Parallel Path Consistency

Steven Y. Susswein, Thomas C. Henderson,
Joseph L. Zachary, Chuck Hansen[†],
Paul Hinker[†], Gary C. Marsden[‡]

UUCS-91-010

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

July 30, 1991

†:Los Alamos National Laboratories,Los Alamos, New Mexico

‡:Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, California

Abstract

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. We also intend to categorize the relation between graph structure and algorithm performance. 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 explored the use of an outer product computational formation of path consistency and have excellent results of its use on a Connection Machine.

1 Introduction

There is a class of problems in computer science known variously as *Consistent Labeling Problems* [3], *Satisfycing Assignment Problems* [1], *Constraint Satisfaction Problems* [7], etc. We will refer to it 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 *Consistent Labeling Problems*, along with a number of current problems in computer vision.

The basic problem can be looked at abstractly in the form of a graph, in which we have:

- A set of *nodes*, N $= \{n_1, n_2, \ldots, n_n\}$; (let $|N| = n$).
- For each node n_i a domain M_i, which is the set of acceptable labels for that node. Often all the M_i's are the same, giving $M_1 = M_2 = \ldots = M_n = M$; (let $|M| = m$).
- A set of *constraint relations* R_{ij}, $i, j = 1, n$, which define the consistent label pairs which can be assigned to nodes n_i and n_j; i.e., $R_{ij}(l_1, l_2)$ means label l_1 at node n_i with label l_2 at node n_j is a consistent labeling. Directed arcs give a visual representation of the relationships.

The problem 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.

It can be shown that CLP is NP-complete[2]. Thus, there are no known efficient solutions. However, there are a number of ways the problem can be solved, including *generate and test*, *standard backtracking*, *Waltz filtering*[15], 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 [8] has shown that the "thrashing" behavior of standard backtracking can be reduced by the incorporation of consistency algorithms (*node, arc,* and *path* consistency). Mohr and Henderson [10] have given an optimal algorithm for arc consistency and an improved algorithm for path consistency.

- In *node consistency*, we look at the label set for a single node and remove any impossible labels.
- In *arc consistency* we look at each pair of nodes and remove those labels which cannot satisfy the arc between them. For example, if we looked at nodes one and two in the above example using arc consistency we would remove the value 1 from $node_1$and the value 4 from $node_2$.

- In *path consistency* we eliminate arcs which cannot satisfy a closed path of three or more nodes. Montanari has shown that if all paths of length two are consistent, then all paths of any greater length than two are consistent; therefore, in practice, only paths of length two need be considered to ensure path consistency.

Path consistency does a much better job of filtering than arc consistency, but is also much slower (i.e., requires a lot more computation); as a result, arc consistency is currently the most widely used filtering technique.

1.1 Parallel Algorithms for AC and PC

Samal and Henderson have explored parallel versions of arc consistency[13, 12]. They showed that the worst case performance of any parallel arc consistency algorithm is $O(mn)$. This means that given a polynomial bound on the number of processors, it takes time proportional to mn to solve the problem in the worst case. Moreover, they explored the dependence of performance on graph structure.

We are interested in providing a similar analysis for parallel path consistency algorithms. We conjecture that the average case time complexity of parallel path consistency is $O(mn)$. 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 mn. In fact, our preliminary results indicate that the innermost loops of path consistency (i.e., those which update the relations) run on the average in constant time, $O(1)$. However, Ladkin et al.[6] have provided a problem that exhibits worse case iteration complexity, $O(m^2n^2)$ complexity.

2 Parallel Path Consistency

The current best path consistency algorithm (PC-3) has a time complexity of $O(n^3m^3)$, compared to the optimal arc consistency algorithm (AC-4) which has a time complexity of $O(n^2m^2)$[4], but 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 main thrust of this research is to define and implement *parallel* versions of the PC algorithms on a multiprocessor to see whether they can outperform the best AC algorithms when used within search to prune the search tree at each node.

2.1 Standalone Parallel PC

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

2.2 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.

2.3 Finding Worst Case Performance

Although theoretical worst case performance of sequential PC-1 is of complexity $O(m^5n^5)$, 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. N-queens and confused n-queens[11] are the standard test cases for performance measurement and comparison.

3 Initial Results

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.*
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.

3.1 Pruning Efficiency of PC vs. AC

We already had a working version of arc consistency created by Samal, so PC-1 was coded based on the algorithm given by Mackworth. Both these programs use identical system calls

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

Note: 1s is sequential code and 1p is parallel code

Table 1: Speedup Linearity for 16-Queens using PC-1

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 did a superior job of pruning the search space. As mentioned earlier, a good example of this is consistent 4-Queens. Figure 1 shows number of nodes expanded for n-queens ($n = 4, 6, 8, 10$).

3.2 Parallel PC-1

As a next step, we modified the PC-1 program mentioned above 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 1). 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.

