Parallel Arc Consistency Algorithms for Pre-processing Constraint Satisfaction Problems

by
James Michael Conrad

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering

Raleigh
1992

Approved By:

Chairman of Advisory Committee

ABSTRACT

CONRAD, JAMES MICHAEL *Parallel Arc Consistency Algorithms for Pre-processing Constraint Satisfaction Problems* (Under the direction of Dr. Dharma P. Agrawal).

Constraint satisfaction problems (CSPs) are prevalent in artificial intelligence applications. Versions of CSPs occur in areas such as resource scheduling, semantic information processing, theorem proving, image processing and database retrieval. Backtracking is often used to solve these CSPs. Straightforward backtracking algorithms, however, are inadequate for solving large problems because they exhibit an excessive amount of thrashing. One approach is to pre-process the problem by eliminating variable assignments which can never result in a solution. This can be done by using consistency algorithms to pre-process a network of constraints before the tree search.

In this dissertation we introduce three Static Parallel Arc Consistency algorithms (SPAC-1, SPAC-2 and SPAC-3) designed for any general-purpose Multiple Instruction-stream, Multiple Data-stream (MIMD) computer. These algorithms are unique because specific parts of a CSP are assigned to specific processors (unlike previous approaches). We ensure arc consistency of a finite domain binary constraint network. Sample traces characterize the work performed by each processor of a parallel computer. Through simulation and actual machine experimentation we measure work performed by the SPAC algorithms and compare it with work performed by existing sequential algorithms, AC-1 and AC-3. Results show that our parallel arc consistency algorithms can be effectively used to pre-process a constraint network.

The algorithms' effectiveness is further demonstrated for large constraint networks on a distributed memory multiprocessor. Several graph partitioning algorithms used for load balancing are examined, along with different communications frequencies. We also propose the use of these Static Parallel Arc Consistency algorithms to aid in the design of interconnection link assignment for multi-chip module packaging.

Keywords: Arc Consistency, Artificial Intelligence, Constraint Satisfaction, Graph Partitioning, Multi-Chip Modules, Parallel Processing, Simulation.

To Stephanie, Jay, Mom, and Dad

BIOGRAPHY

James M. Conrad was born in Des Plaines, IL, on June 18, 1962. He attended Forest View High School in Arlington Heights, IL, and graduated in June, 1980. He attended the University of Illinois in Urbana, IL from August, 1980 to May, 1984, where he received his B.S. in Computer Science. He started working for IBM in July, 1984, as a Programmer, and left in September, 1990, as a Senior Associate Programmer.

While working at IBM, James earned his M.S. in Computer Engineering from North Carolina State University in May, 1987. He continued his studies, and in December, 1990, began working as an instructor and research assistant with the department of Electrical and Computer Engineering.

He is a member of the ACM, IEEE, IEEE Computing Society, AAAI, and SIAM. He served as Associate Editor-in-Chief on the Student Editorial Board of *IEEE Computing Futures* Magazine.

His research interests include parallel system architectures, parallel programming, search, and constraint satisfaction.

LIST OF PUBLICATIONS

1. Conrad, James M. and Dharma P. Agrawal, "Performance of an Asynchronous Parallel Algorithm on a Generic Multiprocessor Simulator," *Proceedings of the Pittsburgh Conference on Simulation and Modeling*, Pittsburgh, PA, pp. 1290-1297, May 1991.
2. Conrad, James M., Dennis R. Bahler and James A. Bowen, "Static Parallel Arc Consistency in Constraint Satisfaction," *Proceedings of the 1991 International*

Symposium on Methodologies for Intelligent Systems, Charlotte, NC, pp. 500-509, October 1991.

