A Multithreaded solving algorithm for QCSP+

Jérémie Vautard, Arnaud Lallouet

A multithreaded solving algorithm for QCSP$^+$

Jérémie Vautard (student)[1] and Arnaud Lallouet[2]

[1] Université d'Orléans — LIFO
BP6759, F-45067 Orléans
jeremie.vautard@univ-orleans.fr
[2] Université de Caen-Basse Normandie — GREYC
BP 5186 - 14032 Caen
arnaud.lallouet@info.unicaen.fr

Abstract. This paper presents some ideas about multi-threading QCSP solving procedures. We introduce a first draft of a multi-threaded algorithm for solving QCSP$^+$ and give some work leads about parallel solving of quantified problems.

1 Introduction

Quantified constraint satisfaction problems (QCSP) have been studied for several years, and many search procedures ([7] [5]), consistency definitions([6] [4]) and propagations algorithms([3] [1]) have been proposed to solve them. However, while several attempts have been done to make a parallel solver for CSPs, we have not found any parallel approach for solving quantified problem. This is what we try to do in this paper. First, we propose a sketch of a quite general framework for solving QCSP$^+$ problems (introduced in [2]) using a multi-threaded approach. Then, we discuss about several leads that can be explored in this domain.

2 The QCSP$^+$ framework

2.1 Formalism

Variables, constraints and 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 ant $T \in D^W$ a set of tuples. The constraint is *satisfied* for the values of the variables of W which form a tuple of T. If $T = \emptyset$, the constraint is said to be *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 $var(c)$ and $sol(c)$.

A *CSP* is a set C of constraints. We denote $var(C)$ the set of its variables, i.e. $\bigcup_{c \in C} var(c)$ and $sol(C)$ the set of its solutions, i.e. the set of tuples on $var(C)$ satisfying all the constraints of C. The empty CSP (denoted $\top$) is true whereas a CSP containing an empty constraint is false and denoted $\bot$.

Quantified problems.

A *quantified set of variables* (or *qset*) is a pair (q, W) where $q \in \{\forall, \exists\}$ is a quantifier and W a set of variables. We call a *prefix* a sequence of qsets $[(q_0, W_0), \ldots, (q_{n-1}, W_{n-1})]$ such that $(i \neq j) \rightarrow (W_i \cap W_j = \emptyset)$. We denote $var(P) = \bigcup_{i=0}^{n-1} W_i$. A *QCSP* is a pair (P, G) where P is a prefix and G, also called the *goal*, is a CSP such that $var(C) \in var(P)$.

A *restricted quantified set of variables* or *rqset* is a triple (q, W, C) where (q, W) is a qset and C a CSP. A $QCSP^+$ is a pair $Q = (P, G)$ where P is a prefix of rqsets such that $\forall i, var(C_i) \subseteq \bigcup_{j=0}^{i} W_j$. Moreover, $var(G) \subseteq var(P)$ still holds.

Solution.

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

Thus, a solution of a quantified problem can not be a simple assignment of all the variables anymore : in fact, the goal has to be satisfied for all values the universally quantified variables may take. Intuitively, such a problem can be seen as a "game" where an *existential player* tries to satisfy all the constraints of G while a *universal player* aims at violating one of them, each player assigning the variables in turn, in the order defined by the prefix. The solution must represent the *strategy* that the existential player should adopt to be sure that, whatever its opponent do, the goal will always be satisfied. This strategy can be represented as a family of Skolem functions that give a value to an existential variable as a function of the preceding universal ones, or by the set of every possible scenario (i.e. total assignment of the variables) of the strategy. In this paper, we use this later representation and organize the set of scenarios in a tree : a root node represents the whole problem, then, inductively :

- if the next qset (q_i, W_i) is universal, the current node gets as many sons as there are tuples in D^{W_i}. Each node is tagged vith one of these tuples ;
- if the next qset is existential, the current node gets one unique son, tagged by an element of D^{W_i}.

Thus, each complete branch of this tree coresponds to a total assignment of the variables of the problem. If every branch of such a tree corresponds to an assignment satisfying G, then it is indeed a solution of the problem.

