Computer Architecture for Solving Consistent Labelling Problems

JULIAN R. ULLMANN
Department of Computer Science, University of Sheffield

ROBERT M. HARALICK
LINDA G. SHAPIRO
Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061

Consistent labelling problems are a family of NP-complete constraint satisfaction problems such as school timetabling, for which a conventional computer may be too slow. There are a variety of techniques for reducing the elapsed time to find one or all solutions to a consistent labelling problem. In this paper we discuss and illustrate solutions consisting of special hardware to accomplish the required constraint propagation and an asynchronous network of intercommunicating computers to accomplish the tree search in parallel.

1. INTRODUCTION

School-timetabling, subgraph isomorphism, graph colouring, propositional theorem proving, and scene labelling problems can be formulated as special cases of the consistent labelling problem.[1] The consistent labelling problem is NP-complete, which means that in the worst case we may have to resort to exhaustive enumeration in order to find a solution, and the time needed for this enumeration may increase exponentially with the number of variables. Despite the possibility of this combinatorial explosion, problems such as school timetabling have to be solved in practice. Haralick and Elliott[5] have shown that the combinatorial explosion of consistent labelling can be mitigated by pruning the search tree.

The present paper is theoretical, and its primary purpose is to show how the successful tree-pruning technique of Haralick and Elliott can be implemented using various forms of parallelism to reduce the elapsed time for solving consistent labelling problems. The present paper also generalises the formulation of the consistent labelling problem as an N-ary constraint satisfaction problem, where N may be different for different constraints. In previous papers (Haralick and Shapiro, 1979),[1] N has been the same for all constraints.

Earlier papers on special architecture for solving the consistent labelling problem include Cherry and Vaswani, who had actually built special architecture for a boolean satisfiability problem[3] (which is a consistent labelling problem (Haralick and Shapiro, 1979)).[1] We believe, however, that the general possibilities of using special architecture to soften the practical effects of the combinatorial explosion have not previously been explored adequately. For example, Schmidt and Strohlein[4] remark that 'recent developments in computer technology and software engineering have not yet reached the area of timetable programming'.

2. A FORMULATION OF THE CONSISTENT LABELLING PROBLEM

Let U be a set of objects called units, and L be a set of possible labels for those units. Let $T \subseteq \{f \mid f \subseteq U\}$ be the collection of those subjects of units from U that mutually constrain one another. That is, if $f = \{u_1, u_2, \ldots, u_k\}$ is an element of T, then not all possible labellings of $u_1, \ldots, u_k$ are legal labellings. Thus there is at least one label assignment $l_1, l_2, \ldots, l_k$ so that u_1 having label l_1, u_2 having label l_2, ..., u_k having label l_k is a forbidden labelling. T is called the unit constraint set. Finally, let $R \subseteq \{g \mid g \subseteq U \times L$, g single-valued, and $\mathrm{Dom}(g) \in T\}$ be the set of unit-label mappings in which constrained subsets of units are mapped to their allowable subsets of labels. If $g = \{(u_1, l_1), (u_2, l_2), \ldots, (u_k, l_k)\}$ is an element of R, then $u_1, u_2, \ldots, u_k$ are distinct units, $\{u_1, u_2, \ldots, u_k\}$ is an element of T meaning $u_1, u_2, \ldots, u_k$ mutually constrain one another, and u_1 having label l_1, u_2 having label l_2, ..., and u_k having label l_k are all simultaneously allowed.

In the consistent labelling problem, we are looking for functions that assign a label in L to each unit in U and satisfy the constraints imposed by T and R. That is, a consistent labelling is one which when restricted to any unit constraint subset in T yields a mapping in R. In order to state this more precisely, we first define the restriction of a mapping. Let $h: U \to L$ be a function that maps each unit in U to a label in L. Let $f \subseteq U$ be a subset of the units. The restriction $h|_f$ (read 'h restricted by f') is defined by $h|_f = \{(u, l) \in h \mid u \in f\}$. With this notation, we define a consistent labelling as follows.

