

The NP-Completeness Column: An Ongoing Guide

DAVID S. JOHNSON

Bell Laboratories, Murray Hill, New Jersey 07974

This is the eighth edition of a quarterly column the purpose of which is to provide continuing coverage of new developments in the theory of NP-completeness. The presentation is modeled on that used by M. R. Garey and myself in our book "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman & Co., San Francisco, 1979 (hereinafter referred to as "[G&J]"; previous columns will be referred to by their dates). A background equivalent to that provided by [G&J] is assumed, and, when appropriate, cross-references will be given to that book and the list of problems (NP-complete and harder) presented there. Readers who have results they would like mentioned (NP-hardness, PSPACE-hardness, polynomial-time-solvability, etc.), or open problems they would like publicized, should send them to David S. Johnson, Room 2C-355, Bell Laboratories, Murray Hill, NJ 07974, including details, or at least sketches, of any new proofs (full papers are preferred). In the case of unpublished results, please state explicitly that you would like the results to be mentioned in the column. Comments and corrections are also welcome. For more details on the nature of the column and the form of desired submissions, see the December 1981 issue of this Journal.

1. INTRODUCTION

This month's column continues the discussion of parallel computing begun in the last issue. Last time I concentrated on the complexity issues involved in getting the most computation for your money out of parallelism, both by clever parallel algorithms and by the efficient scheduling of parallel processors. This column focuses on the lower-level question of how to coordinate collections of concurrent processes so that they may share resources while still avoiding deadlock and incorrect results.

Although personal computers, with their own dedicated disks, printers, etc., are becoming ever more popular, problems involving shared resources are unlikely to go away. Economic considerations still make it necessary for users to share certain resources, such as phototypesetters and high speed "number crunchers." Other resources, such as common databases, are of value precisely because they are shared. Moreover, I would like to be able to run more than one program at a time on my personal computer, with all the sharing of internal resources that this would entail (by processes that are at least *conceptually*

parallel).

I begin in Section 2 with general problems of resource sharing and the potential for deadlock that this implies. Section 3 then specializes this to the case of shared databases, where the maintenance of database consistency also becomes an issue. In contrast to the wide-ranging nature of past columns, this will be a tour of a relatively narrow area, encompassing a concentration of closely inter-related results. (In fact, the column can be viewed as an expansion and update on just three entries in [G&J].) As researchers have continued to explore the issues involved in resource sharing, proposing new models, new protocols, etc., they have obtained more and more detailed complexity results. In order to present these results in a comprehensible fashion, I shall try, where possible, to provide motivation for the underlying models and to explain some of the basic issues involved. The column can thus be viewed as a tutorial on deadlock prevention and concurrency control, biased of course toward those questions that lend themselves to complexity analysis; tutorials with different biases can be found in [3,15]. Readers interested in further information on this active area of research are directed to these and to the other papers cited in this column's references. (Readers who decide that the column already provides far more than they will ever want to know about the subject are directed to the paragraph *before* the references, where I say something about future, less specialized columns.)

2. LIVING WITH DEADLOCK

One of the most basic problems arising from shared resources is that of DEADLOCK AVOIDANCE ([SS22] in [G&J]). If a resource must be shared, then sometimes processes must wait for access to it. A waiting process may in turn hold just enough of another resource to block other processes from proceeding. In this way we could end up with a collection of processes, each waiting on the others and none able to proceed, a situation commonly called *deadlock* (or, in certain New York City vehicular applications where the resources are street intersections, *gridlock*). As the description of the above problem in [G&J] was only slightly less vague than the preceding sentence, I shall begin by presenting the problem over again, this time with the details spelled out.

In the basic model for this and most of the remaining problems, a *process* is viewed as a sequence of steps, and processes are made *concurrent* by interleaving their steps. For convenience in this first problem, we shall view the steps of a process as the vertices of a directed graph, that graph being a simple path with the first step as its *root* and the last step as its (only) *leaf*. Associated with each vertex v is a *resource allocation vector* $\vec{a}(v) = \langle a_1(v), \dots, a_m(v) \rangle$, where m is the number of resources in the system. If $a_j(v)$ is positive, this means that when step v is performed, $a_j(v)$ units of the j th resource are allocated to the process containing v , which will retain exclusive use of them until it returns them at a later step (if it ever does). A negative $a_j(v)$ indicates the deallocation or return

of that many units of the resource to the common pool, where they will be available to other processes.

