Read-by-one, write-by-all variables implement a *test-and-set* operation [27]. When a specific processor wants to use a variable, it locks the desired memory location and prevents other processors from using it. The processor unlocks the memory location once the variable update is complete. We assume that these variables are rarely accessed, so there is little or no delay due to a locked variable. These variables make up 0.1% of the algorithm's variables. Changes to these variables are also temporarily stored during a processing cycle.

We have one single variable which is read and written by all processors. This variable represents the number of active processors and is used to signify the end of the problem computation. This variable is also implemented using a test-and-set operation and is simply available to all the processors all the time, but the value is used only at the end of a cycle for logical operations (looping conditions).

5.1.3 Simulating a Distributed Memory Architecture and Communication

The distributed memory simulator option assumes the access times to local memory are identical. The conventions of some of the variables, however, are different due to the nature of the memory architecture:

Read-only variables are stored on each processor. Only those variables needed by each processor are stored. Our simulator, however, simply has one copy of all these variables.

Variables which are read by all processors, but written by one processor are stored on each processor. Each processor starts with the same variable values, but these are changed once the processing commences. If a processor changes one of these variables, it sends a message with the changes to all those processors which use that variable. Frequency of messages is specified by the user in each simulation. A value-based simulation sends a message every time a value is removed from a domain. An arc-based simulation sends a message if at least one value is removed after one entire arc is checked. A variable-based simulation sends a message if at least one value is removed after all the arcs originating from a single variable have been checked. A processor-based simulation sends a message after all constraint variables assigned to a processor are checked. Each message takes one time slice to reach its destination, and each processor takes one time slice to receive the data.

We simulate these variables by having three copies of each variable. One copy represents the information local to each processor, called “local memory”. Another represents the variables which are sent in messages from one processor to another, called “communication memory”. The third copy represents variable changes which are received from another processor, called “global memory”. We simulate distributed memory and processor-to-processor communication as follows:

1. A processor uses its own “local memory” to read values and make updates. It uses “global memory” to read variables which are changed by other processors.
2. When a processor determines it should send a variable change, it puts the value in the “communication memory”.
3. The simulator moves the change from “communication memory” to global memory after one time slice has passed. The simulator also takes one time slice to “receive” this value.

Each processor has a variable which is written by all processors, but read only by itself. This variable is easily maintained by receiving messages from other processors with instructions to change the variable. Since this is a logical variable, we only need to perform **OR** or **AND** operations. These are handled the same as read-by-all, write-by-one variables described above.

The single read/write-by-all variable is "maintained" by one processor, or by the host processor. We simulate this as a variable on a separate processor, and all processors send messages to it. Messages instruct the processor to increment or decrement the variable's value. On an actual multiprocessor, this variable accurately indicates when all processors are done, but locating this variable on one processor poses a potential bottleneck. This proves to be the biggest shortcoming of our simulator, but appears to be too complex to simulate accurately.

5.1.4 Other Simulator Details

The hybrid memory option is similar to the shared memory option, except that the read-only variables are stored in a processor's private memory, and all other variables are stored in global shared memory. Access to private memory is faster than access to shared memory. However, since each processor accesses private and shared memory in the same proportions (60% for private memory), our hybrid and shared memory options display identical simulation results.

Our simulation is implemented by programming the sequential and parallel algorithms in the C language on an IBM-compatible 80386 personal computer. We use the memory constructs described above.

5.2 Machine Implementation

We implement the SPAC algorithms on the Sequent Symmetry[1] shared memory computer, the Intel iPSC/2[2] hypercube computer, and the BBN TC2000 Butterfly[3] computer.

Although several of these computers have large numbers of processors, we limit most of our measurements to a number common to all machines. Since the Sequent Symmetry has only 26 processors, we chose 20 as the maximum. This number also works well with the constraint networks since the minimum number of constraint variables is also 20.

In only a few cases did we use more than 20 processors. On some trials on the iPSC/2, we used 32 and 64 processors. We report our results of these trials in Sections 6.5 and 7.

5.2.1 Parallel Execution on the Sequent Symmetry

We implement in C the SPAC algorithms on the Sequent Symmetry [57] shared memory multiprocessor (also called a tightly-coupled MIMD computer). The Symmetry S81 has twenty eight 32-bit processors, 64-Kbytes of cache memory per processor, and 32-Mbytes of shared memory.

We program the Sequent computers such that one processor reads the data file and loads the shared memory with the constraint information. The one processor then starts the remaining processors. Each processor (including the "host") is assigned a set of variables to check. The same variable assignments are used with the simulation

---

[1] Symmetry is a trademark of Sequent Computer Systems, Inc.

[2] iPSC/2 is a trademark of Intel Inc.

[3] BBN, TC2000, and Butterfly are trademarks of Bolt, Beranek, and Newman Inc.

and Sequent trials. Work is partitioned equitably as described in Section 3.1. Once the processors complete the work, the first processor reads the shared memory and records the results.

All constraint variables, values, and relations are stored as shared variables, and each processor's counting variables are stored as private variables. The constraint relations are never updated, and hence, this data can be copied into a processor's cache with no coherency problems.

The variable domain values which are read by many processors but written by one, the counting variable *active_procs*, the flag *no_solution*, and the *change* bits were initially implemented as test-and-set variables. Only one processor at a time is allowed to access each variable and this is achieved by employing special "lock" and "unlock" operators provided by the Sequent C library.