A function $h: U \to L$ is a consistent labelling if and only if for every $f \in T$, $h|_f$ is an element of R.

A simple example

Suppose the inputs to the problem are as follows:

$U = \{1, 2, 3, 4, 5\}$

$L = \{a, b, c\}$

$T = \{\{1\},$ unary constraint
$\{1, 2\},$ binary constraints
$\{2, 5\},$
$\{1, 3, 4\}\}$ ternary constraint

$R = \{\{(1, a)\}, \{(1, b)\},$ unary constraint
$\{(1, a), (2, a)\},$ binary constraints
$\{(1, a), (2, b)\},$

$\{(1, b), (2, b)\}$,

$\{(2, a), (5, a)\}$,

$\{(2, b), (5, c)\}$,

$\{(1, a), (3, a), (4, c)\}$, ternary constraints

$\{(1, b), (3, a), (4, a)\}\}$

Then $h = \{(1, a)\ (2, a)\ (3, a)\ (4, c)\ (5, a)\}$ is a consistent labelling. To see this note that

$h|_{\{1\}} = \{(1, a)\}$, $h|_{\{1, 2\}} = \{(1, a), (2, a)\}$, $h|_{\{2, 5\}} = \{(2, a), (5, a)\}$, and

$h|_{\{1, 3, 4\}} = \{(1, a), (3, a), (4, c)\}$ are all elements of R.

Simplified examples of practical consistent labelling problems

In the Appendix we show that a school timetabling problem and a three-dimensional packing problem can be formulated as consistent labelling problems. Further examples of consistent labelling problems are given by Haralick and Shapiro.[1]

The practical usefulness of a consistent labelling formulation depends on the actual constraints that are employed. Subgraph isomorphism is an example in which careful scrutiny of graph-theoretic factors has experimentally yielded greater efficiency than can be obtained using the simplest consistent labelling formulation.[9] The formulations given in the Appendix to the present paper are merely simple examples; and we are not now concerned to explore application-specific refinements that may enhance efficiency. Instead we are concerned, in Sections 4, 5 and 6, with introducing parallelism to reduce elapsed time for all consistent labelling problems.

In the next section we introduce a natural generalisation of the forward checking algorithm of Haralick and Elliott.[5] Our algorithm is identical to the forward checking algorithm in the case where every constraint involves exactly two units.

3. A CONSTRAINT PROPAGATION ALGORITHM FOR CONSISTENT LABELLING PROBLEMS

In principle consistent labellings can be found by exhaustively checking whether each possible assignment of exactly one label to each unit satisfies all the constraints. This inefficient brute-force method can be organised as a backtrack search.

To improve efficiency we check for satisfaction of constraints after each successive unit has been instantiated (i.e. has had a single label assigned to it) in the course of a backtrack search. For the simple example given above in Section 2, suppose that the units are instantiated in the sequence 1, 2, 3, 4, 5. After units 1 and 2 have been instantiated, the labels *currently* permitted for each unit might, for example, be as shown in Table 1.

Table 1. An example of the relation H

Unit	Label
1	a
2	a
3	a, b, c
4	a, b, c
5	a, b, c

In Table 1, units 3, 4 and 5 have not yet been instantiated: that is why they have more than one label. If *at this stage* the constraints cannot be satisfied then we can omit that part of the tree search which would proceed to instantiate units 3, 4 and 5. Our algorithm uses this principle to prune the search tree.

The unit that has just been instantiated is the *current* unit; and units that have not yet been instantiated are *future* units. We denote the set of future units by UF. After the current unit has been instantiated, we check only for satisfaction of constraints belonging to a set Q defined by

$Q = \{f \in T \mid (f \cap UF) \neq \emptyset$ and f contains the current unit$\}$

Thus Q is the subset of T in which each constraint includes the current unit and at least one future unit.

We denote by H, $H \subseteq U \times L$, the relation or unit-label table of possible label assignments currently permitted for each unit. Table 1 is a simple example of a relation H. If a unit u is not in UF then

(1) $(u, l) \in H$ implies that l is the instantiated label for unit u.