3. Conrad, James M., Dharma P. Agrawal, and Dennis R. Bahler, “Scalable Static Parallel Arc Consistency Algorithms for Shared Memory Computers,” *Proceedings of the Sixth International Parallel Processing Symposium*, Beverly Hills, CA, pp. 242-249, March 1992.
4. Teague, W. Michael, James M. Conrad, and Dharma P. Agrawal, “An Innovative Model of Generic Multiprocessors using Simulation,” *Proceedings of the Pittsburgh Conference on Simulation and Modeling*, Pittsburgh, PA, May 1992 (to appear).
5. Conrad, James M. and Dharma P. Agrawal, “Distributed, Scalable, and Static Parallel Arc Consistency Algorithms on Private Memory Machines,” *Proceedings of the 12th International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992 (to appear).
6. Conrad, James M. and Dharma P. Agrawal, “A Graph Partitioning-based Load Balancing Strategy for a Distributed Memory Machine,” *Proceedings of the 21th International Conference on Parallel Processing*, St. Charles, IL, August 1992 (to appear).
7. Conrad, James M. and Dharma P. Agrawal, “Simulation of Generic Multiprocessor Configurations for Asynchronous Algorithms,” *International Journal in Computer Simulation, special Issue on Multiprocessor Networks*, 1993 (to appear).

Acknowledgements

I would like to thank the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Lab for use of their Sequent Symmetry, BBN Butterfly TC2000, and Intel i860 computers. I would also like to thank Oak Ridge National Lab for use of their Intel iPSC/2 computer. The generosity of the Department of Energy has helped me gather useful data on real and expensive machines.

I also want to thank Todd Cook for the use of his mean field annealing software, and Dr. David E. Van den Bout for the use of his Kernighan/Lin software.

I wish to thank the following people for the support they have given me during my graduate school experience:

To Vince Hamrick, Eric Scott, and Mike Teague, for their assistance with some of the parallel programming and data gathering. You helped shorten my learning curves by making my programming mistakes for me (and your work was not without reward). And especially to Mike, whose quest for knowledge wasn't satisfied, for sticking around to dabble in other ventures.

To my advisor, Dr. Dharma Agrawal, who always made sure I had a research or teaching assistantship each semester. You always did believe in me, even when I was down because of a roadblock. I am still amazed at your ability to give impeccable advice. I only regret that I did not always follow it.

To my committee, Drs. Bahler, Reeves, and Van den Bout, and to Dr. Bowen, for their honest opinions. I believe this dissertation is stronger because of your guidance.

To my parents, Ralph and Marilyn Conrad, for providing moral support, encourage-

ment, and the correct perspective of what's important in life.

To my Mother-in-law, Betty Warren, for her encouragement and help reviewing this thesis.

To my son Jay, for giving me a goal, the chance to spend more time with you.

And especially to my wife Stephanie for her patience, forgiveness, and understanding.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most of today's scientific advances involve the marriage of several technologies. This research is no exception. It is a combination of algorithm design, parallel architectures, parallel programming, simulation, and artificial intelligence. It is not our objective to be experts in all of these areas. However, we have culled our knowledge to make an innovative and novel contribution to the computing world.

The specific goals of this work are to:

- Develop several algorithms useful for artificial intelligence applications.
- Develop these algorithms for parallel processing machines.
- Test these algorithms through simulation.
- Prove the effectiveness of these algorithms through experimentation on actual parallel machines.
- Apply these algorithms to a non-trivial real-world problem.

The basic premise of this work is artificial intelligence. Winston [62] states, "Artificial Intelligence is the study of ideas that enable computers to be intelligent." Embodying of AI techniques can make computers more useful by solving problems important to human-kind. This work will, hopefully, advance this usefulness a bit further.

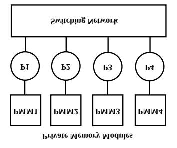
1.1 Parallel Computer Architectures

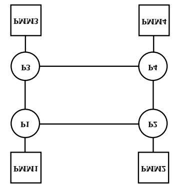
Parallel computers can be classified according to the instruction stream sequence executed by the computer and the data stream sequence used by the individual processors [26]. Stream classification of parallel systems includes *Single Instruction-stream, Multiple Data-stream* (SIMD) computers and *Multiple Instruction-stream, Multiple Data-stream* (MIMD) computers.

