数理解析研究所講究録
1185 巻 2001 年 81-89

Some Complexity Issues in Parallel Computing

Oscar H. Ibarra

Department of Computer Science, University of California
Santa Barbara, California 93106, USA
ibarra@cs.ucsb.edu

Abstract: We give an overview of the computational complexity of linear and mesh-connected cellular arrays with respect to well known models of sequential and parallel computation. We discuss one-way communication versus two-way communication, serial input versus parallel input, and space-efficient simulations. In particular, we look at the parallel complexity of cellular arrays in terms of the PRAM theory and its implications, e.g., to the parallel complexity of recurrence equations and loops. We also point out some important and fundamental open problems that remain unresolved.

Next, we investigate the solvability of some reachability and safety problems concerning machines operating in parallel and cite some possible applications. Finally, we briefly discuss the complexity of the "commutativity analysis" technique that is used in the areas of parallel computing and parallelizing compilers.

Keywords: cellular array, computational complexity, P-complete, parallel complexity, recurrence equations, reachability, safety, commutativity analysis.

1 Introduction

One of the earliest and simplest models of parallel computation is the cellular array, also called the cellular automaton. They have been studied extensively in the literature. Early papers have studied these devices in the context of pattern and language recognition - their recognition power, closure and decision properties, and their relationships to other models of computation, such as Turing machines, linear bounded automata, pushdown automata, and finite automata. In later papers, the study of these arrays has focused on their abilities to perform numeric and nonnumeric computations in various areas such as computational linear algebra and signal and image processing. Such arrays, whose processors need no longer be "finite-state", have also been called systolic arrays.

Here we give an overview of the computational complexity of cellular arrays with respect to well known models of sequential and parallel computation. We discuss results concerning one-way communication versus two-way communication, linear-time versus real-time, serial input versus parallel input, space-efficient simulation of one-way arrays, parallel complexity of cellular arrays, etc. We also point out some important and fundamental open problems that remain unresolved.

Next, we investigate the solvability (existence of algorithms) of some reachability and safety problems concerning machines operating in parallel. Possible applications of the results are in load balancing in parallel machines and model-checking and safety testing in reactive systems.

Two operations commute if they generate the same result regardless of the order in which they execute. Commuting operations enable significant optimizations in the areas of parallel computing and parallelizing compilers. We briefly discuss the computational complexity of commutativity analysis.

2 Cellular arrays

A *linear cellular array* (LCA) is a one-dimensional array of n identical finite-state machines (called nodes) that operate synchronously at discrete time steps by means of a common clock [BUCH84, KOSA74, SMIT70,

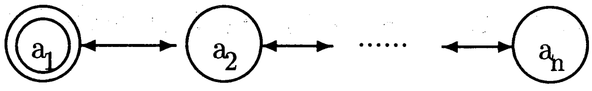

Figure 1. An LCA.

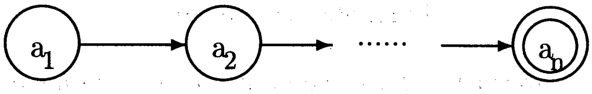

Figure 2. An OLCA.

SMIT71, SMIT72] (see Figure 1). The input $a_1a_2 \cdots a_n$, where a_i is in the finite alphabet Σ is applied to the array in parallel at time 0 by setting the states of the nodes to $a_1, a_2, \cdots, a_n$. The state of a node at time t is a function of its state and the states of its left and right neighbors at time $t-1$. We assume that the leftmost (rightmost) node has an "imaginary" left (right) neighbor whose state is \$ at all times. We say that $a_1a_2 \cdots a_n$ is accepted by the LCA if, when given the input $a_1a_2 \cdots a_n$, the leftmost cell eventually enters an accepting state. The LCA has time complexity $T(n)$ if it accepts inputs of length n within $T(n)$ steps. Clearly, for a nontrivial computation, $T(n) \geq n$. If $T(n) = cn$ for some real constant $c \geq 1$, then the LCA is called a *linear-time* LCA. When $T(n) = n$, it is called a *real-time* LCA. Note that an LCA without time restriction is equivalent to a linear-space bounded deterministic Turing machine (TM).

