International Journal of Parallel Programming, Vol. 16, No. 5, 1987

Parallel Consistent Labeling Algorithms[1]

Ashok Samal[2] and Tom Henderson

Received April 1987; revised April 1988

Mackworth and Freuder have analyzed the time complexity of several constraint satisfaction algorithms.[1] Mohr and Henderson have given new algorithms, AC-4 and PC-3, for arc and path consistency, respectively, and have shown that the arc consistency algorithm is optimal in time complexity and of the same order space complexity as the earlier algorithms.[2] In this paper, we give parallel algorithms for solving node and arc consistency. We show that any parallel algorithm for enforcing are consistency in the worst case must have $\mathbf{O}(na)$ sequential steps, where n is number of nodes, and a is the number of labels per node. We give several parallel algorithms to do arc consistency. It is also shown that they all have optimal time complexity. The results of running the parallel algorithms on a BBN Butterfly multiprocessor are also presented.

KEY WORDS: Arc consistency; parallel algorithms; constraint satisfaction; inherent parallelism.

1. INTRODUCTION

We define a constraint satisfaction problem (after Mackworth[3]) as follows:

- $N = \{i, j, ...\}$ is the set of nodes, with $|N| = n$,
- $A = \{l_1, l_2, ...\}$ is the set of labels, with $|A| = a$,
- $E = \{(i, j) | (i, j)$ is an edge in $N \times N\}$, with $|E| = e$,

[1] This work was partially supported by NSF Grants MCS-8221750, DCR-8506393, and DMC-8502115.

[2] Department of Computer Science, University of Utah, Salt Lake City, Utah 84112. ARPANET: samal@cs.utah.edu.

0885-7458/87/1000-0341$05.00/0

- $A_i = \{l \mid l \in A$ and (i, l) is admissible$\}$,
- R_i^1 is a unary predicate, and (i, l) is admissible iff $R_i^1(l)$,
- R_{ij}^2 is a binary predicate, and $(i, l) - (j, l')$ is admissible iff $R_{ij}^2(l, l')$.

The constraint satisfaction problem is to find all n-tuples in A^n which satisfy the two sets of predicates: R^1 and R^2. Note that A is the set of all possible labels and A_i is the set of possible labels for the i^{th} node. A_i may either be a proper subset of A or identical to A. We should also note at the outset that since predicates and relations are equivalent, they are used somewhat interchangeably. It will be obvious from the context, however, whether we mean a relation or a predicate.

An algorithm for achieving node consistency checks only the unary relations on the different nodes, and keeps in the domain of each node, values satisfying this unary constraint. Arc consistency algorithms check the consistency of labels for each pair of nodes linked by a binary constraint and remove the labels that cannot satisfy this local condition.

Path consistency algorithms ensure that any pair of labelings $(i, l) - (j, l')$ allowed by a direct relation is also allowed by all paths from i to j. It has been proved in Ref. 4 that for complete graphs, path consistency is equivalent to consistency of every path of length 2; therefore, this is equivalent to checking the consistency of every triple. Each graph can always be replaced by an equivalent complete graph by adding the *TRUE* constraint between the nodes which are not connected.

Several authors have presented algorithms to solve the consistency labeling problem. However, since the problem is NP-complete,[5] it has been suggested by others that a preprocessing or filtering step be applied before the backtracking or search procedures.[3,4,6–8] Although node, arc, and path consistency algorithms do not usually result in a solution, they do eliminate any labels failing to satisfy a minimum of consistency constraints. Such techniques have found wide application in artificial intelligence, pattern recognition, and image analysis.

Mackworth[3] has given several algorithms to enforce arc and path consistency in a network of constraints. Later Mohr and Henderson[2] gave algorithms for arc and path consistency, AC-4 and PC-3, which are bounded above by $\mathbf{O}(ea^2)$ and $\mathbf{O}(n^3a^3)$, respectively (worst case complexity). Moreover, the space requirements, although not negligible, are of the same order as Mackworth's algorithms.

In this paper, we study parallel algorithms to achieve node and arc consistency. To make the paper self-contained we give a brief summary of the sequential node and arc consistency algorithms in Section 2. The abstract machine model that is used to derive the theoretical results is described in Section 3. Section 4 contains the parallel algorithms PAC-1, PAC-3, and PAC-4 to achieve arc consistency. First however the lower bound for any algorithm doing arc consistency in parallel is derived. The algorithms are analyzed for their parallel complexity, and it is shown that they are optimal parallel algorithms. All the parallel algorithms are implemented on a BBN Butterfly multiprocessor and its main features are described in Section 5. The results are presented in Section 6. We conclude in Section 7 with a brief summary and possible future work.

2. ALGORITHMS TO ACHIEVE ARC CONSISTENCY

Several algorithms to achieve arc consistency in a network of constraints have been proposed. Mackworth[3] gave three algorithms: AC-1, AC-2, and AC-3. Later Mohr and Henderson[2] gave another algorithm, AC-4, which is an optimal sequential algorithm for arc consistency. Since we have used these algorithms to study parallelism in the arc consistency algorithms, they are briefly summarized in this section. The details of the algorithms can be found in the relevant papers.

2.1. Node Consistency

The first step in enforcing arc consistency is to make sure that none of the nodes violates any unary constraint at that node. The algorithm to enforce consistency at each node is rather straightforward. For each label, l, at each node, i, we check if (i, l) is admissible, i.e., $R_i^1(l)$ is *TRUE*. If it is not, the label l is removed from the label set of the node i. The complexity of the algorithm is $\mathbf{O}(na)$.

2.2. The Revise Function

All the arc consistency algorithms given in Ref. 3 use a common function called **Revise**. It tests for consistency along a single arc, (i, j), and removes the labels at node, i, which do not have any support from the labels at node, j. The function returns *TRUE* if any label is deleted from the label set of any of the nodes and returns *FALSE* otherwise. For each label l in the label set of node i, we check to see if there is a label l' in the label set of node j such that the binary constraint holds; i.e., $R_{ij}^2(l, l')$. In the worst case we have to check all a labels of node j for each label of node i. So, the complexity of the **Revise** procedure is $\mathbf{O}(a^2)$.

An arc (i, j) can be made consistent by calling the function **Revise.** However as a side effect one or more arcs in the graph may now become inconsistent. Therefore, this function has to be called on all the arcs. Also,

it is easy to see that a single pass through the set of arcs is not sufficient to guarantee arc consistency in the graph. An iterative solution is necessary to achieve global consistency.

