Another implementation of the p-stable semantics, a parallel aproach

David López[1], Gabriel López[1], Mauricio Osorio[2], and Claudia Zepeda[2]

[1] Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación
[2] Universidad de las Américas - Puebla,
CENTIA

(Paper received on November 28, 2010, accepted on January 28, 2011)

Abstract. In this paper we review some theoretical results about the p-stable semantics, and based on that, we design some algorithms that search for the p-stable models of a normal program. An important point is that we propose algorithms that can also be used to compute p-stable models in a parallel approach. **Keywords** non-monotonic reasoning, p-stable, parallel.

1 Introduction

Currently, is a promising approach to model features of commonsense reasoning. In order to formalize NMR the research community has applied monotonic logics. In [3], Gelfond and Lifschitz defined the stable model semantics by means of an easy transformation. The stable semantics has been successfully used in the modeling of non-monotonic reasoning (NMR).

Additionally, Pearce presented a characterization of the stable model semantics in terms of a collection of logics in [12]. He proved that a formula is "entailed by a disjunctive program in the stable model semantics if and only if it belongs to every intuitionistically complete and consistent extension of the program formed by adding only negated atoms". He also showed that in place of intuitionistic logic, any proper intermediate logic can be used. The construction used by Pearce is called a weak completion.

In [9], a new semantics for normal programs based on weak completions is defined with a three valued logic called G'_3 logic. The authors call it the P-stable semantics. In [7], the authors define the *p-stable semantics* for disjunctive programs by means of a transformation similar to the one used by Gelfond and Lifschitz in their definition of the stable semantics. The authors also prove that the p-stable semantics for disjunctive programs can be characterized by means of a concept called weak completions and the G'_3 logic, with the same two conditions used by Pearce to characterize the stable semantics of disjunctive programs, that is to say, for normal programs it coincides with the semantics defined in [9].

In fact, a family of paraconsistent logics studied in [7] can be used in this characterization of the p-stable semantics.

In [8], the authors offer an axiomatization of the G'_3 logic along with a soundness and completeness theorem, i.e., every theorem is a tautology and vice-versa.

Advances in Computer Science and Applications
Research in Computing Science 53, 2011, pp. 221-228

We also remark that the authors of [7] present some results that give conditions under which the concepts of stable and p-stable models agree. They present a translation of a disjunctive program D into a normal program N, such that the p-stable model semantics of N corresponds to the stable semantics of D when restricted to the common language of the theories. Besides, they show that if the size of the program D is n then the size of the program N is bounded by An^2 for a constant A. The relevance of this last result is that it shows that the p-stable model semantics for normal programs is powerful enough to express any problem that can be expressed with the stable model semantics for disjunctive programs.

It is important to mention that the p-stable semantics, which can be defined in terms of paraconsistent logics, shares several properties with the stable semantics, but is closer to classical logic. For example, the following program $P = \{a \leftarrow \neg b, a \leftarrow b, b \leftarrow a\}$ does not have stable models. However, the set $\{a, b\}$ could be considered the intended model for P in classical logic. In fact, it is the only p-stable model of P.

In [11], a schema for the implementation of the p-stable semantic using two well known open source tools: Lparse and Minisat is described. In [11], a prototype[3] written in Java of a tool based on that schema is also presented. In [6] the author presents an improved implementation [4] using optimized code and algorithms, resulting in an error-free tool, getting at least the same afficienty that [11]. In [5] there is another implementation where the author manage some work through the use of a high performance and efficiency software Suite call Potassco[5], coded in java without a final version.

Currently one processor computers had reached enormous speeds and they have pushed hardware to its physical limits. This trend has gradually been displaced given the physical limits in design delimiting the computational power we can get with a single processor. Multiple architectures are known that use parallel through the interaction of multiple processing units.

One of these takes place in the form of various processors working together internally which was an initial progress. With advances in computer networks, new protocols and speeds reached by them, it has been created new methods for developing parallelism using small autonomous networks functioning as parallel systems. To achieve the greatest benefits of parallelism, both designers and programmers must understand the virtues and disadvantages associated with the development on multicore systems.

Despite the large advances observed in sequential Answer Set Solving technology as seen in the beginning, only a few implementations have been developed with a parallel approach[13][1][2]. Its alarming, seeing how easily we can have access to devices such as clusters, multiprocessor and/or multicore computers. It is worth mentioning that the development of these parallel systems has been made following the standards of stable semantics, which we know its extent.

