Local consistency in parallel constraint satisfaction networks*

Simon Kasif\textsuperscript{a,}\textsuperscript{*}, Arthur L. Delcher\textsuperscript{b}

\textsuperscript{a} Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, USA
\textsuperscript{b} Computer Science Department, Loyola College in Maryland, Baltimore, MD 21210, USA

Received January 1993; revised August 1993

Abstract

In this paper we present several basic techniques for achieving parallel execution of constraint networks. The major result supported by our investigations is that the parallel complexity of constraint networks is critically dependent on subtle properties of the network that do not influence its sequential complexity.

1. Introduction

One of the key problems facing Artificial Intelligence (AI) is performing efficient inferences over large knowledge bases. Viewed in the context of parallel computation this problem brings forth fundamental research issues such as knowledge representation in parallel environments, problem-decomposition methods, parallelization of classes of AI problems, the role of incremental methods, parallel matching, and parallel search strategies that eliminate redundancy.

Our research is aimed at deriving a precise characterization of the utility of parallelism in constraint networks. In this paper, we relate parallel constraint networks to standard models of computation (Parallel Random Access Machines or PRAMs). We present several basic techniques for achieving parallel execution of constraint networks. We are interested primarily in developing a classification of constraint networks according to whether they admit massively

---

\* Originally submitted as a Research Note.
\* Corresponding author. E-mail: kasif@cs.jhu.edu.
parallel execution. We analyze parallel execution for dense constraint networks, chain networks, tree networks, two-label networks, directed support networks and path consistency in general networks. For example, we show that contrary to common intuition, chain networks do admit parallel solutions (as in fact do all acyclic graphs). While the obvious parallel algorithm for local consistency in constraint networks should work well in practice, we would like to obtain lower and upper bounds on the complexity of the problem on ideal parallel machines (such as the PRAM).

This study may have significant practical implications since it should indicate which parallel primitives are fundamental in the solutions of large constraint systems. Once such primitives are implemented in hardware, they execute in essentially constant time for all practical purposes, e.g., parallel prefix on the Connection Machine. The original design of the Connection Machine was motivated by these considerations. The ultimate goal of our research is to produce a set of primitives that are critical to the solution of constraint problems. In a view of parallelism advocated by many AI researchers, we associate a small simple processor with each data element (thus loading data does not require time). In this perspective logarithmic-time parallel algorithms have important implications since they show how to represent the data and perform parallel search in almost constant time for any reasonable size network.

It is generally believed that humans perform many tasks efficiently (i.e., in almost constant time) by exploiting the massive parallelism in the brain (e.g., Feldman’s 100-instruction-step metaphor). One of the main underlying assumptions of the Connectionist school of thought is that their approach is naturally amenable to a parallel (distributed) implementation. Moreover, the claim is often made that the symbolic approach does not have this advantage. Our analysis suggests that parallel implementations are possible for symbolic constraint systems. In fact, we conjecture that both connectionist and symbolic constraint systems share the same limitations which are caused by the structure of the particular problem or algorithm. The major result supported by our investigations reported in this paper is that the parallel complexity of constraint networks is critically dependent on subtle properties of the network that do not influence its sequential complexity.

2. Constraint satisfaction, local consistency and discrete relaxation

Constraint satisfaction networks are used extensively in many AI applications such as planning, scheduling, natural language analysis, truth maintenance systems, and logic programming [2, 7, 11, 19, 24, 27, 30]. These networks use the principle of local constraint propagation to achieve global consistency (e.g., consistent labelling in vision).

A constraint satisfaction network can be defined as follows. Let \( V = \{X_1, \ldots, X_n\} \) be a set of variables. With each variable \( X_i \) we associate a set of labels \( L_i \). Now let \( \{R_{ij}\} \) be a set of binary predicates that define the compat-
ibility of assigning labels to pairs of variables. Specifically, \( R_{ij}(a, b) = 1 \) iff the assignment of label \( a \) to \( X_i \) is compatible with the assignment of label \( b \) to \( X_j \).

The Constraint Satisfaction Problem (CSP) is defined as the problem of finding an assignment of labels to the variables that does not violate the constraints given by \( \{R_{ij}\} \). More formally, a solution to CSP is a vector \((a_1, \ldots, a_n)\) such that \( a_i \) is in \( L_i \) and for each \( i \) and \( j \), \( R_{ij}(a_i, a_j) = 1 \).

A standard approach to model CSP problems is by means of a constraint graph (see e.g., v. [19,23]). The nodes of the constraint graph correspond to variables of the constraint network, and the edges correspond to the binary constraints. In this context, each edge in the constraint graph is labelled with a matrix that shows which assignments of labels to the objects connected by that edge are permitted. In this interpretation CSP can be viewed as generalized graph coloring.

In this paper we also will use an explicit constraint graph representation, which expresses constraint information in a more detailed fashion. Specifically, for any constraint network \( G = (V, E) \), we define its explicit representation \( G_{\text{expl}} = (V', E') \) to be the graph in which

\[
V' = \bigcup_{i=1}^{n} L_i,
\]

\[
E' = \{(a_{ij}, b_{kt}) \mid a_{ij} \in L_j, b_{kt} \in L_t \text{ and } R_{jt}(a_{ij}, b_{kt}) = 1\}.
\]

We illustrate this construction by an example. Assume the set of labels is \( \{0, 1\} \) for each variable. For each variable \( X \) we create two nodes \((X, 0)\) and \((X, 1)\). For each edge connecting variables \( X \) and \( Y \) we connect \((X, a)\) with an arc to \((Y, b)\) iff assigning \( a \) to \( X \) is consistent with assigning \( b \) to \( Y \). An example is shown below for the relation \( R_{XY} \).