Because of the asynchronous nature of our algorithm, we found that we did not have to use the test-and-set operation for the read-by-one, write-by-all variables. Earlier trials [8] found that locking these variables caused large, unnecessary delays for data accesses.

Executing the SPAC-3 algorithm on a shared memory machine poses additional difficulties. The underlying structure of the queue of arcs could cause a large bottleneck if a single space in memory were allocated. We, therefore, have to ensure that adding an arc to one queue does not cause other queues to be locked. We address this by maintaining separate queues **and** free lists for each processor.

The SPAC algorithms we implement are identical to those already shown in Figures 3.2, 3.3, and 3.5.

5.2.2 Parallel Execution on the Intel iPSC/2

The iPSC/2 computer [28, 29] is a distributed memory hypercube computer (loosely-coupled MIMD computer). The inter-processor communications is handled via a hypercube configured network, although Intel specifies that the network actually works like a fully connected network due to worm-hole routing techniques. Each processing node contains an Intel 80386 processor and four Megabytes of memory. The system we use contains sixty-four processors. For the measurements comparing the iPSC/2 with other multiprocessors, we employed only four, ten, or twenty processors. For other measurements, we used eight, sixteen, thirty-two, and sixty-four processors.

The iPSC/2 operating system allows a host computer to control a set of processing nodes. Our host distributes and loads the work into the processing nodes, instructs the processing nodes to start, monitors the work performed, and gathers the node results and desired statistics. Each processor is assigned a number of constraint network variables and the associated constraint data.

As we mentioned in Section 5.1.3, we need to change our SPAC algorithms and Sprevise functions to execute on a distributed memory multiprocessor. We call them Distributed Static Parallel Arc Consistency (DSPAC) algorithms, and we use SPAC and DSPAC interchangeably to refer to these distributed algorithms. In these DSPAC algorithms, only the processor node is allowed to change the supported values of its variable's domain. Each processor then reports this "write" action to other processors by sending a message. The message is sent only to other processors which use that particular variable's values in their computation. Although we simulate four message frequencies, we only program the arc-based message paradigm for comparing performance against other multiprocessors in the results reported in Chapter 6. The arc-based frequency shows the most consistent speedup value and usually represented the best speedup possible in simulation. We later found that the variable-based frequency actually performs best with large constraint networks. We discuss the variable-frequent version on the DSPAC algorithms in Chapter 7.

Communication is handled asynchronously. Each node processor has several interrupt-driven receive procedures which are activated depending on the message type.

The global variable *no_solution* is implemented by a broadcast message. If a processor finds that a variable is empty (and the constraint network has no solution), it sends a message to all the processors informing them to stop.

The counting variable *active_procs* is maintained in the host. Processing nodes indicate to the host when the processors start work and become idle. When all the processors have stopped, our arc consistency algorithm is complete.

Figures 5.2, 5.3, and 5.4 show the modified DSPAC procedures for distributed memory SPAC-1, SPAC-2, and SPAC-3, respectively. The host programs basically remain the same, except the host is separate from the processing nodes and must gather the results. All node procedures now use the Sprevise function (SPAC-2 for shared memory machines used the Sprevise-2 function).

The major difference with the DSPAC-NODE procedures is that a processor is now responsible for keeping its copy of the constraint network domains current. The Sprevise functions tell the DSPAC-NODE procedures that a domain has changed. The node procedures then send messages to other processors which are affected by the domain change. Each node processor has an interrupt-driven receive procedure which reads domain change messages, updates its own memory, and updates its own *change* or $Q[p]$ variables.

```
procedure DSPAC-1-HOST()
begin
    PNC();
    assign variables to processors;
    $change$ := TRUE; $no\_solution$ := FALSE
    while($change \wedge \neg no\_solution$) do
    begin
        $change$ := FALSE;
        ||for processing node $p$, $change = change \vee$ DSPAC-1-NODE($p$);
    end
    gather results;
end
procedure DSPAC-1-NODE($p$)
begin
    for each assigned variable $i$ do
        for each directed $arc(i, j)$ do
        begin
            $localchange$ = FALSE;
            $localchange$ = Sprevise($arc(i, j)$);
            $change = change \vee localchange$;
            if ($localchange$)
                for each $arc(i, j)$, where $j$ is on processor $m$
                    send($D_i$) to $m$;
        end
    return($change$);
end
```

Figure 5.2: DSPAC-1 Algorithm: Host Processor and Node Processor Components

```
procedure DSPAC-2-HOST()
begin
    PNC();
    $active\_procs := 0$; $no\_solution :=$ FALSE;
    assign variables to processors;
    || for each processing node $p$, DSPAC-2-NODE($p$);
    wait for $active\_procs = 0$;
    gather results;
end
procedure DSPAC-2-NODE($p$)
begin
    $change[p] :=$ TRUE;
    while($(change[p] \vee active\_procs > 0) \wedge \neg no\_solution$)
    begin
        while($change[p] \wedge \neg no\_solution$) do
        begin
            send($active\_procs++$) to host;
            while($change[p] \wedge \neg no\_solution$) do
            begin
                $change[p] :=$ FALSE;
                for each assigned variable $i$ do
                    for each directed $arc(i, j)$ do
                    begin
                        if (Sprevise($arc(i, j)$))
                            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 5.3: DSPAC-2 Algorithm: Host Processor and Node Processor Components

