Reprinted from:
INTERNATIONAL JOURNAL OF PARALLEL PROGRAMMING

Vol. 20, No. 6, December 1991
Printed in Belgium

Parallel Path Consistency

Steven Y. Susswein,[1] Thomas C. Henderson,[1]
Joseph L. Zachary,[1] Chuck Hansen,[2]
Paul Hinker,[2] and Gary C. Marsden[3]

Received October 1991; Revised June 1992

Filtering algorithms are well accepted as a means of speeding up the solution of the consistent labeling problem (CLP). Despite the fact that path consistency does a better job of filtering than arc consistency, AC is still the preferred technique because it has a much lower time complexity. We are implementing parallel path consistency algorithms on multiprocessors and comparing their performance to the best sequential and parallel arc consistency algorithms.[1,2] (See also work by Keretho *et al.*[3] and Kasif[4]). Preliminary work has shown linear performance increases for parallelized path consistency and also shown that in many cases performance is significantly better than the theoretical worst case. These two results lead us to believe that parallel path consistency may be a superior filtering technique. Finally, we have implemented path consistency as an outer product computation and have obtained good results (e.g., linear speedup on a 64K-node Connection Machine 2).

KEY WORDS: Path consistency; parallel implementation.

1. INTRODUCTION

The Constraint Satisfaction Problem can be defined as a problem in logic.[5] Let $X = \{X_1, ..., X_n\}$ be a set of n variables, and let $P = \{P_1, ..., P_n, P_{1,2}, ..., P_{n-1,n}\}$ be a set of unary and binary predicates which the values of the variables must satisfy. Each variable, X_i, takes on a set of values from some domain, D_i. We also require that $P_{i,j}(v_i, v_j) =$

[1] Department of Computer Science, University of Utah, Salt Lake City, Utah 84112.
[2] Los Alamos National Laboratories, Los Alamos, New Mexico.
[3] Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, California.

0885-7458/91/1200-0453$06.50/0

$P_{j,i}(v_j, v_i)$. Then the goal is to find an n-tuple, $v = (v_1, v_2, ..., v_n)$, of values for the variables such that:

$$P_1(v_1) \wedge P_2(v_2) \wedge \cdots \wedge P_n(v_n) \wedge P_{1,2}(v_1, v_2) \wedge P_{1,3}(v_1, v_3) \wedge \cdots \wedge P_{n-1,n}(v_{n-1}, v_n) \quad (1)$$

This problem is also known as the Satisfycing Assignment Problem,[6,7] the Discrete Relaxation Problem,[8] and as the Consistent Labeling Problem.[9,10] We shall refer to this problem as the *Consistent Labeling Problem* (CLP). Many classical computer science problems such as N-queens, magic squares, and the four color map problem can be viewed as CLP.

CLP is equivalent to a graph problem in which we have:

- A set of *nodes*, $N = \{n_1, n_2, ..., n_n\}$; (let $|N| = n$).
- For each node n_i a domain D_i, which is the set of acceptable labels for that node. Often all the D_i's are the same, giving $D_1 = D_2 = \cdots = D_n = D$; (let $|D| = a$).
- A set of *constraint relations* $P_{i,j}$, $i, j = 1, n$, which defines the consistent label pairs which can be assigned to nodes n_i and n_j; i.e., $P_{i,j}(l_i, l_j)$ means label l_i at node n_i with label l_j at node n_j is a consistent labeling. Directed arcs give a visual representation of the relationships.

The problem in this case is to find a complete and consistent labeling such that each node is assigned a label from its label set that satisfies the constraints induced by all its connected arcs. In this work, we restricted ourselves to problems in which the domain of interpretation of each variable is finite. This formulation omits types of constraint problems such as temporal or interval constraint problems, in which the domains can be infinite.[11] Ladkin and Maddux[12] have also formulated arc consistency and path consistency algorithms without assuming finite domains. The algorithms studied there are parallelizable versions of Montanari's original sequential path consistency algorithm,[13] as is our algorithm. The implementations differ, however, and efficiency of implementation is crucial to the success of these algorithms. One of the major goals of this paper is to demonstrate the practical utility of parallel path consistency.

It can be shown that CLP is NP-complete.[9,10] However, there are a number of ways the problem can be solved in practice, including *generate and test* and standard backtracking, etc. In standard backtracking, we assign a label to $node_1$, and using this constraint, attempt to find a valid label for $node_2$. Using these values for nodes one and two, we attempt to

find a valid label for $node_3$, etc. When no valid label exists for a node, we backtrack and make a new assignment for the last node. We continue until all nodes have been assigned labels or all possible assignments have been attempted, and failed.

Mackworth[14] has shown that the "thrashing" behavior of standard backtracking can potentially be reduced in practice by the incorporation of consistency algorithms (*node*, *arc*, and *path* consistency). Mohr and Henderson[15] have given an optimal algorithm for arc consistency and an improved algorithm for path consistency.

In order to achieve (1), several heuristics to limit the graph search can be defined and enforced:

- *node consistency:* for every label at every node, delete the label if it fails to satisfy the unary predicate at that node.

$$\begin{array}{l}\forall i = 1, n \\ \quad \forall v_i \in D_i \\ \quad\quad \text{if } \neg P_i(v_i) \text{ then } D_i \leftarrow D_i - \{v_i\}\end{array}$$

- *arc consistency:*

$$\begin{array}{l}\forall i = 1, n \\ \quad \forall j = 1, n \\ \quad\quad \forall v_i \in D_i \\ \quad\quad\quad \text{if } \neg \exists v_j \in D_j \text{ such that } P_{i,j}(v_i, v_j) \\ \quad\quad\quad\quad \text{then } D_i \leftarrow D_i - \{v_i\}\end{array}$$

  This process is iterated until no labels are deleted.

