Microprocessing and Microprogramming 34 (1992) 147–152
North-Holland



# USING TRANSPUTERS TO INCREASE SPEED AND FLEXIBILITY OF GENETICS-BASED MACHINE LEARNING SYSTEMS

Marco Dorigo
Politecnico di Milano - Dipartimento di Elettronica
Piazza Leonardo da Vinci 32 - 20133 Milano - Italy
E-mail: dorigo@ipmel2.elet.polimi.it
Tel. +39-2-2399-3622
Fax. +39-2-2399-3411

## ABSTRACT

We implemented a distributed environment for machine learning experimentation on a transputer network. The system can be used by a researcher to build modular and efficient learning systems. The algorithms composing the basic structure of the implementation are the genetic algorithm, the bucket brigade algorithm and the inferential engine. We present a parallel version of these algorithms and call it low-level parallelism. Compared to the standard sequential version of the same algorithms, low-level parallelism gives us an increase in performance. To provide the learning system designer with a higher level of flexibility than currently available with standard systems, we also implemented high-level parallelism: subsets of the transputer network can be allocated to different learning systems. In this way a complex learning problem can be decomposed in many simpler problems, each one mapped on a single (possibly low-level parallel) learning system.

KEYWORDS Parallel genetic algorithms Implementation on transputers Genetics-based machine learning

## 1. INTRODUCTION

A major goal of artificial intelligence research is to give computers learning capabilities. A first step to solve this very difficult goal could be to implement systems with adaptive capabilities, i.e. systems that change their behaviour according to the environmental situation in which they operate. The problems raised by the design of such systems are faced by researchers in machine learning. Genetics-based machine learning is a recent approach to machine learning problems, and the comprehension of this model is, as it happens with neural networks, largely dependent on simulations. The present understanding indicates that even the solution of simple learning problems requires the use of large sets of rules, and suggests that real world application will be possible only exploiting the power of parallel computers.
We have therefore developed a parallel distributed system that can be used as a tool to build genetics-based machine learning - GBML - systems. A parallel implementation of a GBML system on the Connection Machine has been proposed by Robertson [1]. That work demonstrates the power of such a solution, but still retains, we think, a basic limitation: as the Connection Machine is a SIMD architecture the resulting implementation is only a more powerful but still classic GBML system. To implement our system we have used a transputer net that, because of its MIMD architecture, permits the presence of many simultaneously active control flows operating on different data sets. This architectural organization allowed us to distinguish between low- and high-level software forms of parallelism in a way that directly maps on the hardware architecture.

## 2. GENETICS-BASED MACHINE LEARNING SYSTEMS

GBML systems are a class of adaptive systems that have recently raised the interest of the artificial intelligence community. They are composed of:

- A performance system, which contains the system knowledge-base, expressed as a set of production rules, and the inferential engine that allows many rules to fire concurrently.
- A rule discovery system that search for new rules whenever changes in the environment require the system to adapt.
- A rule evaluation system, whose task is to rate rules according to their usefulness.

The system as a whole is interacting with the environment: it senses environmental changes through incoming messages, acts on it through actions and receives rewards or punishments as feedback for performed actions. We give now a brief description of the three systems.

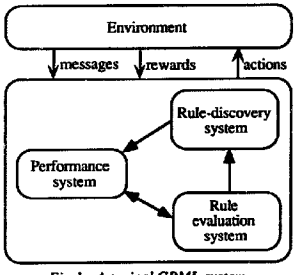


*Fig.1 - A typical GBML system*

### 2.1 The performance system

The main blocks composing the performance system are (see Fig.2):

- a set of rules, called classifiers, with a condiction→action format
- a message list of dimension k+e, used to collect messages sent from classifiers and from the environment to other classifiers, where k is the number of positions for internal messages, i.e. messages sent by classifiers, and e is the number of positions for environmental messages, i.e. messages coming from external environment
- an input and an output interface with the environment (detectors and effectors) to receive/send messages from/to the environment
- a pattern-matching and a conflict-resolution subsystem that identify which rules are active in each cycle and which of them will fire

The performance system algorithm is:

0 • Initialize the system (create a random set of rules).
1 • Read environmental messages and append them to the message list.
2 • Set to status *active* each classifier which has both conditions matched by messages on the message list and then clear the message list.
3 • If number of active rules ≤ k (with k dimension of the message list),
  **then** append their messages to the message list;
  **else** call the conflict resolution module, which takes as input the m competing rules and returns the k rules that have the right to post their message; then append the messages of the k winning rules to the message list.
4 • Set the status of all classifiers to *not-active*.
5 • Repeat from step 1.

