A parallel solving algorithm for quantified
constraints problems

Jérémie Vautard
University of Caen — GREYC
BP 5186 - 14032 Caen
jeremie.vautard@gmail.com

Arnaud Lallouet
University of Caen — GREYC
BP 5186 - 14032 Caen
arnaud.lallouet@info.unicaen.fr

Youssef Hamadi
Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom
youssefh@microsoft.com

Abstract—Quantified constraint satisfaction problems have been the topic of an increasing number of studies for a few years. However, only sequential resolution algorithms have been proposed so far. This paper presents a parallel QCSP+ solving algorithm based on a problem-partition approach. It then discusses work distribution policies and presents several experimental results comparing several parameters.

I. INTRODUCTION

Quantified constraint satisfaction problems (QCSP) allow natural modeling of problems involving two enmies (such as two-player games), or problems with uncertainty (e.g. robust scheduling, testing...), which generally lies out of reasonable reach of the constraint satisfaction framework (CSP). A CSP consists in a set of variables and a set of constraints on these variables. A solution of a CSP is an assignment of the variables such that all the constraints are satisfied. Most CSP solvers find such a solution by recursively enumerating all possible values of one variable and propagating domains reductions. Since several years, parallel CSP solving procedures have been studied and tested against sequential ones. For example, a simple way to parallelize a CSP solver consists in solving every subproblem generated by the enumeration of a variable in parallel. Other frameworks have also been parallelized, such as Boolean formula satisfaction (SAT) and quantified boolean formulas (QBF). As far as we know, no parallel approach has ever been done to solve QCSP. This paper introduces such a parallel algorithm for solving QCSP+, an extension to QCSP introduced in [1], and presents some experiments on several instances of easy and hard problems.

II. PARALLEL SOLVING

Parallelism is largely used in artificial intelligence and operation research problems. In the more particular domain of CSPs, the idea of a parallel constraint solver or optimizer has been studied for years [7] [6]. Several approaches have been followed to have several processors solving one problem. A first one consists in dividing the search space of the problem into parts that the processing units will search independently. This problem partition is naturally given by the search tree: every node is itself a subproblem. In this approach, sharing the jobs can be done in several ways: a naive method consists in ensuring that the problem is split often enough to avoid starvation. Work stealing is a more reactive solution: a subproblem being solved is split as soon as a processor needs work. Another approach is the portfolio-based one: basically, it consists in running several solvers in parallel on the same problem, so that the most efficient solver for this problem will quickly give a solution. It may use communication, each instance sending its own deductions to each other.

The first approach is for example used in the Gecode CSP solver since version 3. The parallel version of the solver uses a work-stealing method to keep every thread active. In Boolean satisfaction (SAT), award-winning solver ManySAT [4] is portfolio-based. It uses a Master/Slaves hierarchy among the processing units and widely relies on information sharing between instances. In the quantified domain, the QBF solver PQSOLVE [2], is based of a problem-dividing approach with a workload balancing mechanism. More recently, Lewis et al. introduced PaQuBE in 2009 [5], which features conflict-clause sharing among processes.

III. THE QCSP+ FRAMEWORK

A. Formalism

1) CSPs: Let \( V \) be a set of variables. Each \( v \in V \) has got a domain \( D_v \). For a given \( W \subseteq V \), we denote \( D_W \) the set of tuples on \( W \), i.e. the cartesian product of the domains of all the variables of \( W \). A constraint \( c \) is a pair \( (W,T) \), \( W \) being a set of variables and \( T \subseteq D_W \) a set of tuples. The constraint is satisfied for the values of \( W \) which form a tuple of \( T \). If \( T = \emptyset \), the constraint is empty and can never be satisfied. On the other hand, a constraint such that \( T = D_W \) is full and will be satisfied whatever value its variables take. \( W \) and \( T \) are also denoted by \( \text{var}(c) \) and \( \text{sol}(c) \). A CSP is a set \( C \) of constraints. We denote \( \text{var}(C) \) the set of its variables, i.e. \( \bigcup_{c \in C} \text{var}(c) \) and \( \text{sol}(C) \) the set of its solutions, i.e. the set of tuples on \( \text{var}(C) \) satisfying all the constraints of \( C \). The empty CSP (denoted \( \top \)) is true whereas a CSP containing an empty constraint is false and denoted \( 
\).. An assignment is a set of couples \( (v,a) \), \( v \) being a variable and \( a \) a value, such that no variable appear twice. Given an assignment \( A \) of variables \( W \), we denote \( C[W \leftarrow A] \) the CSP obtained by reducing the domains of the variables of \( W \) to be values given by the assignment \( A \).
2) Quantified problems: A quantified set of variables (or qset) is a pair \((q, W)\) where \(q \in \{\forall, \exists\}\) and \(W\) is a set of variables. We call prefix a sequence of qsets \(\{(q_0, W_0), \ldots, (q_{n-1}, W_{n-1})\}\) where \(i \neq j \rightarrow (W_i \cap W_j = \emptyset)\). We denote \(\text{var}(P) = \bigcup_{i=0}^{n} W_i\). A Q CSP is a pair \((P, G)\) where \(P\) is a prefix and \(G\) is a CSP called the goal such that \(\text{var}(G) \subseteq \text{var}(P)\).

