Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

CONCURRENT SIMULATION: AN ALTERNATIVE TO DISTRIBUTED SIMULATION

Douglas W. Jones
Department of Computer Science
University of Iowa
Iowa City, IA 52242, U.S.A.

ABSTRACT

The advent of of a new generation of multiprocessors allows new approaches to parallel simulation. Previous work in this area has concentrated on distributed simulation; this approach uses spatial decomposition to allow simulations to be run on networks of machines, where the message flow between processors in the network is related closely to the topology of the system being simulated. This paper presents an alternate approach, concurrent simulation, which is based on temporal decomposition. This allows natural use to be made of the shared memory facilities and load-balancing capabilities of the new multiprocessors, and it overcomes some fundamental limitations of the distributed approach.

1. BACKGROUND

Discrete-event simulation is important to fields as diverse as operations research and VLSI design. In all of these fields, the demand for time-consuming and hence costly simulation studies has lead to a growing interest in the use of parallel computer systems in simulation. The dominant approach to parallel simulation, usually called distributed simulation, involves the partitioning of the simulation model into segments that are then run on separate machines connected in a network with a topology compatible with the logical interconnection structure of the simulated system (Chandy and Misra, 1979) (Peacock, Wong and Manning, 1979b).

The problem with this approach is that many simulation problems are not easily partitioned in this manner. Many distributed simulation experiments have relied on manual decomposition of the simulation model (Peacock, Wong, and Manning, 1979a), although automatic decomposition may be possible. Furthermore, in the limited domain of queueing networks, some otherwise well-behaved simulation models are prone to deadlock when run in a distributed environment (Peacock, Wong and Manning, 1979b), and others gain very little from the distributed approach (Peacock, Wong, and Manning, 1979a). This provides a clear motivation for investigating alternate approaches to the application of multiple processors to discrete-event simulation.

A new generation of symmetric multiprocessor computer systems has recently reached the marketplace (Baskett and Hennessy, 1986) (Bell, 1985). These machines are characterized by between 2 and 20 processors sharing a single main memory. Most of these machines are currently based on monolithic processor implementations and a single shared memory bus, where cache memories are used to reduce bus contention. Systems with more processors or more complex memory arbitration mechanisms are on the horizon, as illustrated by the Illinois Cedar machine (Kuck et al, 1986). Because of their use of shared memory, all of these machines have the potential for concurrency based on problem decompositions other than the strictly spatial ones commonly used in distributed simulations.

The balance of this paper begins with a brief discussion of the operation of a typical discrete-event simulation system running on a single processor. Section 3 continues with a general framework for discussing parallelism in discrete-event models. Section 4 discusses distributed simulation in terms of this framework. Section 5 introduces concurrent simulation, and section 6 continues this subject with an examination of pending event set support for concurrent simulation. The paper ends with a preliminary evaluation of the potential performance of concurrent simulation and a discussion of how it might be combined with distributed simulation.

2. SEQUENTIAL DISCRETE-EVENT SIMULATION

The basic discrete-event simulation algorithm operates by repeatedly extracting and simulating the next event from the set of all events involved in the simulation. Thus, events are simulated in strictly chronological order (as long as simultaneous events are ignored). The simulation of one event may involve changes to state variables of the simulation model and the scheduling of other events that are to be simulated in the future. State variables can represent such things as the voltage on a particular wire or the number of items in some queue. At any instant during the simulation, the pending event set is the subset of all possible events that consists of those events that have not yet been simulated but have already been scheduled. The following invariant constrains both the scheduling of new events and the selection of the next event to be simulated:

I1: The simulated time of the event currently being simulated is no later than the simulated time of any event in the pending event set.

Without loss of generality, the code for simulating an event can be restricted to reading the state variables on which the event depends and making any state variable modifications before any new events are scheduled. This restriction divides the simulation of each event into two phases, a _state-variable phase_, where the state variables are inspected and possibly updated, and an _event-scheduling phase_, where new events are scheduled. Although this restriction has no effect on the sequential discrete-event simulation algorithm, it will be useful in the parallel versions to be discussed next.

