Scalable Parallel Arc Consistency Algorithms for Shared Memory Computers*

James M. Conrad and Dharma P. Agrawal
Electrical and Computer Engineering
North Carolina State University
Box 7911, Raleigh, NC 27695-7911
conrad/dpa@csl36h.csl.ncsu.edu

Dennis R. Bahler
Computer Science
North Carolina State University
Box 8206, Raleigh, NC 27695-8206
drb@adm.csc.ncsu.edu

Abstract

Constraint satisfaction problems (CSPs) are prevalent in artificial intelligence applications. One approach to assist in solving CSPs 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 paper we introduce three scalable Static Parallel Arc Consistency algorithms (SPAC-1, SPAC-2 and SPAC-3) designed for any general-purpose shared memory Multiple Instruction-stream, Multiple Data-stream (MIMD) computer. We ensure arc consistency of a finite domain binary constraint network. Through 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.

Keywords: *Arc Consistency, Artificial Intelligence, Constraint Satisfaction, Parallel Processing.*

1 Introduction

Mackworth [13] 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". Backtracking is often used to solve these CSPs. Straightforward backtracking algorithms, however, are inadequate for solving large constraint networks found in typical AI applications because they exhibit an excessive amount of thrashing. One approach to avoiding thrashing (but not the only one [7, 9]) 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. Algorithms have been described to achieve node, arc, and path consistency [13, 15].

Although much has been said about solving CSPs on sequential computers, more researchers are proposing solving these problems on parallel processing computers. Some contend that constraint satisfaction algorithms are inherently sequential, and that using parallel computers cannot significantly improve the worst case performance [10, 18]. Others, however, believe that *average-case* execution time can be reduced through the use of parallelism. Several authors have solved CSPs using parallel tree search algorithms similar to forward-checking [16] or depth-first search with backjumping [3, 6]. Still others have pre-processed constraint networks using parallel consistency algorithms [17, 8].

No one, however, has measured the specific work performed by the individual processors using arc consistency techniques to pre-process a constraint network. In this paper we introduce three new scalable Static Parallel Arc Consistency algorithms designed for any general-purpose parallel processing computer and measure work performed by each processor on a Sequent Symmetry multiprocessor. Results show that our parallel arc consistency algorithm can be used to effectively pre-process a constraint network. We also address scaling our algorithm to solve larger AI applications.

---

*This work has been supported in part by the Army Research Office, under grant DAAL03-89-K-0142.

2 Consistency Algorithms

To describe consistency algorithms, we refer to the network model of a CSP. We are exclusively concerned with binary constraints, where the constraint between variables i and j is represented by an edge, and each edge consists of two directed arcs: $arc(i,j)$ and $arc(j,i)$.

An arc $arc(i,j)$ is *arc consistent* if, for each value in the domain D_i for i, there is at least one value in D_j that satisfies the constraint of the arc. Arcs can be made consistent by performing the operation $D_i \leftarrow D_i \cap \{l_{i_x} | (\exists l_{j_y})(l_{j_y} \in D_j) \wedge R^2_{ij}(l_{i_x}, l_{j_y})\}$ for every specific assignment l_{i_x} given to variable v_i from a domain D_i of possible items, where R^2_{ij} is a binary constraint relation containing tuples of values satisfying the constraint.

In this paper, we measure the size of a problem as follows:

- the total number of variables in a problem is denoted by n;
- the number of edges (half the number of arcs) of a problem is e;
- the size of each domain is a.

We analyze several algorithms below using these measures to count consistency checks. A *consistency check* is a comparison of a possible value of one variable with the possible value of another variable. This measurement is used in Mackworth and Freuder's paper on consistency algorithm performance [14]. We address arc consistency only of constraint networks with finite domain variables and binary constraints. Although each algorithm ensures node consistency, we do not analyze node consistency algorithms. Similarly, path consistency is beyond our scope here.

