TEXAS STATE UNIVERSITY SAN MARCOS

Department of Computer Science
San Marcos, TX 78666

Report Number TXSTATE-CS-TR-2005-2

daspps a distributed implementation of the aspps system

Deborah East
Jason High

2005-05-11

daspps a distributed implementation of the *aspps* system

Deborah East and Jason High

Texas State University–San Marcos
San Marcos, TX

Abstract. We introduce *daspps*, a distributed general constraint system, based on the *aspps* system [1]. We describe the advantages of basing a distributed system on the *aspps* system and distributing only independent sub-theories. The language of logic PS^+ [2] facilitates the modeling, modifying and programming of search problems and the grounder *psgrnd* [1] instantiates the resulting program with data for a specific instance. During the instantiation, many of the constructs of the logic PS^+ are maintained. The resulting *aspps* theory is concise and retains the structure of the original problem. Despite concise theories scalability is still an issue. As the size of problem instances increase, single processor machines take too long to find solutions or they exhaust their resources. The distributed implementation, *daspps*, increases the resources available thus allowing us to solve problems we otherwise would not be able to solve. We describe how the *daspps* implementation minimizes communication overhead and provides robustness and scalability. We demonstrate the effectiveness of *daspps* by comparing results of executions by both *aspps* and *daspps*.

1 Introduction

We introduce *daspps*, a distributed general constraint system based on the *aspps* system [1], which uses a master-client paradigm for distributing independent sub-theories. Much of the research on general constraint solvers has been focused on satisfiability (SAT) solvers. One of the most important satisfiability algorithms is the Davis-Putnam algorithm [3, 4]. The introduction of the Davis-Putnam algorithm drew considerable interest in developing SAT solvers. However, due to difficulties in scaling, interest in SAT solvers waned until the development of probabilistic algorithms such as GSAT [5] and simulated annealing [6–8]. The success of probabilistic algorithms brought about a renewed interest in SAT solvers as general purpose solvers. The renewed interest in SAT solvers and increased speed in current computer hardware also led to development of new heuristics for Davis-Putnam algorithms. Several fast implementations of the Davis-Putnam algorithm [9, 10] were also developed.

There are still several issues involved in using SAT solvers as general purpose solvers. First is the need for a problem to be encoded in conjunctive normal form (CNF). Second, for each problem a program must be written to encode the problem as a SAT theory. Even minor changes in the problem could result in the need for major changes in the encoding program. More importantly, the encoding of a problem into a SAT theory could entail a polynomial increase in size which would make finding a solution impractical. Another important issue is the lack of scalability of SAT solvers, since search space for SAT increases exponentially as the size of the problem instances increase linearly.

The *aspps* system facilitates the modeling, programming, modifying and solving of search problems. A program in the language of logic PS^+ [1,2] is easy to model and modify. The grounder for the *aspps* [1] system, *psgrnd* [1] instantiates a program with data for a specific instance. During the instantiation, many of the constructs of the language of logic PS^+ are maintained resulting in a concise theory. Although the modeling and instantiation of the *aspps* system addresses some of the issues present in SAT solvers, scalability remains an issue in *aspps*. As the size of problem instances increase, single processor machines take too long to find solutions or they exhaust their resources. Currently there are implementations of distributed SAT solvers which are used as general constraint solvers such as *NAGSAT* [11], *//Satz* [12], *GridSAT* [13,14]. We present *daspps*, a distributed implementation based on the *aspps* system. Distributing processes increases the resources available thus allowing us to solve problems we otherwise would not be able to solve. The *daspps* implementation focuses on minimizing communication overhead and providing robustness and scalability. Section 2 provides background information on the *aspps* system with section 2.3 describing branching heuristics. Section 3 discusses current distributed solvers based on SAT solvers. In section 4 we present *daspps* and describe the generation of sub-theories, initialization of client nodes and the assignment of sub-theories to initialized clients. Section 5 presents results of distributing the n-queens problem and a problem based on VLSI design. Finally, we present conclusions and discuss future work.

2 Aspps

The *aspps* system is an answer-set programming system based on the extended logic of propositional schemata (PS^+) [15], which allows variables but not function symbols (disallowing function symbols ensures that the theories will be finite) in the language. The *aspps* system consists of *psgrnd* and the *aspps* solver. The *psgrnd* module uses as input a program written in logic PS^+ and data for a specific instance of the problem and outputs an *aspps* theory. The *aspps* solver is a complete solver which takes an *aspps* theory as input.

