DASPPS - DISTRIBUTED ANSWER-SET PROGRAMMING WITH PS^+

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Jason High, M.A.

San Marcos, Texas

May 2005

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Propositional satisfiability (SAT) is an important problem in both theoretical and practical computer science. Practical application of the SAT formalism can be found in testing and verification, FPGA routing, path delay analysis, and VLSI. Because SAT is NP-complete, computational resource availability and utilization represents a significant problem area in the design and implementation of efficient SAT solvers.

Several efforts have been made to distribute existing sequential solver heuristics to capitalize on latent CPU cycles of available networked machines. While a noticeable speed up has been achieved by a number of these distributed solvers, several problems become apparent:

1. Many of the leading solver heuristics use a shared database of partial solutions. Distributing these heuristics results in costly communication overhead between nodes. For example, the GradSAT [Chrabakh and Wolski, 2003a] distributed solver relies upon shared learned clauses which must be sent to all nodes as they are produced.

2. Because the run-time is not a function of problem size, it is difficult to efficiently partition the problem between nodes.

3. Many of the leading distributed solvers employ a communication framework

which limits scalability. Parallel Satz [Jurkowiak et al., 2001], for example, relies upon the Network File System (NFS), which limits scalability to a local area network.

Development in the area of distributed logic solvers has, with the exception of NAGSAT [Forman and Segre, 2002], taken an existing sequential solver, such as Satz [Li and Anbulagan, 1995] or zchaff [Moskewicz et al., 2001], and modify the sequential solver to take advantage of networked resources. Recently, a new sequential solver, *aspps* [East and Truszczyński, 2001a], has been developed, based on the extended logic of propositional schemata, denoted PS^+ [East and Truszczyński, 2001b]. Special constructs within the logic PS^+, such as explicit representation of cardinality constraints, allow the *aspps* system to significantly reduce the problem search space. In addition, search space partitioning results in independent sub-theories. These two features suggest that the *aspps* system is a prime candidate for parallelization.

In this thesis, we seek to address the three common problems found in distributed solvers. We design and implement a distributed solver based on the sequential *aspps* solver, and call it *daspps* . The search algorithm of the *daspps* system is designed to take advantage of the special constructs of the logic PS^+ and exploit these features in a distributed framework. In chapter 9 we compare the performance of *daspps* with the sequential *aspps* solver and show that a distributed *aspps* solver is a significant development in the field of distributed SAT solvers.

This thesis is organized as follows. Chapters 2 and 3 present background material on answer-set programming and the logic PS^+. Chapter 4 discusses the aspps sequen-

tial solver. Chapter 5 surveys the distributed solvers currently available. Chapter 6 presents the current technique of segmentation used by the *daspps* system. Chapter 7 and 8 present the current implementation details of the *daspps* system. Chapter 9 presents the comparative runtime results between the *daspps* and *aspps* systems. Chapter 10 gives some concluding remarks on the results of this thesis. Chapter 11 discusses the two primary issues affecting the scalability of the existing system.

CHAPTER 2

ANSWER SET PROGRAMMING

Answer-set programming (ASP) is a framework for declarative programming which emerged from the area of logic programming with stable logic semantics [Gelfond and Lifschitz, 1988]. In ASP, a problem is represented as a theory in some logic so that *models* of the theory represents solutions to the problem [East and Truszczyński, 2004]. Finding models, rather than proofs, is the primary task and is performed by an answer set solver, such as *smodels* [Niemela and Simons, 1997] or *dlv* [Citrigno et al., 1997]. Problems such as search, planning, and diagnosis can be represented as logic programs so that stable models represent solutions. This chapter reviews the basic tenets of the ASP formalism.

An ASP formalism is a declarative programming formalism where programs are constructed from theories whose models represent problem solutions. A formalism in the ASP paradigm is a formalism based on an underlying formal language with well-defined semantics. Given a formalism F of a logic language and a program P, the semantics of F defines a mapping to a program P of F, bound by the semantics of F, to a collection of sets. These sets are the answers sets of program P [East and Truszczyński, 2004].

The ASP framework has been shown to be well suited for search problems [East and Truszczyński, 2004]. A search problem Π is defined

[Garey and Johnson, 1979] as a set D_Π of finite instances such that for each instance $I \in D_\Pi$ there exists a finite set $S_\Pi(I)$ of solutions for I. An algorithm is said to solve a search problem Π if, given as input any instance $I \in D_\Pi$, it returns failure if $S_\Pi(I)$ is empty, otherwise it returns at least one solution s, $s \in S_\Pi(I)$. Consider, for example, the k-graph coloring problem. Given an undirected graph $G = (V, E)$ and a set of k colors, is there a k-color assignment to vertices such that no two vertices of the same color are joined by an edge. The solution set consists of all satisfying k-color assignments.

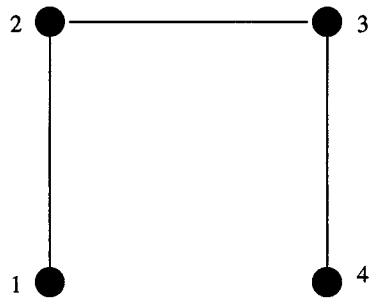

Figure 2.1: Graph for 3-coloring graph coloring problem.

The general principle of the ASP paradigm is that models of the theory represent solutions to the problem instance. This principle is applicable to any logic system where the concept of models is well-defined [East and Truszczyński, 2004]. Propositional logic offers one such example, in which truth assignments correspond to models. As an example, consider the 3-color graph coloring problem for the graph in Figure 2.1. To find solutions to the problem, we find all models which satisfy a given theory. To construct a propositional theory we begin by populating the fact base. First, we define the graph's four vertices: 1,2,3, and 4

1. vertex(1)
2. vertex(2)
3. vertex(3)
4. vertex(4)

Next, we define the edges connecting the vertices of the graph

1. edge(1,2)
2. edge(2,3)
3. edge(3,4)

Finally, we define the three colors: red, blue and green.

1. color(red)
2. color(blue)
3. color(green)

Having defined the fact base, we now define the constraints of the problem. First, we guarantee that each vertex, 1-4, is assigned at least one color

1. $colored(1, red) \vee colored(1, blue) \vee colored(1, green)$
2. $colored(2, red) \vee colored(2, blue) \vee colored(2, green)$
3. $colored(3, red) \vee colored(3, blue) \vee colored(3, green)$
4. $colored(4, red) \vee colored(4, blue) \vee colored(4, green)$

Next, we ensure that each vertex is assigned at most one color

1. $colored(1, blue) \wedge colored(1, green) \rightarrow \perp$

2. $colored(1, green) \wedge colored(1, red) \rightarrow \perp$
3. $colored(1, red) \wedge colored(1, blue) \rightarrow \perp$
4. $colored(2, blue) \wedge colored(2, green) \rightarrow \perp$
5. $colored(2, green) \wedge colored(2, red) \rightarrow \perp$
6. $colored(2, red) \wedge colored(2, blue) \rightarrow \perp$
7. $colored(3, blue) \wedge colored(3, green) \rightarrow \perp$
8. $colored(3, green) \wedge colored(3, red) \rightarrow \perp$
9. $colored(3, red) \wedge colored(3, blue) \rightarrow \perp$
10. $colored(4, blue) \wedge colored(4, green) \rightarrow \perp$
11. $colored(4, green) \wedge colored(4, red) \rightarrow \perp$
12. $colored(4, red) \wedge colored(4, blue) \rightarrow \perp$

Finally, we make sure that if two vertices are connected by an edge, then the two vertices do not have the same color assignment.

1. $colored(1, red) \wedge colored(3, red) \rightarrow \perp$
2. $colored(1, blue) \wedge colored(3, blue) \rightarrow \perp$
3. $colored(1, green) \wedge colored(3, green) \rightarrow \perp$
4. $colored(2, red) \wedge colored(4, red) \rightarrow \perp$
5. $colored(2, blue) \wedge colored(4, blue) \rightarrow \perp$
6. $colored(2, green) \wedge colored(4, green) \rightarrow \perp$

To find solutions to the specific theory we find all models which represent a valid coloring scheme for the problem instance. Given below are the 24 answer sets which yield valid color schemes.

