```
procedure AC-1()
begin
    NC();
    Q ← Q ∪ {arc(i,j)|arc(i,j) ∈ G, i ≠ j}
    repeat
        change := FALSE;
        for each arc(i,j) ∈ Q do
            change := Revise(arc(i,j)) ∨ change;
    until (¬change);
end
```

Figure 2.2: Mackworth's Arc Consistency Algorithm: AC-1

$\mathbf{O}(a^3ne)$:

- AC-1 checks each arc of the graph, $2e$, by calling Revise.
- The complexity of Revise is a^2.
- If only one value is removed from each variable each loop, na loops are executed.

AC-3 (Figure 2.3) is an improvement of the AC-1 algorithm. AC-1's inefficiencies lie in the fact that, when only one value is removed, all arcs 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 $arc(i,j)$:

$$Q \leftarrow Q \cup \{arc(m,i)|arc(m,i) \in G, m \neq i, m \neq j\} \tag{2.4}$$

If we look at the example network in Figure 1.5, when AC-3 examines $arc(1,2)$, Revise removes the value B from the domain D_1. AC-3 would then add $arc(4,1)$ to the queue

```
procedure AC-3()
begin
    NC();
    Q ← Q ∪ {arc(i,j)|arc(i,j) ∈ G, i ≠ j}
    repeat
        for arc(i,j) ∈ Q do
        begin
            Q := Q − {arc(i,j)};
            if (Revise(arc(i,j))) then
                for each arc(m,i)
                    Q := Q ∪ {arc(m,i)|arc(m,i) ∈ G, m ≠ i, m ≠ j};
        end
    until (Q = φ);
end
```

Figure 2.3: Mackworth's Arc Consistency Algorithm: AC-3

(if it is not on the queue already). AC-3 does not add $arc(2,1)$ to the queue because B is never used to support a value in D_2.

AC-3 terminates when the queue is empty. Mackworth and Freuder [46] show 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 by $\Omega(a^2e)$ and $\mathbf{O}(a^3e)$.

We should note, however, that although AC-3 performs fewer consistency checks (in the worst and average cases) than AC-1, a computer running AC-3 spends additional time building and maintaining the queue of arcs for processing.

2.3 Parallel Arc Consistency Algorithms

Samal [55] presents several parallel versions of sequential arc consistency algorithms. He assumes his model computer has an infinite number of processors available, and that the computer implements a shared memory scheme.

```
boolean function Previse(arc(i,j))
begin
    change := FALSE;
    ||for each l_{i_x} := 1 to a do
    begin
        support[l_{i_x}] := FALSE;
        ||for each l_{j_y} := 1 to a do
            if R^2_{ij}(l_{i_x}, l_{j_y}) then support[l_{i_x}] :=TRUE;
        if (¬support[l_{i_x}] ) then
        begin
            D_i := D_i − {l_{i_x}};
            change := TRUE;
        end
    end
    return(change);
end
```

Figure 2.4: Samal's Parallel Arc Consistency Algorithm: Previse

Samal's algorithms use a parallel version of Revise (Previse), shown in Figure 2.4. In Previse, all consistency checks of a single arc are executed concurrently. If the domain size of each variable in an arc is up to a, then up to a^2 checks are made in one time unit. Therefore, up to a^2 processors are needed, and time complexity is $\mathbf{O}(1)$. Also, computational complexity 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 $\mathbf{O}(na)$, while the computational complexity is $\mathbf{O}(a^3ne)$.

```
procedure PAC-1()
begin
    PNC();
    change := TRUE;
    while (change) do
    begin
        change := FALSE;
        || for each arc(i,j) do
            change := Previse(arc(i,j)) ∨ change;
    end
end
```

Figure 2.5: Samal's Parallel Arc Consistency Algorithm: PAC-1

```
procedure PAC-3()
begin
    PNC();
    Q := E;
    while (Q ≠ φ) do
    begin
        || for each arc(i,j) ∈ Q do
        begin
            lchange := Previse(arc(i,j));
            if (lchange) then
            begin
                change := TRUE;
                add_arcs(arc(i,j));
            end
        end
    end
end
```

Figure 2.6: Samal's Parallel Arc Consistency Algorithm: PAC-3

The parallel version of AC-3 is PAC-3 (Figure 2.6). PAC-3 builds a globally available queue like AC-3, and uses Previse like PAC-1. PAC-3 executes a special procedure, add_arcs, to add $arc(i, j)$ to the global queue. Again, only arcs in the queue are checked. If each arc is checked concurrently, then, in the worst case, all $2e$ arcs are checked. 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 negligible and may affect the total computational time.

