Perspective Views: A Technique for Enhancing Parallel Program Visualization*

Alfred A. Hough and Janice E. Cuny

COINS Technical Report 90–02
January 1990

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

*The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research under contract N000014-84-K-0647 and by the National Science Foundation under grants DCR-8500332 and CCR-8712410.

Abstract

Visualization is widely used as an aid to understanding program behavior. Standard visualization techniques, however, do not address the fundamental problems of complexity and concurrency in parallel computations. Complexity can be managed with the use of abstraction but only if visualization tools are able to produce comprehensible displays of nonatomic, concurrent events. In this report, we present techniques for "reordering" events prior to display in order to enhance visualizations. In many cases, the reordering is accomplished without violating dependencies and thus results in equivalent, comprehensible visualizations. In other cases, however, consistent reorderings are not possible because of intertwined dependencies. For these cases, we introduce *perspective views* that enable the user to construct partially consistent reorderings. We demonstrate the use of our techniques in three different tools for the visualization of parallel programs.

Keywords: parallel debugging, visualization, program animation, behavioral abstraction

1. Introduction

Visualization is widely used as an aid in understanding the behavior of programs. Visualization tools for sequential programs have been available for a number of years [21, 24, 3]. Similar aids for parallel programming have been developed [22, 12, 7, 18, 8, 4] but, for the most part, they have been extensions of sequential tools that do not address the fundamental problems of complexity and concurrency in parallel computations.

Parallel computations, with hundreds or perhaps thousands of independent, asynchronous interacting processes, are extremely complex. To be effective, visualization tools must assist the user in managing this complexity. Several existing tools accomplish this with the use of abstraction [22, 1, 12], allowing the programmer to group sets of program actions into single *abstract events* for the purpose of visualization. Consider, for example, an application running on a hypercube in which, on every iteration, nodes communicate in a sequence of phases, each phase crossing a different dimension of the cube. If the processes execute independently without synchronization, an animation system might produce the sequence of snapshots shown in Figure 1. Initially, the processes

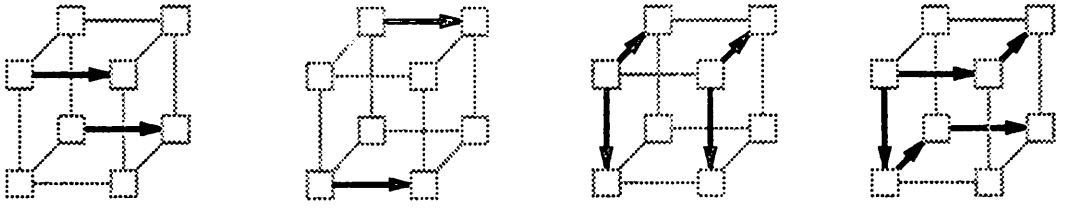


Figure 1: Possible snapshot from animation of cube program showing individual communication events.

communicate at slightly different times within the same phase (Figure 1 a-b). After a number of iterations, however, their execution becomes so skewed (Figure 1c-d) that some of the processes are still completing communications from the second or third phase of an iteration while others have progressed to the first phase of the next iteration. It is

doubtful that this series of snapshots would be meaningful to the programmer who had envisioned all communications within a phase occurring simultaneously. Instead, that programmer would prefer to see his code's behavior in terms of abstract "phase communication events" in which all communications within a phase are treated as a logical unit. To show such abstractions as distinct visual objects, a visualization tool must contend with problems of concurrency.

Abstract, nonatomic events may be concurrent, overlapping in both time and space. In the hypercube example, all three phase communication events can be executing simultaneously on the same set of processors. How are they to be visualized without superimposing, and thus obscuring, their displays? In this report, we present techniques for automatically achieving the visual discrimination of abstract, concurrent events.

Our techniques temporally recorder execution traces prior to display. We attempt first to construct *consistent reorderings* that preserve the partial ordering of events imposed by the sequentiality of processes and by interprocess dependencies. For the purposes of this report, we consider only message-passing systems and thus restrict interprocess dependencies to those resulting from communication; specifically, the "receive" of a message is dependent on its "send." In the hypercube example, a consistent reordering can be achieved by putting first phase events before second phase events and second phase events before third phase events. It results in an animation sequence showing the orderly progression of phases as in Figure 2. The abstract events of interest have been

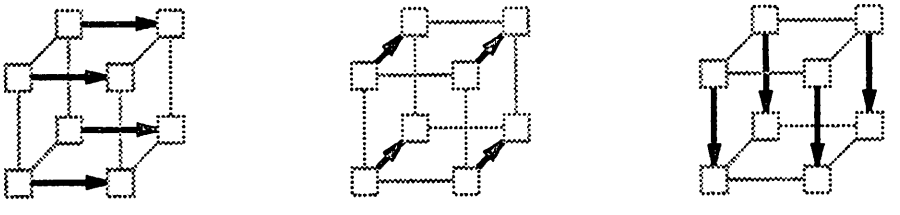


Figure 2: Intended, logical phases of communication in hypercube program.

temporally separated, providing visual discrimination and allowing the user to under-

stand his program's behavior in terms of his own conceptualizations. This visualization shows an execution sequence that is equivalent to the one that actually occurred; that is, all processors execute the same sequences of operations with the same interprocess dependencies.

For many parallel computations, however, such simple reorderings are not possible because of intertwined dependencies. For these cases, we introduce the concept of *perspective views*. Perspective views (the main contribution of this report) enable the user to create comprehensible visualizations by selectively ignoring some events and dependencies in establishing partially consistent reorderings. Each such reordering provides a partial view of the system's behavior; several different partial views may be needed to understand its full behavior. We demonstrate the utility of this approach with a number of examples.

This work was done in the context of our parallel debugger, Belvedere [12], but the techniques are generally applicable to the visualization of concurrent, abstract events. We assume only that a stream of locally-timestamped events is available and that it has been grouped into abstract events for subsequent visualization by some display tool. In order to demonstrate the generality of our techniques, we demonstrate their application to three different types of visualization tools for parallel computation.

In Section 2, we define our notions of ordering between abstract events. In Section 3, we discuss reorderings and introduce the concept of perspective views. In Section 4, we present an algorithm for computing both full and partial perspective views. In Section 5, we demonstrate the use of perspective views and, in Section 6, we present our conclusions.

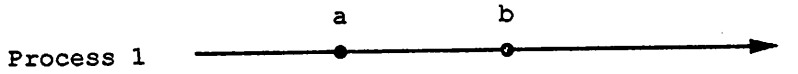
2. Orderings between Abstract Events

We present here three ordering relations — *precedes*, *parallel*, and *overlapped* — that characterize the possible relationships between two abstract events.

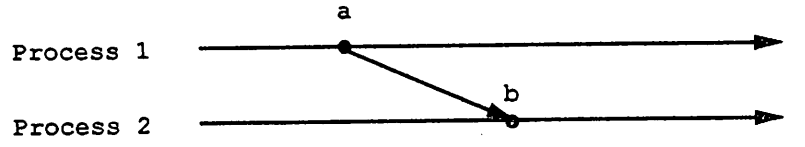
Other ordering relations on abstract events have been proposed[17, 2, 14, 6]; ours is an extension of Lamport's ordering of primitive events[16] that has been tailored to the

needs of visualization systems. Lamport's ordering, called *happened before* and denoted $\rightarrow$, is defined on atomic actions of a distributed system; each action a is assumed to have a process-local timestamp, *timestamp*(a). Positioning events on their process time-line according to their local timestamps, $a \rightarrow b$, iff one of the following conditions holds:

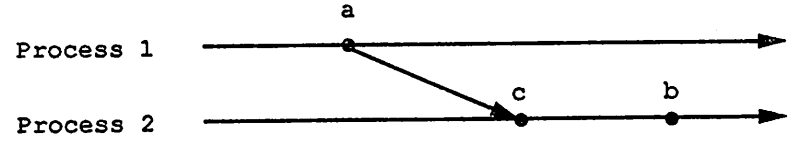
i) events a and b happen on the same process and timestamp(a) < timestamp(b):



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):

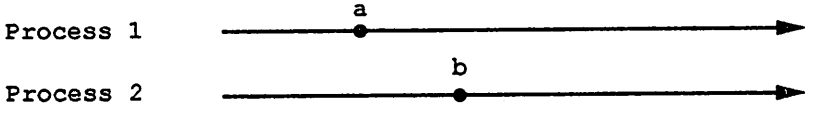


iii) $a \rightarrow b$ is in the transitive closure of i) and ii):



Two events are unordered if they are not related by *happened before* [1]:

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

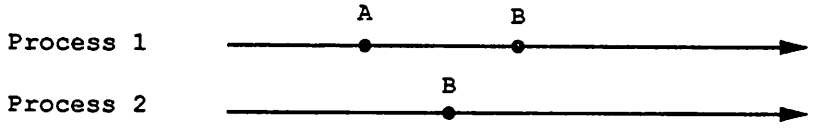


[1]This notation is slightly different than Lamport's.

Happened before is defined on primitive events; for our visualization purposes, we must consider relations between abstract, nonatomic events. Here, we define abstract events to be sets of primitive events (later we extend this to allow hierarchical definitions). A straightforward extension of *happened before* might require that for abstract events A and B, A→B only when each element of A *happened before* each element of B:

$$A \rightarrow B \text{ iff } \forall\, a \in A, b \in B : \; a \rightarrow b$$

This extension, however, is overly restrictive. It does not capture all of the behavior that we might reasonably wish to consider "serializable" for the purposes of visualization. Consider, for example,



where each dot is a primitive event labeled with the name of the abstract event to which it belongs. Under this simple extension, event A would not have *happened before* event B because there is no relation between the event belonging to A on Process 1 and the event belonging to B on Process 2. However, for visualization purposes, we might consider A to have happened before B because we would not violate any dependencies by showing the visualization of A before the visualization of B.

