Abstract—In this paper new parallel versions of some Estimation of Distribution Algorithms are proposed. Particularly, the learning phase of the probability distribution from the selected individuals has been parallelized for different algorithms that use probabilistic graphical models -Bayesian networks and Gaussian networks-. In discrete domains, we explain the parallelization of $EBN_{BIC}$ and $EBN_{PC}$ algorithms, while in continuous domains the selected algorithms are $EGN_{BIC}$ and $EGN_{EE}$. This new implementation allows execution of the algorithms in several architectures. The parallelization was developed mixing two different options: Message Passing Interface and threads. Some experiments were executed for all the algorithms in a cluster of computers showing important performance gains comparing with the sequential versions.
Parallel Estimation of Distribution Algorithms: New Approaches

Alexander Mendiburu, Department of Computer Architecture and Technology
Jose A. Lozano, Department of Computer Science and Artificial Intelligence
Jose Miguel, Department of Computer Architecture and Technology
The University of the Basque Country
San Sebastian, Spain


I. INTRODUCTION

In recent years great improvements have been made in the development and use of heuristic techniques for optimization. Different algorithms have been proposed and applied to optimization problems, usually with very good results. However, there is still a significant drawback which-in most situations-makes the use of such algorithms prohibitive in real environments: computation time. That is, the algorithms are able to solve a complex problem but not as quickly as we need, with execution times lasting hours or even days. One approach to reduce computation time is the use of parallelization techniques.

Some of the algorithms that have received much attention from researchers are those included in the field of evolutionary computation. Such techniques include Genetic Algorithms (GAs), Evolutionary Strategies, Evolutionary Programming and Genetic Programming.

In the literature a number of interesting parallel proposals have been made in the field of evolutionary computation. Efforts in the design of parallel evolutionary algorithms have focused on two areas. On the one hand, looking for algorithms where many populations evolve in parallel, and on the other, concentrating on parallelizing computation of the objective function. A good summary of different ideas for designing parallel GAs can be found in [1].

Recently, a new group of algorithms has been proposed in the evolutionary computation field: Estimation of Distribution Algorithms (EDAs) [2]. Like most evolutionary computation heuristics, EDAs maintain at each step a population of individuals, but, instead of using crossover and mutation operators to evolve the population, EDAs learn a probability distribution from the set of selected individuals and sample this distribution to obtain the new population.

However, in this approach, it is the estimation of the joint probability distribution that causes the bottleneck. To avoid this problem, several authors have proposed simplifying assumptions on the probability model to be learnt at each step.

These assumptions concern the relations between the variables that the probability model takes into account. EDAs range from those that assume that all variables are indepen-
dent, to those that assume second order statistics and those that consider unrestricted models.

Of these, the ones that obtain the best results are those that use probability models with unrestricted dependencies [2]. In these algorithms, the probability distribution is codified by a probabilistic graphical model. Excluding the computation time of the objective function, the main computational cost of the algorithms is the time consumed in learning the probabilistic graphical model at each step. In fact, it takes 50-99% of the total execution time of the algorithm. Therefore, our effort focuses on the parallelization of this learning process, proposing different solutions for certain EDAs that use probabilistic graphical models to learn the joint probability distribution of the selected individuals.

In the field of EDAs, previous parallel implementations can be found for some algorithms. In [3] two different parallel proposals for an algorithm that uses probabilistic graphical models in the combinatorial field (EBN_A_{BIC}) are proposed. These implementations use threads over a multiprocessor with shared memory, so that all data structures are shared between the different processes, with each of them computing a piece of work.

Also interesting is the work done in [4] and [5] where two different parallel versions of the Bayesian Optimization Algorithm (BOA) [6] are proposed using a pipelined parallel architecture and clusters of computers respectively. Recently, in [7] the parallelization of the learning of decision trees using multithreaded techniques has been proposed.

Another way of parallelization can be found in [8] where a basic framework that facilitates the development of new multiple-deme parallel EDAs is presented.

In this paper we propose new parallel approaches of certain EDAs that use probabilistic graphical models to learn the probability distribution of the selected individuals in both discrete and continuous domains. Specifically, we have chosen one algorithm for each different method of learning a probabilistic graphical model used in EDAs.

In our versions we mix two parallel techniques: Message Passing Interface (MPI) [9] and threads [10]. Consequently, unlike the proposal of [3], the parallel solution can be used in a multiprocessor computer (communications are made using shared memory), in a cluster (where each computer communicates with others using messages) or any combination of both scenarios. Also, our work extends [4] and [5] allowing the learning of the Bayesian network without taking into account order restrictions, even though the objective function is not parallelized. Finally parallel approaches for algorithms of the continuous domains are also proposed, particularly: EBN_{A_{BIC}} and EBN_{A_{PC}} algorithms for discrete domains and EGN_{A_{BIC}} and EGN_{A_{EE}} for continuous domains.

A description of EDAs can be seen in section II, where the main characteristics of the algorithms will be explained. In section III general considerations for the parallel proposals are presented. In sections IV and V the parallel solution for the four different algorithms is explained following this schema: a brief explanation of the algorithm, parallel implementation(s) and an evaluation of the results obtained. Finally, the conclusions obtained and some ideas for future work are presented in section VI.

II. ESTIMATION OF DISTRIBUTION ALGORITHMS

EDAs were introduced in the field of evolutionary computation in [11], although similar approaches can be found in [12]. In EDAs there are neither crossover nor mutation operators. Instead, the new population of individuals is sampled from a probability distribution, which is estimated from a database containing the selected individuals from the previous generation. Thus, the interrelations-building blocks-between the different variables representing the individuals are explicitly expressed through the joint probability distribution associated
Fig. 1. EDA approach to optimization

with the individuals selected at each generation. In order to understand the behavior of this heuristic, a common model for all EDAs is introduced:

1) Generate the first population $D_0$ of $M$ individuals and evaluate each of them. Usually this generation is made supposing a uniform distribution on each variable.

2) $N$ individuals are selected from the set of $M$ following a given selection method.

3) The $n$ (size of the individual) dimensional probability model that shows the interdependencies among the variables best is induced.

4) Finally, a new population of $M$ individuals is generated based on the sampling of the probability distribution learnt in the previous step.

Steps 2, 3 and 4 are repeated until some stop criterion is met (e.g., a maximum number of generations, a homogeneous population or no improvement after a certain number of generations). A graphic schema of this process (for $n = 4$) can be seen in Fig. 1.

The probabilistic graphical model learnt at each step has an important influence on the behavior of the EDA algorithm (computing times and obtained results). Therefore, the algorithms are classified below taking into account the complexity of this probability model and, particularly, the dependencies they consider:

- **Without dependencies:** It is assumed that the $n$-dimensional joint probability distribution factorizes as a product of $n$ univariate and independent probability distributions. Algorithms that use this model are, for example, $UMDA$ [13] or $PBIL_c$ [14].

- **Bivariate dependencies:** Estimation of the joint probability can be done quickly without assuming independence between the variables by taking into consideration dependencies between pairs of variables. This group includes $MIMIC$ [15], $MIMIC_c$ ([16], [17]).

- **Multiple dependencies:** All possible dependencies among the variables are considered without taking into account required complexity. $EBNA_{BC}$ ([18], [19]), $BOA$ ([6], [20], [21], [22]) or $EGNA_{EE}$ ([16], [17]) are some of the algorithms that belong to this group.

For detailed information about the characteristics and different algorithms that constitute the family of EDAs, see [2] and [23].

The algorithms of the last group, defined in the previous classification as multiple dependencies, use different paradigms to codify the probabilistic model. Particularly, we are interested in those that use probabilistic graphical models -based on Bayesian or Gaussian networks- and therefore we will explain briefly the main characteristics of both type of networks.

1) **Bayesian Networks:** Formally, a Bayesian network [24] over a domain $\mathbf{X} = (X_1, \ldots, X_n)$ is a pair $(S, \Theta)$ representing a graphical factorization of a probability distribution. The structure $S$ is a directed acyclic graph which reflects the set of conditional (in)dependencies between the variables. The factorization of the probability distribution is codified by $\Theta$: 
In the following section (discrete domain), we propose the parallelization of two EDAs that use these methods: \( EBN_{ABIC} \) and \( EBN_{APC} \), respectively.

2) Gaussian Networks: The other particular case of probabilistic graphical models to be considered in this work is when each variable \( X_i \in \mathbf{X} \) is continuous and each local density function is the linear-regression model:

\[
f(x_i \mid \mathbf{pa}_i, \theta_i) \equiv \mathcal{N}(x_i; m_i + \sum_{j \in \mathbf{pa}_i} b_{ji}(x_j - m_j), v_i)
\]  

where \( \mathcal{N}(\mu; \sigma^2) \) is a univariate normal distribution with mean \( \mu \) and variance \( \sigma^2 \). Given this form, a missing arc from \( X_j \) to \( X_i \) implies that \( b_{ji} = 0 \) in the former linear-regression model. The local parameters are given by \( \theta_i = (m_i, b_i, v_i) \), where \( b_i = (b_{i1}, \ldots, b_{i|\mathbf{pa}_i|}) \) is a column vector. We call a probabilistic graphical model constructed with these local density functions a Gaussian network after [25].

The interpretation of the components of the local parameters is as follows: \( m_i \) is the unconditional mean of \( X_i \), \( v_i \) is the conditional variance of \( X_i \) given \( \mathbf{pa}_i \), and \( b_{ji} \) is a linear coefficient reflecting the strength of the relationship between \( X_j \) and \( X_i \).