![Constraint graph example](attachment:constraint_graph_example.png)

Since CSP is known to be \( \mathcal{NP} \)-complete, several local consistency algorithms have been used extensively to filter out impossible assignments.

Arc consistency allows an assignment of a label \( a \) to an object \( X \) iff for every other object \( X' \) in the domain there exists a valid assignment of a label \( a' \) which does not violate the constraints [19,23]. More formally, we define arc consistency as follows.

Given a constraint network, a solution to the local version of CSP or arc consistency (AC) is a vector of sets \((M_1, \ldots, M_n)\) such that \( M_i \) is a subset of
Li and label \(a\) is in \(M_i\) iff for every \(M_j\), \(i \neq j\), there is a label \(b\) in \(M_j\), such that \(R_{ij}(a, b) = 1\). Intuitively, a label \(a\) is assigned to a variable iff for every other variable there is at least one valid assignment of a label to that other variable that supports the assignment of label \(a\) to the first variable.

We call a solution \((M_1, \ldots, M_n)\) a \textit{maximal} solution for AC iff there does not exist any other solution \((S_1, \ldots, S_n)\) such that \(M_i \subseteq S_i\) for all \(1 \leq i \leq n\). We are interested only in maximal solutions for an AC problem. By insisting on maximality we guarantee that we are not losing any possible solutions for the original CSP. Therefore, in the remainder of this paper a solution for an AC problem is identified with a maximal solution. The sequential time complexity of AC is discussed in [15,20,22]. Discrete relaxation is the most commonly used method to achieve local consistency. Starting with all possible label assignments for each variable, \textit{discrete relaxation} repeatedly discards labels from variables if the AC condition specified above does not hold.

3. Parallel processing of constraint networks

The standard approach for achieving arc consistency is the following discrete-relaxation procedure:

**Procedure Parallel-AC.**

\begin{itemize}
  \item \textit{Step 1.} Start by setting \(M_i = L_i\), for \(1 \leq i \leq n\).
  \item \textit{Step 2.} Repeat the following:
    \begin{itemize}
      \item For each constraint between \(X_i\) and \(X_j\) test whether for each label \(a \in M_i\) there exists a label \(b \in M_j\) that permits it. If there is no such \(b\) then remove \(a\) from \(M_i\).
    \end{itemize}
  \end{itemize}

until no label is removed from any \(M_i\).

It is easy to see that this algorithm will terminate in \(O(EK^2nK)\) time, where \(E\) is the number of edges in the constraint network graph and \(K\) is the number of labels for each variable (recall that \(EK^2\) is the size of the input). In fact, much better sequential algorithms for the problem are discussed in [14,15,20,22].

Clearly, procedure Parallel-AC can be parallelized in a straightforward way. If we assume a CRCW PRAM as our model of parallel computation,\(^1\) we have the following simple result:

\textbf{Claim 3.1.} The parallel complexity of procedure Parallel-AC is \(O(nK)\) on a CRCW PRAM with \(EK^2\) processors.

\(^1\) CRCW PRAM is a standard shared-memory parallel computation model that permits concurrent reads and writes into the same location. Concurrent writes are permitted if they agree on the data being written.
**Proof.** Simply assign $K$ processors to each arc and label, and perform the test for arc consistency in Step 2 in parallel. Arc consistency is essentially a logical OR on set membership of a set of labels, and can be performed in constant time on a CRCW PRAM. At each parallel step, if the algorithm does not halt, then at least one label must be dropped and there are a total of $nK$ labels.

On a more realistic model of computation, such as an EREW (exclusive read/exclusive write) PRAM, we can perform the above procedure in $O(nK \log K)$ parallel time, the extra log $K$ factor being required to compute the logical OR.

The usual way to see parallel computation of constraint networks in the AI community is not via shared-memory models such as the CRCW PRAM. We usually assume that we have processors associated with nodes/arcs of the network. The processors communicate to their neighbors. This perspective requires the full power of a CRCW PRAM model for the following reasons:

1. We assume constant overhead for communication between processors, which in the worst case implies complete graph connectivity.
2. We assume that in constant time all the neighbors of a node can communicate with the node (i.e., propagate constraints, remove labels, etc.), which is equivalent to concurrent read/write capabilities.

It is easy to see that the procedure above has a lower bound of $nK$ steps, i.e., in the worst case it does not fully parallelize in the sense of achieving polylogarithmic parallel time (this a well-known observation). As a simple example, consider a chain constraint graph as shown in Fig. 1 with label sets $L_i = \{0, 1\}$, $2 \leq i \leq n$ and $L_1 = \{1\}$, and assume the only supporting assignment for label 0 is 0. Then procedure Parallel-AC will drop only one 0-label in each parallel iteration of Step 2. In Section 6 we show that, in fact, it is possible to achieve full parallelization of chain networks using a different technique.

In [15] we proved the following much stronger result, namely, that AC is inherently sequential in the worst case for general constraint networks.

**Theorem 3.2** (Kasif 90). The propositional Horn clause satisfiability problem is log-space reducible to the AC problem. That is, AC is $\mathcal{P}$-complete.

Local consistency belongs to the class of inherently sequential problems called log-space complete for $\mathcal{P}$ (or $\mathcal{P}$-complete). Intuitively, a problem is $\mathcal{P}$-complete iff a polylogarithmic-time parallel solution (with a polynomial number of processors) for the problem will produce a polylogarithmic-time
parallel solution for every deterministic polynomial-time sequential algorithm. This implies that unless \( P = \mathcal{NP} \) (\( \mathcal{NP} \) is the class of problems solvable in polylogarithmic parallel time with polynomially many processors) we cannot solve the problem in polylogarithmic time using a polynomial number of processors. The above theorem implies that to achieve polylogarithmic parallel time one (probably) would need a superpolynomial number of processors. We emphasize that this is a worst-case result and indeed several groups have reported successful experiments with massively parallel constraint processing [3,8,10,25,28,31].