2.1 Existing Sequential Arc Consistency Algorithms

Four popular sequential arc consistency (AC) algorithms are Mackworth's AC-1 and AC-3 [13], Waltz's AC-2 [19], and Mohr and Henderson's AC-4 [15]. We do not discuss AC-2 since it is simply a special case of the AC-3 algorithm. Although AC-4 is considered the best sequential arc consistency algorithm, Samal [17] observed that the parallel version of AC-4 performs poorly. We do not, therefore, consider AC-4 in our investigations.

Both the AC-1 and AC-3 algorithms use the Revise subroutine. Revise 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 of the first variable. This is done for each value in the first variable's domain. Each time AC-1 or AC-3 calls the Revise subroutine, up to a values of the first domain and up to a values of the other domain are checked, so up to a^2 checks are performed.

AC-1 is a simple but inefficient sequential algorithm to ensure arc consistency. AC-1 calls Revise for every arc of the constraint network graph. If Revise removes a value from any variable's domain, a *change* bit is set. Once AC-1 has finished checking each arc, it examines the *change* bit. If the bit is set, AC-1 rechecks all of its arcs again. AC-1 ends when it checks all of its arcs and does not remove any values. The worst case complexity of AC-1 is reported by Mackworth and Freuder as $\mathbf{O}(a^3ne)$.

It is clear that AC-1's inefficiencies lie in the fact that when only one value is removed, all variables are checked again. To remedy this, AC-3 maintains a queue of arcs. The queue is first built by adding all of the arcs of the constraint network G to the queue, Q. If Revise removes a value from a variable, AC-3 rechecks the arcs which are attached to that variable (except the current arc and node consistent $arc(i,i)$) by adding the arcs to the queue. That is, given that a value in D_i is removed when processing $arc(i,j)$, then:

$$Q \leftarrow Q \cup \{arc(m,i) | arc(m,i) \in G, m \neq i, m \neq j\} \quad (1)$$

AC-3 terminates when the queue is empty. Mackworth and Freuder have shown that, since the queue may grow and shrink during the execution of AC-3, the complexity is less than AC-1. They show that AC-3 is bounded below by $\Omega(a^2e)$ and above by $\mathbf{O}(a^3e)$.

2.2 Existing Parallel Arc Consistency Algorithms

Samal [17] introduced several parallel versions of arc consistency algorithms. He assumes his model computer has an infinite number of processors in a shared memory environment.

Samal employs a parallel version of Mackworth's Revise, called Previse. In Previse, all consistency checks of a single arc are executed concurrently. If the domain size of each variable of an arc is a, then up to a^2 checks are made in one time unit. Therefore, up to a^2 processors are needed and take $\mathbf{O}(1)$ time. Total work performed is $\mathbf{O}(a^2)$, the same as Revise.

Samal's parallel version of Mackworth's AC-1 is Parallel Arc Consistency algorithm #1 (PAC-1). Although this algorithm requires a large number of processors, PAC-1 ensures consistency quickly since all possible variable assignments of all the arcs are checked concurrently. PAC-1 checks each of the $2e$ arcs of the problem concurrently, while Previse executes up to a^2 consistency checks concurrently. Therefore, each iteration of the PAC-1 loop requires $\mathbf{O}(a^2e)$ processors. If only one value is removed from each variable each loop, up to na loops are executed. The total time complexity is therefore $\mathbf{O}(na)$, and the total work performed is $\mathbf{O}(a^3ne)$.

The parallel version of AC-3 is PAC-3. PAC-3 builds a globally available queue like AC-3, and uses Previse like AC-1. Again, only arcs in the queue are checked, and so we can check up to $2e$ arcs concurrently. Generally, the PAC-3 queue is smaller than the set of all edges, but in the worst case PAC-3 behaves just like PAC-1. In this case, the time and computational complexities are the same as PAC-1. A drawback of PAC-3 is that the time to build the queue is not small and may affect the total computational time.

