A new parallel architecture for QBF tools

Benoit da Mota, Pascal Nicolas, Igor Stéphan

▶ **To cite this version:**

Benoit da Mota, Pascal Nicolas, Igor Stéphan. A new parallel architecture for QBF tools. 2010 International Conference on High Performance Computing and Simulation, HPCS 2010, 2010, Caen, France. pp.324 - 330, 10.1109/HPCS.2010.5547114 . hal-03255427

HAL Id: hal-03255427

https://univ-angers.hal.science/hal-03255427

Submitted on 9 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A New Parallel Architecture for QBF Tools

Benoit Da Mota, Pascal Nicolas, Igor Stéphan
LERIA, University of Angers, France
{*damota, pn, stephan*}*@info.univ-angers.fr*

ABSTRACT

In this paper, we present the main lines and a first implementation of an open general parallel architecture that we propose for various computation problems about Quantified Boolean Formulae. One main feature of our approach is to deal with QBF without syntactic restrictions, as prenex form or conjunctive normal form. Another main point is to develop a general parallel framework in which we will be able in the future to introduce various specialized algorithms dedicated to particular subproblems.

KEYWORDS: Quantified Boolean Formulae (QBF), parallel, tools.

1. INTRODUCTION

The quantified Boolean formula (QBF) validity problem is a generalization of the Boolean formulae satisfiability problem. While the complexity of Boolean satisfiability problem is NP-complete, it is PSPACE-complete for the QBF validity problem. This is the price for a more concise representation of many classes of formulae. Many important problems in several research fields have polynomial-time translations to the QBF validity problem : AI planning [1] and Formal Verification (see [2] for a survey). Most of the recent and efficient decision procedures for QBF have formulae in negative normal form (NNF) as input or even more restrictive format such as formulae in conjunctive normal form (CNF). But rarely, problems are expressed in such a form which destroyed completely their original structures. It is much more natural to use the full expressivity of the QBF language: all the usual connectors (including implication, bi-implication and xor) and quantifiers nested in the formula. So, a first goal of our work is to elaborate a QBF solver without restrictions on the input formula. Today, the availability of multi-core processors, computer clusters and grid computing is an opportunity to elaborate new algorithms to solve difficult computation problems. Then, the second goal of our work is to exploit the power of parallel programming to tackle the QBF validity problem by reusing, adapting, improving, combining,... well known sequential algorithms for QBF inside a parallel architecture (see [2] for a survey on open QBF techniques). Furthermore, beyond the validity problem, we are also interested in more general problems linked to QBF, as compilation for instance. In this case, the answer of the system is not simply YES or NO, but rather a set of formulae. Thus, one feature of our long term goal is to elaborate an architecture, as open as possible, in order to offer a set of QBF tools to the end user. In this present work, after having recalled some fundamental notions about SAT and QBF in section 2 and made a quick survey on sequential and parallel QBF solvers in section 3, we describe the main feature of the parallel architecture that we propose in section 4. In section 5, we detail the first instantiation of this general framework that we have done and in section 6 we give some technical details and experimental results about our implementation, before to conclude in section 7 by giving some future research directions.

2. PRELIMINARIES

Symbols $\bot$ and $\top$ are the propositional constants. Symbol $\wedge$ stands for conjunction, $\vee$ for disjunction, $\neg$ for negation, $\rightarrow$ for implication, $\leftrightarrow$ for bi-implication and $\oplus$ for xor. A literal is a propositional symbol or the negation of a propositional symbol. Definitions of the language of propositional formula **PROP** and semantics of all the Boolean symbols are defined in standard way. A formula is in negation normal form (NNF) if it is only constituted of conjunctions, disjunctions and literals. A formula is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals. A substitution is a function from propositional symbols to **PROP**. This definition is extended as usual to a function from **PROP** to **PROP**: $[x \leftarrow F](G)$ is the

formula obtained from G by replacing each occurrence of x by the formula F. This definition is also extended as usual for the substitution of a formula by another formula. Logical equivalence is denoted $\equiv$. The symbol $\exists$ stands

