PERFORMANCE OF AN ASYNCHRONOUS PARALLEL ALGORITHM ON A GENERIC MULTIPROCESSOR SIMULATOR

James M. Conrad, conrad@csl36h.csl.ncsu.edu
Dr. Dharma P. Agrawal, dpa@csl36h.csl.ncsu.edu
Department of Electrical and Computer Engineering
North Carolina State University
Box 7911, Raleigh, NC 27695-7911
(919) 737-3984

ABSTRACT

To predict a parallel algorithm's behavior without running on a multiprocessor, it is desirable to try the algorithm on a simulator. 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) architecture. The performance of a class of asynchronous parallel algorithms is examined, and as an example, an arc consistency algorithm is considered in detail. Simulation results are compared with actual measurements taken from runs on Sequent Balance and Symmetry computers. The closeness of the results indicate that, for a small number of processors, our simple simulator is adequate in accurately simulating the behavior of a generic multiprocessor.

Keywords: Arc Consistency, Asynchronous Algorithm, Constraint Network, Multiprocessor, Simulation, Speedup, Utilization, Work Ratio.

INTRODUCTION

Most 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* [6]. These algorithms allow some degree of asynchronism between various steps or successive iterations. Whenever any processor needs a value of a global variable, it reads the most currently available value and continues execution using that information. The main objectives of this paper is to check if the steps of such algorithms can be efficiently programmed on many different types of multiprocessors, and to verify this by comparing simulation results with actual runs.

Parallel computers can be classified according to the instruction stream sequence, the data stream sequence, and the memory architecture. Stream classification includes *Single Instruction stream - Multiple Data stream* (SIMD) computers and *Multiple Instruction stream - Multiple Data stream* (MIMD) computers. The three possible memory architectures include shared memory, distributed memory, and hybrid memory.

Our goal is to examine the algorithm, not instruction by instruction, but by a relatively large and identifiable amount of work. It may be noted that an asynchronous algorithms in general can only be run on the MIMD class of multiprocessors and not on SIMD multiprocessors.

We considered many experimental parallel processing simulators [2, 4, 7] and several commercial simulators [4, 10] and observed that either they were too complex, they simulated one specific computer architecture, or they simulated the problem at a machine instruction level.

To examine the affect of running asynchronous parallel algorithms, we decided to develop our own simple yet elegant simulator. We initially 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 an identifiable amount of work performed by an algorithm.

We are able to meet all three goals. With one simulator we can represent any MIMD computer architecture, can represent communications overhead of a distributed memory computer, can examine the work performed by each processor of the simulation, and collect data for overall algorithm performance evaluation. We compared these simulation results with measurements taken on shared memory Sequent Balance and Sequent Symmetry computers. Results indicate that the speedup and processor utilization from the simulation are fairly close to that of the actual runs for a small number of processors.

This paper is organized as follows. Section 2 defines the requirements for achieving the goals of simulating both shared and distributed memory MIMD machines. Section 3 presents a Static Parallel Arc Consistency algorithm as an example of an asynchronous parallel algorithm. Section 4 describes the serial and parallel algorithm for the simulator and the Sequent computers. Section 5 outlines measurement metrics and includes results of both simulation and actual results. Finally, our conclusions are given in Section 6.

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, which implies that the communication time is the same for each source–destination processor pair. This assumption is similar to the Intel iPSC/2, where message passing time is virtually identical [3]. We are concerned, though, with the communications overhead time associated with each message.

The second goal is to have a capability of simulating a shared, distributed, or hybrid memory architecture. We can differentiate between shared and distributed memory by employing a message passing scheme to provide variable updates on each processor needing the variable values.

The third goal is that the simulation measure a large and identifiable amount of work performed by our algorithm. One step of our algorithm, therefore, represents many computer instructions, such as a consistency check in an arc consistency algorithm. In fact, a consistency check is a very convenient measure to accurately compare all arc consistency algorithms (serial and parallel).

To implement a generic parallel system 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 one consistency check.

The time interval where all the processors are given one time slice is called a *cycle*, and after each cycle the memory is updated. A given processor, however, may have no 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 1):

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 various processors' registers.

