a software prototype was developed. The prototype was coded in Java and contains five packages described below:

- **interimage-core** – this package implements core functionalities like communication with the cloud, data management and interpretation control.
- **interimage-geometry** – this package implements vector data functionalities such as geometric and topological operations.
- **interimage-data** – this package implements raster data functionalities like the computation of spectral features.
- **interimage-datamining** – this package implements classification functionalities such as fuzzy rules and machine learning algorithms.
- **interimage-operators** – this package implements basic low-level image processing operators that are released with the platform.
- **interimage-common** – this package implements basic Java classes and UDFs that are used by the other packages.

There is no graphical user interface up to now. To facilitate the creation of an interpretation project, some mechanisms were developed to import an interpretation project created in the previous version of InterIMAGE. It is possible to define the input images, input shapefiles, semantic network and decision rules on InterIMAGE and then import the project to ICP. This process is not yet completely automated but still can save significant time.

5.1.1. Data staging

Figure 23 shows ICP’s folder structure. Before executing an interpretation model, ICP uploads input and configuration data to a local cluster or a cloud service. These data are stored in a folder called *interimage*. This folder has a simple organization. It contains two folders: *lib* and *scripts*.



```
▽ interimage
  ▷ lib
  ▷ scripts
  ▷ project_name
    ▽ 512
      ▽ resources
        ▷ images
        ▷ shapes
        fuzzysets.ser
        semanticnetwork.ser
        tiles.ser
      ▷ tiles
```

Figure 23: ICP’s folder structure.

The *lib* folder contains all the libraries necessary to execute ICP. The platform has six libraries: *core*, *common*, *data*, *geometry*, *datamining* and *operators*. The core library is not shipped to the cluster, but stays on the client machine. Besides these libraries, the system is also shipped with several external libraries that are responsible for image and geometric operations.

The *scripts* folder contains a special Pig Latin script that defines the UDFs that are available in the platform. When a new UDF is added to the system, this file has to be updated in order to make it available to the user.

The folder *interimage* contains also the project folders. A project folder is created along with the project and contains all the project files. The project folder has the following structure:

The first level is defined by the tile size. This way it is possible to have different tile sizes coexisting in the same project folder. Below there are two folders: resources and tiles. The *resources* folder contains resources like fuzzy sets, semantic networks and the tile grid. It is in this folder that image tiles (*folder images*) and input vector objects (*folder shapes*) are stored. The *tiles* folder holds information about the geographic tiles and will be used mainly as input for segmentation operators.

5.2. Dataset

The land cover and land use classification models used in the experiments were proposed by Novack (Novack09; Novack10). For his experiments, Novack had originally an area delimited by the following geographic coordinates: S 23º 38’ 33”, W 46º 24’ 47”, S 23º 35’ 58” and W 46º 41’ 35”. The area has 25 km^2 and comprises part of Vila Sônia, Vila Andrade, Morumbi, Santo Amaro and Itaim Bibi districts in the city of São Paulo.

Since his goal was to run the same classification models on eCognition and InterIMAGE, he had to cut the original QuickBird image into smaller subsets because both systems could not process the whole scene which had 8,000x8,0000 pixels. For eCognition, he selected subsets of 4,000x4,000 pixels. For InterIMAGE, the limit was even smaller: subsets of 1,500x1,500 pixels. Novack used InterIMAGE 0.092 and Definiens Developer 7.0 (eCognition was called Definiens in 2009) in his experiments.

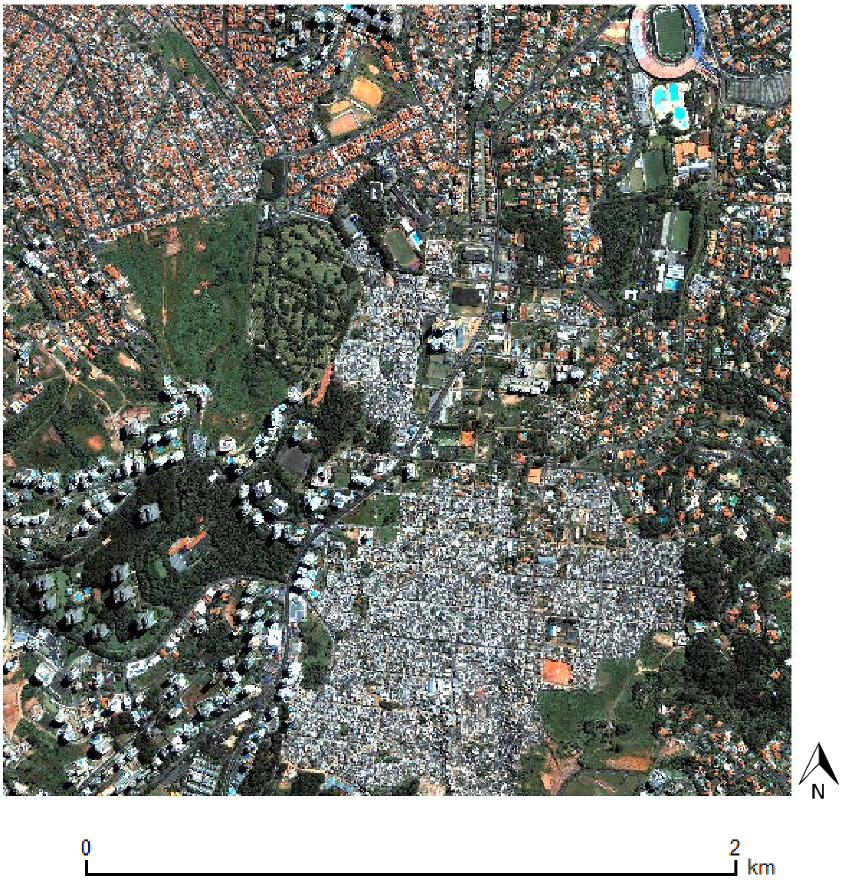


Figure 24: image subset 4K.

For the present work, one of the subsets of 4,000x4,000 (4K) pixels was selected (Figure 24). It is a QuickBird image with 0.61m of spatial resolution and 4 bands (blue, green, red and infrared). As ICP has as its main goal to process very large datasets, the image was replicated to reach image sizes as large as 8,000x8,000 (8K), 16,000x16,000 (16K) and 32,000x32,000 (32K) pixels. The objective of this configuration is to assess if ICP is able to process much larger images than the previous version of InterIMAGE and the commercial suite eCognition can handle.

Novack (Novack09) used machine learning algorithms for the calibration of the segmentation parameters. As the calibration was performed using the segmentation program based on the algorithm proposed by Baatz and Schäpe (Baatz00), present in the original version of InterIMAGE, it was possible to use the software to segment the image using the same parameters. Two segmentations were performed. One for vegetated areas (PS1), and one for roofs and swimming pools (PS2) (Table 1).

Table 1: parameter sets applied for vegetated areas (PS1), and for roofs and swimming pools (PS2).

Parameter set	Scale	Shape	Compactness	Band weights
PS1	21	0.25	0.61	0.24,0.07,0.42,0.26
PS2	40	0.7	0.8	1.0,1.0,1.0,0.0

To segment shadow areas, a thresholding procedure was carried out on InteIMAGE, selecting the pixels of a brightness image (sum of the four spectral bands divided by four) with values below 20.

For the classification of red objects (ceramic tile roofs and bare soil), Novack used GLCM-based texture features (Haralick73) on eCognition. On InterIMAGE, an auxiliary shapefile with the class Bare Soil was imported because the texture features available on InterIMAGE were incompatible with the same features available on eCognition. The same procedure was adopted in our experiments because ICP does not yet provide texture-based features.

As the image was replicated to simulate larger datasets, the segmentations and auxiliary shapefiles had to be replicated as well. Table 2 describes the images, segment sets and auxiliary shapefiles used in the experiments.

Table 2: image sizes, number of objects and file sizes of input segment sets and auxiliary shapefiles.

	4K	8K	16K	32K
Image size (pixels / size)	4,001x4,000	8,002x8,000	16,004x16,000	32,008x32,000
	61 MB	244 MB	976 MB	3.81 GB
segments for vegetated areas (# / size)	62,405	249,620	998,480	3,993,920
	160 MB	660 MB	2.58 GB	10,0 GB
segments for roofs and swimming pools (# / size)	50,890	203,560	814,240	3,256,960
	193 MB	787 MB	3.07 GB	12,0 GB
segments for shadows (# / size)	10,168	40,672	162,688	650,752
	14 MB	58.8 MB	235 MB	897 MB
polygons of bare soil (# / size)	794	3,176	12,704	50,816
	991 KB	3.87 MB	15.5 MB	61.7 MB
polygons of blocks (# / size)	204	816	3,264	13,056
	4.96 MB	19.8 MB	79.5 MB	316 MB

5.3.
Cloud environment

Although the cloud service selected for the experiments was Amazon Web Services (AWS), the system can be easily extended to work with any cloud service (e.g. Microsoft Azure). The Java libraries, input and auxiliary data were stored on an Amazon S3 bucket, following the scheme presented in section 5.1.1.

Amazon EMR (Elastic MapReduce) was used for processing. With this service it is possible to select machines that have Hadoop and Pig already installed and the characteristics of the machines can be easily customized. The available machine types range from basic computers to high-performance nodes. In our experiments, we used *m1.xlarge* machines. These are 64-bit computers, with 4 physical cores (8 logical cores), 15 GB of RAM and 4 disks of 420 GB. Hadoop and Pig configuration parameters can also be easily changed.

Hadoop version was 2.4 with the following parameters:

- *Replication factor*: 3
- *Block size*: 64 MB
- *Speculative execution*: off
- *Hadoop compression*: Snappy
- *Java heap size*: 1.5 GB

Pig version was 0.12. File split combination and temporary file compression were turned on.

5.4.
Land cover classification

This first set of experiments consisted of three steps. Firstly, the land cover interpretation model was executed using a single instance group where absolute processing times and speedups were evaluated. Later on, in order to investigate the graph-level parallelism, the same investigation was carried out for multiple instance groups.

To build the land cover classification model, Novack (Novack09) defined the semantic network shown below (Figure 25). The classes and their hierarchical structure were defined after visual inspection of the image and a visit to the site. In the end, eleven classes were created: *Grass, Trees, Ceramic tile roofs, Bare soil, Dark asbestos tile roofs or new asphalt, Grey asbestos tile roofs or asphalt, Clear asbestos tile roofs or concrete, Bright roofs, Blue-colored roofs, Swimming pools* and *Shadow*. Class definitions can be found in his dissertation (Novack09).

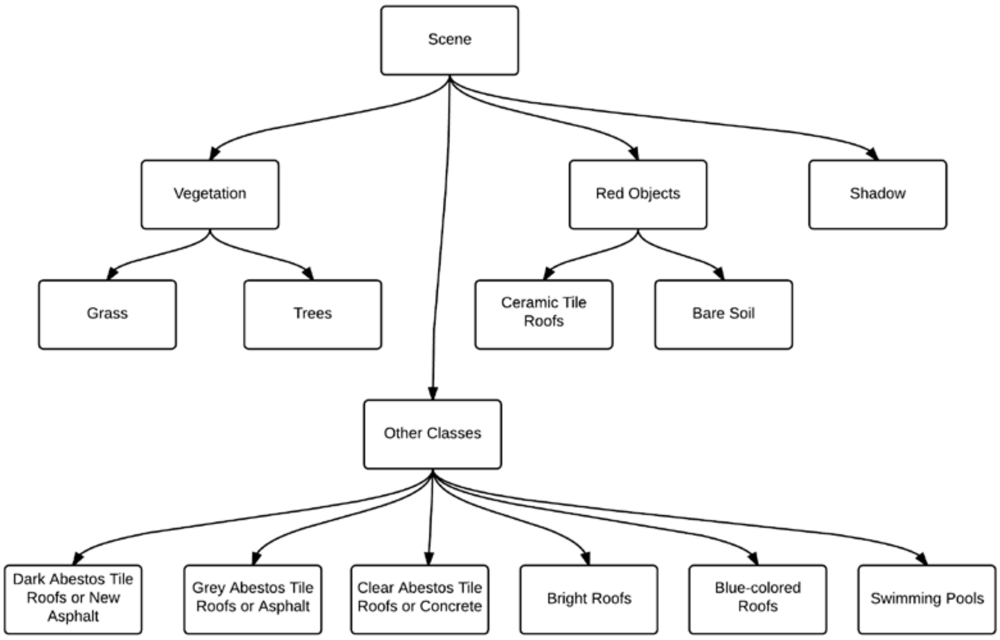


Figure 25: land cover semantic network.

After that, using some sample segments and a decision tree algorithm, Novack determined the most relevant attributes and respective thresholds to classify the segments according to the classes defined previously. The decision trees given by the classification algorithm helped him to define the decision rules in InterIMAGE and eCognition.

In order to emulate the same experiment, the decision rules for each classification used in InterIMAGE were translated to Pig Latin and became operators in ICP. The semantic network was also reproduced. Figure 26 presents the final operator graph.

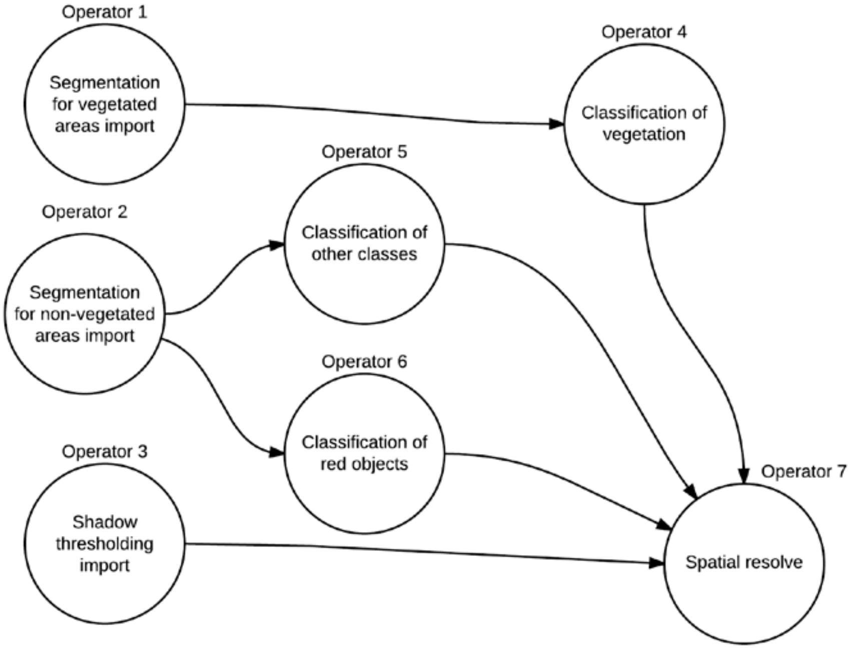

Operator 1
Segmentation for vegetated areas import
Operator 4
Classification of vegetation
Operator 5
Classification of other classes
Operator 2
Segmentation for non-vegetated areas import
Operator 6
Classification of red objects
Operator 3
Shadow thresholding import
Operator 7
Spatial resolve

Figure 26: land cover operator graph.

- *Operator 1* imports the segments obtained from the segmentation for vegetated areas performed on InterIMAGE.
- *Operator 2* imports the segments obtained from the segmentation for non-vegetated areas performed on InterIMAGE.
- *Operator 3* imports the shadow segments obtained with the thresholding procedure on InterIMAGE.
- *Operator 4* classifies Grass and Trees segments according to the decision rules translated from the original project.
- *Operator 5* classifies several roofs and Swimming pools segments according to the decision rules translated from the original project.
- *Operator 6* classifies Ceramic Tile Roof and Bare Soil segments according to the decision rules translated from the original project.
- *Operator 7* resolves spatial conflicts and generates the final classification.

The Pig Latin scripts of each operator are analyzed by Pig and compiled automatically into MapReduce jobs. The MR jobs of each operator are described below:

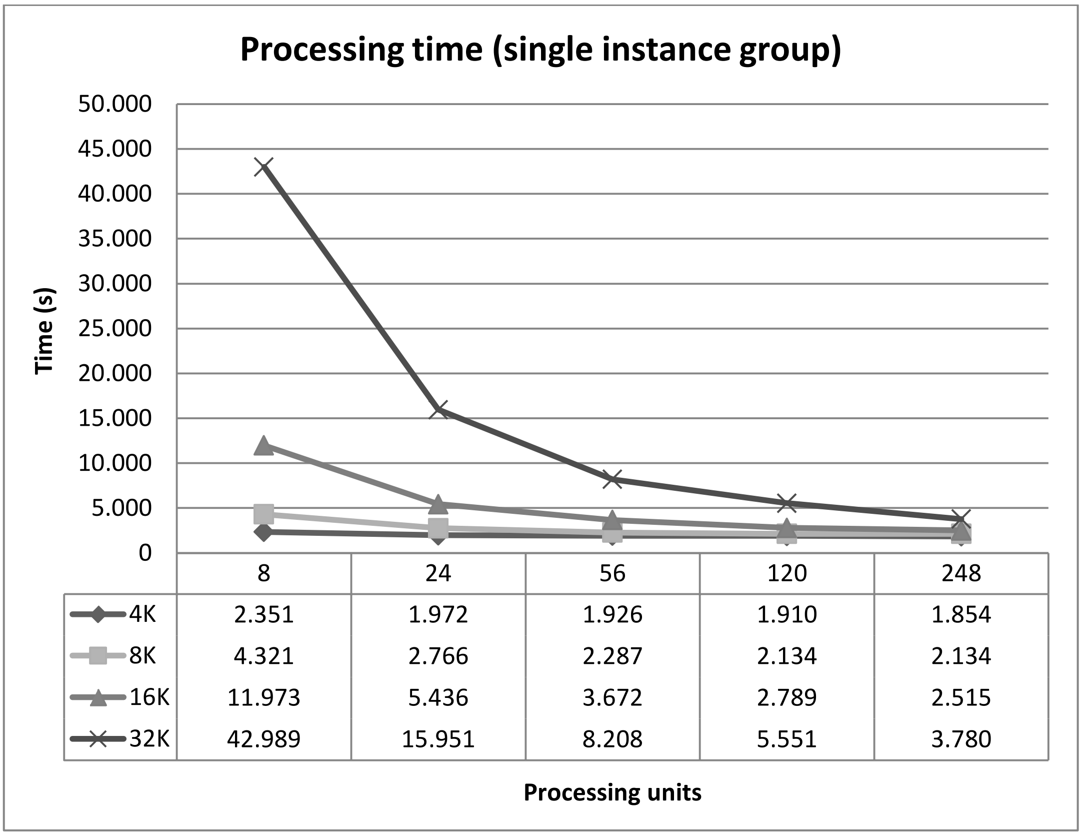
Table 3: MapReduce jobs for the land cover interpretation model.

	Job	Job type
Operator 1	#1	Map only
Operator 2	#2	Map only
Operator 3	#3	Map only
Operator 4	#4	Map-Reduce
	#5	Map-Reduce
Operator 5	#6	Map-Reduce
	#7	Map-Reduce
Operator 6	#8	Map-Reduce
	#9	Map-Reduce
	#10	Map-Reduce
	#11	Map-Reduce
Operator 7	#12	Map-Reduce
	#13	Map-Reduce

The Pig Latin script of each operator is presented in Appendix H. The whole interpretation model takes 13 MR jobs to complete (Table 3). The first three jobs have only the Map phase, while the others have both, Map and Reduce phases.

5.4.1. Single instance group

In this step, all images (4K, 8K, 16K and 32K) were executed in a single instance group with 2, 4, 8, 16 and 32 machines. Each experiment was executed 5 times and the average processing time was taken. As *m1.xlarge* instances have 8 logical cores and one machine is the MapReduce master, the corresponding processing units are 8, 24, 56, 120 and 248. The results are shown in Figure 27.



	8	24	56	120	248
4K	2.351	1.972	1.926	1.910	1.854
8K	4.321	2.766	2.287	2.134	2.134
16K	11.973	5.436	3.672	2.789	2.515
32K	42.989	15.951	8.208	5.551	3.780

Figure 27: processing time for the experiments using a single instance group.

The graph shows that the processing time decreases as the number of processing units increases. The reduction rate depends on the image size and the number of processing units. As the image size increases, the operators can benefit more from the available processing units achieving a higher parallelism. For smaller images, as the instance group size increases, some processing units become idle and parallelism is not improved, producing an almost steady processing time.

One can understand the causes of this behavior by looking at the distributed processing flow. Image 4K, for example, has 62,405 input objects for Operator 1. This number of objects corresponds to 160 MB on disk (Table 2). As MapReduce’s block size is set to 64 MB, this operator executes only 3 map tasks. This means that even for the smallest instance group, with only 2 machines, the 8 available processing units are already underutilized for this operator. For larger images, processing units start being underutilized for larger instance groups.

For image 8K, it starts on the 4-machine instance group (24 processing units). As the input size on disk is equal to 660 MB, 11 map tasks are executed for

Operator 1. For images 16K and 32K, 42 and 161 map tasks for Operator 1 are launched, respectively. Therefore, they can benefit from all processing units until the 8-machine instance group for image 16K, and until the 16-machine group for image 32K.

The same reasoning applies for the map phases of all operators because the parallelization of map tasks is only defined by the size of the input. The larger the image (number of image objects), the more operators can benefit from the available processing units, achieving higher parallelism. In fact, MapReduce does not perform well with several small files (Aji13). In order to mitigate this problem, Pig has a mechanism called *split combination* that joins small input files until they reach the block size. This feature is turned on by default in ICP.

