1981

Testing the Coordination Predicate

Janice D. Cuny

Lawrence Synder

Report Number:
81-391

Testing the Coordination Predicate

Janice E. Cuny
Laurence Snyder

Purdue University

ABSTRACT

A collection of parallel processors is said to be coordinated if each write from one processing element (PE) to another is answered by a read. We report on an efficient algorithm to test coordination for parallel programs in which the code for each PE is a loop. We also test a weaker predicate for parallel algorithms with oblivious PE codes and we show that the general problem is PSPACE-hard.

August 20, 1982

CSD-TR-391

This work is part of the Blue CHiP Project. It is supported in part by the Office of Naval Research Contracts N00014-80-K-0816 and N00014-81-K-0300. The latter is Task SRO100.

Testing the Coordination Predicate

Janice E. Cuny

Lawrence Snyder

Purdue University

Pipelined or systolic [1] multiprocessors often depend critically on data values arriving at the right processing element (PE) at the right time without the benefit of explicit, interprocessor synchronization. In the simplest cases [1], when these algorithms involve a mesh interconnection of identical processors performing simple read-compute-write cycles, it is a straightforward matter to establish that the patterns of interprocessor reads and writes are compatible. Often, however, we find algorithms with multiple processor types, complex interconnection patterns [2], or differing data rates [3] and in these cases, the coordination of interprocess reads and writes can be quite complex. An objective then in simplifying parallel algorithm development is to support the coordination of interprocessor I/O operations.

We have reported earlier [4] on progress toward this objective. Starting with a parallel algorithm which assumes an abstract data flow execution mode, we show that for a limited, but widely practical class of algorithms, we can automatically generate the timing necessary for synchronous execution. But what if the algorithm is not in the class or if manual design is required? In this paper, we report on algorithms that assist the designer by testing programs for incompatibilities in interprocess communication.

The Model of Parallel Programs

We postulate a parallel processor composed of m processing elements (PEs), $M_1, M_2, \ldots, M_m$, which communicate with read and write operations. The PEs are all of the same type and since we are concerned only with interprocessor input/output behavior, it is sufficient to let them be devices capable of defining a regular set. We assume that the PEs execute synchronously and that, on each step, a PE can simultaneously execute a set of operations.

We model such systems as *Interprocessor Communication (IC) Systems.*[†] An IC system is completely defined by a set of reduced, Moore-type machines, $V_1, V_2, \ldots, V_m$, each describing the interprocess input/output behavior of a single PE. The i-th machine describes the behavior of the i-th PE. The alphabet of the machines consists of symbols denoting sets of operations that are to be executed simultaneously. Each symbol is an element of the power set of [††]

$$\{r_{i,\sigma} | i \in [m] \wedge \sigma \in \Sigma\} \cup \{w_{i,\sigma} | i \in [m] \wedge \sigma \in \Sigma\}$$

where Σ is a finite set of values, $r_{j,\sigma}$ denotes an operation that reads σ from PE j, $w_{j,\sigma}$ denotes an operation that writes σ to PE j, and φ, the empty set, takes the place of any operation not involved in interprocessor communication (including operations that transfer values to and from the external environment).

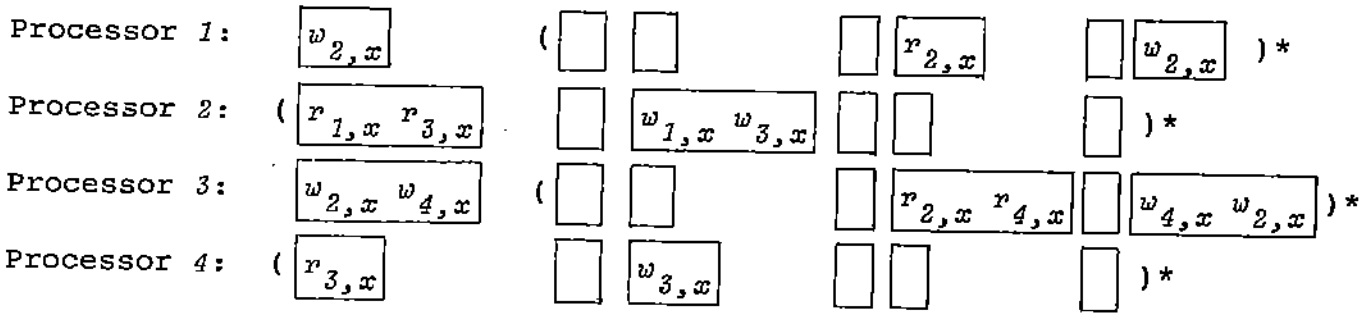
Figure 1(a) is an IC system describing a systolic processor for band

---

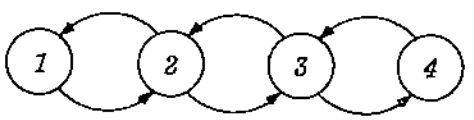
[†] IC Systems can be defined more generally [5] but for the purposes of this paper, we present only a limited version.

[††] $[m]$ denotes the set $\{1,2,\ldots,m\}$. Note that we use standard set notation to represent both sets and the symbols of our alphabet; the distinction will be clear from the surrounding context.

matrix-vector multiplication with a bandwidth of four [7]; only interprocessor reads and writes appear in the model, all other operations are replaced by φ. †



1(a) IC system representing systolic processor for band matrix - vector multiplication.



1(b) Communication graph for the IC system of Figure 1(a).

Figure 1.

For this example, there are no data-dependent branches and so we denote the values passed between processors by a single, generic value x. If PE i writes to PE j or PE j reads from PE i, we say that there is a *communication link* from i to j. Figure 1(b) is a *communication graph* in which the communication links for the system in Figure 1(a) appear as edges.

We define the execution of an IC system in terms of two sequences,

---

† In the figures, we use a rectangle to enclose the elements of a set rather than set braces.

$C^1, C^2, C^3, \ldots$ and $Q^0, Q^1, Q^2, \ldots$. Each element of the first sequence is an an m-vector of symbols, one coordinate for each PE, describing the operations executed by an IC system in a single step. Each element of the second sequence is an $m \times m$ matrix of strings, giving the status of communications; $q_{i,j}^k$ gives the status of communications on the link from PE i to PE j on step k. Values that have been written but that have not yet been read are denoted by elements of an alphabet Σ; values that have been requested but that have not yet been written are denoted by their inverses. The inverse of a symbol σ is written σ^{-1} and $\sigma \cdot \sigma^{-1}$ is defined to be λ, the empty string. Each $q_{i,j}^k$ is of the form $\alpha\beta$ where α is a queue of written values (head on the right end) and β is a queue of requested values (head on the left end) and corresponding writes and reads cancel at the boundary between these queues. For all $k>0$, C^k describes the set of operations executed in the k-th step and Q^k describes the status of communications after those operations complete.