2.3. The AC-1 Algorithm

AC-1 is a simple, but brute force algorithm to enforce arc consistency. To start with, the **Revise** function is called for each arc in the graph. If any label at any node is deleted, then the process is repeated. The algorithm terminates when there is no change in the label set of any of the nodes. In the worst case AC-1 will need na iterations to finish. In each iteration there are e calls to **Revise** which is an $\mathbf{O}(a^2)$ procedure. So the complexity of AC-1 is $\mathbf{O}(ena^3)$.

2.4. The AC-3 Algorithm

In AC-1 all the arcs are checked for consistency during each iteration. This is very inefficient since, in general, only a (small) subset of the arcs is affected by the changes made in the previous iteration. This observation is used in the AC-2 and AC-3 algorithms to reduce the amount of work in each iteration. AC-2 is not discussed here since it is a special case of AC-3.

In AC-3 only the arcs associated with the nodes whose label sets have changed during the previous iteration are added to the set of arcs to be checked for consistency. During each iteration the **Revise** procedure is called on only these arcs. If **Revise**(i, j) returns *TRUE*, which means that at least one label at node i was deleted, we add the arcs $\{(k, i) \mid (k, i) \in E, k \neq i, k \neq j\}$ to the set of arcs to be checked for consistency in the next iteration. These are the arcs which may now have become inconsistent because of the deletion of any label(s) at node i. This is done for all the arcs in the set. If there is any change in the label set of any node, this process is repeated. Like AC-1 the algorithm stops when there is no change in the previous iteration. It can easily be shown[1] that AC-3 is $\mathbf{O}(ea^3)$.

2.5. The AC-4 Algorithm

The AC-4 algorithm is based on the notion of support. As long as a label l at a node i has some support from one or more labels at all other nodes, l is considered a viable label for node i. This information about the support for the labels is stored explicitly in two data structures: *Counter* and *Support*.

The number of supports for label l at node i from node j is stored in $Counter[(i, j), l]$. The detailed support information is stored in an array of

lists, *Support*. Any label at a node supports some labels at other nodes. This list of node-label pairs are stored in *Support*. $Support_{jl'}$ contains all the node-label pairs, (i, l) such that label l' at node j supports label l at node i.

The algorithm works in two stages. First the data structures (*Counter* and *Support*) are constructed. This is done by iterating over all the possible labels for the nodes that constitute the arcs. Also, after the support sets are built, a list (called *List*) of all the node-label pairs which have no support is constructed. It is easy to prove that this procedure takes $\mathbf{O}(ea^2)$ time.

In the second stage all the labels which have no support are deleted. Initially *List* contains this information. A node-label pair, (j, l'), is pulled out from *List* and the number of supports (stored in *Counter*) for the node-label pairs which are supported by it is decremented by one. If this makes the number of supports for that node-label pair to become zero, then the label is deleted and this node-label pair is added to *List*. This process continues until *List* is empty. This procedure also takes $\mathbf{O}(ea^2)$ time.

3. ABSTRACT MACHINE MODEL

Before we discuss the parallel arc consistency algorithms, we describe the absract machine model used to obtain the results in the next section. We also discuss the notation used for the rest of the paper. All the measures that are used to compare the performance of the parallel algorithms are also defined.

In the case of machines with single processors, the computation model as defined by von Neumann is taken to be the standard. However, several models have been proposed to study the computation properties of multiprocessors (see Ref. 9 for a survey). The model that is used for analysis of the algorithms in this section is a variation of the Parallel Random Access Machine (*PRAM*)[10] model.

A PRAM has an infinite number of processors, an unbounded amount of global memory, a set of input registers, and a finite program. The processors are capable of working both synchronously and asynchronously. Each processor resembles a von Neumann machine and has a local accumulator, a memory bank, and a program counter. However, the memory is unbounded. The global memory is shared by all the processors and any processor can read/write from/to any memory location at any time during execution. Both concurrent reads and concurrent writes are allowed in this model. However, if several processors try to write into the same location simultaneously, any one (nondeterminate) of them will succeed.

The global memory is the only means of communication between the processors. This essentially sidesteps a communication network and the

issues involving it. The instruction set of the machine consists of usual operations like, LOAD, ADD, JUMP, etc. In addition a set of new instructions are added to incorporate the multiprocessing capabilities, e.g., fork processes, synchronization primitives, etc. The time to create a process (task) on a processor is assumed to be negligible.

Clearly the PRAM model can't be realized in hardware. However, there are several multiprocessors which come close to it; for example, the class of shared memory multiprocessors (e.g., the BBN Butterfly parallel processor[11]).

3.1. Notation

The notation used throughout the paper is briefly described here. The time taken by the algorithm to run on a single processor is denoted by T_1. T_∞ is the time taken when infinitely many processors are available. It is also called the *unbounded parallel complexity*. Time taken under bounded parallelism (with a finite number (k) of processors) is denoted by T_k. S_k is used for speedup obtained for the sequential algorithm using k processors. In general α is used to denote the time taken to execute a unit-computation. For each arc, x, *x.start* and *x.end* denote the start and end nodes of the arc, respectively.

Several new constructs are used in the algorithms to describe the behavior in a multiprocessor framework. We use a variation of the standard *for* construct, called ‖ *for*, to describe processes running concurrently. The statement

$$\| \textit{for } i := 1 \text{ to } m \text{ do } \mathbf{f}(i);$$

creates m tasks which execute concurrently (at least in theory), the i^{th} task computing $\mathbf{f}(i)$. The statement after ‖ *for* is executed only after all the m tasks are completed. We do not use Dijkstra's[12] primitives, **P** and **V** for synchronization. Instead we use a simple operation available on the BBN Butterfly called *Atomic_add*. The operation *Atomic_add*(x, y) adds y to x atomically. If several processes are trying to add to the same variable x they will be serialized in some arbitrary order. This is not as general as **P** and **V**, but is sufficient for our application.

3.2. Performance Measurements

In case of sequential algorithms, the main two concerns are their time and space complexities. These are very important for parallel algorithms as well. However, these two figures alone do not portray a parallel algorithm either accurately or completely.

For example, they don't indicate how much faster the algorithm runs as compared to a sequential algorithm. Nor do they show how many processors are needed to achieve this. Several measures have been proposed to quantify many of these properties of parallel algorithms. We will define only a few of these and describe them briefly in this section. See Refs. 13 and 14 for more of these measures.

