Implementation and performance evaluation of a Parallelization of Estimation of Bayesian Network Algorithms

A. Mendiburu [a,1] J. Miguel-Alonso [a,2] J.A. Lozano [b,1]

[a] *The University of the Basque Country, Department of Computer Architecture and Technology, San Sebastian, Spain*

[b] *The University of the Basque Country, Department of Computer Science and Artificial Intelligence, San Sebastian, Spain*

Abstract

This paper presents, discusses and evaluates parallel implementations of a set of algorithms designed for optimization tasks: Estimation of Bayesian Network Algorithms (EBNAs). These algorithms belong to the family of Evolutionary Computation.

Two different APIs have been combined: message passing and threads, with the aim of obtaining good performance levels in a wide range of parallel machines. Our approach has been to analyze the most computationally intensive sections of a sequential implementation of EBNAs, and then parallelize those sections, using a master-worker design pattern. This way the resulting program exhibits exactly the same behavior of the sequential one, but runs faster.

To evaluate our proposal, we have chosen a complex scenario where EBNAs can be applied: Feature Subset Selection (FSS) for supervised classification problems. For the experiments, three different computing systems have been tested: two different clusters built from commodity components, and an 8-way multiprocessor. Programs have been executed on the target machines using different combination of message-passing processes and threads. Achieved performance is excellent, with efficiency around 1.

These encouraging results do widen the spectrum of problems (and problem sizes) that can be solved, in reasonable times, with EBNAs.

Key words: Estimation of Bayesian Network Algorithms, Parallel Computing, Estimation of Distribution Algorithms, Message Passing Interface, Threads, Feature Subset Selection

1 Introduction

Evolutionary Computation [1] comprises a collection of techniques with the purpose of finding suitable solutions to complex optimization problems. Among these are Estimation of Distribution Algorithms (EDAs) [2,3]. Like most Evolutionary Computation heuristics, EDAs maintain at each step a population of individuals but, instead of using crossover and mutation operators to make population evolve, EDAs learn a probability distribution from the set of selected individuals and sample this distribution to obtain the new population. The family of EDAs includes many different algorithms, depending on the complexity of the probabilistic model that is learnt. Usually, when dealing with very complex problems, the algorithms that obtain the best results are those using unrestricted probability models. An instance of EDAs that use unrestricted probability models are EBNAs, which use Bayesian networks to codify the probability distribution. The use of this kind of probability models requires large computation times in complex problems: from hours to days using current commodity personal computers. This is clearly excessive if the searched-for solution is needed in a real environment.

An obvious way of dealing with this issue is applying more computing power: faster machines, parallel machines. Clusters made with commodity components are of particular interest, because they are cheap and easy to build.

Some work has been done in this area. In [4] two different parallel proposals for an algorithm that uses probabilistic graphical models in the combinatorial field ($EBNA_{BIC}$) are proposed. This approach requires communication via shared data structures, so it is not directly useful in the context of distributed memory computers (including clusters). Also relevant are the works [5,6] where two different parallel approaches are presented for another algorithm of the EBNA family, the Bayesian Optimization Algorithm (BOA) [7]: one uses a pipelined parallel architecture, and the other uses a cluster.

We propose a new parallel implementation of sequential EBNAs. The functionality of the sequential version is preserved, using parallel processing to accelerate some portions of the algorithm: those consuming the largest part of the execution time. Particularly, we focus on the $EBNA_{BIC}$ algorithm; this way, we extend the work in [5,6], allowing the learning of the Bayesian network without taking into account order restrictions. Moreover, in the im-

Email addresses: amendiburu@si.ehu.es (A. Mendiburu), miguel@si.ehu.es (J. Miguel-Alonso), lozano@si.ehu.es (J.A. Lozano).

[1] This work has been done with the support of the University of the Basque Country (9/UPV 00140.226-15334/2003)

[2] This work has been done with the support of the Spain's MCyT (TIC 2001-05950-C02-02) and the Diputacion Foral de Gipuzkoa (DF 758—2003)

plementation we use two different programming interfaces (APIs): Message Passing Interface (MPI) [8] and POSIX threads [9]. It is possible to tune the program to use only MPI, only threads, or a combination of both. Different mixtures have been tested in order to choose the most satisfactory one for a given computing environment. The achieved performance levels are up to (or even over) our expectations: efficiency is always around 1, due to the successful workload distribution and the small communication and synchronization requirements.

The rest of this paper is organized as follows. A brief description of EDAs -and particularly EBNAs- and their characteristics can be seen in Section 2. Section 3 shows the sequential algorithm and the analysis of the execution times. In Section 4 we present a detailed description of the phases of the sequential version that are susceptible to be parallelized. Section 5 presents the proposed parallel implementation for those phases. The performance of the parallel approaches is discussed in Section 6. Finally, conclusions and future work are presented in Section 7.

