Exploiting idle cycles to execute data mining applications on clusters of PCs

Hermes Senger a,*, Eduardo R. Hruschka a, Fabrício A.B. Silva a, Liria M. Sato b, Calebe P. Bianchini b, Bruno F. Jerosch a

a Universidade Católica de Santos (UniSantos), R. Dr. Carvalho de Mendonça 144, 11070-906 Santos, SP, Brazil
b Escola Politécnica, Universidade de São Paulo Av. Prof. Luciano Gualberto, trav. 3, n. 180, 05508-900 São Paulo, SP, Brazil

Received 23 June 2005; received in revised form 26 May 2006; accepted 28 May 2006
Available online 20 July 2006

Abstract

In this paper we present and evaluate Inhambu, a distributed object-oriented system that supports the execution of data mining applications on clusters of PCs and workstations. This system provides a resource management layer, built on the top of Java/RMI, that supports the execution of the data mining tool called Weka. We evaluate the performance of Inhambu by means of several experiments in homogeneous, heterogeneous and non-dedicated clusters. The obtained results are compared with those achieved by a similar system named Weka-Parallel. Inhambu outperforms its counterpart for coarse grain applications, mainly for heterogeneous and non-dedicated clusters. Also, our system provides additional advantages such as application checkpointing, support for dynamic aggregation of hosts to the cluster, automatic restarting of failed tasks, and a more effective usage of the cluster. Therefore, Inhambu is a promising tool for efficiently executing real-world data mining applications. The software is delivered at the project’s web site available at http://incubadora.fapesp.br/projects/inhambu/.

© 2006 Elsevier Inc. All rights reserved.

Keywords: Parallel and distributed data mining; Commodity clusters; Load sharing

1. Introduction

As proposed by Fayyad et al. (1996), Knowledge Discovery in Databases (KDD) is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data. Although the terms KDD and Data Mining (DM) are sometimes employed interchangeably, DM is often considered as a step in the KDD process that centers on the automated discovery of patterns and relationships in data (Hand et al., 2001; Witten and Frank, 2000). Under this perspective, our work is primarily focused on DM, which usually involves computationally expensive tasks. For this reason, several techniques have been proposed to improve the performance of DM algorithms, such as parallel processing (Freitas and Lavington, 1998), implementations based on cluster of workstations (Baraglia et al., 2000), and computational grids (Canataro and Talia, 2003). These techniques can leverage the deployment of DM applications to production scales.

The potential benefits of applying parallel processing to DM applications are not limited to reducing execution times, i.e. qualitative improvements can also be achieved. As DM methods often are based on machine learning techniques (Witten and Frank, 2000), computational power may be potentially turned into accuracy gains, for instance by means of executing approximation based algorithms with a greater number of steps, training algorithms in larger datasets, and executing more experiments when non-deterministic algorithms are employed.
During the last years, computer clusters have become increasingly used for high performance computing. Also, assembling computer clusters composed of commodity PCs and workstations is an easy way to provide computational power at low cost, since companies and universities often have hundreds or thousands of commodity PCs. In such environments, individual computers often present low levels of utilization, offering great amounts of computational power that can be used to perform intensive computations, such those required by DM algorithms.

While clusters comprised by commodity PCs can be used to carry out the execution of large applications, some challenges may arise due to their heterogeneous and non-dedicated nature. For scalability and protection of investment, companies and universities often add new computers to the existing pool without discarding the existing ones. This practice usually leads to heterogeneity. Also, non-dedicated environments often suffer from fluctuating levels of performance, failures, and unavailability of resources. All of these may render it difficult to support the execution of DM applications on such a computing infrastructure. From this standpoint, the development of tools capable of supporting the efficient usage of the existing resources for DM is important. Currently, DM practitioners and researchers usually lack from tools that support the execution of DM applications on commodity clusters which encompass the following requirements:

- **Implement a good collection of algorithms.** Researchers and practitioners usually execute several DM algorithms in order to assess their performance on particular applications.
- **Ease of use.** DM tools should be easy to install and easy to use, and DM users should not be concerned with operating systems and networking configuration. Ideally, new tools should employ known interfaces and interaction patterns.
- **Resource management.** In order to extend the benefits of high performance computing to DM users, DM tools should provide support for dynamic discovery, allocation and management of existing resources in clusters of PCs and/or workstations.
- **Heterogeneity.** Commodity clusters may be composed of heterogeneous computers with different capacities and resources which should be taken into account for the purpose of scheduling DM tasks.
- **Fault tolerance.** The execution of computing intensive tasks which may take several hours (or a few days) and involve several machines requires DM tools to survive from failures. After crashing, DM tools should be able to resume their execution.

Inhambu is a system that leverages the exploitation of idle resources in a cluster composed of commodity PCs, supporting the execution of DM applications based on the Weka (Witten and Frank, 2000) tool. A brief introduction to Inhambu has been previously presented in Senger et al. (2004). In our current work, we further elaborate on the project and implementation of Inhambu, proposing policies for scheduling, load sharing, heterogeneity, overloading avoidance, and fault tolerance. In addition, we compare its performance with a similar system named Weka-Parallel (Celis and Musicant, 2002). We benefit from our previous experience on designing load sharing policies and scheduling tasks of a generic application on heterogeneous, non-dedicated clusters (Senger and Sato, 2003), showing how a DM framework (Weka) focused on classification algorithms can be adapted for harvesting the computer power of commodity clusters. In brief, classification (Duda et al., 2001) involves the assignment of instances\(^1\) of a dataset to one of a finite number of categories (classes). Algorithms designed for classification are used in several applications such as in bioinformatics, business, text mining, and web mining.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of related work. In Section 3, the architecture of Inhambu, as well as its main components, functionalities, and policies implemented are described. In Section 4, we detail how the Weka tool is adapted to be executed with the support of Inhambu. Afterwards, we empirically evaluate the performance of Inhambu in several experiments, which demanded more than 30 thousand hours of CPU, comparing it against Weka-Parallel. The obtained results are reported in Section 5. Finally, Section 6 concludes the paper.

