A Simple Distributed Conflict-Driven Answer Set Solver

E. Ellguth[1], M. Gebser[1], M. Gusowski[1], R. Kaminski[1], B. Kaufmann[1], S. Liske[1], T. Schaub[1⋆], L. Schneidenbach[2], and B. Schnor[1]

[1] Institut für Informatik, Universität Potsdam, D-14482 Potsdam, Germany
[2] IBM Ireland, Dublin Software Lab, Mulhuddart, Dublin 15, Ireland

Abstract. We propose an approach to distributed Answer Set Solving based on Message Passing. Our approach aims at taking advantage of modern ASP solvers rather than proposing a genuine yet involved parallel ASP solver. To this end, we rely upon a simple master-worker architecture in which each worker amounts to an off-the-shelf ASP solver augmented with a separate communication module being only lightly connected to the actual solver. The overall communication is driven by the workers' communication modules, which asynchronously exchange messages with the master. We have implemented our approach and report upon an empirical study demonstrating its computational impact.

1 Introduction

Despite the progress of sequential Answer Set Solving technology, only little advancement is observed in the parallel setting. This is deplorable in view of the rapidly growing availability of clustered, multi-processor, and/or multi-core computing devices. We address this shortcoming and furnish a distributed approach to ASP solving by focusing on the parallelization of search. Our approach builds upon the Message Passing Interface (MPI; [1]), realizing communication and data exchange between computing units via message passing. Interestingly, MPI abstracts from the actual hardware and lets us execute our system on clusters as well as multi-processor and/or multi-core machines.

We aim at a simple and transparent approach in order to be able to take advantage of the high performance offered by modern off-the-shelf ASP solvers. To this end, we have chosen a simple master-worker architecture, in which each worker consists of an ASP solver along with an attached communication module. The solver is linked to its communication module via an elementary interface requiring only marginal modifications to the solver. All major communication is initiated by the workers' communication modules exchanging messages with the master in an asynchronous way.

2 Distributed Answer Set Solving

We have implemented our approach in `C++` using MPI [1]. The resulting system is called *claspar*, alluding to its underlying ASP solver *clasp* [2]. Although we tried to keep our design generic, we took advantage of some design features of *clasp*, whose basic search procedure can be outlined by means of the following loop [3]:

⋆ Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

```
loop
    propagate                                   // compute deterministic consequences
    if no conflict then
        if all variables assigned then return variable assignment
        else decide                             // choose a non-deterministic consequence
    else
        if top-level conflict then return unsatisfiable
        else
            analyze                 // analyze conflict and add a conflict constraint
            backjump                // undo assignments until conflict constraint is unit
```

At first, the closure under deterministic consequence operations is computed. Then, four cases are distinguished. In the first one, a non-conflicting complete assignment is returned. In the second case, an unassigned variable is non-deterministically chosen and assigned. Or at last, a conflict is encountered. All assignments made before the first non-deterministic choice constitute the *top-level*. Hence, a top-level conflict indicates unsatisfiability. Otherwise, the conflict is analyzed and learned in form of a conflict constraint. Then, the algorithm backjumps by undoing a maximum number of successive assignments so that exactly one literal of the constraint is unassigned.

The *clasp* solver extends the static concept of a top-level by additionally providing a dynamic variant referred to as *root-level* [3]. As with the top-level, conflicts within the root-level cannot be resolved given that all of its variable assignments are precluded from backtracking. We build upon this feature for splitting the search space. Splitting is accomplished according to a so-called *guiding path* [4], the sequence of all non-deterministic choices. Given a root-level $i-1$, a guiding path $(v_1, \ldots, v_{i-1}, v_i, \ldots, v_n)$ can be divided into a prefix $(v_1, \ldots, v_{i-1})$ of non-splittable variables and a postfix $(v_i, \ldots, v_n)$ of splittable variables. We can split the search space at the first splittable variable by incrementing the root-level by one and dissociating a guiding path composed of the first $i-1$ variables and the complement of the ith variable, yielding $(v_1, \ldots, v_{i-1}, \overline{v_i})$. Note that the local assignment remains unchanged, and only the root-level is incremented to i. We have chosen to split at the first splittable variable because, first, this results in cutting off the largest part of the search space and, second, this way the backjumping is least restricted.