- *path consistency:*

$$\begin{array}{l}\forall i = 1, n \\ \quad \forall j = 1, n \\ \quad\quad \forall k = 1, n \\ \quad\quad\quad P_{i,j} \leftarrow P_{i,j} \wedge (P_{i,k} \circ P_{k,k} \circ P_{k,j})\end{array}$$

  Here, $P_{i,j} \circ P_{l,m}$ is the composition operator on relations. This process is iterated until no relation is changed.

Montanari[13] was the first to define path consistency and has shown that if all paths of length two are consistent, then all paths of any greater length than two are also consistent; therefore, in practice, only paths of length two need be considered to ensure path consistency.

Path consistency is a heuristic technique that many believe does a much better job of filtering than arc consistency, but it is also much slower (i.e., it adds a level of complexity from serial quadratic to serial cubic). As a result, arc consistency is currently the most widely used filtering technique.

There are some domains in which arc consistency is trivial but path consistency is an extremely effective heuristic (e.g., interval constraint problems). However, these constraint problems all have infinite domains, and this class of problems is not considered here.

1.1. Parallel Algorithms for AC and PC

Samal and Henderson have explored parallel versions of arc consistency.[16,17] Kasif[4] has shown that arc consistency is inherently sequential, and he has suggested applying parallelism in a controlled way. We are interested here in exploring implementations of parallel path consistency algorithms. We conjecture that the average case time complexity of parallel path consistency is $O(na)$. This means that over populations of standard problems and given a polynomial bound on the number of processors, the average time to solve the path consistency problem is proportional to na. In fact, our preliminary results indicate that the outermost loop of path consistency (i.e., that which updates the relations) runs on the average in constant time, $O(2)$.

Ladkin and Maddux[12] have defined a set of constraint relations that exhibits worse case iteration complexity, $O(n^2)$. It is now known that there is a class of relations on an 84 element set which exhibits the behavior from which the Ladkin-Maddux $O(n^2)$ examples are constructed, so the known lower bound for local, iterative algorithms is now $\Omega(n^2)$ for finite domains.

2. SEQUENTIAL PATH CONSISTENCY

The current best path consistency algorithm (PC-3) has a time complexity of $O(n^3a^3)$, compared to the optimal arc consistency algorithm (AC-4) which has a time complexity of $O(n^2a^2)$[18] but for some problems path consistency does a much better job of pruning the search space. This can be seen by looking at the 4-*Queens* problem. Path consistency will prune 50% of the labels from each node, leaving just two possible positions for each queen; arc consistency on the other hand prunes *none* of the labels, leaving the problem at its original complexity.

The three major sequential path consistency algorithms which have been proposed to date include: PC-1 by Montanari,[13] PC-2 by Mackworth,[14] and PC-3 by Mohr and Henderson[15]; we use PC-4 here, a

corrected version of PC-3, given by Han and Lee.[19] These algorithms are derived from arc consistency algorithms AC-1, AC-2, and AC-4, respectively.[13–15] For a review of discrete relaxation and its application, see Henderson.[18]

2.1. Sequential PC-1

Algorithm PC-1 is a straightforward generalization of AC-1, and is shown in Fig. 1. PC-1 is a brute force algorithm.

The nested loops on lines 4–21 examine all node-label triples to make sure that every node pair—label pair $P_{i,j}(l_i, l_j)$ has support through all length two paths $i-k-j$; thus, lines 4–21 have complexity $O(n^3a^3)$. (Note that some authors do not include the number of labels in the complexity, and would call this $O(n^3)$.) When an inconsistent element $P_{i,j}(l_i, l_j)$ is found, the corresponding entry in the relation matrix is changed from **TRUE** to **FALSE**. Since the relation matrix contains n^2a^2 elements and can only change from **TRUE** to **FALSE**, the worst case complexity of the **repeat**—*until* loop is $O(n^2a^2)$. This gives a worst case complexity for the algorithm of $O(n^5a^5)$.

2.2. Other Path Consistency Algorithms

Just as PC-1 is derived from AC-1, PC-2 is a generalization of AC-2. Rather than check all node triples every time a relation entry is set to

```
1   procedure PC-1
2   begin
3     repeat
4       change := FALSE;
5       for i := 1,n do
6         for l_i := 1,a do
7           for j := 1,n do
8             for l_j := 1,a do
9               begin
10              for k := 1,n do
11                begin
12                k_support := FALSE;
13                  for l_k := 1,a do
14                    k_support := k_support ∨ [P_{i,k}(l_i, l_k) ∧ P_{k,k}(l_k, l_k) ∧ P_{k,j}(l_k, l_j)];
15                if ¬ k_support then
16                  begin
17                  P_{i,j}(l_i, l_j) := FALSE;
18                  change := TRUE;
19                  end;
20              end
21            end
22    until ¬ change;
23  end.
```

Fig. 1. PC-1 Algorithm.

