6.3 Results for SPAC-3 Algorithms

6.3.1 Shared Memory Machine

A comparison of processor cycles for the sequential AC-3 and SPAC-3 algorithms on the simulator and Sequent Symmetry is shown in Table 6.8.

Simulation of SPAC-3 for these networks on a four-processor computer predicts speedup of between 79 and 93% of linear speedup (efficiency), and actual computer execution indicates speedup within 4 to 6% of the simulated speedup (Figure 6.7). Simulation of a ten-processor computer predicts a speedup of between 65 and 82% of linear speedup, but actual runs show that the prediction is off by 8 to 12%. Simulation of a twenty-processor computer predicts a speedup of between 56 and 73% of linear speedup, but actual runs show that the prediction is off by 5 to 11%.

Utilization and redundancy figures display similar trends. Utilization of SPAC-3 simulation is about 92, 87, and 88% for four, ten, and twenty processors, respectively. Utilization of SPAC-3 actual measurements is about 87, 72, and 74% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the SPAC-3 simulation is 2 to 63% higher than AC-3. The redundancy of work performed by the SPAC-3 on the Sequent Symmetry is 3 to 47% higher than AC-3.

Table 6.8: Processing Cycles of AC-3 and SPAC-3 Algorithms for a Shared Memory Machine

Measurement	Algorithm	20 Vars	40 Vars	60 Vars
Sequential	AC-3	1490	3181	4905
Simulation, 4 proc	SPAC-3	474	879	1326
Symmetry, 4 proc	SPAC-3	502	944	1403
Simulation, 10 proc	SPAC-3	228	446	602
Symmetry, 10 proc	SPAC-3	279	511	663
Simulation, 20 proc	SPAC-3	129	233	337
Symmetry, 20 proc	SPAC-3	143	275	387

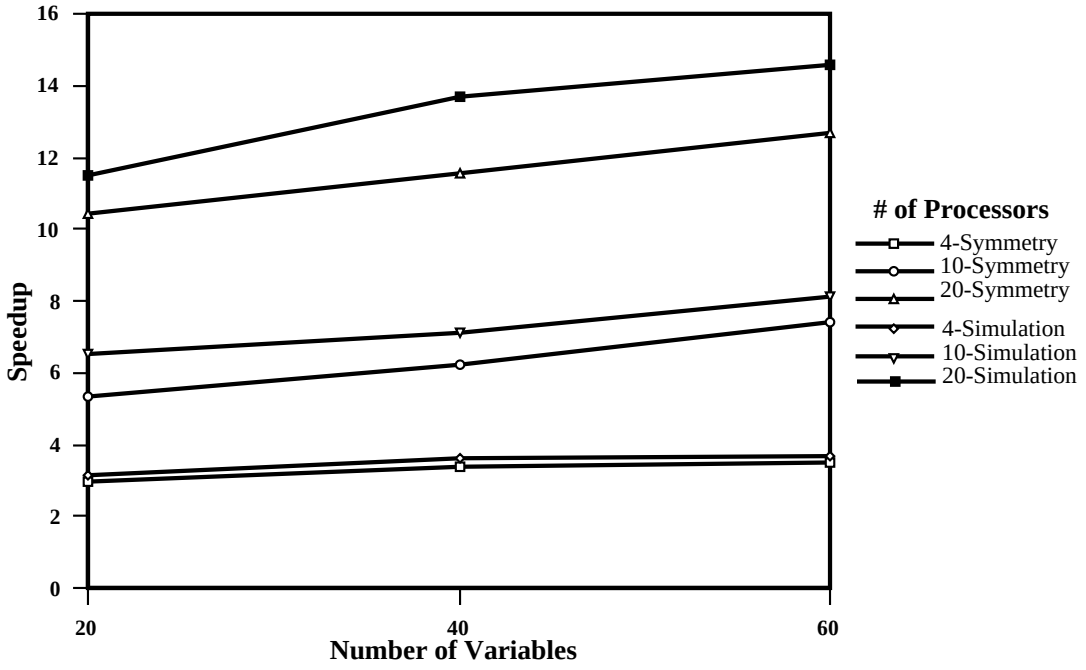

Figure 6.7: Speedup of SPAC-3 over AC-3 for a Shared Memory Machine

6.3.2 Distributed Memory Machine

A comparison of processor cycles for the sequential AC-3 and DSPAC-3 algorithms on the simulator and Intel iPSC/2 is shown in Table 6.9. The "Frequency" column represents the specific frequency of variable updates, as discussed in Section 5.1.3. The simulation data below shows processor cycles for these frequencies. Only the arc-based message frequency was implemented on the iPSC/2.