Upon enumerating answer sets, (locally) using the scheme in [5], the assignment can contain complements of non-deterministically assigned variables of previously enumerated answer sets. Such complements $\overline{u_1}, \ldots, \overline{u_j}$ indicate that the search spaces for answer sets containing $(v_1, \ldots, v_{i-1})$ and at least one of $u_1, \ldots, u_j$ have already been explored, while v_i or $\overline{v_i}$ may have belonged to already enumerated answer sets. In order to avoid repetitions, it is thus important to pass guiding path $(v_1, \ldots, v_{i-1}, \overline{u_1}, \ldots, \overline{u_j}, \overline{v_i})$ in response to a split request. This refinement for repetition-free answer set enumeration is implemented in *claspar*.

Finally, *clasp* incorporates constraint database simplifications wrt variables assigned at the top-level. In particular, conflict constraints can lead to top-level assignments, in which case the corresponding variables are eliminated from all resident constraints. The root-level plays a crucial role for whether such simplifications are applied, as variables assigned at or below the root-level but beyond the top-level are not subject to simplification. This feature is also inherited by *claspar* setting the root-level to

the number of variables in a guiding path. As a consequence, conflicts and resulting assignments due to a nonempty guiding path (or subproblem, respectively) do not involve simplification, while top-level assignments independent of the guiding path lead to simplifications.

3 Communication

Our approach to distribution builds upon message passing, accounting for communication and data exchange. For the sake of simplicity, we have adopted a classical master-worker model. While the purpose of the single master is to handle the overall message exchange, each worker amounts to an ASP solver enhanced by message handling capacities. The workers constitute the active components, initiating all requests, while the master mainly reacts by processing the workers' requests.

Master. The main task of the master is the reception and transmission of search (sub)problems. To accomplish this, the master divides its assigned workers into a set of active and inactive workers. The active workers, i.e., workers assigned a not yet processed guiding path, are arranged in a queue ordered by a workload parameter. On the other hand, the inactive workers have either finished processing their guiding paths or have not yet been assigned any.

At the beginning, the search space has to be distributed among the workers. As initially all workers are inactive, the master receives a work request from each worker. The first incoming work request obtains the empty guiding path, representing the entire search space. It is then successively split and distributed among the other workers.

The overall routine of the master is driven by load balancing. A work request by a worker normally results in a split request to another worker. The split request is sent to a worker with putatively high workload, namely, to one with a short guiding path. Notably, each worker determines whether and/or how often it is asked (and thus interrupted) for work. The master merely maintains its priority queue according to the information supplied by the workers.[3] When a subproblem is returned to the master by a split response, it is forwarded to the first worker in the request queue or put into a cache to allow for immediate response to the next work request. Apart from the guiding path, a split response also contains information on the workload of the sending worker.

Whenever all workers are inactive and the cache is empty, the given problem is found to be unsatisfiable. As soon as the requested number of answer sets is computed or unsatisfiability is established, the master asks all workers to gather runtime statistics and to then terminate. Once all statistics are received, they are aggregated and printed before the master terminates itself.

Worker. Our worker design is driven by the desire to minimize modifications to the given ASP solver while keeping the overall approach as simple as possible. To this end, we attach a module handling communications, providing an interface reacting to incoming messages during search. The interface is used at the end of the conflict analysis in the solver loop given in Section 2. This is the only change done to the ASP solver at hand (except for redirecting its output operations).

[3] Currently, this information consists of the length of the initial guiding path and the number of choices made by the worker's ASP solver since the guiding path was received.

However, in general, the worker's communication module has two modes of operation. When out of work (no guiding path received yet or completed subproblem), the first mode cares about raising a work request and launching its ASP solver with the guiding path obtained from the master. Notably, upon such a response, *claspar* reuses the previously learnt clauses and heuristic information like variable activities. The second mode addresses split requests. For this purpose, the communication module is equipped with a heuristic function for deciding whether a guiding path is extracted from the ASP solver. If so, a guiding path is sent to the master (accompanied with some information on the workload). The current strategy is to return the shortest guiding path.