Note that, in terms of the world views compared by Overstreet (1982) and by Overstreet and Nance (1986), this presentation is clearly based on event scheduling and not process interaction or activity scanning. Nonetheless, the division of the processing of an event into phases resembles the division used in the three-phase formulation of the activity scanning approach (O'Keefe, 1986). In terms of the three-phase model, the events being discussed here are all bound or scheduled events; any computations that would occur as a consequence of conditional or contingent events in a three-phase model are assumed to be folded into the bound events that caused the state changes on which they depended.

3. PARALLELISM IN DISCRETE EVENT SIMULATION

Parallel approaches to discrete-event simulation ultimately rely on the identification of independent events. An event may depend on another because of a _scheduling dependency_ or because of a _state-variable dependency_. Event A has a direct scheduling dependency on event B if event A is scheduled by the simulation of event B. The initial events in a simulation model have no scheduling dependencies, while all other events have a direct scheduling dependency on exactly one other event. Event A has a direct state-variable dependency on event B if the results of simulating event A depend on the value of a state variable that is stored by the simulation of event B. The number of other events on which an event has state-variable dependencies depends on the number of state variables inspected during the simulation of that event. An event depends indirectly on another if it depends in any way on some event that depends directly or indirectly on the other. A pair of events are independent if neither depends on the other.

The dependency relation establishes a partial ordering of the set of all events in a simulation run. If this is a total ordering, there are no independent events and opportunities for parallelism are limited. A parallel simulation system can begin the simulation of any event as soon as all events on which it depends have been simulated. The problem of designing a parallel simulation system, then, is to determine when this is true. Figure 1 illustrates the dependency relations between the events in a simulation model and the subsets of the set of events that can be identified at some point during the simulation.

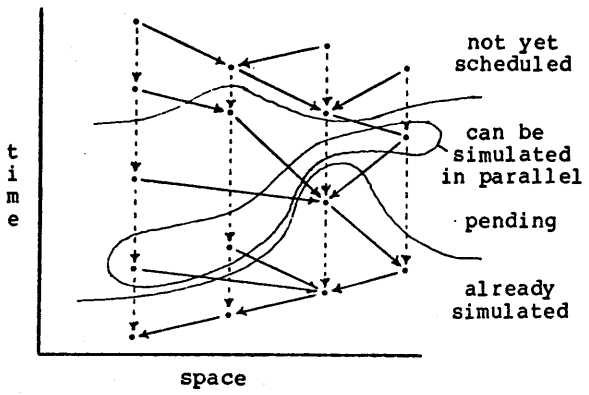

Figure 1. Dependency relations and subsets in the set of events. Event dependencies are shown by arrows, solid for scheduling dependencies, dashed for state-variable dependencies.

It should be noted that simulation models can be modified to change the dependency relationships between events. For example, if event A schedules event B, and event B always schedules event C, where the scheduling of C does not depend on the values of any state variables, the model can be changed so that A schedules both B and C. The former formulation will minimize the size of the pending event set, while the latter may allow greater concurrency.

4. DISTRIBUTED SIMULATION

In a distributed simulation, the set of state variables is partitioned in such a way that, for each event, the state variables needed to simulate that event are in a single block resulting from the partitioning. Each block is then associated with a process that simulates the events that manipulate the state variables in that block. Interprocess communication is required whenever an event associated with one block is scheduled by an event associated with a different block. Within each process, events are simulated strictly in the order of their simulated times; additional interprocess communication is required to prevent the simulation of an event until it can be guaranteed that no events dealing with the same block will be scheduled at prior simulated times.

Although most real simulation models can be partitioned so as to allow distributed simulation, some models cannot and others must be modified to permit this. The primary tool used to modify simulation models to allow partitioning is event splitting. When this is done, one of the events resulting from the split schedules the other to happen at the same simulated time and transmits to it the values of any state variables needed.

As described in Peacock, Wong, and Manning (1979a), events that schedule others at the same simulated time introduce the possibility of deadlock into distributed simulations.

It is natural to hope that distributing a simulation model over m machines would lead to a speedup proportional to m; this is not always the case. One reason for this is that operations on an n element pending event set take O(log n) time in the worst case using the fastest known implementations (Jones, 1986). If the event set is partitioned over m machines, the time is reduced to O(log n/m) per operation. As a result, the speedup s for operations on the pending event set will be (log n)/(log n/m). Solving for m as a function of s and n (the number of machines required to provide a particular speedup when the size of the pending event set is known) yields

$$m = n^{(1 - 1/s)}.$$

For s = 2, this implies

$$m = n^{1/2}.$$

This can severely limit the speedup for systems where the expected number of pending events is large.