Simulation of DSPAC-3 for these networks on a four-processor computer predicts speedup of between 69 and 83% of linear speedup (efficiency), and actual computer execution indicates speedup within 1 to 2% of the simulated speedup (Figure 6.8). Simulation of a ten-processor computer predicts a speedup of between 51 and 71% of linear speedup, but actual runs show that the prediction is off by 3 to 5%. Simulation of a twenty-processor computer predicts a speedup of between 40 and 61% of linear speedup, but actual runs show that the prediction is off by 5 to 6%.

Utilization and redundancy figures display similar trends. Utilization of DSPAC-3 simulation is about 83, 74, and 76% for four, ten, and twenty processors, respectively. Utilization of DSPAC-3 actual measurements is about 84, 70, and 70% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the DSPAC-3 simulation is 3 to 98% higher than AC-3. The redundancy of work performed by the DSPAC-3 on the iPSC/2 is 3 to 108% higher than AC-3.

Table 6.9: Processing Cycles of AC-3 and DSPAC-3 Algorithms for a Distributed Memory Machine

Measurement	Algorithm	Frequency	20 Vars	40 Vars	60 Vars
Sequential	AC-3		1490	3181	4905
Simulation, 4 proc	DSPAC-3	Value	598	1118	1669
Simulation, 4 proc	DSPAC-3	Arc	541	990	1482
Simulation, 4 proc	DSPAC-3	Variable	546	991	1430
Simulation, 4 proc	DSPAC-3	Processor	671	1437	2140
iPSC/2, 4 proc	DSPAC-3	Arc	524	982	1454
Simulation, 10 proc	DSPAC-3	Value	318	597	793
Simulation, 10 proc	DSPAC-3	Arc	294	532	691
Simulation, 10 proc	DSPAC-3	Variable	342	544	695
Simulation, 10 proc	DSPAC-3	Processor	360	719	977
iPSC/2, 10 proc	DSPAC-3	Arc	326	562	718
Simulation, 20 proc	DSPAC-3	Value	175	313	452
Simulation, 20 proc	DSPAC-3	Arc	184	294	401
Simulation, 20 proc	DSPAC-3	Variable	245	348	430
Simulation, 20 proc	DSPAC-3	Processor	245	411	550
iPSC/2, 20 proc	DSPAC-3	Arc	214	330	438