Q CSP+ have been formalized in [1] to restrict the quantifiers of a QCSP to chosen values. Such a restriction implies changing the nature of the qset: it includes a CSP whose solutions define the allowed values for the variables \(W_i\). QCSP+ are defined as follows:

A restricted quantified set of variables or qset is a triple \((q, W, C)\) where \((q, W)\) is a qset and \(C\) a CSP.

A Q CSP+ is a pair \(Q = (P, G)\) where \(P\) is a prefix of qsets such that \(\forall_i, \text{var}(C_i) \subseteq \bigcup_{j=0}^{m} W_j\). Moreover, \(\text{var}(G) \subseteq \text{var}(P)\) still holds.

3) Solution: A QCSP \((P, G)\) where \(P = [(\exists, W_0), (\forall, W_1), \ldots, (\exists, W_n)]\) represents the following logic formula: \(\exists W_0 \in D_{W_0} \exists W_1 \in W_1 \ldots \exists W_n G\).

Thus, a solution can not be a simple assignment of the variables: in fact, the goal has to be satisfied for all values the universally quantified variables may take. Intuitively, such variables: if the restrictor of a qset \(q\) is not relevant anymore (because the branch has been cut upper in the tree by another worker), \(M\) sends a signal ordering it to stop and fetch another job; (2) if \(M\) has no more job in its waiting list, it sends a signal to a worker to make

The set of scenarios of a winning strategy has a special structure: the inductive definition leads to the fact that if a scenario assign the variables \(W_1 \cup \ldots \cup W_i\) (among others) to values \(a_1 \ldots a_i\), one of these situations occur: (1) either \(q_{i+1} = \forall\) and every solutions of \(C +\) where \(W_1 = a_1 \ldots W_i = a_i\) is present at least in one scenario of the winning strategy; or (2) \(q_{i+1} = \exists\) and one solution of \(C +\) where \(W_1 = a_1 \ldots W_i = a_i\) is present at least in one scenario of the winning strategy.

A strategy can be represented a tree where each node at a given depth is labeled an assignment \(W_i\), such that every branch represents a unique scenario. So, given a node such that itself and the set of its ancestors give the assignment \(W_1 = a_1 \ldots W_i = a_i\), either \(q_{i+1} = \exists\) and this node has exactly one child labeled with a solution of \(C_{i+1}\), or \(q_{i+1} = \forall\) and the node has as many children as \(C_{i+1}\) has solutions.

B. QCSP+ solving procedure

The only publicly available QCSP+ solver is Q eCode [8]. It consists in a classical backtracking algorithm: qsets are considered from left to right, and the scenarios of the winning strategy are progressively built: given a QCSP+ \(Q\), a depth \(i\) (pointing to \((q_i, W_i, C_i)\) or \(t h(G)\)) and an assignment \(A = a_1 \ldots a_{i-1}\) of the variables \(W_1 \cup \ldots \cup W_{i-1}\), the recursive procedure returns a winning strategy for the corresponding subproblem: if \(q_i = \forall\), it consists in a set of winning sub-strategies for each solution of \(C_i\), whereas if \(q_i = \exists\), it consists in a winning sub-strategy for one solution of \(C_i\). When viewing the set of scenarios as a tree, each recursive call computes a sub-tree for a given node, which can be identified by the partial assignment \(A\).

IV. MULTITHREADED ALGORITHM

The multithreaded algorithm we propose has a Master-Slaves pattern based structure: a central data-structure called manager maintains a partial strategy (i.e. a strategy with missing scenarios), and calls several (single-threaded) workers to complete it. Those workers are just “classical” sequential QCSP+ solver instances. Because each incomplete branch yields a different subproblem, no direct communication between a worker and another is needed. thus, every messages transit between the manager and a worker.

The main communication between manager \(M\) and worker \(S\) consists in sending jobs from \(M\) to \(S\) and returning sub-strategies from \(S\) to \(M\). Formally, a job is a quadruplet \((Q, i, A, Sol)\) where \(Q\) is a QCSP+, \(i\) a depth and \(A\) an assignment \(W_1 \cup \ldots \cup W_i\), and \(Sol\) a set of solutions of \(C_i[W_1 \ldots W_i \leftarrow A]\). It describes a location of a subproblem needed to be solved, the remaining sub-branches yet to be explored are given by \(Sol\). Once \(S\) has finished solving a subproblem, it returns to \(M\) the corresponding sub-strategy and asks for another job. \(M\) can also send signals to \(S\) to prematurely stop it if needed: (1) if \(S\) is solving a subproblem that is not relevant anymore (because the branch has been cut upper in the tree by another worker), \(M\) sends a signal ordering it to stop and fetch another job; (2) if \(M\) has no more job in its waiting list, it sends a signal to a worker to make
it stop and return the partial sub-strategy already computed. Every incomplete branch being as many jobs available for other workers. Once the whole problem is solved, each worker is killed and the result is returned.