for the existential quantifier and $\forall$ stands for the universal quantifier (q stands for any quantifier). The set **QBF** of quantified Boolean formulae is defined inductively as follows: if F is in **PROP** then it is also in **QBF**; if F is in **QBF** and x is a propositional symbol then $(\exists x\ F)$ and $(\forall x\ F)$ are also in **QBF**; if F is in **QBF** then $\neg F$ is also in **QBF**; if F and G are in **QBF** and $\circ$ is in $\{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ then $(F \circ G)$ is in **QBF**. If a propositional symbol x is not under the scope of a quantifier (as in qx), then it is a free propositional symbol. A QBF is closed if its set of free propositional symbols is empty. A binder is a string $q_1 x_1 \dots q_n x_n$ with $x_1 \dots x_n$ distinct propositional symbols and $q_1 \dots q_n$ quantifiers. A QBF is in prenex form if it is constituted of a binder and a propositional formula called the matrix. A QBF is in conjunctive normal form if it is a prenex QBF and its matrix is in conjunctive normal form. The definition of substitution is extended to QBF as follows: $[x \leftarrow F](G)$ is the formula obtained from G by replacing *free* occurrences of the propositional symbol x by the formula F. The semantics of QBF is defined as follows: for every propositional symbol y and every QBF F, $(\exists y\ F) \equiv ([y \leftarrow \top](F) \vee [y \leftarrow \bot](F))$ and $(\forall y\ F) \equiv ([y \leftarrow \top](F) \wedge [y \leftarrow \bot](F))$.A QBF F is valid if $F \equiv \top$.

3. STATE OF THE ART OF QBF SOLVERS

3.1. State-of-the-art Of Sequential QBF Solvers

Since the verification of a model of a prenex QBF is co-NP-complete [3] there are very few incomplete algorithms based on metaheuristics. As far as we know there are only two procedures which are based on local search: `WalkQSAT` [4] and `QBDD(LS)` [5]. Hence, most of the procedures for QBF are decision procedures which may be separated in three kinds: "monolithic" procedures which are self-sufficient, procedures with a transformation to another decision problem formalism and procedures with an oracle. In the first kind, procedures are based on resolution like `QKN` [6], are bottom-up quantifier elimination procedures like `quantor` [7] (for CNF QBF) or `Nenofex` [8] (for NNF QBF) as extension of Davis and Putnam algorithm, or are top-down quantifier elimination (or search-based) procedures like `Evaluate` [9], `QUBE` [10] or `QSOLVE` [11] (all for CNF QBF) or `qpro` [12] (for NNF QBF) as extension of Davis, Logemann and Loveland algorithm. A transformation procedure interprets QBF in a different decision problem which has already efficient decision procedure: SAT (in CNF) for `sKizzo` [13] or ASP [14] (in those cases with an exponential growth of the formula). Procedures with an oracle come back to the initial concept of "polynomial hierarchy" since an oracle which is able to solve subproblems with smaller complexity is needed: `QBDD(DLL)` [5] uses an NP-complete oracle and `QSAT` [15] uses both NP-complete and co-NP-complete oracles. This latter procedure is detailed in section 5 since the instantiation of our parallel model uses `QSAT`.

3.2. State-of-the-art Of Parallel QBF Solvers

As far as we know, there exist three implementations of parallel solvers for the problem of validity of QBF: `PQSOLVE` [11], `PaQube` [16] and `QMiraXT` [17]. Two important remarks :

- Those procedures are all based on a top-down sequential procedure (`QSOLVE` [11] for `PQSOLVE`, `QUBE` [10] for `PaQube` and `PaMiraXT` [18], a parallel SAT solver, for `QMiraXT`) and then apply a *semantic splitting strategy* which selects a propositional symbol of the most external block of quantifiers and apply the quantifier semantics to split the problem and distribute jobs.
- Those procedures are all dedicated for CNF QBF.