If the worker decides not to split its search space, it can signal to retry later or to send no further split requests. Once an answer set is found, the worker sends it to the master. In the case of unsatisfiability, conflict constraints accumulated over time may eventually yield a top-level conflict that is also signaled to the master, which then asks all workers to send their statistics and to terminate.

4 Experiments

Our experiments consider *claspar*[4] (0.1.0) based on *clasp* (1.0.5); they ran under MPI (mpich2-1.0.7) on the cluster described at `http://www.cs.uni-potsdam.de/bs/research/labs/highland.html`, each individual run restricted to 900s time and 2GB RAM per worker. Each solver instance of *claspar* was run with the default settings of *clasp* (except for the second group of *PigeonHole* benchmarks).

Table 1 summarizes benchmark results capturing the scaling capacities of *claspar*.[5] We consider a master run on a single machine plus increasing numbers of workers running on machines with double cores (and thus at most two MPI processes per machine). The single worker setting amounts to that of a serial run of *clasp*: the same number of choices and conflicts are obtained for each run, and the message passing overhead leads to an increase in execution time of less than one percent. We have selected popular benchmark classes for evaluating our approach. Among them, *BlockedQueens-sat* are satisfiable and terminated after an answer set was obtained. All remaining benchmarks are unsatisfiable and thus necessitate a complete traversal of the search space. Each setting is described by the sums of times,[6] and timeouts over all underlying instances.[7] Moreover, we give the relative speedup wrt the single worker setting and indicate the efficiency by normalizing the speedup with the number of workers in each setting. In general, we observe a steady increase in speedup although the efficiency goes down with more workers due to greater overhead. An unsteady speedup is observed on the class of satisfiable benchmarks. Even though the search for one answer set can be boosted by lucky strikes, it is surprising to see that 8 workers performed better than 16. This is different on the unsatisfiable instances in *BlockedQueens-unsat* that show a steady yet suboptimal speedup. The *GraphColoring* benchmarks are taken from this year's ASP

[4] Available at `http://potassco.sourceforge.net` .

[5] The detailed table is available at `http://www.cs.uni-potsdam.de/claspar` .

[6] Timeouts are taken as maximum time, viz., 900s.

[7] A tarball containing all benchmark instances is available at the URL given in Footnote 5.

claspar 0.1.0	*1 worker*	*2 workers*	*4 workers*	*8 workers*	*16 workers*
Benchmark	*BlockedQueens-sat (7 instances)*				
Time (Timeouts)	678.82 (0)	444.06 (0)	248.47 (0)	99.74 (0)	116.23 (0)
Speedup (Efficiency)	1.00 (1.00)	1.53 (0.76)	2.73 (0.68)	6.81 (0.85)	5.84 (0.37)
Benchmark	*BlockedQueens-unsat (9 instances)*				
Time (Timeouts)	1528.72 (0)	1223.84 (0)	649.57 (0)	401.74 (0)	191.31 (0)
Speedup (Efficiency)	1.00 (1.00)	1.25 (0.62)	2.35 (0.59)	3.81 (0.48)	7.99 (0.50)
Benchmark	*GraphColoring (20 instances)*				
Time (Timeouts)	18000 (20)	16993.80 (18)	12841.70 (11)	9910.80 (6)	7063.68 (3)
Speedup (Efficiency)	1.00 (1.00)	1.06 (0.53)	1.40 (0.35)	1.82 (0.23)	2.55 (0.16)
Benchmark	*PigeonHole (2 instances)*				
Time (Timeouts)	975.66 (0)	988.66 (1)	469.45 (0)	190.48 (0)	153.20 (0)
Speedup (Efficiency)	1.00 (1.00)	0.99 (0.49)	2.08 (0.52)	5.12 (0.64)	6.37 (0.40)
Benchmark	*PigeonHole-norestarts (2 instances)*				
Time (Timeouts)	984.67 (1)	587.38 (0)	293.10 (0)	169.63 (0)	85.32 (0)
Speedup (Efficiency)	1.00 (1.00)	1.68 (0.84)	3.36 (0.84)	5.80 (0.73)	11.54 (0.72)