```
procedure DSPAC-3-HOST()
begin
    PNC();
    $active\_procs$ := 0; $no\_solution$ := FALSE;
    assign variables to processors;
    || for each processing node $p$, DSPAC-3-NODE($p$);
    wait for $active\_procs = 0$;
    gather results;
end
procedure DSPAC-3-NODE($p$)
begin
    for each assigned variable $i$
        $Q[p] \leftarrow Q[p] \cup \{arc(i,j) | arc(i,j) \in G, i \neq j\}$
    while((($Q[p] \neq \phi$) $\vee$ $active\_procs > 0$) $\wedge$ $\neg no\_solution$)
    begin
        while(($Q[p] \neq \phi$) $\wedge$ $\neg no\_solution$) do
        begin
            send($active\_procs++$) to host;
            while(($Q[p] \neq \phi$) $\wedge$ $\neg no\_solution$) do
            begin
                for $arc(i,j) \in Q[p]$ do
                begin
                    $Q[p] := Q[p] - \{arc(i,j)\}$;
                    if (Sprevise($arc(i,j)$)) do
                    begin
                        for each $arc(i,j)$, where $j$ is on processor $m$
                            send($D_i$) to $m$;
                        for each $arc(m,i)$, where $m$ is on this processor
                        $Q[p] := Q[p] \cup \{arc(m,i) | arc(m,i) \in E, m \neq i, m \neq j\}$;
                    end
                end
            end
            send($active\_procs--$) to host;
        end
    end
end
```

Figure 5.4: DSPAC-3 Algorithm: Host Processor and Node Processor Components

5.2.3 Parallel Execution on the Butterfly TC2000

The TC2000 Butterfly computer [3, 4] is a shared memory computer (tightly-coupled MIMD computer). Each processing node can contain up to 32 megabytes of memory which can be configured as private to the processor or shared among the nodes of the computer. Each processor contains a Motorola 88100 processor and 32 kilobytes of cache memory. The system we use contains 45 processors, and each processor has four megabytes of memory. We use only 4, 10, or 20 processors.

We program the Butterfly computer such that one processor reads the data file and loads the shared memory with the constraint information. It then starts processing one or more constraint variables on each node of the configuration. Once the processors complete the work, the first processor reads the shared memory and records the results.

All constraint variables, values, and relations are stored as shared variables, and each processor's counting variables are stored as private variables. The constraint relations are never updated, and hence, this data can be copied into a processor's cache with no coherency problems.

The variable domain values, which are read by many processors but written by one, are implemented as cachable shared variables. The processor which writes an update implements a "write-through" caching scheme, where in writing a value to cache causes the processor to write the same value into the shared memory copy. A processor which uses this shared variable updates its cache each time it uses the variable.

We learned from our Sequent Symmetry implementation that only the counting variable *active_procs* should be implemented as a test-and-set variable. This is achieved by employing special "atomic" operators provided by the Butterfly operating system.

