TU Kaiserslautern & DFKI
Multimedia Analysis and Data Mining

Diplomarbeit

A Distributed System for Pattern Recognition and Machine Learning

Alexander Arimond

Mai 2010

Betreuer:
Prof. Dr. Andreas Dengel
&
Christian Kofler

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Textauszüge und Grafiken, die sinngemäß oder wörtlich aus veröffentlichten Schriften entnommen wurden, sind durch Referenzen gekennzeichnet.

Kaiserslautern, im Mai 2010

Alexander Arimond

Hiermit möchte ich mich bei allen bedanken, die mich während dieser Diplomarbeit unterstützt haben. Vielen Dank an Prof. Dr. Andreas Dengel, der es mir ermöglichte, diese Arbeit am DFKI durchführen zu können. Besonders danke ich Christian Kofler, der mir bei der Wahl des Themas half und meine Arbeit sehr gut betreut hat. Vielen Dank an Adrian Ulges, der mir bei einigen Machine Learning Angelegenheiten sehr weiterhelfen konnte. Danke an Damian Borth für viele hilfreiche Tips und andauernde Motivation, auch an Sonn- und Feiertagen. Ich danke auch Markus Fuchs für die Hilfe bei den technischen Problemen dieser Arbeit. Danke für viel Spass und Ablenkung, als auch für viel Geduld, an Rebecca, Eric, Nadja, Michael, Andi, Helge und alle guten Freunde, auch aus der Heimat. Der größte Dank gilt meinen Geschwistern Janina und Ralf, und besonders meinen Eltern, für den Rückhalt und die Unterstützung, nicht nur während meines Studiums.

Abstract

Pattern Recognition and Machine Learning techniques usually involve data- and compute-intensive methods. Applying such techniques therefore often is very time-consuming and requires expert knowledge. In this context, the state-of-the-art software RapidMiner already provides easy to use interfaces for developing and evaluating Pattern Recognition and Machine Learning applications. However, it has only limited support for parallelization and it lacks functionality to spread long-running computations over multiple machines. A solution to this is distributed computing with paradigms like MapReduce.

This thesis deals with the development and evaluation of a system which integrates distributed computing frameworks into RapidMiner. A special focus is put on utilizing MapReduce as a programming model. The software frameworks Hadoop, GridGain and Oracle Coherence are reviewed and evaluated with respect to their suitablility to fit into the context of RapidMiner. The developed system provides effective means for transparently utilizing these frameworks and enabling RapidMiner processes to parallelize their computations within a distributed environment.

The systems applicability and practicability is demonstrated on two Machine Learning techniques arising from Concept Detection in Videos with the Bag-of-Visual-Words approach: Interest Point Extraction in video frames and k-Means clustering. Evaluations show that the system is able to accelerate these processes by utilizing multiple cores and machines. Furthermore, using GridGain and Coherence as distributed framework within the system can lead to nearly linear speedup with the number of machines.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Today, Pattern Recognition and Machine Learning techniques are found in many applications like face and object recognition in videos and images, speech recognition as given in mobile phones, or optical character recognition (OCR) for the automatic digitization of scanned documents. Because of their great potentials and wide usage in many areas, Pattern Recognition and Machine Learning are also subject of large interest in academics and research. Nonetheless, applying theses techniques usually both requires professional expertise in the field of Machine Learning, as well as software engineering skills in order to integrate into real world applications. Furthermore, the development and adaptation of Pattern Recognition and Machine Learning systems most often differ to those of traditional software systems, since the methods used in such systems are data-driven, i.e. their performance strongly depends on the problems and domains they are applied to.

In this context, the project *PaREn* [7], which is initialized by the *German Research Center for Artificial Intelligence* (DFKI) and funded by the *Federal Ministry of Education and Research*, aims to find ways to support and automatize development, evaluation and application of Pattern Recognition systems. Furthermore, carrying out these processes should be feasible with respect to execution time of the computations involved. However, especially in Pattern Recognition and Machine Learning settings this goal proves challenging, since one often faces very large and heterogenous data sets which have to be processed, and the techniques and algorithms used often require a vast amount of computation time. This leads to situations in which one machine is no longer sufficient and scaling to multiple machines becomes necessary.

One solution to this problem is distributed computing. Nowadays, there exist complex systems and software tools which make performant and reliable computation and distribution of data possible on multiple machines, even scaling to big data centers. Furthermore, programming models like MapReduce [11] - a divide & conquer approach applied to distributed computing - aim to foster utilization and integration of distributed systems into real world applications. Nonetheless, distributed computing remains challenging in many aspects and is far from being easy to handle, even for experts. This particularly holds true when applying it to complex tasks as arising from Pattern Recognition and Machine Learning setups.

One tool box, which aims to provide state of the art and easy to use Machine Learning and Data Mining components, is *RapidMiner* [37]. This open source software is an integral part of PaREn, as it provides intuitive and easy to use interfaces for handling heterogenous data sets and for constructing Pattern Recognition and

Machine Learning systems. However, RapidMiner lacks support for distributed computing capabilities. In opposite to this, there exist a variety of distributed computing software like *Hadoop* [3], *GridGain* [20] and *Oracle Coherence* [34], which already have been successfully used for enabling applications to scale in distributed environments and to increase their performance. An ideal situation would be to have the well-suited Machine Learning interfaces of RapidMiner combined with the benefits of using distributed computing software to scale on multiple machines. However, integrating these tools is not trivial and usually requires detailed knowlegde about the frameworks and a lot of experience with distributed systems in general.

In summary, it can be said that bringing together the techniques of Pattern Recognition and Machine Learning with the capabilities of distributed computing in a performant and comprehensible manner is a difficult task, but becomes mandatory, especially when considering the constantly growing amounts of information in todays world of internet and large scale applications. Projects like PaREn, as well as existing Pattern Recognition and Machine Learning applications, can benefit from these approaches by making their underlying processes scalable and more performant.

1.2 Thesis Objectives

The main subject of this thesis is the exploration of opportunities for applying distributed computing to Pattern Recognition and Machine Learning techniques. It is examined in which way such a combination can lead to significant speed up of these techniques. In this context, a special focus is put on the popular MapReduce paradigm as a possible programming model for distributed computing. It shall be demonstrated how it can be applied to Pattern Recognition and Machine Learning algorithms, and which problems arise in this context.

The thesis takes place in the context of PaREn, which uses RapidMiner as basis tool. Thus, the main goal is to elaborate ways for integrating distributed computing software into RapidMiner. As a result of this, a system shall be developed which seemlessly can be embedded into RapidMiner and by this enables different components of RapidMiner to make use of multicore capabilities and multiple machines, making them more performant and scalable. An emphasis of the development lies on the utilization of MapReduce as a programming model for the system. The system shall be easy to use, both for Machine Learning developers, as well as for RapidMiner users.

In order to prove the applicability of the system, different use cases shall be built up on it. A prominent use case in Pattern Recognition and Machine Learning, as well as in the context of PaREn, is Concept Detection in Videos. One important technique in this field is the Bag-of-Visual-Words approach, which includes two methods which shall be considered for application to the developed system: Interest Point Extraction from video frames and the k-Means clustering algorithm, which is responsible for codebook generation.

1.3 Thesis Outline

In chapter 2, mandatory and useful background topics and issues of this thesis are explained. This includes an introduction into Pattern Recognition and Machine Learning in general, as well as an overview of the project PaREn. Furthermore, principles of distributed computing and distributed systems are shown, including a classification of two types of distributed systems, and giving an introduction into the MapReduce paradigm. In the third part, a presentation and an overview of the used software libraries is given, on the one hand RapidMiner as the Pattern Recognition and Machine Learning tool, on the other hand Hadoop, GridGain and Oracle Coherence as distributed computing software. At last, the chapter includes an introduction into the use case of Concept detection, including a small overview of the techniques of Interest Point Extraction and k-Means clustering.

Chapter 3 deals with the development of the distributed system and its integration with RapidMiner. First, the requirements for the systems are identified and explained. In the second part, there is a discussion about the distributed computing libraries, which is done with respect to the requirements. After this, the developed system is presented. The components of the system, its functionality, and its integration within RapidMiner are demonstrated. It is also shown how the system is realized by using the different distributed computing libraries. After that a demonstration is given on how the system can be applied on the methods Interest Point Extraction and k-Means clustering.

In chapter 4, the system is evaluated with respect to computation performance. Different experiments are conducted, which prove the practicability of the developed system by reference to the use cases of Interest Point Extraction and k-Means clustering. It is shown that the developed system leads to performance gains when applied to these methods. The results of the thesis are concluded and summarized in the last chapter, in which also several ideas for future work are proposed.

Chapter 2

Background

This chapter covers the background topics and issues which are mandatory and useful for a profound understanding of the contents of this thesis. First, a definition is given about Pattern Recognition and Machine Learning in general, followed by an introduction into the project PaREn, which is concerned with the issues of Pattern Recognition and Machine Learning. Second, an introduction and overview in distributed computing is given. This includes a comparison of Computational Grids and Data Grids, as well as an introduction into the MapReduce programming paradigm. The third part presents the software libraries used in this thesis, i.e. RapidMiner, Hadoop, GridGain and Oracle Coherence. At last, there will be an introduction into Concept Detection in Videos using the Bag-of-Visual-Words approach. Bag-of-Visual-Words will serve as a use case for PaREn and especially for this thesis.

2.1 Pattern Recognition and Machine Learning

2.1.1 Definition

The idea of *Pattern Recognition* especially in computer science is concerned with the automatic discovery of regularities in data through the use of computer algorithms and with the use of these regularities to take actions on it [6]. Usually the discovered regularities - also called patterns - are used to construct models in order to represent real world circumstances. Based on these models, which can be seen as approximations of the real world, possible actions are the classification of data into different categories or making predictions based on observed data [2].

The models themselves are normally described with the aid of parameters and can be adapted by changing these parameters. The experience of the last decades has been that the most effective methods for constructing models and developing classifiers involve learning from example patterns [13]. The corresponding techniques, which are referred to as *Machine Learning*, usually include large amounts of training data or past experience in order to perform their task: the optimization of model parameters with respect to specific classifying problems.

The methods used for Pattern Recognition and Machine Learning often tend to be very complex, both in terms of comprehension and computation time. The project PaREn, which is described in the next section, is concerned with the issues of developing Pattern Recognition systems. Different aspects of these development processes are described, especially the reason is pointed out why these processes demand a lot of computation time.

2.1.2 PaREn

Pattern Recognition and Machine Learning are already used in many specialty applications such as spam classification, OCR (Optical Character Recognition) and ad placement [7]. Many functions and behavior of these applications are manually constructed rules that require programming skill, software engineering and extensive testing in order to create and make reliable. The respective code is often developed on a case-by-case basis by specialists. Furthermore, the methods for developing, training and testing Pattern Recognition and Machine Learning modules differ greatly from those of other software systems since the underlying methods are data-driven methods and often change behavior significantly for many input patterns in response to new training data, both properties not shared by "traditional" software systems.

The project PaREn (Pattern Recognition Engineering), which is initialized by the *German Research Center for Artificial Intelligence* (DFKI) and funded by the Federal Ministry of Education and Research, tries to fill this gap between Pattern Recognition, Machine Learning and software engineering. Its goal is to create the methods and tools necessary allowing non-experts to use, train, test, and deploy pattern recognition and Machine Learning modules in real-world software systems.

A big obstacle of the adoption and integration of Pattern Recognition and Machine Learning methods into real-world software systems is the mathematical complexity and sophistication required for adapting them to particular problems. Those methods usually have many parameters representing concepts which are often not very meaningful to developers, and their behavior is highly sensitive to how the underlying Pattern Recognition modules are interconnected. In this context, PaREn set itself to support the development and adaptation of Pattern Recognition systems in real-world environments by pursuing three different goals:

- develop better theories of robustness, self-adaptation, and self-supervision of Pattern Recognition methods.
- develop technologies for automating and supporting model construction and selection in real-world settings, especially as part of whole Pattern Recognition system pipelines.
- develop easy-to-use tools for supporting configuration management, testing, and integrating.

Especially the second goal is central to this thesis. Therefore the following paragraphs survey the different steps involved when designing a Pattern Recognition system. This also includes construction and selection of models. Figure 2.1 further illustrates the whole design process[1].

1. Preprocessing The foundation of a Pattern Recognition system is some given raw data which often has to be prepared first by applying application specific preprocessing algorithms. For example, document images usually have to be binarized before performing OCR or layout analysis. In other scenarios input data may be incomplete due to missing values, but the applied Machine Learning techniques do not accept incomplete data sets. A preprocessing step would fill those gaps e.g. by averaging or using other appropriate statistical methods. Preprocessing steps may also clean up data sets from outliers or noise. Finding and applying the right methods for preprocessing has effect on further steps of a Pattern Recognition system.

[1]Duda et al. interpret design cycles for Pattern Recognition systems in a similar way [13].

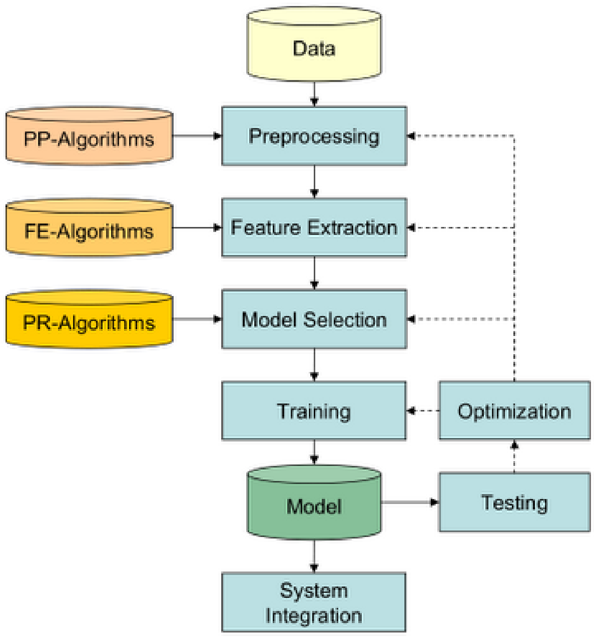

Figure 2.1: Design cycle of a Pattern Recognition system. [7]

2. Feature Selection In general, the patterns to be recognized and classified are represented by measurements referred to as features. The compositions of different kinds of features, also called feature vectors, define points in a multidimensional feature space. In order to classify objects, an appropriate set of features has to be selected. These features are expected to satisfy certain aspects: they should be distinguishing enough for the objects in the domain, invariant to irrelevant transformations of the input data, and compact in dimensionality in order to reduce memory consumption and computation time. Furthermore they should be easy to extract and insensitive to noise. The choice may involve prior knowledge of the problem domain, but finding appropriate features is often not straight forward and often leads to lengthy evaluations.

3. Model Selection The performance of a classifier also depends on the model which is used to approximate the real-world conditions. The better the approximation, the better the classification rate or prediction is. But for different problem domains, some class of models may approximate the real-world better than others. Also the amount and quality of available example data are crucial, since some classes of models are more robust against noisy data than others. Furthermore performance requirements and explanation-awareness may play a role when selecting a model.

4. Training, Testing and Optimization After having selected a classification algorithm it will be evaluated on example data. Therefore it has to be applied on a subset, also called training data. The result of this training step is a model which has to be evaluated in a subsequent testing step by using a subset referred to as test data.

The classification algorithm or model may be adjustable by different parameters. One goal is to select optimal parameter values with respect to classification rate. Existing techniques for performing this parameter optimization for example include grid search or evolutionary computation. According to those techniques, parameter optimization often requires to repeat training and testing many times to evaluate different parameter values.

Even with appropriate parameters the evaluation results may not be satisfying, i.e. classification performance may not be sufficient. This may indicate an inappropriate design of other parts of the pipeline. The model generation procedure may then be reconsidered and one or many of the former steps of preprocessing, feature extraction and model selection may be altered. Training, Testing and Optimization will then be repeated for the modified pipeline. Altogether, the design of a Pattern Recognition system naturally follows a cyclical approach.

As one can imagine, the computation time for each step may become very long, depending on the used techniques. Especially the large number of repititions of training and testing during parameter optimization results in huge computational efforts. Since one of PaREn‘s goals is the automation of design cycles and parameter optimization, it is worth to think about options to accelerate the execution of the individual steps. In some cases, this might even be necessary to make a whole automatized design process feasible in practice.

One way to make the used Machine Learning techniques more performant is introducing parallelization into them. By this they can be enabled to make use of many processors, and even many processing machines at the same time. This is where the idea of distributed computing applies.

This thesis, as a part of PaREn, deals with the issues of accelerating Pattern Recognition and Machine Learning with the aid of distributed computing. Its goal is to develop a system which enables such techniques to utilize distributed computing techniques. More precisely, this thesis aims to integrate the Machine Learning library RapidMiner, which is the main tool for PaREn, with distributed computing software. The next section covers the relevant topics of distributed computing, whereas RapidMiner and different distributed computing frameworks are presented in section 2.3.

2.2 Distributed Computing

Nowadays, not only server machines, but also desktop computers are standardly equipped with multicore processors. However, algorithms and applications do often not make use of the potential speedup which can be achieved by parallelization, and developing software which efficiently utilizes multicore capabilities proves challenging [35]. The same holds true when thinking of multiple machines connected over some local network or the internet. Aspects like communication and synchronization, consistency and replication, fault-tolerance and security [43] are even more difficult to handle within a distributed and heterogenous environment.

Nonetheless, many distributed computing software and middleware exist which provide efficient means to build distributed applications and enable them to scale on multiple machines. In many cases, these distributed systems make the execution of intensive computations and the processing of large amounts of data feasible in the first place.

In the context of this thesis, it is helpful to distinguish two types of such distributed systems: *Computational Grids* and *Data Grids* [38]. The term *grid* is referring to the practice of Grid Computing [5], which has become a popular paradigm in the field of distributed computing during the last two decades. By a definition of

Ian Foster and Carl Kesselman, the Grid is a hardware and software infrastructure which allows reliable, consistent, easy-to-reach and cheap access to the capacities of high performance computers [16]. In this thesis, different distributed computing frameworks are considered, which in many ways fit into this abstract concept of grids. These frameworks, namely Hadoop, GridGain and Oracle Coherence, are presented in 2.3. All of them aim to provide facilities to build up infrastructures which span multiple machines and allow the reliable and easy access to the capacities of these machines. In this sense these frameworks can be possible basis technologies for doing Grid Computing. The further distinction between Computational and Data Grids is made to emphasize the difference between distributed systems which in first place deal with the management of computation processes in a distributed environment, and others which concern themselves with the appropriate management and provision of data. Both Computational Grids and Data Grids include several aspects which are relevant for the discussions in the following chapters. These aspects are introduced and explained in the following subsections. The distinction between both is taken up in this thesis to classify the used distributed systems and components.

Another important subject in this thesis is the MapReduce paradigm. Having a reliable distributed software infrastructure can be the prerequisite to scale applications on multiple machines. However, finding ways to parallelize application logic and mapping it to the functionality of a distributed computing framework usually is a difficult task. MapReduce is a programming model which aligns to the principles of divide & conquer approaches and by this provides an intuitive interface to map applications onto it and enable them to utilize distributed computing environments. The characteristics of MapReduce and its applicability to Pattern Recognition and Machine Learning techniques are examined in this thesis. Therefore, this section includes a general introduction into this topic.

