ATMS and Satisfiability on the DAP

Mike Hopkins
Department of Computer Science,
Queen Mary and Westfield College,
University of London, U.K.
leonardo@dcs.qmw.ac.uk

Abstract

The purpose of the paper is explore the implemetation of an ATMS on a Distributed Array Processor (DAP) based on the parallel ATMS of Dixon and deKleer for the connection machine detailing the simplification of the original serialised ATMS model of deKleer's. We also look at how the implementation of the ATMS can be adapted to, instead of creating all models, generate the models one by one. This then allows the size of problem being considered to be greatly increased though removes the incremental nature of the ATMS.

1 Introduction

Dixon and deKleer [DdK88] describe a parallel ATMS for SIMD machines and, in particular, the connection machine. In reconstructing the ATMS for a parallel architecture they have simplified the ATMS model [dK86a, dK86b, dK86c]. There are four primitive functions for the parallel ATMS:

- adding an *assumption*
- adding a *node*
- adding a *justification*
- returning the *extension* of an *assumption* or *node*

This simplification is achieved by admiting negative assumptions in justifications. The following example demonstrates how they might be used:

Example 1 from [DdK88] *Mr. X wants to meet with Art, Betty and Chris; each at least once and the available times are 1:00, 2:00 and 3:00; Art can't come at 2:00. These represent the nine assumptions:* $\{a_1, a_2, \ldots, c_3\}$.

We can represent the fact that Chris is met at least once by sentence:

$$c_1 \vee c_2 \vee c_3$$

In the ATMS this is represented by the justification:

$$\neg c_1 \wedge \neg c_2 \wedge \neg c_3 \rightarrow \perp$$

There will be similar justifications for Art and Betty.

He would like the following constraints to hold:

1. *meet Art alone.*

2. *meet Art before and meeting with Chris.*

3. *meet Betty before and meeting with Chris.*

4. *to have no meetings a 3:00.*

Of these, the most important are the first three but hopefully the last will also hold; we shall represent these constraints with the four nodes:

$$\{n_1, \ldots, n_4\}$$

We can state that if he meets Betty at 2:00 and does not meet Chris at 1:00 or 2:00 then the third constraint holds:

$$b_2 \wedge \neg c_1 \wedge \neg c_2 \rightarrow n_3$$

There are two such justifications to completely represent this constraint.

If we introduce a further node, n_5, we can decide if the first three constraints are met:

$$n_1 \wedge n_2 \wedge n_3 \rightarrow n_5$$

And then finally, we can discover whether the last constraint holds:

$$n_5 \wedge \neg a_3 \wedge \neg b_3 \wedge \neg c_3 \rightarrow n_4$$

Having added the above justifications along with the others required to fully define the constraints (a total of fourteen), the extension of n_4 will give all possible solutions to this problem (if any) and it transpires that there none! On the other hand there are six solutions in the extension of n_5.

As demonstrated by this example, the use of negative assumptions can make the encoding of particular problems very simple. The introduction of negative assumptions into the parallel ATMS removes the need for the cumbersome addition of extra assumptions and justifications explicitly to represent the negation of an assumption within the serial ATMS. Also by admiting negative assumptions, completeness is achieved without the need for extra mechanisms while in the serial ATMS, completeness is guaranteed using hyper-resolution on the justifications and this turns out to be very expensive [1] . In [HC93] the equivalence is shown between propositional logic and the parallel ATMS without nodes which, though is apparently obvious, highlights the automatic subsumption checks that are obtained in the ATMS, to guarantee that all subspaces are disjoint, therebye improving on the efficiency over, say, a simple semantic tableau system.

In addition to the use of justifications to describe the constraints on assumptions and nodes we consider the addition of *mutual exclusions* which are meta-level constructs that insist that at most one of a set of assumptions and/or nodes can be true at a time; when combined with a justification that states at least one of the set of assumptions/nodes is true then we have a simple mechanism for constraint satifaction problems using variable assignment where the set of assumptions and nodes represent the finite domain of assignments for a particular variable. These mutual exclusions can be defined in terms of justifications or alternatively, as we show below, they can be made a primitive function therebye dramatically increasing the efficiency.