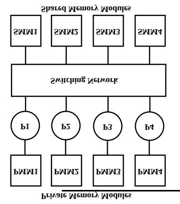
At a given instant, a MIMD parallel processor executes a different instruction on each of its processing elements. The processors may, however, require communication with each other, or may share resources. How the processors access memory or input/output devices totally depends on the underlying computer architecture. We classify MIMD computer architectures relative to the location and distribution of their memory modules. The three possible schemes of MIMD computers are: shared memory, distributed memory, and hybrid memory.

directly between the processors via the switching network. If one processor wants a variable value from the memory module of another processor, the processor in need of data submits a request in the form of a message to the other processor. The remote processor retrieves the information from its private memory and sends it to the requesting processor. One of the many network topologies, such as ring, mesh, or hypercube, could be selected to provide the inter-processor communication.

Figure 1.4 shows an example of a hybrid memory architecture, which combines the


Switching Network
P1
P2
P3
P4
PMM1
PMM2
PMM3
PMM4
Private Memory Modules


PMM3
PMM4
P3
P4
P1
P2
PMM1
PMM2


Shared Memory Modules
SMM1
SMM2
SMM3
SMM4
Switching Network
P1
P2
P3
P4
PMM1
PMM2
PMM3
PMM4
Private Memory Modules

1.2 Parallel Algorithms

Kung [41] defines a parallel algorithm for multiprocessors as a collection of concurrent *processes* that may operate simultaneously to solve a given problem. A process should be an identifiable subset of the entire problem, but can execute on one or more processors.

A problem usually requires interaction between the processes. If the program controlling a process needs to communicate with another process, it can send or receive information via a message, or it can read or store information via a globally available variable. Such points in the program are *interaction points*, and these points divide a process into *stages*.

A *synchronized parallel algorithm* consists of processes wherein "there exists a process such that some stage of the process is not activated until another process has finished a certain portion of its program."[41] For example, the processes $A + B$ and $C + D$ are part of the larger process $X = (A + B) \times (C + D)$. The process to compute X requires that both the processes $A + B$ and $C + D$ first complete, therefore, the beginning of process X is synchronized with the completion of the other two.

Much of the existing work on parallel algorithms has been devoted to synchronous algorithms, where this strict synchronism between successive steps of a parallel computation is observed, and processing of subtasks is done in lockstep [26]. In this work, we study a class of algorithms for multiprocessors called *asynchronous parallel algorithms* [41] (hereafter called asynchronous algorithms). These algorithms allow some degree of asynchronism between various steps or successive iterations. The most important characteristic of asynchronous algorithms is that whenever any processor needs a value of a global variable, it reads the most currently available value and continues execution using that information. The main objective of this dissertation is to verify that the specific steps of these algorithms can be efficiently programmed for many different types of multiprocessors.

Definition 1.1 *An asynchronous algorithm has the following properties [41]:*

1. *A set of global variables is available to all the processes.*
2. *When a stage of a process is complete, the process first reads some global variable(s). Then, based on the value of the variable(s) and the results from the previous processing stage, the process modifies some global variable(s), and then activates the next stage or terminates itself.*

The most important characteristic of asynchronous algorithms is that the processes never wait for inputs. Therefore, a process cannot have deadlock, waiting for synchronization from another process. In most existing work, asynchronous algorithms are executed on shared memory computers. We have modified these algorithms for a distributed memory environment due to the increasing popularity of multiprocessors.

In the following section, we consider one application area where asynchronous algorithms can be effectively used.

1.3 Constraint Satisfaction

Constraint satisfaction is a branch of artificial intelligence where one must find values for a group of variables, or parameters.