2.2.1 Computational Grids and Data Grids

In general one can distinguish two different categories of grids: Computational Grids and Data Grids [38]. A **Computational Grid** allows to take computations, optionally split them into multiple parts, and execute them on different processing nodes in the grid in parallel. A processing node in this sense is a machine or some other kind of processing unit, a grid defines a cluster of connected processing units. The benefit is that the computation may perform faster due to the parallel use of resources from all processing nodes in the grid. Computational Grids improve overall scalability and fault-tolerance of systems by offloading computations onto most available nodes. A common design pattern used in a Computational Grid is MapReduce, which is described in the next subsection. Some of the requirements a Computational Grid should meet are explained in the following.

- **Load Balancing.** Proper load balancing is crucial for system performance. There are many ways to perform load balancing, for example by picking nodes randomly for new jobs, by following a Round Robin algorithm, or by adapting load balance depending on the performance of individual nodes. The major goal is to distribute the workload in a way that jobs optimally utilize the existing processing nodes, not leaving any of them idle for a longer time.

- **Fault-Tolerance.** The reliability of a Computational Grid strongly depends on how failures are handled within the grid. If a processing node crashes or the job causes some failure, the execution should be rescheduled to another node and re-executed automatically. This failover mechanism should work transparently to the user.

- **Automatic Deployment.** Newly developed application software should be automatically deployed on all nodes of a grid without any extra steps from the developer. Automatic deployment not only boosts productivity in development processes by avoiding time-consuming installation and configuration steps on all processing nodes, but it also increases reliabilty, since manual deployment rather tends to induce faults into a distributed application.
- **Data Grid Integration.** Since most applications process data and therefore depend on efficient access to it, compute grids should integrate seamlessly with adequate data grids and utilizing their features within their control mechanisms.

A **Data Grid** allows to distribute data across the cluster of machines. A main goal of a Data Grid is to provide fast access to the data, which could for example be achieved by providing as much data as possible from main memory on each processing node. Furthermore it should ensure coherency of data accross the grid. Further requirements a Data Grid should meet are explained in the following.

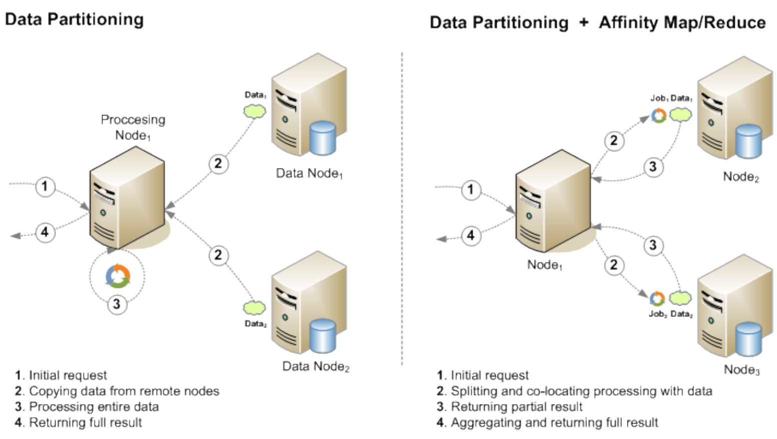
Data Partitioning
Proccesing Node$_1$
Data$_1$
Data Node$_1$
Data$_2$
Data Node$_2$
1
2
3
4
1. Initial request
2. Copying data from remote nodes
3. Processing entire data
4. Returning full result
Data Partitioning + Affinity Map/Reduce
Job$_1$ Data$_1$
Node$_2$
Node$_1$
Job$_2$ Data$_2$
Node$_3$
1. Initial request
2. Splitting and co-locating processing with data
3. Returning partial result
4. Aggregating and returning full result

Figure 2.2: Data affinity of jobs in a distributed computing environment [42].

- **Data Affinity.** An important aspect of distributed data processing is the locality of computations and corresponding data. If nodes often have to fetch data from other nodes in order to run the jobs which have been assigned to them, network bandwidth becomes a bottleneck for overall system performance. Therefore, Data Grids should offer functionality for locating computation to corresponding data. Figure 2.2 demonstrates how jobs are aligned to corresponding data.
- **Data Replication.** The performance of data access strongly depends on the selected replication strategy. Data may be fully replicated to all nodes, which fosters access time, but consumes most resources. Another strategy assumes a fixed number of replica per data unit. More sophisticated strategies may adapt the replication factor depending on access rate of individual data. Others may involve network topology to ensure efficient balancing of data in order to reduce network traffic.
- **Data Backups.** Replication of data is not only crucial to allow quick access, but also to avoid data loss due to node failures. If a node crashes for some reason, the data should be backed up and immediately be accessable on another node.

In this thesis, three different distributed computing frameworks are presented, which mostly satisfy the conditions mentioned above and therefore can be seen as Grid Computing frameworks. The Hadoop framework both implements Computational and Data Grid functionalities. The other frameworks are GridGain and Coherence. The former one can be seen as Computational Grid middleware, whereas the latter can be classified as a Data Grid solution. The frameworks themselves are described in section 2.3.

2.2.2 MapReduce

MapReduce is a programming model and a software framework which has been introduced in [11] and patented by Google [12]. The main goal is to have a simple programming interface which supports distributed computing on large data sets on clusters of computers. The core idea, which is following a divide & conquer approach, is to have a *map* function, which is applied on parts of the input data, and to have a *reduce* function which then aggregates the results of the map step. The idea is inspired by functional programming, where usually a *map* is used to apply a function on each element of a list, whereby a *reduce* construct aggregates a list to a single value[2]. Even though it is flawed in some places [26], this comparison gives a good impression of how MapReduce works.

Since a *map* is thought to be applied independently on each data element of the input list, many map tasks may be executed at the same time in parallel. This is where the interface provides support for distributed computing. In the proposed programming model also the reduce step is abstracted in a way that there may be many reduce tasks which run in parallel. The programming model is described in more detail in the following.

Programming model

The input to the computation is split before and given as a list of key/value pairs $\mathcal{I} = [(k_{1_1}, v_{1_1}), ..., (k_{1_n}, v_{1_n})]$, the output produced also is a list of key/value pairs. The user of a MapReduce framework defines the two functions *map* and *reduce*. The *map* function takes one key/value pair as input and produces a list of intermediate key/value pairs:

$$map(k_{1_i}, v_{1_i}) \rightarrow [(k_{2_1}, v_{2_1}), ..., (k_{2_m}, v_{2_m})]$$

The intermediate key/value pairs of all map steps are then grouped by their keys. The resulting lists are passed in form of key/value-list pairs to the *reduce* function:

$$reduce(k_{2_j}, [v_{2_j}1, ..., v_{2_j}l]) \rightarrow v_3.$$

For each key, the reduce function aggregates the input list and returns one value as output. Figure 2.3 demonstrates the data flow of a MapReduce computation.

Examples

A typical example to illustrate a MapReduce computation is the word count example. The input data is a large collection of documents, for which the number of occurrences for each word are counted. In this function the *map* emits each word of a document with an associated count of occurrences. The *reduce* step sums together those counts for each word to obtain the total count for a particular word. A Java implementation for the map and reduce steps would be similar to this:

[2]One example are the built-in functions map and reduce in the python programming language

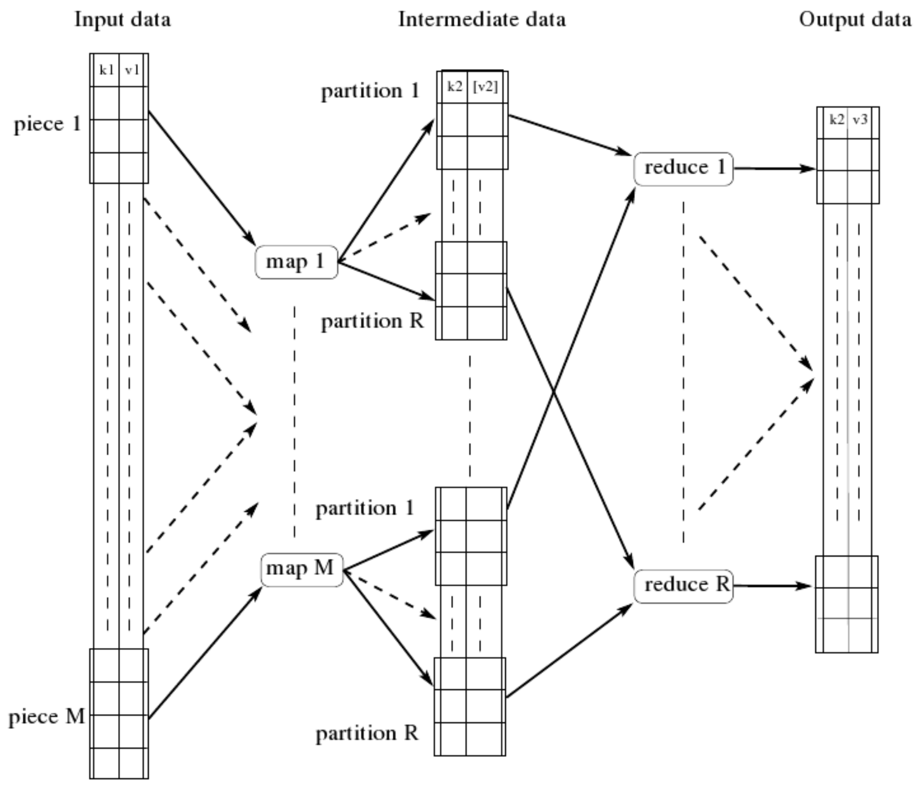

Figure 2.3: MapReduce: First the input data is split and each part is processed independently by a *map* function. The results of the map steps are aggregated by a *reduce* step [26].

Listing 2.1: Example for a *map* and a *reduce* function. The word occurrences of a collection of documents are counted.

```java
public void map(String key, String value){
  // key: document name
  // value: document content
  for (String word : value.split("␣")){
    emitIntermediate(word, new Integer(1));
  }
}

public void reduce(String key, Iterator values){
  // key: a word
  // values: a list of counts
  Integer sum = 0;
  while (values.hasNext()){
    sum += (Integer) values.next();
  }
  emit(sum);
}
```

Other examples include a distributed large scale grep implementation, the construction of reverse web-link graph or building an inverted index [11]. There are many algorithms and tasks which can be formulated in terms of *maps* and *reduces*. If one can solve a problem by following a divide & conquer approach, then it is often also expressible as a MapReduce computation. Especially in the fields of Pattern Recognition and Machine Learning there are many algorithms and tasks which can be modified to fit into the MapReduce concept [10, 19]. Thus, by adapting those algorithms and tasks one may take benefit of an underlying distributed system and accelerate computations which otherwise would take much longer.

Implementation

The idea of dividing a task into sub-tasks, executing them in parallel and merging the results in the end is by no means invented by Google [25]. Nonetheless, MapReduce has gained huge popularity in industry and science, since Google could demonstrate the ease of using this pattern and that it provides a highly effective means of attaining massive parallelism in large data- centers [11]. To achieve this, Google‘s implementation of the MapReduce framework provides solutions for different important aspects of distributed systems, which are hidden to the user. Any implementation of MapReduce should consider these aspects. In the following, three of them are described, namely Load Balancing, Fault Tolerance and Locality Optimization. These correspond to the same-titled aspects described in the context of Computational and Data Grids in the previous subsection. The following explanation focuses on these issues in the context of MapReduce.

- **Load Balancing.** Before executing a MapReduce computation, the program is copied to the machines of the cluster. Exactly one copy of the program defines the master, which is responsible for organizing the distributed computation. This includes the assignment of *map* and *reduce* tasks to worker nodes. Every time a worker node becomes idle, the master node assigns a new *map* or *reduce* task to it. Thus, a load balancing is achieved, since each machine receives a new task when new capacity becomes available.

- **Fault Tolerance.** In large clusters of hundreds or thousands of computers one has permanently to deal with crashes or machines that are not reachable. To recognize possible failures of individual nodes, the master pings every worker periodically. Workers which do not respond in a certain amount of time are marked as failed. The task will then be rescheduled to another worker. This *failover* mechanism is essential for the reliability of a MapReduce system.

- **Data affinity.** One bottleneck of MapReduce usually is network bandwidth. Therefore one goal of a MapReduce implementation should be to reduce network traffic, not only for messaging, but especially for data transfer. Thus, it is important that the input data already is stored locally on the machines where the computation takes place, or in other words, to bring the computation to the data, not vice versa. This *data affinity* plays a key role for performance in a MapReduce framework. Google achieves this by using its Google File System (GFS) [18]. This distributed file system divides each file into 64 MB blocks and stores several copies of each block on different machines. Due to this replication mechanism, the master node can take the locality of input blocks into account when assigning map tasks to a worker. If one worker node fails, the task will be rescheduled to a machine which owns a replica of the input data block. Totally viewed, most of the time the input data is read locally and therefore does not consume network bandwidth.

2.3 Software

In this section, the different software libraries used in this thesis are presented. This includes an introduction into the Pattern Recognition and Machine Learning library RapidMiner, as well as into the distributed computing frameworks Hadoop, GridGain and Oracle Coherence.

2.3.1 RapidMiner

The PaREn project uses the data mining tool *RapidMiner 5.0* [37] (formerly *YALE*) as basis for building Pattern Recognition systems. RapidMiner focuses on rapid prototyping for knowledge discovery, Data Mining and Machine Learning systems. Its goal is to support maximal re-use and innovative combination of existing methods [31]. To achieve this, RapidMiner provides a variety of existing methods, including different input and output mechanisms, state of the art Machine Learning methods as well as facilities for data processing and feature space transformation. Especially the offered evaluation and meta optimization methods are important for PaREn, since they support the selection of suitable components for a Pattern Recognition system. Additionally RapidMiner provides a range of visualization tools like plots of data or experiment results.

Since RapidMiner follows the paradigm of visual programming and provides an XML interface for storing processes, it allows easy construction and automated execution of Pattern Recognition processes. Processes in RapidMiner are expressed as a combination of "operators", which are connected as a directed graph. In RapidMiner, which is entirely implemented in Java, each operator extends the class *Operator*, which essentially represents a method as mentioned above. An operator usually expects some input data on which it performs a defined action, and delivers some output which then again may serve as input for other operators. The graph structure defines the data flow within a process. Figure 2.4 demonstrates a typical process graph as presented by the RapidMiner GUI.

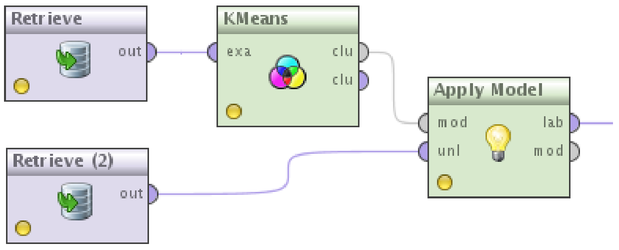

Figure 2.4: A typical Rapidminer process: k-Means clustering on a data set. The resulting cluster model is used for assigning samples of another set to their nearest clusters.

Input and output data passed between operators implement the interface *IOObject*, which especially ensures them to be serializable. All data objects in RapidMiner can therefore be persisted and transfered.

Machine Learning and Data Mining methods usually use data sets consisting of sample vectors, which may have nominal or numerical component values. RapidMiner includes the interface *ExampleSet*. Implementing classes represent a data set, which is essentially a table of multi-dimensional sample vectors. Since ExampleSet also extends the IOObject interface, it can be used as input and output data for Machine Learning and Data Mining operators.

As outlined before, the main goal is to introduce parallelization and distributed computing into PaREn processes. Since RapidMiner is the basis tool for PaREn, this thesis also builds up on RapidMiner. The main focus will be on the integration of RapidMiner and different frameworks for distributed computing. This especially involves the extension, implementation and usage of RapidMiner classes and interfaces like *Operator*, *IOObject* and *ExampleSet*.

2.3.2 Hadoop

The Apache project *Hadoop* [3] develops open-source software for reliable, scalable, distributed computing. The Hadoop framework, which is written in Java, enables applications to work with thousands of processing nodes and petabytes of data. It includes a MapReduce implementation and a distributed file system (HDFS) which are both inspired by Google's MapReduce and GFS publications. According to the categories of grid systems presented in 2.2.1, the HDFS can be classified as Data Grid, whereas the MapReduce part manages the computational aspects.

Hadoop is being used and built by established contributers [21], including AOL, Amazon.com, Facebook, IBM and many others. Especially Yahoo! caused sensation by building web search and advertising applications upon Hadoop, running it on a 10.000 core Linux cluster, using over 5 Petabytes disk space [33].

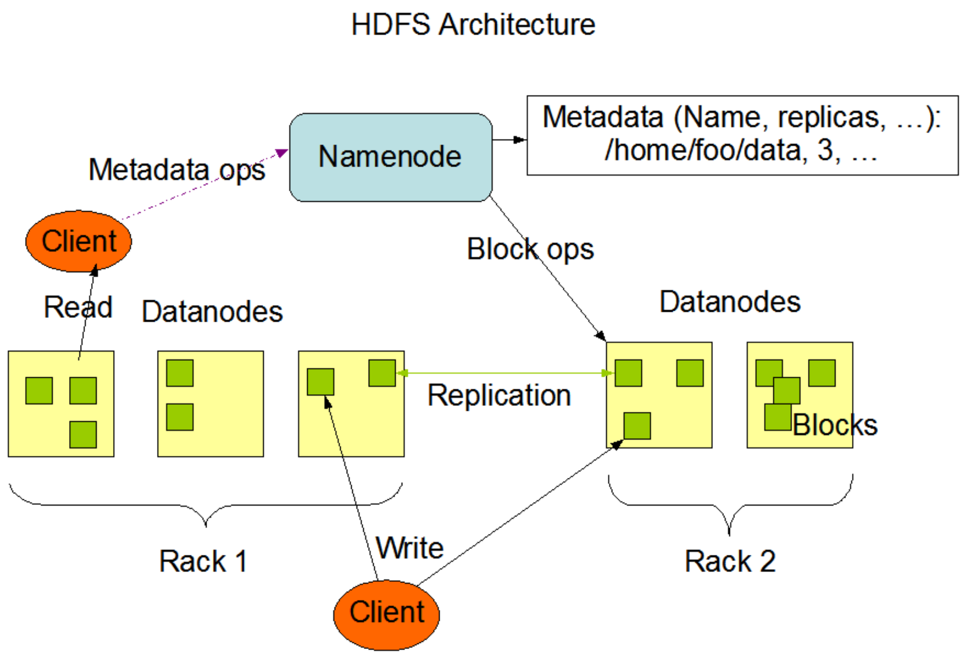

Figure 2.5: HDFS: The name node manages the file system. The data nodes ensure replication and local access on data [3].

HDFS The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. The goals of HDFS are to be highly fault tolerant and to ensure high throughput of large scale data sets. The former one is achieved by replication of data on several storage nodes, similar to the GFS. Each file in the file system is stored by deviding it into blocks (usually 64MB in size), which are replicated and distributed between so called *data nodes*. The data nodes are able to communicate with each other to copy and rebalance the data across the cluster. Since hardware failure is a norm in big clusters of hundreds or thousands of machines, each block redundantly exists three times in the cluster by default[3]. Thus, failure of a data node would normally not result in loosing data. The file system is managed by exactly one *name node*. It stores meta data about the number and position of blocks and to which file they belong. A client who wants

[3]Typically two blocks in the same rack, one block in another one.

to read data only receives the position of a block and therefore does not have to transfer the data over the name node, but receives it directly from the corresponding data node. Figure 2.5 illustrates the collaboration of name nodes and data nodes within a HDFS cluster. HDFS also provides interfaces for applications to make use of data affinity. This minimizes network congestion and further increases the overall throughput of the system. In general, high throughput is crucial to HDFS since it focuses on the support of batch processing on large data sets rather than interactive file access. Therefore, HDFS should not be seen as a usual file system but rather as data set system [24]. According to that view, files usually are very large in size, typically giga- or terabytes.