Unfortunately by admitting the simplified model of the parallel ATMS the quaranteed property of minimal extensions is lost and usually involves some processing each time an extension is asked for if the minimality is required. Fortunately in many problems this lack of minimality does not present any particular problems. In section 5 we detail the method required to obtain these minimal extensions.

2 The Representation of Data

The *assumption space* for a given problem consists of a complete set of *points* where each *point* is a unique valuation for each assumption and node in the ATMS of either True or False such that valuation is consistent with the justifications that have been supplied by the problem solver.

Because of the exponential number of points in an unconstrained ATMS (Dixon and deKleer produced their *algorithm 1* where the assumption space was

[1]The incompleteness with respect to propositional logic of the basic serial ATMS is because the justifications and nogoods are both horn-clauses and there is no mechanism for introducing clauses with more than one unnegated literal. To partially overcome this problem of incompleteness, the serial ATMS introduces the *choose* construct where $choose(C_1, C_2, \ldots)$ is equivalent to $C_1 \vee C_2 \vee \ldots$. Again, the assumptions occuring within the choose construct cannot be negative and therefore the ATMS still falls short of full propositional logic; i.e. there is no representation for non-horn clauses that contain a least one negated literal; further mechanisms are described to overcome this restriction. In addition, hyperresolution is required for any deduction to be made from the choose construct; this mechanism is both expensive and complicated (see [dK86b] for details and further problems with completeness).

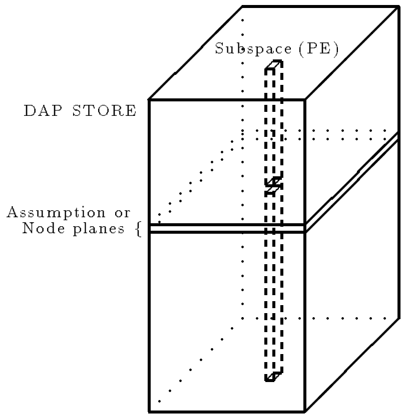

Figure 1: SSA: storage of assumptions and nodes on the DAP

represented explicitly by every possible consistent truth assignment to the assumptions, thereby doubling the number of stored points when ever a new assumption was added), the abstract data representation used here is based on their *algorithm 2* where the assumption space is represented as a set of subspaces (each of which represents a set of points within the assumption space); the value of an assumption within any subspace is represented by $\mathcal{T}$ (true), $\mathcal{F}$ (false) or $\mathcal{B}$ (both). $\mathcal{B}$ specifies that the assumption can be either $\mathcal{T}$ or $\mathcal{F}$. A subspace containing n $\mathcal{B}$'s therefore represents 2^n points within the assumption space. The algorithm must guarantee that all consistent points are stored once and only once (hence all subpsaces must be disjoint.)

Within the DAP, each assumption or node is represented by two parallel logical vectors; each corresponding element of these two vectors refers to the value of that assumption or node within a particular subspace. Two vectors are required because of the three possible values; the first row is the $\mathcal{T}$ row while the second is used for $\mathcal{F}$ (the value is set to true). Hence for the value $\mathcal{B}$, both rows are set to true. We shall refer to the 2D array that represents the ATMS for n assumptions or nodes ($2*n$ rows) and parallel across the columns (the subspaces), as the subspace array (SSA).

Also required is a logical mask, SSMASK, that is the same size as the parallel dimention of SSA, to determine the active columns (or subspaces) in SSA.

By virtue of the *enhanced* Fortan for the DAP, the declaration of these parallel vectors is not limited to the size of the DAP but can be of any size. We can therefore treat these as virtual processors, creating a comparable size to that of the connection machine; unfortunately, this is still not as flexible as the *xectors* (parallel objects!) in *CMLISP* which can change size dynamically. The number of columns in SSA is kept to a multiple of 1024 (for the 510 series or 4096 for the 610) so that each row occupies an exact number of planes, therebye maximising

$$\begin{array}{c|c} l_1 & \mathcal{B} \\ l_2 & \mathcal{B} \\ \vdots & \vdots \\ l_{m-1} & \mathcal{B} \\ l_m & \mathcal{B} \end{array} \quad \xRightarrow{[l_1,\ldots,l_m]\rightarrow n} \quad \begin{array}{cccccc} \mathcal{T} & \mathcal{T} & \mathcal{T} & \cdots & \mathcal{T} & \mathcal{F} \\ \mathcal{T} & \mathcal{T} & \mathcal{T} & \cdots & \mathcal{F} & \mathcal{B} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathcal{T} & \mathcal{T} & \mathcal{F} & \cdots & \mathcal{B} & \mathcal{B} \\ \mathcal{T} & \mathcal{F} & \mathcal{B} & \cdots & \mathcal{B} & \mathcal{B} \end{array}$$

Figure 2: Addition of a justification

the parallelism of the operations on the vectors.

