Experiences Running a Parallel Answer Set Solver on Blue Gene

Lars Schneidenbach[2], Bettina Schnor[1], Martin Gebser[1], Roland Kaminski[1], Benjamin Kaufmann[1], and Torsten Schaub*[1]

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

Abstract. This paper presents the concept of parallelisation of a solver for Answer Set Programming (ASP). While there already exist some approaches to parallel ASP solving, there was a lack of a parallel version of the powerful *clasp* solver. We implemented a parallel version of *clasp* based on message-passing. Experimental results on Blue Gene P/L indicate the potential of such an approach.

Keywords: Applications based on Message-Passing, Answer Set Programming, Performance evaluation

1 Introduction

Answer Set Programming (ASP) [1] has become a popular tool for knowledge representation and reasoning. Apart from its rich modelling language, provably more expressive [2] than the ones of neighbouring declarative programming paradigms, as for example Satisfiability checking (SAT), its attractiveness is due to the availability of efficient ASP solvers. In fact, modern ASP solvers rely on advanced Boolean constraint solving technology (cf. [3]), making them competitive with state-of-the-art SAT solvers.

In particular, the sequential ASP solver *clasp* [4, 5] is a modern ASP solver incorporating conflict-driven learning, backjumping, restarts, etc. Even though *clasp* is a powerful tool for tackling (NP-hard) search problems, improving the solving time is still a challenge. We thus implemented a parallel *clasp* version, the so-called *claspar* solver, which is portable over a wide range of platforms. Making use of the Message-Passing paradigm, it is particularly designed for distributed memory machines, like clusters are.

2 Related Work

There already are a few approaches to parallel ASP solving. The approaches presented in [6, 7] aim at building a genuine parallel solver, running on Beowulf clusters, based on the sequential ASP solver *smodels* [8]. The approach of [9] is to provide a platform for distributing ASP solvers (viz., *smodels* and *nomore++* [10]) in various settings. It supports combinations of Forking, Multi-Threading, and Cluster-Computing using

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

MPI. None of these approaches incorporates modern ASP solving techniques such as conflict-driven learning and backjumping. These are the strengths of the *clasp* solver.

Unlike this, there already are distributed SAT solvers incorporating conflict-driven learning, like *pasat* [11], *ysat* [12], *pamira* [13], and *miraxt* [14].

3 Parallelisation Concept

We use the concept of a *guiding path*, which was first implemented within the parallel SAT solver *psato* [15], for workload description. A guiding path defines a path in the search tree from the root node down to the current node with a sub-tree to investigate.

For partitioning the search space captured by an assignment $(v_1, \ldots, v_i, \ldots, v_n)$, one selects a splittable (Boolean) variable v_i (with $1 \leq i \leq n$) and generates a new guiding path $(v_1, \ldots, \overline{v_i})$ by flipping the truth value assigned to v_i and marking all elements of the path as non-splittable. Also, v_i is marked as non-splittable in the original assignment $(v_1, \ldots, v_i, \ldots, v_n)$. A systematic application of the guiding path method provides a genuine partition of the search space. This can be exploited for dynamically balancing the workload between distributed solver instances. For handling guiding paths, *clasp* extends the static concept of a *top level* assignment by additionally providing a dynamic variant referred to as *root level*. As with the top level, conflicts within the root level cannot be resolved given that all of its variables are precluded from backtracking.

A heuristic method to estimate the remaining load of a worker is the length of its guiding path. A short guiding path means that only few decisions were made, so that a lot of work may still remain to be done. Hence, we always select the first splittable variable v_i in an assignment $(v_1, \ldots, v_i, \ldots, v_n)$ to construct a new guiding path $(v_1, \ldots, \overline{v_i})$.

3.1 Master-Worker versus P2P Approach

If we could evenly distribute the workload (i. e. search sub-trees), we could simply start a number of *clasp* solver processes investigating predefined sub-trees. But such an approach is impossible, as we do not know the size of a sub-tree in advance. Starting from any load distribution, sooner or later there will be a load imbalance: some solver instances are still busy, while others are already finished. Thus we need load balancing.

Load balancing techniques can be divided into central, hierarchical, and distributed P2P algorithms. It is important for an efficient distributed load balancing algorithm to know whether a worker is working on a huge branch of the search tree (considered as high load) or whether a worker is almost finished and will ask for additional work soon (low load). This is in advance unknown in our application and varies between inputs.