This estimate of the speedup resulting from distributing a simulation is pessimistic, in that it deals only with the gain to be obtained by distributing the storage and processing of the pending event set. Although some simulation problems have large pending event sets, many have only small ones and thus may be quite appropriate for distributed simulation. It is also important to note that the speedup expected for distributed simulation when the pending event set is large can be made to look quite good by using a linear list implementation. In this case, the expected time to perform an operation when there are n elements in the pending event set is O(n), so partitioning over m machines reduces the time to O(n/m) giving a speedup of m. While this is an impressive speedup, it is misleading since the run-times will be slower than they would be with a better choice of pending event set implementation. Unfortunately, few if any of the papers on distributed simulation provide clear descriptions of the pending event set sizes or implementations used to obtain the speedups reported. In fairness, it should be noted that for the small queueing networks such as are reported on by Peacock, Wong, and Manning (1979a), the pending event set should be quite small, and thus the choice of event-set implementation should not make much difference in the results reported.

5. CONCURRENT SIMULATION

An alternate approach to the problem of applying parallel processing to discrete-event simulation relies on the observation that events are not simulated instantaneously. As noted above, the process of simulating an event can be divided into two phases, the state-variable phase and the event-scheduling phase. As soon as the state-variable phases of all events on which a pending event has state-variable dependencies have been completed, that event may be simulated. This allows some parallelism even when the event dependency relationship imposes a total ordering on the set of all events. The synchronization constraints that govern this approach to parallel simulation can be summarized as follows:

C1: An event may not be considered for removal from the pending event set until it can be guaranteed that all events on which it has state-variable dependencies have already been scheduled.

C2: The state-variable phases of events that have been removed from the pending event set to be simulated must be simulated in chronological order by simulated time for those events that have state-variable dependencies.

The term <u>concurrent simulation</u> is used here to describe the approaches to exploiting opportunities for parallelism that are not exploited by distributed simulation. Since the goal of this work is to identify an alternative to distributed simulation, no assumptions are made about the possibility of partitioning the state variables of the simulation model. Thus, it is assumed that all events may inspect or modify any of the state variables, so that the set of events is totally ordered by state-variable dependencies.

These constraints lead naturally to a pipelined model of concurrent simulation where a single process simulates the state-variable phases of all events in strict chronological order according to their simulated times. A second process then handles the event-scheduling phases of each event after the first process has finished with it. This distributes the work between the processes based on a temporal decomposition instead of a spatial decomposition such as is used in distributed simulation. This pipelined view of concurrent simulation is illustrated in Figure 2. The specialized VLSI simulation machine built at HP Labs appears to be a direct hardware implementation of this approach (Birnbaum, 1985).

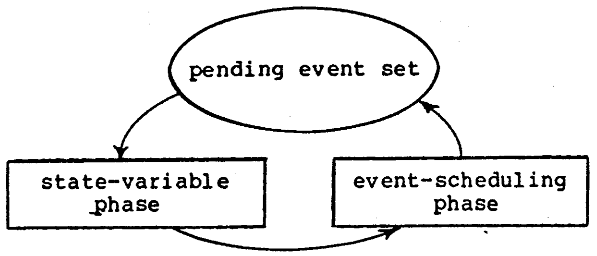

Figure 2. A Pipelined View of Concurrent Simulation

An alternate view of concurrent simulation is based on a symmetrical decomposition of the work to be done. In this case, all of the processes execute identical code, and the number of processes used may be arbitrarily large (although it must be noted that adding additional processes will not necessarily lead to any performance gain). This view is especially appealing for symmetrical shared memory multiprocessors because full advantage can then be taken of the automatic load balancing characteristics of such machines. The main body of each process in this model is outlined below:

```
loop
    wait( statesem );
    e := nextevent;
    statephase( e );
    signal( statesem );
    eventphase( e );
forever;
```

In the above code, the wait and signal operations on the semaphore statesem ensure mutual exclusion, and therefore serialization, of the state-variable phases of all events (Dijkstra, 1971). The procedures statephase and eventphase actually perform the simulation of the two phases of each event, and the function nextevent returns the next event from the pending event set.

