SIMULATION OF GENERIC MULTIPROCESSOR CONFIGURATIONS FOR ASYNCHRONOUS ALGORITHMS

James M. Conrad
conrad@csl36h.csl.ncsu.edu
Dr. Dharma P. Agrawal
dpa@csl36h.csl.ncsu.edu
North Carolina State University, (919) 515-3984
Department of Electrical and Computer Engineering
Box 7911
Raleigh, NC 27695-7911
December 17, 1992

Correspondence should be directed to Dr. Dharma P. Agrawal.

This work has been supported in part by the Army Research Office, under grant DAAL03-89-K-0142.

Advance Copy
International Journal in Computer Simulation
Special Issue on Multiprocessor Networks
1993

ABSTRACT

A multiprocessor simulator is a useful tool to predict a parallel algorithm's behavior. Most existing parallel system simulators assume a particular architecture or simulate algorithms at a machine instruction level. This paper describes a simple but elegant parallel processing simulator environment which could represent a general class of multiple instruction stream - multiple data stream (MIMD) architectures. The performance of a class of asynchronous parallel algorithm is examined, and as an example, an arc consistency algorithm is considered in detail. Simulation results indicate that if our algorithm is programmed appropriately and run on a parallel machine, good speedup and processor utilization, and hence good performance, can be achieved. Actual measurements taken from runs on a Sequent Symmetry computer verify this premise.

Keywords:

Arc Consistency, Asynchronous Parallel Algorithm, Constraint Network, Multiprocessor, Redundancy, Simulation, Speedup, Utilization.

1 Introduction

Much of the existing work on parallel algorithms has been devoted to synchronous algorithms, wherein a strict synchronism between successive steps of a parallel computation is observed, and processing of subtasks is done in lockstep [1]. In this work, we study a class of algorithms for multiprocessors called *asynchronous parallel algorithms* [2] (hereafter called asynchronous algorithms). These algorithms allow some degree of asynchronism between various steps or successive iterations. The most important characteristic of asynchronous algorithms is that whenever any processor needs a global variable, it reads the most currently available value and continues execution using that information. The main objective of this paper is to verify that the specific steps of these algorithms can be efficiently programmed for many different types of multiprocessors.

Parallel computers can be classified according to the instruction stream sequence executed by the computer and the data stream sequence used by the individual processors [1]. Stream classification of parallel systems includes *Single Instruction stream - Multiple Data stream* (SIMD) computers and *Multiple Instruction stream - Multiple Data stream* (MIMD) computers. It may be noted that asynchronous algorithms in general can only be run on the MIMD class of multiprocessors and not on SIMD multiprocessors. Parallel computers can also be classified according to three possible memory architectures: shared, distributed, and hybrid.

To examine the effect of running asynchronous algorithms, we defined three specific goals:

1. The simulation should be able to represent any MIMD multiprocessor architecture.
2. The simulation should be capable of representing the memory operations of either shared, distributed, or hybrid memory architecture.
3. The simulation should be able to measure a relatively large and identifiable amount of work performed by an algorithm, but not measure work instruction by instruction.

We considered many experimental parallel processing simulators [3, 4, 5] and several commercial simulators [4, 6] and observed that either they were too complex, or they simulated one specific architecture, or they simulated the problem at a machine instruction level. Therefore, we decided to develop our own simulator. With a simple yet elegant simulator we are able to represent any MIMD computer architecture, represent communications overhead of a distributed memory computer, examine the work performed by each processor of the simulation, and collect data for overall algorithm performance evaluation. We can also vary the frequency of interprocessor communication and, thereby, meet all three goals described above.

We have compared our shared memory simulation results with measurements taken on a shared memory Sequent Symmetry computer. Results indicate that the speedup and processor utilization from the simulation are fairly close to that of the actual runs.

This paper is organized as follows. Section 2 defines the types of generic parallel processing architectures we simulate. Section 3 describes the general characteristics of asynchronous algorithms. Section 4 presents a Static Parallel Arc Consistency algorithm as a specific asynchronous algorithm under consideration. Section 5 defines the requirements for achieving the goals of the simulation project and describes how simulating shared, distributed, and hybrid memory accesses are achieved. Section 6 introduces the measurement metrics and presents simulation results and actual measurements for the example algorithm. Section 6 includes the conclusion.

2 Parallel Computer Architectures

At a given instant, a MIMD parallel processing computer executes a different instruction on each of its processing elements. The processors may, however, require communication with each other or sharing of resources. How the processors access memory or input/output devices totally depends on the underlying computer architecture. We classify MIMD computer architectures relative to the location and distribution of their memory modules. The three possible schemes of MIMD computers are: shared memory, distributed memory, and hybrid memory.

Figure 1 shows a block diagram of a shared memory architecture with four processors and four memory modules. The switching network is designed in such a way that each processor can access any of the memory modules. The switching network can be a simple bus, a complex crossbar, or a multistage interconnection network. Communication between processors is handled by writing to and reading from the shared memory locations.

Figure 2 shows a block diagram, and Figure 3 an example, of a distributed memory architecture. Each processor has its own private memory. Communication takes place directly between the processors via the switching network. If one processor wants a variable value from the memory module of another processor, the requesting processor submits a message to the other processor. The remote processor retrieves the information from its private memory and sends it to the requesting processor. One of the many network topologies, such as ring, mesh, or hypercube, could provide the interprocessor communication.

