An Investigation of Sharing Strategies for Answer Set Solvers and SAT Solvers

Hung Viet Le and Enrico Pontelli

Department of Computer Science
New Mexico State University
{hle,epontelli}@cs.nmsu.edu

Abstract. This paper describes a parallel engine for Answer Set solving, based on exploitation of search parallelism. The work explores a range of alternative strategies for work sharing, describing their implementations and comparing their efficiency. These results indicate methodologies to combine sharing strategies and select the most effective one depending on properties of the problem.

1 Introduction

In recent years, there has been a significant increase of interest towards the application of logic-based technology—in particular, technology based on propositional and SAT solving—in a variety of application domains. This renewed interest has also been guided by the development of formal modeling and programming paradigms based on these concepts, such as the widely used *Answer Set Programming (ASP)* [9]. ASP builds on logic programming and answer set semantics [5], to provide a set-oriented programming paradigm. In ASP and SAT, the problem is modeled using clauses of a propositional theory, and solutions are represented by minimal or *stable* [5] models of the theory. A significant push to these efforts comes from the development of efficient implementations of ASP and SAT solvers. These implementations are practical and scalable to real-life domains. The execution mechanisms employed by ASP are analogous to those used in general SAT solving, and are based on highly optimized and/or specialized versions of the Davis-Putnam procedure. In this work, we will focus on a state-of-the-art ASP engine—i.e., an implementation of the SMOLETS algorithms [13]—though the proposed ideas are applicable to related systems.

In spite of the efficiency provided by existing systems, there are areas where ASP and SAT solving provide elegant, compact, and highly declarative solutions, but whose execution requirements are beyond the capabilities of existing systems. E.g., ASP is widely used in planning in complex domains [7], but the high computational requirements limit the domains and goals that can be effectively addressed. *Parallelism* has been identified as a natural avenue to further improve applicability of ASP and SAT solving to real-world problems. Preliminary steps have been taken in the design of parallel ASP [4, 11] and SAT solvers [1, 2, 15].

The literature is rich of studies related to the design of parallel engines for traditional logic programming (mostly Prolog) [6], theorem provers [2], and constraint solvers [10]. Nevertheless, recent investigations [11] have highlighted that results from these

related areas are not directly transferable to ASP—e.g., task sharing techniques proved optimal in the context of Prolog have provided suboptimal performances in ASP.

In this paper, we are interested in the development of techniques to exploit parallelism from ASP at the *search level* (a.k.a. *or-parallelism*) and in the context of distributed search strategies [2]. Parallelism at the search level implies the presence of multiple processes (*search agents*) that search in parallel the solution space of the problem; in ASP and SAT, this corresponds to the concurrent construction of *distinct models* of the propositional theory, where each agent explores different truth assignments to the logical variables of the theory. Theoretical research in the area of search parallelism [6, 12] underlined that dynamic distribution of work is vital to achieve adequate parallel performance, and two components of the execution model have the greatest impact: the *sharing* strategy and the *scheduling* strategy. Scheduling determines the policy to be used to select tasks to be exchanged during execution, while the sharing strategy determines how the exchange of tasks takes place. Our focus is on the latter.

The contribution of this paper is the design of the first complete parallel ASP system (supporting the *complete* SMODELS language) on *Beowulfs*. Our focus is on investigating the impact of different sharing strategies. We explore a variety of alternatives, some adaptations to ASP of known methodologies and some novel, and study their behavior on a representative set of benchmarks. The results suggest that flexible dynamic selection of sharing strategies is vital to guarantee high parallel performance.

2 A Parallel Engine for ASP

Sequential Execution Model: The objective of a computation is, given a propositional theory (extended Horn clauses in the case of ASP), to determine one or more minimal models of the theory. For ASP, we are seeking a special set of minimal models, called *stable models* [5]. The execution is a fixpoint computation which alternates two phases: *boolean constraint propagation* and *atom splitting* [2]. During constraint propagation, clauses in the theory are used to extend a partial model, adding to the model those atoms whose truth value is uniquely determined by the theory (w.r.t. the partial model). Whenever constraint propagation is not possible, the system performs *atom splitting*, by selecting an unknown atom and “guessing” its truth value; this corresponds to the creation of a choice-point, since backtracking needs to explore both alternative truth values for such atom. Heuristic strategies (based on estimating the size of the subtrees) are employed in the selection of the atom during splitting, to guarantee effective propagation. Because of the non-determinism of atom splitting, the computation can be visualized as a search tree, where the nodes correspond to the choice-points created by splitting.