3 Assumptions, Nodes and Justifications

To add a new fact, whether an assumption or a node, requires that the fact is indexed to the next two previously unused adjacent rows of SSA. The value for that fact must be initialised within every active subspace; $\mathcal{B}$ for an assumption and $\mathcal{F}$ for a node.

A justification consists of an antecedent list of assumptions (each of which can be either positive or negative) and/or nodes (always positive) and a consequent consisting of either a node or the contradiction $\perp$[2].

To apply a justifiction to a particular subspace, we first need to check if each of the antecedents of the justification are true within the subspace. i.e. we need to check, for each positive literal in the list, that its value in the subspace is $\mathcal{T}$ or $\mathcal{B}$ and, for each negative literal, its value is $\mathcal{F}$ or $\mathcal{B}$. If this is not the case then the justification is not applicable and the subspace already satisfies the justification.

On the other hand if the above check succeeds then we need to extract the subspaces that contain only $\mathcal{T}$'s amd $\mathcal{F}$'s for the antecedent of the justification; this we shall refer to as *separating out* the subspace. To separate out the subspace we need to iterate through all the literals in the justification whose value in the subspace is $\mathcal{B}$. At each iteration the subspace must first be copied and then for the literal in question the value in one subspace is set to $\mathcal{T}$ and in the other $\mathcal{F}$. If the literal is positive in the antecedent list then the subspace which has $\mathcal{F}$ for that literal now satifies the justification; we then proceed with the iterative process in the other subspace. Figure 2 shows the subspaces generated by applying a justification with only positive literals to a subspace that only has $\mathcal{B}$'s for each of the literals (it should be noted that the disjoint nature of the subspaces has been maintained by this process.) This action of splitting the subspace at each iteration and assigning the values $\mathcal{T}$ and $\mathcal{F}$ to the appropriate assumptions in each respectively is refered to, by Dixon and de Kleer, as *forking*.

[2] corresponding to the old notion of a *nogood*

Having produced this subspace where each of the members in the antecedent list are $\mathcal{T}$ ($\mathcal{F}$ for negative assumptions) then if n is a node then it is marked $\mathcal{T}$ in the subspace corresponding to the first column while if n is $\perp$ then this subspace is deactivated. If the value of a node is changed from $\mathcal{F}$ to $\mathcal{T}$ in any subspace then it is necessary for any previously applied justification with that node as an antecedent to be reapplied. This continues until no more nodes are changed; it is because of this operation that the occurrence of negative nodes in justifications is prohibited, thereby ensuring that this process terminates.

The aim is to first locate subspaces where, for each member of the antecedent list, the value is $\mathcal{T}$ ($\mathcal{F}$ for negative assumptions). If the justification has a node as a consequent then we need only process subspaces where the value of that node is $\mathcal{F}$.

In the later case it is then necessary to reapply all justifications with that node as an antecedent. This continues until no more nodes are changed; it is because of this operation that the occurence of negative nodes in justifications is prohibited, therebye ensuring that this process terminates.

It should be observed that there is scope for altering the order that the assumptions are dealt with in this separating out such the compactness of the assumption space is maximised. In particular it may be beneficial to keep as many $\mathcal{B}$'s for the assumptions that less constrained than others; this needs to be explored more fully but may be computationally impractical[3].

Forking on the connection machine takes advantage of the *packet-routing network*, therebye acting across the subspaces in parallel, copying the appropriate subspaces to previously unactivated processors. Within the DAP, we are constrained by the physical shape of the parallel vectors; the method used to achieve parallel forking on the DAP, having isolated all the subspaces that contain no $\mathcal{F}$ values for the antecedents of the justification, is to copy all the subspaces containing $\mathcal{B}$ (for a particular antecedent) by first finding the extremities of the these subspaces in SSA and then copying the whole block to a previously unoccupied part of SSA; the corresponding elements of the SSMASK are set to $\mathcal{T}$ to activate the copied subspaces. The appropriate changes are made to the $\mathcal{B}$ values for each of the blocks in SSA.