For the first parallelisation step, we decided to use a central approach where the load distribution is managed by a central component, the *master*. The running *clasp* processes are called *workers*. When a worker has finished processing its search sub-tree, it sends a `work request` to the master. The master then determines another worker that seems to have high load and asks it via a `split request` to split its search space. If the current assignment of the asked worker contains a splittable variable, it

returns a new guiding path in terms of a `split response`. The master uses a *work cache* (see Section 3.3) to reduce the average answer time to work requests. In summary, the tasks of the master are

- receiving a `work request` from a worker w_1,
- sending a `split request` to a worker w_2 asking to split its search space,
- receiving a `split response` from worker w_2,
- sending a `work response` to worker w_1,
- the calculation of overall search statistics (choices, conflicts, restarts, etc.),
- the output of answer sets and statistics, and
- the termination of all workers.

The master has to do a lot of message handling and may become a bottleneck. Therefore we decided to use the master as a dedicated component without an own solver process. Before we describe the concept of work cache management, we discuss the design of worker processes.

3.2 Multi-Threaded Worker versus Single-Threaded Worker

There are two main interactions of a worker with the master: asking for work and replying to split requests.

If a worker runs out of work, it sends a `work request` to the master and waits for a reply, that is, a new guiding path. This can be done using a straightforward blocking communication since there is nothing else to do for an idle worker.

How and when to respond to a `split request` is a more essential decision to make in terms of performance and load balancing. There are two possible approaches to this: the worker can interrupt its processing to handle an incoming request, or an additional dedicated thread handles requests.

In case of a multi-threaded approach, computation and communication could be done concurrently by different threads. The benefit is that the worker will have a good response time, since the communication thread would process a `split request` immediately. This minimises the waiting time for the master and improves the load balancing if another idle worker already waits for the guiding path to be returned. The drawback of such an approach is that threads need exclusive access to shared data structures. For example, the communication thread has to increment the root level, which interferes with the backjumping capability of the solver. As the additional locking would have to be used by both the computation and the communication thread, it would significantly influence the performance of the *clasp* solver. Thus, the multi-threaded approach was out of question for our parallelisation.

A single-threaded worker needs an entry point in the solver code where it is meaningful to test for a `split request`. From the perspective of search, the solver state resulting from conflict analysis is well-suited for making a reasonable decision on whether to split or not. We thus chose the end of the conflict analysis phase, performed to recover from a dead-end encountered in the search. Experiments will have to show the applicability and behaviour of this approach since the frequency of calls to conflict analysis is generally unpredictable (but usually high enough).

3.3 Work Cache Management

The introduction of a work cache (currently holding a single guiding path) solves two problems. It reduces the average answer time to a `work request`, and it avoids the need for a special treatment of the initialisation of all processes.

On start-up, the work cache is initialised with the empty guiding path, representing the entire search space, while all workers request work. One of the workers (first incoming request) will get the entry from the cache and starts processing the search space. The other work requests are appended to a FIFO queue.

The master maintains a list of busy workers. This list contains the number of a worker along with a priority reflecting its estimated workload. To this end, we take the root level of the worker, calculated from the length of its guiding path, as priority. The assumption is that a short guiding path results in a large search sub-tree to process. In the future, other heuristic methods are possible without changing the master, provided that they are based on integer values.

Whenever the work cache is empty, the master sends a `split request` to a busy worker with highest priority (i. e. a worker with smallest priority value). Reconsidering start-up, note that the work cache runs empty immediately when sending the empty guiding path, so that the receiving worker will be the target of a `split request`.

3.4 Implementation

We have implemented our approach in C++ using MPI. The resulting parallel ASP solver is called *claspar*.

Since the size of messages is not known in advance, the workers probe for a `split request` via `MPI_Probe` and `MPI_Iprobe`. The non-blocking `MPI_Iprobe` is used to check for messages after conflict analysis, since immediate return to the solver is required if no messages have arrived. Otherwise, the message size is determined by `MPI_Get_count`, and blocking receives are used to retrieve the message. A non-blocking receive is unnecessary and would rather increase the communication time, since it requires two calls to the MPI library.

Sending messages is always done using blocking `MPI_Send`. Non-blocking sends at a worker's side would require to interrupt the solver and check for completion of the send. In general, overlapping the sending of data would increase the risk of deferred data transmission, causing longer waiting times of the master and workers. Finally, *claspar* uses `MPI_Pack` and `MPI_Unpack` to create and interpret structured messages.

