A cache-based parallel genetic algorithm for the BDD variable ordering problem

Umberto S. Costa*†, Anamaria M. Moreira, David Déharbe

[1] Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Campus Universitário - Lagoa Nova
59072-970, Natal, RN, Brazil
{umberto,anamaria,david}@dimap.ufrn.br

Abstract—
Binary Decision Diagrams (BDDs) have proved to be a powerful representation for Boolean functions. Particularly, they are a very useful data structure for the symbolic model checking of digital circuits and other finite state systems, as well as other problems. Nevertheless, the size of the BDD representation of these functions is highly dependent on the order of the function arguments, also called variables, and to find such good ordering is an NP-Complete problem. Many heuristics have been proposed to solve this problem, as our Parallel Genetic Algorithm presented in [4], where we got excellent results in terms of parallelization. Now, we present a new version of our algorithm, this time with a cache system, together with experimental results.

Keywords— **binary decision diagrams (BDDs), genetic algorithms, parallel processing, symbolic model checking.**

I. Introduction

Binary Decision Diagrams, referred to only as BDDs from now on in this work, are a data structure with an associated set of handling algorithms that has been shown very useful to deal with propositional Boolean functions [2]. BDDs represent these functions by means of directed acyclic graphs. Along the paths of this graph, variables should be found in a fixed order. For instance, we show in Figure 1 the BDD for the function f = A.B + C, considering the variable ordering A, B, C.

Another important attribute of this data structure is the canonical representation of functions for each given variable ordering, that allows the execution of generally expensive tests, like equivalence and satisfiability, with unitary cost[2]. With BDDs, formal verification tools were able to handle finite-state systems with more than 10^{20} states[11].

Since many problems in the areas of digital logic design, verification and test, artificial intelligence and combinatorics can be expressed in terms of operation sequences on Boolean functions, BDDs arose as a way

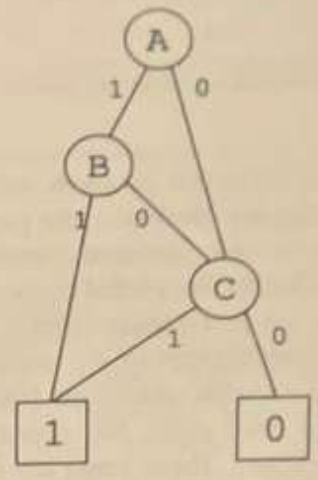

Figure 1. BDD for A.B + C using the variable order A, B, C.

to reduce solution costs on these fields, or even to make the Boolean representation practical [2]. In a special way, one of the most powerful application of BDDs has been symbolic model checking, employed in the formal verification of digital circuits and other finite state systems [13].

However, although manipulation algorithms present complexity linear with the size of the function's graph, this size is highly dependent on the order of the function arguments [6]. Observe that the choice of another variable ordering brings significant changes on the number of nodes of the representation, even for our simple example function: the representation in Figure 2 has one more node than the first one, that is, there was an increase of 20% on its size. Indeed, the choice of this order has a direct consequence on the time and memory costs of BDDs. Unfortunately, to find an order that minimizes the aforesaid costs is an NP-Complete problem [1]. Thus, to reach this goal, it is necessary to make use of approximative methods, that is, heuristics.

In [4] we have shown Parallel Genetic Algorithms as

*Supported by a grant from CAPES (Brazil)
†This work is partially financed by projects *Formal Verification of Computer Systems of Industrial Complexity* (CNPq/ProTeM-CC and NSF) and *Laboratório Concepção de Sistemas* (CNPq/ProTeM-CC).

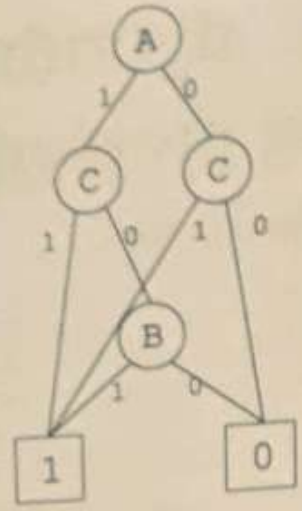

Figure 2. BDD for A.B + C using the variable order A, C, B.

a feasible way to solve this problem, with excellent results of parallelization. However, the genetic algorithm fitness function for this problem remained an expensive task, encumbering the global costs of our solution. In this work we present enhancements to such weakness of the previous algorithm, by introducing a cache system for BDD variable orderings. Naturally, a cache system brings its own costs, but experimental results show that, in practice, these costs are more than balanced by the gains obtained.