A restricted version of an LCA is the *one-way* linear cellular array (OLCA) [DYER80], where the communication between nodes is one-way, from left to right. The next state of a node depends on its present state and that of its left neighbor (see Figure 2). An input is accepted by the OLCA if the rightmost node of the array eventually enters an accepting state. The time complexity of an OLCA is defined as in the case of an LCA.

Although OLCA's have been studied extensively in the past (see, *e.g.*, [BUCH84, CHOF84, DYER80, IBAR85b, IBAR86, UMEO82]) a precise characterization of their computational complexity with respect to space- and/or time-bounded TM's is not known. For example, it is not known whether linear space-bounded deterministic TM's are more powerful than OLCA's, although a positive answer seems likely.

Another simple model that is closely related to linear cellular arrays is the *linear iterative array* [COLE69, HENN61, IBAR85b, IBAR86]. The structure of an LIA is similar to an LCA, as shown in Figure 3. (Note that here we assume that the size of the array is bounded by the length of the input. In some papers, the array is assumed to be infinite.) The only difference between an LIA and an LCA is that in an LIA the input $a_1a_2 \cdots a_n\$$ is fed serially to the leftmost node. Symbol $a_i, 1 \leq i \leq n$, is received by the leftmost node at time $i-1$; after time $n-1$, it receives the endmarker \$. That is, \$ is not consumed and always available for reading. At time 0, each cell is in a distinguished quiescent state q_0. As in an LCA, the state of a node at time t is a function of its state and the states of its left and right neighbors at time $t-1$. For the leftmost node, the next state depends on its present state and the input symbol. An OLIA (the one-way version of the LIA) is defined in a straightforward way.

For a nontrivial computation, the time complexity of an LIA is at least n, and the time complexity of an OLIA is at least $2n$. An LIA operating n steps is called a real-time LIA and an OLIA operating in $2n$ steps is called a *pseudo-real-time* OLIA. One can easily show that an LIA and an LCA can efficiently simulate each other.

Mesh-connected cellular arrays (MCA's) and *mesh-connected iterative arrays* (MIA's) are the two-dimensional analogs of LCA's and LIA's. Here we are mostly interested in the arrays with one-way communication. A one-way mesh-connected cellular array (OMCA) and a one-way mesh-connected iterative array (OMIA) are

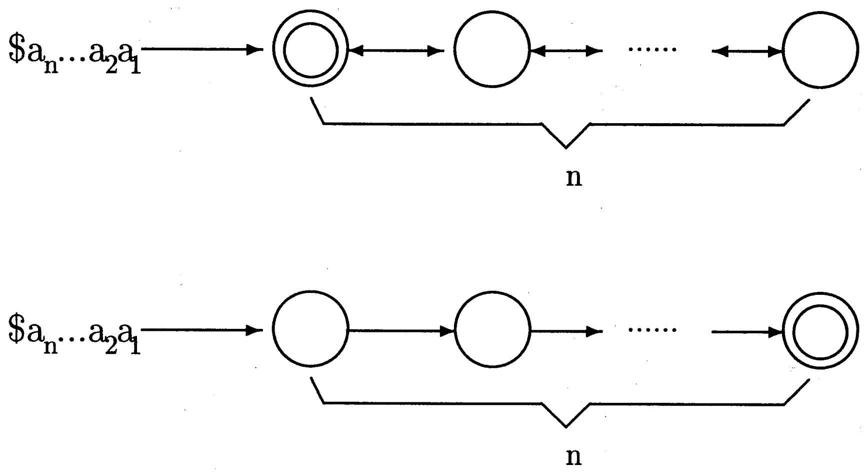

Figure 3. An LIA and an OLIA.

shown in Figure 4.

To simplify the presentation, we introduce the following notations.

1. For any class C of machines and function $T(n)$, $C(T(n))$ denotes the machines in C operating in time $T(n)$.
2. Let C_1 and C_2 be two classes of machines. $C_1 \subseteq C_2$ means that every machine M_1 in C_1 can be simulated by some machine M_2 in C_2. $C_1 \subset C_2$ means that $C_1 \subseteq C_2$ and there is a machine in C_2 that cannot be simulated by any machine in C_1.