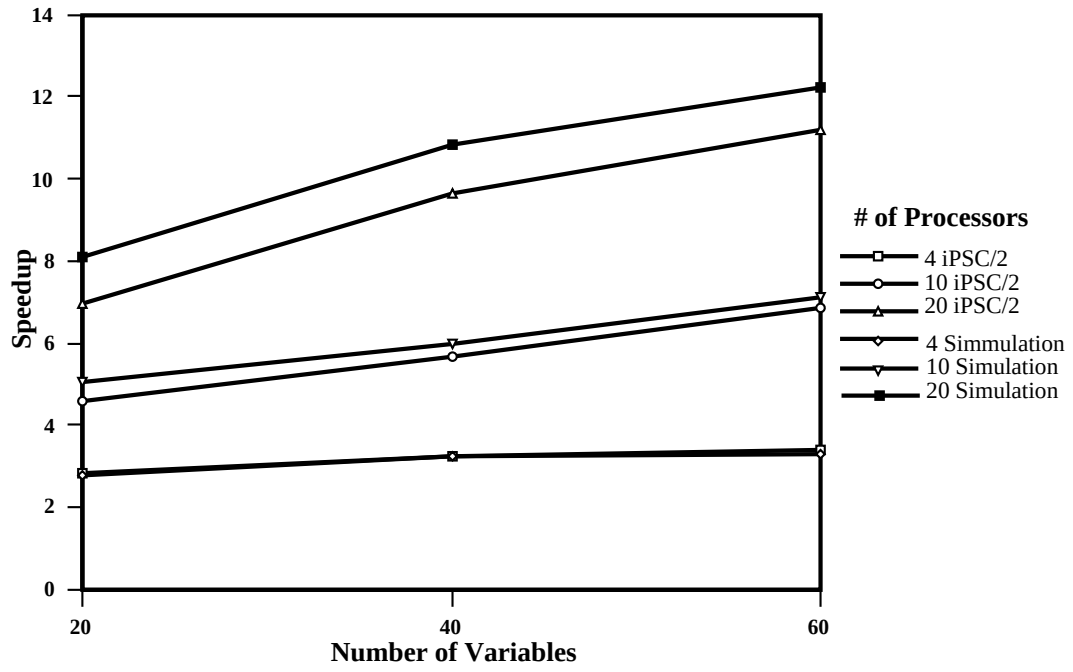

Figure 6.8: Speedup of DSPAC-3 over AC-3 for a Distributed Memory Machine

6.3.3 Hybrid Memory Machine

A comparison of processor cycles for the sequential AC-3 and SPAC-3 algorithms on the simulator and BBN TC2000 is shown in Table 6.10.

Simulation of SPAC-3 for these networks on a four-processor computer predicts speedup of between 79 and 93% of linear speedup (efficiency), and actual computer execution indicates speedup equal to the simulated speedup (Figure 6.9). Simulation of a ten-processor computer predicts a speedup of between 65 and 82% of linear speedup, but actual runs show that the prediction is off by 0 to 1%. Simulation of a twenty-processor computer predicts a speedup of between 56 and 73% of linear speedup, but actual runs show that the prediction is off by 0 to 2%.

Utilization and redundancy figures display similar trends. Utilization of SPAC-3 simulation is about 92, 87, and 88% for four, ten, and twenty processors, respectively. Utilization of SPAC-3 actual measurements is about 91, 83, and 85% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the SPAC-3 simulation is 2 to 63% higher than AC-3. The redundancy of work performed by the SPAC-3 on the BBN Butterfly is 1 to 60% higher than AC-3.

Table 6.10: Processing Cycles of AC-3 and SPAC-3 Algorithms for a Hybrid Memory Machine

Measurement	Algorithm	20 Vars	40 Vars	60 Vars
Sequential	AC-3	1490	3181	4905
Simulation, 4 proc	SPAC-3	474	879	1326
BBN TC2000, 4 proc	SPAC-3	474	880	1328
Simulation, 10 proc	SPAC-3	228	446	602
BBN TC2000, 10 proc	SPAC-3	231	450	604
Simulation, 20 proc	SPAC-3	129	233	337
BBN TC2000, 20 proc	SPAC-3	134	235	338