One common but incorrect interpretation of Theorem 3.2 is that inherent sequentiality is caused by long constraint chains such as the one in Fig. 1. In Section 6 we give a simple technique for parallel processing of constraint chains, and in Section 7 we extend these results to trees. Most importantly we show that while the degree of parallelism is dependent on the diameter of the constraint graph, a high diameter is not a sufficient condition for inherent sequentiality of constraint networks.

We first make an observation which is critical to the understanding of the procedural semantics, and consequently the complexity, of achieving arc consistency. In [13–15] we provided a two-way reduction between arc consistency and Propositional Horn Satisfiability (PHS). Specifically, given a constraint satisfaction problem \( S \) one can construct a propositional Horn formula (AND/OR graph) \( G \) such that arc consistency of \( S \) can be achieved by (essentially) running a satisfiability algorithm for \( G \). For a formal construction see [15]. We sketch the intuition here. For each label \( a \) and a variable \( X \) we construct a propositional atom \( P_{X,a} \) which means \( a \) drops from \( X \). We also use a propositional atom \( Q_{X,a,Y} \) to mean that variable \( Y \) has no label that supports label \( a \) in \( X \). Consider, for example, a variable \( X \) connected in the constraint graph to variables \( Y \) and \( Z \). Assume that \( Y \) has labels \( a_1, a_2 \) and \( Z \) has labels \( a_3, a_4 \) that support \( a \) at \( X \). Thus, we construct the formulae:

\[
\begin{align*}
P_{X,a} & \leftarrow Q_{X,a,Y}, \\
P_{X,a} & \leftarrow Q_{X,a,Z}, \\
Q_{X,a,Y} & \leftarrow P_{Y,a_1} \land P_{Y,a_2}, \\
Q_{X,a,Z} & \leftarrow P_{Z,a_3} \land P_{Z,a_4}.
\end{align*}
\]

We can apply one iteration of procedure Parallel-AC to determine which labels will be dropped initially and add the corresponding assertions.

Note that if the constraint graph has \( E \) edges and \( K \) labels per variable, the size of the formulae is potentially \( EK^2 \). More importantly, the sequential complexity of solving satisfiability of these formulae is \( O(EK^2) \) (see [15] for details). This is a slight improvement over the result in [20] and it matches the algorithm in [22]. The advantage is that we can use this reduction to derive optimal algorithms for AC when the resulting Horn-clause formula is of small size. Thus, for example, when the size of the formula is \( O(EK) \) we get an \( O(EK) \) algorithm. We also can devise efficient parallel algorithms when
the resulting Horn-clause formula has some special graph structure. Note that this reduction is from AC to PHS, as opposed to the reduction from PHS to AC used to prove Theorem 3.2.

4. AC in dense graphs

It has been noted by several researchers that for many $P$-complete problems such as depth-first search, circuit evaluation and unification, optimal speed-up is possible if the underlying graph is very dense, i.e., the number of edges in the graph is quadratic [29]. We illustrate this simple principle with an example. Assume we are given a boolean circuit that consists of NAND gates only. The circuit contains $N$ gates and $E$ edges. We also are given an input to the circuit. Clearly, any sequential algorithm must take $O(E)$ time to evaluate the output of this circuit. The standard technique for evaluating circuits uses a counter for each gate. The initial value of the counter is set to the number of inputs the gate has. We maintain a queue of gates whose value has been computed (initially, just the input gates). We pick any gate on the queue and traverse all its outgoing edges. For each such edge we decrement the counter associated with the gate incident to the edge and drop the edge. If the counter becomes zero we simply add the gate to the queue. For the special case of NAND gates one can obviously reset the counter of a gate to zero when one of the inputs is zero. Note that this simple algorithm is in fact the same algorithm that yields optimal sequential time for both PHS (propositional Horn satisfiability) and AC [15,22]. PHS and AC problems may generate circuits with cycles but this does not fundamentally change the algorithm, as was pointed out in [15].

But now consider a brute-force parallel algorithm, where we essentially perform the sequential algorithm above, with one exception. When we choose a gate $g$ from the queue we use $N$ processors to update the counters of all the gates that are connected to $g$. This can be done trivially in constant time on a PRAM (parallel shared-memory machine). When any counter becomes zero, we add the gate to the queue as in the sequential version (on some models of parallel computation the above two steps may take logarithmic time). Since we are visiting at most $N$ nodes the procedure terminates in $O(N)$ steps using $N$ processors. Thus we achieve linear speed-up when the number of edges in the circuit is $O(N^2)$. This observation holds for propositional Horn satisfiability. It also holds for arc consistency and path consistency by using the reduction to propositional Horn satisfiability (or circuit evaluation). For AC, this observation yields $O(nK)$ performance with $nK$ processors (details are left as an exercise to the reader).

**Theorem 4.1.** AC can be solved in $O(nK)$ time with $nK$ processors.

The algorithm above will work well on any machine that can support broadcasts (to implement traversing edges) and some kind of fast selection operation.
(to pick the next node from the queue). Parallel prefix on the Connection Machine can support both primitives efficiently in logarithmic time. Note, that the parallel algorithm described above can be improved in practice by retrieving (in parallel) all nodes (gates) from the queue. Thus, for a tree-structured graph we obviously will achieve performance proportional to the depth rather than the size of the tree. Since in this paper we are primarily concerned with asymptotic performance, we do not discuss obvious improvements that may work in specific problem instances.

