Parallel and Distributed Algorithms for Finite Constraint Satisfaction Problems

Ying Zhang

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

Alan K. Mackworth*

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

Abstract

This paper develops two new algorithms for solving a finite constraint satisfaction problem (FCSP) in parallel. In particular, we give a parallel algorithm for the EREW PRAM model and a distributed algorithm for networks of interconnected processors. If an FCSP can be represented by an acyclic constraint network of size n with width bounded by a constant then (1) the parallel algorithm takes $O(\log n)$ time using $O(n)$ processors and (2) there is a mapping of this problem to a distributed computing network of poly(n) processors which stabilizes in $O(\log n)$ time.

1 Introduction

A Finite Constraint Satisfaction Problem (FCSP) can be informally described as follows. Given a set of variables, each with a finite domain, and a set of constraints, each specifying a relation on a subset of the variables, find the relation on the set of all the variables which satisfies all the given constraints simultaneously. FCSPs are useful abstractions of many problems in image understanding, planning, scheduling, database retrieval and truth maintenance [7] [2].

Any FCSP can be represented by a constraint network. Graphically, a constraint network is a labelled hypergraph, in which nodes represent variables and arcs represent constraints. Formally,

Definition 1.1
Constraint Network $CN \equiv \langle V, dom, A, con\rangle$ *where*

- *V is a set of variables,* $\{v_1, v_2, \ldots, v_N\}$.
- *Associated with each variable* v_i *is a finite domain* $d_i = dom(v_i)$.
- *A is a set of arcs,* $\{a_1, a_2, \ldots, a_n\}$.
- *Associated with each arc* a_i *is a constraint* $con(a_i)$.

A *constraint*, written $r(R)$, can be considered as a relation r on a relation scheme R [9], where R is a set of variables and r is a set of relation tuples. $r(R)$ is a *universal* constraint iff r includes all the possible tuples. For $X \subseteq R$, the *projection* of r onto X, written $\Pi_X(r)$, is $\{t(X)|t \in r\}$, where $t(X)$ is tuple t restricted to X. The *join* of two constraints $r(R)$ and $l(L)$, written $r \bowtie l$, is $\{t(R \cup L)|\exists t_r \in r, t_l \in l, t(R) = t_r(R), t(L) = t_l(L)\}$. The *semijoin* of $r(R)$ and $l(L)$, written $r \triangleleft l$, is $\Pi_R(r \bowtie l)$.

Let C be the set of constraints of a constraint network CN, $C = \{con(a_i)|a_i \in A\}$. A constraint network is a *binary* constraint network iff $\forall r(R) \in C, |R| \leq 2$. The hypergraph of CN is called the *scheme* of CN [2], i.e. $scheme(CN) = \{R|r(R) \in C\}$. Tuple s is a solution of a constraint network CN, i.e. $s \in sol(CN)$, iff $\forall r(R) \in C$, $s(R) \in r$. A constraint network CN is *minimal* iff $\forall r(R) \in C$, $\Pi_R(sol(CN)) = r$. Two constraint networks CN and CN' are *equivalent*, written $CN = CN'$, iff $V = V', dom = dom', sol(CN) = sol(CN')$.

However, it is well known that the FCSP decision problem is NP-complete. In order to cope with the intractability of FCSPs, two strategies have been followed: (1) finding efficient algorithms for preprocessing, such as arc consistency [6], path consistency [11] and k-consistency [4] algorithms for binary constraint networks; and (2) exploiting the topological features of FCSPs to guide efficient algorithms for solving these problems [2]. In this paper, we develop an approach to combining these two strategies. We generalize the binary arc consistency problem [6] to an arc consistency problem on any constraint network.

The *dual network DN* of a constraint network CN can be considered as an alternative representation of an FCSP. DN is a labelled undirectional graph, in which the nodes are the arcs of CN labelled by constraints, the edges represent the nonempty intersection of two relation schemes. For an edge $e_{ij} = \{a_i, a_j\}$ in DN, with $con(a_i) = r_i(R_i)$ and $con(a_j) = r_j(R_j)$, the label of e_{ij}, written $L(e_{ij})$, is $R_i \cap R_j$. e_{ij} is *arc consistent* iff $\Pi_{L(e_{ij})}(r_i) = \Pi_{L(e_{ij})}(r_j)$. A dual network is *arc consistent* iff all the edges are arc consistent. We say a constraint network is *arc consistent* iff its dual network is arc consistent. This definition reduces to the definition given in [6] for binary constraint networks.