2.1 PS^+

A theory in the logic PS^+ is a pair (D,P), where D is a set of ground atoms (only constant symbols as arguments) representing an *instance of a problem* (input data), and P is a set of PS^+-clauses representing a *program* (an abstraction of a problem). Predicates in a PS^+ theory are classified as *data* and *program* predicates. Ground atoms built of data predicates represent a problem instance. The program P consists of clauses built of atoms involving both data and program predicates. Clauses are written as implications and explicit negation of atoms is not allowed (the implication symbol is omitted if a clause has an empty conjunction of atoms as the antecedent). The program is written to capture all the relevant constraints specifying the problem to be solved. The meaning of a PS^+-theory $T = (D,P)$ is given by a *family* of PS^+-models [15].

The essential difference between the logic PS^+ and the logic of propositional schemata is in the definition of a model. Following the intuition that computation must not modify the data set, a set of ground atoms M is a model of a PS^+ theory (D,P) if M is a propositional model of the grounding of (D,P) and if it coincides with D on the part of the Herbrand Universe given by data predicates.

In some cases, the consequent of a clause must be a disjunction of a set of atoms that depends on a particular data instance. To build such disjunctions, we introduced in the language of the logic PS^+ the notion of an *e-atom*. An example of an e-atom (in the context of a graph-coloring setting) is $color(X,\_)$. It stands for the disjunction of all atoms of the form $color(X,c)$, where c is a constant from the extension of the data predicate *clr*. The current version of logic PS^+ allows also for more complex variants of e-atoms.

Another powerful modeling concept in the language of logic PS^+ is that of a c-atom. An example of a c-atom is $k\{color(X,\_)\}m$. We interpret the expression within the braces as a specification of the *set* of all ground atoms of the form $color(X,c)$, where c is a constant from the extension of the data predicate *clr*. The meaning of the atom $k\{color(X,\_)\}m$ is: at least k and no more than m atoms of the form $color(X,c)$ are true.

In addition to the program and data predicates, the *aspps* implementation includes *predefined* predicates and function symbols such as the equality operator $==$, arithmetic operators $<=$, $>=$, $<$ and $>$, and arithmetic operations $+$, $-$, $*$,/, $abs()$ (absolute value), $mod(N,b)$, $max(X,Y)$ and $min(X,Y)$. These symbols are assigned their standard interpretation. It is necessary to emphasize that the domains are restricted only to those constants that appear in a theory.

2.2 *psgrnd*

The grounding of logic PS^+ programs is performed by the *psgrnd* module. When grounding, we first evaluate all expressions built of predefined operators. We then form ground instantiations of all program clauses. Next, we evaluate and simplify away all atoms built of predefined predicates. We also simplify away all atoms built of data predicates (as they are fully determined by the data part). Therefore, the ground PS^+-theory contains only ground atoms built of program predicates. The structure of c-atoms is preserved in ground theories and the *aspps* solver takes advantage of the structure of the c-atom.

2.3 Branching heuristics

The *aspps* solver uses a depth first backtracking algorithm similar to [3, 4] but extends unit propagation and branching heuristics. The *aspps* solver has branching heuristics which were developed to take advantage of the constructs maintained in the ground theories of the logic PS^+. In the *aspps* solver, as in SAT complete solvers, branching heuristics are designed to choose the most constrained literal. For SAT solvers using theories in CNF the most constrained literal heuristic is typically computed or estimated based on the number of occurrences of the literal in the shortest clauses. This works very well for many problems such as color-ability or random 3-SAT but less well for structured problems where the CNF encoding distorts the structure of the problem. The constructs in *aspps* theories enable the branching heuristics in the *aspps* solver to be more accurate.

Let's look at two examples. First, the color-ability problem where the CNF theory is similar to the *aspps* theory. Assume we have the set of colors $C = \{red, blue, green\}$ and a graph $G(V,E)$ where $V = \{1,2,3\}$ and $E = \{(1,2),(2,3)\}$. The structure of the problem results from the edges in the graph since the clauses which require each vertex to be assigned exactly one color are symmetrical. Branching heuristics for both the *aspps* solver and SAT solvers will find as equally constrained the following atoms: {color(2,red), color(2,green),color(2,blue)}.
Although *aspps* theory has fewer clauses than the CNF theory, the key issue is that the branching heuristics in both cases will find the same set of atoms and consequently SAT solvers perform as well or better on culpability instances as does the *aspps* solver.