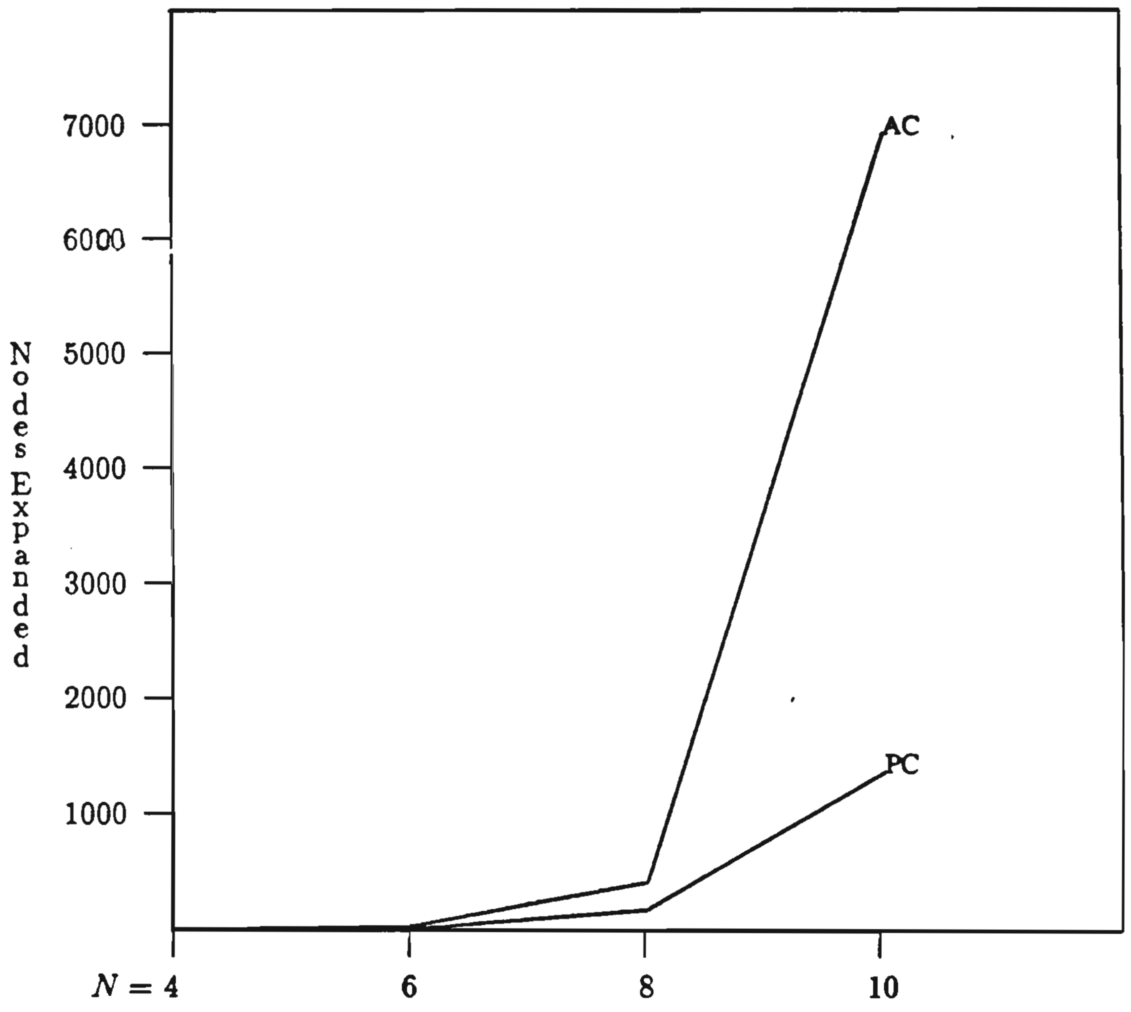


Figure 1: Number of Nodes Expanded in N-Queens for AC and PC

3.3 Worst Case Performance

The graph input to PC-1 is encoded in the form of an $nm*nm$ binary matrix. The algorithm iterates over this matrix until two successive iterations yield no change in the matrix. Each iteration is of complexity $O(m^3n^3)$ 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 m^2n^2 iterations, yielding a worst case performance of $O(m^5n^5)$.

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 most possible relation constraints for small values of m and n 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 $m = 2$ and $n = 3$ yielded a worst case performance of 5 iterations (compared to a theoretical worst case of 30 iterations) and an average case performance of 2.07 iterations (see Figure 2).

Additional experiments varying the value of m and n 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, but the iterations themselves are performed in sequence. The number of iterations required (whose upper bound is theoretically m^2n^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 m and n, then parallel path consistency may prove to be a superior filtering technique.

4 Tools and Facilities

All the code is being 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).

