Non-monotonic Reasoning on Beowulf Platforms

E. Pontelli[1], M. Balduccini[2], and F. Bermudez[1]

[1]Dept. Computer Science
New Mexico State University
epontell@cs.nmsu.edu

[2]Dept. Computer Science
Texas Tech University
marcello.balduccini@ttu.edu

Abstract. Non-monotonic logic programming systems, such as the various implementations of Answer Set Programming (ASP), are frequently used to solve problems with large search spaces. In spite of the impressive improvements in implementation technology, the sheer size of realistic computations required to solve problems of interest often makes such problems inaccessible to existing sequential technology. This paper presents some preliminary results obtained in the development of solutions for execution of Answer Set Programs on parallel architectures. We identify different forms of parallelism that can be automatically exploited in a typical ASP execution, and we describe the execution models we have experimented with to take advantage of some of these. Performance results obtained on a Beowulf system are presented.

1 Introduction

In recent years we have witnessed a rapid development of logical systems—*non-monotonic logics*—that provide the ability to retract existing theorems via introduction of new axioms. In the context of logic programming, non-monotonic behavior has been accomplished by allowing the use of negation as failure (NAF) in the body of clauses. The presence of NAF leads to a natural support for non-monotonic reasoning, allowing for intelligent reasoning in presence of incomplete knowledge. NAF is also important for various forms of database technology (e.g., deductive databases). Stable model semantics [8] is one of the most commonly accepted approaches to provide semantics to logic programs with NAF. Stable model semantics relies on the idea of accepting multiple minimal models as a description of the meaning of a program. In spite of its wide acceptance and its extensive mathematical foundations, stable models semantics have only recently found its way into mainstream "practical" logic programming. The recent successes have been sparked by the availability of very efficient inference engines (such as *smodels* [15], DeRes [3], and DLV [6]) and a substantial effort towards *understanding* how to write programs under stable models semantics [14, 12]. This has led to the development of a novel *programming paradigm*, commonly referred to as *Answer Set Programming (ASP)*. ASP is a computation paradigm in which logical theories (Horn clauses with NAF) serve as problem specifications and solutions are represented by *collection of models*. ASP has been concretized in a number of related formalism—e.g., disjunctive logic programming and Datalog with constraints [6, 5]. In comparison to other non-monotonic logics, ASP is syntactically simpler and, at the same time, very expressive. The mathematical foundations of ASP have been extensively studied; in addition, there exist a large number of *building block* results about specifying and programming using ASP—e.g., results about dealing with incomplete information and abductive assimilation of new knowledge. ASP has been successfully adopted in various domains (e.g., [1, 2, 11, 16]).

In spite of the continuous effort in developing fast execution models for ASP [6, 5, 15], computation of significant programs remains a challenging task, limiting the scope of applicability of ASP in a number of domains (e.g., planning). In this work we propose the use of *parallelism* to improve performance of ASP engines and improve the scope of applicability of this paradigm. The core of our work is the identification of potential sources for *implicit* exploitation of parallelism from a basic execution model for ASP programs—specifically the execution model proposed in the *smodels* system [15]. We show that ASP has the potential to provide considerable amounts of independent tasks, which can be concurrently explored by different ASP engines. Exploitation of parallelism can be accomplished in a fashion similar to the models proposed to parallelize Prolog [10] and constraint propagation [13].

Building on recent theoretical results regarding efficiency of parallel search in computation trees [19], we provide the design of an engine which exploits the two forms of parallelism identified (*Vertical Parallelism* and *Horizontal Parallelism*). The engine design is optimized to take advantage of the specific features of the *smodels* execution, including features such as *lookahead*. The effectiveness of our engine design in extracting parallelism is demonstrated via implementations on a *distributed memory system* (a Pentium-based Beowulf architecture) and the execution of a number of ASP benchmarks. We also investigate the use of parallelism to improve the performance of the local grounding preprocessor [22] used in *smodels*-type systems. The work proposed—which continues the work on shared memory platforms presented in [18]—along with the work concurrently conducted by Finkel et al. [7], represents the first exploration in the use of scalable architectures for ASP computations ever proposed.

The paper is organized as follows. In the next section we present an introduction to answer set programming. In Section 3 we describe the parallelization of local grounding. In Sections 4, 5, and 6 we discuss the design of the engine for the computation of the answer sets of logic programs. Performance results are presented in Section 7, while optimization, related work and conclusions are discussed in Sections 8 and 9.

2 Answer Set Programming

From Answer Set Semantics to Answer Set Programming: *Answer Sets Semantics (AS)* [8] (a.k.a. *Stable Models Semantics*) was designed in the mid eighties as a tool to provide semantics for logic programming with *negation as failure*. The introduction of NAF in logic programming leads to various complications. In particular, it leads to the loss of a key property of logic programming: the existence of a *unique* intended model for each program. In standard logic programming there is no ambiguity in what is true and what is false w.r.t. a given program. This property does not hold true anymore when NAF is allowed in the programs—i.e., programs may admit distinct independent models. Various classes of proposals have been developed to tackle the problem of providing semantics to logic programs with NAF. In particular, one class of proposals allows the existence of a *collection* of intended models (*answer sets*) for a program [8]. *Answer Sets Semantics (AS)* (also known as *Stable Models Semantics*) is the most representative approach in this class, and there is intuitive as well as formal evidence showing that AS "properly" deals with negation as failure. AS relies on a simple definition: Given a ground program P and given a "tentative" model M, we can define a new program P^M (the *reduct* of P w.r.t. M) by: *(i)* removing all rules containing atoms under NAF which are contradicted by M, and *(ii)* removing all the atoms under NAF from the remaining rules. P^M contains only those rules of P that are applicable given M. P^M is a standard logic program, without negation as failure, which admits a unique intended model M'. M is an *answer set* (or *stable model*) if M and M' coincide. In general, a program with NAF may admit multiple answer sets.

