ConcurrentMentor: A Visualization System for Distributed Programming Education *

Steve Carr, Changpeng Fang, Tim Jozwowski, Jean Mayo and Ching-Kuang Shene
Department of Computer Science
Michigan Technological University
Houghton, MI 49931
Email: {carr, cfang, trjozwow, jmayo, shene}@mtu.edu

Abstract ***The study of distributed systems is increasingly fundamental to a Computer Science curriculum. Yet, the design of applications to run over distributed systems is complex and mastery of fundamental concepts is challenging for students. In order to assist in distributed systems instruction, we have developed ConcurrentMentor, a visualization system for distributed programming. This system reveals the behavior of a distributed program and its underlying communication protocols while the program executes. Input to the visualization system is generated by an accompanying communication library that closely follows abstractions of communication found in distributed systems literature. No program instrumentation is required.***

Keywords: Concurrent programming, visualization, education

1 Motivation

Distributed programming is increasingly fundamental to undergraduate Computer Science education [1]. Instruction in this area is correspondingly moving ever earlier into the undergraduate curriculum. Yet, the design of applications to run over distributed systems is significantly more complex than the design of applications that run on a single system. Mastery of fundamental concepts is challenging to students.

We have developed a visualization system, ConcurrentMentor, that helps students understand fundamental concepts of distributed systems by illustrating characteristics of an executing program. The visualization system gets input from a communication library. The library supports interprocess communication using the abstraction of synchronous and asynchronous channels between processes, an abstraction that closely follows that found in the distributed systems literature [2]. Program instrumentation is not required.

The visualization system provides several views of an execution. The **Space-time Diagram** view depicts the causal relationships among events of interest. The **History Graph** view depicts the states through which a process passes, e.g., "blocked on synchronous message send", "running", etc. The **Process Topology** view depicts the channels that exist among system processes, and the **Statistics** view displays characteristics of the message traffic that flows along these channels.

The message passing library and distributed program visualization are part of a larger system designed to present concepts of concurrent programming using a unified approach. The communication classes and accompanying visualization are available to threaded applica-

*This work supported in part by the National Science Foundation under grants DUE-9752244 and DUE-9952509. The fourth author was also supported by National Science Foundation CAREER award CCR-9984862.

tions as well. Further, additional library functions, and accompanying visualizations, specific to threads, e.g., monitors and semaphores, are available [3, 4]. Support for data-parallel programming is under development.

2 Related Work

Visualization systems generally either perform *algorithm animation* or *program behavior visualization*. The former animates the execution of an algorithm, while the latter displays the execution behavior of a program. The visualization can be either *real-time* or *post-mortem*. A post-mortem system saves events that occur during an execution and plays them back with another program. A real-time system generates the displays while the program executes.

Hartley presented an approach for visualization in which a user program saves data about an execution to a file. XTANGO [5] is used to play back an execution when the program completes. Naps and Chan incorporate parallel program animation into Multi-Pascal [6]. One of the most well-known systems for parallel program animation is Stasko's PARADE, which is based on POLKA [7]. Cai presents a system for OCCAM programs [8].

Most freely available and well-known systems are post-mortem, including XTANGO, POLKA, and PARADE. These systems have the advantage that all data relevant to the execution has been saved and can be replayed and analyzed at any time. They have the disadvantage that a large volume of data must be saved to support the replay, the program must be explicitly instrumented, and the execution runs to completion before the replay begins.

The system presented in this paper allows visualization of a program during execution. This is a natural and gratifying approach for students, whose program analysis experience is based upon program debugging systems that operate on an executing program. Students are also pleased by an expeditious response from the software they run. Use of the system does not require explicit instrumentation of user programs. Support is also provided for recording visualization data, at the same time it is displayed, for playback at a later time. User programs are written in C++.

3 Communication Library

The communication library abstracts communication at two levels: channel and topology [2]. Channels or topologies are objects within the user program. The implementation of each object provides input to the visualization system invisibly to the user. The channel and topology classes are described in turn below.