This process of copying followed by the deactivation of subspaces for nogood justifications will give rise to a sparse array of subspaces and so when the total available store is about to be filled up, the whole array is *packed* to the left to allow room for further forking; this packing is achieved by using the technique refered to as *recursive doubling* which enables an n wide array to be packed in $O(\log n)$. It transpires that packing is is the most costly operation, potentially taking in excess of 50% of the processing time. The above method will work until the available store becomes too full for the efficient forking of subspaces; we consider this later.

[3]This is related directly to the subsumption check mentioned earlier.

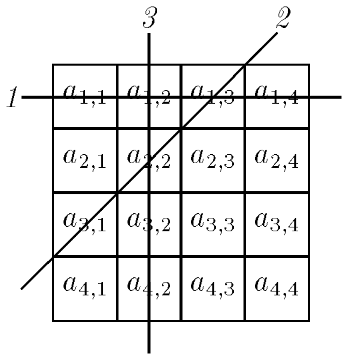

Figure 3: 4-queen problem

4 Mutual Exclusions

In many problems, the mutual exclusion of a set of assumptions is required; that is, only one assumption in the set is true at a time. They can be used to describe many problems in a natural way:

Example 2 *We consider solving the n-queen problem where $n = 4$, say; this is the problem of placing n queens on an $n \times n$ chess board such that no queen attacks another - this means that no two queens can be placed on the same row, same column or same diagonal.*

The first thing we require is n^2 assumptions; one for each square of the board. We then proceed by adding the constraints that define the problem (see figure 3):

1. *We can view this problem as variable assignment problem where there are n variables representing the n rows and for each row there are n possible assignments representing the n columns. That is, we require that for each row there is one and only one queen;*

$$\neg a_{1,1} \wedge \neg a_{1,2} \wedge \neg a_{1,3} \wedge \neg a_{1,4} \rightarrow \perp$$

   ensures that there exists at least one queen in the first row and the mutual exclusion of the set $\{a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}\}$ further specifies that only one queen is in the first row; this is repeated for each row.

2. *We then require that the queens don't attack on the diagonals; again we can use a mutual exclusion, this time ensuring that on each of the diagonals there is no more than one queen is placed: $\{a_{3,1}, a_{2,2}, a_{1,3}\}$*

3. *Finally we need to insist that no more than one queen is placed in each column with a mutual exclusion: $\{a_{1,2}, a_{2,2}, a_{3,2}, a_{4,2}\}$*

This, therefore, fully specifies the problem; in the next section we consider problems of efficiency.

The effect of a mutual exclusion can be achieved by using the current ATMS functions by saying that each pair of assumptions in the set is a nogood. This means that for n assumptions we require $\frac{n^2-n}{2}$ nogoods.

An alternative to this would be to ensure that for each subspace there is at most one assumption in the set that has the value $\mathcal{T}$. This gives us three possible cases for the values of the assumptions within a given subspace:

1. There is one $\mathcal{T}$: change all the $\mathcal{B}$'s to $\mathcal{F}$.
2. There are more than one $\mathcal{T}$: remove it from the SSMASK.
3. There are no $\mathcal{T}$'s: For n $\mathcal{B}$'s there must be n copies where one must be set to $\mathcal{T}$ and the rest are set to $\mathcal{F}$.

It is the last of these cases that is expensive to deal with; if on the other hand we can guarantee that there exists at least one $\mathcal{T}$ in every subspace then it will never arise. Fortunately, it is often the case where not only is the mutual exclusion required but the insistence that exactly one assumption is true; this is the *choose* function of deKleer's. Therefore, before the mutual exclusion is added, the disjunction of the assumptions is added; this is achieved by making the negation of all the assumptions a nogood.

On the other hand, in the case where only the mutual exclusion is required it may be possible to delay forking as mentioned above in *algorithm 3*.