Mackworth [45] describes *constraint satisfaction problems* (CSPs) as "those in which one has a set of variables each to be instantiated in an associated domain and a set of Boolean constraints limiting the set of allowed values for specified subsets of the variables". To best represent this task in mathematical form, the following terms are defined:

Definition 1.2 *A* variable, *i, is a unit which is assigned a meaning, or value.*

Definition 1.3 *A* finite domain, *D_i, is a list of possible values that variable i can be given. We assume that each domain has the same cardinality (size).*

Definition 1.4 *A* value, *l_{i_x}, also called a label, is a specific assignment given to variable i from a list of possible items, that is, $l_{i_x} \in D_i$.*

Definition 1.5 *A* unary constraint, *R_i^1, is a relation by which if a value l_{i_x} from domain D_i is assigned to variable i, then the constraint R_i^1 is true if the relation is satisfied, that is $l_{i_x} \in R_i^1$.*

Definition 1.6 *A* binary constraint, *R_{ij}^2, is a relation by which if a value l_{i_x} from domain D_i is assigned to variable i,* **and** *value l_{j_y} from domain D_j is assigned to variable j, then the constraint R_{ij}^2 is true if the relation is satisfied, that is $(l_{i_x}, l_{j_y}) \in R_{ij}^2$.*

Definition 1.7 *Using a first order predicate logic formula, a constraint problem can be satisfied if:*

$$\begin{aligned}(\exists l_{1_x})(\exists l_{2_x})\ldots(\exists l_{n_x})(l_{1_x} \in D_1)(l_{2_x} \in D_2.)\ldots(l_{n_x} \in D_n)R_1^1(l_{1_x})\\ \wedge R_2^1(l_{2_x}) \wedge \ldots R_n^1(l_{n_x}) \wedge R_{12}^2(l_{1_x}, l_{2_x}) \wedge R_{13}^2(l_{1_x}, l_{3_x}) \wedge \ldots R_{n-1,n}^2(l_{n-1_x}, l_{n_x})\end{aligned} \qquad (1.1)$$

We only list R_{ij}^2 because for $j < i$, $R_{ij}^2 = R_{ji}^2$.

Definition 1.8 *A* constraint network *is a graphical representation of a specific CSP.*

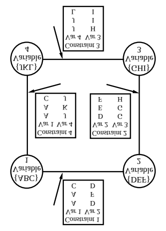
Figure 1.5 shows an example of a four variable constraint network. The circles represent the finite domain variables of the problem, and the boxes represent the constraints between the variables. Here the variable domains are single letters, and the constraints are sets of letter pairs which satisfy an assignment of the two variables.

We restrict our discussion of CSPs to the class where each constraint is binary or unary. Many problems with n-ary constraints can be converted to a binary equivalent [15]. Although Mackworth [44] suggests consistency algorithms can be extended to infinite domain problems, we only address finite domain problems.

1.4 Solving Constraint Satisfaction Problems

CSPs (also called the Consistent Labeling Problem) can be solved using various tree search algorithms like backtracking [30, 49], backmarking [20, 49], or forward-checking [24, 49]. Generally, backtracking algorithms are inadequate for solving a large constraint network because they exhibit an excessive amount of thrashing. Pre-processing the problem, by eliminating variable assignments which can never result in a solution, can reduce the work performed by a tree search algorithm. The task to eliminate variable assignments involves the use of network consistency algorithms to pre-process a network of constraints before the tree search. Such algorithms include node, arc, and path consistency [44, 47, 61]. The CSP with finite domain variables is NP-complete [19, 45].

Although much has been said about solving CSPs on sequential computers, researchers are now proposing solving these problems on parallel processing computers. Some contend that constraint satisfaction algorithms are sequential in nature, and


Constraint 3
Var 4 Var 3
J H
J I
L I
Variable 4
{JKL}
Variable 3
{GHI}
Constraint 4
Var 1 Var 4
A J
A K
C J
Constraint 2
Var 2 Var 3
D G
E G
F H
Variable 1
{ABC}
Variable 2
{DEF}
Constraint 1
Var 1 Var 2
A D
A F
C D

that using parallel computers cannot significantly improve the worst case performance [33, 58]. Other researchers believe that the execution time can be reduced for the average problem using parallel algorithms on parallel computers. Many authors have addressed CSPs using parallel tree search algorithms like a forward-checking algorithm [51, 52] and a depth first search with backjumping [7, 17].