As can be seen, there is a bijection between Gaussian networks and a \( n \)-dimensional multivariate normal distribution, whose density function can be written as:

\[
f(x) \equiv \mathcal{N}(x; \mu, \Sigma) \equiv (2\pi)^{-\frac{n}{2}} |\Sigma|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}
\]  

where \( \mu \) is the vector of means, \( \Sigma \) is an \( n \times n \) covariance matrix, and \(|\Sigma|\) denotes the determinant of \( \Sigma \). The inverse of this matrix, \( W = \Sigma^{-1} \), whose elements are denoted by \( w_{ij} \), is referred to as the precision matrix.

An example of a Gaussian network can be seen in Fig. 3.
different methods to complete Gaussian network learning. The ones we point out are “score + search” and “detect conditional (in)dependencies”. In sections V-A and V-B the parallelization of two algorithms that use these approaches are shown: $E G N A _ { B I C }$ ("score + search") and $E G N A _ { E E }$ (edge exclusion tests).

### III. General Considerations for Parallel Proposals

Before undertaking the implementation of a parallel solution for an existing sequential algorithm it is essential to study the structure and functionality of the sequential solution and find the parts amenable to parallelization. To this end, we have executed the sequential version of the algorithms with different problems and different individual sizes in order to measure the computation time that each part of the algorithm requires. As we have said in the introduction and can be seen in table I (showing the time required to complete the learning step in percentages), the most time-consuming process is the learning phase, which generally accounts for 50-99% of total execution time. Our effort will therefore be focused on understanding how the learning process proceeds and finding a good parallel approach for this phase.

Before explaining the implementation, two important concepts should be explained: MPI and Threads.

#### Message Passing Interface [9] provides a layer that can be used by application process to communicate over the network with processes that are physically executed in other computers. It was designed and standardized by the MPI forum, a group of academic and industrial experts for use in very different types of parallel computers. A wide set of procedures are available for managing, sending and receiving messages.

Concerning threads [10], most operating systems have thread support, that is, they include capabilities to spawn new processes (actually, threads) that share the same variable space. In this paradigm, a set of processes collaborating to perform a common task are implemented as threads that share the same memory space, so they can communicate directly using a set of global variables. The advantage of using this approach is that communication is fast and efficient, although the synchronization between threads has to be explicitly implemented.

With regard to implementations, a two-level Master-Slave scheme has been used. At the first level (communication between computers) the communication is carried out using MPI, where a computer acts as manager, coordinating communication with other computers (workers). These workers execute the orders sent by the manager and return the results to it. In some situations, the manager must wait for a long time.
time until all the workers finish their work, and this makes it possible for the manager to act also as a worker making use of this idle time. At the second level, we present another Master-Slave schema (inside each computer), where the MPI node (computer) is the master, and the workers are the multiple threads that can be created. This is a good choice when computers are multiprocessors. This schema is depicted in Fig. 4.

Implementations of the parallel algorithms were made (except for $EBN_A_{PC}$ algorithm) using both parallel techniques (MPI and threads). In many environments, machines available to run high CPU-consuming programs are multiprocessors, clusters of single-processor nodes or even clusters of multiprocessors. Our implementations can be easily adapted to any of these environments by selecting the right choice of mechanism for inter-node and intra-node communication and synchronization.

Finally, it should be noted that all the pseudocodes introduced in this paper for the different algorithms describe the master-slave communication for the first level (MPI communication between computers). The reason is that the schemes for the second level master-slave communication are similar to the first level (although there are no data send/receive operations because there is a single shared memory space).

It is important to remark that, throughout this paper, results for the parallel implementations are compared to an optimized sequential version of the programs, running on a single processor.

IV. DISCRETE DOMAIN

A. $EBN_{A_{BIC}}$ algorithm [18], [19]

1) Algorithm description: This algorithm maintains the common schema described for EDAs.

During the learning process, it uses a “score + search” approach to learn the structure of the probabilistic graphical model (Bayesian network). This method assigns a score to each possible Bayesian network structure as a measure of its performance compared to a data file of cases. To look for a structure that maximizes the given score it uses an algorithm that basically consists of adding or deleting edges to the existing Bayesian network.

$EBN_{A_{BIC}}$ uses the penalized maximum likelihood score denoted by $BIC$ (Bayesian Information Criterion) [26]. Given a structure $S$ and a dataset $D$ (set of selected individuals), the BIC score can be written as:

$$BIC(S, D) = \sum_{i=1}^{n} \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log \frac{N_{ijk}}{N_{ij}} - \frac{1}{2} \log N \sum_{i=1}^{n} q_i (r_i - 1) \quad (5)$$

where:

- $n$ is the number of variables of the Bayesian network (size of the individual).
- $r_i$ is the number of different values that variable $X_i$ can take.
- $q_i$ is the number of different values that the parent variables of $X_i$, $Pa_i$, can take.
- $N_{ij}$ is the number of individuals in $D$ in which variables $Pa_i$ take their $j^{th}$ value.
- $N_{ijk}$ is the number of individuals in $D$ in which variable $X_i$ takes its $k^{th}$ value and variables $Pa_i$ take their $j^{th}$ value.
An important property of this score is that it is decomposable. This means that the score can be calculated as the sum of the separate local BIC scores for the variables, that is, each variable $X_i$ has associated with it a local BIC score $(BIC(i, S, D))$: \[ BIC(S, D) = \sum_{i=1}^{n} BIC(i, S, D) \] \[ BIC(i, S, D) = \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log \frac{N_{ijk}}{N_{ij}} - \frac{1}{2} \log(N)q_i(r_i - 1) \] where $BIC(S, D)$ is the BIC score for the entire network $S$, and $BIC(i, S, D)$ is the BIC score for variable $X_i$. The structure search algorithm used in EBNA$_{BIC}$ is usually a hill-climbing greedy search algorithm. At each step, an exhaustive search is made through the set of possible arc modifications. An arc modification consists of adding or deleting an arc from the current structure $S$. The arc modification that maximizes the gain of the BIC score is used to update $S$, as long as it results in a DAG (Directed Acyclic Graph) structure (note that the structure of a Bayesian network must always be a DAG). This cycle continues until there is no arc modification that improves the score. It is important to bear in mind that if we update $S$ with the arc modification $(j, i)$, then only $BIC(i, S, D)$ needs to be recalculated.

The structural learning algorithm involves a sequence of actions that differs between the first step and all subsequent steps. In the first step, given a structure $S$ and a database $D$, the change in the BIC is calculated for each possible arc modification. Thus, we have to calculate $n(n - 1)$ terms as there are $n(n - 1)$ possible arc modifications. The arc modification that maximizes the gain of the BIC score, whilst maintaining the DAG structure, is applied to $S$. In remaining steps, only changes to the BIC due to arc modifications related to the variable $X_i$ (it is assumed that in the previous step, $S$ was updated with the arc modification $(j, i)$) need to be considered. Other arc modifications have not changed its value because of the decomposable property of the score. In this case, the number of terms to be calculated is $n - 2$.

We use four memory structures for this algorithm. A vector $BIC[i]$, $i = 1, 2, \ldots, n$, where $BIC[i]$ stores the local BIC score of the current structure associated with variable $X_i$. A structure $S[i]$, $i = 1, 2, \ldots, n$, with the DAG represented as adjacency lists, that is, $S[i]$ represents a list of the immediate successors of vertex $X_i$, a $n \times n$ matrix $G[i,j]$, $i, j = 1, \ldots, n$, where each $(j, i)$ entry represents the gain or loss in score associated with the arc modification $(j, i)$. Finally, a matrix $paths[i,j]$, $i,j = 1, 2, \ldots, n$, of dimension $n \times n$ that represents the number of paths between each pair of vertices (variables). This last structure is used to check if an arc modification produces a DAG structure. For instance, it is possible to add the arc $(j, i)$ to the structure if the number of paths between $i$ and $j$ is equal to 0, that is, $paths[i,j] = 0$.

A pseudocode for the sequential structural learning algorithm can be seen in Fig. 5.

2) Parallel approach. First version: This first version is a straight-forward parallelization of the sequential algorithm. That is, the only changes made in the code are the ones necessary to implement the parallel version maintaining sequential functionality.

As parallelization is proposed only for the learning phase,
all the other phases of the algorithm are carried out sequentially by the manager (master). During this learning phase, the manager sends different orders to the workers identifying the work to be done. To begin, the master sends some information to the workers (e.g., size of the individual, $D$ database, number of workers) and in the subsequent steps different orders are sent - regarding the computation of the decomposed BIC score and maintenance of the local $S$ structure. Workers need their own data structures - remember that they are executed in different machines - to store temporary results. Once initialization has been completed, workers wait for an order. Two different types of orders can be received: one, to compute the BIC score for their respective set of variables (as each worker has a unique identifier it is easy to divide the work that each worker must complete in terms of this identifier) and return the results to the manager; and the second type of order, to maintain the integrity of the $S$ local structure - each node addition or deletion means that the manager must notify the workers that the $(i,j)$ edge has changed.

This is mainly the MPI schema but, as mentioned at the beginning of this part, in this implementation we have also added thread programming techniques. When combined with MPI, a variable number of threads can be created at the beginning of the program execution and, when a BIC computation for a subset of variables must be executed in a worker, it can use the previously created threads. Each of them will independently compute a BIC value for a determined variable (continuing with another variable until all variables have been calculated). It can be seen that in every “MPI worker”, new workers (threads) are locally created, all working together to complete the work assigned to the “MPI worker”. As they use shared memory, all the changes are made in the same set of data structures, with the process being transparent for the MPI communication.

The pseudocode for this algorithm is shown in Fig. 6.