QCSP^+ restricts the "moves" of each player to assignments that satisfy the CSP of the rqsets. The logic formula represented is :
$\exists W_0 \; C_0 \wedge (\forall W_1 \; C_1 \rightarrow (\exists W_2 \ldots \quad G))$

In this case, the notion of solution is the same, except that the restrictions have to be taken into account : for universal rqsets $(\forall, W_i, C_i)$ the current node's sons corresponds to each solution of C_i. For existential rqsets $(\exists, W_j, C_j)$, the son must be tagged by an element such that the partial branch corresponds to an assignment that satisfies C_j.

2.2 A basic solving procedure

One simple way to solve quantified problems is to adapt the classical backtracking algorithm for CSPs :

- first, perform a propagation algorithm on the problem. If an inconsistency is detected, return $false$.
- pick up the leftmost quantified set of variables and enumerate the possible values of its variables, dividing the problem in as many subproblems.
  - in the universal case, solve all the subproblems. If one of them is false, return $false$. Else, group all the corresponding substrategies and return them.
  - in the existential case, solve each problem until one of them does not return $false$. if such a subproblem exists, create a node containing the values that led to the corresponding subproblem, attach the substrategy returned by the subproblem and return the whole. In the other case, return $false$.

3 Multithreaded solving : a first attempt

The multithreaded solving method we propose is based on a central data-structure called *manager* managing several (single threaded) solvers called *workers* : a partial strategy is maintained, where some nodes do not father a substrategy. Each of these nodes corresponds to a *task* that a worker can solve. Formally, a task consists in a pair (Q, τ) where Q is a QCSP$^+$ and τ the partial assignment of the variables of Q corresponding to the branch of the partial strategy where the task is attached. Once a worker has finished solving a task, it returns its result (either a sub-strategy or $false$) that will be taken into account by the manager to update the partial strategy. Then, the worker receives another task to solve.

Whenever the to-do tasks queue empties, the manager sends a signal to one or several workers to stop its current task, and send a partial result. Such a result consists in a partial sub-strategy containing "unfinished" nodes which are as many other pending tasks.

Once the whole problem is solved (i.e. there is neither more to-do task left nor other working thread), each worker thread is killed, and the result can be returned. Here is a list of each procedures and signals used in this framework :

Workers.
Each worker is a thread having a very simple main loop. 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 answer by a *WAIT* pseudo-task, which will cause the worker to sleep until a task becomes available (by calling a special method of the Manager), or by a *STOP* pseudo-task, which will kill the thread. Once the solver finishes, its result is sent to the Manager. This main loop is described in figure 1.

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, along with a list of remaining "sub-tasks".

The *Solve* method can inherit from any original search procedure, but must be modified in order to take the signals into account. Figure 2 shows an adaptation from a basic QCSP$^+$ solving procedure which can be interrupted by these signals.

Finally, a worker provides some methods so that other processes know which subproblem it is working on.

```
Procedure Main
  loop
    task = Manager.fetchWork()
    if task == STOP then
      Exit
    else if task == WAIT then
      Manager.wait()
    else
      result = Solve(task)
      Manager.returnWork(result)
    end if
  end loop
```

Fig. 1. The worker main loop

Manager.

The manager is an object that contains and builds the winning strategy of the problem. During search, this winning strategy is incomplete and some nodes are replaced by tasks remaining to solve. We call *ToDo* this list of remaining tasks. A Manager is also aware of the list *Current_Workers* of the workers currently solving a task and maintains a list of *sleeping* workers that should be waken when tasks become available for solving. Unless sait otherwise, the Manager's methods are called by a worker.

The *fetchWork* method withdraws a task from the ToDo list and returns it. If ToDo is empty, then it sends the signal *Send_partial* to one worker from Current_Workers and returns WAIT. If Current_Workers is also empty, the search has ended and therefore, STOP is returned.

