Computing stable models in parallel

R.A. Finkel, V.W. Marek, N. Moore and **M. Truszczyński**
Department of Computer Science
University of Kentucky
Lexington KY 40506-0046, USA
email: raphael|marek|neil|mirek@cs.uky.edu

Abstract

Answer-set programming (ASP) solvers must handle difficult computational problems that are NP-hard. These solvers are in the worst case exponential and their scope of applicability, despite recent impressive gains in performance, remains limited. One way to deal with limitations of answer-set programming is to exploit parallelism. In this paper, we design and implement a parallel program, *parstab*, that computes stable models of logic programs. We describe preliminary experimental studies of *parstab*, running it on seven machines and comparing its performance to a serial execution. Our results are encouraging. For some problems, significant speedups are obtained by running *parstab* on multiple machines.

Introduction

Stable-model semantics (Gelfond & Lifschitz 1988) is one of the two most commonly accepted semantics of logic programs with negation, the other one being well-founded semantics (Van Gelder, Ross, & Schlipf 1991). Yet, despite its long presence on the logic-programming stage, researchers are only now beginning to appreciate the full potential of stable-model semantics. Two reasons for this recent interest are the development of *smodels* (Niemelä & Simons 1997; 2000), a fast implementation of an algorithm to compute stable models of logic programs without function symbols, and a better understanding of how to apply stable-model semantics in computation and the scope of its applicability (Marek & Truszczyński 1999; Niemelä 1999; Lifschitz 1999a; 1999b).

Work on declarative-programming formalisms based on stable-model semantics has led researchers to introduce *answer-set programming* (ASP), a computational paradigm in which theories (in some formal systems) serve as problem specifications, and different models of these theories determine different solutions (Marek & Truszczyński 1999; Niemelä 1999). Logic programming with stable-model semantics is one answer-set programming system. Researchers have recently proposed several other formalisms and corresponding implementations as well, including disjunctive logic programming and its implementation *dlv* (Eiter *et al.* 1997; 1998), and DATALOG with constraints and its solver *dcs* (East & Truszczyński 2000). Propositional logic with satisfiability checkers to compute models can also be regarded as an ASP formalism (Selman & Kautz 1992).

Computing stable models of logic programs and answer sets of theories of DATALOG with constraints is NP-hard (Marek & Truszczyński 1991; East & Truszczyński 2000); computing answer sets of disjunctive logic programs is Σ_P^2-hard. Despite impressive recent advances in the performance of *smodels*, *dlv*, *dcs*, and propositional satisfiability checkers, the problem of addressing the issue of computational complexity remains a challenge.

In this paper we study parallel computing environments to improve the performance of ASP formalisms. To this end, we design a parallel algorithm to compute stable models of logic programs and implement it in a program, called *parstab*. We have performed preliminary experimental studies of *parstab*, running it on seven machines and comparing its performance to a serial execution. We chose a hard combinatorial problem of computing Ramsey numbers and some standard benchmarks for the initial experiments. The results are highly encouraging. Significant speedups are obtained by running *parstab* on multiple machines. Although these results are preliminary, they provide strong evidence that parallelism, inherent in the search procedures underlying ASP implementations, can be exploited in order to expand the range of applicability of ASP. Related work on parallel implementations of algorithms for computing stable models is reported in (Pontelli & El-Khatib 2001).

The paper is organized as follows. The next section describes details of serial and parallel design of *parstab*. The following section describes our experimental results. The last section provides a brief discussion of these results and directions for future work.

Description of *parstab*

The program *parstab* is a parallel implementation of an algorithm to compute stable models of logic programs. It uses the PVM (Parallel Virtual Machine) package for inter-node communication (Geist *et al.* 1996). The program *parstab* is based on the *stable* algorithm, a part of the *lpsms* collection of programs developed at the University of Kentucky[1].

[1] *lpsms* is available from: `http://www.cs.uky.edu/ai/software/lpsms-current.tar.gz`.

The program *stable* is itself based on *smodels*[2] developed by Niemelä and his collaborators at the Helsinki University of Technology (Niemelä & Simons 1997; 2000).