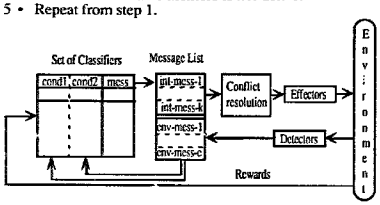


*Fig.2 - The performance system*

The conflict resolution module requires, to work properly, some knowledge about the usefulness of competing rules; only using this information it can decide which rules to fire - and which action to choose in case of proposed inconsistent actions (e.g. "go right" and "go left"). It is the necessity of this information that justifies the introduction of the apportionment of credit algorithm, whose task is to rate rules according to their perceived usefulness.

### 2.2 The rule evaluation system

The main task of the rule evaluation algorithm is to classify rules by their usefulness. The most known and used algorithm is the Bucket Brigade algorithm. In words, it works as follows: a time varying real value called *strength* is associated to every classifier C. At time zero each classifier has the same strength. When an external classifier causes an action on the environment a payoff is generated whose value is dependent on how good the performed action was with respect to the system goal. This reward is then transmitted backward to internal classifiers that caused the external classifier to fire. The backward transmission mechanism causes the classifiers strength to change in time and to reflect their importance for the system performance (with respect to the system goal).
Because of space constraints the bucket brigade algorithm will be presented only in the parallel version.

### 2.3 The Genetic Algorithm

Genetic algorithms are a class of stochastic algorithms which has been successfully used both as an optimization device and as a rule-discovery mechanism [2], [3]. They work modifying a population - set - of solutions (in GBML a solution is a classifier) to a given problem. Solutions are properly codified and a function, called fitness function, is defined to relate solutions to performance (the value returned by this function is a measure of the solution quality). In genetics-based machine learning the fitness of a classifier is given by its usefulness as measured by the apportionment of credit algorithm.
As it uses classifiers strength as a measure of fitness, the genetic algorithm can be usefully applied to the set of classifiers only when the bucket brigade algorithm has reached steady-state, i.e. when a rule strength accurately reflects its usefulness. Therefore, it is applied very seldom, usually every 1000÷10000 bucket brigade steps. In the following we report the steps of a single GA call.

0 • Take the set of classifiers as initial population P.
1.1 • Rank individuals of P in decreasing fitness order using the strength associated to every classifier as a measure of fitness.
1.2 • Choose 2k individuals to be replaced among low ranked - useless - ones.
1.3 • Choose 2k individuals to be replicated among high ranked - useful - ones.
2.1 • Mate the individuals selected at step 1.3, so to get k pairs of useful rules.
2.2 • Apply the crossover operator to each of the k pairs.
2.3 • Apply the mutation operator to each of the 2k individuals resulting from step 2.2.
3 • Replace with the new generated 2k individuals the 2k useless individuals chosen at step 1.2.

## 3. HOW TO APPLY PARALLELISM?

Let's now underline the characteristics of GBML systems that make their parallelization easy. We said that the activation of rules is, at each cycle, based on the set of messages composing the message list. It is then a common situation to have many classifiers simultaneously activated: therefore there is intrinsic concurrency among the rules. This (strong) possibility of parallel activation has the effect to require a large computing power for the matching and competition steps; nonetheless it is clear that these computations can be distributed to a group of processing units, working in parallel. Each processor can take care of matching the elements of the message list with a restricted subset of classifiers.
To be more explicit, let us reconsider the performance algorithm. We can think of it as being executed by four distinct processes (see Fig.3), each one taking care of different operations:

- **DTprocess** (DeTector process): input interface, converting changes in the environmental state into messages to be appended to the message list (ML).
- **EFprocess** (EFfector process): output interface, converting classifier messages into environmental actions.
- **MLprocess** (MessageList process): central manager for operations regarding the message list; tasks of this process are:
  - to append to ML messages coming from **DTprocess** or to send to **EFprocess** messages to interpreted as actions;
  - to send ML to CFprocess;
  - to choose which rules should be replicated or discarded;
  - to apply crossover, mutation or other genetic operators.
- **CFprocess** (ClassiFiers process): classifier-population manager: this process matches each message in ML with the condition part of the various classifiers; it also updates the strength of each rule distributing payments and rewards.