$EBNA_{BIC}$ Algorithm. First parallel version. Manager

Input: $D$, $S$, paths
Step 1. send $D$ to the workers and set the number of variables ($NSet$) to work with
Step 2. send “calculate BIC” order to the workers for each variable $i$ in $NSet$
Step 3. receive BIC results from workers
Step 4. for $i = 1, \ldots, n$ and $j = 1, \ldots, n$ $G[j, i] = 0$
Step 5. for $i = 1, \ldots, n$
send $i$, $BIC[i]$ to the workers
send “calculate $G[k, i]$” order to the workers for each variable $k$ in $NSet$
if ($k \neq i$) calculate $G[k, i]$
Step 6. receive from workers all the changes and update $G$
Step 7. if $G[j, i] > 0$
update $S$ with arc modification $(j, i)$
update paths
send “change arc $(j, i)$” order to the workers
else send workers “stop order” and stop
Step 8. if $G[j, i] > 0$
for each $r, s = 1, \ldots, n$ such that $paths[s, r] = 0$
Step 9. send “calculate $G[r, s]$” to the workers
Step 10. for each variable $k$ in $NSet$
if ($k \neq i$ and $k \neq j$) calculate $G[k, i]$
Step 11. receive from workers all the changes and update $G$
Step 12. go to Step 7

Fig. 6. Pseudocode for the parallel structural learning phase. First version of the $EBNA_{BIC}$ algorithm for the manager.

(manager) and Fig. 7 (workers). It must be remembered that included in the manager’s pseudocode are its duties as a manager and also as a worker.

3) Parallel approach. Second version: In this second version, some changes have been made with respect to the sequential algorithm. Looking at the algorithm, it can be observed that all the computations are made for each $G[i, j]$ but, while looking for the better $G[r, s]$ value, it must be verified that the Bayesian network is still a DAG (Directed Acyclic Graph).

This means that the workers complete some work that can be considered useless. Why must they compute a BIC score if it does not fulfill the DAG property?

The first idea to deal with is to send to the workers the data structure needed for them to check if the DAG property is still maintained ($paths$). However, this solution is not efficient, because work has been distributed among the workers as $num\text{variables}/num\text{workers}$ but it is impossible to know how many BICs must be calculated in each worker (only those
**EBN_{A_{BIC}}** algorithm. First parallel version. Workers

Step 1. create and initialize $S$ local structure
receive $D$
define the set of variables $(NSet)$ to work with
Step 2. wait for an order
Step 3. case order of

"calculate $BIC$"
for each variable $i$ in $NSet$ calculate $BIC[i]$
send $BIC$ results to the manager
"calculate $G[k, i]$"
for each variable $k$ in $NSet$
if $(k \neq i)$ calculate $G[k, i]$
send $G$ modifications to the manager
"calculate $G[k, i]$ for $(i, j)$"
for each variable $k$ in $NSet$
if $(k \neq i$ and $k \neq j)$ calculate $G[k, i]$
send $G$ modifications to the manager
"change arc $(i, j)$"
Update $S$ with $(i, j)$ arc modification
"step" stop
Step 4. go to Step 2

**EBN_{A_{BIC}}** algorithm. Second parallel version. Manager

Input: $D$, $S$, $paths$
Step 1. send $D$ to the workers and set the number of variables ($NSet$) to work with
Step 2. send "calculate $BIC$" order to the workers
for each variable $i$ in $NSet$
calculate $BIC[i]$
Step 3. receive $BIC$ results from workers
Step 4. for $i = 1, \ldots, n$
and $j = 1, \ldots, n$
$G[i, j] = 0$
Step 5. for $i = 1, \ldots, n$
send $i$, $BIC[i]$ to the workers
calculate the structures that fit the DAG property and
set the number of variables ($NSetDAG$) to work with
send to the workers the set of variables ($NSetDAG$) to work with
send "calculate $G[k, i]$" order to the workers
for each variable $k$ in $NSetDAG$
calculate $G[k, i]$
Step 6. receive from workers all the changes and update $G$
Step 7. find $(j, i)$ such that $path[j, i] = 0$ and $G[j, i] \geq G[r, s]$
for each $r, s = 1, \ldots, n$
such that $path[s, r] = 0$
Step 8. if $G[j, i] > 0$
update $S$ with arc modification $(j, i)$
update $paths$
send "change arc $(j, i)$ order" to the workers
else send workers “stop order” and stop
Step 9. calculate the structures that fit the DAG property and
set the number of variables ($NSetDAG$) to work with
send to the workers the set of variables ($NSetDAG$) to work with
and "calculate $G[k, i]$ for $(i, j)$" order
Step 10. for each variable $k$ in $NSet$
calculate $G[k, i]$
Step 11. receive from workers all the changes and update $G$
Step 12. go to Step 7

---

That kept the DAG property). Therefore, great time differences
that can exist between the different workers (unbalanced distribution)
and the manager must wait until the last of them finishes.
As a consequence, it is necessary to determine previously
the number of edges that fulfill the restriction and therefore,
before sending the work, the manager tests for all the possible
changes that maintain the DAG condition, creating a set with those ones. Then, this set is sent to the workers, and each of them calculates the BIC criterion for its proportional subset
assuring an equally distributed work load. In Fig. 8 (manager)
and Fig. 9 (workers) the changes made for this second version
can be seen.

**B. EBN_{A_{PC}}** algorithm [18], [19]

1) Algorithm description: As previous algorithm,
$EBN_{A_{PC}}$ also maintains the common schema described
for EDAs, but, unlike $EBN_{A_{BIC}}$, to complete the learning
phase a “detection of conditional (in)dependencies” method
is used.

This method tries to detect whether or not there are conditional dependencies between all the variables that constitute
the domain. All the detected dependencies are stored and, in
the final step, a Bayesian network is created based on the
detected dependencies.

$EBN_{A_{PC}}$ receives its name from the fact that the algorithm used to detect the conditional (in)dependencies is the $PC$ algorithm, described in [27]. Like almost all recovery algorithms based on independence detections, the PC algorithm
starts by creating the complete undirected graph $G$,
then “thins” that graph by removing edges with zero order
conditional independence relations, “thins” again with first
order conditional relations, second order conditional relations
and so on. The set of variables conditioned on need only be a
subset of the set of variables adjacent to one of the variables of the
pair. The independence test is performed based on the $\chi^2$
distribution. When there are no more tests to do, the orientation
process begins, giving a direction to each edge in $G$.

Concerning the data structures, it is necessary to define
**EBNA\textsubscript{BIC} algorithm. Second parallel version. Workers**

Step 1. create and initialize \( S_w \) local structure  
receive \( D \)  
define the set of variables (\( NSet \)) to work with  
Step 2. wait for an order  
Step 3. \textbf{case order of}  
“calculate \( BIC \)”  
for each variable \( i \) in \( NSet \) calculate \( BIC[i] \)  
send to the manager \( BIC \) results  
“calculate \( G[k, i] \)”  
receive the set of variables (\( NSetDAG \)) to work with  
for each variable \( k \) in \( NSetDAG \)  
if \( k \neq i \) calculate \( G[k, i] \)  
send \( G \) modifications to the manager  
“calculate \( G[k, i] \) for \((i, j)\)”  
receive the set of variables (\( NSet\text{DAG} \)) to work with  
for each variable \( k \) in \( NSet\text{DAG} \)  
if \( k \neq i \) and \( k \neq j \) calculate \( G[k, i] \)  
send \( G \) modifications to the manager  
“change arc \((i, j)\)”  
Update \( S \) with \((i, j)\) are modification  
“stop!” \( \text{stop}! \)  
Step 4. go to Step 2

Fig. 9. Pseudocode for the parallel structural learning phase. Second version of the \( EBNA\textsubscript{BIC} \) algorithm for the workers.

A matrix \( G \) to store the undirected graph structure, where \( G[i, j] = \text{true} \) when there is an edge connecting \( i \) and \( j \) nodes. Also, another structure -\( \text{SepSet} \)- is needed to save the independence relation between two variables and the set of variables conditioned on.

In Fig. 10 a pseudocode for the PC algorithm can be seen. \( \text{Adj}(G, A) \) represents the set of vertices adjacent to the vertex \( A \) in the undirected graph \( G \) and \( I(X_i, X_j | S(X_i, X_j)) \) indicates that \( X_i \) and \( X_j \) are independent given a subset of adjacent variables \(-S(X_i, X_j)\). Note that the graph \( G \) is continually updated, so \( \text{Adj}(G, A) \) is constantly changing as the algorithm progresses. A good review for the induction of Bayesian networks by detecting conditional (in)dependencies can be found in [28].

2) **Parallel approach. First version:** Before implementing any parallel proposal, we did a study of execution times for the sequential algorithm. As can be seen in table I, in this algorithm the learning phase requires nearly 98% of the total execution time, so this is the phase that will be parallelized.

As mentioned earlier, the learning phase can be divided into two different steps: detection of conditional (in)dependencies, and orientation. As can be seen in table II the most important action is the first one -detection of the independencies- and therefore, our parallel approach focuses only on this process.

Once we have explained the \( EBNA\textsubscript{PC} \) algorithm, we propose the parallel algorithm. The model induction begins with a complete undirected graph, and in the first step, all zero order conditional independencies must be searched. This process can be carried out independently by each computational node. Thus, all the pairs are equally distributed between the manager and workers (since the manager must wait for the results it will be used just as another worker). With \( n \) being the number of...
variables, the dependencies between $n(n-1)/2$ pairs must be checked. Thus, each worker takes a set of $(n(n-1)/2)/\text{comp}$ pairs to detect dependencies where $\text{comp}$ is the number of computational nodes (including the manager).

In the following steps, the original algorithm tests sequentially the dependencies between all possible $(X_i, X_j)$ pairs, $i = 1, \ldots, n$ and $j = 1, \ldots, n$, with all the subsets of cardinality $r$ (from $r = 1$ until no subset can be found). The deleted edges and the state of the executions must be controlled, so the manager needs auxiliary structures where the current state is saved:

- **manage_pairs**: stores all the pairs that must be calculated along with each one’s present situation: is being calculated or not, cardinality value and so on.
- **notify_changes**: stores for each worker a list with the edges that must be deleted from the $G$ structure.