[3] http://cxjepa.googlepages.com/home
[4] http://sites.google.com/site/computingpstablesemantics/downloads
[5] http://potassco.sourceforge.net/

This is the reason for us to implement a version that can parallelize the case of p-stable.

Our paper is structured as follows. In section 2, we summarize some definitions, logics and semantics necessary to In section 3, we show how to find the the p-stable models of a program P, in section 4 we desribe the current implementation workflow, in section 5 we introduce the parallel algorithms proposed. Finally, in section 6, we present some conclusions and further work.

2 Background

In this section we summarize some basic concepts and definitions necessary to understand this paper.

2.1 Syntax

A signature $\mathcal{L}$ is a finite set of elements that we call atoms. A *literal* is either an atom a, called *positive literal*, or the negation of an atom $\neg a$, called *negative literal*. Given a set of atoms $\{a_1, ..., a_n\}$, we write $\neg\{a_1, ..., a_n\}$ to denote the set of atoms $\{\neg a_1, ..., \neg a_n\}$. A *normal* clause or *normal* rule, r, is a clause of the form

$$a \leftarrow b_1, \ldots, b_n, \neg b_{n+1}, \ldots, \neg b_{n+m}.$$

where a and each of the b_i are atoms for $1 \leq i \leq n+m$, and the commas mean logical conjunction. In a slight abuse of notation we will denote such a clause by the formula $a \leftarrow B^+(r) \cup \neg B^-(r)$ where the set $\{b_1, \ldots, b_n\}$ will be denoted by $B^+(r)$, the set $\{b_{n+1}, \ldots, b_{n+m}\}$ will be denoted by $B^-(r)$, and $B^+(r) \cup B^-(r)$ denoted by $B(r)$. We use $H(r)$ to denote a, called the head of r. We define a *normal program* P, as a finite set of normal clauses. If for a normal clause r, $B(r) = \emptyset$, $H(r)$ is known as a *fact* . We write $\mathcal{L}_P$, to denote the set of atoms that appear in the clauses of P.

2.2 Semantics

From now on, we assume that the reader is familiar with the single notion of *model*[4]. In order to illustrate this basic notion, let P be the normal program $\{a \leftarrow \neg b.,\ \ b \leftarrow \neg a.,\ \ a \leftarrow \neg c.,\ \ c \leftarrow \neg a.\}$. As we can see, P has five models: $\{a\}$, $\{b, c\}$, $\{a, c\}$, $\{a, b\}$, $\{a, b, c\}$.

Now we give the definition of p-stable model semantics for normal programs.

Definition 1. *[10] Let P be a normal program and M be a set of atoms. We define the reduction of P with respect to M as $\boldsymbol{RED}(P, M) = \{a \leftarrow B^+ \cup \neg(B^- \cap M) | a \leftarrow B^+ \cup \neg B^- \in P\}$.*

Definition 2. *[10] A set of atoms M is a p-stable model of a normal program P iff $\boldsymbol{RED}(P, M) \models M$, where the symbol $\models$ means logical consequence under classical logic semantics. The set of p-stable models of P is denoted by $PS(P)$.*

We say that two normal programs P and P' are equivalent if and only if they have the same set of p-stable models, this relation is denoted by $P \equiv P'$.

The following definition states the definition of stratification and module of a program P.

Definition 3. *Let P be a normal program which can be partitioned into the disjoint sets of rules $\{P_1, ..., P_n\}$. Let $P_i, P_j \in \{P_1, ..., P_n\}, P_i \neq P_j$, we say that $P_i < P_j$ if $\exists r \in P_j : \exists r' \in P_i : H(r') \in B(r)$, if from this condition we do not conclude that $P_i < P_j$ or $P_j < P_i$ then we can choose to hold whether $P_i < P_j$ or $P_j < P_i$ as long as the following properties hold. For every $X, Y, Z \in \{P_1, ..., P_n\}$, the strict partial order relation properties and the totality property hold:*

1. *$X < X$ is false (this property holds trivially).*
2. *If $X < Y$ then ($Y < X$ is false).*
3. *If ($X < Y$ and $Y < Z$) then $X < Z$.*
4. *$P_1 < ... < P_n$*

then we refer to this partition as the stratification of P, sometimes we will write it as $P = P_0 \cup ... \cup P_n$. And we will refer to $P_i, 1 \leq i \leq n$ as a module of P.