2. Related work

The last decade has witnessed a considerable amount of research publications on Parallel and Distributed Knowledge Discovery. For instance, more than three hundred bibliographical references are given by Liu and Kargupta. Most of them address the parallelization of particular DM algorithms or specific classes of algorithms. Our work differs from such approaches, since we focus on the parallel execution of Weka, which is a system that implements a wide range of classification algorithms, on non-dedicated clusters comprised by commodity PCs and workstations.

Weka (Witten and Frank, 2000) is currently one of the most popular DM tools. Originally proposed for didactic purposes, Weka is a framework for the implementation and deployment of DM methods. It is an open source software developed in Java, released under the GNU General Public License (GPL), being currently available to Windows, MAC OS and Linux platforms. Weka provides implementations of several algorithms for DM, and it is continuously incorporating more and more algorithms by means of the contribution of researchers around the world.

Weka-Parallel (Celis and Musicant, 2002) extends the original implementation of Weka for supporting the

---

\(^1\) The terms instance, tuple, example and object are commonly employed by the DM community.
execution of the cross-validation procedure (Witten and Frank, 2000) on a cluster composed of commodity PCs and workstations. Cross-validation is the standard method for evaluating classification algorithms, which are predominant in Weka (Witten and Frank, 2000). In a nutshell, the \( n \)-fold cross-validation process divides a given dataset into \( n \) equal sized parts (named folds). Then, the classification algorithm is trained in \((n-1)\) parts, and tested in the remaining one. This procedure is repeated \( n \) times, by varying the set of folders used for training and for testing a given classification algorithm. Finally, the overall classification accuracy, which is the average accuracy obtained from the \( n \) test sets is provided (see Section 4 for further details). Because cross-validation generates a (potentially larger) number of independent subtasks, Weka-Parallel distributes those subtasks for executing on remote servers.

Although shortening the execution times for cross-validation, Weka-Parallel suffers from several limitations. For instance, it does not provide support for dynamic resource management, neither support for the heterogeneity of the networked computers which may present different capacities. Instead, computers are allocated for executing tasks regardless of the fact that they may be heterogeneous and may be currently executing different workloads. As a consequence, it may lead to low performance to DM users, as well as to make other users sharing the cluster machines to perceive performance degradation. In addition, Weka-Parallel does not provide support to fault tolerance and dynamic aggregation of computers, whose number and availability may vary along the application’s execution time. By taking into account the project requirements mentioned in Section 1, the system to be described in the following provides a more effective sharing of resources, while avoiding that other users sharing the cluster can perceive a reduction of performance due to resource sharing. Differently to Weka-Parallel (which is implemented on the top of socket application programming interface), Inhambu is based on the Java/RMI, which allows to minimize the effort for coding the involved communication. Due to this approach, we can minimize the programming effort to evolve Inhambu, extending its benefits to future versions of Weka.

3. The architecture of Inhambu

This section presents an overview of the Inhambu’s architecture, as well as its main components and strategies. As depicted in Fig. 1, the architecture of Inhambu implements an application layer which consists in a modified implementation of Weka, and a resource management layer which provides the necessary functionalities for the execution of Weka in a distributed environment. In the application layer, specific components are implemented and deployed at the client and server sides. The client component executes the user interface and generates DM tasks, while the server contains the core Weka classes which execute the DM tasks. The trader provides publishing and discovery mechanisms for clients and servers.

3.1. The application layer

The application layer is a modified implementation of Weka with some minor changes. In brief, Inhambu divides the original Weka classes into client and server components replacing the method `crossValidateModel()` of the class `Evaluation` with a parallel implementation of the cross-validation procedure. Such a few modifications allow tasks which comprise the cross-validation procedure to be executed on idle processors in the cluster. Also, such tasks can be executed in parallel as the client implements a multithreaded version of the cross-validation. Therefore, the application layer implementation is able to deliver as many concurrent invocations as the current number of idle processors in the cluster. By concentrating such modifications in a specific class, the programming effort necessary to port Inhambu to future versions of Weka can be reduced. The parallelization of the cross-validation was previously introduced in Senger et al. (2004). In this paper we present a more consistent discussion on its architecture and implementation.

There are two important design aspects regarding the architecture of Inhambu. First, the resource management layer is not specific for data mining tasks. Instead, it provides support for the management of idle cycles in a cluster of PCs, as well as scheduling for coarse granularity
applications composed of independent tasks. Note that our current work is focused on the execution of a class of data mining applications, but other application domains can also take benefit from our approach as long as they present similar characteristics (i.e. independent tasks and coarse granularity). The strategies that guided the design of the resource management layer were detailed in Senger and Sato (2003). Moreover, the benefits of parallelizing the cross-validation process (Witten and Frank, 2000) is not limited to its execution per se. Other classes implemented in Weka, such as the meta-classifiers and feature selection methods, also require the execution of one or more instances of the cross-validation process. Therefore, our implementation can also be useful for such classes. Section 4 provides further details on how the cross-validation was modified.