1. $colored(1, g) \wedge colored(2, b) \wedge colored(3, g) \wedge colored(4, b)$
2. $colored(1, g) \wedge colored(2, b) \wedge colored(3, g) \wedge colored(4, r)$
3. $colored(1, r) \wedge colored(2, b) \wedge colored(3, g) \wedge colored(4, b)$
4. $colored(1, r) \wedge colored(2, b) \wedge colored(3, g) \wedge colored(4, r)$
5. $colored(1, g) \wedge colored(2, b) \wedge colored(3, r) \wedge colored(4, b)$
6. $colored(1, g) \wedge colored(2, b) \wedge colored(3, r) \wedge colored(4, g)$
7. $colored(1, r) \wedge colored(2, b) \wedge colored(3, r) \wedge colored(4, b)$
8. $colored(1, r) \wedge colored(2, b) \wedge colored(3, r) \wedge colored(4, g)$
9. $colored(1, b) \wedge colored(2, g) \wedge colored(3, b) \wedge colored(4, g)$
10. $colored(1, b) \wedge colored(2, g) \wedge colored(3, b) \wedge colored(4, r)$
11. $colored(1, r) \wedge colored(2, g) \wedge colored(3, b) \wedge colored(4, g)$
12. $colored(1, r) \wedge colored(2, g) \wedge colored(3, b) \wedge colored(4, r)$
13. $colored(1, b) \wedge colored(2, g) \wedge colored(3, r) \wedge colored(4, b)$
14. $colored(1, b) \wedge colored(2, g) \wedge colored(3, r) \wedge colored(4, g)$
15. $colored(1, r) \wedge colored(2, g) \wedge colored(3, r) \wedge colored(4, b)$
16. $colored(1, r) \wedge colored(2, g) \wedge colored(3, r) \wedge colored(4, g)$
17. $colored(1, b) \wedge colored(2, r) \wedge colored(3, b) \wedge colored(4, g)$
18. $colored(1, b) \wedge colored(2, r) \wedge colored(3, b) \wedge colored(4, r)$
19. $colored(1, g) \wedge colored(2, r) \wedge colored(3, b) \wedge colored(4, g)$
20. $colored(1, g) \wedge colored(2, r) \wedge colored(3, b) \wedge colored(4, r)$
21. $colored(1, b) \wedge colored(2, r) \wedge colored(3, g) \wedge colored(4, b)$
22. $colored(1, b) \wedge colored(2, r) \wedge colored(3, g) \wedge colored(4, r)$
23. $colored(1, g) \wedge colored(2, r) \wedge colored(3, g) \wedge colored(4, b)$

24. $colored(1, g) \wedge colored(2, r) \wedge colored(3, g) \wedge colored(4, r)$

The above encoding illustrates a problem in using propositional logic as an ASP formalism. For a given search problem Π, a specialized program, P_Π must be created for each instance of Π. That is, the program theory is tied to each specific instance. As such, the use of propositional logic and accompanying SAT solvers are limited in their applicability as a *general-purpose* tool [East et al., 2004].

The above propositional theory may be generalized to extend the theory allowing a representation of k-colorability. Let graph $G = (V, E)$ be an undirected graph, where V is a set of vertices and E is a set of edges. Further, let C be a set of colors, where color $c_i \in C$. Let A be the set of color assignments defined as propositional predicate $colored(v, c) \in A$, with the meaning that vertex v is assigned the color c. The theory has three clauses

1. $\top \rightarrow colored(v_i, c_i) \vee ... \vee colored(v_i, c_k), v_i \in V, c_i \in C for 1 \leq i \leq k$
2. $colored(v_i, c_i) \wedge colored(v_i, c_j) \rightarrow \bot, v_i \in V, c_i, c_j \in C, i \neq j$
3. $colored(v_i, c) \wedge colored(v_j, c) \rightarrow \bot, (v_i, v_j) \in E, c \in C$

This generalization offers the benefits that the constraints of the problem are separated from a particular problem instance. It also show us that, in order for a formalism to support the separation of constraints and data, the formalism must allow constraints to be described without respect to a specific data instance.

One such formalism which supports the separation of data and constraints is the stable logic formalism [Gelfond and Lifschitz, 1988]. The stable logic formalism is a declarative logic semantics which allows negation and whose concept of model is that

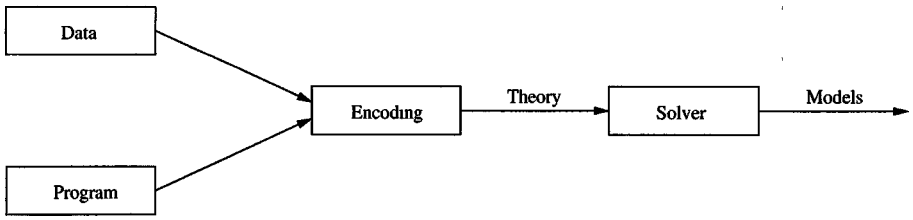

Figure 2.2: Data-constraint separation

of a *minimal* model, i.e. a proper subset of a stable model is not itself a stable model. Note that this differs from the concept of model in propositional logic. Let graph $G = (V, E)$ be an undirected graph where V is a set of vertices and E is a set of edges. Further, let C be a set of colors, where color $c_i \in C$. Let A be the set of color assignments. We then define a propositional predicate $colored(v, c) \in A$, such that vertex v is assigned the color c. We further define $diffcolored(v, c) \in A$.

1. $colored(X, A) \leftarrow diffcolored(X, A) \wedge vertex(X) \wedge color(A)$.
2. $diffcolored(X, A) \leftarrow colored(X, B) \wedge A \neq B \wedge vertex(X) \wedge color(A) \wedge color(B)$.
3. $\bot \leftarrow colored(X, C) \wedge colored(Y, C) \wedge edge(X, Y) \wedge color(C)$.

The above three clauses express the required constraints for the k-coloring problem using the stable logic formalism. The first clause assigns to vertex X a color A only if not previous assigned a different color. This clause, together with the second clause, guarantee only one color assignment for each vertex. The third clause prevents connected vertices from begin assigned the same color.

The separation of data and program (Figure 2.2) allows a single program to be encoded which is capable of solving different problem instances, thus allowing for a general solver mechanism. Fundamental to this concept of generalization through

separation using a given formulation is *uniformly* solving a search problem. A given ASP formalism is said to *uniformly* solve search problems if, for a search problem Π there is

1. an effective encoding, E_{D_n}, for $I \in D_\Pi$ in the language of F and
2. a finite program T_Π defining Π

which yields a one-to-one polynomial time mapping from the answer sets of $T_\Pi \cup E_{D_\Pi}(I)$ to the solutions of Π [Garey and Johnson, 1979].

This chapter discusses the basic concepts of answer-set programming. The next chapter presents the logic PS^+ in the context of the answer-set programming paradigm.

CHAPTER 3

LOGIC PS AND ITS EXTENSION PS^+

In chapter 2 we discussed the basic concepts of the answer-set programming paradigm. Recently [East and Truszczyński, 2001b], the logic PS and its extension PS^+ have been proposed which offers an alternative ASP formalism. The logic of PS is a modification of the logic propositional schemata. A primary concept in the logic PS is the data-program pair (D, P). This concept is based upon the separation of a problem representation and a particular instance of the problem, pursuant to the idea of a *generalized* computational mechanism. This chapter provides an overview of the logic PS and its extension PS^+.

3.1 Syntax

The syntax of logic PS is a subset of first-order logic without function symbols [East and Truszczyński, 2001b]. As such, the logic PS consists of the following

1. infinite denumerable sets R, C, and V of relation, constant, and variable symbols, respectively.
2. symbols $\bot$ and $\top$, interpreted as false and true, respectively.
3. basic logic connectives $\wedge$ (conjunction), $\vee$ (disjunction), and $\rightarrow$ (implication)
4. quantifiers $\exists$ (existential) and $\forall$ (universal)

5. punctuation symbols '(', ')' and ','.

Constants and variables are the only terms, and constants are the only ground terms. Expressions of the form

$$p(t_1, t_2, \ldots, t_n)$$

are *atoms*, where p is an n-ary relation symbol from R, and t_i, $1 \leq i \leq n$ are terms. An atom is said to be grounded if all of terms belonging to the atom are ground.

The use of existential quantifiers is restricted in the logic PS. Consider the expression

$$\exists X_1, X_2, \ldots, X_k \; p(t_1, t_2, \ldots, t_n)$$

where $(t_1, t_2, \ldots, t_n)$ is a set of terms, and $X_1, X_2, \ldots, X_k$ are distinct variables appearing in the set $(t_1, t_2, \ldots, t_n)$ exactly once. Such an expression is an existential atom, or *e-atom.* Existential quantifiers are restricted exclusively to the syntax of e-atoms. The notation of the e-atom

$$\exists X_1, X_2, \ldots, X_k \; p(t_1, t_2, \ldots, t_n)$$

takes a simplified syntax of the form

$$p(t_1', t_2', \ldots, t_n')$$

where $t_i' = t_i$ if t_i is not within the variable set $X_1, X_2, \ldots, X_k$, and $t_i =$ '_' otherwise. For example, the e-atom $\exists X \; p(X, Y)$ is simplified to $p(\_, Y)$.

In the logic PS, *rules* are also referred to as *implications* and are the only allowed formulas. Each rule takes the form

$$\forall X_1, X_2, \ldots, X_k (A_1 \wedge A_2 \wedge \ldots, A_m \rightarrow B_1 \vee B_2 \vee \ldots \vee B_n)$$

where $A_1, A_2, \ldots, A_m$ are atoms and $B_1, B_2, \ldots, B_n$ are either atoms or e-atoms, and $X_1, X_2, \ldots, X_k$ are variables in $A_1, A_2, \ldots, A_m$ and $B_1, B_2, \ldots, B_n$. Either m and n may be zero. If $m = 0$, then the symbol $\top$ replaces the absent conjunct in the antecedent. Likewise, if $n = 0$, then the symbol $\perp$ is used to replace the absent disjunct in the consequence. A collection of rules that contain at least one constant symbol is a *theory*.

3.2 Semantics

In the logic PS, a theory T is considered a representation of the Herbrand class of models of T. This is distinct from the first-order logic interpretation of theories as logical consequences of T. Before discussing this class of models used by logic PS, let us review some preliminary details (these may be found in any logic text).