Researchers have proposed and implemented parallel arc consistency algorithms on shared memory computers [22, 23, 54, 55]. Some have proposed distributed parallel arc consistency algorithms, but they did not make any attempt to actually implement their algorithms [34, 63, 64].

1.5 Applications and Motivation

In this dissertation we introduce three new Static Parallel Arc Consistency algorithms designed for any general-purpose MIMD computer. These algorithms are unique in that pre-partitioned sections of a CSP are assigned to specific processors (unlike previous approaches).

No one has measured the specific work performed by the individual processors using arc consistency techniques to pre-process a constraint network. Through simulation and actual machine experimentation, we measure work performed by each processor. Results show that our parallel arc consistency algorithm can be used to effectively pre-process a constraint network.

Since arc consistency algorithms can greatly reduce the search space of a CSP, it can be used as an effective pre-processor for many AI applications which are viewed as CSPs. Gu [22] lists several CSPs including: resource scheduling, detection of graph and subgraph isomorphism, the graph coloring problem, semantic information processing, theorem proving, image processing and object recognition, VLSI design, neural network computing, and database consistency checking. As examples, we illustrate two possible applications of our SPAC algorithms.

Nadel [50] describes an application based on reasoning about the timing of events. In his example, he represents actual activities and timing relationships as a CSP. If one examines these activities and timing relationships, the problem becomes a resource scheduling application. Larger and more complex scheduling problems can be addressed by pre-processing the CSP using our parallel arc consistency algorithms.

Waltz [61] analyzes a real word scene and represents objects in the scene with lines. He represents the physical possibilities of lines as constraints, and introduces an arc consistency algorithm (AC-2) to reduce the possibilities of the physical representation. Using Waltz's representation of video scenes and our parallel arc consistency algorithms, one could analyze more complex scenes, or analyze scenes in greater detail.

We propose the use of these Static Parallel Arc Consistency algorithms to aid in the design of interconnection link assignment for multi-chip module packaging. Ho, et. al. [25], have proposed a heuristic for assigning these links, but we examine combining Parallel Arc Consistency with Parallel Forward-Checking search to minimize the number of required layers.

1.6 Outline of this Work

In Chapter 2, we describe the concepts of node, arc, and path consistency and evaluate sequential and parallel arc consistency algorithms. In Chapter 3, we describe the task of assigning parts of the constraint network to processors. We then present our three parallel arc consistency algorithms based on the sequential arc consistency algorithms AC-1 and AC-3. Chapter 4 follows sample traces of our algorithms and compares our results with other parallel arc consistency algorithms. Chapter 5 defines the requirements for achieving the goals of simulating our algorithms on shared, distributed, and hybrid memory computers. We describe the programming implementation of the algorithms on commercial parallel processors as well. Chapter 6 presents simulation and actual results of pre-processing constraint networks. Large constraint networks are also tested on a distributed memory multicomputer. Chapter 7 examines two variations of our Distributed Static Parallel Arc Consistency algorithms which differ only in the frequency of communications. It also considers the effect on speedup of simple load balancing versus the more robust mean field annealing graph partitioning heuristic. The effect of arc consistency on the execution time of search is examined. Chapter 8 presents our proposal for using Static Parallel Arc Consistency algorithms as an aid in the assignment of inter-chip links for multi-chip module packaging. Finally, we give our conclusions in Chapter 9.

