Exploiting Vertical Parallelism from Answer Set Programs

Enrico Pontelli and Omar El-Khatib
Department of Computer Science
New Mexico State University
epontell@cs.nmsu.edu

Introduction

In the last ten years we witnessed a rapid development of alternative logical systems, called *non-monotonic logics* (Baral & Gelfond 1994; Apt & Bol 1994; Minker 1993)—which allow new axioms to retract existing theorems, and result to be more adequate for common-sense reasoning and modeling dynamic knowledge bases. One of the outcomes of research in the field of non-monotonic logics is represented by the development of a number of languages for knowledge modeling and manipulation. In particular, in the last couple of years a novel *programming paradigm* has arisen, called *Answer Sets Programming (ASP)* (Marek & Truszczynski 1999; Niemela to appear), which builds on the mathematical foundations of *logic programming* and *non-monotonic reasoning*. ASP offers novel and highly declarative solutions in a number of *well-defined application areas*, including intelligent agents, planning, and software modeling & verification. ASP currently benefits from solid and well-developed mathematical foundations, but its programming aspects still require considerable research. In particular, there is the need to *(1)* develop efficient implementations of inference engines for ASP, and *(2)* develop methodologies for software development in the ASP framework—i.e., methodologies for representing knowledge using the constructs offered by ASP. Indeed, many of the research teams involved in the development of ASP are currently investing considerable effort in these directions (Cholewinski, Marek, & Truszczynski 1996; Eiter *et al.* 1998; Niemela & Simons 1997; Lifschitz 1999). The goal of this project is to tackle some of these issues, using constraint solving and parallel processing technology. In particular, in this paper we present some preliminary ideas that have been used to address the first issue—i.e., improving performance of ASP engines—through the use parallelism.

A Sequential Execution Model for ASP

Various execution models have been proposed in the literature to support computation of answer sets and some of them have been applied as inference engines to support ASP systems (Bell *et al.* 1993; Chen & Warren 1996; Cholewinski, Marek, & Truszczynski 1996; Niemela to appear; Eiter *et al.* 1998).

In this project we propose to adopt an execution model which is built on the ideas presented in (Niemela to appear) and effectively implemented in the popular *Smodels* system (Niemela & Simons 1997). The choice is dictated by the relatively simplicity of this execution model and its apparent suitability to exploitation of parallelism. The system consists of two parts, a compiler—we are currently using the `lparse` compiler (Syrjanen 1998)—which is in charge of creating atom tables and performing program grounding, and an engine, which is in charge of computing the answer sets of the program. Our interest is focused on the engine component. A detailed presentation of the structure of the Smodels execution model (Niemela & Simons 1997) is outside the scope of this paper. In the rest of this section we propose an intuitive overview of the basic execution algorithm.[1] Figure 1 presents the overall execution cycle for the computation of stable models.

As from Figure 1, the computation of answer sets can be described as a non-deterministic process—needed since, in general, each program Π may admit multiple distinct answer sets. The computation is an alternation of two operations, `expand` and `choose_literal`. The `expand` operation is in charge of computing the truth value of all those atoms that have a determined value in the current answer set (i.e., there is no ambiguity on whether they are true or false). The `choose_literal` is in charge of arbitrarily choosing one of the atoms not present in the current answer set (i.e., atoms which do not have a determined value) and "guessing" a truth value for it.

Non-determinism originates from the execution of $\texttt{choose\_literal}(\Pi, B)$, which selects an atom l satisfying the following properties: the atom l appears negated in the program Π, and neither l nor its negation are currently present in B. The *chosen* atom is added to the partial answer set and the expansion process is restarted.

Each non-deterministic computation can terminate in three different ways:

1. *successfully* when B assigns a truth value to all the atoms and B is an answer set of Π;
2. *unsuccessfully* when a conflict is detected—i.e., B as-

[1] The presentation does not have any pretense of completeness.

```
function compute (Π : Program, A : LiteralsSet)
begin
    B := expand(Π, A) ;
    while ( (B is consistent) and
            (B is not complete) )
        l := choose_literal(Π, B);
        B := expand(Π , A ∪ { l }) ;
    endwhile
    if (B stable model of Π) then
        return B;
end
```

Figure 1: Basic Execution Model for ASP

```
function expand (Π : Program, A : LiteralsSet)
begin
    B := A ;
    while ( B ≠ B′ ) do
        B′ := B;
        B := apply_rule(Π, B);
    endwhile
    return B ;
end
```

Figure 2: Expand procedure

signs both values true and false to an atom a;

3. *unsuccessfully* when B assigns a truth value to each atom without any conflict, but B does not represent an answer set of Π. This situation typically occurs when a positive literal[2] a is introduced in B (e.g., it is guessed by `choose_literal`) but the rules of the program do not provide a "support" for the truth of a in this answer set.