5. Path consistency

While arc consistency is useful in many applications, it does not always succeed in pruning the search space. Therefore, arc consistency has been generalized to $i$-consistency as follows [9]. A network of constraints is said to be $i$-consistent (or path consistent) iff for every consistent assignment of labels to a subnetwork of $i-1$ variables, there is a label consistent with the subnetwork for any choice of an $i$th variable. Several $i$-consistency-enforcing algorithms have been proposed [9, 22]. In this section we extend the simple observation from the previous section and derive efficient $i$-consistency algorithms. We believe that the methodology we use here is important since several published path consistency algorithms had errors and our derivation is very simple and easy to prove correct.

Earlier we mentioned the two-way reduction between AC and PHS. Here, we observe that a similar two-way reduction can be provided between path consistency and PHS. We illustrate the reduction from 3-consistency to PHS. The key to solving 3-consistency is deriving all assignments of pairs of labels to two variables that violate the 3-consistency condition. Without loss of generality, assume all labels are unique (otherwise, we can rename the labels). For each pair of labels $a$ and $b$ we create a proposition $P_{a,b}$ that is true iff the pair $(a, b)$ violates the 3-consistency condition, and therefore is dropped from consideration. However, $P_{a,b}$ is true iff there is at least one other variable to which it cannot be extended. We denote this condition by a predicate $Q_{a,b,i}$ which is true iff this assignment cannot be extended to variable $X_i$. Thus,

\begin{align*}
P_{a,b} &\leftarrow Q_{a,b,1}, \\
P_{a,b} &\leftarrow Q_{a,b,2}, \\
P_{a,b} &\leftarrow Q_{a,b,3}, \\
&\vdots
\end{align*}

Now, $Q_{a,b,i}$ is true iff $P_{b,c}$ or $P_{a,c}$ is true for all labels $c$ that potentially support the assignment $(a, b)$ at variable $X_i$. Thus,
We have \( O(\mathcal{E}K^2) \) predicates of the form \( P_{a,b} \). Therefore, the size of the PHS formula is dominated by the clauses of the form

\[
Q_{a,b,1} \leftarrow Q_{a,b,c_1} \land Q_{a,b,c_2} \land \cdots
\]

\[
Q_{a,b,c_1} \leftarrow P_{a,c_1},
\]

\[
Q_{a,b,c_1} \leftarrow P_{b,c_1},
\]

\[\vdots\]

The total size is therefore \( O(\mathcal{E}K^2n\mathcal{K}) = O(n\mathcal{E}K^3) \), or alternatively \( O(n^3\mathcal{K}^3) \).

The assertions of the program are obtained by applying a 3-consistency check once, and asserting all the dropped pairs \((a,b)\) as facts \( P_{a,b} \).

The important corollary of this construction is that now we can utilize the standard linear time PHS algorithm for local consistency. This immediately yields an algorithm that matches (in asymptotic complexity) the best known algorithm for path consistency. This algorithm is optimal if the number of labels is assumed constant. The same algorithm also would work for \( k \)-consistency with similar optimal performance. The bottom-up PHS algorithm has very good performance in practice. The conversion to PHS also can be done efficiently in parallel.

Similar observations were made independently by McAllester, Bibel, Mackworth and Saraswat. Here we have presented a careful analysis of the transformation that leads to optimal complexity results as stated above. The reduction illustrated in this section can be combined with the observation in the previous section to yield an efficient parallel algorithm for path consistency in dense constraint graphs. Parallel path consistency is also studied in [18,22].

6. AC in chains

It is easy to see that a simple separator-based technique can be used to solve constraint satisfaction problems in chain graphs. The technique is based on removing a variable that separates a constraint chain with \( n \) variables into two chains with \( n/2 \) variables. Then we create \( 2\mathcal{K} \) recursive subproblems, where \( \mathcal{K} \) is the number of labels in each variable. We repeat the process \( \log n \) times, so that the complexity is \( O((2\mathcal{K})^{\log n}) = O(n^{\log 2\mathcal{K}}) \). For details see [14]. Thus, when the number of labels is large (i.e., is a function of the number of variables \( n \)), the complexity is no longer polynomial.

Here we propose a simple procedure to achieve AC in a constraint chain. We first construct an explicit constraint graph \( G_{\text{expl}} \). The number of nodes in this graph is \( n\mathcal{K} \), one node for each label in each variable. Recall that in \( G_{\text{expl}} \) two nodes \((X,a)\) and \((Y,b)\) are connected iff the assignment of \( a \) to \( X \) is compatible with assigning \( b \) to \( Y \). Without loss of generality, assume the variables are numbered \( X_1 \) to \( X_n \) in order on the chain. We
now orient all the edges in $G_{\text{expl}}$ in the direction from $X_i$ to $X_{i+1}$. We mark all nodes reachable from the source nodes contained in $X_1$, and discard all other nodes. Next, we reverse the orientation of edges to point from $X_{i+1}$ to $X_i$. We then mark all nodes reachable from the remaining nodes in $X_n$, and discard the rest. An easy induction argument shows that the nodes that remain after this last step correspond to the labels in the solution to the AC problem.

**Theorem 6.1.** AC in a chain constraint network can be solved either with $nK^3/\log n$ processors in $O(\log n)$ time, or with $nM(K)/\log n$ processors in $O(\log K \log n)$ time, on a CRCW PRAM, where $M(K)$ is the number of processors needed to multiply two $K \times K$ boolean matrices in $O(\log K)$ time.