Figure 4 shows an example of a hybrid memory architecture, which combines the inherent characteristics of both shared and distributed memory architectures. The private memory can serve as a fast memory for computer instructions and local data, while the shared memory can be used to store global variables used by some or all the processors.

3 Parallel Algorithms

Kung [2] defines a parallel algorithm for multiprocessors as a collection of concurrent *processes* that may operate simultaneously to solve a given problem. A process should be an

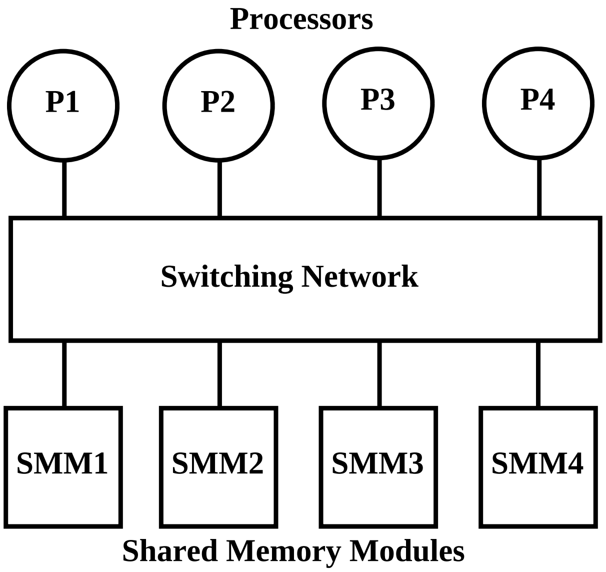

Figure 1: General Block Diagram of a Shared Memory Computer Architecture

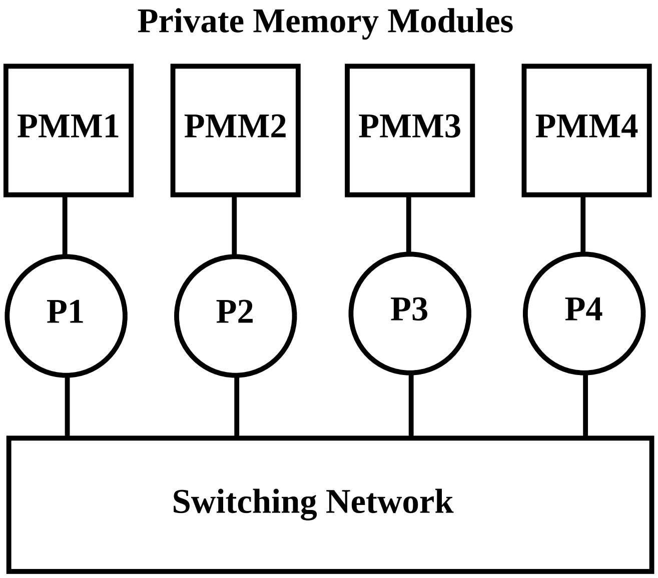

Figure 2: General Block Diagram of a Distributed Memory Computer Architecture

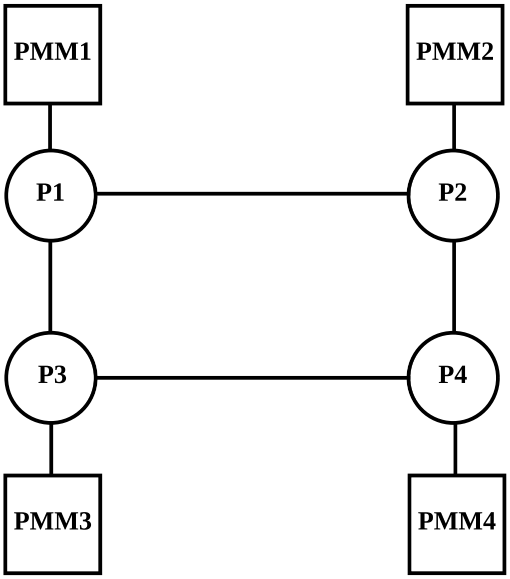

Figure 3: Example of a Distributed Memory Computer Architecture

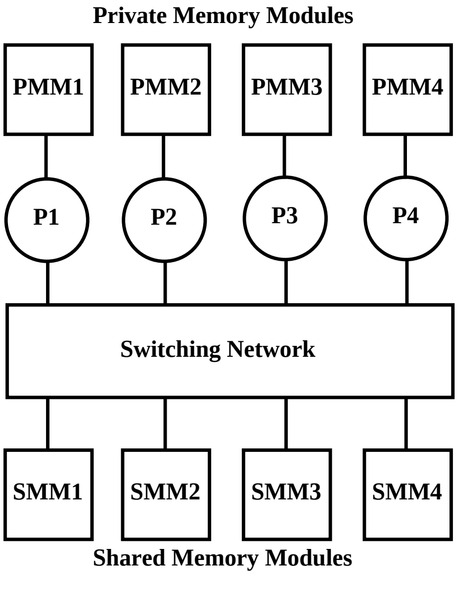

Figure 4: General Block Diagram of a Hybrid Memory Computer Architecture

identifiable subset of the entire problem, but can execute on more than one processor.

A problem usually requires interaction between the processes. If the program controlling a process needs to communicate with another process, it can send or receive information via a message or it can read or store information via a globally available variable. Such points in the program are *interaction points*, and these points divide a process into *stages*.