Serial Algorithm

The *smodels* algorithm upon which we base *parstab* is described in detail in (Simons 1997). It is an earlier version of *smodels* which does not support cardinality and weight constraints available in the current implementation (Niemelä & Simons 2000). We outline here only those features of the algorithm important in our discussion of its parallel implementation.

The *smodels* algorithm uses backtracking search. It alternates between (a) adding to the current model atoms that must be true (or false) given the current knowledge, and (b) guessing atoms to be true or false. The first task, called 'expansion', uses what is basically well-founded semantics to expand the current partial model. The second task, called 'guessing', uses a heuristic we do not discuss here; we use a slight modification of the heuristic described in (Simons 1997). Expansion relies on full lookahead, so it can cause atoms to be added to the model positively or negatively, as it discovers that their negation produces a contradiction. The program *smodels* calls those atoms 'forced'.

We use a stack to assist in backtracking. It lists the atoms that have been added to the model along with an annotation indicating whether the atom is in the model positively or negatively, and whether it is known (expanded or forced) or guessed. As we alternately expand and guess, atoms are pushed onto the stack. When a contradiction or a stable model is reached, atoms are popped off until we reach an atom that was guessed. These atoms are known as 'choice points' and represent nodes in the search tree. Choice points are considered 'undecided' until their negation has been explored.

Backtracking proceeds to the most recent (closest on the stack) undecided choice point. That choice point is then negated and marked as 'decided'. A decided choice point is no longer treated specially by backtracking; all the models, if any, in the unprocessed subtree of the choice tree have been found. A decided choice point is therefore marked as known, making it indistinguishable from an expanded or forced atom. If backtracking reaches the top of the search tree (an empty stack) without encountering an undecided choice point, it has exhausted the search space. The algorithm then terminates.

Parallel Algorithm

The program *parstab* makes use of a master/slave parallel architecture. Its control structure is very similar to that of DIB (Finkel & Manber 1987). The first machine runs the master process. The master starts a user-specified number of slave processes, keeps track of models and logs them, and coordinates dynamic reallocation of work (see below). Initially, the slaves are assigned subtrees of the search tree by the master. Later, if they have no work, they request and obtain subtrees from other slaves. They then use the serial algorithm on these subtrees, sometimes breaking off part of their search space at the request of other slaves. In general, one slave is placed on each machine; the master, since it requires little CPU time, can share a machine with a slave.

PVM is a library for starting distributed algorithms and passing messages between components. Messages are sequences of strongly-typed elements, each of some fundamental data type (such as integer, character, string, and floating-point number). The program *parstab* uses a common format for messages, with eight subtypes. All messages begin with two integers; the first is the type of message, the second the PVM task ID of the machine generating the message. The remaining content of the message depends on the value of the message-type field. The master, since it does not perform a great deal of computation, spends nearly all its time waiting for messages. Slaves, on the other hand, only wait for messages when they have no work. They also accept messages at each choice point and once for every ten executions of the heuristic that do not encounter a choice point. Although the delay between the message reaching a slave and that slave processing could be arbitrarily large, in practice it is not too long.

Initialization

Initialization begins when the user starts the master, either from the command-line or from the PVM console. The master loads the logic program. It then starts slaves using the PVM routine `pvm_spawn()`. PVM then runs the requested number of copies of *parstab*. If the number requested is greater than the number of machines in the PVM configuration, some machines run multiple slaves. The program *parstab* passes to each slave command-line parameters indicating the master's version number as well as logging and timing flags. The version number is used as a crude test for compatibility; if the version numbers do not match, the slaves assume that they are not using a protocol compatible with that of the master and exit.

Once the slaves have been started, the master uses PVM to send each an initialization message. This message contains all the information the slaves need to begin working. It contains the logic program (with atoms replaced by numbers), the stratification, if any, certain flags, the number of slaves, and the serial number (label) of each slave (an integer between zero and one less than the number of slaves; this number is different from the PVM task ID, and is used by the slave to locate its initial subtree of the search space). The flags are mostly tuning parameters for the guessing heuristic.

Initial Choice Points

The slaves all begin at the top of the search tree with an empty stack. If they proceeded to use the *smodels* algorithm without modification, they would each make exactly the same guesses and would uselessly perform exactly the same computation on many different machines. Instead, each machine has its first few guesses determined beforehand. There are enough such initially determined choices to

[2] Available from: `http://www.tcs.hut.fi/Software/smodels/`.

ensure that the slaves reach disjoint subtrees of the search space.