A sentence is said to be *universal* if it takes the form

$$\forall x_1, \forall x_2, \ldots, \forall x_k \varphi$$

so that φ is a formula in the language without quantifiers and where all variables in φ are among $x_1, x_2, ..., x_k$. A set of universal sentences is a *universal theory*. Given a theory T, a *Herbrand universe* of T is the collection of ground terms that can be constructed from the constraints and function symbols of T. For example, consider the following theory T

1. $p(a)$

2. $p(X) \rightarrow p(Y)$

The Herbrand universe would be $\{a\}$, as 'a' is the only constant symbol. The *Herbrand base* is the collection of ground atoms that can be constructed from the ground terms and predicates of T. For the above theory T, the Herbrand universe is $\{p(a)\}$. The *Herbrand models* of T is any subset of the Herbrand base of T. Continuing with our example, the Herbrand model is $\{p(a)\}$.

An important feature of the logic PS is that it is nonmonotonic. For example, let theory $T_1 =$

1. $\top \rightarrow p(a)$
2. $q(a) \rightarrow \bot$
3. $p(a) \rightarrow q(\_)$

and let theory $T_2 = T_1 \cup \{\top \rightarrow p(b)\}$. For T_1, no Herbrand models exist because the Herbrand universe $\{a\}$ does not satisfy the third rule in T_1 for the standard interpretation of implication. T_2, however, does have a Herbrand model, $\{p(a), p(b)\}$, which is unaffected by T_1. For further discussion, see [East and Truszczyński, 2004].

3.3 Computing Models

Computing models in the logic PS requires two steps. Given a PS-theory T, we first ground T to a grounded propositional theory T_{grnd}. By propositional grounding, it is meant the following. Given a theory T, for each rule r, $r \in T$, in the form

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \rightarrow B_1 \vee B_2 \vee \ldots \vee B_n$$

we define $r\delta$ to be

$$A_1\delta \wedge A_2\delta \wedge \ldots \wedge A_m\delta \rightarrow B_1 \vee B_2\delta \vee \ldots \vee B_n\delta$$

where all atoms $A_i, 1 \leq i \leq m$ are not e-atoms. We then define the ground theory T_{grnd} to include all rules $r\delta$, where $r \in T$ and δ is the ground substitution containing all variables in r [East and Truszczyński, 2004]. Second, we compute models of T by computing models for T_{grnd}.

A data-program pair is defined as the pair (D, P), where D is a finite set of ground atoms resenting an encoding of input data, and P is a finite collection of rules serving as a declarative specification of the problem. This pairing of data and rules together represents a specific problem instance. The term *data predicate* is used for all relation symbols in atoms of D. Likewise, we use the term *program predicate* to refer to all relation symbols appearing in P.

3.4 The Logic PS^+

From the viewpoint of a programming formalism, the logic PS is limited. To enhance the effectiveness of the logic PS, it has been extended to support higher-level constraints, such as cardinality. Building upon the logic PS, this extension is called PS^+.

By a propositional cardinality atom, it is meant

$$m\{p_1, p_2, ..., p_k\}n$$

where both m and n are non-negative numbers, representing the lower and upper bound of the constraint, respectively. Either m or n, but not both, may be absent.

3.5 Processing models of PS^+ theories

Similar to the two-step process of computing models with stable logic programming (see chapter 2), obtaining solutions from PS^+-theories is a two-step procedure. Given a data instance and program pair T, the first step is to ground T to obtain a propositional theory, T_{grnd}. The ground theory T_{grnd} is then used to compute models of T.

Grounding in logic of PS^+ is similar to the concept of grounding in first-order logic. Consider the following. A sentence is said to be *universal* if it takes the form

$$\forall x_1, \forall x_2, ..., \forall x_k \varphi$$

so that φ is a formula in the language without quantifiers and where all variables in φ are among $x_1, x_2, ..., x_k$. A set of universal sentences is a *universal theory.* The semantics for universal theories can be obtained by *lifting* propositional semantics to the general predicate case through grounding. Let φ be a universal sentence of the form

$$\forall x_1, \forall x_2, ..., \forall x_k \varphi$$

where φ is a formula without quantifiers. By propositional ground of φ, it is meant a collection of all formulas

$$\varphi(x_1/t_1, ..., x_k/t_k),$$

where $t_1, ..., t_k$ range over all ground terms in the Herbrand universe and

$$x_i/t_i$$

is the substitution of a ground term t_i for variable x_i. For a universal theory T, the propositional ground of T is defined as the union of the propositional ground of all universal sentences in T. Consider the following theory, T,

1. $p(a, b) \rightarrow q(c)$
2. $q(X) \rightarrow p(\_, X)$

Clause (1) is already grounded. It contains no variables but does contains three constants: a, b, and c. Clause (2) contains the variable X. For clause (2), the three constants in clause (1) will result in three ground substitutions, S, where $S_1 = \mathrm{X}/a$. $S_2 = \mathrm{X}/b$. $S_3 = \mathrm{X}/c$. Accordingly, we have the following grounded theory, T_{grnd}.

1. $p(a, b) \rightarrow q(c)$
2. $q(a) \rightarrow p(a, a) \vee p(b, a) \vee p(c, a)$
3. $q(b) \rightarrow p(a, b) \vee p(b, b) \vee p(c, b)$
4. $q(c) \rightarrow p(a, c) \vee p(b, c) \vee p(c, c)$

3.6 Theory Reduction

While the ground theory for the above theory is small, the size of a grounded theory for any non-trivial problem can be prohibitively large in terms of computational requirements. Reduction of the ground theory may be obtained by noting that any propositional theory that has the same models of its grounding may be used in its place [East and Truszczyński, 2004]. Furthermore, truth assignments of ground atoms built upon predefined relation symbols may be evaluated during grounding. As such, theory reduction may be obtained in the following cases:

1. If A appears in the consequence of the clause and is true, the clause is removed.
2. If A appears in the consequence of the clause and is false, A is removed from the consequence.
3. If A appears in the antecedent of a clause and is true, A is removed from the body.
4. If A appears in the body of the clause and is false, the clause is removed.

During grounding, these reductions may further yield truth assignments, and the process is repeated. Consider the case that a rule consists of a single atom. If this atom appears in the consequence of the rule, the atom must be assigned true and the clause is removed. If the atom appears in the rule body, it must be assigned false and the clause is removed. Now consider the case of a forced cardinality atom of the form

$$m\{p_1, p_2, \ldots, p_k\}n$$

If m atoms has been assigned true, then all remaining unassigned atoms must be assigned false. This process is continued while new atoms with forced truth assignments are found. The result of the completed process is referred to as the *ground core.*

Once a grounded theory is constructed, the second step in computing models is to search for models of the theory. Models of T_{grnd} are obtained by using a propositional solver. However, the size of theories can be huge, especially in the case of cardinality. In the logic PS^+, cardinality has direct representation resulting in a comparatively compact ground theory. These theories, however, which maintain the structure of cardinality atoms can not be processed by 'off-the-shelf' propositional solvers. As

such, the next chapter presents the *aspps* system, which is designed to process PS^+-theories and which can take advantage of direct cardinality constraints.

CHAPTER 4

THE ASPPS SYSTEM

The *aspps* system is an answer set programming system based on the logic PS^+ [East and Truszczynski, 2002]. This system consists of two modules. The **psgrnd** module is used to compute theories given a rule and a set of data files. The module **aspps** computes models for the grounded theory produced by **psgrnd**. As discussed in chapter 3, a PS^+ theory consists of a data-program pair (D, P). D is a set of ground atoms representing an instance of the problem, and P is a set of PS^+ clauses representing the constraints of the problem. This chapter gives an overview of the *aspps* system.

4.1 Data File Syntax

An atom statement is terminated by a single '.' character. Comments may be included by a '%' character delimiter, and continue until the end-of-line. Blank lines are ignored. As an example, consider the k-coloring problem. Given a graph G and an integer k, can we color the graph with k colors in such a way that no two vertices are connected by an edge of the same color.For a 3-coloring problem, a possible data file is given below (line numbers are included only to aid readibility)

```
1. color(red).
2. color(blue).
3. color(green).
```

```
4.  vertex(1).
5.  vertex(2).
6.  vertex(3).
7.  vertex(4).

8.  edge(1,3).
9.  edge(2,3).
10. edge(4,1).
```

Lines 1-3 defines three available colors: r (red), b (blue), and g (green). Lines 4-7 defines the four vertices of the input graph. Lastly, lines 8-10 specifies the edges of the input graph. As a syntactic convenience, data predicates may also be defined by unary predicates. For example, the vertex predicates in the example above may also be written

```
vertex(1..4)
```

The *aspps* system also allows constants to be defined at grounding. As such, the color data predicate may be defined

```
color(1..k)
```

where k is a constant defined on the command-line of **psgrnd**. Each occurrence of the constant k is replaced by the value specified on the command-line.

4.2 Rule File

A rule file contains the constraints of the problem instance. Only one rule file is permitted for each instance. Each rule file consists of two parts. The first part is the preamble. In the preamble, program predicates and variables are declared

which restrict the way the program predicates are grounded. A program predicate is declared using the *pred* keyword and take the following form

pred $name(type_1, type_2, ..., type_k)$.

where **pred** is the predicate keyword, *name* is the program predicate and $type_i$ are data predicates .