A *join network JN* of a constraint network is a subnetwork of the dual network DN, with redundant

*Shell Canada Fellow, Canadian Institute for Advanced Research

0-8186-2310-1/91 $1.00 © 1991 IEEE

edges removed. A join network is *arc consistent* iff all the edges are arc consistent. All the join networks of a dual network are equivalent, in the sense that a dual network is arc consistent iff any of its join networks is arc consistent. In other words, a constraint network is arc consistent iff any of its join networks is arc consistent.

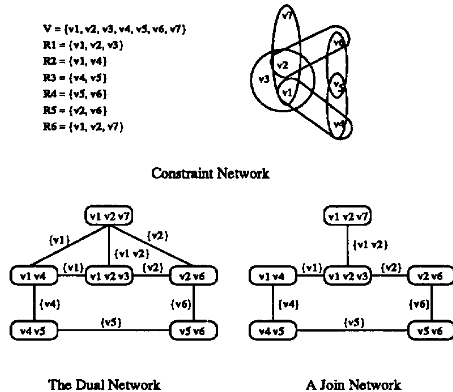

Figure 1: A Constraint Network, the Dual Network and a Join Network

Example 1.1 *Figure 1 shows the scheme of a constraint network, the dual network and a join network.*

Algorithm `AC` is a generalization of `AC-3` in [8].

```
Algorithm AC: Enforce Arc Consistency in CN
Input: join network <A,E>;
Output: arc consistent network;
BEGIN
 q := the set of all nodes in A;
 WHILE (q is not empty) DO
 BEGIN
  remove node a from q; /* con(a) = r(R) */
  s := r;
  FOR (all {a,a1} in E) /* con(a1) = r1(R1) */
   s := s semijoin r1;
  IF s =\= r THEN
  BEGIN
   r := s;
   FOR (all {a,a1} in E) DO
    q := q union {a1};
  END
 END
END
```

If a binary constraint network CN is acyclic, a tree, enforcing arc consistency in CN results in a minimal network [8]. Generalizing, we say a constraint network is acyclic iff its hypergraph is acyclic, a hypertree [9] [12]. On the other hand, it is acyclic iff its join networks are trees [9]. Applying algorithm `AC` to any of its join trees results in a minimal constraint network.

Furthermore, for a join tree, there exist a more efficient algorithm `TAC` for obtaining a minimal network.

```
Algorithm TAC: AC on Acyclic Network
Input: rooted join tree <A,E>;
Output: minimal network;
BEGIN
 q0 := the set of all nodes in A;
 /* ordered from children to parents */
 q := q0;
 WHILE (q is not empty) DO
 BEGIN
  remove node a from q; /* con(a) = r(R) */
  FOR (all (a,a1) in E) /* con(a1) = r1(R1) */
  /* a is the parent of a1 */
   r := r semijoin r1;
 END
 q: = reverse q0;
 /* ordered from parents to children */
 WHILE (q is not empty) DO
 BEGIN
  remove node a from q;
  FOR (all (a1,a) in E) DO
   r := r semijoin r1;
 END
END
```

For an acyclic constraint network CN, let $size(CN) = |A|$, $width(CN) = \max_{R \in scheme(CN)}\{|R|\}$. The complexity of `TAC` is $O(nl \log l)$ where n is the number of nodes in the join tree, i.e. $n = size(CN)$ and l is the maximum number of tuples in any constraint, i.e. if $m = \max\{|d_i|\}$, then $l \leq m^{width(CN)}$. For a minimal acyclic network, finding one solution is in $O(n)$.

Since a constraint network CN may not be acyclic in general, as in the example shown in Figure 1, the solutions for CN can be computed in three steps. First, construct a join tree whose constraint network is equivalent to CN. Second, apply `AC` or `TAC` to the join tree. Third, construct the solutions of the acyclic minimal network. The first step is called *tree clustering*. A tree-clustering scheme can be obtained by applying a tree-clustering algorithm [3] to $scheme(CN)$. Given a tree-clustering scheme TC for CN, we can construct a join tree by adding universal constraints to relation schemes which are in TC but not in CN.

In the rest of this paper, we will present a parallel version of algorithm `TAC`, and a distributed version of algorithm `AC`.

2 Parallel Algorithm and Complexity

Even though arc consistency for a binary constraint network is P-complete, it is in NC for a binary *acyclic* constraint network [5], i.e. there exists an algorithm which takes polylog time using polynomial number of processors in the PRAM model. We present a parallel `TAC` algorithm which generalizes this result to any acyclic constraint network of bounded width.

