Logical Time in Visualizations Produced by Parallel Programs

Janice E. Cuny,[1] Alfred A. Hough, and Joydip Kundu

Department of Computer Science
University of Massachusetts, Amherst MA 01003

Abstract

Visualization tools that display data as it is manipulated by a parallel, MIMD computation must contend with the effects of asynchronous execution. We have developed techniques that manipulate logical time in order to produce coherent animations of parallel program behavior despite the presence of asynchrony. Our techniques "interpret" program behavior in light of user-defined abstractions and generate animations based on a logical rather than a physical view of time. If this interpretation succeeds, the resulting animation is easily understood; if it fails, the programmer can be assured that the failure was not an artifact of the visualization. Here we demonstrate that these techniques can be generally applied to enhance visualizations of a variety of types of data as it is produced by parallel, MIMD computations.

1 Introduction

Visualization tools aid in the understanding of massively parallel, MIMD computations. They are useful both at the application level where data visualization allows the user to better understand simulated, real world phenomena and at the programming level where program animation enables the user to better understand the behavior of his/her code. At either level, however, visualization tools running on MIMD architectures must contend with the effects of asynchronous execution.

Processes in an asynchronous computer system execute without the benefit of a global clock; instead they have their own local clocks which are not mutually synchronized. As a result, it is not always possible to determine the order of events executed by different processes. *The temporal ordering of events in a parallel, MIMD computer system is a partial order, not a linear order.* Animation systems, however, are constrained to sequential, frame-by-frame depictions and therefore they must impose a linear ordering on events. But what if this ordering does not match the programmers expectations? Is the resulting anomaly an artifact of the visualization tool or is it a misunderstanding of the real world phenomena or is it the symptom of a program bug?

To date, our work has focused on program animation tools designed specifically for use in debugging for correctness.[2] We have developed techniques that manipulate logical time in order to produce coherent animations of parallel program behavior despite the presence of asynchrony. Our techniques "interpret" program behavior in light of user-defined abstractions and generate animations based on a logical rather than a physical view of time. If this interpretation succeeds, the resulting animation is easily understood; if it fails, the programmer can be assured that the

[1]The Parallel Programming Environments Project at the University of Massachusetts is supported by the National Science Foundation under grant CCR-9023256.

[2]This is in contrast to a number of program visualization tools designed for use in debugging for performance [2, 5, 15]. In debugging for correctness, it is logical time that is important; in debugging for performance, it is physical time.

failure was not an artifact of the visualization. Here we demonstrate that these techniques can be more generally applied to enhance the visualization of data as it is manipulated by parallel, MIMD computations.

We begin, in Sections 2 and 3, by defining our techniques in their original context, that is, within systems for program animation. In Section 4, we demonstrate their utility in the more general context of data visualization. In Section 5, we present our conclusions.

2 Our Approach: An Overview in the Context of Parallel Program Animation

Massively parallel computer systems — in which hundreds or even thousands of interacting processes execute concurrently — are enormously complex. In order to understand their behavior, programmers rely on informal modeling techniques: information traced during execution is filtered and abstracted to develop a model of the system's actual behavior that is compared with the programmer's conceptual model of the system's intended behavior. Visualization tools aid in this process by providing comprehensible views of program behavior.

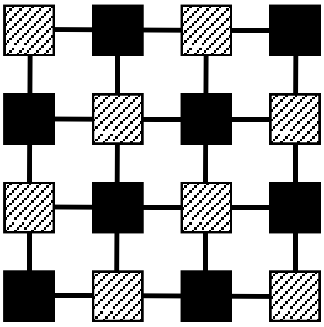


Figure 1: Checkered pattern of striped and solid processes for the SOR program.

Consider, for example, a program iteratively approximating the solution to a PDE using a Successive Overrelaxation (SOR) method [6]. Processes, arranged in a square mesh repeatedly update their values as a function of the values of their neighbors. To speed convergence, their execution alternates in a checkered pattern as in Figure 1: first striped processes execute and then solid processes.

Figure 2 shows snapshots from a straightforward animation of an SOR program but it does not show the expected behavior. What has gone wrong?