Channels The goal of the channel classes is to provide an abstraction of communication that ties closely to that encountered in the literature. Two channel types have been implemented; both types are bidirectional. The first class is a synchronous one-to-one channel. Along this channel, both send and receive are blocking [9].

The second class implements an asynchronous one-to-one channel. Along these channels, sends are non-blocking; receives can be blocking or non-blocking [9]. The asynchronous channel class supports optional message loss. Loss is specified, when the channel is created, as either a value between zero and one, or via an integer function with a single integer input. Messages are dropped immediately prior to the point at which they would be sent along a totally reliable channel.

Vector time is maintained on behalf of user applications. Vector time is used to determine the *happened-before* relation [10] among events that occur within a distributed computation [11]. Users can query, and increment the local component of, the current local vector time within an application.

Figure 1 depicts code that creates an asynchronous channel between processes numbered zero and one.[1] Each process sends and then receives a message on the channel. Messages

[1] Identifiers are assigned by a control process when the processes comprising an application are spawned.

```
char msg[]="False pearls before real swine";
VectorClock VectorTime;
channel1 = new AsynOnetoOneChannel(1,
                    myID,dropSome(myID));
channel1.Send((void *)msg,sizeof(msg));
channel1.Receive((void *)msg,sizeof(msg));
VectorTime[myID]=VectorTime[myID]+1;
VectorTime.Print();
```

(a) Process Zero

```
char msg[]="1,1,2,3,5,8,13,21,34";
VectorClock VectorTime;
channel0 = new AsynOnetoOneChannel(0,
                  myID,0.5);
channel0.Send((void *)msg,sizeof(msg));
channel0.Receive((void *)msg,sizeof(msg));
VectorTime.Print();
```

(b) Process One

Figure 1: Message Exchange along Asynchronous Channel

along the channel from process zero to process one will be dropped whenever the function `dropSome()` (defined elsewhere) evaluates to a value less than one. Fifty percent of the messages from process one to process zero will be dropped. Process zero explicitly increments the local component of its clock. (This will advance the local component beyond the value that results from the previous message receive event.) Both processes print the current value of the vector clock.

Topology Topologies facilitate creation of multiple channels via instantiation of a single class. Standard topologies, derived from the topology class, are also provided including: fully-connected, star, linear array, ring, grid, and torus. The recipient of a message within a topology must be directly connected to the message sender. Topologies are constructed with reliable asynchronous channels.

```
theGrid = new Grid(ROWS,COLS,myID);

if ((myID % COLS) != 3)
  theGrid.send(RIGHT,(void *)&myID,
               sizeof(myID));
if ((myID % COLS) != 0)
  theGrid.send(LEFT,(void *)&myID,
               sizeof(myID));
if (myID >= COLS)
  theGrid.send(UP,(void *)&myID,
               sizeof(myID));
if (myID < (ROWS-1)*COLS)
  theGrid.send(DOWN,(void *)&myID,
               sizeof(myID));
```

Figure 2: Exchange of ID Among Grid Neighbors

Figure 2 depicts code for exchange of identifiers among all neighbors in a grid topology. The macros `RIGHT,LEFT,UP,DOWN` are defined within the system. Similar macros are defined as appropriate to a given topology.

Within the topology class, and each standard topology, the methods `Send(), Receive(), Broadcast(), Scatter(), Gather()`, and `Reduce()` are provided. `Scatter, Gather,` and `Reduce` currently operate on one-dimensional arrays (consecutive storage). Support for operations on some non-consecutive storage in two-dimensional arrays (e.g., sub-matrix) is under development.

A topology editor allows rapid creation of complex topologies via a graphical interface [12]. The editor outputs a file containing specification of a class derived from the topology class. This file can be included by the user in her code to easily create the constructed topology. The derived class supports broadcast, scatter, gather, and reduce functions for each custom topology.