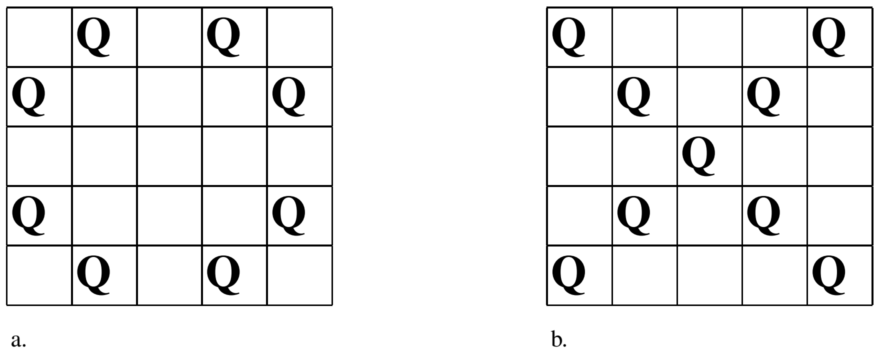

Fig. 1. Diagonal conflict for queens.

In contrast to the color ability problem, SAT solvers perform much worse on the n-queens problem than the *aspps* solvers. This is partly due to the increase in size for the CNF theory; however, it is our contention that the key difference is the branching heuristic in the *aspps* solver which takes advantage of the c-atoms. The branching heuristic in the *aspps* chooses branching points based upon weights assigned to atoms and c-atoms. Each atom in the *aspps* theory is assigned a weight which is the sum of the constraint weight for each clause or c-atom in which it occurs. The constraint weight is maximum for a clause or a c-atom of length two and decreases as the length increases. Weights are also placed on c-atoms which are the sums of the weights of its base atoms (the set of atoms making up the c-atom). A c-atom is branchable if it is forced true and it is satisfiable only if exactly one of the unassigned base atoms is assigned true. The *aspps* branching heuristic finds a c-atom which is branchable, has minimum length and maximum weight. The *aspps* theory for the 5-queens problem contains clauses with c-atoms which are row, column symmetric. These c-atoms are forced atoms and exactly one atom within the c-atom must be true thus they are branchable atoms. Since they are symmetric, without additional clauses or c-atoms they would all have the same weight. For 5-queens problem the row and column clauses are:

$$
\begin{gathered}
1\{queen(1,1)\ queen(1,2)\ queen(1,3)\ queen(1,4)\ queen(1,5)\}1 \\
1\{queen(2,1)\ queen(2,2)\ queen(2,3)\ queen(2,4)\ queen(2,5)\}1 \\
\vdots \\
1\{queen(1,1)\ queen(2,1)\ queen(3,1)\ queen(4,1)\ queen(5,1)\}1 \\
1\{queen(1,2)\ queen(2,2)\ queen(3,2)\ queen(4,2)\ queen(5,2)\}1 \\
\vdots
\end{gathered}
$$

The meaning is that each row and each column must have exactly one queen assigned true. The diagonal constraints consist of clauses with forced c-atoms which are only partially symmetric and are not branchable; however, these c-atoms still they play an important role in the branching heuristics of *aspps*. The clauses with c-atoms to prohibit the queen placements for Fig. 1 a. are:

$$
\begin{gathered}
\{queen(1,2)\ queen(2,1)\}1 \\
\{queen(1,4)\ queen(2,5)\}1 \\
\{queen(4,1)\ queen(5,2)\}1 \\
\{queen(4,5)\ queen(5,4)\}1
\end{gathered}
$$

The meaning for the c-atoms for the diagonal constraints is that each diagonal can have at most one queen. The c-atoms for the diagonals shown in Fig. 1 a. are not branchable because they do not have a lower bound; however, due to their short length they provide a maximum increase to the sum of the individual atom weights. The c-atom chosen by the *aspps* heuristic would contain some of the individual atoms which are in the two–atom c-atoms. Clauses with c-atoms to prohibit the longer diagonal constraints represented by queen placements for Fig. 1 b. are:

$$
\begin{gathered}
\{queen(1,1)\ queen(2,2)\ queen(3,3)\ queen(4,4)\ queen(5,5)\}1 \\
\{queen(1,5)\ queen(2,4)\ queen(3,3)\ queen(4,2)\ queen(5,1)\}1
\end{gathered}
$$