The most obvious problem is that the programmer did not think of the individual behavior of processes but instead thought of a global pattern of activity in which all processes were "doing the same thing at the same time." Here we assume that the visualization tool supports some type of abstraction, allowing the user to group program actions into *abstract events*. In this example, we group *read* and *update* actions into *striped* and *solid* events which we animate with

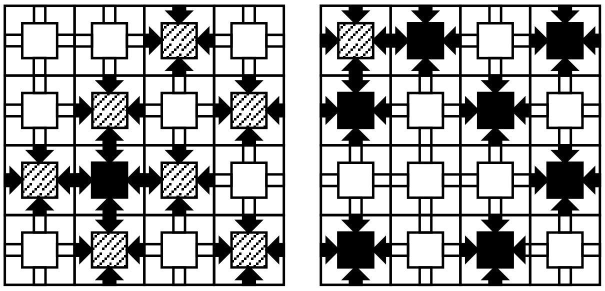
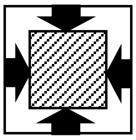
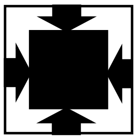


Figure 2: Snapshots from the original animation of the SOR program.



and



Abstraction alone, however, is not enough.

The abstract *striped* and *solid* events are not atomic but occur over a period of time (starting with the first *read* and ending with the last *update*). Because of asynchrony, the time periods for successive events can overlap; thus, the logically sequential *striped* and *solid* events may appear as concurrent. Worse, since they are executed on the same set of processes, their animations may be superimposed making them incomprehensible. (This phenomena occurs in the Dictionary Search example below; it is seen in Figure 4*b*.)

Our visualization techniques produce comprehensible pictures by ordering events in logical rather than physical time. Events are ordered in logical time by

process dependencies: processes are sequential and their actions are totally ordered by local timestamps, and

interprocess dependencies: messages must be sent before they can be received.[3]

For the SOR program, *striped* and *solid* steps are ordered by process dependencies and thus we animate them in successive frames as shown in Figure 3. In the figure, the abstract events of interest have been temporally separated, providing visual discrimination and allowing the user to understand his program's behavior in terms of his/her own conceptualizations.
The behavior in this new animation is logically equivalent to the behavior in the original picture; that is, all processors execute the same sequences of operations with the same interprocess dependencies. For many parallel computations, however, such simple orderings are not possible because of intertwined dependencies.

Consider, for example, a program that implements a dictionary search in which queries are pipelined from the host to a database of key-ordered records stored in a hypercube [14]. Queries are routed within the cube to the proper node using a binary search. In the animation of primitive

[3]Here, we consider only nonshared memory paradigms but the definitions can be extended to shared memory as discussed in Section 4.

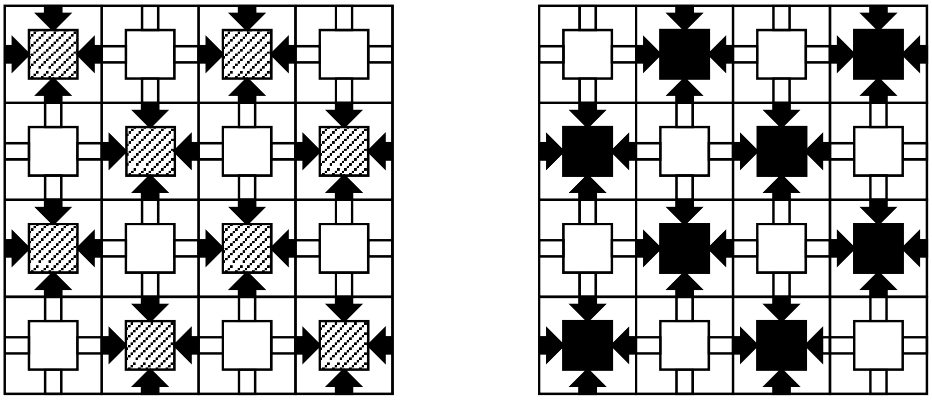


Figure 3: Intended, logical sequence of *striped* and *solid* steps in the SOR computation.

events shown in Figure $4a$, multiple queries are active simultaneously and it is difficult to understand whether or not each query is proceeding as intended.

To view this behavior in a more understandable format, we define an abstract event to be all of the communications generated on behalf of a single query. These new query events cannot be separated on the basis of logical time because they follow data-dependent paths through the cube, arriving in different orders at different processes. There is no single, logically consistent ordering that can be imposed by the animator.