3 One-way versus two-way

The question of whether one-way communication reduces the power of a linear array has remained open for many years. In particular, we do not know if OLCA = LCA and if OLIA = LIA. The following result shows that an LIA and an LCA can simulate each other with a delay of at most n steps [IBAR85b].

Theorem 3.1. For any $T(n) \geq n$,

1. $\text{LCA}(T(n)) \subseteq \text{LIA}(T(n)+n)$;
2. $\text{LIA}(T(n)+n) \subseteq \text{LCA}(T(n)+n)$.

Hence LIA = LCA. It seems difficult to precisely characterize the computational complexity of an OLCA or an OLIA. Nevertheless, it has been shown in [CHAN88b] that OLIA's are actually very powerful since they can simulate the computations of nondeterministic $n^{1/2}$-space bounded TM's, i.e.,

Theorem 3.2 $\text{NSPACE}(n^{1/2}) \subseteq \text{OLIA}$.

Clearly, OLCA $\subseteq$ OLIA. On the other hand, it has also been shown (quite surprisingly) that every OLIA can be simulated by an OLCA [IBAR87]. The difficulty arises from the fact that in an OLIA, *every* node of the array has access to *each* symbol of the input string, whereas in an OLCA, the i-th cell can only access the first i symbols of the input.

Theorem 3.3. OLCA = OLIA.

With respect to one-way versus two-way communication, it seems unlikely that OLCA = LCA. On the other hand, it is easy to show that LCA = DSPACE(n), so proving OLCA $\subset$ LCA would imply $\text{NSPACE}(n^{1/2}) \subset \text{DSPACE}(n)$, which would be an improvement of Savitch's well-known result [SAVI70]. This should explain why the one-way communication versus two-way communication problem for linear arrays is hard.

4 The complexity of mesh-connected arrays

We now consider mesh-connected arrays, especially the ones with one-way communication. Theorem 3.3 can be easily extended to OMCA and OMIA.