4.1 DECStation 3100

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

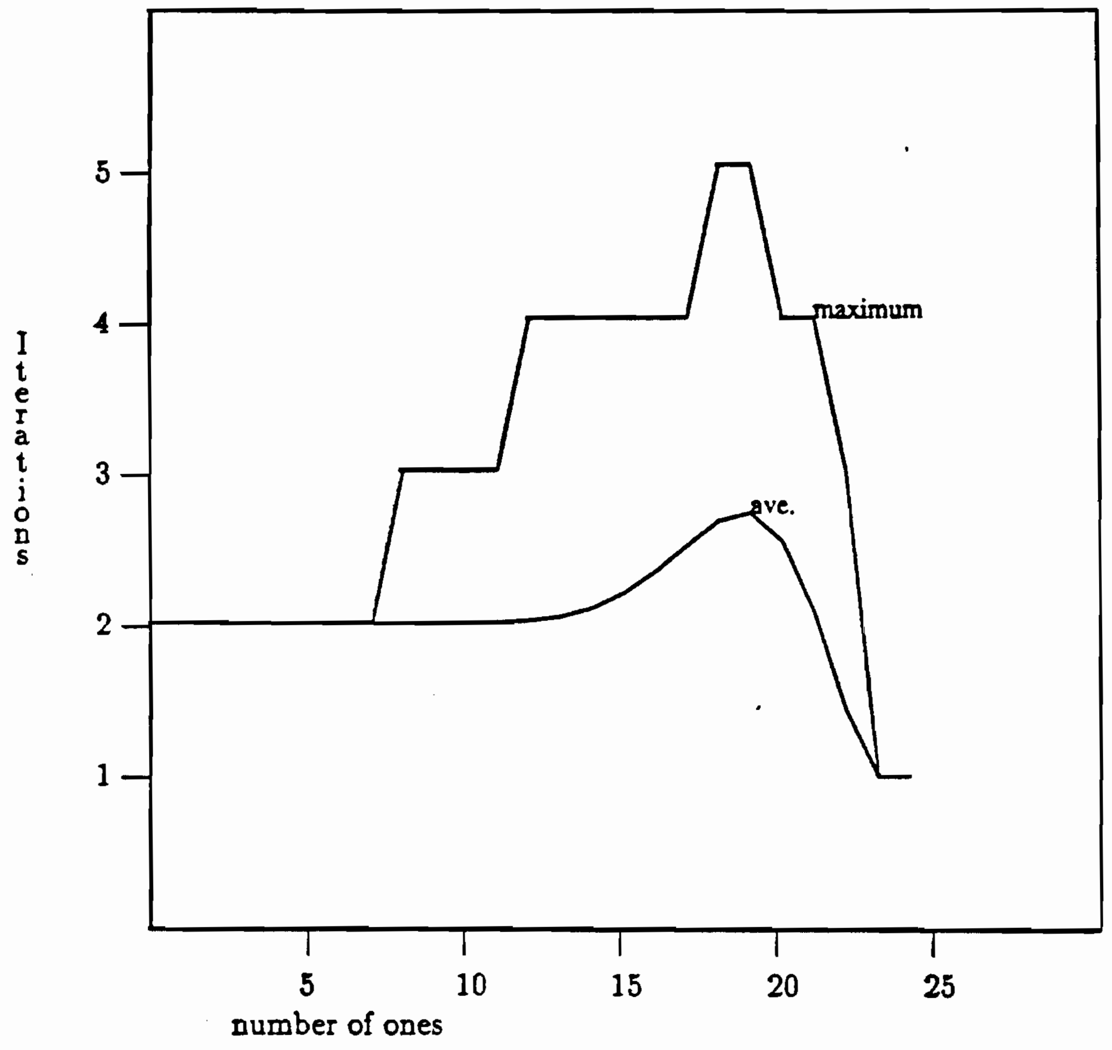


Figure 2: Avg. and Max. Iterations for all relations of $m = 3$ and $n = 2$.

4.2 Butterfly GP1000

Parallel code i=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 allow 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.

The Butterfly is configured with eighteen nodes, which is sufficient for development and testing and to show the effect of parallelized PC. 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 m (ranging from 2 to 10). This shows that the average number of iterations is constant (about 2), and the worst case is linear in nm. Figure 4 shows the speedup linearity per iteration. Finally, Figure 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, since almost all the search time is spent in PC.

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.[9]. 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. 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. also show how the outer product computation can be used to compute k-consistency for any k.)

Recall that there are mn rows in the relation matrix, corresponding to n nodes with m 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 matrix, which stores the running sum of these outer products. After iterating on all labels for a given node (m 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 6 and 7 show a visual representation of this algorithm (adapted from[9]).