The initial choices are based on the binary pattern of the slave's label, with lower-order bits representing earlier choice points. For example, slave 11 of 16 (binary 1011) chooses true at the first two choice points, false at the third, and true at the fourth. If the number of slaves is not a power of two, some slaves have more initially determined choice points than others. For example, if there are 6 = 4 + 2 slaves, slaves 0, 1, 4, and 5 each has three such choice points; slaves 2 and 3 have two.

A slave with a predetermined choice point uses the heuristic as usual to determine which atom to "guess". It then pushes that atom as a decided choice point, true or false, depending on the low bit of its label. Finally, it shifts its label right by one bit (so 11 becomes 5, for instance). Since initial choice points are marked as decided, backtracking does not try their negation, so a slave never backtracks into another slave's portion of the search space.

The choice heuristic must be deterministic to prevent slaves from disagreeing on the initial subdivision of the search tree. Given determinism, the initial subdivision is in fact a complete partition of the search tree.

Dynamic reallocation of work

Since the search tree is not uniform, some slaves finish their subtrees before others. We do not want slaves to be idle for potentially long periods when they could be doing useful work. Therefore we employ a method of dynamically redistributing the work, much as DIB does (Finkel & Manber 1987).

When a slave A has exhausted its search tree, it reports this fact to its master. The master then sends a message to a randomly-picked slave B that is still computing. Slave B, upon receiving this message from the master, breaks off a branch from its search tree at atom X, the first (deepest on the stack) undecided choice point. It marks that choice point as decided. Slave B sends slave A its stack up to and including atom X, with atom X negated (and decided). Thus slave A receives the largest untouched subtree of slave B's search tree. Finally, slave A reports to the master that it has received work.

Slave A always receives a stack with no undecided choice points. Once it completes the tree represented by that stack, it never backtracks into another slave's portion of the search space. Likewise, slave B marks the branched choice point as decided; it never backtracks into the portion of the search space it has just assigned to slave A. This dynamic reallocation therefore ensures that, once the initial choice points (see above) are exhausted, no two slaves are ever in the same portion of the search space. Likewise, it preserves the covering of the search space, so it never loses models (unless, of course, a machine dies[3]).

[3]DIB handles the case of machine failure by a mechanism we could adopt.

Models

The purpose of running *parstab* is to report stable models to the user or to show that they do not exist. PVM slaves cannot communicate directly with the user; only the master can. As a slave finds a model, it sends it as a PVM message to the master, which then formats the model and presents it to the user.

Unfortunately, before the initial choices have been exhausted, two or more slaves may be in the same portion of the search tree. If there is a model that high in the search tree, multiple slaves will find it. We use a filtering method so that only one such slave reports such a model: the one with the lowest label, that is, the one with only 0 bits remaining in its shifted label.

Termination

In order to dynamically reallocate work, slaves report to the master when they have completed their assigned subtrees. The running slaves always cover the remaining portion of the search space. If all slaves have completed their subtrees, no unprocessed portion of the search space remains; all the stable models have been found, and computation is finished. At this point, the master sends a message to all slaves asking that they terminate. After three seconds, the master uses PVM to terminate any remaining slaves that did not terminate for some reason.

The user may specify bound N on the number of models to search for. Once the master receives the Nth model, it initiates termination in the same way. In this case, it is more often necessary to use PVM to terminate remaining slaves; some might be stuck in computation for a long time before they check messages.

The master may die for some unexpected reason, such as power failure or the user killing the process. To take into account this possibility, the slaves request that PVM notify them on the death of the master. This notification is treated as a termination request from the master, except for logging details. In this case, some slaves may continue running for a relatively long period of time, until they next check for messages.

Results

We tested *parstab* using a cluster of 7 identical Sun Sparcstation 20 125MHz workstations connected by a 100 Mbps ethernet. We compared run times on the cluster with those of a sequential algorithm *stable* on a single workstation. Times reported here include initialization but not the three seconds spent waiting for slaves to die.

