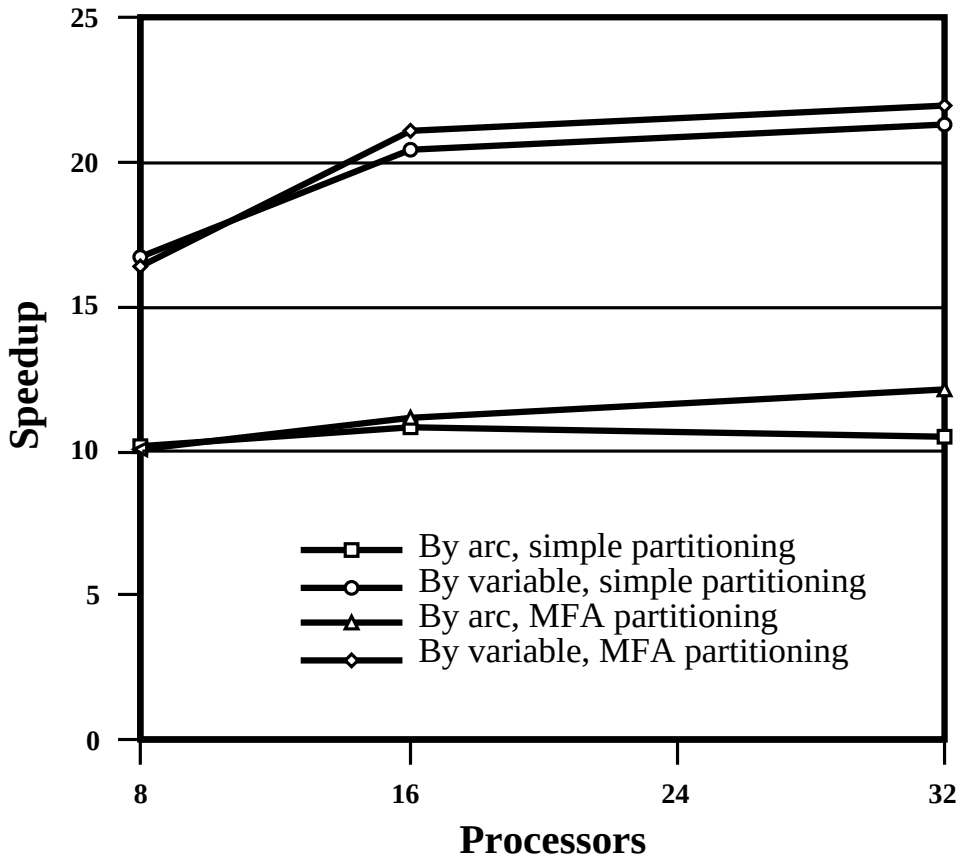

Figure 7.4: Speedup of DSPAC-1 versus AC-1 on a Distributed Memory Machine

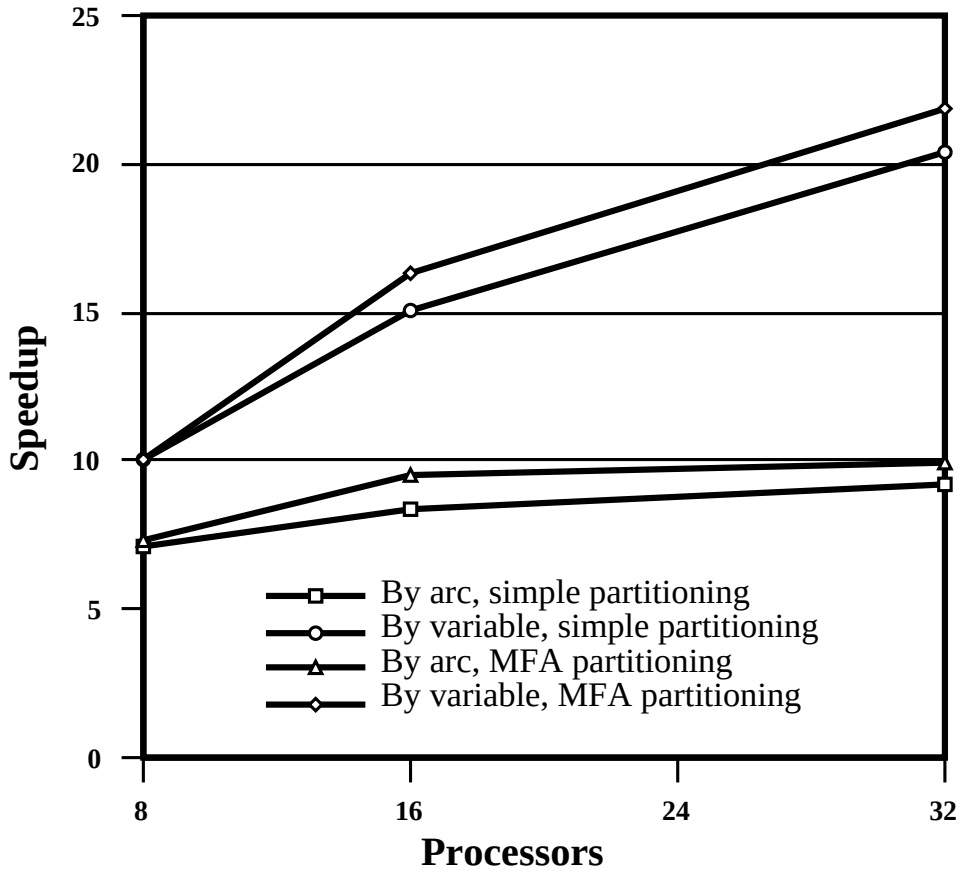

Figure 7.5: Speedup of DSPAC-2 versus AC-1 on a Distributed Memory Machine

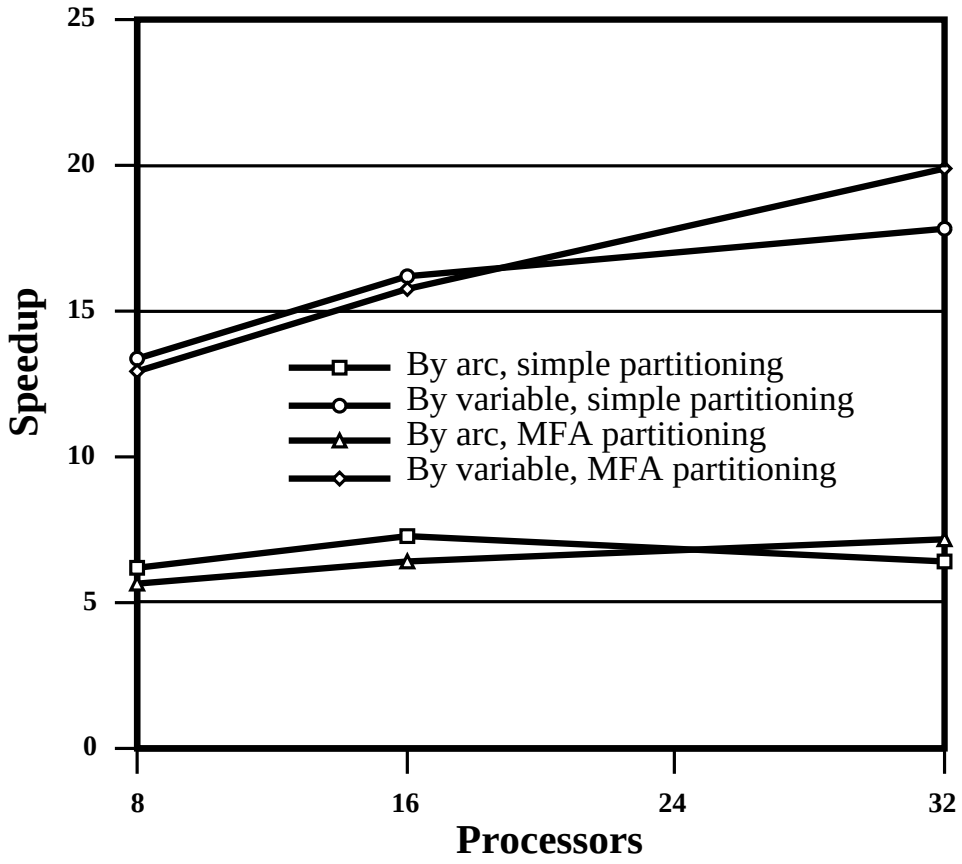

Figure 7.6: Speedup of DSPAC-3 versus AC-1 on a Distributed Memory Machine

The results of our measurements suggest that mean field annealing graph partitioning can moderately improve speedup for large multiprocessor configurations for some DSPAC algorithms.

We notice little or no difference in speedup between simple and MFA load balancing with the DSPAC-1 algorithm. This is as expected, since DSPAC-1 is synchronous with respect to starting consistency checks. The only exception is with the arc-based algorithms on thirty-two processors, where we see a 16% improvement using MFA load balancing. We also notice that the algorithm with less frequent communications performs much better.

With DSPAC-2, we notice very little effect of load balancing for eight processors, but an increase in speedup of 11% and 7% is observed for sixteen and thirty-two processor, respectively, using MFA load balancing. Again, our results also show that the algorithm with less frequent communications performs much better.