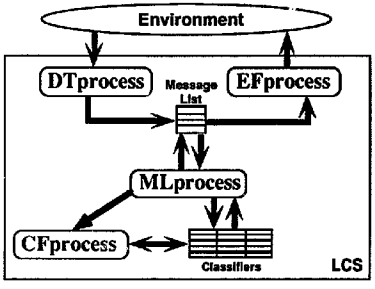


*Fig.3 - Concurrent processes in a standard GBML system*

If we consider the activities of matching and message-production, we see that they can be executed on each classifier independently. So, it is natural to split **CFprocess** into an array of sub-processes **CFprocess.1**, **CFprocess.2**, ... , **CFprocess.n**, each one taking care of a fraction (1/n) of the classifier set (CF). The higher goes n, the more intensive is the concurrency. When **n** is equal to the cardinality of CF, each **CFprocess.***i* manages a single classifier: this is the typical Connection Machine version of a concurrent GBML system. In our transputer-based implementation we allocated about 100-500 rules to each processor [4]. The set of **CFprocesses** can be organized in a hierarchical structure such as, for example, a tree (see Fig.4) or a toroidal grid. The chosen structure deeply influences the distribution of computational loads [5].
Similar remarks hold for the parallelization of the genetic algorithms. We saw in a preceding section how the genetic algorithm works when used in the GBML context. About that algorithm we remark the following aspects:
Step 1 (ranking individuals of P by their fitness [Step 1.1] and choosing which individuals are to be replaced [Step 1.2] and which are to be replicated [Step 1.3]) may be seen as a competition, which can be therefore distributed over the processor network by a "hierarchical gathering and broadcasting" mechanism similar to the one we used for propagating ML to the array of **CFprocess.***i*'s.
Step 2.1 (mating rules for crossover) is hard to parallelize, as it requires a central management unit. Fortunately, the number of pairs involved is usually small and concurrency seems to be unnecessary. Step 2.2 (applying crossover operator) can be parallelized, even if many communications are required.
Step 2.3 (applying mutation operator) and Step 3 (replacement of new generated individuals) are typical examples of local data processing, extremely suited to concurrent distribution.

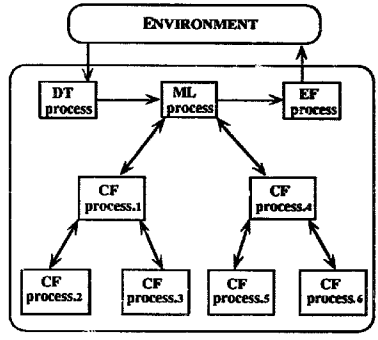


*Fig.4 - The parallel version of a GBML system*

## 4. THE EXPRESS COMMUNICATION SYSTEM

To implement our system we have used Express [6], a software tool written in 3L parallel C [7] and running on transputers, that offers facilities related to data exchange, processes allocation, load balancing, etc.
The typical read/write functions have in the Express communication system the following form:
**exread**(address_of_receiving_buffer,message_size, identifier_of_source_processor, message_tag)
**exwrite**(address_of_buffer_to_send, message_size, identifier_of_destination_processor, message_tag)
The message_tag field associates a numerical tag to each communication, characteristic for each transmitted package of data, both for simple (integer, float, double) or structured (arrays, structures) data. This way it is possible to think of a communication between two processes as of a simple pair of statements: a call to **exwrite**(A_protocol) on the sender, and a corresponding call to exread(A_protocol) on the receiver.