We apply the parallel tree contraction technique in [10] to the problem. Let $T = \langle A, E \rangle$ be a rooted join tree with nodes A and edges E. A sequence of nodes $a_1, \ldots, a_k$ is called a *chain* if a_{i+1} is the only child of a_i for $1 \leq i < k$, and a_k has exactly one child and that

child is not a leaf. The parallel tree contraction algorithm defines two basic contract operations: RAKE and COMPRESS. RAKE is the operation of removing all leaves from T. COMPRESS is the operation on T which contracts all the maximal chains of T in half, by identifying a_i with a_{i+1} for i odd, where a_i is a node on a maximal chain. CONTRACT is the simultaneous application of RAKE and COMPRESS to the entire tree. After $\lceil \log_{5/4} n \rceil$ executions of CONTRACT on a tree of n vertices, the tree is reduced to its root [10].

The parallel TAC algorithm ParAC consists of two phases: ContractAC and ExpandAC. ContractAC, shown below, iterates tree contraction on a rooted join tree T. Semijoin operations are associated with each RAKE; join and projection operations are associated with each COMPRESS. For $a \in A$, let $pt(a)$ be the parent of a. If a has only one child, let $cd(a)$ denote that child. If $arg(a)$ is the number of children of a, let $chain(a)$ be a boolean function defined as $arg(a) = 1$ and $arg(pt(a)) = 1$. We call p the *contracting parent* of a, if a is raked from p or a is compressed to p. Let $cp(a)$ denote the contracting parent of a. Whenever a RAKE operation removes a leaf node with constraint $l(L)$ from its parent with constraint $r(R)$, a semijoin $r \lhd l$ is performed and r, the relation on the parent, is updated. Correspondingly for the COMPRESS operation, suppose a_i, a_{i+1} are two consecutive nodes on a chain and let a_{i-1} be the parent of a_i and a_{i+2} be the child of a_{i+1} with $con(a_k) = r_k(R_k)$ and $L_k = R_k \cap R_{k+1}$, where $i-1 \le k \le i+1$. Whenever a_i is identified with a_{i+1}, an operation $\Pi_{L_{i-1} \cup L_i \cup L_{i+1}}(r_i \bowtie r_{i+1})$ is applied.

```
Algorithm ContractAC: Tree Contraction Phase
Input: rooted join tree T = <A,E>;
Output: directional arc consistent network;

Iterate the following procedure until T=root:

In Parallel for all a in A\{root}
BEGIN
 r(R) := con(a); p(P) := con(pt(a));
 IF (a has a leaf child) THEN /* RAKE */
 FOR (each leaf child c with constraint l(L))
 BEGIN
  r := r semijoin l; remove c;
  /* update links of a */
  cp(c) := a
 END
 ELSE IF (chain(a)) THEN /* COMPRESS */
 BEGIN /* pt(a) is identified with a */
  create a new node a';
  c(C) := con(cd(a));
  p'(P') := con(pt(pt(a)));
  P" := C * R + R * P + P * P';
  /* + denotes union,* denotes intersection */
  p" := project (r join p) on P";
  con(a') := p"(P");
  pt(cd(a)) := a'; cd(a') := cd(a);
  cd(pt(pt(a)) = a'; pt(a') = pt(pt(a));
  cp(a) := a'; cp(pt(a)) := a'
 END
END
```

It is clear that the number of iterations in ContractAC is identical to the number needed for CONTRACT.

During the tree contraction phase, links between a contracting parent and its contracted nodes are established. Let $T' = \langle A', E' \rangle$ be the join tree resulting from applying ContractAC to T, such that $A' = A \cup A''$ where A'' includes all the nodes created in the tree contraction phase, and $(a, a') \in E'$ iff $a' = cp(a)$, i.e., a' is the contracting parent of a. The tree expansion phase starts from the root node of T' and propagates the solutions from root to leaves. Initially, the root is marked. Whenever the parent of a node is marked, the solutions can be computed for the node and then the node is marked.

```
Algorithm ExpandAC: Tree Expansion Phase
Input: result of ContractAC T' = <A',E'>;
Output: minimal network;

marked(root) := 1;

Iterate the following procedure the same
number of times as for ContractAC:

In Parallel for a in A' \{root}
/* at most n nodes at each iteration */
BEGIN
 IF (marked(cp(a)) THEN
 BEGIN
  r(R) := con(a); p(P) := con(cp(a));
  r := r semijoin p;
  marked(a) := 1
 END
END
```