For such cases, we introduced *perspective views* which enable the user to selectively ignore logical dependencies in establishing partially consistent event orderings. In this case, we can choose to ignore all dependencies other than those caused by *send* events on the host process. This creates the perspective views shown in Figure $4c-d$ in which queries are shown in the order that they were issued from the host and each query completes before the next begins.

This animation is clear and understandable. In spite of the fact that it is not logically equivalent to the actual execution trace (queries appear to arrive at some processes out of order), it enables the programmer to easily comprehend relevant aspects of program behavior. In this case, it enabled us to discover a bug that was not apparent in the original animation: in Figure $4d$, a query crosses a dimension of the cube twice indicating an error in the routing of messages. Thus, *ignoring some temporal orderings in the original execution sequence made it possible for the programmer to understand aspects of the program's behavior relevant to its correctness.*

3 Our Approach: The Technical Details

We base our animations on Lamport's *happened before* relation [12] which is defined on primitive actions, each assumed to have a processor-local timestamp, denoted $timestamp(a)$ for an event a. Positioning events on their process time-line according to their local timestamps (increasing from left to right), event a *happened before* event b, denoted $a \rightarrow b$, iff one of the following conditions holds:

i) events a and b happen on the same process and

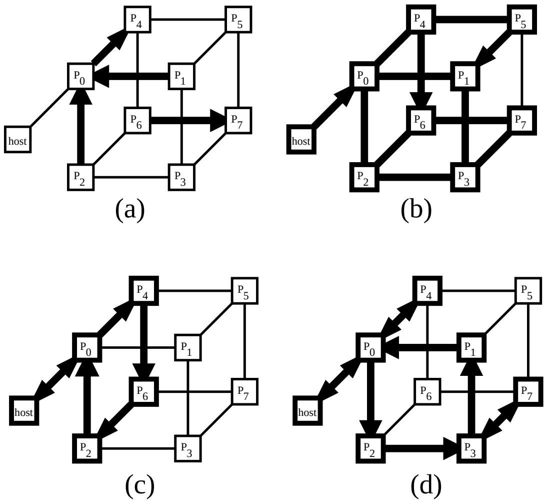
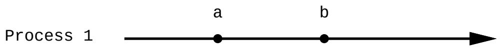
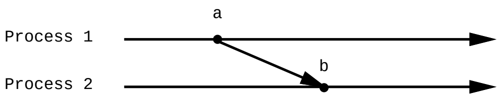


Figure 4: Snapshots of an animation of the Dictionary Search. Low-level communication events (a); abstract events with several concurrent search requests (b); a perspective view of abstract events showing the path taken by an individual request (c); erroneous communication between p3 and p7 (d).

$$\text{timestamp}(a) < \text{timestamp}(b)$$

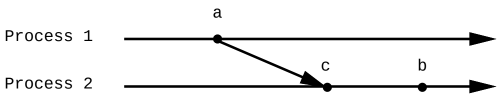


ii) a is the act of sending a message and b is the act of receiving it (denoted by an arrow from a to b)



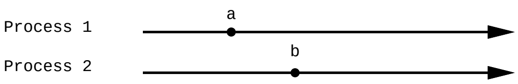
and

iii) there is some event c such that $a \rightarrow c$ and $c \rightarrow b$



Two events are unordered if they are not related by *happened before*:[4]

$$a \parallel b \text{ iff not } (a \rightarrow b) \text{ and not } (b \rightarrow a)$$



[4]This notation is slightly different than Lamport's.

$\rightarrow$ is defined on primitive events. For our purposes, we extend the definition to nonatomic events, defining three relations — *precedes*, *parallels*, and *overlaps* — between abstract events. Other extensions have been proposed [12, 1, 11, 3] but ours has been tailored to the needs of visualization systems. We begin with a relation *partially precedes*, denoted $\mapsto$. Let A and B be sets of events. Informally, $A \mapsto B$ if some part of A *happens before* some part of B or they share a primitive event; formally

i) $A \mapsto B$ if $\exists a \in A, b \in B : a \rightarrow b$ or $a = b$

and

ii) $\mapsto$ is closed under transitivity

Using *partially precedes* we define the three possible relations between two abstract events:

Precedes: $A \Rightarrow B$ iff $A \mapsto B$ and NOT $(B \mapsto A)$

Parallels: $A \parallel B$ iff NOT $(A \mapsto B)$ and NOT $(B \mapsto A)$, (alternatively, iff $\forall a \in A, b \in B : a \parallel b$)

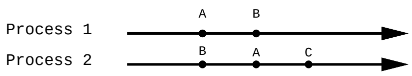
Overlaps: $A \Leftrightarrow B$ iff $A \mapsto B$ and $B \mapsto A$

Precedes captures the notion that one abstract event logically occurs before another. *Parallels* captures the notion that no logical ordering exists between the events and *overlaps* captures the notion that the events are logically intertwined (that is, some part of each event must occur before some part of the other).