As in traditional execution of logic programming, non-determinism is handled via backtracking to the choice points generated by `choose_literal`. Observe that each choice point produced by `choose_literal` has only two alternatives: one assigns the value true to the chosen literal, and one assigns the value false to it.

The `expand` procedure mentioned in the algorithm in Figure 1 is intuitively described in Figure 2. This procedure repeatedly applies expansion rules to the given set of literals until no more changes are possible. The expansion rules are derived from the program Π and allow to determine the truth status of literals; the rules used in this procedure are derived from the work of Niemela and Simons (Niemela & Simons 1997) and Warren and Chen (Chen & Warren 1996). Efficient implementation of this procedure requires considerable care to avoid unnecessary steps, e.g., by dynamically removing invalid rules and by using smart heuristics in the `choose_literal` procedure (Niemela & Simons 1997).

Parallelism in ASP

The structure of the computation of answer sets illustrated previously can be easily seen as an instance of a constraint-based computation (Bell *et al.* 1993; Subrahmanian, Nau, & Vago 1995), where

- the application of the expansion rules (in the `expand` procedure) represents the propagation step in the constraint computation
- the selection of a literal in the `choose_literal` procedure represents a labeling step

From this perspective, it is possible to identify two sources of non-determinism in the computation:

- *horizontal non-determinism:* which arises from the choice of the next expansion rule to apply (in `expand`);
- *vertical non-determinism:* which arises from the choice of the literal to add to the partial answer set (in `choose_literal`)

These two forms of non-determinism bear strong similarities respectively with the *don't care* and *don't know* non-determinism traditionally recognized in constraint and logic programming (Gupta *et al.* 2000; Van Hentenryck 1989).

The goal of this project is to explore avenues for the exploitation of parallelism from these two sources of non-determinism. In particular, we will use the terms

- *vertical parallelism* to indicate a situation where separate threads of computation are employed to explore alternatives arising from vertical non-determinism
- *horizontal parallelism* to indicate the use of separate threads of computation to concurrently apply different expansion rules to a given set of literals

Preliminary experimental considerations have underlined the difficulty of exploiting parallelism from ASP computations. In particular

- considerable research has been invested in the design of algorithms and heuristics to provide fast computation of answer sets; realistically, we desire to maintain as much as possible of such technology;
- the structure of the computation (seen as a search tree where nodes correspond to the the points of non-determinism) can be extremely irregular and ill-balanced. Size of the branches can become very small—thus imposing severe requirements for granularity control;
- neither form of non-determinism dominates on the other—i.e., certain ASP programs perform very few choices of literals (i.e., calls to `choose_literal`), while spending most of the time in doing expansions, while other programs explore a large number of choices. The structure of the computation is heavily dependent on the nature of the program. There are programs which lead directly to answer sets with little or no choices (e.g., positive programs), other which invest most of their time in searching through a large set of literals for answer sets.

This leads to the following conclusions:

- ASP does not allow the immediate reuse of similar technology developed in the context of parallel logic programming (Gupta *et al.* 2000);
- granularity control is a serious issue—techniques have to be adopted to collapse branches of the search tree and produce parallel computations of adequate grain size;
- both vertical and horizontal parallelism need to co-exist within the same system; although it is unclear whether we

[2] If atom a is added to B, then a receives the value true in B; if $not\ a$ is added to B, then a receives the value false in B.

need to exploit them *at the same time*, it is instead clear that both forms of parallelism may need to be (alternatively) employed during the execution of a program.

In the rest of this work we will focus on the exploitation of vertical parallelism. Exploitation of horizontal parallelism is currently under study (El-Khatib & Pontelli 2000).

Exploiting Vertical Parallelism in ASP

Alternative choices of literals during the derivation of answer sets (`choose_literal` in Figure 1) are *independent* and can be concurrently explored, generating separate threads of computation, each potentially leading to a distinct answer set. Thus, *vertical parallelism* parallelizes the computation of *distinct* answer sets.