The procedure `PQSOLVE` is a distributed solver which uses techniques from chess program parallelization [11]. It instantiate a peer-to-peer model: An idle process requests for work to a randomly selected process and becomes the slave of this process for one job. Each process has a stack of jobs to do and a master sends one of these jobs to its momentary slave. A slave may become itself the master of another process. The procedure `QMiraXT` is dedicated to take into account the potential performance of the modern multi-core and/or multithreaded processors. By using a shared memory threaded solver, the learnt conflict clauses [19] are shared between the different search spaces. There is no master process but instead a Master Control Object (MCO) which allows threads to communicate through asynchronous signal global event messages (if subproblems are valid or not). The MCO also handles the semantic splitting strategy, called Single Quantification Level Scheduling (or SQLS) and distributes the jobs. The procedure `PaQube` is designed as a Master/Slave Model where one process is dedicated to the master (which does not need a dedicated CPU) and the others are the slaves which actually perform the search. `PaQube` is a parallel

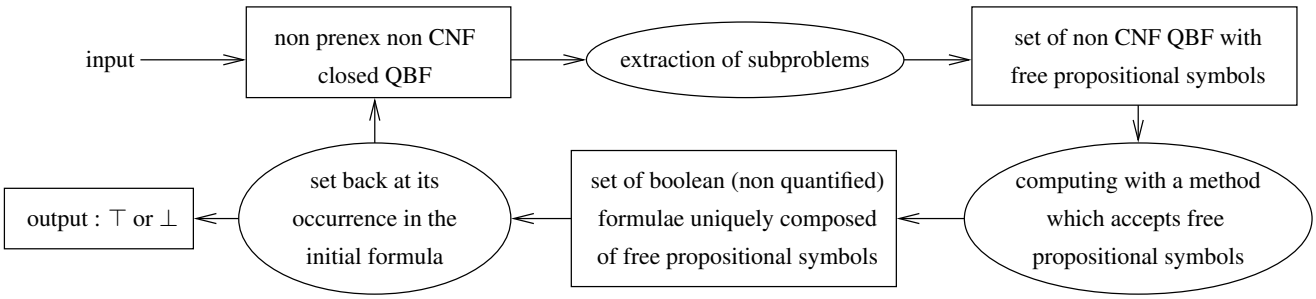

Figure 1. Execution Loop

MPI based QBF solver and the slaves share some conflict learnt clauses or solution learnt cubes through messages added to the local databases. The main work of the master is to realized, as for `QMiraXT`, the SQLS. By their models, `PQSOLVE` and `PaQube` are more extensible to clusters and grids than `QMiraXT` but this latter seems to take more into account the evolution of hardware than the two first ones.

4. A PARALLEL ARCHITECTURE

The general model of our system is a master/slaves architecture implemented by means of a server/clients network on a cluster of computation nodes. Thus, in the sequel we will use indifferently the terms master or server and slaves or clients. Our parallel procedure applies many copies of a sequential procedure to a small part of the original QBF. To do so, disjoint subproblems have to be extracted and then treated in parallel. The splitting of the given QBF can be made syntactically or semantically. In our architecture, we propose two splitting strategies and a way to mix them both. In our parallel architecture, the master node reads the original QBF, extracts syntactically some sub-problems and distributes the jobs to slave nodes. Then, it waits for answers and reinsert the result into the original formula. This loop is repeated until to obtain $\top$ or $\bot$ to ensure the completeness of the procedure. The whole process is illustrated in figure 1 The aim of a slave node is to accept a prenex non CNF QBF with free variables and to return an equivalent non quantified boolean formula built only on these free variables. For instance, in our first implementaion described here, the slave node uses the method described in subsection 5.1. At every iteration of the general loop we can determine a *syntatic width* that is the number of subformulas extracted by the splitting method. The *maximal syntactic width* obtained all along this process indicates the maximal number of computation nodes needed for our syntactic splitting. Because of the chosen splitting method, this maximal value can be not very large and reached at the first iteration. With only this approach, it is obvious that our parallel process would not be very efficient if many problems have a maximal syntactic width lesser or equal to two, and then all slaves nodes, except two, would be useless. But, as described below, the slave nodes can themselves create some new subproblems. Every slave node receiving a formula to process has the responsability to estimate the difficulty of the given task. If it seems easy, then it decides to solve itself the given problem. So, it starts immediately the computation, returns the solution and waits for a new task. If it considers that the task is too difficult, it applies a semantic splitting strategy based on free variables of the QBF. So, it generates a set of new subproblems and returns them to the master node. This whole process is illustrated in the figure 2. By this way, a client is able to add some new tasks in the stack of tasks of the master node, which on its turn will distribute them to slave nodes. This particular client splitting heuristic is a way to optimize the use of numerous computation nodes when the maximal syntactic width is small.