A *partial schedule* for a collection P of processes is a sequence $S = \langle u_1, u_2, \dots, u_T \rangle$ obtained by taking a prefix (initial segment) S_k of each process p_k in P and shuffling them together, so that each S_k is a (not necessarily contiguous) subsequence of S . Given a collection of *resource bounds* B_1, \dots, B_m , S is a *legal* partial schedule if it obeys the resource constraints, i.e., for each j , $1 \leq j \leq m$, and for all t , $1 \leq t \leq T$, $0 \leq \sum_{i=1}^t a_j(u_i) \leq B_j$. (In what follows, we make the reasonable stipulation that each process p_k is *legal*, i.e., is such that the partial schedule consisting of p_k by itself is legal.) A legal partial schedule is a (*total*) *schedule* if it contains all vertices of all processes. A legal partial schedule $S = \langle u_1, \dots, u_T \rangle$ is a *deadlocked* schedule if (a) there is some process that is not entirely contained in S and (b) for each process p_k not entirely contained in S , the first vertex v of p_k that is not in S satisfies $a_j(v) + \sum_{i=1}^T a_j(u_i) > B_j$ for some j .

We are now ready to state our first problem.

[1] DEADLOCK AVOIDANCE

INSTANCE: Set $\{R_1, \dots, R_m\}$ of resources with integer resource bounds B_1, \dots, B_m , set $P = \{p_1, \dots, p_n\}$ of legal processes having resource allocation vectors $\bar{a}(v)$ for each vertex v , and a legal partial schedule S for P .

QUESTION: Can S be extended to a total schedule for P , i.e., is there a total schedule for P which has S as a prefix?

Reference. Araki, Sugiyama, Kasami, and Okui [1,31] and Gold [7]. Transformation from 3SAT.

Comment. NP-complete even if all resources are “unit resources” (all B_j equal 1) and there are no “joint allocations” (no $\bar{a}(v)$ contains more than one non-zero entry) [1]. Solvable in polynomial time if there are no joint allocations and the allocations and deallocations in each process are “well-nested.” (Although the NP-completeness proof in reference [7], implies the current result, it actually concerns a slightly more complicated model, in which processes can *create* as well as use resources. For a discussion of the model and its polynomial-time solvable subcases, see [7].)

The problem as described here models the dilemma faced by an “on-line scheduler” or “resource manager,” a character that will play a major role in some of the later problems presented in this column. Such a scheduler is in charge of deciding which processes get access to the resources, and in which order. Whenever a process is ready to go on to its next step (next vertex v), it sends a request to the scheduler for permission to proceed. The scheduler then can either grant the request (thus causing the allocations and deallocations of

$\bar{a}(v)$ to be made) or put the request on hold (thus delaying the process). It can also choose to grant an earlier request that was previously put on hold. At all times, the sequence of granted requests must form a legal partial schedule (the resource constraints must be obeyed). We assume the scheduler knows in advance the structure of all the processes, but not the order in which their requests will arrive.

An on-line scheduler can easily avoid deadlock if it is allowed to have control from the beginning, when all processes are waiting to execute their roots. For instance, the scheduler could just forget about concurrency and execute process p_1 to completion, then p_2 , etc. More sophisticated schedulers exist, ones that run in polynomial time and do allow some concurrency while guaranteeing freedom from deadlock (e.g., see [20,33]). However, these still limit concurrency by delaying requests according to pre-specified rules, whether the delay is absolutely necessary or not. Ideally, a scheduler should never delay a request unless granting the request would either violate a resource constraint or else make deadlock unavoidable, i.e., yield a partial schedule that is *not* extendable to a total schedule. The current NP-completeness result tells us that, like other ideals, this one is (for all practical purposes) unattainable.

The ideal becomes even further out of reach if we generalize the notion of process to include processes with conditional branches. These can be modeled by out-trees (the branch points correspond to the vertices with out-degree exceeding 1). Here a partial schedule consists of a shuffling together of directed paths from the process roots, and is a total schedule if all the paths end in leaves. If, given a partial schedule, we wish to know whether it can be extended to a total schedule, no matter what choices are made at the remaining branch points, we face a PSPACE-complete problem, as does our poor on-line scheduler, if it wishes to guarantee that deadlock will be avoided, even though it does not know what choices the processes may make in the future [20]. This holds even if all resources are unit resources. However, if the allocations and deallocations are “well-nested” and there are no joint allocations, then even this branching problem can be solved in polynomial time [1]. Also, efficient on-line schedulers again exist if we are allowed to start from the empty partial schedule and sacrifice a bit of concurrency [20].