**Proof.** By the argument of the preceding paragraph the AC problem is reducible to reachability in an $n$-level directed acyclic graph. Let $\text{Reach}_{i,j}$ denote the predicate on $L_i \times L_j$ that indicates for each $a \in L_i$ and $b \in L_j$ whether there is a path from $a$ to $b$ in the directed version of $G_{\text{expl}}$. $\text{Reach}_{i,i+1}$ is the constraint relation given initially, and we must compute $\text{Reach}_{1,n}$.

We can successively compute $\text{Reach}_{i,i+2^s}$, in parallel for $1 \leq i \leq n$, in stages $s = 1,\ldots,\log n$. At stage $s = 1$ we compute $\text{Reach}_{i,i+2}$ for $i \equiv 1$ (mod 2) by composing the predicates $\text{Reach}_{i,i+1}$ and $\text{Reach}_{i+1,i+2}$. Then at stage $s = 2$ we compute $\text{Reach}_{i,i+2^2}$ for $i \equiv 1$ (mod $2^2$) by composing the predicates $\text{Reach}_{i,i+2}$ and $\text{Reach}_{i+2,i+2^2}$. In general, at stage $s$ we compute $\text{Reach}_{i,i+2^s}$ for $i \equiv 1$ (mod $2^s$) by composing the predicates $\text{Reach}_{i,i+2^s-1}$ and $\text{Reach}_{i+2^s-1,i+2^s}$.

Note that the number of compositions at stage $s$ is $n/2^s$ so that the total number of compositions is $n$. By performing $n/\log n$ compositions in parallel, we can finish all $n$ compositions in $O(\log n)$ steps.

Each of the $\text{Reach}_{i,j}$ predicates is a $K \times K$ boolean matrix and the composition can be performed by matrix multiplication. Specifically, when we compose $\text{Reach}_{i,j}$ with $\text{Reach}_{j,k}$ to get $\text{Reach}_{i,k}$, we have

$$\text{Reach}_{i,k}(a,c) = \bigvee_{b \in L_j} \text{Reach}_{i,j}(a,b) \land \text{Reach}_{j,k}(b,c).$$

In the CRCW model, with $K^3$ processors we can accomplish this composition in $O(1)$ time, or alternatively in $O(\log K)$ time with $M(K)$ processors [12].

Note that Dechter and Pearl used a similar technique, i.e., orienting the constraint edges in one direction and then in the opposite direction. They coined the term directed arc consistency for this approach. However, since they were not studying the parallel complexity of chain networks, they did not note that this approach reduces the problem in chains to two computations of directed components in a graph.
7. Trees

In this section we consider the special case of CSP in which the constraint graph contains no cycles, i.e., it is a forest. Since it clearly suffices to consider each connected component of the graph separately, in the remainder of this section we restrict our attention to trees.

7.1. Separator-based methods

We first show how we can take advantage of the separability properties of the constraint graph in order to achieve a recursive parallel decomposition of the problem. A more powerful (but higher overhead) method was presented for planar constraint networks in [16] (see also [4, 26]).

Lemma 7.1. Given any constraint tree $T$, with $|T| = N$, we can construct a corresponding binary constraint tree $T'$, with $|T'| < 2N$, such that $T$ and $T'$ have the same solutions.

Proof. Let $v$ be a node in $T$ with $k$ children, $u_1, \ldots, u_k$; let $L_v, L_{u_1}, \ldots, L_{u_k}$ be their respective label sets; and let $R_{u_i,v}$ be the constraint relation between $L_{u_i}$ and $L_v$. If $k \leq 2$, we make $v$ and its child edges the same in $T'$ as in $T$. If $k > 2$, then we create a chain of nodes $v_0, \ldots, v_{k-2}$ as shown in Fig. 2. To each $v_j$ we assign the same label set $L_{v_j}$, and the constraint relation between $u_i$ and its parent $v_j$ is just $R_{u_i,v_j}$. Between $v_j$ and $v_{j-1}$, $1 \leq j \leq k-2$, we assign the identity constraint. It is now an easy exercise to verify that the solutions to $T$ and $T'$ are identical. Since we create at most one new node $v_j$ for each child node $u_i$, it follows that $|T'| < 2|T|$. □

![Fig. 2. Converting a non-binary tree to a binary equivalent.](image-url)
Notice we are not transforming n-ary constraint networks into binary constraint networks; we are merely modifying the topology of the original constraint tree with a constant increase in size. Thus, without loss of generality we need consider only binary trees.

**Lemma 7.2.** Let $T$ be a binary tree, with $|T| = N$. There is a node $v$ in $T$ such that if $T_v$ denotes the subtree of $T$ rooted at $v$, then $N/3 \leq |T_v| \leq 2N/3$.

**Proof.** Starting from the root of $T$, traverse a path $P$ toward the leaves always moving to the child which roots the larger subtree. Then choose $v$ to be the first node on $P$ with $N/3 < |T_v| < 2N/3$. Clearly such a node exists, since if $u_1$ and $u_2$ are successive nodes on $P$, $|T_{u_1}| > (|T_{u_2}| - 1)/2$.

**Proposition 7.3.** If constraint graph $G$ is a tree, with $K$ labels per node, we can solve CSP for $G$ in $O(\log N)$ time with $O(N^{\log_{1.5} K})$ processors.

**Proof.** As in Lemma 7.2, let $v$ be a node in $G$ such that $N/3 \leq |T_v| \leq 2N/3$. Disconnect $T$ at $v$ to create two trees, each of size less than $2N/3$. Then in parallel for each possible label of $v$, recursively solve CSP for these two trees. Finally determine whether there is a solution for each tree with the same label for $v$. The recurrences for time and processors are, respectively:

$$
T(N) \leq T(2N/3) + c \quad (c \text{ a constant})
$$

$$
= O(\log N),
$$

$$
P(N) \leq K \cdot P(2N/3)
$$

$$
= O(N^{\log_{1.5} K}).
$$

Thus, we can solve CSP in logarithmic time with a polynomial number of processors, where the degree of the polynomial depends on $K$, the number of labels per variable. □

A similar method can be used to solve the AC problem. Details are omitted.