5. A FIRST INSTANCE

In our parallel architecture, the slave nodes have to compute an equivalent propositional formula of a given QBF. Since this QBF is itself a subformula of a larger one, the goal is to replaced it by the equivalent generated propositional formula. For that, we have chosen to instantiate our parallel architecture with a procedure from the state-of-the-art which applies a syntactical partitioning of the search space with an oracle: the `QSAT` procedure [15]. In the rest of this section we recall the `QSAT` procedure, then we describe how work the slave nodes and then we come back to the master node and its syntactic treatment of QBF.

5.1 QSAT Decision Procedure For QBF

`QSAT` [15] is an inside-out quantifier elimination decision procedure for the QBF validity problem. This procedure it-

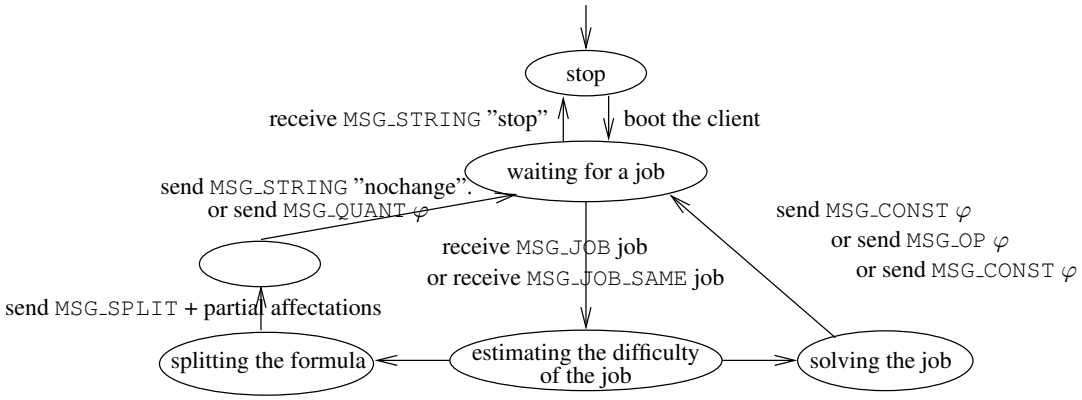

Figure 2. Client Automata

eratively works on the pattern $Q(\exists x\ (F \wedge G))$ such that the propositional symbol x does not appear in F. The logical equivalence $(\exists x\ (F \wedge G)) \equiv (F \wedge (\exists x\ G))$ isolates the subformula $(\exists x\ G)$ then an equivalent formula G' which does not contain x is computed thanks to a procedure $simp$, finally QSAT is recursively applied on the QBF $Q(F \wedge G') (\equiv Q(\exists x\ (F \wedge G)))$. This induction step is similar with a universal quantification. For a formula $Q(\forall x(F \wedge G))$, such that x does not appear in F: the logical equivalences $(\forall x(F \wedge G)) \equiv (F \wedge (\forall x\ G)) \equiv (F \wedge (\neg(\exists x\ (\neg G))))$ isolates again a subformula $(\exists x\ (\neg G))$ for which the above method can be applied. This process is iterated until there is no more quantifier. The $simp$ procedure needs two decision procedures, one for SAT and one for TAUT. Its algorithm is based on the classical construction of the CNF as the conjonction of the negation of the lines, considered as cubes, of the truth table which falsify the propositional formula. It builds a CNF G' on the free propositional symbols of G such that $G' \equiv (\exists x\ G)$. It works as a Davis, Logemann and Loveland algorithm by splitting the search space according to a free propositional symbol y of G by recursive calls on $[y \leftarrow \top](G)$ and $[y \leftarrow \bot](G)$. In [15], QSAT is described under a double restriction to SAT and to CNF formulae. Under this latter restriction, the search of a pattern $Q(\exists x\ (F \wedge G))$ such that x does not appear in F is trivial then only the choice of x enables different heuristics.