For the tests we have selected several difficult search problems commonly used as benchmarks for ASP programs:

1. computing bounds on Ramsey numbers
2. n-queens problems
3. pigeonhole problem with n holes and $n+1$ pigeons

Ramsey numbers. The *Ramsey number* $R(k, m)$ is defined as the least integer n such that in every coloring of the complete graph with n vertices such that each edge is either red or blue, there is a complete subgraph with k vertices with all edges red or a complete subgraph with m vertices and with all edges blue. Even for relatively small values of k and m, the precise value of $R(k,m)$ is not known. For instance, if both k and m are at least 4, only two values are known: $R(4,4) = 18$ and $R(4,5) = 25$ (Graham, Rothschild, & Spencer 1980; McKay & Radziszowski 1995). To show that $R(k,m) > n$, one needs to find a coloring of a complete graph with n vertices in which neither a red copy of a complete graph on k vertices nor a blue copy of a complete graph on m vertices exists. To this end, in the case of $k = 4$ and $m = 5$, we use the following program. Its stable models define colorings that have no monochromatic complete graphs of the requisite size. Thus, if a stable model is found, $R(4,5) > n$.

```
vtx(1..n).

edge(X,Y) :- vtx(X), vtx(Y), X < Y.

blue(X,Y) :- edge(X,Y), not red(X,Y).
red(X,Y) :- edge(X,Y), not blue(X,Y).

:- edge(W,X),edge(W,Y),edge(W,Z),
   edge(X,Y),edge(X,Z),edge(Y,Z),
   blue(W,X),blue(W,Y),blue(W,Z),
   blue(X,Y),blue(Y,Z),blue(X,Z).

:- edge(V,W),edge(V,X),edge(V,Y),
   edge(V,Z),edge(W,X),edge(W,Y),
   edge(W,Z),edge(X,Y),edge(X,Z),
   edge(Y,Z),
   red(V,W),red(V,X),red(V,Y),
   red(V,Z),red(W,X),red(W,Y),
   red(W,Z),red(X,Y),red(X,Z),
   red(Y,Z).
```

We chose this problem because it is very hard, and as n increases, the constraints become tighter: The number of stable models declines and eventually becomes equal to 0. When there are no stable models, the corresponding value of n is the Ramsey number. We ran this problem looking for only a single solution, if any.

n-queens problem. The goal is to find an arrangement of queens on an $n \times n$ chess board such that no two queens attack each other. This problem is often used as a benchmark. Its key feature is that there are many solutions, increasing rapidly with n. We used the following program for our tests:

```
row(1..n).
col(1..n).

queen(X,Y)  :- row(X), col(Y),
               not otherqueen(X,Y).

otherqueen(X,Y) :- row(X), col(Y;Z),
                   queen(X,Z), Y!=Z.

:- row(X;Z), col(Y),
   queen(X,Y), queen(Z,Y), X!=Z.

:- row(U;X), col(V;Y),
   queen(X,Y), queen(U,V),
   (X-U)==(Y-V), X!=U.

:- row(U;X), col(V;Y),
   queen(X,Y), queen(U,V),
   (X-U)==(V-Y), X!=U.
```

In our experiments, we ran this program to find all stable models so as to exhaust the whole search space.

Pigeonhole problem. We considered the case of n holes and $n+1$ pigeons, a situation in which the problem has no solution. To report failure, the program has to search through the entire search space. We used the following program in our tests:

```
pigeon(1..n+1).
hole(1..n).

in_hole(P,H) :- pigeon(P), hole(H),
                not otherhole(P,H).
otherhole(P,H) :- pigeon(P), hole(H;X),
                  in_hole(P,X), H != X.

:- pigeon(P;Q), hole(H),
   in_hole(P,H), in_hole(Q,H), P != Q.
```

The results of our experiments are shown in Tables 1 – 3. The size of the problem is specified in the first column. The column *stable* gives times obtained by running the program *stable* — a sequential version of *parstab*. The column *parstab* gives times obtained by running *parstab* on seven workstations. The last column gives the speedup.

n	*stable*	*parstab*	*speed up*
18	>30m	30.5s	>60
19	>12h	44.5s	>60
20	>12h	71.5s	>60
21	>12h	101s	>60
22	>12h	141s	>60
23	>12h	>12h	?