MapReduce The MapReduce component in Hadoop is very close to the programming model proposed by Google. It is built upon HDFS, so the input data is usually directly read from the distributed file system. The implementation uses the data affinity features of HDFS to ensure having maps applied locally on the input and therefore to gain throughput.

In Hadoop, a MapReduce computation is called a job, whereas computing a simple map or reduce is called a task. Hadoop‘s MapReduce is designed as a master/slave architecture. Similar to the master node in Google‘s proposal, Hadoop runs a *Job Tracker* on one node of the cluster, which manages the execution of a whole MapReduce job. The Job Tracker sends map and reduce tasks to so called *Task Trackers* in order to be done, striving to perform the work on those nodes on which the data resides or at least as close as possible to them.

2.3.3 GridGain

GridGain [20] is an open source Grid Computing framework, which focuses on easy development and deployment of Grid Computing and Cloud application software. It is pureley written in Java, and is a software middleware that allows to develop complex grid applications on the cloud infrastructure [14]. In the context of Grid Computing, it can be seen as Computational Grid, since its main focus is on managing computational tasks within a distributed environment, and explicitely leaves data management to underlying Data Grids.

GridGain has support for MapReduce like computations because it includes a MapReduce implementation which is similar to the one proposed by Google, but less complex. It reduces itself to split a task in several sub-tasks, executes them on different grid nodes, and finally aggregates the results within a single step. Google‘s MapReduce is more abstract, since it allows several reduce steps because of its intermediate key/value-pair mechanism. Furthermore, Google‘s MapReduce is completely data driven, whereas GridGain as a Computational Grid focuses on the computational aspects of task splitting in the first place. In reverse to Hadoop, the overall MapReduce computation is called a Task, whereas a single map is done by a Job.

GridGain is designed as a Computational Grid and therefore is optimized with respect to crucial aspects of these kind of grids (see also 2.2.1). It follows a higly modular design and therefore is very flexible. It is made up of several services implementing so called service provider interfaces (SPI). For example, grid nodes find each other on the network by using a discovery service, which implements the *GridDiscoverySpi* interface. The default implementation uses IP-multicast for discovering other nodes. But there are other implementations to make use of or to integrate with frameworks like Coherence (see 2.3.4), JBoss[4], JGroups[5] and JMS[6],

[4]http://www.jboss.org
[5]http://www.jgroups.org
[6]http://java.sun.com/products/jms/

but also e-mail based discovery is possible. Similar to GridDiscoverySpi there are other service provider interfaces covering issues like communication, deployment, load balancing, fail-over, collision, checkpoint and topology resolution. For all of these GridGain provides several implementations.

In this way GridGain also integrates with several established middleware frameworks out of the box. Communication in GridGain for example is realized with TCP/IP connections by default. But as in the discovery-example, GridGain also includes implementations for Coherence, JGroups, JMS and others, in order to reuse their communication protocols and to run on top of existing grid environments.

As stated in section 2.2.1, a Computational Grid should allow automatic deployment of new software. GridGain achieves this by its P2P-class-loading mechanism. A grid developer does not have to copy newly written Java classes or archives to every node by hand or even has to restart the nodes. New classes are P2P-loaded at deployment time from the remote node which initialized the task.

GridGain not only supports distribution in terms of MapReduce, but offers general Grid Computing facilities. This includes the execution of single task within the whole grid, which are not parallelized in a MapReduce manner. The location of execution thereby is chosen transparently to the user, but with respect to the work load found on individual nodes.

Since GridGain is a Computational Grid, it does not provide data distribution in terms of a Data Grid. But its designed to easily integrate with different Data Grids. Escpecially for MapReduce it is important that computation is located near to data (compare 2.2.2). Therefore GridGain provides the *GridAffinityLoadBalancingSpi*. Implementations for this interface support data affinity for MapReduce jobs. Depending on the kind of underlying data distribution or the used Data Grid, different GridAffinityLoadBalancingSpi implementations can be chosen to ensure an efficient distribution of sub-tasks with respect to the location of data.

In this thesis, GridGain is always considered in conjunction with Oracle Coherence as underlying Data Grid. GridGain provides integration with Coherence out of the box. As Coherence is an in-memory distributed cache, the data can be accessed very fast. Coherence is described in the following subsection.

2.3.4 Coherence

Coherence is a commercial distributed memory data management solution offered by *Oracle* [34]. It provides a reliable distributed data tier with a single, consistent view of the data. This data tier, also called *Distributed Cache*, represents itself as a key/value-store, which is partitioned and/or fully replicated across several processing nodes.

Caching is a concept especially known from the area of hardware in form of memory caches. A cache transparently arranges frequently used data to be fastly accessible for a processing component. A Distributed Cache is a form of caching which allows the cache to span multiple servers so that it can grow in size and transactional capacity [23]. Data can be stored in the Distributed Cache on one node, and transparently received from the Distributed Cache on another node. Usually data in a distributed cache is kept in main memory to allow fast access. In practice a main memory cache is layered above some background persistent storage like a database or a file system. It therefore reduces traffic between application and storage, which arises from communication and serialization issues. Operations on cached data will be done with respect to the underlying background storage. When reading data which has not been cached before, it will be fetched from storage. Writing data can be handled synchronously or asynchronously by using write-through or write-behind mechanisms. A further advantage of a cache is transparent eviction of data. If there are too many objects in the cache and therefore the cache is at risk of overflow, it automatically evicts selected objects and writes them back to the background storage. Usually a cache supports different eviction strategies like Least-Frequently-Used (LFU), Least-Recently-Used (LRU) or First-In-First-Out (FIFO). Which one to select depends on the application, since each strategy has advantages and disadvantages regarding access time of objects. Because of eviction a cache is not only useful to provide fast access to data, but also to support processing on large data sets which do not totally fit in main memory.

As said before, a distributed cache provides a single, consistent view of the data. This means that any updates of data on one cache node will be propagated to the other nodes. This may be done by replicating updated data or by marking it as invalid on the other nodes. A read operation on another node will then cause a refetch of the data.

The different structures and the underlying mechanisms a cache can take up are known as cache topologies. Coherence provides several cache toplogies, including replicated cache, partitioned cache, local cache and near cache. All of these can also be used in conjunction and/or together with some background storages. In the following there is a more detailed description about replicated and partitioned caches, since these are used within the developed system in chapter 3.

Replicated Cache A replicated cache holds a copy of each data unit on each cache cluster node. This means that a replicated cache provides high availability of data and reliability of the cache. If any node of the cache cluster goes down, there will be no loss of data since it is available at any other node. This topology is very efficient and scalable if an application needs to do a lot of read-intensive operations. The more cache servers are added to the cluster the more read-transaction capacity is available. But on the other hand a replicated cache is not the ideal topology for write-intensive operations. Each write will cause an update on all other nodes and therefore can result in large network traffic.

Partitioned Cache In contrast to replicated caches, a partitioned cache does not copy every data unit to each node. A partitioned cache breaks up the whole data into partitions and then stores distributes them across the cluster nodes. This topology facilitates write operations, since they do not have to be propagated within the whole cluster. The partitioned cache therefore scales very well for write- and read-intensive applications. In practice, a partitioned cache is partially organized like a replicated cache, as it replicates every partition as backup to only few nodes in the cluster, but not all. This backup mechanism not only ensures reliability of the cache, but also increases availability.

2.4 Use Case: Concept Detection in Videos

This thesis focuses on the acceleration of Pattern Recognition and Machine Learning methods by using distributed computing techniques. Different issues and problems of introducing distributed computing into such methods are discussed and a system is demonstrated, which integrates distributed computing software into the Machine Learning library RapidMiner. In order to prove the systems practicability, it is applied to an important use case of Pattern Recognition and Machine Learning and especially for PaREn: Concept Detection in Videos using the Bag-of-Visual-Words approach. This use case is explained in the following, including two techniques which are examined in more detail in this thesis: Interest Point Extraction and k-Means clustering.

In multimedia retrieval, one sub-area of research is Concept Detection. The goal of Concept Detection in Videos is to infer high-level semantic concepts from video contents. While humans are naturally able to understand and interpret video content, the same thing is a non-trivial task for a machine, since videos are initially given within a low level representation[7], i.e. as a collection of frames, each consisting of simple pixel values. The mapping of these low-level multimedia features to high-level semantic concepts is usually achieved by using Pattern Recognition and Machine Learning techniques. Based on different low-level features such as colors, textures, shapes, interest points or temporal features, models are constructed and optimized to allow an automatic detection of the high-level concepts. A comprehensive overview of the field of Concept Detection can be found in [41].

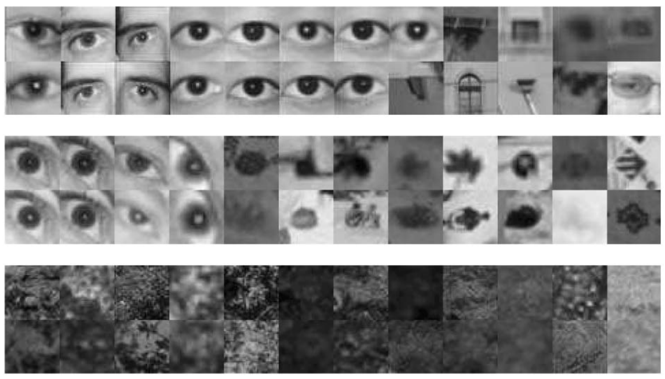

Figure 2.6: Interest points which are located in the same cluster. Each cluster defines a visual word.[36].

A state of the art approach to Concept Detection and Video Retrieval, which is based on local interest points and so called *Bag-of-Visual-Words* features, has been introduced in [39]. In this technique, which will further on be referred to as Bag-of-Visual-Words approach, three major steps are involved, namely interest point detection, interest point description and vector quantization. According to the Pattern Recognition system presented in 2.1.2 all those steps together perform the feature extraction task, which will result in a Bag-of-Visual-Words feature vector for each key frame of a video[8]. In the following, the three steps are explained in more detail.

Interest Point Detection Interest points are used as an efficient means to capture the essence of a scene by detecting the information-rich pixels in an image, such as those representing spots, edges, corners and junctions [41]. Various interest point detectors have been proposed in literature. A detailed comprehension can be found in [44]. Which one to choose or if its reasonable to use a combination of different detectors usually depends on the problem domain. Thus, it is often necessary to perform evaluations of different detectors in different application scenarios, which may result in huge computational demands.

Interest Point Description The second step is to compute a descriptor vector for each interest point. State of the art descriptors are Scale Invariant Feature Transform (SIFT) [30] or Speeded Up Robust Features (SURF) [4] In both descriptor types an interest point is represented as a vector $\mathbf{x} \in \mathbb{R}^n$. By default, SIFT

[7]This problem is also known as *Semantic Gap*[40]

[8]A key frame is a frame of the video which appropriately represents one scene of the video.

is used with $n = 128$, whereas SURF is implemented for example with $n = 64$ or $n = 128$. As the name indicates, the descriptor vectors *describe* an interest point, viz. at best in a way that supports distinctiveness of the described interest points and invariance to transformations like rotating or zooming.

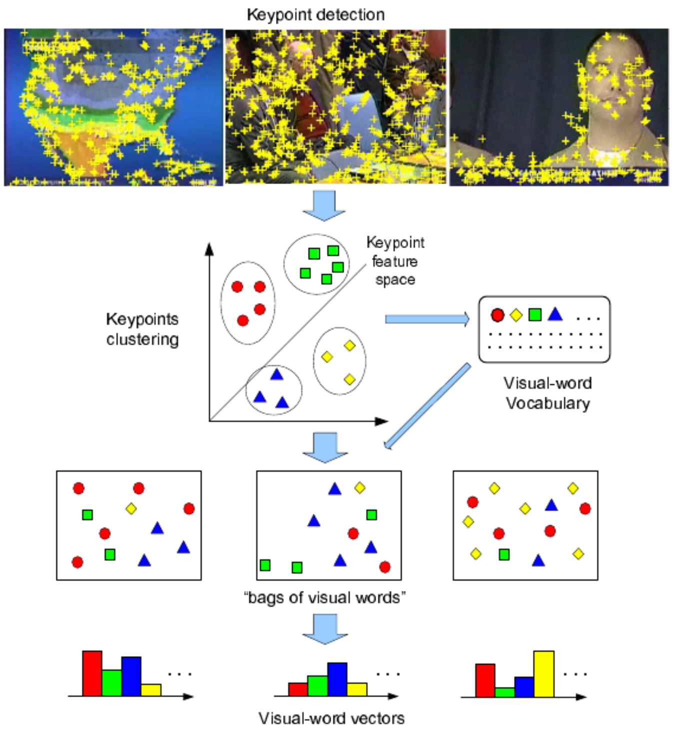

Figure 2.7: Generating visual-word image representations based on vector-quantized interest point descriptors [47].

Vector quantization The Bag-of-Visual-Words idea arose from the field of text retrieval. A text document may be represented by a collection of index terms which do not themselves have internal structure [27]. This model is often called Bag-of-Words. A common approach to get a Bag-of-Words is to count the number of occurrences of all words in the document. The resulting term histogram or term vector defines the Bag-of-Words representation and can be used for document retrieval or classification.

Analogous to the Bag-of-Words document representation, key frames of videos can be represented as a Bag-of-Visual-Words [47]. This representation can be achieved by performing a vector quantization over all interest point descriptors in all key frames. The descriptors therefore are clustered in their feature space into a large number of clusters. This can be done by k-Means or similar methods like k-Medoids or histogram binning. Each cluster can be seen as a 'visual' word that represents a specific local pattern shared by the interest points in that cluster. Thus, the clustering process generates a visual word codebook describing different local patterns in images. Figure 2.6 shows example visual words which can be found in a codebook. The number of clusters determines the size of the codebook, which can vary from hundreds to over tens of thousands. By assigning each descriptor of a frame to its nearest cluster and thereby mapping it to a visual word, the Bag-of-Visual-Words representation can be obtained. According to Bag-of-Words, the visual words for each frame are counted. Figure 2.7 illustrates the vector quantization step.

Depending on application needs, a training set for obtaining a Bag-of-Visual-Words codebook is usually very large. It may consists of thousands, millions, or even billions of video frames. But not only the frames, but also the interest point vectors extracted from those frames are very large in number: one frame usually includes hundreds or thousands of relevant interest points. Furthermore, the size of the codebook influences the classification performance. Finding an appropriate codebook size usually includes evaluations of different sizes.

Because of these issues, following the Bag-of-Visual-Words approach both puts high demands on computational and memory capacity. This thesis takes a look at two methdos of the Bag-of-Visual-Words approach and at the opportunities to accelerate them by using distributed computing techniques. First, this is the detection and description of interest points, which furtheron shall be subsumed by the term Interest Point Extraction. The second method is k-Means clustering. A detailed description of this method can be found in Appendix A. The details of Interest Point Extraction are not crucial for a profound understanding of this thesis. The distributed system developed in the next chapter is applied to these methods. The performance of the system is then evaluated in chapter 4.

Chapter 3

System Development

This chapters deals with development of a distributed system, which integrates distributed computing software into the Machine Learning library RapidMiner. First the requirements for this system are analyzed with respect to technical aspects as well as from a developers and users view. Next, the distributed computing frameworks Hadoop, GridGain and Oracle Coherence are discussed and compared with respect to those requirements. After that, the systems design and its integration into RapidMiner is presented, as well as its realizations with Hadoop and GridGain in conjunction with Coherence. The applicability of the system illustrated by two different use cases: Interest Point Extraction and k-Means clustering.

3.1 Requirements Analysis

The overall goal of this thesis is to find appropriate ways for accelerating Pattern Recognition and Machine Learning techniques by using distributed computing. This especially involves the integration of distributed computing software into Pattern Recognition and Machine Learning applications. A special focus is put on MapReduce as a possible programming model for distribution. The thesis takes place in the context of PaREn, which uses RapidMiner to model its underlying processes and to construct pipelines. Therefore the integration of RapidMiner with several frameworks for distributed computing is emphasized.

Before developing approaches, requirements have to be defined which then are considered during development. In the context of this thesis, the requirements derive from different views, considering different problems and demands. First, the *general requirements* represent the main goals which have to be achieved, abstracting from the special concerns and aspects that arise when looking more deeply into the problem context. More detailed discussions are done within the *technical view*, the *developer view* and the *user view*. The technical view deals with general problems arising from distributed computing and applying the MapReduce paradigm to Machine Learning. The developers view takes a look on the needs of a Machine Learning developer, who wants to apply distributed computing techniques to Machine Learning algorithms within RapidMiner. The last view considers the end user and how he is confronted with the issues of distributed computing while using RapidMiner. The views are discussed in more detail in this section. After discussion of each view there are tables that list the elaborated requirements in a formal way. Functional requirements are numbered as F whereas non-functional requirements are numbered as NF. By this, they can be referenced during the design process.

3.1.1 General requirements

The main goal is to find ways to apply distributed computing to Machine Learning techniques. Since RapidMiner is the considered tool for Machine Learning in the context of this thesis, the goal is to develop an approach or system which introduces distributed computing capabilities into RapidMiner and thereby accelerates execution of different processes provided by RapidMiner. The system should consider the MapReduce paradigm as an approach of distributed programming and should provide a meaningful interface to make efficient use of this approach. The computations shall be accelerated by utilizing many processing nodes, for instance by using multiple cores on one machine as well as spreading the computation over multiple machines.

Nr.	**Requirement**
F0	The proposed system should introduce distributed computing capabilities into RapidMiner.
F1	The MapReduce paradigm should be utilized to provide an efficient means for distributed programming.
F2	The system should make use of multicore capabilities and utilize multiple machines within its distributed environment.

3.1.2 Technical View

It has been shown that it is possible to design many Pattern Recognition and Machine Learning techniques using the MapReduce paradigm [10]. Examples include popular methods like *Naive Bayes*, *PCA*, *Neural Networks*, *SVMs* and *k-Means* clustering (compare 3.4.2). In [10], those have been implemented in MapReduce on Multicore machines, achieving almost linear speedup. There have also been implementations of such methods as MapReduce operations within distributed environments for scaling them on multiple machines and clusters [9, 15, 28, 46]. Nonetheless, there are several problems arising when applying MapReduce on Pattern Recognition and Machine Learning techniques, especially on multiple machines. These problems are often due to limitations of the MapReduce programming model, but also inherent in distributed computing itself. In the following, first a discussion on applying MapReduce to Machine Learning techniques is given. After that the general requirements of distributed computing are outlined.

3.1.2.1 MapReduce and Machine Learning

Machine learning techniques can be classified by their procedural character, i.e. their data processing pattern. In [19], three different classes have been identified: single-pass, iterative and query-based learning techniques. Which class an algorithm belongs to has implications for its adaptability for MapReduce. Some algorithms fit well into the MapReduce paradigm, whereas others entail major problems when trying to adapt. The three classes are outlined in the following.