DSPAC-3 experimentation shows characteristics different from the other two DSPAC algorithms. We notice little or no effect of load balancing with the DSPAC-3 algorithm on eight and sixteen processors using the variable-based frequency implementation. The arc-based frequency implementations for eight and sixteen processors actually show that simple partitioning is better than MFA by about 12%. A closer examination of experimental data shows that some processors perform much more work than others. We suspect that MFA partitioning is not as effective for load balancing for DSPAC-3 since the processes of adding arcs to the queues are not reflected in the weights of the graph nodes.

The thirty-two processor trials show a 10% speedup of MFA over simple load balancing. Again, our results also show that the algorithm with less frequent communica-

tions performs much better.

Our results also illustrate a phenomenon called *super-linear speedup*, where the reported speedup is actually greater than the number of processors in the multiprocessor configuration [1]. This observation was also reported by Janakiram, et. al. [31], in their work with parallel backtracking algorithms, and Samal [55] in his work with parallel arc consistency algorithms on shared memory computers.

Our super-linear speedup is the result of two situations. First, we have mentioned that the DSPAC algorithms remove domain values and make these values available quickly to other processors. The other processors, then, perform the reduced work much sooner because the number of values it has to check decreases rapidly. The second reason for the speedup is attributed to the amount of work the sequential algorithm performs in each pass. We described in Section 2.2 how AC-1 and AC-3 work, and in Sections 3.5 and 5.2.2 how the SPAC and DSPAC algorithms work compared to AC-1 and AC-3. Due to the distributed nature of the DSPAC algorithms, they perform less work, as each processor has a smaller number of domain values to check whenever an arc is checked.

We did not include the time required to partition the graphs in our speedup computation. Our simple load balancing took an average of 0.4 seconds on a DecStation 5000. Mean field annealing took an average of 22, 177, and 378 seconds for dividing each network into eight, sixteen, and thirty-two partitions, respectively. It is obvious that if the problem is to be executed only once, simple load balancing is more effective. If hundreds or thousands of runs are required for the same constraint network topology, then mean field annealing is appealing.

7.6 AC-1 and AC-3 Revisited

Although Mackworth and Freuder state that AC-3 is a more efficient algorithm, our experience is just the opposite. AC-3 requires 31% more time to finish as opposed to AC-1. We mentioned earlier that complexity analysis of AC-3 completely ignores operations of the queue. The work required to first build the queue, and then check the queue when adding arcs, is not trivial.