Map parallelism is highly sensitive to the block size. Although block sizes of 64 MB and 128 MB are the two most common configurations, it is possible to change the block size in the Hadoop configuration file. Higher block sizes produce fewer tasks and minimize the parallelism possibilities. However, having a block size too small may make the overhead of managing the splits and of map task creation begin to dominate the total job execution time (White12). This tradeoff must be considered before changing this feature.

For the reduce phase, the logic is different. The reduce phase depends on the number of keys that are sent to reduce tasks. In ICP, reduce phases are forced by GROUP BY Pig Latin commands that group image objects together based on the geographic tile they belong to. So, in order to discuss reduce parallelism, let us take a look at Table 4 that shows the number of geographic tiles for each image.

Table 4: number of tiles for each image.

	4K	8K	16K	32K
# geographic tiles	81	256	1,024	3,969

For image 4K, for example, there are 81 geographic tiles. It means that until the 8-machine instance group (56 processing units) all processing units are used. For the 16- and 32-machine instance groups, the processing units are underutilized. This reasoning is valid for all operators that have a reduce phase. In this case, the amount of work each reduce task will perform depends on two things. Firstly, on the number of objects assigned to each tile which varies from one operator to another; and, secondly, on the number of tiles that are assigned to each reducer. The fewer machines available, the more tiles are processed per reducer. All the other images use all the available processing units in the reduce phase for all instance group sizes, considering that the largest instance group has 32 machines and 248 processing units.

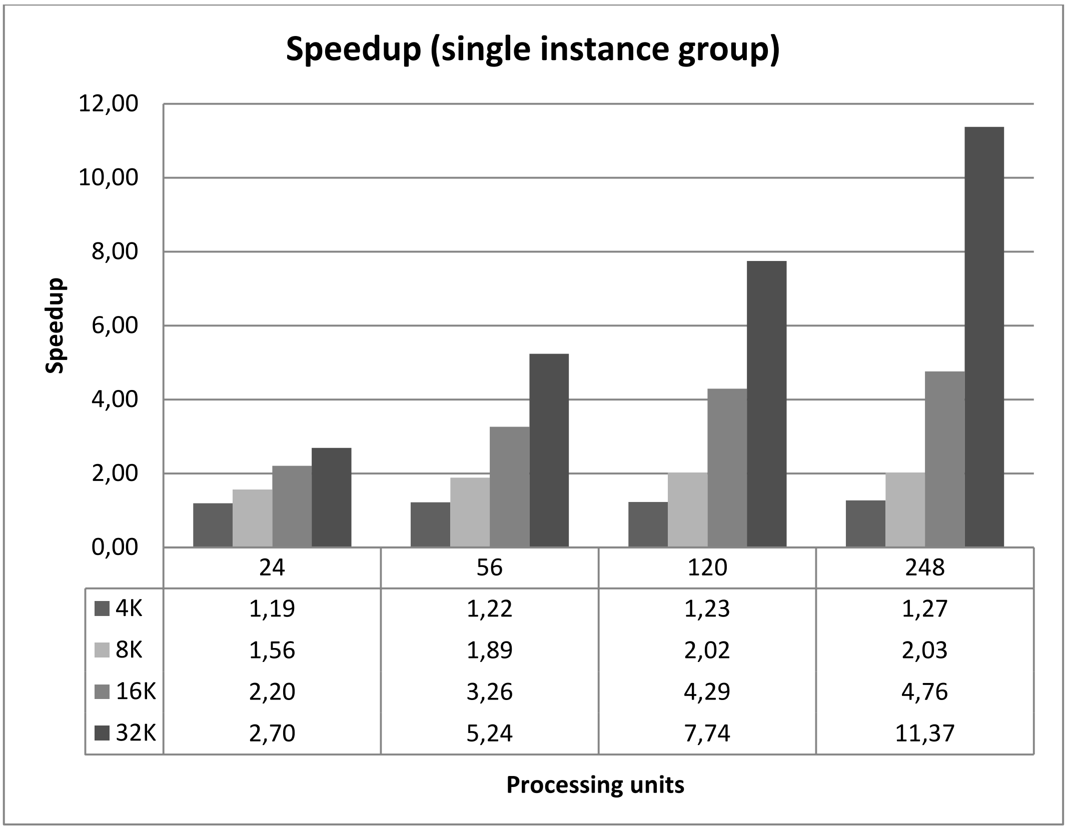
MapReduce uses by default a hash function to determine the reduce tasks that will process each reduce key. As ICP uses strings to represent geographic tiles, there was an unbalance in the number of keys sent to each reducer. This unbalance was imperceptible when the number of keys was much larger than the number of available reduce slots, but was quite strong when these numbers got closer to each other. This problem was solved by using a numeric representation of each tile (considering only the tiles that actually intersect the image). This way, the default *Partitioner* of MapReduce was overwritten with a simple *modulo* operation which produced very good results.

As stated before, reduce parallelism is sensitive to the number of geographic tiles. Thus, the size of the geographic tile (in meters) and, ultimately, the size of the image tile (in pixels) plays a key role in defining the parallelism level of the reduce phase. Larger tiles produce fewer reduce tasks and minimize parallelism possibilities. However, having smaller tiles means grouping more objects in the same machine, which may incur in memory issues. This tradeoff must be considered before defining the tile size.

The experiments show that MapReduce provides its maximum performance for larger images since small images underutilize compute resources. That is why Figure 27 shows a steeper curve for images 16K and 32K. As the number of processing units increases, the operator can take advantage of more processing units, improving the performance.

In absolute numbers, the whole interpretation for image 4K took 2,351s (~39m) with an 8-machine instance group and 1,854s (~31m) with a 32-machine instance group. For image 32K, the total processing time was 42,989s (~12h) for a 2-machine instance group and 3,780 (~1h) for a 32-machine instance group. These results show that it is possible in ICP to process an image of 32,000x32,000 pixels with only two machines without incurring in memory issues.

Taking the 2-machine instance group (8 processing units) as a basis for comparison, Figure 28 shows the speedups for different instance group sizes.



	24	56	120	248
■ 4K	1,19	1,22	1,23	1,27
■ 8K	1,56	1,89	2,02	2,03
■ 16K	2,20	3,26	4,29	4,76
■ 32K	2,70	5,24	7,74	11,37

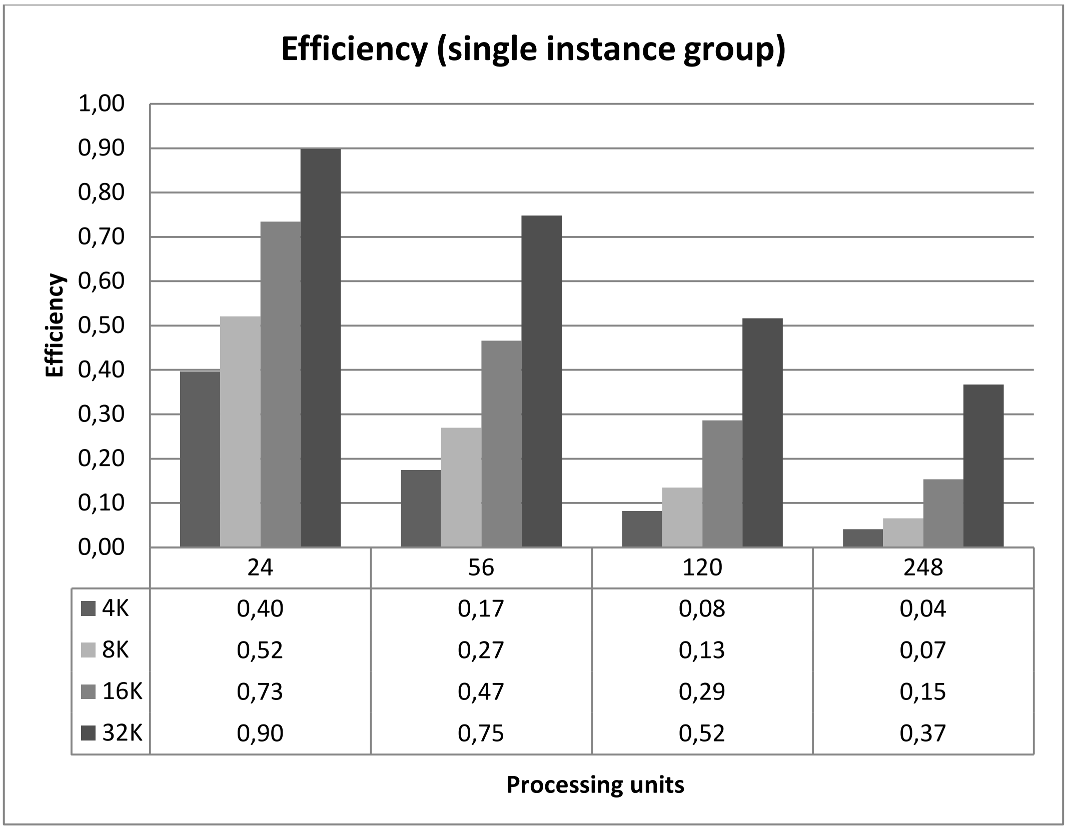
Figure 28: speedups of different group sizes in comparison to a 2-machine group size (8 processing units).

For the reasons presented before, the highest speedups were obtained for the largest images since they benefit more from the available processing units. For the 4-machine instance group (24 processing units) the image 4K achieved a speedup of 1.19 while the image 32K achieved a speedup of 2.70. The latter is a very interesting result if we consider that the number of processing cores was tripled.

For the 8-machine, 16-machine and 32-machine instance groups the speedups grew, but not linearly with the image size. For the images 4K and 8K, the main reason is that at a certain point they start to not use all the processing units, underutilizing the available processing power. For the other images, 16K and 32K, a reason is the network communication between the map and reduce phases (sort and shuffle), when the objects are grouped by tile. The larger the number of machines, the more time consuming is this procedure. For an increase of 7x, 15x and 31x in the number of processing units (in comparison to a 2-machine instance group), the 8-machine, 16-machine and 32-machine instance groups obtained a speedup of 5.24, 7.74 and 11.37, respectively, for image 32K.

In this context, it is interesting to analyze the efficiency graph (Figure 29). This graph shows that for the same number of processing units, as we increase the image size, the efficiency increases too. It means that in our experiments we did not saturate the processing capability of the configurations with different instance group sizes and a larger image could be executed in each of them.

For these experiments, the graph also shows that the efficiency decreases as the number of processing units increases. This is due to the increase in the network communication, as discussed before. Although this is true for these results, we expect that as we saturate the processing capability of a configuration with a specific number of processing units, increasing the number of machines may increase the efficiency as well.



	24	56	120	248
■ 4K	0,40	0,17	0,08	0,04
■ 8K	0,52	0,27	0,13	0,07
■ 16K	0,73	0,47	0,29	0,15
■ 32K	0,90	0,75	0,52	0,37

Figure 29: efficiency of different group sizes in comparison to a 2-machine group size (8 processing units).

It is noteworthy that Hadoop is tailored for large datasets (gigabytes and larger) (Holmes12). Thus, as the graph (Figure 28) shows, the largest images led to the highest speedups. As an example, only the images 16K and 32K produced inputs larger than one gigabyte. However, the other images are considered in the experiments to show that the system also works satisfyingly on small images.

5.4.2. Multiple instance groups

In the second step, all images (4K, 8K, 16K and 32K) were executed in 3 instance groups having 2, 4, 8, 16 and 32 machines. Again, the average processing time was taken out of 5 executions. The number of parallel instance groups was chosen automatically by the system by analyzing the graph structure (Figure 26). It is possible to limit this value by setting a maximum level of graph parallelism in ICP’s configuration file (Appendix B). The underlying idea is to benefit from the cloud environment, where increasing the number of machines is much cheaper than in a cluster, and to achieve graph parallelism by running operators in parallel. By using 3 instance groups, the land cover interpretation model was executed in three *waves* (Figure 30). In the first wave, operators 1, 2 and 3 were executed. The second wave executed operators 4, 5 and 6. Finally, the last wave executed Operator 7. The results are shown in Figure 31.

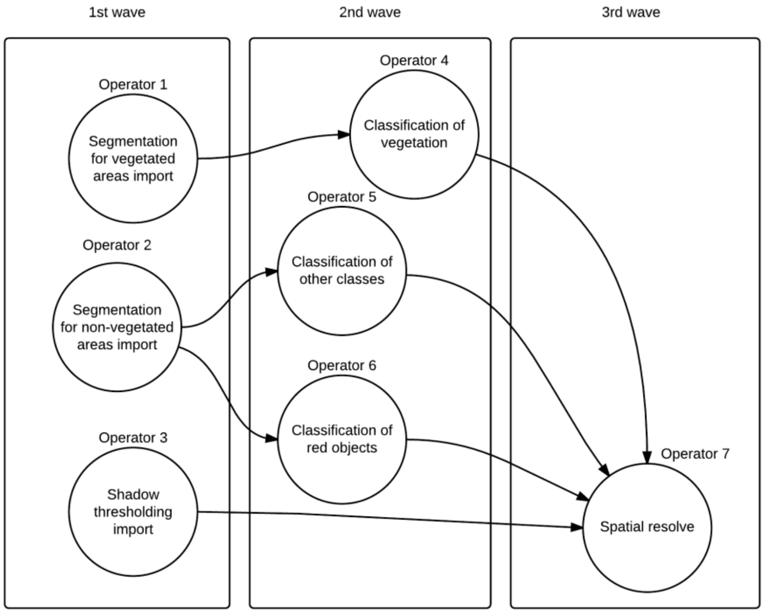
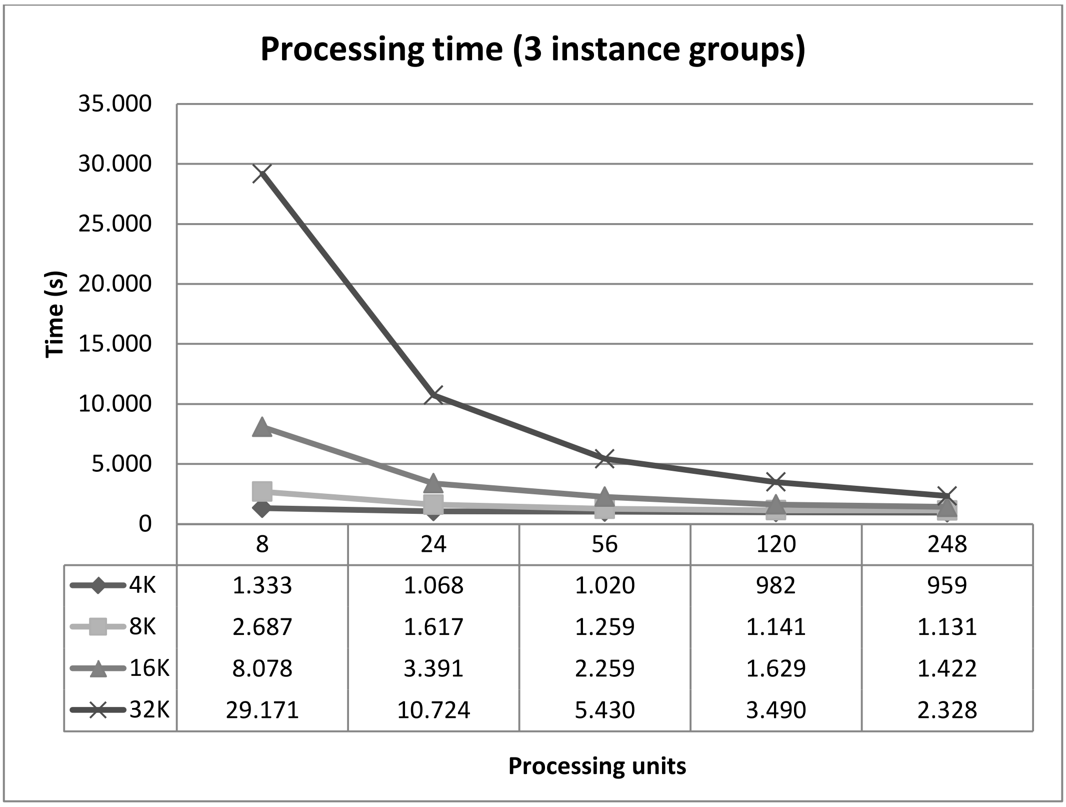

1st wave
2nd wave
3rd wave
Operator 1
Segmentation for vegetated areas import
Operator 2
Segmentation for non-vegetated areas import
Operator 3
Shadow thresholding import
Operator 4
Classification of vegetation
Operator 5
Classification of other classes
Operator 6
Classification of red objects
Operator 7
Spatial resolve

Figure 30: land cover interpretation execution waves using 3 instance groups.



	8	24	56	120	248
4K	1.333	1.068	1.020	982	959
8K	2.687	1.617	1.259	1.141	1.131
16K	8.078	3.391	2.259	1.629	1.422
32K	29.171	10.724	5.430	3.490	2.328

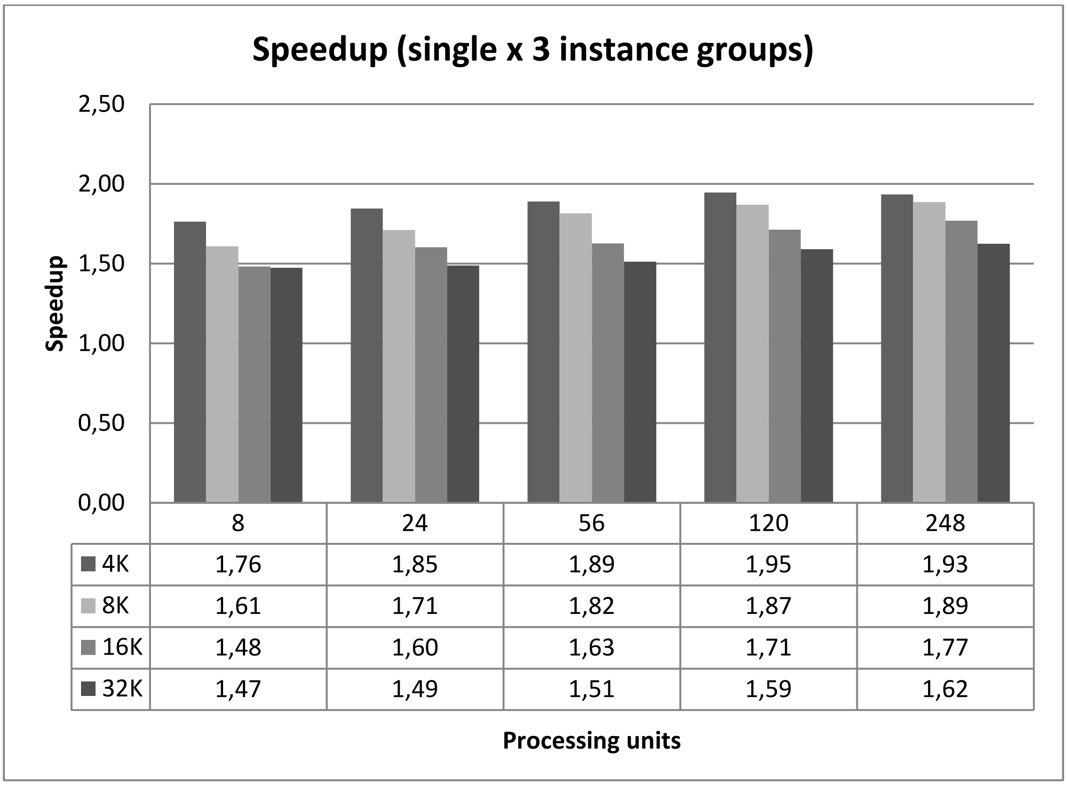
Figure 31: processing time for the experiments considering all images and instance group sizes for a 3-instance-group configuration.

The curves are similar to the graph shown in Figure 27, except for the absolute processing times. Figure 31 shows that for the same images and instance group sizes, the whole interpretation took less time to complete by taking advantage of graph parallelism. Image 4K, for example, took 2,351s (~39m) in a single 2-machine instance group, while using 3 groups of 2 machines it reached 1,333s (~22m), a speedup of 1.84. For image 32K, the interpretation took 3,780s (~1h) in a 32-machine instance group, while it reached 2,328s (~39m) using 3 groups of 32 machines, a speedup of 1.62. In fact, the speedups did not vary much. For all images and instance group sizes, the speedups ranged from 1.47 to 1.93 (Figure 32).

One interesting aspect refers to the operators and the graph structure. The land cover graph structure allowed 3 operators to run in parallel for the two first waves, while the last wave had only one operator running and two idle instance groups. At a first glance this may seem a good tradeoff, however depending on the total processing time of the operators that are executed in each wave, the gain might be modest. In this case, the first wave (the first three operators) represents, in average, around 8% of the processing time whereas the second wave represents around 34%. It means that the last operator, which is not running in parallel due to its dependency on the other operators' results, accounts for 58% of the whole processing time, in average. Thus, although three instance groups are available, the intrinsic sequential structure of this specific graph limits the maximum speedup that can be reached.

That is also why the speedups decrease for larger images (Figure 32). The larger the image, the more time the third wave takes to complete, accounting for a higher percentage of the whole processing time. This is due to the complexity of the *spatial resolution* method that is executed in this operator. For image 4K, for example, the last operator represented 47% of the whole processing time, for a 2-machine instance group. For image 32K, using the same number of machines, the last operator accounted for 68% of the interpretation processing time. A similar increase was also observed for the other instance group sizes.

Another question arises in view of the aforementioned results, specifically if the speedup pays off the costs of running the interpretation on more machines. As stated before, the last wave keeps two instance groups idle. This may incur in high unnecessary costs depending on the cloud service prices and on the processing time of the last operator. Therefore, using multiple instance groups in parallel have a high potential to speed up the processing time but it depends mostly on the graph structure of a given interpretation.