Chapter 2

Consistency Algorithms

2.1 Consistency Algorithm Definitions

To describe consistency algorithms, we refer to the network model of a CSP shown in Figure 1.5. We work exclusively with binary constraints, where the constraint between variables i and j is represented by an edge, and each edge consists of two directed arcs:

Definition 2.3 *A path of length two through a graph is* path consistent *from starting node i through node k to ending node j if and only if:*

$$\begin{aligned}&\forall(l_{i_x} \in D_i, l_{j_y} \in D_j), R_i^1(l_{i_x}) \wedge R_j^1(l_{j_y}) \wedge R_{ij}^2(l_{i_x}, l_{j_y}) \Rightarrow \\ &\qquad \exists l_{k_z} \in D_k, R_k^1(l_{k_z}) \wedge R_{ik}^2(l_{i_x}, l_{k_z}) \wedge R_{kj}^2(l_{k_z}, l_{j_y}) \qquad (2.3)\end{aligned}$$

Definition 2.4 k-*consistency is a concept in a graph where, in choosing $k-1$ variables which satisfy all constraints among the variables, adding the* k*th variable retains consistency of the* k *variables. Node, arc, and path consistency are 1-, 2-, and 3-consistency, respectively.*

Definition 2.5 *A* consistency check *is a comparison of a possible value of one variable with the possible value of another variable (e.g., using the constraint network of Figure 1.5, value A of D_1 is checked with value J of D_4). We use consistency checks to analyze the algorithmic performance.*

Definition 2.6 *The* complexity *of a sequential algorithm is the maximum number of steps the algorithm may take. These steps are identified as a basic unit of work, like an addition operation or a comparison of two values. Our step is a consistency check.*

Definition 2.7 *The* time complexity *of a parallel algorithms is the maximum number of steps, as a measure of time, that are required to solve the problem. This can be viewed as "start-to-stop" time. Some or all of the processors running a parallel algorithm may be active at any one time. Therefore, the* computational complexity *is the maximum number of steps, as a measure of work, that a parallel algorithm performs. This is often the time complexity times the number of processors.*

An example of a finite domain binary constraint problem, shown in Figure 1.5, helps us define the measures we use for the rest of this dissertation:

- The total number of constraint variables in a problem is n. The example has four variables.
- The number of edges (half the number of arcs) of a problem is e. The example has four edges, or eight arcs.
- The size of each domain is

Gu [22] has developed several arc consistency algorithms (which he calls Discrete Relaxation Algorithms) which are designed to run on special-purpose architectures. His DRA1 is designed to execute on a special-purpose sequential computer. His DRA2, DRA3, DRA5, mDRA2, and mDRA5 algorithms are designed to execute on his special-purpose VLSI parallel architecture. While his efforts show speedup in the range of three to ten orders of magnitude in the area of constraint satisfaction, we cannot compare his work to ours because his algorithms are designed for a specific platform, while ours are meant for commercially available MIMD computers.

Zhang and Mackworth [63, 64] introduce two parallel arc consistency algorithms for shared and distributed memory computers. Their algorithms assume that the constraint network is pre-processed, and that the transformed network is acyclic. They do not implement their algorithms on parallel computers.

Kasif [34] introduces a parallel arc consistency algorithm that also requires preprocessing the constraint network into propositional Horn satisfiability problems, which is not implemented on a parallel machine.

2.2 Sequential Arc Consistency Algorithms

Both the AC-1 and AC-3 algorithms use the Revise subroutine. Revise (Figure 2.1) examines an arc by choosing a variable value and checking if the other variable attached to the arc has at least one value which is consistent with the value for the first variable. This is done for each value in the first variable's domain.

A consistency check is either successful or unsuccessful. A check is successful if a constraint contains a paired tuple that matches the current values at each end point of the arc checked. If we examine

```
boolean function Revise(
```

We should note a fundamental difference between our interpretation of the Revise function and the interpretation of Samal [55] and Gu [22]. In particular, we refer to those lines of the function where we assign a value l_{i_x} to variable i and then check to see if a value l_{j_y} can be assigned to variable j which satisfies the constraint $R^2_{ij}(l_{i_x}, l_{j_y})$. Mackworth describes part of the Revise function [44] by:

if there is no $y \in D_j$ such that $R_{ij}(x, y)$ **then** ...

Our interpretation is that, once we find a value l_{j_y} which satisfies the constraint, we need not check any values of domain D_j until we check another value for variable i. If we examine