3.2. The resource management layer

In the resource management layer, Inhambu implements a trading service that supports resource management facilities. The trader enables server programs to publish their services, by invoking its exportService operation and passing service names and remote object references. Such information remains stored in the trader, which can be queried by client programs by means of its importService operation. Clients must import references to remote objects that implement the application services before they can invoke them. Clients can be either application programs, or the Weka’s user interface.

The resource management functionalities are supported by the components depicted in Fig. 2. In each server machine, a load agent gathers static and dynamic information on system resources (e.g. number of processors, amount of memory, swap space, and processor usage). Such information is sent to one or more load managers. The trader provides mechanisms for the advertisement and discovery of application services. Both the trader and load manager are placed at the same node, so that discovery operations executed by the trader can issue local queries on resource availability.

Application tasks may often present specific demands for resources, such as the minimum amount of memory, CPU speed, and network bandwidth they require. A possible approach for obtaining the application’s requirements consists in letting the user to specify them explicitly when submitting applications, for instance, by means of script languages (Czajkowski et al., 1998) or service level agreements (SLA) (Czajkowski et al., 2002). We believe this may be a suitable approach for the design of general purpose systems such as application schedulers and resource managers, but it is not the case for Inhambu. In case of general purpose systems, middleware designers are not likely to have any prior knowledge about specific characteristics of the application, which are usually known by application designers. In contrast, the Inhambu client component is able to estimate the amount of memory required to execute the classifiers implemented in Weka at runtime, and pass such information to the trader when invoking its importService operation. As a consequence, DM users do not need to be aware of such details.

3.3. Scheduling and load sharing

For scheduling DM tasks, we implemented a strategy based on both static and dynamic performance indices (Casavant and Kuhl, 1988; Dinda and O’Halloran, 1998; Ferrari and Zhou, 1987). Static performance indices are usually implemented by static values that express or quantify amounts of resources and capacities. Examples of static...
metrics include the processor speed (e.g. MFLOPS, MIPS), the total amount of memory (in bytes), the cache memory size (in bytes) and nominal transmission rate of network interface. Dynamic indices are often implemented by values that express the utilization of resources within a period of time. Examples of dynamic indices typically include the occupation index of the processor during the last minute, the current amount of free memory in the system, the current number of processes in the execution queue and the current status of a processing node (e.g. BUSY, IDLE).

Every machine of Inhambu executes a load agent which gathers some static information, such as the total amount of memory, the transmission rate of network interface, the number of CPUs, and the estimated processor speed. In order to measure the capacity of each machine $M_i$, the load agent executes a small synthetic benchmark during a few seconds. The result is then expressed as the static capacity index $\text{SC}_i$ which reflects the maximum capacity of the machine $M_i$ when no other application is running. All of these information are gathered only once, and sent to the load manager. Having sent initial information to the load manager, the load agent enters a dynamic monitoring phase, in which they periodically collect and transmit resources utilization indices to the load manager at regular intervals of one second. Dynamic monitoring is important to manage workloads fluctuations and improve the system performance of non-dedicated clusters of computers. An important index for this purpose is $q_i$, which measures the CPU utilization during the last observation period. Once the load manager knows the maximum capacity $\text{SC}_i$ and the current utilization $q_i$ of each machine $M_i$, it is able to choose which of them can best execute DM tasks. It is worth noting that $q_i$ is a simple, widely adopted (Ferrari and Zhou, 1987; Devarakonda and Iyer, 1989; Dinda and O’Halloran, 1998) load estimator that can be computed as the average utilization rate from the recent past. According to Ferrari and Zhou (1987), mean values can better serve as load indexes to reflect resource utilization than instant values, which may not be representative and lead the system to unstable behaviors. In case of using multiprocessors, $q_i$ measures the average utilization index for all processors.

According to Schneekenburger and Stellner (1997), a location policy is responsible to find a server that can execute the application tasks. Our location policy is executed when an importService request is processed. The importService requests are received by the trader that queries the load manager on “which server provides this service and offers the largest amount of idle processing capacity”. The load manager computes the estimated dynamic capacity index $\text{DC}_i$ for each machine $M_i$ in the cluster as

$$\text{DC}_i = \text{SC}_i(1 - q_i).$$

Then, the load manager selects the machine that: (i) is not executing a data mining task, and (ii) is associated to the maximum value of Eq. (1) as being the best server.

Although load indices (computed by averaging the processor usage during some period of time) have the ability of smoothen load variations and avoid instability, they can also cause the system to perceive workload variations only when it is too late. This phenomenon may lead to incorrect scheduling decisions in some specific situations. For instance, when some lightly loaded computer is chosen by the location policy to receive a new load entity, its real workload tends to increase swiftly, while measured load indexes can only reflect this change after some period. Due to this delay, the chosen computer could be erroneously assigned more load entities, thus entering an overloading state. A possible strategy to circumvent this problem involves using migration of tasks (e.g., see Brunner and Kalé, 1999). In this work, the location policy retains an information on the eligibility of every machine in order to prevent a single computer from being assigned an excessive number of requests. Right after choosing a given machine for receiving a request, the location policy marks such computer as ineligible. Only after finishing the processing of its current request, and being marked as eligible again, such computer could receive another DM task. In summary, when the Inhambu’s location policy has to decide on which computer will be assigned to execute a new DM task, it looks for “the computer that offers the largest available capacity and that is not running another DM task”. In case of multiprocessors, the location policy maintains information regarding to the eligibility of each individual processor. The scheduling and load balancing policies adopted for the implementation of Inhambu have been previously presented in Senger and Sato (2003).