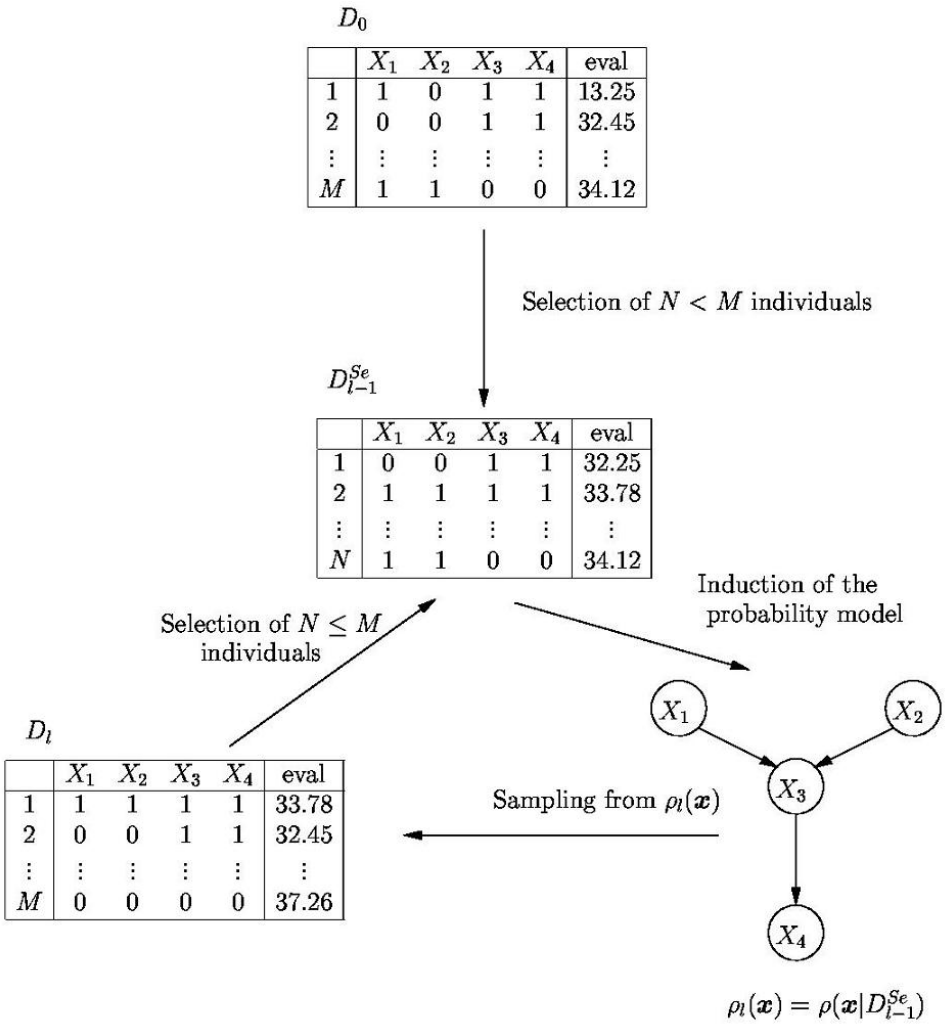
2 Estimation of Distribution Algorithms

EDAs constitute a family of algorithms designed to solve optimization problems. They were introduced in the field of Evolutionary Computation in [2], although similar approaches can be previously found in [10]. The general pattern is very similar to that of other Evolutionary Computation algorithms:

(1) Generate an initial population D_0 of M individuals and evaluate each of them. An individual codifies a possible solution to the problem and it is usually represented as a vector of variables: $X = (X_1, X_2, \ldots, X_n)$.
(2) N individuals are selected from the set of M, following a given selection method.
(3) Learn a probability model from the set of selected individuals.
(4) Finally, a new population of M individuals is generated based on the sampling of the probability distribution learnt in the previous step and, these individuals are evaluated.

Steps 2, 3 and 4 are repeated until some stop criterion is met (e.g., maximum number of generations, homogeneity of the population, or no improvement after a certain number of generations). A diagram of this process (for $n = 4$) is shown in Figure 1.

With regard to the third step, there are many different ways to learn the probability model. Clearly, the probability model learnt at each step has an important influence on the behavior of a particular EDA (in terms of compu-



	X_1	X_2	X_3	X_4	eval
1	1	0	1	1	13.25
2	0	0	1	1	32.45
⋮	⋮	⋮	⋮	⋮	⋮
M	1	1	0	0	34.12

	X_1	X_2	X_3	X_4	eval
1	0	0	1	1	32.25
2	1	1	1	1	33.78
⋮	⋮	⋮	⋮	⋮	⋮
N	1	1	0	0	34.12

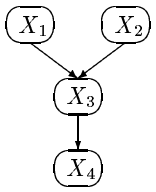
	X_1	X_2	X_3	X_4	eval
1	1	1	1	1	33.78
2	0	0	1	1	32.45
⋮	⋮	⋮	⋮	⋮	⋮
M	0	0	0	0	37.26

Fig. 1. EDA approach to optimization.

tational complexity and quality of the obtained results). Different algorithms can be designed using different models: dependency chains, tree structures, Bayesian networks, and so on. For detailed information about the characteristics of the family of EDAs, see [3,11].

Among the algorithms that use Bayesian networks to codify the probability distribution representing the selected individuals, we can point out BOA [7], Learning Factorized Distribution Algorithm (LFDA) [12], $EBNA_{BIC}$ [3] or $EBNA_{K2+pen}$ [3]. The differences between them lie on the restrictions taken into account during the creation of the Bayesian network, and on the score used to measure the quality of the obtained structure.

A Bayesian network [13] is a pair $(S, \boldsymbol{\theta})$ where S is a directed acyclic graph (DAG) that represents the factorization of the joint probability distribution and $\boldsymbol{\theta}$ is a set of parameters used to define the local probability distributions associated to each node of the structure. Figure 2 depicts an example. Struc-



Structure

Local probabilities

$\boldsymbol{\theta_1} = (\theta_{1-1}, \theta_{1-2})$ $\quad p(x_1^1), p(x_1^2)$

$\boldsymbol{\theta_2} = (\theta_{2-1}, \theta_{2-2}, \theta_{2-3})$ $\quad p(x_2^1), p(x_2^2), p(x_2^3)$

$\boldsymbol{\theta_3} = (\theta_{311}, \theta_{321}, \theta_{331},$ $\quad p(x_3^1|x_1^1, x_2^1), p(x_3^1|x_1^1, x_2^2), p(x_3^1|x_1^1, x_2^3),$

$\theta_{341}, \theta_{351}, \theta_{361},$ $\quad p(x_3^1|x_1^2, x_2^1), p(x_3^1|x_1^2, x_2^2), p(x_3^1|x_1^2, x_2^3),$