These c-atoms are not branchable either because they have no lower bound and in addition they provide less weight to their base atoms due to their relatively long length. There are eight c-atoms representing row 1, 2, 4 and 5 and column 1, 2, 4 and 5 constraints with base atoms which are also base atoms of the c-atoms representing the constraints for Fig 1 a. The *aspps* branching heuristic will choose randomly one of these eight c-atoms for branching. After choosing a c-atom the heuristics order the base atoms by their weight.

The CNF encoding for the 5-queens requires the representation of row and column restrictions as five-literal clauses:

$$
\begin{gathered}
queen(1,1) \vee queen(1,2) \vee queen(1,3) \vee queen(1,4) \vee queen(1,5) \\
queen(2,1) \vee queen(2,2) \vee queen(2,3) \vee queen(2,4) \vee queen(2,5) \\
\vdots \\
queen(1,1) \vee queen(2,1) \vee queen(3,1) \vee queen(4,1) \vee queen(5,1) \\
queen(1,2) \vee queen(2,2) \vee queen(3,2) \vee queen(4,2) \vee queen(5,2)
\end{gathered}
$$

$$\vdots$$

and two-literal clauses:

$$\begin{gathered}\neg queen(1,1) \vee \neg queen(1,2) \\ \neg queen(1,1) \vee \neg queen(1,3) \\ \vdots \\ \neg queen(2,1) \vee \neg queen(2,2) \\ \neg queen(2,1) \vee \neg queen(2,3) \\ \vdots \\ \neg queen(1,1) \vee \neg queen(2,1) \\ \neg queen(1,1) \vee \neg queen(3,1) \\ \vdots \\ \neg queen(1,2) \vee \neg queen(2,2) \\ \neg queen(1,2) \vee \neg queen(3,2) \\ \vdots\end{gathered}$$

and the diagonal constraints as two-literal clauses:

$$\begin{gathered}\neg queen(1,1) \vee \neg queen(2,2) \\ \neg queen(1,1) \vee \neg queen(3,3) \\ \vdots\end{gathered}$$

The limitations imposed by CNF may result in less constrained literals being chosen for branching. In the 5-queen example the literal $\neg queen(3,3)$ occurs most frequently in the two-literal CNF clauses and $queen(3,3)$ occurs with equal frequency in the five-literal CNF clauses. Thus the first branch point will be on $queen(3,3)$ for the SAT solver in contrast to the more constrained atoms chosen by the *aspps* branching heuristic.

The choice of branch points is critical to the efficiency of a solver, either a SAT solver or the *aspps* solver. Maintaining the structure of the underlying problem in the *aspps* theory allows the *aspps* branching heuristics to make choices based on the structure of the problem and not on a structure imposed by the theory format.

3 Current distributed solvers

Recent research into the incorporation of parallel/distributed paradigms into SAT solvers has resulted in several successful systems. We discuss only *complete* distributed solvers.

GridSAT [13, 14] is a complete distributed solver based on the zChaff [16] 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 [17] communication framework. Because GridSAT uses zChaff as its core solver system the exchange of learned clauses among registered client nodes is required. The 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, 2) a registered client finds a solution, 3) the master node times out or 4) a registered client node exceeds allowed resources or becomes otherwise unavailable. 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.

Another complete distributed SAT solver is Parallel Satz [12], based upon the sequential solver Satz [10]. 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. A simple master-slave communication model is used, employing RPC as a message passing framework. Process invocation and termination is facilitated using the Berkeley RSH protocol. All work is begun on the master node. The master node will halt on the following cases: 1) all slaves are idle, 2) a solution is found. Similar to the GridSAT solver, the load-balancing mechanism of Parallel Satz insures that if all slave nodes are in an idle state, then there is no more processing to be done, indicating unsatisfiability of the search problem. Unlike GridSAT, a client node failure does not terminate the master process, indicating a degree of fault-tolerance in Parallel Satz.