4 Experimental Results

We have conducted experiments to analyse the scalability of the master/worker approach. The examples were run on a Blue Gene/L from 16 up to 1024 cores in Virtual Node mode (two available cores per node are used for computation). Each core runs at 700 MHz and has access to 512 MB RAM. A large number of cores has to be employed in order to traverse a large search space in reasonable time. This requires the application to scale. Additionally, the search space must contain sufficient opportunities to split the

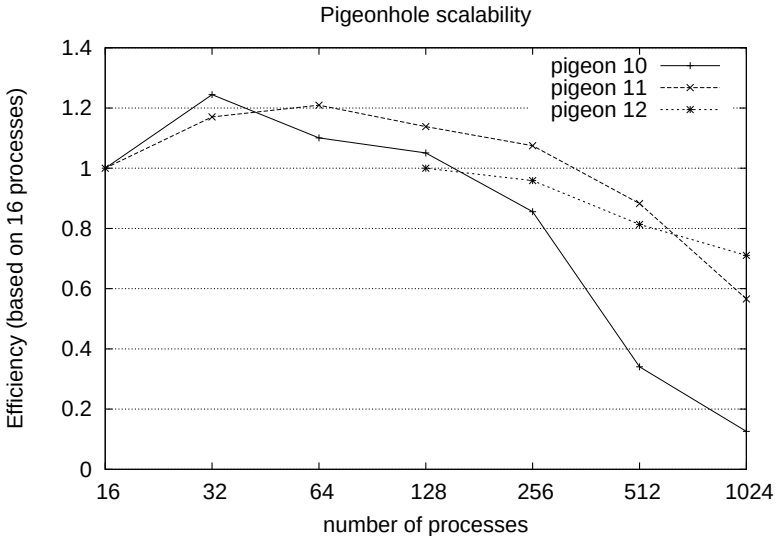

Fig. 1. Efficiency for pigeonhole benchmark.

work among the available workers. A few examples were also run on a Blue Gene/P using 2048 and 4096 cores in Virtual Node mode. The Blue Gene/P has four 850 MHz cores per node and 512 MB RAM per core.

It is common practise to stop a run after a certain time. The limit was set to 1200 seconds. Before termination, *claspar* prints out runtime statistics and, in particular, the number of models found. Both time and number of models can be taken to measure scalability.

The choice of examples covers simple and regular problems, such as *pigeonhole*, where the search space has to be traversed completely (since there is no way to put $N+1$ pigeons into N holes allowing only one pigeon per hole). The more sophisticated *blocked queens* benchmark is about putting N queens onto a $N \times N$ checker board such that they do not attack each other, where queens cannot be placed on particular blocked positions. With the *clumpy graphs* benchmark, Hamiltonian cycles are to be found in graphs of a two-layered structure, which gives rise to a non-uniform distribution of solutions in the search space.

4.1 Pigeonhole

Figure 1 shows the efficiency of the pigeonhole examples with 10, 11, and 12 pigeonholes (resp. 11, 12, 13 pigeons). The small example using 10 pigeonholes finishes within about 40–50 seconds with 16 processes. Sustained linear scaling is achieved up to 128 processes. Using 256 processes reduces the time to (no) solution to about 3 seconds. However, 512 processes introduce too much overhead on this small example such that the runtime increases on greater process counts. In contrast, 11 and 12 pigeonholes induce sufficient work to scale to 1024 processes, although efficiency drops

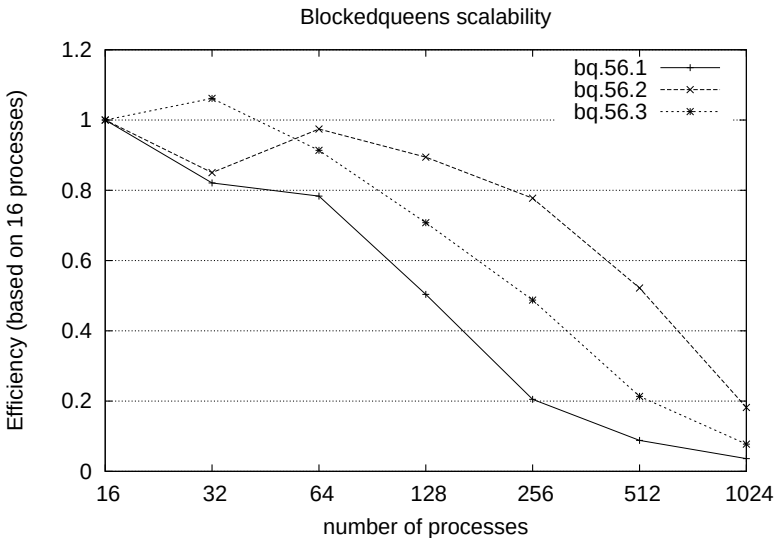

Fig. 2. Efficiency for blocked queens benchmark.