As ensues from research on parallelization of search tree applications and non-deterministic programming languages (Ranjan, Pontelli, & Gupta 1999; Clocksin & Alshawi 1988; Gupta *et al.* 2000), the issue of designing the appropriate data structures to maintain the correct state in the different concurrent branches, is essential to achieve efficient parallel behavior. Observe that straightforward solutions to similar problems have been proved to be ineffective, leading to unacceptable overheads (Ranjan, Pontelli, & Gupta 1999).

The architecture for vertical parallel ASP that we envision is based on the use of a number of ASP engines (*agents*) which are concurrently exploring the search tree generated by the search for answer sets—specifically the search tree whose nodes are generated by the execution of the `choose_literal` procedure. Each agent explores a distinct branch of the tree; idle agents are allowed to acquire unexplored alternatives generated by other agents.

The major issue in the design of such architecture is to provide efficient mechanisms to support this sharing of unexplored alternatives between agents. Each node P of the tree is associated to a partial answer set $B(P)$—the partial answer set computed in the part of the branch preceding P. An agent acquiring an unexplored alternative from P needs to continue the execution by expanding $B(P)$ together with the literal selected by `choose_literal` in node P. Efficient computation of $B(P)$ for the nodes in the tree is a known complex problem (Ranjan, Pontelli, & Gupta 1999).

Since ASP computations can be very ill-balanced and irregular, we opt to adopt a dynamic scheduling scheme, where idle agents navigate through the system in search of available tasks. Thus, the partitioning of the available tasks between agents is performed dynamically and is initiated by the idle agents. This justifies the choice of a design where different agents are capable of traversing a shared representation of the search tree to detect and acquire unexplored alternatives. We will explore a number of optimization schemes to improve efficiency of these mechanisms, via run-time transformations of the search tree.

Implementation Overview

As mentioned earlier, the system is organized as a collection of agents which are cooperating in computing the answer sets of a program. Each agent is a separate ASP engine, which owns a set of *private* data structures employed for the computation of answer sets. Additionally, a number of *global* data structures, i.e., accessible by all the agents, are introduced to support cooperation between agents. This structuring of the system implies that we rely on a shared-memory architecture.

The different agents share a common representation of the ASP program to be executed. This representation is stored in one of the global data structures. Program representation has been implemented following the general data structure originally proposed in (Dowling & Gallier 1984)—proved to guarantee very efficient computation of standard models. This representation is summarized in Figure 3. Each rules is represented by a descriptor; all rules descriptors are collected in a single array, which allows for fast scan of the set of rules. Each rule descriptor contains, between the other things, pointers to the descriptors for all atoms which appear in the rule—the head atom, the atoms which appear positive in the body of the rule, and the atoms which appear negated in the body of the rule. Each atom descriptor contains information such as *(i)* pointers to the rules in which the atom appears as head, *(ii)* pointers to the rules in which the atom appears as positive body element, *(iii)* pointers to the rules in which the atom appears as negative body element, and *(iv)* an *atom array* index. Differently from the schemes adopted in sequential ASP engines (Dowling & Gallier 1984; Niemela & Simons 1997), our atom descriptors *do not* contain the truth value of the atom. Truth values of atoms are instead stored in a separate data structure, called *atom array*. Each agent maintains a separate atom array, as shown in Figure 3; this allows each agent to have an independent view of the current (partial) answer set constructed, allowing atoms to have different truth values in different agents. E.g., in Figure 3, the atom of index i is true in the answer set of one agent, and false in the answer set computed by another agent.

Each agent acts as a separate ASP engine. Each agent maintains a local stack structure (the *trail*) which keeps track of the atoms whose truth value has already been determined. Each time the truth value of an atom is determined (i.e., the appropriate entry in the atom array is set to store the atom's truth value), a pointer to the atom's descriptor is pushed in the trail stack. The trail stack is used for two purposes. During `expand`, the agent uses the elements newly placed on the trail to determine which program rules may be triggered for execution. Additionally, a simple test on the current size of the trail stack allows each agent to determine whether all atoms have been assigned a truth value or not. The use of a trail structure provides also convenient support for exploitation of horizontal parallelism (El-Khatib & Pontelli 2000).