Given that guaranteeing freedom from deadlock seems to be impossible without either substantial computational effort or else loss of concurrency, one might be tempted to try a scheduler that emphasizes concurrency and simply ignores the possibility of deadlock, hoping that Murphy’s famous Law* will not apply. The next two problems relate to this approach. The first concerns the scheduler that grants each request as received, so long as it does not violate the resource constraints. The question now is not “Can deadlock be avoided?” but rather

* “If anything can go wrong, it will” (and if you had to read this footnote, it already has).

“Is it possible that deadlock may occur?”

[2] DEADLOCK POTENTIAL

INSTANCE: As in the previous problem.

QUESTION: Can the partial schedule S be extended to a deadlocked schedule for P , i.e., is there a deadlocked schedule for P that contains S as a prefix?

Reference. Minoura [20]. Transformation from 3SAT.

Comment. In NP even for branching processes, and NP-complete even for straight-line processes and even if the specified partial schedule is the empty (initial) schedule (in which case the DEADLOCK AVOIDANCE problem was polynomial even for branching processes, as deadlock could always be avoided by clever scheduling). Moreover, the above holds even if each process obeys the “two-phase locking” discipline, i.e., consists of a “locking phase,” in which all allocations are made, followed by an “unlocking phase,” during which all resources are deallocated [33]. Can be solved in polynomial time by a clever geometric approach if only unit resources are allowed and there are just K (straight-line) processes, for any fixed K [19,22,28,32,35]. Deadlock prediction can also be accomplished in polynomial time for the following variant of the given problem [10]: each process p is represented by a vector $\bar{m}(p)$, where $m_j(p) \leq B_j$ is the maximum amount of resource R_j that process p can ever use at one time. At any time process p can deallocate some of the resources it holds or request additional amounts, so long as granting the request will not cause the process to exceed any of its maxima. The scheduler must grant the request if the resource is available; otherwise it “blocks” the process, and the process cannot make any further steps until the resource is available and is given to it. Deadlock occurs if there is a non-empty set of blocked states that will remain blocked even if all non-blocked states return all the resources currently allocated to them.

One need not be able to solve the DEADLOCK POTENTIAL problem in order to use the “grant any grantable request” scheduler we have just been discussing. However, since we also haven’t solved the DEADLOCK AVOIDANCE problem, the following may well be relevant.

[3] DEADLOCK RECOVERY

INSTANCE: Set $\{R_1, \dots, R_m\}$ of *resources* with integer *resource bounds* B_1, \dots, B_m , set P of *blocked processes*, each process p having a non-negative integer *cost* $c(p)$, a vector $\bar{r}(p)$ of currently allocated resource amounts, and a vector $\bar{a}(p)$ of current requests satisfying (a) $r_j(p) + a_j(p) \leq B_j$, $1 \leq j \leq m$,

and (b) for some j , $1 \leq j \leq m$, $a_j(p) + \sum_{p \in P} r_j(p) > B_j$, and an overall cost bound K .

QUESTION: Is there a subset $P' \subseteq P$ with $\sum_{p \in P'} c(p) \leq K$ such that if all processes in P' are aborted and their resources returned to the common pool, then the remaining processes in $P - P'$ can all proceed, i.e., such that for all j , $1 \leq j \leq m$, $\sum_{p \in P - P'} r_j(p) + \sum_{p \in P - P'} a_j(p) \leq B_j$?

Reference. Leung and Lai [18]. Transformation from 3-PARTITION.

Comment. NP-complete in the strong sense if the number m of resources is arbitrary; NP-complete but solvable in time pseudo-polynomial in K if $m = 1$ [17]. Note, however, that the deadlock could possibly be broken more cheaply if we did not require that *all* unaborted processes be able to proceed immediately, but only required that at least one of them be allowed to proceed, in hopes that the others might later become unblocked. Minimizing cost for this type of recovery is, of course, also NP-complete [9].

3. DATABASE CONCURRENCY

Let us now turn to the case where the “shared resources” are the entries in a common database. In this case we would like not only to avoid deadlock, but also to maintain the consistency of the database (under the unlikely assumption that databases are consistent in the first place), and to ensure that the transactions have, in some sense, the “correct” effect. If we restrict ourselves to straight-line processes and to the terminology of database theorists, then the processes of the previous section become *transactions*, the vertices become *actions* or *steps*, and schedules are sometimes called *histories*. In addition, actions are viewed as affecting (and being affected by) the shared resources, which in this case are the database variables. The basic actions are *read(x)*, which reads the variable x and stores its value for future reference, and *write(x)* which changes the variable to a new value. We will sometimes, however, consider combinations of these basic actions as atomic steps, such as *read(X)*, where X is a *set* of variables. We view the resources (i.e., variables) as allocated only *during* the steps, not between them, so that no transaction can hold resources while waiting to proceed. Deadlock is therefore not a problem in the model so far elaborated (but just wait a page or two). The notions of “consistency preservation” and “correctness” are captured by the concept of a “serializable schedule.”