Other types of shared variables, the flag *no_solution*, and the *change* bits are not locked because of the asynchronous nature of our algorithm.

Chapter 6

Performance Results

A simulator has been developed [8, 13, 59] to examine the work performed by the SPAC-1, SPAC-2, and SPAC-3 algorithms on a parallel computer. Results were obtained by executing these algorithms and comparing them to their respective sequential algorithm counterparts. These results are compared to actual trials on parallel processors. Actual results are also reported in [9, 10, 11].

We developed many sample constraint networks to use as the data for experimental trials. The networks are computer-generated and have some attributes determined by random processes (Table 6.1). As an example of scaling a problem, we also created several large constraint networks. These networks are also shown in Table 6.1, and are described in Section 6.5.

The characteristics of the computer-generated networks are:

- Size: problems contain 20, 40, and 60 variables.
- Domains: each variable has a domain of 10 values.
- Solutions: Each problem contains one specific solution. The number of solutions is the total number of different consistent assignments that the problem

Table 6.1: Examples of Constraint Network Characteristics

Satisfiability = 0.15

Network	Variables	Minimum Connect.	Maximum Connect.	Average Connect.	Total Edges	Domain Size	Solutions
P20S01-0	20	2	10	6.30	63	10	1
P20S01-1	20	2	10	6.30	63	10	1
P20S01-2	20	2	10	6.00	60	10	1
P20S01-3	20	2	10	4.80	48	10	1
P20S01-4	20	2	10	5.00	50	10	1
P20S01-5	20	2	10	6.00	60	10	1
P20S01-6	20	2	10	6.20	62	10	1
P20S01-7	20	2	10	7.00	70	10	1
P20S01-8	20	2	10	5.80	58	10	1
P20S01-9	20	2	10	5.80	58	10	1
Average				5.92	59		
P40S01-0	40	2	10	5.95	119	10	1
P40S01-1	40	2	10	5.90	118	10	1
P40S01-2	40	2	10	6.30	126	10	1
P40S01-3	40	2	10	6.65	133	10	1
P40S01-4	40	2	10	6.55	131	10	1
P40S01-5	40	2	10	6.20	124	10	1
P40S01-6	40	2	10	6.20	124	10	1
P40S01-7	40	2	10	6.20	124	10	1
P40S01-8	40	2	10	5.95	119	10	1
P40S01-9	40	2	10	6.25	125	10	1
Average				6.22	124		
P60S01-0	60	2	10	5.63	169	10	1
P60S01-1	60	2	10	6.40	192	10	1
P60S01-2	60	2	10	6.13	184	10	1
P60S01-3	60	2	10	5.17	155	10	1
P60S01-4	60	2	10	6.10	183	10	1
P60S01-5	60	2	10	6.00	180	10	1
P60S01-6	60	2	10	6.20	186	10	1
P60S01-7	60	2	10	6.07	182	10	1
P60S01-8	60	2	10	6.20	186	10	1
P60S01-9	60	2	10	6.43	193	10	1
Average				6.03	181		
BIGNET1	120	2	30	15.13	908	16	64
BIGNET2	120	2	30	16.47	988	16	32
BIGNET3	120	2	30	16.82	1008	16	32
BIGNET4	120	2	30	15.68	941	16	16

Satisfiability = 0.40

Network	Variables	Minimum Connect.	Maximum Connect.	Average Connect.	Total Edges	Domain Size	% Values Removed
P400C10C	400	30	49	39.24	7848	64	24.6
P400C10D	400	30	49	39.76	7952	64	25.1
P400C10E	400	30	49	39.48	7895	64	25.0
P400C10F	400	30	49	39.03	7805	64	24.5

network would yield if solved using a tree search algorithm. Although arc consistency may not necessarily remove all but one value from each domain, we chose one-solution networks to provide an easy end point to compare algorithm measurements.

- Connectivity: The connectivity (degree) of each variable is a uniformly distributed random number from two to ten. The order of the binary constraints for each edge has no bearing on the consistency check complexity analysis.