5.2. Client Description

From a global view, a client (running on a slave node) receives a QBF with free variables and must send back a not quantified Boolean formula composed only with these free variables. The QSAT procedure described previously is a good candidate since it is works well for the two kinds of quantifiers $\exists$ and $\forall$. In addition, it is possible to eliminate several consecutive quantifiers of the same type. A prenex QBF requires as many iterations to the QSAT procedure as the number of quantifier alternations. The QSAT procedure has a drawback, it is efficient only on a certain class of formulas called "long and thin" [15]. So, to try to escape to this difficulty, our architecture provides that a client can give up on a too difficult task as it is illustrated in the figure 2 and commented below. Firstly, the client starts and waits for a task. Then, it receives MSG_JOB followed by a job: a formula accompanied with a partial interpretation or MSG_JOB_SAME and only a partial interpretation when the same formula is resubmitted for a new computation. By means of a heuristic function, the client tries to estimate the difficulty of the task and the different cases are the following.

- The job is considered computable : the job is executed and a message is returned: MSG_CONST with $\top$ or $\bot$, or MSG_OP with a propositional formula only build with free propositional symbols, or MSG_CNF with a formula like for MSG_OP but in CNF.
- The job is considered too difficult to compute : so a semantic splitting of the formula is done and the message MSG_SPLIT is sent with a set of partial affectations. Then is sent the message MSG_STRING plus "no change" or MSG_QUANT plus the formula partially treated.

Our heuristic function to estimate the difficulty of a task is very simple: if a task has already been split, then it is easy, otherwise, if the number of free variables is lesser than a constant, then the task is easy, otherwise, the task is no computable. Our semantic splitting procedure is also very simple. When the formula is read, the free variables are stacked. So, if a splitting is necessary, we compute the number of free variables to instantiate without exceeding another constant defining the finest possible splitting. Then, the corresponding free variables are unstacked

and all the partial affectations are generated and sent with MSG_SPLIT. Our system could be improved by using a more sophisticated heuristic to evaluate the difficulty of a task and to choose the variable on which the semantic splitting is done. But, for the moment, we have prefered to implement a fully deterministic behavior (a client always cuts the same task the same way) in order to limit the variations during different runs. Today, the client ends its splitting work by returning MSG_STRING plus "no change" but in the future, by coupling the estimation of difficulty and a partial solving of the problem, we envisage to return some partial affectations accompanied with MSG_QUANT followed by the formula partially treated and rewrote.

5.3. Syntactical Splitting Strategy

The syntatic splitting strategy realized by the master node looks for some subformulas which can be treated separately. In a general way, it consists to take a binary logical connective and to work recursively on its two subformulae. QSAT being a quantifier elimination procedure, at least one of these two sub-formulas should have a quantifier. Our approach of reasoning with QBF is to keep as close as possible to the original formulation of the problem, because we postulate that the one who has encoded the problem has represented some particular pieces of knowledge via some particular subformulae. So, the position of quantifiers makes sense for the author and that is why whe have chosen to extract the set of non CNF prenex subformulae with possibly free propositional symbols. For instance, let $Q(F \leftrightarrow (G \wedge H))$ be a QBF, such that F and G are prenex with free variables and H is a non quantified boolean formula. Our splittig strategy extracts the set of subproblems $\{F, G\}$ that can be treated separately. Our syntactical split does not change from one execution to another, and also limits the number of hypothesis during the study of experimental results. The drawback of this split is the maximal syntactic width which often does not exceed 2.

