A GRAPH PARTITIONING-BASED LOAD BALANCING STRATEGY FOR A DISTRIBUTED MEMORY MACHINE*

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

Abstract

The use of arc consistency algorithms can greatly reduce the search space of a constraint satisfaction problem. Parallel arc consistency algorithms exist for shared and distributed memory machines. Like many parallel algorithms, parameters such as distribution of work, frequency of message passing, and load balancing greatly affect the algorithm's potential speedup. This paper examines two variations of a parallel arc consistency algorithm which differ only in the frequency of communications. This paper also examines the effect of simple load balancing versus the more robust mean field annealing graph partitioning heuristic on speedup. Through actual machine experimentation we determine time required for the different message frequencies and work assignments. Results indicate that the configuration with less frequent communications and the robust load balancing yields the best speedup.

Keywords: *Arc Consistency, Distributed Processing, Graph Partitioning, Mean Field Annealing, Load Balancing.*

Introduction

Constraint Satisfaction Problems (CSPs) are prevalent in artificial intelligence applications. Versions of CSPs occur in areas such as image processing, resource scheduling, semantic information processing, and database retrieval. Backtracking is often used to solve these CSPs. Straightforward backtracking algorithms, however, are inadequate for handling large constraint networks found in typical AI applications because they exhibit an excessive amount of thrashing. To avoid this, one can pre-process the problem by eliminating variable assignments which can never participate in a solution. This can be achieved by using node, arc, and path consistency algorithms [10] to filter a network of constraints before the tree search.

We have developed [2, 3] several parallel arc consistency algorithms which are based on existing sequential algorithms. Unlike previous approaches, our algorithms uniquely assign large parts of a CSP to specific processors. This unique characteristic of our algorithms defines a relationship between constraint variables and parallel processor architecture. Because of the nature of our representation of CSPs, we can use graph partitioning heuristics to assign work statically to processors. Partitioning algorithms include the Kernighan-Lin heuristic [7], simulated annealing [8], and mean field annealing [12]. We examine mean field annealing in more detail since it draws the best characteristics of the other two algorithms.

This paper contrasts two variations of Distributed Static Parallel Arc Consistency algorithm #2 which differ only by the frequency of communications. This paper also considers the effect of simple load balancing versus the more robust mean field annealing graph partitioning heuristic on speedup. Like many parallel algorithms, parameters such as distribution of work, frequency of message passing, and load balancing greatly affect the algorithm's potential speedup. Through actual machine experimentation, we measure time required for the different message frequencies and work assignments. Results indicate that the less frequent communications and the robust load balancing yield the best speedup.

Consistency Algorithms

To describe consistency algorithms, we refer to the network model of a CSP. An example is shown in Figure 1. We also rely on the following definitions:

Definition 1: A *variable*, i, is a unit which is assigned a meaning, or value. This represents a node in our constraint network graph.

Definition 2: 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).

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

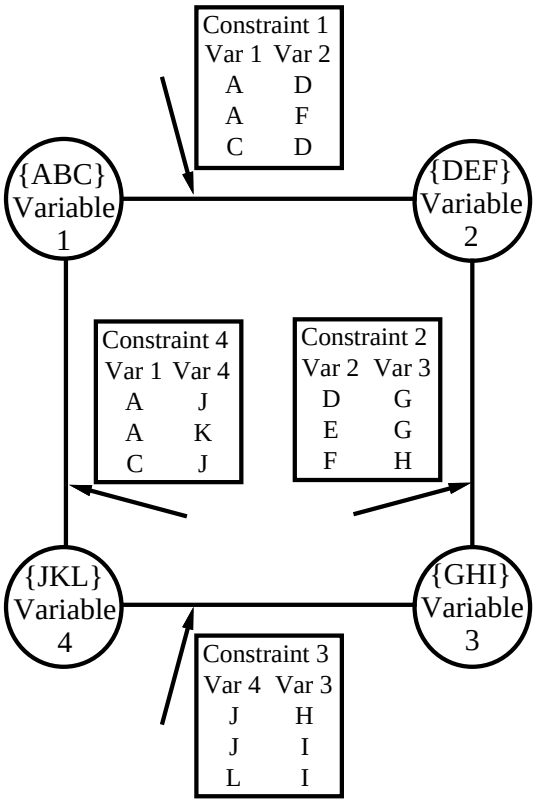

Figure 1: A Four-Variable, Eight-Arc, Three-Domain Value Constraint Network

Definition 3: 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 4: A *binary constraint*, R^2_{ij}, 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^2_{ij} is true if the relation is satisfied, that is $(l_{i_x}, l_{j_y}) \in R^2_{ij}$.