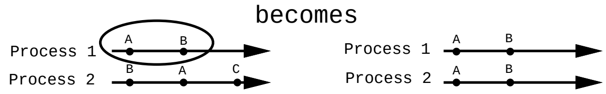
Our techniques associate with each relation a transformation that assigns animation times. In the case of *precedes*, we separate the events by assigning animation times so that all elements of the first event complete before any element of the second event begins. In the case of *parallels*, we assign the the same animation time to the earliest events in each set. The transformation for *precedes*, for example, separates the *striped* and *solid* steps of the SOR program in Figure 3, while the transformation for *parallels* synchronizes the subevents of each step.

For events related by *precedes* or *parallels*, the transformations produce *consistent orderings*; that is, all processes execute the same sequence of events and the *happened before* relation remains unchanged. For events related by *overlaps*, however, we cannot construct consistent reorderings and, thus, we introduce *partially consistent orderings* in which each process executes a subsequence of its original event sequence and the *happened before* relation is a subset of the original *happened before* relation.

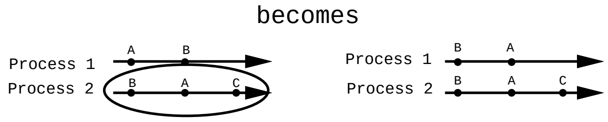
We base our partially consistent orderings on a user-selected subset of the events called a *perspective*. Only events named in the perspective are used in computing ordering relations. Events not named in the perspective and not needed for the display of named events are deleted and the remaining events are reordered in a manner consistent with the computed relations. We might, for example, consider the following behavior where all constituent events of an abstract event are labeled with the same uppercase letter



from the partial perspective of Process 1. We would use only the Process 1 primitive events for computing ordering relations and, thus,



where circles indicate the events used to compute the ordering. The ordering on Process 1 has been applied to all events, including those on Process 2; abstract event C has been deleted since it does not contain elements in the perspective. Alternatively, the same system can be decomposed from the perspective of Process 2, in which case



Each reordering is meaningful; each exposes the order of events on an individual process and provides some insight to the code and environment of that process. The two reorderings together characterize the complete behavior of the system.

Further details of the transformation algorithms can be found in [9] and [10].

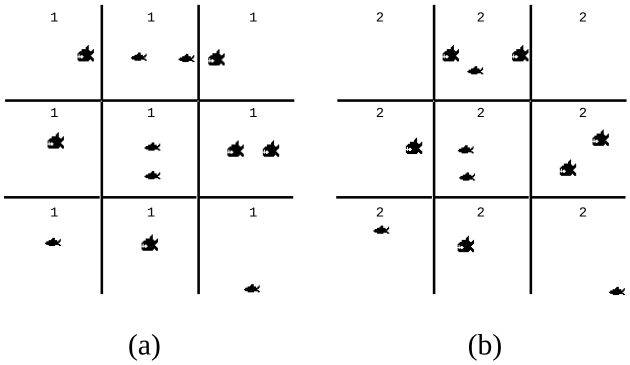
4 Examples: Logical Time in Visualizations Produced by Parallel Computations

We have successfully used the logical time manipulations described here in the Belvedere animation system [7, 8, 10] which was specifically designed to aid in the debugging of parallel programs for correctness. Here, we demonstrate their more general application to visualizations of domain-specific data.

Example 1. Sharks and Fishes [16]. For this example, we use the animation of a program that performs an underwater simulation taken from a paper on the Voyeur animation system [16]. The simulation consists of a 2D world where sharks eat fish that come too close.