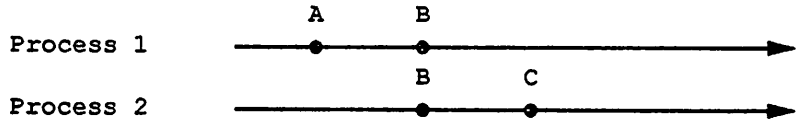
To define the relations that we need, we first define a relation *partially precedes*, denoted $\mapsto$. Informally, $A \mapsto B$ if some part of A *happens before* some part of B or A and B share a primitive event; formally

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

Partially precedes allows us to order abstract events for visualization purposes that would not have been ordered by the simpler extension to *happened before*. In the following example,



partially precedes provides a total ordering for A, B, and C (because *partially precedes* is transitive, $A \mapsto C$ even though no element of A *happened before* any element of C).

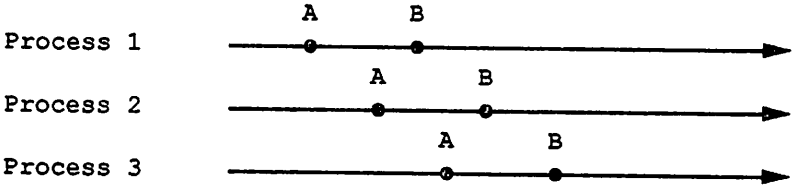
Using *partially precedes* we define three possible relations between two abstract events:

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

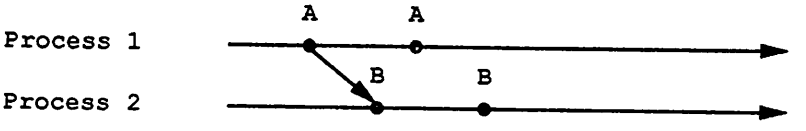
Parallel: $A \parallel B$ iff NOT $(A \mapsto B)$ and NOT $(B \mapsto A)$, (alternatively, iff $\forall a \in A, b \in B : a \parallel b$)

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

Precedes is intended to capture the notion that one abstract event logically occurs before another. $A \Rightarrow B$ if some element of A occurs before some element of B (in the extended sense of *partially precedes*) but no element of B occurs before any element of A. In the following example,

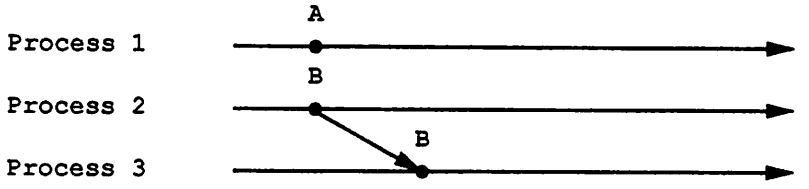


$A \Rightarrow B$ because each element of A *happened before* the element of B on the same process but is unrelated to the elements of B on the other two processes. Note that $A \Rightarrow B$ even though the element of B on Process 1 has an earlier local timestamp than the element of A on Process 3. In the following example,



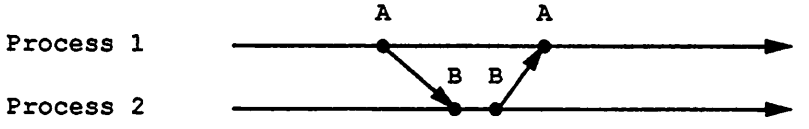
$A \Rightarrow B$ because communication imposes the *happened before* relation between the send event of A and the receive event of B.

Parallel events are *unordered.* In the following example,

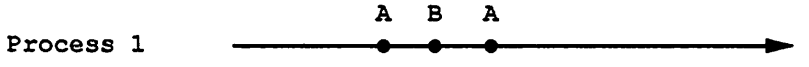


$A \parallel B$ because the *happened before* relation does not hold between any pair of A and B elements. Note that *parallel* events must occur entirely in disjoint processes because events within a process would necessarily be ordered.

Overlap is intended to describe events which are logically intertwined: some part of each event happens before some part of the other. In the example,



$A \Leftrightarrow B$ because the first element of A on Process 1 *happens before* the first element of B on Process 2 but the second element of B on Process 2 *happens before* the second element of A on Process 1. Overlapped events can also occur on a single process:



In the next section, we consider the use of these relations in reordering abstract events to provide more comprehensible visual displays.

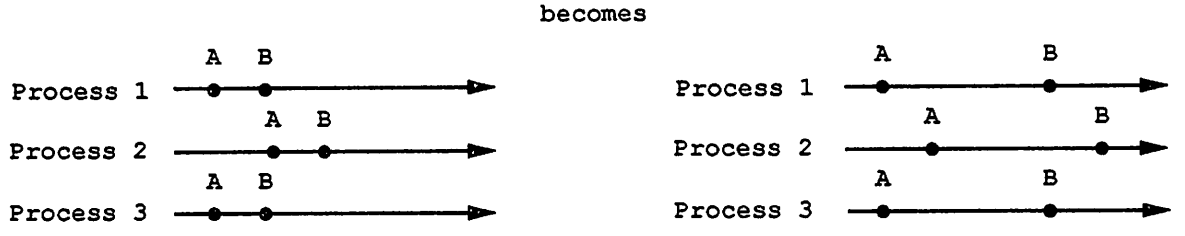
3. Reordering Transformations and Perspective Views

For abstract events related by *precedes* or *parallel*, we provide transformations that produce *consistent reorderings.* A reordering is *consistent* if (1) all processes execute the

same sequence of events and (2) its *happened before* relation is identical. To an observer inside the system (such as a process), consistent behaviors are indistinguishable. For events related by *overlap*, consistent reorderings are not possible. For these events, we introduce the notion of a *partially consistent reordering* in which (1) each process executes a subsequence of its original event sequence and (2) the *happened before* relation is a subset of the original *happened before* relation.

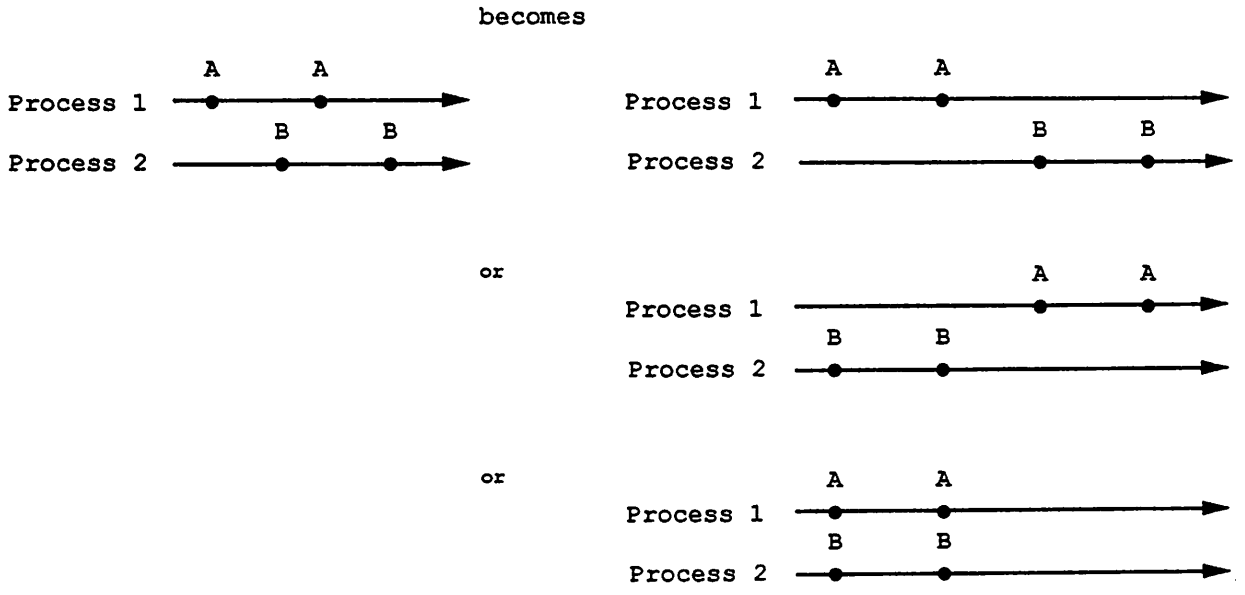
We first describe the consistent reorderings needed for *precedes* and *parallel* and then we describe the partially consistent reorderings needed for *overlap*.

The transformation associated with *precedes* separates two abstract events (by assigning new timestamps) so that all elements of the first event complete before any element of the event begins. A visualization of the transformed system would show the abstract events as distinct visual units. Thus,



The transformed behavior is consistent with the original behavior. In addition, it is easier to understand because logically related events are grouped together.