3.2.1. Speedup

This is the most commonly used and probably the most important characteristic of parallel algorithms. The most common definition of *speedup* is the ratio of the time taken for a given algorithm to run on a uniprocessor and the time taken by it on a multiprocessor.

$$S_k(A) = \frac{\text{Time taken by an algorithm } A \text{ on 1 processor } (T_1)}{\text{Time taken by the same algorithm } A \text{ on } k \text{ processors } (T_k)}$$

It is a very good indicator of the success of a multiprocessor implementation. There are several problems with this definition however. First, it can be misleading. A high speedup does not necessarily mean a better algorithm. It is important to realize that this speedup number is only for the particular algorithm at hand. So it is possible that an algorithm A has a higher speedup than an algorithm B, but actually runs slower on a multiprocessor. This can happen if the sequential algorithm for A is very slow to start with. Secondly, by transforming a sequential algorithm to run on a multiprocessor, we necessarily change its structure. Hence, it is not the same algorithm any more. So, it is not entirely accurate to say that algorithm B is the parallel version of A. In fact, two sequential algorithms might be similar, but their corresponding parallel algorithms may have very different structures.

3.2.2. Efficiency

Efficiency is defined as the ratio of the speedup obtained to the total number of processors used by the algorithm.

$$\eta = \frac{S_k}{k}$$

This is important because it indicates how effectively the processors are used. Higher efficiency means better utilization and vice versa.

3.2.3. Inherent Parallelism

Both speedup and efficiency measure how the algorithm performs in an actual multiprocessor configuration. They are important yardsticks

since they are obtained by actually running the algorithms. But these numbers don't indicate a very important property of algorithms. It is what we call *inherent parallelism.* Intuitively this means the amount of parallelism that is intrinsic to the algorithm. If this quantity for an algorithm is 1 (one) it means that it is in essence a very sequential algorithm and no speedup can be expected by running it on a multiprocessor. On the other hand, if it is large then it should mean that there is a large amount of parallelism in the algorithm which can be exploited by a multiprocessor.

This *inherent parallelism* (*IP*) is important for several reasons. First, it indicates how much speedup one can expect from an algorithm and indirectly, the expected utilization of the processors. This can be obtained theoretically without actually running the algorithm in a multiprocessor. Another important reason is that it helps in the design and performance prediction of a system where a set of algorithms with different characteristics have to be put together to achieve the task. We define inherent parallelism (*IP*) of an algorithm *A*, as follows:

$$IP(A) = \frac{\text{Serial complexity of } A}{\text{Unbounded parallel complexity of } A}$$

Unbounded parallel complexity means the time taken by the algorithm (in some abstract time unit), given an infinite number of processors; i.e., the algorithm can use as many processors as it wants. Here complexity is used to denote the worst case time complexity.

3.2.4. Maximum Effective Processors

Except for a few cases (combinatorial implosion,[15–17] speedup is less than the number of processors used. Ideally one would like the speedup to go up linearly (with slope 1) with the number of processors. However, depending on the problem size, at a certain point adding processors does not increase the speedup. We call this the maximum number of effective processors, and it is denoted by $P_{\max}$.

There are several other measures one could use to characterize parallel algorithms, e.g., synchronization costs, equivalence point, etc. We restrict ourselves to the ones previously described, in addition to the time and space complexities. Both *IP* and $P_{\max}$ are used to analyze the algorithms theoretically. To measure the performance of the algorithms on the Butterfly multiprocessor we use speedup, S_k and efficiency, η.

4. PARALLEL ALGORITHMS FOR ARC CONSISTENCY

In this section we give the details of the parallel algorithms for arc consistency. The order in which algorithms are analyzed is the same as in

Section 2. Each parallel algorithm is first described, and then its performance measures as defined in the last section are deduced. A lower bound for parallel arc consistency is also derived.

First, however, we put a restriction on the number of processors the PRAM model can have. It can be shown that if $n^a n^2$ processors are available the CLP can be solved in constant time. Essentially there are n^a possible solutions for the problem and they all can be checked in parallel. However, the number of processors required to achieve constant time is exponential and as the problem size increases it is not possible to have that large a number of processors. For example, for $n = 10$, and $a = 10$, which are in the low end of the problem size, the number of processors needed to achieve constant time is 10^{10}. Clearly this is not feasible. Also, the efficiency of the system is going to be extremely low. So, we restrict ourselves to algorithms which need only a polynomial number of processors, although a PRAM model allows us to have them. Also, if we had exponential number of processors we can solve CLP in constant time and hence there would be no need to enforce consistency in the network.

4.1. Parallel Node Consistency

The parallel algorithm to achieve node consistency is given in Fig. 1. The serial complexity of the node consistency algorithm is $\mathbf{O}(na)$. Essentially there are *na* independent tasks which may be done in parallel. If *na* processors are available, it is clear that node consistency can be achieved in constant time. There is no need for any kind of synchronization because all the tasks are independent. Thus $T_\infty = \mathbf{O}(1)$. If there are only k ($< na$) processors, then the *na* tasks can be divided among the k processors and hence the time taken in this case is $\mathbf{O}(na/k)$.

$$IP = T_1/T_\infty = \mathbf{O}(na); \qquad P_{\max} = na$$

```
procedure PNC()
begin
    ||for each unit := 1 to n do
        ||for each label := 1 to a do
            if ¬ R¹unit(label) then
                Aunit := Aunit − {label};
end
```

Fig. 1. Parallel node consistency algorithm: PNC.

4.2. Lower Bound for Parallel Arc Consistency

Theorem 1. Any parallel algorithm for enforcing arc consistency in the worst case must have $\mathbf{O}(na)$ sequential steps, where n is number of nodes, and a is the number of labels per node.

Proof. Consider a network of n nodes connected to form a cycle as shown in Fig. 2. Let the label set of node i be the set of numbers defined as: $\{x \mid x = jn + i, 1 \leqslant j \leqslant a\}$. The relation used here is the *greater-than* relation such that values at node $i+1 \pmod n$ should be greater than values at node i, $1 \leqslant i \leqslant n$. It is assumed that the network is node consistent before any algorithm is applied. (Dechter and Pearl[18] use a cyclic graph to prove the lower bound for sequential arc consistency. However, the label sets of the nodes are different and the constraint used is also different.)