Computing a vector outer product from two vectors of length mn has a complexity of

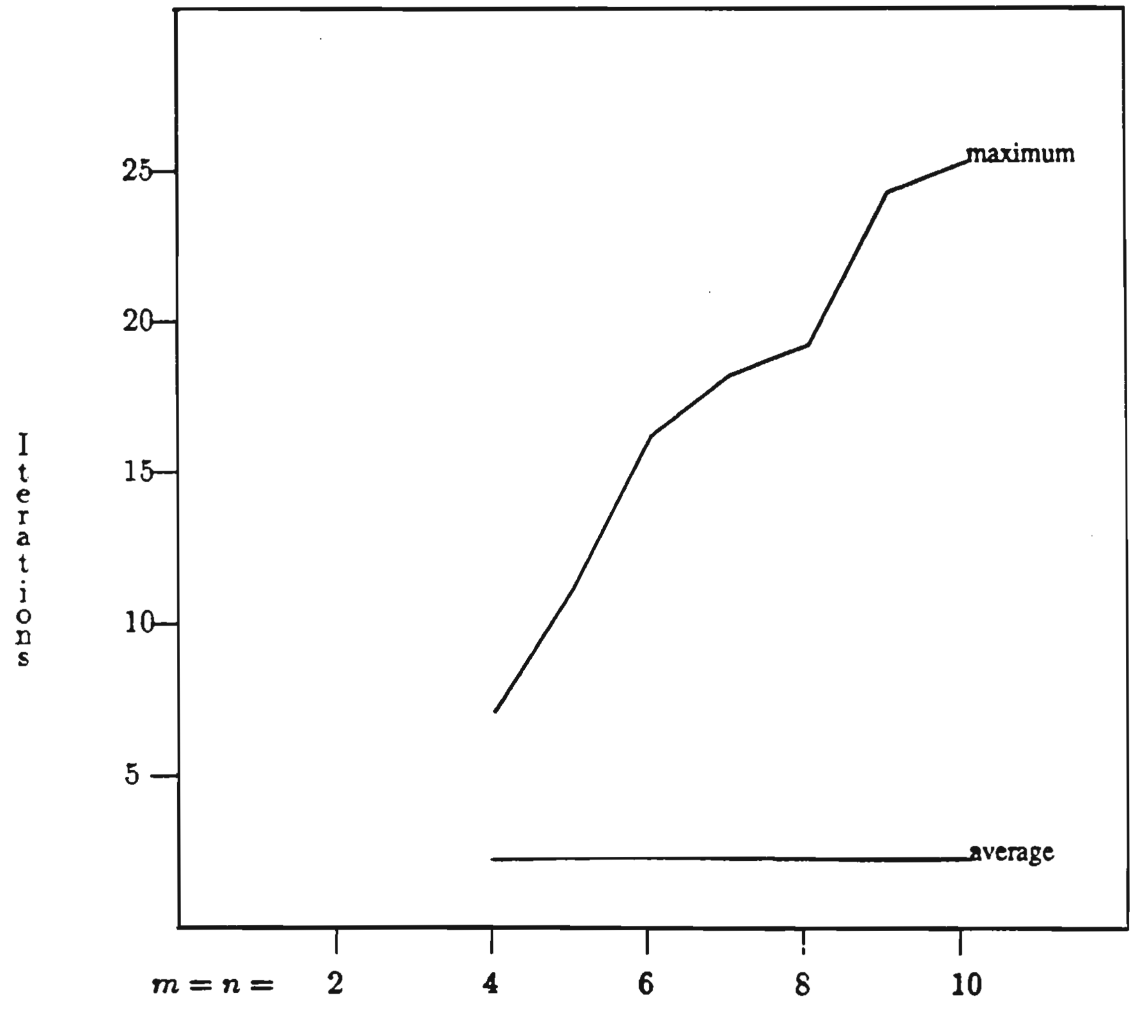


Figure 3: Avg. and Max. Iterations for $m = n = 2, \ldots, 10$.