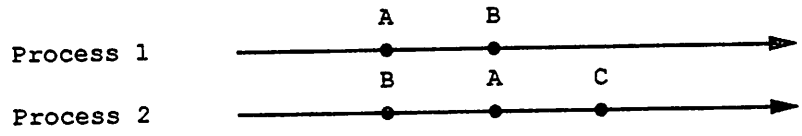
For the transformation associated with *parallel*, there are a variety of options:



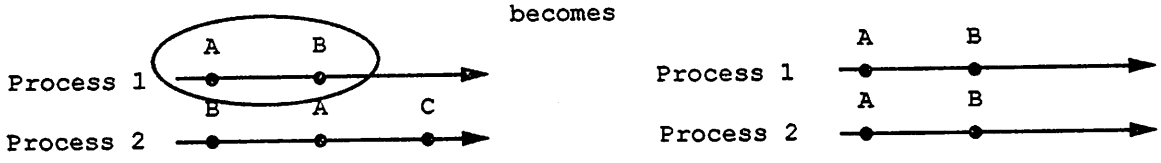
We chose the third — in which *parallel* events are started at the same time — for aesthetic reasons. Parallel events must occur on separate processes so this does not conflict with our goal of making the events visually distinct.

Both transformations were used in producing the comprehensible display of the hypercube computation in Figure 2. The transformation for *precedes* separated the abstract events and the transformation for *parallel* synchronized their subevents.

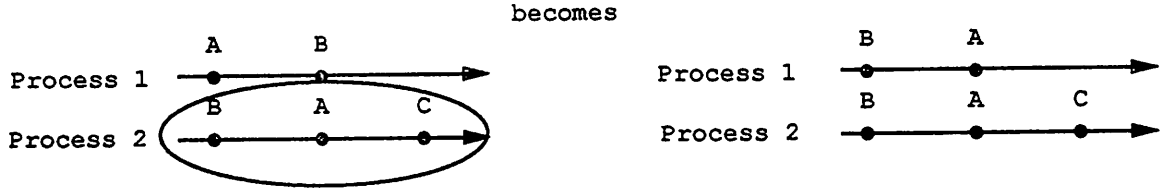
If two events are related by *overlap*, it is not possible to assign new timestamps that will both separate the events and preserve the *happened before* relation. For these events, we use partial reorderings based on a user-selected subset of the events called a *perspective*. If all events are selected, the subset is called a *full perspective*; otherwise it is called a *partial 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 (they are not passed on to the display tool) and the remaining events are reordered in a manner consistent with the computed relations. We might, for example, consider the following behavior



from the partial perspective of Process 1; in this case, we would use only the Process 1 primitive events for computing ordering relations. Thus,

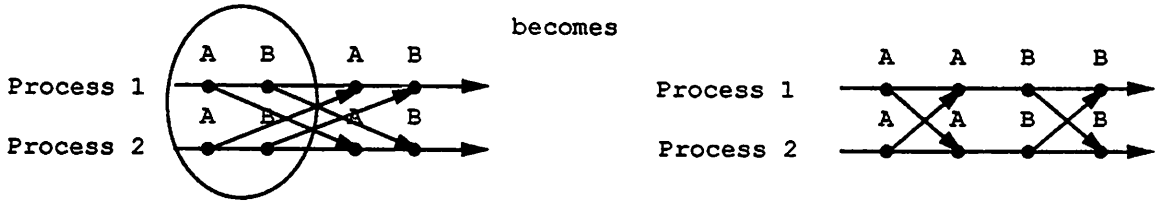


where circles indicate the events used to compute the ordering. The ordering on Process 1 has been applied to all events, including those on Process 2; abstract event C has been deleted since it does not contain elements in the perspective. Alternatively, the same system can be decomposed from the perspective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

We define a *perspective view* to be a consistent or partially consistent reordering of an execution sequence from a perspective for the purposes of visualization. Previously, we reported on a version of perspective views [11, 12] that allowed only perspectives from a process or set of processes. We found those perspective views to be too limited. Here we allow arbitrary subsets of events to serve as perspectives. Thus, for example, in visualizing the following pair of "message exchange events," we might consider only send operations, in which case,



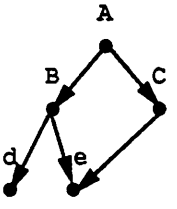
because each process sends a message as part of the A exchange before it sends a message as part of the B exchange. From the perspective of the send events, A *precedes* B. (The perspective view would have been the same, in this case, if we had used just the receives as our perspective.)

In the next section, we provide an algorithm for computing perspective views.

4. Algorithm for Computing Perspective Views

The algorithm is assumed to operate in the context of a pattern recognizer that provides a stream of locally-timestamped events (both primitive and abstract) and a display system that produces a visualization of the reordered behavior.[2] Abstract events are generated by the pattern recognizer, reordered by the perspective algorithm, and visualized by the display system.

The algorithm permits hierarchical definitions of abstract events. Each event is represented as a directed acyclic graph in which the nodes are events (primitive or abstract) and the edges signify "contains". Thus, for example,



event A consists of the two abstract events B and C; event B consists of the two primitive events d and e; and abstract event C contains the single primitive event e. Our

[2]We currently use the EBBA tools [1] as our pattern recognizer and the Belvedere debugger [12] as our display tool.

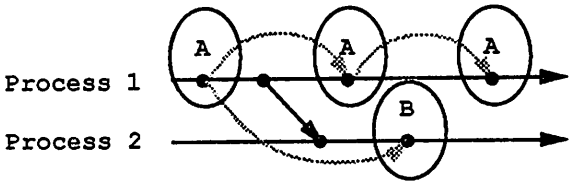
algorithm constructs a graph that represents the entire behavior of the system by adding dependency edges between nodes in these graphs.

There are three inputs to the algorithm: (1) the set of timestamped events to be viewed; (2) the set of events defining the perspective; and (3) a set of dependencies representing constraints imposed by the display system. These last inputs are intended to insure that the reordered behavior can be successfully displayed. Our display tool, for example, requires that a message transmission precede every message receipt; the explicit inclusion of this as a constraint prevents such a dependency from being deleted in a partial perspective. Constraints are supplied to the perspective algorithm as a list of artificial *happened before* relations. All input events are represented as records. An abstract event is represented a variable length record that contains pointers to the records representing the immediate children of the abstract event.

The perspective algorithm has four steps:

1. *Construct a graph representation of the input.*

   Nodes in the graph represent events (both primitive and abstract). Edges in the graph come from three sources: (1) the binding of events to abstract events; (2) the constraints supplied from the display system; and (3) the *happened before* dependencies between primitive events. In the first case, we add a bidirectional edge between an abstract event and each of its immediate children, copying the information from the input record representing the abstract event. In the second case, edges are added to represent the dependencies introduced by display system constraints. In the third case, we add a minimal set of edges sufficient to generate the restriction of the *happened before* relation to the events of the perspective. Thus, for example, in the following, only edges representing the dependencies shown in gray are added for the circled perspective:



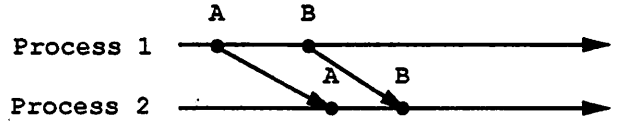
Initially, the graph contains only nodes representing events in the perspective; other nodes are added as needed to complete dependency edges required from sources (1) and (2).

2. *Find the strongly connected components of the above graph.*
   The strongly connected components of our graph correspond to sets of events that are to be treated as distinct visual units. We use an algorithm due to Tarjan[23] to find the strongly connected components.

3. *Assign depths to components.*
   The depth of each component is the length of the longest path to that component, counting along strongly connected components. To compute this length we first compact the graph so that each strongly connected component is represented by a single node; the compacted graph is acyclic. Path lengths are computed using a special case of the Critical Path Method[15, 5] normally used for job scheduling.

4. *Reorder events.*
   The events are reordered so that all components at the same depth start at the same time. Each strongly connected component is moved as a unit. Reordering is accomplished with a sort on the depth of each component followed by a pass over the events in sorted order to assign new timestamps.

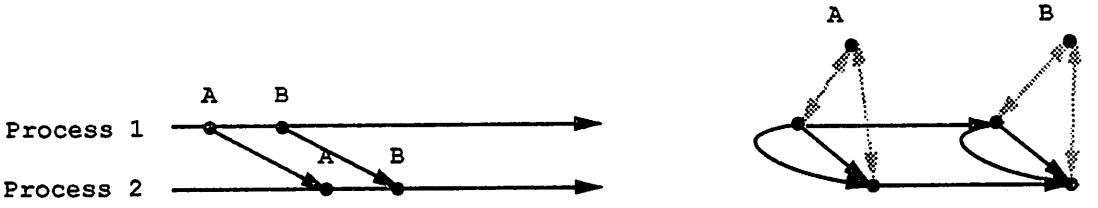
We illustrate the algorithm first with a full perspective and then with a partial perspective. For the full perspective, we assume that the following behavior



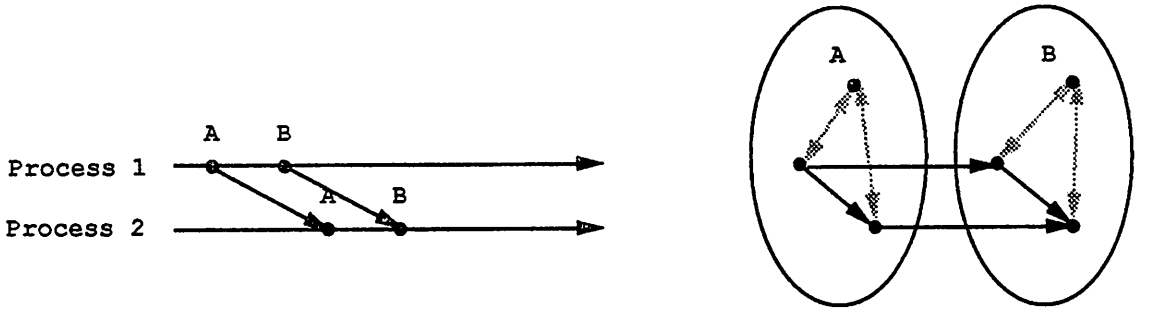
is to be reordered prior to animation.