Gu [8] developed several arc consistency algorithms (which he calls Discrete Relaxation Algorithms) designed to run on a special-purpose computer architectures. His DRA1 was designed to execute on a special-purpose sequential computer. His DRA2, DRA3, DRA5, mDRA2, and mDRA5 algorithms were designed to execute on his special-purpose VLSI parallel architecture computer. 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 designed for commercially available shared memory MIMD computers.

3 Static Parallel Arc Consistency

We have developed three parallel arc consistency algorithms which are similar to Mackworth's AC-1 and AC-3 algorithms. The unique characteristic of our algorithms is the relationship between constraint variables and parallel processor architectures. Our algorithms are unique because specific yet large parts of a CSP are statically assigned to specific processors (unlike previous approaches). These assignments allow our algorithm to run on any MIMD computer, and allow us to scale a problem simply by adding more processors. Each processor of our target multiprocessor ensures local arc consistency of its assigned part of the network.

An algorithm which dynamically assigns tasks would best balance the workload of a parallel processing computer, but the task assignment and recombination overheads may overtake any benefits of load balancing [2]. A static approach is desirable only if work can be assigned equitably.

Although consistency checks of arcs represent the work of consistency algorithms, distributing network information can have an impact on the speed of an algorithm. For example, if one assigns arcs to processors, several processors may have the task of removing values from a single domain. The variable, therefore, has to be locked to prevent overwriting changes. This "test-and-set" operation [1] locks the domain from use until the variable write is complete. This lock causes processors to wait for the variable to become free, and slows the parallel computation. This is, in fact, how Samal implemented his PAC-1 and PAC-3 algorithms on actual computers.

We assign one or more constraint variables to each processor. With one constraint variable per processor the distribution of work is not necessarily equitable. If we assign several constraint variables to each processor, we can ensure that the number of arcs per processor is approximately equal.

Our algorithms are examples of *asynchronous parallel algorithms* [11]. These algorithms allow some degree of asynchronism between various steps or successive iterations. The most important characteristic of asynchronous parallel algorithms is that whenever any processor needs a global variable, it reads the most currently available value and continues execution using that information. It is because of this asynchronism that we can achieve a consistently high amount of speedup.

Our three Static Parallel Arc Consistency Algorithms, SPAC-1, SPAC-2, and SPAC-3, utilize a Boolean function called Static Parallel Revise (Sprevise), which is similar to Mackworth's Revise function. Sprevise checks an arc and removes all variable values which could never participate in a solution. If a value is removed, it informs the calling SPAC algorithm of a change (although this differs somewhat with SPAC-2 and is described below). Sprevise also checks to ensure that the variable domain still has at least one value. If it removes the last value from a domain, it sets the global flag *no_solution*, and all SPAC algorithms halt.

3.1 Static Parallel Arc Consistency Algorithm #1

The SPAC-1 algorithm (Figure 1) has been split into two components, the host algorithm and the node algorithm. The host part resides in only one processor and assigns work to other processors. The host maintains the *change* bit and determines if the arcs need to be rechecked. The host component of the SPAC-1 al-

```
procedure SPAC-1-HOST()
begin
  assign variables to processors;
  $change$ := TRUE; $no\_solution$ := FALSE
  while($change \wedge \neg no\_solution$) do
  begin
    $change$ := FALSE;
    ||for processing node $p$, $change = change \vee$ SPAC-1-NODE($p$);
  end
end

procedure SPAC-1-NODE($p$)
begin
  for each assigned variable $v_i$ do
    for each directed $arc(i, j)$ do
      $local\_change = local\_change \vee$ Sprevise($arc(i, j)$);
  return($local\_change$);
end
```

Figure 1: SPAC-1 Algorithm: Host Processor and Node Processor Components.

gorithm starts the node component of the algorithm. Each processor works with one or more assigned constraint network variables and the arcs which emanate from those variables.