4 Visualization

The visualization system intends to help student visualize the execution of distributed programs. To enhance its utility, the software runs

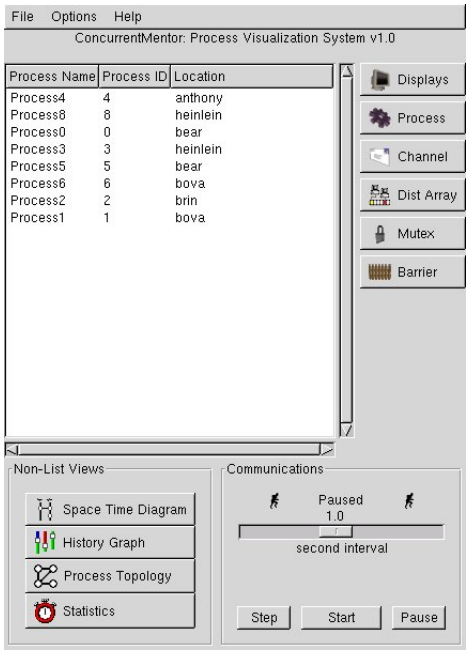

Figure 3: Main Window

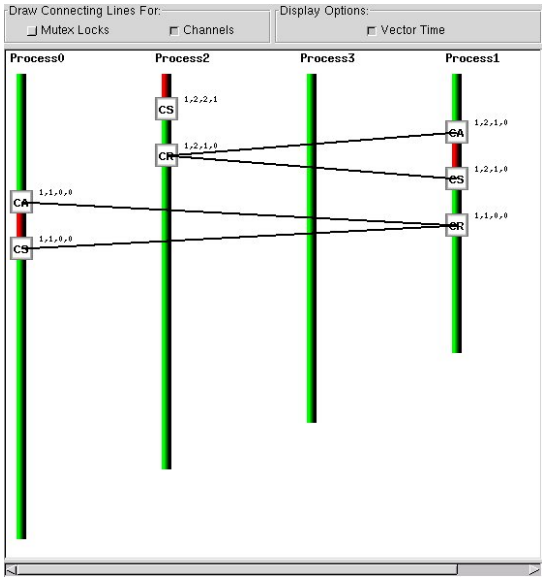

Figure 4: History Graph View

over Linux and Solaris. Use of the visualization system, when using the communication library, is optional. Each of the main visualization system windows is described in turn below.

Main Window The main window, depicted in Figure 3, functions in three ways. First it provides a way to navigate through the different windows of ConcurrentMentor. Windows that can be reached through the main window include: **Space-time Diagram**, **History Graph**, **Process Topology**, **Statistics**, **Process**, and **Channel**. The first four may be reached by their corresponding buttons in the lower left portion of the window. These views are described in more detail below. The remaining buttons, in the upper right portion of the window, lead to display of a text list of the currently created elements within the main window.[2] Figure 3 contains a list of created processes, from activation of the **Process** button. Selection of a list item causes a corresponding window to open displaying more detailed information about the chosen element. For example, selecting the text list item associated with a particular channel will open a window that displays messages that have been sent along, or that are currently on, the selected channel.

The main window is also used to control the display of events throughout the system. The **Communications** panel, in the lower right corner of the main window (Figure 3), allows one to step through the events of the program, playing the incoming events at different speeds, or to pause the execution.

Finally, the main window can be used to save and play back a particular execution of the program. If, for example, a student finds a deadlock within an execution, the visualization data may be saved and then re-played.

History Graph View The **History Graph** view depicts states and events of interest through which a process passes during its lifetime. A history graph view for a four process execution, using synchronous message passing, is depicted in Figure 4. Each process has a corresponding event line with events

[2]The **Displays**, **Dist Array**, **Mutex**, and **Barrier** buttons invoke functionality that is currently under development.