### 3.4. Overloading and contention avoidance

Although Inhambu does not provide guarantees concerning the quality of services, it does implement a “best-effort” policy aimed at preventing local applications and users to perceive a severe lost of performance. To avoid overloading, a computer can only be chosen to receive load entities if its load index is below a given threshold. Such criterion is important for two reasons. Firstly, it prevents other users and local applications that are using the computers from perceiving a significant lost of performance. Second, it prevents DM tasks from executing in busy machines, which could lead to poor performance. The choice of a suitable value for the threshold is an important project decision, which can be supported by results obtained from the literature. As shown in Jain (1991), there is a relation between the utilization index and the response time of a computer system. Such relation shows that the response time increases very rapidly when the utilization index of a computer system goes beyond 0.8. Furthermore, response times can increase exponentially when utilization goes beyond 0.9. Thus, in this work the default value chosen for the threshold is 0.7.

In summary, as the Inhambu’s goal is to take advantage of idle cycles, DM applications should wait while cluster
machines are above this threshold. However, if a local application is launched after some DM task is already executing in the same machine, both of them are likely to experience a poor performance due to resources contention. To solve this problem, Inhambu always execute its tasks with lower priority, therefore benefiting the execution of local applications.

3.5. Heterogeneity

Commodity clusters may be comprised by heterogeneous computers. Heterogeneity is explicitly considered by Inhambu, since the index $SC_i$ (Eq. (1)) is computed by the load agents when they start to run in each cluster machine. Therefore, the index $DC_i$ (which is considered for selecting the best computer to receive a new request) also reflects heterogeneous capacities.

3.6. Fault tolerance

As Inhambu has been primarily designed for DM applications whose execution may require several hours (or days) and involve a large number of computers, failures are likely to become frequent. Thus, recovering from failures is an important requirement. Also, recovering strategies should be aware that failures can happen both at the client and server sides of Inhambu (see Fig. 2). Checkpointing and recovery can be addressed under different approaches. For instance, they can be implemented by instrumenting the application code (Lawall and Muller, 2000). Such approach may not be interesting when it involves a large amount of work to provide specific implementations of checkpointing mechanisms for each algorithm implemented in Weka. Also, under such approach the implementation work must be redone every time one is interested in updating Inhambu according to a new version of Weka. The second approach consists in modifying the run-time environment (such as the Java Virtual Machine) (Bouchenak et al., 2004; Napper et al., 2003). Such approach is challenging because of the lack of tools that support checkpointing and recovery. For instance, although Condor (The Condor Project Website) and Platform’s LSF (Lumb and Smith, 2003) support checkpointing capabilities, these are currently limited to sequential jobs, while checkpointing of parallel applications in heterogeneous systems is an ongoing research theme (Garbacki et al., 2005; Stone et al., 2004).

The Inhambu’s client is a central point of failure. If the client fails, all of the results produced by the already completed tasks could be lost. Therefore, implementing checkpointing and recovery mechanisms at the client side is essential. Although Weka implements several algorithms for classification, its design and implementation allow checkpoint and recovery mechanisms to be implemented at the client side in a simple and uniform way. All classification algorithms are implemented as subclasses of the Weka’s class named Classifier, which defines the general structure of any scheme for classification. Similarly, the Weka’s class named Evaluation implements the cross-validation procedure for any classifier which is an instance of Classifier. Thus, we implemented the Inhambu’s fault tolerance mechanisms by modifying the Evaluation class as follows. The cross-validation is partitioned into several independent tasks (as it will be described in Section 4) that are created by an instance of Evaluation. The result of every concluded task is stored in a local or remote disk by means of a network file system. If an execution fails, Inhambu is capable to recover the results of all completed tasks, and it restarts only the remaining ones. Also, by means of storing checkpoints right after the completion of each task, performance can be improved twofold. First, it reduces the number of checkpoints to one checkpoint per executed tasks. Second, it minimizes the size of a checkpoint to a few bytes. This is an important feature as the overhead produced by introducing checkpoint is proportional to its size (Elnozahy et al., 1992).

If the client crashes before finishing the execution of a DM application, Inhambu allows the application to be restarted, resuming its execution from the last checkpoint on. Implementing checkpointing only in the client side was a project decision as it arose a trade-off relationship involving fault-tolerance, performance, and programming effort. Implementing application level checkpointing at the server side may be a hard job, as it would involve specific solutions for each classification algorithm implemented in Weka. For the sake of illustration, by the time of this writing Weka has 68 classification algorithms, and new implementations are likely to be incorporated on future versions as well as the existing ones may be modified. Furthermore, checkpointing at the server side could produce an overhead proportional to the number of servers, since each server would periodically save its checkpoints in the cluster’s file system. For these reasons, we adopted a simpler solution to cope with failures. If a server fails, Inhambu just looks for another server to execute the same task and relaunches the same request.

4. Parallel and distributed cross-validation

By the time of this writing, the latest version of Weka is 3.4.4, which contains implementations of 68 classification algorithms, 5 algorithms for clustering, 3 algorithms for finding association rules, and 12 algorithms for attribute selection. In fact, it is continuously growing, incorporating more and more data mining algorithms. All these algorithms can be either applied directly to a dataset by means of a GUI, by means of a command line interface, or even called from Java programs. Classification algorithms, also called classifiers, are predominant in relation to the other algorithms implemented in Weka. Therefore, we decided to investigate potential benefits that Inhambu could bring to the classifiers package.