Variable declaration use the *var* keyword and takes the form

var type $name_1, name_2, \ldots, name_n$.

where **var** is the variable declaration keyword, *type* is the data predicate, and $name_i$ is a variable name. A complete preamble for a color-ability program is (line numbers are added for readability)

```
1. pred colored(vertex,color).
2. var vertex X,Y.
3. var color K,C.
```

where *colored* is a program predicates, and *vertex* and *color* are data predicates.

Following the program preamble are the clauses which define the constraints to the problem instance. Note that conjunction is written using the ',', disjunction with the '|', and implication with the '->'. Continuing with the graph 3-coloring example, a possible set of constants may be

```
4. colored(X,red) | colored(X,green) | colored(X,blue).
5. colored(X,K), colored(X,C), K != C ->.
6  colored(X,K), colored(Y,K), edge(X,Y) ->.
```

Line 4 states that each vertex, X, is assigned at least one color, *red*, *blue*, or *green*. Line 5 states that a contradiction is reached if a vertex, X, is assigned two different

colors. Finally, line 6 states that a contradiction is reached if the same color is assigned to two vertices connected by an edge.

As defined above, the rule file, while correctly representing the problem, does not take advantage of some of the built-in constructs of PS^+. As discussed in chapter 3, the logic of PS^+ introduced the notion of the existential-atom. The disjunction in line 4, in the above example, could be written

```
colored(X,_).
```

In addition to the use of e-atoms, the modeling concept of the cardinality atom, which has a direct representation in the *aspps* system, can be used to rewrite lines 5-6 as

```
1{colored(X,_)}1.
```

In addition, the *aspps* system allows for several *predefined* predicates and function. Available predicates symbols include the equality operator ==, the arithmetic comparators <=, >=, < and >, and the arithmetic operations +, −, ∗ ,/. Available function symbols include *abs*() (absolute value), $mod(N, b)$, $max(X, Y)$ and $min(X, Y)$. All of these symbols are assigned their standard interpretation.

4.3 Processing Theories

With the data and program files, the grounded program instance is generated using the **psgrnd** module. During the grounding process, the following tasks are performed. First, all predefined operators are evaluated. Second, all program clauses are instantiated. Third, both atoms built upon predefined predicates and data predicates are

simplified away. The result is a grounded theory consisting of only ground atoms built from program predicates. The output is similar to that of DIMACS CNF format, with the following deviations

1. The output file begins with a program header.

2. Clauses are terminated by the newline character rather than a '0'.

3. C-atoms and Horn clauses are preserved using special notation.

Cardinality atoms are directly represented in the following form

$$\{\ l\ u\ int_1,\ int_2,\ \ldots,\ int_n\ \}$$

where l and u represent the lower and upper bound, respectively, of the constraint and $int_1, int_2, \ldots, int_n$ are the integer representations of constituent ground atoms.

The required input to execute **psgrnd** is a single program file, one or more data files and optional constants passed on the command line. If no errors occurs while reading the files and during the grounding process, a machine readable file is constructed. An example ground file appears in figure 4.3. Line 1 is the program header, indicating the number of atoms, cardinality atoms, and clauses, respectively. Lines 2-5 are the direct representation of the cardinality atoms representing the single color assignment for each vertex. Lines 6-14 are clauses where the values are the numerical representation of the grounded atoms found in lines 15-26.

Once the grounded theory has been constructed, the *aspps* system uses a modified Davis-Putnam [Davis and Putnam, 1960] algorithm to perform search.

```
1.  p 12 4 13
2.  ,  { 1 1  1  2  3  }
3.  ,  { 1 1  4  5  6  }
4.  ,  { 1 1  7  8  9  }
5.  ,  { 1 1  10  11  12  }
6.  1  4  ,
7.  2  5  ,
8.  3  6  ,
9.  4  7  ,
10. 5  8  ,
11. 6  9  ,
12. 7  10  ,
13. 8  11  ,
14. 9  12  ,
15. c 1 colored(1,r)
16. c 2 colored(1,b)
17. c 3 colored(1,g)
18. c 4 colored(2,r)
19. c 5 colored(2,b)
20. c 6 colored(2,g)
21. c 7 colored(3,r)
22. c 8 colored(3,b)
23. c 9 colored(3,g)
24. c 10 colored(4,r)
25. c 11 colored(4,b)
26. c 12 colored(4,g)
```

Figure 4.1: Sample data file from the psgrnd module.

The pseudo-code for the DPLL is given in Figure 4.2. This algorithm is based upon unit propagation and case splitting. Input to the algorithm is a theory T and a list of partial assignments A. Each time an unassigned variable is assigned a truth value the program is divided into two independent sub-problems. This method is the basis of the *search-space partitioning* mechanism employed by the majority of distributed solvers (see chapter 5).

An important detail of any DPLL implementation is the heuristics implemented to select split points. The branching heuristic currently used by *aspps* is as follows

```
DPLL(T,A)
   T' = UnitPropagate(T,A)
   if Satisfied(T') then
      return satisfied
   else if Contradiction(T') then
      return contradiction
   p = ChooseUnassignedVariables(A)
   A' = AssignTrue(p)
   if DPLL(T',A') = satisfied then
      return satisfied
   else
      A' = AssignFalse(p)
   return DPLL(T',A')
```

Figure 4.2: Davis-Putnam-Logemann-Loveland Algorithm

[East and Truszczyński, 2004]. For each rule r, a weight is defined

$$W(r) = k^{m-l}$$

where k is a constant (currently 13), m is the minimum(L,10), where L is the maximum length of a rule, and l is the length of the rule r. The weight of an atom is the sum of the weights of all rules in which it appears. If the atom is a cardinality atom, the weight is the sum of the weights of unassigned atoms which appear in it.

When looking for a way to branch, the system considers all propositional atoms that are currently unassigned. It also considers c-atoms that have been forced by earlier assignments. If there are multiple c-atoms satisfying the conditions, it will select one with the greatest weight. It will then generate all truth assignments to unassigned atoms that are consistent with a true assignment, and will use these assignments to split the search space. Otherwise, it will branch on an atom with the greatest weight and split the search space into two parts.

This chapter presents the sequential *aspps* solver. The following chapters discuss the details of incorporating a distributed framework using the *aspps* solver. The next chapter provides a survey of currently available distributed solvers.

CHAPTER 5

EXISTING DISTRIBUTED SAT SOLVERS

Recent research into the integration of parallel/distributed paradigms with SAT solvers has resulted in several successful systems. With few exceptions, these distributed solvers take an existing sequential SAT solver as a solver core, and incorporate various parallel/distributed mechanisms. This chapter concerns itself with *distributed* SAT solver systems, that is, systems which aggregate independent computer systems into a single process image. Additionally, it is concerned with *complete* distributed solvers, i.e. those which employ core solver systems which are guaranteed to find an instance of satisfiability if the problem is satisfiable, or to terminate indicating the problem is unsatisfiable. The following is a brief overview of those complete, distributed SAT solvers, which are readily available.

In discussing the various features of existing distributed solvers it is necessary to clarify the terminology that will be used. The term *node*, *resource*, *processor* all refer to a single computer system, with the understanding that the system is accessible over a network. The term *node* is used for its clarity. The term *master node* is used to refer to the system which initiates a search problem. The term *client node* is used to refer to a single system acting on behalf of the master node. These two terms combined form what is referred to as the *master-client* paradigm, also referred to as the *master-slave* paradigm. Note that while some solvers may possess properties that

resemble a peer-to-peer architecture, all communication models considered in regards to the master-client paradigm. This is due to the fact that while these peer-to-peer qualities may exist, the essential roles played by both master and client nodes remain those of the traditional understanding of the master-client paradigm.

5.1 GridSAT

GridSAT [Chrabakh and Wolski, 2003b] is a complete distributed solver based on the zChaff [Moskewicz et al., 2001] sequential SAT solver. It is the successor of the GradSAT distributed solver. GridSAT is designed and implemented for Computational Grid environments. Source code for the GridSAT system, or that of the GradSAT system, is not publicly available at this time.

The GridSAT system employs a master-client communication model. Communication between nodes is facilitated by the EveryWare [Wolski et al., 1999] message system.

The master node performs no search space processing, but rather is responsible for the following tasks

Resource management: The master node is responsible for selection and monitoring its set of available resources. Such information is gathered from the Grid information system (globus mds or nws).

Client management: The master node is also responsible for monitoring the state of each client node. This allows the dynamic inclusion of client nodes during runtime.

Resource Scheduling: It is the responsibility of the master node to assign a given sub-theory to a the highest ranked client node within the its resource pool. This includes only the selection of the client node; as will be discussed below, requests for additional resources is a task assigned to the each participating client node.

When the master process is invoked, it first queries the resource discovery subsystem for a list of all available clients. Each discovered resource is initialized without data, i.e., set to an idle state. Once initialized, each client registers with the master node to indicate resource availability. The master node's list of registered clients is subsequently prioritized according to its resource evaluation and ranking gathered from its Grid information system, if configured, or from a pre-compiled, static index.

The first registered client is sent the entire theory by the master node. This client will process the assigned search space until it detects that the threshold of available resources is about to be reached. At this point, the client sends the master node a request to split the search space. The master node, in turn, searches among registered client pool for the highest ranked idle node. This newly selected node is then initialized and begins direct communication with the original client node which requested the split. These two client nodes proceed to divide the search space and receive relevant shared clauses pertaining the the theory. Note that the master node acts only as temporary intermediary for resource selection.