In general the arcs of the graph may represent different types of constraints. Se we may not be able to use any special technique to enforce global arc consistency. The only way is to enforce the global arc consistency along each arc. If unlimited processors were available we could enforce consistency along all the arcs concurrently. In this problem instance, during the first time step only the label 1 of node 1 is removed, since it doesn't get any support from node n. But all the labels at all other nodes have some support and hence remain intact. During the second time step only label 2 of node 2 gets deleted since, it was being supported by 1 at node 1 which was just deleted in the last time step. So, during the first $(a-1)n$ time steps, $(a-1)$ labels are removed from each node and the

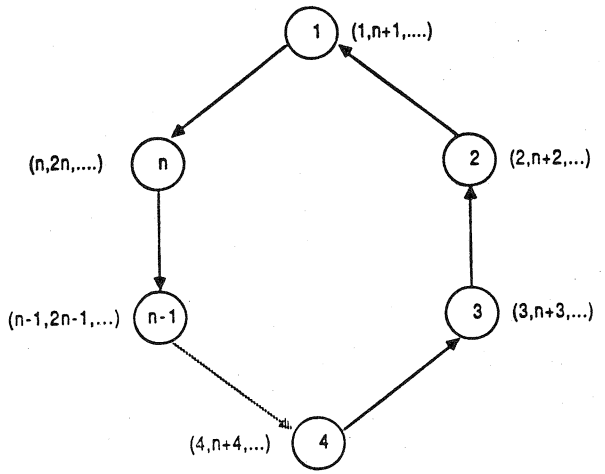

Fig. 2. A network of constraints.

order of removal of labels from nodes is cyclic; 1, 2,..., n, 1,..., n, and so on. Then the label set of node 1 becomes empty, since it has no support from node n. After that the label sets of 2, 3,..., n, all become empty in 1 time unit, since the empty label set for one unit implies that there can be no globally consistent labeling. Thus the number of steps needed before any algorithm can terminate is $(a-1)n+2$. So, parallel arc consistency is $\Omega(na)$. In the previous proof it is assumed that there are at least e processors and they are working simultaneously on the arcs of the graph. If there are more processors the time taken in the worst case doesn't change. The theorem also holds if there are fewer processors.

4.3. Parallel Revise

The **Revise** procedure is used by both AC-1 and AC-3 to enforce the consistency along a single arc, (i, j). It removes all labels at node i which don't have any support from node j. The algorithm (called **Previse**) to enforce consistency along an arc in parallel is given in Fig. 3.

The algorithm goes through the same steps the **Revise** procedure does, but now the supports for all the labels are checked in parallel. A globally shared variable (*change*) is used to indicate if there is any change in the label set of some node in the network. If a^2 processors are available, the **Previse** procedure can be done in constant time. If there were only k ($<a^2$) processors then the time taken by the algorithm is $\mathbf{O}(a^2/k)$. So,

$$IP = T_1/T_\infty = \mathbf{O}(a^2); \qquad P_{\max} = a^2$$

```
boolean function Previse(arc)
begin
    i := arc.start; j := arc.end;
    change := FALSE;
    ||for each l := 1 to a do
    begin
        support[l] := FALSE;
        ||for each l' := 1 to a do
            if R^2_ij(l, l') then support[l] := TRUE;
        if (¬ support[l]) then
            A_i := A_i − {l}; change := TRUE;
    end
    return(change);
end
```

Fig. 3. The Previse function.

Since the support for all the labels is checked in parallel, we now need one variable per label to store this information. So only the tasks checking for support for label l at node i may access *support*[l]. We don't need any synchronization between the tasks which are looking for support for a label. It does not matter how many, or which labels at node j support a label, l, at node i as long there is one. This information is stored in *support*[l]. If this is set to TRUE by any task it means there is some support. Its initial value is FALSE and when a task finds support for it, it sets it to TRUE. If several tasks find support for it and try to update it simultaneously, any one of them will succeed and set it to TRUE. If there is no support, none of the tasks will update the variable and it will have the value FALSE at the end. In all the scenarios the algorithm produces the correct result.

4.4. Parallel AC-1

In the PAC-1 we take a similar approach to AC-1, i.e., check for consistency along each arc in the graph in each iteration. However all the arcs are checked for consistency simultaneously. Depending on the circumstances PAC-1 may take more or less number of iterations as compared to AC-1. PAC-1 is sketched in Fig. 4.

In this algorithm *change* is a Boolean variable which is globally shared between all the processes. There is no need for its protection since it is only the effect that matters; i.e., it doesn't matter which (or how many) labels have changed. Any change implies one more iteration.

The algorithm consists of both inherently serial and parallel parts. The repeat loop has to be done sequentially while the "revise" along each arc may be done simultaneously. Also, as previously described, the **Previse** procedure can itself be done in parallel. So, potentially the algorithm can utilize up to ea^2 processors; a^2 processors by **Previse** for each of the e arcs. So the $\|$ *for* loop in the PAC-1 procedure can be done in constant time. So,

```
procedure PAC-1()
begin
    PNC(); change := TRUE;
    repeat
        change := FALSE;
        ||for each arc := 1 to e do
            change := Previse(arc) ∨ change;
    until (¬ change) ;
end
```

Fig. 4. The PAC-1 algorithm.

the time taken by the algorithm is the number of times the *repeat* loop is executed, times some constant (say α), for the code inside the loop. During each iteration (except the last) at least one label gets removed, because otherwise the algorithm would terminate. Since the total number of labels is na, the maximum number of iterations possible is also na.

If there are only k ($<ea^2$) processors available, then total parallelism inherent in the algorithm cannot be exploited, but a definite speedup can be obtained. During each repeat loop, there are ea^2 tasks which are executed concurrently. When there are only k processors, only k tasks can be done in parallel, thus the time taken to complete all these tasks is $\mathbf{O}(ea^2/k)$.

Thus, the expected speedup for the algorithm is linear in the number of processors up to ea^2 processors. An important observation is that PAC-1 is an optimal parallel algorithm, since it takes the minimum possible time ($\mathbf{O}(na)$).

$$T_\infty = \alpha na = \mathbf{O}(na); \qquad T_1 = \mathbf{O}(a^3ne)$$

$$IP = T_1/T_\infty = \mathbf{O}(ea^2); \qquad P_{\max} = ea^2$$

It can easily be shown, using arguments similar to that for **Previse**, that PAC-1 works correctly. The procedure **Previse** is called for all the arcs simultaneously and if it returns TRUE for any arc the variable *change* is set to TRUE. Again it doesn't matter which task sets it to TRUE as long as some task does. This means at least one more iteration. If *change* has a value of FALSE at the end of the iteration, it implies there has been no change in the label set of any of the nodes, and hence the process has to be terminated. The algorithm works correctly in either case.

