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 algorithm, *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

The stable model semantics (Gelfond & Lifschitz 1988) is one of the two most commonly accepted semantics of logic programs with negation, the other one being the well-founded semantics (Van Gelder, Ross, & Schlipf 1991). Yet, despite its long presence on the logic programming stage, the full potential of stable model semantics is only now becoming to be fully appreciated. The two factors 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 the stable model semantics in computation and what is the scope of its applicability (Marek & Truszczyński 1999; Niemelä 1999; **?**; **?**; Syrjänen 1999).

Work on declarative programming formalism based on the stable model semantics 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 an answer-set programming system but several other formalisms and corresponding implementations were proposed recently. They include disjunctive logic programming and its implementation *dlv* (Eiter *et al.* 1997; 1998), and DATALOG with constraints and its solver *dcs* (East & Truszczyński 2000a; East & Truszczyński 2000b). Propositional logic with satisfiability checkers to compute models can also be regarded as an ASP formalism.

The general problems these computational tools are solving are complex. Computing stable models of logic programs and answer sets of theories of DATALOG with constraints is NP-hard (Marek & Truszczyński 1991; East & Truszczyński 2000b), computing answer sets of disjunctive logic programs is Σ^2_P-hard. Despite impressive recent advances progress 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 the use of parallel computing environments to improve the performance of of ASP formalisms. To this end, we design and implement a parallel algorithm, *parstab*, that computes stable models of logic programs. We performed preliminary experimental studies of *parstab*, running it on seven machines and comparing its performance to a serial execution. We chose hard combinatorial problems for the initial experiments. The results are highly encouraging. Significant speedups are obtained by running *parstab* on multiple machines. These results are preliminary. Still, they provide strong evidence that parallelism, inherent in search procedures underlying ASP implementations, can be exploited in order to expand the range of applicability of ASP.

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*

parstab is a parallel implementation of an algorithm to compute stable models of logic programs. It uses the PVM (Parallel Virtual Machine) system for internode communication (Geist *et al.* 1996), see also `http://www.epm.ornl.gov/pvm/pvm\_home.html`. The basic algorithm whose parallel version was developed and studied in this work is *stable*, a part of the *lpsms*

collection of programs developed at the University of Kentucky[1]. *stable* is itself based on the *smodels* program[2] developed by Ilkka Niemelä at the Helsinki University of Technology (Niemelä & Simons 1997; 2000).

parstab must be compiled separately for each machine type (processor/OS combination) it is to be run on. It should then be placed in the PVM binary directory on each machine appropriate for that architecture. This is necessary so that PVM can spawn copies of *parstab* on machines other than the master (see below).

Serial Algorithm

The algorithm used by *parstab* is a modification of that used by *stable* which is a slight modifications of the algorithm used by *smodels* version WHAT. This algorithm is described in more detail in (Simons 1997). We outline here only those features of the algorithm important in our discussion of its parallel implementation.

The *stable* algorithm uses backtracking. The algorithm alternates between adding to the current model atoms which must be true (or false) given the current knowledge, and guessing atoms to be true or false. The first task is called 'expansion'. It uses what is basically the well-founded semantics to expand the current partial model. The second task is called 'guessing'. We do not discuss here the heuristics used to determine which atoms to guess; we use a slight modification of the heuristic described in (Simons 1997). One thing to note about expansion is that it relies on full lookahead and can cause certain atoms to be added to the model positively or negatively, as we discover that their negation produces a contradiction. *smodels* refers to those as 'forced'. We keep that terminology here.

In implementing the algorithm, we make use of two stacks, which are operated on in concert. The first lists the atoms which have been added to the model; the second, stores annotations indicating whether the corresponding 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 which was guessed. These atoms are known as 'choice points', and represent nodes in the search tree. Choice points are initially 'undecided' as their negation has not been tried.

As stated above, backtracking proceeds to the most recent (highest on the stack) undecided choice point. That choice point is then negated and marked as 'decided'. A decided choice point will no longer be treated specially when backtracking; all the models, if any, in the uncomputed subtree of the choice tree have been found. Thus, a decided choice point is marked as known — making it indistinguishable from an expanded or forced atom. If we backtrack to the top of the search tree (an empty stack) without encountering an undecided choice point, we know that we have exhausted the search space. The algorithm then terminates.

Parallel Algorithm

parstab makes use of a master/slave parallel architecture. The first node, generally the one executed by the user, is the master. The master spawns a user-specified number of children, keeps track of models and logs them, assigns work to the children, and coordinates branching (see below). Initially, the children are assigned subtrees of the search tree by the master. Later in the execution, they obtain subtrees from other children. They then use the serial algorithm on these subtrees, sometimes breaking off part of their search space at the request of other children. In general, there will be one child node on each machine; the master, since it requires so little CPU time, can share a node with a child.