In its simplest form the detection on the number of $\mathcal{T}$'s is done using a parallel counter across all the subspaces (treating the $\mathcal{B}$'s as $\mathcal{F}$).

Given this description of a *mutual exclusion*, it may be useful to generalise it to allow at most n assumptions can be true at a time. In the case of fault diagnoses for, say, digital circuits it is desirable to find the diagnosis with the minimal number of faulty components. It has been shown [HC93] that it could be more efficient to run the problem looking for a single faulty gate and then if this fails to deliver the desired result then the problem is rerun but by applying a mutual exclusion that limits the number of faulty gates to two and so on and so forth. This may well solve the problem of an explosion in the number of assumption spaces; in the case of circuit diagnoses, given an efficient representation for the parallel ATMS the number of subspaces is exponential w.r.t. the number of internal connections.

Generalisation is achieved by adapting the above method but given the value n for the maximum true assumptions then changing the the $\mathcal{B}$'s to $\mathcal{F}$ if there are exactly n $\mathcal{T}$'s and removing those subspaces where there are more than n $\mathcal{T}$'s. For the case of less than n $\mathcal{T}$'s we can again adopt the above strategy of fork delaying to remove that expence.

$$\begin{array}{c|c} a_1 & \mathcal{B} \\ a_2 & \mathcal{B} \\ n & \mathcal{F} \end{array} \stackrel{[a_1,a_2]\to n}{\Longrightarrow} \begin{array}{ccc} \mathcal{T} & \mathcal{F} & \mathcal{T} \\ \mathcal{F} & \mathcal{B} & \mathcal{F} \\ \mathcal{F} & \mathcal{T} & \mathcal{T} \end{array} \stackrel{[a_1,\neg a_2]\to n}{\Longrightarrow} \begin{array}{ccc} \mathcal{T} & \mathcal{F} & \mathcal{T} \\ \mathcal{F} & \mathcal{B} & \mathcal{F} \\ \mathcal{T} & \mathcal{T} & \mathcal{T} \end{array}$$

Figure 4: The addition of resolvable justifications

The counting of the number of $\mathcal{T}$'s can be improved upon if we are only interested in the original form of mutual exclusion where only one proposition is allowsd to be true at a time. We require only two logical masks instead of a counter, mask1 and mask2; mask1 records those subspaces where there exists at least one $\mathcal{T}$ and mask2 records those where there are more than one $\mathcal{T}$; mask1 is created by taking the disjunction of the assumption rows (the $\mathcal{B}$'s are counted as $\mathcal{F}$ here) while mask2 created by taking the disjunction of itself with the conjunction of each given assumption row and the mask1 for the rows before (again the $\mathcal{B}$'s are counted as $\mathcal{F}$).

Having produced these masks, appropriate changes can be made to both SSA, grounding to $\mathcal{F}$ those assumptions that have the value $\mathcal{B}$,and SSMASK, removing those where more than one assumption is $\mathcal{T}$.

5 Returning Extensions

DeKleer defines the extension of an assumption or node as being the points where the assumption or node is $\mathcal{T}$; that is, as set of sets of truth assignments to assumtions for the given assumption or node. Unfortunately this does not always produce minimal extensions.

Example 3 *Given a problem with two assumptions, a_1 and a_2, and one node, n, if we add the justifications $[a_1, a_2] \to n$ and $[a_1, \neg a_2] \to n$ then we get the ATMS in figure 4:*

From this we see that there are two points in the extension for n but they are not minimal; they represent one minimal point where n is dependant on a_1.

The process of finding the minimal sets of assumptions that represent the node's extension involves resolving the different subspaces until no more can be resolved, where for each resolution it may possible to discarded either or both of the resolvants.

Definition 1 *Given two subspaces, then a* resolvable *assumption is one that has the value $\mathcal{T}$ in one subspace and $\mathcal{F}$ in the other.*

Two subclasses can be resolved if they contain one resolvable assumption. To resolve them the resolvable assumption is set to $\mathcal{B}$ in each resolvant and then they are *merged* such that if the values are $\mathcal{T}$ and $\mathcal{B}$ (or $\mathcal{F}$ and $\mathcal{B}$) respectively then value is set to $\mathcal{T}$ (or $\mathcal{F}$). Two subclasses can also be resolved if they contain no resolvable assumptions; this is only constructive if one subspace subsumes the other:

Definition 2 *One subspace* subsumes *another if the value for every assumption in the second subspace is the same as the corresponding value in the first or the value in the first is* $\mathcal{B}$ *.*

Obviously, then, if two subspaces subsume each other then they are the same subspace. If there is no resolvable assumption for two subspaces then either one or both subsume the other.