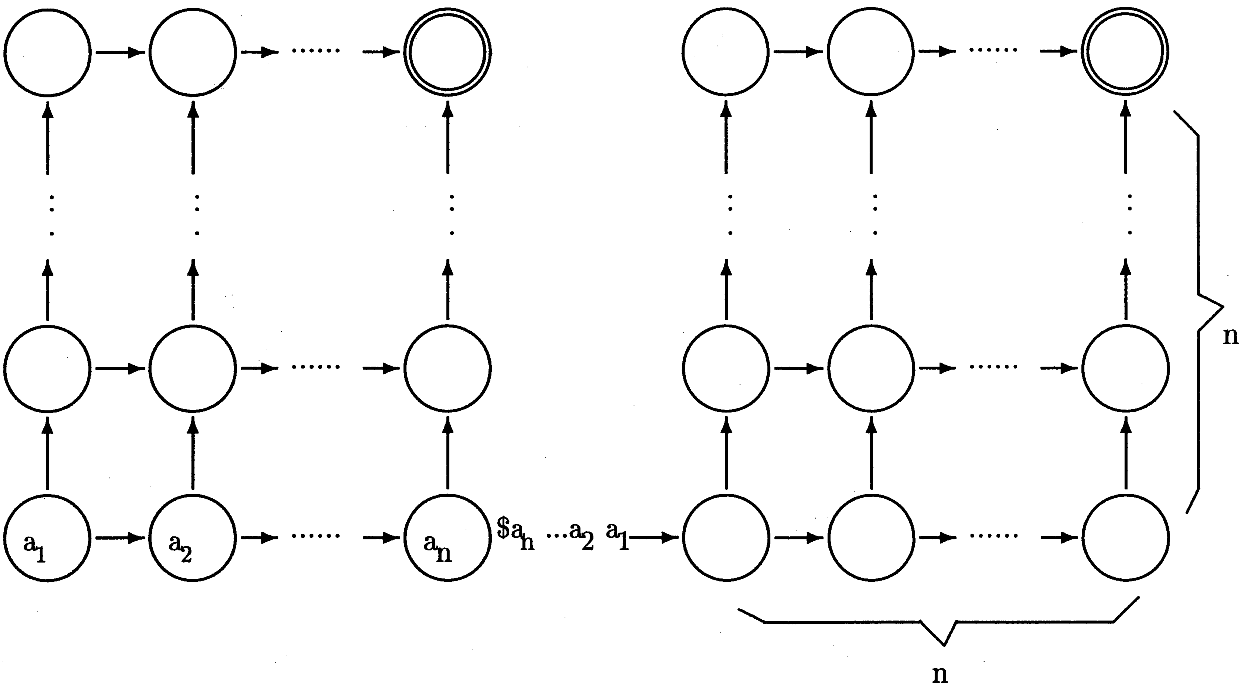

Figure 4. An OMCA and an OMIA.

Theorem 4.1. OMCA = OMIA.

The one-way communication versus two-way communication question can be answered for mesh-connected arrays, because of the following space-efficient simulation result for OMCA's and OMIA's [CHAN88a].

Theorem 4.2. OMCA = OMIA $\subseteq$ DSPACE($n^{3/2}$).

It is not known if the the above space bound is the best possible. In fact, we do not know if OMCA's are more powerful than OLCA's. We also do not know the relationship between OMCA's and LCA's. It is easy to show that MCA = MIA = DSPACE(n^2). It follows from Theorem 3.2 and the space hierarchy theorem for TM/s that one-way mesh-connected arrays are weaker than their two-way counterparts:

Theorem 4.3. OMCA (= OMIA) $\subset$ MCA (= MIA).

OMCA's and OMIA's are quite powerful. They can accept fairly complex languages efficiently. For example, the following result can be shown [IBAR86]:

Theorem 4.4. OMIA's (OMCA's) can accept context-free languages in $2n - 1$ time ($3n - 1$ time), which is optimal with respect to the model of computation.

In our definition of an MCA (or an MIA, or their one-way versions), the number of nodes is the square of the length of the input. It is also interesting to consider mesh-connected arrays where the number of nodes is equal to the length of the input. Denote these models as MCA_1 and MIA_1. The one-way version of these arrays are shown in Figure 5.

We do not know if MCA_1 = OMIA_1. Clearly each OMCA_1 can be simulated by an OMIA_1. It seems difficult to prove the converse. We also do not know if OMCA_1 =OLCA, if OMIA_1 =OLIA, and if OMCA_1's and OMIA_1's can simulate nondeterministic $n^{1/2}$ space-bounded TM's.

5 The parallel complexity of cellular arrays, recurrence equations, and nested loops

In this section, we look at the parallel complexity of real-time OLCA's and pseudo-real-time