4.5. Parallel AC-3

In AC-3 only the arcs associated with the nodes whose label sets have changed during the previous iteration are added to the pool of arcs to be checked for consistency. There are two ways one can parallelize AC-3. The first approach is to exploit parallelism in the **Revise** procedure, but leave the structure of the algorithm untouched. Then arcs are selected and added to the queue in the same way as before. This algorithm looks exactly the same as AC-3, except now we call **Previse** instead of **Revise**. This algorithm can in the worst case have $2e + a(2e - n)$ iterations.[1] Thus, its time complexity is $\mathbf{O}(ae)$.

Here we give another parallel algorithm in which all the arcs in the queue check for consistency simultaneously. During the next iteration the

queue is constructed from the arcs associated with the nodes whose label sets have changed. This is similar to PAC-1, in the sense that here we use only a subset of the arcs. In PAC-1 the queue is static and consists of the complete set of arcs during every iteration. The new algorithm (PAC-3) is sketched in Fig. 5.

The procedure **add_arcs**(*arc*), where $arc = (j, m)$, adds to Q all the arcs, (i, j), in the network such that $i \neq j$ and $i \neq m$. The maximum number of arcs in Q is e since no arc is duplicated. So the maximum number of processors needed is ea^2. Also the **add_arcs** procedure can be done in constant time if there are n^2 processors available. This can be accomplished by a globally shared array of arcs with a field to indicate the inclusion/exclusion of an arc in the queue. Similar arguments hold for the initialization of Q. Clearly, in the worst case the number of times the *repeat* loop is performed is na, since there are at most that many labels and at least one is deleted in each iteration. Thus PAC-3 is also $\mathbf{O}(na)$.

$$T_\infty = \mathbf{O}(na); \qquad T_1 = \mathbf{O}(ea^3)$$
$$IP = T_1/T_\infty = \mathbf{O}(ea^2/n); \qquad P_{\max} = ea^2$$

The correctness of PAC-3 can be proved by using the same arguments for PAC-1.

Although the parallel complexity of PAC-3 is the same as PAC-1, PAC-3 has some drawbacks. First, the cost of building the list of arcs, Q, is not negligible. Although in general PAC-3 uses fewer processors than PAC-1 in the worst case it may end up using the same number of processors to get maximum parallelism. Since the number of processors needed keeps fluctuating from iteration to iteration, it may not be convenient to effectively employ the unused processors for some other purpose.

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

Fig. 5. The PAC-3 algorithm.

4.6. Parallel AC-4

AC-4 uses several data structures to reduce the time taken to remove inconsistent labels. The algorithm consists of two stages: building the appropriate data structures (*Support* and *Counter*), and pruning the inconsistent labels. We do each stage separately in parallel.

The **Pbuild** procedure which builds the data structures in parallel is given in Fig. 6. The number of supports for a label along an arc is stored in the variable *total*. Since the tasks are done in parallel, we now need a variable per arc per label to keep the support information. Also the updates have to be done atomically since the support checks are done simultaneously and more than one label at node j may support the same label at node i.

The procedure **Pbuild** can utilize up to ea^2 processors. They build and update the data structures in parallel. The bottleneck here is in updating *total*. In the worst case all the updates of *total* are serialized and since a unit-label can have at most a supports from any other node, this procedure take $\mathbf{O}(a)$ steps. It should be pointed out here that appending and maintaining *Support* can be done in unit time, by using a 4 dimensional bit array. Thus $T_\infty = \mathbf{O}(a)$ and $P_{\max} = \mathbf{O}(ea^2)$.

To prove the correctness of the procedure it is sufficient to show that the two data structures *Support* and *Counter* are updated correctly. *Support*

```
procedure Pbuild()
begin
    ||for each arc ∈ E do
    begin
        i := arc.start; j := arc.end;
        ||for each l ∈ A_i do
        begin
            total[arc][l] := 0;
            ||for each l' ∈ A_j do
                if R^2_ij(l,l') then
                begin
                    Atomic_add(total[arc][l],1);
                    Support_jl' := Support_jl' ∪ {(i,l)};
                end
            if (total[arc][l] = 0) then
                A_i := A_i − {l}; List := List ∪ {(i,l)};
            else Counter[(i,j),l] := total[arc][l];
        end
    end
end
```

Fig. 6. The Pbuild procedure of PAC-4.

can implemented as a 4-dimensional bit-array, i.e., $Support[i, l, j, l'] = 1$ iff the label l' at node j supports label l at node i. Initially each element of the array is set to zero, and an element is set to one if the corresponding support is found. Under these conditions, the updates of *Support* in the *Pbuild* procedure are always done correctly, since no two processes access the same location of the array at the same time. To make sure that *Counter* is updated properly, it is necessary to keep a separate counter corresponding to each element in *Counter*. These are updated atomically to ensure correctness. After the innermost $\|$*for* loop is completed, the variable $total[arc, l]$ contains the correct number of supports. This is then used to update *Counter*.

Figure 7 explains how the pruning process is done in parallel. There are two levels of parallelism in the algorithm. First, each unit-label pair in the *List* can be processed in parallel. Also, each element in the support set of the unit-label can also be worked on concurrently. Updating of *Counter* has to be done atomically for correctness.

Clearly the maximum number of elements in *List* is na. Once a pair, (i, l), is removed from *List*, it is never processed again. In the worst case, a single element is processed at each iteration of the *while* loop. Hence the maximum number of iterations performed by the outer loop is na. The maximum length of $Support_{jl'}$ is $a(n-1)$, since label l' at node j may support all labels at the other $n-1$ nodes. Due to the synchronization in the inner most loop, in the worst case it can take $\mathbf{O}(a)$ time. However, if there are more than one (say x) identical counter updates (with same i, j, l) it means there are x labels in node j which support label l at node i. It also implies that all of these x labels are deleted, since they are in *List*. Thus, there are x label-node pairs (j, l') which get processed during the same iteration, which means there can now be at most $na - x$ iterations. Thus the overall complexity remains $\mathbf{O}(na)$. The number of processors needed to exploit all parallelism is $\mathbf{O}(n^2a^2)$. Thus, $T_\infty = \mathbf{O}(na)$ and $P_{\max} = n^2a^2$.

```
procedure Pprune()
begin
    while (List ≠ φ) do
        ||for each (j,l') ∈ List do
            ||for each (i,l) ∈ Support_jl' do
            begin
                Atomic_add(Counter[(i,j),l], −1);
                if ( (Counter[(i,j),l] = 0) ∧ (l ∈ A_i) ) then
                  List := List ∪ {(i,l)}; A_i := A_i - {l};
            end
end
```

Fig. 7. The Pprune procedure of PAC-4.