Definition 5: The constraint between variables i and j represents an edge in our constraint network graph, and each edge consists of two directed arcs: $arc(i,j)$ and $arc(j,i)$.

Definition 6: An arc $arc(i,j)$ is *arc consistent* if, for each value in the domain D_i for variable 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 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.

Definition 7: A *consistency check* is a comparison of a possible value of one variable with the possible value of another variable. We use consistency checks to analyze algorithm performance.

Popular sequential arc consistency (AC) algorithms include Mackworth's AC-1 and AC-3 [10].

AC-1 is a simple but inefficient sequential algorithm to ensure arc consistency. AC-1 checks every arc of the constraint network graph. If any value was removed from any domain, AC-1 rechecks all of its arcs again. AC-1 ends when it checks all of its arcs and does not remove any values.

To avoid the shortcomings of AC-1, AC-3 builds a queue containing all of the arcs of the constraint network to the queue. While an arc is checked, if a value is removed from a domain, AC-3 rechecks only the arcs which are attached to that variable (except the current arc) by adding the arcs to the queue.

Samal [11] introduces several parallel versions of arc consistency algorithms for a shared memory environment with an infinite number of processors.

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 the network arcs are checked simultaneously.

The parallel version of AC-3 is PAC-3. PAC-3 builds a globally available queue like AC-3. Again, only arcs in the queue are checked, and so we can check all arcs simultaneously. Generally, the PAC-3 queue is much smaller than the set of all edges, while in the worst case PAC-3 behaves just like PAC-1. A drawback of PAC-3 is that the time to build the queue is not negligible and may affect the total computational time.

Distributed Static Parallel Arc Consistency

We have developed three Distributed Static Parallel Arc Consistency Algorithms, DSPAC-1, DSPAC-2, and DSPAC-3, which are based on Mackworth's sequential arc consistency algorithms [3]. Unlike previous approaches, our algorithms uniquely assign large parts of a CSP to specific processors. This unique characteristic of our algorithms defines a relationship between constraint variables and parallel processor architectures.

This paper addresses only one of these algorithms, the Distributed Static Parallel Arc Consistency Algorithm #2 (DSPAC-2). We observed that, although DSPAC-2 performed more consistency checks than DSPAC-3, it finished its computation faster. This is most likely due to the additional time needed for

queue maintenance by DSPAC-3. DSPAC-2 also requires less work and took less time than DSPAC-1.

In DSPAC-2, we statically 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. We examine this load balancing in the next section.

Since we can not share variables, we designed our algorithms to best exploit the nature of a distributed memory architecture as follows:

- Read-only variables are stored on each processor. Only those variables needed by each processor are stored.
- Variables which are read by all processors, but written by one processor are stored on each processor. Each processor starts with the same variable values, but these are changed once the processing commences. If a processor changes one of these variables, it sends a message with the changes to all those processors which use that variable. In fact, the difference between the two algorithms below is the frequency that these variables are updated.
- Each processor has one variable which is written by all processors, but read only by itself. This variable is easily maintained by receiving messages from other processors with instructions to change the variable. Since this is a logical variable, we only need to perform **OR** or **AND** operations.
- One variable is read and written by all processors and indicates when all processors are done. It is maintained on the host processor. Messages from the node processors to the host processor instruct the host to increment or decrement the variable's value. When the variable indicates that all processors are done, the host gathers the results from node processors.

Our algorithms are examples of *asynchronous parallel algorithms* [9]. These algorithms allow some degree of asynchronism between various steps or successive iterations. The most important characteristic of asynchronous algorithms is that the processes never wait for inputs. Therefore, 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 and the distributed nature of our implementation that we can achieve a consistently high degree of speedup.

---

boolean function Sprevise($arc(i,j)$)
begin
  for each $l_{i_x} \in D_i$ and ($\neg support$)
  begin
    $support$:= FALSE;
    for each $l_{j_y} \in D_j$ while ($\neg support$)
      if $R^2_{ij}(l_{i_x}, l_{j_y})$ then $support$:= TRUE;
    if ($\neg support$) then
    begin
      $D_i := D_i - \{l_{i_x}\}$;
      if $D_i = \phi$ then $no_solution$:= TRUE;
      $local_change$:= TRUE;
    end
  end
  return($local_change$);
end

Figure 2: Static Parallel Revise Function: Sprevise

Static Parallel Revise Function

Our Distributed Static Parallel Arc Consistency Algorithms utilize a procedure called Static Parallel Revise Boolean function (Sprevise). Sprevise (Figure 2) checks an arc and eliminates all variable domain values which could never participate in a solution. If a value is removed, it informs the calling DSPAC algorithm of a change. 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 sends a *no_solution* message to all processors, and the DSPAC algorithm stops.