$\theta_{312}, \theta_{322}, \theta_{332},$ $\quad p(x_3^2|x_1^1, x_2^1), p(x_3^2|x_1^1, x_2^2), p(x_3^2|x_1^1, x_2^3),$

$\theta_{342}, \theta_{352}, \theta_{362})$ $\quad p(x_3^2|x_1^2, x_2^1), p(x_3^2|x_1^2, x_2^2), p(x_3^2|x_1^2, x_2^3)$

$\boldsymbol{\theta_4} = (\theta_{411}, \theta_{421}, \theta_{412}, \theta_{422})$ $\quad p(x_4^1|x_3^1), p(x_4^1|x_3^2), p(x_4^2|x_3^1), p(x_4^2|x_3^2)$

Factorization of the joint mass-probability

$$p(x_1, x_2, x_3, x_4) = p(x_1)p(x_2)p(x_3|x_1, x_2)p(x_4|x_3)$$

Fig. 2. Structure (S), local probabilities and resulting factorization for a Bayesian network with four variables (X_1, X_3 and X_4 with two possible values, and X_2 with three possible values.)

$EBNA_{BIC}$ algorithm. Sequential version.

Step 1. **for** $i = 1, \ldots, M$
create a new individual, X_i=(X_i^1, X_i^2,...,X_i^n)
evaluate X_i
Step 2. select the best N individuals
Step 3. learn a Bayesian network from the previously selected individuals
Step 4. generate a new population by sampling the Bayesian network
evaluate each individual
Step 5. **go to** Step 2 until a stop criterion is met

Fig. 3. Pseudocode for the sequential $EBNA_{BIC}$ algorithm.

ture S shows the dependencies between the nodes and gives the factorization and parameter θ_{1-1}, for instance, represents the probability of variable X_1 to take its first value. Detailed information about Bayesian networks can be consulted in [13].

In this paper, we focus on $EBNA_{BIC}$ algorithm. However, this work can be easily adapted to other EBNAs -like $EBNA_{K2+pen}$-, that use different scores. Previous work about different parallel approaches of other EDA algorithms can be consulted in [14].

3 Analysis of the Sequential Implementation

In order to check if the parallelization of an algorithm is viable, its behavior must be carefully studied. In Figure 3 the pseudocode of the $EBNA_{BIC}$ algorithm is shown. It can be observed that, once the initial population is created, steps 2 and 3 are repeated until the algorithm converges.

As we have stated before, this kind of algorithms have been designed to solve complex optimization problems. Depending on the characteristics of the particular target problem, some steps of the algorithm may require more execution time than others. For example, the longer the number of variables that constitute the individual $X = (X_1, X_2, \ldots, X_n)$, the harder the creation of the Bayesian network. Other aspect is the complexity of the fitness function used to evaluate each individual.

We have studied the behavior of a sequential version of the $EBNA_{BIC}$ algorithm when solving two very different, but representative problems: $OneMax$ and Feature Subset Selection (FSS) [15]. $OneMax$ problem is the optimization of the following function, defined as:

$$OneMax(\boldsymbol{x}) = \sum_{i=1}^{n} x_i$$

where $x_i \in \{0, 1\}$.

The best individual is that with $x_i = 1$ for all i values. The evaluation of the objective function is straightforward and therefore its computational cost is insignificant. Two different individuals sizes have been selected: 150 and 250.

FSS is used in Data Mining to look for a subset of features in a supervised classification problem. The fitness function is quite complex since to evaluate a candidate solution (individual) several classifiers must be created. More detailed information about this problem is provided in Section 6. We have evaluated the sequential program with two well-known datasets: "Arrythmia" and "Internet Advertisement" [16].

In Table 1, the cost of the most important steps of the algorithm is shown. It can be observed that, when evaluation functions have a small computational cost ($OneMax$), the learning of the Bayesian network consumes almost all the execution time. Otherwise, when the evaluation function is more complex, the execution percentages are shared between the learning of the Bayesian network and the "sampling and evaluation" of the new population.

Table 1
Profiling of the sequential implementation of $EBNA_{BIC}$ for different problems: proportion of execution time used by the learning and "sampling and evaluation" phases.

Problem	**Indiv. sizes**	**% Learning**	**% Sampl. and Eval.**
$OneMax$	150	98.6	0.4
$OneMax$	250	99.4	0.1
$FSS_{Arrhythmia}$	279	82.6	17.3
$FSS_{InternetAdv.}$	1558	58.6	41.3

4 Description of the Sequential Algorithm

Once the more computationally expensive phases have been detected (learning and "sampling and evaluation"), a detailed study of their characteristics is necessary to obtain the parts that can be parallelized. In the following subsections both phases are presented, and in Section 5 the parallel alternatives are explained.

4.1 The learning phase

Once the population is selected, a Bayesian network is learnt from it. In EBNA, a greedy approach is used to learn the structure of the Bayesian network. Each possible network structure will be assigned a score that represents its goodness for the current population. The search will be done adding or deleting edges to the existing Bayesian network when this addition or deletion implies a better score.

Obviously, the score used during this process plays an important role in the algorithm, as it conditions the obtained Bayesian network. EBNA_{BIC} uses the penalized maximum likelihood score denoted by BIC (Bayesian Information Criterion) [17]. Given a structure S and a dataset D (set of selected individuals), the BIC score can be defined as:

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

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 (those from which an arc exists towards X_i), $\boldsymbol{Pa_i}$, can take.
- N_{ij} is the number of individuals in D in which variables $\boldsymbol{Pa_i}$ 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 $\boldsymbol{Pa_i}$ 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 a local BIC score ($BIC(i, S, D)$) associated to it:

$$BIC(S, D) = \sum_{i=1}^{n} BIC(i, S, D) \tag{2}$$

where

$$BIC(i, S, D) = \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log \frac{N_{ijk}}{N_{ij}} - \frac{1}{2} \log(N) q_i (r_i - 1) \tag{3}$$

At each step, an exhaustive search is made through the whole set of possible arc modifications. An arc modification consists of adding or deleting an arc from the current structure S. The modification that maximizes the gain of the BIC score is used to update S, as long as it results in a DAG structure. 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 to update the global score.