Note that *Counter* is decremented atomically since there may be more than one process trying to update the same location at the same time, since more than one label may support the same label at a node. Therefore, *Counter* is updated correctly. If there is no support for a label and it has not been deleted yet, each process updates *List* and A. Since we are only deleting labels from A, it doesn't matter which process succeeds in deleting it. Also, we can use a two-dimensional bit array to implement *List*; addition of a unit-label pair to *List* is done by setting the appropriate bit. If the appropriate bits of the array are reinitialized to zero at the beginning of each iteration, no further synchronization is necessary to ensure the correctness of updating *List*.

PAC-4 consists of calls to **Pbuild** and **Pprune** sequentially. Thus, the time taken by PAC-4 is the sum of the two procedures and the processor requirement is the maximum of the two.

$$T_1 = \mathbf{O}(ea^2); \qquad T_\infty = \mathbf{O}(a) + \mathbf{O}(na) = \mathbf{O}(na)$$

$$P_{\max} = \mathrm{Max}\{\mathbf{O}(ea^2), \mathbf{O}(a^2n^2)\} = \mathbf{O}(a^2n^2)$$

$$IP = T_1/T_\infty = \mathbf{O}(ea/n)$$

4.7. Comparison of the Parallel ACs

In this section the three parallel algorithms are compared. First it should be noted that all three algorithms are optimal parallel algorithms, since each is $\mathbf{O}(na)$. However, the inherent parallelism in the algorithms is different. The inherent parallelism of PAC-1 is $\mathbf{O}(ea^2)$, while that of PAC-3 is $\mathbf{O}(ea^2/n)$. PAC-4 has $\mathbf{O}(ea/n)$ inherent parallelism. So, PAC-1 has the most inherent parallelism and PAC-4 the least.

The processor requirements of the three algorithms are also different. PAC-1 always needs $\mathbf{O}(ea^2)$ processors to exploit all the parallelism in the algorithm. Although PAC-3 needs fewer processors on the average; in the worst case it still needs $\mathbf{O}(ea^2)$ processors. PAC-4 needs $\mathbf{O}(a^2n^2)$ processors. However for a complete graph, where $e = \mathbf{O}(n^2)$ the processor requirement of all the algorithms is the same.

Since all three algorithms have optimal time complexity, the obvious question is which algorithm performs best on the average? How much time is taken up by the communication and synchronization overheads? None of the algorithms has any explicit communication overheads since all the communication is done through shared memory. All the algorithms need synchronization steps at the end of the $\|$*for* iteration. PAC-4 needs further synchronization while building the data structures, as explained before. PAC-3 on the average does less computation during each iteration, but it

has additional costs in maintaining some lists. This cost can be minimized by trading off space. The structure PAC-4 is totally different from PAC-1 and PAC-3, and hence it is difficult to compare their average execution times. But the space requirement for PAC-4 is greater in general.

It has been independently shown by Kasif[19] that removing inconsistencies in a constraint network is inherently sequential. We have shown that it is indeed so in terms of the labels of the network, i.e., in the worst case the labels of the network will be removed sequentially (proof of Theorem 1). However, it does not imply that we can't get any speedup by using a parallel processor. In fact it can be shown that all these algorithms entail linear speedup (at least in theory) over their sequential counterparts. It is fairly straightforward and will not be proved here. To see which algorithm works better it is imperative to implement them in a parallel processor and analyze their performance.

5. THE BBN BUTTERFLY PARALLEL PROCESSOR

All the algorithms in the previous section are implemented on a BBN Butterfly multiprocessor. In this section a brief discussion of the main features of the machine is given. The Butterfly is a shared-memory MIMD machine capable of having up to 256 processors. All the processors in the machine are identical. Each processor may run its own program independently of the other processors. The memory is shared between all the processors and may be used to communicate between the processors. The memory is physically partitioned among the processors, but each processor can access the memory of every other processor through a switch. Thus even though it is tightly coupled, it is not a true shared-memory multiprocessor. (In contrast, the Cosmic Cube[20] is loosely coupled, where a processor can access only the memory that is local to it. Sequent[21] is an example of a truly shared memory machine.)

5.1. Hardware

The Butterfly hardware consists of two main subsystems: the processor nodes and the butterfly switch. The processor nodes are responsible for the computing job of the machine, while the switch forms the communication system of the machine, and is responsible for the communication between the processor nodes.

The Butterfly parallel processor can have up to 256 processor nodes. Each processor node consists of an MC68020 microprocessor, a *processor node controller* (PNC), and up to 4 megabytes of memory. The machine used to obtain the results in this paper has 18 processors, with all but two

nodes with 1MB of memory. Each processor node has a M68881 floating point coprocessor along with the MC68020 microprocessor.

The PNC occupies a central role in the machine architecture. All the accesses to the memory including the local memory accesses are directed to the PNC. It uses the switch for the remote memory references. In addition it performs several other operations, e.g., atomic operations, I/O interfaces, etc.

All the processor nodes are connected through a switching network called the Butterfly switch. It is a self-routing, packet switching omega network. One important characteristic of the Butterfly switch is that the number of switching elements grows as $N \log N$, where N is the number of processing elements. The bandwidth of the network grows linearly with the number of processors. The maximum data transfer rate between two processors nodes is 32Mbit/second per channel. The ratio of time taken for a remote memory access to a local memory access is roughly 5 to 1 (see Ref. 22 for some tests and precise numbers).

5.2. Software

The application programs run on the Butterfly under the Chrysalis Operating System. Currently *C* and a few other languages are supported. All the low level Chrysalis subroutines can be called from the application program. These include primitives for process creation, synchronization, creation of memory objects, creation and maintenance of queues, etc.

An application library called the Uniform System is also provided. It supports a methodology for programming, where all the processors share a common address space. It is written on top of the Chrysalis system. The application is structured such that all the available processors are used. While it is convenient to use for many applications, it is not very flexible. The programs written for obtaining the results in this paper are written using the low level Chrysalis function directly. The features that have direct bearing on our application are discussed in Section 6.