6. EXPERIMENTATIONS

6.1. Technical Choices

Since we want to escape the difficulties arising from the translation of general QBF into prenex CNF ones, one of the specific and original feature of our approach is to be able to deal directly with non prenex, non CNF QBF. In the elaboration of the data structure for representing QBF we have searched the maximum of expressivity and extensibility and not performance or saving of memory space. A QBF is represented by a set of abstract elements of formula, linked to each other in a tree. An abstract element of formula imposes only two actions to the concrete elements : to start an abstract visitor and to (de)serialize itself. Among the concrete elements of formula we find the tree nodes for representing logical (unary or binary) connectives or quantifiers and the tree leaves for representing literals or propositional constants. By this way, our data structure is a set of composite elements all inheriting from the abstract element of formula. Following the design pattern composite+visitor, the visitors are external treatments that adapt automatically to the concrete element met along the visit. A concrete formula is serializable to be sent on the network and deserializable to be restored in memory. In order to represent more efficiently the formulas having a part in CNF, we have define a concrete element CNF leaf storing a matrix of literals.

All the communication part is done by using the standard Message Passing Interface (MPI) and we have chosen the Open MPI implementation. For our approach with a composite data structure, MPI has a shortcoming: it is simply not possible to send and receive complex data types natively. The Boost.MPI C++ library is a wrapper dedicated to respond this constraint by using Boost.Serialization. All our communications are synchronous and use the send/recv of Boost.MPI.

To use the method from section 5.1, we have to answer two questions: Is the formula tautological? Otherwise, is the formula unsatisfiable? To answer, we have included the MiniSat decision procedure [20]. Besides the fact that this procedure is very efficient, its sources are available and distributed freely under the MIT licence (a strictly freer licence than the LGPL). Data structures in MiniSat are very different from ours and are geared for performance. So we have an interface (a visitor) to formulate our queries in the data format of MiniSat. Conversely, we can interpret the answers of MiniSat to integrate them directly into our data structure. From our expressive and extensible data structure, it is thereby possible to benefit of efficient treatments in another model by means of a conversion in linear time with respect to the size of the formula.

To reduce the need in bandwidth, each computing node keeps in cache the MiniSat solver object of the last treated subformula. If a new task on the same formula comes, the first benefit is the saving of the transfer of this formula. The second benefit is the use by MiniSat of the clause learning. The already trained instance of MiniSat can answer more quickly to a new question. The ideal configuration would be to share the learning between the different nodes, but that is not the case at the moment in our

system. In addition, we should study the extra traffic generated by sharing this information.

6.2. Experimental Results

To evaluate our architecture, we ran few preliminary tests on some instances of *qbflib* (www.qbflib.org) and on some self-generated problems. The semantic splitting is strictly the same whatever the number of computing clients. Experiments are performed on a high-performance cluster containing 12 Bull Novascale R422, connected by a double gigabit ethernet network. Each R422 server has 2 motherboards. Each computer is a 2x Int12el(R) Quad-Core Xeon(R) E5440 at 2.83GHz with 16GB of local memory and runs on a 64 bits version of the Linux 2.6.18 kernel.

We select some results to illustrate some interesting cases. The first column of our tables represents the number of running clients. The label on the top of the columns are shortcuts for the instances names. In addition, we have a dedicated processor to run the master. Our results match the pattern: *time in seconds speedup*.

Table 1. Results without pretreatment

	c8_8		c8_16		r4_5		s5_4	
1	105	1	7264	1	18483	1	58812	1
2	52	2.0	6627	1.1	11215	1.6	27741	2.1
4	28	3.8	4560	1.6	3601	5.1	19938	2.9
8	15	7.0	2437	3.0	4940	3.7	5537	10.6
16	11	9.5	3247	2.2	2268	8.1	4320	13.6
32	7	15.0	967	7.5	857	21.6	1027	57.2
64	6	17.5	3950	1.8	247	74.8	582	101.1
128	7	15.0	3548	2.0	128	144.4	780	75.4