Within a **single-pass** Machine Learning algorithm, the data is only passed once to extract relevant statistics and information for further learning and usage during inference. One example for this is feature extraction for Naive Bayes Classifiers: estimation of the desired probabilities may be essentially done by summing up occurrences of feature values over the whole data set [10]. This may also involve computation-intensive extraction of features in the first place [19] (which especially would be worth parallelizing). Another example is the extraction of interest points discussed in section 3.4.1. The map tasks are performed per datum or on a subset of the whole data set, extracting local contributions of each datum, which are then combined by the reduce step to compute relevant statistical information - for instance means, variances or histograms - about the data set as a whole. This class of algorithms usually fit very well in MapReduce. Thus, implementations within the programming model are mostly straightforward.

In contrast to this, **iterative** Machine Learning algorithms, which are perhaps the most commonly applied within Machine Learning research, can also be expressed within the programming model of MapReduce by doing multiple operations consecutively. A common characteristic of these methods is that a set of parameters is matched to the data set via iterative improvement. One example for such an iterative algorithm is k-Means (see Appendix A). In each iteration the means are adjusted and therefore improved. Parameter updates decompose into per-datum contributions, i.e. updates depend on the whole data set, which is typical for many algorithms and methods like SVMs or perceptrons. Furthermore, the contribution from each datum depends in a meaningful way on the output of the previous iteration. In k-Means, the means of clusters can only be updated if all data samples have been reassigned to their nearest cluster means, which have been computed in the previous iteration. In consequence, these parameters have to be available to each map task within the distributed environment. Thus, the result of a map task not only depends on the input data, but also on further parameters. This especially becomes a problem if the parameter set for a given Machine Learning instance is very large. One example for this, which can be found in the area of machine translation, is the problem of word alignment in bilingual corpora. In word alignment models, parameters include word-to-word translation probabilities. The parameters therefore can be in the number of millions. Scaling in the number of training example sentences would quickly lead to a point where a simple node cannot handle all the parameter data at once. At least, the communication overhead for broadcasting the whole information would dominate computation time. In the case of word alignment, where a map task rarely needs all of the parameter data, even the MapReduce topology itself would not be the best choice for efficiently solving the problem [46].

Considering iterative algorithms as a common class of Machine Learning algorithms, a MapReduce framework should allow performant access to additional "static" information in form of parameters or configurations in each map task. Furthermore, iterative algorithms often work on the same main input data in each iteration and the computation may only differ on given parameters or configurations. A MapReduce framework therefore should also include fast loading mechanisms for reused data. The best is to keep such input data in memory and locate the same job to the corresponding machine.

The third class of algorithms represents **query-based** learning with distance metrics. These are Machine Learning applications that directly reference the training set during inference, such as the nearest-neighbor classifier. In this setting, a query instance must be compared to each training datum. This can be done by splitting up the data set and perform queries concurrently on the map tasks. Again there is a need to broadcast static information, i.e. the query instances, to all map tasks. However, multiple queries need not to be processed concurrently, but they can be broken up into multiple MapReduce operations. Hence, in contrast to the example of word alignment, static information in query-based algorithms tends to be of managable size. Considering the problems arising from query-based learning, it can be seen as a mixture of single-pass and iterative techniques: A single query passes the dataset only once, but multiple queries may lead to several walks through the same dataset, which is similar to performing iterative computations on the same dataset. Figure 3.1 illustrates the processing patterns of the three classes.

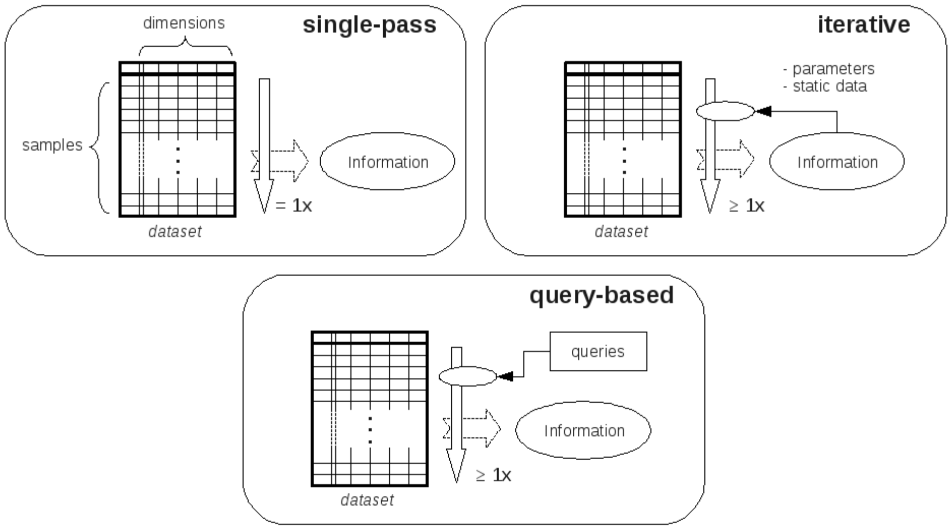

Figure 3.1: Processing patterns of single-pass, iterative and query-based Machine Learning algorithms. Note that iterative and query-based algorithms usually pass the dataset multiple times.

Finally, independent of the class of an algorithm, the input data for a Machine Learning task is sometimes very complex in structure. One sample of a data set may consist of inputs in different formats, perhaps originating from several sources or even external applications. Considering Java as the programming language of RapidMiner, input may arise from different objects, which are logically independent in the first place. But the original MapReduce model expects input to be in form of key/value pairs. An application developer would have to fit the complex input data into this model by bundling the relevant parts of each sample into one complex value object. It is therefore meaningful to provide some mechanisms to easily bundle input data for map and reduce tasks.

The discussion above revealed some basic requirements, which should be met by the distributed system to make a MapReduce application on Machine Learning techniques feasible. These requirements are also subsumed in the following table.

F3	Map Tasks often require access to common static data. The proposed system should provide a performant mechanism to **receive common static data** for map tasks.
F4	Since input data often has a very complex structure, the system should include an easy to use **interface for providing complex input data** to map and reduce tasks.
NF0	In iterative algorithms, the data is often used multiple times. Therefore the system should allow **fast access to reused data** within maps and reduces.

3.1.2.2 Distributed Computing

There are several general issues and requirements arising from the field of distributed computing, which have been partially explained in section 2.2. Computational and Data Grids, as well as realizations of MapReduce should provide appropriate solutions for those.

When looking at **fault-tolerance**, there are generally two types of failures which can occur when executing a Machine Learning task in a distributed environment: First, the node may fail because of external influences which kill the process or make it no longer available. One example for this may be a crashed machine or some network failure. The other type of failures is caused by faults within the Machine Learning task, either because of errors within the code, or maybe because of a wrong usage of the distributed computing framework. The first type of failures should be transparently handled by the system in form of failover mechanisms. The task will then be rescheduled and re-executed on another processing node. This makes less sense when considering the second type of failure. The fault is inherent in the Machine Learning task and will occur again if the task is rescheduled to another node. The developer or user should be informed about the failure to allow him to fix the fault within the code.

Machine learning tasks are rarely given without data to process. Most tasks will work on large data sets, which are costly to transfer and sometimes even too big to fit in memory. Thus, **data affinity** plays an important role when distributing those tasks, especially with MapReduce. In particular, data affinity is useful for iterative algorithms, i.e. when the same data is processed multiple times (see NF0). The proposed system should provide effective means for performing computations with respect to the location of data.

If new tasks are performed within the distributed system, the work load should be balanced appropriately over the nodes. As explained in section 2.2, there do exist several strategies to ensure **load balancing**. In the setting of Machine Learning and MapReduce, it is important to align those strategies to the requirement of data affinity. Load balancing therefore should be done with respect to data locality.

To work on the data in a distributed fashion, it first must be transferred and replicated across the cluster. The distributed system should provide an efficient data grid to do **replication**. There are different means for replicating data, either by copying all data to all nodes or by partitioning the data between the nodes while also maintaining one or more backups of each partition. The system should implement the technique which is most efficient in the case of Machine Learning with MapReduce.

F5	**Failover** mechanisms should be provided in the case of node failure. Task-specific failures should be appropriately reported to the user.
F6	The tasks should be executed with respect to data locality. This regards **load balancing** with respect to **data affinity**.
NF1	**Data replication** should be done efficiently in the context of the system and its goals.

3.1.3 Developer View

The proposed system shall serve as a starting point for developers to introduce parallelization into Pattern Recognition and Machine Learning techniques. Therefore it is important to figure out the needs of developers, especially considering RapidMiner as the main Machine Learning environment.

First, the question arises on how much of the underlying distributed computing technology shall be “visible” to the developer. This refers to the abstraction level of the proposed system. Since it will build up on the MapReduce paradigm, it is crucial how this part of the system will present itself to the developer. MapReduce for itself already abstracts the many issues of distribution and parallelization, but each concrete implementation of MapReduce will provide different interfaces to the paradigm. The proposed system should hide the underlying distribution frameworks as far as possible, while retaining enough flexibility to use them appropriately. By this, the developer would not have to deal with the APIs of external distributed computing libraries and concentrate on the application of MapReduce within RapidMiner. This also reduces the probability of wrong usage and failures.

One important aspect regarding the integration of a distribution framework with RapidMiner is whether it is able to handle data types and classes of RapidMiner. Escpecially serialization and deserialization of objects should be performed in a meaningful way, since this often is a bottleneck in distributed environments. Furthermore, the RapidMiner library must be available on all nodes of the distributed environment. This regards the class loading mechanisms of the distribution framework. The system should allow the use and transfer of RapidMiner data types and classes within the underlying MapReduce environment.

The kind of class loading mechanism not only determines the availability of RapidMiner classes. If a developer creates new functionality in form of new classes, the nodes within the distributed environment need to reload them in order to perform their tasks correctly. Depending on how this reloading mechanism is realized, the developer will spend more or less time for broadcasting the classes to all nodes. This refers to the issue of Automatic Deployment (see 2.2.1). Reducing deployment time is one way to fasten the overall development process, which is especially important considering RapidMiner as a rapid prototyping framework.

NF2	The system should **appropriately abstract** the interfaces of underlying MapReduce frameworks.
NF3	The usage of **RapidMiner and other external libraries** should be transparently possible within the distributed system.
NF4	There should be an easy way for a developer to **deploy** changes during development to all the nodes of the distributed system.

3.1.4 User View

The typical user of the system comes from the Pattern Recognition and Machine Learning community, for instance from the context of PaREn, using RapidMiner as a tool to perform his tasks. As a Machine Learning expert and RapidMiner user in first place, he will mostly not be interested in how the Machine Learning algorithms are implemented in detail, he just wants to use them and be sure they work correctly as expected. This also includes the distribution aspects of the system. When defining a pipeline of Machine Learning tasks by selecting and connecting different components within RapidMiner, it is not essential - from a Machine Learning point of view - that some components may support distributed computing. But a user doubtlessly is interested in accelerating his tasks. In some cases it might be even infeasible to perform the tasks without distributed computing, especially when considering time or memory aspects.

The user should be aware of the possibility of distributing his tasks, but also should be able to use it transparently within his Machine Learning environment. The integration of distributed computing should be intuitively possible, without having to deal too much with the underlying concepts. This especially means that building and configuring new clusters of processing nodes should be as easy and fast as possible. Starting new processing nodes and connect them to a cluster should furthermore not require too much knowledge about distributed computing and network issues.

A further aspect of usability is monitoring: the user might want to know whether his cluster is up, how many processing nodes are in it and where these nodes are located. This allows better control of the whole system. In the case of failure it usually is important to know what and on which node the failure has happened.

NF5	The system should **hide** the **underlying distribution concepts** as much as possible to the user.
NF6	**Cluster initialization** should be as easy and fast as possible.
F7	The system should provide appropriate **monitoring** of processing nodes and feedback in the case of failures within the cluster.

3.2 Discussion of Frameworks

Several approaches and frameworks exist for distributed and grid computing. One approach or framework may fit the needs of a distributed application context very well, whereas others may entail huge drawbacks or even have negative impact on overall application performance. Which framework to choose for a particular application strongly depends on the goals that a framework trys to fulfill, i.e. which problems they usually address and which applications they are optimized for. In the following, different frameworks are discussed, referring to the needs and problems of distributed Machine Learning with MapReduce within RapidMiner, which have been outlined in the previous section. Each requirement is referenced and discussed for the frameworks. The discussion furthermore provides the basis for integrating the appropriate components within the design step.

The discussion takes a look at three frameworks for distributed computing: *Hadoop* and *GridGain*, the latter one in conjunction with *Coherence* as underlying data grid. There are several other approaches and frameworks, which may be possible candidates for an integration with RapidMiner, but considering and discussing all of them in detail would go beyond the scope of this thesis. However, there are several reasons why Hadoop, GridGain and Coherence have been chosen for detailed discussion: The first important aspect is that all frameworks are written in Java, which fosters integration with RapidMiner, as it is also purely written in Java. Hadoop, as the first framework under consideration, is widely used in popular applications (compare section 2.3.2) and often referenced in scientific publications, so its suitability for distributed computations has been proven by a big community. Furthermore, it is a direct implementation of Google's MapReduce paradigm, which is especially inspected as a programming model for Pattern Recognition and Machine Learning techniques in this thesis. GridGain on the other hand also provides ability to perform MapReduce like computations and by this also serves as a candidate for inspection of MapReduce. In addition to this, it seamlessly integrates with several grid and and enterprise software solutions, especially data grids. Thus, it is far more easy to build up and evaluate a distributed system on different technologies. Furthermore, GridGain's concepts and interfaces appeared to be very intuitive and the framework is very well-documented. Coherence has been chosen as Data Grid solution, since GridGain provides integration with it out of the box. A big advantage of Coherence is the ability to have data hold in memory instead of managing it on file system. It provides several different cache topologies, which are highly configurable. As with GridGain, the documentation is very clear and gives many tips and suggestions for development.

3.2.1 MapReduce Support

Both Hadoop and GridGain do support MapReduce-like computations (**F1**). Hadoops MapReduce is very close to the specification made by Google, whereas GridGain has a more rudimentary MapReduce implementation, which mainly focues on providing a divide & conquer solution. A main focus of Hadoop is to make computations on very large data sets possible. These data sets may be tera-bytes in size [22]. Hadoops main responsibility basically is splitting large data into smaller data subsets for processing. It is therefore very close to Google's idea of MapReduce. GridGain, as a computational grid, on the other hand focuses on splitting logic and not data in the first place. The MapReduce part of GridGain therefore splits a task into subtasks and distributes the computation, thereby not looking so much on how data is efficiently split and distributed. GridGain assumes to have this done by some underlying data grid. It therefore integrates with several Data Grid solutions out of the box, the user may choose a Data Grid which fits his problems best, but on the other hand has to rely on further technology.

In both frameworks it is possible to utilize multicore capabilities as well as multiple machines (**F2**). They both provide options to configure the number of working processes or threads on a machine, which implicitly allows control over the number of cores which are used on a single machine. By default, they are designed to operate on multiple machines and aim to scale with the number of machines.

3.2.2 Data Handling

Fast access to the data within maps and reduces is not only crucial for single MapReduce jobs, but especially when performing multiple jobs iteratively on the same data (**NF0**). One important aspect therefore is data affinity (compare 2.2.1), which is captured by requirement F6. Assuming that data affinity is given on every single machine, there is still a need to have the data in a form which allows fast access in each iteration. Hadoop's MapReduce mechanisms are strongly coupled to the file management of its underlying HDFS. This means that input data is typically given in form of large raw files which are distributed as blocks within the file system cluster. This concept performs well for very large text or similar input, but entails drawbacks when working with Java objects. The objects have to be serialized when writing to files and deserialized when reading from files. But especially fast reading is crucial for overall performance when doing multiple MapReduce jobs in sequence on the same data. Reading from file system and time-consuming deserializations of the same input objects in each iteration may be a huge bottleneck for performance.

In a distributed cache solution like Coherence the data is usually hold in memory and therefore reading from file is not needed. The need for deserialization of data only arises when objects are kept in cache in a serialized form. Using a cache as underlying data grid therefore can have advantages over using a file system or some disk approach when doing iterative MapReduce operations. However, when considering very large input data that do not fit into memory at once, reads from disk and deserialization usually become necessary. In cases like this Hadoop probably performs better since it is thought to do performant disk reads in large scale data extensive applications [45].

In RapidMiner, the intended data sets usually do fit in memory, since it is designed as standalone desktop application. Therefore a cache solution may result in better performance in many use cases, especially for iterative data access. When thinking of Concept Detection as a use case, the emerging data sets become very large. As outlined in chapter 2.4, a performant Concept Detection system needs to be trained on a very large data base, consisting of thousands, millions or even billions of images. Not only the images themselves require much storage space, but also the extracted interest point vectors which are then clustered to build the visual codebook. Considering large scale data processing scenarios like Concept Detection in Videos, Hadoop may be a better choice for distributing data.

In most cases data affinity of computations is not only crucial for fast distributed data processing, but it is necessary to make worth the effort of distributing data and computations (**F6**). Hadoop is explicitely designed to use data affinity as it drives computations depending on locality of data blocks within HDFS. Without that mechanism, the data blocks, which are by default 64 MB in size, would have to be transfered to the place of task execution, which would not be feasible. GridGain is not designed to drive computations depending on data locality in the first place. It just allows fine-grained control over how jobs are distributed within the grid. In conjunction with some underlying Data Grid the computations can be scheduled in a way to respect locality of data. For example, when looking at Coherence as an example for such a Data Grid, GridGain provides the *GridCoherenceLoadBalancingSpi*, which can be used to associate jobs with certain keys stored in the cache. By this way it is possible to schedule the jobs to a node which contains a replica of the data corresponding to that key. Figure 3.2 illustrates this load balancing mechanism.

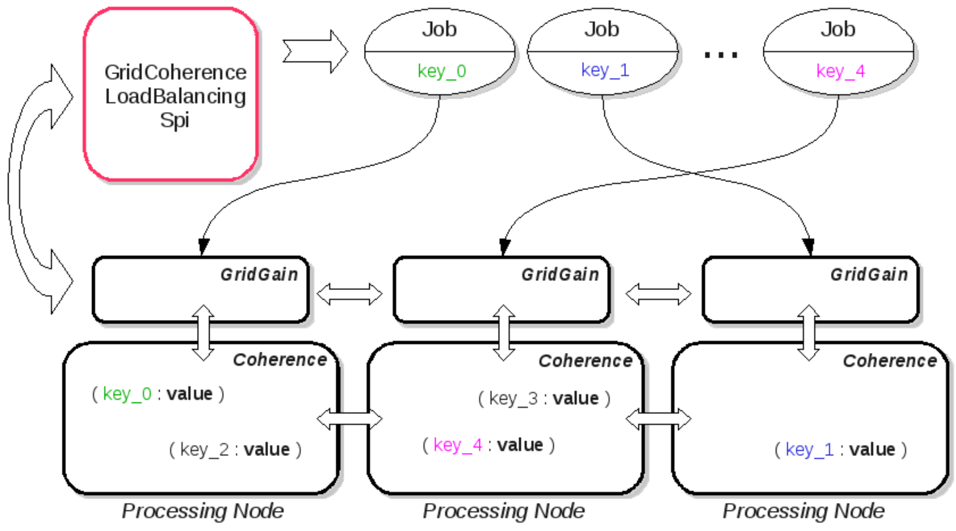

Figure 3.2: GridCohrenceLoadBalancingSpi: Each job contains the key which is used to store the corresponding data in the distributed cache. The jobs therefore can be scheduled with respect to the location of this key.