Note that the parallel version must repeat exactly the same steps as the sequential algorithm. Starting with zero order dependencies, the pairs can be distributed between the manager and the workers without any additional control. When these computations have finished, the real “problematic” work begins. The manager waits for requests from workers asking for pairs to calculate. To determine whether a pair $(X_i, X_j)$ is susceptible to be calculated, there can not be another pair $(X_k, X_l)$ being calculated such that $X_i = X_k$, $X_i = X_l$ or $X_j = X_k$. In this way, we assure that the order followed to distribute the work is exactly the same as the one used in the sequential algorithm. If there is no possibility of obtaining a pair, the manager obliges the worker to wait until a pair can be selected. This process is repeated until no more dependencies can be found.

In those steps (from first order until the end) the manager will distribute the work to be done among the workers, sending the next pair to be calculated. As this calculation can change the $G$ and $\text{SepSet}$ structures, the manager receives the result of each worker’s execution (the edge must be removed or not) and updates its $G$ and $\text{SepSet}$ structures when necessary, changing also the structure used to notify each worker of the changes made in the $G$ graph -**notify_changes**. This is necessary because each worker needs to have an updated copy of the structure of the $G$ graph before doing any conditional detection and it is cheaper in terms of computation and communication to send the changes than to send the whole $G$ structure each time. The $\text{SepSet}$ structure is only needed in the manager, because, as said earlier, the orientation process will be made sequentially.

The pseudocode of this parallel version of the $\text{EBNA}_\text{PC}$ algorithm can be seen in Fig. 11 (manager) and Fig. 12 (workers).

3) **Parallel approach. Second version**: Observing the operation mode of the parallel proposal when calculating dependencies with adjacent sets of cardinality greater than zero, the manager selects the next pair to calculate and sends it to the worker that has asked for a job, repeating this process for all the requests. During these requests -until the workers compute the assigned pair- some time elapses and the manager does nothing.

An improvement taking advantage of this idle time, is the utilization of the manager as a worker to complete independence tests whilst waiting for another worker’s request. That is, if there is a worker asking for a pair to calculate, the manager selects a new pair and sends it to the worker but, while there are no more requests, the manager selects a new pair and calculates it itself.

In Fig. 13 the modifications made in the manager version of the parallel program can be seen. The worker’s schema is not repeated again because it is the same as in the first version of the $\text{EBNA}_\text{PC}$. 
**EBNA_{PC} algorithm. First parallel version. Manager**

Step 1. form the complete undirected graph \( G \) on vertex set \( V = \{X_1, \ldots, X_n\} \)
Step 2. send \( D \) structure to the workers
Step 3. define the set of pairs \((X_i, X_j)\) that are adjacent in \( G \)
Step 4. select a subset of pairs \((PairsSet)\) to work with
Step 5. for each pair \((X_i, X_j)\) in \((PairsSet)\)
    test conditional dependencies
    if “test fulfilled”
        delete \((i, j)\) edge
Step 6. receive from workers the edges they have deleted and update \( G \)
Step 7. send the updated \( G \) to the workers
Step 8. \( r = 1 \)
Step 9. repeat
    for each worker \( w \) without work
    search for a pair to send
    if there is a pair \((X_i, X_j)\)
        send “notify changes” order to the worker
        send edges deleted by other workers to the worker
        if it is not possible to send a pair (due to priority)
            do not send anything to worker
        if all pairs for all possible \( r \) values have been calculated
            send “stop” order to the worker
        wait for an answer
    case request of “result”
        update the manage.pairs structure
        if “result=true”
            delete from \( G \) the edge that the worker \( w \)
            has calculated and update notify_changes structure
        “stop”
            take note that worker \( w \) has finished its execution
    until all the workers have finished
Step 10. for each triplet of vertices \( X_i, X_j, X_k \) such that the pair \((X_i, X_j)\) and the pair \((X_j, X_k)\) are both adjacent in \( G \)
    but the pair \((X_i, X_k)\) is not adjacent in \( G \), orient \( X_j \rightarrow X_k \rightarrow X_i \)
    if and only if \((X_i, X_k)\) is not in \( StepSet(X_i, X_k) \)
Step 11. repeat
    if \((X_i, X_j)\) and \((X_j, X_k)\) are adjacent, \( X_i \) and \( X_k \) are not adjacent,
    and there is no arrowhead at \( X_k \)
    orient \( X_j \rightarrow X_i \) as \( X_j \rightarrow X_i \)
    if there is a directed path from \( X_i \) to \( X_j \)
    and an edge between \( X_i \) and \( X_j \)
    orient \( X_i \rightarrow X_j \) as \( X_j \rightarrow X_i \)
    until no more edges can be oriented

**EBNA_{PC} algorithm. First parallel version. Workers**

Step 1. form the complete undirected graph \( G \) on vertex set \( V = \{X_1, \ldots, X_n\} \)
Step 2. receive \( D \) structure from manager
Step 3. define the set of pairs \((X_i, X_j)\) that are adjacent in \( G \)
Step 4. select a subset of pairs \((PairsSet_w)\) to work with
Step 5. for each pair \((X_i, X_j)\) in \((PairsSet_w)\)
    test conditional dependencies
    if “test fulfilled” save \((i, j)\) edge in \( DeletedEdges_w \)
Step 6. send to the manager edges in \( DeletedEdges_w \)
Step 7. receive \( G \) from the manager
Step 8. wait for an order
Step 9. case order of “notify changes”
    receive the edges deleted by other workers and update \( G \)
    receive \( i, j, r \) values and test conditional dependencies
    for the \((X_i, X_j)\) pair with all possible sets
    with cardinality \( r \)
    send result (deleted or not) to the manager
    “stop”
    send confirmation and stop
Step 10. \textbf{go to Step} 8

**Fig. 12.** Pseudocode for the parallel structural learning phase. First version of the \( EBNA_{PC} \) algorithm for the workers.

The experiments were executed combining MPI (between the nodes) and threads (two inside each node, the same as the number of processors). An exhaustive study should be made to determine the best combination of MPI nodes and threads for a particular algorithm, problem and machine, but this is outside the scope of this paper. Hence, in our tables and figures only the results obtained by using the configuration outlined in figure 14 will be taken into account.

Since we focus on the learning phase, we selected two problems that have a very simple objective function: the well-known \textit{OneMax} and the \textit{EqualProducts} functions.

The \textit{OneMax} problem consists of maximizing:

\[
OneMax(x) = \sum_{i=1}^{n} x_i
\]

where \( x_i \in \{0,1\} \).

Clearly \textit{OneMax} is a linear function, which is easy to optimize. The computational cost of evaluating this function is tiny.

For the \textit{EqualProducts} function a set of random real numbers \( \{a_1, a_2, \ldots, a_n\} \) in the interval \([0,u]\) has to be given (in our case we used \([0,2]\)). The objective is to select

---

**C. Selected problems and experiments for the discrete domain**

This section reports the scenario used for the experiments, the selected problems and the results obtained.

Beginning with the computer resources, the machine used -which is the same for discrete and continuous domains-, is a cluster of 10 nodes, where each node has two AMD ATLON MP 2000+ processors (CPU frequency 1.6GHz), 512kB cache memory and 1GB of RAM. The operating system is GNU-Linux and the MPI implementation is LAM (6.5.9. version).

All the computers are interconnected using a switched Gigabit Ethernet network.

For the \textit{EqualProducts} function a set of random real numbers \( \{a_1, a_2, \ldots, a_n\} \) in the interval \([0,u]\) has to be given (in our case we used \([0,2]\)). The objective is to select

---

**Fig. 11.** Pseudocode for the parallel structural learning phase. First version of the \( EBNA_{PC} \) algorithm for the manager.
**EBNAPC algorithm. Second parallel version. Manager**

Step 1. form the complete undirected graph $G$ on vertex set $V = \{X_1, \ldots, X_n\}$
Step 2. send $D$ structure to the workers
Step 3. define the set of pairs $(X_i, X_j)$ that are adjacent in $G$
Step 4. select a subset of pairs ($PairsSet$) to work with
Step 5. for each pair $(X_i, X_j)$ in ($PairsSet$)
   test conditional dependencies
   if “test fulfilled”
   delete $(i, j)$ edge
Step 6. receive from workers the edges they have deleted and update $G$
Step 7. send the updated $G$ to the workers
Step 8. $r = 1$
Step 9. repeat
   for each worker $w$ without work
   search for a pair to send
   if there is a pair $(X_i, X_j)$
   send “notify changes” order to the worker
   send edges deleted by other workers to the worker
   send $i, j$ values to the worker
   if it is not possible to send a pair (due to priority)
   do not send anything to worker
   if all pairs for all possible $r$ values have been calculated
   send “stop” order to the worker
   if there is an answer
   case request of “result”
   update the $managePairs$ structure
   if “result=true”
   delete from $G$ the edge that the worker $w$ has calculated and update $notifyChanges$ “stop”
   take note that the worker $w$ has finished its execution
   else
   search for a pair
   if there is a pair $(X_i, X_j)$
   test conditional dependencies
   if “test fulfilled”
   delete $(i, j)$ edge and update $notifyChanges$
until all the workers have finished
Step 10. For each triplet of vertices $X_i, X_j, X_k$ such that the pair $X_i, X_j$ and the pair $X_j, X_k$ are both adjacent in $G$
   but the pair $X_i, X_k$ are not adjacent in $G$, orient $X_i \rightarrow X_j \rightarrow X_k$ as $X_i \rightarrow X_k \leftarrow X_j$ if and only if $X_i$ is not in Step5 $(X_i, X_j)$
Step 11. repeat
   if $X_j \rightarrow X_i$, $X_j$ and $X_i$ are adjacent, $X_j$ and $X_i$ are not adjacent, and there is no arrowhead at $X_j$
   orient $X_j \rightarrow X_i$ as $X_j \rightarrow X_i$
   if there is a directed path from $X_i$ to $X_j$
   and an edge between $X_i$ and $X_j$
   orient $X_i \rightarrow X_j$ as $X_i \rightarrow X_j$
until no more edges can be oriented

---

some numbers in such a way that the difference between the products of selected and unselected numbers is minimized.