Table 1 presents the results for 4 instances: counter8_8, counter8_16, ring4_5 and semaphore5_4. They are part of the QBF1.0 set available on *qbflib*. All these formulas are prenex, then the maximal syntactic width is 1. The unique choice we have to treat this formulas in a parallel way is to use the semantic extraction of subproblems described in subsection 5.2. The first problem, counter8_8 is simple, so increasing the number of clients is really efficient only until we use 8 of them. The explanation is easy: greater is the number of used nodes, longer is the initialization time of all the machinery of MPI. For instance it takes 5 seconds for 128 nodes. To evaluate our approach with many calculation resources, we need bigger problems. The second problem, counter8_16, satisfies this constraint, however the times are irregular and the speedup is relatively bad. Two phenomena occur here. First, some subproblems are very difficult to solve. For this example, some tasks take several hundreds of seconds. A task taking 10% of the total time, will limit to 10 the maximal speedup in the best case, ie when starting the search with this task. The second phenomena is linked to the use of `MiniSat` as an oracle. Each of these 4 problems has only one alternation of quantifiers. So, each computation node receives the formula one time and creates a unique instance of the `MiniSat` solver. All other tasks will use this instance in cache and will take benefits of the clause learning already locally done during previous tasks. The proper of a parallel execution is the impossibility to predict the task distribution. In extension, each semantic subproblem trains the local instance of `MiniSat` and this training is unpredictable and different at each execution. More computation nodes we have, more the probability to learn, locally, a good information before to solve a hard problem decreases. Using more processors does not necessarily implies a quicker resolution if we are not able to share information. For the ring4_5 problem we find a super-linear speedup with 64 and 128 computation nodes. As for the previous problem, we think that the clause learning of `MiniSat` has an effect, but positive this time. It is possible that some subproblems train efficiently most of the `MiniSat` solvers. With the increase of the number of nodes, each solver receives less subproblems to solve and perhaps keeps longer some more relevant information's. We note similar durations: no task monopolize many resources. But, because of the random nature of the parallel execution, we have to do multiple runs to consolidate our hypothesis. We have a super-linear speedup for the semaphore5_4 problem with 32 and 64 clients. With 128 the gain is lesser than with 64 clients. Unlike ring5_4, some tasks are very long and can take 50% of the total time with 64 clients and more. As counter8_16, more the clause learning is distributed on different clients more the long task are penalizing.

Table 2. Results with pretreatment

	c8_8		c8_16		r4_5		s5_4	
1	52	1	6069	1	8718	1	25647	1
2	27	1.9	2959	2.1	3536	2.5	6530	3.9
4	14	3.7	1855	3.3	1597	5.4	1855	13.8
8	10	5.2	3338	1.8	914	9.5	2226	11.5
16	6	8.7	1114	5.4	349	25.0	1993	12.9
32	5	10.4	1527	4.0	202	43.2	778	33.0
64	5	10.4	1249	4.9	89	98.0	240	106.9
128	5	10.4	1750	3.5	38	229.4	588	43.6

Table 2 shows the results on the same problems but with few optimizations on the input formula. The master, just after reading the formula, applies recursively a naive propagation comparable to unit propagation on CNF formula and seeks monotonic literals. The results with this simple pretreatment show that it is possible to improve the performances of our resolution procedure with state-of-the-art techniques. This pretreatment could be applied to subproblems. All results for ring4_5 correspond to a super-linear

speedup and the gain with 128 clients reaches 229.4.

Table 3. Results for adder_6 and chaine_30

	adder_6		chaine_30	
1	3479	1	68343	1
2	1377	2.5	33068	2.1
4	995	3.5	16181	4.2
8	1788	1.9	7504	9.1
16	708	4.9	3780	18.1
32	1242	2.8	2043	33.5
64	1238	2.8	1023	66.8
128	1007	3.5	438	156.0

Unlike previous problems, adder_6 and chain_30 are not prenex and contain bi-implication and/or xor. The adder_6 problem has a syntactic width of 2, except for the last iteration. the chain_30 have a syntactic width of 30 only during the first iteration, 1 otherwise. Table 3 summarizes the different results. For adder_6, the speedup is good until 4 clients. For this problem, the last iteration is the longer and this task does not have free variable: only one client is working. The speedup for chain_30 is super-linear. First, the master distributes 30 subproblems which are quickly simplified, then the syntactic width decreases to 1: the semantic extraction takes over. As seen with ring5_4, the times to solve the tasks are very similar.

