High-Performance Knowledge Extraction from Data on PC-based Networks of Workstations

Cosimo Anglano[1] *, Attilio Giordana[1], Giuseppe Lo Bello[2]
[1] DSTA, Università del Piemonte Orientale, Italy.
[2] Data Mining Laboratory, CSELT, Torino - Italy.

Abstract. The automatic construction of classifiers (programs able to correctly classify data collected from the real world) is one of the major problems in pattern recognition and in a wide area related to Artificial Intelligence, including Data Mining. In this paper we present G-Net, a distributed algorithm able to infer classifiers from pre-collected data, and its implementation on PC-based Networks of Workstations (PC-NOWs). In order to effectively exploit the computing power provided by PC-NOWs, G-Net incorporates a set of dynamic load distribution techniques that allow it to adapt its behavior to variations in the computing power due to resource contention. Moreover, it is provided with a fault tolerance scheme that enables it to continue its computation even if the majority of the machines become unavailable during its execution.

1 Introduction

The induction of classification rules from a database of instances pre-classified by a teacher is a task widely investigated both in Pattern Recognition and in Artificial Intelligence. Image interpretation, medical diagnosis, troubleshooting, text retrieval, and many other applications can be seen as specific instances of this task. The popularity recently gained by the so called *Data Mining* [5, 10] has brought a renovated interest on this topic, and has set up new requirements that started a new trend aimed at developing new algorithms. As a matter of fact, most Data Mining applications are characterized by huge amounts of data that exceed the capabilities of most existing algorithms, such as neural networks, decision trees, and so on. Therefore, a new generation of induction algorithms have been recently proposed in the literature, and the research in this area is currently very hot. Among the emerging approaches, evolutionary algorithms are gaining more and more interest since, on the one hand, they proved to be flexible and easy to apply to a wide variety of problems, obtaining excellent performance, and on the other, they naturally lend themselves to exploit large computational resources such as the ones offered by a network of workstations.

In this paper we will present G-Net [2], a distributed evolutive algorithm for inferring classification rules from data that integrates strategies borrowed from

* Corresponding author. Address: Dipartimento di Scienze e Tecnologie Avanzate, Universitá del Piemonte Orientale, Corso Borsalino 54, 15100 Alessandria, Italy. email: mino@di.unito.it

evolutionary computation, genetic algorithms and tabu-search, and we will discuss its implementation on PC-based Networks of Workstations (PC-NOWs). Although the G-Net algorithm is equally suited to various kinds of parallel computing systems having different granularities (ranging from SIMD machines to massively parallel processors), thanks to its architecture that couples a very fine granularity of the computational objects with the absence of global synchronization points, we decided to devise an implementation targeted to PC-NOWs for two main reasons. First, the dramatic advances in low-cost computing technology, together with the availability of new high speed Local Area Networks have enabled the construction of PC-NOWs whose raw processing and communication performance are comparable to those of traditional massively parallel processors. Second, relatively large systems can be built without massive investments in hardware resources, since PCs are relatively inexpensive and, more importantly, already existing machines can be exploited as well. Therefore, by exploiting PC-NOW platforms, G-Net can be used to solve problems of considerable complexity without the need of massive investments in computing resources.

The above advantages, however, do not come for free. In order to exploit the opportunities provided by PC-NOWs, it is necessary to develop applications able to deal with some peculiar features of these platforms, and this makes the development much harder than in the case of traditional parallel systems. In particular, applications must be able to adapt to the occurrence of machine failures during their execution, and to tolerate the effects of resource contention (i.e., variations in the performance delivered by the various PCs) due to the presence of several users sharing the machines. Therefore, the main part of the work required to port G-Net on PC-NOWs platforms was devoted to the development of techniques allowing the application to effectively deal with the above issues.

The paper is organized as follows. Section 2 gives a brief presentation of the G-Net algorithm. In Section 3 we discuss the main issues that we had to face when we designed and implemented G-Net on a PC-NOW. In particular, Section 3.1 discusses the software architecture of our application, Sections 3.2 and 3.3 present the dynamic load balancing techniques devised to deal with heterogeneity and resource contention, while Section 3.4 illustrates the fault-tolerance mechanisms. Section 4 presents some experimental results, and Section 5 concludes the paper and outlines future work.

2 The G-Net Algorithm

In this section we give a brief description of the G-Net algorithm, focusing on the logical behavior of its various computational entities (the interested reader may refer to [2] for a more detailed presentation of its features).

G-Net learns classification rules for a class h of objects starting from a dataset $D = D^+ \cup D^-$, being D^+ and D^- sets of elements belonging and not belonging to h, respectively. In particular, its goal consists in finding a set of classification rules (henceforth referred to as a *classification program* or *classifier*) of the type

$\{\varphi_1 \rightarrow h, \varphi_2 \rightarrow h, \dots \varphi_n \rightarrow h\}$, where h denotes the class to discriminate and $\varphi \rightarrow h$ means "if conditions φ is true for an object x, then $x \in h$".