To support the exploitation of vertical parallelism, we have also introduced an additional simple data structure: a *choice point* stack. The elements of the choice point stack are pointers to the trail stack. These pointers are used to identify those atoms whose truth value has been "guessed" by the `choose_literal` function. The choice points are used during backtracking: they are used to determine which atoms should be removed from the answer set during backtracking, as well as which alternatives can be explored to compute other answer sets. This is akin to the mechanisms

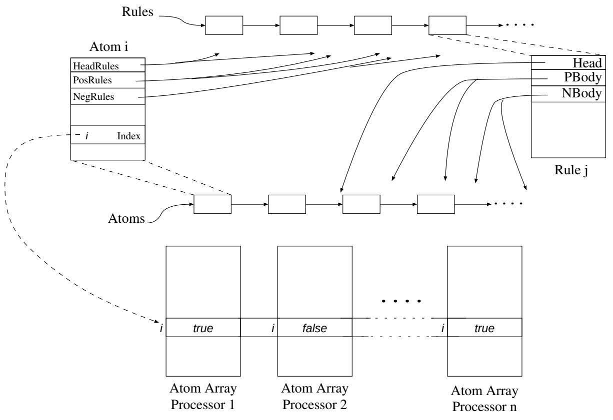

Figure 3: Representation of Rules and Atoms

used in trail-based constraint systems (Schulte 1997).

The open issue which remains to be discussed is *how* agents *interact* in order to exchange unexplored alternatives—i.e., how agents *share* work. Each idle agent attempts to obtain unexplored alternatives from other active agents. In our context, an *unexplored alternative* is represented by a partial answer set together with a new literal to be added to it. In this project we have explored two alternative approaches to tackle this problem. In *Recomputation-based Work Sharing*, agents share work by exchanging the list of *chosen* literals which had been used in the construction of an answer set; the receiving agent will use these to *reconstruct* the answer set and then perform local backtracking to explore a new alternative. In *Copy-based Work Sharing* instead, agents share work by exchanging a complete copy of the current answer set (both *chosen* as well as *determined* literals) and then performing local backtracking. The two schemes provide a different balance between amount of data copied and amount of time needed to restart the computation with a new alternative in a different agent.

Another important aspect that has to be considered in dealing with this sort of systems is termination detection. The overall computation needs to determine when a global fixpoint has been reached—i.e., all the answer sets have been produced and no agent is performing active computation any longer. In the system proposed we have adopted a centralized termination detection algorithm. One of the agents plays the role of controller and at given intervals polls the other agents to verify global termination. Details of this algorithm are omitted for lack of space.

Recomputation Based Approach

The idea of Recomputation-based sharing is derived from similar schemes adopted in *or-parallel* execution of logic programs (Clocksin & Alshawi 1988). In the Recomputation-based scheme, an idle agent obtains a partial answer set from another agent in an *implicit* fashion. Let us assume that agent $\mathcal{A}$ wants to send its partial answer set B to agent $\mathcal{B}$. To avoid copying the whole partial answer set B, the agents exchange only a list containing the literals which have been chosen by $\mathcal{A}$ during the construction of B. These literals represent the "core" of the partial answer set; in particular, we are guaranteed that an `expand` operation applied to this list of literals will correctly produce the whole partial answer set B. This is illustrated in Fig. 4.

The core of the current answer set is represented by the set of literals which are pointed to by the choice points in the choice point stack. In particular, to make the process of sharing work more efficient, we have modified the choice point stack so that each choice point not only points to the trail, but also contains the corresponding chosen literal (the literal it is pointing to in the trail stack). As a result, when sharing of work takes place between agent $\mathcal{A}$ and agent $\mathcal{B}$, the only required activity is to transfer the content of the choice point stack from $\mathcal{A}$ to $\mathcal{B}$. Once $\mathcal{B}$ receives the chosen literals, it will proceed to install their truth values (in its atom array) and perform an `expand` operation to reconstruct the partial answer set. The last chosen literal will be automatically complemented to obtain the effect of backtracking and to start construction of the "next" partial answer set.

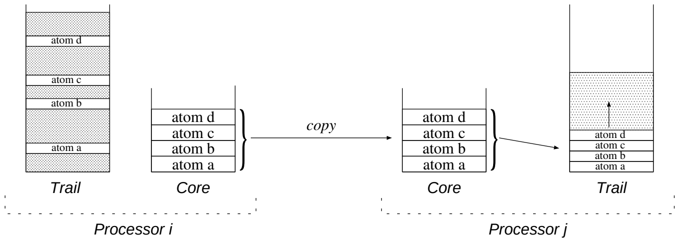

Figure 4: Recomputation-based Sharing of Work

This copying process can be made more efficient by making it *incremental*: agents exchange only the *difference* between the content of their choice point stacks. This reduces the amount of data exchanged and allows to reuse part of the answer set already existing in the idle agent.