Efficient handling of static data like parameters or configurations within MapReduce is an important requirement especially for Machine Learning applications (**F3**). Hadoop provides an interface named *DistributedCache* which allows specifying different read-only files or archives on the HDFS for transferring statically to all slave nodes before map or reduce tasks are executed. This includes drawbacks if the static data is given in form of Java objects. These have to be read and deserialized from file before map and reduce executions, which consumes additional time. Furthermore, there neither is an obvious way to store or change static data for a next iteration within map or reduce tasks, nor it is possible to have a shared synchronized memory space for tasks.

GridGain provides a functionality named *DistributedTaskSession*, which allows specifying of attributes which can be seen and modified by all sub-tasks within some distributed task. This may be used to transfer static data to all sub-tasks before execution. Another way would be to utilize the underlying data grid to replicate all data to the nodes before execution. In Coherence it is possible to define a replicated cache which can be used to provide static data on all machines. The data stays in memory and changes to it can be synchronized with all other machines.

The replication of data is important to ensure data protection in the case of node failure (**NF1**). If a machine crashes and therefore loses the data on its side, there still should be copies of that same data units around somewhere in the cluster. But it may take up too much disk space to store each data unit on every node. Hadoop by default stores each data block three times in the HDFS. If one node failes and a data block is lost, it is replicated again from one of the other blocks to a number of three copies. The number of replica is configurable and therefore can be adjusted to the needs of specific applications. In Coherence there are different cache topologies, each allowing other ways of distributing the data. A "replicated cache" does hold all the data on every node, which may be best suited for providing static data needed by all jobs. Another cache topology is the "partitioned cache" which works similar to the replication mechanism in Hadoop's HDFS. The whole cache is paritioned into subsets and every subsets is replicated certain times on other nodes of the cache. Therefore data loss is also prevented in case of using Coherence as underlying data grid.

The need for supporting complex input data for maps and reduces has been identified (**F4**). Since Hadoop is very close to Google's idea of MapReduce, it defines the input of maps and reduces as key/value pairs. This implies that complex input data must be bundled into one value object per key. As Gillick et al. figured out, much of the existing code of Hadoop assumes that input data is packaged in one file that can be distributed across the network independently of other data [19]. Tying together multiple different input files therefore is not trivially possible. The most straightforward way therefore is to bundle all data within the same input file. GridGain itself does not provide sufficient functionality for splitting data and distributing it, it rather controls computations. The bundling of complex data therefore has to be done by the underlying data grid. Coherence as a possible Data Grid solution has the ability to co-locate data which is relational in nature. It is therefore possible to tie together multiple input objects on the same machine and use it as one input source.

Summing it up, both Hadoop and GridGain with Coherence provide effective means to handle data in a way that fits the needs identified in 3.1. The central point in which they differ and which is important in the context of RapidMiner, is the way in how they provide the data, i.e. provision from file system vs. provision from memory. Since RapidMiner first of all handles data sets which fit into memory, it is reasonable to choose a distribution approach which holds data in memory instead of writing and reading it from file when accessing. As said before, in cases where data does not fit into memory any more, a file system approach probably results in better performance. In chapter 4, it is also shown that both approaches lead to significant differences in performance.

3.2.3 Failure Handling

In any distributed system, especially when thinking of large scale applications, it is a rule rather than an exception that single tasks or nodes will fail or go down. Appropriate failover mechanisms are therefore important to ensure trouble-free distributed computations (**F5**).

In Hadoop, failover is done four times by default for each failed task until finally marking the task as failed. When rescheduling the task in the case of failure, Hadoop tries to avoid executing the task on a processing node where it previously has failed. A task that finally fails after four (or some configurable number) of attempts will bring the whole job in a state of failure. In some cases, this may not be an appropriate behavior: In many data processing applications there sometimes are corrupted data sets, for instance some text lines which do not fit the expected format. This means that small parts of the whole data can cause single tasks to fail, but the overall computation results may nevertheless not be influenced very much by these failures and therefore still be valuable. For such cases Hadoop allows to configure the maximum percentage of tasks that are allowed to fail without triggering job failure.

GridGain's failover mechanism is handled by *GridFailoverSpi*. By implementing this SPI one gets fine-grained control over how jobs are rescheduled in case of failure. Default implementations include a failover mechanism very similar to the one of Hadoop: The *GridAlwaysFailoverSpi* reschedules a job a maximum number of times until it is finally marked as failed. For each attempt another node is chosen to try execution. Another implementation is the *GridNeverFailoverSpi* which never tries to failover a job, but directly marks the whole task as failed. This may be useful in applications where execution of a job on each node is necessary.

Totally viewed, both frameworks fit the requirements for reliable distributed computing, as identified in context of this thesis.

3.2.4 Object Serialization

Since both Hadoop and GridGain are purely written in Java, they can be integrated into RapidMiner within the language, which is a great development advantage. A certain integration issue is the distributed handling of specific RapidMiner object types within Hadoop and GridGain (**NF3**). In order to transfer data within the distributed environment, it is necessary to serialize and deserialize objects. In RapidMiner, the IOObject interface, which is responsible for representing data that can be passed within RapidMiner processes and persisted on disk, can be serialized in two ways. The first is to use Java Object Serialization[1], which is included in the Java language. The IOObject interface extends the Serializable interface in order to do so. The second way is XML serialization with XStream[2].

Hadoop provides its own interface for this purpose, namely the *Writable* interface. It gives fine-grained control over how objects are serialized and deserialized. Several implementations are available for common object types like Integer, String or Arrays. For specific application types the user has to implement the serialization procedures by himself, which can result in huge development efforts. But Hadoop also provides a plugin mechanism for custom serialization frameworks. This for example allows the usage of the Java Object Serialization included in the Java language, but also external frameworks like Apache Thrift[3] or Google Protocol Buffers[4].

GridGain and Coherence both can also be used with different serialization frameworks. They both provide support for standard Java Object Serialization, but also for own implementations or external serialization solutions. GridGain by default uses JBoss serialization[5] to transfer objects between nodes, but also allows to use XStream or to implement custom serialization via the *GridMarshaller* interface.

[1]http://java.sun.com/javase/6/docs/api/java/io/Serializable.html
[2]http://xstream.codehaus.org/
[3]http://incubator.apache.org/thrift/
[4]http://code.google.com/p/protobuf/
[5]http://labs.jboss.com/serialization

Much more important are the serialization mechanisms of the underlying data grid, since data is usually much bigger in size, so that serialization and transfering objects often becomes a bottleneck in distributed environments. Coherence therefore provides its own interface, namely the *Portable Object Format*[6] (POF), which is similar to the Hadoop Writable interface in that the developer has to implement serialization by himself. But it is designed to allow indexing fields of Java Objects and by this supports the access on single fields of cached objects without loading and deserializing the object as a whole.

The right serialization framework not only is a precondition to distribute data, but also for doing it performantly. It is therefore important that the serialization used in this frameworks can be changed easily. Both Hadoop as well as GridGain and Coherence provide effective means for doing this.

3.2.5 Deployment

Deployment and redeployment of classes within distributed environments often inhibits fast development and testing of applications. Thus, deployment should be as automized as possible to gain development productivity (**NF4**). In Hadoop, deployment of new software is done on starting time of jobs. The class files have to be packaged into a Java archive (Jar), which can then be transfered to all nodes. When the TaskTracker starts a new JVM to execute map or reduce tasks on the remote node, the Jar file and its classes are available on classpath.

GridGain provides a remote class loading mechanism. When starting a new task, the classes are automatically deployed on all nodes that participate in executing the task. By this it is not necessary to put new classes at classpath. When modifying code, only the node that starts the task has to be restarted to load the new classes locally, remote nodes stay running and reload classes each time when executing jobs.

From a developers point of view, it appears to be more productive to have class loading automated on starting time of the application. But as experienced during work with GridGain in conjunction with Coherence, there often arise problems caused by classloading issues: the class loaders used by GridGain and those used by Coherence are not necessarily the same, which for example may lead to failures in which classes are not found within Coherence and cannot be deserialized. Thus, in some cases the use of remote class loading might be problematic. Furthermore, more complex deployment process like packaging Jar files as given in Hadoop, might also be supported by appropriate building tools, so this overhead in development would be negligable.

3.2.6 Cluster Initialization

From a users point of view, it is important that building a new grid can quickly and easily be done, even without having specific knowledge about distributed computing or the used frameworks (**NF6**). In both Hadoop and GridGain it is possible to start new nodes relatively fast. Nonetheless, for both it is usually necessary to do some configuration in order to start nodes, connect them to an existing cluster and to adjust all the components to the given environment. Most of these configuration issues are concerning IP addresses and ports of local and remote nodes, number of threads or processes on the node, Java heap size, logging mechanims and others. Sometimes it is also necessary to do some environment specific settings, for example when having to pass a firewall. Even though some of the configuration parameters can be set up and aligned prior to the release of a specific application package, there most often still will be open configuration issues.

[6]http://coherence.oracle.com/display/COH35UG/The+Portable+Object+Format

As experienced during working with both frameworks, it turned out that GridGain tends to be more comfortable to set up and start than Hadoop, especially when having little knowledge about the frameworks. For example, Hadoop requires to have a master computing node, including a running NameNode and a JobTracker process. This means that every node has to be configured with IP addresses and ports which these processes are listening to. This requires specific knowledge about the machines of the cluster, and it implies the user having knowledge about the concepts of Hadoop, i.e. the topology of a Hadoop cluster. GridGain has different discovery and communication mechanism, but by default new nodes are discovered by IP multicasting, which means that nodes can find each other and connect automatically[7]. Other examples of more complex configurations for Hadoop include SSH settings and the creation of a dedicated Hadoop user on Linux systems[8]. However, in most cases both Hadoop and GridGain require at least minimal knowledge about the frameworks or network issues to start and adjust to the environment.

3.2.7 Monitoring

A user may want to have a concrete overview over his cluster. Appropriate monitoring mechanisms are therefore important (**NF7**). In Hadoop, all components such as JobTracker, TaskTracker, NameNode and DataNode do expose themselves via a web front end. By this, monitoring is possible within common web browsers. The different nodes provide information about file system structure, actual running jobs, map and reduce status, output of tasks and reports of failures. In GridGain, monitoring is supported via JMX MBeans. Most parts of GridGain expose themselves via MBeans and can be monitored using JConsole[9]. It is therefore possible to view configurations for each node, have a look at actual work loads and get an overview over the cluster. All this information is also directly accessible within applications via the API. The monitoring facilities of both frameworks are appropriate solutions for this thesis.

In summary, it can be said that comparing Hadoop and GridGain with Coherence is not really possible in general, as both frameworks are quite different in many aspects and pursuing different goals. Hadoop concentrates on making large scale data processing possible, providing a distributed file system which is designed to store tera- or petabytes of data. GridGain focuses on beeing a Computational Grid software which aims to allow fine-grained control over scheduling and execution of jobs, independent of how data is provided in this context. Coherence, as a distributed caching solution, first of all aims to provide fast access to data, for example as an interlayer between a database and an application. To provide a compact overview of the presented discussion, the main points which have been considered for comparison are listed in Table 3.1.

3.3 System Design

In this section, the architecture of a concrete system for distributed computing with MapReduce and its integration with RapidMiner is presented. The design decisions made are mainly based on the requirements identified in section 3.1. If a certain design approach addresses a specific requirement, then it is annotated with the

[7]This is an ideal case. Firewalls and network must support this.
[8]These configrations are not absolutely required, but often recommended, for example in [45].
[9]http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html

Features	**Hadoop**	**GridGain & Coherence**
MapReduce Support	Yes (++)	Yes (+)
MultiCore	Yes (++)	Yes (++)
Multiple Machines	Yes (++)	Yes (++)
Data Provision	File System (-)	In-Memory (++)
Data Affinity	HDFS (++)	GridCoherenceLoad-BalancingSpi (++)
Static Data Provision	File System (-) (*DistributedCache*)	In-Memory (++) (Replicated Cache)
Replication	HDFS (++)	Partitioned and Replicated Cache (++)
Complex Input Objects	Have to be bundled within one file (-)	Co-location of objects within the Cache (+)
Fault-Tolerance	Failover Mechanism (++)	GridFailoverSpi (++)
Serialization	Writable Interface (++)	Portable Object Format (++)
Deployment	Jar-Files (+)	P2P-Classloading (+)
Cluster Initialization	Start Scripts (+)	Start Scripts (++)
Monitoring	Web Interface (++)	MBeans (++)

Table 3.1: Comparison of Hadoop and GridGain with Coherence in the context of RapidMiner.

corresponding reference number. After the abstract description of the system there follows a more detailed explanation on how the system is realized, first with Hadoop, second with GridGain and Coherence. To avoid confusion, note that Hadoop and GridGain have different denotations for whole computations and single map and reduce computations (compare 2.3). In context of the developed system, whole computations are denoted as jobs, whereas map and reduce computations shall be referred to as tasks.

3.3.1 Architectural Components

The developed system consists of three major components, namely *Distribution*, *MapReduceSpecification* and *DataLayer*. The **Distribution** interface exposes one major method, the *startComputation()* method. It is the main interaction point for RapidMiner (or any software) to perform the distributed computation and to get output from it (F0). Furthermore, the Distribution interface is meant to encapsulate specific characteristics of the underlying distributed computing infrastructure, i.e. the frameworks in use. This includes for example configuration issues, processing instance management and job preparation.

The computation itself can be specified by some **MapReduceSpecification**. It defines the logic of a computation in form of a *map* and a *reduce* function (F1). The map and reduce functions are semantically similar to the map and reduce functions in Google's MapReduce specification in that they perform parallel computations and aggregate the partial results. But they differ to them in that there is only one reduce task, which gets all the results from the map tasks in order to aggregate them in a defined way. The reason for designing the semantics of the system's map and reduce functions in this way is to decrease complexity of the system[10]. Even though such modelling may lead to performance degradation in cases where parallel reduces are possible, it turns out that allowing multiple reduce tasks in parallel adds more complexity to the system and therefore puts greater demands on the underlying distribution software. Furthermore, the requirements analysis has not explicitely revealed a special need for multiple parallel reduce tasks.

Data access to map and reduce tasks is given by the **DataLayer** interface. It exposes methods to provide and access two types of data: **individual mapping data** and **common static data**. Individual mapping data refers to parts of the whole main input data, which can be worked on in parallel by individual map tasks. Static data refers to data which is common to all map and reduce tasks. It must not be split and is accessable by all tasks (F3). The distinction of those two types of data is meaningful for example when considering how data can be distributed across processing nodes: Individual mapping data must be stored at least once, ideally at the location of map computation to improve access-time consumption (NF0, F6), static data on the other hand should be accessible by all map tasks, which implies to have it replicated several times across the cluster, at best once for every processing node. Figure 3.3 illustrates the relationship between the components of the system.

This thesis explored the opportunities of the MapReduce model, especially in the context of Pattern Recognition and Machine Learning (compare 3.1.2.1). It has been shown that MapReduce can be applied to many of these algorithms and techniques (and also to other applications) in a straighforward manner, and by this transparently gain computation perfomance for these techiques within a distributed environment. The components of the developed system, especially the MapReduceSpecification interface, provide this MapReduce functionality to Rapid-

[10]It can be easily shown that every regular MapReduce computation which may use multiple reduce tasks also can be modeled as a MapReduce computation with only one reduce task: change the output (key, value) of the map function to (default_key, (key, value)). The whole map output will be delivered to only one reduce task which still can aggregate with respect to key.

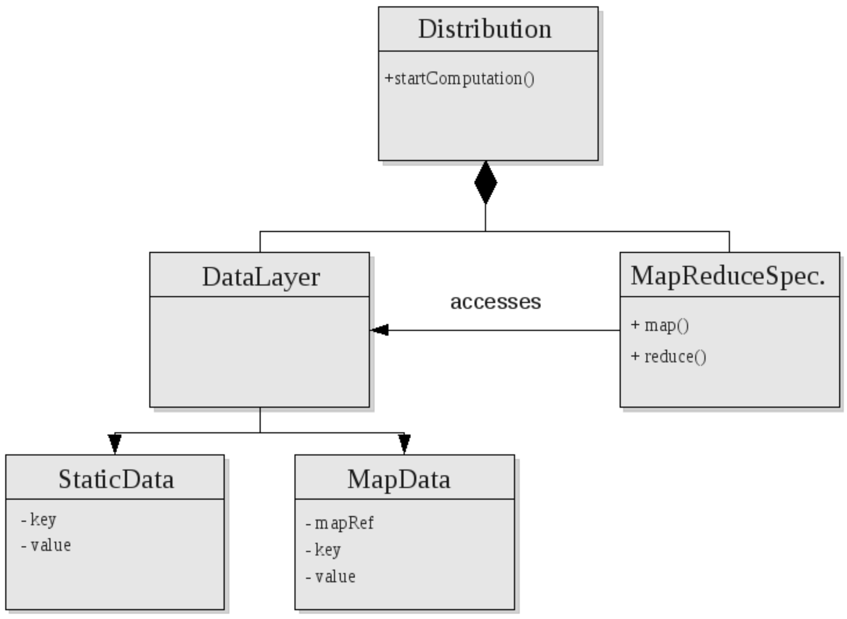

Figure 3.3: Relationship between the components of the developed system.

Miner, PaREn and other Machine Learning developers in a meaningful and easy to use way.

3.3.2 Functionality

Before the user starts computation, he specifies input data by means of the DataLayer. First, the whole data must be split into individual parts which serve as input to the parallel map tasks. How to split the whole data into such parts is left to the user. By not dictating how to split data to provide map inputs, the system stays flexible and gives a more fine-grained control over how input is worked with in the map tasks[11].

The map and reduce tasks can access the DataLayer during computation and fetch corresponding data. The input data for a single map task can consist of several data units, which may be logically independent in the first place, but group together to a complex input data entity (F4). Each individual map task has assigned a map reference number. Data units, which group together as input for an individual map task, are all referenced by the same map reference number and therefore can be handled together by the system. Each map task receives exactly the input data units which correspond to its reference number. The results of the map tasks are forwarded to the reduce task and aggregated to a single result, which is then returned to the user who invoked startComputation().

Since there is only one reduce task, there is no need to control data flow according to the keys produced by the map step. Data therefore are not given as key/value pairs in the sense of Google's MapReduce. But each map task has access to the collection of data units corresponding to its map reference number. The data units are represented in the system as a key/value pair, the data layer can therefore be seen as a key/value-store. This is conceptually not the same as the key/value

[11]The use cases in 3.4 provide examples on how to split data: a list of images can be split into single images, or a set of vectors can be split into subsets of vectors or even single vectors.

pairs passed between maps and reduces within Google's MapReduce paradigm. It just allows to provide logically independent data units for a single map task in a consistent manner within the DataLayer and by this allows passing complex input collections to the computations.

3.3.3 Integration with RapidMiner

The developed system may be embedded into the context of RapidMiner without any changes for the user. The only thing the user might be aware of is that there are other processing nodes involved, but he will use the system transparently without realizing how distribution is realized, just recognizing that certain methods or components of RapidMiner run faster.

From a developers point of view, the system totally abstracts from the frameworks which are used to perform the distributed computation (NF2). In principle, it is possible to implement the system by using differnet frameworks, from a data distribution view as well as when considering the computational aspects of the MapReduce part. The system has been realized in two ways, first with Hadoop, second with GridGain and Coherence. Both realizations allow to utilize multicore capabilities and to scale out on multiple machines (F2). This will be explained in the following subsections.

Within RapidMiner, the two types of realizations can be chosen by means of a Factory pattern [17]. The class **DistributionFactory** takes care of the proper instantiation of imlementations of the Distribution component, either with Hadoop or with GridGain and Coherence. By this design, decoupling of RapidMiner, the developed system, and the underlying distribution frameworks is fostered and further implementations can easily be integrated.