Table 1. Scaling *claspar* from 1 to 16 workers.

claspar 0.1.0	*1 worker*	*2 workers*	*4 workers*	*8 workers*	*16 workers*
Benchmark	*ClumpyGraphs (12 instances)*				
Models	83,866,664	153,764,698	312,272,614	610,673,252	1,247,804,500
Speedup (Efficiency)	1.00 (1.00)	1.83 (0.92)	3.72 (0.93)	7.28 (0.91)	14.88 (0.93)

Table 2. Scaling *claspar*'s enumeration of answer sets within 900s from 1 to 16 workers.

competition. Here the speedup values are insignificant because one worker alone cannot even solve a single instance. However, increasing the number of workers leads to a steady decrease of timeouts, demonstrating that distribution can make a difference. Finally, we consider the well-known *PigeonHole* example, spanning a very uniform search tree. The default behavior of *claspar* allows solver instances to restart, which has a negative effect on the individual and thus global solver performance. Once restarts are inhibited, reasonable speedups are obtained.

Finally, we consider in Table 2 how *claspar* scales regarding the enumeration of answer sets. For this, we substitute runtimes by the number of answer sets obtained within 900s. We consider Hamiltonian cycles in *ClumpyGraphs*, a benchmark designed in [6] for showing the advantage of conflict-driven learning ASP solvers. We observe that the distribution of search incurs only minor overhead in *claspar*'s efficiency, and the speedup closely follows the number of workers. This nicely demonstrates the computational impact of distributed ASP solving.

5 Discussion

We proposed a simple approach to distributed ASP solving based on the well-known Message Passing Interface. To this end, we rely upon a master-worker architecture in which each worker amounts to an off-the-shelf ASP solver augmented with a separate

communication module being only lightly connected to the actual solver. The simplicity is pragmatically best reflected by the fact that *claspar* changes less than a dozen lines of code in *clasp* while adding another $\sim$350 lines for handling distribution.

Our approach differs from existing work in several respects. In fact, it provides the first distributed version of an ASP solver using conflict-driven learning. Unlike this, existing distributed ASP solvers rely on classical backtracking schemes that provide a much tighter control over the search space. Second, our approach endows each solver instance with great independence, principally allowing for a variety of different ASP solvers at the same time. Although the *platypus* framework [7, 8] also aims at a certain genericity, this applies only to the deterministic part of the solvers, while distributed search is controlled by *platypus* itself. The approach of [9, 10] goes even further in building a genuine parallel solver based on sequential ASP solver *smodels*.

Acknowledgments. This work was partially funded by DFG grant SCHA 550/8-1.

References

1. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press (1999)
2. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proceedings IJCAI'07, AAAI Press/MIT Press (2007) 386–392
3. Eén, N., Sörensson, N.: An extensible SAT-solver. Proceedings SAT'03. (2003) 502–518
4. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **21**(4) (1996) 543–560
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. Proceedings LPNMR'07, Springer (2007) 136–148
6. Ward, J., Schlipf, J.: Answer set programming with clause learning. Proceedings LPNMR'04, Springer (2004) 302–313
7. Gressmann, J., Janhunen, T., Mercer, R., Schaub, T., Thiele, S., Tichy, R.: Platypus: A platform for distributed answer set solving. Proceedings LPNMR'05, Springer (2005) 227–239
8. Gressmann, J., Janhunen, T., Mercer, R., Schaub, T., Thiele, S., Tichy, R.: On probing and multi-threading in platypus. Proceedings ECAI'06, IOS Press (2006) 392–396
9. Pontelli, E., Balduccini, M., Bermudez, F.: Non-monotonic reasoning on Beowulf platforms. Proceedings PADL'03, Springer (2003) 37–57
10. Balduccini, M., Pontelli, E., El-Khatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. Parallel Computing **31**(6) (2005) 608–647