The memory management system of the Butterfly has important consequences in the design of algorithms. The first Butterfly was designed using the MC68000, which supports only 24 bit addresses. Now even though MC68020 processors are used, which support 32 bit addresses, only 24 bit addresses can be used because of other hardware restrictions. So, the problem here is to map the 24 bit virtual address to a 32 bit physical address. The details of this scheme are discussed elsewhere.[11,23,24]

6. IMPLEMENTATION AND RESULTS

In this section implementation details of the parallel arc consistency algorithms on the BBN Butterfly are described. Our machine has 18 nodes of which two nodes are used by the system and hence are unavailable. All the code is written using the low level Chrysalis primitive functions. This allows us more control over the process creation, task partitioning, memory management, etc.

There are several ways to divide the tasks in order to exploit the parallelism in the system, since the algorithms are so rich in parallelism. However, since we have only 16 processors available, we take a multi-sequential programming approach. The basic idea here is that we create exactly one process on each processor which runs until the whole program is terminated. Since process creation is expensive on the Butterfly, it is a good design decision.

In PAC-1 we partition the arcs equally among all the processors. Each processor works on its arcs, waits for others to finish, and then repeats if necessary. In PAC-3 we keep a queue (DualQ, in Butterfly terminology) to keep all the tasks that need to be executed. Each process waits for a task in the queue. Once it finds a task, it dequeues it, and does the work specified by the task. We use a strategy similar to PAC-1 for the **Pbuild** procedure in PAC-4. However for pruning the labels we use a DualQ which emulates *List* in **Pprune**.

Several low level Chrysalis function calls are very useful. Among them are Atomic_Add to do the atomic operations, Make_DualQ, Enq_DualQ, and Wait_DualQ to create and manipulate queues, etc. The memory is partitioned over all the processors to reduce memory contention. We use only one centralized queue whose address is known to all the processes. Although it is not a big vector in our system, it will be a bottleneck in a large multiprocessor implementation. It would be more efficient to use a multiple-queue scheme in such a system.

6.1. Results

The results of running the algorithms on the Butterly for the cyclic graph (see Fig. 2) are presented in this section. In the ideal case one would like to get a linear speedup with a slope of one. It is also very hard to obtain in practice. The speedups as a functions of the number of processors for PAC-1 and PAC-3 are given in Figs. 8 and 9 respectively. For PAC-4 we give the performance of the **Pbuild** and **Pprune** procedures separately in Figs. 10 and 11 respectively. The speedups are measured for four different problem sizes (number of nodes): $n = 20$, 30, 40, and 50. The number of

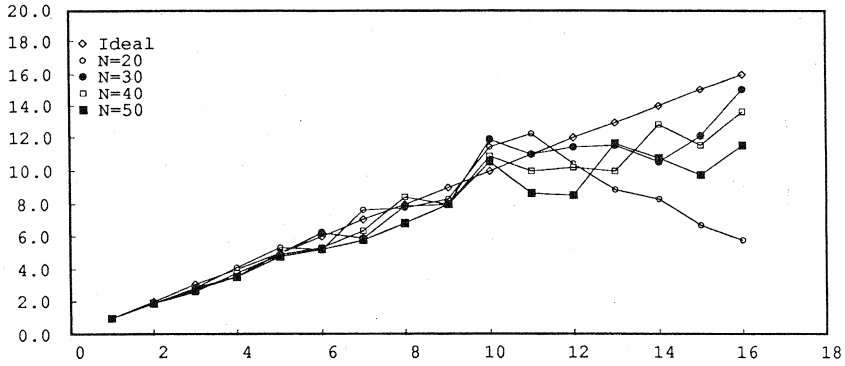

Fig. 8. Speedup vs. number of processors for PAC-1.

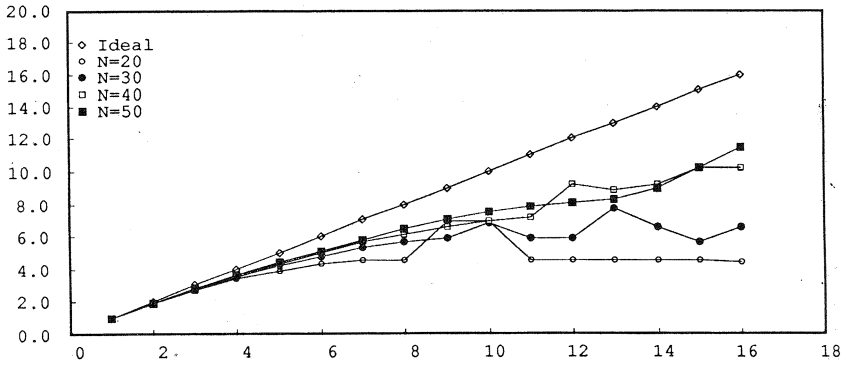

Fig. 9. Speedup vs. number of processors for PAC-3.

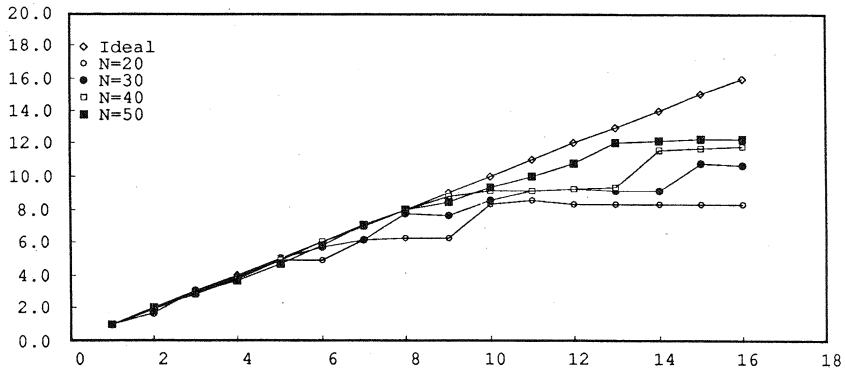

Fig. 10. Speedup vs. number of processors for PAC-4: Pbuild.

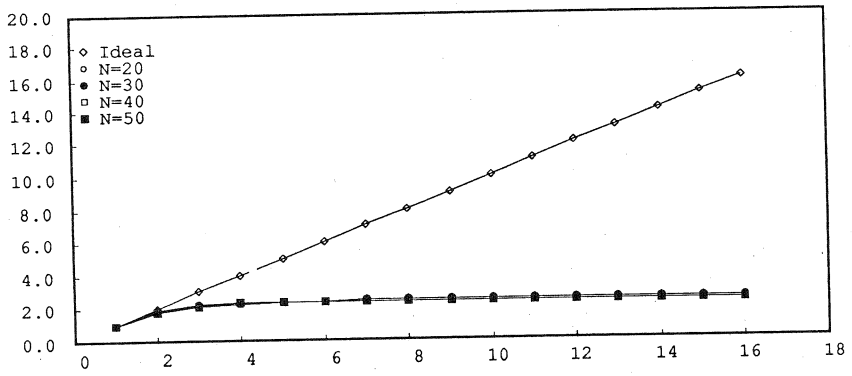

Fig. 11. Speedup vs. number of processors for PAC-4: Pprune.

labels is taken to be the same as the number of nodes. The ideal speedup curves are also shown in the graphs.