Table 1: Computing Ramsey number $R(4,5)$

n	*stable*	*parstab*	*speed up*
8	10.7s	5.75s	1.86
9	59.7s	22.8s	2.62
10	282.6s	189.0s	1.49
11	1113s	445.1s	2.50
12	>30m	>30m	?

Table 2: n-queens problem

The results show impressive speedups in the case of Ramsey numbers, where we stop as soon as the first solution is found. The very large speedups indicate that the guessing heuristic of stable (or *smodels*) is not well tuned to this problem; the first solution is not found on the principal path (first

n	*stable*	*parstab*	*speed up*
6	4.38s	1.17s	3.74
7	36.6s	7.02s	5.21
8	317.2s	52.15s	6.08
9	>30m	>30m	?

Table 3: Pigeonhole problem

path in sequential depth-first search). The program *parstab* searches in parallel in several places of the search tree in same time. In some of them, a few decided atoms quickly led to a solution.

In the other two examples, *parstab* needed to traverse the whole search space. The speedups for the n-queens problem are much lower than those obtained for the pigeonhole problem. This difference is almost surely due to the large number of results that are reported, a procedure in which the master is a serial bottleneck.

The timing results for the pigeonhole problem are very promising, indicating that, if there are have few models and not too many machines, *parstab* gets fairly close to a perfect speedup.

Conclusions and future directions

This is a preliminary report. We are still conducting experiments. They involve computing on larger clusters of workstations (up to 25) and more detailed gathering of timing data. In particular, we collect compute, wait and communication times to understand the impact of each of these three components on the overall performance.

We are also developing a similar parallel implementation of the most recent version of *smodels* with its cardinality, and weight constraints.

References

East, D., and Truszczyński, M. 2000. Datalog with constraints. In *Proccedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, 163–168.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1997. A deductive system for non-monotonic reasoning. In *Logic programming and nonmonotonic reasoning (Dagstuhl, Germany, 1997)*, volume 1265 of *Lecture Notes in Computer Science*, 364–375. Springer.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. A KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceeding of the Sixth International Conference on Knowledge Representation and Reasoning (KR '98)*, 406–417. Morgan Kaufmann.

Finkel, R., and Manber, U. 1987. DIB — a distributed implementation of backtracking. *ACM TOPLAS* 9:235–256.

Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R.; and Sunderam, V. 1996. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, MA.

Gelfond, M., and Lifschitz, V. 1988. The stable semantics for logic programs. In Kowalski, R., and Bowen, K., eds., *Proceedings of the 5th International Conference on Logic Programming*, 1070–1080. Cambridge, MA: MIT Press.

Graham, R.; Rothschild, B.; and Spencer, J. 1980. *Ramsey Theory*. Wiley.

Lifschitz, V. 1999a. Action languages, answer sets, and planning. In Apt, K.; Marek, W.; Truszczyński, M.; and Warren, D., eds., *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 357–373.

Lifschitz, V. 1999b. Answer set planning. In *Logic programming, Proceedings of the 1999 International Conference on Logic Programming*, 23–37. MIT Press.

Marek, W., and Truszczyński, M. 1991. Autoepistemic logic. *Journal of the ACM* 38(3):588–619.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In Apt, K.; Marek, W.; Truszczyński, M.; and Warren, D., eds., *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 375–398.

McKay, B. D., and Radziszowski, S. P. 1995. $r(4, 5) = 25$. *J. Graph Theory* 19:309–322.

Niemelä, I., and Simons, P. 1997. Smodels — an implementation of the stable model and well-foundd semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning (the 4th International Conference, Dagstuhl, Germany, 1997)*, volume 1265 of *Lecture Notes in Computer Science*, 420–429. Springer-Verlag.

Niemelä, I., and Simons, P. 2000. Extending the smodels system with cardinality and weight constraints. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3-4):241–273.

Pontelli, E., and El-Khatib, O. 2001. Experiments in parallel execution of answer set programs. In working notes of the AAAI Spring Symposium on Answer Set Programming, 2001.

Selman, B., and Kautz, H. A. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*.

Simons, P. 1997. Towards constraint satisfaction through logic programs and the stable model semantics. Technical Report 47, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland.

Van Gelder, A.; Ross, K.; and Schlipf, J. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3):620–650.