PVM is a message-passing library; messages are sequences of strongly-typed elements, each of some fundamental data type (integer, long integer, character, string, floating-point number, etc.). *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. Children, on the other hand, only wait for messages when they have no work, once at each choice point, and once for every ten executions of the heuristic which do not encounter a choice point. Thus, though the delay between the message reaching a child and that child processing it is in theory arbitrarily large, it is in practice not too long.

Initialization

Initialization begins when the user runs the master, either from the command-line or from the PVM console. The master loads the logic program (and optionally a stratification; see the documentation of *stable* for more information on this). It then spawns children 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 will run multiple children. *parstab*, when spawning children, passes them the version number of the master, as well as logging and timing flags. The flags must be sent through the command line so that the children know whether to log the initialization messages. The version number is used as a crude test for compatibility; if the version numbers do not match, the children assume that they are not using a protocol compatible with that of the master, and exit.

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

[2] The program is available from: `http://saturn.tcs.hut.fi/pub/smodels/`.

Once the children have been started, the master uses PVM to send each an initialization message. This message contains all the information the children need to begin working. It contains the program (with atom names stripped; all user interaction is done by the master so slaves can refer to atoms by number alone), the stratification if any is used, certain flags, the total number of children, and the number (label) of each child (an integer between zero and one less than the number of children; this number is different from the PVM task ID, and is used by the child to locate its initial subtree of the search space). The flags are in general tuning parameters for the heuristic, and are beyond the scope of this document.

Initial Choice Points

The children 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 we 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 ensure that the children reach disjoint subtrees of the search space.

The initial choices are based on the binary pattern of the machine's child number (the one transmitted in the initialization message), with lower-order bits representing earlier choice points. For example, machine 11 of 16 (binary 1011) would choose true at the first two choice points, false at the third, and true at the fourth. If the number of machines is not a power of two, some machines will have more initially determined choice points than the others. For example, if there are 6 (4 + 2) machines, machines 0, 1, 4, and 5 will have three such choice points, while machines 2 and 3 will have two.

While a child still has predetermined choice points left, it uses the heuristic as normal to determine which atom will be the choice point. It then pushes that atom as a decided choice point, true or false, depending on the low bit of its child number. Finally, it shifts its child number right by one bit (so 11 becomes 5, etc.). Since initial choice points are marked as decided, the algorithm will not try their negation when backtracking. Thus a child will not backtrack into another child's portion of the search space.

Note that, for the above to not omit parts of the search tree, running the heuristics twice with the same stack must give the same result. This way, machines 3 and 11, for example, will always choose the same atom at the fourth choice point (though one will mark it true and the other false). This requires that the heuristics be deterministic — not a problem currently, as the *smodels* heuristics is so. Assuming this determinism, the initial choices partition the remaining search space (that below the initial choice points). Each machine receives a subtree, these trees are disjoint, and they, taken with the common portions of the tree (where not all of the initial choices have been made yet), cover the entire search space.

Branching

Since the search tree is not uniform, it is quite likely that some children should finish their subtrees before others. We do not want them to be idle for potentially long periods of time while they could be doing useful work. Therefore we employ a method of branching the computation to other machines.

When a child has exhausted its search tree, it reports this fact to its master. The master then selects at random a child which is still computing, and sends a message to that child. For the sake of illustration, let us call the child that wants work A, and the child selected by the master B. Child B, upon receiving this message from the master, breaks off its search tree at the first (lowest on the stack) undecided choice point (atom X, say). That choice point is then marked as decided. Child B sends to child A the stack up and including atom X, with atom X negated (and still marked as decided). Thus child A receives the largest untouched subtree of child B's search tree. Finally, child A reports to the master that it has received work.

Using this method, children always get stacks with no undecided choice points. Thus, once they complete the tree they are assigned, they will not backtrack into another child's portion of the search space. Likewise, the child that is giving up work marks the branched choice point as decided; thus it will not backtrack into the portion of the search space it has just assigned to another machine. We therefore ensure that, once the initial choice points (see above) are exhausted, no two children will ever be in the same portion of the search space. Likewise, we preserve the covering of the non-computed search space, so we can be sure of not losing models (unless, of course, a machine dies).