Example 1. Given a database containing information regarding people working in different departments, e.g.,

```
dept(hartley,cs). dept(pfeiffer,cs). dept(gerke,math). dept(prasad,ee).
```

we would like to select the existing departments and one (arbitrary) representative employee from each of them:

```
depts_employee(Name,Dep) :- dept(Name,Dep), not other_emps(Name,Dep).
other_emps(Name,Dep) :- dept(Name1,Dep), depts_employee(Name1,Dep), Name ≠ Name1.
```

The rules assert that `Name/Dep` should be in the solution only if no other member of the same department has already been selected. AS produces 2 possible answer sets (for the `depts_employee` predicate):

$$\{\langle \texttt{hartley}, \texttt{cs}\rangle, \langle \texttt{gerke}, \texttt{math}\rangle, \langle \texttt{prasad}, \texttt{ee}\rangle\}$$
$$\{\langle \texttt{pfeiffer}, \texttt{cs}\rangle, \langle \texttt{gerke}, \texttt{math}\rangle, \langle \texttt{prasad}, \texttt{ee}\rangle\}$$

As recognized by a number of authors [12, 14], the adoption of AS requires a *paradigm shift* to reconcile the peculiar features of AS with the traditional program view of logic programming. First of all, we need to provide programmers with a way of handling multiple answer sets. One could attempt to restore a more "traditional" view, where a single "model" exists. This has been attempted, for example, using *skeptical semantics* [12], where a formula is considered entailed from the program only if it is entailed in *each* answer set. Nevertheless, skeptical semantics is often inadequate—e.g., in many situations it does not provide the desired result, and in its general form provides excessive expressive power [12]. The additional level of non-determinism, is indeed a real need for a number of applications. Maintaining multiple answer sets bears also close resemblance to similar proposals put forward in other communities—such as the *choice* and *witness* constructs used in the database community. This creates an additional level of *non-determinism* on top of the non-determinism in traditional logic programming. Both are forms of *don't know* non-determinism: the difference is in the granularity of the choices made at each level.

Additionally, the presence of multiple answer sets leads to a new set of requirements on the *computational mechanisms* used. Given a program, the goal of the computation is not to provide a goal-directed tuple-at-a-time answer (i.e., a true/false answer or a substitution), as in traditional logic programming, but the objective is to return *whole answer sets*—i.e., set-at-a-time answers. The traditional resolution-based control used in logic programming is largely inadequate, and should give place to different control and execution mechanisms.

To accommodate for all these novel aspects, we embrace a different view of logic programming under AS, interpreted as a *novel programming paradigm*—that we will refer to as *Answer Sets Programming (ASP)* [14, 12]. In simple terms, the goal of an ASP program is to identify a *collection of answer sets*—i.e., each program is interpreted as a specification of a *collection of sets of atoms*. Each rule in the program plays the role of a *constraint* [14] on the collection of sets specified by the program: a generic rule $Head :- B_1, \ldots, B_n, not\, G_1, \ldots, not\, G_m$ indicates that, whenever $B_1, \ldots, B_n$ are part of an answer set and $G_1, \ldots, G_m$ are not, then *Head* has to be in the answer set as well. The shift of perspective from traditional logic programming to ASP is very important. The programmer is led to think about writing programs as manipulating *sets* of elements, and the outcome of the computation is a collection of sets. This perspective comes natural in a large number of application domains—e.g., graph problems deal with set of nodes/edges, planning problems deal with sets of actions. ASP has received consideration in knowledge representation and deductive database communities, as it enables to represent default assumptions, constraints, uncertainty and non-determinism *in a direct way* [2].

Sequential Implementation Technology: 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 [3, 14, 6]. In this work we adopt an execution model which is built on the ideas presented in [14] and effectively implemented in the popular *smodels* system [15]. 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 preprocessor (called *lparse* in the *smodels* system [22]) that 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 ground program. The work performed by the preprocessor is based on a local grounding for programs with a restricted syntax (strongly range restricted programs [22]). Intuitively, rules are required to contain *domain predicates*—i.e., predicates not relying on any recursive definition—and each variable in a rule is required to appear in a domain predicate. Domain predicates and their extensions are identified and computed through dependency graphs. These are used to perform local grounding of each rule—taking a natural join of the positive domain predicates in the body and then checking them against the negative ones.

Our main interest is focused on the engine component. A detailed presentation of the structure of the *smodels* engine [15] is outside the scope of this paper. In this section we propose an *intuitive* overview of the basic execution algorithm. Fig. 1 presents the overall execution cycle for the computation of stable models: the computation of answer sets can be described as a non-deterministic process—since 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. We will refer to B as *partial answer set*. The general objective is to try to expand a partial answer set into a stable model.

The meaning of the partial answer set is that, if atom a belongs to B, then a will belong to the final model. If *not* a belongs to B, a will *not* belong to the final model.

Non-determinism originates from the execution of `choose_literal`(Π, B), which selects an atom l such that neither l nor its negation are present in B. The *chosen* atom is added to the partial answer set and the expansion process is restarted. The choice of literals makes use of *lookahead* [15] to quickly exclude literals not leading to answer sets (see Sect. 6).

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

Fig. 1. Basic Execution Model for ASP

```
function expand (Π : Program, A : LiteralsSet)
    B := A ;
    do
        B' := B;
        B := apply_rule(Π, B);
    while ( B ≠ B' );
    return B;
```

Fig. 2. Expand procedure

Each non-deterministic computation can terminate either successfully—i.e., B assigns a truth value to all the atoms and it represents an answer set of Π—or unsuccessfully—if either the process tries to assigns two distinct truth values to the same atom or if B does not represent an answer set of the program (e.g., truth of certain selected atoms is not "supported" by the rules in the program). As in traditional 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 which literals have a definite truth value w.r.t. the existing partial answer set. This is accomplished by applying the rules of the program Π in different ways [15]. Efficient implementation of this procedure requires care to avoid unnecessary steps, e.g., by dynamically removing invalid rules and by using smart heuristics in `choose_literal` [15].