Symbolic induction algorithms as G-Net essentially combine two types of strategies: Hypothesis (rule) generalization and set-covering. The former strategies concern the way hypotheses are generated, that is, how the hypothesis space is actually searched; different approaches apply different inductive operators for moving through the search space and different evaluation criteria to determine the order in which different alternatives are explored. Set-covering strategies deal instead with the global focus of the search: The coverage of all given positive objects (D^+) and possibly none of the negative ones (D^-). A fundamental issue of learning disjunctive classifiers (i.e., classifier containing more than one classification rule) is the number of different rules (disjuncts) to put into the final classifier. Set-covering strategies try to cover the learning set with the smallest number of classification rules.

In G-Net both hypothesis generalization and set-covering strategies are based on Genetic Algorithms (GAs) [6]; in particular, hypothesis generalization is based on task specific genetic operators, while set-covering is based on cooperative coevolution [12, 11].

Let us now describe in more detail the G-Net algorithm. G-Net has a distributed architecture that encompasses two levels. The lower level is a distributed genetic algorithm that stochastically explores the search space, while the upper level applies a coevolutive strategy that directs the search process of the genetic algorithm towards globally best solutions. The above architecture is schematically depicted in Fig. 1, and encompasses three kind of processes, namely Genetic Nodes (G-Nodes), Evaluator Nodes (E-Nodes), and a Supervisor Node. Each G-

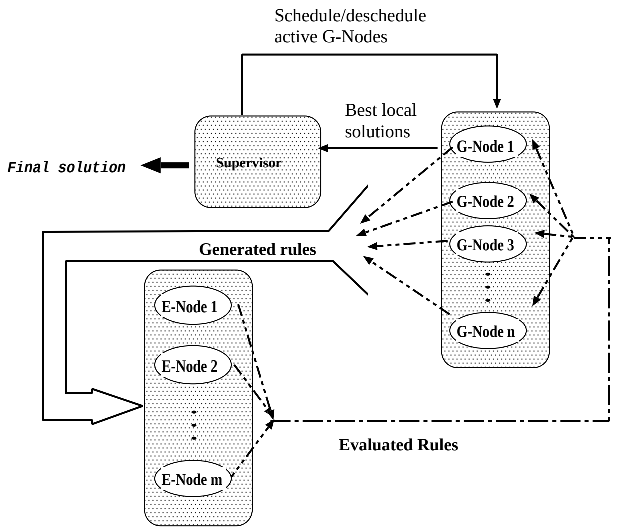

Fig. 1. Distributed Architecture of G-Net

Node corresponds to a particular element $d \in D^+$, and is in charge of finding rules covering d, which are stored into the G-Node's *local memory*, by executing a genetic algorithm schematically working as follows:

1. Select two rules from the local memory with probability proportional to their fitness;
2. Generate two new rules φ_1 and φ_2 by applying suitable genetic operators;
3. Send φ_1 and φ_2 to an E-Node for evaluation in order to obtain their fitness values
4. Receive from the E-Nodes a set of evaluated rules (if available) and stochastically replace some of the rules already in the local memory with them.
5. Go to step 1

As indicated in the above algorithm, E-Nodes perform *rule evaluation*: Each time a G-Node generates a new rule, sends it to an E-Node (since all the E-Nodes use the same dataset D, anyone of them may be used) that matches it against each element of the dataset D, and assigns it a *fitness* value, a numerical quantity expressing the "goodness" of the rule. When the evaluation of a rule is completed, the E-Node broadcasts it (together with its fitness value) to all the G-Nodes, so during Step 4 of the genetic algorithm a G-Node may receive also rules generated by other G-Nodes. G-Nodes and E-Nodes work asynchronously, that is a G-Node can continue generating new rules while already-generated ones are still under evaluation. However, for the sake of convergence, the number of rules that a G-Node can generate without using new rules is limited to a predefined threshold (called the *G-Node credit*).

The Supervisor Node coordinates the execution of the various G-Nodes according to the coevolutive strategy. Its activity basically goes through a cycle (called *macro-cycle*), having period measured in terms of iterations performed by the G-Nodes (*micro-cycles*), in which it receives the best rules found by the G-Nodes, selects (by means of a hill-climbing optimization algorithm) the subset of rules (the *hypothesis pool*) that together give the best classification accuracy for class h, and chooses the subset of G-Nodes that will be *active* (i.e. allowed to execute their genetic algorithm) during the next macro-cycle. As a matter of fact, although in principle all the G-Nodes might be simultaneously active if enough computational resources were available, in real situations the number of processors is limited. Since in a typical Data Mining application the number of G-Nodes ranges from hundred of thousands to several millions, it is evident that even a very large PC-NOW cannot provide enough resources, so each machine will be shared among several (possibly many) G-Nodes. In order to deal with this issue, only a subset of G-Nodes (henceforth referred to as *G-Nodes working pool*) will be activated during each macro-cycle, and in order to ensure a proper coverage of the search space, the composition of the working pool is dynamically varied by the Supervisor (across different macro-cycles) by means of a stochastic criterion that tends to favor those G-Nodes whose best rules belong to hypothesis pool and need further improvement (see [2] for more details). The size of the working pool is specified by the user, and depends on several factors, such as the particular classification task; the duration of a macro-cycle is defined as $G \times \mu$, where G and μ denote the number of active G-Nodes and the number of micro-cycles per G-Node per macro-cycle, respectively.