## 5. LOW-LEVEL PARALLELIZATION

On the basis of what we said in previous sections, supported by a performance analysis on different models of parallelized GBML systems [5], we decided to implement our system as a centrally-driven, distributed algorithm. In this way we see the whole system as distributed over a net composed of one leader (MLprocess) and an array of slaves (CFprocess.i). To optimize the CPU usage, we allocated on the node hosting MLprocess also one of the elements of the CFprocess.i array. This way, while waiting for its net

of "slaves" to give their answers, the MLprocess node can process itself a part of the classifiers population. The parallel GBML system is then composed of two different programs, Root and Net, interfaced with a third program Host, implementing environment and I/O processes (DTprocess and EFprocess). Consider the case in which we have m+1 processors: then the Root program is downloaded to one processor while the Net program is allocated to each of the remaining m nodes. Each node, be it running a Root or a Net program, takes care of a fraction of the global classifiers population. The set of "slave" processes may be organized in an arbitrary fashion, regardless of the underlying hardware architecture. Of course, the greater the correspondence between the software hierarchy and the transputer physical structure, the more efficient the communication system will be. An example: with an underlying pipeline of 9 transputers (Fig.5.1), our system allows the distribution of a learning system both on a binary tree hierarchy (Fig.5.2), and on a double-branch structure (Fig.5.3). But the former results in longer, non-hardware paths for broadcasting data from the Root to each Net program and backward. These paths have a strong influence upon the execution time, as the parallelized learning system works by distributing data over the processors net and by gathering results from it, along the same paths. In fact, all the data structures are distributed through the net in a hierarchical way, each node communicating only with nodes of immediately higher and/or lower level (if any). This means, for example (see Fig.5.4), that node Y will receive data from node X (when broadcasting from the Root node) and node Z (when gathering data towards the Root node). And, obviously, node Y will send data towards node Z (when broadcasting) and towards node X (when gathering).

The basic cycle of the Host program is:

0 • Generate an initial environmental state.
1 • Code the environmental messages into proper messages (DTprotocol).
2 • Send these messages to the leader process, i.e. the Root node: **exwrite(*DTprotocol*)**.
3 • Receive from the Root node the answer, directed to the effectors, by TEprotocol (To Effector protocol): **exread(*TEprotocol*)**.
4 • Decode the messages into environmental actions.
5 • Competition step: decide which action is to be performed on the environment, choosing among (feasible) suggested actions.
6 • Perform the selected action and receive the reward (or punishment).
7 • Send to the Root node the environmental reward through FEprotocol (From Effector protocol): **exwrite(*FEprotocol*)**.
8 • If EndTest = True then Stop else Goto Step 1.

The basic cycle of the Root program is:

0 • Generate an initial population. Set time t=0.
1 • Receive an environmental message from the Host node: **exread(*DTprotocol*)**.
and put it into a Message List structure (MLprotocol).
2 • Distribute MLprotocol towards the neighbouring Net nodes (if any): **exwrite(*MLprotocol*)**.
3 • Operate the matching phase.
4 • Set up an internal competition among bidding classifiers, resulting in a list of winning rules (WMprotocol: Winning Messages protocol); other lists of bidding classifiers may come from connected Net nodes by calls to: **exread(*WMprotocol*)**.
and are then set into competition with the local list; the result of this merging operation is a final list of classifiers that will append their messages on the new message list to be used in cycle t+1.
5 • Choose which messages are directed to the effectors, build a structure TEprotocol out of them, and send it to the Host node: **exwrite(*TEprotocol*)**.
6 • Receive a reward or a punishment from the environment, via a communication with the Host node. **exread(*FEprotocol*)**
and put this data into the MLprotocol.
7 • Receive an environmental message from the Host node: **exread(*DTprotocol*)**
and add it to the Message List structure (MLprotocol).
8 • Distribute MLprotocol towards the neighbouring Net nodes (if any): **exwrite(*MLprotocol*)**.
9 • Update the values of strength of the classifiers allocated to this node, using the data contained into the MLprotocol structure.
10 • If EndTest = True then Stop else t=t+1 and Goto Step 3.