(2) $(u, l) \in H$ and $(u, n) \in H$ imply $n = l$, since only one label can have been assigned to an instantiated unit.

If $u \in UF$ then u is a unit yet to be instantiated, and $H(u)$ is the set of labels still permitted for unit u. The algorithm is designed so that H is always defined everywhere; that is, every unit always has at least one possible label ($H(u) \neq \emptyset$ for every u).

After instantiating the current unit, the result of constraint checking is to delete labels that cannot possibly belong to any consistent labelling that is a subset of the current H. The result is a new H given by

$$H := \bigcap_{f \in Q} R(H, f)$$

where $R(H, f)$ is the set of unit-label pairs that constraint f does not rule out. That is,

$$R(H, f) = \{(u, l) \in H \mid (u \notin f) \quad \text{or} \quad (u \in f)$$

and there exists

$$g \in R \quad \text{with} \quad (u, l) \in g,$$

satisfying

$$\text{dom}\,(g) = f \quad \text{and} \quad g \subseteq H\}$$

If $u \in f$ then $(u, l) \in R(H, f)$ only if l belongs to at least one of the labellings of all the units in f allowed by R and included in H.

From the definition of consistent labelling it immediately follows that a function $h: U \to L$ satisfies

$$H = \bigcap_{f \in T} R(h, f)$$

if and only if h is a consistent labelling. When the algorithm yields a relation H that comprises exactly one label for each unit, H is recognized as a consistent labelling. At an intermediate stage, before there is exactly one label for each unit, the current H must contain all the consistent labellings that can possibly be produced from it. If there were any unit in H having no label then no consistent labelling could possibly be produced from H;

and our algorithm would therefore not try further instantiation within H. Our algorithm is a *constraint propagation* algorithm because deletion of one inconsistent label may cause deletion of another, which may cause deletion of another, and so on.

The algorithm can be implemented using a recursive procedure TREESEARCH for which the inputs are

UF: the set of units requiring labels (initially all the units),
T: the unit constraint set,
R: the allowable unit-label mappings, and
H: the partial or incomplete labelling (initially $H = \{(u, l) \in U \times L \mid (u, l)$ is not ruled out by any constraint in $R\}$.

Note that all unary constraints have been removed from T and used to produce the initially constrained H.

The predicate ISEMPTY returns **true** if its argument is the empty set and **false** otherwise. The predicate DEFINED_EVERYWHERE returns **true** if its argument is a set of unit-label pairs including every unit and **false** otherwise. The procedure OUTPUT outputs a mapping. The function DELETEFIRST removes and returns the first element of its argument set. The function RESTRICT inputs a binary relation H, a unit u and a label l and returns a subset of H consisting of all pairs (v, m) such that if $v = u$, then $m = l$. Procedure TREESEARCH is given below.

```
procedure TREESEARCH (UF, T, R, H)
local u, Q, S, l, H_{u,l}, H'
by-reference T, R, H,
by-value UF
u := DELETEFIRST(UF):
Q := {f ∈ T | f ∩ UF ≠ ∅ and u ∈ f}
S := {l | (u, l) ∈ H};
while not ISEMPTY(S) do
  begin
    l := DELETEFIRST(S);
    H_{u,l} := RESTRICT(H, u, l)
    H' := ⋂_{f∈Q} R(H_{u,l}, f)
    if DEFINED_EVERYWHERE (H')
    then if SINGLE_VALUED(H')
      then OUTPUT(H')
      else call TREESEARCH(UF, T, R, H') endif
    endif;
  end;
return;
end TREESEARCH
```

For the simplified example of Section 2, we have initially

$UF = U = \{1, 2, 3, 4, 5\}$,
$T = \{\{1, 2\}, \{2, 5\}, \{1, 3, 4\}\}$
(the unary constraints have been removed, since they will be used to determine the initial H),
$R = \{\{(1, a), (2, a)\}, \{(1, a), (2, b)\}, \{(1, b), (2, b)\}, \{(2, a), (5, a)\}, \{(2, b), (5, c)\}, \{(1, a), (3, a), (4, c)\}, \{(1, b), (3, a), (4, a)\}\}$
(the unary unit-label pair sets have been removed here also),
and
$H = \{(1, a), (1, b), (2, a), (2, b), (3, a), (4, a), (4, c), (5, a), (5, c)\}$
(the initial R was used to determine the legal labels for each unit).