	8	24	56	120	248
■ 4K	1,76	1,85	1,89	1,95	1,93
■ 8K	1,61	1,71	1,82	1,87	1,89
■ 16K	1,48	1,60	1,63	1,71	1,77
■ 32K	1,47	1,49	1,51	1,59	1,62

Figure 32: speedups between three and single instance groups.

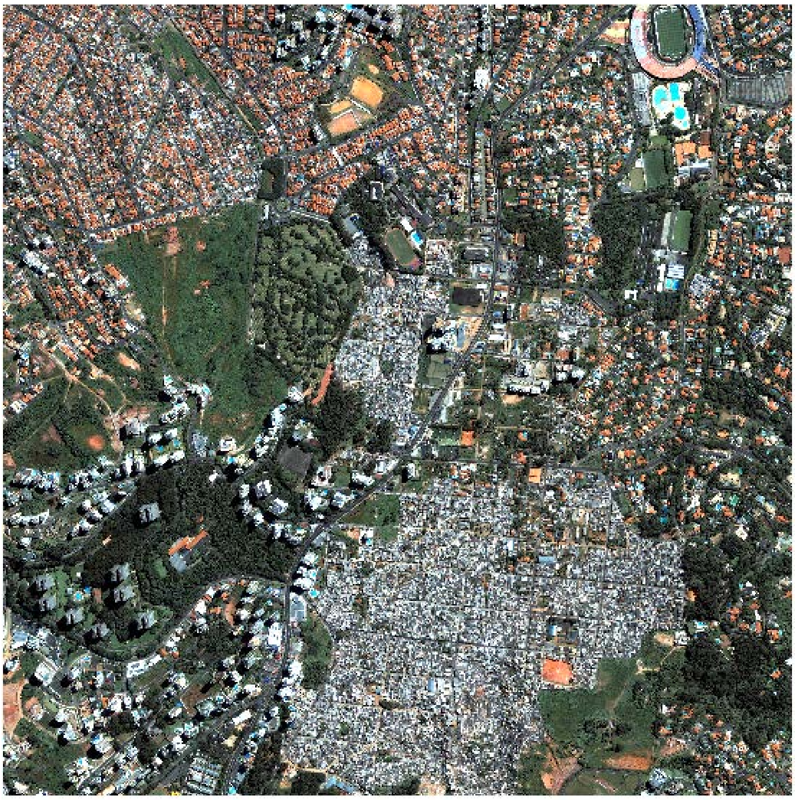
As an example of the costs of running a single experiment for the single instance group in a cloud infrastructure, Table 5 shows the costs for AWS. For image 32K, using 248 processing units (32 machines), a user would pay in average US$ 13.54 to process the whole land cover model in about 1 hour.

Table 5: costs of a single experiment for the single instance group.

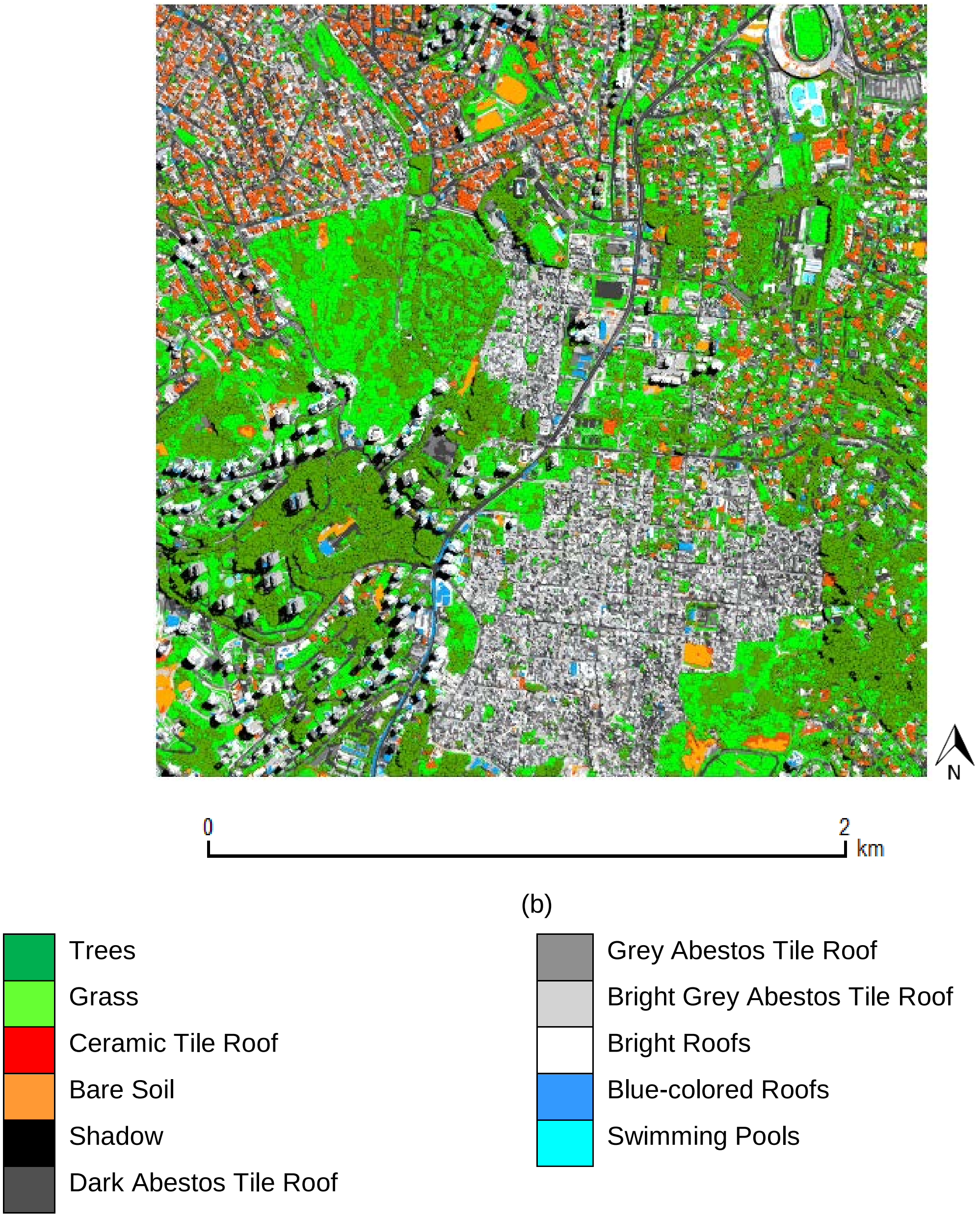
Processing units Images	8	24	56	120	248
4K	US$ 0.70	US$ 1.40	US$ 2.80	US$ 5.60	US$ 11.20
8K	US$ 0.77	US$ 1.47	US$ 2.87	US$ 5.67	US$ 11.27
16K	US$ 2.64	US$ 1.94	US$ 3.34	US$ 6.14	US$ 11.74
32K	US$ 10.74	US$ 7.94	US$ 7.94	US$ 7.94	US$ 13.54

5.4.3. Classification qualitative analysis

It is beyond the scope of this work to assess quantitatively the classification accuracy; nevertheless it is interesting to visually verify how the land cover classification performed in ICP (Figure 33 (a)) compares to the original image (Figure 33 (b)). A visual inspection shows that the obtained classification is very consistent with image 4K. It is possible to easily correlate the image objects classified in ICP and the image objects represented in Figure 33 (a).



(a)



(b)

Figure 33: image 4K (a) and land cover classification for image 4K (b).

5.5.
Land use classification

The second set of experiments consisted of the same three steps as the previous one. Firstly, the land use interpretation model was executed using a single instance group. Absolute processing times and speedups were evaluated. After that, the same investigation was carried out for multiple instance groups.

To build the land use classification model, Novack (Novack09) defined the semantic network shown in Figure 35. The class definitions and the semantic network structure were defined by visual interpretation of the image and based on the land use map of the city of São Paulo for 2005 (PMSP15). The land use classification was performed per city block. Figure 34 shows the shapefile of city blocks.

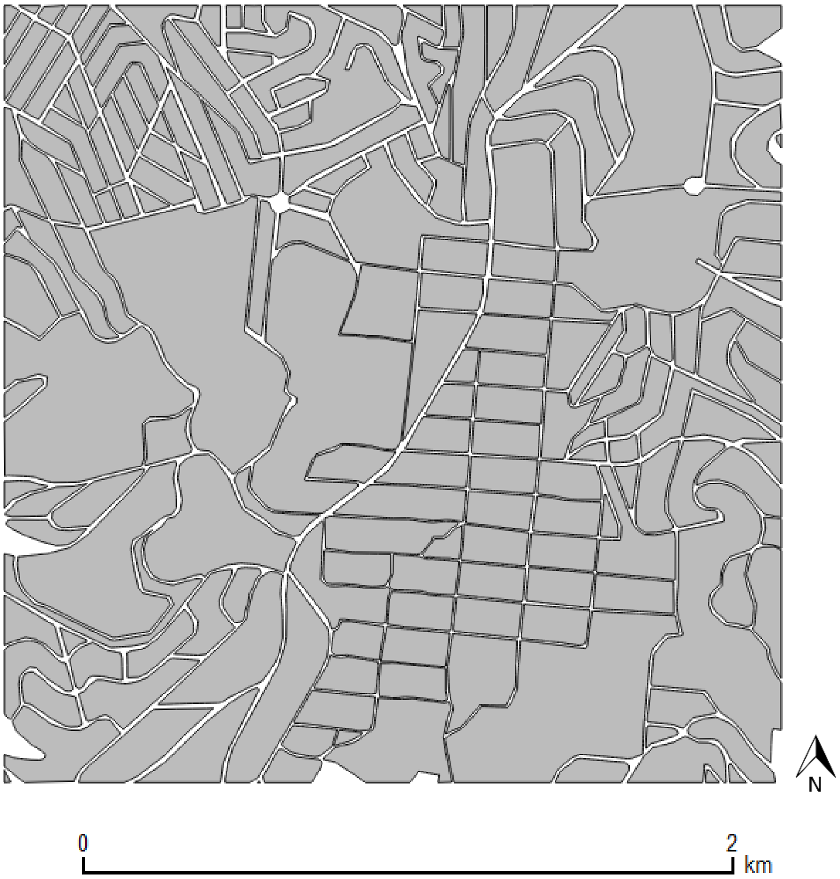


Figure 34: shapefile of city blocks.

Some classes present in the official map were discarded because they could not be classified automatically using remote sensing. In the end, nine classes were defined: *Industrial services (IS), Favelas (F), Horizontal residential of high standard (HRHS), Horizontal residential of low standard (HRLS), Vertical residential of high standard (VRHS), Vertical services (VS), Mixed residential services (MRS), Partially unoccupied land (PUL)* and *Sport clubs (SC).* Class definitions can be found in (Novack09).

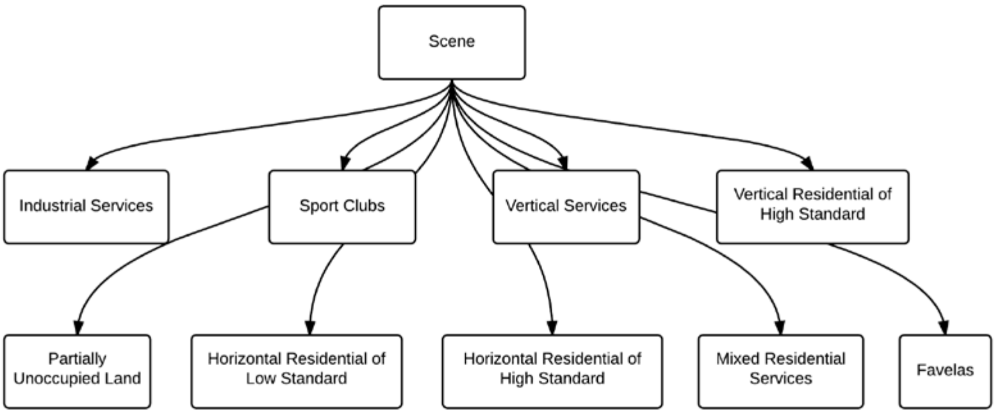

Scene
Industrial Services
Sport Clubs
Vertical Services
Vertical Residential of High Standard
Partially Unoccupied Land
Horizontal Residential of Low Standard
Horizontal Residential of High Standard
Mixed Residential Services
Favelas

Figure 35: land use semantic network.

After the definition of the semantic network, some land cover classes had to be defined to produce the input for the land use model. These new classes are listed below:

- **Building Shadow** – shadows with more than 200 m^2.
- **Big Blue** – blue objects with more than 1900 m^2.
- **Big Bright** – bright objects with more than 1900 m^2.
- **Big Bright Grey** – bright grey objects with more than 1900 m^2.
- **Big Grey** – grey objects with more than 1900 m^2.
- **Big Dark** – dark objects with more than 1900 m^2.
- **Big Roofs** – blue, bright, bright grey, grey and dark objects with more than 1900 m^2.
- **Various Roofs** – blue, bright, bright grey, grey and dark objects with less than 1900 m^2.

Besides these classes, the classes Vegetation (Trees and Grass), Pools and Ceramic Roof were also imported. After that, the attributes selected by the visual interpretation of the image were visually explored and the thresholds and fuzzy sets were defined (Novack09). This information was used to build the decision rules on InterIMAGE.

Once again, the decision rules for each classification used in InterIMAGE were translated to Pig Latin and became operators in ICP. The semantic network was also reproduced in ICP. Figure 36 presents the final operator graph.

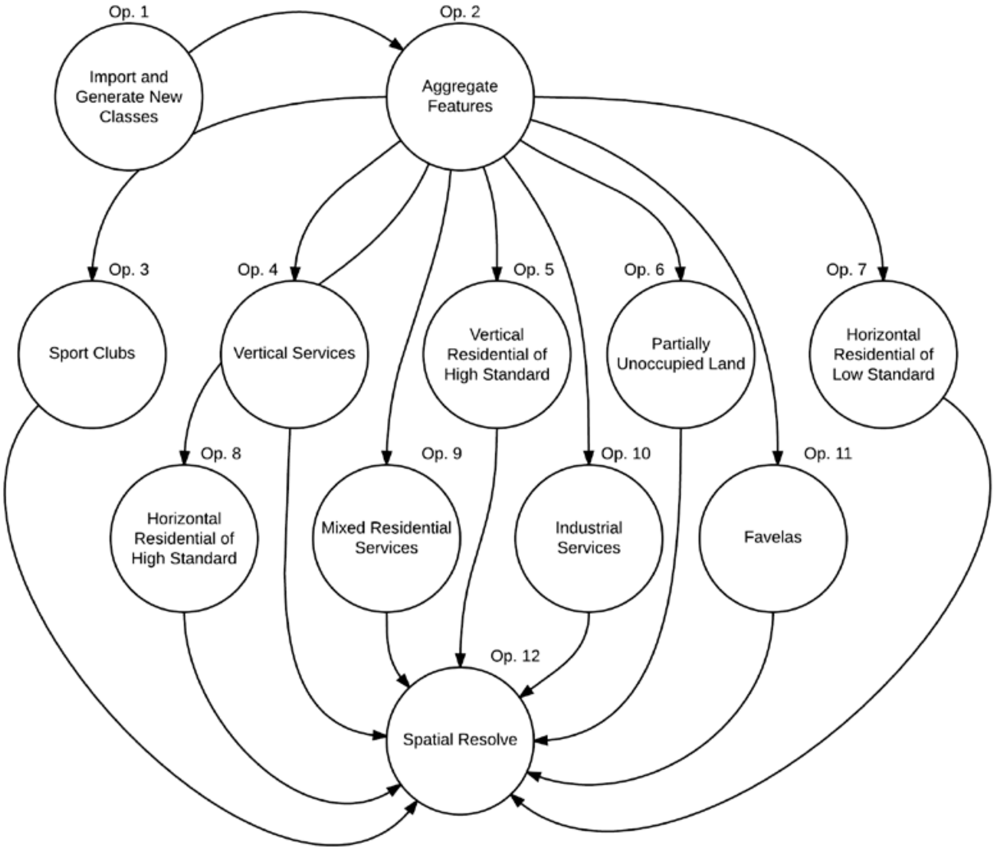

Op. 1
Import and Generate New Classes
Op. 2
Aggregate Features
Op. 3
Sport Clubs
Op. 4
Vertical Services
Op. 5
Vertical Residential of High Standard
Op. 6
Partially Unoccupied Land
Op. 7
Horizontal Residential of Low Standard
Op. 8
Horizontal Residential of High Standard
Op. 9
Mixed Residential Services
Op. 10
Industrial Services
Op. 11
Favelas
Op. 12
Spatial Resolve

Figure 36: land use operator graph.

- *Operator 1* imports the land cover classes, merges neighbor objects that belong to the same class and generates new classes.
- *Operator 2* computes the hierarchical features that will be used to classify the blocks. From this point on, the land cover segments are discarded.
- *Operators 3 to 11* classify the blocks into the different land use classes:
  - *Operator 3* – Sport Clubs
  - *Operator 4* – Vertical Services
  - *Operator 5* – Vertical Residential of High Standard
  - *Operator 6* – Partially Unoccupied Land
  - *Operator 7* – Horizontal Residential of Low Standard
  - *Operator 8* – Horizontal Residential of High Standard
  - *Operator 9* – Mixed Residential Services

    - *Operator 10* – Industrial Services
    - *Operator 11* – Favelas
- *Operator 12* resolves spatial conflicts and generates the final classification.

The Pig Latin scripts of each operator are analyzed by Pig and compiled automatically into MapReduce jobs. The MR jobs of each operator are described below:

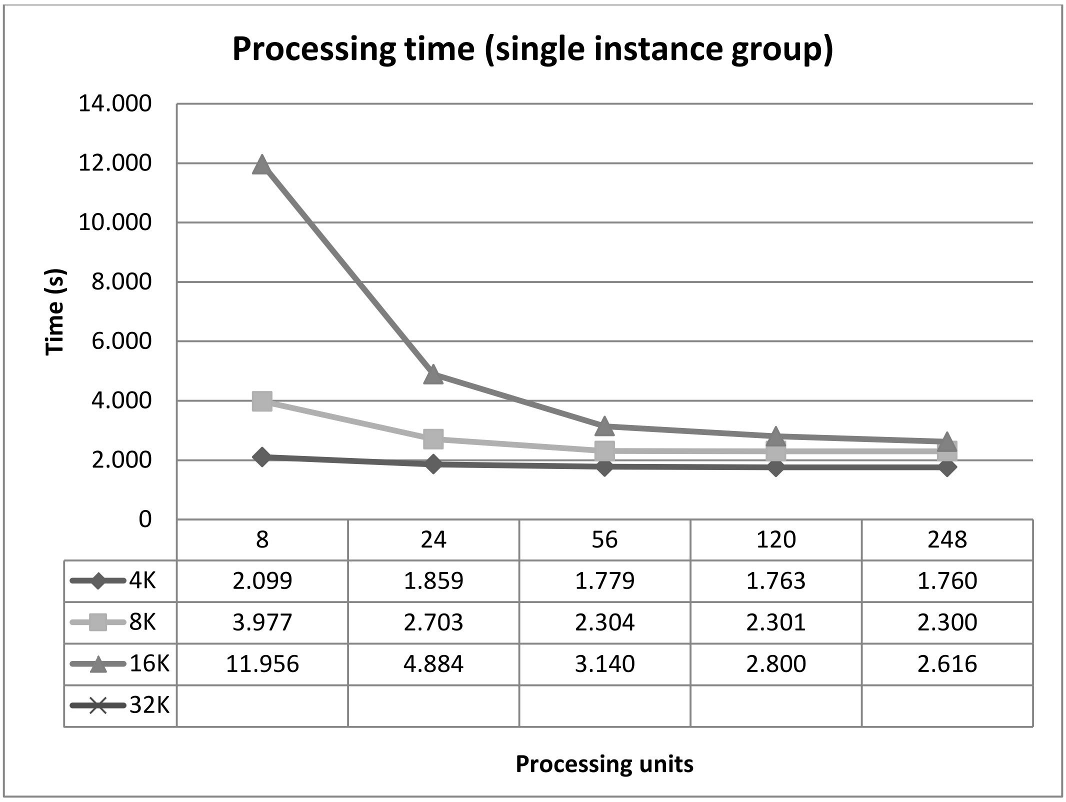
Table 6: MapReduce jobs for the land use interpretation model.

	Job	Job type
Operator 1	#1	Map-Reduce
	#2	Map only
Operator 2	#3	Map-Reduce
Operator 3	#4	Map only
Operator 4	#5	Map only
Operator 5	#6	Map only
Operator 6	#7	Map only
Operator 7	#8	Map only
Operator 8	#9	Map only
Operator 9	#10	Map only
Operator 10	#11	Map only
Operator 11	#12	Map only
Operator 12	#13	Map-Reduce

The Pig Latin script of each operator is presented in Appendix I. The whole interpretation model takes 13 MR jobs to complete (Table 6). Operators 3 to 11 have single Map-only jobs. Operator 1 has two jobs, one with Map and Reduce phases and one with only the Map phase. Operators 2 and 12 have single jobs, with both Map and Reduce phases.

5.5.1. Single instance group

In these experiments, all images (4K, 8K, 16K and 32K) were analyzed in a single instance group having 2, 4, 8, 16 and 32 machines. Each experiment was executed 5 times and the average processing time was taken. As *m1.xlarge* instances have 8 logical cores and one machine is the Hadoop master, the corresponding processing units are 8, 24, 56, 120 and 248. The results are shown in Figure 37.



	8	24	56	120	248
4K	2.099	1.859	1.779	1.763	1.760
8K	3.977	2.703	2.304	2.301	2.300
16K	11.956	4.884	3.140	2.800	2.616
32K					

Figure 37: processing time for the experiments using a single instance group.