In all three algorithms we get fairly good speedup figures. Although the performance is not ideal it is close to it. For PAC-1 we get a maximum speedup of 13.6 using 16 processors ($\eta = 85\%$). The performance of PAC-3 is not as good as the PAC-1. The maximum speedup is 11.4 ($\eta = 71\%$) for PAC-3. The building of data structures in PAC-4 entails good speedup with a maximum of 12.3 ($\eta = 77\%$) using 16 processors. However, for the pruning part there is not much speedup. It is because of the fact that the labels are deleted sequentially one after the other and there is not much parallelism that can be exploited for this graph. The speedup (of 2.0) is obtained because of a cascading effect, where a second process picks up a task immediately when it is put in the queue even though the first task hasn't finished its work. For smaller problem sizes the speedup is not very good for the large number of processors, since there is not enough work to do and quite a bit of time is spent on synchronization. It should be noted that the number of arcs here is also very small (same as the problem size). For PAC-1 we get, in some cases, what is called super-linear speedup, i.e., speedup of greater than the number of processors. It happens in this case because of combinatorial implosion. Since the processes are working concurrently, a change effected by any one of them is immediately seen by the others. This may save some work for the parallel algorithm which the sequential algorithm has to do.

7. CONCLUSIONS AND FUTURE WORK

This work is in part an effort to study inherent parallelism in computer vision algorithms. We have given parallel algorithms for node consistency and arc consistency. The algorithms have been examined for their inherent parallelism and parallel complexity. It has been shown that parallel arc consistency is $\Omega(na)$. Although the serial arc consistency algorithms have different complexities, the parallel algorithms all have the time complexity ($\mathbf{O}(na)$). Also, even though all the algorithms are optimal, they have different inherent parallelisms. The algorithms are implemented on a BBN Butterfly multiprocessor and the results are also presented.

Although the speedups obtained in the algorithms are very good, we believe they can be improved upon. It should also be noted that the results are obtained for a very sparse graph, i.e., the number of edges is small. We are positive that the results would be much better for other types of graphs. We are currently extending the work to other kinds of graphs as well. We have not considered path consistency algorithms here, but we are also looking into inherent parallelism in these algorithms.

REFERENCES

1. Alan K. Mackworth and Eugene C. Freuder, The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence* **25**(1):65–74 (January 1985).
2. Roger Mohr and Thomas C. Henderson, Arc and Path Consistency Revisited, *Artificial Intelligence* **28**(2):225–233 (March 1986).
3. Alan K. Mackworth, Consistency in Network of Relations, *Artificial Intelligence* **8**:99–118, (1977).
4. Ugo Montanari, Networks of Constraints: Fundamental Properties and Applications to Picture Processing, *Information Sciences* 7:95–132 (1974).
5. Donald E. Knuth, Estimating the Efficiency of Backtrack Programs, *Mathematics of Computation* **29**(129):121–136 (January 1975).
6. John Gaschnig, *Performance Measurements and Analysis of Certain Search Algorithms*, PhD thesis, Carnegie-Mellon University, Department of Computer Science (May 1979).
7. Robert M. Haralick, Larry S. Davis, Azriel Rosenfeld, and David Milgram, Reduction Operations for Constraint Satisfaction, *Information Sciences* **14**:199–219 (1978).
8. Azriel Rosenfeld, Robert A. Hummel, and Steven W. Zucker, Scene Labelling by Relaxation Operations, *IEEE Transactions On Systems, Man, And Cybernetics, SMC* **6**(6):420–433 (June 1976).
9. Michael J. Quinn and Narsingh Deo, Parallel Graph Algorithms, *Computing Surveys*, **16**(3):319–348 (September 1984).
10. Walter J. Savitch and Michael J. Stimson, Time Bounded Random Access Machines with Parallel Processing, *Journal of the Association for Computing Machinery* **26**(1):103–118 (January 1979).
11. BBN Laboratories Inc., *Butterfly Parallel Processor Overview*, Report Number 6148 (March 1986).
12. E. W. Dijkstra, Co-operating Sequential Processes, In *Programming Languages, NATO Advanced Study Institute*, F. Genuys, ed., Chapter 2, Academic Press, pp. 43–112 (1968).
13. Ashok Samal, Segmentation and Classification in a Multiprocessor Environment, Ph.D. Dissertation Proposal (December 1986).

14. Leah J. Siegel, Howard Jay Siegel, and Phillip H. Swain, *Parallel Algorithm Performance Measures*, Academic Press, New York, pp. 241–252 (1982).
15. Thomas C. Henderson and Ashok Samal, Parallel Split-Level Relaxation, *International Journal of Pattern Recognition and Artificial Intelligence*, to appear (1988).
16. William A. Kornfeld, *The Use of Parallelism to Implement a Heuristic Search*, Technical Report A.I. Memo No. 627, MIT, MIT AI Lab (March 1981).
17. William A. Kornfeld, *Using Parallel Processing for Problem Solving*, Technical Report A.I. Memo No. 561, MIT, MIT AI Lab (December 1979).
18. Rina Dechter and Judea Pearl, *A Problem Simplification Approach that Generates Heuristics for Constraint Satisfaction Problems*, Technical Report UCLA-ENG-REP-8497, UCLA (1986).
19. Simon Kasif, On the Parallel Complexity of Some Constraint Satisfaction Problems, In *Proceedings of AAAI-86, AAAI*, pp. 349–353 (August 1986).
20. Charles L. Seitz, The Cosmic Cube, *Communications of the ACM* **28**(1):22–23 (January 1985).
21. Sequent Computer Systems, Inc., *BALANCE 8000 Guide to Parallel Programming* (1985).
22. Peter Tinker, *The Design and Implementation of an OR-Parallel Logic Programming System*, PhD thesis, Department of Computer Science, University of Utah (1987).
23. BBN Laboratories Inc., *The Butterfly RAMFile System*, Report Number 6351 (September 1986).
24. BBN Laboratories Inc., *Chrysalis Programmers Manual Version 2.3.1.* BBN Report Number 6191 (August 1986).