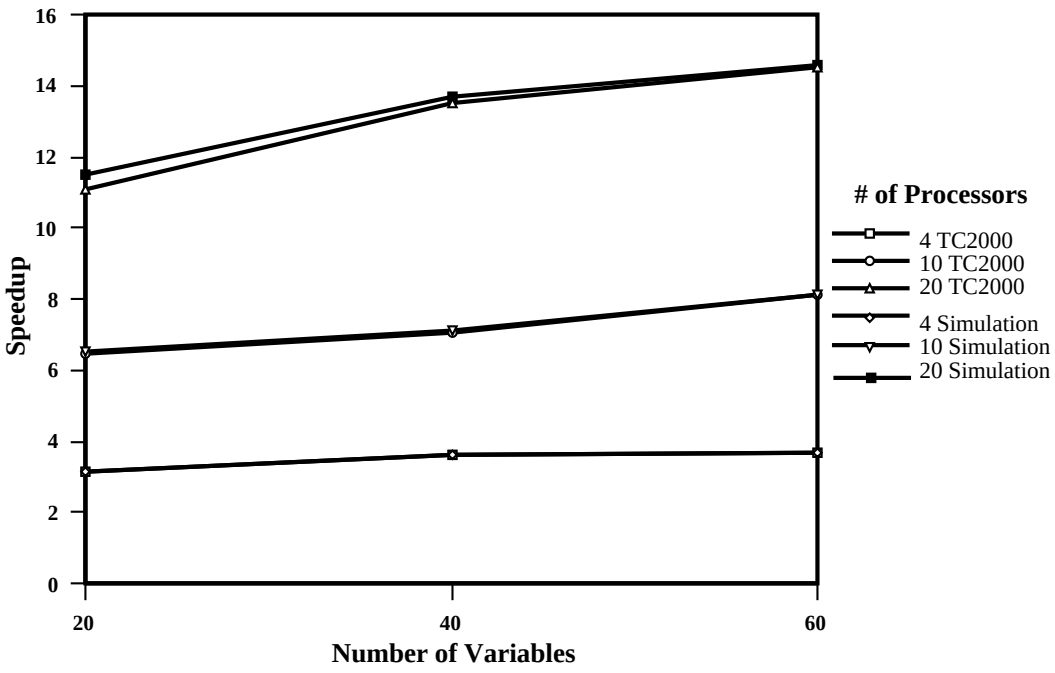

Figure 6.9: Speedup of SPAC-3 over AC-3 for a Hybrid Memory Machine

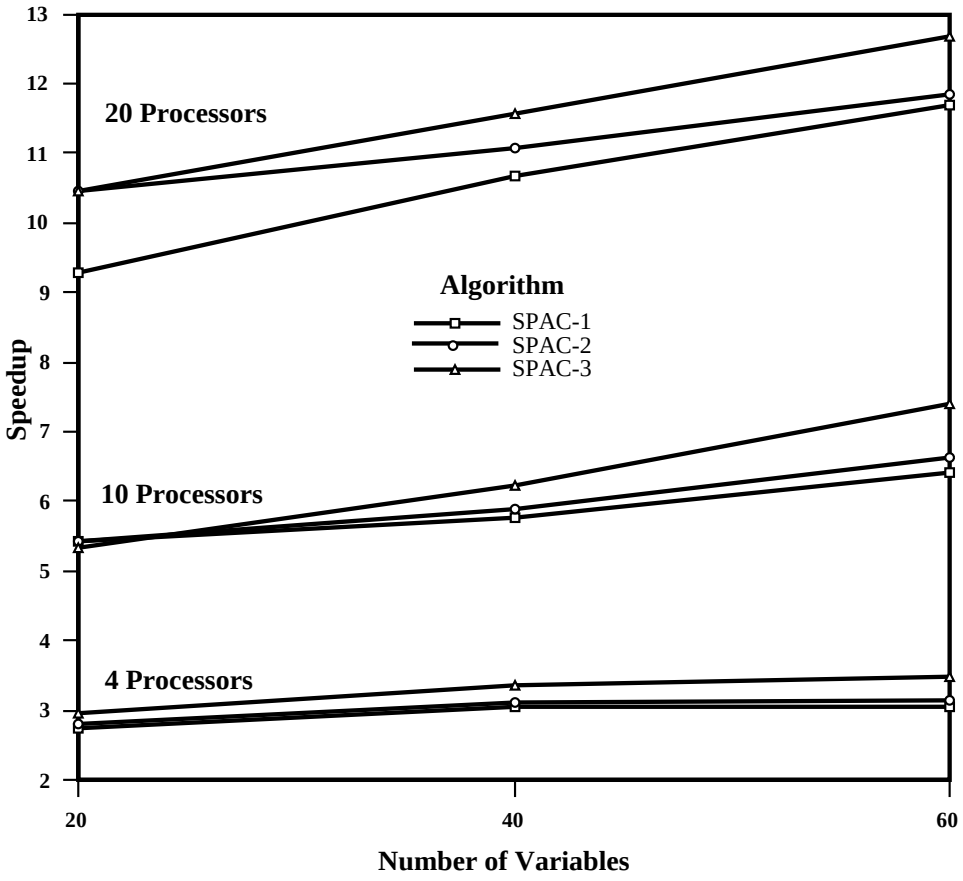

Figure 6.10: Speedup of SPAC over AC-3 for a Shared Memory Machine

6.4 Comparisons Between SPAC Algorithms

Of particular interest is how these algorithms behave as compared to each other. We compute SPAC-1, SPAC-2, and SPAC-3 speedup based on the AC-3 sequential algorithm. As one would expect, SPAC-3 performs the least amount of work, so it shows the best speedup. Also, as we predicted in Section 3.5, SPAC-2 performs better than SPAC-1. Figure 6.10 shows comparisons of speedup for the Sequent Symmetry, and Figure 6.11 shows comparisons of speedup for the Intel iPSC/2. Measurements for the BBN Butterfly TC2000 are similar to the Sequent Symmetry.