A similar exchange is performed during load-balancing, that is, the dynamic evaluation and balancing of resource utilization among a resource pool. If a client node

exceeds it allocated resources, determined by either memory usage or processing time, it will send the master node a request to split the search space, and proceed as outlined above.

One additional communication exchange is necessary. Because GridSAT uses zChaff as its core solver system, it itself must handle the necessary exchange of learned clauses among registered client nodes. The communication and exchange of these newly learned clauses is performed by the client nodes. As learned clauses are generated, the generating client node sends the other nodes its list. These newly received clauses are merged into the client node's local clause database only after the algorithm has backtracked to the first decision level of its search space. This is a significant necessity due to the additional communication overhead.

This process of search-space splitting and learned-clause exchange continues for the duration of the problem processing as monitored by the master node. The following four cases will cause the master node to terminate:

1. All registered clients are idle. Because of the load-balancing mechanism discussed above, an idle client pool signifies that the problem instance is unsatisfiable.

2. A registered client finds a solution. If a registered client finds a solution within its search space, the client will send the master node the solution. Once received, the master will verify that the solution satisfies the problem.

3. The master node times out. A time threshold may be placed on the master node. This mechanism is not described in detail in the available literature.

4. A registered client node exceeds available resources or becomes unavailable due to a failure in network connectively, etc.

The first 3 cases are standard among distributed solvers. The last case, however, is illustrative of GridSAT's lack of fault-tolerance. Consequently, failure of a registered client node results in failure of the master node.

5.2 Parallel Satz

Another complete distributed SAT solver is Parallel Satz [Jurkowiak et al., 2001], based upon the sequential solver Satz [Li and Anbulagan, 1995]. Parallel Satz is designed for distributed problem solving within a clustered environment, i.e. a tightly coupled pool of independent systems on a local-area network. Source code for Parallel Satz is publicly available at `http://www.laria.u-picardie.fr/%7Ecli/ParaSatz.tar.gz`.

A simple master-slave communication model is used, employing Remote Procedure Call (RPC) as a message passing framework. Process invocation and termination is facilitated using the standard Unix Berkeley Remote Shell (RSH) protocol.

Working within this master-client paradigm, the master node is responsible for the following tasks.

1. Context exchanges between slave nodes.

2. Context storage and reassignment.

3. Slave node work load evaluation.

4. Begin/Halt a problem resolution.

Likewise, tasks assigned to participating slave nodes include

1. Context exchanges with master node.

2. Search space construction from context assignment.

3. Context solicitation to master node.

4. Solution submission to master node.

All work is begun on the master node. Once all client nodes are initialized, the master node will arbitrarily choose a slave node to begin resolution. All other client nodes are idle. These idle client nodes will then send a work request to the master. For each work request received, the master node evaluates the load of all client nodes by requesting the position of their first and second remaining subtrees. The master will send the first remaining subtree of the most heavily loaded slave to the slave requesting work.

This form of load balancing is similar to the mechanism employed by the GridSAT solver in that it is initiated by the client nodes. It differs from in the following regards. Client nodes under Parallel Satz send requests when when they are idle; such a structure insures that all well-known client nodes are actively performing a search. It does not, however, offer load-balancing in the sense of offloading excessive resource utilization due to problem at hand. In addition, communication is always performed master-to-client. No communication occurs between client nodes.

The master node will halt on the following cases:

1. All clients are idle

2. A solution is found

Similar to the GridSAT solver, the load-balancing mechanism of Parallel Satz insures that if all client nodes are in an idle state, then there is no more processing to be done, indicating unsatisfiability of the theory. If a solution is found by a client node, the master node will request all runtime information from the client nodes, and subsequently halt all client nodes and itself. Unlike GridSAT, there is no timeout facility available to the master node. The master will simple proceed until it is sent a solution, halt indicating no solution is found, or is explicitly killed. Secondly, note that client node failure does not terminate the master process, unlike GridSAT, indicating a degree of fault-tolerance in Parallel Satz.

In fact, the matter of fault-tolerance is simplified in Parallel Satz, due to the fact that its core solver, Satz, does not employ any look-back techniques, such as learning. As such, the search space may be split into independent search spaces with no common clause database. If a client node becomes inaccessible during the processing of a search space, the master sends the failed node's most recent search context to an idle slave node. The search re-assignment is performed during load balancing limiting communication overhead.

5.3 NAGSAT

Both Parallel Satz and GradSAT employ search space partitioning to facilitate distributed data process. This is not the only mechanism available to incorporate a

distributed framework with a sequent solver. A different approach is taken by the distributed solver NAGSAT [Forman and Segre, 2002]. While NAGSAT is limited to 3-SAT problems, it has some unique features which makes it noteworthy.

As defined in [Segre et al., 2002] and [Paarsch and Segre, 1999], nagging is a general-purpose, asynchronous, parallel search technique where a single master node, or processor, performs a standard DPLL search procedure. This search process is augmented by one or more client nodes, or naggers, each of which performs an identical search procedures on perturbed search spaces. While NAGSAT performs a standard sequential search, unlike Parallel Satz and GradSAT, it is not based on an existing sequential solver, but rather has its own DPLL procedure.

The process of nagging is by definition a distributed task. Nagging begins with a single master processor alerting all available naggers to its own work on a theory. The master node is responsible for the following tasks

1. Perform the primary DPLL search

2. Provide search state information to requesting nagger clients.

3. Incorporate search results of nagger clients into its search.

Upon invocation of its own search, the master node will initialize its client nodes by sending the complete, original theory to each client node. Once initialized, a nagger node will request a nag-point from the master. This begins the nagging episode exchange. Once the nag-point is received, the nagger process transforms, using various techniques, the search space into a perturbed, but semantically equivalent, search

space. This nagging process can be viewed as a race: the nagger processors try to beat the master to a solution. This competitive cooperation between master and nagger processes can yield the following three outcomes.

ABORT: If the master process backtracks over a previously assigned nag-point, the master process will signal the assigned nagger to ABORT the nagging episode. Having released the nagging episode, the nagger process enters an idle state and subsequently requests a new nag-point from the master process.

PRUNE: Conversely, if a nagger process completes its search of its subtree, and finds no solution, the nagger process will signal the master process to backtrack to the nag-point, reducing the master's search space.

SOLVE: A nagger finds a true assignment. The nagger terminates its own search and reports the solution to the master, itself subsequently terminating the search.

The technique of nagging has several inherent benefits that can be exploited by a distributed framework. First, nagging does not require any explicit load balancing mechanisms to be implemented. Second, nagging is, by its definition, fault-tolerant. A nagger process simply performs an extension of the master processor's search. Consequently, a nagging process that becomes unavailable will not affect the master's search. This is subject, of course, to the obvious condition of failure of the master process.

This chapter provides a survey of complete, distributed SAT solvers which are currently available. Several problem areas become apparent in the design and imple-

mentation of the *daspps* system. First, the *daspps* system is to employ the technique of search space partitioning to segment a problems search space into independent segments. As such, an explicit load balancing system much be developed. Second, because each sub-theory within the search space is constructed independently of the others, fault-tolerance is an a relatively simple matter to address. However, if a client node becomes unavailable, we do not want to lose the results of the assigned search space. As such, a simple redundancy mechanism must be put into place. Third, communication between nodes must be as minimal as possible. These three design issues are addressed in the succeeding chapters.

CHAPTER 6

SEARCH SPACE PARTITIONING

The primary task of employing a distributed framework within an existing sequential solver is the development of an efficient segmentation mechanism. The *daspps* system employs a distributed search technique where the search space is partitioned into independent sub-theories. Each of these sub-theories is then independently processed. This technique is called *search-space partitioning* and is used by the majority of existing solvers. The use of search-space partitioning within the *daspps* system differs significantly from its usage in other solvers (see chapter 5). Rather than initializing a client with the complete search problem, and split upon a specified criteria, the *daspps* system dynamically partitions the search space such that each client node receives only its assigned sub-theory. Currently, due to the difficulty of determining partition points, client nodes perform no further partitioning, i.e., the sub-theory is treated sequentially by the client. This chapter will discuss the distributed framework of the *daspps* as it pertains to search space partitioning.

As discussed in chapters 3 and 4, the *aspps* system benefits from the direct representation of cardinality atoms. In the *aspps* system, the grounded PS^+ theories maintain the structure of the cardinality atom. A grounded PS^+ theory T

$$T = (\zeta_1 \wedge \zeta_2 \wedge ... \wedge \zeta_n)$$

where each clause ζ_i has the form

$$\zeta_i = a_1 \wedge a_2 \wedge ... \wedge a_m \rightarrow b_1 \vee b_2 \vee ... \vee b_r (m, r \geq 0)$$

and each a_j and b_k is either a cardinality atom or grounded atom. A cardinality atom is a collection of grounded atoms such that if at least p and at most q are true then the c-atom is true. Otherwise, the c-atom is false. Therefore, a c-atom is assigned the value of true if the collection of ground atoms making up the c-atom have been assigned values such that the cardinality requirements cannot be satisfied than the c-atom is assigned the value of false. Thus the value of a c-atom can be determined by the assignment of values to the ground atoms making up the c-atom. If a c-atom is forced during propagation its cardinality requirements must be enforced. If the c-atom must be true then the collection of ground atoms not already assigned must be assigned values in such a way that the cardinality requirements are met. Likewise, if the c-atom must be false, then the collection of ground atoms must be assigned values which do not satisfy the constraints. A forced c-atom can be used to partition the search-space into multiple independent sub-theories.