FALSE, we consider only the length two paths that are related to the nodes i and j. Algorithm PC-2 is of complexity $O(n^3a^5)$. Similarly, PC-4 is derived from AC-4. Compared to PC-1 and PC-2, PC-4 uses considerably more space due to its use of counters. The space complexity of PC-4 is $O(n^3a^3)$, and its time complexity is also $O(n^3a^3)$. While AC-4 is an optimal sequential algorithm, it has not yet been determined whether PC-4 is an optimal path consistency algorithm. We explain later why we do not explore parallel versions of PC-2 and PC-4; for sequential versions of these algorithms, see Mackworth[14] and Han.[19]

2.3. Using PC in Search

The next step involves creating a standard backtracking program, in which various parallel AC and PC routines are embedded. At each node of the search tree we run the chosen AC or PC code to check for consistency. Again, we measure the raw performance and speedup, as well as the average, minimum, and maximum search depth and the number of nodes traversed.

Although there has been some study of parallel algorithms for search, we have restricted our attention to sequential search with parallelization of the path consistency heuristic. Another issue that arises is the finding of Dechter and Meiri, [20] that in sequential versions of search with arc and path consistency heuristics, arc consistency gives better performance. This is consistent with our results for sequential versions of these algorithms. However, in the parallel implementations, our results indicate that the path consistency heuristic gives better performance.

Although theoretical worst case performance of sequential PC-1 is of complexity $O(n^5a^5)$, early experiments have shown actual performance to be much better. We are attempting to find and categorize the worst case performance based on the type of graph and constraint relation. We have also tested the algorithm on randomly generated relation matrixes for a range of values for n and a. We have included the standard test cases of N-queens and confused n-queens[21] for performance measurement and comparison.

We already had a working version of arc consistency created by Samal, and PC-1 was coded based on the algorithm given by Mackworth. Both these programs use identical system calls to report timing information and were run on a number of both consistent and inconsistent graphs. These graphs mostly corresponded to the *N-Queens* problem (for various values of N), but other graphs were also examined. As expected, arc consistency ran much faster than path consistency, but path consistency

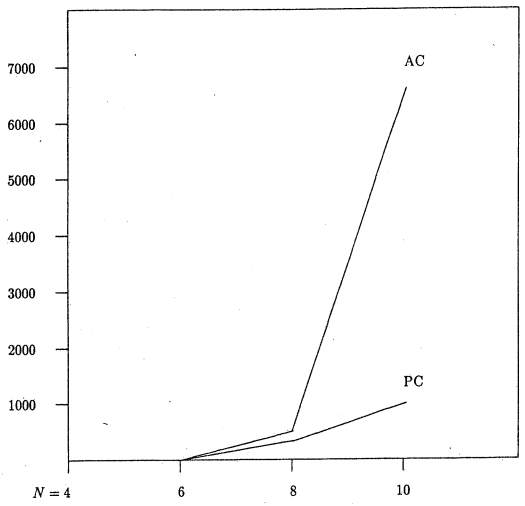

Fig. 2. Number of nodes visited in N-Queens for AC and PC.

did a superior job of pruning the search space. As mentioned earlier, a good example of this is consistent 4-Queens. Figure 2 shows number of nodes visited in the search tree for n-queens ($n = 4, 6, 8, 10$).

3. PARALLEL PATH CONSISTENCY

We have conducted some simple experiments. These experiments support the following claims:

1. *Path consistency* prunes the search space to a greater extent than *arc consistency*. i.e., our experiments support previous work on this.
2. Highly parallelized versions of *path consistency* can achieve near-linear speedup.
3. *Path consistency* will normally run in much better than theoretical worst case performance.

In fact, on average over a random sample of relation matrixes, parallel PC-1 runs in constant time. In addition, its worst case performance seems to be linear, and we have found that the implementation exhibits nearly linear speedup.

3.1. Parallel PC-1

We are currently investigating parallel versions of the PC-1 algorithm and comparing its performance to the parallel AC algorithms. The best sequential AC algorithm is not necessarily the best basis for a parallel algorithm. For each algorithm we measure its raw speed as well as its efficiency, with the goal of finding a parallel PC algorithm with at least linear speedup. Efficiency is a measure of how well we are utilizing the additional processors, and it is defined as (*time for sequential algorithm*)/($N \times$ *time on N processors*). The speedup is given as (*time of sequential algorithm*)/(*time of parallel algorithm*).

PC-1 is clearly the most easily parallelizable of the three path consistency algorithms. Assuming a sufficient number (polynomial in n and a) of processors, each iteration through the outer loop can check all label assignments on all length two paths in parallel, thus giving a time complexity of $O(1)$ for this part of the algorithm. The remaining complexity factor in the algorithm is the number of iterations through the outer loop, which has a worst case complexity of $O(n^2a^2)$, giving a worst case complexity for parallel PC-1 of $O(n^2a_2)$. The parallel PC-1 algorithm is the same as sequential PC-1, except that the **for** loops are parallel **for** constructs which create k tasks (where k is the upper limit of the **for** loop).

Table I. Efficiency for 16-Queens using PC-1[a]

processors	raw time (ms)	efficiency	
		2 iterations	3 iterations
1s	602980	1.00	
1p	626305	0.96	
2	322272	0.94	
3	213479	0.94	
4	244888		0.62
5	128830	0.94	
6	169108		0.59
7	142597		0.60
8	123289		0.61
9	113087		0.59
10	101053		0.60
11	93206		0.59
12	84602		0.59
13	78071		0.59
14	72184		0.60
15	44201	0.91	

[a] Note: 1s is sequential code and 1p is parallel code.