The execution of the algorithm terminates when a satisfactory global solution is found, that is when the quality of the found classifier does not improve for an assigned number of macro-cycles. Finally, the classifier constructed in this way is validated on a new dataset T such that $D \cap T = \emptyset$.

3 Porting G-Net on PC-based NOWs

Let us now discuss the issues involved in the design of the PC-NOW version of G-Net. Porting G-Net on a traditional parallel computing system is relatively simple, since the distributed architecture of the algorithm can be easily translated into a set of parallel processes. Our implementation, however, was targeted to PC- NOWs, and this required to take into considerations several aspects that can be disregarded in the case of traditional parallel computers.

First of all, we had to choose a software environment allowing to use the ensemble of PCs as a parallel computer. Since one of our goals was portability, we have chosen PVM [13], that is practically available for every platform. This choice, however, is not critical, since PVM is used only for the implementation of a few communication operations and for the heterogeneity support, so replacing it with a different system is straightforward (for instance, an MPI implementation is planned for the near future).

Second, PC-NOWs are characterized by the presence of phenomena not arising in traditional parallel computing systems that make the porting task much more challenging. In particular, since the PCs making up a NOW may deliver different performance from each other (if they are equipped with different processors), and their performance may vary over time (if several users share the machines), it was necessary to adopt a dynamic load distribution policy able to assign to each machine an amount of work proportional to its speed, and to keep the above distribution balanced during the execution of the application.

Finally, the PC-NOW used to execute the application may be relatively unstable, since some of the machines may become unavailable at run-time. Therefore, in order to allow G-Net to continue its execution when one or more machines become unavailable, it was necessary to incorporate into the application a suitable mechanism providing it with some form of fault tolerance.

In the following of this section we present in detail the mechanisms mentioned before. We start with the presentation of the software architecture of our implementation (Sec. 3.1), we continue with the load distribution techniques in Sections 3.2 and 3.3, and we conclude with a brief discussion of the fault tolerance mechanisms (Section 3.4).

3.1 Software Architecture

The software architecture of our implementation of G-Net on PC-NOW platforms, schematically depicted in Fig. 2, encompasses two kinds of processes, namely *Searchers* (that generate rules) and *Matchers* (that perform rule evaluation), besides the Supervisor Process (corresponding to the Supervisor Node and not shown in the figure).

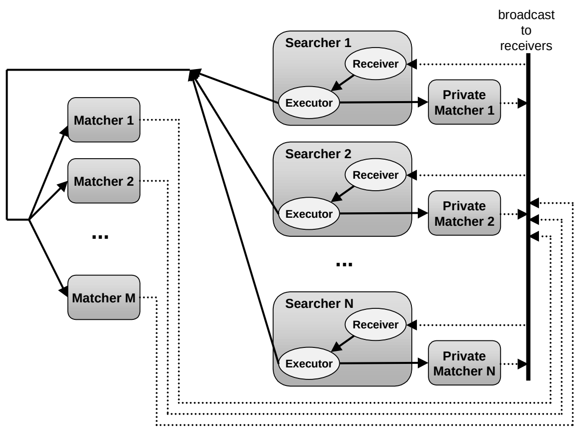

Fig. 2. Software Architecture of G-Net. Ovals correspond to threads, gray boxes to processes, continuous arrows to rules sent for evaluation, and dotted arrows to evaluated rules.

Searcher processes are made up by two threads, called *Executor* and *Receiver*, respectively [2], and split among them the computation load required by the G-Nodes. In particular, each Executor receives a portion of the G-Nodes working pool (called *local working pool*) whose size depends on the computing power allocated to it, multiplexes the processor among its locally active G-Nodes, and alters across different macro-cycles (as the Supervisor Node described in the previous section) the composition of the local working pool. The duration of the macro-cycle of Executor E_i is $wps(E_i) \cdot \mu$, where $wps(E_i)$ is the size of its local working pool. It is worth pointing out that only the G-Nodes working pool is partitioned, and *not* the set of G-Nodes; that is, each Executor chooses the G-Nodes that will be active during a macro-cycle among all the G-Nodes defined in the system.