As an example, consider the following c-atom

$$1\{\ a\ b\ c\ d\ e\ \}1$$

where the set {a,b,c,d,e} are ground atoms and the partial assignment of values consists of {a=false,b=false}. If the c-atom is forced to true, then the theory may be split into three sub-theories

a=false,b=false,c=true,d=false,e=false

a=false,b=false,c=false,d=true,e=false

a=false,b=false,c=false,d=false,e=true

Similarly, if the c-atom is forced to false, then we have all assignments which evaluate to false

a=false,b=false,c=false,d=false,e=false

a=false,b=false,c=true,d=true,e=false

a=false,b=false,c=true,d=true,e=true

a=false,b=true,c=true,d=true,e=true

a=true,b=true,c=true,d=true,e=true

⋮

This technique is used by *daspps* to split the theory into independent sub-theories. Currently, we use the ratio of assigned atoms/number of atoms to determine a split. If the ratio is 0.01 (the value determined through testing), then the current assignment stack is sent to a pending client node. Note that sub-theory partitioning is not, therefore, determined by the depth of the search, but by the percentage of atoms assigned. The actual depth of the search at which sub-theories are generated may vary greatly because atoms may be forced. Consider, for example, Figure 6.1. If we branch on the c-atom 1{ a b }1, we have the two partial assignment lists { $a = true$, $b = false$ } and { $a = false$, $b = true$ }. In the case of the assignment { $a = true$, $b = false$ }, the atoms c and d are forced (to satisfy the cardinality constraints). Atoms f and g are not forced, and we only have sub-theories following the assignments of f and g. On the other hand, in the case of the partial assignment

$\{\ a = false,\ b = true\ \}$, atoms c, d, f, and g are all forced (to satisfy the cardinality constraints), thus producing a sub-theory.

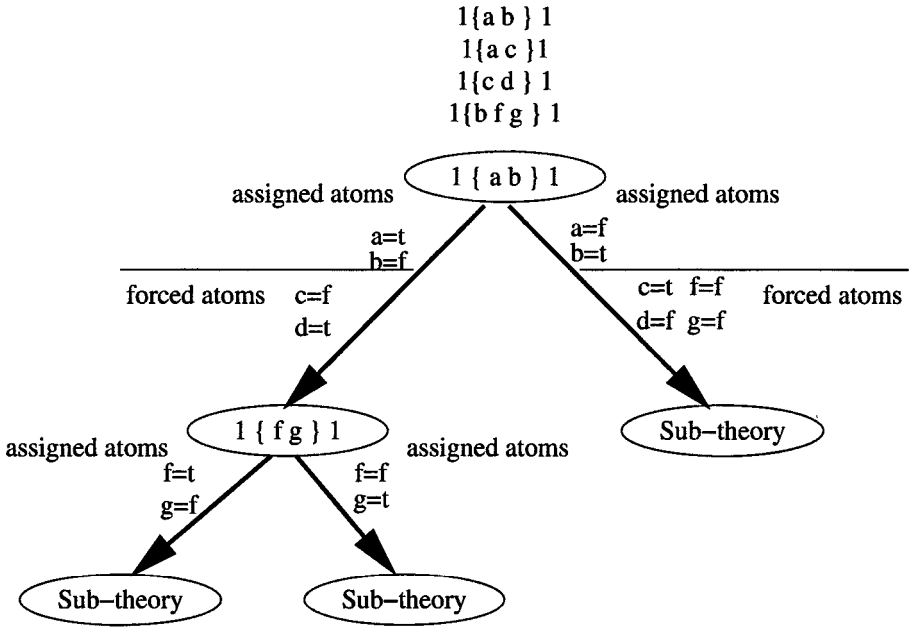

Figure 6.1: Partitioning is not dependent on depth of search.

Because each search space partition is independent, it is a relatively simple matter of assigning each sub-theory to an available client node for processing. Sub-theory assigned is performed as follows. Let N be a set of available clients, S be the set of sub-theories available to be processed, and A be the set of existing assignments. While S is not empty, for each available client we create the pair $a_n = (n_i, s_k)$, where $n_i \in N$, $s_k \in S$, and $a_n \in A$. Note that multiple clients may be assigned the same sub-theory, such that the pairs (n_i, s_k) and (n_j, s_k) exist. Assignments continue to be made until either a solution is returned from a client, or all sub-theories have been processed. If all clauses are satisfied in any of the sub-theories, then the entire theory

is satisfied. If a contradiction is found in a sub-theory, then that sub-theory does not contain a solution and the client can request an additional segment. Having discussed the segmentation design of the *daspps* system, the next chapter discusses the overall structure of the communication system of *daspps* .

CHAPTER 7

DASPPS SYSTEM DESIGN

The design of the *daspps* system was motivated by three primary goals. The first is that the system must be scalable, such that resource utilization would not be limited to a particular subset of clients or local area network. Second, the system must be fault-tolerant, such that failure of one client node does not affect the search of other client nodes. Third, the system must be as transparent as possible, such that details of distribution does not introduce unnecessary complexity.

The *daspps* system consists of a communication sub-system and a modified sequent version of the *aspps* sequential solver. The communication sub-system implements a simple master/client communication model, using the Berkeley sockets and POSIX threads APIs. The master node runs a modified version of the *aspps* solver designed to partition the search space and assign the sub-theories to available client nodes (Figure 7.1).

The master node is responsible for the following tasks

- Problem session initialization and termination.
- Search-space partitioning.
- Sub-theory assignment and data transfer
- Solution retrieval

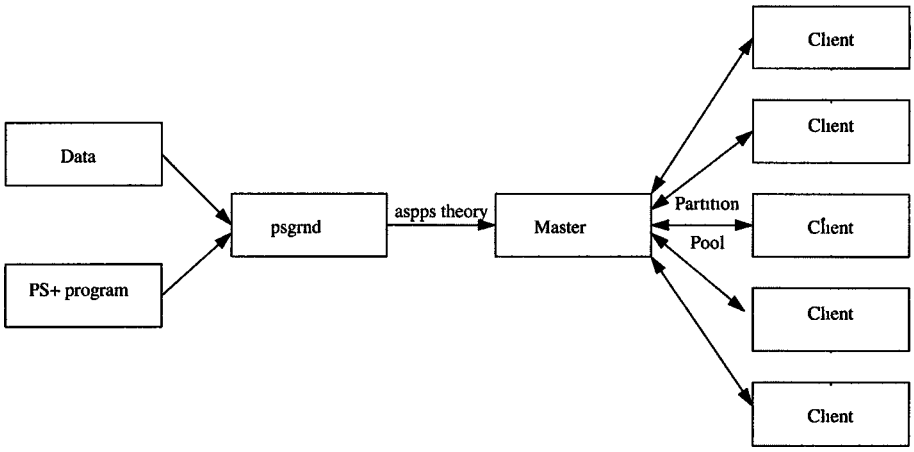

Figure 7.1: Daspps flow control

Given a grounded theory T, a *session* consists of (1) partitioning the search space, (2) assignment and distribution of segments, and (3) obtaining result. With the current implementation of *daspps* , a master node may only process one session at any one time.

Client nodes performs the follow tasks

- Sub-theory processing
- Send solution
- Request additional data

Within a session, a client node performs a complete, sequential search on the assigned sub-theory. If a solution is found within a given assignment, the client will notify the master node. On the other hand, if a solution is not found, the client will request additional work from the master. As such, once initialized, a client node will continuously work on behalf of the master until it is released.

Within the context of a session, interaction between master and client may be categorized as follows: session initialization, sub-theory assignment, and session termination. Each of these categories is described below.

7.1 Session Initialization

Session initialization of *daspps* clients is straight-forward. For each client node, the master node will send an INIT request, indicating the availability of a new session. If the client node is currently idle, i.e. not processing a current session on behalf of another master node, it will send a REPLY/OK response, indicating that the client node is now in a PENDING state, and will accept sub-theories from the master client. If, on the other hand, the client node is currently performing a search on behalf of another master node, the client will send a REPLY/BUSY to the currently requesting node, indicating that the client node is currently unavailable. The receipt of a REPLY/BUSY message will result in the master node removing that client from its host list.

The master node performs no search for the current problem. As such, if, after each client node has been queried, no clients are currently available, the master node will abort the current session. Otherwise, if at least one client node is PENDING for the master node, it will begin the process of search space partitioning discussed in chapter 6.

7.2 Sub-theory Assignment

Once the initial client pool has been initialized, the master begins the process of partitioning the search space. Assignment of client node/segment occurs dynamically: as each sub-theory is generated, the sub-theory is assigned to a currently PENDING client node. If a client node is available, the assignment is made. The sub-theory data is transfered to the client node following a SUBMIT request, after which the client node is marked BUSY. If, on the other hand, all nodes have been previous assigned, i.e. in a BUSY state, the sub-theory is queued until a client node issues a REQUEST.

Once all available client nodes have been assigned, the master node itself enters a PENDING state. If a previously assigned client node issues a work REQUEST, the master will mark the previously assigned sub-theory as PROCESSED, indicating no solution was obtained. If another sub-theory is available, the master node will respond to the request with another SUBMIT request followed by data.