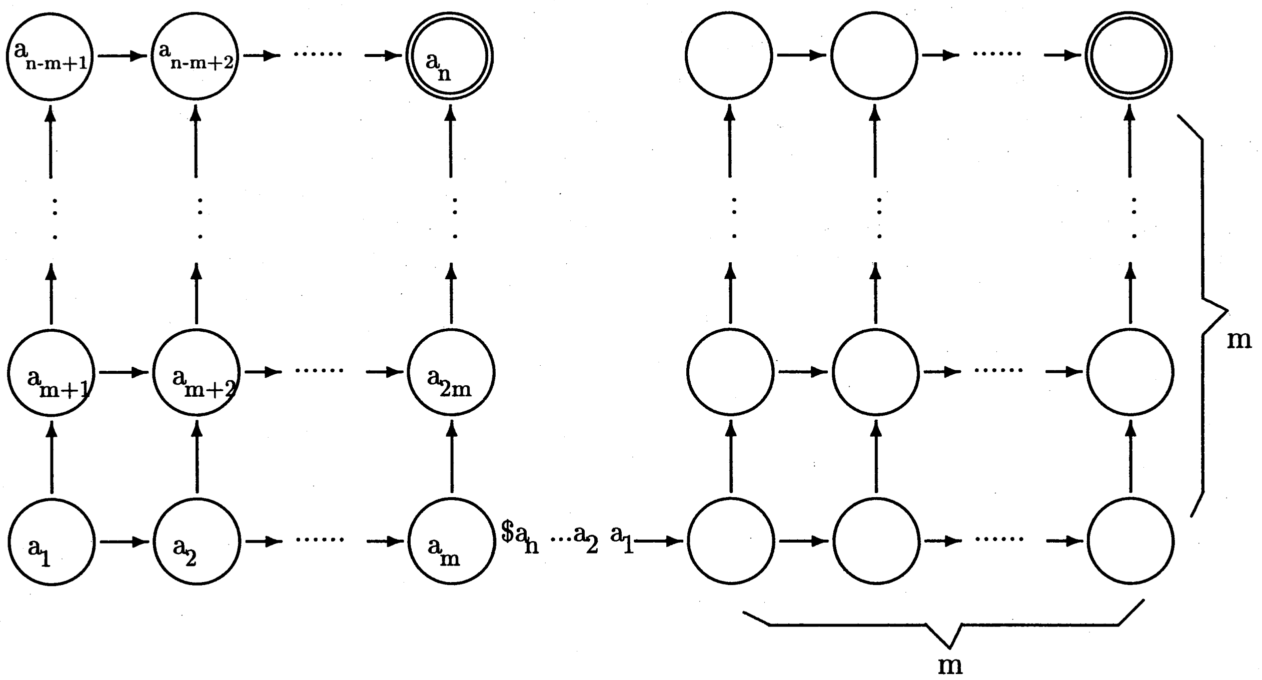

Figure 5. An OMCA_1 and an OMIA_1. (Here, $m = n^{1/2}$.)

OLIA's. We show that it is unlikely that the classes of languages accepted by these arrays are contained in the class NC, which is defined as follows: NC^i is the class of languages accepted by uniform boolean circuits of polynomial size and depth $O(\log^i n)$, and $NC = \bigcup_{i \geq 1} NC^i$ [RUZZ81, COOK85]. Thus, it is unlikely that a general technique can be found that maps any real-time OLCA (or pseudo-real-time OLIA) algorithm into a parallel random-access machine (PRAM) algorithm that runs in polylogarithmic time using a polynomial number of processors.

Formally, we show that there is a language accepted by a real-time OLCA (and by a pseudo-real-time OLIA) that is P-complete. Hence, if such a language is in NC, then P (= the class of languages accepted by deterministic Turing machines in polynomial time) equals NC, which is widely believed to be unlikely.

Theorem 5.1. There is a real-time OLCA (respectively a pseudo-real-time OLIA) that accepts a P-complete language L.

Proof. (Omitted.)

Many computational problems can often be expressed in terms of recurrence equations or simple nested loops. Examples are problems in computational linear algebra, signal processing, and dynamic programming. Thus, efficient sequential and parallel algorithms for solving recurrence equations are of great practical interest.

We can use Theorem 5.1 to show that recurrence equations, even the simple ones, are not likely to admit fast parallel algorithms, i.e., they are not likely to be in NC. Consider the following recurrence equation:

$$R(0,0) = c$$

$$R(i,j) = f(a_i, R(i-1,j), b_j, R(i,j-1)),$$

for $0 \leq i \leq n, 0 \leq j \leq m$ such that $i + j \geq 1$, where $c, a_r, b_s (1 \leq r \leq n, 1 \leq s \leq m)$ and the values of the $R(i,j)$'s are symbols from some fixed finite alphabet, and f is a finite function of four arguments. We assume without loss of generality that $m \leq n$. For notational convenience, let $a_0 = b_0 = \epsilon$, and the boundary conditions $R(i,-1) = R(-1,i) = \epsilon$, where ϵ is a dummy symbol. The objective is to compute $R(n,m)$. We can have f depend also on $R(i-1,j-1)$; however, this dependence can be removed by a simple coding technique. Note that the above recurrence can be written in a form a doubly-nested loop.

Clearly, this recurrence equation can be solved by a parallel algorithm in linear time using a linear number of processors by computing along the diagonals of the recurrence table $R(i,j)$. We can show that it is unlikely that it can be solved by a parallel algorithm in polylogarithmic time using a polynomial number of processors, i.e., it is unlikely that it belongs in the class NC.

Theorem 5.2. There is a recurrence equation of the form above that accepts a P-complete language. Thus, it is unlikely that such a recurrence can be solved by a parallel algorithm in polylogarithmic time using a polynomial number of processors.