For DM purposes, the stratified cross-validation (Witten and Frank, 2000) is a widely adopted way of predicting the
accuracy of a classifier given a sample of data. In the tenfold cross-validation process, the dataset is divided into ten equal parts (folds). Then the classifier is trained in nine parts and tested in the remaining one. This procedure is repeated ten times, by varying the set of folders used for training and testing, and the estimated accuracy is the average in the test sets. Clearly, these experiments can be performed in parallel, taking advantage of Inhambu, so that all classifiers implemented in Weka can benefit from such approach. A generalized version of this method is the \( n \)-fold cross-validation (Witten and Frank, 2000). In this method, the cross-validation is executed \( n \) times, by using \( n - 1 \) folds for training (i.e. building a classifier) and the other one for testing (i.e. to assess accuracy). For instance, the \( n \)-fold cross-validation over a dataset with 45,000 instances (and assuming \( n = 45,000 \)) would produce 45,000 computational tasks, potentially demanding large amounts of computational power (which depends on the classifier to be employed). Clearly, these experiments are independent and can be performed in parallel by up to \( n \) computers of the cluster.

In this paper, we focus on the parallelization of the cross-validation by distributing different folds to execute in different nodes of the cluster. To accomplish this, we have modified some classes of Weka. The modifications concentrate into two classes, named Classifier and Evaluation. The former defines the general structure of any classification scheme, while the later implements the cross-validation procedure. All classification algorithms are implemented as subclasses of the Classifier. This class contains two methods, named buildClassifier() and classifyInstance(), which must be implemented in a specific way for each classification algorithm, so that the behavior associated to a new instance of this class can be defined according to how each algorithm builds a classifier and how it classifies data instances. Therefore, the Classifier creates a uniform interface for building and using all the classification algorithms. Thus, an evaluation module implemented by the class Evaluation can be used to evaluate the performance and accuracy of any classification algorithm in Weka. For instance, the MultiSchemme class implements a meta-classifier that also uses the same module for the cross-validation. MultiSchemme implements a benchmark for choosing the best classification algorithm from a list of algorithms selected by the user, for a given dataset. In this case, the number of tasks may raise even further. For example, suppose again that a dataset with 45,000 instances was submitted to a list of 20 classification algorithms. This experiment would produce 900,000 (i.e. 20 times 45,000) tasks, assuming \( n = 45,000 \). The evaluation module is implemented by the class Evaluation, whose method crossValidateModel() implements the stratified \( n \)-fold cross-validation procedure, as mentioned before. The number of folds in cross-validation can be supplied by the user as an execution parameter. If no other value to \( n \) is provided by the user, Weka assumes \( n = 10 \) by default.

As mentioned before, classifiers can be combined, aiming at higher classification accuracy than that of a single classifier. This issue has been studied under different names, such as combination of multiple classifiers, ensembles, collective mining, classifier fusion, committees of classifiers, bagging, boosting, stacking and collective recognition. Roughly speaking, such schemes can potentially benefit from the Inhambu’s support for resource management, as long as they can be modeled and implemented as sets of independent tasks. Some of these schemes are implemented in the Weka system, which allows the combination of multiple classification models by using the class known as “meta-classifiers”, which belong to the classifiers package. More interestingly, the assessment of multiple classifiers often makes use of the cross-validation procedure, thus benefiting from our implementation. This is precisely the case for the Weka’s MultiSchemme metaclassifier, which can run faster with the support of Inhambu.

5. Evaluating the performance of Inhambu

In this section, we evaluate both the performance and main functionalities of Inhambu, comparing it with the Weka-Parallel system (Celis and Musicant, 2002). Our experiments were carried out with more than 30 thousand hours of computing time equivalent to a 2 gigahertz IA-32 CPU.

5.1. Test environment

In order to carry out performance tests, we have run several experiments with two classifiers that are popular in the DM community: J4.8 and PART. J4.8 is the Weka’s implementation of the popular C4.5 decision tree learner (Witten and Frank, 2000). In fact, J4.8 is the later and slightly improved version, called C4.5 revision 8, which was the last public version of this family of algorithms before the commercial C5.0 was released. In brief, decision trees are constructed recursively. First, one attribute is selected to be placed at the root node. The selected attribute is the one that produces the purest daughter nodes (according to a numeric criterion, Witten and Frank, 2000). Then, the instances are separated according to the values of the selected attribute, originating branches of the decision tree. The process of selecting attributes and splitting instances is repeated for each branch. In this sense, any leaf with only one class will not have to be split further, and the recursive process down that branch will complete. The full complexity of decision tree induction is \( O(a \cdot N \cdot (\lg N)) \), where \( a \) is the number of attributes and \( N \) is the number of instances (Frank and Witten, 1998). The PART classifier provides rules from pruned partial decision trees (Witten and Frank, 2000), adopting the divide-and-conquer strategy. Basically, it builds a rule, removes the instances it covers, and keeps creating rules recursively for the remaining instances until none is left. For each rule, a pruned decision tree is built for the current set of instances. Then, the leaf with the larg-
Adults Census Income

The first dataset, known as "Adult", is a well-known benchmark for DM methods available at the UCI Machine Learning Repository (Merz and Murphy). The "Adult" dataset concerns real data from US census, and it contains 45,222 examples formed by 14 attributes. There are two classes, which describe whether incomes exceed $50K per year. The size of this dataset is 1.47 MB. The second dataset encompasses biopsy samples of diffuse large-B-cell lymphoma from 240 patients which were examined for gene expression with the use of DNA microarrays and analyzed for genomic abnormalities (Rosenwald et al., 2002). The 240 samples were divided into two groups: a preliminary group (training) of 160 patients and a validation group (testing) of 80 patients. The number of microarray features is 7399. This dataset is known as the "Diffuse Large-B-Cell Lymphoma" (here DLBCL, for short). The training set, whose size is 7.96 MB, was used in our experiments.