Step 1: Dependencies are assigned. In the following example dependencies due to the sequential nature of processes and interprocess communication are shown in black and bidirectional dependencies between each primitive event and the abstract event that

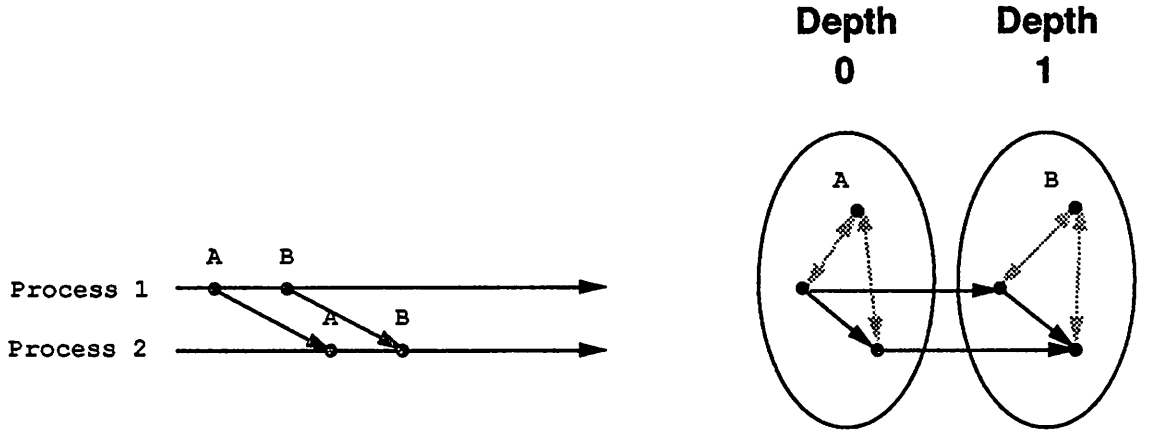
contains it are shown in gray. Because we intend to animate the behavior, application specific dependencies are added to insure that messages are always sent before they are received; these are indicated by curved arrows. (The curved arrows in the following figure indicate only that a dependency is also supplied by the display system. The graph being built is not a multigraph.) The resulting graph is



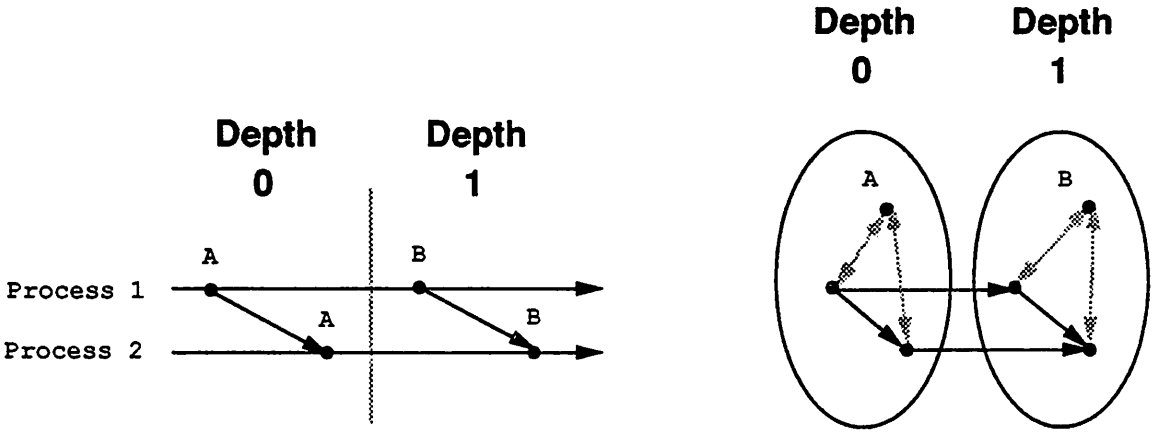
Step 2: The strongly connected components of the graph are calculated(shown circled).



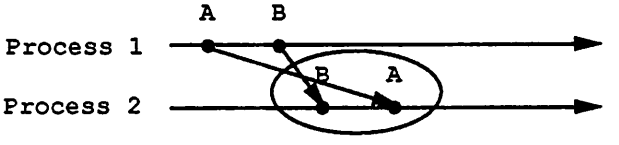
Step 3: Depths are assigned to each component.



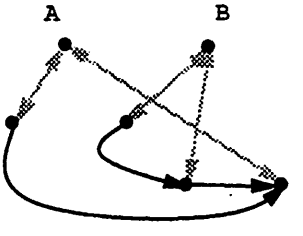
Step 4: The components are separated and moved to new times corresponding to the assigned depths.



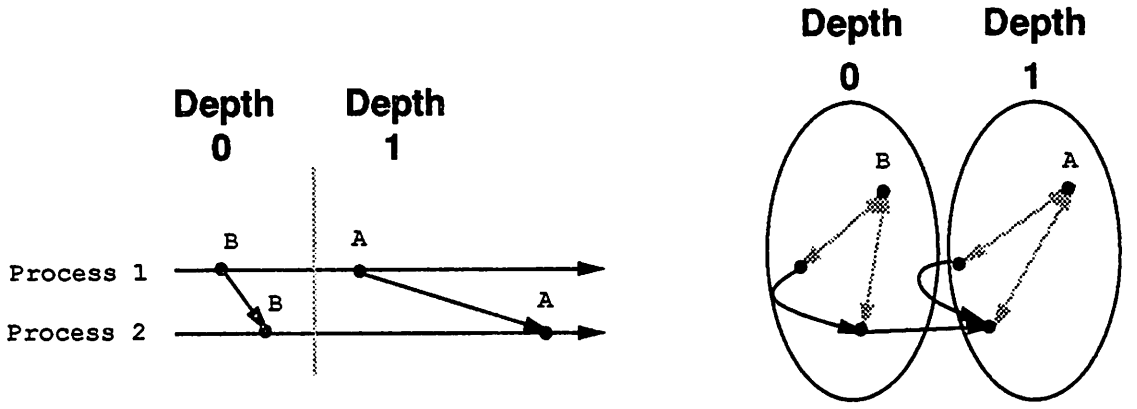
For a partial perspective, we assume that the following behavior



is to be reordered prior to animation. The same steps are followed except that edges of the first two types are only recorded for events in the perspective. When viewed from the perspective of Process 2, the behavior results in the graph

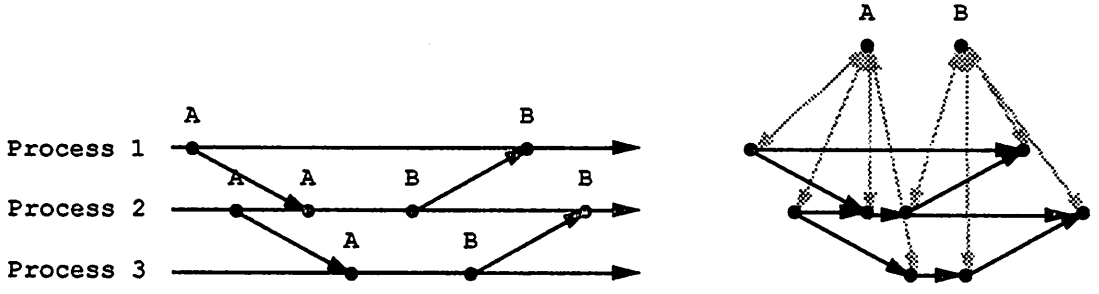


which, in turn, results in the reordering



The algorithm computes a reordering based on a perspective views in $O(N \log N + E)$ time where N is the total number of primitive and abstract events in the system and E is the number of dependencies(ordering relationships) between the events. E can be as large as N^2 in the worst case but in practice it is considerably smaller. A more complete analysis of this algorithm appears in Appendix 6.

A perspective view reduces asynchrony by changing the relative timing between abstract events but does not treat asynchrony within an abstract event. The following system, reordered using a full perspective,

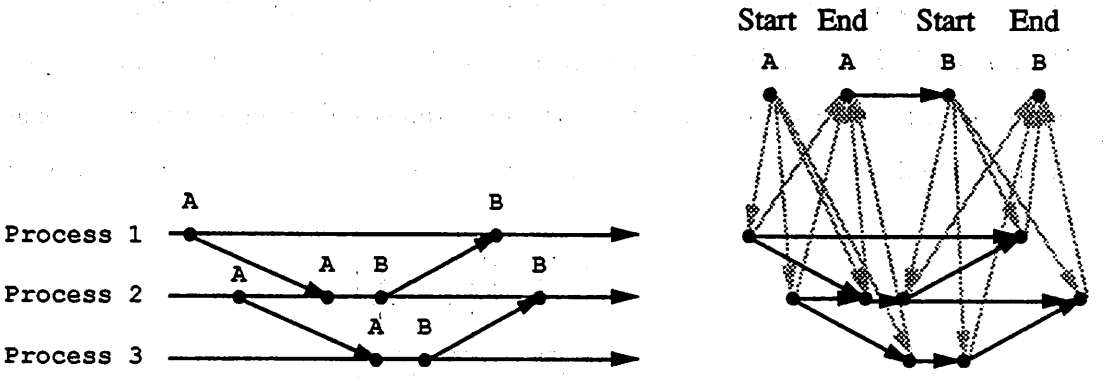


shows asynchrony remaining within the A and B events; that is, the send events within the abstract events are *parallel* but they have not be moved to start at the same time.