The previous 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 score 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 the remaining steps, only changes to the BIC score 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 $n - 2$.

The following data structures are used to implement the algorithm:

- A vector $BIC[i]$, $i = 1, 2, \ldots, n$, where $BIC[i]$ stores the local BIC score of

Learning algorithm. Sequential version.

Input: D, S, *paths*
Step 1. **for** $i = 1, \ldots, n$ calculate $BIC[i]$
Step 2. **for** $i = 1, \ldots, n$ and $j = 1, \ldots, n$ $G[j, i] = 0$
Step 3. **for** $i = 1, \ldots, n$ and $j = 1, \ldots, n$
if $(i \neq j)$ calculate $G[j, i]$ /* the change of the BIC produced by the arc modification (j, i) */
Step 4. find (j, i) such that $paths[i, j] = 0$ and $G[j, i] \geq G[r, s]$ for each $r, s = 1, \ldots, n$ such that $paths[s, r] = 0$
Step 5. **if** $G[j, i] > 0$
update S with arc modification (j, i)
update *paths*
else stop
Step 6. **for** $k = 1, \ldots, n$
if $(k \neq i$ **and** $k \neq j)$ calculate $G[k, i]$
Step 7. **go to** Step 4

Fig. 4. Pseudocode for the sequential structural learning algorithm.

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 $G[j, i]$, $j, i = 1, \ldots, n$, where each (j, i) entry represents the gain or loss in score associated with the arc modification (j, i).
- 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 data 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, $paths[i, j] = 0$.

The pseudocode for the sequential structure learning algorithm is shown in Figure 4.

4.2 The "sampling and evaluation" phase

Once we have represented the desired probability distribution as a Bayesian network, new individuals must be generated by means of the joint probability distribution encoded by the network. Individuals are generated by sampling directly from the Bayesian network using an adaptation of the Probabilistic Logic Sampling algorithm (PLS) [18]. This sampling process is repeated, creating and evaluating an individual at each iteration, until all new individuals have been created.

"Sampling and evaluation" algorithm. Sequential version.

Input: $Order$, $Probs$
Step 1. **for** each i in $NewPopSet$
for $j = 1$ to n obtain X_i^j
evaluate the individual
add the individual to the population

Fig. 5. Pseudocode for the sequential "sampling and evaluation" algorithm.

Before starting the sampling process, it is necessary to set some data structures: $Order$ and $Probs$. From the Bayesian network we establish an order between the variables, being always child variables preceded by their parent variables. This information is stored in the $Order$ data structure. Probabilities are calculated and stored in the $Probs$ data structure, using the learnt structure and the current population. For a more detailed description of this algorithm, see [3].

A pseudocode for this "sampling and evaluation" process is shown in Figure 5.

5 The Parallel Algorithm

The use of parallel computers to solve a problem requires new programs that, explicitly or implicitly, are adapted to their execution in a collection of processors, instead of only one. Common wisdom tells us that the best way to take advantage of parallel computers is to design parallel algorithms, that is, to think about a group of entities collaborating to solve the problem, instead of just one. This usually means looking for new, alternate ways of solving the problem.

A more conservative, but often successful way of using parallel computers, is to stick to the sequential algorithm, using the power of many processors to accelerate portions of it. This can be done automatically, using parallelizing compilers; however, the ability of these compilers to extract parallelism is still limited. A manual reimplementation that takes into account the particular characteristics of the algorithm, and the capabilities of the target parallel computer, can be very efficient for many problems. As we are attesting in this paper, EBNAs are one of such problems.

We already stated that our goal is to accelerate a sequential implementation of $EBNA_{BIC}$, focusing in the reimplementation of the most computational intensive phases. The advantage of this approach is that, if the sequential algorithm is considered correct, the parallel will be correct too: we are neither

defining a new paradigm for the implementation of EBNAs, nor a new algorithm in this family. The disadvantage is that we will not be able to fully exploit the available parallelism, because some portions of the problem have to be solved sequentially. However, if this portion is small (less than 2%) we still have plenty of room for acceleration.

We have chosen a well-known design pattern for parallel programming: the master-worker paradigm. The master runs the sequential parts. When it reaches one of the two costly phases, it distributes parts of its workload among a collection of workers. Then, it collects and summarizes the partial results, and continues normal operation. Sometimes, depending on the characteristics of the work that must be completed, it is interesting to design the master in such a way that it can make use of the idle time. While waiting for the workers to finish, the master completes part of the work just as another worker. This approach is particularly useful in small-scale parallel computers.

This paradigm can be easily implemented using threads inside a computer (common data structures are used to communicate master with workers, the usual primitives are used to synchronize the system) or message passing between processes in the same or in different computers (message passing combine information exchange with synchronization). We have chosen to use two widely-known programming interfaces for the implementation: POSIX threads and Message Passing Interface (MPI). The selected programming language has been C++.

Our target machines range from small-size, shared memory multiprocessors to cheap clusters of single-processor nodes, being a cluster of multiprocessors a frequent platform in many current research centers. Taking this into consideration, our implementation combines both APIs (threads, message passing), and the user is responsible of choosing the right combination for his particular machine. In Section 6 we provide a performance evaluation of several of the possible combinations, for different machines.

5.1 Parallelization of the learning phase

We have explained before that the learning phase (creation of the Bayesian network) is a greedy process where each network will be measured using the BIC criterion. This criterion is decomposable, so each worker (master included) will compute a subset of variables returning the results to the master, which then actualizes the structure of the Bayesian network applying the arc addition/deletion that improves the score the most. This process is repeated until the best structure is obtained.