The analysis of this graph leads to similar conclusions as the ones drawn from the previous set of experiments. As the number of processing units grows, the processing time decreases. The reduction rate of the processing time depends on the image size and the number of processing units. As the image size increases, the operators can benefit more from the available processing units achieving a higher parallelism. Smaller images produce an almost steady processing time as the instance group size increases. This is because some processing units start to become idle and parallelism is not improved.

As for the analysis of the processing flow, this graph has a peculiar characteristic. Operators 1 and 2 are the only ones that handle a high number of objects. Operator 1 imports the land cover objects, merge neighbor objects that belong to the same class and generate new classes for the land use interpretation model. After that, Operator 2 aggregates some features like *area* and *count* of these classes into the city blocks level (section 4.5.5), discarding the land cover objects in the end. From Operator 3 on, the operators only process the city blocks. Each operator is presented in detail in Appendix I.

In Operator 1, for example, image 4K has 121,496 input objects. This number corresponds to 303 MB on disk. As MapReduce’s block size is equal to 64 MB, this operator executes 5 map tasks. This means that even for the smallest instance group, with 2 machines, the 8 available processing units are already underutilized for this operator.

For image 8K, the input size on disk is equal to 1.2 GB for Operator 1. Thus, 20 map tasks are executed and processing units are fully utilized until the 4-machine instance group (24 processing units). For image 16K, 77 map tasks are launched for an input size of 4.9 GB. Processing units are fully utilized until the 16-machine instance group. Operator 2 presents a similar reasoning.

For operators 3 to 12, the map phases had as input the city block objects for each image. As shown in Table 7, the number of city blocks per image is low, by far not comparable to the number of image objects produced by image segmentation. These city block objects have for images 4K, 8K, 16K and 32K, respectively, 4.96 MB, 19.8 MB, 79.5 MB and 316 MB on disk. Thus, they represent a major underutilization of instance groups compute power. In fact, all these operators took less than 1 minute to execute individually. Although this data size is not big enough to achieve the best performance on MapReduce, they show that the system can also process small datasets efficiently.

For the reduce phases of operators 1 and 2, the parallelism is defined by the number of city blocks. This is because in these operators, the parallelism unit is not the geographic tile, but the city block. Table 7 shows the number of city blocks for each image.

Table 7: number of city blocks for each image.

	4K	8K	16K	32K
# city blocks	204	816	3,264	13,056

For image 4K, for example, there are 204 city blocks. It means that until the 16-machine instance group (120 processing units) all processing units are used. For the 32-machine instance group, from the 248 processing units, 44 remain idle. All other images use all the available processing units in the reduce phase for all the instance group sizes, considering that the larger instance group has 32 machines and 248 processing units. For operator 12, the geographic tile was used as parallelism unit.

In absolute numbers, the land use interpretation, for image 4K, took 2,099s (~35m) with a 2-machine instance group and 1,760 (~29m) with a 32-machine instance group. For image 16K, the total processing time was 11,956s (~3h) for a 2-machine instance group and 2,616 (~43m) for a 32-machine instance group. As in the previous set of experiments, the distributed architecture allowed a faster image processing by using more machines.

Taking the 2-machine instance group (8 processing units) as a basis for comparison, Figure 38 shows the speedups for different instance group sizes.

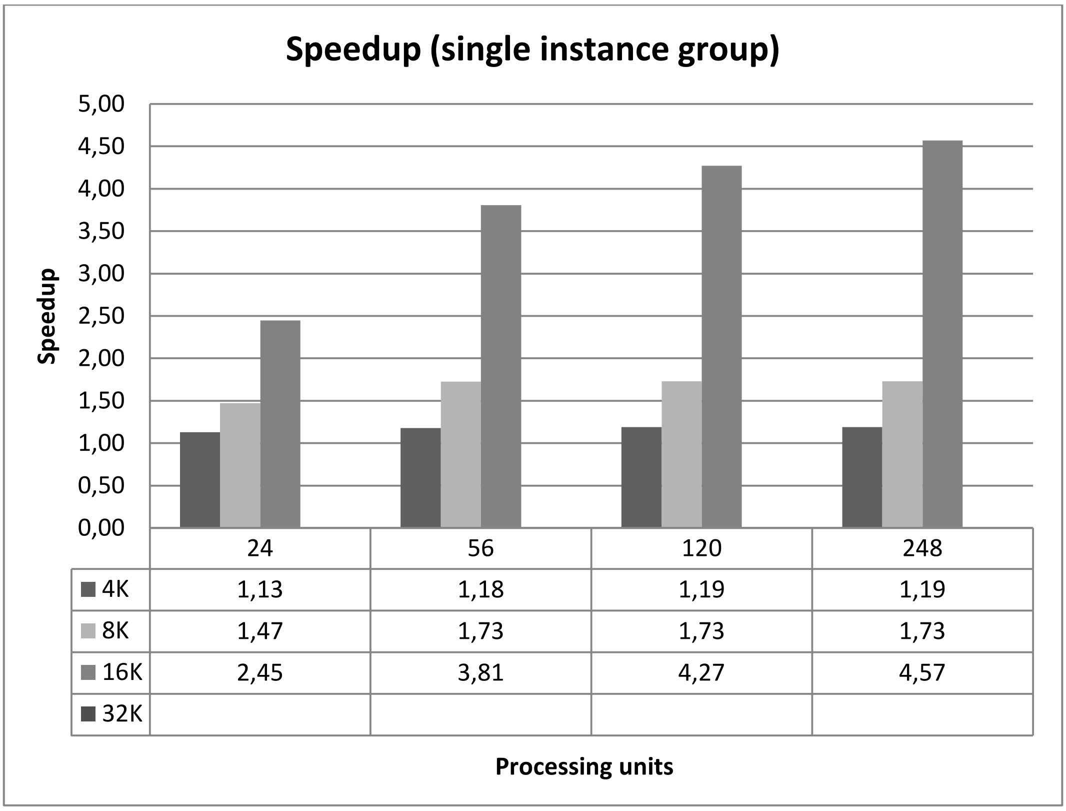
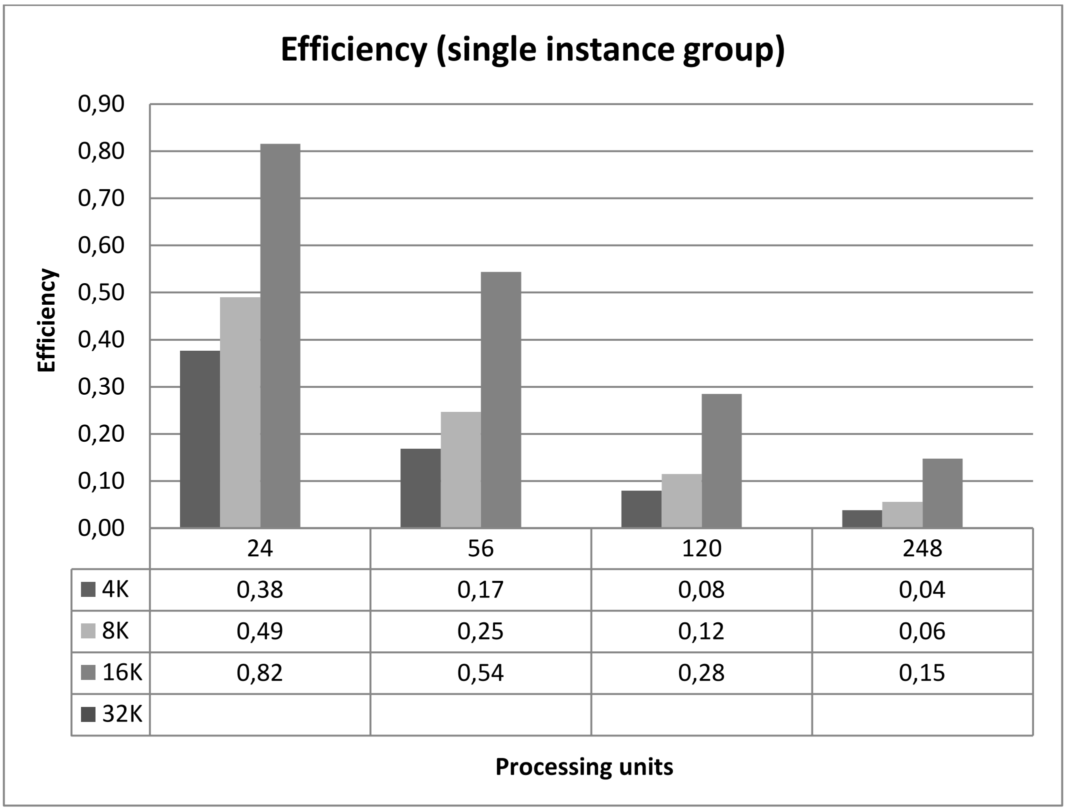


Figure 38: speedups of different group sizes in comparison to a 2-machine group size (8 processing units).

As in the previous set of experiments, the highest speedups were obtained for the largest images, since they benefit more from the available processing units. For the 4-machine instance group (24 processing units) the image 4K achieved a speedup of 1.13 while the image 16K achieved a speedup of 2.45.

The speedups obtained for this interpretation model are not much different from the speedups obtained for the land cover interpretation model. Images 4K and 8K do not present good speedups mainly because at a certain point they start to not use all the processing units, underutilizing the available processing power. Image 16K takes more advantage of the available processing units and present better speedups. As stated in the discussions of the previous set of experiments, the speedups for the largest instance groups are impaired by the network communication between the map and reduce phases. The larger the number of machines, the heavier is this procedure.

If we analyze the efficiency graph (Figure 39), we see that for the same number of processing units, as the image size is increased, the efficiency increases too. This means that in these experiments the processing capability of the configuration with different instance group sizes was not saturated. In these cases, the graph shows that the efficiency decreases as the number of processing units increases, due to the increase in the network communication. However, as a specific configuration of machines is saturated, increasing the number of processing units may increase the efficiency.



	24	56	120	248
■ 4K	0,38	0,17	0,08	0,04
■ 8K	0,49	0,25	0,12	0,06
■ 16K	0,82	0,54	0,28	0,15
■ 32K				

Figure 39: efficiency of different group sizes in comparison to a 2-machine group size (8 processing units).

Image 32K is not present in this analysis because it produced *out-of-memory* errors during the execution of the experiments. This issue has two explanations. The first one is related to the parallelism unit. As stated before, instead of the geographic tile, the parallelism unit in most of the operators in this set of experiments was the city block. As some city blocks are larger, in area, than the geographic tiles, this may have led to city blocks with a large number of objects that could not fit in the memory of a single machine. That is also why the tile size chosen for ICP was 512 (in pixels). Larger tile sizes would lead to memory issues, as more objects would be considered at the same time.

The second explanation is that Operator 1 creates new classes, based on the input land cover classes. Objects of the input classes Blue, Bright, Bright Grey, Grey and Dark are replicated to the new classes Big Blue, Big Bright, Big Bright Grey, Big Grey, Big Dark, Big Roofs and Various Roofs, and during the processing, the number of objects is greatly increased. The combination of both situations led to the memory issues experienced during the experiments.

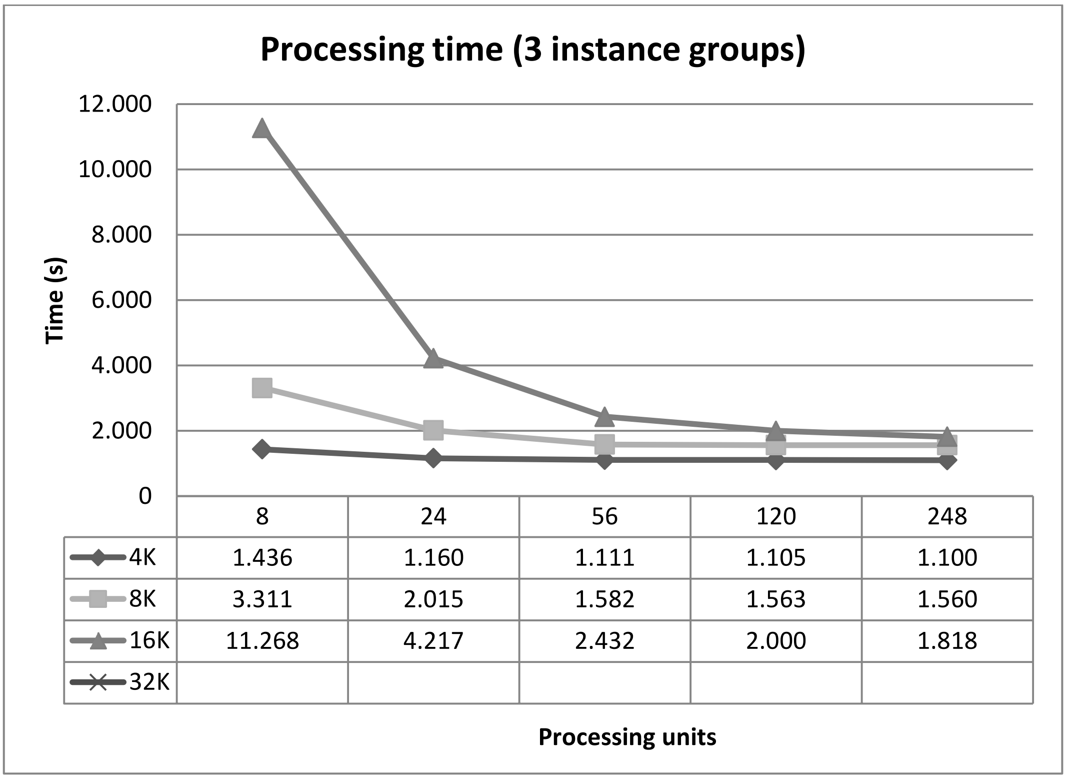
In MapReduce, as in any computation, there is a tradeoff between memory and processing. In these experiments, machines have 15 GB of RAM and 8 logical cores. If we reserve 3 GB for MapReduce standard processes, there are 12 GB for Map and Reduce tasks. Considering that a machine can have up to 8 tasks running at the same time, ICP sets 1.5 GB of maximum memory for each process in the Java heap memory.

A solution for this memory issue would be to increase the available memory for each task. In order to do that, the number of concurrent tasks would have to be reduced from 8 to 6, or 4 tasks. This would provide each task with 2 GB or 3 GB of RAM, but parallelism would be reduced.

5.5.2. Multiple instance groups

In the second step, all images (4K, 8K, 16K and 32K) were executed in 3 instance groups with 2, 4, 8, 16 and 32 machines. Again, the average processing time was taken out of 5 experiments. The number of parallelization instance groups automatically chosen by ICP was 9. However, running 9 instance groups would be very expensive. Thus, the number of instance groups was set to 3, as in the previous set of experiments.

By using 3 instance groups, the land use interpretation model was executed in six waves. In the first two waves, operators 1 and 2 were executed. In the third wave, operators 3, 4 and 5 were executed, and the fourth wave executed operators 6, 7 and 8. In the fifth wave, operators 9, 10 and 11 were executed, and the last wave executed operator 12. The results are shown in Figure 40.



	8	24	56	120	248
4K	1.436	1.160	1.111	1.105	1.100
8K	3.311	2.015	1.582	1.563	1.560
16K	11.268	4.217	2.432	2.000	1.818
32K					

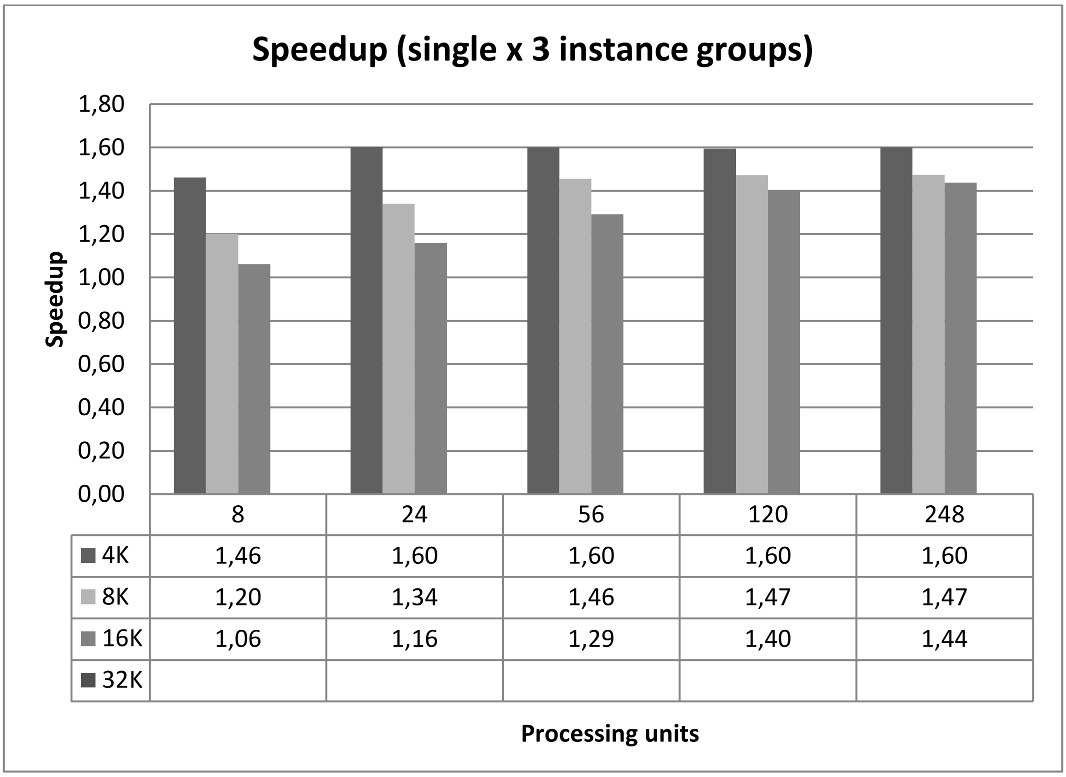
Figure 40: processing time for the experiments considering all images and instance group size for a 3-instance-group configuration.

The graph is similar to the graph obtained for a single instance group configuration (Figure 37). Figure 40 shows that for the same images and instance group sizes, the interpretation took less time to complete by using 3 instance groups. Image 4K, for example, took 2,099s (~35m) in a single 2-machine instance group, while using 3 groups of 2 machines it reached 1,436s (~24m), a speedup of 1.46. For image 16K, the whole interpretation took 2,616s (~43m) in a 32-machine instance group, while it reached 1,818s (~30m) using 3 instance groups of 32 machines, a speedup of 1.44. As in the previous set of experiments, the speedups did not vary much. For all images and instance group sizes, they ranged from 1.06 to 1.60 (Figure 41).

The interpretation could only benefit from the parallel instance groups in operators 3 to 11, where the six operators are executed in three waves. As operators 1, 2 and 12 have to be executed sequentially, they are executed in separate waves and two instance groups remain idle in these cases. For the land use interpretation model, the first wave represents, in average, around 55% of the processing time while the other waves account for 8% to 11% percent of the whole processing time. As in the previous set of experiments, although there are 3 instance groups available, the intrinsic sequential structure of the graph limits the maximum speedup that can be reached.

The speedups decrease for larger images (Figure 41) because as images get larger, the first wave (operator 1) takes more time to complete, accounting for a higher percentage of the whole processing time. This is due to the complexity of the *Merge Neighbors* operation that is executed in this operator. For image 4K, for example, operator 1 represented 53% of the whole processing time, for a 2-machine instance group. For image 16K, using the same number of machines, the first operator accounted for 89% of the processing time. A similar increase was also observed for the other instance group sizes.

Although the graph structure allows a higher level of parallelism – 9 instance groups – this would not be interesting as the speedup would probably not pay off the cloud service costs. This tradeoff must be analyzed for each interpretation model. An interpretation model in which the part that can be parallelized in multiple instance groups represents a high percentage of the whole processing time would benefit much more from this mechanism and produce higher speedup values. Finally, the image 32K was not considered in these experiments for the same reasons mentioned previously.