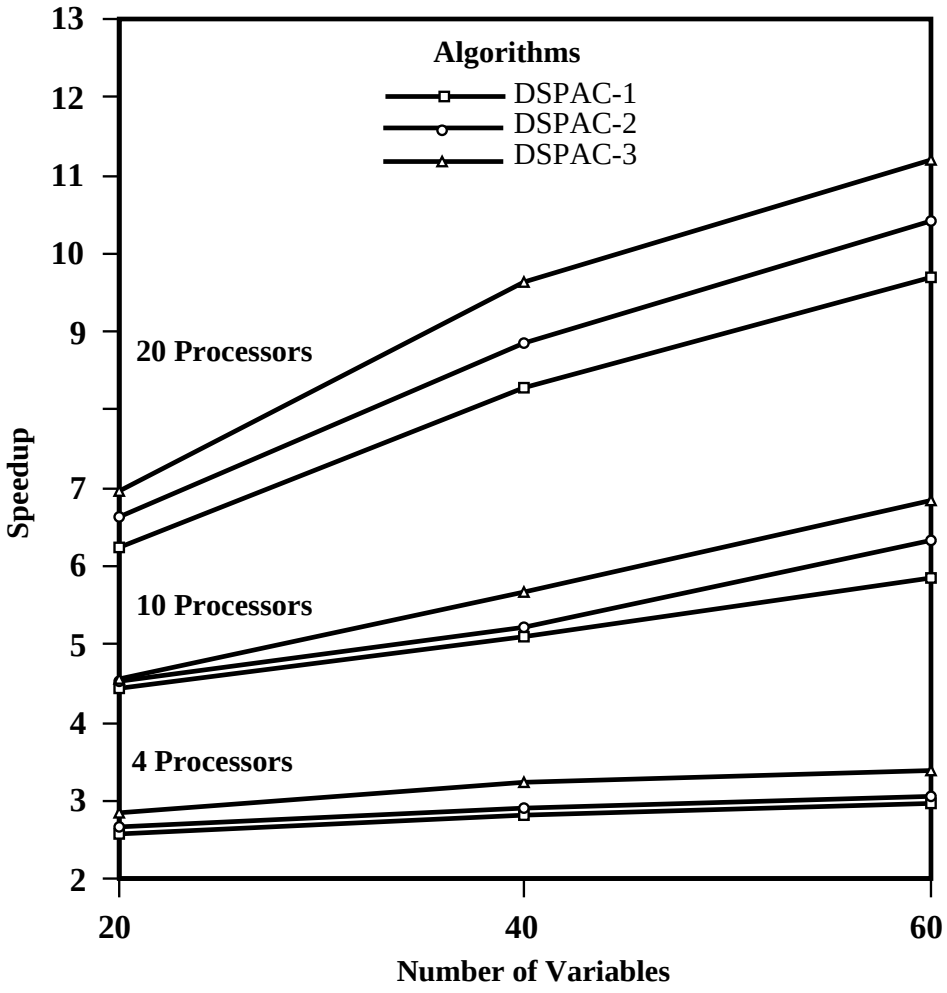

Figure 6.11: Speedup of DSPAC versus AC-3 on a Distributed Memory Machine

6.5 Scaling of Problems using Parallelism

Since arc consistency algorithms can greatly reduce the search space of a CSP, they can be used as an effective pre-processor for many AI applications which are viewed as CSPs. Gu [22] lists several CSP problems including: resource scheduling, detection of graph and subgraph isomorphism, the graph coloring problem, semantic information processing, theorem proving, image processing and object recognition, VLSI design, neural network computing, and database consistency checking.

The constraint networks examined thus far represent small problems with a single solution. These networks can be scaled to larger problems by processing more constraint variables on each processor. From the measurements, one can see that our algorithms perform best when each processor is assigned many constraint variables. Our best speedup is measured when 60 constraint variables are assigned to four processors. Also, due to the distributed nature of the algorithms, problems can also be scaled to a greater number of processors.

As an example of scaling of a problem, we create several large constraint networks and compare results of the DSPAC-2 algorithm on five different distributed memory multiprocessor configurations. These networks are also shown in Table 6.1. In particular, our network has 120 variables, the domain size is 16 values, the connectivity of each variable is from 2 to 30, and the satisfiability is 15%. When executed on an Intel iPSC/2 with 4, 10, 20, 32, and 64 processors, we still realize an appreciable speedup (although speedup and utilization decreases with configuration size). Speedup and utilization of these large networks as compared to the 60 variable networks are shown in Figure 6.12.