A *synchronized parallel algorithm* consists of processes where "there exists a process such that some stage of the process is not activated until another process has finished a certain portion of its program"[2]. For example, the processes $A + B$ and $C + D$ are part of the larger process $X = (A + B) \times (C + D)$. The process to compute X requires that the $A + B$ and $C + D$ processes both complete, therefore the X process is synchronized with the other two.

An asynchronous algorithm has the following properties [2]:

1. A set of global variables is available to all processes.
2. When a stage of a process is complete, the process first reads some global variables. Then, based on the values of the variables and the results from the previous process stage, the process modifies some global variables, and then activates the next stage or terminates itself.

The most important characteristic of asynchronous algorithms is that its processes never wait for inputs, therefore, a process cannot be deadlocked waiting for synchronization from another process. Although asynchronous algorithms are typically executed on shared memory computers, we have modified these algorithms for a distributed memory environment. We describe the needed changes in Section 5.

4 Arc Consistency as an Example of an Asynchronous Algorithm

Constraint satisfaction is a branch of Artificial Intelligence where one must find values for a group of variables, or parameters. Mackworth [7] describes constraint satisfaction problems as "those in which one has a set of variables, each to be instantiated in an associated domain, and a set of Boolean constraints limiting the set of allowed values for specified subsets of the variables."

4.1 Arc Consistency Algorithms

We examine a class of constraint satisfaction pre-processing algorithms called *arc consistency algorithms*. These algorithms remove values from variable domains which can never participate in a solution to the constraint satisfaction problem. To represent these tasks in mathematical form, we need to define some terms:

- A *variable*, v_i, also called a parameter, is a unit which is assigned a meaning, or value.
- A *finite domain*, D_i, is a list of possible values that variable v_i can be given.
- A *value*, l_{i_x}, also called a label, is a specific assignment given to variable v_i from a list of possible items, that is, $l_{i_x} \in D_i$.
- A *binary constraint*, R^2_{ij}, is a relation by which if a value l_{i_x} from domain D_i is assigned to variable v_i, **and** value l_{j_y} from domain D_j is assigned to variable v_j, then the constraint R^2_{ij} is true if the relation is satisfied, that is $(l_{i_x}, l_{j_y}) \in R^2_{ij}$.

The arc $arc(i,j)$ is *arc consistent* if, for each value in D_i, there is at least one value in D_j that satisfies the constraint of the arc. Arcs can be made consistent by performing the operation:

$$\forall l_{i_x},\ \ D_i \leftarrow D_i \cup \{l_{i_x} | (\exists l_{j_y})(l_{j_y} \in D_j) \wedge R^2_{ij}(l_{i_x}, l_{j_y})\} \tag{1}$$

Arc consistency algorithms typically do not find a single assignment set for a constraint problem. Therefore, a constraint satisfaction algorithm, like backtracking, must be used after arc consistency to search for a specific assignment set.

A *constraint network* is a graphical representation of a specific constraint satisfaction problem. Figure 5 shows an example of a four variable constraint network. The circles represent the finite domain variables of the problem, and the boxes represent the constraints between the variables. Here the variable domains are single letters, and the constraints are letter pairs which satisfy an assignment of the two variables. An *arc* is a directed edge from one variable to another variable. A *consistency check* is a comparison of a possible value of one variable with the possible value of another variable. If we look at Figure 5, one consistency check includes assigning the value A to Variable 1 and the value D to Variable 2 and checking if this pair of values satisfies Constraint 1. The network in Figure 5 has only one set of assignments which satisfy the constraints: Variable 1 is assigned A, Variable 2 is assigned F, Variable 3 is assigned H, and Variable 4 is assigned J.

Mackworth and Freuder [8] proved the complexity bounds of sequential arc consistency algorithms using consistency checks as the unit of measure. The number of checks depends on the number of variables (n), values per variable (a), and edges (e) in the network. It is desirable to embrace this complexity measure as an important parameter, and hence, consistency checks are counted. As the process of preparing and executing a consistency check requires several machine instructions, we decided to define our identifiable amount of work as single consistency check for each processor. We shall further define one processor *cycle* as the time needed to perform one consistency check.