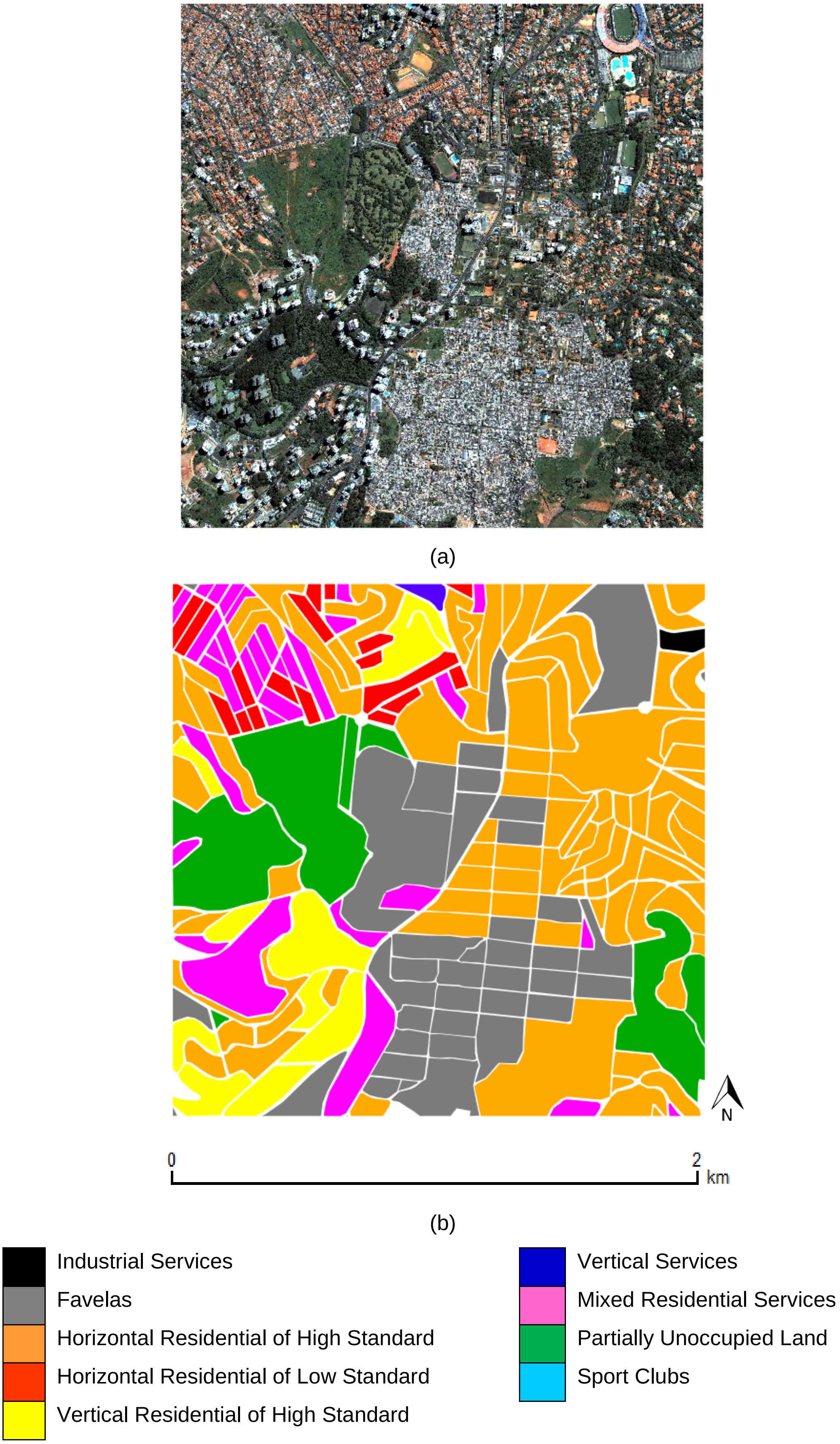


	8	24	56	120	248
■ 4K	1,46	1,60	1,60	1,60	1,60
■ 8K	1,20	1,34	1,46	1,47	1,47
■ 16K	1,06	1,16	1,29	1,40	1,44
■ 32K					

Figure 41: speedups between multiple and single instance groups.

5.5.3. Classification qualitative analysis

It is beyond the scope of this work to assess quantitatively the classification accuracy; nevertheless it is interesting to visually verify how the land use classification performed in ICP (Figure 42 (b)) compares to the original image (Figure 42 (a)). Although a visual inspection shows that most classified city blocks are consistent with image 4K, there is a block near the upper-right corner that is clearly misclassified. In the original image (Figure 42 (a)), we see a soccer stadium and a large pool which indicates that the correct class should be *Sport Club* and not *Favela*. This may have been caused by a threshold in the land use model that was not precisely translated from Novack's model (Novack09) or by a misclassification in the land cover model that may have influenced the land use classification.


N
0
2
km
⬛ Industrial Services
⬛ Favelas
🟧 Horizontal Residential of High Standard
🟥 Horizontal Residential of Low Standard
🟨 Vertical Residential of High Standard
🟦 Vertical Services
🟪 Mixed Residential Services
🟩 Partially Unoccupied Land
🟦 Sport Clubs

(a)

(b)

Figure 42: image 4K (a) and land use classification for image 4K (b).

6
Conclusions

The architecture presented in this work provides a cloud-based, distributed platform for object-based image analysis that enables the interpretation of very large image datasets. The platform achieves robustness by partitioning the data with respect to a geographic tile grid and delivers good performance by distributing processing across potentially hundreds or thousands of machines.

This novel architecture allows images and vectors (polygons) to be processed together using the MapReduce model. While vectors are treated as MapReduce's main input, image tiles are loaded on demand from an auxiliary storage service. This approach delivers a simple and powerful architecture that takes advantage of MapReduce's ability to deal with text data while image data are considered in the interpretation process only when image-related features are needed. In the cases where image data are considered, ICP provides strategies that minimize network communication by grouping image objects according to the geographic tiles and by computing all requested spectral features at once.

ICP provides two knowledge representation components – the *semantic network* and the *operator graph*. While the semantic network represents only declarative knowledge, the operator graph is responsible for representing the procedural knowledge and for controlling the interpretation strategy. It is a flexible and powerful structure that allows different types of operators to be easily combined in an interpretation model, making it easy for experts to embed their knowledge into the system.

The operator graph provides two levels of parallelism. In the *operator level*, each operator is executed in the cloud in a distributed fashion exploring the maximum parallelism available. In the *graph level*, it allows users to divide their cloud infrastructure in separate *instance groups* and execute different operators in each group in parallel, enabling a potential performance gain depending on the graph and operators' characteristics. By storing operators' partial results in a cloud storage service, the architecture also allows users to interactively create their interpretation models and easily recover from errors.

The architecture allows the inclusion of new operators that can be written in the high-level language called Pig Latin. This language makes it easier for non-technical users to extend the system by writing new operators that perform specific image interpretation tasks. This mechanism also allows operators to be created upon other existing operators. For example, an operator that classifies vegetation can be included in another operator that classifies land cover classes. If the system does not provide all the necessary functions one needs to create an interpretation model, the Pig Latin language itself is extensible through the use of UDFs (User Defined Functions) which can be created in a growing number of programming languages.

Raster and vector input data are partitioned according to a *Space Filling Curve* technique which enables the system to easily work on different levels of the geographic grid. This mechanism preserves locality information and allows operators to work with image tiles and image objects in multiple geographic scales.

ICP implements distributed strategies for spectral, topological and hierarchical feature computation; spatial conflict resolution and recursive computations. These strategies were designed for the MapReduce model and utilize its map and reduce primitives to work with very large datasets in a distributed computing environment.

All these features were implemented in a software prototype called ICP. Devised as a collection of Java libraries, the platform allows users to embed their knowledge into the system and execute an interpretation model in parallel transparently. In order to validate the platform, experiments were carried out using two classification models on a QuickBird image of an area of the São Paulo municipality.

The results obtained show that the system presents some robustness to large datasets (in the order of gigabytes), although a deeper investigation is necessary to confirm that this is also true for larger datasets (in the order of terabytes, for example). In the experiments performed, ICP was able to process images of up to 32,000x32,000 pixels (~3.81 GB), with up to 8 million image objects (~23.2 GB).

These datasets are dozens to hundreds of times larger than any dataset that could be executed in eCognition or InterIMAGE.

It is also possible to attest some scalability. The results showed that, in general, the speedups increased with the number of processing units. It is clear that the gain was higher for the largest images since they could benefit more from the available processing units. This result is consistent with the efficiency measures. They showed that the images used in the experiments did not saturate the processing power of any of the processing unit configurations. Because of this, increasing the number of processing units did not increase the efficiency. These results suggest there is a cutoff point where increasing the processing units for a specific image does not improve the speedup. In the case where the experiments got close to this saturation point, the efficiency came to 90%. This result indicates that increasing the image size for this processing unit configuration would soon saturate its processing power. In such a scenario, increasing the number of processing units would likely improve the efficiency.

In the experiments with multiple instance groups, the parallelism was impaired by the characteristics of the graphs in which sequential operators accounted for a high percentage of the whole processing time, limiting the speedups. However, the experiments indicate that graph-level parallelism may reduce significantly the processing time depending on the operator graph's characteristics.

Lastly, the technologies ICP is built upon, more specifically Hadoop and Pig, are open-source and have a large supporting community. These characteristics indicate that these technologies are reliable and are likely to evolve in a fast pace. Besides that, the fact that ICP is oriented towards cloud environments enables its use by students and scientists without the resources to set up a local cluster since most cloud services offer cheap prices and charge users in a per-use basis.

6.1.
Platform limitations

This section presents a summary of the limitations of the platform as it is today. For some discussions about possible solutions and future developments, the reader should refer to the respective section, when indicated, and the section Future Research (6.2).

The first important limitation of the platform regards the operation type called *Recursive Operation*. As discussed in the respective section (4.5.4), this operation must be further developed in order to be robust to a larger number of algorithms. The way it is designed today, it can lead to poor parallelism and memory issues when the processing reaches the highest levels of the hierarchical structure. Section 4.5.4 presents an alternative approach that should be tested on large datasets to assess its feasibility.

Another limitation regards the hierarchical features computation (section 4.5.5). As discussed in the respective section and in the results of the experiments (section 5.5.1), when the parent objects get larger, this operation may lead to memory issues, since many image objects are grouped in the same machine. The next section presents an alternative approach to solve this problem.

Another important limitation is related to image processing algorithms. The platform was tested using basic spectral features which can be easily aggregated using the algorithm presented in section 4.5.2.1. However, the consideration of more complex features like texture features would require a more robust management of image tiles. Although the architecture does not impede the computation of such features, it is not possible to compute them in the platform as it is today.

There is a limitation regarding the cloud service the platform can be used with. Although it is extensible to other services, the platform is only readily integrated to Amazon Web Services. Besides that, some features, e.g. the fault tolerance mechanism explained in section 4.2.2.3, depend heavily on a graphical user interface to be fully available to the end user.

Finally, the default Java image library that was used to cut the image tiles (JAI15) is not robust to very large images. This does not limit the distributed processing of very large images as the processing in the cloud is performed by ImgLib2 (Pietzsch12), but limits the capability of the system to generate image tiles for images larger than 46,000x46,000 pixels. This library should be replaced by a more robust one in the near future.

6.2. Future research

During the development of this thesis some directions for future researches emerged. Firstly, it would be interesting to assess the robustness of the proposed architecture on different datasets. Although the platform was conceived to interpret several images with different spatial resolutions simultaneously, the performed experiments did not evaluate this feature. Future experiments could also consider larger images and images from different sensors.

The second one regards image segmentation. It is one of the most important steps in image analysis and there is no trivial solution for distributed systems. Proposing a distributed image segmentation algorithm was out of the scope of this thesis, and is certainly a necessary step towards making ICP a fully operational, distributed, image analysis software alternative.

Another consequence of this thesis would be the development of an intuitive graphical interface. The reason for this feature is twofold. On the one hand, it would allow users of the original system to easily migrate to the new platform, as they will find in ICP an interface as user-friendly as the one provided by InterIMAGE. On the other hand, without a proper user interface, ICP hampers its use by its end-users: geologists, geographers, etc. A related desirable feature would be the development of a middleware that would allow the interaction with the system as a web service.

Another interesting direction is related to other types of data. ICP was developed targeting remote sensing data but there is no architectural limitation that would impede its use on other types of data like medical images (including three-dimensional data). It would also be interesting to extend the platform with multi-temporal classification methods.

It would be interesting to investigate the performance impact of varying the tile size, as its size is not only related to the parallelism level but also to the number of image objects that is considered in the same machine. Hopefully, it will be possible to create a mechanism that may suggest or select the best tile size based on performance and segmentation granularity. The objective of such a mechanism would be to deliver the best possible performance without incurring in memory issues.

A related feature regards the hierarchical features computation. The system must be robust to any parent object size. This type of computation must be improved in order to compute the hierarchical features of large objects (that are likely to have a large number of sub-objects) without incurring in memory issues. A possible strategy would be to compute these features using an algorithm similar to the one presented in section 4.5.2, where partial results are computed in parallel before the final result is obtained.

Another interesting feature would be to create a mechanism that allows developers to inform the system about the computational cost of each operator. This way, the system could provide the user with suggestions about the necessary computational infrastructure (number of processing units) and graph parallelism level (number of instance groups) for a specific interpretation model.

Another direction is related to MapReduce and Pig. They are two general-purpose frameworks that expose hundreds of parameters to their users. Although some parameters were mentioned and discussed in the thesis, tuning and investigating every parameter and how they affect performance constitutes a separate research that was not in the scope of this work.

Finally, as everything related to technology, distributed system technologies continue to evolve. Hadoop ecosystem itself launches a new framework every year, sometimes oriented towards specific data types, at other times seeking better performances. One example is the Apache Spark framework (Spark15) launched last year that claims to be the Hadoop MapReduce next generation. It is a never-ending objective to keep studying and evaluating these new technologies and how they can help to improve system performance and simplify data analysis.

6.3.
Final considerations

Available image data are growing in a fast pace and the tools for image analysis must continue to evolve accordingly. The final objective of these tools is to allow scientists and data analysts to process this increasing amount of image data in a faster and more reliable way. Therefore, authorities and decision makers can have in their hands timely information to make better decisions and improve people’s lives.

This project was designed to provide a robust platform that can handle huge image datasets by distributing data and processing across many computers in a cloud computing environment. The objective of proposing and validating a software platform for this purpose was also fulfilled.

The architecture is generic enough to be used with different images and sensors. It also allows the construction of very powerful interpretation models by providing a flexible graph structure and multiple semantic networks which facilitates the modeling of the user's knowledge.

The results are promising and future research is encouraged including experiments with other images and interpretation models. Moreover, distributed tools for image analysis are still very incipient, being available mainly in expensive commercial softwares. Thus, the development of free, open-source tools for such analysis is always welcome.

Bibliography

[Aji12] Aji, A.; Wang, F.; Saltz, J. H.. **Towards building a high performance spatial query system for large scale medical imaging data**. In SIGSPATIAL/GIS, pages 309–318, 2012.

[Aji13] Aji, A.; Wang, F.; Vo, H.; Lee, R.; Liu, Q.; Zhang, X.; Saltz, J.. **Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce**. Proceedings of the VLDB Endowment, Vol. 6, No. 11, Riva del Garda, Trento, Italy, 2013.

[Baatz00] Baatz, M.; Schäpe, A.. **Multiresolution segmentation – an optimization approach for high quality multi-scale image segmentation**. Strobl, J., & Blaschke, T., eds. Angewandte Geographische Informationsverarbeitung XII. Beiträge zum AGIT Symposium Salzburg, Herbert Wichmann, Karlsruhe, 2000.

[Bay06] Bay, H.; Tuytelaars, T.; Van Gool. **Surf: Speeded up robust features**. European Conference on Computer Vison, 2006, 1:404-417.

[Blaschke01] Blaschke, T.; Strobl, J.. **What's wrong with pixels? Some recent developments interfacing remote sensing and GIS**. GIS–Zeitschrift für Geoinformationssysteme 14 (6), 12–17, 2001.

[Blaschke04] Blaschke, T.; Burnett, C.; Pekkarinen, A.. **New contextual approaches using image segmentation for object-based classification**. De Meer, F., de Jong, S. (Eds.), Remote Sensing Image Analysis: Including the spatial domain. Kluver Academic Publishers, Dordrecht, pp. 211–236, 2004.

[Blaschke14] Blaschke, T.; Hay, G. J.; Kelly, M.; Lang, S.; Hofmann, P.; Addink, E.; Feitosa, R. Q.; Der Meer, F.; Der Werff, H.; Coillie, F.; Tiede, D..

Geographic Object-Based Image Analysis – Towards a new paradigm. ISPRS Journal of Photogrammetry and Remote Sensing 87, 180–191, 2014.

[Bückner01] Bückner, J.; Pahl, M.; Stahlhut, O.; Liedtke, C. E.. **GeoAIDA - A Knowledge Based Automatic Image Data Analyser for Remote Sensing Data**. CIMA 2001, Bangor, Wales, UK, 2001.

[Burnett03] Burnett, C.; Blaschke, T.. **A multi-scale segmentation/object relationship modelling methodology for landscape analysis**. Ecological Modelling 168 (3), 233–249, 2003.

[Camargo12] Camargo, F. F.; Almeida, C. M.; Costa, G. A. O. P.; Feitosa, R. Q.; Oliveira, D. A. B.; Heipke, C.; Ferreira, R. S.. **An open-source, object-based framework to extract landform classes**. Expert Systems with Applications, 2012.

[Castilla08] Castilla, G.; Hay, G. J.. **Image objects and geographic objects**. In: Blaschke, T., Lang, S., Hay, G. (Eds.), Object-Based Image Analysis. Springer, Heidelberg, Berlin, New York, pp. 91–110, 2008.

[Centeno03] Centeno, J. A. S.; Antunes, A. F. B.; Trevizan, S.; Correa, F.. **Mapeamento de áreas permeáveis usando uma metodologia orientada a regiões e imagens de alta resolução**. Revista Brasileira de Cartografia, 55 (1), pp. 48-56, 2003.

[Chang08] Chang, F.; Dena, J.; Ghemawat, S.; Hsieh, W. C.; Wallach, D. A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R. E.. **Bigtable: A distributed storage system for structured data**. ACM Trans. Comput. Syst., vol. 26, pp. 4:1-4:26, 2008.

[Chi05] Chi, H. M.; Ersoy, O. K.. **A statistical self-organizing learning system for remote sensing classification**. IEEE Transactions on Geoscience and Remote Sensing, 43, pp.1890-1900, 2005.

[Costa08] Costa, G. A. O. P.; Pinho, C. M. D.; Feitosa, R. Q.; Almeida, C. M.; Kux, H. J. H.; Fonseca, L. M. G.; Oliveira, D. A. B.. **InterIMAGE: uma Plataforma Cognitiva Open Source para a Iinterpretação Automática de Imagens Digitais.** Revista Brasileira de Cartografia, SBC – Sociedade Brasileira de Cartografia, Geodésia, Fotogrametria e Sensoriamento Remoto, Rio de Janeiro, v. 60(4), p. 331-337, 2008.

[Costa10] Costa, G.; Feitosa, R.; Fonseca, L.; Oliveira, D.; Ferreira, R.; Castejon, E.. **Knowledge-based interpretation of remote sensing data with the interimage system: major characteristics and recent developments**. In: Addink, E., Van Coillie, F. (Eds.), GEOBIA. ISPRS Working Groups, Gent, Belgium, 2010.

[Crevier97] Crevier, D.; Lepage, R.. **Knowledge-Based Image Understanding Systems: A Survey**. Computer Vision and Image Understanding , 67 (2), pp. 61-185, 1997.

[Dean04] Dean, J.; Ghemawat, S.. **MapReduce: Simplified Data Processing on Large Clusters**. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, California, 2004.

[eCognition15] **eCognition**. http://www.ecognition.com/ (accessed in January 2015).

[Eldawy13] Eldawy, A.; Li, Y.; Mokbel, M.; Janardan, R.. **CG_Hadoop: Computational Geometry in MapReduce**. SIGSPATIAL’13, Orlando, Florida, 2013.

[Eldawy14] Eldawy, A.; Mokbel, M. F.. **Pigeon: A Spatial MapReduce Language**. 30th IEEE International Conference on Data Engineering, Chicago, Illinois, 2014.

[EOSDIS15] **EOSDIS – Earthdata website**. https://earthdata.nasa.gov/ (accessed in January 2015).

[ESRI15] **ESRI Geometry API Java**. https://github.com/Esri/geometry-api-java (accessed in January 2015)

[Fisher97] Fisher, P.. **The pixel: a snare and a delusion**. International Journal of Remote Sensing 18 (3), 679–685, 1997.

[Fowler12] Fowler, M.. **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. Addison Wesley Longman, Inc., 2012.

[Gates11] Gates, A.. **Programming Pig**. O'Reilly Media, Inc., Sebastopol, California, 2012.

[Ghemawat03] Ghemawat, S.; Gobioff, H.; Leung, S.. **The Google File System**. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[Güting94] Güting, R. H.. **An introduction to spatial database systems**. The VLDB Journal, vol. 3, pp. 357-399, 1994.

[Guttman84] Guttman, A.. **R-Trees: A Dynamic Index Structure for Spatial Searching**. SIGMOD, 1984.

[Hadoop15] **Welcome to Apache Hadoop!** http://hadoop.apache.org/ (accessed in January 2015).

[HadoopTutorial15] **Hadoop Tutorial**. https://developer.yahoo.com/hadoop/tutorial/module1.html (accessed in January 2015).

[Haralick73] Haralick, R. M.; Shanmugan, K.; Dinstein, I.. **Textural features for image classification**. IEEE Transactions on Systems, Man and Cybernetics, v. 3, n. 6, p. 610-621, Nov.1973.

[Haralick85] Haralick, R. M.; Shapiro, L. G.. **Image Segmentation Techniques**. Proc. SPIE 0548, Applications of Artificial Intelligence II, 2 (April 5, 1985); doi:10.1117/12.948400.

[Hay96] Hay, G. J.; Niemann, K. O.; McLean, G.. **An Object-Specific Image-Texture Analysis of H-Resolution Forest Imagery**. Remote Sensing of Environment 55, 108–122, 1996.

[Hay01] Hay, G. J.; Marceau, D.; Dube, P.; Bouchard, A.. **A multiscale framework for landscape analysis: Object-specific analysis and upscaling**. Landscape Ecology 16 (6), 471–490, 2001.

[Hay08] Hay, G. J.; Castilla, G.. **Geographic Object-Based Image Analysis (GEOBIA): a new name for a new discipline**. In: Blaschke, T., Lang, S., Hay, G. (Eds.), Object-Based Image Analysis. Springer, Heidelberg, Berlin, New York, pp. 75–89, 2008.

[HBase15] **Apache HBase**. http://hbase.apache.org/ (accessed in January 2015).

[Hive15] **Apache Hive**. http://hive.apache.org/ (accessed in January 2015).

[Holmes12] Holmes, A.. **Hadoop in Practice**. Manning Publications Co., Shelter Island, New York, 2012.

[IEEESpectrum13] **Start-up Profile: Skybox Imaging**. http://spectrum.ieee.org/at-work/innovation/startup-profile-skybox-imaging (accessed in January 2015).