Initially, the simulation might be configured as in Figure 5a. After a single logical timestep, it might be configured as in Figure 5b where both sharks in the first row have moved into the top middle square and the shark on the right has eaten a fish. Figure 5 displays the logical timesteps for each process; it shows synchronized processes moving together from Step 1 to Step 2. If, however, the processes are not synchronized, the confusing display shown in Figure 6 might be seen. In that figure, some processes are at Step 1 while others have advanced to Step 2. The image is difficult to interpret; sharks in the top row, for example, are not displayed at all because during Step 2 they have moved to processes still displayed at Step 1. The problem is not in the code but in its visualization.

Existing animation tools solve this problem by requiring the programmer to explicitly simulate a global or real world time. Each process must output clock "ticks" and all of the data produced between successive ticks is displayed at the same time. This simulated time has been called "phase time" [13] or "generation" time [16] and it works well for SPMD (Single Program Multiple Data) programs where all processes proceed in unison. It does not work well for less regular programs in which processes independently update data at different points in their execution. In these programs,



(a) (b)

Figure 5: Logically consecutive time steps in the lives of sharks and fishes. Each process is labeled with its current time.

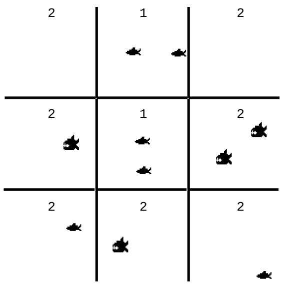


Figure 6: Actual execution of the sharks and fishes program. Processes are executing at different time steps concurrently.

processes that are not involved in an update do not have any way of knowing that time should be advanced. Examples of programs that do not proceed in unison include the dictionary search discussed above and the FIFO queue discussed below.

With our techniques, the user defines abstract events composed of single updates to shark and fish positions on all processes. These abstract events are ordered by *precedes* and, thus, they are automatically separated to produce the desired animation (as shown in Figure 5).

Example 2. Recursive Matrix Transposition. In this example, we consider the animation of a program that performs a recursive matrix transposition as shown in [5]. Starting with the matrix as displayed in Figure 7, we successively transpose submatrices of sizes 2×2, 4×4 and 8×8. A visualization in which processes asynchronously update the display as they compute produces the incomprehensible snapshots of Figure 8. Using our techniques, abstract events corresponding to the

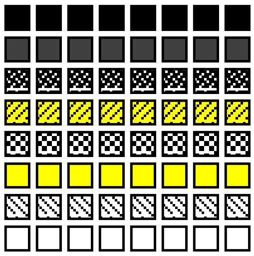


Figure 7: Snapshot from of the matrix before transposition.

recursion levels are shown in a logically coherent manner as in Figure 9. This enhanced visualization is considerably easier to understand than the original.

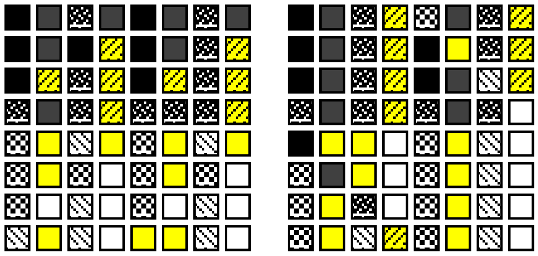


Figure 8: Snapshots from the original animation of the recursive transposition program.

It should be noted that for this algorithm, where all processes recurse to the same depth, the use of phase time or generation time would also suffice. If, however, the depth of recursion was not uniform as, for example, in a binary search, then only our techniques would produce coherent pictures. In the next example, we demonstrate a program that cannot be successfully animated with phase or generation time alone.

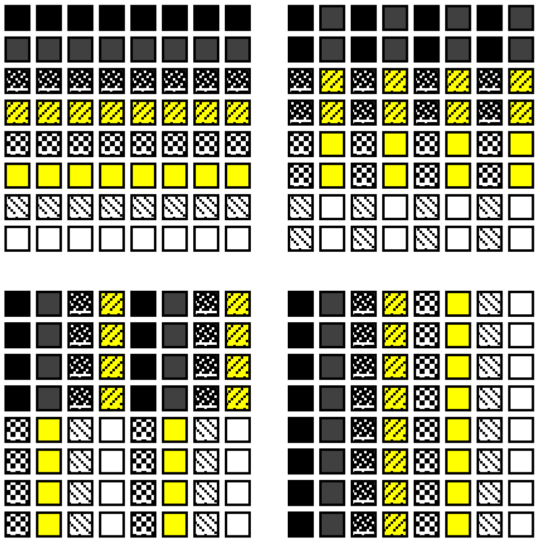


Figure 9: Snapshots from the enhanced animation of the recursive transposition program showing the matrix after the transposition of 2×2, 4×4 and 8×8 submatrices.