2.3 Parallel Computing

Two types of information flow through a processor: instructions and data. The instruction stream is defined as a sequence of instructions by the processing unit. The data flow is defined by the exchange of data between memory and processing unit, according to Flynn's taxonomy any of the two streams can be individual or multiple, given these configurations Flynn proposed the following categories to identify a system:

- A block of instructions and a data stream (SISD)
- A block of instructions and multiple data streams (SIMD)
- Multiple blocks of instructions and a data stream (MISD)
- Multiple blocks of instructions and multiple data streams (MIMD)

Parallel processing can occur in SIMD or MIMD architectures.

Models of parallel algorithms There are different paradigms of parallel programming, below we name yhe ones with relevance to this article.

Work Pool Model

The *work pool* or the *task pool* model is characterized by a dynamic mapping of data or tasks among processors, emulating a list of available tasks where virtually each processor can work with any source of the list.

Master-Slave Model

Also known as the *boss-worker* model, it is said that one or more teachers distribute the flows between the different slaves, on a random or a-priori way.

3 Computing the p-stable models

Now we present the implementation of a p-stable model solver.

To find the p-stable models of a program P we can first apply the transformations to P, however the application of the transformations is not absolutely necessary nor sufficient to find the p-stable models of P. In this section we start presenting the application of the transformations, and then we give two approaches to find the p-stable models of P, both following the theorem 1 which is also presented in this section, each one with a different model of parallel algorithms.

In most cases the application of the transformations is not enough to find a p-stable model of a normal program, and other techniques are required. One of those techniques is to partition the program into sets of rules called modules. Those modules are created based on its graph of dependencies [5][6].

Theorem 1. *[10] Let P be a normal logic program, and M a model of P with stratification $P = P_1 \cup P_2$, then $\boldsymbol{RED}(P, M) \models M$ iff $\boldsymbol{RED}(P_1, M_1) \models M_1$ and $\boldsymbol{RED}(P_2', M_2) \models M_2$ with P_2', M_1, and M_2 defined as follows: $M = M_1 \cup M_2$, $M_1 = h(P_1, P) \cap M$, $M_2 = h(P_2, P) \cap M$, and P_2' is obtained by transforming P_2 as follows:*

1. *Removing from P_2 the rules r' such that $B^-(r') \cap M_1 \neq \emptyset$ or $B^+(r') \cap (h(P_1, P) \setminus M_1) \neq \emptyset$, obtaining a new program P_2''.*
2. *For every $r \in P_2''$, removing from $B(r)$ the occurrences of the atoms in $h(P_1, P)$, obtaining P_2'.*

In other words M is a p-stable model of P iff M_1 is a p-stable model of P_1 and M_2 is a p-stable model of P_2', where P_2' is obtained by removing from P_2 the occurrences of the atoms in $h(P_1, P)$ according to the theorem 1. If P can be stratified as $P = P_1 \cup ... \cup P_n$, $n > 2$, then $P = P_1 \cup Q$ with $Q = P_2 \cup ... \cup P_n$ is also an stratification of P that has only two modules, and then we can apply the theorem 1.

4 Workflow of implementation

Now that we have described the basics for obtaining p-stable models of a program p, it is necessary to explain the workflow followed by our implementation and the part which can be parallelized.

This time we use an ordered list of processes to show the workflow, a brief description of each one may get you to understand the funtion of the implementation.

1. *Reading of logic program p.* We use the lparse format due to its compatibility with a large amount of systems.
2. *Grounding.* Necessary to make the work easier, in our application we use GrinGo from Potassco suite. This gets us a program with no variables just facts.

3. *Transformations.* As deffined before, those are made to reduce the program lenght.
4. *Stratification if possible.* Our parralel approach only gets necessary if the program can be stratified.
5. *Model generation.* Using ClaspD from the Potassco suite, we generate candidates for the program.
6. *Model verification.* This part needs a longer explanation given its importance for parallel application.
7. *Showing results.*

4.1 Model verification

After performing the analysis to the implementation we decided to perform the parallelization of the part that makes the verification of the models, since in this part of the code shows that there is a sequential process where there are no strict dependencies when performing checks. Once again we show the process of model verification through an ordered list.