Distributed Static Parallel Arc Consistency Algorithm #2

An AC-1-based algorithm rechecks each of its arcs if one value in the network has been removed from a variable's domain. To better exploit the distributed nature of AC-1 and reduce the amount of unnecessary processing, we have developed another parallel arc consistency algorithm, DSPAC-2.

The DSPAC-2 algorithms (Figures 3 and 4) have 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 DSPAC-2 algorithms starts the node component of the algorithms. Each node processor has an interrupt-driven receive procedure which reads domain change messages, updates its own memory, and updates its own *change* variable.

Each processor has its own *change* bit. If processor x is assigned variable i and DSPAC-2 running on pro-

```
procedure DSPAC-2-HOST()
begin
  active_procs := 0; no_solution := FALSE;
  assign variables to processors;
  || for each processing node p, DSPAC-2-NODE(p);
  wait for active_procs = 0;
  gather results;
end

procedure DSPAC-2-NODE(p)
begin
  change[p] := TRUE;
  while((change[p]∨
    active_procs > 0) ∧ ¬no_solution)
  begin
    if change[p] send(active_procs + +) to host;
    while(change[p] ∧ ¬no_solution) do
    begin
      change[p] := FALSE;
      for each assigned variable i do
        for each directed arc(i, j) do
        begin
          if (Sprevise(arc(i, j)))
            for each arc(i, j), where j is on processor m
              send(D_i) to m;
        end
    end
    if done processing send(active_procs − −) to host;
  end
end
```

Figure 3: DSPAC-2 Algorithm: Host Processor and Node Processor Components for Arc-Based Message Frequency.

```
procedure DSPAC-2-HOST()
begin
  active_procs := 0; no_solution := FALSE;
  assign variables to processors;
  || for each processing node p, DSPAC-2-NODE(p);
  wait for active_procs = 0;
  gather results;
end

procedure DSPAC-2-NODE(p)
begin
  change[p] := TRUE;
  while((change[p]∨
    active_procs > 0) ∧ ¬no_solution)
  begin
    if change[p] send(active_procs + +) to host;
    while(change[p] ∧ ¬no_solution) do
    begin
      change[p] := FALSE;
      for each assigned variable i do
      begin
        var_chg := FALSE;
        for each directed arc(i, j)
          var_chg := var_chg∨ (Sprevise(arc(i, j)));
        if var_chg
          for each arc(i, j), where j is on processor m
            send(D_i) to m;
      end
    end
    if done processing send(active_procs − −) to host;
  end
end
```

Figure 4: DSPAC-2 Algorithm: Host Processor and Node Processor Components for Variable-Based Message Frequency.

cessor x removes a value from the domain of D_i, processor x sends messages to set to "on" only the *change* bit of all processors y, where $arc(i,j) \in E$ and variable 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.

The versatility of these parallel algorithms lie in the fact that domain changes are available to all processors almost immediately. Each node processor has an interrupt-driven receive procedure which reads domain change messages, updates its own memory, and updates its own *change* variable. Therefore, values which are removed are never checked again by the Sprevise function.

The counter *active_procs* is maintained by the host to signify the number of processors starting and stopping local arc consistency. When a processor starts (stops) its local arc consistency, it sends a message to increment (decrement) the counter. When the host finds the *active_procs* variable is zero, it instructs the node processors to send the results.

With the DSPAC-2 algorithms, 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 AC-1-based algorithms. In the DSPAC-2 algorithms, all processors 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 supplied new variable domain data.

The main difference between the two algorithms shown in Figures 3 and 4 is the frequency of communications between processors. Messages can be sent

after each arc is checked (Figure 3), or after all of the arcs associated with a variable are checked (Figure 4). Here we see a trade-off between work performed and frequency of communications. If messages are sent more frequently, additional time must be devoted sending and receiving the messages, but less time is spent performing consistency checks. We measure the execution time of these two algorithms to determine the best approach.

Complexity analysis of the DSPAC algorithms is addressed in [3].

Data Partitioning and Load Balancing

An algorithm which dynamically assigns tasks would best balance the workload of a parallel computer, but the task assignment and recombination overheads may overtake any benefits of load balancing. 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 variable domain. The variable, therefore, has to be shared between the processors. To ensure data coherency, these two processors need frequent communications and execution time increases.

If one assign variable domains to processors, then only one processor updates a particular variable domain. This allows one to exploit the asynchronous behavior of parallel arc consistency.