with 1024 processes. With 12 pigeonholes, the efficiency is based on the runtime with 128 processes because on smaller number of processes the search space could not be traversed within the time limit of 1200 seconds. The other calculations are based on the time measured with 16 processes.

4.2 Blocked Queens

The scalability of the blocked queens examples, shown in Figure 2, varies between instances. However, the available instances are yet too small for a large number of processes, so that the efficiency rapidly drops below 50 percent. Up to 64 or 128 processes, the examples still scale well. Thus it seems promising to go for larger instances of this class (if they were available).

4.3 Clumpy Graphs

Figure 3 shows the number of detected Hamiltonian cycles in three different graphs of size 9×9. The experiments were stopped after 1200 seconds. If an experiment has been aborted, the calculation of speedup and efficiency is based on the number of detected cycles. Since *claspar* is able to finish the second example within the time limit when running on more than 256 processes, the efficiency based on the number of detected cycles is omitted for 512 and 1024 processes in the figure. Using 512 processes, finishing the second example takes 1134 seconds, and 625 seconds with 1024 processes. This is a speedup of 1.81 and an efficiency of about 90 percent.

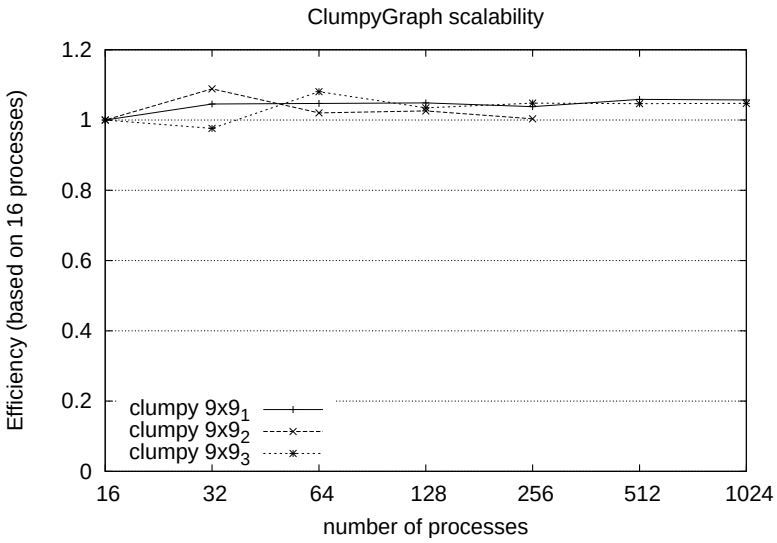

Fig. 3. Efficiency for clumpy graphs benchmark based on number of Hamiltonian cycles found.

4.4 Scalability

With the current examples, we could not find a limit for the number of messages the master can handle. The number of messages per second, shown in Table 1, is no indicator of the scalability of the tested environment.

The limiting factor for the above examples is the difficulty in terms of the number of conflicts (encountered dead-ends). The results indicate that the efficiency correlates to the ratio between messages and conflicts. Table 1 shows the ratio and the efficiency with 1024 processes based on the measurement on 16 cores. If the number of messages is more than two orders of magnitude smaller than the number of conflicts, efficiency is good. For one, this correlation is a result of the design of the workers handling incoming requests after conflict analysis. Therefore, the number of conflicts impacts the response time and the load balancing. For another, few messages per conflict indicate that workers perform a significant amount of search before running out of work.

Table 2 shows results for some examples on 2048 and 4096 cores of a Blue Gene/P. The example with 12 pigeonholes does not scale there. A look at the message/conflict ratio (2.86) reveals that it is too small for the larger numbers of processes. In fact, *claspar* completes even the example with 13 pigeonholes on 2048 or more cores within 1200 seconds, while fewer processes could not finish. Therefore, this example was omitted previously. On the clumpy graphs benchmark, the number of detected Hamiltonian cycles for example 09x09_01 and 09x09_03 doubles with the number of processes. This is a promising result for further examples and even larger numbers of processes.