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.

Step 4 The current state information is stored into the simulator's memory. Again, this step would be necessary in a simulated model and 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 the number of active processors. In a distributed memory computer, one processor or the host computer can hold this variable.

The operation of reading the multiprocessor memory is not dependent on memory architecture. The operation of writing to a multiprocessor's memory is dependent of the memory organization. Two different types of variables have been defined and implemented. The first type of variable can be updated only by one processor. The second type of variable is readable and writable by all the processors, but only one processor at a time is given the write access.

In a shared memory multiprocessor, when a specific processor wants to use a variable, it locks the desired memory location and prevents other processors from using it. The processor then unlocks it once the update is complete. This is known as a "test-and-set" operation [1]. In a distributed memory multiprocessor, one of the processors is assigned the global variable and handles the "test-and-set" operations requested by other processors.

We examine a class of asynchronous parallel algorithms [6]. Although asynchronous parallel algorithms are typically executed on shared memory computers, we have modified these algorithms for distributed memory environment as well. To allow these algorithms to run efficiently, we assume that each processor works with a subset of the problem data. If one processor changes a global variable, it sends the new value to all the processors requiring that variable. Special counting "test-and-set" variables are maintained by one processor.

ARC CONSISTENCY AS AN EXAMPLE OF ASYNCHRONOUS PARALLEL ALGORITHMS

Constraint satisfaction is a branch of Artificial Intelligence where one must find values for a group of variables, or parameters. Mackworth [8] 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."

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 v_i, there is at least one value in v_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 [5], must be used after satisfying arc consistency to search for a specific assignment set.

A *constraint network* is a graphical representation of a specific constraint satisfaction problem. Figure 2 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 2, one consistency check includes assigning the value A to Variable 1 and the value D to Variable 2 and checking to see if this pair of values satisfies Constraint 1. The network in Figure 2 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's [8] Arc Consistency #1 sequential algorithm (AC-1) is a simple algorithm to ensure arc consistency. Every arc of the constraint graph 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 will continue to recheck it's arcs until no more values are removed.

The worst case complexity of AC-1 is reported [9] as $\mathbf{O}(a^3ne)$ consistency checks, where n is the number of variables, e is the number of edges, and a is the size of the variable domains of a constraint graph. If we examine the constraint network of Figure 2, the sequential AC-1 algorithm makes 58 consistency checks while finding a single solution assignment. For our work we embrace this complexity measure as an important parameter, and we established our time slice to encompass a single consistency check for each processor.

The Static Parallel Arc Consistency algorithm (SPAC), is a modification of AC-1. The major difference is that the procedure ends only when all processors have verified local arc consistency. Once each processor has verified local arc consistency, it accesses a "test-and-set" counting variable. If the variable indicates that at least one other processor is still executing, the processor becomes idle. A processor restarts when a processor connected by a constraint deletes a value of a variable from the domain of the idle processor. A restarted processor tries to ensure arc consistency again. SPAC uses a procedure called Static Parallel Revise Boolean function (Sprevise), which is similar to Mackworth's Revise function. Another difference is that the procedure stops if a domain of a variable is empty, as no solution could ever be found.

The worst case time complexity of the SPAC algorithms is $\mathbf{O}(a^3en/p)$ consistency checks, where p is the number of processors. The worst case processing complexity of work performed is n times the the time complexity, or $\mathbf{O}(a^3ne)$. On the average, the actual work performed is quite less. If we examine the constraint network in Figure 2, the parallel SPAC algorithm makes 55 consistency checks in 15 cycles.

Our parallel algorithm (Figure 3) relies on a static distribution of work [1]. We can assign one or more constraint variables to each processor. With one constraint variable per processor the distribution of work is not necessarily equitable. If we assign several constraint variables to each processor, we ensure that the number of arcs per processor is approximately equal.

We randomly generated several different constraint networks and ran both the sequential and parallel algorithms. Since arc consistency is a pre-processing algorithm for constraint satisfaction search algorithms, we find all possible solution assignments to the network. To ensure comparison of similar processes, we have created networks with only one solution set.