To decide which resolvants can be discarded we follow the following rules:

> Given two resolvable subspaces SS1 and SS2 then remove the resolvable assumption producing SS1′ and SS2′ respectively.
>
> If SS1′ subsumes SS2′ then discard SS2.
>
> If SS2′ subsumes SS1′ then discard SS1.
>
> (if both the above hold then discard both SS1 and SS2)

This process is repeated until no more can be resolved; we can guarantee that the process terminates if we insist that two subspaces are resolved only if SS1′ subsumes SS2′ or visa versa. It is then up to the problem solver to deduce any other minimal subspaces if required:

Example 4 *If the subspaces for node, n_1, are:*

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ a_1 & \mathcal{F} & \mathcal{T} & \mathcal{F} & \mathcal{T} \\ a_2 & \mathcal{T} & \mathcal{F} & \mathcal{F} & \mathcal{T} \\ a_3 & \mathcal{T} & \mathcal{T} & \mathcal{T} & \mathcal{F} \end{array}$$

the first and last subspaces resolve leaving:

$$\begin{array}{c|ccc} & 1,4 & 2 & 3 \\ a_1 & \mathcal{F} & \mathcal{T} & \mathcal{F} \\ a_2 & \mathcal{T} & \mathcal{F} & \mathcal{F} \\ a_3 & \mathcal{B} & \mathcal{T} & \mathcal{T} \end{array}$$

and then resolving the second two subspaces we get:

$$\begin{array}{c|cc} & 1,4 & 2,3 \\ a_1 & \mathcal{F} & \mathcal{B} \\ a_2 & \mathcal{T} & \mathcal{F} \\ a_3 & \mathcal{B} & \mathcal{T} \end{array}$$

These are therefore an abbreviation for:

$$\neg a_1 \wedge a_2 \rightarrow n_1 \ \textit{and} \ \neg a_2 \wedge a_3 \rightarrow n_1$$

The user can then resolve these to also get:

$$\neg a_1 \wedge a_3 \rightarrow n_1$$

To find out whether an assumption is resolvable between two subspaces, the conjunction of both subspaces for the two components of the assumption is taken. If neither are true then it is a resolvable assumption.

If there is one or less resolvable assumptions then they can merged. This is achieved by setting the resolvable assumption to $\mathcal{B}$ and then taking the conjunct of both subspaces for each component of each subspace.

Finally, if the resultant is equal to one of the merging subspaces then that subspace is subsumed by the other and hence the appropriate resolvant can be discarded (as described above).

This work can be carried out in the workspace described in section reflps. It is possible to see if one subspace resolves with any one of the other subspaces in one go. Therefore to find two resolvable subspaces would take $O(n)$ and therefore the entire operation would be $O(n^2)$.

As we can see, this process of calculating the minimal sets of assumptions that make an extension could be very costly. Fortunately, in many applications, there is no need to produce these minimal sets.

6 Large Problems Spaces and Efficiency

[DdK88] solved the 13-queen problem on a connection machine with 16k processors where each processor is divided into a number virtual processors, say 4 or 8, therebye making the maximum number of subspaces equal to 64k or 128k respectively; this problem produces 73,712 solutions.

It is clear that the number of available subspaces will be a problem when considering large problems and, as shown above, even for a modest n-queen problem a very large number of subspaces is required for the final solution. In addition to this the number of subspaces required during the calculation will normally be far in excess of the final solution space. Of course this heavily relies on the order that the justifications that define any given problem are presented to the ATMS. In the n-queen problem, the maximum number of subspaces required, if the justifications and mutual exclusions where added in the most naive way, would be n^n or, in the case of the 13-queen problem, 302,875,106,592,253.

There are a number of techniques to try and reduce this problem; Dixon and DeKleer discuss two:

1. *delay* the forking, recording that the justification must be reapplied before any inquiry is made of the ATMS; the application of a subsequent nogood justification way well reduce the number of subspaces leaving room for the delayed justification. It may not be possible to always achieve this method. They use fork delaying technique in their *algorithm 3*.