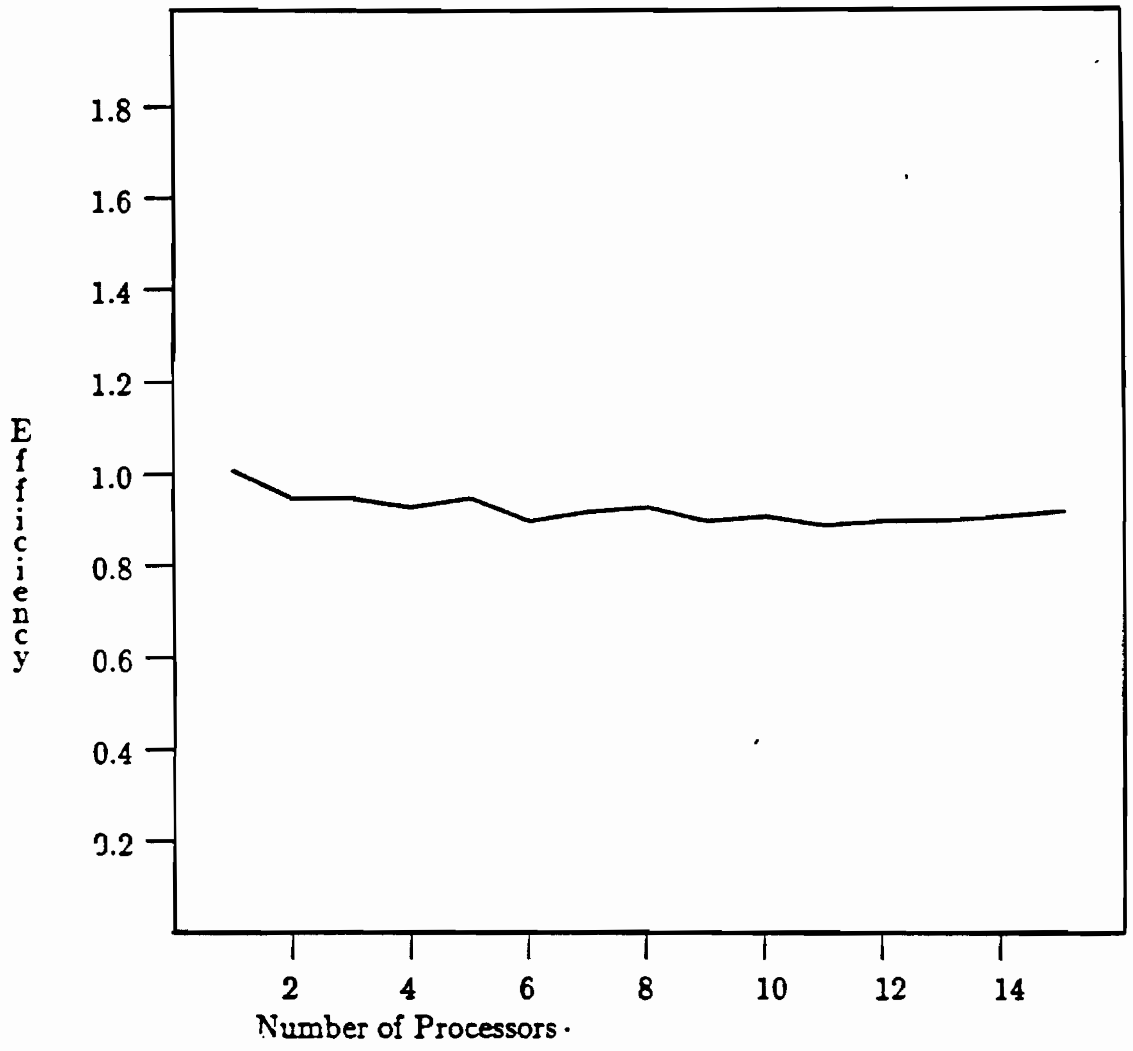


Figure 4: Speedup Linearity per Iteration.