3 Parallel Local Grounding

The first phase of the execution is characterized by the grounding of the input program. Although most interesting programs invest the majority of their execution time in the actual computation of models, the execution of the local grounding can still require a non-negligible amount of time. We decided to investigate simple ways to exploit parallelism also from the preprocessing phase. The structure of the local grounding process, as illustrated in [22], is based on taking advantage of the strong range restriction to individually ground each rule in the program. The process can be parallelized by simply distributing the task of grounding the different rules to different agents, as in Fig. 3. The **forall** indicated in the algorithm represents a parallel computation: the different iterations are independent of each other. The actual solution adopted in our system is based on the use of a *distribution function* which statically computes a partition of the program Π (after removing all rules defining the domain predicates) and assigns the elements of the partition to the available computing agents. The choice of performing a static assignment is dictated by *(i)* the large amount of work typically generated, and *(ii)* the desire to avoid costly dynamic scheduling in a distributed memory context. The various computing agents provide as result the ground instantiations of all the rules in their assigned component of the partition of Π. The partitioning of Π is performed in a way to attempt to balance the load between processors. The heuristic used in this context assigns a weight to each rule (an estimation of the number of instances based on the size of the relations of the domain predicates in the body of the rule) and attempts to distribute balanced weight to each agent. Although simplistic in its design, the heuristics have proved effective in the experiments performed.

function ParallelGround(Π)
    $\Pi_G = \{a \mid \texttt{a is instance of domain predicate}\}$
    $\Pi = \Pi \setminus \Pi_G$
    forall $R^i \in \Pi$
        $R^i_G = \texttt{GroundRule}(R^i)$
    endall
    $\Pi_G = \bigcup R^i_G$
end

Fig. 3: Parallel Preprocessing

The preprocessor has been implemented as part of our ASP system, and it is designed to be compatible in input/output formats with the *lparse* preprocessor used in *smodels*. The preprocessor makes use of an internal representation of the program based on structure sharing—the input rule acts as skeleton and the different instantiations are described as environments for such skeleton. The remaining data structures are essentially identical to those described for the *lparse* system [22]. The implementation of the preprocessor, developed on a Beowulf system, has been organized as a master-slave structure, where the master agent is in charge of computing the program partition while the slaves are in charge of grounding the rules in each partition.

4 Parallelizing the ASP Engine

The structure of the computation of answer sets previously illustrated can be easily interpreted as an instance of a constraint-based computation [21], where the application of the expansion rules (`expand` procedure) represents the *propagation* step of the constraint computation, and the selection of a literal in `choose_literal` represents a *labeling* step. From this perspective, it is possible to identify two sources of non-determinism: **horizontal non-determinism:** which arises from the choice of the next expansion rule to apply (in `expand`), and **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 to the *don't care* and *don't know* non-determinism traditionally recognized in constraint and logic programming [10]. The goal of this project is to explore avenues for the exploitation of parallelism from these two sources of non-determinism—by exploring the different alternatives available in each point of non-determinism in parallel. In particular, we will use the terms *(i) Vertical Parallelism* to indicate a situation where separate threads of computation are employed to explore alternatives arising from vertical non-determinism; and, *(ii) Horizontal Parallelism* to indicate the use of separate threads of computation to concurrently apply different expansion rules to a given set of literals. Horizontal parallelism is aimed at the use of different computation agents to construct *one* of the models of the program—thus, the different agents cooperate in the construction of one solution to the program. Vertical Parallelism on the other hand makes use of separate computing agents for the computation of *different* models of the program—each execution thread is working on a differen answer set of the program. In the rest of this paper we focus on the exploitation of Vertical Parallelism, and on a particular form of Horizontal Parallelism, that we call *Parallel Lookahead*.

5 Vertical Parallelism

The essential idea behind Vertical Parallelism is the concurrent exploration of different alternatives associated to the guessing of the truth value of chosen literals (`choose_literal` operation). Each time a literal is guessed, two independent computations can be spawned, one which assumes the literal to be true and one that assumes the literal to be false. Exploitation of Vertical Parallelism shares the same roots as or-parallelism for Prolog [10] and search parallelism in constraint programming [20, 17]. Recent studies have underlined the inherent complexity of maintaining the correct view of execution during parallel search [19].

The overall design of the engine used for our experiments has been directly derived from the design of the engine used in the *smodels* system [15]. In this paper we are drawing our experience from two prototypical implementations (both having a very similar structure), one developed at New Mexico State University (NMSU) and one concurrently developed at Texas Tech University (TTU). The design used builds on the design previously proposed by the authors for executing ASP on shared memory architectures [18].

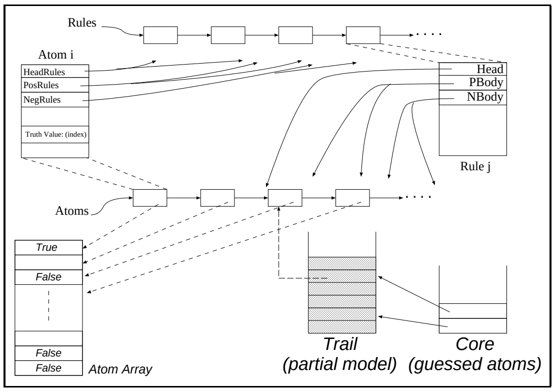

Fig. 4: Structure of an Engine

Answer set programs are internally represented using a collection of structures (both for rules and atoms) which are interlinked to allow direct access from each rule to the associated atoms and from each atom to the rules in which such atom appears (following the scheme for linear-time computations originally described in [4]). Since we are relying on a share-nothing model, each processor maintains a copy of the representation of the program. Each agent makes use of two stacks for supporting its computation. One stack (called *trail*) is used to represent the current partial answer set—each element in the current answer set is represented by an entry in the stack. Each entry is a pointer to the data structure representing the atom–and the truth value in the data structure identifies whether the atom appears positively or negatively in the answer set. For efficiency reasons the truth values are maintained in a separate array structure (the *Atom Array* in Fig. 4). The second stack, called *core*, keeps track of the answer set elements which have been "guessed" during the computation. The elements in the core allow to identify the computation points where unexplored alternatives may be available—to support backtracking and/or work sharing between agents. This structure is depicted in Fig. 4.

The architecture for vertical parallel ASP that we envision is based on the use of a number of ASP engines (*or-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 or-agent explores a distinct branch of the tree; idle agents are allowed to acquire unexplored alternatives generated by other agents.

As ensued from research on parallelization of search tree applications and non-deterministic languages [19, 10], 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. Straightforward solutions have been formally proved to be inefficient, leading to unacceptable overheads [19]. The major issue in the design of such architecture is to provide efficient mechanisms to support 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. Efficiently computing $B(P)$ for the different nodes P in the tree is a known difficult problem [19]. Due to the irregular structure of the computation (branches in the computation tree may have different and unpredictable size) effective parallel implementation of ASP requires the use of dynamic distribution of work. Mechanisms have to be designed to allow dynamic exchange of tasks during the computation.

Exploitation of Vertical Parallelism requires tackling two major issues: *(i)* work sharing: i.e., allowing idle agents to acquire unexplored tasks from active agents, efficiently reproducing the necessary computation state to restart execution; *(ii)* scheduling: i.e., guiding idle agents in the search for unexplored tasks. In [18] we have sketched solutions to these issues in the context of shared memory architectures. In the successive sections we explore how these problems have been tackled and solved in the context of share-nothing architectures.