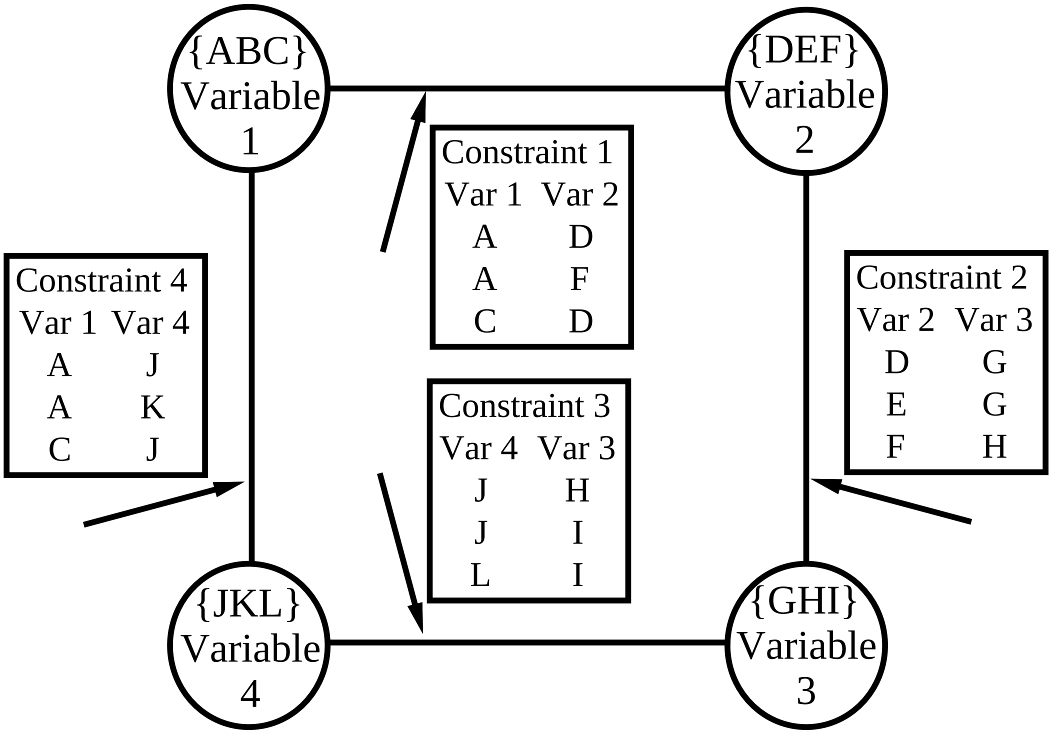

Figure 5: A Four-Variable Constraint Network

4.2 Arc Consistency Sequential Algorithm #1

Mackworth's [9] Arc Consistency sequential algorithm #1 (AC-1) is a simple algorithm to ensure arc consistency. Every arc of the constraint is checked. The AC-1 algorithm uses the Revise subroutine, which examines an arc by choosing a variable value and checking if the variable connected by the arc has at least one value which is consistent with the value for this variable. This is done for each value in the first variable's domain. If a value is removed from a variable's domain, a *change* bit is set with a logical "1" and AC-1 rechecks each of it's arcs at least one more time. AC-1 continues to recheck it's arcs until no more values are removed.

The worst case complexity of AC-1 is reported by Mackworth and Freuder [8] as $\mathbf{O}(a^3ne)$ consistency checks. If we examine the constraint network of Figure 5, the sequential AC-1 algorithm makes 58 consistency checks to find a single solution assignment. This sample trace of the consistency checks for AC-1 is shown in Figure 6. We show a check mark ($\surd$) next to a successful consistency check, and a "×" next to an unsuccessful check. We also show if the change bit is set to logical "1" ("on"). The domain changes are shown in the third column.

4.3 Parallel Arc Consistency Algorithms

Samal and Henderson [10] proposed a parallel version of the AC-1 algorithm, but their version only executes on a shared memory architecture. Gu and Sosic [11] also proposed a parallel version of the AC-1 algorithm, but their version is only valid for a special-purpose SIMD computer. We have developed a true asynchronous algorithm called the Static Parallel Arc Consistency algorithm (SPAC), which can be executed on any MIMD computer architecture.

Our SPAC algorithm is a modification of Mackworth's AC-1 algorithm [12] with several changes which lead to performance enhancements. Each processor is assigned a part of a constraint network, and each processor ensures local arc consistency of it assigned part of the network. SPAC utilizes a procedure called Static Parallel Revise Boolean function (Sprevise), which is similar to Mackworth's Revise function. Sprevise will check an arc and remove all variable values which could never participate in a solution. If a value is removed, all processors which use the value are informed and will need to recheck their assigned arcs. The end of the SPAC procedure occurs only when all processors have verified local arc consistency. Once each processor has verified local arc consistency, it accesses a globally available counting variable indicating the number of active processors. If the variable indicates that at least one other processor is still working on arc consistency, the processor becomes idle. An idle processor restarts when an active processor connected by a constraint deletes a variable value used by the idle processor. A restarted processor tries to ensure local arc consistency again.

The SPAC algorithm also will stops if a domain of a variable is empty because no solution could ever be found.

Clock Cycle	Con Chks	Chg Bit	Domain Values		Clock Cycle	Con Chks	Chg Bit	Domain Values
			ABC DEF GHI JKL		30	AD√		***Loop 2***
1	AD√		***Loop 1***		31	CD√		
2	BD×				32	AJ√		
3	BE×				33	CJ√		
4	BF×	on	AC DEF GHI JKL		34	DA√		
5	cd√	on			35	FA√		
6	AJ√	on			36	DH×	on	AC F H J
7	CJ√	on			37	FH√	on	
8	DA√	on			38	HF√	on	
9	EA×	on			39	HJ√	on	
10	EC×	on	AC DF GHI JKL		40	JA√	on	
11	FA√	on			41	JH√		
12	DG√	on			42	AF√		***Loop 3***
13	FG×	on			43	CF×	on	A F H J
14	FH√	on			44	AJ√	on	
15	GD√	on			45	FA√	on	
16	HD×	on			46	FH√	on	
17	HF√	on			47	HF√	on	
18	ID×	on			48	HJ√	on	
19	IF×	on	AC DF GH JKL		49	JF√	on	
20	GJ×	on			50	JH√	on	
21	GK×	on			51	AF√		***Loop 4***
22	GL×	on	AC DF H JKL		52	AJ√		
23	HJ√	on			53	FA√		
24	JA√	on			54	FH√		
25	KA√	on			55	HF√		
26	LA×	on			56	HJ√		
27	LC×	on	AC DF H JK		57	JF√		
28	JH√	on			58	JH√		
29	KH×	on	AC DF H J					