If a client node issues a SUBMIT request, indicating that the previously assigned sub-theory yielded a solution, the master will accept the subsequent data and the session will be ended.

7.3 Termination

Normal termination of a session will result in the following cases.

- A client node returns a solution, indicating satisfiability.
- All client nodes have been released, indicating unsatisfiability.

If a client node returns a solution, the current theory is satisfiable. All remaining BUSY client nodes are released from the current session, and the master node halts. If all client nodes have been released, no assignment yielded a solution, and the problem is considered unsatisfiable and the master halts.

7.4 Messaging

Communication between master and client nodes is facilitated by simple message passing. All messages passed between nodes consist of the following fields, followed by optional data.

target A 32-bit buffer containing the address and port of the master node.

type An 8-bit buffer containing the message type.

flags An 8-bit buffer containing optional informational flags to augment message type.

status An 8-bit buffer containing status information.

datalen A 32-bit buffer containing the length any data following the message buffer.

The **target** field is used to reference the address and port number of the master node. This field is used by a client node when issuing either SUBMIT or REQUEST messages. The **type** field is used to indicate the message type. The **flag** field is not currently used. The **status** field is used only by a REPLY message type to indicate the status of a previous request. The **datalen** field is used to indicate the amount of data following the message buffer. This value is zero unless the message type is

Message Type	Sender	Recipient	Description
INIT	Master	Client	Initialize a new problem session
SUBMIT	Master	Client	Data segment submission
	Client	Master	Problem solution submission
REQUEST	Client	Master	Data segment request
RELEASE	Master	Client	Resources no longer required
REPLY	-	-	Atomic transaction result message

Table 7.1: Summary of Message Types

a SUBMIT, in which case datalen indicates the length of the submission. Currently implemented messages types are described below and summarized in Table 7.1.

7.4.1 INIT

The INIT message type is sent by a master node to a client node indication the availability of a new problem session. Replies to an INIT message are

REPLY/OK The client has successfully been initialized for the requesting master node. The client is now in a PENDING state.

REPLY/BUSY The client is currently working on a session for another master node.

REPLY/DUP The client node has already been initialized by the requesting master node.

7.4.2 SUBMIT

The SUBMIT message type may be sent by both the master and client nodes. If sent by a master node and received by a client node, the message is used to indicate that the following sub-theory has been assigned to the recipient node.

If received by the master node, and sent by a cached client node, the message represents a solution submission.

7.4.3 REQUEST

A REQUEST message type is sent by a client node to the master node. REQUEST is used to indicate that the previously assigned sub-theory did not yield a solution. As such, the master node will remove the sub-theory from the its queue. If additional sub-theories are available for processing, the requesting client node is sent the new assignment. Otherwise, the master node will release the client node from the current problem session.

SUBMIT Additional data segment is available.

RELEASE All sub-theories have been processed. The client node is released from the current session

7.4.4 RELEASE

A RELEASE message type is sent by the master node to client node. Receipt of a RELEASE indicates that the client node is no longer needed with the current session.

7.4.5 REPLY

A REPLY message type is an auxiliary type used with atomic transactions.

This chapter has discussed the communication mechanism used in the *daspps* system. The next chapter provides information on *daspps* usage. We then present the results of the current implementation of the *daspps* system.

CHAPTER 8

DASPPS USAGE

8.1 Installation and Configuration

The DASPPS system uses the autoconf/automake build system, allowing for easily tailored configuration and installation. To build the system, the only required software is the Berkeley sockets API and the POSIX threads (or compatible) API. Source code for the current implementation is

`http://www.cs.txstate.edu/~jh38107/daspps/current`. To build the package, simply type the command '**configure** followed by '**make**' in the daspps source directory. When the source is built, you can install the software by typing '**make install**'. The default location for installation is /usr/local, but this may be changed by running '**configure**' with the –prefix=/some/other/place option. For example, to build and install the package in the /opt directory

```
[jh38107@hawks daspps0.1-src]$ ./configure --prefix=/opt
[jh38107@hawks daspps0.1-src]$ make
[jh38107@hawks daspps0.1-src]$ make install
```

Note that during each step, output will be displayed on the console.

8.2 Usage

Computing models with the *daspps* system is similar to computing models with the *aspps* system. To compute models of a PS^+ theory (D, P), the theory must first be

grounded. To ground a theory, we use the program **psgrnd**. The required input to **psgrnd** is a single program file, one or more data files and optional constants. If no errors are found while reading the files and during the grounding process, an output file is constructed. The output file is a machine readable file which is the program file used by the *daspps* solver.

Once a program file has been generated, it is processed by the *daspps* master node. To invoke the *daspps* master, two arguments must be passed to it of the form

```
daspps -N <clientfile> -f <programfile>
```

The **-N** command-line flag is used to indicate the file to use to gather client information. This file is an ASCII text file. Each line contains the information used to connect to a particular client. The form of each line is

```
<clientname> <port>
```

where <clientname> is either a hostname, such as myhost.txstate.edu, or an address string, such as 111.222.333.444. Note that the address string is IPv4. The current implementation of *daspps* does not support IPv6. An optional <port> value may be given following the <clientname>, separated by whitespace. This value may omitted if the default port, 15321, is used. A place-holder string '–' may also be used in place of the port, in which case the default is used. Empty lines or lines whose first character is '#' are ignored. For example, the file

```
# An example host file for daspps
owls.cs.txstate.edu     --
hawks.cs.txstate.edu    --
eagles.cs.txstate.edu   --
condors.cs.txstate.edu 13231
```

includes a comment on the first line, followed by four entries. The first three hosts use the default port. The fourth host entry is configured to use port 13231.

The *daspps* system uses this client file to construct its initial pool of available clients. It is necessary that the *daspps* client process is running on systems provided in the client file. During initialization, the *daspps* master node will attempt to initialize each of the clients from this pool. Consequently, the client must be running prior to invoking the master. If a client is not available, it is removed from the pool.

8.2.1 Running the Client

The client process is actually the solver process running in the background on the client, i.e. the sequential *aspps* process. To invoke the client process, the *daspps* executable is passed the single argument:

```
[jh38107@eagles /opt/bin]$ daspps --client
```

8.2.2 Running the Master

The master node is the node on which the problem instance is begun. Required input for the master process is the client file and the grounded theory file. For example, the command

```
[jh38107@eagles /opt/bin]$ daspps -N hostfile.daspps -f nq128.aspps
```

would invoke the daspps master with the client file hostfile.daspps and the grounded theory file nq128.aspps.

CHAPTER 9

TEST RESULTS

The performance of the *daspps* system was measured against the sequential *aspps* system. Two problems were used to evaluate the performance of the *daspps* system. The first problem is the n-queens problem. The second problem comes from VLSI design. The test cases for both *daspps* and *aspps* were performed on a small testbed consisting of four Sun Blade 2000 UltraSPARC III+ workstations running at 900 MHz with 1GB of RAM. For all of the *daspps* executions four nodes were initialized and the actual number used is given in each table.

9.1 Results for n-queens

The n-queens problem consists of determining the position of n queens on an $n \times n$ chess board such that no queen is attacking another queen. In other words, we cannot have more than one queen on a row, or on a column or on the same diagonal. The n-queens program in the language of PS^+ follows:

```
1: pred queen(number, number).
2: var number C, R, I.

3: 1{queen(_,C)}1.
4: 1{queen(R,_)}1.

5: {queen(R+I-1,I)[I]}1 .
6: {queen(I,C+I-1)[I]}1 .
7: {queen(R-I+1,I)[I]}1 .
8: {queen(q - I + 1,R+I-1)[I]}1 .
```

Problem	aspps secs	daspps secs	Speed up ratio	Number of nodes
32-queens	0.06	0.51	0.12	3
64-queens	6.24	0.76	8.21	2
128-queens	****	1.88	–	4
**** denotes timeout after 10 minutes				

Table 9.1: Test results for n-queens

Line 1 defines *queen* as a program predicate with arity of two. Both arguments must be of type *number* where number is a data predicate. The second line declares program variables C, R, I of type *number*. The following lines are clauses where each clause contains one cardinality atom. Because there is no implication symbol in the clauses the cardinality atom is assumed by the **psgrnd** module to be the consequent and thus must be true. Line 3 clause maintains row restrictions and line 4 clause maintains column restrictions. Lines 5-8 contain clauses which provide diagonal restrictions.

We tested *daspps* on the n-queens problem using 32, 64, and 128 queens. We show results from executing the program sequentially, distributed, and the speedup in table 9.1. Runs which exceeded the alloted time of 10 minutes were terminated and are marked ****.

As can be seen from Table 9.1 for the smallest theory, 32-queens, the overhead resulting for *daspps* distributing the theory causes an increase in time rather than a speed up. This is to be expected, for the communication overhead required by *daspps*

is greater than the actual time required to find a solution. For larger theories, 64 and 128 queens the speed up is dramatic.