Proof. The idea is to show that the computation of a real-time OLCA can be reduced to solving the above recurrence. The result then follows from Theorem 5.1.

6 Reachability and safety in parallel machines

We consider machines that can only use instructions of the form:

$q : x \leftarrow x + c$ then goto p

$q : x \leftarrow x - c$ then goto p

$q :$ if $x \# c$ then goto p_1 else goto p_2

$q :$ goto p

$q :$ goto p_1 or goto p_2

Here x denotes a counter (or variable) that can only assume integer values, $q, p, \ldots$ are states (or labels), c is an integer constant, and $\#$ is $<$, $=$, or $>$. Note that we allow a nondeterministic instruction "goto p_1 or goto p_2". This is sufficient to simulate all other types of nondeterminism. Without loss of generality, we assume the counters can only take on nonnegative values (since the states can remember the sign). We also assume that each instruction takes one time unit to execute and that the instructions are labeled $1, 2, \ldots, n$. Machines with no counters correspond to *finite-state* machines since the only instructions are the "goto" instructions.

Suppose we are given a problem with input domain X, n nondeterministic machines $M_1, \ldots, M_n$, and an input x in X which can be partitioned into n components $x_1, \ldots, x_n$. Each machine M_i is to "work" on x_i to obtain a partial solution y_i. The solution to x can be derived from $y_1, \ldots, y_k$. We want to know if there is a computation (note that since the machines are nondeterministic, there may be several such computation) in which each M_i on input x_i outputs y_i and their running times t_i's satisfy a given linear relation (definable by a Presburger formula). An example of a relation is for each t_i to be within 5% of the average of the running times (i.e., the load is approximately balanced among the M_i's), or for the t_i's to satisfy some precedence constraints. Note that the t_i's need not be optimal as long as they satisfy the given linear relation. A stronger requirement is to find the optimal running time t_i of each P_i and determine if $t_1, \ldots, t_n$ satisfy the given linear relation.

The questions above are unsolvable (no algorithms exist) in general, even when the machines work independently (no sharing of data/variables). This is because a machine with two counters is equivalent to a TM and, hence, the halting problem is undecidable.

Now suppose we restrict the operation of each counter to be reversal-bounded in the sense that it changes mode from nondecreasing to nonincreasing and vice-versa by at most a fixed constant independent of the computation. Call such counters *reversal-bounded.* For example, a counter with the following behavior: 0 1 1 2 3 3 3 4 5 5 5 4 3 2 1 1 1 2 2 3 4 is 2-reversal.

We are able to show that the problems above are solvable (algorithms exits) for reversal-bounded multicounter machines, even when they are augmented with a pushdown stack.

Formally, let M_1 and M_2 be nondeterministic reversal-bounded multicounter machines with a pushdown stack but no input tape operating independently in parallel. Call them PCMs. For $i = 1, 2$, denote by α_i a configuration (q_i, X_i, w_i) of M_i (state, counter values, stack content). Let $L(m, n)$ be a linear relation definable by a Presburger formula. Define $Reach(M_1, M_2, L)$ to be the set of all 4-tuples $(\alpha_1, \beta_1, \alpha_2, \beta_2)$ such that for some t_1, t_2, M_i when started in configuration α_i can reach configuration β_i at time t_i, and t_1 and t_2 satisfy L, i.e., $L(t_1, t_2)$ is true. (Thus, e.g., if the linear relation is $t_1 = t_2$, then we want to determine if M_1 when started in configuration α_1 reaches β_1 at the same time that M_2 when started in α_2 reaches β_2.) We can show the following [IBAR00]:

Theorem 6.1 *Reach*(M_1, M_2, L) is effective computable, i.e., there is an algorithm to decide, given a 4-tuple of configurations $(\alpha_1, \beta_1, \alpha_2, \beta_2)$, whether it is in *Reach*. Moreover, the emptiness problem for *Reach* (i.e., determining if the set is empty) is decidable (an algorithm exits).

The ability to decide whether *Reach* is empty is important and very useful in optimizing load distributions in parallel machines, model-checking, and verification of reactive systems.