5.1 Work Sharing

The results presented in [19] lead to the following conclusions in the context of parallel ASP: at least one of the following operations will incur a cost which is $\Omega(\lg n)$ (where n is the size of the computation tree): *(i)* access to the atoms in the partial answer set; *(ii)* execution of a `choose_literal` operation; *(iii)* acquisition of unexplored alternatives from another agent. Practical experience [10] suggests that parallel engine designs where operations *(i)* and *(ii)* are performed in constant time are preferable—i.e., the non-constant time cost should be concentrated in operation *(iii)*. The intuition behind this is that, since the non-constant time cost is unavoidable, it is favorable to locate it in operations whose frequency can be controlled by the engine—and only operation *(iii)* has this property. On top of this, the majority of the methods proposed in the literature for handling work sharing in parallel search (see [10] for a survey on the topic) heavily rely on the use of shared data structures, and are thus unsuitable for a share-nothing architecture—as the Beowulf platforms we intend to use in this project. We have identified two methods suitable to support ASP on a distributed memory architectures: *model copying* and *model recomputation*.

Model Recomputation: The idea of recomputation-based sharing of work is derived by similar schemas adopted in the context of *or-parallel* execution of logic programs [10]. 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 communication process is illustrated in Fig. 5. The core of the current answer set is represented by the set of literals which are pointed to by the choice points in the core stack (see Fig. 4). In particular, to make the process of sharing work more efficient, we have modified the core 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 core stack from $\mathcal{A}$ to $\mathcal{B}$. Once $\mathcal{B}$ receives the chosen literals, it will proceed to install their truth values (by recording the literals' truth values in the Atom Array) and perform an `expand` operation to reconstruct (on the trail stack) the partial answer set. The last chosen literal will be automatically complemented to obtain the effect of backtracking and constructing the "next" answer set. This copying process can be also made more efficient by making it *incremental*: agents exchange only the *difference* between the content of their core stacks. This reduces the amount of data exchanged and allows to reuse part of the partial answer set already existing in the idle agent.

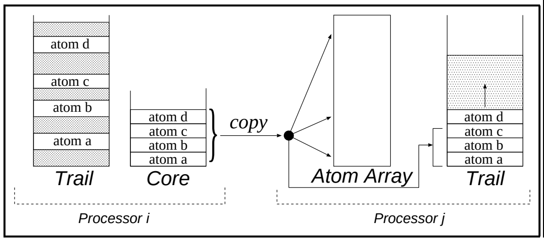

Fig. 5. Recomputation Sharing of Work

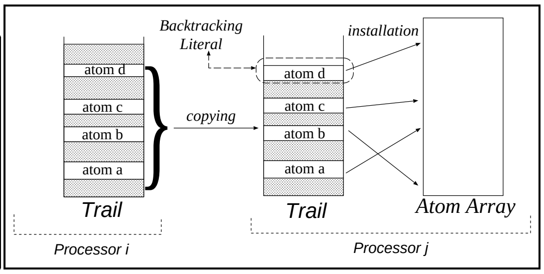

Fig. 6. Copy-based Sharing of Work

Model Copying: The copying-based approach to work sharing adopts a simpler approach then recomputation. Upon 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 [20] as well as to support or-parallel execution of logic and constraint programs [10]. 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 6. 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 recently literal whose value has been "guessed" and performing local backtracking to it. The identification of the backtracking literal is immediate as this literal lies always at the top of copied trail stack. 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 the one in $\mathcal{B}$.

Hybrid Scheme: The experiments performed on shared memory architectures [18] have indicated that Model Copying behaves better than Model Recomputation in most of the cases. This is due to the high cost of recomputing parts of the answer set w.r.t. the cost of simply performing a memory copying operation. This property does not necessarily hold any longer when we move to distributed memory architectures (as the Beowulf platform used in this project), due to the considerably higher cost for copying data between agents.

To capture the best of both worlds, we have switched in our prototype to a hybrid work sharing scheme, where both Model Recomputation and Model Copying are employed. The choice of which method to use is performed dynamically (*each time* a sharing operation is required). Various heuristics have been considered for this selection, which take into account the size of the core and the size of the partial answer set. Some typical observations that have been made from our experiments include: *(i)* if the size of the core is sufficiently close to the size of the answer set, then recomputation would lead to a loss w.r.t. copying. *(ii)* if the size of the answer set is very large compared to the size of the core, then copying appears still to be more advantageous than recomputation. This last property is strongly related to the speed of the underlying interconnection network—the slower the interconnection network, the larger is the partial answer set that we can afford to recompute. We have concretized these observations by experimentally identifying two thresholds (*low* and *high*) and a function f

which relates the size of the core and the size of the answer set; Recomputation is employed whenever $low \leq f(\,sizeof(Core), sizeof(Partial\ Answer\ Set)\,) \leq high$.

5.2 Scheduling

In the context of our system, two scheduling decisions have to be taken by each idle processor in search of work: *(1)* select from which agent work will be taken; *(2)* select which unexplored alternative will be taken from the selected agent. In the current prototype, we have tackled the first issue by lazily maintaining in each agent ($\mathcal{P}$): **(a)** an approximated view of the load in each other agent. Each agent maintains an array with an entry for each agent in the system; the i^{th} entry in the array indicates what is believed to be the load in the i^{th} agent. The entries in the load array are managed by broadcasting the updated load whenever a sharing operation occurs; **(b)** an approximated view of what is the lowest choice point in common with each other agent in the system. This information is updated via multicast each time an agent backtracks over a copied choice point. The scheduling strategy gives preference to agents which are "near" the idle one (allowing for incremental copying) and which have a sufficiently high load.

Regarding the selection of the unexplored alternatives, in [18] we explored two approaches, respectively called *top* and *bottom* scheduling. Top scheduling selects alternatives from choice points which lie closer to the root of the tree (i.e., the oldest choices made during the computation), while in bottom scheduling the most recently guessed literals are considered. From the experiments reported in [18] we observed that in general top scheduling leads to faster sharing operations (as they typically allows the agents to deal with smaller answer sets), but to more frequent calls to the scheduler. Considering the higher cost of communication in presence of share-nothing architectures, we have reverted to a variation of bottom scheduling, similar to the *Stack Splitting* method presented in [9]. In a single sharing operation, two agents share not just one unexplored alternative (taken from the youngest choice point), but a set of them—half of the unexplored alternatives available in the active agent. This method has been implemented as follows: *(i)* the last choice point is easily detected as it lies on the top of the core stack; this allows to determine what is the part of the trail that has to be copied/recomputed; *(ii)* splitting is performed by allowing the idle agent to take control of each other choice point in the core stack.