Organization of the Parallel Computation: Forcing an atom into the partial model, via atom splitting, will create two branches of the search tree, corresponding to the two roles of the atom (true and false). The system needs to completely traverse both sides of this computation subtree separately, and this is the core of the parallel search process—i.e., assign distinct branches to different search agents. The left (right) branch of each node corresponds to setting up the atom’s value to true (false). In search parallelism, the two branches may be explored concurrently by separate agents.

The initial steps of the parallel computations are performed following a static partition of work. A divide and conquer scheme is applied to the first $\approx \lg_2 n$ levels of the search tree (n is the number of agents). Initially all agents independently perform the initial boolean propagation; at the first splitting, processors are partitioned along the two branches—distribution of processors between the two branches is based on the estimated size of the two subtrees (using the same heuristic strategy of [13]). The process continues until individual agents are assigned to distinct subtrees. This process does not require inter-process communication and resembles the *guiding path* generation process of PSATO [15]. After this initial setting, the system switches to a fully dynamic distributed scheduling strategy, where an idle agent secures new tasks by performing a *sharing* operation with another agent. We consider scheduling strategies that are *receiver-initiated* (i.e., idle agents initiate scheduling). The dynamic scheduler (briefly described in Sect. 3.3) is expected to determine a pair $\langle \mathcal{P}, \mathcal{N} \rangle$, where $\mathcal{P}$ is the active agent from where tasks should be acquired (*work-sender*), and $\mathcal{N}$ is the node (*target node*) in the tree containing the task to be exchanged. Once the pair $\langle \mathcal{P}, \mathcal{N} \rangle$ has been determined, a sharing operations has to be performed between the idle agent and $\mathcal{P}$. We will refer to the idle agent as the *work-receiver*. An idle agent which is searching for new tasks is called *work-checker*. The location of the idle agent in the search tree is called *start node*. A node with an unexplored alternative is said to be *open*.

Essential Data Structures: The basic data structures employed extend the traditional design used for linear time computation of declarative closures. Inter-connected objects are used to represent each rule and each atom. The computation is based on the use of a *stack*, which maintains a representation of the partially constructed stable model (as references to atoms). If an atom is determined to be true/false, it is pushed on the stack along with its truth value. An array (*history record*) has been introduced to record the choice-points created during the construction of the current branch of the search tree—i.e., it provides a compact representation of the branch in the search tree built by the agent. During the execution, search agents operate on the leaves of the search tree; to assist the sharing operations, each search agent maintains an array (*relative positions*) recording the *nearest common ancestor* (*nca*) nodes between the agent's leaf and the leaf of any other agent. In a distributed setting, determination of the *relative positions* require inter-process communication. In this work, the updates are lazily done via broadcast messages during the sharing operations.

3 Parallel Work Sharing Strategies

The objective of the sharing operation is to transfer tasks between two agents. In order to allow an agent to restart its computation from a different node, it is necessary to reconstruct the correct execution state; i.e., if the agent moves to node $\mathcal{N}$, then it will have to instantiate its data structures to reflect the partial model associated to node $\mathcal{N}$. [6] surveys over 20 different schemes to address the sharing problem for Prolog.

The formal analysis of search parallelism [12] recognized methods based on constant-time access to the partial model and constant-time alternative selection to be optimal. This restricts our choice to sharing methods that maintain the same model and

program representation as in sequential models, and that reconstruct the necessary segments of the computation *only* when sharing takes place. We can recognize two key approaches: *copying*—where the necessary segments of computation are copied from the work-sender to the work-receiver—and *recomputing*—where the work-receiver rebuilds the segments of computation with minimal information from the work-sender. Copying requires significant communication but it minimizes the amount of work done by the receiver, while recomputing minimizes communication but requires high computation effort by the receiver. Copying has been widely adopted in parallel logic programming and considered as the most effective scheme [6]. We explore a solution where copying and recomputation can alternate depending on the application.

3.1 Recomputation-Based Methods

In the recomputation-based methods, the idle agent acquires work by recreating the state of computation existing at the time the node where a new alternative is available was created. In its most direct version, the content of the data structures representing the clauses and atoms and the content of the stack (representing the partial model) are recomputed, starting either from the root of the search tree or from the nca node of the target node and the start node, and ending in the target node [6]. The *work-sender* only needs to send a part of its history record, used to guide the *work-receiver* in making the correct choices during recomputation (i.e., what alternative to take in each choice-point created during recomputation). We introduce 4 recomputation methods, which differ in the amount of recomputation performed and in the way agents move in the search tree.