The SPAC-1 algorithm is similar to Mackworth's AC-1 algorithm. All the arcs assigned to a processor are checked for consistency. One difference between SPAC-1 and AC-1 is that our algorithm ends only when **all** processors have verified local arc consistency. Once each processor has verified local arc consistency of the variables and arcs it was assigned, it waits for the host to check consistency of the entire network. If any processor has set the *change* bit, each processor again ensures local arc consistency by checking all of its arcs. The algorithm ends only when **all** processors have verified local arc consistency.

The power of this parallel algorithm lies in the fact that domain changes are available to all processors immediately. Therefore, values which are removed are never checked again by the Sprevise function. One disadvantage, however, is that the processors are somewhat loosely synchronized. Finished processors sit idle until all processors have completed checking their arcs.

The complexity of a parallel algorithm is important. Assume that we run the SPAC-1 algorithm on a computer with p processors, and each processor is assigned $2e/p$ constraint arcs. The possibility exists that only one value is removed from the entire constraint problem during each SPAC-1-NODE call, and that being during the consistency check of the last variable's value. Each pass makes up to a^2 consistency checks in Sprevise, therefore each pass in each processor takes up to $\mathbf{O}(a^2e/p)$ time units. If one value is removed from one of the p processors each pass, $\mathbf{O}(na)$ values can be removed from the entire constraint network. If the worst case time complexity is $\mathbf{O}(a^3ne/p)$, there is a possibility that all processors are active all the time. In this case, the complexity of work performed is p times the time complexity, or $\mathbf{O}(a^3ne)$. This, of course, is the worst case. On the average (as we show with the experimental results in Section 4.3), the actual work performed is much less.

3.2 Static Parallel Arc Consistency Algorithm #2

SPAC-1 also operates in the same manner as PAC-1 and AC-1, where all arcs are checked if one value in the network has been removed from a variable's domain. Therefore, SPAC-1 still suffers from the inefficiencies of the other algorithms. To better exploit the distributed nature of AC-1 and reduce the amount of unnecessary processing, we have developed another parallel arc consistency algorithm, SPAC-2.

The SPAC-2 algorithm (Figure 2) has been split into host and node components. The host part executes in only one processor and assigns work to other processors. The host maintains the *active_procs* variable. This variable is used by all processors to verify network arc consistency. The host component of the SPAC-2 algorithm starts the node component of the algorithm.

The SPAC-2 algorithm is similar to SPAC-1 algorithm with two exceptions. First, each processor has its own *change* bit. If processor x is assigned variable v_i and SPAC-2 running on processor x removes a value from the domain of D_i, processor x sets to "on" only the *change* bit of all processors y where $arc(i, j) \in E$ and v_j has been assigned to processor y. Therefore, processor x does not cause processor z to perform work when the results of processor z do not depend on the results of processor x. In the AC-1 and SPAC-1 algorithms, the main arc consistency algorithm modifies the *change* bit. In the SPAC-2 algorithm, Sprevise-2 actually modifies the *change* bits. This means that processors are notified of variable value changes earlier than if only the main arc consistency algorithm modified the *change* bit.

The second difference is the *active_procs* variable. This variable counter is used to signify the number of processors starting and stopping local arc consistency. When a processor starts its local arc consistency, it increments the counter. When it completes local arc consistency, it decrements the counter. When a node processor running SPAC-2-NODE observes that the *active_procs* variable is zero, it ends execution of SPAC-2-NODE.

```
procedure SPAC-2-HOST()
begin
  $active\_procs$ := 0; $no\_solution$ := FALSE;
  assign variables to processors;
  || for each processing node $p$, SPAC-2-NODE($p$);
end

procedure SPAC-2-NODE($p$)
begin
  $change[p]$ := TRUE;
  while(($change[p] \vee active\_procs > 0) \wedge \neg no\_solution$))
  begin
    while($change[p] \wedge \neg no\_solution$) do
    begin
      $active\_procs$ + +;
      while($change[p] \wedge \neg no\_solution$) do
      begin
        $change[p]$ := FALSE;
        for each assigned variable $v_i$ do
          for each directed $arc(i, j)$ do
            Sprevise-2($arc(i, j)$);
      end
      $active\_procs$ − −;
    end
  end
end
```

Figure 2: SPAC-2 Algorithm: Host Processor and Node Processor Components.