A schedule S is a *serial* schedule if it corresponds to a permutation $T_{\pi(1)}, \dots, T_{\pi(n)}$ of the transactions, with all the steps of $T_{\pi(i)}$ preceding all the steps of $T_{\pi(i+1)}$, $1 \leq i < n$. If the individual transactions preserve consistency and yield correct results, one would expect a serial schedule to do likewise. A schedule S is *serializable* if there is some serial schedule that yields the same

final values for the variables as S does, assuming that the result of each *write* depends on the values of all variables that have previously been *read* by the transaction containing the *write*, and no matter how it depends on those values. (This assumption also underlies the other definitions of “equivalence” we shall be encountering). Note that this is essentially a *syntactic* definition that makes no use of any information about the purposes of the transactions or the semantics of the database contents. It is thus clearly appropriate for database systems designed with multiple applications in mind. However, it is not unreasonable to use it even in situations where the database semantics are known. (Using such semantic information might lead to a less-restrictive consistency criterion, but the path is treacherous, with undecidability, not just NP-completeness, lurking around every corner.) See [21,23] for further discussions and justifications.

In general, testing for serializability is NP-complete, even if each transaction is a “two-step” transaction, i.e., consists of an atomic *read*(X) followed by an atomic *write*(Y), where X and Y are possibly distinct sets of variables [21,23]. (See also SERIALIZABILITY OF DATABASE HISTORIES [SR33] in [G&J].) Note that although NP-completeness continues to hold if each atomic *read*(X) is replaced by a sequence of *read*(x)’s for the variables $x \in X$ and similarly for the *write*(Y)’s, splitting the atoms in this way gives rise to a wider variety of possible schedules, since now the sub-actions of the *read*(X)’s and *write*(Y)’s may interleave. Testing whether such a schedule is equivalent to one in which the atoms are not split is itself NP-complete, according to [5].

Returning to questions of equivalence to serial schedules, let us consider some variants on the basic notion, determined by different notions of “equivalence.” A recently studied variant, having the same complexity breakdown as the original notion, is the more restrictive “view serializability,” where two schedules, to be equivalent, must agree not only on all final values, but also on all *read*’s as well [34]. Another important variant of serializability is the still more restrictive *conflict serializability* (sometimes called *D-serializability*), based on the notion of “conflict equivalence” [21,29]. Two schedules S and S' are *conflict equivalent* if and only if each variable x receives the same sequence of values in both S and S' . This is the same as saying that S can be obtained from S' simply by a sequence of interchanges involving non-conflicting actions (*read*’s do not conflict with other *read*’s, and a *write*(X) conflicts with a *read*(Y) or a *write*(Y) only if $X \cap Y \neq \emptyset$). Conflict serializability can be tested in polynomial time even if arbitrary atomic combinations of *read*’s and *write*’s are allowed, it being equivalent to the absence of a cycle in an appropriately defined directed graph [34]. (The related concept of “live conflict serializability” can be tested in polynomial time using a similar approach [34].) It should be noted that if no transaction writes a variable before it reads it, then all the notions of serializability discussed in this column become polynomial-time testable, and all but ordinary serializability become identical to conflict serializability [24]. The next two problems concern two further variants of serializability, each of which raises

different issues.

[4] STRICT SERIALIZABILITY

INSTANCE: Schedule S for a set $\{T_1, \dots, T_n\}$ of database transactions, where the actions are atomic $read(X)$'s and $write(X)$'s.

QUESTION: Is S *strictly* serializable, i.e., is it equivalent to a serial schedule S' such that for any pair of transactions T and T' , if all actions of T precede all actions of T' in S , then they also do so in S' ?

Reference. Sethi [25,26]. Transformation from 3SAT.

Comment. Remains NP-complete for two-step transactions, as with ordinary serializability. However, solvable in polynomial time for two-step transactions if we consider only schedules having no "useless" actions (i.e., every variable value that is written by a transaction either survives to the end of the schedule or else is read by some other transaction) [26]. If useless actions are allowed, NP-completeness returns, as it does if multistep transactions are allowed [26]. Note that the presence or absence of useless actions does not alter the NP-completeness of the original version of serializability.