To start the sequences, we define c_i^1 for all $i \in [m]$, to be the first symbol in some sequence generated by V_i and $q_{i,j}^0$ for all $i,j \in [m]$ to be λ. C^1 shows all PEs executing their first set of operations and Q^0 shows that there are no outstanding reads or writes. The remainder of the sequence of Cs is defined so that a PE moves to a new set of operations on each time step and the operations that it executes form sequences generated by its associated finite state machine: $c_i^{k+1} = c$ for some c such that

$$c_i^1 \cdot c_i^2 \cdot c_i^3 \cdot \cdots \cdot c_i^k \cdot c \in L(V_i).$$

The remainder of the sequence of Qs is defined to reflect the execution of read and write operations: $q_{i,j}^{k+1} = a \cdot q_{i,j}^k \cdot b$ where

$$a = \begin{cases} \sigma & \text{if } w_{j,\sigma} \in c_i^{k+1} \\ \lambda & \text{otherwise} \end{cases}$$

and

$$b = \begin{cases} \sigma^{-1} & \text{if } r_{i,\sigma} \in c_j^{k+1} \\ \lambda & \text{otherwise} \end{cases} .$$

We intend for the operations $r_{i,\sigma}$ and $w_{j,\sigma'}$ to correspond only if $\sigma = \sigma'$. To enforce the matching, we define the *legal computation sequences* of an IC system to be the set of all computations with the property that for all i, j, and k

$$q_{i,j}^k \in \Sigma^* \cup (\Sigma^{-1})^* .$$

This restriction allows us to express the dependency of branching on transmitted values because, unless all corresponding reads and writes match, some link will have a status in $\Sigma^* \cdot (\Sigma^{-1})^*$.

The definition of our model allows computations in which a PE executes *before* the corresponding write; this is acceptable in a formal model but not in an actual system. In order to be correct, the reads and writes of a synchronous system must be coordinated. We say that a system is *strongly coordinated* if for all i, j, and k

$$q_{i,j}^k = \lambda ,$$

that is, corresponding reads and writes are simultaneous. † We say that a system is *weakly coordinated* if for all i, j, and k

$$q_{i,j}^k \in \{\lambda\} \cup \Sigma \ \wedge \ ((k>0 \ \wedge \ q_{i,j}^{k-1} \in \Sigma \ \wedge \ q_{i,j}^k = a \cdot q_{i,j}^{k-1} \cdot b) \Rightarrow a = \lambda)$$

that is, every read is preceded by its corresponding write and there are no consecutive writes.

---

† It is more customary to assume some unit time delay between a write and the subsequent read. We have chosen them to be simultaneous to simplify our presentation but our algorithms can be trivially modified to incorporate any fixed delay in communications between PEs.

One of the most complex aspects of programming for parallel processors is the problem of insuring that the resulting system is correctly coordinated. In this paper we address this problem by providing algorithms to questions of the form

Given an IC system, is it strongly (weakly) coordinated?

We consider the problem for a sequence of cases, based on increasingly complex IC system structure. For the first two cases, which are sufficient to cover most of the existing parallel algorithms, we present efficient algorithms to test coordination. For the third, general case, we show that the problem is computationally intractable.

Loop Programs

In the simplest case, we restrict our attention to *loop programs* in which each PE first executes an initialization sequence and then repeatedly executes a single cycle of instructions. While this restriction seems prohibitive, many highly parallel systems, including most of the systolic processors, can be characterized in this way. Strong coordination across a single communication link of a loop program can be tested with the following algorithm.

Algorithm 1. Verification of strong coordination on a communication link between two loop programs.

Input: Two finite state machines, V_1 and V_2, representing the source and destination PEs of the communication link to be tested respectively.

Output: CORRECT if the input/output behavior across the given communication link is strongly coordinated; ERROR otherwise.

Method:

(1) For each of the PE's, determine the length of its initialization sequence f_i and its cycle l_i. Let *MAX* be the maximum of f_1 and f_2 and let *LCM* be the least common multiple of l_1 and l_2.

(2) Construct the IC system V_1', V_2' where V_i' is V_i with all I/O operations to links other than the given one replaced by φ.

(3) Execute the newly constructed IC system for *MAX+LCM* steps. If a strong coordination error is found, report ERROR; otherwise report CORRECT.

(4) HALT.

Theorem 1. Algorithm 1 correctly detects strong coordination errors in loop programs.

Proof: The machines V_1' and V_2' have the same behavior across the given link as the machines V_1 and V_2 respectively, so it is sufficient to test the coordination of the newly constructed system. For all $k > MAX+LCM$, the PEs in this new system execute the same operations at time k as they do at time $(k - MAX)\ mod\ LCM$ and so the test in step (3) of the algorithm covers all possible execution steps. //

In order to test weak coordination, the algorithm must be modified slightly. After *MAX+LCM* steps, both PEs are at the same point in their cycles as they were after *MAX* steps. In the case of strong coordination, we can be sure that any coordination errors would have shown up by this point. In the case of weak coordination, however, it is possible that there is an outstanding write on step *MAX* and not on step *MAX+LCM* or vice versa; in either case, this would cause a coordination error to show up on the next I/O operation. To guarantee weak coordination, therefore, step (3) of the algorithm must be replaced by

(3) Execute the newly constructed IC system for *MAX+LCM* steps. If a weak coordination error is found or there is an outstanding write after step *MAX* and no outstanding write after step *MAX+LCM* or vice versa, report ERROR; otherwise report CORRECT.

If we assume that each machine of the IC system has at most n states, then we test at most n^2 execution steps and the algorithm requires $O(n^2)$ time per communication link. If we further assume that a system is composed of a small number of distinct PE types which are connected in analogous ways, then it is sufficient to test each link type just once. For a system with t link types and at most n states per machine, we have

Theorem 2. The coordination of a system of interconnected, loop programs can be tested in $O(n^2t)$ time.