2. revert to a sequential ATMS for each of the subspaces.

It is clear that the second of these is only required if the final solution space is larger than the available space. Alternative solutions may be to store parts of the solution space externally but this would prove to be very expensive.

The first technique is more to limit the explosion of the solution space during the application of large sets of justifications that define any given problem.

For the implementation on the DAP, because of the relatively small number of processors compared with the connection machine, a more sophisticated strategy for delaying justification application is employed; in addition to ASS we also have a workspace, ASSWS, which is the same size as ASS except the parallel dimension is equal to the size of the dap, along with the appropriate ASSWSMASK. We also have, in addition to ASSMASK, a mask that marks a buffer, ASSBUFMASK; these two masks should always be disjoint (this can be achieved by storing at either end of the parallel vectors) and if not then there is an overflow.

Therefore given a new set of justifications and mutual exclusions the current assumptions spaces are put into a buffer. These are then processed in dapsize groups until the buffer is empty:

1. a set of assumption spaces is copied into a dapsize workspace.

2. the justifications and mutual exclusions are applied as described in sections 3 and 4. If the there is not enough room in this workspace then the excess subspaces are copied into the buffer.

3. when the assumptions in the workspace are complete then they are copied out to the solution area.

When the subspaces are to be copied from the buffer to ASSWS then the last subspaces to be stored in the buffer are taken in preference.

To improve the performance of the above method, on the application of each justification, all the remaining justifications are applied with the forking *delayed* along with the mutual exclusions, therebye pruning out unwanted subspaces.

Given this method of fork delaying, the n-queen problem can be treated as a variable assignment problem where each row is considered as a variable and each position in the row as a posible assignment to that row. The optimal search can be produced by considering the assignment of the most constrained variables first. In this problem these are the rows nearest the centre. So we supply these justifications to the ATMS first in the order just specified and then all the others concerning the diagonals and the columns afterwards. We can also get a large improvement by adding extra justifications for the column saying that at least one queen must exist in each column; these are redundant in theory but have a significant effect.

7 Satisfiability Engine vs. ATMS

Given the limitations that we mentioned above to do with the size of problems it maybe reasonable with some problems to sacrifice the production of all models simultaneously along with the incremental nature of the ATMS and produce a satisfiability engine that hase the same semantics as the parallel ATMS but generates the solutions in a depth first search method. The effect of this is to greatly increase the size of problems that can be tackled because of the reduction in size of the subspace array, SSA.

At a simple level this can be done by taking the ATMS and stopping whenever a subspace is produced that satisfies all the constraints and delivering this back, only proceeding when further subspaces are requested; it is often the case that these subspaces will be completed in groups producing a fairly eratic amount of behaviour.

It is, on the otherhand, possible to partially revert to Dixon and deKleer's algorithm 1 method; one of the most expensive parts of the method described so far in producing the ATMS was the forking mechanism. An alternative strategy is, instead of fully using the algorithm 1 method, is to use the algorithm 2 method for storing subspaces but, instead of applying justifications that give rise to the forking, to then work through the assumptions in the ASSWK giving all posible truth assignments to as many assumptions as possible before the ASSWKMASK becomes full.

To clarify this, if we consider a state where there are, for example, three activated subspaces in the SSAWKMASK, then, for a DAP 510 which has 1024 parallel processes and is accordingly the size of the SSAWKMASK, we can first duplicate these three subspaces 256 ($256*3 < 1024 < 512*3$) times and then for each of these copies we can give a unique truth assignment for eight ($= \log_2 256$) previously unassigned assumptions.

At this stage, all the justifications should be applied (with their forking inhibited!) and hopefully a large number of the subspaces with be deactivated (During this process, a number of assumptions which have not been previously assign to in the above duplication stage will be grounded within particular subspaces. At a later point when this assumption is being assigned to, for speed and simplicity, it can be ignored that they have been grounded and should be treated as still having the value $\mathcal{B}$. This will not affect the final outcome because the justifications will be applied again and consistency with again be maintained.)