Chapter 3

Static Parallel Arc Consistency Algorithms

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 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 enable us to scale a problem simply by adding more processors. Each processor of our target multiprocessor ensures local arc consistency for its assigned part of the network.

3.1 Distribution of Work

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 [6, 51, 52]. 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.

If one assigns a variable domain and outgoing arcs to a processor, then only that processor writes to the domain variable. In fact, since only one processor is writing to the variable, the variable does not need to be locked. This allows one to exploit the asynchronous behavior of parallel arc consistency algorithms. As example of this assignment scheme, if variable i is assigned to processor m, processor m works with values l_{i_x} of domain D_i and all constraints represented by arcs $arc(i,j)$. Since several variables can be assigned to one processor, variable j may be assigned to processor m or any other processor.

Although one may want to distribute the work optimally on the processors, optimization is itself NP complete. Kernighan and Lin [35] present a heuristic which partitions weighted nodes and weighted edges of a graph into groups of equal cost. They show that, depending on the number of nodes to process, the heuristic may actually partition optimally. Their algorithm works by making an arbitrary but even partition of nodes of a graph. The algorithm then exchanges nodes until a minimum cutset is found.

Another graph partitioning algorithm researchers employ is simulated annealing [37]. Simulated annealing involves moving nodes to other partitions if the overall edge cutting and balance cost is lowered. If a proposed move does not reflect a cost reduction, the move is made or not made based on a decreasing random probability. Johnson, et.al. [32] show that, although the Kernighan-Lin heuristic does not produce

an optimal partition as often as simulated annealing, it can obtain its near-optimal result in 1/100 of the time. They say that the Kernighan-Lin heuristic is still the standard graph partitioning algorithm that researchers use to compare their own algorithms.

In our implementation of the Kernighan-Lin heuristic, the variables of the constraint network represent the nodes of the graph. The number of arcs originating from the constraint variable times the size of the domain represents the weight of a node. The arcs of the network represent the edges of the graph. The number of tuples of the cross product between variable domains of a constraint represents the weight of the edge.

Researchers have developed algorithms for graph partitioning based on the Kernighan-Lin algorithm for shared memory machines [53] and distributed memory machines [21]. Our implementation, however, runs on a sequential processor. It examines our constraint network and determines an assignment such that:

- The amount of anticipated work in each processor is approximately equal.
- The clustering of the constraint variables yields an approximate minimum cost of communication between processors based on the number and weight of edges between clusters.

3.2 Parallel Node Consistency

Each of our parallel arc consistency algorithms ensures node consistency. Each processor checks for node consistency for the part of the network assigned to it. This work is denoted by PNC() in the arc consistency algorithms. As stated earlier, we do not analyze node consistency.

```
boolean function Sprevise(arc(i,j))
begin
    for each l_ix ∈ D_i and (¬support)
    begin
        support := FALSE;
        for each l_jy ∈ D_j while (¬support)
            if R^2_ij(l_ix, l_jy) then support := TRUE;
        if (¬support) then
        begin
            D_i := D_i − {l_ix};
            if D_i = φ then no_solution := TRUE;
            local_change := TRUE;
        end
    end
    return(local_change);
end
```

Figure 3.1: Static Parallel Revise Function: Sprevise

3.3 Static Parallel Revise Function

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 (Figure 3.1) 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 make sure 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.

3.4 Static Parallel Arc Consistency Algorithm #1

The SPAC-1 algorithm (Figure 3.2) is split into host and node components. 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 algorithm 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 verify local arc consistency. Once each processor verifies local arc consistency of the variables and arcs it is assigned, it waits for the host to check consistency of the entire network. If any processor sets the *change* bit, each processor again ensures local arc consistency by checking all of its arcs.

The power of this parallel algorithm lies in the fact that any 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 complete checking their assigned arcs.

Theorem 3.1 *The time complexity of SPAC-1 is* $\mathbf{O}(a^3ne/p)$.

Proof: 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

```
procedure SPAC-1-HOST()
begin
    PNC();
    assign variables to processors;
    change := TRUE; no_solution := FALSE
    while(change ∧ ¬no_solution) do
    begin
        change := FALSE;
        ||for processing node p, change = change∨ SPAC-1-NODE(p);
    end
end
procedure SPAC-1-NODE(p)
begin
    for each assigned variable i do
        for each directed arc(i, j) do
            local_change = local_change∨ Sprevise(arc(i, j));
    return(local_change);
end
```

Figure 3.2: Static Parallel Arc Consistency Algorithm #1: Host Processor and Node Processor Components

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. Therefore, the time complexity is $\mathbf{O}(a^3ne/p)$. □

Theorem 3.2 *The computational complexity of SPAC-1 is* $\mathbf{O}(a^3ne)$.