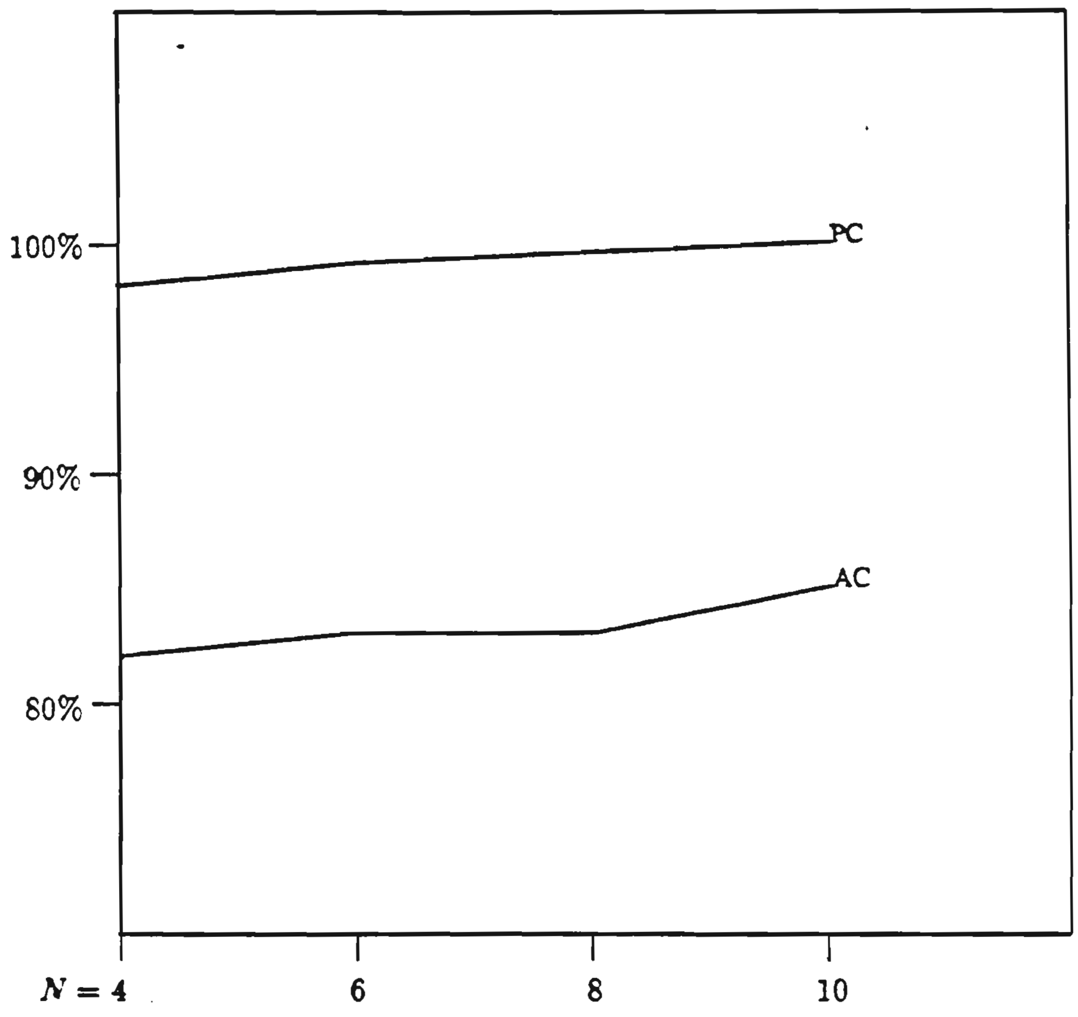


Figure 5: Percentage of Execution Time Spent in Filter Code in N-Queens Search.