The Receiver thread is in charge of receiving evaluated rules from the set of Matchers, and acts as a filter for the corresponding Executor. In particular, it does not forward to its Executor all the evaluated rules generated elsewhere (*non-local* rules) since, for the sake of convergence, it is necessary that a) the percentage of non local rules used by an Executor does not exceed a given threshold and b) the difference between the micro-cycle executed by the Executor (on the behalf of its various G-Nodes) and the micro-cycle during which the rule was generated does not exceed a predefined value (i.e., the rule is not too *old*). Finally, Matchers correspond to E-Nodes, and are in charge of evaluating on the

[2] Since PVM does not provide threads, at the moment Searchers are implemented as a pair of communicating processes executed on the same machine.

dataset D the rules generated by the various Searchers.

The last issue that needs to be discussed is concerned with the choice of the number of Searchers and Matchers. As a general rule, the number of Searchers should be smaller than that of Matchers, since the computationally heavy activity is rule evaluation and not rule generation. Moreover, in order to speed-up the execution as much as possible, the number of Searchers and of Matchers should be chosen in such a way that the rule generation rate and the rule evaluation rate are simultaneously maximized. The (usually) limited number of available machines, however, is often insufficient to simultaneously satisfy the above requirements, so the machine usage must be carefully balanced. As a matter of fact, giving a large number of machines to Searchers increases the rule generation rate, but decreases the amount of computation power assigned to Matchers and, hence, the rule evaluation rate. Conversely, an increase in the number of machines dedicated to Matchers increases the rule evaluation rate, but decreases the generation rate. This choice is further complicated by the fact that the power of the various machines may vary over time as a consequence of resource contention. Therefore, finding the optimal allocation is very hard, if not unfeasible.

In the light of the above considerations, we have followed a different approach: Rather then trying to find an optimal allocation, we require the user to specify (using his/her expertise) the number of Searchers and Matchers (their sum must not exceed the number of available machines), and we use a flexible allocation scheme that allows G-Net to exploit the computing power that may be left unused as a consequence of a poor distribution. In particular, given N Searchers and P hosts ($N < P$), the Searchers are scheduled on the N fastest machines, while Matchers are executed on the remaining $P - N$ machines. Since the above mapping may yield idle time on the Searchers' machines (this happens when Matchers do not have enough computing power to generate an evaluation rate high enough to avoid that Executors have to wait for evaluated rules to come back), each Executor is provided with a *private* Matcher, which is executed on the same machine with the aim of exploiting the above idle time. The private Matcher can be used only by the corresponding Executor, although the rules it evaluates are broadcast to all the Searchers. Each Executor starts to use the private Matcher instead of a shared one as soon as it determines that the arrival rate of evaluated rules is becoming insufficient to sustain its execution (of course, the Executor has to feed its private Matcher *before* coming to a stopping point). In particular, the Executor monitors the value of its credit, that is the amount of new rules that can be generated without receiving evaluated rules (see Section 2): Each time an Executor passes Step 4 of the genetic algorithm without actually receiving an evaluated rule, its credit is decremented by one, while it is incremented of the same amount each time an evaluated rule is received. When the current credit value drops below 50% of the initial value, the Executor sends the new rules to the private Matcher, otherwise it chooses one of the shared Matchers (the criterion used to choose a shared Matcher is discussed in Section 3.3). Note that the exploitation of the private Matcher increases the rule arrival rate while decreasing the speed of the Executor, so after some time the credit value will exceed the 50% threshold and the Executor will start using again the pool of shared Matchers.

3.2 Dynamically Balancing the Workload among Executors

As mentioned in the previous section, each Executor is associated with a local working pool. The size of this pool must be chosen in such a way that all the Executors complete their macro-cycles approximately at the same rate, otherwise the convergence speed and the quality of the computed classifiers would be hampered. As a matter of fact, if some Executors are excessively slower than the other ones, the rules generated by them will be discarded by the faster Executors because of their age. In order to have all the Executors to proceed approximately at the same pace, the number of G-Nodes assigned to each of them should be proportional to the computing power delivered by the machine on which it is allocated. Moreover, since the above power varies as the load due to other users varies, the local working pool size must vary as well. To achieve this goal, we have developed a dynamic load balancing algorithm based on the *hydrodynamic approach* proposed by Hui and Chanson [8].

The hydrodynamic approach defines a framework for the development of *nearest-neighbor* dynamic load balancing algorithms, that is algorithms that allow the workstations to migrate tasks with their immediate neighbors only. Each workstation W_i is associated with two real functions, the *load* $l_i(t)$ and the *capacity* $c_i(t)$ (reflecting its workload and its speed in processing it at time t, respectively), and is represented as a cylinder having a cross-sectional area given by $c_i(t)$ and containing a quantity of liquid given by $l_i(t)$. The *height* of the liquid in cylinder i at time t is defined as $h_i(t) = l_i(t)/c_i(t)$ (for the sake of readability, in what follows we denote capacity, load, and height of workstation W_i as c_i, l_i and h_i, respectively). The network connections among workstations correspond to infinitely thin liquid channels that allow the liquid to flow from one cylinder to another. The goal of the dynamic load balancing algorithm is to distribute the liquid among the various cylinders (by means of the liquid channels) in such a way that their heights are equal. In [8] Hui and Chanson prove that all the distributed nearest-neighbors algorithms that satisfy certain conditions converge geometrically to the optimal distribution.