The sometimes super-linear speedup is a phenomenon that was also observed in previous work [9]. A possible explanation is the reduced size of the search sub-trees to

example	msg/s	msg/conflicts*100	eff. (1024:16)
pigeon10	180387	7.35	0.126
pigeon11	84861	0.91	0.566
pigeon12	9302	0.08	0.71[3]
blockedqueens.56.1	136966	113.07	0.036
blockedqueens.56.2	129509	18.44	0.182
blockedqueens.56.3	134341	47.31	0.078
clumpy-09x09_01	18.5	0.0014	1.05
clumpy-09x09_02	1113	0.0925	(0.25)[4]
clumpy-09x09_03	18.7	0.00066	1.04

Table 1. Number of messages and conflicts for examples.

example	time (2048)	time (4096)	models (2048)	models (4096)
pigeon12	110.593	329.914	0	0
pigeon13	812.751	572.625	0	0
clumpy-09x09_01	1200.061	1200.193	9014355023	18014969357
clumpy-09x09_02	416.835	349.127	2134183512	2134183512
clumpy-09x09_03	1200.061	1200.193	7313232805	14325376947

Table 2. Results on Blue Gene/P.

be processed separately, whose fewer constraints imply less processing time. However, this effect strongly depends on the example and is subject to future investigation.

5 Conclusion and Future Work

We presented a performance analysis of our parallel ASP solver *claspar* on Blue Gene machines with up to 4096 cores. The current results are encouraging, even though they cover only a few problem classes and one cannot assume that *claspar* will behave similarly on every problem. The analyses indicate that *clasp* is well-suited for message-passing parallelisation. This gives us the vision that parallel ASP solving will be useful for even harder problems.

In the current *claspar* version, the communication topology is a star with the master as the centre. This was expected to have a limited scalability. Nevertheless, an efficient design of the master-worker interaction allows us to achieve a scalable behaviour as long as an instance induces a sufficient number of conflicts.

However, since it makes only limited sense to desperately try to achieve linear scalability while the problems and their search spaces usually grow exponentially, this parallel approach was intended to be kept simple. The workers also apply *learning*, i. e. constraints identified on conflicts are locally stored in memory. This advanced technique of the sequential *clasp* solver is planned to be applied to *claspar* (in a distributed manner) in the near future to further speed up the parallel search.

[3] based on 128 processes; see text

[4] all solutions found before timeout

As another future work, we aim at a hybrid version of *claspar* adapted also to multi-core architectures. Between physically distributed workers, learned constraints seeming important based on the *clasp* heuristics are subject to exchange via messages, while workers with shared memory use a shared region for the exchange (cf. [14]).

Acknowledgements. This work was supported by DFG under grant SCHA 550/8-1.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics **16**(1-2) (2006) 35–86
3. Mitchell, D.: A SAT solver primer. Bulletin of the European Association for Theoretical Computer Science **85** (2005) 112–133
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/MIT Press (2007) 386–392
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Volume 4483 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2007) 136–148
6. Pontelli, E., Balduccini, M., Bermudez, F.: Non-monotonic reasoning on Beowulf platforms. In Dahl, V., Wadler, P., eds.: Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03). Volume 2562 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2003) 37–57
7. 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
8. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
9. Gressmann, J., Janhunen, T., Mercer, R., Schaub, T., Thiele, S., Tichy, R.: On probing and multi-threading in platypus. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06), IOS Press (2006) 392–396
10. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The `nomore++` approach to answer set solving. In Sutcliffe, G., Voronkov, A., eds.: Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05). Volume 3835 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2005) 95–109
11. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing **29**(7) (2003) 969–994
12. Feldman, Y., Dershowitz, N., Hanna, Z.: Parallel multithreaded satisfiability solver: Design and implementation. Electronic Notes in Theoretical Computer Science **128**(3) (2005) 75–90
13. Schubert, T., Lewis, M., Becker, B.: Pamira - a parallel SAT solver with knowledge sharing. In Abadir, M., Wang, L., eds.: Proceedings of the Sixth International Workshop on Microprocessor Test and Verification (MTV'05), IEEE Computer Society (2005) 29–36
14. Lewis, M., Schubert, T., Becker, B.: Multithreaded SAT solving. In: Proceedings of the Twelfth Asia and South Pacific Design Automation Conference (ASP-DAC'07). (2007) 926–931
15. 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