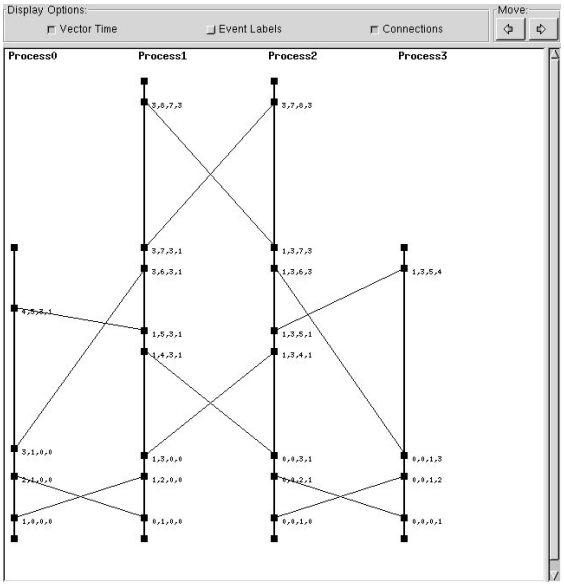

Figure 5: Space-time Diagram View

of interest noted by tags. The color of the event line for a process reflects the process' state. Events are depicted, one per horizontal line, as they are received by the visualization system. Selection of an event brings up a window with the code line that generated the event highlighted.

Events that currently appear on the history graph include initiation of a message send and completion of a message receive. When a process blocks on a synchronous send, the event line changes from green to red. The event line changes from red to green when the send is complete. The vector time associated with events can optionally be displayed in the process history view.

Messages are optionally depicted via a line from the event that corresponds to initiation of a send with the event that corresponds to completion of a message receive. When message passing is synchronous, an acknowledgment message is depicted on the history graph, in addition to the message that travels from originator to recipient.[3]

[3]A receive is complete upon send of an acknowledgment; a send is complete upon receipt of the acknowledgment.

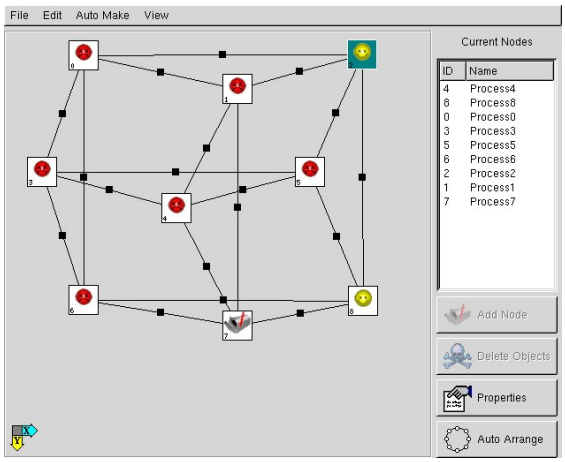

Figure 6: Process Topology View

We have found that the message passing programs of inexperienced students often contain mismatched sends and receives. The history graph helps a student to detect resulting deadlocks, as well as the offending lines of code.

We are currently adding support for additional events including: request for a critical section, acquisition of a critical section, release of a critical section, entry into a barrier, exit from a barrier, and entry into, and exit from, broadcast, scatter, gather, and reduce.

Space-Time Diagram View The **Space-time Diagram** depicts the causal relationship among events in an execution. (See Figure 5.) Event lines, similar to those for the process history view, depict the events that occur within each process. Events in this view are limited to message send and message receive. Vector times may optionally be displayed within the space time diagram. Message send and receive events may also optionally be connected.

Students can explore the causal relationship among events by sliding events up and down the event lines. Event height is (only) restricted so that the causal order is not violated. (More precisely, if $a \rightarrow b$, where '$\rightarrow$' denotes Lamport's *happened-before* relation [10], then the height of event b must be greater than the height of event a.)