From the point of view of implementation, the issue of data consistency is of

major importance. It must be taken into account that each worker could be in a different machine (MPI implementation), so data structures must be sent and updated in each node that takes part in the execution. For example, we have said that the master changes the Bayesian network in terms of the better score. Once this has been done, workers need to know this in order to maintain a correct structure. There are different ways to provide this information to the workers. An option is to send all the structure each time the Bayesian network changes; another is to send only the arc that has been modified indicating when it has been deleted or added. After some tests we have adopted the second option. Other structures also need to be actualized after each master's modification.

With regard to the execution schema, once initialization has been completed, workers wait for an job request. Two different types of job requests can be received: (1) to compute the initial BIC scores for their respective set of variables -each worker receives a chunk of variables to work with- and (2) to update the BIC scores and maintain the integrity of the local structures -each node addition or deletion means that the master must notify the workers that the (i, j) edge has changed and they must update their own copy of the network as well as BIC scores-.

The previous description assumes the use of MPI. However, the user can request the use of several threads inside each MPI process. If this is the case, the specified number of worker threads are pre-forked when the process starts. When a worker receives the request to perform a BIC computation for a chunk of variables, it can distribute this workload among its set of threads. As we can see, an MPI-worker behaves as a master for its set of worker threads. Obviously, communication inside a process is performed via shared variables. The whole schema can be seen as a two-level parallelization, where communication between computers is carried out using MPI and communication inside each computer is either via MPI or using a collection of variables shared by a set of threads.

Figure 6 (master) and Figure 7 (workers) show the pseudocode for master and workers.

5.2 Parallelization of the "sampling and evaluation" phase

The "sampling and evaluation" phase may require a considerable amount of time. In Table 1 it can be observed that, for FSS problems, this phase can require as much computation time as the learning phase. The percentage of each one depends on the size of the individual and on the characteristics of the particular problem. Therefore, we also propose to parallelize the "sam-

Learning algorithm. Parallel version. Master.

Input: D, S, $paths$
Step 1. send D to the workers
set the number of variables ($NSet$) to work with
Step 2. send "calculate BIC" order to the workers
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
calculate the structures that fit the DAG property
set the number of variables ($NSetDAG$) to work with
send "calculate $G[k, i]$" order to the workers
send to the workers the set of variables ($NSetDAG$) to work with
Step 6. receive from workers all the changes and update G
Step 7. find (j, i) such that $paths[i, j] = 0$ and $G[j, i] \geq G[r, s]$
for each $r, s = 1, \ldots, n$ such that $paths[s, r] = 0$
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
set the number of variables ($NSetDAG$) to work with
send to the workers "calculate $G[k, i]$ for (i,j)" order
send to the workers the set of variables ($NSetDAG$) to work with
Step 10. receive from workers all the changes and update G
Step 11. **go to** Step 7

Fig. 6. Parallel structural learning phase. Pseudocode for the master.

pling and evaluation" phase: each worker samples and evaluates its corresponding amount of individuals. The parallelization approach is as follows: being $NewPopSet$ the amount of new individuals to be created and *comp* the number of computational nodes, the master sends to each worker the order to create $NewPopSet/comp$ new individuals (the master also creates new individuals). Once all these new individuals have been created, they are sent to the master, which adds them to the population and continues with the algorithm.

In our experiments we have always operated with sets of identical computers; this is the reason we assign the same (or similar) number of computations to each worker. In other situations, where the parallel computer is heterogeneous, it would be better to use an on-demand schema, where only one piece of work is sent to each worker and, when it is completed, another one is sent, until all the computations have been terminated. This alternative needs more communication than the one we use. Yet another possibility could be to previously

Learning algorithm. 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 to the master BIC results
"calculate $G[k,i]$"
receive the set of variables ($NSetDAG$) to work with
for each variable k in $NSetDAG$
calculate $G[k,i]$
send G modifications to the master
"calculate $G[k,i]$ for (i,j)"
receive the set of variables ($NSetDAG$) to work with
for each variable k in $NSetDAG$
calculate $G[k,i]$
send G modifications to the master
"change arc (j,i)"
Update S with (j,i) arc modification
"stop" stop
Step 4. **go to** Step 2

Fig. 7. Parallel structural learning phase: Pseudocode for the workers.

"Sampling and evaluation" algorithm. Parallel version. Master.

Input: $Order$, $Probs$
Step 1. send $Order$ and $Probs$ structures to the workers
Step 2. set the number of individuals ($NewPopSet$) to create
Step 3. send $NewPopSet$ to the workers
Step 4. receive the new individuals from the workers
update the population

Fig. 8. Pseudocode for the parallel "sampling and evaluation" phase for the master.

measure the computation capacity of each computer and to assign to each of them a workload adapted to its capacity.

The algorithms for the parallel "sampling and evaluation" phase are shown in Figure 8 (master) and Figure 9 (workers).

"Sampling and evaluation" algorithm. Parallel version. Workers.

Step 1. receive *Order* and *Probs* structures
Step 2. receive the number of individuals ($NewPopSet$) to create
Step 3. **for** $i = 1, \ldots, NewPopSet$
    create a new individual
    evaluate the new individual
Step 4. send the new individuals to the master

Fig. 9. Pseudocode for the parallel "sampling and evaluation" phase for the workers.

6 Performance evaluation

In this section we evaluate our proposal of parallel implementation of $EBNA_{BIC}$, applied to some realistic problems. For the evaluation, we have focused on the well-known Feature Subset Selection (FSS) problem in supervised classification [15]. The basic problem of supervised classification in Data Mining is related to the induction of a model that classifies a given object into one of several pre-defined classes. In order to induce the classification model, each object is described by a pattern of d features. Usually, objects have many features and data files are large. Due to this characteristic, the following question is formulated by researchers: "Are all the features useful for learning the classification model?". The FSS approach to this question could be presented as follows: "given a set of candidate features, select the best subset that really matters for constructing the classifier". Different kinds of algorithms have been presented in order to solve the FSS problem, including EDAs [19,20].