We can allow the machines M_1 and M_2 to share some common read-only data, i.e., each machine has a one-way read-only input head. A configuration α_i will now be a 4-tuple (q_i, X_i, w_i, h_i), where h_i is the position of the input head on the common input x. If only one of M_1 and M_2 has a pushdown stack, then the reachability set is still computable and its emptiness decidable. However, this result is not true if both M_1 and M_2 have a pushdown stack, even in the case when each stack is one-turn (after "popping"the stack can no longer "push"), there are no counters, and the linear relation is $t_1 = t_2$.

Looking now at parallel machines that communicate, consider two nondeterministic reversal-bounded multicounter machines (without pushdown) M_1 and M_2 that are connected by a queue. Thus, there is an unrestricted queue that can be used to send messages from M_1 (the "writer") to M_2 (the "reader"). There is no bound on the length of the queue. When M_2 tries to read from an empty queue, it receives an "empty-queue" signal. When this happens, M_2 can continue doing other computation and can access the queue at a later time. Again, M_1 and M_2 operate at the same clock rate, i.e., each transition (instruction) takes one time unit. There is no central control. Call the two machines connected by a queue a queue-connected system M. We have investigated the decidable properties of such queue-connected systems. For example, we can show that it is decidable (an algorithm exists) to determine, given such a system, whether there is some computation in which M_2 attempts to read from an empty queue. Define a configuration of M (at a given time) to be a 5-tuple $\alpha = (q_1, X_1, w, q_2, X_2)$, where w is the content of the stack, and q_i and X_i are the state and set of counter values of M_i, $i = 1, 2$. Let *Reach*(M_1, M_2) = set of all pairs of configurations (α, β) such that M when started in α at some time t will reach β at some time t'. We can show that the binary reachability set *Reach* is computable, and its emptiness is decidable.

Generalizing the model of the queue-connected system slightly yields unsolvable results (no algorithms exist). For example, even when M_1 and M_2 have *no* counters (i.e., they are finite-state): (1) If there are two queues that M_1 can use to send messages to M_2, then reachability is not computable. (2) If M_2 can also send messages to M_1 via another queue, then reachability is not computable. Also, interestingly, if there is a central control that can coordinate the computations of M_1 and M_2, then reachabilty is not computable.

In the area of verification, a typical problem (*safety analysis problem*) is the following: Given a machine M and two sets of configurations I and G, verify if "from any configuration in I, M can only reach configurations in G." Let B be the complement of G. Then we can rewrite the question to "Is there a configuration α in I that can reach some configuration in B?" Assuming that I and G are computable (hence also B), then M is safe if *Reach*$(M) \cap (I \times B) = \emptyset$. Thus, the safety question is reducible to the decidability of emptiness of *Reach*(M).

7 The complexity of commutativity analysis

Program analysis has been widely used to extract program properties of interest. Here we look at a simple property between two program operations – commutativity. We say two operations A and B, each composed of a sequence of basic instructions, commute if they generate the same result regardless of the order in which they execute. Knowledge of commuting operations is of practical significance. In the context of optimizing compilers, commuting program transformations can be used to reduce the search space for the optimal program transformation sequence, hence reducing the algorithmic complexity of the compiler optimization algorithms [SETH74]. In the context of parallel computing, commuting operations enable concurrent execution because they can execute in any order without changing the final result [STEE90,SOLW93]. Parallelizing compilers that recognize commuting operations can exploit this property to automatically generate parallel code for computations that consist only of commuting operations [RINA95].

This broad range of applications motivates the design of static analysis techniques capable of automatically detecting commuting operations — commutativity analysis. We have investigated the theoretical aspects of commutativity analysis and have identified classes of programs for which commutativity analysis is undecidable, PSPACE-hard, NP-hard, polynomial, and probabilistically polynomial-time decidable. For some cases we have shown that the class of programs is complete for the corresponding complexity class. The results rely on known complexity results from the area of theoretical computer science. They serve two purposes. First, they formally establish the complexity of commutativity analysis. Second, they should make researchers working in more applied areas aware of the inherent limitations of any commutativity analysis algorithm.

Acknowledgment: This work was supported in part by NSF Grant IRI-9700370.