The last complete distributed solver we discuss is NAGSAT which is limited to 3-SAT problems. Nagging [18, 11] is a general-purpose, asynchronous, parallel search technique where a single master node, or processor, performs a standard DPLL search procedure which is not derived from a current SAT solver. 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.

4 Distributed Aspps

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

The *daspps* consists of a communication sub-system and a modified version of the *aspps* sequential solver. The communication sub-system implements a simple master/client communication model, using the Berkeley sockets and POSIX thread APIs. The master node runs a modified version of the *aspps* solver designed to partition the problem search space and assign the resulting sub-theories to available client nodes (Figure 2). The master node does not perform explicit search-space processing, however it is possible for the master node to find a solution during partitioning.

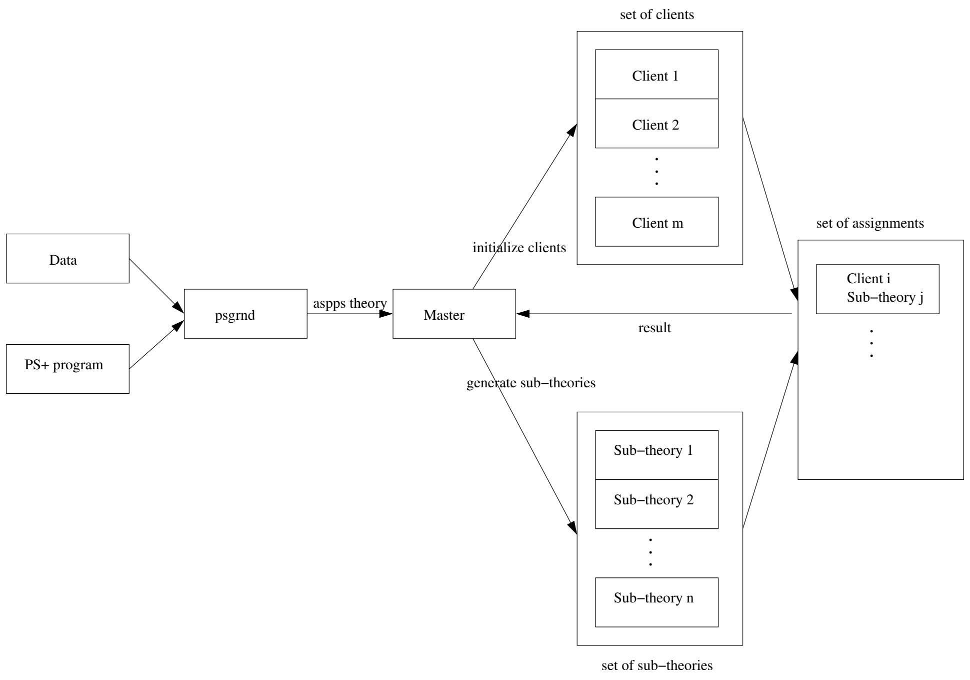

Fig. 2. Flow control of *daspps*

4.1 Initialization of Client nodes

A list or lists of possible clients nodes are maintained as files on the master node. The client nodes are identified either by a host name or an ip address. The clients are not restricted to a local area network. In fact, there is no restrictions on either the number or location of client nodes maintained in the list. At the beginning of a session, the master node initializes client nodes from a client list. If a client node is available when the initialization attempt is made then the client node is placed in a queue of initialized nodes, otherwise, the client node is ignored for this session.

4.2 Partitioning

The primary task of employing a distributed framework within an existing sequential solver is the development of an efficient partitioning 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 processed by a client node. This technique is called *search-space partitioning* and is used by the majority of existing solvers. Search-space partitioning within the *daspps* system differs significantly from other solvers (see Section 3). Rather than initializing a client with the complete search problem, and partitioning based upon a specified criteria, the master node in the *daspps* system partitions the search space dynamically. Each client node receives only