With the SPAC-2 algorithm we see that changes are "localized," so only processors which have an arc connected to a changing variable are informed of the change. This differs from the classic AC-1, Samal's PAC-1, and the SPAC-1 algorithms. In the SPAC-2 algorithm, each processor with a variable connected by a constraint to the changed variable rechecks **all** of its arcs. This selective rechecking still prevents **all** processors from checking their arcs, even if no changes to any connected variables are made. Processors which have verified local arc consistency, therefore, can immediately start useful processing when they are provided new variable domain data.

The SPAC-2 complexity analysis is identical to that for SPAC-1. If SPAC-2 removes only one value from some domain, that being on the last consistency check of the last active processor, SPAC-2 performs identically to SPAC-1. We should notice a slight improvement in speed over SPAC-1 in the average case because SPAC-2 in not synchronized, and Sprevise-2 updates *change* bits frequently.

3.3 Static Parallel Arc Consistency Algorithm #3

All parallel and sequential arc consistency algorithms based on AC-1 operate in the same manner, where all or many arcs are checked if one value in the network has been removed from a variable's domain. Therefore, even SPAC-2 still suffers from the inefficiencies of these AC-1 based algorithms where all arcs on a processor are checked when only a few arcs need to be checked. We have developed another parallel arc consistency algorithm to exploit the efficiency of AC-3 and reduce the amount of unnecessary processing. The Static Parallel Arc Consistency #3 algorithm (SPAC-3) is listed in Figure 3.

Our parallel arc consistency algorithm differs from Samal's PAC-3 algorithm in that each processor works with one or more variables and the constraints which contain the variables. For example, if variable v_i is assigned to processor m, processor m works with values l_{i_x} of domain D_i and all constraints represented by arcs $arc(i, j)$. Variable v_j may be assigned to processor m or any other processor.

Our algorithm also differs from PAC-3 with respect to the queue of arcs to be processed. Samal assumes a shared memory parallel processing model with one queue available for all processors to use. In SPAC-3 each processor maintains its own queue of arcs to make consistent. The queues hold only the arcs that have been assigned to that processor, i.e., only work involving assigned variables and all arcs that emanate from those variables. If Sprevise indicates that a value has been deleted from a variable's domain, the queue of arcs of **only** the processors that use that variable are modified according to the equation $Q \leftarrow Q \cup \{arc(m, i) | arc(m, i) \in G, m \neq i, m \neq j\}$.

Like SPAC-1 and SPAC-2, SPAC-3 has host and node components. The host part executes in only one processor and assigns work to other processors. The host maintains the *active_procs* variable. This variable is used by all processors to verify network arc consistency, as in the SPAC-2 algorithm. The host component of the SPAC-3 algorithm starts the node component of the algorithm.