Figure 6: Consistency Check Trace of Sequential AC-1 Algorithm

Since worst case time complexity of the AC-1 algorithm is $\mathbf{O}(a^3ne)$ consistency checks, and since we distribute work to p processors (such that $p \leq n$), the worst case time complexity of the SPAC algorithm is $\mathbf{O}(a^3ne/p)$ consistency checks. The worst case processing complexity of work performed is p processors times the time complexity, or $\mathbf{O}(a^3ne)$. This, of course, is the extreme worse case. On the average, the actual work performed is quite less. If we examine the constraint network in Figure 5, the parallel SPAC algorithm makes 55 consistency checks to find the single solution assignment, but does so in 15 cycles.

The parallel trace (Figure 7) shows the work performed by each processor. Here, each processor is assigned one variable and all the directed arcs that originate from that variable. Consistency checks are carried out in the same manner as described earlier. The change bit is somewhat different. A processor will alter only other processors' change bits. The processor altering this bit is noted in the "change bit" column (this information is not recorded by a processor, it is only provided here for added readability). The domain changes column shows the variable values for each processor.

Clk Cyc	Proc 1	Chg Bit	Dom 1 Value	Proc 2	Chg Bit	Dom 2 Value	Proc 3	Chg Bit	Dom 3 Value	Proc 4	Chg Bit	Dom 4 Value
			ABC			DEF			GHI			JKL
1	AD√	***	Loop 1	DA√	***	Loop 1	GD√	***	Loop 1	JA√	***	Loop 1
2	BD×			EA×			HD×			KA√		
3	BE×			EB×			HE×			LA×		
4	BF×	on^2	AC	EC×	on^1	DF	HF√	on^2		LB×	on^1	
5	CD√	on^4		FA√	*on*		ID×	on^4		LC×	*on*	JK
6	AJ√	*on*		DG√	on^3		IF×	*on*	GH	JG×	on^3	
7	CJ√	*on*		FG×	*on*		GJ×	*on*		JH√	*on*	
8	AD√	***	Loop 2	FH√	on^3		GK×	*on*	H	KG×	on^3	
9	CD√	on^4		DA√	***	Loop 2	HJ√	on^4		KH×	*on*	J
10	AJ√	*on*		FA√			HD×	***	Loop 2	JA√	***	Loop 2
11	CJ√	on^2		DH×		F	HF√	on^2		JH√		
12	AF√	***	Loop 3	FH√			HJ√	*on*		idle		
13	CF×		A	idle	on^1		HF√	***	Loop 3	idle	on^1	
14	AJ√			FA√	***	Loop 3	HJ√			JA√	***	Loop 3
15	idle			FH√			idle			JH√		

Figure 7: Consistency Check Trace of SPAC Algorithm

5 Parallel Processing Simulation Requirements

The first goal of our simulation project is to consider the operations performed by the processors and not the underlying interconnections between processors. We assume that, regardless of the distributed memory computer's processor interconnections, communication time is the same. This assumption is similar to the processor to processor communication of the Intel iPSC/2, where message passing from one processor to all other processors is virtually identical [13]. We are concerned, though, with the communications and setup time associated with sending and receiving a message.

We also want to make our simulator flexible enough to allow any arbitrary number of processors.

The second goal is to have a capability of simulating a shared, distributed, or hybrid memory architecture. The algorithm design and execution should be independent of any specific memory architecture and the code should be easily portable to different parallel computers (although one should fine-tune the code to exploit features of each computer). This enables us to concentrate on the algorithm and we need not worry about such details as the memory accesses handled by each computer's operating system. This does not mean that we are not interested in the memory architecture. We differentiate between shared and distributed memory, but by minimizing the number of writable variables we have lessened the memory access concern.

The third goal is that the simulation measure a large and identifiable amount of work performed by our algorithm. Even though measurement of the individual computer instructions would represent a true parallel processing simulator, we do not want such a fine-grain measurement. One step of our algorithm, therefore, represents many computer instructions. We are interested in measuring the work at a larger grain level, such as a consistency check in an arc consistency algorithm. A single consistency check is easily measured and accurately represents several steps of our algorithm. In fact, consistency checks provide an identifiable metric which can be used to compare all arc consistency algorithms (sequential and parallel). Once we have executed these algorithms on our simulator, we can identify those which show the greatest promise of speedup, and implement the algorithms on actual multiprocessors.

To implement a parallel processing simulator on a uniprocessor and to meet our goals, we employ a "time-shared" scheme. In this scheme, during a *time slice*, one processor performs work equivalent to one consistency check. A cycle is defined as the time where every processor is given one time slice. At the end of a cycle, the memory of our multiprocessing system is updated. Note that a given processor may not have any work to perform during a time slice and could remain idle.

During each processor's time slice, the following operations are performed (refer to Figure 8):

Step 1 The previous-state information (i.e. counting values or loop variables) is read from the simulator's memory. In an actual multiprocessing application, this information is not stored in memory, but is located in the registers of each processor.

Step 2 From the previous-state information, the next identifiable amount of work is defined (in our example, this work is one consistency check).

Step 3 The work consisting of reading a "multiprocessor's memory" and, if needed, writing a change to the "multiprocessor's memory" is performed. This change is actually stored in a temporary memory location so the change will not be available to "processors" executing their time slice later.