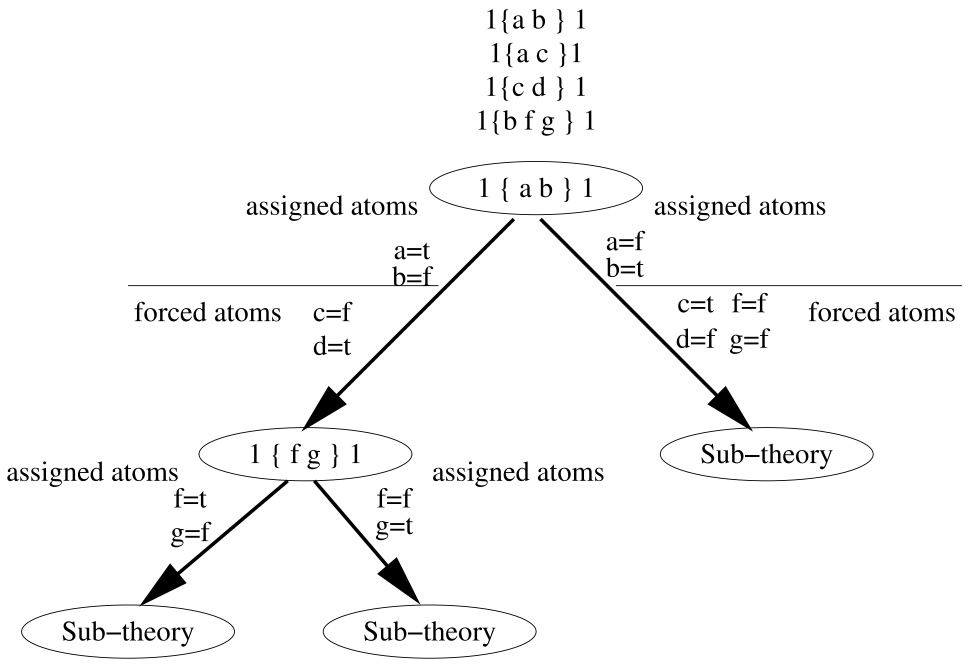

Fig. 3. Partitioning of sub-theories in *daspps*

its assigned sub-theory. Client nodes perform no further partitioning, i.e., the sub-theory is solved sequentially by the client node.

As discussed in the Section 2, the *aspps* system benefits from the direct representation of c-atoms. In the *aspps* system, the grounded PS^+ theories maintain the structure of the c-atom. A grounded PS^+ theory T such that,

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

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 c-atoms or ground atom. A c-atom is a collection of ground atoms such that if at least p and at most q of the ground atoms are true then the c-atoms is true. Otherwise, the c-atom is false. Thus the value of a c-atom is 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 with the following partial assignments:
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 five sub-theories with the following partial assignments:
a=false,b=false,c=false,d=false,e=false
a=false,b=false,c=false,d=true,e=true
a=false,b=false,c=true,d=false,e=true
a=false,b=false,c=true,d=true,e=false
a=false,b=false,c=true,d=true,e=true

Sub-theories resulting from either c-atoms or ground atoms are 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 3. 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.

4.3 Assignment of sub-theories to initialized client nodes

Each sub-theory is assigned to one or more available client nodes for processing. We say processing of a sub-theory is complete if a solution is found or no solutions is found after exhausting the search space. Sub-theory assignment is performed as follows. Let N be a set of available clients, S be the set of sub-theories 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 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 sub-theory.

The master node initializes the session, partitions the theory into sub-theories, assigns sub-theories to client nodes, receives results from the client nodes and terminates the session. The client node performs a complete, sequential search on the assigned sub-theory. If a solution is found within the assigned sub-problem, the client will notify the master node. Otherwise, after determining that the sub-theory does not contain a solution the client will request additional work from the master. Each client node, once initialized, will work on behalf of the master until it is released.

5 Results

We show the results of testing *daspps* on two problems. The first problem is the n-queens problem. The second problem comes from VLSI design. The test cases for both *aspps* and *daspps* were executed on 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.

For a comparison of execution times for *aspps*, SAT solvers and stable model semantics solver *smodels* on a number of different problems see [1] and for more information on modeling in the language of logic PS^+ see [2].