Mathematically:

$$EqualProducts(x) = \prod_{i=1}^{n} h(x_i, a_i) - \prod_{i=1}^{n} h(1 - x_i, a_i)$$

where function $h$ is defined as:

$$h(x, a) = \begin{cases} 1 & \text{if } x = 0 \\ a & \text{if } x = 1 \end{cases}$$

In this problem we do not know the optimum, but the closer the solution is to zero, the better.

Regarding the experiments, several executions were done with different numbers of nodes, measuring different aspects or dimensions: time-related, solution quality-related and algorithm performance-related. Each different experiment was repeated 10 times. The results presented here are the average of all the values obtained.

1) **Time-related dimension**: The main goal of a parallelization process is usually to reduce the computation time required for a problem to be solved. In the following tables and figures, execution time and efficiency as well as the speedup are presented. These last two concepts are defined as:

- **Speed-up** =\[
\frac{\text{sequential time}}{\text{parallel time}}
\]

- **Efficiency** =\[
\frac{\text{speed-up}}{\text{number of processors}}
\]

For both the $EBNABTC$ and $EBNAPC$ algorithms, the size of the population is 2,000. From this population, the best 1,000 individuals are selected at each generation and the executions are stopped when the 15th generation is reached. The size of the individual is different for each algorithm: 200
Tables III and IV summarize the results of the experiments with OneMax and EqualProducts functions for both algorithms. It can be seen that there is a dramatic time reduction in all cases as the number of workers increases.

Particularly, for $EBNA_{BIC}$ algorithm, the times obtained by the second parallel version are better than those of the first, and it is significant that when 7 or more nodes are used the computation time decreases more slightly -see Fig. 15 and 16.

Due to the parallelization process, communication between the manager and the workers is necessary to maintain data integrity, but when the number of workers is excessive, the time spent to communicate and update the structures is greater than the computation reduction obtained.

Before explaining the results obtained for the $EBNA_{PC}$ algorithm, it is necessary to point out that for this particular algorithm only the first level parallelization (MPI) was used. Remembering the pseudocode (Fig. 11), notice that workers manage one pair each time, so it is not worth it to create second level workers (threads) to have each one compute part of the work.

Once this has been clarified, we can see in Fig. 17 and 18 the different tendencies of the two parallel versions of the $EBNA_{PC}$ algorithm. In the first one (the manager waits for the workers to finish without doing anything except distributing pairs) when there are few workers the waiting-time is very high, while in the second version this idle time is used by the manager to act as a worker, yielding better results for the second parallel version. However, when the number of workers increases (for 6 or more nodes), there are more requests from workers asking for a pair and therefore the manager has less
idle time. If the manager also computes pairs, this forces the workers to wait, increasing the waiting-time.

2) quality-related dimension: As mentioned before, all the algorithm versions have exactly the same behavior: that of the sequential version. The stopping criteria for those EDA algorithms is diverse. In some cases, algorithms stop when a given quality of the solution is reached (without a-priori knowledge of the execution time required to reach it). In many others, we look for the best possible solution given a fixed execution time (common in real environments).

In Fig. 19, 20, 21, 22, 23, 24, 25 and 26 we show the evolution of the solution’s quality with time. It can be seen that when execution time is fixed, quality of parallel solutions are considerably better.

3) performance-related dimension: When a parallel solution for an algorithm is proposed, the work must be distributed correctly between all the nodes that will work in parallel. The execution time for each node is shown in tables V, VI, VII and VIII (EBNA_BIC algorithm with OneMax and EqualProducts problems) and tables IX, X, XI and XII (EBNA_PC algorithm with OneMax and EqualProducts problems). The node with the highest computation times is obviously the manager, since it is the one that centralizes all the processes,
and as we can see the work is equally distributed among the workers. There are slight differences due to the work distribution (it is not always an exact division) and the communication connections managed by the MPI implementation, but distribution of the work is properly balanced.
### TABLE VI
Algorithm performance-related experimental results for EBN_A1C second version with OneMax problem. Execution times for manager and workers.

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>464</td>
<td>418</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>373</td>
<td>310</td>
<td>309</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>317</td>
<td>244</td>
<td>242</td>
<td>240</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>247</td>
<td>189</td>
<td>179</td>
<td>178</td>
<td>175</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>242</td>
<td>192</td>
<td>168</td>
<td>166</td>
<td>166</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>221</td>
<td>173</td>
<td>158</td>
<td>147</td>
<td>146</td>
<td>146</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>199</td>
<td>154</td>
<td>140</td>
<td>141</td>
<td>131</td>
<td>130</td>
<td>129</td>
<td>128</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>196</td>
<td>134</td>
<td>122</td>
<td>122</td>
<td>112</td>
<td>112</td>
<td>112</td>
<td>110</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>183</td>
<td>138</td>
<td>113</td>
<td>112</td>
<td>104</td>
<td>103</td>
<td>102</td>
<td>101</td>
<td>102</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### TABLE VII
Algorithm performance-related experimental results for EBN_A1C first version with EqualProducts problem. Execution times for manager and workers.

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>574</td>
<td>522</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>430</td>
<td>365</td>
<td>358</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>346</td>
<td>273</td>
<td>272</td>
<td>271</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>299</td>
<td>225</td>
<td>224</td>
<td>223</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>268</td>
<td>216</td>
<td>189</td>
<td>189</td>
<td>187</td>
<td>188</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>244</td>
<td>193</td>
<td>180</td>
<td>169</td>
<td>164</td>
<td>164</td>
<td>163</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>218</td>
<td>167</td>
<td>154</td>
<td>154</td>
<td>144</td>
<td>145</td>
<td>145</td>
<td>146</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>220</td>
<td>159</td>
<td>140</td>
<td>141</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>132</td>
<td>131</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>205</td>
<td>157</td>
<td>132</td>
<td>132</td>
<td>122</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>124</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### TABLE VIII
Algorithm performance-related experimental results for EBN_A2C second version with EqualProducts problem. Execution times for manager and workers.

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>495</td>
<td>446</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>368</td>
<td>305</td>
<td>304</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>304</td>
<td>233</td>
<td>232</td>
<td>229</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>270</td>
<td>207</td>
<td>198</td>
<td>196</td>
<td>194</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>239</td>
<td>190</td>
<td>167</td>
<td>168</td>
<td>165</td>
<td>165</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>215</td>
<td>165</td>
<td>151</td>
<td>140</td>
<td>138</td>
<td>139</td>
<td>138</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>204</td>
<td>156</td>
<td>142</td>
<td>141</td>
<td>132</td>
<td>131</td>
<td>130</td>
<td>130</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>199</td>
<td>137</td>
<td>123</td>
<td>125</td>
<td>114</td>
<td>114</td>
<td>115</td>
<td>114</td>
<td>111</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>193</td>
<td>143</td>
<td>116</td>
<td>116</td>
<td>106</td>
<td>106</td>
<td>106</td>
<td>106</td>
<td>104</td>
<td>105</td>
<td>-</td>
</tr>
</tbody>
</table>

### TABLE IX
Algorithm performance-related experimental results for EBN_A1C first version with OneMax problem. Execution times for manager and workers.

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>983</td>
<td>2149</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>690</td>
<td>1206</td>
<td>1204</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>544</td>
<td>852</td>
<td>851</td>
<td>852</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>458</td>
<td>665</td>
<td>663</td>
<td>663</td>
<td>663</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>399</td>
<td>542</td>
<td>540</td>
<td>540</td>
<td>540</td>
<td>538</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>358</td>
<td>461</td>
<td>459</td>
<td>458</td>
<td>458</td>
<td>456</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>328</td>
<td>400</td>
<td>397</td>
<td>397</td>
<td>396</td>
<td>396</td>
<td>395</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>305</td>
<td>354</td>
<td>352</td>
<td>351</td>
<td>351</td>
<td>351</td>
<td>350</td>
<td>349</td>
<td>351</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>286</td>
<td>315</td>
<td>313</td>
<td>312</td>
<td>311</td>
<td>311</td>
<td>311</td>
<td>313</td>
<td>311</td>
<td>313</td>
<td>311</td>
</tr>
</tbody>
</table>

### TABLE X
Algorithm performance-related experimental results for EBN_A2C second version with OneMax problem. Execution times for manager and workers.

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1308</td>
<td>1042</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>969</td>
<td>715</td>
<td>717</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>790</td>
<td>567</td>
<td>566</td>
<td>563</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>677</td>
<td>468</td>
<td>467</td>
<td>465</td>
<td>463</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>599</td>
<td>403</td>
<td>399</td>
<td>398</td>
<td>396</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>540</td>
<td>353</td>
<td>351</td>
<td>350</td>
<td>348</td>
<td>348</td>
<td>344</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>491</td>
<td>315</td>
<td>312</td>
<td>310</td>
<td>308</td>
<td>306</td>
<td>306</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>454</td>
<td>286</td>
<td>283</td>
<td>281</td>
<td>282</td>
<td>279</td>
<td>279</td>
<td>277</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>425</td>
<td>262</td>
<td>259</td>
<td>257</td>
<td>258</td>
<td>257</td>
<td>255</td>
<td>254</td>
<td>254</td>
<td>255</td>
<td>-</td>
</tr>
</tbody>
</table>

### TABLE XI
Algorithm performance-related experimental results for EBN_A1C first version with EqualProducts problem. Execution times for manager and workers.