The complexity of a SPAC-3 is similar to the complexity of AC-3. Assume that we run the algorithm on a computer with p processors, and each processor is assigned $2e/p$ arcs (and the variables associated with those arcs). The queue of arcs on each processor grows and shrinks during execution. The possibility exists that only one value is removed from each successful call to Sprevise on any of the processors. Mackworth and Freuder [14] show that the largest number of arcs added to the queues is $a(2e - n)$. These arcs are added to the queues of p processors, so the greatest number of original and added arcs checked on any of the processors is $(2e + a(2e - n))/p$. Sprevise makes up to a^2 consistency checks. For all constraint networks, $e \geq n - 1$, so time complexity becomes $\mathbf{O}(a^3 e/p)$. If the worst case time complexity occurs, there is a

```
procedure SPAC-3-HOST()
begin
  $active\_procs$ := 0; $no\_solution$ := FALSE;
  assign variables to processors;
  || for each processing node $p$, SPAC-3-NODE($p$);
end

procedure SPAC-3-NODE($p$)
begin
  for each assigned variable $v_i$
    $Q[p] \leftarrow Q[p] \cup \{arc(i,j) | arc(i,j) \in G, i \neq j\}$
  while$(((Q[p] \neq \phi) \vee active\_procs > 0) \wedge \neg no\_solution))$
  begin
    while$((Q[p] \neq \phi) \wedge \neg no\_solution)$ do
    begin
      $active\_procs + +$;
      while$((Q[p] \neq \phi) \wedge \neg no\_solution)$ do
      begin
        for $arc(i,j) \in Q[p]$ do
        begin
          $Q[p] := Q[p] - \{arc(i,j)\}$;
          if(Sprevise($arc(i,j)$))
            for each $arc(m,i)$, where $m$ is on processor $p$
            $Q[p] := Q[p] \cup$;
              $\{arc(m,i) | arc(m,i) \in E, m \neq i, m \neq j\}$;
        end
      end
      $active\_procs - -$;
    end
  end
end
```

Figure 3: SPAC-3 Algorithm: Host Processor and Node Processor Components.

possibility that all processors are active all the time. In this case, the complexity of work performed is p times the time complexity, or $\mathbf{O}(a^3e)$. This, of course, is the worst case. On the average (as shown in the experimental results in Section 4.3), the actual work performed, as with SPAC-1 and SPAC-2, is much less.

4 Implementation and Measurement

4.1 Parallel Execution on the Sequent Symmetry

We have implemented the SPAC algorithms on a Sequent Symmetry shared memory multiprocessor (a tightly-coupled MIMD computer). The Symmetry S81 has twenty eight 32-bit processors, 64-Kbytes of cache memory per processor, and 32-Mbytes of shared memory.

We programmed the Sequent computer such that one processor reads the data file and loads the shared memory with the constraint information. This processor then starts the remaining processors. Once the processors complete the work, the first processor reads the shared memory and records the results.

All constraint variables, values, and relations are stored in shared memory, and each processor's counting variables are stored in private memory. The constraint relations themselves are never modified, and hence, this data can be copied into a processor's cache with no coherency problems.

The variable domain values which are read by many processors but written by one, the counting variable *active_procs*, the flag *no_solution*, and the *change* bits were initially implemented as test-and-set variables. Only one processor at a time is allow to access each variable and this is achieved by employing special "lock" and "unlock" operators provided by the Sequent C library. Because of the asynchronous nature of our algorithm, we found that we did not need to use the test-and-set operation for most variables. Earlier trials ([4, 5]) found that locking these variables caused large, unnecessary delays for data accesses. Therefore, the only locked variable is *active_procs*.

Executing the SPAC-3 algorithm on a shared memory machine posed additional difficulties. The underlying structure of the queue of arcs could cause a large bottleneck if a single space in memory were allocated. We, therefore, had to ensure that adding an arc to one queue would not cause other queues to be locked. We solved this by maintaining separate queues and free lists for each processor, and locking the queue only when an arc is added.

4.2 Network Data

We developed many sample constraint networks to use as the data for experimental trials. The networks are computer-generated and have some attributes determined by random processes. The characteristics of the computer-generated networks are:

- Size: problems contain 20, 40, and 60 variables.
- Domains: each variable has a domain of 10 values.
- Solutions: Each problem contains one specific solution. That is, if the constraint network were solved using a tree search algorithm, only one consistent assignment could be made.
- Connectivity: The connectivity (degree) of each variable is a uniformly distributed random number from two to ten. The order of the binary constraints of each edge has no bearing on the consistency check complexity analysis.
- Satisfiability: The relative satisfiability is the success rate of a consistency check. More specifically, it is the ratio of relation value pairs to the total

possible pairs (domain × domain). The satisfiability of our networks are (15 relation pairs)/(10 domain values × 10 domain values)= 0.15.