The above code _does not_ correctly capture all of the details of the interprocess synchronization needed to ensure correct simulation results. Specifically, this code does nothing to prevent the event-scheduling phase of one event from scheduling an event prior to some other event that is already being simulated. As a result, condition C1 can be violated in those cases where the event already being simulated has state-variable dependencies on the new event, which leads to a violation of C2. The following conservative invariant ensures that this does not happen:

I2: No event in the pending event set has a simulated time before the simulated times of any events that have been or are currently being simulated.

This new invariant is a simple generalization of I1, but it is much harder to enforce. The problem is that it must be enforced by preventing events from being removed from the pending event set, not by preventing the scheduling of events. One way to ensure that I2 is satisfied is to have the event simulation phase of each event currently being simulated maintain a record of the _limit time_ of that event. The limit time of each event is initially the same as the simulated time of that event, but it may be advanced as long as the following local invariant is met:

I3: When an event is scheduled, the simulated time of that event does not come before the limit time of the event that scheduled it.

At any instant during a simulation, it is possible to determine the _minimum limit time_, the minimum of the set of limit times of the events currently being simulated. It is easy to see that the minimum limit time is the time before which no new events will be scheduled by events currently being simulated. Thus, if the head event in the pending event set is scheduled at a time prior to the minimum limit time, it can be guaranteed that no as yet unscheduled events will be scheduled before the head event. The following code outlines how this can be accomplished:

```
loop
    limit := infinity;
    wait( statesem );
    while nexteventtime > minlimit
      do nothing;
    e := nextevent;
    limit := e.time;
    statephase( e );
    signal( statesem );
    eventphase( e, limit );
forever;
```

In the above code, the function nexteventtime returns the time of the next event in the pending event set, and the function minlimit computes the minimum of the limit times of all active simulation processes. The above code will not introduce any concurrency unless the eventphase procedure advances the limit time.

The problem of assuring that the limit times of all but the most trivial events are advanced by the event phases of those events can be surprisingly easy to solve. All that is needed is a guarantee that all new events scheduled by an event are computed (and scheduled) in order of increasing simulated time. Assuming that this is done, the limit time can always be set to the simulated time of the most recently scheduled event, as in the following code:

```
procedure eventphase( e, limit );
    while eventsremain( e ) do
        n := nextremaining( e );
        schedule( n );
        limit := n.time;
```

The eventsremain function in the above code determines whether any events remain to be scheduled; the nextremaining function returns events from among those that remain in the order of their simulated times. Clearly, the feasibility of using this approach depends strongly on the ease with which events which remain to be scheduled can be computed in this order. As an example illustrating the ease with which this may be done in some applications, consider the problem of building a logic simulator.

In a logic simulator, events that cause the output of a gate to change result in the scheduling of input change events at the inputs of each gate connected to that output. The delay after which each of these new events is scheduled is usually determined by the wire length and type of wire used. The number of events caused by an output change event depends on how many inputs are driven by that output. For most gate types, a change to an input causes at most one output to change, and frequently, nothing changes. As a result, the event-scheduling phases of events, other than output change events, are fairly simple. When an output changes, however, the simulator must scan the list of inputs connected to that output and schedule a new event for each of those inputs. If this list is sorted in order of increasing delay, these new events will be naturally produced in the correct order; furthermore, this sorting can be done at the time the data structure representing the circuit to be simulated is built, before the actual simulation begins.

The above code suggests the use of a rather complex computation each time the minlimit function is computed. Since this is evaluated inside a polling loop, this could lead to unnecessary delays. One simple alternative to this would be to recompute minlimit each time the limit time of any process changes. Although this might help, it is unnecessary because the priority-queue that is used to order the pending event set can itself be used to perform this computation! This is true because during the event scheduling phase of the simulation of an event, the limit time is always the same as the simulated time of the most recently scheduled event; this event can itself be used as a record of the limit time. As a result, the simulation of an event may begin only when it is at the head of the pending event set and is not being used as a record of the limit time of any other event. The following code illustrates this:

```
loop
    wait( statesem );
    e := nextevent;
    statephase( e );
    eventphase( e );
forever;

procedure eventphase( e );
    if eventsremain( e ) then
        n := nextremaining( e );
        schedule( n );
        signal( statesem );
        while eventsremain( e ) do
            old := n;
            n := nextremaining( e );
            schedule( n );
            release( old );
        end;
        release( n );
    else
        signal( statesem );
```

In the above code, the nextevent function does not return the head event record from the pending event set until that event has been scheduled and released. Scheduling an event places that event in the pending event set as a record of the limit time of the event that scheduled it, while releasing an event indicates that that event no longer records a limit time and may be simulated if it is the head event in the pending event set. The extension of the basic event set abstract data type by the addition of the release operation was perhaps the hardest single step in the development of this approach to concurrent simulation.