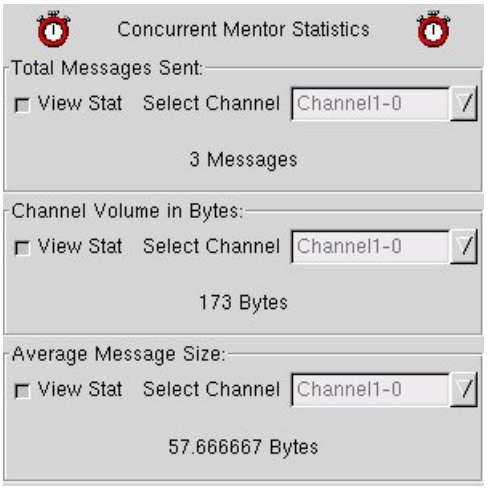

Figure 7: Statistics View

Process Topology View The **Process Topology** view depicts the channels among processes. Figure 6 depicts a topology window for a torus containing nine nodes. An icon appears for each process in the system. An icon reflects the current state of the corresponding process: processing, sending message, receiving message, or terminated. A user may move the icons to create the best possible depiction.

Statistics View The **Statistics** view displays characteristics of the message traffic that has flowed across a particular channel (Figure 7). The top third of the window displays the total number of messages sent along a channel, the middle third displays the total number of bytes sent along the channel, and the lower third displays average message size in bytes. A statistic is displayed for the selected channel, and a different channel may be selected for each statistic.

5 Conclusions and Future Work

We have developed a visualization system that reveals the execution behavior of distributed programs. The system takes input, invisibly to the user, from an included communication library. The library provides two levels of abstraction, channel and topology, for the communication that occurs among processes and threads. A topology editor allows development of custom topologies via a graphical interface.

We are currently adding class libraries for distributed arrays, as well as accompanying visualization. Comprehensive, detailed information on our work is available at `http://www.cs.mtu.edu/~shene/NSF-3/index.html`.

References

[1] ACM. Computing Curricula 2001 (Steelman Draft, August 1, 2001). `http://www.acm.org/sigs/sigcse/cc2001/steelman/`, 2001.

[2] Steve Carr, Changpeng Fang, Tim Jozwowski, Jean Mayo, and Ching-Kuang Shene. A communication library to support concurrent programming courses. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 360–364. ACM Press, 2002.

[3] Michael Bedy, Steve Carr, Xianglong Huang, and Ching-Kuang Shene. A visualization system for multithreaded programming. In *Proceedings of the 31st Annual SIGCSE Technical Symposium on Computer Science Education*, pages 1–5, Austin, TX, March 2000.

[4] Steve Carr and Ching-Kuang Shene. A portable class library for teaching multithreaded programming. In *Proceedings of the Fifth Annual Conference on Innovation and Technology in Computer Science Education*, pages 124–127, Helsinki, Finland, July 2000.

[5] Stephen J. Hartley. Animating operating systems algorithms with XTANGO. In Daniel Joyce, editor, *Proceedings of the 25th Technical Symposium on Computer Science Education*, volume 26(1) of *SIGCSE Bulletin*, pages 344–348, New York, NY, USA, March 1994. ACM Press.

[6] Thomas L. Naps and Eric E. Chan. Using visualization to teach parallel algorithms. In Daniel Joyce, editor, *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, volume 31.1 of *SIGCSE Bulletin*, pages 232–236, N. Y., March 24–28 1999. ACM Press.

[7] John T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, Graphics, Visualization, and Usability Center Georgia Institute of Technology, Atlanta, GA, January 1995.

[8] Wentong T. Cai, Wendy J. Milne, and Stephen J. Turner. Graphical views of the behavior of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):223–230, June 1993.

[9] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*. Addison-Wesley, third edition, 2001.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[11] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[12] Steve Carr, Ping Chen, Timothy R. Jozwowski, Jean Mayo, and Ching-Kuang Shene. Channels, visualization, and topology editor. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 106–110. ACM Press, 2002.