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>983</td>
<td>2081</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>691</td>
<td>1173</td>
<td>1170</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>545</td>
<td>829</td>
<td>827</td>
<td>827</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>458</td>
<td>641</td>
<td>640</td>
<td>640</td>
<td>640</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>400</td>
<td>528</td>
<td>527</td>
<td>527</td>
<td>527</td>
<td>527</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>359</td>
<td>446</td>
<td>443</td>
<td>443</td>
<td>443</td>
<td>443</td>
<td>441</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>329</td>
<td>389</td>
<td>388</td>
<td>387</td>
<td>387</td>
<td>385</td>
<td>385</td>
<td>385</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>305</td>
<td>340</td>
<td>338</td>
<td>337</td>
<td>338</td>
<td>337</td>
<td>336</td>
<td>336</td>
<td>338</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>287</td>
<td>307</td>
<td>306</td>
<td>304</td>
<td>305</td>
<td>302</td>
<td>302</td>
<td>303</td>
<td>306</td>
<td>303</td>
<td>-</td>
</tr>
</tbody>
</table>
TABLE XII
ALGORITHM PERFORMANCE-RELATED EXPERIMENTAL RESULTS FOR
ENA_{pc} SECOND VERSION WITH EqualProducts PROBLEM. EXECUTION TIMES FOR MANAGER AND WORKERS.

<table>
<thead>
<tr>
<th></th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1301</td>
<td>-</td>
</tr>
<tr>
<td>722</td>
<td>-</td>
</tr>
<tr>
<td>560</td>
<td>-</td>
</tr>
<tr>
<td>462</td>
<td>-</td>
</tr>
<tr>
<td>398</td>
<td>-</td>
</tr>
<tr>
<td>349</td>
<td>-</td>
</tr>
<tr>
<td>281</td>
<td>-</td>
</tr>
<tr>
<td>259</td>
<td>-</td>
</tr>
</tbody>
</table>

V. CONTINUOUS DOMAIN

A. ENA_{BIC} algorithm [2]

1) Algorithm description: As we have already noted, EDA algorithms can be used in discrete and continuous domains. In this section we propose the parallelization of the ENA_{BIC} algorithm, which is similar to discrete ENA_{BIC}, but modified for use in continuous domains. A Gaussian network is created instead of a Bayesian network, using a “score + search” approach. That is, the main idea under this approach consists of having a measure for each candidate Gaussian network in combination with a smart search through the space of possible structures. All the comments made concerning ENA_{BIC} are also valid for this algorithm.

The score used to verify the performance of the Gaussian network obtained is the Bayesian Information Criterion (BIC).

A general formula for this criterion is as follows:

\[
BIC(S, D) = \sum_{i=1}^{n} BIC(i, S, D)
\]

where

- \( n \) is the number of variables in the Gaussian network.
- \( N \) is the number of selected individuals.
- \( v_i \) is the conditional variance for the variable \( X_i \) given \( Pa_i \).
- \( m_i \) is the mean of the variable \( X_i \).
- \( b_{ji} \) is the regression coefficient for variable \( X_j \) in \( Pa_i \).

As was explained for ENA_{BIC} algorithm, this score can also be broken down to calculate separately the score for each variable. Accordingly, each variable \( X_i \) has associated with it a local BIC score (\( BIC(i, S, D) \)):

\[
BIC(i, S, D) = \sum_{r=1}^{N} \left[ -\ln \left( \sqrt{2\pi v_i} \right) - \frac{1}{2v_i} (x_{ir} - m_i - \sum_{x_j \in Pa_i} b_{ji} (x_{jr} - m_j))^2 \right] - \frac{1}{2} \log(N)(2n + |Pa_i|) \]  

Consequently, the steps followed to parallelize this algorithm are similar to those seen for the ENA_{BIC}, decomposing the BIC criterion and sending each piece of the score to a different worker.

2) Parallel approach: As parallel approaches, we maintain the two proposals made for the ENA_{BIC} algorithm. The first one, where workers calculate all possible arc modifications without considering whether the DAG property is fulfilled, and the second, where the manager makes a pre-process step to obtain the arc changes that maintain the DAG property, then sending to each worker a subset of those possible modifications.

Due to the similarity of this algorithm to the discrete one (ENA_{BIC}) it is unnecessary to explain the entire process again. To obtain a complete view of the characteristics of the
algorithm and the parallel solution, see section IV-A.

B. EGNABE algorithm [16], [17]

1) Algorithm description: In this algorithm the structure learning of the Gaussian network follows a “detecting conditional (in)dependencies” method.

Particularly, this method begins with a completed graph, where there is a connection from each variable $X_i$, $i = 1, \ldots, n$ to each variable $X_j$, $j = i + 1, \ldots, n$, and then a statistical test is completed for each edge, deleting the edge if the null hypothesis is fulfilled.

To complete this test, the likelihood ratio test is used - borrowed from [29]. The statistic to exclude the edge between $X_i$ and $X_j$, respectively, of a graphical Gaussian model, is $T_{ijk} = -n \log (1 - r_{ij|rest}^2)$ where $r_{ij|rest}$ is the sample partial correlation of $X_i$ and $X_j$ adjusted for the remaining variables.

The latter can be expressed [30] in terms of the maximum likelihood estimates of the elements of the precision matrix as $r_{ij|rest} = \hat{\omega}_{ij}(\hat{\omega}_{ii}\hat{\omega}_{jj})^{-\frac{1}{2}}$, where the precision matrix $W$ is the inverse of the covariance matrix $\Sigma$.

[29] obtain the density and distribution functions of the likelihood ratio test statistic under the null hypothesis. These expressions are of the form:

$$f_{ik}(t) = g_X(t) + \frac{1}{4}(t - 1)(2n + 1)g_X(t)N^{-1} + O(N^{-2}) \quad (11)$$

$$F_{ik}(x) = G_X(x) - \frac{1}{2}(2n + 1)xg_X(x)N^{-1} + O(N^{-2}) \quad (12)$$

where $g_X(t)$ and $G_X(x)$ are the density and distribution functions, respectively, of a $\chi^2$ variable.

Once the structure has been obtained, to complete the Gaussian network the parameters have to be learnt. This can be done using the following formulas:

$$\hat{m}_i = \bar{X}_i \quad (13)$$

EGNABE algorithm. Sequential version.

Input: $D$
Step 1. calculate the means $m$ and the covariance matrix $\Sigma$
Step 2. initialize the structure of the Gaussian network $S$
Step 3. calculate the inverse of $\Sigma$
Step 4. for $i = 1, \ldots, n$
  for $j = i + 1, \ldots, n$
    calculate $r_{ij|rest}$
    if “test fulfilled”
      update $S$ with $(i, j)$ arc deletion
Step 5. calculate the parameters $m$, $b$ and $v$

Fig. 27. Pseudocode for the sequential learning algorithm. EGNABE.

<table>
<thead>
<tr>
<th>Ind. size</th>
<th>%Learn.</th>
<th>%Sampl.</th>
<th>%Learn.</th>
<th>%Sampl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>10.3</td>
<td>89.6</td>
<td>9.7</td>
<td>90.3</td>
</tr>
<tr>
<td>150</td>
<td>44.6</td>
<td>55.1</td>
<td>42.8</td>
<td>57.1</td>
</tr>
<tr>
<td>350</td>
<td>75.5</td>
<td>24.3</td>
<td>77.4</td>
<td>22.4</td>
</tr>
</tbody>
</table>

(14)

$$\hat{b}_{ji} = \frac{\hat{\sigma}_{ji}}{\hat{\sigma}_{jj}}$$

$$\hat{v}_i = \hat{\sigma}_{ii}^2 - \sum_{x_j \in P a_i} \hat{\sigma}_{ji}^2 + 2 \sum_{x_j \in P a_i, x_k \in P a_{i k > j}} \hat{\sigma}_{jk} \hat{\sigma}_{ji} \hat{\sigma}_{ki} \hat{\sigma}_{jk}$$

(15)

where $\mathbf{m}$ is the estimated means, $\mathbf{v}$ the estimated conditional variances and $\mathbf{b}$ the estimated regression coefficients.

Six main structures are needed to implement this algorithm. The graph structure is stored as $S[i], i = 1, \ldots, n$ where for each node its parents are represented in an adjacency list. Two $n \times n$ structures $\Sigma$ and $W$ where the covariances matrix and its inverse are stored respectively. Two vectors, $m[i], i = 1, \ldots, n$ and $v[i], i = 1, \ldots, n$ for means and conditional variances and finally an $n \times n$ matrix, $b[i, j], i = 1, \ldots, n$ and $j = 1, \ldots, n$ where the regression coefficients will be saved.

A pseudocode for the sequential structure learning algorithm is presented in Fig. 27.

2) Parallel approach. First version: As can be seen in table I (presented in section III), the learning phase does not take as long as with previous algorithms. Due to this, a deeper
TABLE XIV
TIME MEASUREMENT FOR DIFFERENT PROCESSES OF THE \textit{EGNAXE} LEARNING PHASE.

<table>
<thead>
<tr>
<th>Ind. size</th>
<th>%(\Sigma)</th>
<th>%Prec. matr.</th>
<th>%test</th>
<th>%Parm. learn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>92.0</td>
<td>2.0</td>
<td>0.5</td>
<td>5.3</td>
</tr>
<tr>
<td>150</td>
<td>71.1</td>
<td>2.5</td>
<td>0.8</td>
<td>25.5</td>
</tr>
<tr>
<td>350</td>
<td>65.3</td>
<td>5.7</td>
<td>1.2</td>
<td>27.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ind. size</th>
<th>%(\Sigma)</th>
<th>%Prec. matr.</th>
<th>%test</th>
<th>%Parm. learn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>92.0</td>
<td>2.0</td>
<td>0.5</td>
<td>5.4</td>
</tr>
<tr>
<td>150</td>
<td>69.0</td>
<td>2.6</td>
<td>0.7</td>
<td>27.6</td>
</tr>
<tr>
<td>350</td>
<td>66.5</td>
<td>5.3</td>
<td>1.0</td>
<td>27.2</td>
</tr>
</tbody>
</table>

Analysis has been done to know how the computation time is distributed among the different procedures. Table XIII shows the most important procedures that are executed by the present algorithm: learning and sampling of the new individuals.