An important change was made in modifying the previous code to produce the above: The end of the critical section established by the semaphore statesem was moved to include not only the state-variable phase of the event but the first call to schedule in the event-scheduling phase. This does not change the extent of parallelism from the previous version since the initial setting of the limit time of the event also prevented the simulation of any other events until that event had a chance to schedule the next event.

It may be possible to introduce additional parallelism into this approach to concurrent simulation by adding semaphores to specific state variables. If this is done, and events claim exclusive use of the state variables that they inspect or modify, the global semaphore statephase may be released as soon as the appropriate state variables have been claimed and it is determined that an event will schedule no new events. Since this modification introduces some overhead into the processing of all events, but only allows a limited increase in concurrency, it may not actually lead to any speedup.

6. PENDING EVENT SET SUPPORT

The model of the pending event set used by the sequential discrete-event simulation algorithm has only two main entry points, nextevent and schedule. These correspond, respectively, to the dequeue and enqueue operations on a priority-queue, sometimes called delete-min and insert (Jones, 1986). The concurrent model of discrete-event simulation presented above requires an additional operation, release. Although the problems of implementing priority queues are relatively well understood, this new operation will clearly complicate any implementation of the pending event set.

Concurrent simulation algorithms, whether symmetrical or pipelined, will all involve multiple processes contending for access to the pending event set. Thus, it is important to find a pending event set implementation that supports concurrent operations. It is well known that a FIFO queue between a single producer and a single consumer can be implemented without critical sections (Dijkstra, 1971). Unfortunately, all known concurrent priority queue implementations require critical sections.

A concurrent implementation of implicit heaps is described in Quinn and Deo (1984). In this implementation, the heap is represented conventionally in a shared array. One processor is dedicated to each level in the heap, so it requires O(log n) processors for an n item heap. Although the presentation in Quinn and Deo (1984) is based on a parallel bottom-up heapify operation using O(n/4) processors, it is clear O(log n) processors would be sufficient for top-down insertions in the heap. Thus, both scheduling of new events and removing the next event from the pending event set should be possible in constant time.

More recently, a concurrent version of skew heaps has been developed (Jones, 1985). In this implementation, each operation on the pending event set is accomplished by a single process in O(log n) time. The key to the efficiency of this scheme is that each operation on the pending event set blocks other operations for only a short, constant interval, thus allowing event-set operations to be performed at a constant rate as long as O(log n) processors are available to accomplish them. The advantage of this scheme is that no processors must be dedicated to pending event set management except when there are operations on the pending event set in progress.

The concurrent implementation of skew heaps, as it currently stands, does not include any facilities to support a release operation, and neither of the concurrent implementations mentioned here has been modified to support the arbitrary deletion of previously scheduled events. Both of these operations will be required if concurrent simulation is to become a widely used approach to discrete-event simulation.

It should also be noted that the computations needed to manage the event-set are almost evenly divided between the enqueue and dequeue operations when concurrent skew heaps are used. Among the sequential priority queue algorithms discussed by Jones (1986), some, such as pairing heaps, perform most of the computations in the dequeue operation, while others, such a sorted linear list, perform more of the computations in the enqueue operation. It would be highly desirable to find a fast concurrent priority queue algorithm which performs most of its computations in the enqueue operation because the concurrent simulation approach described here allows the enqueues to be done in parallel while requiring the dequeues to be done serially.

7. CONCLUSION

Although neither a formal analysis nor any empirical results have yet been obtained for the concurrent simulation model presented here, it may be able to provide almost linear speedup up to O(log n) processors when there are n pending events, especially if there is a wide variance in the number of events scheduled as a consequence of each event. If each event causes exactly one new event to be scheduled, it is clear that this approach has little hope of providing any speedup. Above O(log n) processors, the pending event set may well become a bottleneck so additional speedup may be difficult to attain.