Each worker is constructed along with a thread executing the worker’s main loop. The manager’s controls and datastructure update are performed when a worker calls one of its methods. A mutual exclusion mechanism is placed on the Manager’s methods, so that only one thread may update the master at a time.

A. Workers

Each worker is an object having a very simple looping main procedure. This loop fetches a task from the Manager, and tries to solve it by calling an internal (single threaded) Solve method. The Manager can also possibly return a WAIT pseudo-task, which will cause the worker to sleep until a task becomes available, or by a STOP pseudo-task, which will exit from the main loop, and consequently stop the worker. Once the solver returns a solution, it is sent to the Manager. A worker is able to catch two signals called Terminate and Send_partial. Both indicates that the search procedure should stop, but the former means that the task has become useless while the later calls for returning a partial result plus the jobs left to do. The Solve method inherit from the QCSP\(^+\) search procedure presented above, modified in order to catch the signals in a reasonably short time. Figure 1 presents an example of interaction between a worker and the Manager.

As a worker is just a simple-threaded QCSP\(^+\) solver, any tuning of a sequential quantified constraint solver also apply here. In particular, choosing search heuristics is a natural issue that might greatly impact the solving time as it already does for CSP and sequential QCSP\(^+\) solvers.

B. Manager

The manager is an object containing a partial winning strategy of the problem, and distributes jobs to the workers to complete it. During search, this strategy miss some scenarios, but also potentially have some in excess: in fact, several workers may be at the same time solving subproblems corresponding to several solutions of the same existential restrictor. In this case, each may return a partial sub-strategy (if asked to stop and share its work), though only at most one of these will be kept in the final winning strategy. Considering the tree representation of the strategy, it corresponds to brothers existential nodes, that will eventually be cut when a worker extracts a complete sub-strategy for one of them.

During search, the winning strategy is incomplete. In order to keep trace of incomplete branches in the strategy, special “todo” nodes are inserted at their position. The corresponding jobs are listed in a list called ToDo. A list Current_Workers of the workers currently solving a job, as well as a list of sleeping workers (waiting for a job) are also maintained.

The fetchWork method is called by a worker and returns a task from the ToDo list. If ToDo is empty, it sends the Send_partial signal to one worker from Current_Workers and returns WAIT to the calling worker. If Current_Workers is also empty, the search has ended and the manager returns STOP.

The returnWork method is called by a worker which has finished a job. It attaches the returned sub-strategy to the incomplete strategy and cuts the branches that are no longer necessary. Each worker that was solving a node on a cut branch are sent the Terminate signal. A worker calling this method may also submit an incomplete work. In this case, the incomplete substrategy is attached, the remaining jobs for this subproblem are queued in the ToDo list and the workers contained in the sleeping list are awoken.

1) Manager tuning: The manager has to set various priorities that may notably affect the solving time. First, a priority has to be defined when choosing which job, among the ones in the ToDo list, will be assigned to a worker first: which are the most “promising” jobs? The manager not being a solver, this comparison between jobs can only consider their position in the strategy (in particular: depth and quantifier). Another choice to make is: which worker do we stop in order to share its work when needed? This choice may consider the position of the subproblem being solved (a job near the root is more likely to be cut into more pieces), as well as the execution time of the worker (it is less interesting to share an almost done work).

V. EXPERIMENTAL RESULTS

A. Experiments

The connect-4 is a two-players game consisting in a vertical board having 7 rows and 6 columns in which each player drops a counter in turn (one player has yellow counters, and its opponent red ones). The aim of the game is to be the first to align 4 counters of its color, horizontally, vertically or diagonally. In this game, each player choose only the column to play, as the counter will drop to the lower unoccupied place. The game of Nim consists in a heap of counters from which each player take in turn one, two or three elements, the aim of the game being picking the last element. The variant we
modeled consists in letting the players take up to twice the number of counters that have been picked up by their opponent at the previous turn.

We solved the $6\times7$ connect-4 game with a depth from 11 to 14 (14 being solved by QeCode in less than 1 hour), and the variation of the Nim game with an initial number of counters from 30 to 39 (this last cast taking 1000 seconds to qecode to solve it). Lower values leads to immediately-solvable problems, which are not relevant for testing. The workers used an appropriate branching heuristics (especially, the connect-4 has an ad-hoc value heuristics). We used several jobs-priority heuristics for each instance: first-in-firt-out, deepest first, shallowest first, universal first (then, deepest or shallowest first), and existential first (idem). Each one of these instances was run with a maximum number of workers varying from 1 to 8. Those tests were run on machines equipped with two quad-core Opteron at 2.3 GHz, thus every worker had a dedicated processor core to execute itself. Each instance was run 7 times, in order to measure the influence of the parallelism-inherent indeterminism on computation time.

### B. Results

The most striking result is the great difference of execution time in a same set of runs: even for the hardest problems, the fastest run may complete in half the time taken by the lowest. The benefit of the jobs-ordering heuristics can very hardly be demonstrated. Average times does not give a clear speed-up for hard problems. This constitutes a first approach to parallel quantified constraint satisfaction solving, that shows noticeable benefits in terms of resolution time, and calls to further investigations on efficiency of job-sharing policies and search heuristics in a parallel solver.

### REFERENCES