- 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 × domain). The constraint network in Figure 1.5 has three constraint value pairs, and nine possible pairs, for a satisfiability of 0.33. The satisfiability of our sample networks are (15 relation pairs)/(10 domain values × 10 domain values)= 0.15.

While gathering the experimental data, ten random constraint networks were run for each of the configurations.

The same set of constraint networks were pre-processed using the AC algorithms on a sequential computer, the SPAC algorithms on our parallel processing simulator, and the SPAC algorithms on actual parallel processors. We assess performance by counting consistency checks.

We examine three different parallel architectures: shared memory, distributed memory, and a hybrid of the two. These three architectures have been simulated. In our actual measurements, the Sequent Symmetry represents a shared memory machine, the Intel iPSC/2 Hypercube represents a distributed memory machine, and the BBN TC2000 Butterfly represents a hybrid machine.

A comparison of processor cycles for the DSPAC algorithms on the distributed memory simulator appear different from the other architectures. As mentioned in Section 5.1.3, we can simulate four frequencies of messages based on the completion of consistency checks. The simulation data below shows processor cycles for these frequencies. Only the arc-based message frequency was implemented on the iPSC/2. The only simulation data plotted and discussed, therefore, is also the arc-based frequencies.

6.1 Results for SPAC-1 Algorithms

6.1.1 Shared Memory Machine

A comparison of processor cycles for the sequential AC-1 and SPAC-1 algorithms on the simulator and Sequent Symmetry is shown in Table 6.2.

Simulation of SPAC-1 for these networks on a four-processor computer predicts speedup of between 79 and 91% of linear speedup (efficiency), and actual computer execution indicates speedup within 0 to 1% of the simulated speedup (Figure 6.1). Simulation of a ten-processor computer predicts a speedup of between 66 and 77% of linear speedup, while actual runs show that the prediction is off by 3 to 5%. Simulation of a twenty-processor computer predicts a speedup of between 53 and 69% of linear speedup, but actual runs show that the prediction is off by 1 to 4%.

Utilization and redundancy figures display similar trends. Utilization of SPAC-1 is high, but not as high as for the SPAC-2 algorithm (see Section 6.2). Utilization of SPAC-1 simulation is about 90, 81, and 78% for four, ten, and twenty processors, respectively. Utilization of SPAC-1 actual measurements is about 88, 75, and 73% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the SPAC-1 simulation is 2 to 50% higher than AC-1. The

Table 6.2: Processing Cycles of AC-1 and SPAC-1 Algorithms for a Shared Memory Machine

Measurement	Algorithm	20 Vars	40 Vars	60 Vars
Sequential	AC-1	1671	3592	5530
Simulation, 4 proc	SPAC-1	531	1011	1522
Symmetry, 4 proc	SPAC-1	544	1037	1613
Simulation, 10 proc	SPAC-1	254	517	718
Symmetry, 10 proc	SPAC-1	274	552	766
Simulation, 20 proc	SPAC-1	157	282	399
Symmetry, 20 proc	SPAC-1	161	298	420

redundancy of work performed by the SPAC-1 on the Sequent Symmetry is 4 to 40% higher than AC-1.

40

20

Figure 6.1: Speedup of SPAC-1 over AC-1 for a Shared Memory Machine