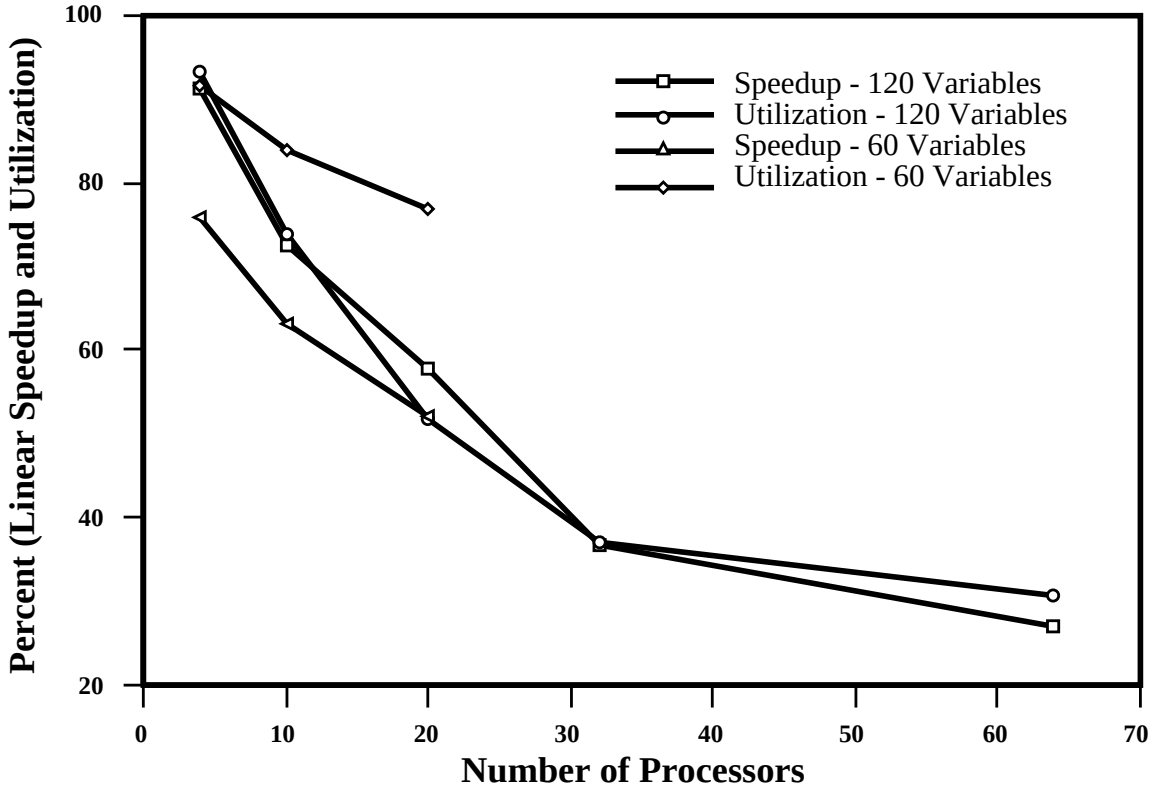

Figure 6.12: Speedup and Utilization of DSPAC-2 for a Large Network

Chapter 7

An In-Depth Examination of DSPAC

This chapter contrasts three Distributed Static Parallel Arc Consistency algorithms by varying the frequency of communications. We also consider the impact on speedup of static, simple load balancing versus the more robust mean field annealing graph partitioning heuristic. Like many parallel algorithms, parameters such as distribution of work, frequency of message passing, and load balancing greatly affect the algorithm's potential speedup. Through actual machine experimentation, we measure time required for different message frequencies and work assignments. Results indicate the less frequent communication and, in most cases, the robust load balancing, yield the best speedup [12] when we do not consider execution time of the load balancing algorithms. If we consider the execution time of load balancing in the computation of speedup, simple load balancing performs better. The effect of arc consistency on the execution time of search is examined.

7.1 Variable-frequent DSPAC Algorithms

The main difference between arc-based and variable-based DSPAC algorithms is the frequency of communications among processors in the node component of the code. Messages can be sent after each arc is checked (Figures 5.2, 5.3, and 5.4), or after all

```
procedure DSPAC-1-NODE(p)
begin
    for each assigned variable i do
        for each directed arc(i,j) do
        begin
            localchange = FALSE;
            localchange = Sprevise(arc(i,j));
            change = change ∨ localchange;
        end
        if (localchange)
            for each arc(i,j), where j is on processor m
                send(D_i) to m;
    return(change);
end
```