Notice that this result is dependent, not on the number of PEs, but on the variety of their interconnection structure.

Oblivious Programs

To test the coordination of more general systems, we define *oblivious programs*, in which we allow arbitrarily complex finite state machines but we replace all data values in Σ with a single, generic value. Since all computation sequences of an oblivious machine are legal, PEs do not have the capability of selectively branching on incoming data values. For these systems, we can test only worst case coordination, that is, we can answer the question

Given an oblivious IC system, is there a potential coordination error? If our algorithm reports CORRECT then the system is coordinated; if our algorithm reports POTENTIAL ERROR, it is possible that the detected error will never show up in any legal computation of the system.

To test worst case coordination along a single communication link, we form the "cross product" machine for the two PEs involved as in Figure 2. The new machine has a state set of size q where, if n is the maximum number of states of any of the machines in the IC system, q is at most n^2. Each sequence generated by the cross product machine represents a simultaneous execution of the two oblivious programs involved. If we define an *I/O state* for a particular link to be any state in which a read or write to that link occurs, the question of strong coordination is reduced to the question

> Is there a reachable I/O state in which a read and write do not both occur?

This is just the state reachability question for finite state machines which can be answered in $O(q) = O(n^2)$ time. The question for weak coordination is more complicated.

To obtain an appropriate test for weak coordination, we represent the behavior of a finite state machine with a *computation tree* in which nodes are labelled by states and edges signify transitions. We are interested in the computation tree for a particular link, that is, the tree in which all I/O operations not pertaining to that link have been replaced by φ. Figure 3 shows the computation tree for the link from PE A to PE B of the cross product machine depicted in Figure 2.

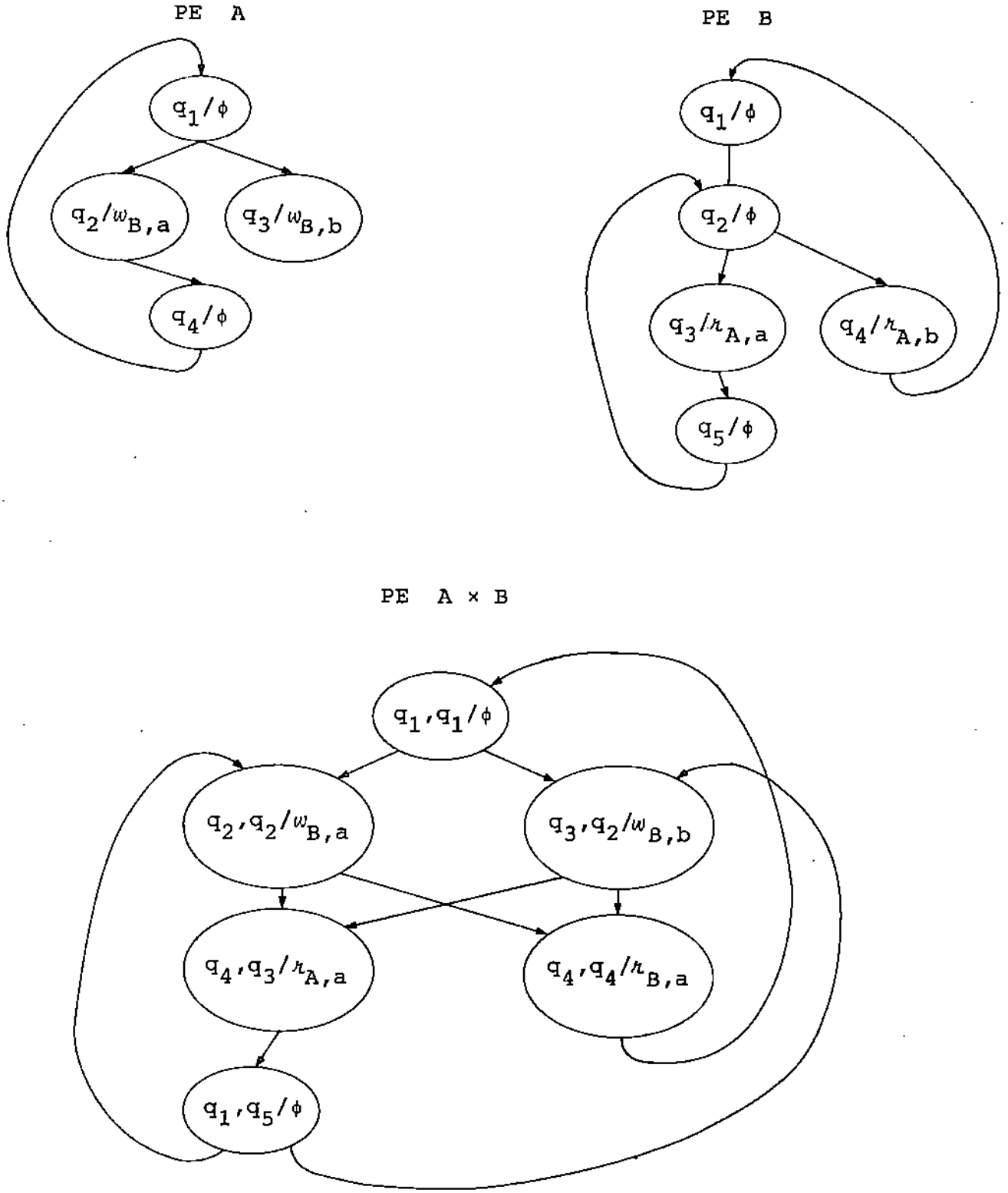


Figure 2. Two PEs and their cross product machine.