Proof: The worst case time complexity is $\mathbf{O}(a^3ne/p)$, and 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 $\mathbf{O}(a^3ne)$. □

These complexities represent the worst case. On the average (as we show with the experimental results in Section 6), the actual time and work performed is substantially smaller.

3.5 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 is 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 3.3) is 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 the SPAC-1 algorithm with two exceptions. First, each processor has its own *change* bit. If processor x is assigned variable i and SPAC-2 running on processor x removes a value from the domain of D_i, processor x sets only the *change* bit of all processors y where $arc(i,j) \in G$, and variable j is 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 (Figure 3.4) 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* global variable.

Each SPAC-2 processor uses this variable counter 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 the node processors running SPAC-2-NODE observe that the *active_procs* variable is zero, it ends execution of SPAC-2-NODE.

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

```
procedure SPAC-2-HOST()
begin
    PNC();
    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] ∨ active_procs > 0) ∧ ¬no_solution)
    begin
        while(change[p] ∧ ¬no_solution) do
        begin
            active_procs + +;
            while(change[p] ∧ ¬no_solution) do
            begin
                change[p] := FALSE;
                for each assigned variable i do
                    for each directed arc(i, j) do
                        Sprevise-2(arc(i, j));
            end
            active_procs − −;
        end
    end
end
```

Figure 3.3: Static Parallel Arc Consistency Algorithm #2: Host Processor and Node Processor Components

```
boolean function Sprevise-2(arc(i,j))
begin
    for each l_ix ∈ D_i and (¬support)
    begin
        support := FALSE;
        for each l_jy ∈ D_j while (¬support)
            if R^2_ij(l_ix, l_jy) then support := TRUE;
        if (¬support) then
        begin
            D_i := D_i − {l_ix};
            if D_i = φ then no_solution := TRUE;
            for each arc(i,m), where m is on processor p
                change[p] := TRUE;
        end
    end
end
```

Figure 3.4: Static Parallel Revise Function: Sprevise-2

start useful processing when they obtain new variable domain data.

Theorem 3.3 *The time complexity of SPAC-2 is* $\mathbf{O}(a^3ne/p)$ *and the computational complexity of SPAC-2 is* $\mathbf{O}(a^3ne)$.

Proof: 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 notice a slight improvement in speed over SPAC-1 in the average case because SPAC-2 in not synchronized and updates *change* bits frequently.

3.6 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 is 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 3.5.

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 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 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 and, therefore, only one queue is available for all processors to use. We do not assume a shared memory model, so we do not maintain one queue. Each processor maintains its own queue of arcs to check for consistency. The queues hold only those arcs which are 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 is deleted from a variable's domain, the queue of arcs associated with **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 exe-

```
procedure SPAC-3-HOST()
begin
    PNC();
    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 i
        Q[p] ← Q[p] ∪ {arc(i,j)|arc(i,j) ∈ G, i ≠ j}
    while(((Q[p] ≠ φ) ∨ active_procs > 0) ∧ ¬no_solution)
    begin
        while((Q[p] ≠ φ) ∧ ¬no_solution) do
        begin
            active_procs + +;
            while((Q[p] ≠ φ) ∧ ¬no_solution) do
            begin
                for arc(i,j) ∈ 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] ∪ {arc(m,i)|arc(m,i) ∈ E, m ≠ i, m ≠ j};
                end
            end
            active_procs − −;
        end
    end
end
```

Figure 3.5: Static Parallel Arc Consistency Algorithm #3: Host Processor and Node Processor Components

cutes on 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.

Theorem 3.4 *The time complexity of SPAC-3 is* $\mathbf{O}(a^3e/p)$.

Proof: 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 dynamically grows and shrinks during execution. A possibility exists that only one value is removed from each successful call to Sprevise on any of the processors. Mackworth and Freuder [46] 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 $\mathbf{O}(a^3e/p)$. □

Theorem 3.5 *The computational complexity of SPAC-3 is* $\mathbf{O}(a^3e)$.

Proof: If the worst case time complexity occurs, 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 $\mathbf{O}(a^3e)$. □

These complexities represent the worst case. On the average (as we show with the experimental results in Chapter 6), the actual work performed is substantially smaller.

Table 3.1 shows a summary and comparison of the sequential and parallel arc consistency algorithms we have discussed.