II. Parallel Genetic Algorithm

A. The Genetic Algorithm

Genetic Algorithms (GAs) are approximative methods founded on Darwin's Evolutionary Theory, observing that individuals of a population evolve, driven by environmental factors, to form a more fitting population. In the scope of computing, these individuals represent solutions that are enhanced by genetic operators at each iteration, the equivalent to the biological generation. Genetic operators should be defined regarding the characteristics of the specific problem. The most common operators are crossover and mutation, or better, one of their many variants. In particular, GAs have been applied to combinatorial optimization problems [12], which is the case of the BDD variable ordering problem to be solved.

In our algorithm, each individual, also called chromosome, represents a specific BDD variable order. Each chromosome has n genes, where n is the number of variables of the Boolean function whose BDD we wish to optimize. Every gene represents one variable and assumes one integer value, called *allele*, belonging to the range $[0 : n-1]$, without duplicates. Each gene occupies a specific position inside a chromosome, its *locus*. For example, the chromosome representation for the previous BDDs is presented in Figure 3, where we used the mapping $0 \rightarrow$ A, $1 \rightarrow$ B and $2 \rightarrow$ C.

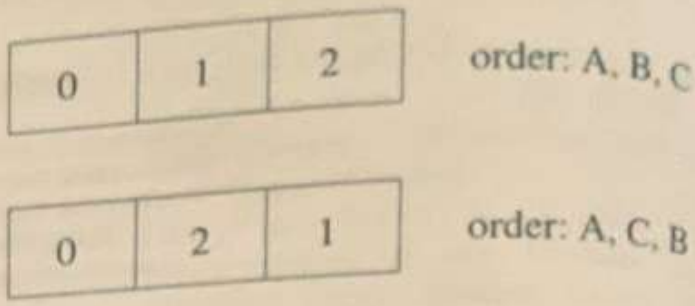

Figure 3. Chromosome representation for the previous BDD variable orderings.

Each chromosome may be evaluated according to a fitness function, which indicates how close to a good solution it is. In our problem, this function is the number of nodes of the corresponding BDD and the smaller this value, the better. In the simple example above, the first chromosome has a fitness of 5, while the second one has a fitness of 6.

The initial population is built from valid individuals created randomly, random variable orders in our case. These orders undergo repeated changes by means of the chosen genetic operators, ending when the algorithm reaches a given maximal number of generations without improvement of its best chromosome's fitness, our stop criterion. At this moment we get an improved population, from which we obtain the fittest chromosome, that represents the best found order.

For this improvement of population fitness to happen two important points have to be considered:

1. Better fitted chromosomes should have a greater influence in the generation of new solutions (new chromosomes). The first BDD in the example above is surely closer from an optimal solution than the second one (in this case it is this solution) and, so, it should have a greater chance of surviving and generating offspring (new chromosomes with some of its characteristics).
2. The process of generation of these new solutions should try to preserve the characteristics that made best chromosomes best.

The first point is ensured by the GA selection procedure which varies little from one implementation to another. We chosen the roulette wheel as GA selection procedure. This selection procedure simulates the spinning of a roulette whell and the random selection of one of its slots, where each slot corresponds to one chromosome's fitness. The greater the chromosome's fitness, the greater the slot size and the probability of

the chromosome being selected. On the other hand, the second point depends largely on a good choice of the genetic operators used.

The genetic operators we employ are crossover and mutation. As crossover operator we use the Order Crossover (OX). This operator randomly selects two crossover points in the parent chromosomes and copies the genes between these points from the first progenitor chromosome to a descending chromosome, respecting the same positions. After that, remaining positions of that descendant are fulfilled with genes of the other progenitor, starting from its second crossover point and skipping already inserted alleles. The other descendant is created by a symmetric procedure. The choice of this operator was due to its compromise with the preservation of the variables relative order, since it is the feature at issue. As mutation operator we use the Swap operator, suitable to the chosen representation. This operator selects two positions of a chromosome and exchanges their alleles, corresponding to a fine tuning of an already promising chromosome .

B. Parallelism

Taking advantage of the natural parallel characteristics of GAs, we decided to build a Parallel Genetic Algorithm (PGA). PGAs have successfully solved many problems [3], such as global optimization [5]. Compared to sequential GAs, parallel GAs benefits from local selection on each sub-population, asynchronous behavior and independence of their processes [12].