To execute performance tests, we used a cluster composed of PCs interconnected by a 100 Mbps Ethernet switch. Each cluster node is a Pentium IV 1.8 Ghz single processor with 512 KB cache, 128 MB of memory, running Linux operating system (kernel version 2.4.21) and JDK 1.4.1.

5.2. Scalability test

Initially, we executed the J48 and PART algorithms in the Adult and DLBCL datasets for 1, 10, 20, 30, 40, and 50 nodes in the cluster. The test for one node refers to a sequential execution with the original, non-modified Weka software. Each test comprised a 50-fold cross-validation, being repeated 10 times under the same conditions for Inhambu and Weka-Parallel, and the average results are reported in Tables 1 and 2. In general, both systems show similar performance with fine granularity tasks, while Inhambu performs better for applications with coarse granularity tasks.

For this first scenario, the cluster was fully dedicated to our experiment, avoiding interferences from other applications. As shown in Table 1, some reduction in the execution times can be achieved until 20 nodes for J48 and until 50 nodes for the PART method, for both Weka-Parallel and Inhambu running on the Adult dataset. As shown in Table 2, a similar reduction in the execution times can be achieved until 40 nodes for J48 for both Weka-Parallel and Inhambu running on the DLBCL dataset. For the PART method on the DLBCL dataset, reduction in the execution times can be achieved until 40 nodes for WP, and 50 nodes for Inhambu.

In this test, the maximum theoretical speedup can be considered for a better estimate of the enhancements achieved. The speedup is given by

\[ S = \frac{T_1}{T_P}, \]

where \( T_1 \) is the execution time for the sequential algorithm and \( T_P \) refers to the time for a parallel execution with \( P \) processors. Furthermore, it is worth comparing the speedup achieved against the maximum theoretical speedup. According to the Amdahl's law, performance improvements due to the implementation of parallel algorithms are limited by the fraction of the problem in which parallelism cannot be applied. Then, let \( f \) be the sequential fraction of the problem, the speedup to be achieved is limited as follows:

\[ S \leq \frac{1}{f + \frac{1-f}{P}}. \]

The preparation of the dataset comprises the sequential part of the cross-validation. As we mentioned in Section 4, the stratified \( n \)-fold cross-validation is implemented by the method evaluateModel() of the class Evaluation. Such method performs some actions to build an instance of Classifier and prepare to invoke crossValidateModel(). As the crossValidateModel() is the only part that can be parallelized by means of Inhambu, the fraction of the problem that was parallelized can be computed as the ratio between the execution times for performing crossValidateModel() and evaluateModel(). The sequential fraction can be estimated as: 0.014 for the J48-Adult, 0.019 for PART-Adult, 0.033 for J48-DLBCL and 0.025 for PART-DLBCL.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>The Adult dataset</th>
<th>The DLBCL dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>J48</td>
<td>PART</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>Inhambu</td>
</tr>
<tr>
<td>1</td>
<td>2688.24</td>
<td>2688.24</td>
</tr>
<tr>
<td>10</td>
<td>259.95</td>
<td>308.82</td>
</tr>
<tr>
<td>20</td>
<td>243.24</td>
<td>236.96</td>
</tr>
<tr>
<td>30</td>
<td>240.37</td>
<td>243.81</td>
</tr>
<tr>
<td>40</td>
<td>269.01</td>
<td>281.07</td>
</tr>
<tr>
<td>50</td>
<td>308.04</td>
<td>252.02</td>
</tr>
</tbody>
</table>

Table 1: Total run times (s) for the Adult dataset

Table 2: Total run times (s) for the DLBCL dataset

The achieved speedup as well as the maximum theoretical speedup for the classifier J48 running on the top of both Inhambu and Weka-Parallel with Adult and DLBCL datasets are depicted in Figs. 3 and 4, respectively, whereas similar experiments for the PART classifier are depicted in Figs. 5 and 6. For better confidence, the results depicted are the mean values for 10 executions of the same test under the same conditions.

Figs. 3–6 show that the speedup achieved for both systems is significantly lower than the maximum (theoretical) speedup when the number of slaves grows. It is worth noting that the performance results shown in Fig. 5 are slightly better than the results shown in the remaining ones. In this particular test, the achieved speedup grows up to 50 slaves (and perhaps more) when both systems execute the PART method on the Adult dataset. It happens because Inhambu and Weka-Parallel implement the master-slave model, in which the process that implements the Weka’s Evaluation object behaves as the master, thus becoming the system’s bottleneck. This bottleneck appears because the master is the only machine responsible for the transmission of the dataset to the slave processors, as well as for the reception of results. In such scenario, when the number of slaves exceeds a given threshold, the exceeding slaves begin to spend some time idle as the master is not capable to transmit datasets and receive results as fast as the slaves can execute tasks. Thus, there is a performance limitation which dictates the maximum number of processors that can be added to the system without losing performance. Such a threshold, here named as the effective number of slaves (ENS), can be estimated as a function of some application and architectural parameters as follows. Assume that files are sequentially transmitted to slave processors, i.e. the master does not perform concurrent transmissions.³ Also,

³ Obviously, this is a simplified model for the execution and transmission times. Although it does not reflect the overlapping of processing and communication at the master as observed in the Inhambu, it aims at explaining the scalability limitations observed in the tests. Such model is likely to be more representative for the Weka-Parallel, whose master node does not perform concurrent transmission of files.
assume that the size of input files are several orders of magnitude larger than the size of output files. This is precisely the case observed in these experiments, in which the size of the input files used for the cross-validation are 1.47 MB and 7.96 MB, while the output has a constant size of a few bytes (to represent the number of correctly classified instances). For the sake of simplicity, under such conditions the time required to transmit the output can be neglected. In such scenario, the maximum number of slave processors without loss of performance can be estimated as

$$\text{ENS} = 1 + \left[ \frac{\text{ET}}{\text{TTIF}} \right]$$

where ET is the mean execution time of a given set of tasks, and TTIF is the mean time required to transmit input files.