The next version of serializability arises from the following scheme to help databases maintain their long-term consistency [2,4,16,24,30]. Let us return to the assumption that the atomic actions are simply $read(x)$'s and $write(x)$'s. The idea is to save old values of variables along with the updated ones, and to supply each $read(x)$ action with the past value of x that most promotes serializability. In this framework, a schedule can have a variety of *interpretations*, where an interpretation is an assignment to each $read(x)$ action of either the output of some earlier $write(x)$ action or else the original value of x . (In the *standard* interpretation, only the most recent values are used.)

[5] MULTIVERSION SERIALIZABILITY

INSTANCE: Schedule S for a set $\{T_1, \dots, T_n\}$ of database transactions, the $read$ and $write$ actions of which access only single variable sets.

QUESTION: Is S *multiversion serializable*, i.e., is there an interpretation for S that yields the same collection of final variable values as does the standard interpretation of some serial schedule S' ?

Reference. Papadimitriou and Kanellakis [24] (see also [4]). Transformation from POLYGRAPH ACYCLICITY, for which see [21] (and my upcoming column on the complexity of lie detection).

Comment. NP-complete. There is actually a hierarchy of serializability

notions, *k-multiversion serializability*, $1 \leq k \leq \infty$, depending on the maximum number of versions of any one variable that may be retained at any one time. This is a strict hierarchy, with the number of schedules that are “serializable” increasing with k . Ordinary serializability is the $k = 1$ case, multiversion serializability is the $k = \infty$ case, and all cases are NP-complete when a *write(x)* need not be preceded by a *read(x)* [24]. The hierarchy collapses (all notions become equivalent) if the actions are all of the form *update(x)*, an atomic combination of *read(x)* followed by *write(x)*, in which case testing is once again polynomial [24].

Note that testing a given schedule for serializability (of any sort) is a different matter from devising an on-line scheduler that will guarantee it (the analog for deadlock problems would simply be testing whether one’s current state is deadlocked). To guarantee serializability, practitioners have developed such strategies as “two-phase locking” [6], “tree locking” [8,27], and others [13,32]. According to specified rules, these add *lock(x)* and *unlock(x)* actions to the transactions, thus allowing a transaction to prevent other transactions from accessing a variable, even though the first transaction is not currently accessing the variable itself. (*Lock*’s are thus like allocations of the previous section, and *unlock*’s are like deallocations.) The basic idea is to add *lock*’s and *unlock*’s in such a way that, given a simple-minded on-line scheduler that immediately grants any request not violating a lock, one can only generate serializable schedules. In this case we say we have a *safe* locking policy. The above-mentioned locking policies are all safe (note that I am using the term “policy” to refer both to a general strategy for introducing locks and to a specific strategy as applied to a particular transaction system). For more on the theory of locking policies, see the cited references and especially [32,33], where a unifying theory in terms of hypergraphs is developed.

As with allocations and deallocations, *lock*’s and *unlock*’s limit the possibilities for concurrency in the system. Ideally, one would like to maximize concurrency by locking variables only when absolutely necessary to insure serializability, given the current set of transactions. Unfortunately, one is faced with the following analog of the DEADLOCK POTENTIAL problem. (For simplicity we shall state the problem in its most restrictive form, where all actions are atomic “updates” as described in the comments to the previous problem, and hence testing for serializability is easy.)

[6] UNSAFE LOCKING POLICY

INSTANCE: Set $X = \{x_1, \dots, x_m\}$ of variables and a set T_1, \dots, T_n of transactions, each consisting of a sequence of steps of the form *lock(x)*, *update(x)*, and *unlock(x)*, where each *update(x)* occurs between a *lock(x)*-*unlock(x)* pair (although not necessarily contiguous to either), and no transaction contains more

than one $lock(x)$ - $unlock(x)$ pair for any $x \in X$.

QUESTION: Is there a “legal” schedule for the transactions that is not serializable, i.e., is there a non-serializable schedule S such that between any two occurrences of $lock(x)$ in S (for any $x \in X$), there is an occurrence of $unlock(x)$?

Reference. Papadimitriou [22] and Yannakakis [32] (see also [35]). Transformation from 3SAT.

Comment. Solvable in polynomial time if there are just K transactions, for any fixed K [22,32], and under other, more complex restrictions [22,32]. If there are no locks, but separate *read* and *write* actions are allowed, the problem is also NP-complete. The NP-hardness is an old result (see [21,23] and the SAFETY OF DATABASE TRANSACTION SYSTEMS [SR34] problem in [G&J]), but membership in NP has only recently been shown [34]. This latter problem can be solved in polynomial time, however, if “serializability” is replaced by the stronger notion of “view serializability” [34]. As an aside, suppose we are given a set T of transactions without *locks* or *unlocks*. Each safe locking policy L for T defines a set of serializable schedules S_L for T : just delete the *locks* and *unlocks* from the schedules that are legal under L . Given a schedule S and a locking policy L , testing whether $S \in S_L$ is, of course, easy. Surprisingly, given S , testing whether there *exists* an L such that $S \in S_L$ is NP-complete [34].