Our PGA consists of a model centralized by a master process. The master process distributes individuals into *np* sub-populations, that evolve independently in *np* slave processes, and coordinates the *cc* communication cycles, where *np* and *cc* are integer numbers fixed by the application user. This strategy is highly based on the PGA developed at [10], falling into the category of coarse grained PGAs [12].

Our algorithm, sketched in figures 4 and 5, is relatively straightforward: the master process spawns *np* slave processes, chooses the global fittest individual among the best local individuals sent by the slave processes (master's synchronization point 1), and sends the winner back to all slave processes (master's synchronization point 2), as described in Figure 4. On the other hand, every slave process executes a simple sequential GA, sends its fittest individual to the master process (slave's synchronization point 1) and receives the best global individual from the master process (slave's synchronization point 2). Such an individual replaces the receiver sup-population's worst individual (see Figure 5).

The rationale of our choice of distribution and communication is that, on the one hand, spawning a number of different slave processes results in a richer range of sub-populations. On the other hand, since best intermediate results are distributed to all slave processes, stagnation in fruitless search domains is thus avoided.

Master

```
Spawn slave processes
Repeat cc times
    Receive best local individuals
        from each slave process (1)
    Select best global individual
    Send best global individual
        to slave processes (2)
```

Figure 4. General algorithm for the master process.

Slave

```
Repeat cc times
    Run sequential GA
    Send best local individual
            to master process (1)
    Receive best global individual
            from master process (2)
    Replace worst local individual
            with best global individual
```

Figure 5. General algorithm for the slave process.

III. Cache System

In our original algorithm, the fitness function is liable for most of the run time. Since Genetic Algorithms work randomly, we do not have direct control on how new individuals of the population are generated, or better, if they are really brand-new individuals. Consequently, already evaluated individuals can be generated, between generations or even in the same generation. Thus, many of the executions of the fitness function evaluate the same variable orders, encumbering the algorithm unnecessarily. So, we have decided to employ a cache system in order to avoid useless evaluation steps.

Obviously, the cache system brings its own overhead in managing chromosome codes and the respective number of nodes, beyond an increase in memory costs. Nevertheless, such costs are relatively low when compared to the time and memory required by the construction of BDDs, specially for large ones.

The cache system we developed is a multi-directed tree where each node has an array of node pointers and one integer. The array of node pointers keeps track of the various paths (BDDs orderings) along the tree, the integer records the number of BDD nodes at the leaf nodes, which correspond to a complete variable ordering. The cache is initialized with a single node, which we call root, with null pointers in all fields. Each level of our tree corresponds to a specific locus of the chromosome structure, level zero (root level) corresponding to the first locus and so on. The cache system is sketched on Figure 6, where we consider two chromosomes with four genes (variables) each.

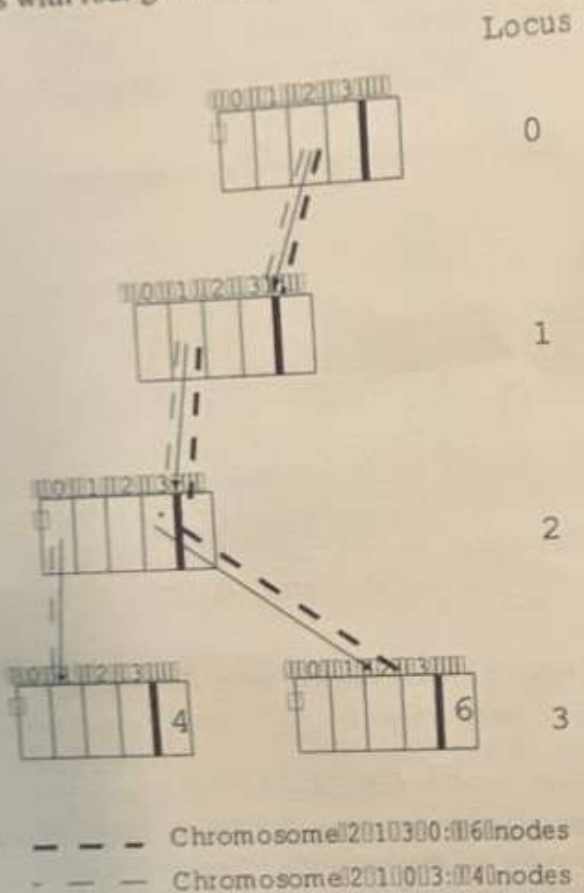

Figure 6. Cache

When a variable order's fitness is required, the cache system looks for a path that matches this ordering, starting from root. Non-initialized nodes found during the path scanning are dynamically created, appending new routes to the tree. Such a search goes on until level $n-1$ is reached, where n is the number of variables of the Boolean function at issue. If the last node of the path, at level $n-1$, has its number of nodes already computed, the cache system returns such value. Otherwise, the BDD construction module is started to compute the required number of nodes. After that, the cache system records the resulting fitness value at the corresponding cache node and returns the variable order's fitness.