Actual measurement of the queue maintenance time is not available due to difficulty in measuring such a small-grained operation. We were able to measure the time spend building the queues. For the networks described in Section 7.4, an average of 15,750 arcs are initially put in the queue. That means we need a queue of at least 16,000 cells to handle our AC-3 problem. Our queue building time includes these operations:

- Queue index initialization
- Queue initialization
- Queue building - putting arcs in the queue

As one would expect, one can greatly affect execution time by wisely coding the algorithm. We quickly found that an unwise implementation of the queue is to have one queue for all variables, with one queue head. To add an arc to the queue, the AC-3 algorithm has to check **all** of the arcs to ensure the arc is not currently on the queue.

To speed up the queue reference time, we maintain a separate queue for each of the variables of the constraint network. That way, if $arc(l, i)$ needs to be added to the queue, only the queue with arcs originating at variable l are examined. We figure that this implementation saves one or two orders of magnitude of work.

When first building the queue, we keep a reference to the end of the queue so that we can quickly add arcs without going from the head to the tail. Queue building time for AC-3, however, still averaged 16.015 seconds.

Even though we code the algorithms efficiently, AC-3 and DSPAC-3 do not perform as well as the AC-1-based algorithms. In fact, we are surprised that DSPAC-1 performs slightly better than DSPAC-2. This is because of the update timing in DSPAC-1 and the host control of the loops. In the examples of Section 4.3, all processors running SPAC-1 checks each of its arcs four times. All processors running SPAC-2 checks each of its arcs only three times. With our DSPAC implementation, we noticed that one or more processors running DSPAC-2 checks all of their arcs one additional time than the DSPAC-1 implementation. This may be a characteristic of the constraint networks.

7.7 The Effect of Arc Consistency on Search

Arc consistency algorithms usually do not find a solution to a CSP [40, 44], so some search algorithm must be employed to find a solution. Several sequential paradigms exist which rely on varying degrees of assignment, search, and consistency. These paradigms are reviewed by Kumar [40] and Nadel [49]. Nadel also empirically compares the performance of these algorithms. Each of these algorithms employ a greater degree of arc consistency at the node of the search tree:

- Generate and Test
- Simple Backtracking
- Forward Checking
- Partial Lookahead
- Full Lookahead

- Really Full Lookahead

Several sequential search algorithms have been adapted for parallel processors. These include backtracking [16, 31, 39] and Forward Checking [51, 52].

No one, however, has made empirical measurements showing the effect of applying arc consistency algorithms to a CSP before search. Nadel's performance comparisons were only of the search algorithms with consistency imbedded, rather that consistency followed by search. We examine this effect, although not rigorously. Since this dissertation is on arc consistency and not search, our results should only be used as an approximation of the time saved.

We gather time measurements of identical constraint networks with the following arc consistency and search combinations:

- Sequential AC-1, followed by sequential depth-first backtracking [2, 38]
- Only sequential depth-first backtracking tree search

We also approximate the performance of a parallel depth-first search by assuming a maximum speedup of 12 using 16 processors (75% of possible speedup). This assumption reflects measurements gathered by Janakiram [30]. Each specific constraint network is represented by a different predicted speedup. If the solution is found at the beginning of depth-first tree search, no speedup due to parallelism is realized. If the solution is in the middle of search, we assume speedup is half of the maximum speedup. If no solution exists, tree search would check all possible assignments, and maximum speedup is possible. These assumptions also allows us to predict execution time of parallel depth-first search using results of sequential search of the following combinations:

Table 7.2: Examples of Constraint Network Characteristics for Tree Search

Satisfiability = 0.30

Network	Variables	Min./Max. Connect.	Avg. Conn.	Total Edges	Domain Size	% Values Removed	Solution Location
P020C10A	20	3/5	3.6	36	10	35.0	none
P020C10B	20	3/5	3.5	35	10	37.5	middle
P020C10C	20	3/5	3.6	36	10	32.5	begin.
P020C10D	20	3/5	3.5	35	10	36.0	begin.
P020C10E	20	3/5	3.6	36	10	39.0	none