While gathering the experimental data, ten random constraint networks were run for each of the configurations.

4.3 Metrics and Results

From the performance point of view, we limit our measurements to the algorithmic parameter speedup. Speedup is the effective speed of a parallel computation in operations executed per time unit of a parallel computer with p processors [12]. This is expressed as:

$$speedup = S(p) = \frac{time(sequential\ algorithm)}{time(parallel\ algorithm)} \quad (2)$$

We interpret "time" as the number of consistency checks for the sequential algorithm and processing cycles for the parallel algorithm.

The results of our measurements suggest that our parallel algorithms provide good speedup. We compute SPAC-1, SPAC-2, and SPAC-3 speedup based on the AC-3 sequential algorithm.

Execution of SPAC-1 on a four-processor computer configuration yields speedup of between 2.74 and 3.07, SPAC-2 yields a range of 2.80 to 3.14, and SPAC-3 yields a range of 2.97 to 3.50.

Execution of SPAC-1 on a ten-processor computer configuration yields speedup of between 5.43 and 6.41, SPAC-2 yields a range of 5.44 to 6.64, and SPAC-3 yields a range of 5.35 to 7.40.

Execution of SPAC-1 on a 20-processor computer configuration yields speedup of between 9.28 and 11.68, SPAC-2 yields a range of 10.46 to 11.84, and SPAC-3 yields a range of 10.44 to 12.68.

Actual speedup measurements are shown in Figure 4.

Scaling using Parallelism

Since arc consistency algorithms can greatly reduce the search space of a CSP, they can be used as an effective pre-processor for many AI applications which are viewed as CSPs. Gu [8] lists several CSP problems 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.

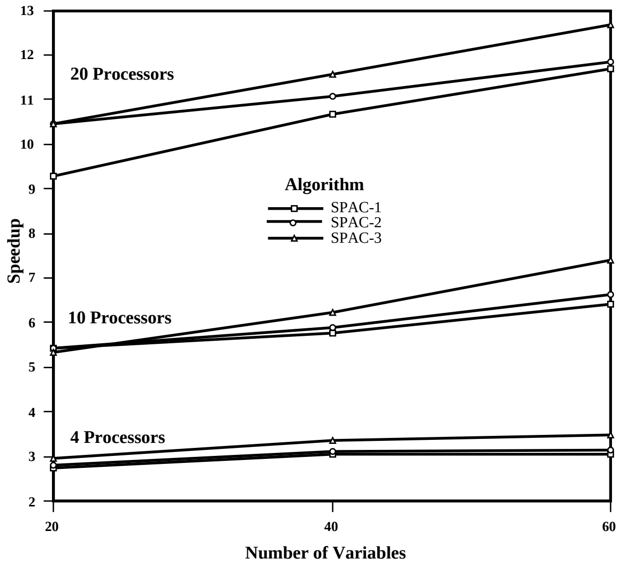

Figure 4: Speedup of SPAC over AC for a Shared Memory Machine

The constraint networks examined represent small problems with a single solution. These networks can be scaled to larger problems by processing more constraint variables on each processor. Also, due to the distributed nature of the algorithms, problems can also be scaled to a greater number of processors.

6 Conclusions

We have reviewed several arc consistency algorithms for sequential and parallel processing computers. We introduced three parallel arc consistency algorithms: SPAC-1, SPAC-2, and SPAC-3, and compared our algorithms with existing algorithms. We ran the algorithms in a parallel processing environment and compared their performances with that of Mackworth's AC-3 algorithm. SPAC-1 achieved speedup of 51 to 76% of linear speedup over the AC-3 algorithm. SPAC-2 achieved speedup of 52 to 78% of linear speedup over the AC-3 algorithm. SPAC-3 achieved speedup of 52 to 87% of possible speedup over the AC-3 algorithm.