Method 1: Recomputation with Backtracking (*ReBack*). The key requirement in this method is that each agent maintains its *relative positions* in the search tree w.r.t. the other agents. The *work-receiver* acquires work by first backtracking to the nca node of the start node and the target node (a.k.a. the *relative position*), and then initiating the recomputation operation from there. The stack and the other data structures, including the rule and atom objects, are reconstructed from the *relative positions*. The path that runs from the nca node to the target node in the search tree is called the *connection*. The description of the connection is present in the work-sender—it is a segment of its *history record* (Sect. 2). The connection is exchanged between the two agents, and used by the work-receiver to perform recomputation. Both the *work-receiver* and the *work-sender* will also mark the target node as explored (to avoid duplication of work). The *relative positions* between agents also need to be updated during the sharing operation. When the receiver has completed the recomputation process, it will replace its *relative positions* array with a copy of the *relative positions* of the work-sender. Meanwhile, the *work-sender* broadcasts the new position of the work-receiver to all agents, allowing them to update their own *relative positions* arrays (w.r.t. the work-receiver).

The lack of guarantee regarding the order of arrival of messages sent by different agents might create situations where agents incorrectly update their *relative positions*. The overall effect is that the system may be unable to schedule work to idle agents. We refer to this problem as the *mismatch situation*.

Method 2: Recomputation by Backtracking-Compare History (*ReBackHis*). In this method, instead of maintaining the *relative positions* between each pair of agents, the

agents explicitly exchange their history records at the beginning of the sharing operation, determining at such moment the *nca* node in the search tree w.r.t. their positions. Whenever two agents exchange work, the *work-receiver* sends its own history records to the *work-sender*. Having compared to the received list, the *work-sender* figures out and sends back to the *work-receiver* the *nca* node and the *connection*. The main advantage of this alternative is that the work exchange is done locally, between two agents *without notifying anyone else*. On the other hand, the agents do not know their relative positions, and they may attempt to seek work from agents that have a *nca* node close to the root (i.e., agents that are “far” from the receiver). This may cause longer backtrack/recomputation phases, but we avoid the messages to maintain the relative positions between agents, reducing traffic and avoiding the mismatch situation.

Method 3: Recomputation Reset (*ReReset*). In the recomputation reset scheme, the initial backtracking to the *nca* node between the positions of the two agents is avoided. The backtracking step is replaced by a recomputation that starts *always* from the root of the search tree. Such scheme requires each agent to store its state at the root of the tree—i.e., the result of the first constraint propagation—and the ability to make an agent efficiently switch back to such initial state. This operation is called *Reset*, and it can be accomplished by (a) emptying the stack; (b) over-writing the rule and atom objects with a permanently saved copy of their state at the root node; (c) removing all the atoms present in the various temporary queues used by the boolean constraint propagation procedures. The relative positions between the agents are no longer necessary, and the only communication required is the connection from the root to the target node (which might be significantly bigger than the connection in *ReBackHis*).

Method 4: Recomputation Reset Split (*ReResetSplit*). In this method, instead of sharing the highest node which contains alternatives, the sender sends to the *work-receiver* the complete path from the root to the lowest node in its branch. All the nodes with unexplored alternatives along the branch are distributed between the two agents according to an interleaved scheme—i.e., the first open node is kept by the *work-receiver*, the *sender* keeps the second one, the *work-receiver* keeps the third, etc. This scheme bears some similarities to the *stack splitting* scheme used in some parallel Prolog systems [14], where the partitioning is made in contiguous blocks of choice-points; our scheme makes use of an interleaved distribution—impractical in Prolog (due to the need of handling side-effects) but effective in ASP, and expected to give rise to more balanced distributions—since it is unpredictable whether the “richer” nodes are in the upper or lower levels of the search tree. Compared to the other recomputation methods, *ReResetSplit* requires larger amount of data exchanged between agents (the complete history record of the *work-sender*), and the sender is also required to travel more to get to the new computation state. However, the agents can share in a single interaction a large number of open nodes, quickly accessible via local backtracking.