In order to use EDAs, the problem must be characterized as an optimization one. As we have said before, the goal is to select, from the set of features that describe an object, the subset that provides the best classifier. Therefore, we define an individual as $X = (X_1, X_2, \ldots, X_n)$, being n the number of features. Each variable (X_i) of the individual can take two possible values: 0 to indicate that the i^{th} feature has not been selected or 1 to indicate that it has been selected. Each individual is evaluated measuring the accuracy of the classifier obtained starting from the data cases and using only the selected variables for the individual. Usually, the quality of a classifier is measured in terms of the percentage of rightly classified instances (accuracy percentage). A good review of FSS methods can be found in [15].

In [19], EDAs are proposed to solve the FSS problem, however signifying the impossibility of using some potentially interesting EDAs (those using Bayesian networks to learn the probability distribution of the selected cases) when problems have a large number of attributes (close to one hundred or more), due to the long computation times required. Therefore, our goal is to use our fast,

parallel implementation of the $EBNA_{BIC}$ algorithm for this type of problems, demonstrating that, in fact, they can be used if enough computing power is available. As the target problem, we will use "Internet Advertisements", a middle-large size problem from the UCI Machine Learning Repository [16].

It must be pointed out that among the results presented in this paper we neither show accuracy percentages, nor other parameters that indicate the quality of the obtained classifier. Our aim in this paper is to focus only on the quality of our parallel solution in terms of execution speed, trying to extract conclusions from the behavior of the developed parallel algorithm. In this way, we prove that this parallel version of $EBNAs$ can be applied to FSS problems of medium to large sizes, extending the work in [19].

"Internet Advertisements" database has 1,558 features, 2 classes, and 3,279 instances. In consequence, the size of the individual is 1,558 and each variable has two possible values (0 or 1, that is, not selected or selected). For the $EBNA_{BIC}$ algorithm, we have fixed the rest of the parameters as follows: size of the population is 2,000, size of the selected population is 1,000 and 1,999 new individuals are sampled and evaluated in each new generation. These are common recommended values [3].

With regard to the fitness function, each individual will be evaluated using Naive Bayes, a simple but good supervised classification algorithm. In order to approximate our computation times to a real problem, a 10-fold cross-validation is used to estimate the accuracy of the classifier (fitness function). This is to maintain the same fitness function chosen in the previously mentioned works [19,20].

Three different machines have been available for experimentation. Two of them are clusters, and the other is a 8-way multiprocessor. The main characteristics of these machines are presented below:

Athlon Cluster: A cluster of 8 nodes. Each node has two AMD Athlon MP 2000+ processors (1.6GHz), with 512KB of cache memory each and 1GB of (shared) RAM. Operating system is Linux. C++ compiler is version 3.3.2 of GNU's gcc. MPI implementation is LAM (version 6.5.9). Nodes are interconnected using a switched Gigabit Ethernet network.

Xeon Cluster: A cluster of 4 nodes. Each node has two Intel Xeon processors (2.4GHz, hyper-threading enabled), with 512KB of cache memory each and 2GB of (shared) RAM. Operating system is Linux. MPI implementation is LAM (version 7.0.2). C++ compiler is version 8 of Intel's icc. Nodes are interconnected using a switched Gigabit Ethernet network. From the operating system point of view, each physical processor is seen as a dual processor (Linux actually presents each node as a 4-way multiprocessor). This is the way the operating system takes advantage of hyper-threading.

Table 2
Arrangement of processes/threads per node in the machines used in the experiments.

machine	*MPI version*	*MPI&Threads version*
Athlon	2 single-threaded processes	1 process with 2 threads
Xeon	4 single-threaded processes	1 process with 4 threads
Altix	8 single-threaded processes	1 process with 8 threads

Altix: A SGI Altix 3300 system with 8 Intel Itanium-2 processors (1.3GHz), 3MB of cache memory each and 8GB of shared RAM. Operating system is Linux. MPI implementation is MPICH. C++ compiler is version 8 of Intel's icc. Processors use a proprietary SGI interconnection network that provides, at hardware level, a memory space shared among all processors. Message passing is also very efficiently supported.

Results are expressed in terms of the execution times of the different versions of the programs. For the comparisons to be fair, a common termination criterion must be established. We have decided to end programs after a fixed number of generations: 4.

Each experiment has been repeated 10 times. For the chosen termination criterion, variations of execution times are very small: the observed deviation is less than 2%; therefore, we consider that 10 executions are representative enough. In all tables we collect the mean of the 10 obtained values.

Experiments have been made using different combinations of parallel programming paradigms:

MPI: All communications (either intra-node or inter-nodes) are performed using only MPI.

MPI&Threads: A process is spawned at each node. Communication and synchronization between processes is done via MPI. Each process creates a collection of threads that use shared variables to communicate and synchronize.

The baseline experiments for performance comparison have been performed using the sequential program in one node of each one of the available machines. For the parallel experiments, the arrangement of processes and threads in the machines depends on the machine's characteristics. Details are provided in Table 2.

As explained before, workload distribution among processes is done by allotting "chunks" of work to each of them. In the case of using threads (which are pre-forked when process starts), an on-demand schema is used: threads await for a piece of work, execute it, return results to the master thread, and then wait for another piece.