Performance Results: In this section we present performance results for a prototype which implements an ASP engine with Recomputation-based vertical parallelism. The prototype has been developed in C and the performance results have been obtained on a Sun Enterprise. The prototype is capable of computing the answer sets of standard ASP programs, pre-processed by the lparse grounding program (Syrjanen 1998). The prototype is largely unoptimized (e.g., it does not include many of the heuristics adopted in similar ASP engines (Niemela & Simons 1997)) but its sequential speed is reasonably close to that of Smodels[3].

All figures are in milliseconds and have been achieved as average execution times over 10 runs on a lightly loaded machine. The benchmarks adopted are programs obtained from various sources (all written by other researchers); they include some large scheduling applications (sjss, rcps[4]), planners (logistics 1,2, strategic), graph problems (color), as well various synthetic benchmarks (T4, T5, T15, T8,

[3]Comparisons made with the lookahead feature turned off.

[4]These two programs have been modified from their original version to reduce the size of the ground program—to accommodate some current limitations in our memory allocation scheme.

P7). These benchmarks range in size from few tens of rules (e.g., T4, T5) to hundreds of rules (e.g., rcps).

As can be seen from the figures in Table 1, the system is capable of producing good speedups from most of the selected benchmarks. On the scheduling (sjss, rcps), graph coloring, and planning (strategic, logistics) benchmarks the speedups are very high (mostly between 6 and 8 using 10 agents). This is quite a remarkable result, considering that these benchmarks are very large and some produce highly unbalanced computation trees, with tasks having very different sizes. The apparently low speedup observed on the logistics with the first plan (logistics 1), is actually still a positive result, since the number of choices performed across the computation is just 4 (thus we cannot expect a speedup higher than 4). On the very fine-grained benchmarks T4 and T5 the system does not behave as well; in particular we can observe a degradation of speedup for a large number of agents—in this case the increased number of interactions between agents overcome the advantages of parallelization, as the different agents attempt to exchange very small tasks. In T4 we even observe a slow-down when using more than 8 agents. Two slightly disappointing results are in T8 and P7. T8 is a benchmark which produces a very large number of average-to-small size tasks; the top speedup is below 5 and denotes some difficulty in maintaining good efficiency in presence of frequent task switching. P7 on the other hand has a very low number of task switching, but generates extremely large answer sets. The speedup tends to decrease with large number of agents because some agents end up obtaining choice points created very late in the computation, and thus waste considerable time in rebuilding large answer sets during the recomputation phase.

Note that the sequential overhead observed in all cases (the ratio between the sequential engine and the parallel engine running on a single processor) is extremely low, i.e., within 5% for most of the benchmarks.

Copy-Based Approach

The Copy-based approach to work sharing adopts a simpler approach than recomputation. During work sharing from agent $\mathcal{A}$ to $\mathcal{B}$, the entire partial answer set existing in $\mathcal{A}$ is directly copied to agent $\mathcal{B}$. The use of copying has been frequently adopted to support computation in constraint programming systems (Schulte 1999) as well as to support or-parallel execution of logic and constraint programs (Gupta *et al.* 2000; Van Hentenryck 1989).

The partial answer set owned by $\mathcal{A}$ has an explicit representation within the agent $\mathcal{A}$: it is completely described by the content of the trail stack. Thus, copying the partial answer set from $\mathcal{A}$ to $\mathcal{B}$ can be simply reduced to the copying of the trail stack of $\mathcal{A}$ to $\mathcal{B}$. This is illustrated in Figure 5.

Once this copying has been completed, $\mathcal{B}$ needs to *install* the truth value of the atoms in the partial answer set—i.e., store the correct truth values in the atom array. Computation of the "next" answer set is obtained by identifying the most recent literal in the trail whose value has been "guessed", and performing local backtracking to it. The receiving agent $\mathcal{B}$ maintains also a register (bottom_trail) which is set to the top of the copied trail: backtracking is not allowed to proceed below the value of this register. This allows avoidance of duplicated work by different agents.