6 Horizontal Parallelism: Parallel Lookahead

The (sequential) *smodels* algorithm presented earlier builds the stable models of an answer set program incrementally. The algorithm presented in Fig. 1 can be refined to introduce the use of lookahead during the "guess" of a literal. The algorithm is modified as follows: *(1)* Before guessing a literal to continue expansion, unexplored literals are tested to verify whether there is a literal l such that $\mathsf{expand}(\Pi, B \cup \{l\})$ is consistent and $\mathsf{expand}(\Pi, B \cup \{not\, l\})$ is inconsistent. Such literals can be immediately added to B. *(2)* After such literals have been found, choose_literal can proceed by guessing an arbitrary unexplored literal. Step 1 is called the `lookahead` step. It is important to observe that any introduction of literals performed in this step is *deterministic* and does not require the creation of a choice point. In addition, the work performed while testing for the various unexplored literals can be used to choose the "best" literal to be used in step 2, according to some heuristic function.

During the lookahead step, every test performed on a pair $\langle l, not\ l\rangle$ is substantially independent from the tests run on any other pair $\langle l', not\ l'\rangle$. Each test involves up to two calls to expand (one for l, the other one for $not\ l$), thus resulting in a comparatively expensive computation. These characteristics make the lookahead step a natural point where the algorithm could be parallelized. Notice that *Parallel Lookahead* is an instance of the general concept of Horizontal Parallelism, since the results of the parallel execution of lookahead are combined, rather than being considered alternative to each other, as in Vertical Parallelism.

The appeal of exploiting Horizontal Parallelism at the level of `lookahead`, rather than at the level of `expand`, lies in the fact that the first involves a coarser-grained type of parallelism.
Basic Design: The parallelization of the lookahead step is obtained in a quite straightforward way by splitting the set of unexplored literals, and assigning each subset to a different agent. Each agent then performs the test described in step 1 on the unexplored literals that it has been assigned. Finally, a new partial answer set, B' is built by merging the results generated by the agents. Work sharing is based on the Model Copying technique.

Notice that, even in the parallel implementation, the lookahead step can be exploited in order to determine the best literal to be used in `choose_literal` (provided that the results returned by the agents are suitably combined). This significantly reduces the computation performed by `choose_literal`, and provides a simple way of combining Vertical and Horizontal Parallelism by applying a work-sharing method similar to the *Basic Andorra Model* [10], studied for parallelization of Prolog computation.
Scheduling: The key for the integration of Vertical and Horizontal Parallelism is in the way work is divided in work units and assigned to the agents. Our system is based on a central scheduler, and a set of agents that are dedicated to the actual computation of the answer sets. Every work unit corresponds to a lookahead step performed on a partial answer set, B, using a set of unexplored literals, U. Work units related to different partial answer sets can be processed at the same time by the system. Whenever all the work units associated with certain partial answer set have been completed, the scheduler gathers the results and executes `choose_literal` – which, as we stated before, requires a very small amount of computation, and can thus be executed directly on the scheduler. `choose_literal` returns two (possibly) partial answer sets[1], and the scheduler generates work units for both of them, thus completing a (parallel) iteration of the algorithm in Fig. 1, extended with `lookahead`. Under this perspective, Horizontal Parallelism corresponds to the parallel execution of work units related to the same partial answer set. Vertical Parallelism, instead, is the parallel execution of work units related to different partial answer sets. The way the search space is traversed, as well as the balance between Vertical and Horizontal Parallelism, are determined by: *(1)* the number agents among which the set of unexplored literals is split, and *(2)* the priority given to pending work units. In our implementation we assign priorities to pending work units according to a "simulated depth first" strategy, i.e., the priority of a work unit depends first on the depth, d, in the search space, of the corresponding node, n, and second on the number of nodes of depth d present to the left of n. This choice guarantees that, if a computation based only on Horizontal Parallelism is selected, the order in which nodes are considered is the same as in a sequential implementation of our algorithm. This is an important feature, because it allows us to exploit the same search heuristics present in the original smodels algorithm. These heuristics have been thoroughly tested in the past few years and proved to perform very well in most applications.

The number of agents among which the set of unexplored literals is split is selected at run-time. This allows the user to decide between a computation based on Horizontal Parallelism, useful if the answer set(s) are expected to be found with little backtracking, and a computation based on Vertical Parallelism, useful if more backtracking is expected.

7 Performance Results

In this section we show some of the experimental results collected from the implementation of the ideas presented in the previous sections. The results have been obtained on the Pentium-based Beowulf (purely distributed memory architectures) at NMSU—Pentium II (333Mhz) connected via Myrinet. The results reported have been obtained from two similar implementations of ASP, one developed at NMSU and one at TTU. Both systems have been constructed in C using MPI for dealing with interprocessor communication. The experiments

[1] Our version of `choose_literal` runs `expand` on the two partial answer sets before returning them.

have been performed by executing a number of ASP programs (mostly obtained from other researchers) and the major objective was to validate the feasibility of parallel execution of ASP programs on Beowulf platforms.

Parallel Local Grounding: We have analyzed the performance of the parallel preprocessor by comparing its execution speed with varying number of processors. The parallel preprocessor is in its first prototype and it is very unoptimized (compared to *lparse* we have observed differences in speed ranging from 4% to 48%). Nevertheless, the current implementation was mostly meant to represent a proof of concept concerning the feasibility of extracting parallelism from the preprocessing phase. The first interesting result that we have observed is that the rather embarrassingly parallel structure of the computation allowed us to make the parallel overhead (i.e., the added computation cost due to the exploitation of parallelism) almost negligible. This can be seen in Fig. 7, which compares the execution times for a direct sequential implementation of the grounding algorithm with the execution times using a single agent in the parallel preprocessor. In no cases we have observed overhead higher than 4.1%. Very good speedups have been observed in each benchmark containing a sufficient number of rules to keep the agents busy. Fig. 8 shows the preprocessing time for two benchmarks using different numbers of processors. Note that for certain benchmarks the speedup is slightly lower than linear due to slightly unbalanced distribution of work between the agents—in the current scheme we are simply relying on a static partitioning without any additional load balancing activities.