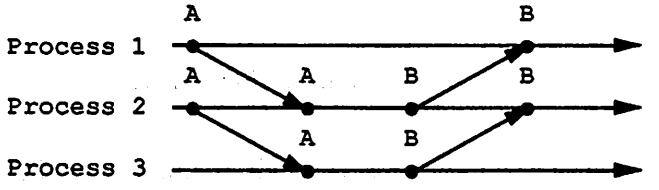
Since we allow abstract events to contain other abstract events, however, we can recursively apply the algorithm to sub-events within each abstract event. In the implementation, we do not actually apply the algorithm recursively but instead achieve the same effect by modifying the graph and running the algorithm as usual. The graph is

modified so that instead of a single node to collect all of the dependencies in an abstract event, we use pseudo "start" and "end" nodes; the start node *happens before* all members of the associated abstract event and all members of an event *happen before* its end node. The bidirectional arcs between an element and its parent node are thus replaced with directional arcs from the start node to each element and from each element to the end node. When a *happened before* dependency is found between primitive events a and b, an arc is added between the corresponding end and start nodes of the topmost abstract events they belong to. (This usually adds no additional arcs, but in pathological cases this can force the time complexity of the algorithm to be $O(N \log N * E)$).

Using this "recursive" version of the algorithm, the above example results in the graph



and the reordering



5. Examples of the Utility and Generality of Perspective Views

Perspective views can be used to enhance visualizations of parallel systems. In this section, we demonstrate their application in three different types of visualization

tools: tools producing process-time graphs such as Moviola [7]; tools facilitating user-controlled animations such as Voyeur [22]; and tools providing automatic animations such as Belvedere.

5.1 Process-Time Graphs

Several display tools[7, 9] produce process-time graphs showing processes along one axis and time along the other. We present two examples that demonstrate the use of perspectives in making these graphs more comprehensible; the first uses a full perspective and the second a partial perspective. It should be noted that we are primarily interested in visualizations for debugging purposes and that our temporal reorderings would not be useful in applications of process-time graphs to performance debugging.

Example 1. Successive Overrelaxation (SOR) [10]. In this iterative approximation to a PDE, processes arranged in a square mesh repeatedly update their values as functions of their neighbors' values. Even and odd processes alternate their execution: first odd

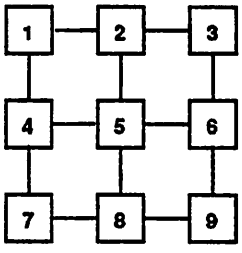


Figure 3: SOR mesh.

processes send their current values to their neighbors while even processes read their neighbor's values and update their own; then the processes reverse roles. Processes send values in a counter-clockwise sweep beginning with the neighbor above them; processes read values in a complementary, clockwise sweep beginning with the neighbor below them. The resulting communication for the labeled mesh of Figure 3 is shown in the process-time graph of Figure 4. Because of asynchrony, the communication patterns of these processes are difficult to see in the graph.

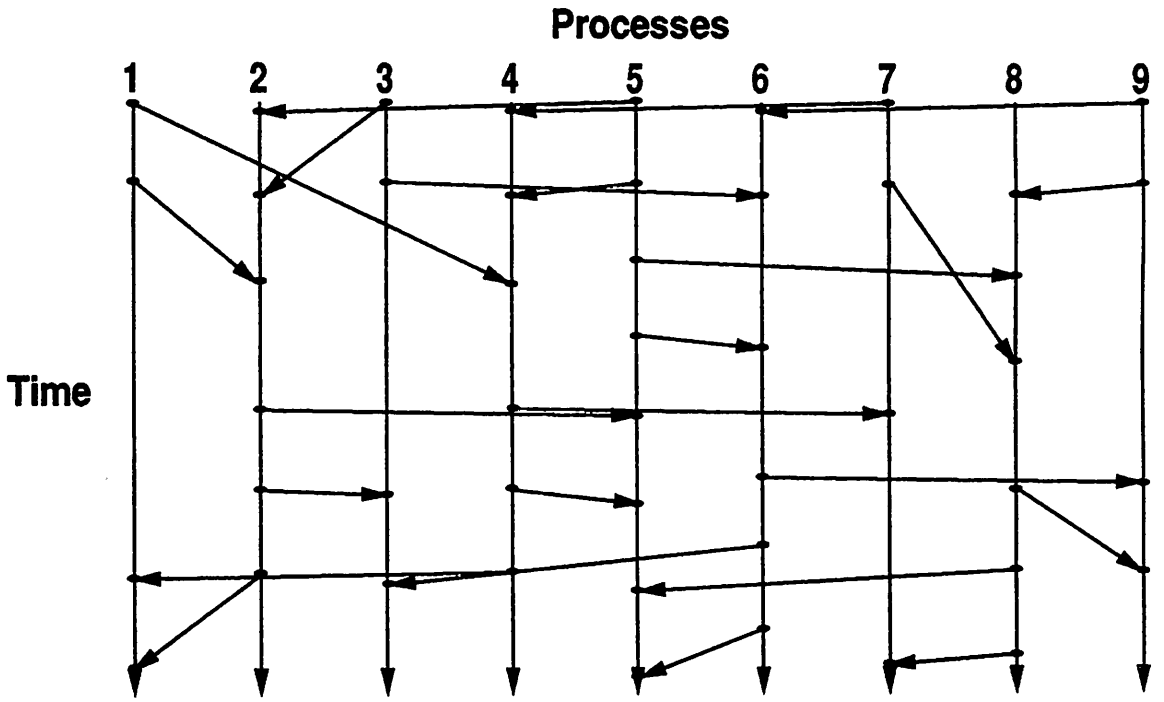


Figure 4: Primitive communication on a process-time graph for the SOR program.

To use our reordering tool, we first define a two-level hierarchy of abstract events: "messages" are defined as appropriately matched pairs of send and receives and are then used in defining abstract events of the form "north messages from even processes", "west messages from odd processes", *etc.* Because the processes send and receive in complementary sweeps, these abstract events are pairwise related by *precedes*. Thus, a recursive, full perspective successfully separates them into distinct bands as shown in Figure 5.

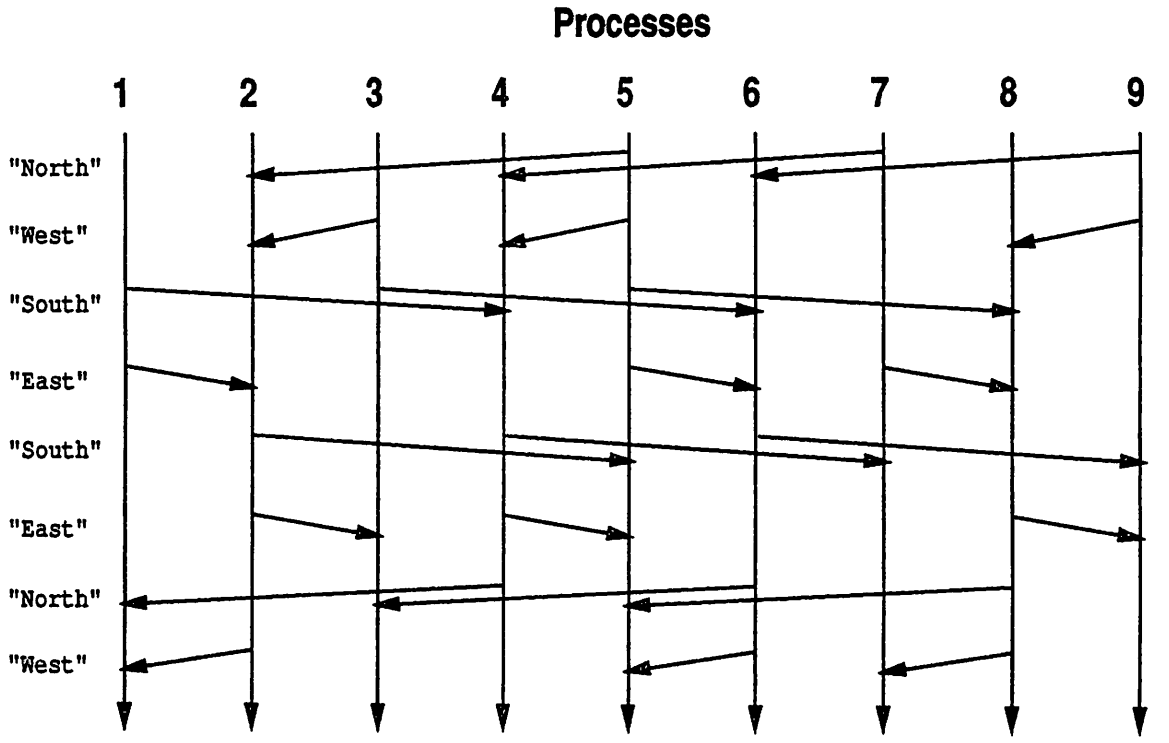


Figure 5: Logical communication on process-time graph from the SOR algorithm.