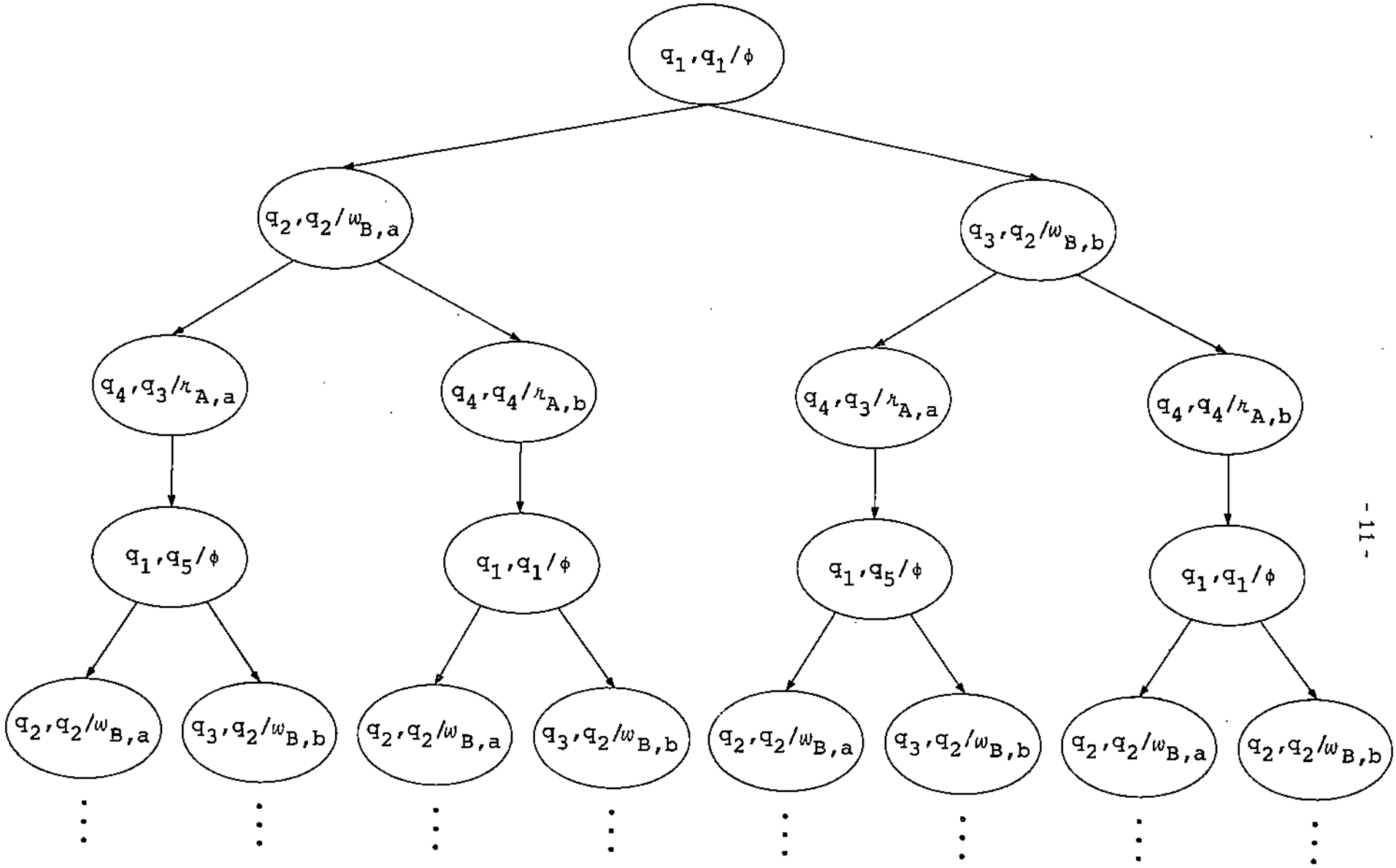


Figure 3. Computation tree for cross product machine in Figure 2.

We define the *outcome* of a path p in a computation tree to be

$$\begin{cases} 1 & \text{if there is 1 more write than read along } p \\ & \text{and there are no consecutive writes along } p \\ 0 & \text{if there are an equal number of reads and writes along } p \\ & \text{and there are no consecutive writes along } p \\ -1 & \text{if there is 1 more read than write along } p \\ & \text{and there are no consecutive writes along } p \\ \perp & \text{otherwise} \end{cases}$$

The following lemma relates the outcome of paths to potential weak coordination errors

Lemma 1. The computation tree for a link contains no potential weak coordination errors if and only if all paths from the root have an outcome of either 0 or 1.

Proof: For a given link i,j, consider the IC system that is composed of the two PEs using i,j with all of their other I/O operations replaced with φ. This system has a weak coordination error if and only if the original IC system had such an error on the given link. Each path in the computation tree of the cross product machine for the new system corresponds directly to one of its execution sequences and it is easily shown by induction that

(i) the path from the root to a node at level l in the computation tree has outcome 0 if and only if in the corresponding execution sequence

$$q_{i,j}^{l} \in \{\lambda\} \wedge ((k>0 \wedge q_{i,j}^{k-1} \in \Sigma \wedge q_{i,j}^{k} = a \cdot q_{i,j}^{k-1} \cdot b) \Rightarrow a = \lambda)$$

and

(ii) the path from the root to a node at level l in the computation tree has outcome 1 if and only if in the corresponding execution sequence

$$q^l_{i,j} \in \Sigma \wedge ((k>0 \wedge q^{k-1}_{i,j} \in \Sigma \wedge q^k_{i,j} = a \cdot q^{k-1}_{i,j} \cdot b) \Rightarrow a = \lambda) \quad .$$

Thus the path to a node at level l in the computation tree has outcome 0 or 1 if and only if

$$q^l_{i,j} \in \{\lambda\} \cup \Sigma \wedge ((k>0 \wedge q^{k-1}_{i,j} \in \Sigma \wedge q^k_{i,j} = a \cdot q^{k-1}_{i,j} \cdot b) \Rightarrow a = \lambda)$$

that is, if and only if there is no potential weak coordination error in the tree. //

As a result of this lemma, we can reduce the question of potential weak coordination errors to the question

> Is there a path from the root in the computation tree for the link that has outcome -1 or $\perp$?

We now introduce a series of lemmas to show that we can determine the answer to this question after having seen only a finite amount of the computation tree. We show

Lemma 2. No path between two nodes in an error-free computation tree has an outcome of $\perp$

Proof: Let $p = t_1, t_2, \ldots, t_r$ be the shortest path in the tree with outcome $\perp$; p must have at least two nodes. Since p is shortest, there are three possibilities for the outcome of the path $p' = t_1, t_2, \ldots, t_{r-1}$

(*i*) p' has outcome 1: then there is one more write than read along p'. Since reads and writes must alternate on any path in an error-free tree, the last I/O operation in p' must be a write and t_r cannot contain a write. The outcome of p must be 0 or 1, a contradiction.