These tasks execute concurrently, and the statement immediately following the parallel **for** is only performed once all k tasks generated by the parallel **for** have completed.

We have not explored parallel versions of PC-2 and PC-4 because both contain an outer loop of similar complexity to PC-1's outer loop, which means that even in the best case, parallelized versions of these algorithms could not improve on the performance of parallel PC-1. In addition, they both require more space than PC-1 (a great deal more in the case of PC-4). Another factor in choosing PC-1 as a candidate for parallelization is the outcome of experiments done by Samal.[16] He examined parallelized versions of the arc consistency algorithms AC-1, AC-2 and AC-4, and concluded that the best parallel performance was achieved by AC-1.

As a next step, we modified the PC-1 program mentioned earlier to run as a parallel program on the Butterfly. We employed a straightforward parallelization, where the number of parallel processes generated is based on the size of the initial graph. Larger graphs have shown an approximately linear speedup, up to the number of processors available (see Table I). Note that the number of iterations varies slightly due to interactions caused by the parallelization, and the speedup remains linear only for equal iteration counts.

Figure 3 shows the worst case and average case number of iterations over a set of 10,000 randomly selected trials per selection of n and a (ranging from 2 to 10). This shows that the average number of iterations is constant (about 2), and the worst case is linear in na. Figure 4 shows the

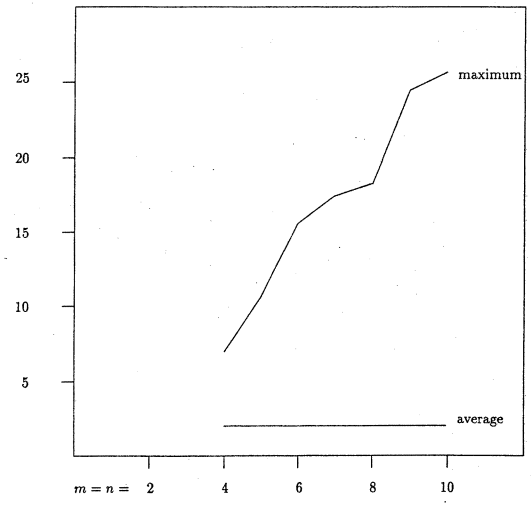

Fig. 3. Avg. and max. iterations for $n = a = 2,..., 10$.

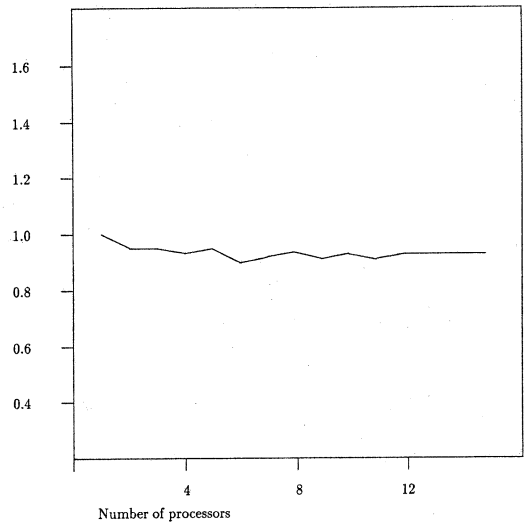

Fig. 4. Efficiency per iteration.

efficiency per iteration. (Efficiency is the ratio of the time of the sequential algorithm over k times the time of the parallel algorithm with k processors.) Finally, Fig. 5 shows the percentage of time spent in the consistency part of the code (versus in the backtracking), and indicates that it is advantageous to parallelize PC-1, since almost all the search time is spent in PC-1.

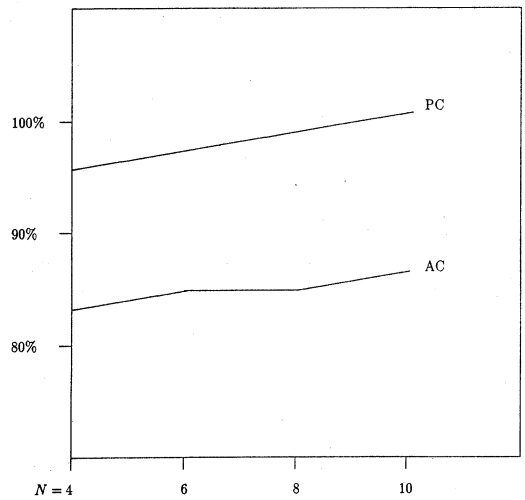

Fig. 5. Percentage of execution time spent in filter code in N-Queens search.

3.2. Worst Case Performance

The Consistent Labeling Problem input to PC-1 is encoded in the form of an $na \times na$ binary matrix (this can also be viewed as an $n \times n$ matrix of the $P_{i,j}$ relations each of which is an $a \times a$ matrix). The algorithm iterates over this matrix until two successive iterations yield no change in the matrix. Each iteration is of complexity $O(n^3a^3)$ and can only simplify the matrix (i.e., change a "1" to a "0"). Since each iteration simplifies at least one element in the matrix, we require as a worst case n^2a^2 iterations, yielding a worst case performance of $O(n^5a^5)$.