6.1.2 Distributed Memory Machine

A comparison of processor cycles for the sequential AC-1 and DSPAC-1 algorithms on the simulator and Intel iPSC/2 is shown in Table 6.3. 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-1 for these networks on a four-processor computer predicts speedup of between 70 and 83% of linear speedup (efficiency), and actual computer execution indicates speedup within 0 to 2% of the simulated speedup (Figure 6.2). Simulation of a ten-processor computer predicts a speedup of between 49 and 63% of linear speedup, but actual runs show that the prediction is off by 1 to 3%. Simulation of a twenty-processor computer predicts a speedup of between 36 and 54% of linear speedup and actual runs show that the prediction is off by 1 to 3%.

Utilization and redundancy figures display similar trends. Utilization of DSPAC-1 simulation is about 83, 68, and 65% for four, ten, and twenty processors, respectively. Utilization of DSPAC-1 actual measurements is about 84, 71, and 65% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the DSPAC-1 simulation is 2 to 80% higher than AC-1. The redundancy of work performed by the DSPAC-1 on the iPSC/2 is 3 to 81% higher than AC-1.

Table 6.3: Processing Cycles of AC-1 and DSPAC-1 Algorithms for a Distributed Memory Machine

Measurement	Algorithm	Frequency	20 Vars	40 Vars	60 Vars
Sequential	AC-1		1671	3592	5530
Simulation, 4 proc	DSPAC-1	Value	665	1270	1876
Simulation, 4 proc	DSPAC-1	Arc	593	1109	1661
Simulation, 4 proc	DSPAC-1	Variable	601	1127	1628
Simulation, 4 proc	DSPAC-1	Processor	726	1616	2396
iPSC/2, 4 proc	DSPAC-1	Arc	579	1101	1652
Simulation, 10 proc	DSPAC-1	Value	365	752	1066
Simulation, 10 proc	DSPAC-1	Arc	340	634	872
Simulation, 10 proc	DSPAC-1	Variable	388	635	847
Simulation, 10 proc	DSPAC-1	Processor	420	821	1111
iPSC/2, 10 proc	DSPAC-1	Arc	336	610	838
Simulation, 20 proc	DSPAC-1	Value	227	417	614
Simulation, 20 proc	DSPAC-1	Arc	231	384	508
Simulation, 20 proc	DSPAC-1	Variable	296	427	522
Simulation, 20 proc	DSPAC-1	Processor	296	510	657
iPSC/2, 20 proc	DSPAC-1	Arc	239	275	507

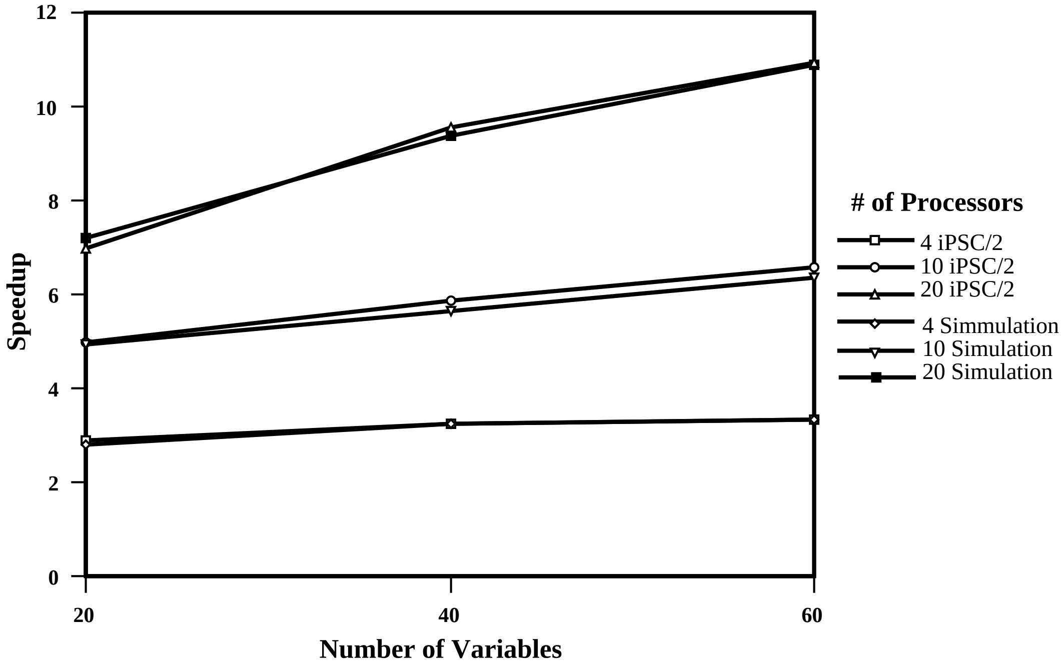

Figure 6.2: Speedup of DSPAC-1 over AC-1 for a Distributed Memory Machine

6.1.3 Hybrid Memory Machine

A comparison of processor cycles for the sequential AC-1 and SPAC-1 algorithms on the simulator and BBN TC2000 is shown in Table 6.4.

Simulation of SPAC-1 for these networks on a four-processor computer predicts speedup of between 79 and 91% of linear speedup (efficiency), and actual computer execution indicates speedup nearly equal to the simulated speedup (Figure 6.3). Simulation of a ten-processor computer predicts a speedup of between 66 and 77% of linear speedup, while actual runs show that the prediction is off by only 0 to 1%. Simulation of a twenty-processor computer predicts a speedup of between 53 and 69% of linear speedup, while actual runs show the prediction to be off by only 0 to 1%.

Utilization and redundancy figures display similar trends. Utilization of SPAC-1 is high, but not as high as for the SPAC-2 algorithm (see Section 6.2). Utilization of SPAC-1 simulation is about 90, 81, and 78% for four, ten, and twenty processors, respectively. Utilization of SPAC-1 actual measurements is about 90, 81, and 79% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the SPAC-1 simulation is 2 to 50% higher than AC-1. The redundancy of work performed by the SPAC-1 on the BBN Butterfly is 1 to 48% higher than AC-1.