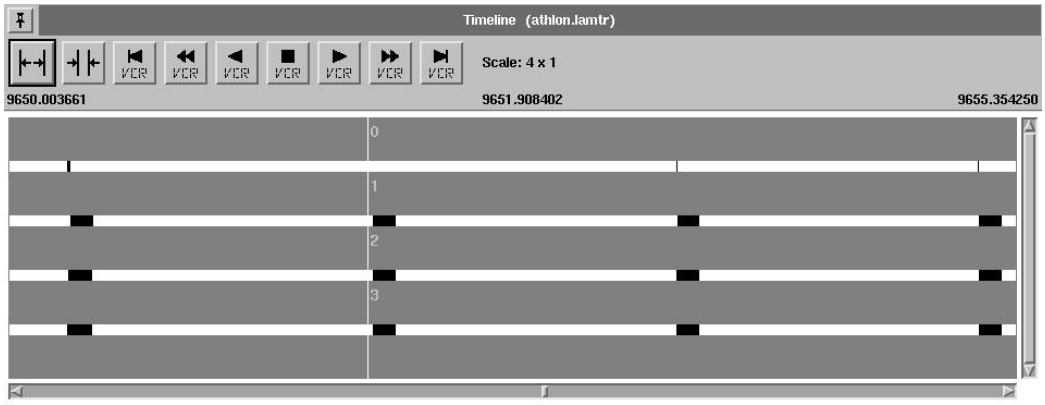


Fig. 10. Trace of the learning process, for the experiment with 4 CPUs in the Athlon cluster.

In terms of measurements of the experiments, we have focused on two interrelated values: efficiency of the parallel program, and load distribution among processes. We have also obtained some traces of the program's execution to illustrate the blocking times caused by message interchange.

6.1 Efficiency

The main goal of a parallel algorithm is to reduce the execution time as much as possible. In addition, scalability is also a highly desirable property, because it allows efficient execution even for a large number of processes.

We have measured the performance of our program in terms of speed-up (execution time of the sequential program / execution time of the parallel program) and efficiency (speed-up / number of processors), when running in the different platforms available. Table 3 summarizes the obtained results.

Taking into account that our parallel algorithm follows the same steps of the sequential version and, thus, only some sections of the program can be parallelized, the efficiency range of our programs (0.84-1.09) is very good. This has been achieved because workload distribution is even and communication and synchronization are carefully tuned. Figure 10 shows a portion of the learning process where master and workers look for a new arc addition/deletion that improves the BIC score; communication and synchronization periods (those in dark color) are really short compared to those periods of useful work (those in light color).

The difference between the pure MPI and MPI&Threads versions in the Xeon cluster is due to the hyper-threading technology. Intel has optimized their processors (and compilers) to efficiently support the concurrent execution of several threads, and the MPI&Threads version of our program takes advan-

tage of this. A rule-of-thumb would be that, in the absence of mechanisms to accelerate thread execution, a pure MPI version of the program is capable of obtaining good levels of performance in most parallel machines.

It could be surprising to see efficiencies over 1. There are several causes of this super-linear speed-up. Some of those are:

- A parallel computer does not only add CPU power: each CPU has its own, fast cache memory. A parallel program can store more of its data structures in the caches than its sequential counterpart, thus improving execution speeds.
- Parallel and sequential programs are not exactly the same. Different program structures can allow the compiler to perform different optimizations that have impact in execution times.

These sources of acceleration compensate the overheads imposed by communication and synchronization.

6.2 Load distribution

A good load distribution is essential to achieve good levels of efficiency and scalability. As all parallel processes work together to execute a particular program, the work that each one completes must be equally distributed, avoiding great differences between them, because the slowest process determines the overall execution time. Sometimes -like in this situation-, the work cannot be exactly distributed between the processes (perfect division) due to the characteristics of the problem to be solved. However, given the large number of tasks to be distributed, this difference is almost unnoticeable.

Table 4 shows the execution times (actual CPU usage) for the different scenarios and implementations of our program. As expected, the master process (which is in charge of all the sequential part, the distribution of the workload and the collection of partial results) uses a larger portion of CPU time. However, distribution is very satisfactory, because (1) the extra CPU-time required by the master is not very significant, and (2) distribution of workload among workers is also fairly even. This means that our choice of phases to parallelize, and the approach used to implement them, has been successful in fulfilling our design objectives.

Table 3
Measured performance for the different machines and implementations. Total execution time as well as speed-up and efficiency is shown.

Athlon Cluster

CPUs	*MPI version*			*MPI&Threads version*		
	Exec. time	*Speed-up*	*Efficiency*	*Exec. time*	*Speed-up*	*Efficiency*
Seq.	52h 34' 25"	-	-	-	-	-
2	24h 18' 32"	2.16	1.08	25h 7' 48"	2.09	1.05
4	12h 19' 31"	4.27	1.07	12h 50' 17"	4.10	1.02
8	6h 6' 30"	8.61	1.08	6h 26' 48"	8.16	1.02
16	3h 18' 57"	15.86	0.99	3h 25' 5"	15.38	0.96

Xeon Cluster

CPUs	*MPI version*			*MPI&Threads version*		
	Exec. time	*Speed-up*	*Efficiency*	*Exec. time*	*Speed-up*	*Efficiency*
Seq.	21h 36' 17"	-	-	-	-	-
2	13h 9' 47"	1.64	0.82	9h 52' 33"	2.19	1.09
4	6h 44' 10"	3.21	0.80	5h 2' 33"	4.28	1.07
8	3h 32' 22"	6.10	0.76	2h 34' 48"	8.37	1.05

Altix Multiprocessor

CPUs	*MPI version*			*MPI&Threads version*		
	Exec. time	*Speed-up*	*Efficiency*	*Exec. time*	*Speed-up*	*Efficiency*
Seq.	41h 16' 2"	-	-	-	-	-
8	5h 29' 53"	7.51	0.94	6h 7' 40"	6.73	0.84

7 Conclusions and future work

In this paper a new parallel version of the $EBNA_{BIC}$ algorithm has been introduced and evaluated. This is one of the most computationally expensive algorithms inside the family of EDAs. It uses a Bayesian network to represent the probability distribution and therefore, even if good results are obtained, the required computation time is too high for complex problems.