### 7.2. Expression evaluation methods

In the previous subsection we suggested a separator-based algorithm to solve CSP and AC when the underlying graph is a tree. Similar to chains, the complexity of that algorithm is $O(\log N)$ with $O(K^{\log_{1.5} N})$ processors. Here we propose a different algorithm to solve AC in trees. This algorithm operates in $O(\log N \log K)$ time and uses a polynomial number of processors, where the degree of the polynomial does not depend on $K$. More importantly, it reduces the consistency problem to reachability and expression evaluation on trees.

For non-constant values of $K$, the algorithm in this subsection is asymptotically better than the previous algorithm. However, the algorithm is probably
impractical, and is presented to demonstrate the substantial difficulty in getting asymptotically optimal parallel algorithms, even for tree-like constraint networks.

Let $T$ be the constraint tree, which without loss of generality we assume is binary and rooted at some arbitrary node. We first construct the explicit constraint graph $G_{\text{expl}}$. Let $\ell_{i,v}$ denote the $i$th label in node $v$, and let $R_{u,v}$ denote the compatibility (support) predicate between nodes $u$ and $v$. Note that $R_{u,v}$ is also the adjacency matrix for the subgraph of $G_{\text{expl}}$ restricted to labels in nodes $u$ and $v$.

The algorithm now proceeds as follows:

**Step 1.** Mark the following set of labels, defined bottom up in the tree:
- $\ell_{i,v}$ is marked if $v$ is a leaf.
- $\ell_{i,v}$ is marked if for each child $w$ of $v$ in $T$ there exists $j$ such that $\ell_{j,w}$ is marked and $\ell_{i,v}$ and $\ell_{j,w}$ are adjacent in $G_{\text{expl}}$.

The entire system is consistent iff the set of labels marked at the root is non-empty.

**Step 2.** Discard all unmarked labels, and consider the original problem restricted only to marked labels. Clear all marks, and do the following second marking procedure:
- Mark all labels at the root.
- Mark $\ell_{i,v}$ iff there exists a marked $\ell_{j,w}$ where $w$ is the parent of $v$ and $\ell_{i,v}$ and $\ell_{j,w}$ are adjacent in $G_{\text{expl}}$.

The solution is the set of labels marked at the end of Step 2.

The sequential version of the algorithm outlined above is a version of directed arc consistency that was originally developed by Dechter and Pearl [4]. The reader should note that while sequentially directed arc consistency can be implemented optimally for both chains and trees, in the parallel environment the two problems are very different. The parallel implementation of directed arc consistency that achieves sublinear time is nontrivial and is discussed below.

For completeness, we first discuss the correctness of the algorithm outlined above. By induction on the height of nodes in the tree, it is clear that labels marked in Step 1 are precisely those that potentially have support from nodes below them in the tree. In particular, it follows that labels marked at the root of $T$ will be contained in the solution. We now can argue by induction on the depth of each node in $T$ that nodes marked in Step 2 are precisely those that have support from their parent in the tree. Since we are only considering nodes that already have support from their child edges, it follows that the labels marked in Step 2 are the solutions to the AC problem.

It is easy to see that the sequential version executes in linear time. We now describe how this simple algorithm can be implemented in parallel. Our implementation is nontrivial, but we are unaware of a simpler way to achieve sublinear time for trees in which the depth may be proportional to the size of the tree.
Step 1 can be viewed either as an expression-evaluation calculation, or as a reachability computation.

- As an expression we are computing a bit vector at each node, one bit marker for each label at that node. If \( b_x \) denotes the bit vector computed at node \( x \), then if nodes \( v \) and \( w \) are children of node \( u \) in \( T \), we have

\[
    b_u = (b_v \times R_{v,u}) \odot (b_w \times R_{w,u}),
\]

where \( \times \) represents boolean matrix multiplication and \( \odot \) is a bitwise AND operation. We can evaluate this expression by parallel expression-evaluation algorithms such as the raking procedure described in [1,17].

In brief, the raking procedure works by evaluating and removing (in parallel) leaves and their parents from the expression tree. At each stage approximately half of the leaves are removed, so that after \( O(\log N) \) stages the entire expression has been evaluated. The partial evaluation results are stored on the edges of the tree. In our case, when a leaf is raked, we are applying an intersection of a known (i.e., constant) bit vector between two matrix multiplications. This is simply a restriction of the matrix product to the rows and columns of the intersection vector. Thus the result of the rake operation can be represented as a single matrix, and can be computed in \( O(\log K) \) time using the number of processors required to multiply two \( (K \times K) \)-bit matrices, \( M(K) \). The entire evaluation can then be done in \( O(\log N \log K) \) time with \( N \cdot M(K)/\log N \) processors.

- As a reachability computation, we can identify all labels reachable from a leaf label (assuming \( G_{\text{exp}} \) is now directed with all arcs oriented toward the root of \( T \)). We then mark all labels that are reachable from all leaves under them. Finally, restricting ourselves only to these marked nodes, we redo the reachability-from-leaves calculation. Computing reachability as transitive closure by matrix multiplication, we can accomplish Step 1 in this manner also with \( N \cdot M(K)/\log N \) processors in \( O(\log N \log K) \) time.