3.2 Copy-Based Methods

In copying, instead of reconstructing the computation state, the idle agent acquires work by copying the data structures stored in the *work-sender* agent. In most of the cases, the data to be transferred include the components of the rule and atom objects that are part of

the computation state (e.g., the truth value of the atoms, the counters in the objects used to keep track of the state of the clause), the stack, the history record, and the *relative positions* of the agent. In order to facilitate the copy process, we have separated the rule objects in two parts—a static part (set during the initialization phase and never copied) and a dynamic part (modified during the remainder of the execution). The dynamic parts are collected in arrays to facilitate copying. Also, changes to atom objects are *trailed*, and only the modifications are copied. Observe that, while in recomputation the execution develops from the root of the tree towards the open node, in copying the computation restarts from the open node and moves up via backtracking.

Method 5: Incremental Copying Split-Maintain the Relative Positions (*IncCopySplit*). In this method, we exploit the basic copying mechanism, enriched with the idea of *incrementality*—incrementality derives from the fact that the idle worker has already traversed the part of the branch in the search tree from the root node to the nca node, thus, there is no need to copy it. Like in the case of recomputation by backtracking, each agent maintains its *relative positions* with the other agents. The *work-receiver* backtracks to the nca node, while the *work-sender* transfers the part of the stack from the nca node to the selected open node. The set of clauses and atoms with up-to-date parameters and links are included in this copying operation. The *work-receiver* unpacks the data set and updates its data structures. The update process is performed by replacing the dynamic parts of rules and atoms with the received ones and adding the content of the received stack to the local stack. Similarly to the *ReBack* case, all other agents have to be notified (via broadcast) of the new position of work-receiver in the search tree. Concurrently to the exchange of data, the agents perform an interleaved partition of the open choice-points as described in the method *ReResetSplit*.

Method 6: Incremental Copying Split-Compare History (*IncCopySplitHis*). This method is simpler than *IncCopySplit*. The *relative positions* between the agents, employed to support the incremental copying behavior, are no longer required. Analogously to the *ReBackHis* scheme, the agents exchange their history records to find out the nca node. As soon as such node has been determined, the sharing process proceeds exactly as in the *IncCopySplit* scheme. The only additional difference is that the broadcast messages to notify the change of position of the work-receive agents are not required. As for the *ReBackHis* scheme, scheduling has to be performed blindly, without knowledge of the relative positions of the agents, but the message traffic is dramatically reduced and the mismatch situations are avoided.

Method 7: Copying All Split (*CopyAll*). This method is the simplest between our copying approaches. Whenever a sharing operation is invoked, the sender transfers the complete database of up-to-date rules and atoms (dynamic parts only), along with the complete stack. The receiver empties its stack and it installs the received data. Compared to the *IncCopySplitHis* scheme, the *CopyAll* allows the same type of blind scheduling (with the same advantage in terms of reduced traffic), but it does not require the initial exchange of history records (since knowledge of the nca node is not required). On the other hand, the method requires the copying of the complete stack, which might contain a substantially larger amount of data than in the incremental case.

3.3 Further Implementation Details – Scheduling and Termination

The irregular structure of ASP search trees requires the use of a dynamic distributed scheduling scheme [6, 14]. Each agent alternates between execution phases and scheduling phases. The lack of a central scheduler leads to a situation in which agents do not have knowledge about the location and status of other agents in the tree. Consequently, they need to exchange information (e.g., the *relative positions* broadcasts) to ensure a good load balancing. The scheduler addresses two aspects: (i) establishes policies for exchange of scheduling information; (ii) determines global termination (using a token-ring termination detection). All communications rely on a time-out mechanism, where unanswered task requests are discarded by the idle agent—this has shown large improvements in performance w.r.t. schemes requiring acknowledgments.

Fig. 1. Puzzle (left) and Queens Benchmarks

The Searching Agent: Whenever an agent has completely exploited all the alternatives locally available in its branch, it starts requesting work. The work requests are arranged according to a dynamic ordering of agents (stored in a *priority array*). By using a good ordering, a searching agent can reduce the volume of data in copying or recomputation. Depending on the individual approach, there are alternative strategies to compute the priority array. E.g., the methods that keep the *relative positions* of agents, sort the priority array based on the level of the nca nodes. The other methods start with a random order and re-sort the array based on the observed communications (e.g., a late message indicating tasks availability will increase the priority of that agent). Agents are contacted in the order they appear in the priority array. Time-outs lead to generation of a new request sent to the next agent in the array. The agent enters the idle state when all its attempts to get work failed. If the idle agent receives the white token, it will forward the token to the next agent in the ring. If it receives a black token, then it will know that agents are still working, and it will revert to work-search and scheduling.