In the first call to TREESEARCH, u is set to 1, UF to $\{2, 3, 4, 5\}$, and S to $\{a, b\}$. Next l is set to a, S reduced to $\{b\}$, and $H_{u,l}$ becomes $\{(1, a), (2, a), (2, b), (3, a), (4, a), (4, c), (5, a), (5, c)\}$. Now the constraint propagation calculates

$$R(H_{u,l}, \{1, 2\}) = \{(1, a), (2, a), (2, b), (3, a), (4, a), (4, c), (5, a), (5, c)\}$$

$$R(H_{u,l}, \{2, 5\}) = \{(1, a), (2, a), (2, b), (3, a), (4, a), (4, c), (5, a), (5, c)\}$$

$$R(H_{u,l}, \{1, 3, 4\} = \{(1, a), (2, a), (2, b), (3, a), (4, c), (5, a), (5, c)\}.$$

Thus the intersection H' becomes

$$H' = \{(1, a), (2, a), (2, b), (3, a), (4, c), (5, a), (5, c)\}.$$

Since H' is defined everywhere but not single-valued, TREESEARCH is called again. This time we have

$UF = \{2, 3, 4, 5\}$, T and R remain the same, and

$$H = \{(1, a), (2, a), (2, b), (3, a), (4, c), (5, a), (5, c)\}.$$

In this activation, u becomes 2, UF becomes $\{3, 4, 5\}$, S becomes $\{a, b\}$, l becomes a, S is reduced to $\{b\}$, and $H_{u,l}$ becomes $\{(1, a), (2, a), (3, a), (4, c), (5, a), (5, c)\}$. Now the constraint propagation calculates

$$R(H_{u,l}, \{2, 5\}) = \{(1, a), (2, a), (3, a), (4, c), (5, a)\}.$$

$R(H_{u,l}, \{1, 3, 4\})$ is not calculated since unit 2 is not an element of $\{1, 3, 4\}$. The intersection H' becomes $\{(1, a), (2, a), (3, a), (4, c), (5, a)\}$. It is defined everywhere and single-valued, so a consistent labelling has been found. The procedure will go on to find a second consistent labelling also.

The next section introduces various forms of parallelism for speeding up the solution of consistent labelling problems.

4. COMBINATIONAL CONSTRAINT CIRCUITS

In procedure TREESEARCH, the computation of $R(H_{u,l}, f)$ involves at least $w_f = |f| \times |\{g \in R$ and dom $(g) = f\}|$ logic operations. Thus the computation of

$$\bigcap_{f \in Q} R(H_{u,1}, l) \quad \text{involves} \quad \sum_{f \in Q} w_f$$

logic operations, normally done serially. To speed up the computation, these logic operations can be done in parallel outside the CPU in a combinational constraint network as shown in Fig. 1. This network is of interest because of its simplicity.

We implement H as a bit matrix that has one row per unit and one column per label. The bit in row u and column l is a predicate

$$H(u, l) = (u, l) \in H.$$

Corresponding to each $g \in R$ the network contains an AND gate whose output is

$$g(H) = \underset{(u,l) \in g}{\text{AND}} (H(u, l)) = g \subseteq H.$$

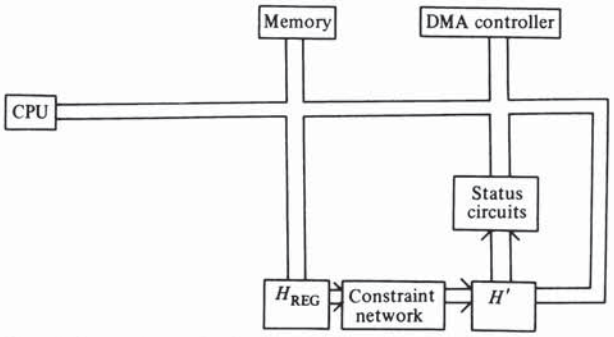

Fig. 1 is a schematic diagram that includes a combinational constraint network that can perform logic operations in parallel outside the CPU.