[Indyk97] Indyk, P.; Motwani, R.; Raghavan, P.; Vempala, S.. **Locality-Preserving Hashing in Multidimensional Spaces**. STOC’97, El Paso, Texas, 1997.

[JAI15] **Java Media Downloads**. http://www.oracle.com/technetwork/java/current-142188.html

[Jain00] Jain, A. K.; Duin R. P. W.; Mao, J.. **Statistical Pattern Recognition: A Review**. IEEE Transactions on Pattern Analysis and Machine Inteligence, 22 (1), pp. 4-37, 2000.

[JTS15] **JTS Topology Suite**. http://www.vividsolutions.com/JTS/JTSHome.htm (accessed in January 2015).

[Kummert98] Kummert, F.. **Interpretation von Bild- und Sprachsignalen – Ein hybrider Ansatz**. Shaker Verlag, 1998.

[Lakshman10] Lakshman, A.; Malik, P.. **Cassandra: a decentralized structured storage system**. ACM SIGOPS Operating Systems Review, vol. 44, pp. 35-40, 2010.

[Lee11] Lee, R.; Luo, T.; Huai, Y.; Wang, F.; He, Y.; Zhang, X.. **Ysmart: Yet another sql-to-mapreduce translator**. ICDCS, 2011.

[Li00] Li, D.; Di, K.; Li, D.. **Land Use Classification of Remote Sensing Image with GIS on Spatial Data Mining Tehcniques**. International Archives of Photogrammetry and Remote Sensing, Vol. XXXIII, Part B3, pp. 238-245, 2000.

[Liedtke99] Liedtke, C.-E.; Bückner, J.; Grau, O.; Growe, S.; Tönjes, R.. **AIDA: A system for the knowledge-based interpretation of remote sensing data**. In: Proceedings of the Third International Airborne Remote Sensing Conference And Exhibition, Copenhagen, Denmark, 1999.

[Lin13] Lin, F.; Chung, L.; Ku, W.; Chu, L.; Chou, T.. **The Framework of Cloud Computing Platform for Massive Remote Sensing Images**. IEEE 27th International Conference on Advanced Information Networking and Applications, 2013.

[Liu12] Liu, Y.; Chen, L.; Xiong, W.; Liu, L.; Yang, D.. **A MapReduce Approach for Processing Large-scale Remote Sensing Images.** 20th International Conference on Geoinformatics (GEOINFORMATICS), 2012.

[Liu13] Liu, Y.; Chen, B.; He, W.; Fang, Y.. **Massive image data management using HBase and MapReduce**. 21st International Conference on Geoinformatics (GEOINFORMATICS), 2013.

[Lo96] Lo, M.-L.; Ravishankar, C.V.. **Spatial hash-joins**. In SIGMOD, pages 247-258, 1996.

[Lu12] Lu, J.; Guting, R. H.. **Parallel Secondo: Boosting Database Engines with Hadoop**. ICPADS, 2012.

[Mahout15] **Apache Mahout: Scalable machine learning and data mining**. http://mahout.apache.org/ (accessed in January 2015).

[MarketWired14] **U.S. Department of Commerce Relaxes Resolution Restrictions DigitalGlobe Extends Lead in Image Quality**. http://www.marketwired.com/press-release/us-department-commerce-relaxes-resolution-restrictions-digitalglobe-extends-lead-image-nyse-dgi-1919482.htm (accessed in January 2015).

[Mciver01] Mciver, D. K.; Friedl, M. A.. **Estimating pixel-scale land cover classification confidence using nonparametric machine learning methods**. IEEE Transactions on Geoscience and Remote Sensing, 39 (9), pp. 1959-1968, 2001.

[Mell11] Mell, P.; Grance, T.. **The NIST Definition of Cloud Computing – Recommendations of the National Institute of Standards and Technology**. 2011.

[Nievergelt84] Nievergelt, J.; Hinterberger, H.; Sevcik, K.. **The Grid File: An Adaptable, Symmetric Multikey File Structure**. TODS, 9(1), 1984.

[Nishmura11] Nishimura, S.; Das, S.; Agrawal, D.; Abbadi, A. E.. ***MD*-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services**. In MDM, June 2011.

[Novack09] Novack, T.. **Classificação da cobertura da terra e do uso do solo urbano utilizando o sistema InterIMAGE e imagens do sensor QuickBird**. MSc. Dissertation. São José dos Campos (Brazil): National Institute for Space Research. Available at: http://mtc-m18.sid.inpe.br/rep/sid.inpe.br/mtc-m18@80/2009/08.31.21.234.

[Novack10] Novack, T.; Kux, H. J. H.. **Urban land cover and land use classification of an informal settlement area using the open-source knowledge-based system InterIMAGE**. Journal of Spatial Science, 55: 1, 23- 41, 2010.

[Olston08] Olston, C.; Reed, B.; Srivastava, U.; Kumar, R.; Tomkins, A.. **Pig Latin: A Not-So-Foreign Language for Data Processing**. SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

[Pahl08] Pahl, M.. **Arquitetura de um sistema baseado em conhecimento para a interpretação de dados de sensoriamento remoto de múltiplos sensores**. 95 p. (INPE-15211-TAE/71). PhD Thesis - Universidade de Hannover, São José dos Campos. 2008. Available in: http://urlib.net/sid.inpe.br/mtc-m17@80/2008/03.07.18.31 (accessed in January 2015).

[Patel96] Patel, J. M.; DeWitt, D. J.. **Partition Based Spatial-Merge Join**. SIGMOD 96, 1996.

[Pietzsch12] Pietzsch, T.; Preibisch, S.; Tomančák, P.; Saalfeld, S.. **ImgLib2 – generic image processing in Java**. Bioinformatics, 28(22), 3009-3011, 2012.

[Pig15] **Welcome to Apache Pig!** http://pig.apache.org/ (accessed in January 2015).

[PiggyBank15] **PiggyBank**. https://cwiki.apache.org/confluence/display/PIG/PiggyBank (accessed in January 2015).

[PMSP15] **Município em Mapas**. Prefeitura Municipal da Cidade de São Paulo, São Paulo, 2002. http://www9.prefeitura.sp.gov.br/sempla/mm/ (accessed in January 2015).

[Reuters14] **DigitalGlobe gains U.S. govt license to sell sharper satellite imagery.** http://www.reuters.com/article/2014/06/11/digitalglobe-imagery-idUSL2N0OR2UX20140611 (accessed in January 2015).

[Rigaux02] Rigaux, P.; Scholl, M.; Voisard, A.. **Spatial Databases: With Application To GIS**. Morgan Kaufmann, San Francisco, 2002.

[Sagan94] Sagan, H.. **Space-Filling Curves**. Springer-Verlag, 1994.

[Sagerer97] Sagerer, G.; Niemann, H.. **Semantic networks for understanding scenes. Advances in Computer Vision and Machine Intelligence**. 1. ed., New York: Plenum Publishing Corporation, 500p, 1997.

[Sammer12] Sammer, E.. **Hadoop Operations**. O’Reilly Media Inc., Sebastopol, California, 2012.

[Sample10] Sample, J. T.; Ioup, E.. **Tile-Based Geospatial Information Systems – Principles and Practices**. Springer Science+Business Media, LLC, New York, NY, 2010.

[Schiewe01] Schiewe, J.; Tufte, L.; Ehlers, M.. **Potential and problems of multi-scale segmentation methods in remote sensing**. Geo-Informations-Systeme, 6, pp. 34-39, 2001.

[Shekhar03] Shekhar, S.; Chawla, S.. **Spatial Databases: A Tour**. 1st ed., Prentice Halll, 2003.

[Sosinsky11] Sosinsky, B.. **Cloud Computing Bible**. Wiley Publishing, Inc., Indianapolis, Indiana, 2011.

[Sousa11] Sousa, G. M.; Santos, F. V.; Ferreira, R. S.; Fernandes, M. C.. **Modelagem do Conhecimento Aplicada ao Estudo da Susceptibilidade à Ocorrência de Incêndios no Maciço da Pedra Branca/RJ**. XV Simpósio Brasileiro de Sensoriamento Remoto, Curitiba, Paraná, 2011.

[Spark15] **Apache Spark – Lightning-fast cluster computing**. http://spark.apache.org/ (accessed in January 2015).

[Strahler86] Strahler, A. H.; Woodcock, C. E.; Smith, J. A.. **On the nature of models in remote sensing**. Remote Sensing of Environment 20, 121–139, 1986.

[Thusoo09] Thusoo, A.; Sen, J. S.; Jain, N.; Shao, Z.; Chakka, P.; Anthony, S.; Liu, H.; Wyckoff, P.; Murthy, R.. **Hive: A Warehousing Solution over a Map-Reduce Framework**. PVLDB, 2009.

[Vatsavai13] Vatsavai, R. R.. **Object based image classification: state of the art and computational challenges**. Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial '13). ACM, New York, NY, USA, 73-80, 2013.

[Venner09] Venner, J.. **Pro Hadoop**. Apress, Inc., Berkeley, California, 2009.

[Wang12] Wang, H.; Shen, Y.; Wang, L.; Zhufeng, K.; Wang, W.; Cheng, C.. **Large-Scale Multimedia Data Mining Using MapReduce Framework**. 4th International Conference on Cloud Computing Technology and Science, 2012.

[Wang09] Wang, K.; Franklin, S. E.; Guo, X.; He, Y.; McDermid, G. J.. **Problems in remote sensing of landscapes and habitats**. Progress in Physical Geography 33 (6), 747–768, 2009.

[Webb02] Webb, A. R.. **Statistical Pattern Recognition**, Second Edition, Wiley, 2002.

[Weng09] Weng, Q.. **Remote Sensing and GIS Integration - Theories, Methods, and Applications**. McGraw-Hill, New York, 2009.

[White12] White, T.. **Hadoop: The Definitive Guide, Third edition**. O'Reilly Media Inc., Sebastopol, California, 2012.

[Zhong08] Zhong, P.; Zhang, P.; Wang, R.. **Dynamic Learning of SMLR for Feature Selection and Classification of Hyperspectral Data**. IEEE Geoscience and Remote Sensing Letters, 5 (2), pp. 280-284, 2008.

[Zhong12] Zhong, Y.; Han, J.; Zhang, T.; Li, Z.; Fang, J.; Chen, G.. **Towards Parallel Spatial Query Processing for Big Spatial Data**. IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, 2012.

[Zhou98] Zhou, X.; Abel, D. J.; Truffet, D.. **Data partitioning for parallel spatial join processing**. Geoinformatica, 2:175–204, June 1998.

[Zookeeper15] **Apache Zookeeper**. http://zookeeper.apache.org/ (accessed in January 2015).

Appendix

A. Pig Latin

Data Model

Pig has two basic categories of data types: *scalar types* and *complex types*. Scalar types are simple types that contain a single value and appear in most programming languages: *int*, *long*, *float*, *double*, *chararray* and *bytearray*.

Complex types are types that can contain other types, including other complex types. Pig has three complex data types: map, tuple and bag.

A *map* is a collection of mappings from chararrays to data elements. The chararray is called a key and the element is referred to as the value. The key is used as an index to find the corresponding value. There is no requirement that the elements in a map must be of the same type. A typical map representation would be: `[‘id’#1, ‘class’#‘trees’]`.

A *tuple* is a fixed-length, ordered collection of data elements. It is divided into fields, where each field contains a data element. A tuple is analogous to an SQL row, with the fields being table columns. Because a tuple is ordered, it is possible to reference the fields in a tuple by position. It is possible to associate a schema to a tuple, describing the field’s names and types. This allows Pig to type-check the data and allows users to reference the fields by name. A typical tuple representation would be: `(1, ‘trees’)`.

A *bag* is an unordered collection of tuples. Because it has no order, it is not possible to reference tuples in a bag by position. Like tuples, it is possible to associate a schema to a bag. Bag is the only type in Pig that is not required to fit into memory, Pig has the ability to spill bags to disk when necessary. A typical bag representation would be: `{(1, ‘trees’), (2, ‘grass’), (3, ‘bare soil’)}`.

Pig Latin commands

This section briefly describes the most important Pig Latin commands.

`LOAD` and `STORE`. The `LOAD` statement loads data into a Pig Latin script. This command defines where the input data is and how it is going to be read (converted into Pig's data model). The output of any command is a data set or a *relation*.

```
objects = LOAD ‘objects.txt’ USING MyLoadFunc() AS (id:int,
tile:chararray, class:chararray, geometry:chararray);
```

In the above statement, the `LOAD` command is loading the file `objects.txt` using the load UDF `MyLoadFunc`. Any load UDF can be plugged to a `LOAD` command. The clause `AS` defines the schema of the output dataset.

The STORE command is analogous:

```
STORE objects INTO ‘objects.txt’ USING MyStoreFunc();
```

`FOREACH..GENERATE`. This command applies a set of expressions to every tuple in a data set.

```
new_objects = FOREACH objects GENERATE id, tile, class, geometry,
area(geometry);
```

The above command processes each tuple of the relation `objects` independently and returns an output tuple. The first fields of the output tuple are the `id`, `tile`, `class` and `geometry` fields of the input tuple, and the last field of the output tuple is the result of applying the evaluation UDF `area` to the `geometry` field of the input tuple. Any evaluation UDF can be used in a `FOREACH` command.

`FILTER`. This command allows users to retain some tuples and discard others according to a predicate.

```
valid_objects = FILTER objects BY isValid(geometry);
```

The above statement discards the objects with invalid geometries. The predicate is coded within the filter UDF `isValid`. The `FILTER` command supports comparison operators (`==`, `!=`, etc.) and logical connectors (`AND`, `OR`, etc.). Any filter UDF can be used in a `FILTER` command.

`(CO)GROUP`. This command collects together records with the same key. It forces a reduce phase.

```
grouped_objects = COGROUP objects1 BY tile, objects2 BY tile;
```

This statement groups two different image object datasets based on the geographic tile information. For readability, `GROUP` is used when only one relation is being grouped.

B.
Configuration files

Main configuration file

ICP has a configuration file where the user can define some parameters that will determine the system's behavior. The main parameters are the following:

- **interimage.tileSize** – defines the tile size in pixels. Default: 512.
- **interimage.clusterSize** – defines the number of machines in each cluster (or instance group). Default: 8.
- **interimage.maxNumberOfClusters** – defines the maximum number of concurrent clusters (or instance groups). The system will try to parallelize the execution as much as possible, what might incur in high costs. This is, therefore, a critical parameter that controls the maximum parallelization level. Default: 1.
- **interimage.cores** – defines the number of physical cores in a cluster node. The system executes automatically 2 tasks per physical core. Default: 4.
- **interimage.storageService** – defines the cloud service to be used. So far, only *AWS* is supported.

The following parameters are specific for AWS. They show how cloud service specific parameters can be passed to the system.

- **interimage.aws.accessKey** – AWS account access key.
- **interimage.aws.secretKey** – AWS account secret key.
- **interimage.aws.S3Bucket** – AWS S3 bucket name.
- **interimage.aws.logging** – defines the S3 folder where the logs will be copied to. If left blank, no copy is made.
- **interimage.aws.instanceType** – defines the instance type. Default: m1.xlarge.
- **interimage.aws.bidPrice** – defines the bid price for the SPOT market. If the price in the market goes above this threshold, the instances are shutdown. Default: 0.35.
- **interimage.aws.amiVersion –** defines the AMI (Amazon Machine Image) version. Defaults to version 3.3.1 which provides the latest versions of Hadoop and Pig.
- **interimage.aws.region** – defines the AWS region. So far, only *us-eat-1a* is supported.
- **interimage.aws.market** – defines whether the instances will be acquired in the *SPOT* market or *ON_DEMAND*. SPOT instances are usually cheaper. Default: SPOT.

Operator configuration file

In ICP, operators are defined as Pig Latin scripts. In order to add a new operator to the system, the user has to provide an XML file that defines the Pig Latin script template, among other information. Here is an example of the configuration file of the operator *Import Segmentation*, used in the experiments:

```
<?xml version="1.0" encoding="UTF-8"?>
    <operators>
        <operator name="ImportSegmentation" oldName="TA_ShapeFile_Import">

            <input type="String">INPUT_PATH</input>
            <input type="String">CLASS</input>
```

```
<input type="Double">RELIABILITY</input>

<input type="URL">INPUT_ROI</input>
<output type="URL">OUTPUT_ROI</output>

<template>

    DEFINE II_CalculateTiles
    br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles
    ('$TILES_FILE','single','$MIN_RESOLUTION','negative');

    --Loads image objects
    load = LOAD '$INPUT_PATH' USING
    org.apache.pig.builtin.JsonLoader('geometry:chararray,
    data:map[chararray], properties:map[bytearray]');

    --Filters out objects with invalid geometries
    selection = FILTER $LAST_RELATION BY II_IsValid(geometry,
    properties, '');

    --Computes tile info
    projection = FOREACH $LAST_RELATION GENERATE geometry,
    II_ToProps(II_CalculateTiles(geometry,
    properties#'tile'),'tile',properties) AS properties;

    --Imports another operator. In this case, an operator
    that imports target regions
    BEGIN IF $INPUT_ROI
    INCLUDE ImportROI
    END IF $INPUT_ROI

    --Sets objects’ class
    projection = FOREACH $LAST_RELATION GENERATE geometry,
    II_ToProps('$CLASS','class',properties) AS properties;

    --Sets objects’ membership
    projection = FOREACH $LAST_RELATION GENERATE geometry,
    II_ToProps($RELIABILITY,'membership',properties) AS
    properties;

</template>

<description>
```

```
            Operator that imports an existing segmentation. The
            operator also accepts target regions, an output class and
            membership values.
        </description>

        <author>Rodrigo Ferreira</author>

    </operator>
</operators>
```

The *operator* tag defines the operator *name*. It is also possible to define a field *oldName* that may be used in the future to associate the operators in ICP and the operators in InterIMAGE.

The operator tag has *input, output, template, description* and *author* as sub-tags. The input tags define the input parameters and their data types. The template tag defines the Pig Latin script template. The script may refer to any UDF present in the default libraries or added by the user. It also may contain some special terms beginning with the $ symbol. These terms are variables that will be replaced by explicit parameters of the operator or by default parameters of the system. These are the main default parameters:

- $IMAGES_PATH – URL of the image tiles.
- $SHAPES_PATH – URL of the folder with vector data.
- $TILES_FILE – URL of the file with geographic tile information.
- $FUZZYSETS_FILE – URL of the file that contains fuzzy sets information.
- $SEMANTICNET_FILE – URL of the semantic network file.
- $TILES_PATH – URL of the folder with geographic tile information.
- $TILE_SIZE_METERS – tile size in meters.
- $MIN_RESOLUTION – minimum resolution among all input images.
- $PARALLEL – default parallel capacity of the cluster (number of processing units).
- $CRS – default coordinate reference system (accepts EPSG codes).

To implement some specific functionalities, ICP implements a preprocessor for the original Pig Latin language. It is possible for example to insert an IF clause that only considers a specific part of the code if an input parameter is set. This preprocessor also allows the operator designer to use terms like $LAST_RELATION, what makes it easier to write Pig Latin templates. In Pig, all relations must be directly addressed.

The INCLUDE command is also a preprocessor command. With this command it is possible to include an operator into another one. This is a powerful mechanism that allows very complex operators to be built by combining other operators.

C. Spatial-blind operations

- **Morphological operations**
  - **Angle** – the main angle of an image object. It is obtained by calculating the smallest surrounding ellipse, and the angle of the longest radius of the ellipse corresponds to the object's angle.
  - **Area** – the real area of the image object (in meters).
  - **Buffer** – computes a buffered polygon around the image object.
  - **Centroid** – centroid of the image object.
  - **Compactness** – computed according to the following formula: $compactness = \frac{P/A}{\sqrt{A}}$, where P is the object's perimeter and A is the object's area.
  - **Convex Hull** – computes the convex hull of a given image object.
  - **Density –** the density of an image object is calculated by the ratio between its area and its radius (the maximum distance between the polygon centroid and all its vertices).
  - **Ellipse Fit –** finds the smallest surrounding ellipse and returns the ratio between the image object's area and the ellipse's area.

  - **Envelope** – object’s geographic bounding box.
  - **Fractal Dimension** – computed according to the following formula: $fractalDimension = 2\frac{\log(\frac{P}{4})}{\log A}$, where P is the object’s perimeter and A is the object’s area.
  - **Gyration Radius** – this attribute equals the mean distance between each pixel in the image object and the image object centroid.
  - **Length** – the longest side of the smallest surrounding rectangle.
  - **Length Width Ratio –** calculates the ratio between the length and the width of an image object.
  - **Perimeter** – object’s perimeter
  - **Perimeter Area Ratio –** calculates the ratio between the perimeter and the area of an image object.
  - **Rectangle Fit –** finds the smallest surrounding rectangle and returns the ratio between the image object’s area and the rectangle area.
  - **Roundness** – computed according to the following formula: $roundness = 1 - \frac{A}{\pi R^2}$, where A is the object’s area.
  - **Shape Index** – computed according to the following formula: $shapeIndex = \frac{P}{4\sqrt{A}}$, where P is the object’s perimeter and A is the object’s area.
  - **Smallest Surrounding Ellipse** – computes the smallest surrounding ellipse.
  - **Smallest Surrounding Rectangle** – computes the smallest surrounding rectangle.
  - **Width** – the smallest side of the smallest surrounding rectangle.

- **Data Mining Operations**
  - **Bayesian Classifier** – classifies image objects using a Bayesian classifier.

  - **Decision Tree Classifier** – classifies image objects using a decision tree classifier.
  - **Membership** – computes the membership value of an attribute according to a given fuzzy set.
  - **Random Forest Classifier** – classifies image objects using a random forest classifier.
  - **SVM Classifier** – classifies image objects using a SVM classifier.