Figure 7.1: DSPAC-1 Algorithm: Node Processor Component for Variable-Based Message Frequency

of the arcs associated with a variable are checked (Figures 7.1, 7.2, and 7.3). Here we see a trade-off between work performed and frequency of communications. If messages are sent more frequently, more time is devoted to sending and receiving the messages, but less time is spent performing consistency checks. We measure the execution time of these two algorithms to determine the best approach.

```
procedure DSPAC-2-NODE(p)
begin
    change[p] := TRUE;
    while((change[p] ∨ active_procs > 0) ∧ ¬no_solution)
    begin
        while(change[p] ∧ ¬no_solution) do
        begin
            send(active_procs + +) to host;
            while(change[p] ∧ ¬no_solution) do
            begin
                change[p] := FALSE;
                for each assigned variable i do
                begin
                    localchange := FALSE;
                    for each directed arc(i, j)
                        localchange := localchange∨ (Sprevise(arc(i, j)));
                    if localchange
                        for each arc(i, j), where j is on processor m
                            send(D_i) to m;
                end
            end
            send(active_procs − −) to host;
        end
    end
end
```

Figure 7.2: DSPAC-2 Algorithm: Node Processor Component for Variable-Based Message Frequency

```
procedure DSPAC-3-NODE(p)
begin
    for each assigned variable i
        Q[p] ← Q[p] ∪ {arc(i,j)|arc(i,j) ∈ G, i ≠ j}
    localchange := FALSE;
    while(((Q[p] ≠ φ) ∨ active_procs > 0) ∧ ¬no_solution)
    begin
        while((Q[p] ≠ φ) ∧ ¬no_solution) do
        begin
            send(active_procs + +) to host;
            while((Q[p] ≠ φ) ∧ ¬no_solution) do
            begin
                for arc(i,j) ∈ Q[p] do
                begin
                    Q[p] := Q[p] − {arc(i,j)};
                    localchange := localchange∨ (Sprevise(arc(i,j))
                    if (localchange∧ next arc in Q[p] not (arc(i,x))) do
                    begin
                        for each arc(i,x), where x is on processor m
                            send(D_i) to m;
                        for each arc(x,i), where x is on this processor
                            Q[p] := Q[p] ∪ {arc(x,i)|arc(x,i) ∈ E, x ≠ i};
                        localchange := FALSE;
                    end
                end
            end
            send(active_procs − −) to host;
        end
    end
end
```

Figure 7.3: DSPAC-3 Algorithm: Node Processor Component for Variable-Based Message Frequency

7.2 Simple Data Partitioning

We represent CSPs as a graph with nodes and edges. We can, therefore, employ existing graph partitioning algorithms to distribute work among processors of a multiprocessor configuration. In fact, graph partitioning clusters nodes together while attempting to minimize the number of connections between clusters. The importance of a minimum cutset becomes apparent if one considers that an edge between two clusters represents potential communications for a distributed memory computer.

One simple method of partitioning data is to fill the partitions with approximately equal weights, with no regard at all to the edges. Our simple load balancing fills one partition at a time. Weights of the nodes are equal to the degree of the node. Nodes are removed from the pool of all nodes by order of size. The largest node that fits in the partition is selected until the weight of the partition is approximately equal to the predicted average of all partitions. While this is not elegant, it is fast, as shown in Section 7.5.

7.3 Mean Field Annealing

The quality of a graph partition is described by an objective function, H, that indirectly indicates the cost of a partition. The main attributes which contribute to the cost of a given partition are the cost of edges between partitions (H_e) and the cost of processor imbalance (H_b) given by [60]:

$$H = H_e + \alpha H_b \tag{7.1}$$

where α is a scaling factor reflecting the computational significance of processor imbalance.

The objective of the partitioning process is to find the specific partition with the smallest cost. One pitfall of such convergence is potential minimization of cost which could actually be a local minima. Simulated annealing [37] avoids such local minima by moving nodes based on a probability. Mean field annealing (MFA) [60] avoids local minima in a similar manner. MFA calculates a graph node's probability that it should be in a certain partition. The MFA algorithm forces these probabilities to converge to either 0 or 1. MFA (as well as simulated annealing) is an algorithm which models a collection of atoms being slowly cooled to a ground state. The algorithm uses the concept of a decreasing temperature to aid in the convergence of a solution.