Since the input matrix defines both the list of possible labels for each node and the constraint relation between nodes, it is possible to exhaustively examine all possible relation constraints for small values of n and a through a brute-force approach of constructing all possible input matrixes. The purpose of this experiment was to find which constraint relations produced the worst results (greatest number of iterations). While we haven't been able to fully characterize which constraint relations produced the most iterations, we were surprised by the maximum and average number of iterations required. Using values of $a=2$ and $n=3$ yielded a maximum performance of 5 iterations (compared to a theoretical worst case of 30 iterations) and an average case performance of 2.07 iterations (see Fig. 6). (Note: although there are nine 2×2 matrixes with 4 elements each, making 36 bits in the relations matrix, the diagonal $P_{i,i}$ matrixes can

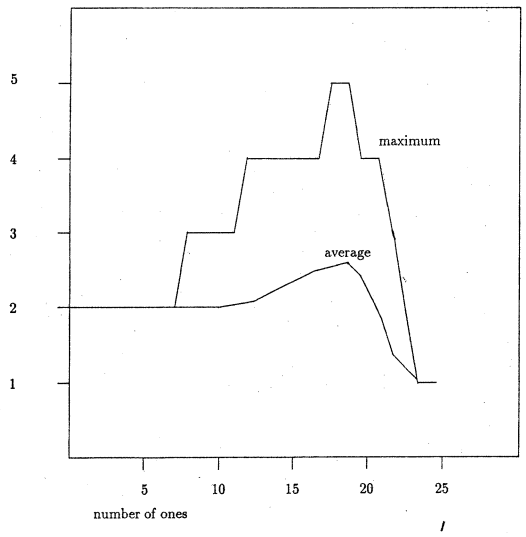

Fig. 6. Avg. and max. iterations for all relations of $a=2$ and $n=3$.

only have 0's in their nondiagonal elements; thus, the 6 off-diagonal bits of the diagonal $P_{i,i}$ matrixes are always 0, and only 30 bits are free to take on either binary value.) The maximum number of iterations shown in Fig. 6 is not the theoretically worst case possible, but is the maximum number of iterations observed for any relation matrix tried.

Additional experiments varying the value of n and a for a fixed relation showed that the number of iterations required remains small and relatively constant for at least some relations. If found to be generally true for all relations this would make parallel path consistency even more attractive. Each iteration in PC-1 can be highly parallelized (we parallelize the **for** loops over the nodes and label triples), but the iterations themselves are performed in sequence. The number of iterations required (whose upper bound is theoretically n^2a^2) places an upper bound on the efficiency of parallel PC; if the number of iterations required is found to be small and relatively constant for large values of n and a, then parallel path consistency may prove to be an excellent filtering technique.

4. TOOLS AND FACILITIES

All the code is written in standard C. Timing information is gathered using standard Unix system calls (for the sequential code) and Uniform built-in timing routines (for the parallel code on the Butterfly).

4.1. DECStation 3100

Sequential code was developed and run on a dedicated DECStation 3100, a high-performance RISC workstation. Code developed here under ULTRIX is source-code compatible with the HP Bobcat workstations, but its high performance (approximately 3× an HP370) and lack of contending jobs means that large runs can be completed quickly.

4.2. Butterfly GP1000

Parallel code has been developed and run on the BBN Butterfly multiprocessor. The Butterfly offers two means of accessing its multiprocessor features: direct system calls to the Mach operating system, and the Uniform system. The Uniform system consists of a library of routines which allows easy access to the multiprocessing features. While not as powerful as direct Mach calls, it is much easier to use and supplies all the features needed to implement parallel path consistency.

Due to its regular nature, parallel PC-1 was most easily implemented using the Uniform library routines. The sequential version of PC-1 used for

the sequential experiments was modified to include the Uniform system calls in such a way that different levels of parallelization could be tried. Although the Butterfly used is configured with 18 nodes, only 15 of these were available for parallel applications, thus limiting the maximum speedup that could be obtained. These nodes were used as a common pool and allocated to processes as needed.

After experimenting with different levels of parallel granularity, it was found that maximum performance was obtained with a granularity of no more than n^3 processes, each with an internal complexity of a^3. Reducing the process size below this level led to unacceptable levels of overhead which reduced performance. This level of granularity still resulted in many more processes than the number of processors available, so the actual performance would not be improved by a smaller granularity in any case. To achieve this level of parallelization, the three node related **for** loops were parallelized.

The 18 nodes of the Butterfly is sufficient for development and testing and to show the effect of parallel PC-1.

5. PARALLEL OUTER PRODUCT FORMULATION OF PATH CONSISTENCY

In addition to the path consistency algorithms discussed above, another method of computing path consistency has been suggested by Marsden *et al.*[22] This method is based on vector outer products and matrix summation and intersection, and is well suited to a highly parallel implementation. As with the other PC algorithms, it is also based on the relation matrix data structure.

The following algorithm is a slight variation of the original Marsden algorithm, in that it does not assume a symmetric relation matrix. However, it should be noted that a symmetric matrix formulation could be exploited, since $P_{i,j}(l_i, l_j)$ can be set to zero if $P_{j,i}(l_j, l_i)$ is zero. That is, for $i = 1, n; j = i + 1, n$; $P_{i,j}$ can be set equal to $P_{i,j} \wedge (P_{j,i})^T$ before starting the path consistency check. This algorithm computes the same results as the three loops in PC-1 and must be repeated until the relation matrix stabilizes. (Note that Marsden *et al.*[22] also show how the outer product computation can be used to compute k-consistency for any k.)