4 Experimental Results

The system has been implemented on a Beowulf cluster (Xeon 1.7GHz, 1GB RAM, Linux, and Myrinet), using Java+MPIJava. The parallel system has an average overhead of less than 10% over the sequential solver (the SMOELS [13]). We tested the

Fig. 2. Speedups for Benchmarks

implementation on a set of benchmarks, focusing on evaluating the different sharing strategies. All the timings presented are averages over 5 runs.

The *Puzzle* benchmark (4×4 numeric puzzle) is characterized by a balanced search tree with fairly short branches and a balanced distribution of choice-points in the different branches; as a result (Fig. 1) the performance and the scalability is good, and all methods provide comparable results. On the other hand, in the *Queens* benchmark (14-queen problem, Fig. 1), we can observe that the speedup of *ReBack* is significant lower due to (1) the mismatch problem and (2) the high cost of recomputation. In this problem, the branches are long and the cost for generating them is comparatively higher than just copying the corresponding data structures. Furthermore, splitting methods tend to perform worse; when the system works near the end of the branches, the agents exchange work more frequently and the work has “poorer” quality; in this case splitting schemes tend to generate larger messages (to secure larger chunks of work) without being balanced by advantages in terms of work obtained. The analogous behavior occurs in the *Vertex* problem (30-node vertex covering, Fig. 2(i)). The speedups are higher thanks to the large number of choice-points generated (towards the root part of the tree).

In the *Seating* party benchmark (Fig. 2(ii)) we see marked differences in performance. All speedups tend to flatten for many processors, due to the small number of choice-points, located in the top part of the tree. In this experiment, the winners are the simple methods: *ReReset* and *CopyAll* where the idle agents start from the root of computation tree. In the *Pigeon Hole* experiment (pigeonhole problem with 9 holes and 10 pigeons, Fig. 2(iii)), all methods reach excellent speedups (11 using 12 agents). The

benchmark has both long and short branches, with a balanced distribution of choice-points on each branch. The *Car Plan* (planning with continuous time, Fig. 2(iv)) has a peculiar behavior. It offers a large number of choice-points but many with small branches. The recomputation schemes are superior, as they quickly reconstruct short branches, while copying requires transferring all atom and rule objects.

The problems *Lp1-Lp4* are synthetic problems proposed in ASSAT [8]—and challenging for SAT solvers. The speedups of the system are almost linear. *ReBack* and *IncCopySplit* slow down in some cases due to mismatch situations. Similarly, *CopyAll* tends to produce worse performance due to the large size of the models constructed—making non-incremental copying considerably slower.

Discussion: The experiments conducted led to important conclusions regarding the development of efficient solvers: (a) There is considerable less uniformity than other frameworks (i.e., no clear winner); (b) The simple methods (e.g., *ReReset*) behave better than the complicated ones. (c) The recomputation approach appears to be relatively better than the copying approach in most of the experiments.

In greater details, we have that:

- Sharing methods that rely on exchange of histories for determining the nca node are faster than methods that rely on approximated *relative positions* and broadcast.
- If the number of agents is small, the methods maintaining the *relative positions* behave very well. Although the system must handle more messages, the speedups show the benefits of choosing the “right” agents to share work with.
- Short executions are not suitable to copying (due to messages overhead).
- In recomputation, stack splitting does not greatly improve performance. In *ReReset*, the *receiver* stops recomputing at the first open node, while in *ReResetSplit* it has to recompute until the last node. The gain from the extra open nodes has to balance the extra cost; this is not the case in search trees with work near the root.

This indicates the need to support different sharing strategies in the same solver; different strategies might lead over 60% variations in speedups. The most suitable strategy can be selected based on various factors. At the application level, relevant factors include the *number of agents* (if small, methods based on *relative positions* are better), the *hardware platform* (if the ratio of cpu speed vs. interconnection speed is high, then recomputation is preferred), and the *size of the application* (recomputation seems better for small applications). Factors arising *during* the computation can also be employed to dynamically select the most effective strategy; let *expand_rate* denote the ratio $\frac{\text{numberOfAtoms}}{\text{numberGuessedAtoms}}$, where *numberGuessedAtoms* is the number atom splits and *numberOfAtoms* is the total number of atoms in the stack (size of the partial model).