This visualization clearly shows the patterns of communication that the user intended. Because it is a consistent, full perspective, it preserves all relevant dependencies (that is, it does not change the *happened before* relation) and allows the user to view and debug his program at an appropriate level of abstraction.

Example 2. Median Filter [20]. In this algorithm, each node repeatedly updates

itself with the median of its neighbors' values. It differs from the SOR algorithm in that all processes execute simultaneously: each sends values to all of its neighbors; receives values from those neighbors; and then updates its own value. Both send and receive cycles use the same counter-clockwise sweep, beginning with the process above. Again, the patterns of communication are difficult to discern in the standard process-time graph, shown in Figure 6.

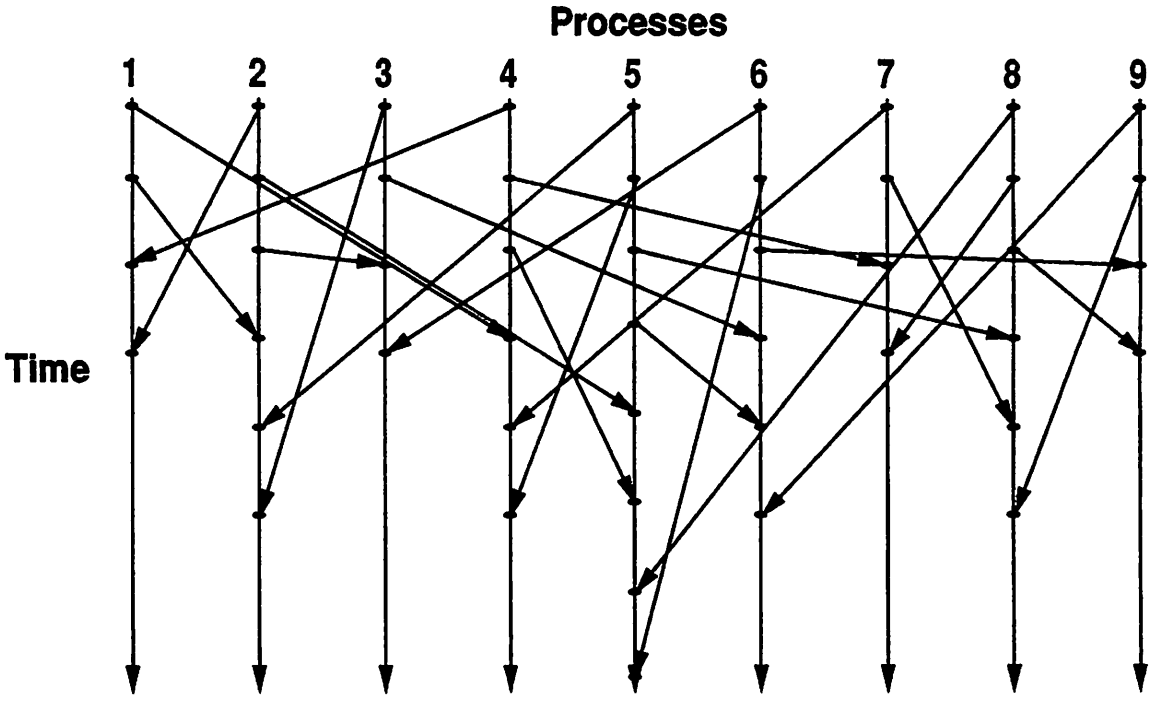


Figure 6: Primitive communication on process-time graph from the median filter algorithm.

The abstract events of interest are similar to those used in the SOR example, events of the form "north messages from all processes", "west messages from all processes", *etc.* Unlike SOR events, however, these abstract events will not be related by *precedes* because each process overlaps its participation in many abstract events: first sending in all directions (to begin its participation in four events) and then reading from all directions (to end its participation in those events). Thus, for example, if we look at Processes 2 and 5, they exchange two messages, one as part of the "north message" event and one as

a part of the "south message" event. These two events are related by *overlap* and they cannot be consistently separated in a full perspective.

To choose a partial perspective that will separate the events, we note that all processes send messages in the same counter-clockwise pattern. A partial perspective that includes only send events produces a process-time graph in which the abstract events are shown as bands of messages as in Figure 7. A partial perspective based on receive events would

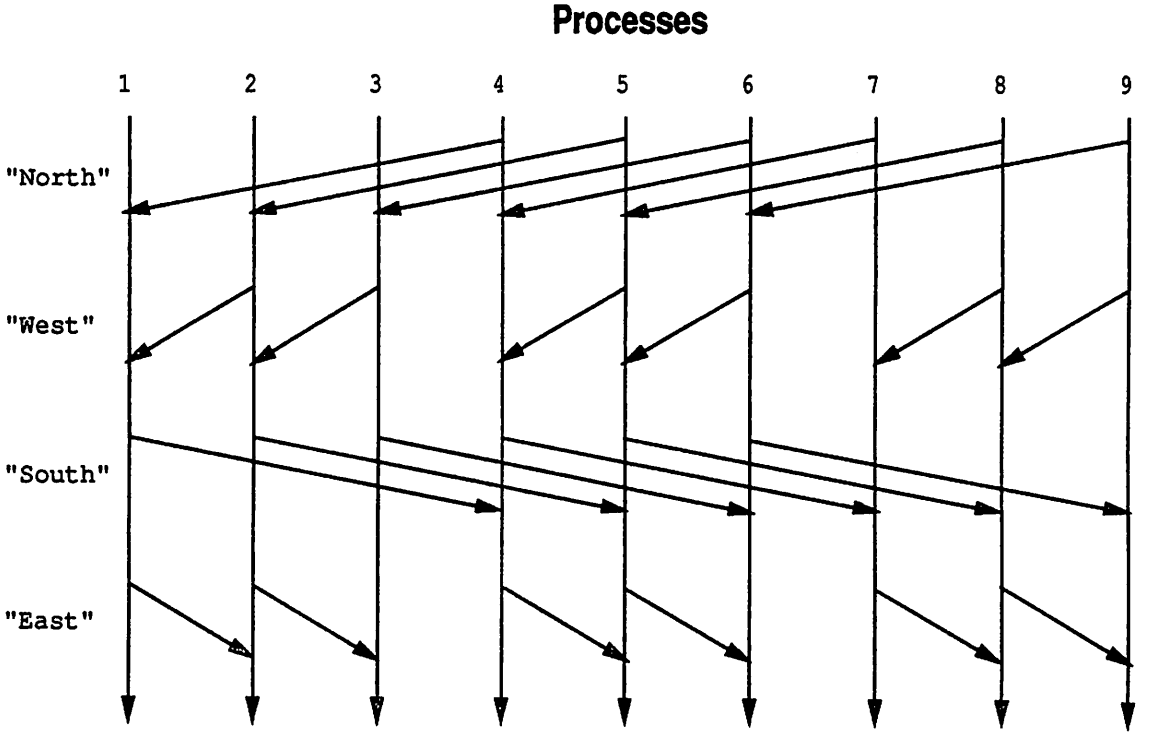


Figure 7: Logical communication on process-time graph from the median filter algorithm.

also separate abstract events but the resulting order of the abstract events would differ because receiving processes try to read first from their north neighbors as part of a "south messages" event. The reordering is not entirely accurate because it omits some temporal dependencies. On the other hand, these views provide a more comprehensible visualization of behavior then would be possible if all temporal relations were preserved. We anticipate that programmers will want to create a number of partial views, each

providing different insights into their program's behavior.

5.2 User-Directed Animations

User-directed animations can provide high-level, application-oriented visualizations of parallel programs. Unless special care is taken, however, even these animations will be difficult to understand.

Example 3. Sharks and Fish [22]. For this example, we use a load-balancing algorithm operating on an underwater simulation taken from a Voyeur paper [22]. The simulation contains sharks and fish that can move at most once per unit time; sharks eat fish. Initially, the simulation might be configured as in Figure 8a.[3] After a single logical

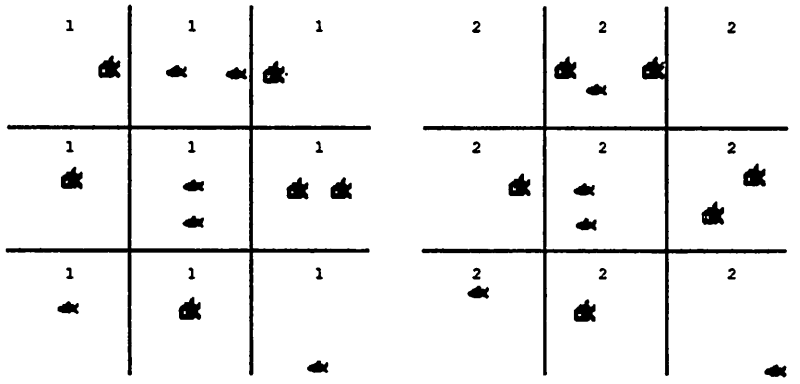


Figure 8: Logically consecutive time steps in the lives of sharks and fishes. Each process is labeled with its current time.

timestep, it might be configured as in Figure 8b where both sharks in the first row have moved into the top middle square and the shark on the right has eaten a fish. The load balancing algorithm seeks to balance the number of animals under the control of each process.

[3]The figures in this section were not produced by Voyeur. They are hand simulations of Voyeur's actions based on the published example and conversations with the author.