Our cache system prioritizes search procedure speed over memory spent. Each search is always executed in $n-1$ steps, reducing eventual processing time overhead. Although theoretical cache size is upper limited to $\sum_{i=0}^{n-1} n^i$, corresponding to a exhaustive search, in practice cache size is bounded by the number of generated chromosomes, a significantly smaller number which can be easily controlled by application user parameters, namely population size and communication cycles number. Besides, the cache size tends to be constant as individuals evolve to later generations of the population. Tables with some experimental results showing actual cache sizes are shown in section IV.

Every spawned process has its own cache system. A cache system is initialized when the corresponding process' sub-population is evaluated for the first time. After that, it undergoes successive queries and updates until the end of the last communication cycle, when the cache is freed.

IV. Implementation and Results

PVM (Parallel Virtual Machine)[7] is the parallel processing software adopted in this project. The most important factor to make this decision was the heterogeneity of the available executing environment, a cluster of Linux PCs, considering PVM's ability to run over a network of heterogeneous processing nodes. The large adoption and portability inherent to PVM were also important factors in this choice. As BDD package we employ bddlib[9], due to its facilities in implementing our fitness function.

Our experiments were conducted on a 10Mb/s Ethernet network in star topology. Four processing nodes were used, each one a 300MHz Pentium Celeron with 64Mb of RAM memory. The tests were made for two families of comparison functions on bit vectors: equality (eq) and less or equal (le), defined as:

$$\forall n \geq 1, eq_n = \lambda(\vec{x}_n^1, \vec{y}_n^1)[\bigwedge_{i=1}^{n} x_i \leftrightarrow y_i]$$

$$le_1 = \lambda(\vec{x}_1^1, \vec{y}_1^1)[\neg x_1 + y_1]$$
$$\forall n > 1, le_n = \lambda(\vec{x}_n^1, \vec{y}_n^1)\ [x_n \leftrightarrow y_n . le_{n-1}(\vec{x}_n^2, \vec{y}_n^2) + \neg x_n . y_n]$$

Two kinds of experiments were realized, the first compares a sequential GA implementation with our uncached parallel GA implementation. The second one compares the uncached parallel GA with its cached version. The results presented for both classes correspond

Table I
COMPARATIVE RESULTS FOR SEQUENTIAL AND PARALLEL GA (*equality*)

	eq_4	eq_8	eq_{16}
seq. time (sec.)	5.2	23.6	241
par. time (sec.)	2	6.6	72
factor	2.60	3.58	3.35
seq. results (# nodes)	12	24.2	49.8
par. results (# nodes)	12	24	49.2
factor	1.00	1.01	1.01

Table II
COMPARATIVE RESULTS FOR SEQUENTIAL AND PARALLEL GA (*less or equal*)

	le_4	le_8	le_{16}
seq. time (sec.)	5	85.8	442
par. time (sec.)	1.8	23.6	102.4
factor	2.78	3.64	4.32
seq. results (# nodes)	9	20	73.6
par. results (# nodes)	9	17.6	47.4
factor	1.00	1.14	1.55

to the mean of 5 runs (to eliminate discrepancies due to the randomness of the choice of the initial population), and all algorithm parameters were maintained.

The obtained results for the first kind of experiments are summarized in tables I and II, shown below. We note the excellent results in terms of parallelization, with a gain about linear when compared to the sequential algorithm. The best parallelization results we can expect for non-random problems is a linear speed-up [8], but the GA randomness allows us get super linear speed-ups. In our example, we achieve a time gain factor of 4.32 using four processors in the le_{16} function (see table II). Such super linear speed-up is due to the beneficial influence of communication between subpopulations.

About 90 percent of the run time of the uncached parallel algorithm is spent executing the fitness function. Communication overhead is between 5 and 10 percent, allowing to decrease the execution time by the use of more processing nodes without significant performance loss. The program always finds optimal or near optimal solutions. For example, we got 12, 24 and 49.2 BDD nodes for the 4, 8 and 16 bits equality problems, respectively. The corresponding optimal results are 12, 24 and 48 BDD nodes, in the same order (see table I).