The load balancing algorithm developed for G-Net is executed only by Executors. In particular, given a set of Executors $\{E_1, E_2, \ldots, E_n\}$, we associate with each E_i its capacity c_i, measured in terms of micro-cycles executed per second (computed by means of a benchmark executed in absence of contention on the workstation on which it is allocated), and its load l_i, that corresponds to the number of micro-cycles that have to be performed to complete a macro-cycle (computed as $wps(E_i) \cdot \mu$). Consequently, the height h_i corresponds to the amount of seconds needed to E_i to complete a macro-cycle, so the goal of having the same height for all the Executors corresponds, as expected, to keep the Executors proceeding at the same speed.

Initially, $wps(E_i) = c_i \cdot (\sum_{j=1}^{n} l_j / \sum_{j=1}^{n} c_j)$, so all the Executors have the same height $h_i = \mu \cdot \sum_{j=1}^{n} l_j / \sum_{j=1}^{n} c_j$ and, hence, proceed at the same speed. During the execution of each macro-cycle, E_i periodically measures the average speed s_i at which it completes the micro-cycles and broadcasts it to the other Executors (note that s_i may increase or decrease according to the presence of additional jobs). At the end of each macro-cycle, E_i builds a list (the *speed list*) in which all Executors (including itself) are ranked in increasing order according to their speed, and divides it into two parts of equal size. Therefore, the upper part of the list will contain *overloaded* Executors that are slower than the *underloaded* Executors contained into the lower part. If E_i is overloaded, it decreases the size of its local working pool of a suitable amount x of active G-Nodes (see below), i.e. $wps(E_i) = wps(E_i) - x$, and chooses an underloaded Executor to assign the execution of this surplus. If, conversely, E_i is underloaded, it waits for a notification, sent by some overloaded Executor, concerning the amount x of active G-Nodes that must be added to its local working pool (i.e., $wps(E_i) = wps(E_i) + x$). At the end of the balancing step, each Executor broadcasts to all the other Executors the size of its local working pool.

Each overloaded Executor E_i whose position in the list is $pos(E_i)$ ($pos(E_i) < n/2$) balances its workload with the underloaded Executor listed at the position $n - pos(E_i) + 1$. In this way, a given underloaded Executor may be chosen only by a single overloaded Executor (note that the list is the same for all the Executors, since the speeds are globally known), so that excessive fluctuations in the load assigned to underloaded Executors are avoided, and a balanced state should be reached in less balancing steps.

The amount of active G-Nodes sent by the overloaded Executor E_i to an underloaded Executor E_j is computed as $\gamma(c_i * c_j/(c_i + c_j))(h_i - h_j)$, where $\gamma = 0.7$ (the minimum value recommended by [8]), $c_j = 1/s_j$, and h_i is computed by setting $l_j = wps(E_j) \cdot \mu$ (note that s_j and $wps(E_j)$ are globally know to all the Executors).

The above load balancing algorithm satisfies all the conditions required by the hydrodynamic framework [1], so it is guaranteed to converge geometrically to the optimal solution.

3.3 Dynamically Distributing Rule Evaluations among Matchers

As discussed in Section 3.1, when an Executor generates a rule and the private Matcher cannot be used, one of the shared Matchers must be chosen to evaluate it. This choice is performed autonomously by each Executor by means of a *rule distribution policy* whose goal is to reduce, as much as possible, the time required to evaluate the rules in such a way that the amount of time they spend idle is minimized.

Although the problem of Matchers selection is equivalent to that of assigning tasks to servers in distributed systems [7], many of the solutions developed in this context cannot directly be applied to G-Net since they do not deal with heterogeneity and resource contention. As a matter of fact, it is crucial for G-Net that rule distribution is performed is such a way that all the Matchers proceed at the same evaluation rate in spite of possible differences in the speed of the corresponding machines, otherwise it may happen that the work performed by the slower Matchers is completely wasted. To illustrate this phenomenon, consider the simple case were all the Matchers proceed at the same speed, except one which is slower. In this case, if no special care is taken, the rules evaluated by the fast Matchers will allow the Executors' computations to proceed without interruptions. Consequently, when a rule evaluated by the slow Matcher is received by an Executor, it is very likely that it has completed a significant number of micro-cycles since when the rule was generated, so the rule may be too old and, consequently, discarded by the Receiver.