Recall that there are na rows in the relation matrix (which combines all the $P_{i,j}$'s into one matrix), corresponding to n nodes with a labels each. The algorithm iterates over the rows of the relation matrix, with nested iterations for each node (i) and label (j). For each iteration ij, the ij-th row is extracted from the relation matrix, and an outer product of this row with column ij is computed. This outer product matrix is added to a summation

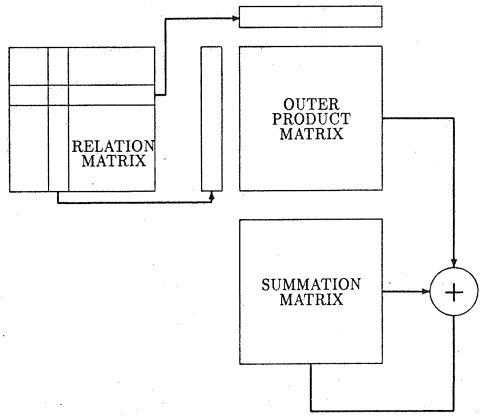

Fig. 7. Outer product PC: computation of summation matrix.

matrix, which stores the running sum of these outer products. After iterating on all labels for a given node (a outer products), the summation matrix is intersected with the relation matrix to yield an updated relation matrix. This process is iterated for all nodes (n times). Figures 7 and 8 show a visual representation of this algorithm (adapted from Ref. 22).

Computing a vector outer product from two vectors of length na has a complexity of $O(n^2a^2)$. This outer product is computed for each row in the relation matrix (na times), giving a total complexity of $O(n^3a^3)$ for each iteration of path consistency. This is exactly equivalent to the complexity of each iteration of PC-1.

We have mapped the outer product computation onto a suitable highly parallel processor, and have analyzed its performance there. The Connection Machine is a reasonable choice for such an analysis, and it is used to model and implement parallel outer product PC (POPPC).

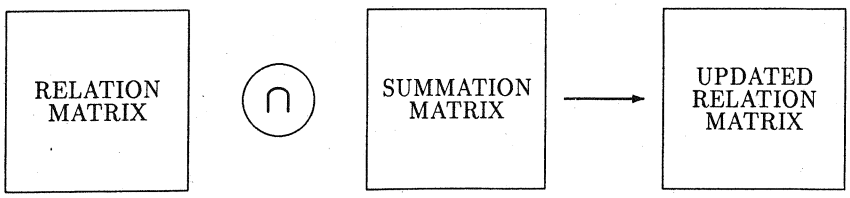

Fig. 8. Outer product PC: computation of updated relation matrix.

5.1. The Connection Machine

The connection machine is a massively parallel, fine-grained SIMD processor.[23] It provides a large number of tiny processor/memory cells connected by a programmable communications network. The Connection Machine 2, for example, has 65,536 processor nodes, each with 32K bytes of local memory. The Connection Machine is easily scalable, but this is considered a large configuration. Parallel data structures are spread across the processor cells, with a single element stored in each processor's memory. When parallel data structures have more than 65,536 data elements, the hardware operates in virtual processor mode, presenting the user with a larger number of processors, each with a correspondingly smaller memory. Communications between elements of a parallel data structure is carried out over the high-speed routing network. Processors that hold related data elements store pointers to one another. When data are needed, they are passed through the routing network to the appropriate processor. The interface to the Connection Machine is via a sequential front-end processor, which also holds scalar data elements and performs nonparallel computations.

Given a data structure that has been spread across the processor cells, many operations can be computed in unit time ($O(1)$). This is because each processor element acts independently on a single element of the data structure. Examples of such operations are *search* and *delete*, and matrix operations such as *copy*, *intersection*, and *addition*. Other operations that involve counting, reducing, or numbering the elements take place in logarithmic time, because they are implemented by algorithms using balanced trees.[24] Quoted performance for a 65,536 node machine is 2500 MIPS for 32-bit fixed point instructions; the routing network has a throughput of 250 million 32-bit messages per second.

We implemented path consistency on the Connection Machine 2 (CM-2) using the outer product algorithm previously described. On the total machine, there is 8 gigabytes of physical memory. The machine can by split into quadrants of 16K processors each or can be accessed as half the machine (32K processors) or as a full 64K device. The CM-2 executes in a SIMD parallel fashion where each processor executes in lock-step with the others. Certain processors can be masked out of the operation but cannot concurrently execute different instructions.

Path Consistency maps extremely well onto this type architecture. We can represent the relations as 2D bit-maps. Thus, for a relation graph with n nodes and a labels, we only need a binary matrix of size n^2a^2. The algorithm proceeds in two steps:

1. set up the initial relation matrix

2. iterate over the nodes and labels using the outer product technique to reduce incompatible relations.

The first phase of setting up the initial relation matrix would be typically down-loaded from the front-end. For the timing results given here, we generated the initial relation matrix on the CM-2. Another method for loading the CM-2 with the proper initial relation matrix is to utilize the CM-2 file server and load the file directly from the Data Vault (a massive storage device which is directly attached to the CM-IO bus) or through the Data Vault's front-end via ethernet. Once the initial relation matrix is in physical memory on the CM-2, the path consistency computation can proceed.