The *returnWork* method attaches the returned sub-strategy and cuts the branches that are no longer necessary (for example, every brothers of a complete substrategy of an existential node, or directly the father node of a universal substrategy if one of the subproblems have been found to be inconsistent). Each worker that was solving a node on a cut branch are sent the *Terminate* signal, and the workers contained in the *sleeping* list are awoken. Finally, the *wait*

```
Procedure
Solve_e ([(∃, W, C)|P'], G)
  S_C = Set of solutions of C
  while S_C ≠ ∅ do
    choose t ∈ S_C ; S_C = S_C − t
    if Signal Terminate then
      return STOP
    end if
    CURSTR := Solve( (P', G)[W ← t] )
    if CURSTR ≠ Fail then
      if Signal Send_Partial then
        return
        PartialResult(tree(t,CURSTR),S_C)
      else
        return tree(t,CUR_STR)
      end if
    end if
  end while
  return Fail
```

```
Procedure Solve_u ([(∀, W, C)|P'], G)
  STR := ∅
  S_C = Set of solutions of C
  while S_C ≠ ∅ do
    choose t ∈ S_C ; S_C = S_C − t
    if Signal Terminate then
      return STOP
    end if
    CURSTR := Solve( (P', G)[W ← t] )
    if CURSTR = Fail then
      return Fail
    else
      STR := STR ⋃ CURSTR
    end if
    if Signal Send_Partial then
      return PartialResult(STR,S_C)
    end if
  end while
  return STR
```

Fig. 2. Search procedure.

method records the calling thread in the *Sleeping* list and puts it in sleeping mode, until it is waken by the previous method.

4 Work leads

Search heuristics.

The time taken by a search procedure to solve a CSP greatly depends on the heuristics used to choose which subproblem should be explored first. Unfortunately, most parallel approaches tends to be incompatible with this heuristics, thus ruining solving performances. QCSP (and QCSP$^+$), reduce the alternatives, as a solver have to follow the order of the prefix, and the impact of the heuristics used to perform these choices remains unclear, because they were not originally tailored for QCSPs. In 2008, Verger and Bessière presented in [8] a promising heuristics for QCSP$^+$ that accelerate solving by several orders of magnitude on some problems. Parallelizing the search might, as for CSPs, drawn the benefit of theses heuristics.

Task priority.

In the presented parallel approach, each task could be given a priority, in order to minimize pointless subproblems solving. The method used to calculate this priority will most likely the solving time: in fact, solving one given subproblem might be pointless or not according to the result of another given task, and the priority given to the tasks should take that into account. For example, it sounds reasonable to give top priority to leftmost universal nodes as every branch from a universal node must be verified whatsoever, a single failure cutting the whole

subproblem. After that, solving rightmost pending existential tasks should help finishing to build complete sub-strategies.

Several kind of workers.

The workers run independently from each other, and their communications with the Manager are not specific to a particular search procedure. Thus, using the algorithm of figure 2 in the workers is not mandatory. Any procedure able to return the sub-strategy of a problem is *a priori* appropriate. However, algorithms that can not return partial work and generate remaining tasks should not be used, as they will tend to prematurely dry the ToDo list, bringing other workers into sleep mode.

5 Conclusion

Solving quantified constraint satisfaction problems in a parallel way is new, and even this contribution is far from being achieved. thus, while looking interesting, this still needs to prove its worth. We presented here a quite simple and general framework that has to be implemented and tested against traditional solvers before drawing definitive conclusions.

References

1. Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Reusing csp propagators for qcsps. In Francisco Azevedo, Pedro Barahona, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 4651 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2006.
2. Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Qcsp made practical by virtue of restricted quantification. In Manuela M. Veloso, editor, *IJCAI*, pages 38–43, 2007.
3. Lucas Bordeaux, Marco Cadoli, and Toni Mancini. Csp properties for quantified constraints: Definitions and complexity. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 360–365. AAAI Press / The MIT Press, 2005.
4. Lucas Bordeaux and Eric Monfroy. Beyond np: Arc-consistency for quantified constraints. In Pascal Van Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 371–386. Springer, 2002.
5. Ian P. Gent, Peter Nightingale, and Kostas Stergiou. Qcsp-solve: A solver for quantified constraint satisfaction problems. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 138–143. Professional Book Center, 2005.
6. Peter Nightingale. Consistency for quantified constraint satisfaction problems. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 792–796. Springer, 2005.
7. Guillaume Verger and Christian Bessiere. Blocksolve: a bottom-up approach for solving quantified csps. In *Proceedings of CP'06*, pages 635–649, Nantes, France, 2006.
8. Guillaume Verger and Christian Bessiere. Guiding search in qcsp$^+$ with backpropagation. In *Proceedings of CP'08*, pages 175–189, Sydney, Australia, 2008.