Figure 8 displays the logical timestep for each process. The figure shows synchronized processes moving together from step 1 to step 2. If, however, the processes are not synchronized, the confusing display shown in Figure 9 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 shown at step 1. The problem is not in the code but in its visualization.

Either Voyeur or perspective views can be used to produce the coherent animation shown in Figure 8. In Voyeur, the user explicitly simulates a global clock by associating a step number with each process action during execution. Voyeur displays all process actions with the same step number at the same time. In Belvedere, the user defines abstract events (after execution) that correspond to single movements of sharks and fish on all of the processes. Such events are ordered by *precedes*, and thus are automatically separated by a full perspective to produce the desired animation.

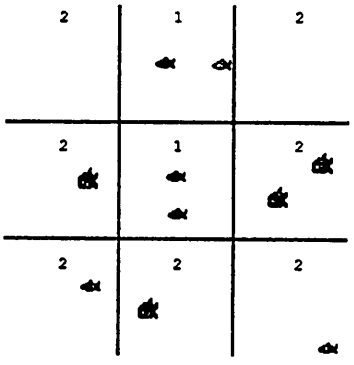


Figure 9: Actual execution of the sharks and fishes program. Processes at different time steps are executing concurrently.

5.3 Automatic Animations

Perspective views were developed to enhance the automatic animations provided by our "pattern-oriented" parallel debugger, Belvedere. In the following examples, we

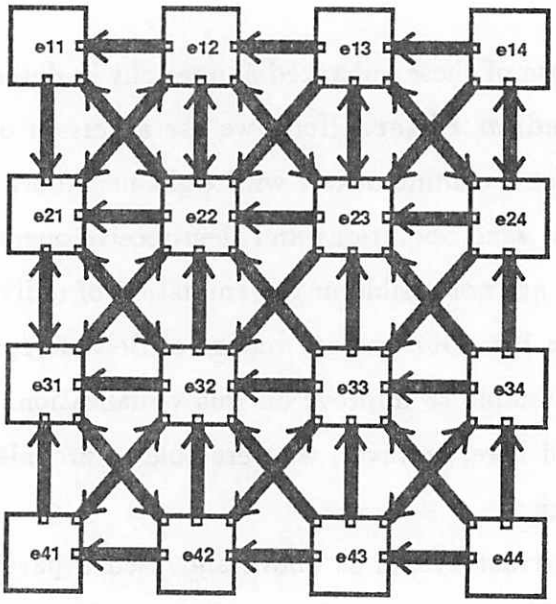
demonstrate the use of these enhanced animations in detecting program bugs.

Example 3. Median Filter. Here, we use a version of the median filter algorithm in which each process communicates with eight neighbors. On every iteration, the processes perform first send operations and then receive operations. The expected patterns of communication are not visible in the animation of individual send and receive events as shown in Figure 10a. Our original version of Belvedere, which supported only process perspectives, was unable to improve on this visualization. Using the more general perspectives presented here, however, we were able to provide animations that were useful in uncovering a bug.

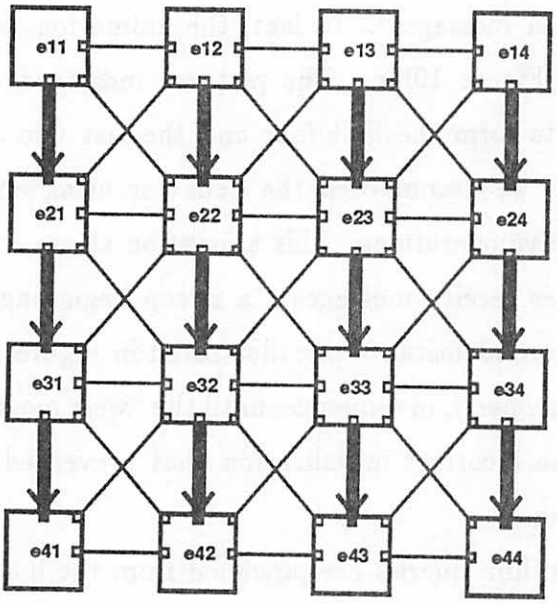
We defined abstract events as above and used a partial perspective including just send operations. The resulting animation of each iteration should show a sequence of eight distinct abstract events in the order that processes send messages: a counter-clockwise sweep beginning with the "north messages". In fact, the animation we saw included just two complete events as in Figure 10b-c. The pictures indicated a bug resulting in the loss of messages needed to form the first four and the last two of the eight abstract events. Pursuing the bug, we decomposed the behavior using another partial perspective that included just receive operations. This animation shows the the abstract events in the order that processes receive messages : a sweep beginning with the "south messages". The actual animation, identical to one illustrated in Figure 10b-c, showed us that messages were completing properly, in sequence, until the "west messages" failed. This led us to discover the bug, an incorrect initialization that prevented some east to west communication from occurring.

4. Dictionary Search[19]. In this algorithm, queries are pipelined from the host to a database of key-ordered records stored in a hypercube.[4] Queries are routed within the cube to the proper node using a binary search technique. In the animation of primitive send and receive events shown in Figure 11a, multiple queries are active simultaneously and it is difficult to understand whether or not each query is proceeding as intended. For debugging purposes, then, we wanted to view the progress of each query as it moved

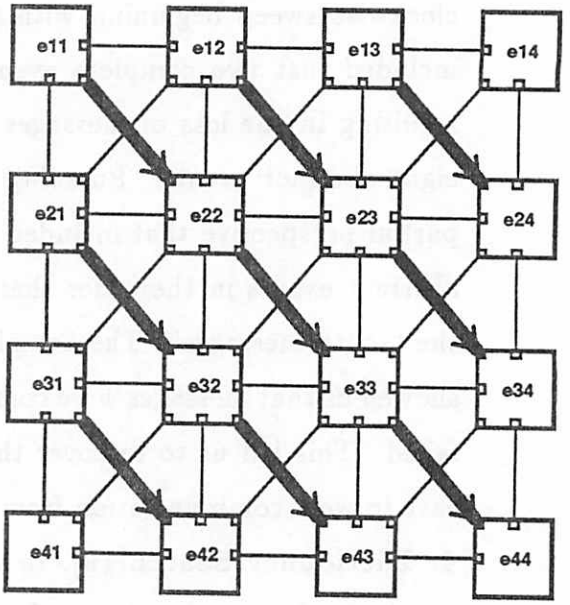
[4]Our implementation is a simplified version of the cited algorithm.



(a)



(b)



(c)

Figure 10: Median filter algorithm. Processes are shown as large squares, ports as small squares, channels as connecting lines. Animated events are highlighted; thick channels indicate outstanding SENDS, and arrowheads indicate a message in transit. Low-level communication events (a); logically sequential abstract communication phases (b) and (c).

through the cube.

We defined an abstract event to be all of the communication generated on behalf of a query. Figure 11b is a snapshot from the animation of a full perspective on these events; the path followed by each query is still obscured by concurrent queries. To separate the queries, we chose the set of send events on the host process as our perspective. This resulted in clear animations, as in Figure 11c-d, where queries are shown in the order that they were issued from the host and each query completes before the next begins.

The query depicted in 11c was processed correctly, but that of 11d was not. A query packet should be sent across a dimension of the cube at most once; if it is returned, we know the target node is not in that half of the cube. In Figure 11d, we can easily see a packet returning from the back plane of the cube twice. The extra communication uncovered a bug that occurred because the packets returning to the host were initially being sent in the wrong direction from some processes.

Again, ignoring some of the temporal ordering in the original execution sequence allowed the programmer to understand the behavior of his code in a more meaningful manner.

6. Conclusions

Visualizing the execution of parallel programs is difficult because low-level primitive events do not reflect the programmer's view of the system. Program behavior is better understood at a higher level in terms of user-defined, abstract events. Visualizations of abstract events, however, are often obscured by concurrency and asynchrony. Perspective views enhance visualization by reordering behaviors so that abstract events can be seen as distinct units. For program executions that cannot be consistently reordered, partial perspectives can be used to transform the system to one in which only user-selected temporal relationships are preserved. It is easy to create a variety of partial perspectives for the same behavior.

We have applied perspective views to the animations of parallel programs generated by our debugger and have found them to be effective in enhancing the depiction of high-