Example 3. Parallel FIFO Queue [4]. Here, we discuss an animation of a parallel queue[5] but the scenario is typical of many simulations that model concurrent, nonatomic updates of data.

Figure 10*a* shows an empty circular queue. Each trapezoid represents a location in the queue; the head pointer is labeled I and the tail pointer is labeled D. In Figures 10*b* and *c*, we use a solid trapezoid below the queue to indicate that an *Insert* is in progress and a blank trapezoid above the queue to indicate that a *Delete* is in progress. . Note that Figure 10 shows both an *Insert* and a *Delete* in progress at the same location but this does not indicate an error. Neither *Inserts* nor *Deletes* are atomic. A *Delete* takes three steps: first it gets the position of the next queue element, then it waits until any *Inserts* on that location have completed, and then it removes the element. Its start can legitimately overlap with the previous *Insert*. Figure 10*d* shows several *Inserts* and *Deletes* in progress; shaded queue locations are full. All of these pictures show the queue behaving as expected. But what if there is an error?

Suppose that the wait is omitted during *Deletes* so that it is possible for them to access the queue before data has been inserted. Figure 11 shows snapshots from the animation of an execution sequence in which this actually occurred. It looks as if the first *Insert* has completed and the first *Delete* is still in progress. Why doesn't the error show up?

The problem is in the timing of the screen updates. In this trace, the *Delete* accessed the queue first but the sequence of operations that made up the *Insert* finished before the sequence of operations that made up the *Delete*. The screen was updated as the sequences completed and, thus, it showed the *Insert* first, masking the error. Using logical time, however, this anomaly does not occur. Figure 12, for example, shows the same execution trace animated with our techniques. *Inserts* and *Deletes* were treated as abstract events and they were ordered logically by the dependencies

[5]Note: We have assumed a nonshared memory paradigm but for this example we simulate a shared memory location as a process: if p is a process and m a memory location, a write from p to m is treated as a message from p to m and a read by p from m is treated as a message from m to p.

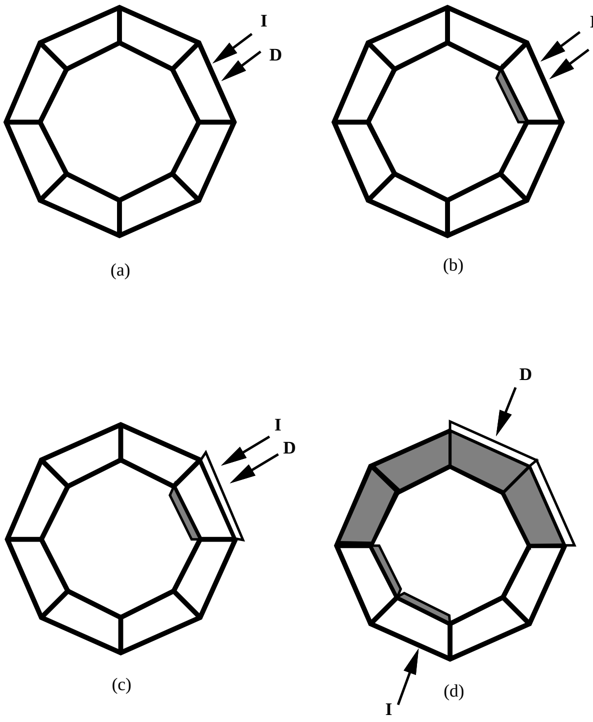


Figure 10: Snapshots from the animation of the parallel queue.

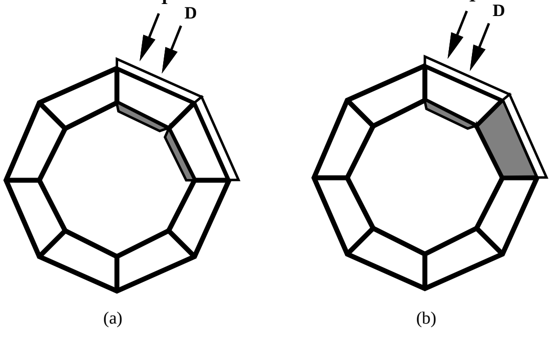


Figure 11: Snapshots from the original animation of an erroneous version of the parallel queue.

caused by queue accesses (dependencies resulting from the process waits were ignored). The error is clearly visible.