Obviously, this number depends on the granularity of the application tasks, since it reflects the ratio between the times involved in the execution of tasks and in the transmission of files. Above the number of slaves defined in Eq. (4), speedup will be lower than the theoretical speedup. Also, as evidenced in Figs. 3–6, both systems suffer from such limitation that is intrinsic to the master-slave model they implement.

Although the bottleneck can occur for some applications with low values of the computation/communication ratio (e.g. in Figs. 3, 4 and 6), it is worth noting that J4.8 (Figs. 3 and 4) is considered a lightweighted classifier whose execution time is slightly above the linear, and the size of the employed dataset is only 1.47 MB. Furthermore, although DLBCL’s (in Fig. 6) size is 7.96 MB, it has only 160 instances. In practice, other real-world datasets are expected to be larger. Also, several classification algorithms implemented in Weka present execution times asymptotically greater than linear. Consequently, the computation per communication ratio tends to raise for such datasets, and this bottleneck is not likely to appear for clusters with less than a few hundred hosts.

In Fig. 5 (PART-Adult), one can observe that no such performance limitation was verified for the cluster size evaluated. However, as shown in Senger et al. (2006), Silva et al. (2004a), Silva et al. (2004b), the bottleneck appears for master-slave architectures running applications whose tasks present low values for the ratio of computation per communication. Under this perspective, such problem may even occur for the PART algorithm when running on a larger cluster. For example, let us assume an application with similar execution and transmission times (i.e. whose tasks take around 54 s for J4.8-Adult and 1090 s for the PART-Adult) and dataset size (1.47 MB) to estimate the cluster size for which a bottleneck could occur. For the J4.8-Adult, the bottleneck was achieved for a cluster composed of approximately 20 processors. Since the execution times for the PART-Adult tasks are around 20 times longer than J4.8-Adult tasks, the corresponding bottleneck would be achieved when approximately 400 slaves or more are used, considering the simplified computational model used in Eq. (4).

Possible solutions to the scalability limits of DM tasks include, for instance, grouping tasks that share common files to execute in the same processor (Silva et al., 2004b). Under such approach, each file is transmitted only once for every worker processor. As the number of tasks is likely to be much larger than the number of workers, such optimization could lead to performance improvements as reported in a previous work (Silva et al., 2004b). Also, the master-slave architecture implemented by Inhambu could be adapted to a more scalable one which alleviates the master’s bottleneck, such as the hierarchical architecture described in Senger et al. (2006). As far as Inhambu is concerned, although such optimizations could potentially improve its scalability, the corresponding implementation would raise the complexity of the system, and require a complete refactoring and reimplementation of several Weka’s classes (e.g. Evaluation and Classifier, to mention a few). Consequently, the adaptation of Inhambu for future releases of Weka would be significantly harder to perform. In this sense, there is a compromise relationship between performance optimization and software development. We have tried to manage it in our project and Inhambu has shown a good performance when compared with Weka-Parallel (which does implement the former optimization, by sending each dataset only once for each computer). Inhambu performs better than Weka-Parallel, mainly because Weka-Parallel has only one thread that transmits the dataset sequentially for each computer, as opposed to Inhambu, which spawns a pool of threads that concurrently transmit the dataset for the remote computers within RMI calls. Performance gains in favor of Inhambu are due to the better utilization of resources at the master computer.

As the conclusion for this test, Inhambu and Weka-Parallel performs roughly similar for fine granularity tasks (see J4.8’s experimental results), and Inhambu performs better than Weka-Parallel when running tasks whose granularity is coarser (e.g. see the results for PART). Also, it is worth noting that Weka implements many algorithms whose granularity is even coarser than PART.

5.3. Non-dedicated scenario

In order to show that Inhambu properly deals with dynamic resource sharing, tests were performed with three different scenarios which simulate realistic applications. To do so, we produced background workloads in random patterns at each machine. Such applications continuously alternate from periods of activity and idleness. Each machine $M_i$ runs a background application with a main loop which performs a number of $n_i$ computing intensive operations with matrices of $b_i$ bytes, followed by idle periods of random duration, whose length is exponentially distributed with average $\lambda_i$. Such background application has been previously used in Senger and Sato (2003) to produce...
a resource sharing scenario for performance evaluation. The length of idle periods are generated randomly by using the inverse transform technique, which is commonly used to produce synthetic workloads for discrete event simulation systems (Cassandras, 1993). The test procedure comprises the tenfold cross-validation (which is the Weka's default cross-validation procedure). The cross-validation was executed with the PART algorithm on the Adult dataset.

The scenarios 1 and 2 were produced for testing how well Inhambu is able to choose the idle machines when a subset of the cluster machines may have background workloads. The last scenario was created for testing how well Inhambu can chose the best machines when all of the cluster machines present concurrent workloads. The first scenario involves 21 homogeneous machines that are employed as following. Ten machines are allocated in a dedicated fashion to execute DM tasks. Other 10 machines are shared by DM tasks and background applications, which were configured to produce low-to-moderated levels of load. Finally, the remaining machine is used to execute: (i) DM tasks, (ii) the client application, and (iii) the application manager, which can be the Inhambu’s trader process or the Weka-Parallel’s main process. The second scenario was created by changing the background application’s parameters, so that moderated-to-high levels of load were produced in the non-dedicated machines. Finally, the third scenario was produced with 16 machines, in which 15 machines execute the background application with different levels of load. These varying levels of workload were produced by choosing different values of \( n, b, \) and \( \lambda \) for each node \( M, \) as described in Senger and Sato (2003).

