# Distributed Pattern Recognition in RapidMiner

Alexander Arimond, Christian Kofler, and Faisal Shafait
Multimedia Analysis and Data Mining Competence Center
German Research Center for Artificial Intelligence (DFKI GmbH)
D-67663 Kaiserslautern, Germany
a_arimon@cs.uni-kl.de, christian.kofler@dfki.de, faisal.shafait@dfki.de

***Abstract*— 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. In this paper, we present a system called *DisPaRe*, which integrates distributed computing frameworks into RapidMiner. A special focus is put on utilizing MapReduce as a programming model. The frameworks GridGain and Oracle Coherence are reviewed and evaluated with respect to their suitability to fit into the context of RapidMiner. The system provides effective means for transparently utilizing these frameworks and enabling RapidMiner processes to parallelize their computations within a distributed environment.**

## I. Introduction

Today, Pattern Recognition and Machine Learning techniques are found in many applications like face and object recognition in videos or images, speech recognition as given in mobile phones, or optical character recognition (OCR) for the automatic digitization of scanned documents. 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 are highly data-driven, i.e. their performance strongly depends on the problems and domains they are applied to.

In this context, the Pattern Recognition Engineering (PaREn) project was initialized. PaREn 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 [1] - 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* [2]. 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. Unfortunately, RapidMiner lacks support for distributed computing capabilities. In opposite to this, there exist a variety of distributed computing software like *GridGain* [3] and *Oracle Coherence* [4], 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 knowledge about the frameworks and experience with distributed systems in general.

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.

In the following section, the architecture and functionality of *DisPaRe* is presented. It is shown how this system can be seemlessly embedded into RapidMiner and how it enables different components of RapidMiner to make use of multicore capabilities and multiple machines. An emphasis of the system's development lies on the utilization of MapReduce as a programming model. After this, the applicability of the

system is shown by building two different use cases upon it, arising from the field of Pattern Recognition - interest point extraction from images and k-means clustering. Evaluation results are presented which prove that *DisPaRe* is capable to reach significant performance gains in terms of computation time.

## II. System Design

### A. Architectural Components

*DisPaRe* consists of three major abstract components as illustrated in Figure 1, namely *Distribution*, *MapReduceSpecification* and *DataLayer*. From a developers point of view, *DisPaRe* totally abstracts from the frameworks which are used to perform the distributed computations. In principle, *DisPaRe's* modularity allows to implement the system by using differnet frameworks, from a data distribution view as well as when considering the computational aspects.

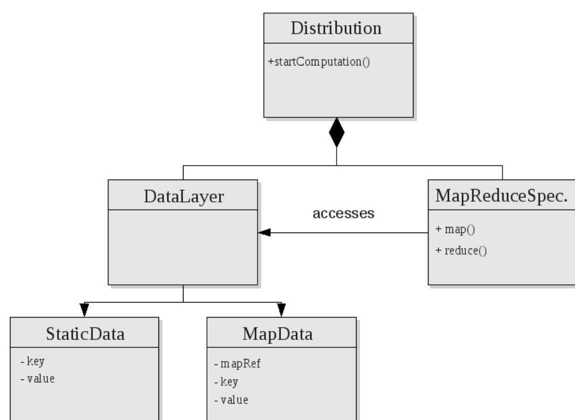


Fig. 1. Relationship between the components of the developed system.

The **Distribution** interface is the main interaction point for RapidMiner to perform the distributed computation and to get output from it. Furthermore, it 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**. This interface defines the logic of a computation in form of a *map* and a *reduce* function. MapReduce is a programming model and a software framework which has been introduced in [1] and patented by Google [5]. 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 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. Even though it is flawed in some places [6], 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. 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, e.g. k-Means, logistic regression, neural networks and SVMs [7], [8].

The map and reduce functions in *DisPaRe* 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. The only difference is 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[1]. It therefore is less complex, which fosters modularity for integrating other distributed computing software.

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. The distinction of those two types of data is not made by "classical" MapReduce, but is meaningful when considering how data can be distributed across processing nodes: Individual mapping data must be stored (at least) once, ideally at the location where the map computation takes place in order to improve access-time consumption. 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.

MapReduce can be applied to many algorithms and techniques (and also to other applications) in a straightforward manner, and by this it is possible to transparently gain computation perfomance for these techniques within a distributed environment. The components of the developed system, especially the MapReduceSpecification interface, provide this MapReduce functionality to RapidMiner, PaREn and other Machine Learning developers in a meaningful and easy to use way.