The second kind of experiments, in turn, compares the uncached parallel GA with its new version, with cache system. Tables III and IV summarize the found

Table III
COMPARATIVE RESULTS FOR PGA AND PGA WITH CACHE SYSTEM (*equality*)

	eq_4	eq_8	eq_{16}
PGA time (sec.)	2	6.6	72
Cached PGA time (sec.)	0.8	2	28.4
factor	2.5	3.3	2.54

Table IV
COMPARATIVE RESULTS FOR PGA AND PGA WITH CACHE SYSTEM (*less or equal*)

	le_4	le_8	le_{16}
PGA time (sec.)	1.8	23.6	102.4
Cached PGA time (sec.)	0.6	7.6	37.2
factor	3	3.11	2.75

results. Note the considerable gains of our new PGA over the uncached one, confirming our expectations. With a cache system, the fitness function uses only about 30% of the time spent by the uncached one. The remaining run time would be spent unnecessarily by the construction of BDDs already evaluated. Such optimization allows us decrease the fitness function run time from about 90% to about 70% of the global time. Naturally, the computation of the fitness function still costs an important time slice, but certainly the use of a cache system brings significant improvement to our algorithm. On the whole, the cached PGA spends only about 35% of the time of the uncached PGA.

Another important benefit resulting from the introduction of a cache system is the addition of a new stop criterion for the GA: sample space covering rate. We can implement such a stop criterion in a easy way, just comparing the cache size with the number of possible variable combinations for a given function to be optimized.

Unlike the theoretical estimate, the cache sizes for our experiments are very reasonable, confirming the positive trade-off of the cache system, as shown in tables V and VI.

V. CONCLUSION AND FUTURE WORKS

With this work we intend to show not only that the variable ordering problem can be susceptible to be sucessfully dealt by a GA, and especially by a parallel GA, but how a simple refinement of our algorithm can reduce overall costs significantly. Now, we intend to optimize the essential and unavoidable population eval-

Table V
THEORETICAL AND ACTUAL CACHE SIZES *(equality)*

	Number of cache nodes	
	Theoretical	Actual
eq_4	2.40×10^6	247
eq_8	1.23×10^{18}	4,118
eq_{16}	4.71×10^{46}	45,259

Table VI
THEORETICAL AND ACTUAL CACHE SIZES *(less or equal)*

	Number of cache nodes	
	Theoretical	Actual
le_4	2.40×10^6	343
le_8	1.23×10^{18}	17,907
le_{16}	4.71×10^{46}	60,698

uation, making this tecnique even more efficient, and to define some PGA parameters automatically, reducing the number of decisions to be made by the application users.

REFERENCES

[1] B. Bollig et al. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 1996.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[3] E. Cantú-Paz. A summary of research on parallel genetic algorithms. Technical report, Illinois Genetic Algorithms Laboratory, 1995.

[4] U. S. Costa, D. Déharbe, and A. M. Moreira. Variable ordering of bdds with parallel genetic algorithms. In *Proceedings of PDPTA'2000*, 2000.

[5] P. S. de Souza. *Asynchronous Organizations for Multi-Algorithm Problems*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1993.

[6] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *1993 IEEE/ACM international conference on Computer-aided design: ICCAD'93*, pages 38–41. IEEE/ACM, 1993.

[7] A. Geist et al. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[8] K. Hwang and P. A. Briggs. *Computer Architecture and Parallel Processing*. Mc Graw-Hill, 3rd edition, 1987.

[9] D. Long. *bddlib - a binary decision diagram (bdd) package*. available on http://www.cs.cmu.edu/~modelcheck/bdd.html.

[10] F. Marin, O. Trelles-Salazar, and P. Sandoval. Genetic algorithms on an message-passign architectures using PVM: application to the routing problem. In Springer-Verlag, editor, *Parallel Problem Solving from Nature: PPSN111*, Lecture Notes in Computer Science, pages 534–543, 1994.

[11] K.E. Partridge. Bddtcl: an environment for visualizing and manipulating binary decision diagrams. In *CHI'96 conference compation on human factors in computing systems: common ground*, pages 111–112, 1996.

[12] M. G. Resende. Parallel metaheuristics for combinatoral optimization. International School on Advanced Algorithmic Techniques for Parallel Computation with Applications, 1999.

[13] B. Yang, R.E. Bryant, D.R. O'Hallaron, A. Biere, O. Coudert, G. Jaanssen, R.K. Ranjan, and F. Somenzi. A performance study of bdd-based model checking. In *FMCAD'98. Formal Methods in Computer-Aided Design*, number 1512 in Lecture Notes in Computer Science, Palo Alto, Ca, 1998. Springer Verlag.