The basic cycle of the Net program is:

0 • Generate an initial population. Set time t = 0.
1 • Receive the Message List structure, either directly from the Root node, or from an upper level Net node: **exread(*MLprotocol*)**.
2 • Distribute MLprotocol towards the neighbouring Net nodes (if any) of a lower level: **exwrite(*MLprotocol*)**.
3 • Operate the matching phase.
4 • Internal competition among bidding classifiers, resulting in a list of winning rules (WMprotocol: Winning Messages protocol); other lists of bidding classifiers may come from connected Net nodes of a lower level: **exread(*WMprotocol*)**
and are set into competition with the local list; the result of this merging operation is a list of classifiers sent either directly to the Root node, or to upper level Net nodes: **exwrite(*WMprotocol*)**.
5 • Receive the Message List structure, either directly from the Root node, or from an upper level Net node: **exread(*MLprotocol*)**.
6 • Distribute ***MLprotocol*** towards the neighbouring Net nodes (if any) of a lower level: **exwrite(*MLprotocol*)**.
7 • Update the values of strength of the classifiers allocated to this node, using the data contained into the MLprotocol structure.
8 • If EndTest = True then Stop else t=t+1 and Goto Step 3.

The parallel genetic algorithm, not reported here because of space constraints (see [4]), is organized in the same way. We insert it into the basic cycle after the EndTest step (step 10 in Root and step 8 in Net), i.e. before beginning any change in the population fitness. The genetic algorithm is distributed on m+1 processes, as in the performance algorithm case. We therefore built the implementation of the parallel GA by means of two programs, a "leader" and a "slave", which have been inserted inside Root and Net.

Also in this case, we organized the Net "slaves" hierarchically, obtaining a "flow" of ***GAprotocol*** data structures similar to the one we used for ***WMprotocol*** in the basic cycle: that is, we broadcast and gather data from the Root node to the Net nodes and back.

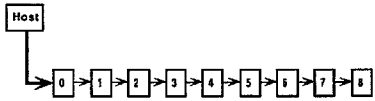


*Fig.5.1 - Pipeline of transputers: hardware structure*

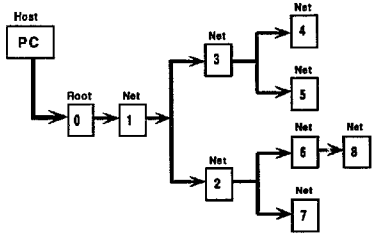


*Fig.5.2 - A parallel learning system with binary-tree structure*

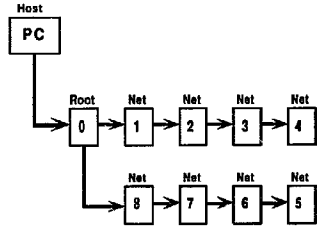


*Fig.5.3 - A parallel learning system with double-branch structure*

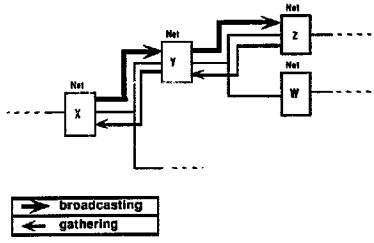


*Fig.5.4 - Broadcasting and gathering data in the parallel system*

## 6. HIGH-LEVEL PARALLELIZATION

We have shown how to parallelize a single GBML system, in order to obtain computing speed improvements. What we did was only to build a parallel version of the GBML system, with no significant difference with the sequential model, except for average execution times.
This approach shows its weakness when a GBML system is applied to multi-goal problems - as unluckily seems to be the case in most of real problems. To solve multi-objective tasks is hard because they don't have explicit mechanisms to handle different sets of rules (where each set is dedicated to the solution of a different goal). For these reason, a low-level parallelized GBML system, though faster and capable of managing larger sets of rules, seems still unable to be of practical use. Moreover, scalability problems arise in a low-level parallelized system: distributing a GBML system on larger processor networks makes the communication load grow fast, obtaining less and less in computing speed. Adding processors to the existing network is therefore less and less effective, tending to an asymptotical limit.
A better way to deal with complex problems could be to code them as a set of easier subproblems, each one to be solved by a smaller GBML system.
Therefore, we have partitioned the processor network into subsets, each having its own size and topology. Every subset is allocated a single GBML system, distributed by a low-level parallelization, whenever the number of nodes used by each single system is greater than one (see Fig.6).