In this first parallel approach, the parallelization of the learning phase is presented, and due to the execution time distribution, this version will be completed -in the second approach- with the parallelization of the creation of new individuals.

In this particular case, the learning phase has several independent procedures and we can descend one level in depth to obtain the exact computation time spent on each of these processes. As explained for the sequential algorithm, the learning phase completes the following steps:

1) Calculate means and \(\Sigma\) from the selected individuals.
2) Obtain the precision matrix.
3) Compute the tests.
4) Carry out the parameter learning.

Table XIV shows the times required by each of these processes. We can see that the two most computationally expensive phases are the first one, where means and \(\Sigma\) are calculated, and the fourth one, where the parameters are learnt. In the second step -calculate the inverse of the covariance matrix- the LU decomposition method is used and execution is very fast. The third step -test- is also quickly executed because it uses the values computed in the previous phases to obtain the \(r_{ij}\) values. Clearly, this distribution of the time is conditioned by the parameters of the particular problem. That is, if the size of the population grows or the size of the individuals changes the computation distribution between the four steps can be quite different. Concerning our problem, our parallelizing efforts must focus on steps one and four.

To calculate \(\Sigma\) it is necessary first to obtain the means for each variable, so the database of cases is sent to each worker and the number of means to be calculated (number of variables) is distributed among all the workers (manager included). If \(n\) is the number of variables and \(\text{comp}\) the number of nodes, each node will have \(n/\text{comp}\) variables to compute. Once the means have been calculated, all the nodes must send their results and receive the ones computed by the others. Thus, all the nodes have an updated \textit{means} structure, which is needed for the next step: compute \(\Sigma\) matrix. The idea is the same, where each node computes \((n(n - 1)/2)/\text{comp}\) values of the matrix. Next, each worker sends its results to the manager and it updates the \textit{\Sigma} structure, continuing with the sequential processes: calculate the precision matrix (inverse of \(\Sigma\)) and compute the tests. Finally, the parameters must be learnt, and again, a distribution similar to the previous one is made, with each node computing a piece of \(b\) and \(v\).

Fig. 28 and Fig. 29 show the pseudocode for the parallel version.

3) Parallel approach. Second version: Due to the fact that the learning process is not as computationally weighty as in the previous algorithms, the results obtained are not as good as in the other programs. If we look at table XIII, we can see that there is another process that consumes a large portion of the total execution time: creation of the new individuals (sampling).

For this reason, we present as a second version, the parallelization of both phases: learning and sampling. Remembering
the general structure of EDAs, an initial population is created, the best \( N \) individuals are selected, a probabilistic model is induced (learning phase) and finally, a new population is generated based on the induced model. We use an adaptation of the Probabilistic Logic Sampling (PLS) proposed in [31].

In this method the instances are generated one variable at a time in a forward way. That is, a variable is sampled after all its parents have already been sampled. To do that an ancestral ordering of the variables is given \((\pi(1), \ldots, \pi(n))\) where parent variables are before children variables. Once the values of \( \mathbf{P}a_\pi(i) \) have been assigned, we simulate a value for \( X_{\pi(i)} \), using the distribution \( f(x_{\pi(i)} | Pa_{\pi}(i)) \). For the simulation of a univariate normal distribution, a simple method based on the sum of 12 uniform variables is applied [32].

A pseudocode for this sampling process can be seen in Fig. 30.

The parallelization approach is the following: If \( New\text{-}PopSet \) is the amount of new individuals to be created and \( \text{comp} \) the number of computational nodes, the manager sends to each worker the order to create \( New\text{-}PopSet / \text{comp} \) new individuals (the manager also creates new individuals). Once all these new individuals have been created, they are sent to the manager which adds them to the population and continues with the algorithm.

In Fig. 31 (manager) and Fig. 32 (workers) the parallel proposal for the sampling phase can be seen.
**EGNA\_EE sampling algorithm. Parallel version. Workers**

Step 1. receive $S$, Order ($\pi(i)$), idets, $\pi(n)$, $m$, $b$ and $w$ structures

Step 2. for each $i$ in NewPopSet
- calculate a new individual

Step 3. send the new individuals to the manager

Fig. 32. Pseudocode for the parallel sampling phase. EGNA\_EE algorithm for the workers.

**C. Selected problems and experiments for the continuous domain**

The scenario used for the continuous domain is the same as used for the discrete one, so this information can be obtained in section IV-C

In order to test the different proposals, we have chosen two standard functions broadly used in the literature for comparing optimization techniques.

The *Sphere* model is a simple minimization problem. It is defined so that $-600 \leq x_i \leq 600$, $i = 1, \ldots, n$, and the fitness value for an individual is as follows:

$$ Sphere(x) = \sum_{i=1}^{n} x_i^2 \quad (16) $$

As the reader can see the fittest individual is the one whose components are all 0, which corresponds to the fitness value 0.

The second function is *Rosenbrock generalized*, a minimization problem proposed by [33], extending to $n$ dimensions the original proposal [34].

It is defined as:

$$ Rosenbrock\_generalized(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \quad (17) $$

The optimum value is 0, which is obtained again for the individual whose components are all set to 1. The range of values is of $-10 \leq x_i \leq 10$, $i = 1, \ldots, n$.

Concerning analysis of the experiments, the same dimensions as for discrete domain algorithms have been taken into account for these continuous algorithms. Therefore we analyze different dimensions such as time-related, performance-related and solution quality-related.

1) *time-related dimension:* In this section, the execution time as well as efficiency and speed-up are presented. The experiments were executed with a population of 2,000, with the number of individuals selected at each generation set at 1,000. As particular parameters, EGNA\_BIC was executed with an individual size of 60 and the number of generations was limited to 15 while for EGNA\_EE algorithm, the size of the individuals was 300 and the execution was stopped when the 300th generation was reached (looking for an execution time of the sequential algorithm close to 30 minutes).

The results obtained for EGNA\_BIC algorithm can be seen in Fig. 33 and 34 and table XV.

It can be observed that the parallel algorithm represents a great improvement over the sequential version. In particular, the results obtained with the second parallel version are really good: for six or less nodes, efficiency is always over 0.8, being even more than 1 when 2, 3 or 4 nodes are used. It should be remembered that the program is executed in different machines (using MPI) and therefore, depending on the characteristics of the algorithm and the available architecture, results like...
TABLE XVI

<table>
<thead>
<tr>
<th>Nod</th>
<th>Sphere model</th>
<th>Rosenbrock gener.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1st version</td>
<td>2nd version</td>
</tr>
<tr>
<td>seq.</td>
<td>secs</td>
<td>eff</td>
</tr>
<tr>
<td>1</td>
<td>1850</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>877</td>
<td>0.53</td>
</tr>
<tr>
<td>3</td>
<td>807</td>
<td>0.38</td>
</tr>
<tr>
<td>4</td>
<td>790</td>
<td>0.29</td>
</tr>
<tr>
<td>5</td>
<td>789</td>
<td>0.23</td>
</tr>
<tr>
<td>6</td>
<td>787</td>
<td>0.20</td>
</tr>
<tr>
<td>7</td>
<td>782</td>
<td>0.17</td>
</tr>
<tr>
<td>8</td>
<td>790</td>
<td>0.15</td>
</tr>
<tr>
<td>9</td>
<td>797</td>
<td>0.13</td>
</tr>
<tr>
<td>10</td>
<td>812</td>
<td>0.11</td>
</tr>
</tbody>
</table>

These results can be seen in Fig. 35 and 36 and Table XVI. However, when combining the parallelization of learning with the parallelization of the sampling phase -the second parallel version of $E\!G\!N\!A_{EE}$-, the results are better, reaching a reasonable speed-up. It can also be seen -for this second parallel version- that the best results are obtained when seven or eight nodes are used, decreasing the efficiency with nine or more. It is important to keep in mind that a good balance between communication and computation must be maintained, because if too many nodes are used (last cases) the process of work distribution and recollection supposes an important delay.

Concerning the parallelization of the $E\!G\!N\!A_{EE}$ algorithm, it can be seen that the results are not as good as in previous algorithms. In the explanation given for this algorithm, the learning phase is not as important computationally as in the other algorithms, and therefore the improvement obtained can not be excellent: for the first version, there is nearly naught time reduction when three or more nodes are used.

2) quality-related dimension: As said before, the proposed parallel version follows exactly the same steps as the sequential version, so it is interesting to compare the objective function values that are obtained by the different parallel versions over the time. Fig. 37, 38, 39 and 40 show the
difference of the sequential version compared to the parallel ones for the $EGNA_{BIC}$ algorithm, where better objective function values can be obtained in a short time.

Concerning to $EGNA_{EE}$ algorithm, take note that in the first version, the difference when three or more nodes are used is really small, being slightly more appreciable for the second parallel version (see Fig. 41, 42, 43 and 44.)