### B. 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. The use cases in the performance evaluation section

[1] In Google's programming model also the reduce step is abstracted in a way that there may be many reduce tasks which run in parallel.

provide examples on how to split data: a list of images can be split into single images, a set of vectors (ExampleSet) can be split into single vectors or subsets of vectors.

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. Data units, which group together as input for an individual map task, are all referencing this map task and therefore can be handled together by the system, so that each map task receives exactly the input data units which correspond to it. 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 the MapReduce computation.

### *C. Integration with RapidMiner*

The *DisPaRe* 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.

Within RapidMiner, the type of realization can be chosen by means of a Factory pattern [9]. The class **DistributionFactory** takes care of the proper instantiation of imlementations of the Distribution component, in case of this paper with GridGain and Coherence. By this design, decoupling of RapidMiner, *DisPaRe*, 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 *DisPaRe*. 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 *DisPaRe*. 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 *DisPaRe*.

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 2 illustrates how the AbstractDistributionOperator fits into the context of RapidMiner.

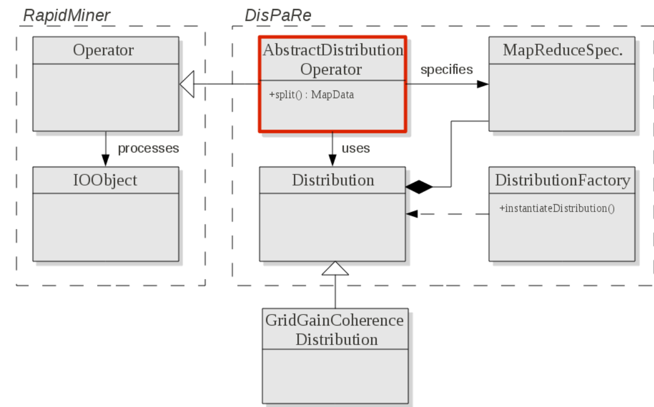


Fig. 2. Integration of the developed system within RapidMiner by means of the *AbstractDistributionOperator*. The *DistributionFactory* instantiates concrete implementations of the *Distribution* interface, in this case with GridGain and Coherence. The whole functionality can be transparently used as Operator within RapidMiner processes.

### *D. Realization with GridGain and Coherence*

The components of *DisPaRe* can be realized by using external frameworks: on the one hand GridGain for controlling computations within the MapReduceSpecification component, on the other hand Oracle Coherence for the DataLayer.

GridGain [3] is an open source Grid Computing framework, which focuses on easy development and deployment of Grid Computing and Cloud application software. It is purely written in Java, and is a software middleware that allows to develop complex grid applications on the cloud infrastructure [10]. In the context of Grid Computing, it can be seen as so called 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 supports MapReduce like computations similar to the one proposed by Google, but less complex.

*Coherence* is a commercial distributed memory data management solution [4]. 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. It provides several different cache topologies, which are highly configurable.

The global architecture of the realization of *DisPaRe* with GridGain and Coherence can be seen in Figure 3. 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 task. It therefore fits the needs of the MapReduce model of *DisPaRe*. The maps must be aligned with the locality of data within the Coherence cache. GridGain provides functionality which allows to realize this ”data affinity” by aligning jobs with the data given in a cache. This is an important performance aspect, because computations should always be located to corresponding data, not vice versa.

The DataLayer component specifies two types of input data, which can be stored in two ways in Coherence. Individual mapping data can be stored within a “partitioned cache“. By using this type of cache, the input for a single map task can only be stored once[2] in the cluster, i.e. on the machine where the map computation takes place. Fast access is guaranteed since data is kept in memory by the cache. 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.

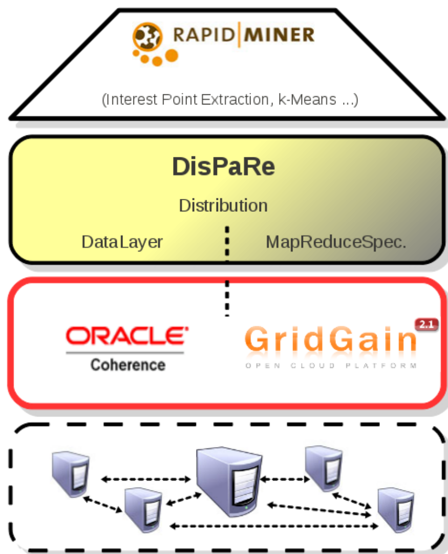


Fig. 3. Realization of *DisPaRe* with GridGain and Oracle Coherence.