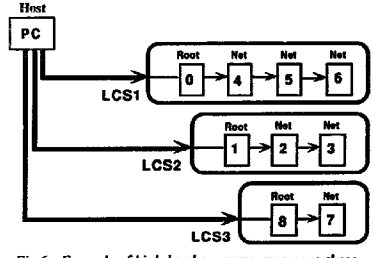


*Fig.6 - Example of high-level concurrency among three parallel cooperating learning systems*

Each of these systems learns to solve a specific subgoal, depending on the inputs it receives: each learning system perceives the external environment by its own detectors while the output interface is obviously common to all GBML systems, thus requiring some type of interaction among systems proposing actions.
We give an example of the kind of flexibility provided by high-level parallelism in designing a learning system [8]. Consider the following learning task: a simple autonomous robot has to learn to trace a light source and at the same time should learn to avoid heat sources; we can give the first task to the learning system $LS_1$ and the second task to the learning system $LS_2$. Moreover, as these tasks are simple components of a more complex coordination task (to avoid dangerous objects while tracing the light source), we can use a third learning system $LS_3$ that has as its goal the coordination of these activities (i.e. how to behave in con-

flicting situations, e.g. when the light source goes too near to a heat source for the system to continue to follow it without being injured).

We are now ready to present the algorithm for concurrent GBML systems, noting that the flow of data among different learning systems is restricted to communications among Root nodes and the Host node: each GBML system can be distributed over a transputer subnet, with a low-level parallelization as described in the previous section; but from the point of view of the high level parallelization, a learning system - LS - will operate in the same way, be its Root node processing data on its own, or with a subnet of underlying Net slaves. The resulting system then works as follows:

1 • each $LS_i$ receives from the environment its own input messages: **exread (*DTprotocol*_i)**.
2 • by processing them each $LS_i$ deduces an action to be performed in the environment (usually a different one for each $LS_i$).
3 • proposed actions, with their associated bids, are communicated to the Host node:
**exwrite (*TEprotocol*_i)**.
4 • the Host node takes care of informing the coordinator ($LS_3$ in the example) about the data received at step 3; this is done by sending a new type of protocol to the LS implementing the coordinator:
**exwrite (*DTprotocol*_Coordinator)**.
5 • the coordinator decides a resulting action, which is communicated to the Host node by sending protocols of data from the Root node of the coordinating LS ($LS_3$) to the Host node itself which receives them by a call to: **exread (*TEprotocol*_Coordinator)**.
6 • the action is performed by means of the output interface and rewards are given to the Root nodes:
**exwrite (*FEprotocol*_Coordinator)**
**exwrite (*FEprotocol*_i)**.
7 • If EndTest = True then Stop else t=t+1 and Goto Step 1.

Building a concurrent network of (if necessary parallelized) LSs requires then only three type of modules: Host (Environment, Input/Output interfaces, simulators and connections between *coordinator* and *workers*), Root (**MLprocess/GAprocess**), and Net (**CFprocess.*i***). These units can be arbitrarily connected to obtain the desired network.

## 7. CONCLUSIONS

In this paper we have presented a parallel architecture implementing a general purpose genetics-based machine learning system. By low-level parallelism we have enhanced the computational speed, by high-level parallelism the system overall flexibility.

We are using it with the following results:

- the time required to design a GBML system has dropped dramatically;
- the computational power of each module can be defined by the machine learning researcher according to the task complexity.

## REFERENCES

[1] Robertson,G.G., "Parallel Implementation of Genetic Algorithms in a Classifier System", Proceedings of the Second International Conference on Genetic Algorithms, July 28-31 1987, Lawrence Erlbaum.
[2] Holland,J.H., "Adaptation in natural and artificial systems", Ann Arbor: The University of Michigan Press, 1975.
[3] Goldberg,D.E. "Genetic Algorithms in Search, Optimization & Machine Learning".Addison-Wesley,1989.
[4] Dorigo,M., Sirtori,E., "A Parallel Distributed Environment for Genetics-based Machine Learning", Technical Report No. 91-015, Politecnico di Milano, Italy.
[5] Camilli,A., Di Meglio,R., "Classifiers systems in massively parallel architectures", Master thesis, University of Pisa - Italy, 1990.
[6] Express 3.0 User's Guide, ParaSoft Corporation, 2500 Foothill Blvd., Pasadena, CA 91107, 1990.
[7] 3L Parallel C User's Guide, 3L Ltd, 1988.
[8] Dorigo,M., Schnepf,U. "Organisation of Robot Behaviour Through Genetic Learning Processes", to appear in the proceedings of the Fifth IEEE International Conference on Advanced Robotics - June 20-22, 1991 - Pisa - Italy.