In order to select a suitable rule distribution policy, we have considered three different policies that represent extensions of algorithms developed for task allocation, and we have compared them via simulation. The first policy, henceforth referred to as *work stealing*, was adopted in the first implementation of G-Net [2], and is based on a client/server scheme in which an Executor sends a new rule to a Matcher only if there is a Matcher immediately available for evaluation, otherwise stores it into a local FIFO queue. When a Matcher finishes to evaluate a rule, it signals to the Executor that generated it that it is willing to accept a new rule for evaluation. Each Matcher servers the various Executor in a round robin fashion, in order to avoid that an Executor has exclusive use of the Matcher resources. The rationale behind this algorithm is that faster Matchers will reply more often than slower Matchers and, consequently, will receive more rules to evaluate. Also, if the speed of a Matcher decreases, it will start to reply less often and, consequently, will receive less and less rules to evaluate.

The second policy we consider is the *Join the Shortest Queue* [14] (JSQ) policy. In this case, unlike work stealing, Executors do not store locally a new rule if no free Matchers are available, but rather they send it to a Matcher as soon as it is generated. If the chosen Matcher is busy, the rule waits into a FIFO queue and is evaluated later. Each time an Executor has to send a rule, it chooses the Matcher with the shortest queue, since a shorter queue indicates that the Matcher is able to process its rules faster than Matchers with longer queues. This algorithm is inherently dynamic, since if a Matcher becomes slower, its queue will grow and, as consequence, it will be chosen less frequently. The implementation of the algorithm is straightforward: Each time a Matcher broadcasts an evaluated rule to all the Executors, it piggy-backs the information concerning its queue length on the messages containing the evaluated rule.

Finally, the last policy we analyzed is called *Join the Fastest Queue* (JFQ), and is derived from the Join the Shorted Expected Delay Queue [14]. This policy is similar to the JSQ one, with the difference that now an Executor chooses the Matcher that minimizes the expected time required to evaluate a new rule. In order to allow Executors to perform their choices, each Matcher M_i keeps track of the average time RET_i it needs to evaluate a rule, and computes the expected time that a rule should spend in its queue before it is evaluated as $RET_i \cdot qlen(E_i)$ (where $qlen(E_i)$ denotes the length of its queue). This information is piggy-backed on each evaluated rule and, consequently, broadcast to all the Executors.

In order to evaluate the effectiveness of these policies, we have developed a discrete event simulator, and we have evaluated their performance for two different scenario. As will be shown by our results, the JFQ policy results in the best performance for various system configurations that represent the opposite extremes of the whole spectrum, so it has been chosen for use in G-Net.

Our analysis focused on two performance indices that allow to assess the suitability of each policy to the needs of G-Net. The first one is the average *rule round trip time* (RTT), defined as the time elapsing from when a rule is generated to when the evaluated rule is received by the Executor that generated it, and indicates the speed at which evaluated rules are received by the Executors. The second index is the *evaluation time range*, and is computed as follows. Let ET_{M_i} be the average time elapsing from when a rule is generated to when its evaluation is completed by M_i, and M_{min}, M_{max} the slowest and fastest Matchers, respectively. The evaluation time range is computed as $ET_{M_{max}} - ET_{M_{min}}$, and indicates the maximal difference in the speeds of Matchers and, hence, the probability of discarding rules because of their age. For all the results reported in this Section, the 95% confidence interval is within 3% of the sample mean.

In the first scenario, corresponding to the graphs shown in Figs. 3 and 4, we have kept fixed the number of Executors (8), and we have progressively increased the number of Matchers from 8 to 24. The rule generation speed has been kept equal to a given value S_G for all the Executors in all the experiments. Also, in all the experiments we have used 4 “slow” Matchers, having an evaluation speed 5 times smaller than S_G, while we have progressively increased the number of “fast” Matchers having a speed only 2 times smaller than S_G. As can be

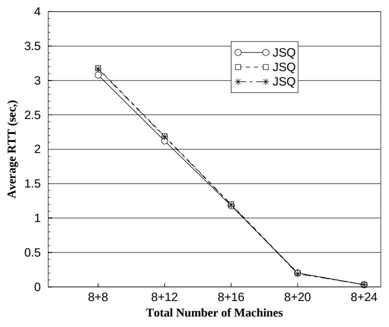

Fig. 3. Average rule RTT

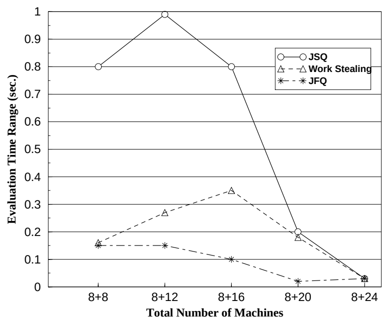

Fig. 4. Evaluation Time Range

observed from the graph in Fig. 3, all the three policies have practically the same average rule RTT, while the results for the evaluation time range (Fig. 4) are quite different. In particular, we can observe that the JSQ policy has the largest evaluation time range, that corresponds in the highest probability that evaluated rules are discarded because of their age, while the best performance is provided by the JFQ policy.