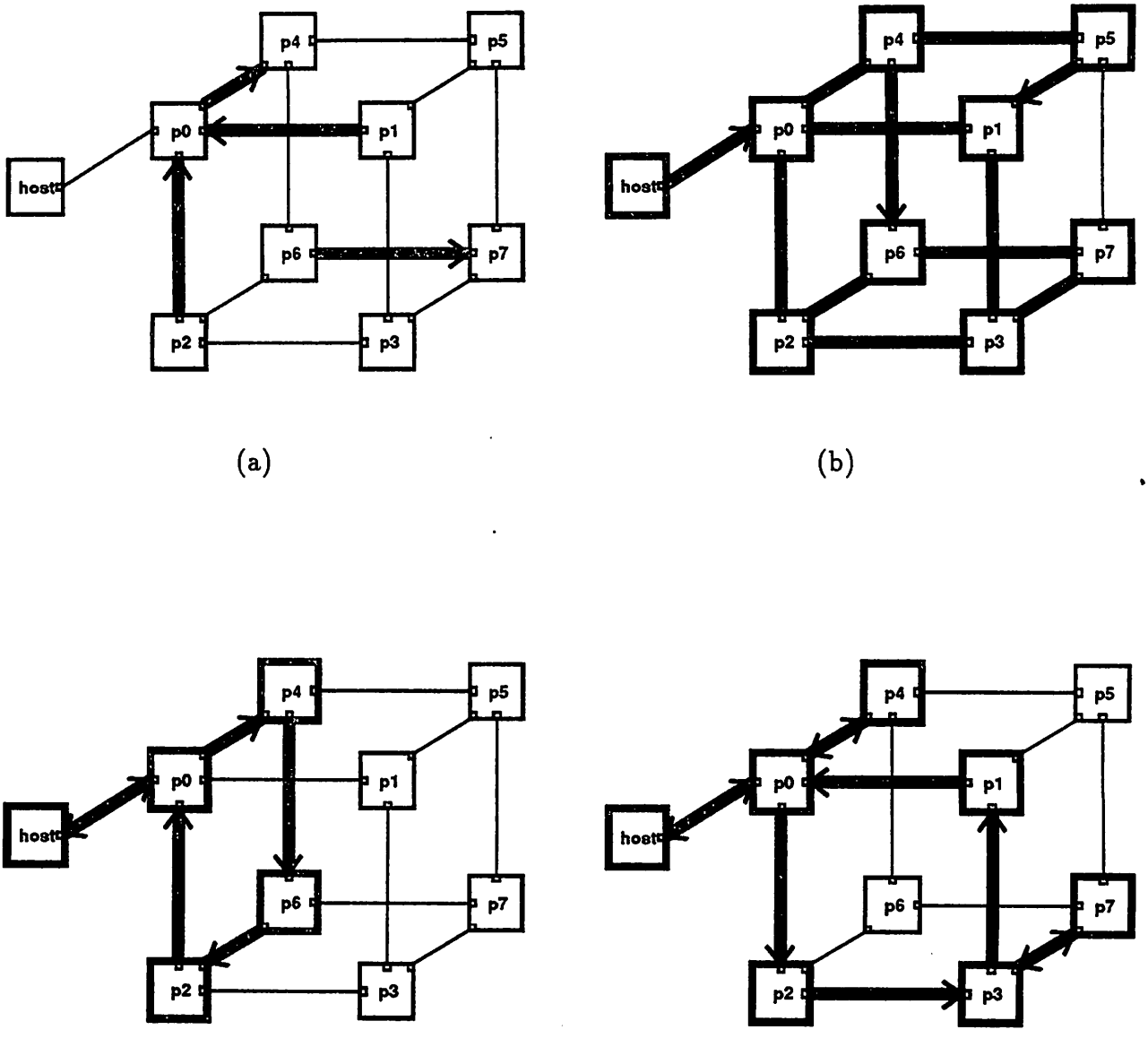


Figure 11: Snapshot of Belvedere's animation of the Dictionary Search. Low-level communication events (*a*); high-level events with several simultaneous search requests (*b*); a perspective view of high-level events showing the path taken by an individual request (*c*); erroneous communication between p3 and p7 (*d*).

level behaviors. These temporal remappings can also be used in conjunction with many other types of visualization tools, as we have demonstrated. We have found that their use significantly enhances the visualization of natural abstractions of parallel program behavior.

Appendix A:

Complexity of the Perspective Algorithm

The time needed to compute a perspective view is $O(N \log N + E)$ where N is the total number of events in the system and E is the number of dependencies between the events.

To arrive at this result we analyze each of the four steps of the algorithm in turn:

1. *Construct a graph representation of events and dependencies.*
   The constructed graph has N nodes and E edges representing events and the dependencies between events, respectively. Dependencies have three sources: (1) the connection between an abstract event and its subevents; (2) the implicit *happened before* dependencies between primitive events; and (3) the constraints supplied from the display system.

   Dependencies from each source are processed to produce edges in the graph as follows:

   (a) *Connect each abstract event to its immediate child events.*
   The record describing each abstract event contains pointers to the records representing the immediate children of the event. For each pointer we create a bidirectional edge in the graph between the abstract event and the corresponding child event. For a system with N events and E_1 pointers to child events the complexity of this step is $O(N + E_1)$, because we visit each abstract event and each pointer once. The total number of pointers is at most N^2 because there is at most one parent-child pointer between each pair of events.

   (b) *Add a minimal set of* happened before *dependencies.*
   In this step we add a set of edges sufficient to generate the *happened before* relation for the events in the perspective. Edges are only added between

events in the perspective. First, an edge is created between each event and the next event on the same process. Second, for each message an edge is created between the event on the sending process immediately prior to the sending of the message and the event on the receiving process immediately following the receipt of the message. For a system with E_2 messages the complexity of this step is $O(N + E_2)$, because there is one edge added to the graph per node and one edge added per message. The number of messages is at most $N/2$ because each message has two events associated with it, the send event and the receive event.

(c) *Add visualization dependencies.*

Each visualization dependency causes an edge to be added to the graph. Given E_3 dependencies, the complexity of this step is $O(E_3)$ because each dependency is processed once. The number of visualization dependencies is at most N^2 because there is at most one dependency between each pair of events.

The complexity of step 1 is the sum of the complexities of the substeps, $O(N + E_1 + E_2 + E_3)$. This expression can be simplified by expressing it in terms of the number of edges in the graph. The total of $2E_1 + E_2 + E_3$ is the number of times step 1 attempts to add an edge to the graph. This total can be at most $3E$, where E is the number of edges in the graph, because each of the three substeps adds a given edge to the graph at most once. Substituting $3E$ for $E_1+E_2+E_3$ gives a complexity of step 1 of $O(N + E)$.

2. *Find strongly connected components.*

Tarjan's algorithm[23] finds the strongly connected components of a graph in time $O(N + E)$.

3. *Assign depths to components*

The depth of each component is the length of the longest path to each component, counting along strongly connected components. To compute this length we first compact the graph so that each strongly connected component is represented by a

single node; the compacted graph is acyclic. The longest path to each component can then be computed by the critical path method, normally used for scheduling jobs ordered by a graph of dependencies. The critical path method schedules a job to start when the jobs it depends on have completed; if we assign each job a unit execution length, then the time a job is scheduled to start is the length of the longest path of dependencies to that job. The critical path method schedules jobs in time $O(N + E)$[5].

4. *Reorder events.*

   Reordering the events has two substeps. Events are first sorted by depth and then a pass over the events in sorted order assigns new timestamps. Sorting can be accomplished in time $O(N \log N)$. The pass to assign new timesteps is $O(N)$.

No algorithm step requires more than $O(N \log N)$ or $O(N+E)$ time, so summing these complexities and simplifying gives the complexity of the entire algorithm, $O(N \log N+E)$. To compute the complexity in terms of N we observe that the upper bound of E is the maximum number of unique edges in a directed graph, $2N^2$, because there can be at most two directed edges between each pair of nodes. Substituting this for E, the complexity of the algorithm in terms of N becomes $O(N^2)$.

REFERENCES

[1] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 11–22, 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[2] Peter C. Bates and Jack C. Wileden. Event definition language: An aid to monitoring and debugging complex software systems. In *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, pages 86–93, January 1982.

[3] Marc H. Brown. Exploring algorithms with Balsa-II. *Computer*, 21(5):14–36, May 1988.

[4] Alva L. Couch. *Seecube User's Manual.* Department of Computer Science, Tufts University, 1988.

[5] Shimon Even. *Graph Algorithms*, pages 138–139. Computer Science Press, Inc., 1979.

[6] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[7] Robert J. Fowler, Thomas J. Leblanc, and John M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[8] Vincent A. Guarna Jr., Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur. Faust: An integrated environment for parallel programming. *IEEE SOFTWARE*, pages 20–27, July 1989.

[9] Paul K. Harter, Dennis M. Heimbigner, and Roger King. IDD: an interactive distributed debugger. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 498–506, 1985.

[10] R. W. Hockney and C. R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Bristol, England, 1981. SOR algorithm as presented by Michael J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, pp. 140–142, (1987).

[11] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735–738, 1987.

[12] Alfred A. Hough and Janice E. Cuny. Initial experiences with a pattern-oriented parallel debugger. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 195–205, 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[13] Alfred A. Hough and Janice E. Cuny. Perspective Views: A technique for enhancing parallel program visualization. Technical Report 90-02, COINS Department, University of Massachusetts, Amherst, MA 01003, 1990.

[14] Wenway Hseush and Gail E. Kaiser. Data path debugging: Data-oriented debugging for a concurrent programming language. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 236–247, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[15] James E. Kelley, Jr. and Morgan R. Walker. Critical path planning and scheduling. In *1959 Proceedings of the Eastern Joint Computer Conference*, 1959.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[17] Leslie Lamport. The mutual exclusion problem: Part I–a theory of interprocess communication. *Journal of the Association for Computing Machinery*, 33(2):313–326, April 1986.

[18] Kathleen. M. Nichols and John. T. Edmark. Modeling multicomputer systems with PARET. *Computer*, 21(5):39–48, May 1988.

[19] Amos R. Omondi and J. Dean Brock. Implementing a dictionary on hypercube machines. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 707–709, 1987.

[20] William K. Pratt. *Digital Image Processing.* John Wiley and Sons, 1978.

[21] Steven P. Reiss. Graphical program development with PECAN program development systems. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, 1984. Proceedings also published as SIGPLAN Notices, 19(5), May 1984.

[22] David Socha, Mary L. Bailey, and David Notkin. Voyeur: Graphical views of parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 206–215, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[23] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972.

[24] Warren Teitelman. A tour through Cedar. In *Proceedings of the 7th International Conference on Software Engineering*, pages 30–41, March 1984.