The DistributionFactory can be used within RapidMiner Operators to instantiate and utilize the capabilities of the developed system. In this context, an abstract class named **AbstractDistributionOperator** has been developed, which can be used as a foundation or just as example for implementing concrete distribution-enabled Operators. It allows any type of input and output data, especially the types which implement the RapidMiner IOObject interface. Therefore the input and output objects within RapidMiner can be directly used within the developed system. Furthermore, the AbstractDistributionOperator provides an abstract method named *split()*. By implementing this method the developer can specify how data is divided into parts, which then serve as input for the individual map jobs. At last, the developer must provide a MapReduceSpecification for his implementation of the AbstractDistributionOperator. This will then be used to perform the computation within the MapReduce framework of the developed system.

Implementations of the AbstractDistributionOperator seemlessly integrate with the RapidMiner GUI, as they also extend Operator. Within the RapidMiner GUI, it is possible to make configurations on actual processes on which the user is working on. Especially the Operators within a process can be selected and, depending of the kind of Operator and its provided functionality, the user can change the configurations and parameters of this Operator. In this context, implementations of the AbstractDistributionOperator provide one configuration option out-of-the-box: The user can directly select whether he wants to use Hadoop or GridGain with Coherence as underlying distributed computing framework. Summarizing, the AbstractDistributionOperator embodies an easily reusable component for quickly enabling computations to be executed in a parallelized and distributed manner, which also can be seemlessly embedded into existing RapidMiner processes. Figure 3.4 illustrates how the AbstractDistributionOperator fits into the context of RapidMiner. An example implementation is the use case of Interest Point Extraction, presented in 3.4.1.

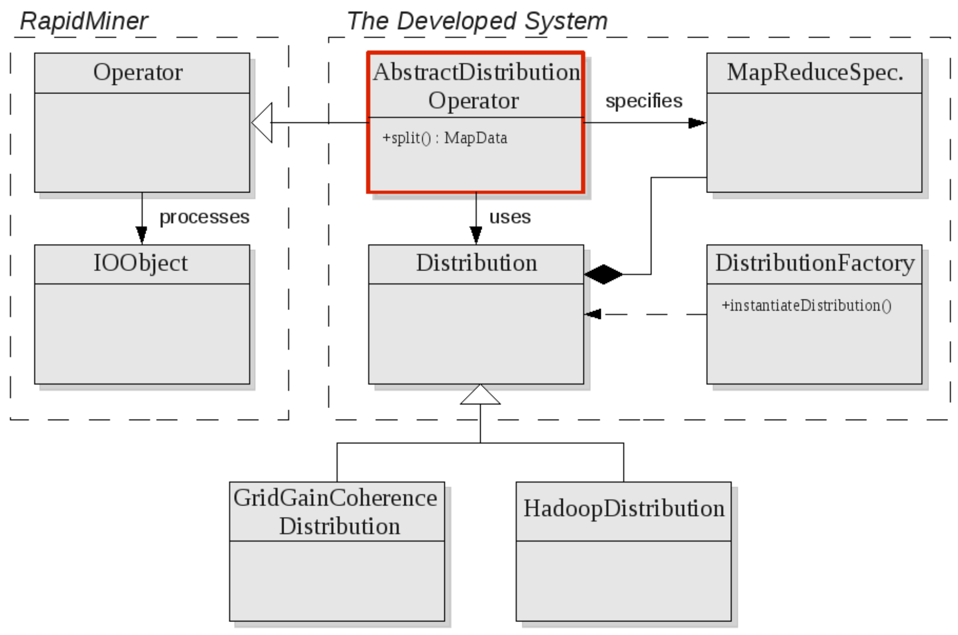

Figure 3.4: Integration of the developed system within RapidMiner by means of the *AbstractDistributionOperator*. The *DistributionFactory* instantiates concrete implementations of the *Distribution* interface, either with Hadoop or with GridGain and Coherence. This functionality can be transparently used as Operator within RapidMiner processes.

3.3.4 Realization with Hadoop

The realization of the system within Hadoop has been done by mapping the DataLayer component to the HDFS, while modelling the MapReduceSpecification part as a Hadoop MapReduce job. Figure 3.5 shows the architecture of the system when using Hadoop as distribution framework. The input data units for individual map tasks, which are given as Java Objects, are collected by the DataLayer and stored in serialized form within a file on HDFS before starting a job. Appropriate replication and coordination of data with respect to the MapReduce framework is handled by the HDFS (NF1). Therefore, the file which holds the input data, is automatically split and passed to the map tasks by Hadoop. Individual mapping data are read from and deserialized from HDFS file before task execution.

Static data that can be accessed by all map tasks and the reduce task are provided via the DistributedCache interface (compare 3.2). Since files on HDFS cannot be modified, but only totally overwritten, it is not trivially possible to change individual mapping data or static data within map and reduce tasks. Thus, the Hadoop realization of the developed system does not provide this functionality. Nonetheless, it turned out that also without this functionality, the system is sufficient for applying to the use cases presented in section 3.4, especially for doing evaluation with Hadoop.

By now, the DataLayer does only accept objects which implement the Java Serializable interface, which is sufficient for passing IOObjects of RapidMiner within Hadoop, even though it probably lacks in performance compared to other serialization mechanisms. By customizing Hadoop to utilize other serialization mechansims, it would also be possible to use those within the developed system.

The MapReduceSpecification is appropriately embedded into the MapReduce

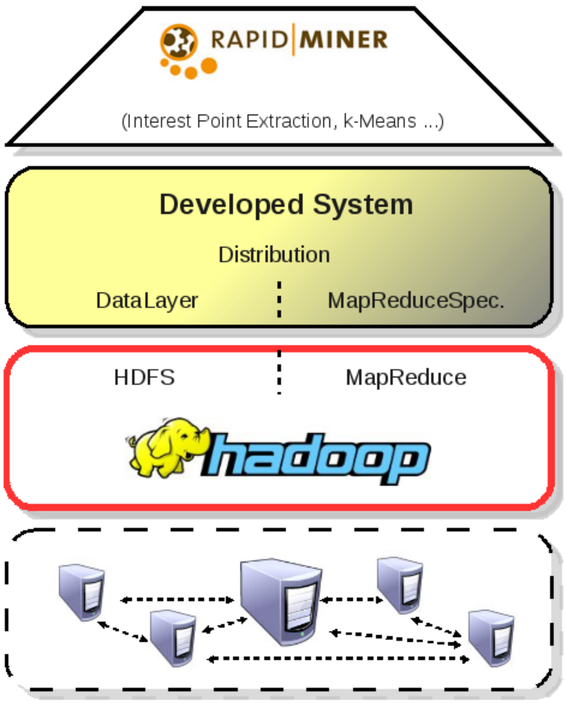

Figure 3.5: Realization of the developed system with Hadoop.

framework of Hadoop. The map tasks within Hadoop pass the corresponding input data to the map task given by the MapReduceSpecification, whereas one reduce task in Hadoop executes the reduce task given in the MapReduceSpecification. By this the systems MapReduceSpecification utilizes the parallelization capabilities of Hadoop's MapReduce framework. Also failover of single tasks is done automatically by Hadoop (F5) and therefore must not be handled explicitely by the system.

Using RapidMiner libraries or other external libraries is only possible if they are packaged into the Jar file which is deployed during starting the job, or by setting the classpath of every processing node in the cluster to include those libraries (NF3, NF4). Both methods have certain drawbacks: the Jar file may become very large in size if there are many dependencies in the code. Furthermore, these dependencies also have to be identified, i.e. missing dependencies would result in exceptions indicating that classes are not found. Adding the missing classes to the classpaths of all processing nodes implies that classes must be available on all machines of the cluster, which means increased deployment efforts.

As said in the discussion about frameworks in the previous section, building a Hadoop cluster may demand configuration efforts from the user and therefore forces the user to learn specific Hadoop concepts and maybe other concepts like network issues (NF6, NF7). Nonetheless, there already may exist a Hadoop cluster within the users environment, which is used for other purposes. This could then easily be utilized for the developed system and RapidMiner.

Another aspect of the developed system is monitoring (F7). Since Hadoop does already provide web interfaces for this purpose, there is no need to include detailed monitoring within the system or RapidMiner. The user may be referred to the monitoring capabilities of Hadoop directly.

Hadoop has been proven within many real world applications to be a very powerful framework for distributed computing and large scale data processing (compare 2.3.2). The distributed system is able to utilize these capabilities and by this integrate them into the the context of RapidMiner. This can lead to perfomance gains and

3.3.5 Realization with GridGain and Coherence

The developed system can also be realized by utilizing Coherence as DataLayer and GridGain to control the computations of the MapReduceSpecification. The architecture of the developed system can be seen in figure 3.6. The two types of input data can be stored in two ways in Coherence. Individual mapping data can be stored within a partitioned cache. Data units which form the input for a single map task can be colocated by using the *KeyAssociation* interface of Coherence. Objects with the same associated key are stored in the same partition, i.e. on the same machine. By this and using a partitioned cache, the input for a single map task is only stored once (not considering backup) in the cluster, i.e. on the machine where the map computation takes place (F6). Fast access is guaranteed since data is kept in memory by the cache (NF0). Static data can be stored within a second cache, which is configured as a replicated cache. Data is then replicated to all nodes and therefore accessible by all tasks (NF1).

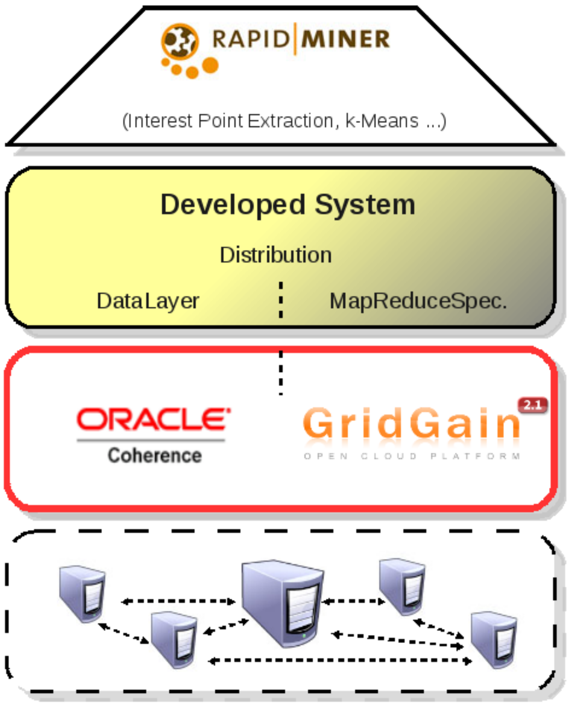

Figure 3.6: Realization of the developed system with GridGain and Oracle Coherence.

As in the Hadoop realization, the system only supports input objects which implement the Java Serializable interface, which is sufficient for the IOObject interface within RapidMiner. But this can also be changed by utilizing other serialization mechansims within Coherence and GridGain and probably would lead to performance gains.

The MapReduceSpecification can be executed by using the MapReduce capabilites of GridGain. It is designed to have many parallel map tasks and exactly one reduce tasks. It therefore fits the needs of the MapReduce model of the developed system. The maps must be aligned with the locality of data within the Coherence cache. GridGain provides the *GridCoherenceLoadBalancingSpi*, which allows to align single jobs with the data given in a cache. Data in the cache is given as key/-value pairs. The jobs can be associated with some key and GridGain automatically controls locality of computation by means of the GridCoherenceLoadBalancingSpi.

In case of failure, the computation is rescheduled by the *GridFailoverSpi* to a node which holds a backup of the corresponding data (F5). By default, this failover mechanism re-executes a single computation job three times before skipping the whole computation. It also would be possible to define certain exception types in which the job is not re-scheduled, but directly leads to global failure of the whole computation.

Deployment of RapidMiner classes or other external libraries is handled automatically by GridGain's peer-classloading mechanism (NF3, NF4). As discussed in 3.2, the developer or user does not have to copy any class files or libraries (Jar files) to the processing nodes. This can be an advantage for development time.

Building a grid can be fairly easy with GridGain and Coherence. GridGain ships with integration of Coherence out of the box, so that each processing node is a conjunction of GridGain and Coherence. In the best case it is possible to just start a script on a different machine and by this start another processing node for the cluster without doing any configuration and even without knowing any concepts of GridGain or Coherence (NF6). The distribution concepts could be totally hidden to the user in this way (NF5). Nonetheless, this is only possible if network and environment (e.g. firewall settings) support this.

Monitoring of the cluster can also be done by external interfaces. As explained in 3.2, it is possible to monitor GridGain with JConsole and JMX (F7). Furthermore, GridGain provides an API for accessing certain aspects which are interesting for monitoring. By this, monitoring could be easily integrated into the RapidMiner GUI and provided to the user within his working environment without dealing with external software like JConsole.

3.4 Use Case Application

In this section, the applicability of the developed system presented in 3.3 is demonstrated on two use cases: Interest Point Extraction and k-Means clustering. It will be shown that these methods can be adapted to the system's programming model, and by this can utilize its parallelization and multi-machine capabilites for accelerating computations. Both methods are important parts within Concept Detection in Videos with the Bag-of-Visual-Words approach, and both - but especially k-Means clustering - can be found as components in many other Machine Learning and Pattern Recognition applications. Thus, it is valuable to gain performance of these methods by utilizing distributed computing technologies. But they also serve as appropriate examples for Machine Learning algorithms in general.

3.4.1 Interest Point Extraction

In section 2.4 Concept Detection in Videos has been introduced. In this context the Bag-of-Visual-Words approach has been explained which includes two major steps: Interest Point Detection and Interest Point Description. In the first step, the interest points are located by a specific interest point detector. The coordinates of the interest points are then used to compute a description vector of the area around those points. Since these steps are naturally performed in sequence on a single video frame, they are handled as a single wrapped-up process in this thesis. This process is referred to as Interest Point Extraction. The details of Interest Point Extraction are not central to this thesis. Instead it will focus on how this process can be executed and accelerated within a distributed environment.

Training sets for Concept Detection consist of a large number of video frames (compare 2.4). Doing Interest Point Extraction sequentially on the whole data set therefore takes a long time. But Interest Point Extraction can be done independently for each individual video frame. This is where parallelization, and especially MapReduce can be applied: by modelling the extraction process for a single frame as a *map* task, it can be done separately for each frame on different processing nodes. The *reduce* step simply collects the extracted descriptor vectors. The process of Interest Point Extraction can also be seen as a *single-pass* learning algorithm as explained in 3.1.2.1 The whole data set is passed once to extract all the relevant information, i.e. the interest point vectors. Therefore, the corresponding MapReduce specification is very straightforward and would be as follows[12]:

$$map(frame\_id,\ frame) \rightarrow [(frame\_id,\ \vec{desc}_1), ..., (frame\_id,\ \vec{desc}_n)]$$

The *map* step takes a single frame as input. The key is some unique identifier for this frame, for example the file name. The resulting intermediate key/value pairs are again the frame identifier, together with descriptor vectors as values. These pairs are then grouped by frame identifier and passed to the *reduce* function, which is defined as the *identity*-function[13]:

$$reduce(frame\_id, [\vec{desc}_1,\ ...\ ,\ \vec{desc}_n]) \rightarrow (frame\_id,\ [\vec{desc}_1,\ ...\ ,\ \vec{desc}_n])$$

This MapReduce specification has been implemented upon the developed system shown in 3.3, and has been encapsulated into an *Operator* in RapidMiner. More precisely, the class **InterestPointExtraction** has been designed. Since there do exist many different methods of interest point detection and description (for instance SIFT and SURF), the operator has been designed as abstract. By this, new Interest Point Extraction operators can be implemented very fast. For this thesis, a concrete implementation for SURF detection and description has been created, which uses external libraries.

The InterestPointExtraction class inherits from AbstractDistributionOperator. By this, it is enabled to do its computations in a distributed manner. In the context of Interest Point Extraction, the input data is meant to be a collection of images, which shall be processed in parallel within the distributed system. RapidMiner does not support handling of images out of the box. It would be feasible to utilize the specific RapidMiner type *ExampleSet* in order to handle a plain list of images. However, a solution like this would lack proper representation of images and would not support distinguishing images from the nominal or numerical data types usually used in RapidMiner. It furthermore would not support any extensions for image handling or processing within RapidMiner.

Because of this, the new class **ImageObjectList** has been designed. It implements the IOObject interface and therefore can serve as input for the interest point extraction operator. It abstracts a list of images by referencing each image by its file name. In the context of this thesis, the images themselves are stored on NFS[14].

[12]The same approach can be found in [9]. In this scenario 6 million images have been processed within three days.

[13]This assumes multiple parallel reduce tasks, but can easily be adapted to the developed system, which only entails one reduce task.

[14]Network File System: http://tools.ietf.org/html/rfc3530

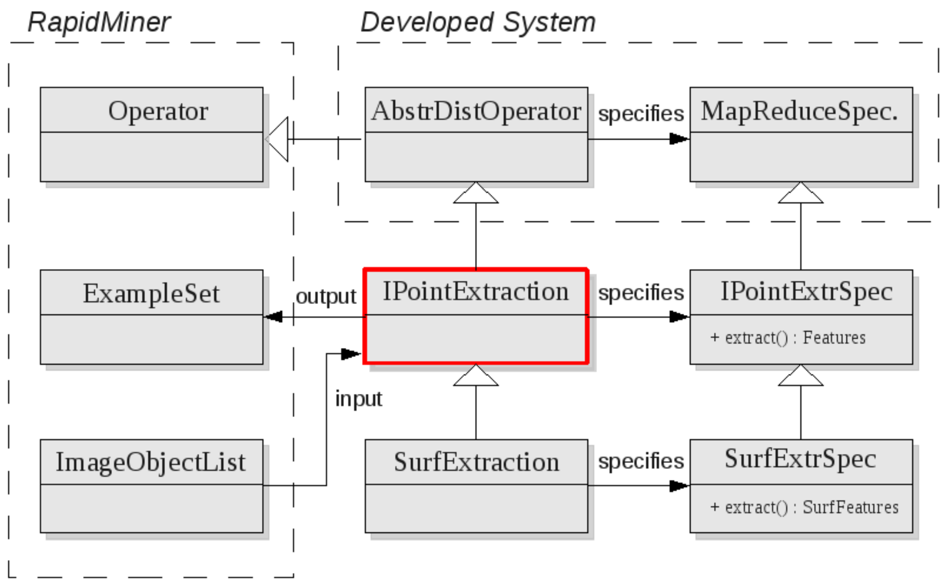

Figure 3.7: Distributed Interest Point Extraction within RapidMiner: the InterestPointExtraction class shares the functionality of a RapidMiner Operator, while utilizing the capabilites of a distributed system.

By this, they are accessible on all processing nodes, but are not distributed via the Data Grid of the developed system.

The extracted interest point vectors are collected within the specific RapidMiner type *ExampleSet*. They therefore can be easily reused for further processing within RapidMiner, e.g. for some kind of dimensionality reduction or for codebook generation with k-Means clustering. In section 3.4.3 it is illustrated, how such a data flow may look like. The relationship of the different components used in this scenario is illustrated in Figure 3.7.

3.4.2 k-Means Clustering

The second example use case in this thesis is k-Means clustering, which is described in Appendix A. As explained in 3.1.2.1, k-Means clustering is an *iterative* learning algorithm. It therefore is more complex than the previous example of Interest Point Extraction.