5.1 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 remove 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 .
```

The 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 c-atom is assumed by the *psgrnd* to be the consequent and thus must be true. The Line 3 clause maintains row restrictions and the Line 4 clause maintains column restrictions. The Line 5-8 clauses provide for diagonal restrictions.

We tested *daspps* on the n-queens problem using 32, 64 and 128 queens and stopping upon finding the first solution. We show results from executing the program sequentially, distributed, and the speedup.

As can be seen from Table 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 which can be expected. For larger theories, 64 and 128 queens the speed up is dramatic.

Problem	*aspps*	*daspps*	Speed up	Number
	secs	secs	ratio	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
*** time out after 10 minutes				
** not calculated due to time out of *aspps*				

Table 1. Results of n-queen executions.

5.2 VLSI design

VLSI design has several steps. In this paper we are only looking at the physical layout of components on the chip on 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. We stopped execution when the first solution was found.

Instance	*aspps*	*daspps*	Speed up	Number
	secs	secs	ratio	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
*** time out after 10 minutes				
** not calculated due to time out of *aspps*				

Table 2. Results of executions of the component placement problem.

All instances demonstrate a reduction in time for finding solutions.

6 Conclusions

The *daspps* system is scalable. We can increase the number of sub-theories generated by the master node by increasing the ratio of assigned atoms to total atoms with that the number or location of client nodes which can be initialized is not restricted to a local area network or specific network. Communication overhead is minimized since there is no communication between client nodes and communication between master nodes and client nodes is minimal.

The *daspps* system is robust since the failure of any client node does not cause the session to fail. In addition, a sub-theory can be sent to multiple client nodes until it is processed. Thus if a client node fails during processing of a sub-theory that sub-theory will still be processed by another client node. This results in a minimal amount of redundancy.

The *daspps* system uses the language of PS^+ with no additional information or control needed by the programmer. The user of the *daspps* system must only have a list of clients and an *aspps* theory to execute a distributed session.

There are several additional issues related to the efficiency of *daspps*. First, modeling problems with PS^+ and grounding instances with *psgrnd* results in *aspps* theories which are concise. Second, *aspps* branching heuristic uses the underlying structure of the original problem. Finally, as is well known, in depth first searches early decisions on branching greatly impact the total number of branches. By distributing sub-theories from relatively high branch points, the problem of a bad choice of which branch to take first is minimized.

References

1. East, D., Truszczyński, M.: Predicate–Calculus based logics for modeling and solving search problems. ACM Transactions on Computational Logic (2004)
2. East, D., Iakhiaev, M., Mikitiuk, A., Truszczyński, M.: Tools for modeling and solving search problems. In: Third International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, Toronto (2004)
3. Davis, M., Logemann, G., Loveland, D.: A machine learning for theorem-proving. Comm. Assoc. for Computing Machines **5** (1962)
4. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of Association for Computing Machines **7** (1960)
5. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Proccedings of the Tenth National Conference on Artificial Intelligence(AAAI-92). (1992)
6. Johnson, D., Aragon, C., McGeoch, L., Schevon, C.: Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning. Operations Research **39** (1991)
7. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science (1983)
8. Spears, W.M.: Simulated annealing for hard satisfiability problems. DIMACS Cliques, Coloring and Satisfiability **26** (1996)
9. Dubois, O., Andre, P., Boufkhad, Y., Carlier, J.: SAT versus UNSAT. DIMACS Cliques, Coloring and Satisfiability **26** (1996)
10. Li, C., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of IJCAI-97. (1997) 366–371
11. Forman, S.L., Segre, A.M.: Nagsat: A randomized, complete, parallel solver for 3-sat. In: Fifth International Symposium on the Theory and Applications of Satisfiability Testing. (2002)
12. Jurkowiak, B., Li, C.M., Utard, G.: Parallelizing Satz Using Dynamic Workload Balancing. In Kautz, H., Selman, B., eds.: Electronic Notes in Discrete Mathematics. Volume 9., Elsevier Science Publishers (2001)
13. Chrabakh, W., Wolski, R.: Gradsat: A parallel sat solver for the grid. In: In Proceedings of IEEE SC03, November 2003. (2003)
14. Chrabakh, W., Wolski, R.: GridSAT: A chaff-based distributed SAT solver for the grid (2003)
15. East, D., Truszczyński, M.: Propositional satisfiability in answer set programming. In: Proccedings of KI-2001), Springer Verlag (2001)
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001)
17. Wolski, R., Brevik, J., Krintz, C., Obertelli, G., Spring, N., Su, A.: Running EveryWare on the computational grid (1999)
18. Segre, A.M., Forman, S., Resta, G., Wildenberg, A.: Nagging: a scalable fault-tolerant paradigm for distributed search. Artif. Intell. **140** (2002) 71–106