As soon as we introduce locking into the model, we introduce the potential for deadlock. An on-line scheduler following a safe locking policy is not guaranteed to avoid deadlock; all that is guaranteed is that if a total schedule is generated, then that schedule will be serializable. In fact, the question of whether there is a potential for deadlock is, in general, NP-complete. See the comments about “two-phase locking” under DEADLOCK POTENTIAL.

There is an alternative method for guaranteeing serializability that avoids such dangers by using precedence constraints instead of locks. Given a set T of transactions, we construct a partial order \leq_T on the set of all actions in T such that all the actions of any one transaction are totally ordered (in the order specified by the transaction) and such that any total order consistent with \leq_T represents a serializable schedule. The scheduler then need only delay a ready action A when one of A 's predecessors under \leq_T has not yet finished.

A straightforward way of generating an appropriate partial order, suggested in [14], starts with a serial schedule S for T . Given S , we construct a directed graph $G(S)$ with the actions of T as vertices and with an arc from action u to action v if u precedes v in S and the two actions “conflict” (as in the definition of “conflict serializability”). $G(S)$ then induces our desired partial order, which I shall denote by $\leq_{S(T)}$. All total orders consistent with $\leq_{S(T)}$ will be conflict serializable (in fact, conflict equivalent to S). Given this scheme, it now becomes sensible to consider certain explicit measures of (potential) concurrency, and try

to optimize for them, which is precisely what the next problem attempts to do.

[7] MINIMUM DEPTH D-SERIALIZABLE PARTIAL ORDER

INSTANCE: Set $T = \{T_1, \dots, T_n\}$ of transactions, positive integer K .

QUESTION: Is there a serial schedule S for T such that no directed path in the partial order $\leq_{S(T)}$ has length exceeding K ?

Reference. Krishnamurthy and Dayal [14]. Transformation from GRAPH 3-COLORABILITY.

Comment. Also NP-complete is the related problem where the cost criterion is the number of arcs in $\leq_{S(T)}$. If the cost criterion is the number of total orders consistent with $\leq_{S(T)}$, the problem is NP-hard, but not known to be in NP.

Our next problems concern the maintenance of consistency in *distributed* databases, and extend the use of partial orders to both the transactions and the schedules themselves. In this model the variables are partitioned into sets X_1, \dots, X_m , where each set X_k resides at a different *site*, which we shall identify with the set. Because of uncertainties in communication delays, it is no longer appropriate to view a transaction as simply a sequence of actions. Although all the actions of transaction T at site X may be assumed to be performed in the order requested by T , there is less certainty about the relative starting times of two actions at different sites, unless we are prepared to accept the communication delays required to be sure that one action is finished before we request the other. To avoid such delays when they are unnecessary, we generalize the notion of a *transaction* from a sequence of actions to a partially ordered set $T = (A, \leq)$ of actions, subject to the constraint that the actions involving variables at each individual site are totally ordered. A *schedule* is then a partial order on all the actions in all the transactions that is consistent with each of the individual partial orders and that, for each site X , induces a total order on the actions involving variables at X . A *serial* schedule is one that induces an ordinary serial schedule at each site, and such that the induced total orders on the transactions at the various sites are all consistent with each other. Given this definition of “serial schedule,” the definition of *serializability* is analogous to that for single-site databases.

For a first result about this model, let us restrict our attention to schedules that are total orders. Then the previous problem, UNSAFE LOCKING POLICY, which was polynomial-time solvable if there were just two (single-site) transactions, remains so if the two transactions each involve two sites, but becomes NP-complete if an arbitrary number of sites is allowed in each transaction [12].

For a second result about the model (and for the last result of this column), we come full circle with the following rough analog of DEADLOCK AVOIDANCE, this month’s first problem. In this final problem, “deadlock” is

replaced by “non-serializability,” a central scheduler is replaced by a collection of distributed schedulers, and the possibility of conditional branches whose choices are unpredictable is replaced by the need for minimizing inter-process communication when *delays* are unpredictable. Furthermore, having taken one look at the phrase “NON-SERIALIZABILITY AVOIDANCE,” I think I shall use the following problem name instead.

[8] DISTRIBUTED SERIALIZABILITY ASSURANCE

INSTANCE: Pair X_1, X_2 of sites, set $T = \{T_1, \dots, T_n\}$ of transactions (partial orders in the sense mentioned above), each action of which is of the form $update(x)$ for some variable $x \in X_1 \cup X_2$, positive integer K .