1. *Receiving stratifications and proposed models of program P* We use an stack were the stratifications are stored.
2. *Test the feasibility of each stratification and its model assigned* This step repeats until each stratification and its model have been analised or when one isnt satisfactible. Here we can easily see a natural parallelizable code.
3. *Analysis of results*

5 Parallel algorithms

From theorem 1,and the analisys of the model verification code two different approaches to compute the p-stable models of a program P in parallel are proposed. The first one uses an hybrid work-pool schema as we can see below *Algorithm 1.* The second one uses another hybrid using as a base the master-slave paradigm *Algorithm 2.*

6 Conclusions and further Work

The development of hybrid algorithms based on the paradigms of the workpool and master-slave brings great benefits when parallelized, since both approaches have a natural way to be implemented the first allows each node look for work in a list, which allows a more dynamic data manipulation; the second approach is classic, but on examination it was found that the master should have a well constructed load balancer to avoid becoming a bottleneck when allocating tasks, since in many cases segments of P are very small and may cause lost of time during communications. Currently the basic application its made in Java, and its being migrated to C++ in order to use MPI [6] to run the application in parallel plataforms such a cluster and a multiprocessor server getting faster services natural to C applications.

[6] http://www.open-mpi.org/

Algorithm 1 Work_pool_check

Require: Pi and Mi {Pi are the stratiffied elements of P and Mi their proposed models}
Ensure: Shows if Mi are valid {Otherwise it finish early}
  while Board.hasElements= $TRUE$ and Board.status=NoError **do**
    setPtemporal(null);
    setMtemporal(null);
    findPi() and findMi() {on board}
    setPtemporal(Pi);
    setMtemporal(Mi);
    deleteFromBoard(Pi, Mi);
    Pi.setStatus(pendant);
    if testFactibility(Pi)= $TRUE$ **then**
      Pi.setStatus(factible);
    else
      Board.status=Error;
    end if
  end while

Algorithm 2 Master_Slave_check

Require: Pi and Mi {Pi are the stratiffied elements of P and Mi their proposed models}
Ensure: Shows if Mi are valid {Otherwise it finish early}
  if amIMaster= $TRUE$ **then**
    while PiList.hasElements= $TRUE$ and PiList.status=NoError **do**
      checkPiList();
      giveWorkToNodes(); {free nodes receive Pi and Mi sets}
      waitResponseFromNodes();
      updatePiList();
    end while
    giveEndNoticeToNodes();
  else
    while moreWorkToCome= $TRUE$ **do**
      waitWorkFromMaster();
      verifyPiMiFactibility();
      sendResultsToMaster();
    end while
  end if

References

1. M. Balduccini and E. Pontelli. Issues in parallel execution of nonmonotonic reasoning systems. In *Parallel Computing*, number 31 in 6, pages 608–647, 2005.
2. E. Ellguth and M. Gebser. A simple distributed conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 490–495. Springer Berlin / Heidelberg, 2009.
3. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
4. J. W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, second edition, 1987.
5. G. López. Solver para p-stable. To be publish at BUAP.
6. A. Marín. Algoritmos para semánticas de programación lógica. Master's thesis, BUAP, 2011.
7. M. Osorio, J. Arrazola, and J. L. Carballido. Logical weak completions of paraconsistent logics. *Journal of Logic and Computation, doi: 10.1093/logcom/exn015*, 2008.
8. M. Osorio and J. L. Carballido. Brief study of G'$_3$ logic. *Journal of Applied Non-Classical Logic*, 18(4):79–103, 2008.
9. M. Osorio, J. A. Navarro, J. Arrazola, and V. Borja. Logics with common weak completions. *Journal of Logic and Computation*, 16(6):867–890, 2006.
10. S. Pascucci. Syntactic properties of normal logic program under pstable semantics: theory and implementation. Master's thesis, March 2009.
11. S. Pascucci and A. Lopez. Implementing p-stable with simplification capabilities. *Submmited to Inteligencia Artificial, Revista Iberoamericana de I.A.*, Spain, 2008.
12. D. Pearce. Stable Inference as Intuitionistic Validity. *Logic Programming*, 38:79–91, 1999.
13. E. Pontelli and M. Balduccini. Non-monotonic reasoning on beowulf platforms. In Springer, editor, *PADL '03 Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 37–57, 2003.