3) performance-related dimension: The last results are those obtained from the execution times of manager and workers. These show how much work is executed by each type of node (in this case manager or worker) and can also be used to verify whether the load is equally distributed between the workers. Fig. XVII, XVIII, XIX and XX show the detailed information obtained from the execution of the
Fig. 42. Solution produced by the second version of the \textit{EGNA}_{E/E} algorithm throughout the time with the \textit{Sphere} model problem

Fig. 43. Solution produced by the first version of the \textit{EGNA}_{E/E} algorithm throughout the time with the \textit{Rosenbrock generalized} problem

Fig. 44. Solution produced by the second version of the \textit{EGNA}_{E/E} algorithm throughout the time with the \textit{Rosenbrock generalized} problem

\textit{EGNA}_{BIC} algorithm with the two problems (\textit{Sphere} model and \textit{Rosenbrock generalized}). In most of the cases, manager and workers require a similar amount of computation time (remember that the manager carries out the sequential steps as well as the communications) and, if we focus on the workers, we see that load is quite well balanced. Particularly, it can be observed that the first workers consume generally more computational time than the others; this is mainly due to the work division: it is not always possible to evenly distribute the variables among the workers, so some of them (the first one’s) may be receiving a bit more work.

For the first version of the \textit{EGNA}_{E/E} algorithm, the behavior is different. As the learning phase does not take much more than half of the total execution time, it can be seen that the work distribution between the manager and the workers is very uneven, becoming more evident as the number of workers increases. In the second parallel version, more code has been parallelized, so the manager-worker duties are better distributed and therefore better results are obtained. Even so, the differences between the execution times of manager and workers are greater than in the previous parallel algorithms.

### Table XVII

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>808</td>
<td>787 - - - - - - -</td>
</tr>
<tr>
<td>567</td>
<td>543 538 - - - - - -</td>
</tr>
<tr>
<td>445</td>
<td>420 419 417 - - - - -</td>
</tr>
<tr>
<td>369</td>
<td>346 343 343 341 - - - -</td>
</tr>
<tr>
<td>318</td>
<td>298 295 294 294 294 - - -</td>
</tr>
<tr>
<td>293</td>
<td>273 272 268 246 246 245 - - -</td>
</tr>
<tr>
<td>269</td>
<td>249 249 246 228 225 219 218 - -</td>
</tr>
<tr>
<td>247</td>
<td>227 225 221 218 199 197 197 -</td>
</tr>
<tr>
<td>221</td>
<td>203 199 201 200 198 197 198 199 196</td>
</tr>
</tbody>
</table>

### Table XVIII

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>612</td>
<td>581 - - - - - - -</td>
</tr>
<tr>
<td>438</td>
<td>413 401 - - - - - -</td>
</tr>
<tr>
<td>348</td>
<td>324 316 314 - - - - -</td>
</tr>
<tr>
<td>295</td>
<td>273 265 263 254 - - - - -</td>
</tr>
<tr>
<td>260</td>
<td>242 234 233 228 224 - - - -</td>
</tr>
<tr>
<td>235</td>
<td>217 213 208 200 198 194 - - -</td>
</tr>
<tr>
<td>208</td>
<td>196 189 188 181 179 175 176 - -</td>
</tr>
<tr>
<td>198</td>
<td>181 179 179 172 168 163 164 157 -</td>
</tr>
<tr>
<td>181</td>
<td>166 162 162 157 154 152 152 147 146</td>
</tr>
</tbody>
</table>
### TABLE XIX
**Algorithm performance-related experimental results for EGN$_{AIC}$ first version with Rosenbrock generalized problem. Execution times for manager and workers.**

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>769</td>
<td>746 - - - - - - -</td>
</tr>
<tr>
<td>533</td>
<td>508 509 - - - - - - -</td>
</tr>
<tr>
<td>422</td>
<td>399 396 396 - - - - - - -</td>
</tr>
<tr>
<td>352</td>
<td>330 327 330 326 - - - - - - -</td>
</tr>
<tr>
<td>305</td>
<td>285 282 285 285 284 - - - - - - -</td>
</tr>
<tr>
<td>283</td>
<td>261 261 262 233 237 233 - - - - - - -</td>
</tr>
<tr>
<td>260</td>
<td>237 236 240 213 215 211 214 - - - - - - -</td>
</tr>
<tr>
<td>233</td>
<td>211 209 213 212 214 187 190 189 - - - - - - -</td>
</tr>
<tr>
<td>209</td>
<td>191 190 194 190 193 190 191 188 192 - - - - - - -</td>
</tr>
</tbody>
</table>

### TABLE XXI
**Algorithm performance-related experimental results for EGN$_{EIE}$ second version with Sphere model problem. Execution times for manager and workers.**

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>768</td>
<td>748 - - - - - - -</td>
</tr>
<tr>
<td>533</td>
<td>508 509 - - - - - - -</td>
</tr>
<tr>
<td>422</td>
<td>399 396 396 - - - - - - -</td>
</tr>
<tr>
<td>352</td>
<td>330 327 330 326 - - - - - - -</td>
</tr>
<tr>
<td>305</td>
<td>285 282 285 285 284 - - - - - - -</td>
</tr>
<tr>
<td>283</td>
<td>261 261 262 233 237 233 - - - - - - -</td>
</tr>
<tr>
<td>260</td>
<td>237 236 240 213 215 211 214 - - - - - - -</td>
</tr>
<tr>
<td>233</td>
<td>211 209 213 212 214 187 190 189 - - - - - - -</td>
</tr>
<tr>
<td>209</td>
<td>191 190 194 190 193 190 191 188 192 - - - - - - -</td>
</tr>
</tbody>
</table>

### TABLE XXII
**Algorithm performance-related experimental results for EGN$_{AIC}$ second version with Rosenbrock generalized problem. Execution times for manager and workers.**

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>741</td>
<td>432 - - - - - - -</td>
</tr>
<tr>
<td>553</td>
<td>277 276 - - - - - - -</td>
</tr>
<tr>
<td>420</td>
<td>213 213 213 - - - - - - -</td>
</tr>
<tr>
<td>433</td>
<td>179 174 176 174 - - - - - - -</td>
</tr>
<tr>
<td>403</td>
<td>160 148 148 148 148 - - - - - - -</td>
</tr>
<tr>
<td>378</td>
<td>141 134 131 130 131 128 - - - - - - -</td>
</tr>
<tr>
<td>362</td>
<td>126 120 120 115 115 115 115 - - - - - - -</td>
</tr>
<tr>
<td>356</td>
<td>116 108 108 104 104 105 105 - - - - - - -</td>
</tr>
<tr>
<td>343</td>
<td>113 99 99 95 96 96 96 96 - - - - - - -</td>
</tr>
</tbody>
</table>

### TABLE XXIII
**Algorithm performance-related experimental results for EGN$_{EIE}$ first version with Rosenbrock generalized problem. Execution times for manager and workers.**

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>880</td>
<td>173 - - - - - - -</td>
</tr>
<tr>
<td>810</td>
<td>121 120 - - - - - - -</td>
</tr>
<tr>
<td>790</td>
<td>94 94 94 - - - - - - -</td>
</tr>
<tr>
<td>777</td>
<td>83 78 79 79 - - - - - - -</td>
</tr>
<tr>
<td>765</td>
<td>78 68 68 68 67 - - - - - - -</td>
</tr>
<tr>
<td>754</td>
<td>71 65 61 60 61 59 - - - - - - -</td>
</tr>
<tr>
<td>750</td>
<td>65 57 59 53 53 53 54 - - - - - - -</td>
</tr>
<tr>
<td>715</td>
<td>61 54 53 49 49 48 50 49 - - - - - - -</td>
</tr>
<tr>
<td>748</td>
<td>63 50 50 45 45 45 46 46 - - - - - - -</td>
</tr>
</tbody>
</table>

### TABLE XXIV
**Algorithm performance-related experimental results for EGN$_{EIE}$ second version with Rosenbrock generalized problem. Execution times for manager and workers.**

<table>
<thead>
<tr>
<th>M.</th>
<th>Wkrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>768</td>
<td>398 - - - - - - -</td>
</tr>
<tr>
<td>566</td>
<td>273 272 - - - - - - -</td>
</tr>
<tr>
<td>491</td>
<td>209 210 211 - - - - - - -</td>
</tr>
<tr>
<td>445</td>
<td>176 171 172 171 - - - - - - -</td>
</tr>
<tr>
<td>410</td>
<td>157 146 146 146 145 - - - - - - -</td>
</tr>
<tr>
<td>389</td>
<td>138 132 128 128 128 126 - - - - - - -</td>
</tr>
<tr>
<td>373</td>
<td>124 118 119 113 113 113 - - - - - - -</td>
</tr>
<tr>
<td>366</td>
<td>113 107 107 102 102 102 102 102 102 102 - - - - - - -</td>
</tr>
<tr>
<td>355</td>
<td>110 96 98 94 93 94 94 94 94 95 - - - - - - -</td>
</tr>
</tbody>
</table>
VI. CONCLUSIONS AND FUTURE WORK

In this paper several parallel solutions have been proposed for different EDAs. For each one, a previous study of the execution times of the procedures that conform the sequential algorithm was made. In most of the cases, the step that requires the most execution time is the learning phase, where a probabilistic graphical model is induced from the database of individuals. In the final section, we have seen that, depending on the algorithm, there may be other steps that also take up much computational time: for example, the creation of new individuals once the structure has been learnt.

Different parallel approaches have been developed for each sequential algorithm, mixing MPI and thread techniques. The experiments have been made using a cluster of ten two-processor computers, changing the number of nodes from two to ten (MPI communication) where in each node two threads have been created.

Looking at the obtained results, it can be seen that parallelization of the learning phase notably improves the performance of the algorithms. This suggests that applying parallelization techniques to EDAs to solve complex problems can bring them nearer to practical use. However, it is important to realize that in a practical situation not only the parallelization of the learning or sampling phases has to be taken into account but, the parallelization of the objective function. Moreover, if an accurate and efficient solution is wanted, it would be necessary to find a specific parallelization of the objective function in used.

Finally, we have began a deeper study of the advantage that supposes mixing MPI and threads, measuring different times and characteristics (communication times, synchronization, work distribution) of the resulting algorithm combined with more complex problems.

REFERENCES