QUESTION: Is there a pair of “optimistic” on-line scheduling strategies $STRAT_1$ and $STRAT_2$, one for each site, that is guaranteed to construct a serializable schedule while sending K or fewer inter-site messages, no matter what the order of arrival at each site of requests from the transactions (assuming that the requests arrive in an order consistent with the partial orders of the individual transactions) and no matter what the communication delays for the messages sent from site to site? A *strategy* is a sequential process that can receive requests from a transaction to perform an action, grant such requests, send messages to other sites, receive such messages, and do local computation. By “optimistic” I mean that if the requests arrive in the order induced by a serializable schedule and all communication delays are 0, then each request must be granted as soon as it is received. This qualification is introduced in order to promote concurrency by preventing unnecessary delays in granting requests.

Reference. Kanellakis and Papadimitriou [11]. Transformation from QUANTIFIED BOOLEAN FORMULAS [LO11].

Comment. PSPACE-complete even if in each transaction the actions at different sites are totally unrelated. NP-complete if $K = 0$ (no communication allowed). Solvable in polynomial time if both restrictions hold [11]. A related question concerns the existence of *general* schedulers, pairs of strategies for which the set T of transactions is itself an input, and which then guarantee serializability as above while being appropriately “optimistic.” In [11] it is shown as a corollary of the above PSPACE-completeness result that, if $NP \neq PSPACE$, no general scheduler can both run in polynomial time (as a function of the size of T) and always send the minimum possible number of messages. (If we do not require communication optimality, polynomial-time schedulers do exist.) In the non-distributed case, similar questions can be asked [21], although now the definition of “optimistic” is tightened to require that requests be granted in the order received, so long as there is some serializable schedule that has the current sequence of requested actions as a prefix. The likelihood of a general scheduler

existing depends both on the notion of serializability being used and the type of actions considered. If arbitrary $read(X)$'s and $write(X)$'s are allowed, then a general polynomial-time scheduler guaranteeing ordinary serializability can exist only if $P = NP$, although polynomial-time schedulers do exist for conflict-serializability as well as a number of other classes [21].

This last problem introduces the issue of coordinating the *communication* between processes, a topic that has a large and varied literature of its own, and to which I shall return in the near future. However, the next column marks our second anniversary, and so deserves special treatment. I shall take it as an opportunity to update many of the results presented during the first two years (and correct a few), as well as to celebrate with a variety of other fun and games. Be sure to invite your friends.

REFERENCES

1. T. ARAKI, Y. SUGIYAMA, T. KASAMI, AND J. OKUI, Complexity of the deadlock avoidance problem, in "Proceedings 2nd IBM Symp. on Mathematical Foundations of Computer Science," pp. 229-252, IBM Japan, Tokyo, 1977.
2. R. BAYER, H. HELLER, AND H. REISER, Parallelism and recovery in databases, *ACM Trans. Database Syst.* **5** (1980), 139-156.
3. P. A. BERNSTEIN AND N. GOODMAN, Concurrency control in distributed database systems, *Comput. Surveys* **13** (1981), 185-222.
4. P. A. BERNSTEIN AND N. GOODMAN, Concurrency control algorithms for multiversion database systems, in "Proceedings 1st ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing," pp. 209-215, Association for Computing Machinery, New York, 1982.
5. K. EKANADHAM AND A. NIGAM, "On serializability," Report No. RC 9257, IBM Research, Yorktown Heights, N.Y., 1981.
6. K. P. ESWARAN, J. N. GRAY, R. A. LORIE, AND I. L. TRAIGER, The notions of consistency and predicate locks in a database system, *Comm. ACM* **19** (1976), 624-633.
7. E. M. GOLD, Deadlock prediction: Easy and hard cases, *SIAM J. Comput.* **7** (1978), 320-336.
8. J. N. GRAY, R. A. LORIE, AND G. R. PUTZOLU, Granularity of locks in a shared data base, in "Proceedings Int. Conf. on Very Large Data Bases," pp. 428-451, Association for Computing Machinery, New York, 1975.
9. D. S. JOHNSON, Exercise for the reader (1983).
10. T. KAMEDA, Testing deadlock-freedom of computer systems, *J. Assoc. Comput. Mach.* **27** (1980), 270-280.
11. P. C. KANELLAKIS AND C. H. PAPADIMITRIOU, The complexity of distributed concurrency control, in "Proceedings 22nd Ann. Symp. on Foundations of Computer Science," pp. 185-197, IEEE Computer Society, Los Angeles, 1981.
12. P. C. KANELLAKIS AND C. H. PAPADIMITRIOU, Is distributed locking harder?, in "Proceedings 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems," pp. 98-107, Association for Computing Machinery, New York, 1982.
13. Z. KEDEM AND A. SILBERSCHATZ, Controlling concurrency using locking protocols, in "Proceedings 20th Ann. Symp. on Foundations of Computer Science," pp. 274-285, IEEE Computer Society, Los Angeles, 1979.
14. R. KRISHNAMURTHY AND U. DAYAL, Theory of serializability for a parallel model of transactions, in "Proceedings 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems," pp. 293-305, Association for Computing Machinery, New York, 1982.