Corresponding to each triple (f, u, l) the network contains an OR gate whose output is

$$f_{u,l}(H) = \underset{\{g|(u,l)\in g}{\text{OR}} \quad \underset{\text{and}}{} \quad \underset{\text{dom}(g)=f\}}{} (g(H))$$
$$= \text{there exists} \quad g \in R \quad \text{with} \quad (u,l)\in g$$
$$\text{satisfying} \quad \text{dom}(g) = f \quad \text{and} \quad g \subseteq H.$$

This definition can easily be related to our definition of $R(H, f)$. If $f_{u,l}(H) = \emptyset$, then constraint f has ruled out $(u, l) \in H$.

The output of the constraint network is a bit matrix H' that has the same dimensions as H. There is one AND gate for each bit in H'; and the bit in row u and column l of H′ is the predicate

$$H'(u,l) = \underset{\{f\in T|u\in f\}}{\text{AND}} (f_{u,l}(H)) \qquad (1)$$
$$= \{(v,n)\in H \mid (v,n) \quad \text{has not been ruled out by any constraint} \quad f\in T.\}.$$

It is easy to see that H' is exactly the H' computed within the procedure TREESEARCH, although the network does some unnecessary but harmless extra work. For example, if u is the current unit, then the procedure TREESEARCH intersects $R(H, f)$ over $f \in Q$ whereas the network ANDs $f_{u,l}$ (H) over $\{f \in T \mid u \in f\}$ which is a superset of Q. ANDing over Q is all that is really necessary, but the network is simplified by omitting the condition $f \cap UF \neq \emptyset$.

H' is a matrix of product-of-sum-of-product functions of H. This structure makes the constraint network an obvious candidate for programmable logic array implementation. If the PLAs are electronically re-programmable,[6] then our constraint network can be re-used for different consistent labelling problems provided that the dimensions of H are adequate.

Referring now to Fig. 1, H' is the bit matrix of outputs from the constraint network. All the bits of H' are input to status circuits whose two outputs are the predicates DEFINED_EVERYWHERE(H') and SINGLE_VALUED (H') that we have defined and used previously. The provision of these simple status circuits is intended to save CPU time.

H_{REG} is a two-dimensional register that sometimes stores H and sometimes stores $H_{u,l}$, as we shall explain. Recursive CALLs of TREESEARCH automatically stack H', and RETURNs restore the previous H. In Fig. 1, H_{REG} and H' are hardwired to the constraint network, and it is therefore expedient to handle the stacking and unstacking of bit matrices explicitly. For this purpose the memory in Fig. 1 includes, along with the program and variables, blocks (or bit planes) $C_1, C_2, \ldots, C_u, \ldots, C_N$, where N is the number of units. Each of these blocks has the same dimensions as H. At the time of the initial call of the procedure, block C_1 contains the initial H.