Models

The purpose of running *parstab* is, of course, to find stable models (or to show that they do not exist). We therefore need some way of reporting stable models to the user. The children cannot communicate directly with the user; they may be running on any number of machines, which the user may not have easy non-PVM access to. Thus the master must be the one to report all models. Children therefore send models to the master through PVM messages — one per model. Since PVM messages may be arbitrarily large, and are broken up into transmittable chunks by lower-layer protocols, this does not pose a communication problem.

There is only one complication with regard to models. Recall that, before the initial choices have been exhausted, two or more different children may be in the same portion of the search tree. If there is a model so high in the search tree, multiple children will find it. This is a problem, because we want the master to report each model only once. There is a simple solution, though. Children which are in a common part of the search tree must have made the same choices

so far; hence they share the lowest, already shifted-off, bits of their child numbers. Since children have unique child numbers, this means they differ in the remaining bits. Thus we can have only the lowest-numbered child which would come across the model report it. Since child numbers are assigned sequentially, this would be the child with 0 for all the unshifted bits. Hence, while there are still initial choices to be made, only children with 0 for the unshifted portion of their child numbers will report stable models.

Ending Computation

Recall that, for the purposes of branching, children report to the master when they have completed work. The master uses these messages for another purpose. As noted above, the running children always cover the remaining portion of the search space. Thus, if all children have completed their search trees, there must be no uncomputed portion of the search space remaining; thus all the stable models have been found, and computation is finished. When this happens, the master sends a message to all children asking that they terminate. Then, after three seconds, the master has PVM kill any remaining children which did not terminate for some reason.

It may also be the case that the user specified a fixed number N of models to search for. Once the master receives the Nth model, it ends computation in the same way. In this case, the second step, of having PVM kill any remaining children, is necessary more often; those children can be stuck in computation for a potentially long time before they check messages.

Finally, the master may die for some reason: a power failure, the user killing the process, etc. To take into account this possibility, the children use the `pvm_notify()` routine to have PVM notify them on the death of the parent. This notification is treated just like any other message; when the child receives it, it behaves exactly as though the parent had asked it to die (except that it logs a different message). In this case, the master cannot ask PVM to kill its children; thus children 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 Sun workstations. We compared run times on the cluster with those of a sequential algorithm on a single Sun machine. Times reported here include initialization but not the 3 seconds that are spent waiting for slaves to die.

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

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 so that each edge is either read or blue, there is a complete subgraph with k vertices and with all edges red or a complete subgraph with m vertices with all edges blue. Even for relatively small values of k and m 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 $R(4,5) = 25$. To show that $R(k,m) > n$ one needs to find a coloring 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 without required monochromatic complete graphs. 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).
```

This problem was chosen as it is very hard and, as n increases the constraints become tighter, the number of stable models goes down and eventually becomes equal to 0 (when this first happens, the corresponding value of n is the Ramsey number).

n-queens problem. In this problem the goal is to find an arrangement of queens on an $n \times n$ chess board so that no two queens attack each other. This problem is often used as a benchmark. Its key feature is that there are very many solutions. 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 in which we wanted to study the effect of parallelism, we ran this program to find all stable models so that to exhaust the whole search space.

Pigeonhole problem. We considered the case of n holes and $n+1$ pigeons. Clearly the problem has no solution and to discover that 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 - 2. 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 Sun 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

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

The results show impressive speedups in the case of Ramsey numbers. They seem to indicate that the heuristics of stable (or *smodels* is not well tuned to the problem of computing Ramsey numbers. Running *parstab* we are searching in parallel in several places of the search tree in same time. In some of them, few decided atoms allow us to identify a solution quickly.

In two other examples we ran our algorithms so that they had to search through 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 a result of having to report many models present in the case of the former problem. The results for the pigeonhole problem are very promising—they indicate that, if we have few models and not too many machines, we get asymptotically close to a linear speedup.

Conclusions and future directions

This is a preliminary report. We are still conducting experiments. They will be presented and discussed in the full version of this paper. We will also experiment with larger clusters of workstations.

In the future, we intend to develop parallel implementations of the most recent version of *smodels* including its choice, cardinality, and weight constraints. We will also develop parallel implementations of algorithms for ASP formalisms based on the Message Passing Interface (MPI) approach.

References

East, D., and Truszczyński, M. 2000a. Datalog with constraints. Unpublished manuscript.

East, D., and Truszczyński, M. 2000b. 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 nonmonotonic 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 Repre-*

sentation and Reasoning (KR '98), 406–417. Morgan Kaufmann.

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.

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.

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.

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.

Syrjänen, T. 1999. A rule-based formal model for software configuration. Technical Report A55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 1999.

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.