Trying to obtain a faster version of this algorithm that does not change its functionality, a parallel implementation has been proposed. To ensure efficient execution in a range of low price parallel systems, this program can make use of two communication and synchronization APIs: pure MPI or a combination of

Table 4
Load distribution of the different experiments. Execution time for the master and the mean execution time and deviation for the workers

Athlon Cluster

CPUs	*MPI version*		*MPI&Threads version*	
	t. master	*t. worker±deviation*	*t. master*	*t. worker±deviation*
2	87,512	86,194 ± 372	90,468	89,637 ± 120
4	44,371	42,950 ± 639	46,217	45,038 ± 360
8	21,999	20,841 ± 98	23,209	22,127 ± 222
16	11,937	10,598 ± 124	12,305	11,101 ± 154

Xeon Cluster

CPUs	*MPI version*		*MPI&Threads version*	
	t. master	*t. worker±deviation*	*t. master*	*t. worker±deviation*
2	47,387	46,458 ± 233	35,553	35,070 ± 96
4	24,250	22,991 ± 316	18,153	17,515 ± 206
8	12,742	11,597 ± 172	9,288	8,644 ± 121

Altix Multiprocessor

CPUs	*MPI version*		*MPI&Threads version*	
	t. master	*t. worker±deviation*	*t. master*	*t. worker±deviation*
8	19,792	18,392 ± 312	22,060	21,104 ± 49

MPI and threads. Our program has been evaluated in three different parallel computers, obtaining excellent levels of performance (efficiency close to or even over 1). Arranging the program in threads is advantageous only in some particular cases: Intel Xeon processors, with hyper-threading technology.

In terms of future work, we have different lines opened. Now that execution times of EBNAs are shorter and, therefore, acceptable, we want to use them for FSS in complex problems; for example, classification of microarray data, where an individual is defined by many thousand of features. Also, continuing in the field of parallelization, our aim is to make use of the knowledge obtained from this work to design new parallel versions of other Evolutionary Computation algorithms, and to test them in different architectures, looking for the ideal computer and/or cluster configuration for a particular algorithm in terms of execution time and price.

References

[1] T. Bäck, Evolutionary Algorithms in Theory and Practice, Oxford University Press, 1996.

[2] H. Mühlenbein, G. Paaß, From recombination of genes to the estimation of distributions i. binary parameters, in: Lecture Notes in Computer Science 1411: Parallel Problem Solving from Nature - PPSN IV, 1996, pp. 178–187.

[3] P. Larrañaga, J. A. Lozano (Eds.), Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation, Kluwer Academic Publishers, 2002.

[4] J. A. Lozano, R. Sagarna, P. Larrañaga, Parallel estimation of distribution algorithms, in: P. Larrañaga, J. A. Lozano (Eds.), Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation, Kluwer Academic Publishers, 2002, pp. 129–145.

[5] J. Ocenasek, J. Schwarz, The parallel bayesian optimization algorithm, in: Proceedings of the European Symposium on Computational Intelligence, 2000, pp. 61–67.

[6] J. Ocenasek, J. Schwarz, The distributed bayesian optimization algorithm for combinatorial optimization, in: EUROGEN - Evolutionary Methods for Design, Optimisation and Control, CIMNE, 2001, pp. 115–120.

[7] M. Pelikan, D. E. Goldberg, E. Cantú-Paz, BOA: The Bayesian optimization algorithm, in: W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, R. E. Smith (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99, Orlando FL, Vol. 1, Morgan Kaufmann Publishers, San Francisco, CA, 1999, pp. 525–532.

[8] M. P. I. Forum, MPI: A message-passing interface standard, The International Journal of Supercomputer Applications and High Performance Computing 8 (3/4) (1994) 159–416.

[9] D. R. Butenhof, Programming with POSIX Threads, Addison-Wesley Professional Computing Series, 1997.

[10] A. A. Zhigljavsky, Theory of Global Random Search, Kluwer Academic Publishers, 1991.

[11] M. Pelikan, D. E. Goldberg, F. Lobo, A survey of optimization by building and using probabilistic models, Computational Optimization and Applications 21 (1) (2002) 5–20.

[12] H. Mühlenbein, T. Mahning, FDA - a scalable evolutionary algorithm for the optimization of additively decomposed functions, Evolutionary Computation 7 (4) (1999) 353–376.

[13] E. Castillo, J. M. Gutiérrez, A. S. Hadi, Expert Systems and Probabilistic Network Models, Springer-Verlag, New York, 1997.

[14] A. Mendiburu, J. A. Lozano, J. Miguel-Alonso, Parallel estimation of distribution algorithms: New approaches, Tech. Rep. EHU-KAT-IK-1-3, Department of Computer Architecture and Technology, The University of the Basque Country (2003).

[15] H. Liu, H. Motoda, Feature Selection for Knowledge Discovery and Data Mining, Kluwer Academic Publishers, 1998.

[16] C. Blake, C. Merz, UCI repository of machine learning databases (1998). URL `http://www.ics.uci.edu/~mlearn/MLRepository.html`

[17] G. Schwarz, Estimating the dimension of a model, Annals of Statistics 7 (2) (1978) 461–464.

[18] M. Henrion, Propagating uncertainty in bayesian networks by probabilistic logic sampling, in: J. F. Lemmer, L. N. Kanal (Eds.), Uncertainty in Artificial Intelligence 2, North-Holland, Amsterdam, 1988, pp. 149–163.

[19] I. Inza, P. Larrañaga, B. Sierra, Feature Subset Selection by Estimation of Distribution Algorithms, in: P. Larrañaga, J. A. Lozano (Eds.), Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation, Kluwer Academic Publishers, 2002, pp. 269–294.

[20] I. Inza, P. Larraaga, R. Etxeberria, B. Sierra, Feature subset selection by Bayesian networks based optimization, Artificial Intelligence 123 (1-2) (2000) 157–184.