The second simulation scenario is identical to the first one, the only difference being that the evaluation speed of fast Matchers is 5 times smaller than S_G, while that of slow Matchers is 100 times smaller than SG. From the graphs of

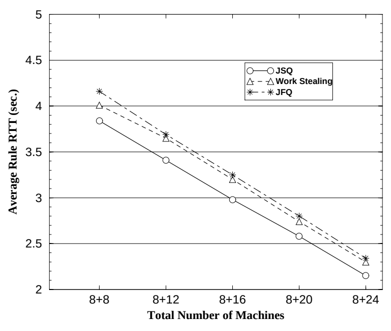

Fig. 5. Average rule RTT

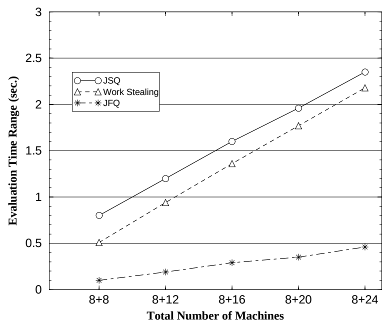

Fig. 6. Evaluation Time Range

Figs. 5 and 6 it is evident that the JFQ policy results in a slightly larger average rule RTT but, as in the first scenario, in a much lower evaluation time range, especially when the number of Matchers gets large.

3.4 Fault Tolerance

As discussed at the beginning of this section, PC-NOWs tend to be relatively unstable, especially if they are made up by machines non-dedicated to the execution of parallel applications (e.g., machines used for teaching purposes that become idle during night hours). In these cases, it may happen indeed that a given machine becomes temporarily or permanently unavailable (for instance, when a user decides to reboot it to run a different operating system). Since the application may have been running for a long time (for realistic data sets the computation may last entire days), it would be an unacceptable waste of resources if, as a consequence of the failure of one or more machines, the application crashed or its execution had to be prematurely terminated.

The basic mechanism employed to allow G-Net to continue its execution, albeit with a possible slowdown or loss of accuracy, when one or more machines fail is nothing else that its stochastic nature and the lack of synchronization points. As a matter of fact, the lack of synchronization points guarantees that the failure of a process will not block other processes, while stochastic search implies that a particular region of the search space is not explored exclusively by a particular process, and consequently the failure of some process does not exclude possibly significant solutions from being found. In order to clarify this point, let us consider how the failure of Searchers and Matchers affect the execution of the entire application.

If the failing machine is running a Matcher, then all the rules enqueued at the Matcher are inevitably lost. This loss, however, is not a major problem because the stochastic nature of the algorithm ensures that losses of new rules do not affect the convergence of the algorithm (since the lost rules have a non null probability of being generated again). Moreover, each Executor does not need to receive back all the rules it generates, since it can continue its execution using also rules generated by somebody else. It is however important that the Executors are informed of the Matcher crash so that they avoid to keep sending it rules. This is achieved by a simple timeout mechanism in which each Executor uses a timer, set to a suitable value each time an evaluated rule is received, for each Matcher: If the timer expires before another rule is received by the above Matcher, the Executor assumes that it is no longer operational.

If, conversely, the failing machine is running a Searcher, then the execution will be certainly slowed down, since the size of the G-Nodes working pool will be reduced by the size of the local working pool of the crashed Executor. However, all the G-Nodes have still the possibility of being selected by the remaining Executors, so the application execution can continue without major problems.

Finally, the last case occurs when the Supervisor process crashes. In this case, the application cannot continue its execution, and has to be prematurely terminated. In order to avoid that the computation has to be restarted from the beginning, G-Net adopts a checkpointing strategy in which periodically its saves the status of its computation. Therefore, when a premature termination occurs, the computation can be restarted from the last checkpoint.

4 Experimental Evaluation

In this section we present some experimental results concerning the performance of G-Net on the *Mutagenesis* problem [9], for which the task consists in learning rules for discriminating substances having cancerogenetic properties on the basis of their chemical structure. This task is a specific instance of the more general problem called *Predictive Toxicology Evaluation*, recently proposed as a challenging Data Mining application [4], and is very hard since the rules are expressed using First Order Logics, and the complexity of matching formulas in First Order Logics limits the exploration capabilities of any induction system.

When assessing the performance of systems for classifiers induction, two different aspects must be considered. The first one is related to the quality of the classifiers that the system is able to find, that is expressed in terms of the number of disjuncts (rules) contained in the classifiers (the smaller the number of disjuncts, the better the classifiers) and of the relative error (misclassification of objects). In [3] we have shown that G-Net is able to find classifiers whose quality is significantly better than those generated by other systems representative of the state of the art.