The parallel AC algorithm ParAC simply applies ContractAC to T and then applies ExpandAC to T'.

```
Algorithm ParAC: Parallel Arc Consistency
Input: rooted join tree T;
Output: minimal network T":
BEGIN
 T' = ContractAC(T);
 T" = ExpandAC(T')
END
```

Theorem 2.1 *The result of applying* ParAC *to a join tree T is an arc consistent join tree whose constraint network is minimal and equivalent to the constraint network of T.*

Theorem 2.2 *The algorithm* ParAC *takes $O(\log n)$ time using $O(n)$ processors in the EREW (Exclusive Read Exclusive Write) PRAM model, given a join tree of an acyclic constraint network with bounded width.*

Proof: see [13].

The procedures associated with RAKE and COMPRESS for arc consistency can be associated with other parallel tree contraction algorithms. By associating semijoin with PRUNE and associating join and projection with BYPASS in the algorithm given by [1], arc consistency for an acyclic constraint network of bounded width can be done optimally in $O(\log n)$ time using $O(n/\log n)$ processors in an EREW PRAM.

3 Distributed Complexity

Here, we develop a distributed **AC** algorithm **DistAC** for reconfigurable interconnected processors with distributed memory and asynchronous communication. Let the nodes and edges of a join network map to processors and bidirectional channels in a distributed computing network, respectively. The algorithm is uniform: all processors have the same program. Let $r(R)$ be the local constraint and **propagate** be a subroutine for propagating the local constraint to its neighbors.

```
propagate:
 FOR (all channel c) send r(R) to c

Algorithm DistAC: Distributed AC
 propagate;
 LOOP
 BEGIN
  s := r;
  FOR (all channel c)
   IF (there is a message at channel c) THEN
   BEGIN
    receive r1(R1) from c;
    s := s semijoin r1;
   END
  IF s =\= r THEN
  BEGIN r := s; propagate END
 END
```

Proposition 3.1 *If the width of constraint network CN is bounded by a constant, the complexity of* DistAC *is $O(n)$, where $n = size(CN)$.*

Proposition 3.2 *For a join tree JT of an acyclic constraint network of bounded width and JT is of bounded degree, the complexity of* DistAC *is $\Theta(D)$ where D is the diameter of JT.*

Theorem 3.1 *Let n and w be the size and width of an acyclic constraint network ACN. One can construct a balanced binary join tree such that its acyclic constraint network ACN' is equivalent to ACN with $size(ACN') = poly(n)$ and $width(ACN') \leq 3w$.*

Proof: see [13].

Thus, there exists a mapping from an acyclic constraint network of size n with bounded width to a network of $poly(n)$ processors and it takes $O(\log n)$ time to find the minimal network.

4 Conclusions

We have presented parallel and distributed algorithms for solving FCSPs and shown that for an FCSP that can be represented by an acyclic constraint network of bounded width, there are efficient algorithms in both parallel and distributed environments. The bounded width property of acyclic constraint networks characterizes a set of tractable FCSPs [2] as well as efficiently parallelizable FCSPs. It is not generally true that a problem solvable in linear sequential time also has an efficient parallel algorithm, but it does happen to be the case for FCSPs.

Acknowledgements

We wish to thank Feng Gao, Nick Pippenger and Runping Qi for valuable suggestions and comments. The first author is supported by the University Graduate Fellowship from University of British Columbia. This research was supported by the Natural Sciences and Engineering Research Council and the Institute for Robotics and Intelligent Systems.

References

[1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[2] R. Dechter. Constraint networks: A survey. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. Wiley, N.Y., 1991. (to appear).

[3] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):257–388, April 1989.

[4] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11), November 1978.

[5] S. Kasif and A. L. Delcher. Analysis of local consistency in parallel constraint satisfaction networks. In *Proc. AAAI Symposium on Constraint Based Reasoning, Stanford*, pages 154 – 163, 1991.

[6] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[7] A. K. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205 – 211. Wiley, N.Y., 1987.

[8] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65 – 74, 1985.

[9] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[10] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proc. 26th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 478–489, 1985.

[11] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

[12] G. Shafer and P. P. Shenoy. Local computation in hypertrees. Technical report, School of Business, University of Kansas, August 1988. Working paper 201.

[13] Ying Zhang and Alan K. Mackworth. Parallel and distributed algorithms for constraint networks. Technical Report 91-6, Department of Computer Science, Univ. of British Columbia, May 1991.