15. W. H. KOHLER, A survey of techniques for synchronization and recovery in decentralized computer systems, *Comput. Surveys* **13** (1981), 149-184.
16. G. LAUSEN, "Serializability problems of interleaved database transactions," Technical Report, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Karlsruhe, West Germany, 1981.
17. J. Y.-T. LEUNG, Complexity of optimal deadlock recovery, manuscript (1983).
18. J. Y.-T. LEUNG AND E. K. LAI, On minimal cost recovery from system deadlock, *IEEE Trans. Computers* **C-28** (1979), 671-677.
19. W. LIPSKI, JR. AND C. H. PAPADIMITRIOU, A fast algorithm for testing safety and detecting deadlocks in locked transaction systems, *J. Algorithms* **2** (1981), 211-226.
20. T. MINOURA, Deadlock avoidance revisited, *J. Assoc. Comput. Mach.* **29** (1982), 1023-1048.
21. C. H. PAPADIMITRIOU, The serializability of concurrent database updates, *J. Assoc. Comput. Mach.* **26** (1979), 631-653.
22. C. H. PAPADIMITRIOU, Concurrency control by locking, *SIAM J. Comput.* **12** (1983), 215-226.
23. C. H. PAPADIMITRIOU, P. A. BERNSTEIN, AND J. B. ROTHNIE, Computational problems related to database concurrency control, in "Proceedings Conf. on Theoretical Computer Science" pp. 275-282, Department of Computer Science, University of Waterloo, Waterloo, Ontario, 1977.
24. C. H. PAPADIMITRIOU AND P. C. KANELLAKIS, On concurrency control by multiple versions, in "Proceedings 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems," pp. 76-82, Association for Computing Machinery, New York, 1982.
25. R. SETHI, A model of concurrent database transactions, in "Proceedings 22nd Ann. Symp. on Foundations of Computer Science," pp. 175-184, IEEE Computer Society, Los Angeles, 1981.
26. R. SETHI, Useless actions make a difference: Strict serializability of database updates, *J. Assoc. Comput. Mach.* **29** (1982), 394-403.
27. A. SILBERSCHATZ AND Z. KEDEM, Consistency in hierarchical database systems, *J. Assoc. Comput. Mach.* **27** (1980), 72-80.
28. E. SOISALON-SOININEN AND D. WOOD, An optimal algorithm for testing for safety and detecting deadlocks in locked transaction systems, in "Proceedings 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems," pp. 108-116, Association for Computing Machinery, New York, 1982.
29. R. E. STEARNS, P. M. LEWIS, AND D. J. ROSENKRANTZ, Concurrency control for database systems, in "Proceedings 17th Ann. Symp. on Foundations of Computer Science," pp. 19-32, IEEE Computer Society, Los Angeles, 1976.
30. R. E. STEARNS AND D. J. ROSENKRANTZ, Distributed database concurrency controls using before values, in "Proceedings ACM-SIGMOD 1981 Int. Conf. on Management of Data," pp. 74-83, Association for Computing Machinery, New York, 1981.
31. Y. SUGIYAMA, T. ARAKI, J. OKUI, AND T. KASAMI, Complexity of the deadlock avoidance problem, *Trans. IECE Japan* **60-D** (1977), 251-258 (in Japanese).
32. M. YANNAKAKIS, A theory of safe locking policies in database systems, *J. Assoc. Comput. Mach.* **29** (1982), 718-740.
33. M. YANNAKAKIS, Freedom from deadlock of safe locking policies, *SIAM J. Comput.* **11** (1982), 391-408.
34. M. YANNAKAKIS, Serializability by locking, manuscript (1982).
35. M. YANNAKAKIS, C. H. PAPADIMITRIOU, AND H. T. KUNG, Locking policies: Safety and freedom from deadlock, in "Proceedings 20th Ann. Symp. on Foundations of Computer Science," pp. 286-297, IEEE Computer Society, Los Angeles, 1979.