Table 6.4: Processing Cycles of AC-1 and SPAC-1 Algorithms for a Hybrid Memory Machine

Measurement	Algorithm	20 Vars	40 Vars	60 Vars
Sequential	AC-1	1671	3592	5530
Simulation, 4 proc	SPAC-1	531	1011	1522
BBN TC2000, 4 proc	SPAC-1	531	1008	1520
Simulation, 10 proc	SPAC-1	254	517	718
BBN TC2000, 10 proc	SPAC-1	253	516	711
Simulation, 20 proc	SPAC-1	157	282	399
BBN TC2000, 20 proc	SPAC-1	154	282	396

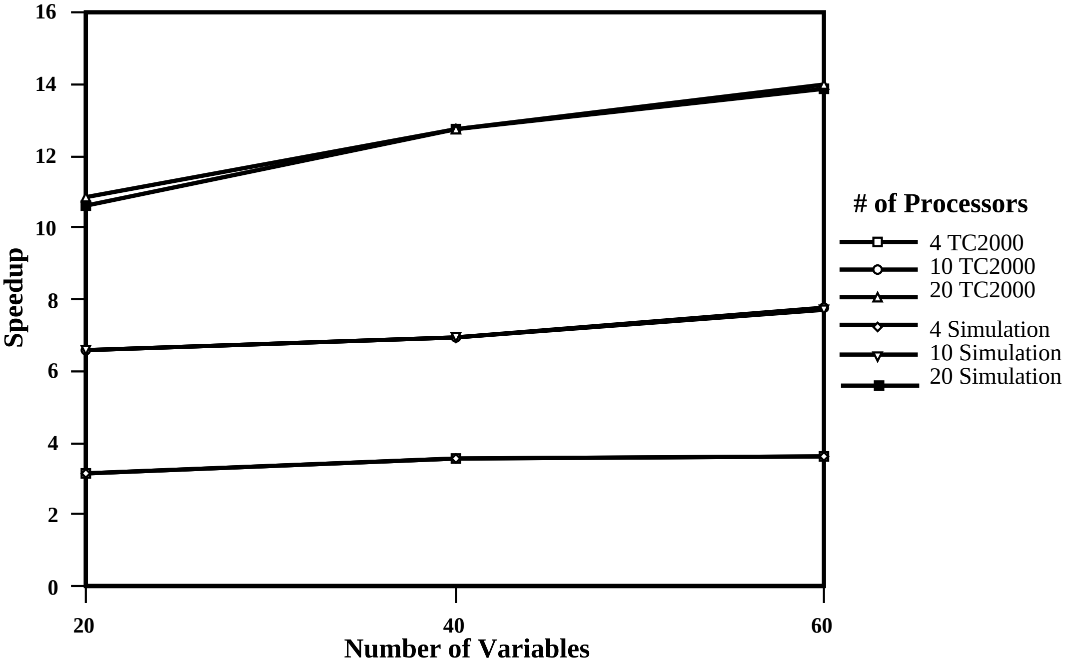

Figure 6.3: Speedup of SPAC-1 over AC-1 for a Hybrid Memory Machine

6.2 Results for SPAC-2 Algorithms

6.2.1 Shared Memory Machine

A comparison of processor cycles for the sequential AC-1 and SPAC-2 algorithms on the simulator and Sequent Symmetry is shown in Table 6.5.

Simulation of SPAC-2 for these networks on a four-processor computer predicts speedup of between 80 and 94% of linear speedup (efficiency), and actual computer execution indicates speedup within 2 to 5% of the simulated speedup (Figure 6.4). Simulation of a ten-processor computer predicts a speedup of between 68 and 82% of linear speedup, while actual runs show that the prediction is off by 6 to 8%. Simulation of a 20-processor computer predicts a speedup of between 60 and 75% of linear speedup, while actual runs show that the prediction is off by 1 to 8%.

Utilization and redundancy figures display similar trends. Utilization of SPAC-2 is quite high; we never observe processor utilization below 76% for SPAC-2. Utilization of SPAC-2 simulation is about 97, 93, and 90% for four, ten, and twenty processors, respectively. Utilization of SPAC-2 actual measurements is about 91, 81, and 79% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the SPAC-2 simulation is 5 to 55% higher than AC-1. The redundancy of work performed by the SPAC-2 on the Sequent Symmetry is 4 to 44% higher than AC-1.

Table 6.5: Processing Cycles of AC-1 and SPAC-2 Algorithms for a Shared Memory Machine

Measurement	Algorithm	20 Vars	40 Vars	60 Vars
Sequential	AC-1	1671	3592	5530
Simulation, 4 proc	SPAC-2	519	986	1473
Symmetry, 4 proc	SPAC-2	532	1021	1563
Simulation, 10 proc	SPAC-2	245	492	671
Symmetry, 10 proc	SPAC-2	274	539	738
Simulation, 20 proc	SPAC-2	140	258	370
Symmetry, 20 proc	SPAC-2	142	287	414

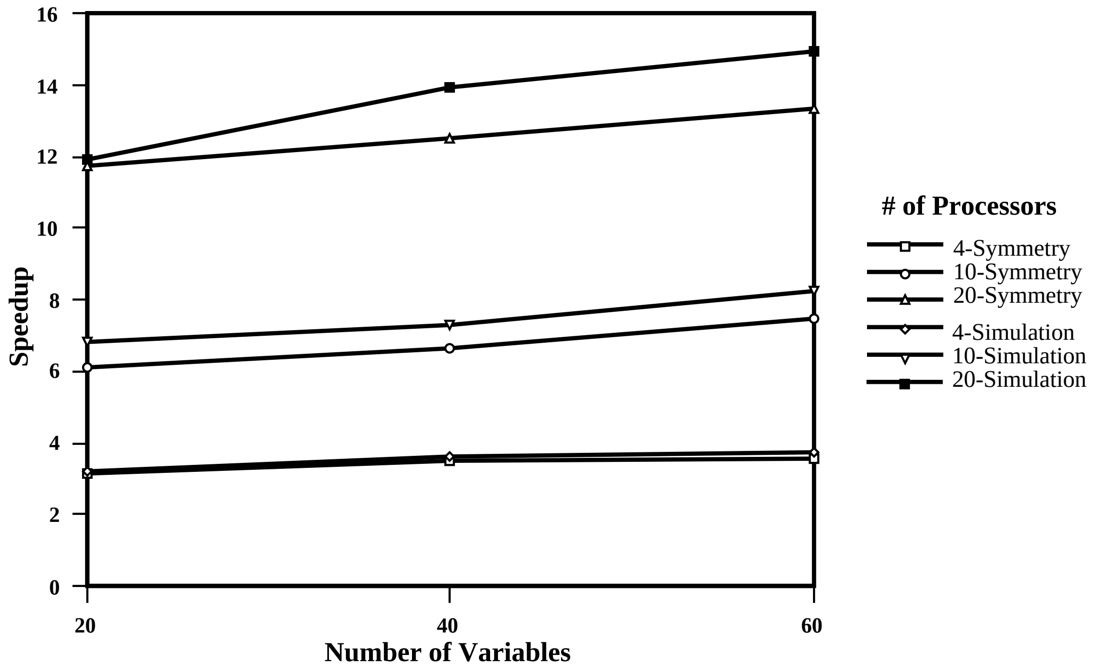

Figure 6.4: Speedup of SPAC-2 over AC-1 for a Shared Memory Machine

6.2.2 Distributed Memory Machine

A comparison of processor cycles for the sequential AC-1 and DSPAC-2 algorithms on the simulator and Intel iPSC/2 is shown in Table 6.6. 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-2 for these networks on a four-processor computer predicts speedup of between 72 and 85% of linear speedup (efficiency), and actual computer execution indicates speedup within 1 to 2% of the simulated speedup (Figure 6.5). Simulation of a ten-processor computer predicts a speedup of between 51 and 67% of linear speedup, but actual runs show that the prediction is off by 0 to 4%. Simulation of a twenty-processor computer predicts a speedup of between 42 and 58% of linear speedup, but actual runs show that the prediction is off by 0 to 4%.