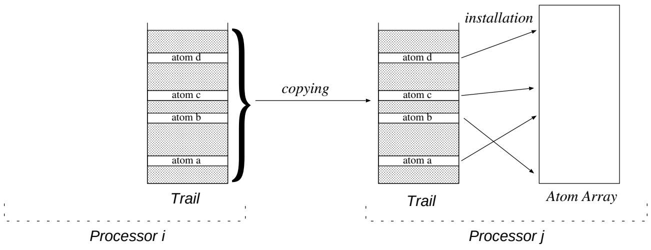

Figure 5: Copy-based Sharing of Work

As in the recomputation case, we can improve performance by performing incremental copying, i.e., by copying not the complete answer set but only the difference between the answer set in $\mathcal{A}$ and $\mathcal{B}$. A design to perform incremental copying has been completed but not implemented yet.

Performance Results: We have modified our implementation to support Copy-based work sharing, and we have tested its performance on the same pool of benchmarks.

The results reported in Figure 6 are remarkable. The large benchmarks (e.g., the two scheduling applications) report speedups in the range $8.5 - 10$ for 10 agents, maintaining linear speedups for small number of agents (from 2 to 5 agents). The fine grained benchmarks (such as T4 and T5) provide speedups similar (usually slightly better) to those observed earlier. In both cases we note a slight degradation of speedup for large number of agents. As in the case of recomputation, this indicates that if the tasks are too fine grained, additional steps are needed in order to achieve performance improvements. We have experimented with a simple optimization, which semi-automatically unfolds selected predicates a constant number of times, in order to create larger grain tasks (by effectively combining together consecutive tasks). The simple optimization has produced improvements in the speedups, as show in Table 2.

Name	2 Agents	4 Agents	8 Agents	10 Agents
T5	1.99/1.91	1.97/1.85	1.95/1.68	1.93/1.60
T4	1.92/1.60	1.95/1.50	1.93/1.51	1.91/1.49

Table 2: Improvement using Task-collapsing (new/old)

The Copy-based scheme behaves quite well in presence of a large number of average-to-small tasks, as seen in the T8 benchmark. The speedups reported in this case are excellent. This is partly due to the lower cost, in this particular case, of copying w.r.t. recomputation, as well as the adoption of a smarter scheduling strategy, made possible by the use of copying, as discussed in the next section.

For what concerns the benchmark P7, the situation is sub-optimal. In this case the need of copying large answer sets during sharing operations penalizes the overall performance. We expect this case to become less of a problem with the introduction of *incremental copying* techniques—i.e., instead of copying the whole answer set, the agents compute

Name	1 Agent	2 Agents	3 Agents	4 Agents	8 Agents	10 Agents
Scheduling (sjss)	131823.88	66146.75	44536.16	34740.25	19132.63	16214.49
Scheduling (rcps)	72868.48	36436.24	28923.33	18040.35	13169.61	10859.68
Color (Random)	1198917.24	599658.62	389737.88	300729.31	178942.87	158796.98
Color (Ladder)	1092.81	610.73	441.61	353.80	325.00	306.67
Logistics (1)	10054.68	10053.78	10054.71	4545.31	3695.67	3295.23
Logistics (2)	6024.67	3340.44	2763.14	2380.36	1407.53	1371.47
Strategic	13783.51	7317.02	5018.43	4005.83	2278.18	1992.51
T5	128.21	67.32	69.97	72.11	76.55	77.62
T4	103.01	78.14	78.33	84.11	91.79	118.21
T8	3234.69	1679.29	1164.25	905.21	761.69	710.37
P7	3159.11	1679.86	1266.91	981.75	445.33	452.19
T15	415.73	221.70	178.94	132.10	135.99	137.11
T23	3844.41	1991.76	1595.75	1433.56	1341.79	1431.70

Table 1: Recomputation-based Sharing: Execution Times (msec.)

the difference between the answer sets currently present in their stacks, and transfer only such difference. Our current prototype does not include this optimization.

Comparison: Figure 7 illustrates the differences in speedups observed by using recomputation vs. copying on some of the benchmarks. In most benchmarks the two approaches do not show relevant differences—the speedups observed differ only of a few decimal points. On the other hand, we have observed more substantial differences on the larger benchmarks (e.g., the sjss and rcps scheduling applications). In these cases the performance of the Copy-based scheme is substantially better than the Recomputation-based scheme. These differences arise because:

- the copied answer sets in these cases are very large; the cost of performing a memory copying operation of a large block of memory is substantially smaller than the cost of recomputing the answer set starting from its core. Experimental results have indicated that for answer sets with less than 350 elements recomputation provides better results than copying, while for larger answer sets copying is better. In benchmarks such as sjss and rcps the answer sets exchanged have an average size of 2500 elements. This observation is also confirmed by the behavior observed in benchmark P7: in this case, the answer sets exchanged have sizes in the order of 300 elements, and, as from Figure 7, recomputation indeed performs better than copying.
- another justification for the difference in performance is related to the style of scheduling adopted; this is discussed in detail in the next section.