The second one is instead related to the efficiency of the system, and is measured in terms of the time taken to find the above solution. The graph shown in Fig. 7 reports the speed-up obtained when running G-Net on a PC-NOW made up by 8 PentiumII 400 Mhz PCs, connected by a switched Fast Ethernet. In the above experiments we have kept fixed the number of Searchers

(2), while we have progressively increased that of Matchers (from 2 to 6). As

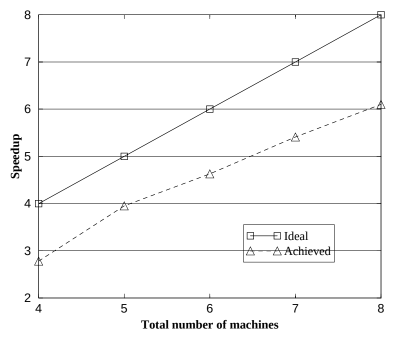

Fig. 7. Speedup achieved by G-Net on the *Mutagenesis* dataset.

can be seen by the graph the speed-up, although sub-optimal (because of the presence of serialization points due to coevolution), can be considered reasonable, meaning that G-Net's performance scaled reasonably well with the number of machines dedicated to the execution of Matchers. This is due to the fact that the evaluation task is significantly demanding in terms of computing power (since, as mentioned before, the Mutagenesis data set is described by means of First Order logics, and the structure of the rules implies that each of them must be evaluated 20^4 times in the worst case). Consequently, the evaluation time dominates over communication time, so an increase in the number of Matchers results in a significant reduction of the execution time.

5 Conclusions and Future Work

In this paper we have presented G-Net, a parallel system for the construction of classifiers, whose implementation has been targeted to PC-based NOWs. G-Net is able to deliver performance higher than other systems of the same type for what concerns both the quality of the found classifiers and the achieved speed-up. Thanks to the dynamic load balancing strategies it incorporates, and to the simple but effective fault tolerance capabilities provided by its stochastic nature, G-Net is able to profitably exploit the computing power provided by non-dedicated PC-based NOWs, with the consequence that it can be effectively used to address real problems characterized by a very high complexity. For instance, at the moment G-Net is being used to solve molecular biology problems that, in order to be solved, required a very high amount of computing power.

Although G-Net can be considered a mature system, several evolution directions, concerning both the algorithm and its PC-NOW implementation, are being considered at the moment. For what concerns the algorithm, we are developing a version of G-Net that extends the numerical capabilities already present, in order to allow it to deal with a broader class of problems. As for the architectural evolutions, for the near future it is planned a G-Net implementation based on MPI rather than on PVM, in order to achieve an even greater portability.

References

1. Anglano, C., Giordana, A., and Lo Bello, G. High Performance Knowledge Extraction from Data on PC-based Networks of Workstations. Tech. rep. Available from http://www.di.unito.it/˜mino/papers.html.
2. Anglano, C., Giordana, A., Lo Bello, G., and Saitta, L. A Network Genetic Algorithm for Concept Learning. In *Proceedings of the 7th International Conference on Genetic Algorithms* (East Lansing, MI, July 1997), Morgan Kaufman.
3. Anglano, C., Giordana, A., Lo Bello, G., and Saitta, L. An Experimental Evaluation of Coevolutive Concept Learning. In *Proceedings of the 15th International Conference on Machine Learning* (Madison, WI, July 1998), Morgan Kaufman.
4. Dehaspe, L., Toivonen, H., and King, R. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knoweldge Discovery and Data Mining* (New York, NY, August 1998), AAAI Press, pp. 30–36.
5. Fayyad, U. Data Mining and Knowledge Discovery: Making Sense out of Data. *IEEE Expert 11*, 5 (1996), 20–25. October 1996.
6. Goldberg, D. *Genetic Algorithms.* Addison-Wesley, Reading, MA, 1989.
7. Harchol-Balter, M., Crovella, M., and Murta, C. On Choosing a Task Assignment Policy for a Distributed Server System. In *Proceedings of the 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (1998), Springer-Verlag.
8. Hui, C., and Chanson, S. Theoretical Analysis of the Heterogeneous Dynamic Load-Balancing Problem Using a Hydrodynamic Approach. *Journal of Parallel and Distributed Computing 43*, 2 (June 1997).
9. King, R., Srinivasan, A., and Stenberg, M. Relating chemical activity to structure: an examination of ILP successes. *New Generation Computing 13* (1995).
10. Piatetsky-Shapiro, G., and Frawley, W., Eds. *Knowledge Discovery in Databases.* AAAI Press / The MIT Press, Menlo Park, CA, 1991.
11. Potter, M. *The design and analysis of a computational model of cooperative coevolution.* PhD thesis, George Mason University, Fairfax, VA, 1997.
12. Potter, M., De Jong, K., and Grefenstette, J. A coevolutionary approach to learning sequential decision rules. In *6th Int. Conf. on Genetic Algorithms* (Pittsburgh, PA, 1995), Morgan Kaufmann, pp. 366–372.
13. Sunderam, V. S., Geist, G. A., Dongarra, J., and Manchek, R. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing 20*, 4 (April 1994), 531–45.
14. Weber, R. On the Optimal Assignment of Customers to Parallel Servers. *Journal of Applied Probability 15* (1978), 406–413.