0. If *expand_rate* is high then copying is better than recomputation. If the test is performed dynamically, and the focus is on incremental/relative positions methods, then *expand_rate* is limited to the part of the search tree branch below the nca node.
1. If recomputation is chosen, then backtracking can be selected if the path from the root to the nca node is significantly longer than the path from the start node to the nca node, otherwise reset can be employed. In copying, the copy-all schemes are more effective than incremental-copy if the nca node is closer to the root.

2. Based on the number of atoms between contiguous choice-points, an agent can decide whether to use splitting. E.g., in recomputation, if the sender estimates that the number of atoms between the consecutive open nodes is large, or the *expand-rate* is large, and there are only a few open nodes, then it transfers work without splitting.
3. If the system uses the *relative positions* and a mismatch situation occurs, then it can temporarily, or permanently, switch to a reset or an history comparison scheme.

Work is in progress to build a self-adapting sharing procedure based on these ideas.

Communication Frequency: All the communications between agents are asynchronous. It is very important to determine the frequency used by the agents to check for incoming messages, and the interval used to wait for a reply before giving up. Our experiments indicate that the frequency and delay intervals should be tied to the estimated size of the tasks available in the open-nodes—to avoid giving away small tasks or depriving the work-sender of work. Another significant factor is the sharing strategy adopted, in particular: (1) the presence of splitting requires a lower frequency; (2) approaches having higher communication requirements (e.g., copying or methods using *relative positions*) impose lower frequencies and higher delays; (3) approaches requiring more time to reach the alternative (e.g., recomputation) need longer delays.

5 Conclusions and Future Work

In this paper, we presented preliminary results from an investigation of efficient methodologies for the execution of ASP and SAT solvers on Beowulf clusters. In particular, we focused on the analysis of distinct task sharing strategies, obtained as variations of the *copying* and the *recomputation* schemes—which have been theoretically proved [12] to be optimal in the context of search parallelism. The different schemes have been developed to cover a significant spectrum of alternatives, balancing computation and communication. The ideas have been developed in a complete parallel ASP solver, and evaluated on a variety of benchmarks.

Relatively little work has appeared in the literature regarding parallelization of SAT and ASP solvers. A preliminary proposal in this area has been presented in [11], based on a simple solver; [4] presented a master-slave ASP search parallel engine, based on PVM (with limited evaluation). [15] describes a distributed implementation of the SATO SAT solver, based on a master-slave structure; the model relies on a fairly standard copying scheme. PaSAT [1] is a parallel SAT solver based on shared memory and dynamic scheduling. [3] presents a large scale copy-based SAT solver.

Current work is focused on analyzing dynamic scheduling strategies and on the investigation of how scheduling interacts with the sharing strategies presented here.

References

1. W. Blochinger et al. Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. *Theory & Apps. of Satisfiability Testing*, ENDM, 2001.
2. M. Bonacina. Taxonomy of Parallel Strategies for Deduction. *Annals Math & AI*, 29, 2000.
3. W. Chrabakh, R. Wolski. A Parallel SAT solver for the Grid. UCSB TR 2003-05, 2003.

4. R. Finkel et al. Computing Stable Models in Parallel. *AAAI Spring Symposium*, 2001.
5. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. *ILPS*, 1988.
6. G. Gupta et al. Parallel Execution of Prolog Programs. *ACM TOPLAS*, 23(4):472–602, 2001.
7. V. Lifschitz. Answer Set Planning. *LPNMR*, 373–374. Springer, 1999.
8. F. Lin and Y. Zhao. Computing Answer Sets By SAT Solvers. *AAAI*, 2002.
9. V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. *The Logic Programming Paradigm*. Springer Verlag, 1999.
10. L. Perron. Search and Parallelism in Constraint Programming. *CP*, Springer, 1999.
11. E. Pontelli and O. El-Kathib. Construction of a Parallel Engine for ASP. *PADL*, 2001.
12. D. Ranjan et al. On the Complexity of Or-Parallelism. *NGC*, 17(3):285–308, 1999.
13. P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD, HUT, 2000.
14. K. Villaverde et al. A methodology for order-sensitive execution of non-deterministic languages on beowulf platforms. *Euro-Par*, pages 694–703, 2003.
15. H. Zhang et al. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *J. Symbolic Computation*, 11:1–18, 1996.