The mean field annealing algorithm can be represented as [14, 60]:

1. Initialize a starting temperature, T.
2. Initialize the *spin* averages, (s_{ik}). A *spin average* is the probability that node i should be in bin k, where $\sum_k (s_{ik}) = 1$.
3. Repeat until a steady state is reached:
   (a) Select a node, i, at random.
   (b) Set all of the spins for i to zero.
   (c) For each spin, compute the objective function H_{ik} when $(s_{ik}) = 1$.
   (d) For each spin, compute the a new spin based on its objective function and the current temperature.
4. Decrease the temperature: $T_{new} = \beta T_{old}$, where β is the cooling rate. Repeat step 3 until the ground state is reached.

MFA executes in the same time-order as the Kernighan-Lin algorithm [60], but obtains optimal partitions as often as the simulated annealing algorithm.

7.4 Network Data

We have created several sample constraint networks for experimental trials. The networks are computer-generated and have some attributes determined by random processes as follows:

- Size: problems contain 400 variables.
- Domains: each variable has a domain of 64 values.
- Solutions: Each problem contains a random number of solutions. Since arc consistency algorithms are typically not used to solve constraint satisfaction problems, each domain has many values intact after the networks are made consistent.
- Connectivity: The connectivity (degree) of each variable is a uniformly distributed random number, with a minimum of 30, a maximum of 49, and an average of 39.25.
- Satisfiability: The relative satisfiability is the success rate of a consistency check. More specifically, it is the ratio of actual constraint value pairs to the total possible pairs (domain $\times$ domain). The satisfiability of our networks are (1638 relation pairs)/(64 domain values $\times$ 64 domain values)= 0.40.

While gathering the experimental data, four random constraint networks were run for each of the configurations. The detail of these networks is shown in Table 6.1.

7.5 Metrics and Results

From the performance point of view, we limit our measurements to one algorithmic parameter, the speedup. *Speedup* is the effective speed of a parallel computation in operations executed per time unit of a parallel computer with p processors [1]. This is expressed as:

$$speedup = S(p) = \frac{time(best\,sequential\,algorithm)}{time(parallel\,algorithm)} \tag{7.2}$$

We measure start-to-stop time, including data loading and parallel operating system overhead. We should note that the speedup should be compared with the execution time of the best sequential algorithm. We address the analysis of sequential algorithms later in this chapter.

We gather time measurements of identical constraint networks with these following algorithm and load balancing configurations:

- Sequential AC-1 and AC-3 algorithms (executed on a single iPSC/2 node processor).
- DSPAC-1, 2, and 3, arc-frequent communications, with simple load balancing.
- DSPAC-1, 2, and 3, variable-frequent communications, with simple load balancing.
- DSPAC-1, 2, and 3, arc-frequent communications, with mean field annealing load balancing.
- DSPAC-1, 2, and 3, variable-frequent communications, with mean field annealing load balancing.

Table 7.1: Execution Times of Communications/Load Balancing Configurations for Constraint Networks of Section 7.4 on a Distributed Memory Machine

All values are in seconds

Algorithm	Number of Processors	Arc Comm. Simple Part.	Var. Comm. Simple Part.	Arc Comm. MFA Part.	Var. Comm. MFA Part.
DSPAC-1	8	8.730	5.275	8.805	5.375
DSPAC-1	16	8.150	4.335	7.895	4.198
DSPAC-1	32	8.460	4.140	7.290	4.020
DSPAC-2	8	12.497	8.832	12.155	8.850
DSPAC-2	16	10.552	5.890	9.252	5.412
DSPAC-2	32	9.562	4.345	8.913	4.035
DSPAC-3	8	14.227	6.637	15.865	6.845
DSPAC-3	16	12.258	5.482	13.818	5.630
DSPAC-3	32	13.825	4.967	12.398	4.460

We execute the DSPAC algorithms using the above algorithm/load balancing configurations on eight, sixteen, and thirty-two processors of an Intel iPSC/2 multiprocessor. We use the constraint networks described in Section 7.4 as our data. Actual time measurements are shown in Table 7.1. Figure 7.4 shows the speedup of each of our four algorithm/load balancing configurations for the DSPAC-1 algorithm. Figure 7.5 shows the speedup of each of our four algorithm/load balancing configurations for the DSPAC-2 algorithm. Figure 7.6 shows the speedup of each of our four algorithm/load balancing configurations for the DSPAC-3 algorithm.

Each time measurement entry of the table and point of the figure represents the average of our four constraint networks. The average execution time of the sequential AC-1 algorithm is 88.363 seconds, while the average execution time of the sequential AC-3 algorithm is 115.760 seconds.