(*ii*) p' has outcome 0: as defined, the outcome of p cannot be $\perp$

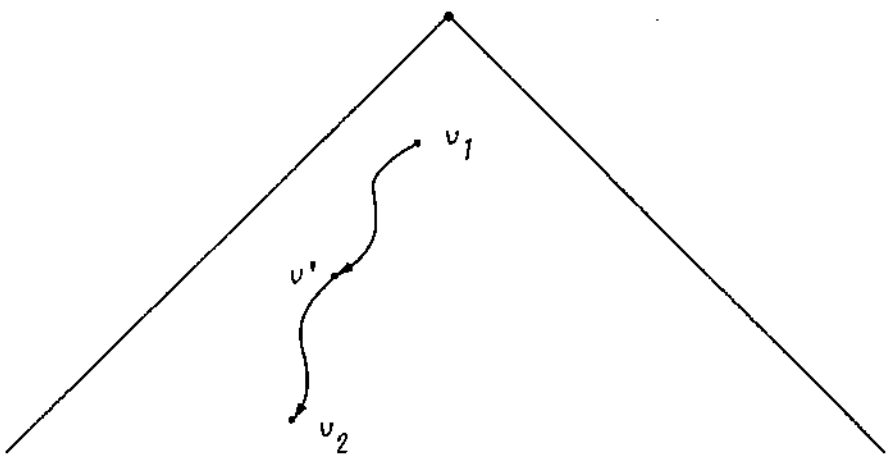
(*iii*) p' has outcome -1: then as in case (*i*), p must have outcome either 0 or 1, a contradiction.

Thus, p cannot have outcome $\perp$ which is a contradiction. //

Defining a *cycle* in a computation tree to be a path from a node to (but not including) the next occurrence of a node labelled with the same state, we show

Lemma 3. The outcome of a cycle in an error-free computation tree must be 0.

Proof: Suppose there is a cycle from node v_1 to node v_2 with outcome other than 0. By Lemma 2, the outcome of the cycle must be either 1 or -1. If the outcome is 1, then there must be one more write than read on the cycle and so there must be at least one node v' on the cycle labelled by a state containing a write operation but no read operation:



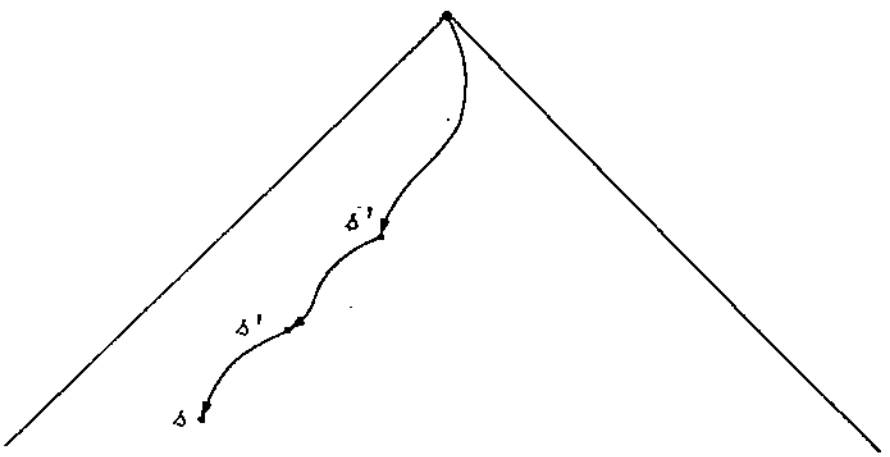
Let p_1 be the path from v_1 to the node immediately preceding v' and let p_2 be the path from the node immediately following v' to v_2. p_1 and p_2

together must contain an equal number of reads and writes. In addition, there must be a path in the tree which starts at v' and has the same labels as v' followed by p_2 followed by p_1 followed by v' again. This path must have an outcome $\perp$ since it has two more writes than it has reads. By Lemma 2, this is a contradiction. Likewise if the outcome of the original cycle was -1. //

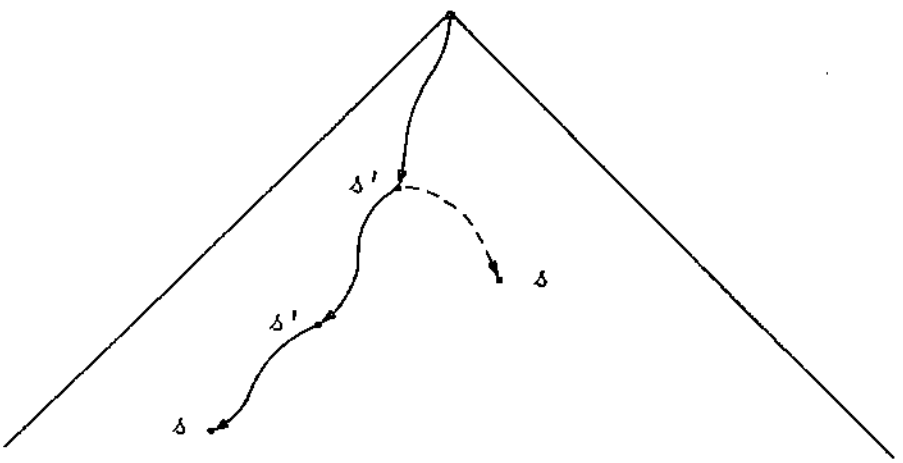
The algorithm for testing worst case weak coordination builds a partial computation tree breadth-first, computing the outcome of paths from the root to each node as it is added. If q is the maximum number of states in the cross product machine, the following lemma bounds the depth of the tree that must be searched in order to determine if there is an error.

Lemma 4. The existence of a potential coordination error in a computation tree can be determined after examining at most $2q$ levels of the tree.