- Only *predicted* parallel depth-first backtracking tree search
- DSPAC-1, followed by *predicted* parallel depth-first backtracking (16 processors)

We execute the AC-1 and tree search algorithms using the above consistency/search configurations on a DecStation 5000. We use five constraint networks described in Table 7.2 as our data. This table also contains information as to the location of the first solution. The search continues until one solution is found, or all possibilities are examined and no solution is found. Actual time measurements and predicted times are shown in Figure 7.7. Since the time needed for AC is small, we assume AC and SPAC take the same amount of time.

The results show that, for our sample networks, arc consistency can significantly reduce the search time needed to find at least one solution. Sequential arc consistency reduces the sequential search time to 3.2% of the time needed when arc consistency is not employed. Also based on our parallel search speedup assumption, parallel arc consistency reduces the parallel search time to 3.7% of the time needed when parallel arc consistency is not employed. Expected speedup due to parallelism is 7.5% for SPAC and parallel tree search (compared with AC and sequential tree search) and 8.5% for only parallel tree search (compared with only sequential tree search). Table 7.3 shows execution times and work performed.

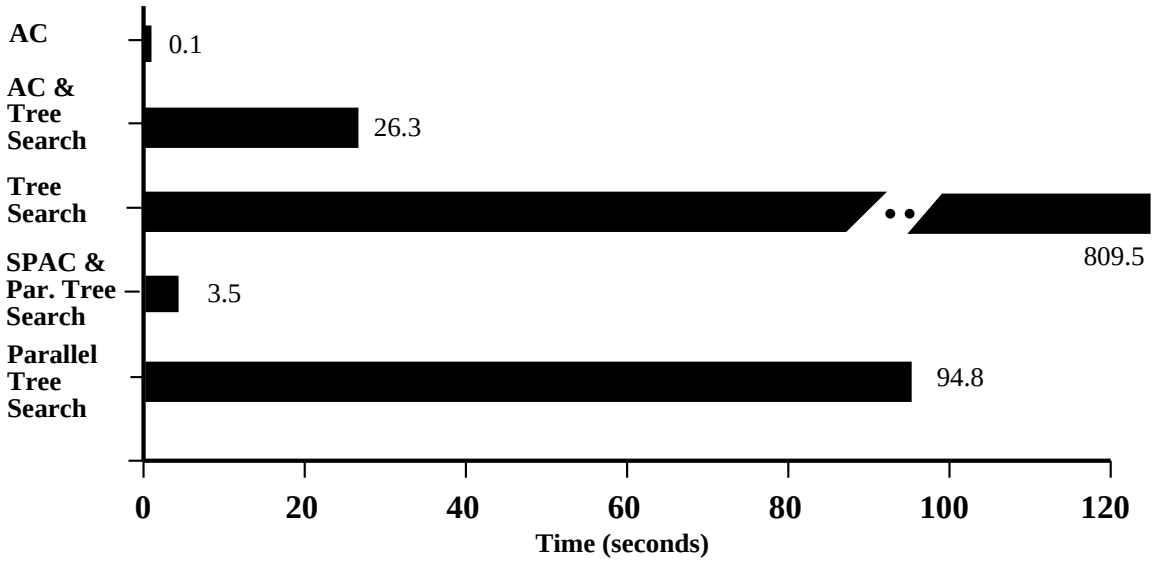

Figure 7.7: The Effect of Arc Consistency on Search Time

These measurements only examine measurements for simple depth-first backtrack search, and not a more efficient forward checking search. A more thorough examination should be made, and is one of our future plans.

Table 7.3: AC and Tree Search Execution Times and Work

All time measurements are in seconds.