GridGain ships with integration of Coherence out of the box. That means each processing node appears as a conjunction of GridGain and Coherence. Ideally, it is possible to just start a script on a different machine to initiate a new processing node. This node would join the cluster automatically without doing any configuration[3]. A RapidMiner user can do this without knowing any concepts of GridGain or Coherence.

Furthermore, by using GridGain and Coherence, many important aspects of distributed computing are already covered. Requirements like proper load balancing, fault-tolerance, automatic deployment of new software, data replication and backups, serialization and monitoring are very difficult to realize by hand. GridGain and Coherence provide sophisticated mechanisms to fulfill these requirements. RapidMiner users and developers therefore can concentrate on parallelizing and performing their tasks instead of thinking about distributed computing issues.

## III. Performance Evaluation

The performance of *DisPaRe* has been evaluated by measuring computation times for two different tasks which are typical for Pattern Recognition and Machine Learning. The first 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. The interest point detector and descriptor used for extraction are both SURF [11]. The total number of extracted interest points for all images is 188176, each point given as field of 128 double values, making the whole result dataset 184 MB large. 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. Parallelizing an interest point extraction task is easily possible within MapReduce by performing the extraction as a map task on each image (or a subset of images) and collecting the interest points in the reduce step.

The second task under consideration is k-Means clustering according to Lloyd [12]. In this setting, the output of the interest point extraction serves as input to the clustering, i.e. 188176 samples, each with 128 dimensions. K-Means parallelization also is straightforward within MapReduce: extract the relevant information within the map step in parallel for subsets of the data and compute the new means within the reduce step. This computation can be done in every optimization iteration of the algorithm. 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, and therefore is amortized by speedup after few iterations. Initialization time in the presented setting is about 2-3 minutes, depending on the number of machines in the cluster. On a single machine with one thread, it takes about 80 seconds for k=100 and 170 seconds for k=200 to perform one optimization iteration. For each experiment, at least ten optimization iterations have been measured, outliers have been removed and the results have been averaged.

First, the experiments have been conducted on a single machine, while varying the number of working threads for execution. Second, the number of machines has been varyied while staying with two threads per machine. The machines, each equipped with Intel Atom CPU 330 1.60GHz Dual Core with Hyperthreading, 512KB Cache and 3 GB RAM, are connected over a 1GBit full duplex intra-network. As the machines are all equally equipped, they build up a very homogenous environment. Furthermore, the used hardware can be seen as commodity hardware, which for example can be found in offices.

### A. Performance on a Single Machine

In the first experiment, the performance of the realization with GridGain and Coherence has been investigated on a single machine. This is done by setting up a cluster which exactly contains one machine. GridGain allows to control

[2]This not considers backups.

[3]Network and environment (e.g. firewall settings) must support this.

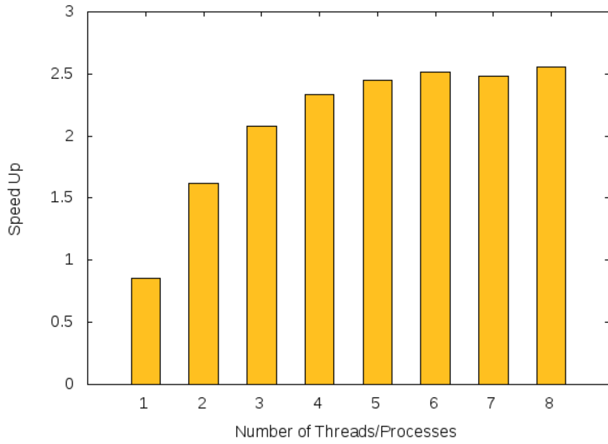


Fig. 4. Performance of interest point extraction, parallelized on a single machine: speedup is limited by the number cores.