There are two approaches to parallelize the k-Means algorithm: The first approach is based on the recommendation that k-Means should be done several times with different random starting means. A possible parallelization could be achieved by copying the whole data set to all processing nodes and start the k-Means algorithm on each node with different random starting means. At the end of the parallelized computation, the algorithm would just pick the best result of all k-Means runs. This approach does not divide the data into subsets, but replicates all data to all nodes. Thus, it is not a data-driven approach and also does not really hit the intention of MapReduce, which explicitly aims to split computation logic according to how data is split. Nonetheless it is possible to implement this approach on the developed system presented in 3.3. In cases where the dataset is not exceedingly large, it may be feasible to copy it to all nodes and perform the compute-intensive k-Means method on it in this way. The k-Means algorithm can then also be seen as *single-pass* learning algorithm, but with a dataset, which is just multiple times larger in size. Therefore this approach is not discussed in more detail.

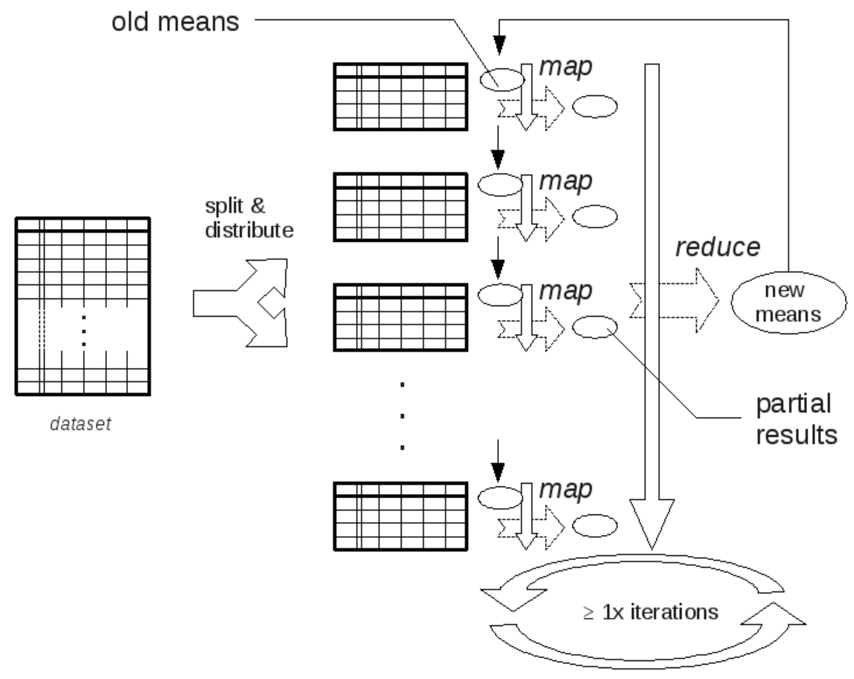

Figure 3.8: MapReduce applied to k-Means. The means are updated and serve as static input for the next iteration.

The second approach already has been demonstrated in different publications, e.g. in [10, 15]. It parallelizes the k-Means method when optimizing the means. The whole dataset is split into parts, which then serve as input for the *map* tasks. In many publications, the splitting is done on a per sample basis, but it also is possible to divide the dataset into subsets and let those serve as input for a single map task. This is more feasible in settings where the MapReduce framework produces much overhead time for instantiating single map tasks and loading the input data. When considering a single vector as input to a map, computation time for this map would be very short, but instantiating the task would take too much time and therefore diminish overall performance or even completely eliminate the advantages of parallelization. In this discussion the data therefore is divided into subsets, which also fosters integration and usage of RapidMiners ExampleSet type.

In each optimization iteration, the k-Means algorithm walks through the whole dataset, assigns each sample in the dataset to its nearest cluster mean, and computes the new means by averaging over the members of each cluster. This process can be parallelized on subsets of the dataset: For each subset, its samples are assigned to the nearest cluster means, and the new means are partially computed for this subset. This can be done independently for all subsets in a *map* step. Then, the partial contributions from all subsets are aggregated to compute the new means in the *reduce* step. The whole process is repeated until the means convergate or until fixed maximal number of iterations is reached. The whole process is illustrated in Figure 3.8.

In section 3.1.2.1, the class of *iterative* learning techniques have been described. Applying the MapReduce concept to them mainly entails two major challenges:

static data must be provided to all map tasks and the maps must have fast access to data in each iteration. In k-Means, the actual means must be provided to all map tasks in each optimization iteration. This can be done by using the DataLayer of the developed system, which provides an interface for broadcasting static data to all maps. Furthermore, the subsets which serve as input for the map tasks have to be quickly accessible in each optimization iteration. The DataLayer also provides an interface in order to do this, i.e. specifying individual and quickly accessible map input data. When using Hadoop as underlying solution, fast access is achieved by having the subsets on the place of computation for further iterations. When using GridGain in conjunction with Coherence, this also holds true. Furthermore, Coherence provides the map input data within memory, which allows a fast access in each iteration.

As in the case of Interest Point Extraction, this distributed k-Means clustering has been realized as an Operator in RapidMiner. RapidMiner itself already includes an Operator which performs k-Means clustering, but its implementation does not support computing on multicore or multiple machines. It therefore has been modified and adapted to make use of the developed system. The modified Operator can be used in the same way within RapidMiner as the standard implementation, but it furthermore automatically and transparently uses the capabilities of a distributed environment to speed up its computations.

3.4.3 Bag-of-Visual-Words

The two methods presented above can be easily connected as operators within a RapidMiner process. By this, thinking of Concept Detection in Videos, the Bag-Of-Visual-Words approach can be constructed within RapidMiner. For a RapidMiner user, it makes no difference - in terms of usability - wether he defines this Bag-of-Visual-Words process with distributed computing enabled operators or not. However, the whole process will profit from the speed-up provided due to the ability of scaling to multiple machines. Figure 3.9 demonstrates a possible Bag-Of-Visual-Words process in RapidMiner.

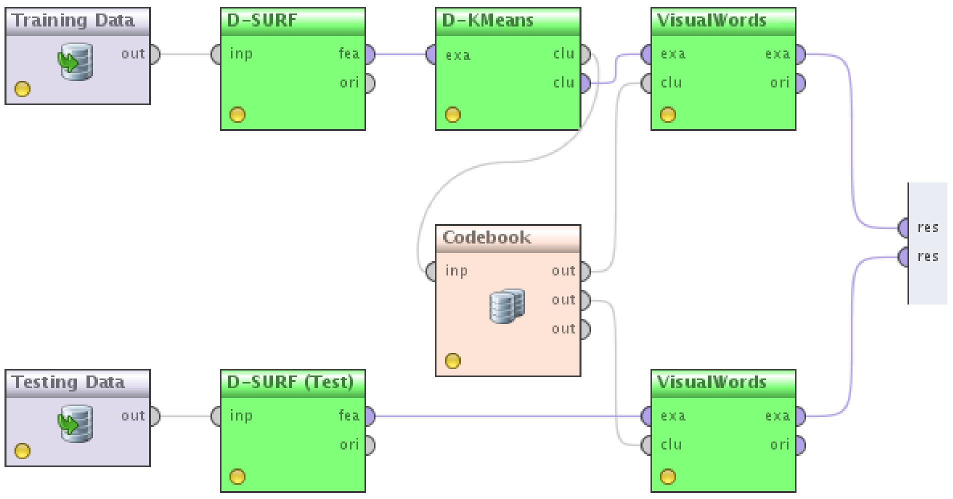

Figure 3.9: Bag-of-Visual-Words process in RapidMiner: the D-SURF operator receives ImageObjectList as input and extracts SURF interest points from the images. These are given as ExampleSet to the k-Means clustering, which generates the codebook. With it, visual word histograms for training and testing data can be constructed, which can be used for further classification processes. The D-SURF and D-KMeans, as distributed computing enabled operators, seemlessly embed into the process and are transparently executed within the distributed environment. By this the process is more performant as when using standard operators of RapidMiner.

Chapter 4

Performance Evaluation

The focus of this thesis is the application of distributed computing on Pattern Recognition and Machine Learning techniques. The main goal is to accelerate these processes by utilizing multiple machines and perform computations in parallel. In this chapter, the developed system is evaluated with respect to this goal. In order to do such an evaluation, appropriate experiments have to be done. Those require a well-considered choice of experimental parameters and an appropriate environment in which the experiments can take place. The experiments conducted in this chapter are all based on the two use cases presented in this thesis, i.e. Interest Point Extraction and k-Means. By this, the evaluation obtains a practical foundation and the system can be proven on real world problems. A further aspect of evaluation is the comparison of the two realizations of the developed system, first with Hadoop, second with GridGain and Oracle Coherence. In the following, the different experiments and their results are described in more detail.

4.1 Experimental Environment

All experiments are performed on five machines, each with same hardware, all running Linux as operation system. The machines are connected via a 1GBit full duplex intra-network. Each machine is equipped with Intel Atom CPU 330 1.60GHz Dual Core, with 512KB Cache and 3 GB RAM. The machines support Hyper-Threading[1], which means that each machine has four logical cores, summing up to a number of 20 logical cores. Note that Hyper-Threading on one physical core is usually not as performant as having two independent physical cores, since processes sharing the resources of this core will affect each other [8].

As the machines in this setting are all equally equipped, they build up a very homogenous environment. This is necessary for having meaningful results when scaling out on multiple machines. Among other things, the experiments aim to explore the dependency between computation time and number of machines. When increasing the number of cores or machines, it is expected that computation time should decrease proportionally. Invastigating this in a heterogenous environment is hardly possible, since different speeds of machines would lead to a distortion of measurements. The same holds true if a machine is occupied by other processes. Therefore it has been ensured that the machines are not under further usage during the experiments. Furthermore, Hyper-Threading must be considered when treating the environment as a collection of logical multi cores, since having two logical cores does not necessarily mean having the performance of two physical cores.

[1] http://www.intel.com/technology/platform-technology/hyper-threading/index.htm

Another important aspect is that the given environment is set up with commodity hardware, which for example might be found in offices. The presented experiments therefore prove the applicability of the developed system within a usual practical environment.

4.2 Tasks

The first experimental task is the extraction of interest points from a fixed collection of images. The number of images is 1500, each with a resolution of 320x240 pixels. The images themselves are provided via NFS on all machines, access time to them is therefore negligible. Basically, the inputs to the map jobs are references of the images into NFS, therefore the underlying Data Grid of the developed system (i.e. HDFS or Coherence) is not heavily used in this setting.

The interest point detector and descriptor used for extraction are both SURF. The number of extracted interest points is 188176, each point given as field of 128 double values, making the whole result dataset 184 MB large. This means that about 125 interest points are extracted from each image in average. The average time for interest point extraction of one image in this collection has been measured as 673 ms[2]. The overall time for extraction of interest points of the whole collection in sequence on a single machine has been measured as 15.8 min in average. All experiments have been repeated at least five times, outliers have been manually removed, and the results have been averaged.

The second task under consideration is k-Means clustering. In this setting, the output of the Interest Point Extraction serves as input to the clustering, i.e. 188176 samples, each with 128 dimensions. In contrast to Interest Point Extraction, the input data is given directly and not referenced on NFS. This means the data must be appropriately distributed to all nodes before execution. This can be seen as initialization, which consumes additional time in contrast to the standard non-distributed implementation in RapidMiner. However, this must only be done once, before starting to iteratively optimize the means. For each experiment, at least ten optimization iterations have been measured, outliers have been removed and the results have been averaged.

4.3 Parameters

By changing different parameters of the experimental setting, several aspects can be investigated which have influence on computation performance. These aspects concern distributed computing in general, but also characteristics of the used frameworks or specifics of the tasks under consideration. A main aspect is the overhead time produced by introducing distributed computing capabilities into the computations. Inherent in any parallelization framework like MapReduce is some computation overhead for managing the framework and distributing the jobs and the data. This overhead time usually decreases relative performance, even though one might exepect that increasing the number of processing instances would lead to a proportionately speedup. In the following, different parameters and their expected impact on computation performance is explained.

Distributed Computing Framework One essential topic of this thesis is the comparison of different distributed computing frameworks with respect to the goal of developing and integrating distributed computing capabilities into RapidMiner.

[2]This includes time for copying the image to a temporary folder and converting it into a grayscale image.

In chapter 3, different aspects of Hadoop, as well as GridGain in conjunction with Oracle Coherence have been reviewed and discussed. Nonetheless, the probably most important aspect which has to be considered as a decision criterion for one of the two solutions is performance. Therefore the two realizations of the developed system presented in chapter 3 are evaluated with respect to this aspect.

Single Machine: Number of Cores One requirement identified in section 3.1 is to make performant utilization of multicore capabilities possible. The frameworks Hadoop and GridGain with Coherence both support this feature. By just running a single instance on one machine, it could be investigated how well these frameworks scale with respect to performance using only the multicore capabilities of a single machine. A good performance in this setting would be useful for a RapidMiner user in cases where only one machine is available for computation. Therefore experiments may be set up which run the computations on a single machine, whilst varying the number of parallel threads or processes used within the MapReduce framework on this machine. A special case in this setting is to have exactly one thread for execution. Compared to the case of using no distributed computing at all, it demonstrates best how much overhead the frameworks produce, since the whole computation is done sequentially as when using no distributed computing.

Multiple Machines: Number of Machines A main requirement is the utilization of multiple machines to gain performance in a distributed environment. Therefore, experiments should be set up in which the number of machines is varied for the same computations. By this, the developed system mainly can be proved to be an appropriate solution for accelerating computations with the aid of multiple machines. Again the overhead time of distributing jobs and data can be investigated, but this time in a distributed environment, including the drawbacks of network communication and management of processing instances over multiple machines.

4.4 Results

In this section, the results of the different experiments are presented. The results are discussed with respect to the different aspects mentioned in the previous section.

4.4.1 Performance on a Single Machine

In the first experiment, the performance of the realizations of Hadoop and GridGain with Coherence are investigated on a single machine. This is done by setting up a "cluster" which only contains one machine. Both Hadoop and GridGain allow to control the number of threads (resp. processes) running on a single instance, i.e. on one machine. This number is varied in this experiment. The results for doing Interest Point Extraction are presented graphically in figure 4.1. The number of images per job has been fixed to 50.

It can be seen that computation performance increases for both realizations when increasing the number of threads. Considering enough threads, both frameworks prove that they can perform better than the standalone, sequentially working implementation without distribution. Hadoop could nearly double up the performance of the standalone version, whereas GridGain with Coherence achieves a speedup of about 2.5.

But as predicted, both frameworks perform worse compared to the standalone version when only allowing one thread for execution of the map tasks. This is due to the additional overhead of distribution management. The results also show that

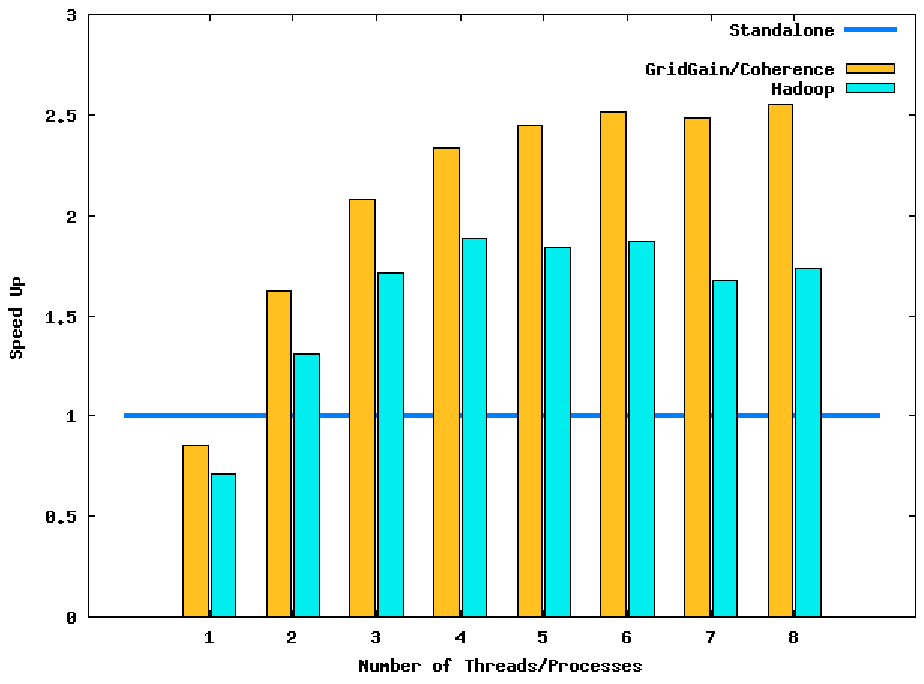

Figure 4.1: Performance of Interest Point Extraction, parallelized on a single machine: speedup is limited by the number cores.

increasing the number of threads does not lead to a linear increase of performance. In this setting, this can be explained by looking at the machine hardware: the machine only has four logical cores (resp. two physical cores with support for Hyper-Threading), which forces multiple threads to share these resources. Therefore performance not only does not increase linearly, but also decreases when having too much threads running at the same time, as can be seen when having more than five or six threads with Hadoop.

Also notable is the observation that performance relatively decreases a bit when looking at Hadoop with 5 and 7 threads and GridGain/Coherence with 7 threads. The reason for this is not known, but it could be due to a load imbalance because of an odd number of threads.

The second experiment investigates the k-Means algorithm. In contrast to the Interest Point Extraction setting, doing k-Means clustering requires to distribute data sets which may be very large. Since the following experiment is done on a single machine, distribution of data is actually not needed, but done anyway due to the design of the developed system. Nonetheless, this must only be done once, the data then is distributed for the following, usually large number of optimization iterations. Assuming a performance gain for these iterations, the additional overhead for distributing the data during initialization usually is amortized.

The experiment has been done with k = 100. In contrast to the Interest Point Extraction, not the duration of the whole process has been measured, but the average duration of a single optimization iteration. By this, the overhead for initialization and distribution of data is discarded in the results, but as said before this overhead is negligible when doing a large number of iterations. The performance results of doing k-Means clustering on a single machine are shown in figure 4.2.

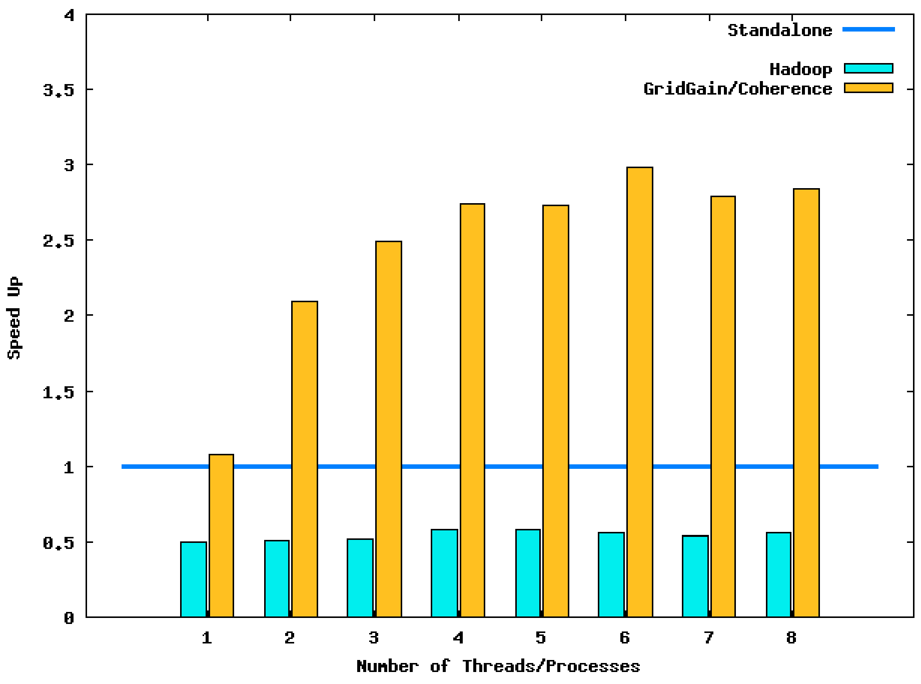

Figure 4.2: Performance of k-Means, parallelized on a single machine. Hadoop underperforms because it reads data from file system for each optimization iteration.