Based on this machine specification, the following relatively simple parallel algorithm was developed, called POP PC. The outer product computation at each $P_{i,j}(l_i, l_j)$ element is:

$$P_{i,j}(l_i, l_j) = P_{i,j}(l_i, l_j) \wedge \prod_k \sum_{l_k} [P_{i,k}(l_i, l_k) \wedge P_{k,j}(l_k, l_j)]$$

Let $c = P_{i,k}(l_i, l_k)$ and $r = P_{k,j}(l_k, l_j)$, then Fig. 9 shows the contribution of (k, l_k) to the outer product at $P_{i,j}(l_i, l_j)$. Thus, to parallelize the algorithm, each CM processor must get 1 bit from the complete P matrix, 1 summation bit, and finally, so as to go as fast as possible, the row and column which are used to compute the $P_{i,j}$ bit. This leads to the algorithm in Fig. 10.

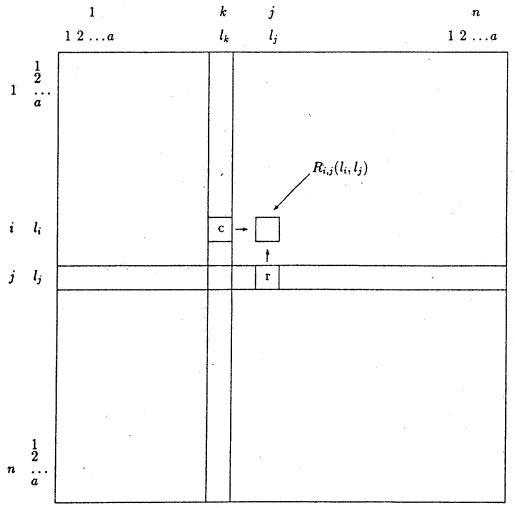

Fig. 9. Outer Product Computation.

```
1   procedure POP PC
2   begin
3     while change
4       for node := 1,n do
5         begin
6         for label := 1,a do
7           begin
8           index := offset(node,label);
9           SUM := SUM ∨ (Row[index] ∧ Col[index]);
10          end;
11        R := R ∧ SUM;
12        end;
13  end.
```

Fig. 10. Parallel Outer Product Path Consistency (POP PC).

While the relation matrix is changing, we iterate over each node. For each node, we form an outer product for each label (represented by a row and column in the relation matrix) and OR those into a temporary matrix we call the SUM matrix. When all labels are exhausted for a particular node, we AND the SUM matrix with the relation matrix and iterate over the next node's labels. When we have exhausted all nodes, we either quit, signified by the relation matrix not changing as a result of the AND'ing of the SUM matrixes, or start the iteration again, signified by a modification of the relation matrix.

5.1.1. Implementation Details

For the actual implementation, we were forced to slightly modify the algorithm for more efficient execution on the CM-2. All matrixes on the CM-2 must be powers of two. Thus when allocating storage, we needed to determine the closest power of two which was greater than or equal to n^2a^2. Since each of the matrixes can be represented with a bit map, we only allocated a single bit per matrix entry. We assigned a processor to each memory location. Since we weren't guaranteed to have a relation matrix of size less than or equal to the physical number of processors, we allocated Virtual Processor Sets (VP-sets); if we were restricted to physical processors, we couldn't process a relation matrix greater than 128×128 on the 16K CM-2. The VP ratio is the ratio of the number of virtual processors to physical processors. Since we might have allocated some extra VP's if our relation matrix was not a power of two, we masked out those processors such that they performed no computations.

There are three types of CM-2 communication: nearest neighbor, utilizing the router (any processor to any processor) and scans. The CM-2

can be configured with hypercube connectivity. Since we are dealing with 2D bitmaps, we simply use a 2D nontoroidal mesh topology. Nearest neighbor communication is thus left-right-up-down (north-south-east-west); this is the fastest method of communication. Conversely, router communication is the slowest since it depends on the routing system to determine the destination of the communication. The scan methodology is somewhere in the middle. It should be $O(\log k)$ where k is the number of VP's. When we actually compute the outer product, we must take a row and a column from the relation matrix and generate an outer product matrix using those. The fastest way of generating this is to use two scans and logically AND the two to form the outer product matrix. We implemented this by forming one temporary matrix with a copy_spread command which takes a column (na-vector) and forms an $na \times na$ matrix by replicating the column. We generated a second temporary matrix by using the copy_spread along the row axis which forms an $na \times na$ matrix which replicated the row. If these two matrixes are logically AND'ed together, this produces the outer product.

For the implementation, each VP has the relation matrix, a copy of the relation matrix, the two temporary matrixes and the SUM matrix. Recall that these are all bit-maps, thus each VP only needs 5 bits to store all the information. As we are iterating over the labels within a node, we can logically OR into the SUM matrix, the result of AND'ing the two temporary matrixes together. When we have exhausted the labels, we AND the SUM matrix with the relation matrix. When we have exhausted the nodes, we bit-wise compare the relation matrix with the copy of the relation matrix. If the two are equivalent, we have finished since the relation matrix didn't change. If the two aren't equivalent, we copy the relation matrix and iterate again.

5.1.2. Timings

As the timings in Table II indicate, we found linear speedup with respect to the number of physical processors. As the VP ratio increased, there was some speed degradation. This was due to the fact that there were more labels; thus, the iterations were longer as well as the fact that the scan-based communication was slowed down due to the increased number of virtual processors.