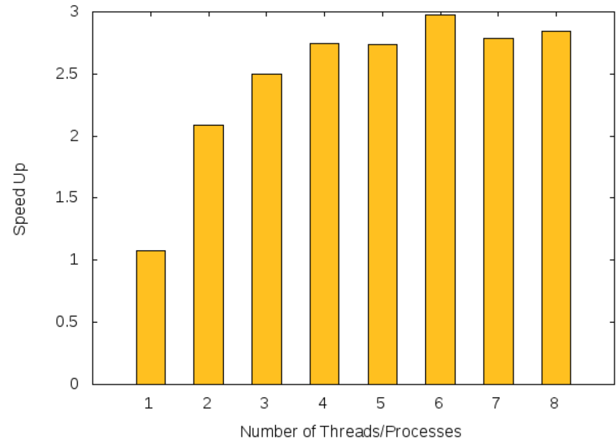


Fig. 5. Performance of k-Means, parallelized on a single machine.

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. The number of images per job has been fixed to 50.

It can be seen that computation performance increases when increasing the number of threads. Considering enough threads, the system proves that it can perform better than the standalone, sequentially working implementation without distribution. On dual core with Hyperthreading, a speedup of about 2.5 can be achieved. Having more physical cores may even lead to better performances.

The second experiment investigates the k-Means algorithm. The experiment has been done with k = 100. 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 5.

First of all, it can be seen that the system performs similarly well as in the first experiment. The minor performance benefit over the standalone version for one thread is due to a slightly different implementation for the distributed version.

### *B. 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 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. Figure 6 shows the performance gain 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.

As can be seen 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 [13]. If the 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.

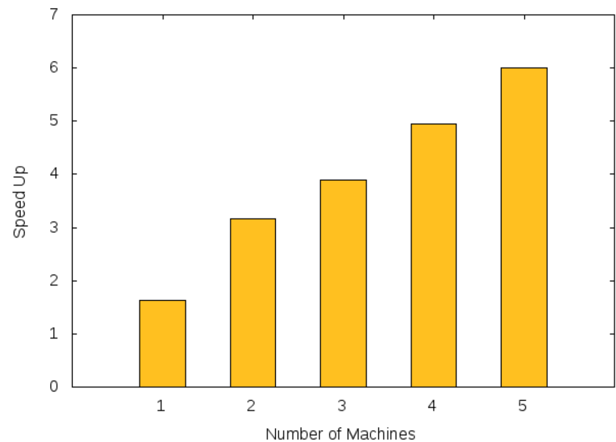


Fig. 6. Performance on multiple machines: for interest point extraction, speedup increases proportionally with the total number of threads.

In the last experiment, k-Means clustering has been distributed on multiple machines (Figure 7). Again, each machine runs with two threads, the parameter k has been set to 200, the number of tasks is 20 in this setting. The system

reaches a speedup of 7.5 with respect to 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.

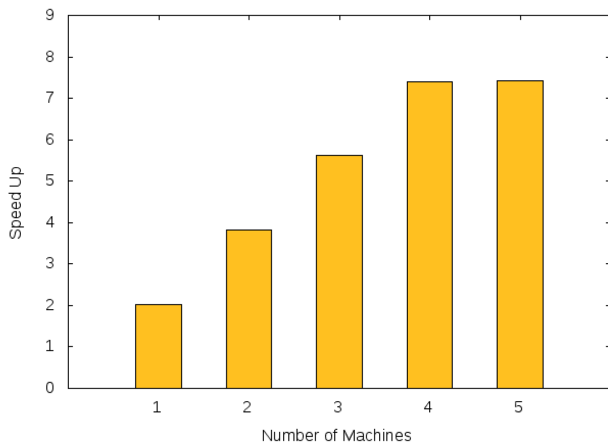


Fig. 7. Performance on multiple machines: using GridGain/Coherence, k-Means achieves nearly linear speedup with the total number of threads. However, load imbalance stops this increase for five machines in this setting.

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 5 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.

## IV. Conclusion

The main goal of *DisPaRe* has been the integration of distributed computing frameworks with the Machine Learning software RapidMiner. The software frameworks GridGain and Oracle Coherence have been reviewed and integrated successfully into RapidMiner. *DisPaRe* 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 the field of Pattern Recognition: interest point extraction from images 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 *DisPaRe*, it is possible to significantly accelerate these processes. Using GridGain and Coherence as distributed computing framework, the system is able to achieve nearly linear speedup with the number of threads and machines as seen for the k-Means clustering.

## References


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

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

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

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

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

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

[7] C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. 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.

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

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

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

[11] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, 2008.

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

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