References

[BUCH84] Bucher, W. and K. Culik II, On real time and linear time cellular automata, *R.A.I.R.O. Informatique theorique/Theoretical Infomatics* 18-4, 1984, pp. 307-325.

[CHAN88a] Chang, J., O. Ibarra, and M. Palis, Efficient simulations of simple models of parallel computation by space-bounded TMs and time-bounded alternating TMs, in the *Proceedings of the 14th ICALP*, 1988, Finland; expanded version in *Theoretical Computer Science* 68, 19-36, 1989.

[CHAN88b] Chang, J., O. Ibarra, and A. Vergis, On the power of one-way communication, *J. ACM* 35, 1988, pp. 697-726.

[CHOF84] Choffrut, C. and K. Culik II, On real-time cellular automata and trellis automata, *Acta Inform.* 21, 1984, pp. 393-409.

[COLE69] Cole, S., Real-time computation by n-dimensional iterative arrays of finite-state machines, *IEEE Trans. Comput.* 18-4, 1969, pp. 346-365.

[COOK85] Cook, S., A taxonomy of problems with fast parallel algorithms, *Information and Control* 64, 1985, pp. 2-22.

[DYER80] Dyer, C., One-way bounded cellular automata, *Information and Control* 44, 1980, pp. 54-69.

[HENN61] Hennie, F., Iterative arrays of logical circuits, *MIT Press, Cambridge, Mass.,* 1961.

[IBAR85b] Ibarra, O., M. Palis, and S. Kim, Some results concerning linear iterative (systolic) arrays, *J. of Parallel and Distributed Computing* 2, 1985, pp. 182-218.

[IBAR86] Ibarra, O., S. Kim, and M. Palis, Designing Systolic Algorithms using sequential machines, *IEEE Trans. on Computers* C35-6, 1986, pp. 31-42; extended abstract in *Proc. 25th IEEE Symposium on Foundations of Computer Science*, 1984, pp. 46-55.

[IBAR87] Ibarra, O. and Jiang, T., On one-way cellular arrays, *SIAM J. on Computing* 16, 1987, pp. 1135-1154; prelim. version in *Proc. 13th ICALP*, 1987, Karlsruhe, West Germany.

[IBAR88a] Ibarra O., and M. Palis, Two-dimensional systolic arrays: characterizations and applications, *Theoretical Computer Science* 57, 1988, pp. 47-86.

[IBAR00] Ibarra O., Reachability and safety in queue systems with counters and pushdown Stack, *Int. Conf. on Implementation and Application of Automata, 2000.*

[KOSA74] Kosaraju, S., On some open problems in the theory of cellular automata, *IEEE Trans. on Computers* C-23, 1974, pp. 561-565.

[RINA95] Rinard,M. and P. Diniz., Automatically parallelizing serial programs using commutativity analysis, submitted for publication.

[RUZZ81] Ruzzo, W., On uniform circuit complexity, *Journal of Computer and System Sciences* 22, 1981, pp. 365-383.

[SAVI70] Savitch, W., Relationships between nondeterministic and deterministic complexities, *J. Comp. System Sci.* 4, 1970, pp. 177-192.

[SETH74] Sethi, R., Testing for the Church-Rooser property, *Journal of the ACM*, 21(4), 1974, pp. 671-679.

[SMIT70] Smith, A., III, Cellular automata and formal languages, *Proc. 11th IEEE Ann. Symp. on Switching and Automata Theory*, 1970, pp. 216-224.

[SMIT71] Smith, A., III, Cellular automata complexity trade-offs, *Information and Control* 18, 1971, pp. 466-482.

[SMIT72] Smith, A., III, Real-time language recognition by one-dimensional cellular automata, *J. Comp. System Sci.* 6, 1972, pp. 233-253.

[SOLW93] Solworth, J. and B. Reagan, Arbitrary order operations on trees, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Portland, OR, August 1993.

[STEE90] Steele, G., Making asynchronous parallelism safe for the world, *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 218–231, San Francisco, CA, January 1990.

[UMEO82] Umeo, H., K. Morita, and K. Sugata, Deterministic one-way simulation of two-way real-time cellular automata and its related problems, *Inform. Process. Lett.* 14, 1982, pp. 159-161.