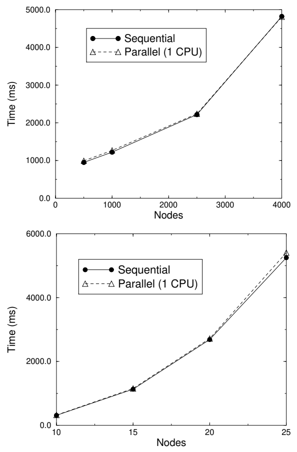

Fig. 7. Preproc. Overhead (Pigeon,Coloring)

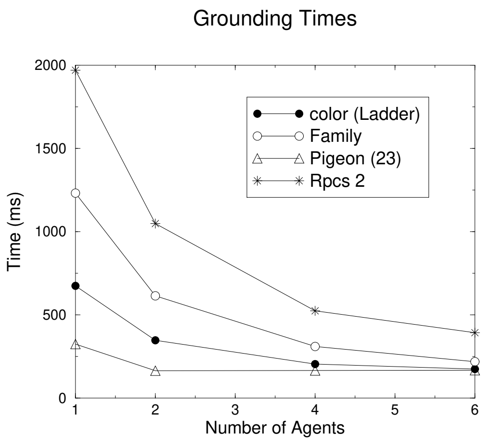

Fig. 8. Parallel Exec. of the Preprocessor

Parallel Literal Selection: The experiments for exploitation of Vertical Parallelism through parallel literal selection have been conducted using the Beowulf ASP engine developed at NMSU—an evolution of the shared memory engine previously described in [18]. All timings presented have been obtained as average over 10 runs. As mentioned in Sect. 5.1, in our design we have decided to adopt a Hybrid Method to support exchange of unexplored tasks between agents. This is different from what we have observed in [18], where Model Copying was observed to be the winning strategy in the last majority of the benchmarks. In the context of distributed memory architectures, the higher cost of communication between processors leads to a higher number of situations where the model copying provides sub-optimal performances.

Table 1 reports the execution times observed on a set of benchmarks, while Fig. 9 illustrates the speedups observed using the hybrid scheme on a set of ASP benchmarks. Some of the benchmarks, e.g., `T8` and `P7`, are synthetic benchmarks developed to study specific properties of the inference engine, while others are ASP programs obtained from other researchers.

`Color` is a graph coloring problem, `Logistics` and `Strategic` are scheduling problems, while `sjss` is a planner. Note also that `sjss` is executed searching for a single model while all others are executed requiring all models to be produced. The tests marked `[*]` in Fig. 9 indicate those cases where Recomputation instead of Copying has been triggered the majority of the times. The results presented have been accomplished by using an experimentally determined threshold to discriminate between copying and recomputation. The rule adopted in the implementation can be summarized as: if $min \leq \frac{size(Partial\ Answer\ Set)}{size(Core)} \leq max$ then model recomputation is applied, otherwise model copying is used. The intuition is that *(i)* if the ratio is too low, then, there is no advantage in copying just the core, while *(ii)* if the ratio is too high, then the cost of recomputing the answer set is likely to be excessive. The *min* and *max* used for these experiments where set to 1.75 and 12.5. Fig. 10 shows the impact of using recomputation in the benchmarks marked with `[*]` in Fig. 9. Some benchmarks have shown rather low speedups—e.g., `Color` on a ladder graph and `Logistics`. The first generates very fine grained tasks and suffers the penalty of the cost of communication between processors—the same benchmarks on a shared-memory platform produces speedups close to 4. For what concerns `Logistics`, the results are, after all, quite positive, as the maximum speedup possible is actually 5 and there seem to be no degradation of performance when the number of agents is increased beyond 5.

Name	**1 Agent**	**2 Agents**	**3 Agents**	**4 Agents**	**8 Agents**
Color (Ladder)	345201	249911	235421	292932	295420
Color (Random2)	2067987	1162905	829685	604586	310622
Logistics 2	3937246	2172124	1842695	1652869	1041534
Strategic	76207	40169	28327	21664	12580
sjss	93347226	46761140	31012367	22963465	13297326
T8	1770106	865175	590035	444730	226930
P7	1728001	918172	690924	536646	216040

Table 1. Execution Times (in μs.) on Beowulf

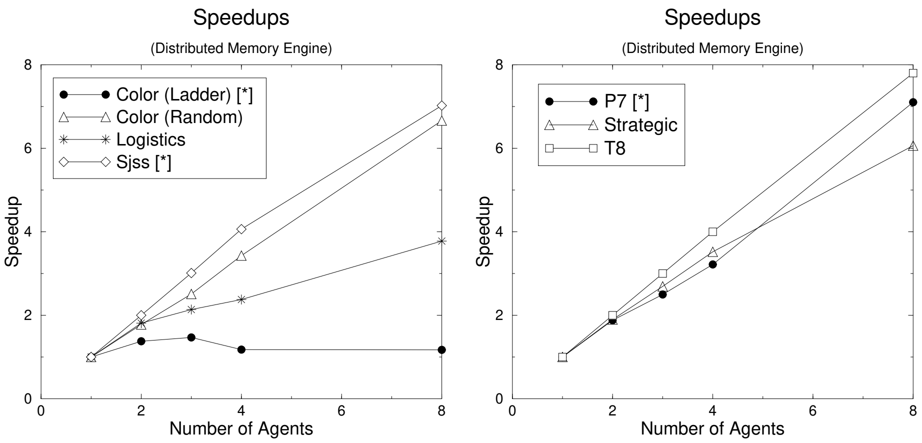

Fig. 9. Speedups from Vertical Parallelism

It is interesting to compare the behavior of the distributed memory implementation with that of the shared memory engine presented in [18]. Fig. 11 presents a comparison between the speedups observed on selected benchmarks in the shared memory and the distributed memory engines. In the majority of the cases we observed relatively small degradation in

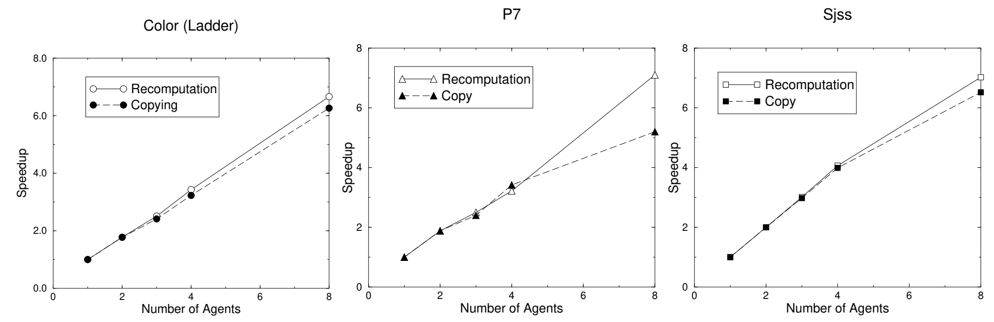

Fig. 10. Impact of using Recomputation

the speedup. Only benchmarks where frequent scheduling of small size tasks is required lead to a more relevant difference (e.g., `Color` for the ladder graph).