- **Other operations**
  - **Spatial Clip** – clips image objects based on the given target regions.
  - **Spatial Filter** – filters image objects based on the given target regions.
  - **Replicate** – replicates image objects according to the geographic tiles they intersect.
  - **Calculate Tiles** – computes the geographic tiles an image object intersects.
  - **Is Valid** – verifies if an image object has a valid geometry.
  - **To Props** – adds a new property to the image object's properties field.
  - **To Classification** – adds a new classification to the classification list.
  - **Classify** – classifies an image object based on its classification list.
  - **Select Class** – selects the objects that belong to the given class.
  - **Min** – computes the minimum value of all input values.
  - **Mul** – computes the product of all input values.
  - **Sum** – sums all input values.
  - **Mean** – computes the mean value of all input values.
  - **Max** – computes the maximum value of all input values.

D.
Spatial-aware operations

- **Simple Spatial Resolve** – resolves spatial conflicts between image objects that have the same geometry.

E.
Spatial-aware operations with replication

- **Spectral Features**
    - **Mean** – mean value of the pixels found inside an image object for the given image band.
    - **Maximum pixel value** – the maximum pixel value found inside an image object for the given image band.
    - **Minimum pixel value** – the minimum pixel value found inside an image object for the given image band.
    - **Band ratio** - represents the amount that a given layer contributes to the total brightness of an image object.
    - **Brightness** - represents the brightness of an image object.
    - **Band mean arithmetic** – adds, multiplies, divides or subtracts the mean values of two given image bands.
    - **Amplitude value** - represents the difference between the maximum and minimum pixel values of an image object for the given image layer.
    - **Standard deviation** - the standard deviation represents the numerical data dispersion degree of pixel values surrounding the mean for a given image band.

- **Spatial Resolve** – resolves spatial conflicts between image objects with different geometries.

- **Topological Features**
    - **Number of** – number of neighboring objects of an image object that belong to the given class.

  - **Border to** – border length (in meters) an image object shares with neighboring objects of a given class.
  - **Relative border to** – the ratio between the border length an image object shares with neighboring objects of a given class and the object's area.
  - **Area of** – the total area (in meters) of neighboring objects that belong to a given class.
  - **Relative area of** – the ratio between the total area of neighboring objects that belong to a given class and the object's own area.

F.
Recursive operations

- **Merge Neighbors** – merges the connected image objects that belong to the same class.

G.
Hierarchical features

- **Min** – computes the minimum value of a specific feature for all child objects that belong to a given class.
- **Max** – computes the maximum value of a specific feature for all child objects that belong to a given class.
- **Count** – computes the number of child objects that belong to a given class.
- **Mean** – computes the mean value of a specific feature of for all child objects that belong to a given class.
- **Sum** – computes the sum of the values of a specific feature for all child objects that belong to a given class.

H.
Land cover operators

Operator 1

Defines

```
DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://s3.amazonaws.com/…/inte
rimage/project/512/resources/tiles.ser','single','0.6000000237999484','negative','id');
```

Job #1 – Map

```
load_1 = LOAD
's3n://…/interimage/project/512/resources/shapes/segmentation_vegetation.json' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

selection_1 = FILTER load_1 BY II_IsValid(geometry, properties, '');

projection_1 = FOREACH selection_1 GENERATE geometry,
II_ToProps(II_CalculateTiles(geometry, properties#'tile'),'tile',properties) AS
properties;

projection_2 = FOREACH projection_1 GENERATE geometry,
II_ToProps('None','class',properties) AS properties;

projection_3 = FOREACH projection_2 GENERATE geometry,
II_ToProps(0.3,'membership',properties) AS properties;

STORE projection_3 INTO 's3n://…/interimage/project/512/results/op1_segmentation' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 43: Pig Latin script for Operator 1.

Operator 1 is compiled into a single map-only job. This operator imports a JSON file with the segmentation for vegetated areas. After that, invalid geometries are filtered out and object tiles, class names and membership values are set.

Operator 2

Defines

```
DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://s3.amazonaws.com/…/inte
rimage/project/512/resources/tiles.ser','single','0.6000000237999484','negative','id');
```

Job #2 – Map

```
load_1 = LOAD
's3n://…/interimage/project/512/resources/shapes/segmentation_non_vegetation.json' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

selection_1 = FILTER load_1 BY II_IsValid(geometry, properties, '');

projection_1 = FOREACH selection_1 GENERATE geometry,
II_ToProps(II_CalculateTiles(geometry, properties#'tile'),'tile',properties) AS
properties;
```

```
projection_2 = FOREACH projection_1 GENERATE geometry,
II_ToProps('None','class',properties) AS properties;

projection_3 = FOREACH projection_2 GENERATE geometry,
II_ToProps(0.3,'membership',properties) AS properties;

STORE projection_3 INTO 's3n://…/interimage/project/512/results/op2_segmentation' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 44: Pig Latin script for Operator 2.

Operator 2’s Pig Latin script is similar to Operator 1’s. The only difference is that the segmentation imported is meant for non-vegetated areas.

Operator 3

Defines

```
DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://s3.amazonaws.com/…/inte
rimage/project/512/resources/tiles.ser','single','0.6000000237999484','negative','id');
```

Job #3 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/resources/shapes/segmentation_shadow.json'
USING org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

selection_1 = FILTER load_1 BY II_IsValid(geometry, properties, '');

projection_1 = FOREACH selection_1 GENERATE geometry,
II_ToProps(II_CalculateTiles(geometry, properties#'tile'),'tile',properties) AS
properties;

projection_2 = FOREACH projection_1 GENERATE geometry,
II_ToProps('Shadow','class',properties) AS properties;

projection_3 = FOREACH projection_2 GENERATE geometry,
II_ToProps(0.3,'membership',properties) AS properties;

STORE projection_3 INTO 's3n://…/interimage/project/512/results/op3_shadow' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 45: Pig Latin script for Operator 3.

Operator 3’s Pig Latin script is similar to the previous two. The imported segmentation is the result of the thresholding procedure performed on InterIMAGE and the objects are classified as hypotheses of the *Shadow* class.

Operator 4

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');
```

```
DEFINE SpectralFeatures
br.puc_rio.ele.lvc.interimage.data.udf.SpectralFeatures('https://s3.amazonaws.com/…/interi
mage/project/512/resources/images/','mean2 = mean(image_layer2);mean3 =
mean(image_layer3);ratio4 = ratio(image_layer4);',
'https://s3.amazonaws.com/…/interimage/project/512/resources/tilenames.ser');

DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://s3.amazonaws.com/…/inte
rimage/project/512/resources/tiles.ser','multiple','0.6000000237999484','negative','id');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('BareSoil,Blue,Bright,BrightGrey,Ceramic
Roof,Dark,Grass,Grey,Pools,Shadow,Trees');
```

Job #4 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op1_segmentation' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

SF_B = FOREACH load_1 GENERATE geometry,  II_ToProps(II_CalculateTiles(geometry,
properties#'tile'),'tile',properties) AS properties;

SF_C = FOREACH SF_B GENERATE FLATTEN(II_Replicate(geometry, properties)) AS
(geometry:chararray, properties:map[]);

SF_D = GROUP SF_C BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #4 – Reduce

```
SF_E = FOREACH SF_D GENERATE FLATTEN(SpectralFeatures(SF_C)) AS (geometry:chararray,
properties:map[]);
```

Job #5 – Map

```
SF_F = GROUP SF_E BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #5 – Reduce

```
group_1 = FOREACH SF_F GENERATE FLATTEN(II_CombineSpectralFeatures(SF_E)) AS
(geometry:chararray, properties:map[]);

selection_1 = FILTER group_1 BY II_IsValid(null, properties, 'mean2,mean3,ratio4');

selection_2 = FILTER selection_1 BY properties#'ratio4' > 0.2988;

projection_1 = FOREACH selection_2 GENERATE geometry,
II_ToClassification('Grass',II_Min(II_Membership('ml2grass',properties#'mean2'),
II_Membership('ml3grass',properties#'mean3')),properties) AS properties;

projection_2 = FOREACH projection_1 GENERATE geometry,
II_ToClassification('Trees',II_Min(II_Membership('ml2trees',properties#'mean2'),
II_Membership('ml3trees',properties#'mean3')),properties) AS properties;

projection_3 = FOREACH projection_2 GENERATE geometry, II_Classify(properties) AS
properties;

STORE projection_3 INTO 's3n://…/interimage/project/512/results/op4_vegetation' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 46: Pig Latin script for Operator 4.

In this script, image segments are classified between *Grass* and *Trees*. The script was obtained by translating the original decision rule to Pig Latin. The general procedures of the script are the spectral features computation as presented in section 4.5.2 and the classification of the objects based on the computed features and given fuzzy sets.

In the map phase of Job 4, the segmentation imported in Operator 1 is read and the segments are replicated and grouped by tile. In the reduce phase, the spectral features are calculated for each image object.

In the map phase of Job 5, objects are regrouped by tile (original tile). After that, in the reduce phase, objects' spectral features are combined and the membership values are computed based on the fuzzy sets. Then, the final class is computed.

Operator 5

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');

DEFINE SpectralFeatures
br.puc_rio.ele.lvc.interimage.data.udf.SpectralFeatures('https://s3.amazonaws.com/…/interi
mage/project/512/resources/images/','brightness = brightness(image);mean1 =
mean(image_layer1);bandMeanDiv31 = bandMeanDiv(image_layer3,image_layer1);maxPixVal1 =
maxPixelValue(image_layer1);ratio2 = ratio(image_layer2);',
'https://s3.amazonaws.com/…/interimage/project/512/resources/tilenames.ser');

DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://s3.amazonaws.com/…/inte
rimage/project/512/resources/tiles.ser','multiple','0.6000000237999484','negative','id');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('BareSoil,Blue,Bright,BrightGrey,Ceramic
Roof,Dark,Grass,Grey,Pools,Shadow,Trees');
```

Job #6 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op2_segmentation' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

SF_B = FOREACH load_1 GENERATE geometry,  II_ToProps(II_CalculateTiles(geometry,
properties#'tile'),'tile',properties) AS properties;

SF_C = FOREACH SF_B GENERATE FLATTEN(II_Replicate(geometry, properties)) AS
(geometry:chararray, properties:map[]);

SF_D = GROUP SF_C BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #6 – Reduce

```
SF_E = FOREACH SF_D GENERATE FLATTEN(SpectralFeatures(SF_C)) AS (geometry:chararray,
properties:map[]);
```

Job #7 – Map

```
SF_F = GROUP SF_E BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #7 – Reduce

```
group_1 = FOREACH SF_F GENERATE FLATTEN(II_CombineSpectralFeatures(SF_E)) AS
(geometry:chararray, properties:map[]);

selection_1 = FILTER group_1 BY II_IsValid(null, properties,
'brightness,mean1,bandMeanDiv31,maxPixVal1,ratio2');

wfeatures_1 = FOREACH selection_1 GENERATE geometry,
II_ToProps(II_Area(geometry),'area',properties) AS properties;

projection_1 = FOREACH wfeatures_1 GENERATE geometry, ( CASE WHEN properties#'brightness'
> 208.0 THEN II_ToClassification('Bright', 1.0, properties) ELSE properties END ) AS
properties;

projection_2 = FOREACH projection_1 GENERATE geometry, ( CASE WHEN (properties#'ratio2' <
0.2976) OR ((properties#'ratio2' > 0.2976) AND (properties#'maxPixVal1' < 116.0)) THEN
II_ToClassification('Dark',II_Min(II_Membership('ml1dark',properties#'mean1'),
II_Membership('bdark',properties#'brightness')),properties) ELSE properties END ) as
properties;

projection_3 = FOREACH projection_2 GENERATE geometry, ( CASE WHEN (properties#'ratio2' <
0.2976) OR ((properties#'ratio2' > 0.2976) AND (properties#'maxPixVal1' < 116.0)) THEN
II_ToClassification('Grey',II_Min(II_Membership('ml1grey',properties#'mean1'),
II_Membership('bgrey',properties#'brightness')),properties) ELSE properties END ) AS
properties;

projection_4 = FOREACH projection_3 GENERATE geometry, ( CASE WHEN (properties#'ratio2' <
0.2976) OR ((properties#'ratio2' > 0.2976) AND (properties#'maxPixVal1' < 116.0)) THEN
II_ToClassification('BrightGrey',II_Min(II_Membership('ml1brightgrey',properties#'mean1'),
II_Membership('bbrightgrey',properties#'brightness')),properties) ELSE properties END ) as
properties;

projection_5 = FOREACH projection_4 GENERATE geometry, ( CASE WHEN (properties#'ratio2' <
0.355) AND (properties#'brightness' < 208.0) AND (properties#'bandMeanDiv31' < 1.5) AND
(properties#'maxPixVal1' > 116.0) AND (properties#'ratio2' > 0.2976) THEN
II_ToClassification('Blue', 0.0, properties) ELSE properties END ) as properties;

projection_6 = FOREACH projection_5 GENERATE geometry, ( CASE WHEN (properties#'ratio2' >
0.355) OR ((properties#'ratio2' < 0.355) AND (properties#'area' < 80.0)) THEN
II_ToClassification('Pools', 0.0, properties) ELSE properties END ) AS properties;

selection_2 = FILTER projection_6 BY II_HasClassification(properties);

projection_7 = FOREACH selection_2 GENERATE geometry, II_Classify(properties) AS
properties;

STORE projection_7 INTO 's3n://…/interimage/project/512/results/op5_other_classes' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 47: Pig Latin script for Operator 5.

In this script, image segments are classified in *Bright*, *Dark*, *Grey*, *BrightGrey*, *Blue* and *Pools* classes. The general procedures of the script are the spectral features computation as presented in section 4.5.2 and the classification of the objects based on the computed features and given fuzzy sets.

In the map phase of Job 6, the segmentation imported in Operator 2 is read and the segments are replicated and grouped by tile. In the reduce phase, the spectral features are calculated for each segment.

In the map phase of Job 7, objects are regrouped by tile (original tile). After that, in the reduce phase, objects' spectral features are combined and the membership values are computed based on the fuzzy sets. Then, the final class is computed.

Operator 6

Defines

```
DEFINE SpectralFeatures
br.puc_rio.ele.lvc.interimage.data.udf.SpectralFeatures('https://s3.amazonaws.com/…/interi
mage/project/512/resources/images/','bandMeanDiv31 =
bandMeanDiv(image_layer3,image_layer1);','https://s3.amazonaws.com/…/interimage/project/51
2/resources/tilenames.ser');

DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://s3.amazonaws.com/…/inte
rimage/project/512/resources/tiles.ser','multiple','0.6000000237999484','negative','id');

DEFINE SpatialResolve
br.puc_rio.ele.lvc.interimage.data.udf.SpatialResolve('0.3600000285599387','https://s3.ama
zonaws.com/…/interimage/project/512/resources/images/','image','https://s3.amazonaws.com/…
/interimage/project/512/resources/tilenames.ser','BareSoil,Blue,Bright,BrightGrey,CeramicR
oof,Dark,Grass,Grey,Pools,Shadow,Trees');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('BareSoil,Blue,Bright,BrightGrey,Ceramic
Roof,Dark,Grass,Grey,Pools,Shadow,Trees');
```

Job #8 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op2_segmentation' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

SF_B = FOREACH load_1 GENERATE geometry,  II_ToProps(II_CalculateTiles(geometry,
properties#'tile'),'tile',properties) AS properties;

SF_C = FOREACH SF_B GENERATE FLATTEN(II_Replicate(geometry, properties)) AS
(geometry:chararray, properties:map[]);

SF_D = GROUP SF_C BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #8 – Reduce

```
SF_E = FOREACH SF_D GENERATE FLATTEN(SpectralFeatures(SF_C)) AS (geometry:chararray,
properties:map[]);
```

Job #9 – Map

```
SF_F = GROUP SF_E BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #9 – Reduce

```
group_1 = FOREACH SF_F GENERATE FLATTEN(II_CombineSpectralFeatures(SF_E)) AS
(geometry:chararray, properties:map[]);

selection_1 = FILTER group_1 BY II_IsValid(null, properties, 'bandMeanDiv31');

selection_2 = FILTER selection_1 BY properties#'bandMeanDiv31' > 1.5;

projection_1 = FOREACH selection_2 GENERATE geometry,  II_ToClassification('CeramicRoof',
0.8, properties) AS properties;
```

```
ceramicroof_1 = FOREACH projection_1 GENERATE geometry, II_Classify(properties) as
properties;
```

Job #10 – Map

```
load_2 = LOAD 's3n://…/interimage/project/512/resources/shapes/bare_soil' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

selection_3 = FILTER load_2 BY II_Area(geometry) >= 0.3600000285599387;

projection_2 = FOREACH selection_3 GENERATE geometry, II_ToClassification('BareSoil', 1.0,
properties) AS properties;

baresoil_1 = FOREACH projection_2 GENERATE geometry, II_Classify(properties) AS
properties;

union_1 = UNION ceramicroof_1, baresoil_1;

SF_B = FOREACH union_1 GENERATE geometry,  II_ToProps(II_CalculateTiles(geometry,
properties#'tile'),'tile',properties) AS properties;

SF_C = FOREACH SF_B GENERATE FLATTEN(II_Replicate(geometry, properties)) AS
(geometry:chararray, properties:map[]);

SF_D = GROUP SF_C BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #10 – Reduce

```
SF_E = FOREACH SF_D GENERATE FLATTEN(SpatialResolve(SF_C)) AS (geometry:chararray,
properties:map[]);
```

Job #11 – Map

```
SF_F = GROUP SF_E BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #11 – Reduce

```
group_2 = FOREACH SF_F GENERATE FLATTEN(II_MergeResolved(SF_E)) AS (geometry:chararray,
properties:map[]);

STORE group_2 INTO 's3n://…/interimage/project/512/results/op6_ceramic_roof' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 48: Pig Latin script for Operator 6.

In this script, image segments are classified between *BareSoil* and *CeramicRoof*. The general procedures of this script are the spectral features and spatial resolve computations as presented in section 4.5.2 and the classification of the objects based on the computed features.

In the map phase of Job 8, the segmentation imported in Operator 2 is read and the segments are replicated and grouped by tile. In the reduce phase, the spectral features are calculated for each segment.

In the map phase of Job 9, objects are regrouped by tile (original tile). After that, in the reduce phase, objects’ spectral features are combined and *CeramicRoof* objects are classified.

In the map phase of Job 10, the auxiliary shapefile with *BareSoil* objects is imported. The objects are classified and joined with *CeramicRoof* objects. Then, all objects are replicated and grouped by tile. In the reduce phase, spatial conflicts are resolved.

Finally, in the map phase of Job 11 objects are regrouped by tile (original tile) and, in the reduce phase, resolved objects are merged.

Operator 7

Defines

```
DEFINE II_SelectClass
br.puc_rio.ele.lvc.interimage.common.udf.SelectClass('https://s3.amazonaws.com/.../interim
age/project/512/resources/semanticnetwork.ser');

DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://s3.amazonaws.com/…/inte
rimage/project/512/resources/tiles.ser','multiple','0.6000000237999484','negative','id');

DEFINE SpatialResolve
br.puc_rio.ele.lvc.interimage.data.udf.SpatialResolve('0.3600000285599387','https://s3.ama
zonaws.com/…/interimage/project/512/resources/images/','image','https://s3.amazonaws.com/…
/interimage/project/512/resources/tilenames.ser','BareSoil,Blue,Bright,BrightGrey,CeramicR
oof,Dark,Grass,Grey,Pools,Shadow,Trees');
```

Job #12 – Map

```
load_1 = LOAD
's3n://…/interimage/project/512/results/op4_vegetation,s3n://…/interimage/project/512/resu
lts/op5_other_classes,s3n://…/interimage/project/512/results/op3_shadow,s3n://…/interimage
/project/512/results/op6_ceramic_roof' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry,  ( CASE WHEN
II_SelectClass(properties#'class','Shadow') THEN II_ToProps(1.0, 'membership', properties)
ELSE properties END ) AS properties;

projection_2 = FOREACH projection_1 GENERATE geometry,  ( CASE WHEN
II_SelectClass(properties#'class','Red') THEN II_ToProps(0.8, 'membership', properties)
ELSE properties END ) AS properties;

projection_3 = FOREACH projection_2 GENERATE geometry,  ( CASE WHEN
II_SelectClass(properties#'class','OtherClasses') THEN II_ToProps(0.4, 'membership',
properties) ELSE properties END ) AS properties;

projection_4 = FOREACH projection_3 GENERATE geometry,  ( CASE WHEN
II_SelectClass(properties#'class','Vegetation') THEN II_ToProps(0.6, 'membership',
properties) ELSE properties END ) AS properties;

SF_B = FOREACH projection_4 GENERATE geometry,  II_ToProps(II_CalculateTiles(geometry,
properties#'tile'),'tile',properties) AS properties;

SF_C = FOREACH SF_B GENERATE FLATTEN(II_Replicate(geometry, properties)) AS
(geometry:chararray, properties:map[]);

SF_D = GROUP SF_C BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #12 – Reduce

```
SF_E = FOREACH SF_D GENERATE FLATTEN(SpatialResolve(SF_C)) AS (geometry:chararray,
properties:map[]);
```

Job #13 – Map

```
SF_F = GROUP SF_E BY properties#'tile' PARTITION BY
br.puc_rio.ele.lvc.interimage.common.TilePartitioner PARALLEL $PARALLEL;
```

Job #13 – Reduce

```
group_1 = FOREACH SF_F GENERATE FLATTEN(II_MergeResolved(SF_E)) AS (geometry:chararray,
properties:map[]);

STORE group_1 INTO 's3n://…/interimage/project/512/results/op6_all' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 49: Pig Latin script for Operator 7.