First of all, it can be seen that GridGain and Coherence perform similarly well as in the first experiment, whereas Hadoop never reaches performance of the standalone version. This can be explained by the fact, that Hadoop has to read the partial data sets from file in each iteration. This produces too much overhead and completely undermines a possible performance gain due to parallelization in this setting. GridGain and Coherence perform better in this case, since the partial data sets do not have to be read from file, but are kept in memory for all iterations. By this, the overhead for reading the input data in each map is small enough to gain performance due to parallelization.

One may notice that the results show a small performance increase of GridGain with Coherence over the standalone version, even when allowing only one thread for computation. As expected, the performance should stay below the standalone version because of the overhead of the distributed system. In this case, the reason for this relies in slightly different implementations of the two versions[3], which results in a better overall performance of the distributed version. This also shows that the time overhead of GridGain with Coherence does not preponderate that much.

4.4.2 Performance on Multiple Machines

In this experimental setting, the number of machines is varied. This means, in contrast to the former experiment on a single machine, that this setting really utilizes a

[3]More precisely, the k-Means implementation of RapidMiner follows a very strict object-oriented design, wrapping some array operations into method calls. This implementation has been modified to adapt to the MapReduce model of the developed system, thereby making use of arrays more directly in some places.

distributed environment, including potential drawbacks of network communication and cluster management. The number of threads per machine has been chosen as 2 in order to have a "natural" utilization of the two physical cores on each machine, trying to avoid affects arising from Hyperthreading or from having too much threads competing for resources on the machines. First, Interest Point Extraction has been done on multiple machines with both Hadoop and GridGain with Coherence. Figure 4.3 shows the performance gain with both frameworks when increasing the number of machines in the cluster. The number of images per job has been fixed to 20. By this, the number of jobs is large enough to appropriately make use of the available processing slots on all machines.

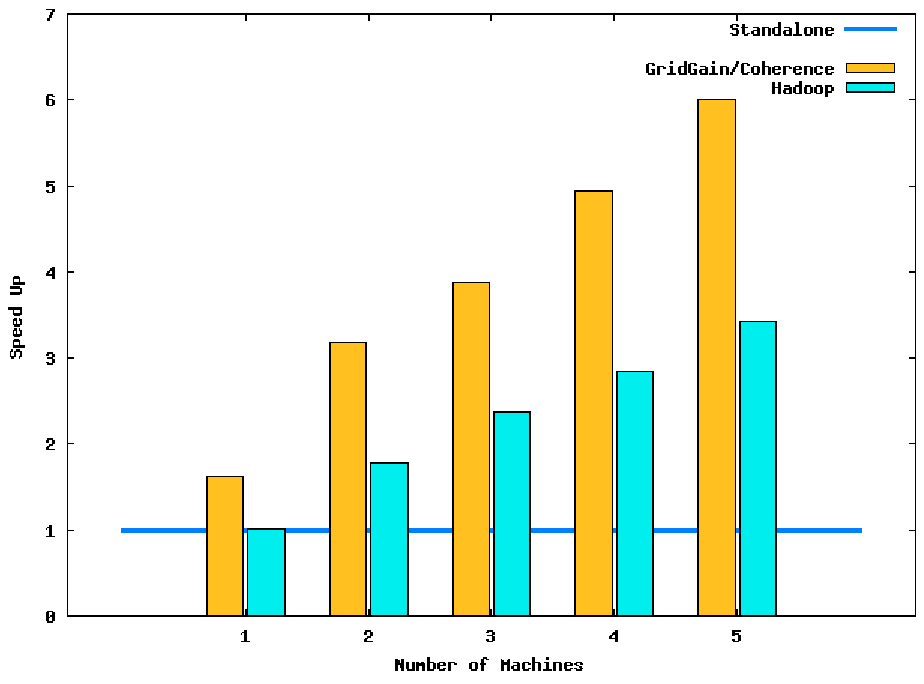

Figure 4.3: Performance on multiple machines: for Interest Point Extraction, speedup increases proportionally with the total number of threads.

When increasing the number of machines, both frameworks achieve speedup, even though Hadoop again does not perform as well as GridGain/Coherence due to its greater overhead. However, as one can see in the Figure, performance increase for both frameworks has a proportional relationship with the number of machines. Even though this experiment only shows performance scalability on five machines, it is reasonable to assume that adding more machines further increases performance in a similar way. However, a limit is at least given by the granularity of the tasks [32]. If number of tasks is too small, the job computation cannot be balanced properly on newly added machines and performance increase would not be proportional anymore.

This can be seen in the following experiment (Figure 4.4). K-Means clustering has been distributed on multiple machines. Again, each machine runs with two threads, the parameter k has been set to 200, the number of tasks is 20 in this setting. As in the single machine case, Hadoop does not perform well for k-Means, since it has to load the subsets for task computation from file in each iteration. In contrast to this, k-Means with GridGain/Coherence is up to 7.5 times faster than the standalone version when using 4 machines (i.e. 8 threads in total), which is almost a linear speedup. By this result, it is shown that the developed system can bring almost ideal performance gains in some cases.

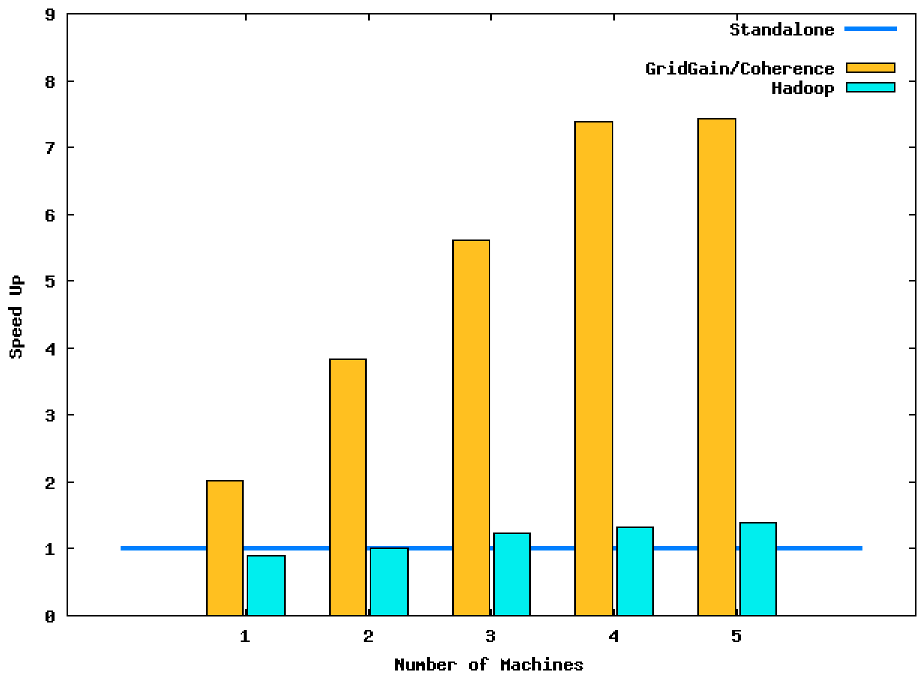

Figure 4.4: Performance on multiple machines: using GridGain/Coherence, k-Means has achieves nearly linear speedup with the total number of threads. However, load imbalance stops this increase for five machines in this setting.

However, the previously indicated limits due to task granularity can also be seen in this Figure. In the case of five machines no more speedup is achieved. As said before there are 20 tasks in this setting, which implies an optimal load balancing of 4 tasks per machine. After several repititions of the experiment, it turned out that some of the machines were assigned five or 6 tasks, which decreased performance down to the case of four machines. This imbalance is caused by Coherence, which gives no control over how data is distributed exactly and which not necessarily aligns data distribution to the needs of computational aspects.

Summarizing the results, it turns out that GridGain in conjunction with Coherence is a more performant solution than Hadoop when considering the problem contexts of Interest Point Extraction and k-Means within RapidMiner. It furthermore has been shown that both frameworks can lead to performance gains on multiple cores as well as on multiple machines. Especially GridGain and Coherence are able to almost linearly scale in performance in some cases. However, task granularity sets limits for performance increase.

Chapter 5

Conclusion

In the final part of this thesis a short summary about the contents of this work and its results is given, as well as an outlook on possible future work.

5.1 Summary and Results

In this work, the challenges and opportunities of applying distributed computing to Pattern Recognition and Machine Learning techniques have been examined. It has been figured out, especially in the context of the project PaREn, that these techniques usually are very data and computation intensive, which means that scaling them on multiple machines often becomes a precondition for feasible applications of such techniques. In this context, it has been shown that the MapReduce paradigm provides effective means for developers to parallelize computations within a distributed environment.

The main goal of this thesis has been the integration of distributed computing frameworks with the Machine Learning software RapidMiner. Several requirements for such an integration have been identified, covering issues arising from distributed computing with MapReduce and Machine Learning. Probably most important in this context are data affinity, i.e. the alignment of computations to data in a distributed environment, and fault tolerance.

The software frameworks Hadoop and GridGain in conjunction with Oracle Coherence have been reviewed in detail as possible candidates for an integration. The crux of the matter of this discussion has been that Hadoop holds data on file system, whereas a GridGain/Coherence setting holds the data in memory. The latter one has performance advantages in cases where datasets are small enough to fit in memory, as it is in the case of RapidMiner.

A system has been developed which integrates the presented distributed computing frameworks into RapidMiner. The system provides an intuitive MapReduce-like interface which takes care of the special needs arising when applying MapReduce to Machine Learning techniques. In addition to this, an abstract Operator has been designed, which seemlessly embeds the systems functionality into RapidMiner processes, thereby allowing RapidMiner developers to enable their processes and algorithms to utilize distributed computing capabilities.

The applicability of the system has been shown by implementing two techniques on top of the system, which arise from Concept Detection in Videos with the Bag-of-Visual-Words approach: Interest Point Extraction and k-Means clustering. The methods could easily be designed and modified to fit into the MapReduce model offered by the system. Performance evaluation results have shown, that by utilizing multiple cores or machines by means of the developed system, it is possible to significantly accelerate these processes. Specifically when using GridGain and Coherence as distributed computing framework, the system is able to achieve nearly linear speedup with the number of threads and machines in some cases, for example when doing k-Means clustering. It also has been shown that Hadoop performs worse than GridGain with Coherence in the context of RapidMiner, due to its file system data management.

5.2 Future Work

There are different issues concerning RapidMiner and the developed system which are of interest for further development and research efforts. Since the developed system already has shown good results regarding the applicability and performance of Interest Point Extraction and k-Means, it seems natural to explore the possibilities for applying other Machine Learning techniques like SVMs, PCA or neural networks on top of it.

Other techniques, which are especially important in the context of PaREn, are cross-validation or different parameter optimization methods. It turned out that the implementations of these techniques in RapidMiner are very difficult to parallelize within a distributed environment, since they naturally include completely nested RapidMiner pocesses which cannot trivially be serialized and executed on another machine. Further efforts should consider the exploration of possibilities to distribute these methods, and especially nested RapidMiner processes, on multiple machines.

From a software engineering point of view there are several apspects which are worth considering. The testing of different serialization frameworks may lead to better performances regarding the communication, distribution and access of data. Moreover, other frameworks for distributed computing could be reviewed and evaluated in the context of the developed system. Regarding the computational aspects, this could be *Terracotta*[1], which is a framework for distributing applications on JVM level, or the more heavy-weight Grid Computing framework *Globus Toolkit* [2]. Regarding the data grid functionality, different caches could be tested within the developed system, for example *Jboss Cache*[3] or *Hazelcast*[4]. Exploring the opportunities of other frameworks may bring benefits in performance and foster further integration with established tools for distributed applications.

In general, RapidMiner is not intended for distributed computing in the first place. The handling of data sets therefore is mostly designed to fit in main memory and is limited by the heap size of the JVM. Even though RapidMiner already can do large scale data processing by means of connecting to databases and reading large data tables in batches, it lacks on doing this in a parallelized and distributed manner. Combining the principles of distributed computing with large scale data access for example by means of databases involves great potentials for RapidMiner. A possible step in this direction has been taken in this thesis by using a distributed cache for spreading the data on multiple machines, while holding it in-memory. Further developments could extend the data handling of RapidMiner to support transparent representation of distributed in-memory datasets within RapidMiner processes. Additionally, such an representation could make use of underlying databases within a distributed environment.

[1]http://www.terracotta.org/
[2]http://www.globus.org/toolkit/
[3]http://www.jboss.org/jbosscache
[4]http://www.hazelcast.com

Appendix A

k-Means Clustering

The k-means clustering problem is to determine k subsets ($S = S_1, ..., S_k$) in a set of n observations $x_1, ..., x_n$, in which all observations belong to the subset S_i with the nearest mean μ_i. Additionally, the subsets must be chosen in order to minimize the within-cluster sum of squares.

$$\underset{S}{argmin} \sum_{i=1}^{k} \sum_{x_j \in S} \|x_j - \mu_i\|^2 \tag{A.1}$$

Since the general k-Means problem is NP-hard [1] there is no algorithm known which always finds the optimal solution in appropriate time with respect to practicability. Therefore usually a heuristic algorithm is used. This standard k-Means clustering algorithm was first proposed by Lloyd [29] and consists of the following steps:

1. Pick k cluster means $\mu_1, ..., \mu_k$ randomly.
2. Assign each observation x_i to the subset S_j with nearest mean μ_j.
3. Recalculate all cluster means μ_j for all subsets S_j.
4. Go to step (2) as long as at least one x_i changes its assignment in step (2) or as long as there has not been a maximum number of iterations.

Since the algorithm is heuristic, it is not guaranteed to find the optimal solution according to Equation A.1. Thus, it is common practice to have several runs of the algorithm with different starting means and choose the result with minimal within-cluster sum of squares. Even though the solution may not be optimal, it is often sufficient in practice.

Bibliography

[1] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, 2009.

[2] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning).* The MIT Press, 2004.

[3] Apache Hadoop. available from `http://hadoop.apache.org` (retrieved: Dec'09).

[4] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, 2008.

[5] F. Berman, G. Fox, and A. Hey. *Grid computing: making the global infrastructure a reality.* John Wiley & Sons Inc, 2003.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] T. Breuel. PaREn - Pattern Recognition Engineering, June 2007. Project Proposal.

[8] J. Bulpin and I. Pratt. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD04)*, 2004.

[9] S. Chen and S. Schlosser. Map-Reduce Meets Wider Varieties of Applications. Technical report, IRP-TR-08-05, Technical Report, Intel Research Pittsburgh, 2008.

[10] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, page 281. The MIT Press, 2007.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107, 2008.

[12] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing. United States Patent 7,650,331, Jan. 2010.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition).* Wiley-Interscience, November 2000.

[14] G. Dynamics. Partners. available from `http://www.griddynamics.com/partners/gridgain.html` (retrieved: March'10).

[15] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284. IEEE Computer Society Washington, DC, USA, 2008.

[16] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[18] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):43, 2003.

[19] D. Gillick, A. Faria, and J. DeNero. MapReduce: Distributed Computing for Machine Learning, 2006.

[20] GridGain. available from `http://www.gridgain.com` (retrieved: Dec'09).

[21] A. Hadoop. PoweredBy. available from `http://wiki.apache.org/hadoop/PoweredBy` (retrieved: March'10), March 2010.

[22] N. Ivanov. GridGain 1.5 - Open Source Grid Computing For Java. available from `http://www.theserverside.com/news/thread.tss?thread_id=46568` (retrieved: April'10), August 2007.

[23] I. Khan. Distributed Caching On The Path To Scalability. available from `http://msdn.microsoft.com/en-us/magazine/dd942840.aspx` (retrieved: March'10), July 2009.

[24] A. Kimball. Cloudera Hadoop Training: MapReduce and HDFS. available from `http://vimeo.com/3584536` (retrieved: March'10), 2009.

[25] G. Kovoor, J. Singer, and M. Luján. Building a Java MapReduce Framework for Multi-core Architectures. 2010.

[26] R. Lämmel. Google's MapReduce programming model - Revisited. *Science of Computer Programming*, 68(3):208–237, 2007.

[27] D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. *Lecture Notes in Computer Science*, 1398:4–18, 1998.

[28] T. Liu, C. Rosenberg, and H. Rowley. Clustering billions of images with large scale nearest neighbor search. In *IEEE Workshop on Applications of Computer Vision, 2007. WACV'07*, pages 28–28, 2007.

[29] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[30] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[31] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: rapid prototyping for complex data mining tasks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940, New York, NY, USA, 2006. ACM.

[32] I. S. Network. Granularity and Parallel Performance. available from `http://software.intel.com/en-us/articles/granularity-and-parallel-performance/` (retrieved: May'10), February 2010.

[33] Y. D. Network. Yahoo! Launches World's Largest Hadoop Production Application. available from `http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html` (retrieved: March'10), February 2008.

[34] Oracle Coherence. available from `http://www.oracle.com/technology/products/coherence/index.html` (retrieved: Jan'10).

[35] V. Pankratius, C. Schaefer, A. Jannesari, and W. Tichy. Software engineering for multicore systems: an experience report. In *Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60. ACM, 2008.

[36] P. Quelhas, F. Monay, J. Odobez, D. Gatica-Perez, and T. Tuytelaars. A thousand words in a scene. *IEEE transactions on pattern analysis and machine intelligence*, 29(9):1575, 2007.

[37] Rapid-I RapidMiner. available from `http://rapid-i.com` (retrieved: December'09).

[38] D. Setrakyan. Compute Grids vs. Data Grids. available from `http://gridgain.blogspot.com/2008/07/compute-grids-vs-data-grids.html` (retrieved: February'10), July 2008.

[39] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision*, page 1470, Washington, DC, USA, 2003. IEEE Computer Society.

[40] A. W. M. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain. Content-based image retrieval at the end of the early years. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(12):1349–1380, 2000.

[41] C. G. M. Snoek and M. Worring. Concept-based video retrieval. *Found. Trends Inf. Retr.*, 2(4):215–322, 2009.

[42] G. Systems. GridAffinityLoadBalancingSpi. available from `http://www.gridgainsystems.com/wiki/display/GG15UG/GridAffinityLoadBalancingSpi` (retrieved: April'10), 2008.

[43] A. Tanenbaum and M. Van Steen. *Distributed systems - Principles and Paradigms.* 2002.

[44] T. Tuytelaars and K. Mikolajczyk. Local invariant feature detectors: a survey. *Found. Trends. Comput. Graph. Vis.*, 3(3):177–280, 2008.

[45] T. White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 2009.

[46] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. In *Proceedings of the 25th international conference on Machine learning*, pages 1184–1191. ACM, 2008.

[47] J. Yang, Y. Jiang, A. Hauptmann, and C. Ngo. Evaluating bag-of-visual-words representations in scene classification. In *Proceedings of the international workshop on Workshop on multimedia information retrieval*, page 206. ACM, 2007.