To explain how the CPU actually uses the external hardware we now give an appropriately modified version of TREESEARCH with explanatory comments enclosed between '(*' and '*)'. UF is implemented as a bit vector that has one bit for each possible label. $H_{REG}[u]$ is the row of H_{REG} that corresponds to unit u.

```
procedure EXTREESEARCH(UF);
u := DELETEFIRST(UF);
H_REG := C_u; (*block transfer done by DMA*)
S := H_REG[u]; (*bit vector copied from H_REG to memory*)
For each l in S do
begin
  B := bit vector comprising the selected l in S with all
       other bits of B reset to ∅;
  H_REG [u] := B; (*in effect H_REG := H_u,l*)
  (*constraint network computes H' using contents of
  H_REG as a new H*)
  if DEFINED_EVERYWHERE(H') (*CPU gets this
  from status circuits*)
  then if SINGLE_VALUED(H') (*CPU gets this from
  status circuits*)
    then OUTPUT(H') (*CPU reads and encodes H'*)
    else
      begin
        C_u+1 := H'; (*done by DMA*)
        CALL EXTREESEARCH (UF);
        H_REG := C_u; (*done by DMA*)
      end
    endif
  endif
end;
return;
end EXTREESEARCH
```

This formulation is simplified in that complete transfer of blocks, e.g. $H_{REG} := C_u$, is unnecessary when $u > l$. Because units $1, 2, \ldots, u-1$ have already been instantiated we only need to keep copies of rows $u, \ldots, N$ of H. Otherwise if, for example, $u = 6$, then the first four rows of C_5 will be identical to the first four rows of C_4. We can eliminate this inefficiency by appropriate elaboration of EXTREESEARCH.

5. ARRAY PROCESSOR IMPLEMENTATION

School timetabling and most of the consistent labelling problems reviewed in Haralick and Shapiro[1] can be formulated such that the cardinality of f is two for all $f \in T$. Subject to this restriction on T we now introduce a parallel (SIMD) array processor algorithm for computing H' from H. This is faster than a conventional computer implementation, but with greater storage requirements than the Fig. 1 implementation. The array processor implementation has no problem of re-programmability and could easily run on many of the

SIMD processors that are commercially available or described in the literature.[7,8]

Our array processor algorithm processes each $(u, l) \in H$ in turn to evaluate $H'(u, l)$. At any given time, variables u and l have had values assigned to them by this sequential process. If $g \in R$ contains $(u, l) \in H$; that is $g = \{(u, l), (u', l')\}$, and $H(u, l) = 1$, then

$$g(H) = H(u, l) \text{ AND } H(u', l')$$
$$= H(u', l'),$$

and for $f = \{u, u'\} \in T$, we have

$$f_{u,l}(H) = \underset{\{(u',l')|((u,l),(u',l'))\in R\}}{\text{OR}} (H(u', l')) \quad (2)$$

For given $f_{u,l}(H)$, u' is fixed, so (2) can be rewritten

$$f_{u,l}(H) = \underset{l' \in L}{\text{OR}} (H(u', l') \text{ AND } (((u, l), (u', l')) \in R)) \quad (3)$$

Expression (1) can be rewritten

$$H'(u, l) = \underset{u' \in U}{\text{AND}} ((u, u') \notin T \text{ OR } f_{u,l}(H)) \quad (4)$$

To manipulate this into a useful form, note that for any given u'

$$\underset{l' \in L}{\text{OR}} (H(u', l'))$$

has truth value 1, so we can write

$$(u, u') \notin T = (\underset{l' \in L}{\text{OR}} (H(u', l'))) \text{ AND } ((u, u') \notin T)$$

Substituting this into (4), using (3) and factoring we obtain

$$H'(u, l) = \underset{u' \in U}{\text{AND}} (\underset{l' \in L}{\text{OR}} (H(u', l') \text{ AND } ((u, u') \notin T \text{ OR } ((u, l), (u', l')) \in R))) \ldots$$

Our array processor implementation uses $|U| \times |L|$ bit matrices $K_{u,l}$ whose contents remain unchanged during the entire tree search. Each matrix $K_{u,l}$ is such that

$$K_{u,l}(u', l') = ((u, u') \notin T) \text{ OR } ((u, l), (u', l')) \in R)$$

Substituting this into (5),

$$H'(u, l) = \underset{u' \in U}{\text{AND}} (\underset{l' \in L}{\text{OR}} (H(u', l') \text{ AND } K_{u,l}(u', l')))$$
$$= \text{DEFINED\_EVERYWHERE}(H \text{ AND } K_{u,l}) \quad (6)$$

where the AND is of corresponding bits in H and $K_{u,l}$. Thus DEFINED_EVERYWHERE is evaluated by ORing over each row and ANDing these ORs over all rows. In our array processor implementation H' is computed by

for each $u \in U$ **do**

for each $l \in L$ **do**

evaluate (6).