Step 4 The current state information is stored into the simulator's memory. Again, this step is necessary in a simulated model but not in a true multiprocessor system.

The algorithm stops when all processors are idle. This can be determined in a shared memory computer by using a single variable to indicate how many processors are active. In a distributed memory computer, one processor, or the host computer, will hold this variable.

```
while at least one processor is active and has work to perform
    for each processor
        if idle processor has no work to perform, remain idle
            else if idle processor has work to perform
                set state information to initial state
                perform time slice information
                if all work complete, become idle processor
            else
                retrieve state information
                perform time slice information
                if all work complete, become idle processor
    move the temporary variables over to the permanent variables
```

Figure 8: Simulation Pseudo-code

5.1 Simulating Variables

Our algorithms use four types of variables:

1. Those that are read-only and used by all processors.
2. Those that are read by all processors, but written by only one processor.
3. Those that are read by one processor, but written by all processors.
4. Those that are read and written by all processors.

The operations using these variables differ depending on the architecture we simulate.

5.2 Simulating a Shared Memory Architecture

The shared memory simulation option assumes that the access time to all memory locations is identical. The treatment of each of the above variables is:

Read-only variables are read by all processors and may be loaded into cache memory. These variables make up the bulk of the algorithm's variables (almost 98%). These variables are simulated simply by making them available to all simulated processors.

Read-by-all, write-by-one variables remain in shared memory. These variables are always available, even if a processor has read the variable with the intention of changing the value. This is possible because of the nature of asynchronous algorithms; the variable value does not have to be current. These variables make up about two percent of the algorithm's variables. These are the variables temporarily stored until all processors have executed their consistency check.

Read-by-one, write-by-all variables implement a *test-and-set* operation [14]. 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 reports 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 type nondivisible operation. This variable is simply available to all the processors all the time, but the value is only used for logical operations (looping conditions) at the end of a cycle.

5.3 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.

Read-by-all, write-by-one variables 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. Four different possibilities are considered in this work:

1. A value-based simulation sends a message every time a value is removed a domain.
2. An arc-based simulation sends a message if at least one value is removed after one entire arc has been checked.
3. A variable-based simulation sends a message if at least one value is removed after all the arc originating from a single variable have been checked.
4. A processor-based simulation sends a message after all constraint variables assigned to a processor have been 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 have been received from another processor, called "global memory". We simulate distributed memory and processor to processor communication as follows:

1. A processor will use its own "local memory" to read values and make updates. It will use "global memory" to read variables which are changed by other processors.
2. When a processor determines that it must send a variable change, it puts the value in the "communication memory".
3. The simulator will move the change from "communication memory" to global memory after one time slice has passed. The simulator will also take one time slice to "receive" this value.

Each processor has one read-by-one, write-by-all variable. If a processor "receives" a message, it updates that variable. It, therefore, performs its own test-and-set operation. These are handled the same way as done for 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 will accurately indicate 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 too complex to simulate accurately. When we implement this on an actual machine, we may use a "hierarchical" scheme. Several processors use one parent processor to store this value, and that one processor then reports to its parent processor, and so on. This will not reduce the number of messages, but will spread the message activity across all involved processors.

5.4 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 will display identical simulation results.

Our parallel algorithm relies on a static distribution of work [1]. We can either assign one constraint variable to each processor or any number of constraint variables to each processor. With one constraint variable per processor, this distribution is fast and easy, but we have also observed that the distribution of work is not entirely equitable. If we assign several constraint variables to each processor, we distribute the variables by ensuring the number of arcs per processor is approximately equal.

We created several different constraint networks to run using both the sequential and parallel algorithms. These constraint networks are randomly generated. A typical constraint network may have any number of solution sets. Since arc consistency is a pre-processing algorithm for constraint satisfaction search algorithms, we find all possible solution assignments to the network. For this paper we have created networks with only one solution set to ensure our measurements are comparing similar processes.

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

5.5 Parallel Processor Application

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

The read-only variables are read from a file prior to execution, are duplicated and stored in each processor's private memory, and are cachable. The constraint relations are never updated, and hence, this data can be copied into a processor's cache with no coherency problems.

All other variables are stored in shared memory and can be updated by all processors. We found that we did not need to use the test-and-set operation for the read-by-one, write-by-all variables because of the asynchronous nature of our algorithm. Earlier trials [15] found that locking these variables caused large, unnecessary delays for data accesses.

The only locked variable was one read/write-by-all counter variable.

6 Simulation Performance Results

From the performance point of view, we limit our measurements to three algorithmic parameters: speedup, utilization, and redundancy of work performed.

By the common definition of speedup [1]:

$$speedup = S(p) = \frac{time(sequential\, algorithm)}{time(parallel\, algorithm)} \tag{2}$$

We can interpret "time" as the number of consistency checks for the sequential algorithm and time slice cycles for the parallel algorithm. For the parallel algorithm, we found that one of the processors is never idle, so the number of cycles is equal to the maximum number of consistency checks performed by any of the processors. For example, the speedup of the constraint network shown in Figure 5 is 58/15 = 3.9. Efficiency can be viewed as the percentage of speedup realized of possible speedup. Efficiency is simply speedup divided by the number of processors used for the parallel computation.