Network	Time AC	Time AC&TS	Time TS	Pred.Time SPAC&ParTS	Pred.Time ParTS	Con.Chks. AC	Con.Chks. AC&TS	Con.Chks. TS
P020C20A	0.1	82.8	2407.6	6.9	200.6	4155	15430696	534042827
P020C20B	0.1	34.1	1117.1	5.7	186.2	5144	7006500	243687039
P020C20C	0.1	3.5	23.7	3.5	23.7	4104	773420	56031060
P020C20D	0.1	0.5	24.0	0.5	24.0	4896	108572	5347005
P020C20E	0.1	10.5	475.3	0.9	39.6	5792	2161378	110990548
Average	0.1	26.3	809.5	3.5	94.8	4818	5096113	190019696
Speedup				7.5	8.5			

Chapter 8

SPACs Applied to MCM Layout

In this chapter, we consider a possible application of SPAC algorithms for determining the interconnection layout of a multi-chip module (MCM). A MCM is a high performance packaging for the interconnection of several hundred chips. The dense packaging results in a smaller delay between chips than the conventional printed circuit board approach. In MCMs, the chips are placed on the top layer. Below the top layer, there is a stack of signal distribution layers for interconnecting the pins of the chips. The signal distribution layers are usually paired similarly as the Manhattan x-y planar model [1].

The design of a MCM includes three steps: placement of chips, layer assignment of nets, and routing of interconnects [25]. We focus on the steps of layer minimization and assignment, and routing of interconnects. It is important to minimize the number of layers, since the cost of fabricating and cooling a MCM are directly related to the number of layers.

Several heuristic algorithms [25] are proposed to solve the layer assignment of MCM layout, yet the previous approaches do not ensure the number of required layers is minimal.

Layer minimization and interconnection assignment for a MCM layout can be trans-

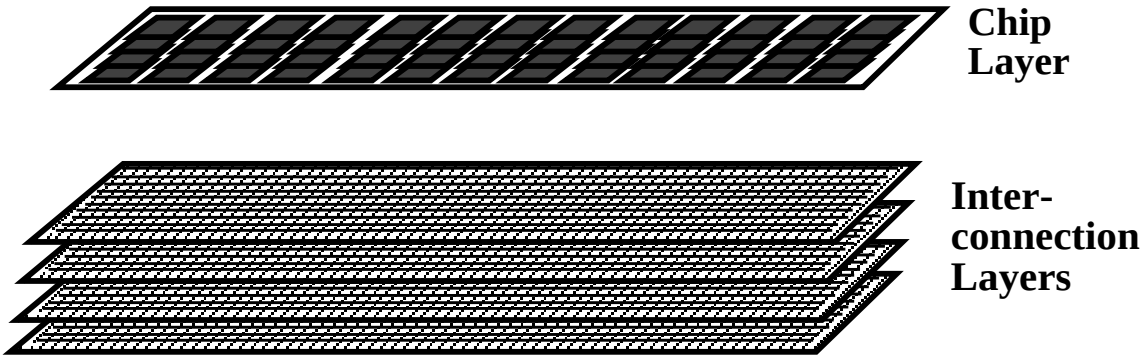

Figure 8.1: Example of Layers of our Multi-chip Module Package

lated into a constraint satisfaction problem. Constraints are used to ensure that nets assigned to the same layer do not cross each other. One major advantage of this translation is that we can utilize our parallel consistency algorithms to reduce the search space. Since the size of the domain of each variable needs to be fixed, we assume some value for the number of layers in the beginning. After iterative executions of the constraint satisfaction, we find the minimal number of the layers we need for a given interconnection pattern.

8.1 Definition of MCM Terms and Value

The following terms are used to describe our MCM package and interconnection assignment process, and are also defined in [25] and [48].

Definition 8.1 *A* layer *is a single piece of substrate used either for the chips or for the interconnections between chips of our system. We stack one chip layer and l interconnection layers to make our MCM package, as shown in Figure 8.1.*