For each scenario, we have performed 50 tests, for which the mean values and standard deviations are reported in Table 3. As reported in Table 3, Inhambu led to a remarkable performance improvement for these scenarios, by reducing the total execution times and standard deviations. In addition, if the number of tasks is less than the number of processors, the Weka-Parallel’s scheduling policy replicates tasks that are already running (but not concluded) in the remaining processors. Clearly, this policy leads to the waste of computing cycles. Also, Weka-Parallel does not distinguish between idle and busy processors. Thus, other applications that are executing on the cluster machines may experience lost of performance due to replicated tasks.

### Table 3

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Weka-Parallel</th>
<th>Inhambu</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>Scenario 1</td>
<td>3651.07</td>
<td>216.02</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>4155.67</td>
<td>362.79</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>3983.41</td>
<td>268.55</td>
</tr>
</tbody>
</table>

5.4. Heterogeneous clusters

An heterogeneous cluster composed by seven PCs interconnected by a 100 Mbps Ethernet switch was used for evaluating how well Inhambu deals with heterogeneity. Architecture details are described in Table 4, and performance results are reported in Table 5. Inhambu led to a reduction in the total execution times for the PART algorithm, for both dedicated and shared scenarios as shown in Table 5. The shared scenario is similar to the third scenario used in Section 5.3, with all nodes executing concurrent applications. Such gains reflect its ability of choosing suitable machines in a heterogeneous scenario. Notice that Weka-Parallel can lead to better performance in presence of shorter tasks, such as J4.8, due to its low communication overhead (remember that Weka-Parallel uses sockets). Despite of the higher overhead due to the use of RMI, Inhambu led to better performance in presence of longer tasks, showing that the extra overhead can be compensated by the choice of suitable computers.

### Table 4

<table>
<thead>
<tr>
<th>Characteristics of the server hosts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1–4</td>
</tr>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Cache</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Kernel</td>
</tr>
</tbody>
</table>

### Table 5

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Weka-Parallel</th>
<th>Inhambu</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>J4.8-Adult (dedicated)</td>
<td>531.51</td>
<td>2.70</td>
</tr>
<tr>
<td>J4.8-Adult (shared)</td>
<td>626.96</td>
<td>39.14</td>
</tr>
<tr>
<td>PART-Adult (dedicated)</td>
<td>11,470.71</td>
<td>169.42</td>
</tr>
<tr>
<td>PART-Adult (shared)</td>
<td>14,970.77</td>
<td>225.87</td>
</tr>
</tbody>
</table>

5.5. Low intrusiveness

Low intrusiveness was also a requirement for the design and implementation of Inhambu’s performance sensors. Monitoring agents are implemented as 3.2 KB bytecode files. Agents running on each server computer wake up after a few seconds, take measures from the kernel by reading a few dozen of memory positions, send such information over the network, and turn to sleep again. The processor consumption due to the execution of an agent is less than 1%, which can be considered negligible. Furthermore, search operations (e.g. the service lookup executed within an importService, and the updating of utilization indices of machines) are implemented by hash functions whose complexity is \( O(1) \).
5.6. Ease of use

Only two minor modifications to the original Weka’s interfaces (Witten and Frank, 2000) have been necessary for the user choose between the serial or parallel execution. In the command line interface, a new flag -P has been included for this purpose, as illustrated in the following:

```
java weka.classifiers.rules.PART -P data-set.arff
```

Similarly, the original Weka’s graphical interface (Witten and Frank, 2000) has been maintained, except for one minor modification. A checkbox was included to allow the user to choose between the sequential and the parallel execution with a single click. In addition, the user is only required to have a valid login account in the cluster machines, and to provide a text file containing the IP addresses of the nodes in the cluster.

6. Conclusion

We have presented and assessed Inhambu, a distributed object-oriented system that supports the execution of data mining applications on clusters of PCs and workstations. The main features and capabilities of Inhambu include: (i) the support for dynamic detection and management of idle cycles to execute data mining tasks, (ii) the management of differentiated capacities of heterogeneous computers, (iii) the dynamic aggregation of computers to the cluster, and (iv) the automatic restarting of failed tasks. This system is implemented as a resource management layer, built on the top of Java/RMI, that supports the execution of the data mining tool named Weka. We have evaluated the performance of Inhambu by means of several experiments in homogeneous, heterogeneous and non-dedicated clusters, comparing its results with those achieved by a similar system named Weka-Parallel. In our tests, Inhambu and Weka-Parallel presented roughly similar performance for fine grain tasks. However, Inhambu showed better performance for coarse grain tasks. Indeed, the execution of most Weka’s classification algorithms produce coarse granularity tasks and can benefit from using Inhambu. By monitoring the cluster machines, Inhambu gathers both static and dynamic performance indices that are used for scheduling DM tasks. Such a feature allows better scheduling decisions to be taken, mainly in presence of heterogeneous and non-dedicated hosts. As a consequence, performance can be optimized in two ways. First, execution times of DM tasks can be reduced with the support of Inhambu. Second, the performance of other applications executing on the hosts can be better preserved. As our final remark, Inhambu can leverage the use of available computer power existing in companies, laboratories and universities to execute real-world data mining applications based in the Weka tool.

References


Merz, C.J., Murphy, P.M. UCI repository of machine learning databases. Available from: <https://www.ics.uci.edu>, University of California, Irvine, CA.