Proof: Let $l > 2q$ be the earliest level at which an outcome from the root to some node s becomes either -1 or $\perp$. There must be some state s' that repeats on the path to the node labelled s before level l



The cycle from the first occurrence of s' to the next along this path must have outcome 0 or, by Lemma 3, we could have detected the existence of a coordination error when the cycle was encountered. If the cycle outcome is 0, the paths from the root to the two nodes labelled s' must have the same outcome and so the outcome at the node labelled s must be the same as the outcome at a node reached by a shorter path which does not contain the cycle:

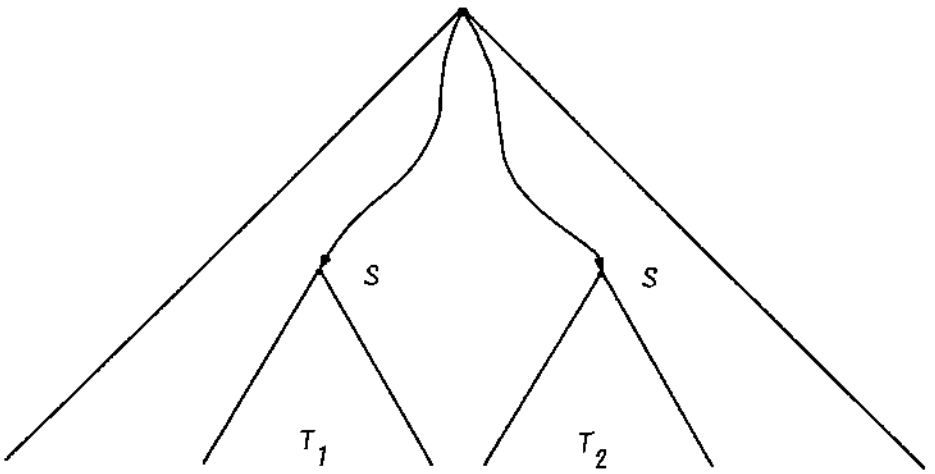


The outcome at s was $\perp$ and so there must be a $\perp$ outcome before level l which is a contradiction. //

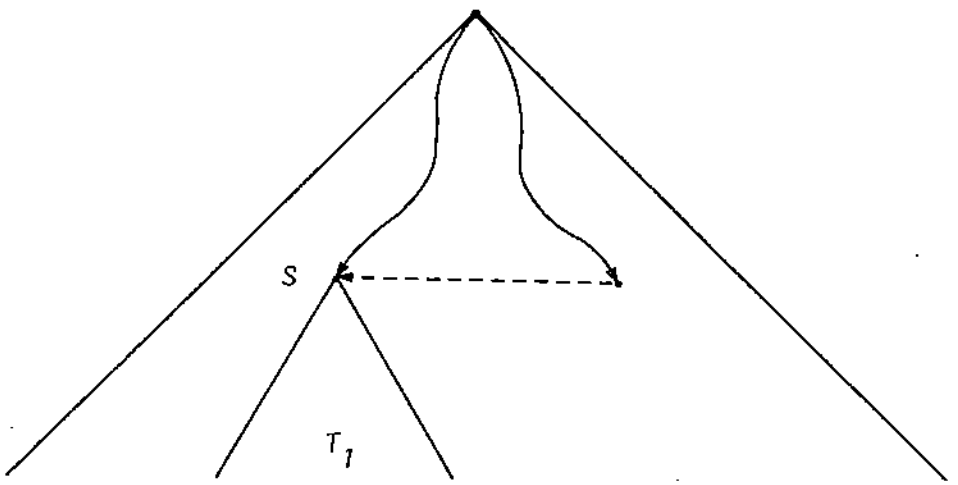
For a cross product machine with at most q states, the following lemma bounds the number of nodes that must be retained on any given level

Lemma 5. The computation tree for determining worst case weak coordination requires at most $2q$ nodes per level.

Proof: Consider two nodes with the same label s on some level of the tree and the subtrees below them:



The subtrees T_1 and T_2 must be identical. Furthermore, if the paths from the root to these nodes have the same outcomes, the outcomes at all corresponding nodes in those subtrees must also be the same. Thus we can "merge" the two nodes



and examine only one of the subtrees to detect coordination errors. Since there are 2 possible legitimate outcomes for each node and q possible states, we need to retain at most $2q$ nodes per level. //

We will use these results in constructing an algorithm for testing

worst case, weak coordination. The algorithm maintains a descriptor for each node on the current level which contains

(*i*) the union of the outcomes of all paths from the root to the node

(*ii*) for each of the q states, an indication of whether or not that state has appeared on any of the paths to this node,

and

(*iii*) for every state, a set containing the possible outcomes from all nodes labelled with that state on some path to (but not including) the current state.

The complete algorithm for finite state machines with q states is as follows

Algorithm 2. Verification of worst case, weak coordination on a single communication link.

Input: Two finite state machines, V_1 and V_2, representing the source and destination PEs of the communication link to be tested respectively.

Output: POTENTIAL ERROR if there are any potential coordination errors on the input/output behavior across the communication link from PE 1 to PE 2; CORRECT otherwise.

Method:

(1) Build a partial computation tree in a breadth-first manner for $2q$ levels below the root:

For each descendent v on a level:

(a) Compute the set of possible outcomes from the root to v using the set of outcomes previously computed for the parent of v. If the set for v contains either -1 or 4, report POTENTIAL ERROR and HALT; otherwise build the descriptor for the node.

(b) If the label of v has appeared earlier on any path to this node, check to see that the outcome on each of those paths since its last occurrence is 0; if not, report POTENTIAL ERROR and HALT.

(c) If the label of v has already appeared on this level, merge the two occurrences by merging their descriptors (unioning outcomes, states encountered and the outcomes since their last occurrences). Remove the current node.

(2) Report CORRECT and HALT.

Theorem 3. Algorithm 2 correctly detects all worst case, weak coordination errors for the given link.

Proof: PART I. Suppose that the algorithm halts after reporting an error. This could happen in either of two ways: as a result of step (a) or as a result of step (b). If it occurred as a result of step (a) then, by Lemma 2, the tree contains an error. If it occurred as a result of step (b) then the tree also contains an error because, by Lemma 3 the outcomes of all cycles in an error-free tree must be 0. Thus, whenever the algorithm reports an error, there is an error in the tree.

PART II. Suppose the tree computation tree for the link contains an error. Then by Lemma 4, that error can be detected in at most $2q$ levels of the tree either because of an error in the outcome on a path (reported in step (a)) or because of a cycle outcome error (reported in step (b)). Thus, if there is an error in the tree, the algorithm reports it. //

For IC systems with t link types and machines with at most n states each, we can combine the results for strong and weak coordination testing in oblivious programs, to get

Theorem 4. The worst case coordination of a system of interconnected, oblivious programs can be tested in $O(n^6 t)$ time.

Again, the results are not dependent on the number of PEs involved but just on the variety of their interconnections. While a bound containing a factor of n^6 seems large, we expect that in most cases n will be quite small. In addition, there are a number of optimizations that we can make on the finite state machines, such as collapsing parallel branches with the same I/O characteristics, to reduce the size of their state sets.