In this script, all objects from operators 3, 4, 5 and 6 are read and classified according to a crisp membership value. After that, spatial conflicts are resolved (as seen in section 4.5.2).

In the map phase of Job 12, the objects generated by operators 3, 4, 5 and 6 are read, their membership values are set and the objects are replicated and grouped by tile. In the reduce phase, spatial conflicts are resolved.

Finally, in the map phase of Job 13 objects are regrouped by tile (original tile) and, in the reduce phase, resolved objects are merged.

I. Land use operators

Operator 1

Defines

```
DEFINE II_MergeNeighbors
br.puc_rio.ele.lvc.interimage.geometry.udf.MergeNeighbors('BigBlue,BigBright,BigBrightGrey
,BigDark,BigRoofs,BigGrey,BuildingShadow,CeramicRoof,Pools,VariousRoofs,Vegetation');

DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://…/interimage/project/51
2/resources/tiles.ser','single','0.6000000237999484','negative','id');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('BigBlue,BigBright,BigBrightGrey,BigDark
,BigRoofs,BigGrey,BuildingShadow,CeramicRoof,Pools,VariousRoofs,Vegetation');

DEFINE II_SpatialFilter
br.puc_rio.ele.lvc.interimage.geometry.udf.SpatialFilter('https://…/interimage/project/512
/resources/shapes/blocks.ser','https://…/interimage/project/512/resources/tiles.ser','inte
rsection','id');

DEFINE II_SpatialClip
br.puc_rio.ele.lvc.interimage.geometry.udf.SpatialClip('https://…/interimage/project/512/r
esources/shapes/blocks.ser','https://…/interimage/project/512/resources/tiles.ser','0.3600
000285599387','0.6000000237999484','id');
```

Job #1 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op6_all' USING
```

```
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry, data,
II_ToProps(II_CalculateTiles(geometry, properties#'tile'),'tile',properties) AS
properties;

selection_1 = FILTER projection_1 BY II_SpatialFilter(geometry, properties#'tile');

projection_2 = FOREACH selection_1 GENERATE FLATTEN(II_SpatialClip(geometry, data,
properties)) AS (geometry:chararray, data:map[chararray], properties:map[]);

group_1 = GROUP projection_2 BY properties#'parent';
```

Job #1 – Reduce

```
merged_1 = FOREACH group_1 GENERATE FLATTEN(II_MergeNeighbors(projection_2)) AS
(geometry:chararray, data:map[chararray], properties:map[]);

wfeatures_1 = FOREACH merged_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) as properties;
```

Job #2 – Map

```
projection_3 = FOREACH wfeatures_1 GENERATE geometry, data, ( CASE WHEN (properties#'area'
> 200.0) AND (properties#'class' == 'Shadow') THEN
II_ToClassification('BuildingShadow',1.0,properties) ELSE properties END ) as properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data, ( CASE WHEN
(properties#'area' > 1900.0) AND (properties#'class' == 'Blue') THEN
II_ToClassification('BigBlue',1.0,properties) ELSE properties END ) as properties;

projection_5 = FOREACH projection_4 GENERATE geometry, data, ( CASE WHEN
(properties#'area' > 1900.0) AND (properties#'class' == 'Bright') THEN
II_ToClassification('BigBright',1.0,properties) ELSE properties END ) as properties;

projection_6 = FOREACH projection_5 GENERATE geometry, data, ( CASE WHEN
(properties#'area' > 1900.0) AND (properties#'class' == 'BrightGrey') THEN
II_ToClassification('BigBrightGrey',1.0,properties) ELSE properties END ) as properties;

projection_7 = FOREACH projection_6 GENERATE geometry, data, ( CASE WHEN
(properties#'area' > 1900.0) AND (properties#'class' == 'Grey') THEN
II_ToClassification('BigGrey',1.0,properties) ELSE properties END ) as properties;

projection_8 = FOREACH projection_7 GENERATE geometry, data, ( CASE WHEN
(properties#'area' > 1900.0) AND (properties#'class' == 'Dark') THEN
II_ToClassification('BigDark',1.0,properties) ELSE properties END ) as properties;

projection_9 = FOREACH projection_8 GENERATE geometry, data, ( CASE WHEN
II_SelectClass(properties#'class','Vegetation') THEN
II_ToClassification('Vegetation',1.0,properties) ELSE properties END ) as properties;

projection_10 = FOREACH projection_9 GENERATE geometry, data, ( CASE WHEN
properties#'class' == 'CeramicRoof' THEN II_ToClassification('CeramicRoof',1.0,properties)
ELSE properties END ) as properties;

projection_11 = FOREACH projection_10 GENERATE geometry, data, ( CASE WHEN
properties#'class' == 'Pools' THEN II_ToClassification('Pools',1.0,properties) ELSE
properties END ) as properties;

selection_2 = FILTER projection_11 BY II_HasClassification(properties);

others_1 = FOREACH selection_2 GENERATE geometry, data, II_Classify(properties) AS
properties;

selection_3 = FILTER wfeatures_1 BY (properties#'area' > 1900.0) AND ((properties#'class'
== 'Blue') OR (properties#'class' == 'Bright') OR (properties#'class' == 'BrightGrey') OR
(properties#'class' == 'Grey') OR (properties#'class' == 'Dark'));

big_roofs_1 = FOREACH selection_3 GENERATE geometry, data,
II_ToClassification('BigRoofs',1.0,properties) AS properties;
```

```
big_roofs_2 = FOREACH big_roofs_1 GENERATE geometry, data, II_Classify(properties) AS
properties;

selection_4 = FILTER wfeatures_1 BY (properties#'area' <= 1900.0) AND ((properties#'class'
== 'Blue') OR (properties#'class' == 'Bright') OR (properties#'class' == 'BrightGrey') OR
(properties#'class' == 'Grey') OR (properties#'class' == 'Dark'));

various_roofs_1 = FOREACH selection_4 GENERATE geometry, data,
II_ToClassification('VariousRoofs',1.0,properties) AS properties;

various_roofs_2 = FOREACH various_roofs_1 GENERATE geometry, data, II_Classify(properties)
AS properties;

union_1 = UNION others_1, big_roofs_2, various_roofs_2;

STORE union_1 INTO 's3n://…/interimage/project/512/results/op7_new_classes' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 50: Pig Latin script for Operator 1.

In the map phase of Job 1, the objects coming from the land cover classification are read, clipped and grouped by city block. In the reduce phase, the neighboring objects belonging to the same class are merged.

In the map phase of Job 2, objects are replicated in new classes.

Operator 2

Defines

```
DEFINE II_AggregationFeatures
br.puc_rio.ele.lvc.interimage.geometry.udf.AggregationFeatures('max_area_pools =
max(area,Pools);max_rect_building_shadow =
max(rectangle_fit,BuildingShadow);sum_area_vegetation =
sum(area,Vegetation);sum_area_ceramic_roof = sum(area,CeramicRoof);count_building_shadow =
count(BuildingShadow);count_ceramic_roof = count(CeramicRoof);sum_area_various_roofs =
sum(area,VariousRoofs);max_rect_big_blue = max(rectangle_fit,BigBlue);max_rect_big_bright
= max(rectangle_fit,BigBright);max_rect_big_bright_grey =
max(rectangle_fit,BigBrightGrey);max_rect_big_grey =
max(rectangle_fit,BigGrey);max_rect_big_dark = max(rectangle_fit,BigDark);count_big_roofs
= count(BigRoofs);mean_rect_big_roofs = mean(rectangle_fit,BigRoofs)');

DEFINE II_CalculateTiles
br.puc_rio.ele.lvc.interimage.geometry.udf.CalculateTiles('https://…/interimage/project/51
2/resources/tiles.ser','single','0.6000000237999484','negative','id');
```

Job #3 – Map

```
blocks_1 = LOAD 's3n://…/interimage/project/512/resources/shapes/blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

blocks_2 = FOREACH blocks_1 GENERATE geometry, data,
II_ToProps(II_CalculateTiles(geometry, properties#'tile'), 'tile', properties) AS
properties;

load_1 = LOAD 's3n://…/interimage/project/512/results/op7_new_classes' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

group_1 = COGROUP load_1 BY properties#'parent', blocks_2 BY properties#'iiuuid' PARALLEL
8;
```

Job #3 – Reduce

```
projection_1 = FOREACH group_1 GENERATE FLATTEN(II_AggregationFeatures(blocks_2, load_1))
AS (geometry:chararray, data:map[chararray], properties:map[]);

STORE projection_1 INTO 's3n://…/interimage/project/512/results/op8_blocks' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 51: Pig Latin script for Operator 2.

In the map phase of Job 3, objects and city blocks are spatially joined. In the reduce phase, the hierarchical features are computed (section 4.5.5).

Operator 3

Defines

```
DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #4 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

selection_1 = FILTER load_1 BY (properties#'max_area_pools' > 2000.0);

projection_1 = FOREACH selection_1 GENERATE geometry, data,
II_ToClassification('SportClubs',1.0,properties) AS properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data, II_Classify(properties) AS
properties;

STORE projection_2 INTO 's3n://…/interimage/project/512/results/op9_sport_clubs' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 52: Pig Latin script for Operator 3.

In the map phase of Job 4, city blocks with total area of pools greater than 2,000 are classified as Sport Clubs.

Operator 4

Defines

```
DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');

DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');
```

Job #5 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

selection_1 = FILTER load_1 BY (properties#'max_rect_building_shadow' > 0.7);

projection_1 = FOREACH selection_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) AS properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) AS properties;

projection_3 = FOREACH projection_2 GENERATE geometry, data,
II_ToClassification('VerticalServices',II_Membership('rel_area_veg_vs',properties#'rel_are
a_vegetation'),properties) AS properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data, II_Classify(properties) AS
properties;

STORE projection_4 INTO 's3n://…/interimage/project/512/results/op10_vertical_services'
USING br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 53: Pig Latin script for Operator 4.

In the map phase of Job 5, city blocks are filtered by their maximum rectangularity of building shadows and the relative area of vegetation is used as input of a fuzzy set to compute the membership values for the class Vertical Services.

Operator 5

Defines

```
DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');

DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');
```

Job #6 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

selection_1 = FILTER load_1 BY (properties#'max_rect_building_shadow' > 0.7);

projection_1 = FOREACH selection_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) AS properties;
```

```
projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) AS properties;

projection_3 = FOREACH projection_2 GENERATE geometry, data,
II_ToClassification('VerticalResidentialHighStandard',II_Membership('rel_area_veg_vrhs',pr
operties#'rel_area_vegetation'),properties) AS properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data, II_Classify(properties) AS
properties;

STORE projection_4 INTO
's3n://…/interimage/project/512/results/op11_vertical_residential_high_standard' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 54: Pig Latin script for Operator 5.

In the map phase of Job 6, city blocks are filtered by their maximum rectangularity of building shadows and the relative area of vegetation is used as input of a fuzzy set to compute the membership values for the class Vertical residential of High Standard.

Operator 6

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #7 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) as properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) as properties;

projection_3 = FOREACH projection_2 GENERATE geometry, data, ( CASE WHEN
properties#'rel_area_vegetation' > 0.9 THEN
II_ToClassification('PartiallyUnoccupiedLand',1.0,properties) ELSE properties END ) as
properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data, ( CASE WHEN
(properties#'rel_area_vegetation' <= 0.9) AND (properties#'sum_area_vegetation' >
120000.0) THEN
II_ToClassification('PartiallyUnoccupiedLand',II_Membership('rel_area_veg_pul',properties#
'rel_area_vegetation'),properties) ELSE properties END ) as properties;

selection_1 = FILTER projection_4 BY II_HasClassification(properties);
```

```
projection_5 = FOREACH selection_1 GENERATE geometry, data, II_Classify(properties) as
properties;

STORE projection_5 INTO
's3n://…/interimage/project/512/results/op12_partially_unoccupied_land' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 55: Pig Latin script for Operator 6.

In the map phase of Job 7, city block objects are read and attributes related to the area of vegetation objects are computed. After that, these attributes are used as input of a fuzzy set that computes the membership values for the class Partially Unoccupied Land.

Operator 7

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #8 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) as properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_ceramic_roof' /
properties#'area','rel_area_ceramic_roof',properties) as properties;

projection_3 = FILTER projection_2 BY (properties#'rel_area_ceramic_roof' > 0.3);

projection_4 = FOREACH projection_3 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) as properties;

projection_5 = FOREACH projection_4 GENERATE geometry, data,
II_ToClassification('HorizontalResidentialLowStandard',II_Membership('rel_area_veg_hrls',p
roperties#'rel_area_vegetation'),properties) as properties;

projection_6 = FOREACH projection_5 GENERATE geometry, data, II_Classify(properties) as
properties;

STORE projection_6 INTO
's3n://…/interimage/project/512/results/op13_horizontal_residential_low_stantard' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 56: Pig Latin script for Operator 7.

In the map phase of Job 8, city block objects are read and attributes related to the area of vegetation and ceramic roof objects are computed. After that, these attributes, together with other attributes computed in Operator 2, are used as input of a fuzzy set that computes the membership values for the class Horizontal Residential of Low Standard.

Operator 8

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #9 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) as properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) as properties;

projection_3 = FOREACH projection_2 GENERATE geometry, data,
II_ToClassification('HorizontalResidentialHighStandard',II_Min(II_Membership('rel_area_veg
_hrhs',properties#'rel_area_vegetation'),II_Membership('number_building_shadow_hrhs',prope
rties#'count_building_shadow'),II_Membership('number_ceramic_roof_hrhs',properties#'count_
ceramic_roof')),properties) as properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data, II_Classify(properties) as
properties;

STORE projection_4 INTO
's3n://…/interimage/project/512/results/op14_horizontal_residential_high_stantard' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 57: Pig Latin script for Operator 8.

In the map phase of Job 9, city block objects are read and attributes related to the area of vegetation objects are computed. After that, these attributes, together with other attributes computed in Operator 2, are used as input of a number of fuzzy sets to compute the membership values for the class Horizontal Residential of High Standard.

Operator 9

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #10 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) as properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_ceramic_roof' /
properties#'area','rel_area_ceramic_roof',properties) as properties;

projection_3 = FOREACH projection_2 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) as properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data,
II_ToProps(properties#'sum_area_various_roofs' /
properties#'area','rel_area_various_roofs',properties) as properties;

projection_5 = FOREACH projection_4 GENERATE geometry, data,
II_ToClassification('MixedResidentialServices',II_Min(II_Membership('rel_area_veg_mrs',pro
perties#'rel_area_vegetation'),
II_Mean(II_Membership('rel_area_ceramic_roof_mrs',properties#'rel_area_ceramic_roof'),
II_Membership('rel_area_various_roofs_mrs',properties#'rel_area_various_roofs'))),properti
es) as properties;

projection_6 = FOREACH projection_5 GENERATE geometry, data, II_Classify(properties) as
properties;

STORE projection_6 INTO
's3n://…/interimage/project/512/results/op15_mixed_residential_services' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 58: Pig Latin script for Operator 9.

In the map phase of Job 10, city block objects are read and attributes related to the area of vegetation, various roofs and ceramic roof objects are computed. After that, these attributes are used as input of a number of fuzzy sets to compute the membership values for the class Mixed Residential Services.

Operator 10

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');

DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #11 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) as properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) as properties;

projection_3 = FOREACH projection_2 GENERATE geometry, data,
II_ToProps(properties#'sum_area_ceramic_roof' /
properties#'area','rel_area_ceramic_roof',properties) as properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data,
II_ToProps(properties#'sum_area_various_roofs' /
properties#'area','rel_area_various_roofs',properties) as properties;

projection_5 = FOREACH projection_4 GENERATE geometry, data,
II_ToClassification('IndustrialServices',II_Min(II_Max(II_Membership('max_rect_big_blue_is
',properties#'max_rect_big_blue'),II_Membership('max_rect_big_bright_is',properties#'max_r
ect_big_bright'),II_Membership('max_rect_big_bright_grey_is',properties#'max_rect_big_brig
ht_grey'),II_Membership('max_rect_big_grey_is',properties#'max_rect_big_grey'),II_Membersh
ip('max_rect_big_dark_is',properties#'max_rect_big_dark'),II_Membership('rel_area_various_
roofs_is',properties#'rel_area_various_roofs')),II_Max(II_Membership('max_rect_building_sh
adow_is',properties#'max_rect_building_shadow'),II_Membership('inexistence_building_shadow
_is',properties#'count_building_shadow')),II_Membership('rel_area_ceramic_roof_is',propert
ies#'rel_area_ceramic_roof'),II_Membership('rel_area_veg_is',properties#'rel_area_vegetati
on')),properties) as properties;

projection_6 = FOREACH projection_5 GENERATE geometry, data, II_Classify(properties) as
properties;

STORE projection_6 INTO 's3n://…/interimage/project/512/results/op16_industrial_services'
USING br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 59: Pig Latin script for Operator 10.

In the map phase of Job 11, city block objects are read and attributes related to the area of vegetation and ceramic roof objects are computed. After that, these attributes, together with other attributes computed in Operator 2, are used as input of a number of fuzzy sets to compute the membership values for the class Industrial Services.

Operator 11

Defines

```
DEFINE II_Membership
br.puc_rio.ele.lvc.interimage.datamining.udf.Membership('https://s3.amazonaws.com/…/interi
mage/project/512/resources/fuzzysets.ser');
```

```
DEFINE II_Classify
br.puc_rio.ele.lvc.interimage.common.udf.Classify('Favelas,HorizontalResidentialHighStanda
rd,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServices,PartiallyU
noccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #12 – Map

```
load_1 = LOAD 's3n://…/interimage/project/512/results/op8_blocks' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

projection_1 = FOREACH load_1 GENERATE geometry, data,
II_ToProps(II_Area(geometry),'area',properties) as properties;

projection_2 = FOREACH projection_1 GENERATE geometry, data,
II_ToProps(properties#'sum_area_vegetation' /
properties#'area','rel_area_vegetation',properties) as properties;

projection_3 = FOREACH projection_2 GENERATE geometry, data,
II_ToProps(properties#'sum_area_ceramic_roof' /
properties#'area','rel_area_ceramic_roof',properties) as properties;

projection_4 = FOREACH projection_3 GENERATE geometry, data,
II_ToProps(properties#'sum_area_various_roofs' /
properties#'area','rel_area_various_roofs',properties) as properties;

projection_5 = FOREACH projection_4 GENERATE geometry, data,
II_ToClassification('Favelas',II_Min(II_Max(II_Min(II_Membership('inexistence_big_roofs_f'
,properties#'count_big_roofs'),II_Membership('rel_area_veg_2_f',properties#'rel_area_veget
ation'),II_Membership('rel_area_various_roofs_f',properties#'rel_area_various_roofs')),II_
Min(II_Membership('existence_big_roofs_f',properties#'count_big_roofs'),II_Membership('mea
n_rect_big_roofs_f',properties#'mean_rect_big_roofs'))),II_Max(II_Membership('max_rect_bui
lding_shadow_f',properties#'max_rect_building_shadow'),II_Membership('inexistence_building
_shadow_f',properties#'count_building_shadow')),II_Membership('rel_area_ceramic_roof_f',pr
operties#'rel_area_ceramic_roof'),II_Membership('rel_area_veg_f',properties#'rel_area_vege
tation')),properties) as properties;

projection_6 = FOREACH projection_5 GENERATE geometry, data, II_Classify(properties) as
properties;

STORE projection_6 INTO 's3n://…/interimage/project/512/results/op17_favelas' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 60: Pig Latin script for Operator 11.

In the map phase of Job 12, city block objects are read and the total area of vegetation, ceramic roofs and various roofs is computed. After that, these attributes, together with other attributes computed in Operator 2, are used as input of a number of fuzzy sets to compute the membership values for the class Favelas.

Operator 12

Defines

```
DEFINE II_SimpleSpatialResolve
br.puc_rio.ele.lvc.interimage.geometry.udf.SimpleSpatialResolve('Favelas,HorizontalResiden
tialHighStandard,HorizontalResidentialLowStandard,IndustrialServices,MixedResidentialServi
ces,PartiallyUnoccupiedLand,SportClubs,VerticalResidentialHighStandard,VerticalServices');
```

Job #13 – Map

```
load_1 = LOAD
's3n://…/interimage/project/512/results/op9_sport_clubs,s3n://…/interimage/project/512/res
ults/op10_vertical_services,s3n://…/interimage/project/512/results/op11_vertical_residenti
al_high_standard,s3n://…/interimage/project/512/results/op12_partially_unoccupied_land,s3n
://…/interimage/project/512/results/op13_horizontal_residential_low_stantard,s3n://…/inter
image/project/512/results/op14_horizontal_residential_high_stantard,s3n://…/interimage/pro
ject/512/results/op15_mixed_residential_services,s3n://…/interimage/project/512/results/op
16_industrial_services,s3n://…/interimage/project/512/results/op17_favelas' USING
org.apache.pig.builtin.JsonLoader('geometry:chararray, data:map[chararray],
properties:map[]');

group_1 = GROUP load_1 BY properties#'tile' PARALLEL 8;
```

Job #13 – Reduce

```
projection_1 = FOREACH group_1 GENERATE FLATTEN(II_SimpleSpatialResolve(load_1)) AS
(geometry:chararray, data:map[chararray], properties:map[]);

STORE projection_1 INTO 's3n://…/interimage/project/512/results/op18_all' USING
br.puc_rio.ele.lvc.interimage.common.udf.JsonStorage();
```

Figure 61: Pig Latin script for Operator 12.

In the map phase of Job 13, the city block objects from the previous operators are read and grouped by tile. In the reduce phase, the simple spatial resolve method is applied to resolve spatial conflicts.