Since the concurrent approach to parallel simulation attacks the problem from a different direction than that followed in distributed simulation, it is interesting to ask whether the two approaches might not be combined. There is some hope of this, since the key concept of the limit time of an event has parallels in some approaches to distributed simulation. For example, in Peacock, Wong and Manning (1979b), a _link time_ is maintained with each communications path between the blocks resulting from the partitioning of the model. The link time for each path is the simulated time before which no new events will be scheduled over that path. The basic rule used to ensure correct handling of time is that no event may be simulated until the simulated time of that event is less than the simulated times of all other pending events in the same block and less than the link times of all paths leading into that block. The link times of all paths leading out of each block are set equal to the time of the event currently being simulated in that block.

The expected O(log n) limit on the potential speedup of concurrent simulation is surprisingly well matched to the number of processors that is typical of the current generation of commercially available symmetric, shared-memory multiprocessors (for example, 12 on Sequent's current offering, 20 on Encore's current offering). Although there is reason to believe that larger symmetric multiprocessors may be hard to build, larger systems using message passing or relatively slower shared memory appear to be quite practical. If such large systems are constructed from symmetric multiprocessor clusters, as is the case with the Illinois Cedar machine (Kuck et al, 1986), an appealing approach to using these systems for discrete-event simulation would be to use a distributed approach to split the model over multiple clusters, and use the concurrent approach within each cluster.

REFERENCES

Baskett, F. and J. Hennessy J. (1986). Small Shared-Memory Multiprocessors. _Science 231_, 963-967.

Bell, C. G. (1985). Multis: A New Class of Multiprocessor Computers. _Science 228_, 462-467.

Birnbaum, J. S. (1985). Towards the Domestication of Microelectronics. _Communications of the ACM 28_, 1225-1235.

Chandy, K. M., and Misra, J. (1979). Distributed Simulation: A Case Study. _IEEE Trans. on Software Engineering SE-5_, 440-452.

Dijkstra, E. W. (1971). Hierarchical Ordering of Sequential Processes. _Acta Informatica 1_, 115-138.

Jones, D. W. (1985). Concurrent Operations on Priority Queues. submitted for publication.

Jones, D. W. (1986). An Empirical Comparison of Priority-Queue and Event-Set Implementations. _Communications of the ACM 29_, 300-311.

Kuck, D. J., et al. (1986). Parallel Supercomputing Today and the Cedar Approach. _Science 231_, 967-974.

O'Keefe, R. M. (1986). The Three-Phase Approach. In: Proceedings of the 1986 Winter Simulation Conference.

Overstreet, C. M. (1982). Model Specification and Analysis for Discrete Event Simulation. Unpublished Ph.D. Thesis, Computer Science Department, Virginia Polytechnic Institute, Blacksburg, Virginia.

Overstreet, C. M. (1986). World View Based Discrete Event Model Simplification. In: Modelling and Simulation Methodology in the Artificial Intelligence Era. North Holland, Amsterdam.

Peacock, J. K., Wong, J. W., and Manning, E. G. (1979a). A Distributed Approach to Queueing Network Simulation. In: Proceedings of the 1979 Winter Simulation Conference. Institute of Electrical and Electronics Engineers, 399-406.

Peacock, J. K., Wong, J. W. and Manning, E. G. (1979b). Distributed Simulation Using a Network of Processors. Computer Networks 3, 44-56.

Quinn, M. J. and Deo, N. (1984). Parallel Graph Algorithms. ACM Computing Surveys 16, 319-348.

AUTHOR'S BIOGRAPHY

DOUGLAS W. JONES is an assistant professor in the Department of Computer Science at the University of Iowa. He received a B.S. in physics from Carnegie-Mellon University in 1973, and M.S. and Ph.D. degrees in computer science from the University of Illinois in 1976 and 1980 respectively. He has developed a gate level logic simulator that has been used for teaching digital systems since 1983; this lead to an investigation of event-set implementations and parallel simulation. His other research interests include system programming languages and computer architecture. He is a member of ACM and AAAS.

Douglas W. Jones
Department of Computer Science
University of Iowa
Iowa City, IA 52242, U.S.A.
(319) 353-7479
CSNET: jones@cs.uiowa.edu