General Programs

Finally, we consider the general case with unrestricted, data dependent control flow within processors.

We show

Theorem 5. For an arbitrary system of interconnected processors, the problem of testing communication interfaces for strong coordination is PSPACE-hard.

Proof: We reduce the language recognition problem for linear bounded automata (lba's), which is known to be PSPACE-complete[6], to the coordination problem. Given an lba and an input string, we construct an IC system which has a coordination error if and only if the lba accepts the given string.

The IC system has the structure shown in Figure 4 where the Memory PEs are storage devices and the Control PEs are modified instances of the lba. There is one Memory PE, Control PE pair for each tape square. The Memory PE keeps track of the current symbol written on its corresponding tape square and the symbol is transfered back and forth between the two PEs, enabling the Control PE to read and branch on its value. The lsa

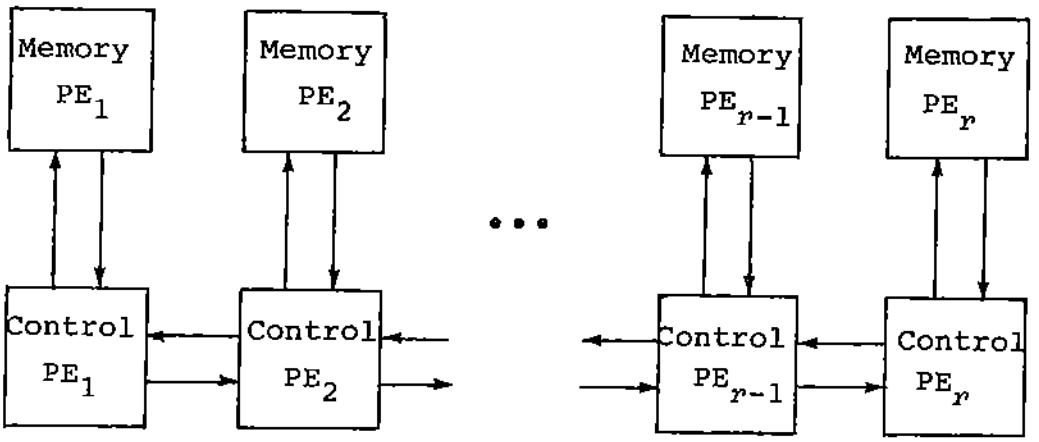


Figure 4.

IC system configuration for simulating an lba.

for the Memory PEs is shown in Figure 5. We use a two symbol alphabet (0 and 1) and the appropriate initial state is determined by the initial value of the tape square.

The Control PEs, in addition to reading the current tape symbol from their corresponding Memory PEs, also read two tokens from their adjacent neighbors: one indicates whether the tape square corresponding to this Control PE will have the read head on the next state and the other is the index of the next state. As indicated in Figure 6, all Control PEs except for the PE at the square where the head initally resides, have the same starting state and the initial state for the PE with the head depends on the initial state of the lba.

The Control PEs read from the Memory PEs (the third level of nodes

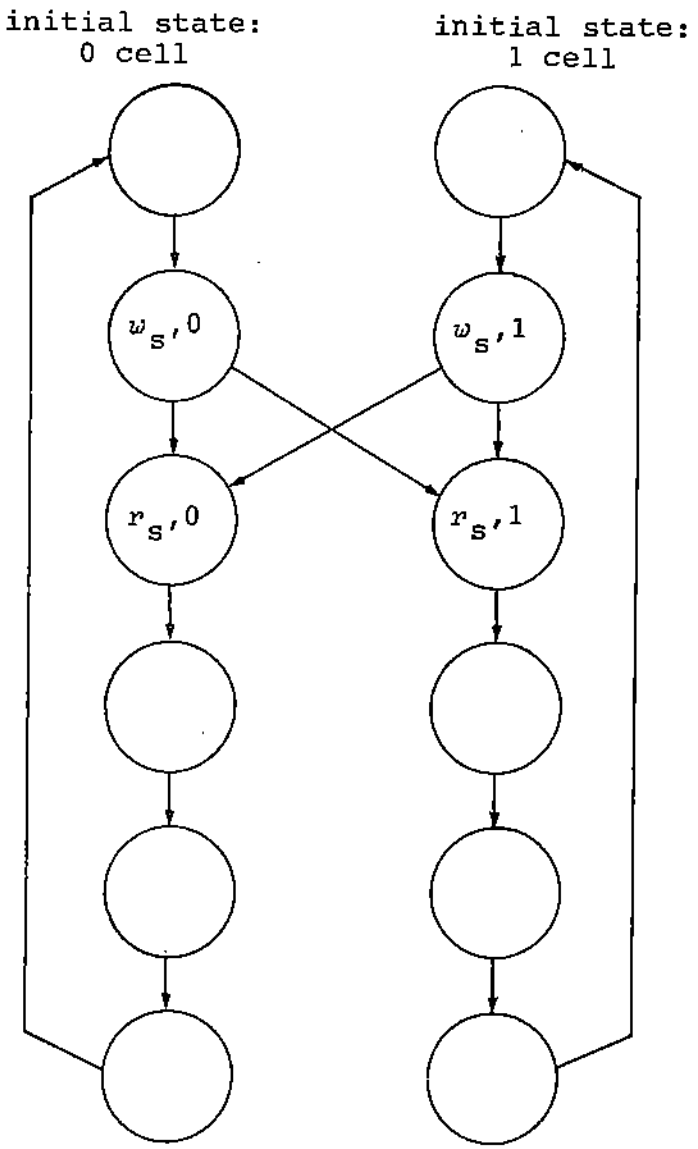


Figure 5.

Finite state machine for Memory PE.