To take advantage of the positive aspects of both methodologies, we have adopted an hybrid scheme, where sharing strategy is selected based on the size of the answer set.

Conclusions and Future Work

The problem tackled in this paper is the efficient execution of Answer Set Programs. Answer Set Programming is an emerging programming paradigm which has its roots in logic programming, non-monotonic reasoning, and constraint programming. This blend has lead to a paradigm which provides for very declarative programs (more declarative, e.g., then traditional Prolog programs). ASP has been proved to be very effective for specific application areas, such as planning and design of common-sense reasoning engines for intelligent agents.

The goal of this work is to explore the use of parallelism to improve execution performance of ASP engines. We have determined two forms of parallelism which can be suitably exploited from a constraint-based ASP engine. We have focused on the exploitation of one of these two forms of parallelism (what we called vertical parallelism) and presented an efficient parallel engine based on this idea. Alternative approaches to perform work sharing and scheduling are currently under investigation, in order to provide a more consistent and efficient exploitation of parallelism. Performance results for our prototype running on a Sun Enterprise system have been presented and discussed.

The project is currently focusing on integrating the second form of parallelism, horizontal parallelism, in our engine. It is clear from our experiments that a number of applications will take considerable advantage from the exploitation of this alternative form of parallelism.

Acknowledgments

The authors wish to thank A. Provetti, M. Gelfond, G. Gupta, and S. Tran for their help. The authors are partially supported by NSF grants CCR9875279, CCR9900320, CDA9729848, EIA9810732, and HRD9906130.

References

Apt, K., and Bol, R. 1994. Logic Programming and Negation: A Survey. *Journal of Logic Programming* 19/20.

Baral, C. and Gelfond, M. 1994. Logic Programming and Knowledge Representation. *J. Logic Programming* 19/20:73–148.

Bell, C.; Nerode, A.; Ng, R.; and Subrahmanian, V. 1993. Implementing Stable Semantics by Linear Programming. In *Logic Programming and Non-monotonic Reasoning*, 23–42. MIT Press.

Chen, W., and Warren, D. 1996. Computation of Stable Models and its Integration with Logical Query Processing. *Transactions on Knowledge and Data Engineering* 8(5):742–757.