The remaining subspaces are then packed to beginning of the array and the process is repeated until there is no room to duplicate the ative subspaces completely. At such at time, if all the assumptions have not been dealt with, then some of the subspaces will need to be copied to the SSA to be processed at a later time if required while the algorithm immediately continues with the rest.

The number of subspaces that are left behind in this process can vary but there is clearly a trade off between time and space. If too many are processed at each stage then the number of subspaces that are buffered in ASS will tend to grow to quickly and is this aspect of the storage that need to be controlled. Fortunately we know that the problem will completed for a given subspace when all the justifications are assigned to and therefore it is possible vary the number of subspaces that are processed at each stage by refering to the proportion of unassigned assumptions outstanding.

A further improvement can be produced if it can be recognised that the problem is a variable asignment problem, either completely or partially; given that we are assigning the assumtions contained in the domain for a particular variable (eg. a row of the n-queen problem) then we require to only assign one assumption $\mathcal{T}$ in any given subspace. This clearly saves the waste of doing a complete assignment and then pruning out immediately all the subspaces where more than one $\mathcal{T}$ assignment has been made.

It transpires that both the above methods of assignment are very cheap to implement on the DAP; they lend themselves naturally to the architecture.

8 Conclusion

Tests of the ATMS on the DAP have been promising in comparison with Dixon and deKleer's ATMS who solve the 13-queen problem in 60 seconds on a 16K CM1 as opposed to 4235 seconds for a serial implementation on a LISP symbolic machine. The timing on the DAP 510 is about 300 seconds; 5 times slower than the CM1 on a machine with 16 times less parallelism.

Unfortunately, it is clear that the limit of the ATMS is reached with such a problem. A major problem with the parallel representation of the ATMS is highlighted by the fact that if two imdependant problems based on different assumptions were being solved and they gave rise to m and n subspaces respectively then the total number of suspaces would be $m * n$ producing an explosion in the amount of space required; this is not a problem in the serial ATMS because of the way that ATMS contexts are implemented; each fact has a label containing a set of environments where an environment is a set of assumptions that support the belief of that particular fact. Therefore equivalent models that are explicitly stored in the parallel ATMS have to calculated. This means that coexisting problems in different domains do not cause an explosion in required memory. It is therefore clear that the parallel ATMS should be restricted to single self-contained problems.

It was for this reason that the production of a satisfiability engine was explored allowing much larger problems to be tackled given the limitations of space. It allowed, for instance, solutions upto the 60-queen problem in a few minutes.

Unfortunately (understandedly) there is still a large gap between the performance and size of problems that can attempted of this type of system and incomplete satisfiability engines, like GSAT [SLM92], for certain applications; GSAT can solve the 100-queen problem which can be improved upon by again including a primitive mutual exclusion construct to produce solutions for the 500-queen problem.

Interestingly enough, where we observed above that the addition of extra 'true' facts can improve the performance of the ATMS, (eg. the justifications that states that there exists at least one queen in every column) it causes GSAT an enourmous amount of problems, significantly increasing the amount of time required to find a solution.

In addition to this, Kautz and Selman [KS92] show that for some toy planning problems, a Davis-Putnam method [DP60] for satisfiability works much better than their incomplete method thereby showing that standard tree searching method can be useful. This is further demostrated in [MJPL90] where solutions to the 1000000-queen problem have been solved by both a complete and an incomplete method.

References

[DdK88] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. In *Proc. 2nd International Workshop on Nonmonotonic Reasoning*, pages 131–143, Grossau, FRG, 1988.

[dK86a] J. de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[dK86b] J. de Kleer. Extending the atms. *Artificial Intelligence*, 28:163–196, 1986.

[dK86c] J. de Kleer. Problem solving with the atms. *Artificial Intelligence*, 28:197–224, 1986.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7:201–215, 1960.

[HC93] M. S. Hopkins and M. R. B. Clarke. Tableau theorem-proving, default logic and the parallel atms. Technical report 657, QMW, 1993.

[KS92] H.A. Kautz and B. Selman. Planning as satisfiabilty. In *ECAI–92*, pages 359–363, 1992.

[MJPL90] S. Minton, M. D. Johnson, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *AAAI–90*, pages 17–24. MIT Press, 1990.

[SLM92] B. Selman, H.J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI–92*, 1992.