Evaluation of (6) is fast because $K_{u,l}$ is simply obtained from memory, and the AND operation is a parallel AND operation on bit planes.

This implementation is slower than the Fig. 1 system because it processes all (u, l) pairs serially. The Fig. 1 system does not require storage of bit planes $K_{u,l}$, but instead uses a substantial number of combinational gates.

6. NETWORK COMPUTER IMPLEMENTATION

Even with the use of parallel hardware, the execution time may be intolerably slow for man–machine interactive school timetabling or for real-time control of an automatic packing machine. To reduce elapsed time for finding consistent labellings, we can subdivide the search tree into M subtrees and use M separate processors to search these subtrees simultaneously, with no need for any synchronisation between these fully independent processors. Each processor could have its own constraint propagation hardware as discussed in the previous section, or several processors could share the same constraint propagation hardware.

A specific method for this is to partition the label set L into M subsets, $L_1, \ldots, L_M$. The first processor would try to solve the consistent labelling problem, restricting the label assigned to the first unit to come from L_1. The second processor would try to solve the consistent labelling problem, restricting the label of the first unit to come from L_2, and so on. Each processor would, to avoid memory access delays, have its own memory containing copies of all required data and code, and would execute the backtrack algorithm of section 4, thus searching a disjoint subtree.

Unfortunately this simple idea may not make optimal use of the M processors to find all consistent labellings in minimal elapsed time. For it is the case that even if each of the M subsets contains the same number of possible labels, the M processors may not all take an equal amount of time to complete their subtree search, exactly because of differences in the effectiveness of tree pruning. Practical experience with algorithms of this type suggests that the elapsed time for one processor may turn out to be many times greater than that for another. Processors that have finished their work may wait idly for others to finish. Thus by using M parallel processors we may not succeed in reducing the overall elapsed time by a factor of M.

Overall elapsed time could be further reduced by interconnecting the M processors in a computer network as in ref. 2. One of the many possible operating policies is that when a processor completes its subtree search it interrogates all other processors that are still searching and then takes over half of the remaining search of the processor whose search is furthest from completion, leaving this processor to complete only the other half of its subtree search. When this network starts operating, with all processors searching (hopefully) equal-sized subtrees, there is at first no delay due to exchange of messages between processors. When more and more processors finish searching subtrees, more and more messages are exchanged, and this eventually constitutes a significant overhead. To prevent this overhead from exploding, we impose a restriction that no processor ever starts searching a subtree of less than a threshold size: if no subtree greater than or equal to this size is available for a processor than this processor becomes idle and is in effect deleted from the network. The threshold size

should of course be chosen so as to minimise overall elapsed time.

This networking policy depends on splitting the subtree whose search is furthest from completion. How far a processor is from completion of a subtree search is easily determined by the number of as yet uninstantiated units.

Simulation of a variety of network architectures uniformly indicates that all other things being equal, (1) processors should execute the search in a depth-first rather than breadth-first manner so that there are as large subtrees as possible that the busy processors can give to a free processor, (2) busy processors should first hand off subproblems to the free processors least centrally located in the network wherever there is a choice.

7. CONCLUSIONS

In this paper we have given several examples of consistent labelling problems and have generalised the original fixed dimensional relational form of the consistent labelling problem. Using this generalised form, we described a forward-checking-like constraint propagation technique to help perform the tree search required to solve a consistent labelling problem. Then we sketched the design of some special-purpose parallel hardware that executes the constraint propagation. Finally we indicated how the entire design could be done in a multiple CPU network.