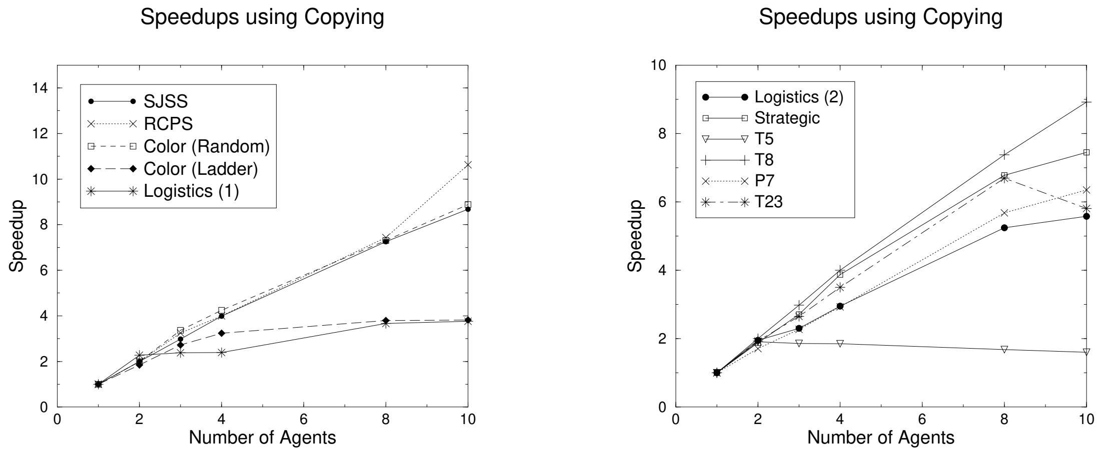

Figure 6: Speedups using Copying

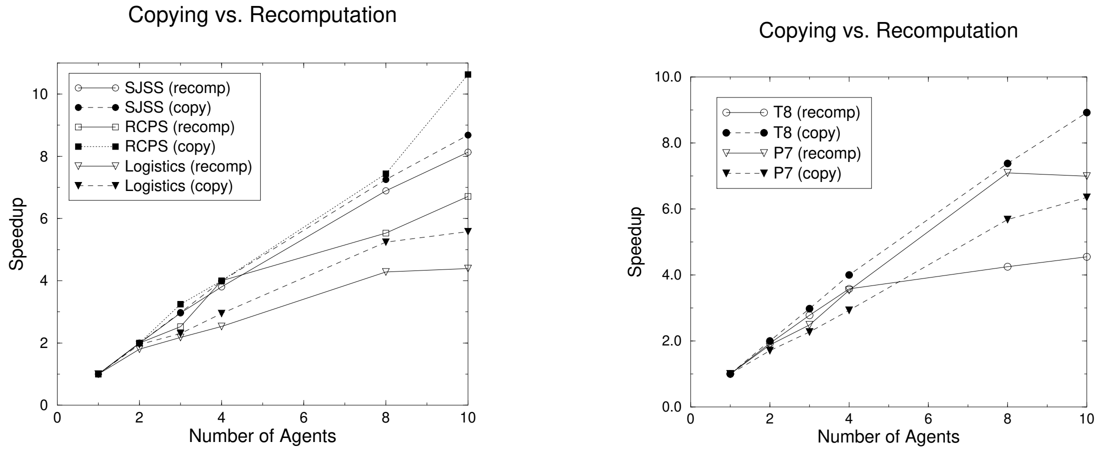

Figure 7: Comparison between Recomputation and Copying

Cholewinski, P.; Marek, V.; and Truszczynski, M. 1996. Default Reasoning System DeReS. In *Int. Conf. on Principles of Knowledge Repr. and Reasoning*, 518–528. Morgan Kauffman.

Clocksin, W., and Alshawi, H. 1988. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing* 5:361–376.

Dowling, W., and Gallier, J. 1984. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming* 3:267–289.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*, 406–417.

El-Khatib, O., and Pontelli, E. 2000. Parallel Evaluation of Answer Sets Programs Preliminary Results. In *Workshop on Parallelism and Implementation of Logic Programming*.

Gupta, G.; Pontelli, E.; Ali, K.; Carlsson, M.; and Hermenegildo, M. 2000. Parallel Execution of Prolog Programs: a Survey. Technical report, New Mexico State University.

Lifschitz, V. 1999. Action Languages, Answer Sets, and Planning. In *The Logic Programming Paradigm*. Springer Verlag.

Marek, V., and Truszczynski, M. 1999. Stable Models and an Alternative Logic Programming Paradigm. In Apt, K.; Marek, V.; Truszczinski, M.; and Warren, D., eds., *The Logic Programming Paradigm*. Springer Verlag.

Minker, J. 1993. An Overview of Nonmonotonic Reasoning and Logic Programming. *Journal of Logic Programming* 17(2/3/4).

Niemela, I., and Simons, P. 1997. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Logic Progr. and Non-monotonic Reasoning*, Springer Verlag.

Niemela, I. (to appear). Logic Programs with Stable Model Semantics as a Constraint Progr. Paradigm. *Annals of Math. & AI*.

Ranjan, D.; Pontelli, E.; and Gupta, G. 1999. On the Complexity of Or-Parallelism. *New Generation Computing* 17(3):285–308.

Schulte, C. 1997. Programming Constraint Inference Engines. In *Principles and Practice of Constraint Progr.*, Springer Verlag.

Schulte, C. 1999. Comparing Trailing and Copying for Constraint Programming. In *Int. Conf. on Logic Programming*, MIT Press.

Subrahmanian, V.; Nau, D.; and Vago, C. 1995. WFS + Branch and Bound = Stable Models. *Transactions on Knowledge and Data Engineering* 7(3):362–377.

Syrjanen, T. 1998. Implementation of Local Grounding for Logic Programs with Stable Model Semantics. Technical Report, HUT.

Van Hentenryck, P. 1989. Parallel Constraint Satisfaction in Logic Programming. In *Proc. of the Sixth International Conference on Logic Programming*. MIT Press.