Step 2 is a simple calculation of reachability from labels at the root of \( T \), and so can be done within the same processor-time bounds. Thus we have shown:

**Theorem 7.4.** If constraint graph \( G \) is a tree, with \( K \) labels per node, we can solve AC for \( G \) in \( O(\log N \log K) \) time with \( N \cdot M(K)/\log N \) processors.

### 8. Two-label CSP

In this section we reduce the two-label CSP to 2-SAT (satisfiability of formulae in conjunctive normal form with at most two atoms per clause). 2-SAT is equivalent to a reachability problem in directed graphs. Therefore, if the number of possible labels per variable is two or less we can process the constraint graph in polylogarithmic parallel time. Another implication of this result is that two-label CSPs can be solved in linear time.
**Lemma 8.1.** Given a CSP $P$, where $|L_i| \leq 2$, $1 \leq i \leq n$ (i.e., the number of labels for each variable is at most 2), then $P$ can be converted to a 2-SAT problem $S$ such that $I$ is a solution for $S$ iff $I$ is a solution for $P$.

**Proof.** We merely regard the labels in each variable as representing truth values. We then translate each constraint relation $R_{ij}$ into an equivalent 2-CNF form, and take the conjunction over all such relations in $P$. For instance, the constraint

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$R_{XY}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

is converted to the set of clauses $(X \lor Y) \land (\neg X \lor \neg Y)$. □

Given a set $S$ of clauses in 2-SAT form, we can create a new, directed constraint graph that captures the constraints of the initial constraint graph more explicitly. For each variable $X$ occurring in $S$ we create two nodes, $X$ and $\neg X$. Then for each clause in $S$ of the form $\alpha \lor \beta$, we create two directed arcs: $\neg \alpha \rightarrow \beta$ and $\neg \beta \rightarrow \alpha$. As an example, for the constraint $(X \lor Y) \land (\neg Y \lor Z)$ we generate the following graph:

![Graph Diagram](image)

It is known that using the above construction of an explicit directed graph we can find solutions to the original problem by tracing the strongly connected components of the constraint graph. The precise claim is captured in the next intuitive proposition which we state without proof.

**Proposition 8.2.** Let $S$ be a 2-SAT formula and $C$ its explicit constraint graph constructed as above. Then $S$ is satisfiable iff no strongly connected component of $C$ contains both $X$ and $\neg X$ for any variable $X$ in $S$. Moreover, if $S$ is satisfiable, the strongly connected components of $C$ can be used to derive satisfying assignments for the variables of $S$.

In other words, the formula is unsatisfiable iff some node $\neg X$ is reachable from node $X$ and vice versa. However, reachability problems are well understood and can be done by transitive closure methods in parallel in $O(\log^2 N)$.
time with $M(N)$ processors on a CRCW PRAM [12]. Thus, we have the following:

**Proposition 8.3.** Given a CSP $P$ where $|L_i| \leq 2$, i.e., the number of possible labels (solutions) per variable is at most 2, $P$ can be solved in $O(\log^2 N)$ time with $M(N)$ processors on a CRCW PRAM.

Thus, in the special case when the number of labels per object is at most 2, we can compute the solution in sublinear time with a relatively large number of processors ($N^{1.5}$ processors where $N$ is the size of the entire input). We note that we are actually solving the constraint problem directly without attempting to achieve arc consistency first. Since computing directed components can be done sequentially by depth-first search, we obtain an immediate corollary.

**Corollary 8.4.** Let $P$ be a CSP with $|L_i| \leq 2$. Then $P$ can be solved in linear time by a single processor.

This last result apparently has been known for some time [21], however it first appeared in print in [6,14]. In fact, a more general version of the above construction seems to be folklore, namely that arc consistency for networks with $K$ labels is easily transformable in linear time to $K$-SAT. Thus, our translation of 2-CSP to 2-SAT is a special case of this result. We leave the proof to the reader.

9. Consistency in directed support networks (DC)

Traditionally, AC was considered as a filtering step used to solve constraint satisfaction problems. However, there is an interesting modal-logic-like interpretation of AC in the context of support networks. We start with a collection of agents (variables) each holding a set of beliefs (labels). For each pair of variables $X$ and $Y$ we define a directional support relation that indicates which beliefs at one agent get supported by beliefs at the other agent.

An agent keeps a belief $a$ iff it is supported by at least one belief at each of the other agents associated with it. Note that we are not seeking a global interpretation of the variables, but rather would like to filter unsupported beliefs. This is analogous to the standard AC problem. In this case, however, the support relations are directed. Dechter and Pearl have defined the notion of directed arc consistency, which is used to compute arc consistency in trees. The standard directed arc consistency allows one to orient the edges from one variable to another in a certain direction. Our formalization allows one to direct edges from any label to any label in any prespecified direction (see also [5] for a related idea). The intuition is, that a certain belief an agent may have could be supported by some beliefs from other agents, but another belief is independent of other agents. Consistency in directed support networks
(DC) is naturally described by an explicit constraint graph. Edges in the graph describe directed support (see the example below). The only label that drops is $e$ from $Y$.

\begin{center}
\begin{tikzpicture}[scale=0.8]
  \node at (0,0) {$X:$};
  \node at (2,0) {$Y:$};

  \node at (0,-1) {$(X,a)$};
  \node at (2,-1) {$(Y,b)$};
  \node at (0,1) {$(X,b)$};
  \node at (2,1) {$(Y,e)$};

  \draw[->] (0,0) -- (0,-1);
  \draw[->] (0,-1) -- (0,1);
  \draw[->] (2,0) -- (2,-1);
  \draw[->] (2,-1) -- (2,1);
  \draw[->] (0,0) -- (2,0);
  \draw[->] (0,0) -- (0,1);
  \draw[->] (2,0) -- (2,1);

\end{tikzpicture}
\end{center}

The following surprising result indicates that the parallel complexity of solving DC is considerably more complicated than standard AC problem. Essentially, this result indicates that the parallel complexity is dependent on the structure of the explicit constraint graph, rather than the structure of the constraint graph. This follows from the fact that we can encode general logical dependency in such a graph.