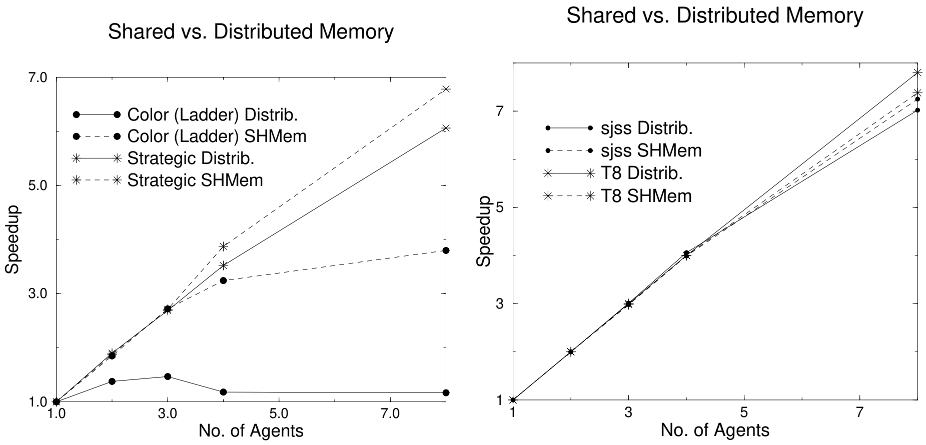

Fig. 11. Comparison of Shared and Distributed Memory Engines

Parallel Lookahead: The experiments on Parallel Lookahead have been conducted using the distributed ASP engine developed at TTU. For our tests, we have used a subset of the benchmarks available at `http://www.tcs.hut.fi/pub/smodels/tests/lp-csp-tests.tar.gz`: *(1)* `color`: c-colorability (4 colors, 300 nodes), *(2)* `pigeon`: put N pigeons in M holes with at most one pigeon in a hole ($N = 24, M = 24$), *(3)* `queens`: N-queens problem ($N = 14$), and *(4)* `schur`: put N items in B boxes such that, for any $X, Y \in \{1, \ldots, N\}$: items labeled X and $2X$ are in different boxes, and if X and Y are in the same box, then $X + Y$ is in a different box ($N = 35, B = 15$).

The tests consisted in finding one answer set for each of these programs. Since, for all of these programs, this can be accomplished with a comparatively small amount of backtracking, the engine was run so that Horizontal Parallelism was given a higher priority than Vertical Parallelism by acting on the number of agents among which the set of unexplored literals is split. The experiments show, in general, a good speedup for all programs. The speedup, for 13 processors, is 5 for `schur` and `pigeon`, almost 6 for `color`, and 120 for `queens`. The speedup measured for `queens` is indeed surprising. It is interesting to note that `queens` requires (with *smodels*) the highest amount of backtracking. We conjecture that the speedup observed is the result of the combined application of both types of parallelism. However this issue deserves further investigation before any precise statement can be made. The results are definitely encouraging if we consider that: to the best of our knowledge, our system is one of the first exploiting Horizontal Parallelism; the way parallelism is handled is

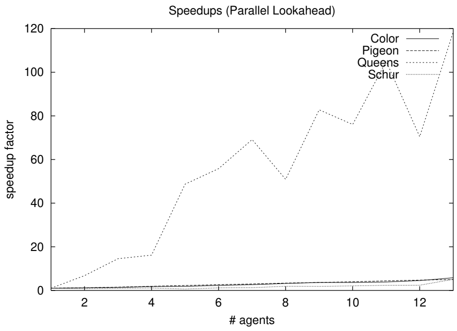

Fig. 12. Speedups for Parallel Lookahead

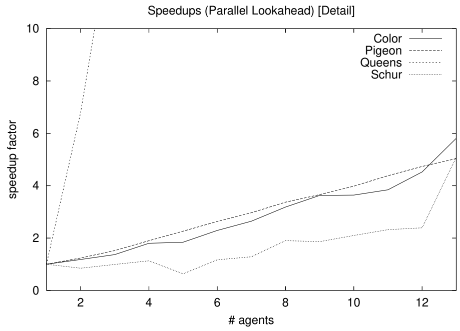

Fig. 13. Speedups for Parallel Lookahead

still very primitive if compared with the other existing parallel systems; the level of refinement of the algorithms for the computation of answer sets is still far beyond *smodels* (we expect the optimizations exploited in *smodels* to significantly improve speedup).

8 Optimizations

Optimizing Vertical Parallelism: Various optimizations can be envisioned to improve the performance of the basic vertical parallel engine. Many of the general optimization principles discussed for parallel execution of Prolog [10] are likely to reduce the parallel overhead. We have applied two optimizations in the development of the parallel engine. Whenever a sharing operation is performed (either using copying or recomputation), the copying agent needs to perform an "installation" operation used to erase the truth value of those literals which have been removed from the partial answer set and add the truth value of those literals copied from the remove agent. This process is typically accomplished by forcing the copying agent to backtrack to the nearest common ancestor in the computation tree between the position of the two agents (for removing literals) and by an explicitly installing the truth value of the copied literals. While the installation is a fairly fast operation (especially when recomputation is used), the backtracking step can be fairly expensive. We have introduced an optimization which trades the cost of backtracking for the cost of copying additional data from the remote agent. The idea is that if the common ancestor is "too far away" and close to the root of the tree, it may be cheaper to avoid backtracking, removing *all* the literals from the partial answer set (using a brute force operation, e.g., the system call `memzero`), and then copy the complete answer set from the remote agent. Fig. 14 shows the improvements observed by triggering this optimization whenever the size of the answer set at the common ancestor is less than 512.