MACHINE IMPLEMENTATION

Our simulator is implemented in C language on an IBM-compatible 80386 personal computer using the memory construct and the simulator structure mentioned earlier for both the sequential and parallel versions. Time sharing allows us to examine consistency checks and ensures that each processor is working with the same set of data. We can examine operations and assess the efficiency of the algorithm's operations.

We implemented in C the SPAC algorithm on the Sequent Balance and Symmetry shared memory multiprocessors (also called tightly-coupled MIMD computers). The Balance B8 computer has ten 32-bit processors, 8-Kbytes of cache memory per processor, and 8-Mbytes of shared memory. The Symmetry S81 has twenty eight 32-bit processors, 64-Kbytes of cache memory per processor, and 32-Mbytes of shared memory.

We programmed 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 was assigned a set of variables to check. The same variable assignments are used with the simulation and Sequent trials. Work is partitioned equitably to processors based on the total possible consistency checks required for each variable. 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 in shared memory, and each processor's counting variables are stored in private memory. 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 are 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.

SIMULATION AND RUNTIME PERFORMANCE RESULTS

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

By the common definition of speedup [1]:

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

We 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 2 using the simulator is 58/15 = 3.9.

Utilization is defined as the total number of consistency checks of the parallel algorithm divided by the total number of time slices available on the multiprocessor. For example, the utilization of the constraint graph shown in Figure 2 using the simulator is 55/60 = 91.7%.

The work ratio is the number of consistency checks performed by the parallel algorithm divided by the number of consistency checks performed by the serial algorithm. For example, the work ratio of the constraint graph shown in Figure 2 using the simulator is 55/58 = 0.95.

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 computers for a small number of processors. We selected networks where the AC-1 and SPAC algorithms remove 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 the network of medium connectivity.

Comparisons of consistency checks for the serial AC-1 and the SPAC algorithms on the simulator and actual multiprocessors is shown in Figure 4.

Simulation of these networks on a four-processor computer predicts speedup of between 80 and 94% of possible speedup, and actual computer execution indicate speedup within 4 to 6% of the simulated speedup (Figure 5). Simulation of a ten-processor computer predicts a speedup of between 68 and 82% of possible speedup, but actual runs show that the prediction is off by 9 to 12%. Utilization and work ratio figures display similar trends.

Simulation of 20-processor computers, however, do not closely match actual multiprocessor runs. We believe this is because of the difficulty to simulate simultaneous memory accesses on a shared memory multicomputer. Obviously we will need to make some adjustments to our simulator to compensate for simultaneous accesses.

CONCLUSIONS

We are pleased with our generic parallel algorithm simulation environment and we can test several other parallel algorithms. We believe that this conceptual simulation is a good way to examine the performance of parallel algorithms before implementing them on a parallel computer. We have demonstrated that the simulator closely represents the work performed on a small number of processing nodes of actual multiprocessors. The asynchronous nature of the algorithm indicates good promise for its future use in many applications.

ACKNOWLEDGEMENTS

Dr. Agrawal's work has been partially supported by the Army Research Office under contract DAAL03-91-G-0031.

The authors would like to thank the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Lab for use of their Sequent Symmetry Computer.

References

[1] Agrawal, Dharma P., Advanced Computer Architecture (Tutorial Text), IEEE Computer Society Press, 1986.

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

[3] Intel Scientific Computers Corporation, iPSC/2 User's Guide, Beaverton, OR, 1988.

[4] Kheir, Naim A., ed., Systems Modeling and Computer Simulation, Marcel Dekker, New York, NY, 1988.

[5] Janakiram, V.K., Agrawal, D.P., and Mehrotra, R., "A Random Parallel Backtracking Algorithm," IEEE Transactions on Computers, Vol. C-37, No. 12, December 1988, pp. 1665-1676

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

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

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

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

[10] Science Applications International Corporation, ANSPEC: An Actor Oriented Environment for PDP Specification, San Diego, CA, 1989.

[11] Sequent Computer Systems, Inc., Guide to Parallel Programming on Sequent Computer Systems, Prentice Hall, Englewood Cliffs, NJ, 1989.