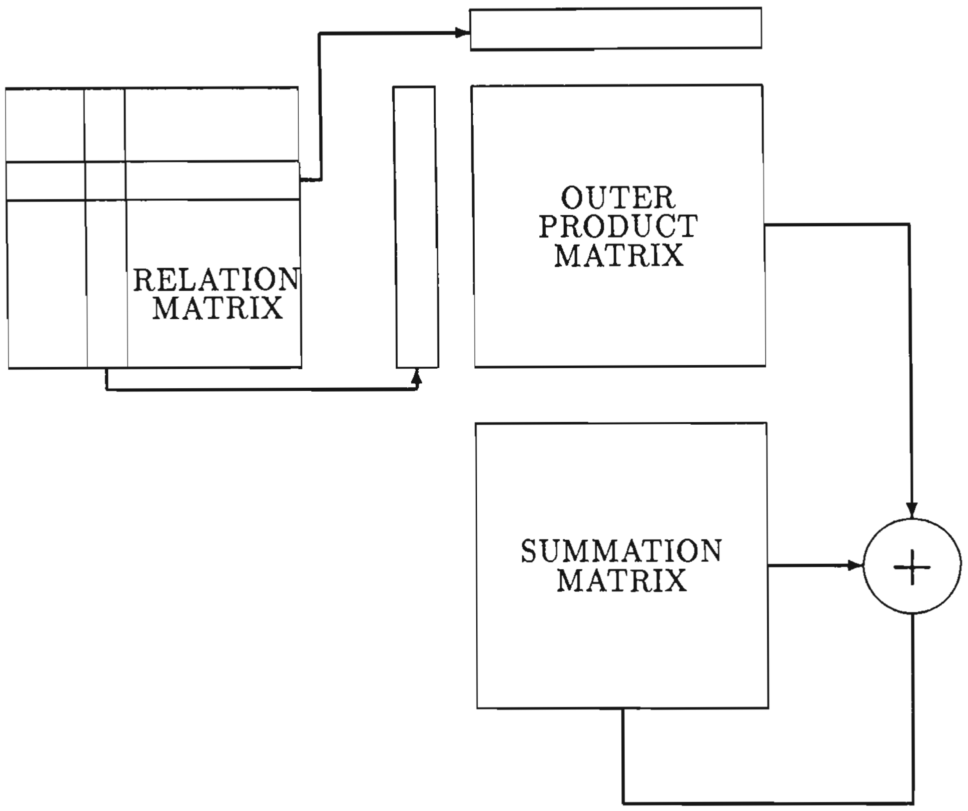


Figure 6: Outer product PC: computation of summation matrix

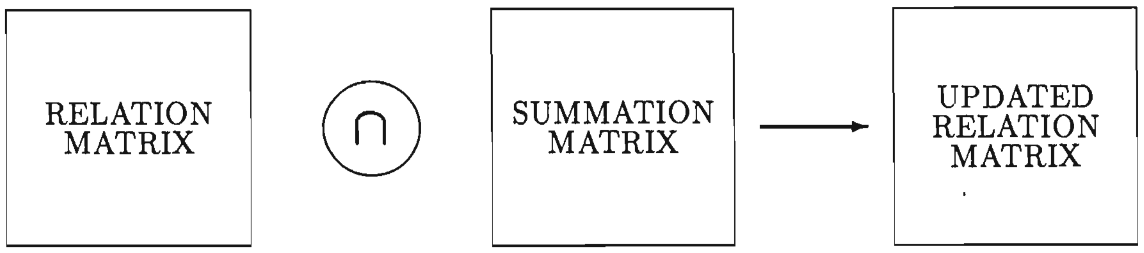


Figure 7: Outer product PC: computation of updated relation matrix

$O(m^2n^2)$. This outer product is computed for each row in the relation matrix (mn times), giving a total complexity of $O(m^3n^3)$ 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 is used to model and implement parallel outer product PC (POP PC).

5.1 The Connection Machine

We implemented path consistency on the CM-2 using a modified outer product algorithm described by Marsden et al. [9]. The Connection Machine 2 (CM-2) is a 64K processor massively parallel device. 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 can not 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 m labels, we only need a binary matrix of size n^2m^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.

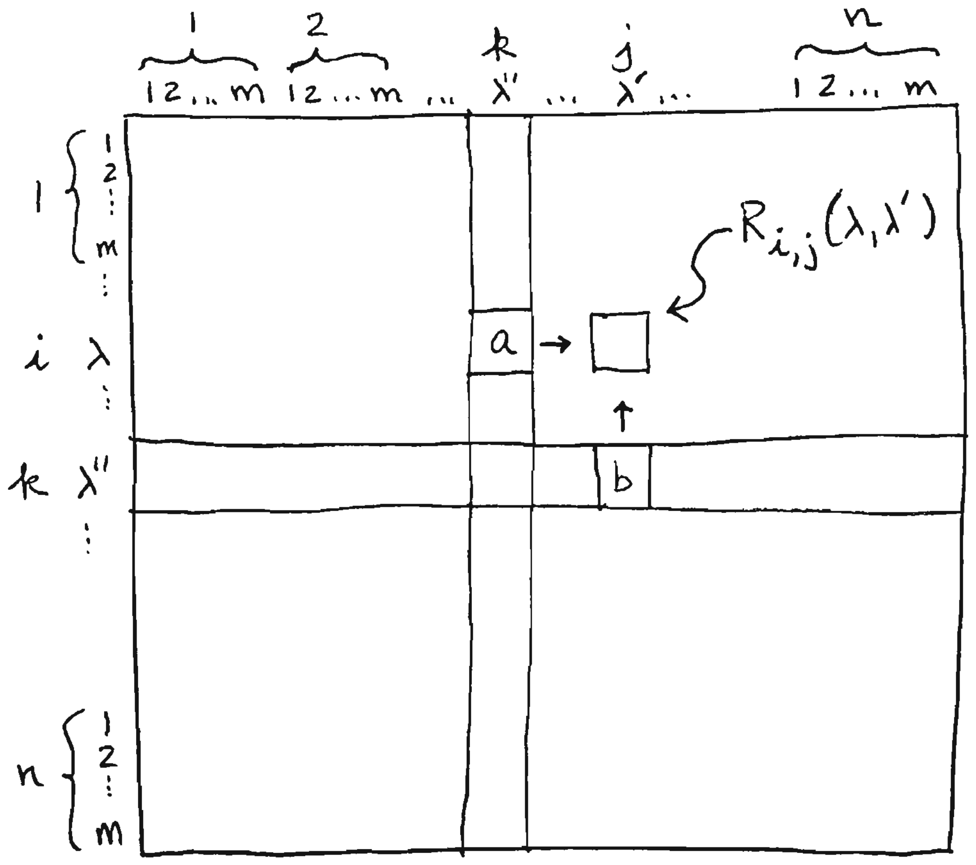
The first phase of setting up the initial relation matrix would be typically down-loaded from the front-end. For the timing results given here, we generated the initial relation matrix on the CM-2. Another method for loading the CM-2 with the proper initial relation matrix is to utilize the CM-2 file server and load the file directly from the Data Vault[1] or through the Data Vault's front-end via ethernet. Once the initial relation matrix is in physical memory on the CM-2, we are ready to proceed with the path consistency algorithm.

Based on this machine specification, the following relatively simple parallel algorithm was developed, called POP PC.