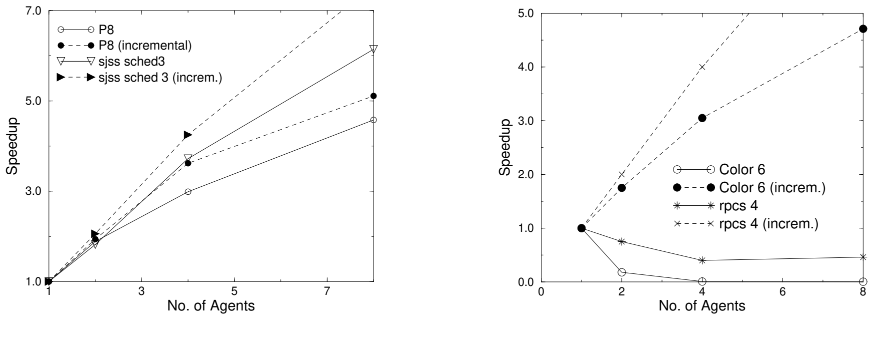

Fig. 14. Speedup Curves with and without Memory Zeroing Optimization

Optimizing Parallel Lookahead: Further research is needed in order to improve the efficiency of the system. Different types of improvements can be identified.
(1) Design improvements, aimed at decreasing the overhead due to communications. Improvements will probably need to be focused on the selection of the correct work sharing model, for which the hybrid method is a good candidate. The development of better scheduling techniques will also be important to achieve a higher efficiency.
(2) Optimization of the heuristic function used to find the "best" literal for `choose_literal`, in order to exploit the features of the parallel implementation: we are currently using a heuristic function close to the one used in *smodels*, designed for sequential implementations.
(3) Improvements aimed at making the system able to *self-adapt* according the type of logic program whose answer sets are to be found. Research has to be conducted on techniques for selecting the correct balance between Vertical Parallelism and Horizontal Parallelism depending on the task to be performed.

9 Related Work and Conclusions

The problem tackled in this work is the efficient execution of Answer Set Programs. Real-life ASP applications can easily become very time consuming, to the point that various programs (e.g., large planning applications) are beyond the computational capabilities of existing inference engines. The goal of this work is to explore the use of parallelism to improve execution performance of ASP engines. Starting from the basic design of an inference engine for ASP (the one proposed in the *smodels* system) we have identified two major sources of parallelism—*Horizontal* and *Vertical* Parallelism. We have focused on the design of technology to allow the exploitation of Vertical Parallelism in the context of a distributed memory architecture. Within Vertical Parallelism, we have distinguished between standard parallel branching and parallel lookahead to provide further scope of exploitation of parallelism. The various issues related to the exploitation of this form of parallelism have been analyzed and solutions proposed. We have also briefly explored the issue of parallelization of the preprocessing phase required for the execution of answer set programs.

The potential for exploitation of parallelism from ASP computations has been recently recognized by other authors as well: [7] proposes a PVM-based implementation of a *smodels*-type engine with Vertical Parallelism—parallelism is extracted from the actual operation of guessing the truth value of a chosen literal, and scheduling is centralized. The work we propose has also strong ties to the work on parallel execution of logic programs [10] and non-deterministic languages [19]. With respect to parallel execution of logic programs, the vertical parallelism used in our work can be related to or-parallelism in Prolog, and horizontal parallelism can be related to deterministic parallelism. With respect to non-deterministic languages, there are similar aspects in the construction of the search tree – each branch represents a solution, and the way nodes are handled involves the ability to reconstruct part of the computation (e.g., "environments").
Acknowledgments: The authors wish to thank G. Gupta, S. Tran, and M. Gelfond for their help. E. Pontelli and F. Bermudez were partially supported by NSF grants CCR9875279, CCR9900320, CDA9729848, EIA0130887, EIA9810732, and HRD9906130. M. Balduccini was partially supported by United Space Alliance under Research Grant 26-3502-21 and Contract COC6771311, and by NASA under Contracts 1314-44-1476 and 1314-44-1769.

References

1. M. Balduccini and M. Gelfond. Diagnostic Reasoning with A-Prolog. *Theory and Practice of Logic Programming* (to appear), 2002.
2. C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.
3. P. Cholewinski et al. Default Reasoning System DeReS. In *Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kauffman, 1996.

4. W.F. Dowling and J.H. Gallier. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 3, 1984.
5. D. East and M. Truszczyński. Datalog with Constraints. In *National Conference on Artificial Intelligence*, pages 163–168. AAAI/MIT Press, 2000.
6. T. Eiter et al. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *Int. Conf. on Principles of Knowledge Representation and Reasoning*, 1998.
7. R. Finkel et al. Computing Stable Models in Parallel. In *AAAI Spring Symposium on Answer Set Programming*, pages 72–75, 2001. AAAI/MIT Press.
8. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *Int. Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.
9. G. Gupta and E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-Parallelism on Distributed Machines. In *ICLP*, pages 290–304, 1999. MIT Press.
10. G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, 2001.
11. K. Heljanko and I. Niemela. Answer Set Programming and Bounded Model Checking. In *AAAI Spring Symposium*, pages 90–96, 2001.
12. V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm*. Springer Verlag, 1999.
13. T. Nguyen and Y. Deville. A Distributed Arc-Consistency Algorithm. *Science of Computer Programming*, 30(1–2):227–250, 1998.
14. I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, 2001.
15. I. Niemela and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *LPNMR*, Springer Verlag, 1997.
16. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog Decision Support System for the Space Shuttle. In *PADL*, Springer Verlag, 2001.
17. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Int. Conf. on Principles and Practice of Constraint Programming*, 1999. Springer Verlag.
18. E. Pontelli and O. El-Kathib. Construction and Optimization of a Parallel Engine for Answer Set Programming. In *PADL*, 2001. Springer Verlag.
19. D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *New Generation Computing*, 17(3):285–308, 1999.
20. C. Schulte. Comparing Trailing and Copying for Constraint Programming. In *International Conference on Logic Programming*, pages 275–289. MIT Press, 1999.
21. V.S. Subrahmanian, D. Nau, and C. Vago. WFS + Branch and Bound = Stable Models. *Transactions on Knowledge and Data Engineering*, 7(3):362–377, 1995.
22. T. Syrjanen. Implementation of Local Grounding for Logic Programs with Stable Model Semantics. Technical Report B-18, Helsinki University of Technology, 1998.