7. CONCLUSION

In this paper we have exposed our ideas about a new parallel approach for QBF tools. Our first implantation demonstrates the validity of our model. Even though its computing power is not very high for the moment our architecture is the only one able to deal with non prenex, non CNF QBF with bi-implication and xor.

In the future, we will work on the integration of other methods of computation and on the development of some heuristic of choice to better select the subproblems and the method to apply to solve them. Another line of research will be to elaborate some techniques of knowledge sharing. Indeed, every node learns some information during its proper computations and then it would be certainly very efficient to propagate, at least one part of, this learning to the other nodes. Last but not least, we do not have to forget that problems in QBF are in PSPACE class of complexity. So, some strategies of resolution or enunciation of solutions of some problems can require data with an exponential size. So, we need measures of size of our data structures and of the cost of their transfer on the network in the cluster.

REFERENCES

[1] A. Ayari and D. Basin, "Qubos: Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers," in *FMCAD'02*. Springer-Verlag, 2002.

[2] M. Benedetti and H. Mangassarian, "Experience and perspectives in qbf-based formal verification," *Journal on Satisfiability, Boolean Modeling and Computation*, 2008.

[3] H. Kleine Büning and X. Zhao, "On Models for Quantified Boolean Formulas," in *Logic versus Approximation, In Lecture Notes in Computer Science 3075*, 2004.

[4] I. Gent, H. Hoos, A. Rowley, and K. Smyth, "Unsing Stochastic Local Search to Solve Quantified Boolean Formulae," in *CP'03*, 2003.

[5] G. Audemard and L. Sais, "A Symbolic Search Based Approach for Quantified Boolean Formulas," in *SAT'05*, 2005.

[6] H. Kleine Büning, M. Karpinski, and A. Flögel, "Resolution for quantified Boolean formulas," *Information and Computation*, vol. 117, no. 1, pp. 12–18, 1995.

[7] A. Biere, "Resolve and Expand," in *SAT'04*, 2004, pp. 59–70.

[8] F. Lonsing and A. Biere, "Nenofex: Expanding NNF for QBF Solving," in *SAT'08*, 2008, pp. 196–210.

[9] M. Cadoli, A. Giovanardi, and M. Schaerf, "Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae," in *AIIA'97*, 1997, pp. 207–218.

[10] E. Giunchiglia, M. Narizzano, and A. Tacchella, "QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability," in *IJCAR'01*, 2001, pp. 364–369.

[11] R. Feldmann, B. Monien, and S. Schamberger, "A Distributed Algorithm to Evaluate Quantified Boolean Formulae," in *AAAI'00*, 2000.

[12] U. Egly, M. Seidl, and S. Woltran, "A Solver for QBFs in Nonprenex Form." in *ECAI'06*, 2006, pp. 477–481.

[13] M. Benedetti, "skizzo: a suite to evaluate and certify QBFs," in *CADE'05*, 2005, pp. 369–376.

[14] I. Stéphan, B. Da Mota, and P. Nicolas, "From (Quantified) Boolean Formulas to Answer Set Programming," *Journal of logic and computation*, vol. 19, no. 4, pp. 565–590, 2009.

[15] D. Plaisted, A. Biere, and Y. Zhu, "A satisfiability procedure for quantified Boolean formulae," *Discrete Applied Mathematics*, vol. 130, pp. 291–328, 2003.

[16] M. Lewis, P. Marin, T. Schubert, M. Narizzano, B. Becker, and E. Giunchiglia, "PaQuBE: Distributed QBF Solving with Advanced Knowledge Sharing," in *SAT'09*, 2009, pp. 509–523.

[17] M. Lewis, T. Schubert, and B. Becker, "QMiraXT - A Multithreaded QBF Solver," in *MBMV'09*, 2009, pp. 7–16.

[18] T. Schubert, M. Lewis, and B. Becker, "PaMiraXT: Parallel sat solving with threads and message passing," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 203–222, 2009.

[19] L. Zhang and S. Malik, "Conflict Driven Learning in a Quantified Boolean Satisfiability Solver," in *ICCAD'02*, 2002.

[20] N. Een and N. Srensson, "An Extensible SAT-solver," in *SAT'03*, 2003.