The out product computation at each $R_{ij}(\lambda, \lambda\prime)$ element is:

$$R_{ij}(\lambda, \lambda') = R_{ij}(\lambda, \lambda') \wedge \prod_{k} \sum_{\lambda''} [R_{ik}(\lambda, \lambda'') \wedge R_{kj}(\lambda'', \lambda')]$$

Let $a = R_{ik}(\lambda, \lambda'')$ and $b = R_{kj}(\lambda'', \lambda')$, then the following figure shows the contribution of (k, λ'') to the outer product at $R_{ij}(\lambda, \lambda')$:



[1]a massive storage device which is directly attached to the CM-IO bus

Thus, to parallelize the algorithm, each CM processor must get 1 bit from the complete R matrix, 1 summation bit, and finally, so as to go as fast as possible, the row and column which are used to compute the R bit. This leads to the following algorithm:

```
POP PC:

  while (change){
    for (node=1; node<n; node++){
      for (label=1; label<m; label++){
        index = offset(node,label);
        S = S | (Row[index] & Col[index]);  /* computes sum over lambda prime */
      }
      R = R & S;  /* computes product over k */
    }
  }
```

The algorithm uses a modified version of the outer product algorithm described by Marsden. 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 col 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 ANDing 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 which the closest power of two which was greater than or equal to n^2m^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)[2] The VP ratio is the ratio of the number of virtual processors to physical processors. Since we might have allocated some extra VPs if our relation matrix was not a power of two, we masked out those processors such that they don't perform any 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

[2]If we were restricted to physical processors, we couldn't process a relation matrix greater than 128x128 on the 16K CM-2.

connectivity. Since we are dealing with 2D bitmaps, we simply use a 2D non-toriodal 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 Olog(n) where n is the number of VPs. 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 matrix. We implemented this by forming one temporary matrix with a copy_spread command which takes a column (n-vector) and forms an $n \times n$ matrix by replicating the column. We generated a second temporary matrix by using the copy_spread along the row axis which forms an $n \times n$ matrix which replicated the row. If these two matrixes are logically ANDed 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 ANDing 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 timing below indicate, we found linear speed up with respect to the number of physical processors. As we increased the VP ratio, we notice 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.

Problem Size	Processors	SpeedUp	VP-Ratio	Elapsed Time(sec)
256x256	16384	1	4	0.25029
484x484	16384	1	16	0.739879
1024x1024	16384	1	64	4.022633
2025x2025	16384	1	256	26.220621
4096x4096	16384	1	1024	204.557243
256x256	32768	1	2	0.260105
484x484	32768	1.33	8	0.555307
1024x1024	32768	1.66	32	2.429805
2025x2025	32768	1.84	128	14.287050
4096x4096	32768	2.00	512	102.245058
256x256	65536	1	1	0.216853
484x484	65536	1.52	4	0.488031
1024x1024	65536	2.52	16	1.596955
2025x2025	65536	3.24	64	8.090925
4096x4096	65536	3.80	256	53.884622

The connection machine is massively parallel, fine-grained SIMD processor [14]. It provides a large number of tiny processor/memory cells connected by a programmable communications network. The CM-2, for example, has 65,536 processor nodes, each with 8K bytes of local memory. The connection machine is easily scalable, and this is considered a small 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 hold 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 [5]. 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.

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= nm (i.e., the length of 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 that you must have a integral vp ratio. So, the smallest problem that can be run on a 16K machine is 128x128 and the smallest problem that can be run on the whole machine is 256x256. Since the N-queens problem causes the shape of the problem space to be nmxnm, 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 and in solving temporal logic problems. We have described several methods here which should be of great value to this aim. However, their useful application requires further study on a wider class of problems and graph geometries.

References

[1] John Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, May 1979.

[2] R. Haralick, L. Davis, A. Rosenfeld, and D. Milgram. Reduction operations for constraint satisfaction. *Information Sciences*, 14:199–219, 1978.

[3] R. Haralick and L. Shapiro. The consistent labeling problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):173–183, April 1979.

[4] T. Henderson. *Discrete Relaxation Techniques.* Oxford University Press, New York, 1990.

[5] D. Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachusetts, 1985.

[6] Peter B. Ladkin and Roger D. Maddux. Parallel path consistency algorithms for constraint satisfaction. Technical Report TR-89-045, International Computer Science Institute, ICSI, 1947 Center Street, Suite 600, Berkeley, CA 94704-1105, August 1989.

[7] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[8] 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.

[9] 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, January 1991.

[10] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisitied. *Artificial Intelligence*, 28(2):225–233, March 1986.

[11] B. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5(4):188–224, November 1989.

[12] A. Samal and Thomas C. Henderson. Parallel consistent labeling algorithms. *International Journal of Parallel Programming*, 16(5):341–364, 1988.

[13] A.K. Samal. *Parallel Split-Level Relaxation.* PhD thesis, University of Utah, Salt Lake City, Utah, August 1988.

[14] Thinking Machines Corporation. *Model CM-2 Technical Summary*, April 1987.

[15] D. Waltz. Understanding line drawings of scenes with shadows. In ed. P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91, New York, 1975. McGraw-Hill.