Because solving consistent labelling problems is so closely allied to solving general combinational reasoning problems, parallel algorithms and associated computer architectures for their fast solution are important to have in our toolbox. Knowledge of them will be of definite help in creating the parallel algorithms and associated parallel computer architectures for efficiently solving the most general predicate calculus types of problems. The efficient solution of this kind of problem will be the hallmark of the next generation of smart computers. We shall be discussing these issues in a future paper.

REFERENCES

1. R. M. Haralick and L. G. Shapiro, The consistent labelling problem, part 1, *IEEE Transactions Pattern Analysis and Machine Intelligence* **PAMI 1**, 173–184 (April 1979); part II **PAMI 2** (3), 193–203 (1980).
2. O. I. El-Dessouki and W. H. Huen, Distributed enumeration on network computers, *IEEE Transactions, Computers* **C 29**, 818–825 (1980).
3. C. Cherry and P. K. T. Vaswani, A new type of computer for problems in propositional logic, with greatly reduced scanning procedures. *Information and Control* **4**, 155–168 (1961).
4. G. Schmidt and T. Strohlein, Timetable construction – an annotated bibliography. *The Computer Journal* **23**, 307–316 (1980).
5. R. M. Haralick and G. L. Elliott, Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* **14**, 263–313 (1980).
6. M. Tanaka, S. Ozawa and S. Mori, Rewritable programmable logic array of current mode logic. *IEEE Transactions, Computers* **C 30**, 229–234 (1981).
7. A. P. Reeves, A systematically designed binary array processor. *IEEE Transactions, Computers* **C 29**, 278–287 (1980).
8. H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, H. E. Smalley and S. D. Smith, PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions, Computers* **C 30**, 934–947 (1981).
9. G. Fowler, R. Haralick, F. G. Gray, C. Feustel and C. Grinstead, Efficient graph automorphism by vertex partitioning. *Artificial Intelligence* **21**, 245–269 (1983).

APPENDIX A

A school timetabling problem

We consider the scheduling of lessons over a fixed period such as one week, assuming that instructors have already been assigned to lessons. The problem is to assign, to each lesson, a time and classroom satisfying the constraints that (1) any pair of lessons attended by the same instructor or same student must be at different times, and (2) no distinct pair of lessons is assigned to the same time and room. This problem fits the consistent labelling model as follows.

U is the set of all lessons. For example, if there are three history lessons then for each of these there is a separate element of U. L is a set of pairs of the form (time, classroom) which includes all possible lesson times and, for each time, all possible classrooms available at that time. The unit constraint set T consists of unary constraints and binary constraints. The unary constraints are for those lessons that *a priori* cannot be scheduled in a particular (time, classroom) pair. The binary constraints consist of all pairs of distinct lessons since (1) these are constrained not to meet in the same classroom at the same time, and (2) a subset of these are constrained not to meet at the same time. From this we get

$$T = T_1 \bigcup T_2,$$

where $T_1 = \{\{u\} \mid u \in U$ and u cannot be scheduled at time t, *classroom* c for some pair $(t, c) \in L\}$

and $T_2 = \{\{u_1, u_2\} \mid u_1, u_2 \in U$ and $u_1 \neq u_2\}$.

Furthermore, $R = R_1 \bigcup R_{21} \bigcup R_{22}$ where

$R_1 = \{\{(u, l)\} \mid \{u\} \in T_1$ and *a priori* knowledge says that label l can be assigned to $u\}$.

$R_{21} = \{\{(u_1, l_1), (u_2, l_2)\} \mid \{u_1, u_2\} \in T_2$ and $l_1 \neq l_2\}$,

and $R_{22} = \{\{(u_1, l_1), (u_2, l_2)\} \mid \{u_1, u_2\} \in T_2$, there exists a person who must attend both u_1 and u_2, and time$(l_1) \neq$ time$(l_2)\}$.

Other constraints can be added to the model as required. For instance, if there are pairs of lessons that must be given in consecutive hours, we can define

$T_3 = \{\{(u_1, u_2\} \mid u_1, u_2 \in U$ and u_2 must be scheduled the hour after $u_1\}$