Utilization and redundancy figures display similar trends. Utilization of DSPAC-2 simulation is about 91, 82, and 79% for four, ten, and twenty processors, respectively. Utilization of DSPAC-2 actual measurements is about 90, 80, and 75% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the DSPAC-2 simulation is 8 to 95% higher than AC-1. The redundancy of work performed by the DSPAC-2 on the iPSC/2 is 7 to 102% higher than AC-1.

Table 6.6: Processing Cycles of AC-1 and DSPAC-2 Algorithms for a Distributed Memory Machine

Measurement	Algorithm	Frequency	20 Vars	40 Vars	60 Vars
Sequential	AC-1		1671	3592	5530
Simulation, 4 proc	DSPAC-2	Value	648	1252	1850
Simulation, 4 proc	DSPAC-2	Arc	580	1093	1635
Simulation, 4 proc	DSPAC-2	Variable	593	1119	1610
Simulation, 4 proc	DSPAC-2	Processor	724	1624	2417
iPSC/2, 4 proc	DSPAC-2	Arc	563	1070	1613
Simulation, 10 proc	DSPAC-2	Value	351	724	1017
Simulation, 10 proc	DSPAC-2	Arc	327	610	822
Simulation, 10 proc	DSPAC-2	Variable	369	613	812
Simulation, 10 proc	DSPAC-2	Processor	398	799	1108
iPSC/2, 10 proc	DSPAC-2	Arc	328	597	775
Simulation, 20 proc	DSPAC-2	Value	198	378	575
Simulation, 20 proc	DSPAC-2	Arc	201	349	476
Simulation, 20 proc	DSPAC-2	Variable	253	394	489
Simulation, 20 proc	DSPAC-2	Processor	253	471	615
iPSC/2, 20 proc	DSPAC-2	Arc	225	351	471

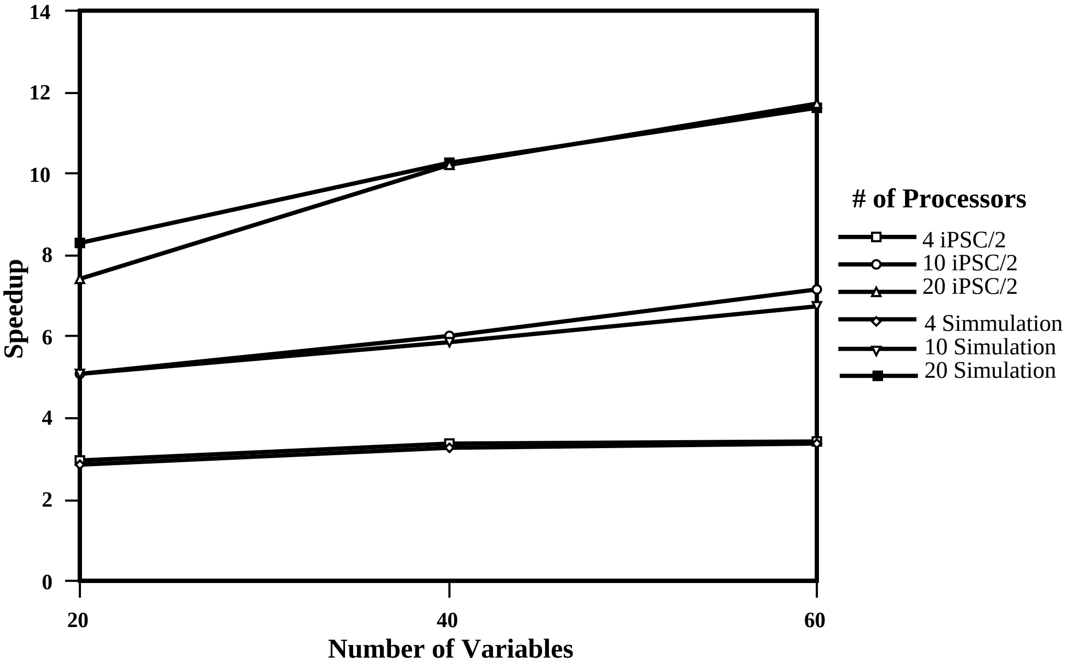

Figure 6.5: Speedup of DSPAC-2 over AC-1 for a Distributed Memory Machine

6.2.3 Hybrid Memory Machine

A comparison of processor cycles for the sequential AC-1 and SPAC-2 algorithms on the simulator and BBN TC2000 is shown in Table 6.7.

Simulation of SPAC-2 for these networks on a four-processor computer predicts speedup of between 80 and 94% of linear speedup (efficiency), and actual computer execution indicates speedup nearly equal to the simulated speedup (Figure 6.6). Simulation of a ten-processor computer predicts a speedup of between 68 and 82% of linear speedup, and again actual runs show that the prediction is nearly equal. Simulation of a twenty-processor computer predicts a speedup of between 60 and 75% of linear speedup, while actual runs show that the prediction is off by 0 to 1%.

Utilization and redundancy figures display similar trends. Utilization of SPAC-2 is quite high; we never observe processor utilization below 86% for SPAC-2. Utilization of SPAC-2 simulation is about 97, 93, and 90% for four, ten, and twenty processors, respectively. Utilization of SPAC-2 actual measurements is about 97, 93, and 89% for four, ten, and twenty processors, respectively. We observed that the redundancy of work performed by the SPAC-2 simulation is 5 to 55% higher than AC-1. The redundancy of work performed by the SPAC-2 on the BBN Butterfly is 5 to 58% higher than AC-1.

Table 6.7: Processing Cycles of AC-1 and SPAC-2 Algorithms for a Hybrid Memory Machine

Measurement	Algorithm	20 Vars	40 Vars	60 Vars
Sequential	AC-1	1671	3592	5530
Simulation, 4 proc	SPAC-2	519	986	1473
BBN TC2000, 4 proc	SPAC-2	520	989	1469
Simulation, 10 proc	SPAC-2	245	492	671
BBN TC2000, 10 proc	SPAC-2	246	494	672
Simulation, 20 proc	SPAC-2	140	258	370
BBN TC2000, 20 proc	SPAC-2	142	263	371

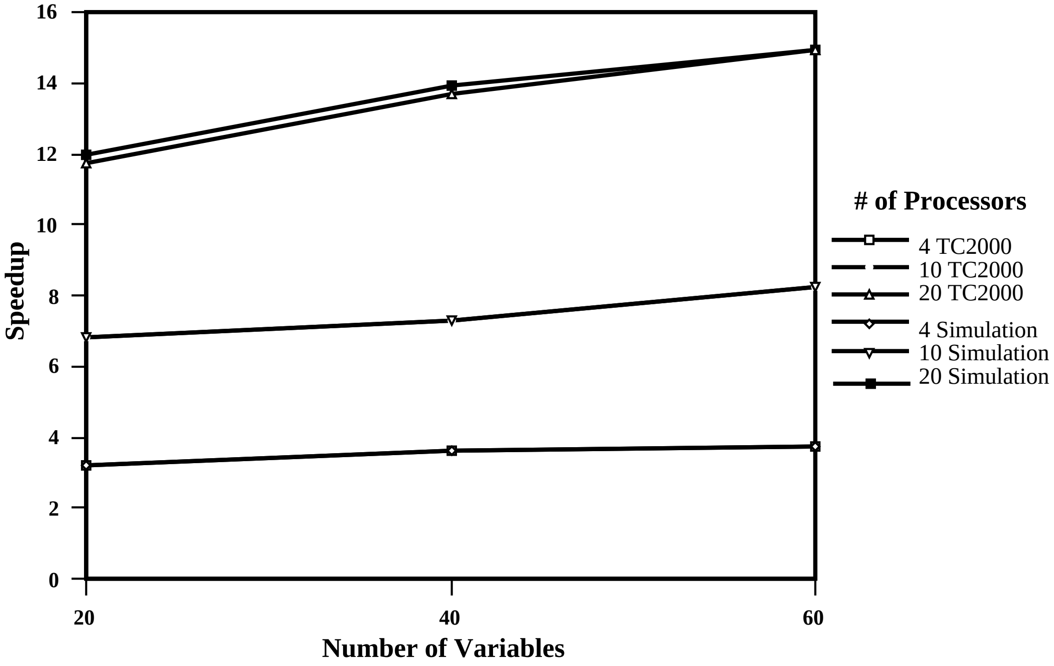

Figure 6.6: Speedup of SPAC-2 over AC-1 for a Hybrid Memory Machine