Utilization of the parallel algorithm is defined as [16]:

$$utilization = U(p) = \frac{total\,\#\,time\,slices\,where\,work\,was\,performed}{total\,\#\,time\,slice\,cycles\ \times\ \#\,processors} \tag{3}$$

For example, the utilization of the constraint network shown in Figure 5 is 55/60 = 91.7%.

We define the redundancy [1] of work performed as:

$$redundancy = R(p) = \frac{\#consistency\,checks\,(parallel\,algorithm)}{\#consistency\,checks\,(sequential\,algorithm)} \tag{4}$$

The redundancy is a measure of how many more consistency checks are performed executing the parallel algorithm than the sequential algorithm. This parameter shows the extra work, or superfluous computation overhead [17], performed by the parallel algorithm. For example, the redundancy of the constraint network shown in Figure 5 is observed to be 55/58 = 0.95. Although the redundancy of this small sample network is below 1.0, we find that the figure is typically above 1.15 for shared memory simulation and above 1.30 for distributed memory simulation.

The results of our measurements suggest that our parallel algorithm shows great promise, and the simulator reports results very close to the trials of the Sequent computer. We select networks where the AC-1 and SPAC algorithms removes all but one value from each variable (which represents the one and only solution to our constraint problem). Each point in our figures represent the average value of ten trial networks. These figures show the data for constraint networks with a consistency check success rate of 15% in a network of medium connectivity.

Simulation of these networks on a shared memory architecture indicate speedup between 60 and 94% of possible speedup (Figure 9). Speedup is higher in the trials with a smaller numbers of processors because processors do not sit idle for any length of time. This is evident by examining processor utilization. Utilization of processors of a shared memory architecture is between 87 and 98%. The work performed by the parallel algorithm is not much more than that performed by the sequential algorithm. The redundancy is between 1.05 and 1.55.

The actual measurements of speedup deviate slightly from simulation trials. Recorded speedup is between 59 and 88% of possible speedup, with the actual measurements between 1 and 7% below simulated possible speedup values. Utilization and redundancy values are also similar.

Simulation of these networks on a ten processor distributed memory architecture indicates speedup between 42 and 73% of p one parallel algorithm, while others could be simulated in an eq
effective way. The asynchronous nature of the algorithm indicates good promise for its f
use in many applications. We plan to experiment with similar algorithms for simulatio
for actual machine implementation utilizing an Intel iPSC/2 hypercube and a BBN Butt
TC2000.

Acknowledgements

The authors would like to thank the Advanced Computing Research Facility, Mathem

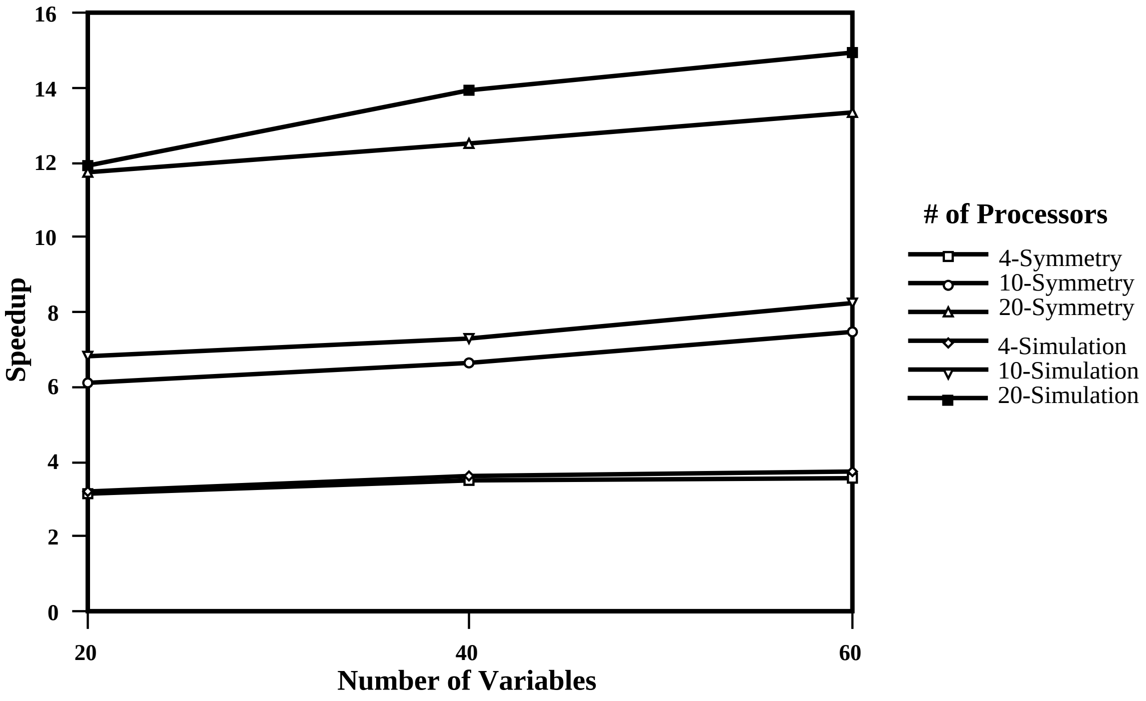

Figure 9: Speedup of Parallel Algorithm as a Percentage of Possible Speedup for a Shared Memory Architecture - Simulated and Actual