**Theorem 9.1.** Consistency in directed support networks (DC) is $\mathcal{P}$-complete, even if the underlying graph is a 3-node chain.

**Proof.** Our reduction is from propositional Horn-clause satisfiability. Without loss of generality, we assume the Horn-clause program contains a single assertion $T$, and that every other variable $A_i$ appears as the head of either exactly one rule of the form $A_i \leftarrow A_j \land A_k$, or else two or more rules of the form $A_i \leftarrow A_j$. Let $A_n$ be the goal of the program.

We construct an instance of DC containing three nodes: $L$ (left), $M$ (middle) and $R$ (right). Let each of these three nodes contain a label for $T$ and for each $A_i$. Now construct arcs for the explicit support graph as follows:

1. From each label in $M$ to the corresponding label in both $L$ and $R$.
2. From $T_L$ to $T_M$ and from $T_R$ to $T_M$.
3. For every clause of the form $A_i \leftarrow A_j, A_k$, construct an arc from $A_{j,L}$ to $A_{i,M}$ and from $A_{k,R}$ to $A_{i,M}$.
4. For every clause of the form $A_i \leftarrow A_j$, construct an arc from $A_{j,L}$ to $A_{i,M}$ and from $T_R$ to $A_{i,M}$.

It is now easy to show that the labels that remain in $M$ correspond exactly to the variables that are true in the Horn-clause program. In particular $A_n,M$ remains iff the goal $A_n$ of the program is true. \qed
10. Summary of parallel AC results

In this section we summarize our knowledge of the parallel complexity of computing local consistency in constraint satisfaction problems. The results appear in Tables 1 and 2. We have classified the parallel complexity of problems into two classes: \( P \)-complete problems and \( A\!) \)-complete problems. \( P \)-complete problems are perceived to be difficult to parallelize (in the same sense that \( NP \)-complete problems are considered intractable), and \( A\!) \)-complete problems can be solved in polylogarithmic time with a polynomial number of processors. \( A\!) \)-complete problems are often amenable for optimal speed-up on parallel machines. In both tables \( R \) denotes the binary compatibility predicate.

The main practical conclusions that we can draw from our study are as follows:

1. Local consistency in constraint networks is generally \( P \)-complete. Practi-
### Table 2
Complexity of arc consistency for fixed-size label sets. ($K$ is the size of the label set $L$)

<table>
<thead>
<tr>
<th>$G$</th>
<th>CSP</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed $K$</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Chain                | $\mathcal{NC}$ | Undirected $R$'s: $\mathcal{NC}$  
                             Directed $R$'s: $\mathcal{NC}$ |
| Tree                 | $\mathcal{NC}$ | Undirected $R$'s: $\mathcal{NC}$  
                             Directed $R$'s: $\mathcal{NC}$; from above |
| Simple cycle         | $\mathcal{NC}$ | Undirected $R$'s: $\mathcal{NC}$  
                             Directed $R$'s: $\mathcal{NC}$; from above |
| **Arbitrary graph**  |       |                                         |
| $K = 2$: Linear sequential algorithm by reduction to 
2-SAT which is $\mathcal{NC}$ |       | Undirected $R$'s: For $K = 2$, 
$\mathcal{NC}$ by reachability along 
"singleton paths"; For 
$K \geq 3$, $\mathcal{P}$-complete from 
propositional Horn-clause solvability |
| $K \geq 3$: $\mathcal{NP}$-complete; 
reduction from 3-colouring graphs |       | Directed $R$'s: $\mathcal{P}$-complete for 
$K \geq 2$ |

- **tial experience suggests** that it is difficult to obtain optimal parallel algorithms for such problems. By optimal parallel algorithms we mean algorithms that obtain $P$-fold speed-up of the best sequential algorithm with $P$ processors. However, it is often easy to obtain optimal speed-ups for these (and other) problems when the number of processors is small.

2. Substantial speed-ups for parallel local consistency algorithms are possible if the constraint graph is dense. This can be accomplished using a simple obvious algorithm which is likely to be efficient in practice. Specifically, given a constraint network with $n$ nodes and $K$ labels per node, it is easy to obtain an $O(nK)$ algorithm for arc consistency with $nK$ processors on a shared-memory parallel model of computation.

3. The application of the obvious parallel version of the arc consistency algorithm to such networks does not yield a sublinear algorithm. The parallel complexity of local consistency in chain networks has been shown equivalent to reachability problems in directed graphs. While this class of networks can theoretically be solved very fast, in practice our results imply that the "transitive closure" bottleneck may apply to chain networks. It currently is not known how to get optimal speed-ups for transitive closure problems in graphs unless the number of processors is smaller than the number of nodes in the graph.

4. For tree networks we suggested several algorithms that achieve sub-
linear time with many parallel processors. We have been unable to find an obvious practical algorithm to achieve optimal speed-up in tree networks.

(5) We provided a reduction from $i$-consistency to propositional Horn satisfiability which allows us to derive optimal sequential algorithms for problems such as path consistency in a simple manner.

Acknowledgements

This research has been supported partially by the Air Force Office of Scientific Research under grant AFOSR-89-1151, the National Science Foundation under grant IRI-88-09324 and NSF/DARPA Grant CCR-8908092. Thanks are due to David McAllester, Judea Pearl and Rina Dechter for their constructive comments.

References