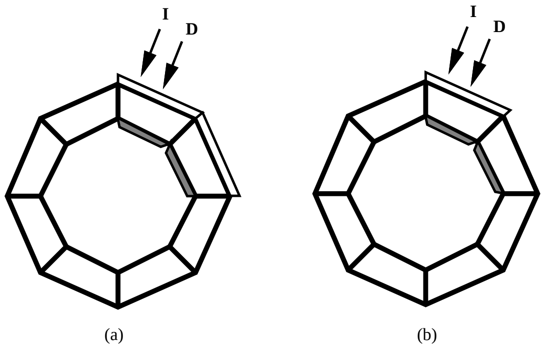


Figure 12: Snapshots from the logically ordered animation of the erroneous version of the parallel queue.

This scenario is typical of simulations in which different parts of a data structure are concurrently updated by nonatomic operations. Simulated global time is not useful because not all processes participate in each update and, thus, not all processes are aware of the need to advance their clocks. Our techniques allow the user to focus on just those temporal orderings that are relevant to the logical behavior under investigation. They also allow the user to change that focus as different aspects of the behavior come under scrutiny.

5 Conclusions

All visualization tools that display data as it is manipulated by a parallel, MIMD computation must contend with the effects of asynchronous execution. In some cases, this is done with the explicit simulation of a global clock but that is not always feasible. Here we propose a less restrictive mechanism in which logical time is manipulated in order to produce coherent animations of parallel program behavior. Our techniques "interpret" program behavior in light of user-defined abstractions and generate animations based on a logical rather than a physical view of time. In this paper, we have demonstrated the application of these techniques more generally to a variety of types of data visualizations.

References

[1] P. C. Bates. *Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior.* Ph.D. Thesis, Department of Computer Science, University of Massachusetts, Amherst MA (1986).

[2] Alva Couch. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* Tufts University, Technical Report 88-4 (1988).

[3] C. J. Fidge, "Partial Orders for Parallel Debugging," *Proceedings of the ACM SIPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183–194 (1988).

[4] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph and Mark Snir, "The NYU Ultracomputer – Designing and MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers* C-32(2), pp. 175-189 (1983).

[5] M. T. Heath and J. A. Etheridge. *Visualizing Performance of Parallel Programs.* Oak Ridge National Laboratory Technical Report ORNL/TM-11813 (1991).

[6] R. W. Hockney and C. R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Bristol, England (1981).

[7] Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," *Proceedings 1987 International Conference on Parallel Processing*, pp. 735-738 (1987).

[8] Alfred A. Hough and Janice E. Cuny, "Initial Experiences with a Pattern-Oriented Parallel Debugger," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 195-205 (1988).

[9] Alfred A. Hough and Janice E. Cuny, "Perspective Views: A Technique for Enchancing Visualizations of Parallel Programs", *Proceedings 1990 International Conference on Parallel Processing*, pp. II 124-132 (1990).

[10] Alfred A. Hough. *Debugging Parallel Programs Using Abstract Visualizations.* Ph.D. Thesis, Computer Science Department, University of Massachusetts, Amherst MA (1991).

[11] W. Hsuesh and G. E. Kaiser, "Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language," *Proceedings of the ACM SIPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 236-247 (1988).

[12] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM* 21(7), pp. 558-565 (1978).

[13] Thomas J. LeBlanc and John M. Mellor-Crummey and Robert J. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing* 9, pp. 203-217 (1990).

[14] A. R. Omondi and J. D. Brock, "Implementing a Dictionary on Hypercube Machines," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 707-709 (1987).

[15] Daniel A. Reed, Robert D. Olson, Ruth A. Aydt, Tara M. Madhyastha, Thomas Birkett, David W. Jensen, Bobby A. A. Nazief, and Brian K. Totty, "Scalable Performance Environments for Parallel Systems," *Proceedings Sixth Distributed Memory Computing Conference*, pp. 562-569 (1991).

[16] David Socha, Mary L. Bailey, and David Notkin, "Voyeur: Graphical Views of Parallel Programs", *Proceedings of the ACM SIPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 206-217 (1988).