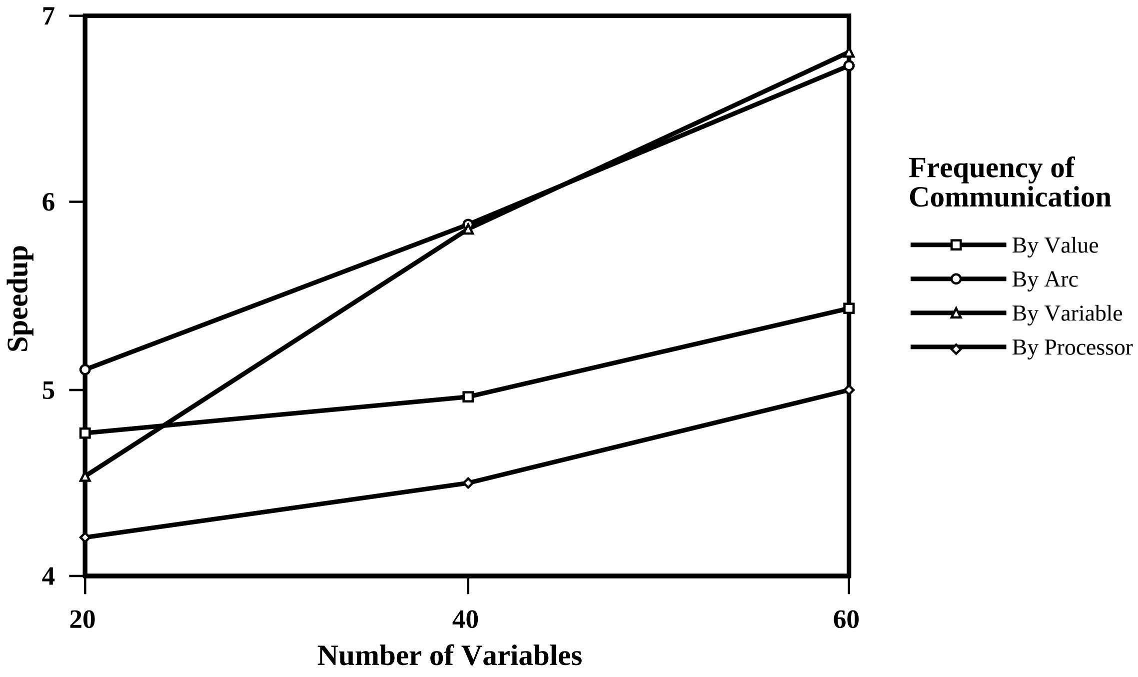

Figure 10: Speedup of Parallel Algorithm as a Percentage of Possible Speedup for a Distributed Memory Architecture - 10 Processors

and Computer Science Division, Argonne National Lab for use of their Sequent Symmetry Computer.

References

[1] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[2] H.T. Kung, "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," in *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub, editor, Academic Press, 1976, pp. 153-200.

[3] R.M. Fujimoto and W.B. Campbell, "Efficient Instruction Level Simulation of Computers," *Transactions of the Society for Computer Simulation*, Vol. 5, No. 2, pp. 109-123, April 1988.

[4] N.A. Kheir, ed., *Systems Modeling and Computer Simulation*, Marcel Dekker, 1988.

[5] A.E. Love, Jr. and M.F. Aburdene, "Simulation of a Distributed Algorithm using OCCAM," *Simulation*, Vol. 55, No. 2, pp. 86-95, August 1990.

[6] Science Applications International Corporation, *ANSPEC: An Actor Oriented Environment for PDP Specification*, 1989.

[7] A.K. Mackworth, "Constraint Satisfaction," In *Encyclopedia of Artificial Intelligence*, S. Shapiro - editor, Wiley, pp. 205-211, 1987.

[8] A.K. Mackworth and E.C. Freuder, "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence*, No. 25, pp. 65-74, 1985.

[9] A.K. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence*, No. 8, pp. 99-118, 1977.

[10] A. Samal and T. Henderson, "Parallel Consistent Labeling Algorithms," *International Journal of Parallel Programming*, Vol. 16, No. 5, pp. 341-364, October 1987.

[11] J. Gu and R. Sosic, "A Parallel, Optimal Arc Consistency Algorithm," *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, pp. 599-600, August 1990.

[12] J.M. Conrad, D.R. Bahler, and J.A. Bowen, "Static Parallel Arc Consistency in Constraint Satisfaction," to appear in the *Proceedings of the 1991 Int'l Symp. on Meth. for Intelligent Systems*, Charlotte, NC, October 1991.

[13] Intel Scientific Computers Corporation, *iPSC/2-iPSC/860 User's Guide*, Beaverton, OR, 1991.

[14] R.N. Ibbett and N.P. Topham, *Architecture of High Performance Computers Volume II*, Springer-Verlag, 1989.

[15] J.M. Conrad and D.P. Agrawal, "Performance of an Asynchronous Parallel Algorithm on a Generic Multiprocessor Simulator," to appear in the *Proceedings of the Pittsburgh Conference on Simulation and Modeling*, Pittsburgh, PA, May 1991.

[16] D.P. Agrawal, *Advanced Computer Architecture (Tutorial Text)*, IEEE Computer Society Press, 386 pages, 1986.

[17] H. Burkhart and R. Millen, "Performance-Measurement Tools in a Multiprocessor Environment," *IEEE Transactions on Computers*, Vol. 38, No. 5, pp. 725-737, May 1989.