Our future work may also include making actual computation trials on other parallel processors, such as the Intel iPSC/2 distributed memory computer and the BBN Butterfly TC2000 shared/hybrid memory computer.

Acknowledgements

The authors would like to thank the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Lab for use of their Sequent Symmetry Computer, and Vince Hamrick, Eric Scott, and Mike Teague for their programming assistance.

References

[1] Agrawal, Dharma P., *Advanced Computer Architecture (Tutorial Text)*, IEEE Computer Society Press, 1986.

[2] Burkhart, Helmar and Roland Millen, "Performance Measurement Tools in a Multiprocessor Environment," *IEEE Transactions on Computers*, Vol. 38, No. 5, pp. 725-737, May 1989.

[3] Collin, Zeev and Rina Dechter, "A Distributed Solution to the Network Consistency Problem," *Proceedings of the 1990 International Symposium on Methodologies for Intelligent Systems*, pp. 242-251, October, 1990.

[4] 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*, 1992 (to appear).

[5] 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.

[6] Freuder, Eugene C. and M.J. Quinn, "Parallelism in an Algorithm that takes Advantage of Stable Sets of Variables to Solve Constraint Satisfaction Problems," Technical Report 85-21, Dept. of Comp. Science, U of NH, 1985.

[7] Gaschnig, J., "Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisficing Assignment Problems," *Proc. of the 2nd Nat. Conf. of the Canadian Society for Comp. Studies of Intel.*, 1978.

[8] Gu, Jun, "Parallel Algorithms and Architectures for Very Fast AI Search," Technical Report UUCS-TR-88-005, PhD Dissertation, Department of Computer Science, University of Utah, Salt Lake City, UT, August 1989.

[9] Haralick, Robert M., and Gordon L. Elliott, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems," *Artificial Intelligence*, No. 14, pp. 263-313, 1980.

[10] Kasif, Simon, "On the Parallel Complexity of Some Constraint Satisfaction Problems," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 349-353, 1986.

[11] Kung, H.T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," in *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub, editor, Academic Press, pp. 153-200, 1976.

[12] Lee, Ruby Bei-Loh, "Empirical Results on the Speed, Efficiency, Redundancy, and Quality of Parallel Computations," *Proceedings of the International Conference on Parallel Processing*, pp. 91-100, August 1980.

[13] Mackworth, Alan K., "Consistency in Networks of Relations," *Artificial Intelligence*, No. 8, pp. 99-118, 1977.

[14] Mackworth, Alan K. and Eugene C. Freuder, "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence*, No. 25, pp. 65-74, 1985.

[15] Mohr, Roger and Thomas C. Henderson, "Arc and Path Consistency Revisited," *Artificial Intelligence*, No. 28, pp. 225-233, 1986.

[16] Natarajan, K.S. and Vivek Sarkar, "Processor Scheduling Algorithms for Constraint Satisfaction Search Problems," *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 140-147, August 1988.

[17] Samal, Ashok Kumar, "Parallel Split-Level Relaxation," PhD Dissertation, Department of Computer Science, University of Utah, August 1988.

[18] Swain, Michael J., "Comments on Samal and Henderson: 'Parallel Consistent Labeling Algorithms'", *International Journal of Parallel Programming*, Vol. 17, No. 6, pp. 523-528, 1988.

[19] Waltz, D.L., "Understanding Line Drawings of Scenes with Shadows," In *The Psychology of Computer Vision*, P.H. Winston - Ed., McGraw-Hill: NY, pp. 19-91, 1975.

Advance Copy: Proceedings of the 1992 International Parallel Processing Symposium, Beverly Hills, CA, March 1992.