9.2 VLSI design

VLSI design has several steps. In this thesis we are only looking at the physical layout of components on the chip and in particular the placement of components without partitioning. Traditionally, specifications are given as a mesh or hyper graph and the first step in layout is to partition each mesh, the next step is to determine a configuration for the graph, layout is the next step and the last step is connecting the components through wire routing. Here we are modeling the layout without performing partitioning thus we can require all the components in a mesh to be *near* one another. We believe this will reduce total distance during wire routing and help prevent skews where the distance between components in a mesh can vary enough to cause timing problems. This is a simplification of VLSI layout and is used to illustrate the speedup of *daspps* .

```
1: pred placement(component,xcoord,ycoord).

2: var xcoord I,J.
3: var ycoord M,P.
4: var component  A,B,C.
5: var mesh X.

6: {placement(_,I,M)}1.

7: 1{placement(A,_,_)}1.

8: meshsize(X,C), inmesh(X,A), inmesh(X,B),
   A < B, (abs(I-J) + abs(M-P)) > C ,
   placement(A,I,M),  placement(B,J,P) ->.
```

The program or problem definition for component placement is given above. The line numbers are not part of the program but are added for explanation purposes. Line 1 defines the single program predicate used for placement. The arguments are the component label and the x and y grid coordinates. The coordinates are different allowing for a rectangular chip configuration. Lines 2 - 5 are variable declarations using the data predicates for the problem. Line 6 is a clause with a single cardinality constraint which restricts each coordinate position on the chip to having a most one component. Line 7 requires that each component be placed in exactly one coordinate position. Line 8 is a constraint which is used to ensure that components are *near* each other.

The chip specifications are randomly generated where the size of the chip and the number of components and meshes are input. The results reflect ten randomly generated 8×8 chips with 64 components and 32 meshes.

All tests demonstrated a reduction in time for finding a solution. We consider this as a demonstration of the benefits of adding a distributed framework to the *aspps* system. Having evaluated the performance of the current version of the *daspps* system, we provide some concluding remarks in the next chapter followed by a discussion of several specific areas requiring attention to improve the overall system.

Problem	aspps secs	daspps secs	Speed up ratio	Number of nodes
chip0	14.94	6.61	2.26	2
chip1	****	7.95	–	3
chip2	4206.20	8.88	473.67	4
chip3	****	7.08	–	3
chip4	6239.63	11.93	523.02	4
chip5	****	11.15	–	4
chip6	****	26.64	–	4
chip7	****	305.89	–	4
chip8	****	7.03	–	3
chip9	****	315.36	–	4
**** denotes timeout after 10 minutes				

Table 9.2: Test results for VLSI chip design

CHAPTER 10

CONCLUSION

This thesis presents a new distributed solver based on the sequential *aspps* solver. The design of a distributed framework with the *aspps* solver core takes advantage of special constructs within the logic PS^+. Applying a distribute framework, we are able to minimize the affects of branching on poor choice which must be tolerated by sequential solvers. The affects of these poor branch choices are realized only by the client node which receives it. Other client nodes are likely to receive sub-theories which branched on better heuristic choices. The direct representation of cardinality constraints, rather than requiring further processing, along with independent branching, allow efficient sub-theory distribution and minimum communication overhead. The results in chapter 9 indicate that this new system is a promising new direction in the distributed SAT solver community.

CHAPTER 11

FUTURE WORK

The primary goal of this thesis was to evaluate the benefits of a distributed framework within the *aspps* system. The results of our experiments, presented in chapter 9, indicate that this is a promising new direction. A primary emphasis on the design was scalability, such that the system would not be bound to a particular network infrastructure. While the current version the *daspps* system is pursuant to this end, there remains limitations. To fully achieve our goal, two topics must be addressed: authentication and client-side partitioning. This chapter discusses these two topics as they pertain to the *daspps* system.

11.1 Authentication

Considering the existing distributed solvers, authentication does not appear to be of concern. There are at least two possible reasons for this. First, the scalability of existing solvers is limited, by design, to a local network which is assumed trusted. As such, the need for authentication is greatly reduced. The exception to this is GridSAT [Chrabakh and Wolski, 2003b], which is designed to work within a computational grid. This environment demands some form of authentication/authorization. This requirement is fulfilled by its use EveryWare software [Wolski et al., 1999], which contains the necessary mechanisms. The second possible reason is that the emphasis

of current solvers is evaluating the benefits of distributed search techniques to speed up search. Additional requirements added by network uses is secondary.

Similar to Parallel Satz [Jurkowiak et al., 2001] and NAGSAT [Forman and Segre, 2002], the *daspps* solver does not include any authentication or authorization mechanisms. Now that the benefits of a distributed paradigm have been evaluated, the additional requirements of authentication must be address. Currently, the true scalability of the system is limited to a trusted network. There are a number of portable libraries available which offer authentication and authorization mechanisms, such as SASL. The use of these services are currently being evaluated.

11.2 Client-side Involvement

The second issue that must be address is client-side participation in directing search. Currently, a *daspps* client is a modified sequential *aspps* solver. There is no mechanism for client-initiated partitioning or off-loading. Both these features are desirable, as they allow the client to respond to local resource utilization, such as memory usage or timeout threshold.

Similar to Parallel Satz, the *daspps* system relies upon the its fault-tolerance to handle such events. If a client process dies during its search, due to memory limitations, for example, the master process will detect the failure and assign the sub-problem to another client node. In contrast, GridSAT use of Everyware software allows it to dynamically monitor and correct client processes, possibly off-loading a clients current assignment to another client via a peer-to-peer mechanism.

The approach to partitioning taken by the *daspps* system does not include any

mechanism to allow client-initiated partitioning. In contrast to other solvers, *daspps* performs dynamic partitioning, where each independent sub-theory is assigned to and processed by a client node. The client will continue to process the search space until either a solution is found, or the search space is exhausted. If found, the client will submit the solution. Otherwise, the client will simply ask for another sub-theory. This method was chosen to minimize communication overhead.

This chapter has discussed two issues limiting the scalability of the *daspps* system: authentication and client-side involvement. Addressing these two issues will dramatically increase the scalability of the *daspps* system. Authentication will facilitate increased scalability by allowing resource utilization which extends beyond a trusted network. A hierarchical partitioning mechanism will further extend the scalability by allowing a finer granularity of sub-theory distribution.

BIBLIOGRAPHY

[Chrabakh and Wolski, 2003a] Chrabakh, W. and Wolski, R. (2003a). Gradsat: A parallel sat solver for the grid.

[Chrabakh and Wolski, 2003b] Chrabakh, W. and Wolski, R. (2003b). GridSAT: A chaff-based distributed SAT solver for the grid.

[Citrigno et al., 1997] Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. (1997). The dlv system: Model generator and advanced frontends (system description). In *Workshop Logische Programmierung.*

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of Association for Computing Machines*, 7.

[East and Truszczyński, 2001a] East, D. and Truszczyński, M. (2001a). aspps – An Implementation of Answer-Set Programming with Propostional Schemata. In *Proccedings of 6th International Conference, LPNMR-2001).* Springer Verlag.

[East and Truszczyński, 2001b] East, D. and Truszczyński, M. (2001b). Propositional satisfiability in answer set programming. In *Proccedings of KI-2001).* Springer Verlag.

[East and Truszczynski, 2002] East, D. and Truszczynski, M. (2002). The aspps system.

[East and Truszczyński, 2004] East, D. and Truszczyński, M. (Accepted 2004). Predicate–Calculus based logics for modeling and solving search problems.

[East et al., 2004] East, D., Truszczynski, M., Mikitiuk, A., and Truszczynski, M. (2004). Tools for modeling and solving search problems.

[Forman and Segre, 2002] Forman, S. L. and Segre, A. M. (2002). Nagsat: A randomized, complete, parallel solver for 3-sat.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. (1979). *Computers and Intractability: A Guide to the theory of NP-Completeness.* W. H. Freeman and Company.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K., editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts. The MIT Press.

[Jurkowiak et al., 2001] Jurkowiak, B., Li, C.-M., and Utard, G. (2001). Parallelizing SATZ Using Dynamic Workload Balancing.

[Li and Anbulagan, 1995] Li, C. and Anbulagan, M. (1995). Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference on Principles and Practices of Constraint Programming*, volume 1330, pages 342–356.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01).*

[Niemela and Simons, 1997] Niemela, I. and Simons, P. (1997). Smodels — an implementation of the stable model and well-founded semantics for normal lp.

[Paarsch and Segre, 1999] Paarsch, H. J. and Segre, A. M. (1999). Extending the computational horizon: Effective distributed resource-bounded computation for intractable problems. Technical report, Society for Computational Economics.

[Segre et al., 2002] Segre, A. M., Forman, S., Resta, G., and Wildenberg, A. (2002). Nagging: a scalable fault-tolerant paradigm for distributed search. *Artif. Intell.*, 140(1-2):71–106.

[Wolski et al., 1999] Wolski, R., Brevik, J., Krintz, C., Obertelli, G., Spring, N., and Su, A. (1999). Running EveryWare on the computational grid.

VITA

Jason High was born in Nyack, New York, on March 10, 1975, the son of Joanie and Charlie High. After completing his work at Coronado High School in El Paso, Texas, he entered Texas State University-San Marcos. In the Fall of 1998, he received the degree of Bachelor of Arts from Texas State University-San Marcos. In Spring 1999, he entered the Graduate College of Texas State University-San Marcos. In Fall of 2002, he received the degree of Master of Arts from Texas State University.

Permanent Address: 6801 Morrill Road

El Paso, Texas 79932

This thesis was typeset using LaTex 1.0.7 by Jason High.