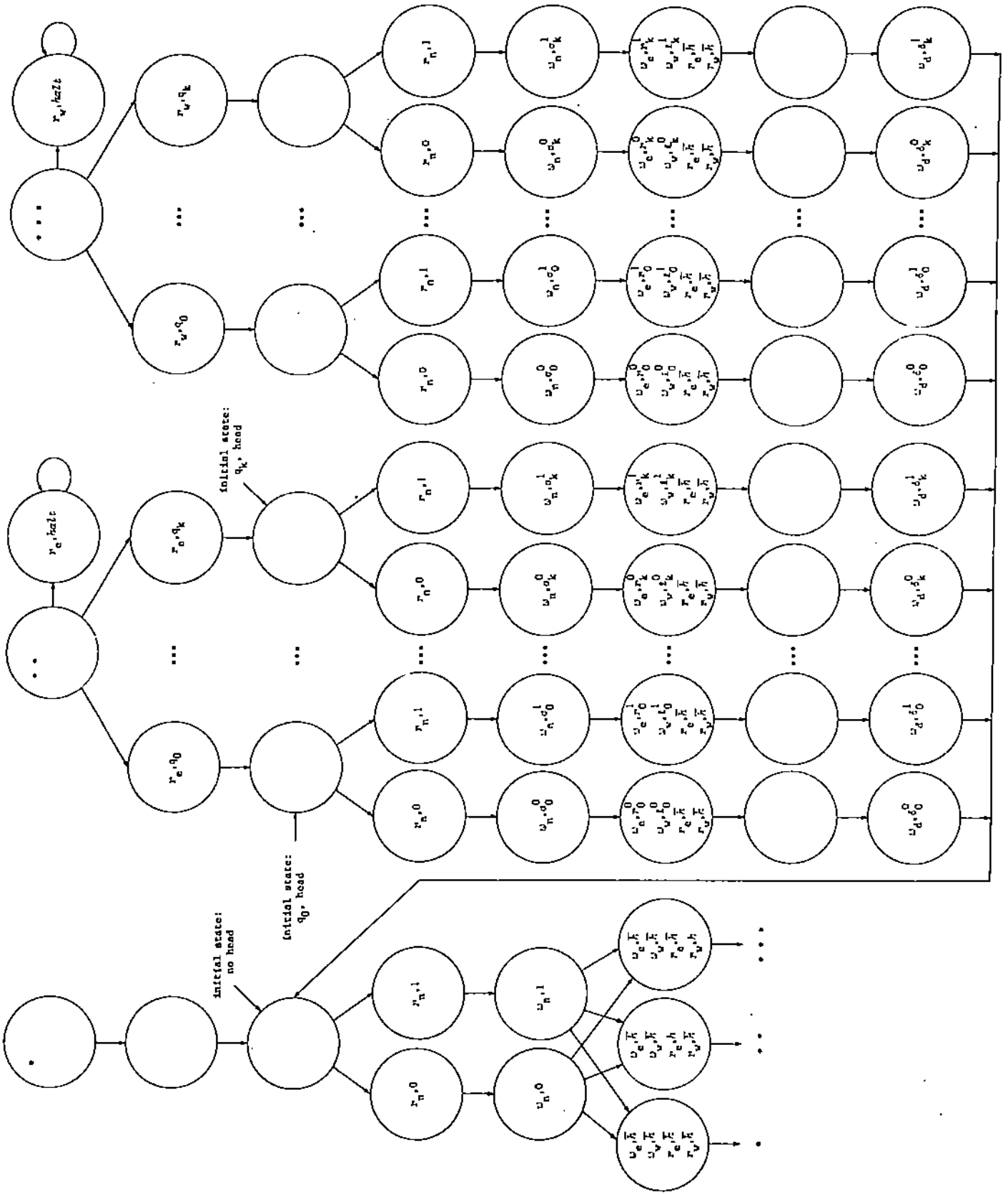


Figure 6. Finite state machine for Control PEs.

in Figure 6) and then write a value back (the fourth level of nodes). If the PE does not have the head, the input value is echoed back. If the PE does have the head, the new value, determined from the transition function of the lba, is written (σ_i^j denotes the value written for state j and symbol i).

On the next step (the fifth level of nodes), the Control PEs simultaneously write and read to their east and west neighbors, indicating the head movement; h and $\bar{h}$ represent the messages "head" and "no head" respectively.

At this point, the Control PE that has the head is on the null state at the seventh level of the figure, the PE that is about to receive the head is either at the state labelled ** or the state labelled *** (depending on whether the head is to be passed from the east or the west), and the remainder of the PEs are at the state labelled *. In the next step, the new state information is passed to the PE receiving the head and the cycle repeats.

The IC system continues simulating the behavior of the lba until a *halt* is reached. The *halt* is passed to the control PE with the head instead of a next state, causing that PE to repeat the read which in turn causes a coordination error. //

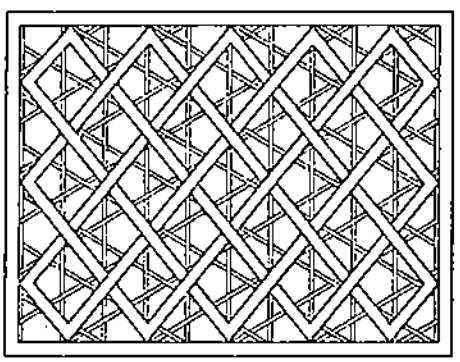
Discussion

Although the complexity theory results indicate that coordination testing is a very complex task, it is important to notice that many recently developed parallel algorithms are covered by Theorem 2. The testing algorithms presented here are currently being implemented and we expect that they will be of significant assistance to programmers

working on parallel algorithms. In addition, as libraries of well understood and well tested parallel modules become available, we expect that these same testing algorithms can be used to check automatically the interface compatibilities of modules.

Acknowledgements

We owe a debt of gratitude to Dennis Gannon for useful discussions concerning coordination and to Cathy Cole who implemented an initial version of these algorithms.



Street-vender's stall, Chengtu, Szechwan, 1918 A.D.

References

[1] H. T. Kung and C. E. Leiserson, Systolic Arrays (for VLSI), In Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison Wesley, 1980, pp. 271-292

[2] D.B. Gannon and L. Snyder, Linear Recurrence Algorithms for VLSI: The Configurable Highly Parallel Approach, International Conference on Parallel Processing, pp. 259-260.

[3] L.J. Guibas, H.T. Kung and C.D. Thompson, Direct VLSI Implementation of Combinatorial Algorithms, Caltech Conference on VLSI, California Institute of Technology, 1979.

[4] Janice E. Cuny and Lawrence Snyder, Conversion from Data-Flow to Synchronous Execution Mode in Loop Programs, Technical Report CSD-TR-391, Purdue University, 1982.

[5] Janice E. Cuny and Lawrence Snyder, A Model for Analyzing Generalized Interprocessor Communication Systems, Technical Report CSD-TR-406, Purdue University, (in preparation).

[6] Michael R. Garey and David S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., p. 271 (1979).