All of these represent one pass through the path consistency check. We believe that the computation is slowed because of the spread that must be done when performing the outer product. This involves a broadcast (two actually) which is fairly expensive with respect to everything else the code does. Since the spread is done $2N$ times, where $N = na$ (i.e., the length of

Table II. Summary of CM-2 Results

Problem Size	Processors	Speed Up	VP-Ratio	Elapsed Time (sec)
256×256	16384	1	4	0.25029
484×484	16384	1	16	0.739879
1024×1024	16384	1	64	4.022633
2025×2025	16384	1	256	26.220621
4096×4096	16384	1	1024	204.557243
256×256	32768	1	2	0.260105
484×484	32768	1.33	8	0.555307
1024×1024	32768	1.66	32	2.429805
2025×2025	32768	1.84	128	14.287050
4096×4096	32768	2.00	512	102.245058
256×256	65536	1	1	0.216853
484×484	65536	1.52	4	0.488031
1024×1024	65536	2.52	16	1.596955
2025×2025	65536	3.24	64	8.090925
4096×4096	65536	3.80	256	53.884622

either the x or y axis), this causes more and more time to be spent doing the spread as the problem size increases.

These test size cases were picked because the CM-2 has a restriction that the length of an axis must be an integral power of two. Another restriction is an integral VP ratio. So, the smallest problem that can be run on a 16K machine is 128×128 and the smallest problem that can be run on the whole machine is 256×256. Since the N-queens problem causes the shape of the problem space to be $na \times na$, the test cases are the largest that fit into the successive, acceptable CM-2 geometries.

6. CONCLUSIONS

A fast, efficient path consistency computation can greatly aid in filtering backtrack search. We have described several methods here which should be of great value to this aim. The linear worst case performance of the parallel algorithm is unexpected; moreover, constant time average case complexity indicates that parallel path consistency may be of great use once parallel computation is readily available. However, its useful application requires further study on a wider class of problems and graph geometries.

ACKNOWLEDGMENTS

This research was supported in part by NSF Grant CDA-9024721. We would also like to thank the reviewers for their suggestions and comments.

REFERENCES

1. Steven Y. Susswein, Parallel Path Consistency, Master's Thesis, University of Utah, Salt Lake City, Utah (March 1991).
2. Steven Y. Susswein, Thomas C. Henderson, Joe Zachary, Chuck Hansen, Paul Hinker, and Gary C. Marsden, Parallel Path Consistency, Technical Report TR-91-010, University of Utah, University of Utah, Department of Computer Science, Salt Lake City, Utah (August 1991).
3. S. Keretho and R. Loganantharaj, On the Parallel Complexity of Constraint Propagation Algorithms for Temporal Reasoning, Technical Report TR-91-2-4, Center for Advanced Computer Studies, University of Southwest Louisiana (1991).
4. S. Kasif, On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction networks, *Artificial Intelligence*, **45**(11):275–286 (1990).
5. A. K. Mackworth, Consistency in Networks of Relations, *Artificial Intelligence*, **8**:99–118, (1977).
6. J. Gaschnig, A Constraint Satisfaction Method for Inference Making, *Proc. 12th Annual Allerton Conf. Circuit and Systems Theory*, pp. 866–874 (1974).
7. J. Gaschnig, Performance Measurement and Analysis of Certain Search Algorithms, Technical Report CMU-CS-79-124, Carnegie-Mellon University (May 1979).
8. R. A. Hummel and S. W. Zucker, On the Foundations of Relaxation Labeling Processes, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **5**(3):267–286 (May 1983).
9. R. Haralick and L. Shapiro, The Consistent Labeling Problem, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **1**(2):173–183 (April 1979).
10. R. M. Haralick and G. Elliot, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence*, **14**:263–313 (1980).
11. J. F. Allen, Maintaining Knowledge about Temporal Intervals, *Communications of the ACM*, **26**(11):832–843 (November 1983).
12. Peter B. Ladkin and Roger D. Maddux, Parallel Path Consistency Algorithms for Constraint Satisfaction, Technical Report TR89-045, International Computer Science Institute, Berkeley, California (August 1989).
13. U. Montanari, Networks of constraints: Fundamental Properties and Applications to Picture Processing, *Information Sciences*, **7**:95–132 (1974).
14. A. K. Mackworth and E. C. Freuder, The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence*, **25**:65–74, (1985).
15. Roger Mohr and Thomas C. Henderson, Arc and Path Consistency Revisitied, *Artificial Intelligence*, **28**(2):225–233 (March 1986).
16. A. K. Samal, *Parallel Split-Level Relaxation*, PhD Thesis, University of Utah, Salt Lake City, Utah (August 1988).
17. Ashok Samal and Thomas C. Henderson, Parallel Consistent Labeling Algorithms, *IJPP*, **16**(5):341–364 (1988).
18. Thomas C. Henderson, *Discrete Relaxation Techniques*, Oxford University Press, New York (1990).

19. Ching-Chih Han and Chia-Hoang Lee, Comments on Mohr and Henderson's Path Consistency Algorithm, *Artificial Intelligence*, **36**(1):125–130 (August 1988).
20. Rina Dechter and Itay Meiri, Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems, *Proceedings of IJCAI*, pp. 271–277 (1989).
21. B. Nadel, Constraint Satisfaction Algorithms, *Computational Intelligence*, **5**(4):188–224 (November 1989).
22. Gary C. Marsden, Fouad Kiamilev, Sadik Esener, and Sing H. Lee, Highly Parallel Consistent Labeling Algorithm Suitable for Optoelectronic Implementation, *Applied Optics*, **30**(2):185–194 (1991).
23. Thinking Machines Corporation, *Model CM-2 Technical Summary* (April 1987).
24. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Massachusetts (1985).

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium