ADVANCES IN PARALLEL DEBUGGERS:
NEW APPROACHES TO VISUALIZATION

Technical Report CSE-89-012

Cherri M. Pancake
Sue Utter

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

December 29, 1989
Advances in Parallel Debuggers:  
New Approaches to Visualization

Sue Utter  
Cornell National Supercomputer Facility  
Cornell University  
Ithaca, New York 14853  
email: psu@cornellf.tn.cornell.edu

Cherri M. Pancake  
Department of Computer Science and Engineering  
Auburn University  
Auburn, Alabama 36849  
email: pancake@ducvax.auburn.edu

Abstract

Programmers who write serial code can take advantage of a wide range of debuggers whose features draw on more than three decades of research in how humans design and verify computer programs. Debugging tools are generally designed to facilitate a cyclic, cause-effect approach to error detection and correction. This strategy is effective in serial environments, where program execution is repeatable; in parallel settings, however, many traditional techniques are rendered useless by the effects of nondeterminism.

This essay reviews recent research in parallel debuggers. In particular, it addresses the difficulties involved in developing effective schemes for portraying program behavior. A debugger's visualization system encompasses representational and interpretive concepts as well as graphical display techniques. The goal of visualization design is to furnish relevant views of program execution as clearly and succinctly as possible. Inadequate or inappropriate visualization systems jeopardize the usefulness and acceptability of debugging tools by making it difficult for the user to draw accurate inferences about the relationship between program code and run-time errors.

Industrial, governmental, and academic research efforts in debugger visualization are traced, with emphasis on the evolution of new strategies and techniques. Three basic approaches are identified — corresponding to functional, structural, and behavioral perspectives — and examples of recent developments are presented from each area. This is followed by a summary of the problems still confronting visualization designers.

Keywords: parallel debuggers, debugging, parallel programming, programming tools, program behavior, visualization
## Contents

1. Introduction .................................................. 2  
2. Terminology .................................................. 6  
3. Overview of Parallel Debugging .............................. 7  
   3.1. Observing Program Behavior ............................ 7  
   3.2. Modifying Program Behavior ........................... 9  
   3.3. History of Parallel Debuggers ........................ 9  
4. New Methods for Visualization .............................. 12  
   4.1. Functional Visualization Schemes ....................... 12  
   4.2. Structural Visualization Schemes ....................... 13  
   4.3. Behavioral Visualization Schemes ....................... 19  
5. Visualization Systems from the User Perspective ......... 20  
   5.1. Portraying the Dimensionality of Program Behavior 21  
   5.2. Reducing Display Complexity .......................... 25  
   5.3. Supporting Conceptual Models ........................ 26  
   5.4. Interface Mechanics .................................. 28  
6. Conclusions .................................................. 29  
References ..................................................... 33
1. Introduction

Program development involves a transformation from the programmer's mental model of problem solution to a second model in the form of workable code. Correctness, consistency, and reliability all derive from how effective that transformation is. The term bug refers to a situation where the process is inaccurate, with the result that program behavior during execution diverges from what was intended in the mental model. The central role of a debugger is to facilitate the isolation and correction of the factors responsible for such deviations.

Programmers who write serial code can take advantage of a wide range of debuggers whose features draw on more than three decades of research in how humans design, verify, and correct computer programs. Debugging activities typically proceed in cyclic fashion, as illustrated in Figure 1. When an error is recognized, the programmer postulates a probable cause, employs information gathering facilities (hand-coded instrumentation or automated tools) to confirm or reject the hypothesis, alters the program accordingly, and re-executes it to determine if the repair was effective. This cause-effect approach is appropriate because the execution of a serial program is deterministic: if identical inputs are supplied, subsequent program runs will repeat the same steps and yield the same results.

When programmers develop code for parallel or distributed systems, they usually start by extending the serial programming techniques with which they are already familiar. Parallelism is seen as the overlapping execution of a number of serial programs, each proceeding deterministically on a separate processor. This “extended serial view” of parallelism is not a reliable approach to program development, however, because it leads to confusion and misconceptions about program behavior. The results of parallel programs, unlike their serial counterparts, are not implicitly reproducible. When two or more activities are carried out in parallel on multiple processors, the order in which they finish is nondeterministic.

This phenomenon has considerable impact on debugging activities, since most tools and techniques that support the cyclic debugging paradigm rely implicitly on the repeatability of program execution. The nondeterminism latent in parallel codes can cause bugs to appear intermittently from one program run to another in an unpredictable way. As there is no
Figure 1. The cyclic debugging process.

guarantee that the same bug will occur again upon re-execution, the programmer must begin debugging by stabilizing the error so that it can be provoked consistently (see Figure 2). In serial programs, this usually is accomplished by repeating execution with identical input values, but parallel bug stabilization can be a formidable task.

Debugging tools traditionally add instrumenting statements or software hooks to the program in order to monitor its run-time behavior. In parallel programs, the insertion of code to trace execution may actually decrease the chance that an error will be repeated, since the new instructions alter normal timing sequences. This probe effect\textsuperscript{4} can disrupt stabilization activities by masking the presence of errors. The perturbances introduced by debugging statements may also provoke new errors that distort or counteract the effects of the original bug; the term \textit{Heisenbugs}\textsuperscript{5} has been coined for these, reflecting application of the Heisenburg Uncertainty Principle to debugging.
Figure 2. Effects of parallelism on the debugging process (shading indicates susceptibility to the effects of nondeterminism).

Subsequent debugging activities are also more complicated in a parallel programming environment (see Figure 2). The isolation of a bug to a particular program region can be especially problematical. The programmer often needs to juggle mental images of fifty or
more interacting units to infer which ones are most likely to be involved in the error. Even after a candidate region has been identified, it may be difficult to pinpoint possible causes. The correct interpretation of parallel control flow errors (deadlock, race conditions, or other access anomalies), for example, generally requires a considerable level of expertise.

A complete discussion of the problems in recognizing and correcting parallel program errors is beyond the scope of this article. Research efforts directed to the development of transparent debugging mechanisms (i.e., non-intrusive actions that will not affect the normal outcome of execution) have been described in depth in other studies\textsuperscript{6,7}. General treatments of debugging tools and user interface mechanisms have also appeared elsewhere\textsuperscript{8,9,10}. These topics will not be covered here. Instead, attention will focus on one of the most controversial — and often neglected — aspects of parallel debugger design, the way program behavior is presented graphically to the user.

The representational and interpretive concepts which form the basis for graphical displays, together with the screen configurations actually employed, are referred to collectively as a debugger's visualization system. Since the study of program behavior is the primary motivation in using a debugger, the goal of visualization techniques is to furnish relevant views of behavior as clearly and succinctly as possible. When the debugger's representational scheme is incomplete or inconsistent with the user's notion of program structure, it is unlikely that accurate inferences will be drawn about the relationship between program code and run-time errors. Inadequate visualization systems can therefore jeopardize the usefulness and acceptability of debugging tools.

Several difficulties must be overcome in any scheme for visualizing parallel programs. Paramount among these is the issue of display complexity. Even serial debuggers are frequently criticized for the large amount of irrelevant data they produce\textsuperscript{11}; when multiple interacting processes are involved the quantity can grow at an alarming rate. Special techniques must be employed to discard or ignore repetitive and uninteresting data. Second, where the visualization of serial execution essentially involves two dimensions — work accomplished over time — parallelism adds another, corresponding to the distribution of work over a processor topology. The nature of output media means that a parallel debugger must make use of two-dimensional representations to express three-dimensional concepts. Yet another problem is the extent to which debuggers rely on information supplied by the programmer, since the need to encode specialized commands as part of the debugger interface introduces a new source of potential error. Lastly, the design of a visualization system often must compensate for biases imposed by the parallel programming paradigm in which it will be used. These include restrictions associated with the decompositional
methods used for algorithm development, the programming language structures used in implementing the algorithm, the translators responsible for preparing the program for execution, and the architecture where program execution takes place.

2. Terminology

A minimal set of terms will be used in the discussion which follows; their definitions have been generalized from those of Hwang and Briggs. A process is understood to be a sequential unit of work; correspondingly, processors are the physical machines on which processes execute. A single computer incorporating multiple processors is a multiprocessor, as opposed to a multiple computer system, which combines autonomous computers, possibly heterogeneous. A multiprocessor is said to be tightly-coupled when its processors communicate through common access to a shared memory. This is contrasted with loosely-coupled systems, where the lack of any shared memory necessitates communication through some form of message-passing. The latter term is also applied to multiple computer systems allowing communications among the autonomous components.

A parallel processing environment is one wherein activities (specifically, processes) are carried out simultaneously on a multiprocessor, while distributed processing involves simultaneous activities on a multiple computer system. Here, a parallel program is understood to include two or more portions that execute simultaneously, on either a parallel or a distributed system. The term concurrent is sometimes used interchangeably in this context. The concept of concurrency, however, is usually extended to include the simulation of parallelism on uniprocessor machines through the use of multiprogramming. Since concurrent programs are not always sensitive to reproducibility and timing problems, our discussion assumes the stricter requirements of true parallelism.

Parallel debugging refers to the isolation and explanation of behavioral errors in parallel programs. Correspondingly, a parallel debugger is a software tool to facilitate parallel debugging. In its broadest sense, the term can be applied to virtually all program analysis tools, from type-checking compilers to portability verifiers. Here, however, treatment will be restricted to software which analyzes the run-time behavior of parallel programs.

It should be noted that a distinction must be drawn between the tool and its target programs. Some debuggers implemented on parallel or distributed systems are only capable of analyzing serial targets (e.g., DICE, Blit); as these are not in a strict sense parallel debuggers, they will be omitted from discussion. Another class of software tools interactively guides the parallelization of serial programs (e.g., PED, PAT). Although it could be
argued that the information provided by these program restructurers could be used in debugging activities, they are not pertinent here because they do not make use of run-time information. Finally, some publications make reference to so-called “performance debuggers,” tools intended to monitor run-time behavior and identify areas which would be promising candidates for efficiency improvement efforts. In this case, the tools are also useful in the detection and correction of program errors, so they will be included when appropriate.

3. Overview of Parallel Debugging

To correctly identify the cause of a programming error, a programmer must be able to investigate the events leading up to its occurrence. Traditionally, debugger systems have supported investigative efforts by providing services for examining, and perhaps altering, “program state.” The approach is based on the notion that program execution may be viewed as a sequence of discrete operations; consequently, program behavior may be characterized as a series of transitions from one “state” to another. Spirk defines an execution state as “a logical interval of task activity during which the behavior of interest remains statistically invariant.” As shall be seen, much of the variation in debugger output derives from the ways in which designers have interpreted the concept of program state.

The services provided by a debugger fall into two major categories: (1) facilities for observing state changes; and (2) facilities for modifying or controlling state changes. Although all debugging tools support program observation in some form or another, mechanisms for altering behavior are quite limited, even when the targets are serial programs.

3.1. Observing Program Behavior

The observation of program behavior requires techniques for monitoring the state of the program and recording information at arbitrary points during execution. The data is used to construct a trace, or execution history, suitable for post-mortem analysis. Runtime data is gathered in a series of synchronous “snapshots,” occurring at specific time intervals (referred to as checkpoints), or tied to the occurrence of asynchronous events (dubbed tracepoints when keyed to the flow of execution control, or watchpoints if applied to data storage values). The run-time overhead for gathering and storing trace data can be prohibitive even for serial programs, increasing execution time by as much as 10 to 40 times. Monitor
overhead is also the primary culprit in the transparency problems of parallel debuggers. In serial programs monitoring produces delays in the execution stream, but it does not change the sequencing of events. Parallel programs, on the other hand, are extremely sensitive to such delays. A minor change in the timing relationships of multiple execution streams can alter the overall sequence of activities radically.

Another difficulty posed by monitoring is the potential for generating copious amounts of information. In a serial environment, a snapshot taken at any moment reflects the state of the program as a whole, so the upper limit on output volume is some function of active data space requirements and the number of snapshots made. In parallel processing, however, the notion of program state becomes fuzzy. To capture the global state of a parallel program, snapshots of the parallel processes executing on all processors must be collected simultaneously and correlated with information on temporarily inactive processes, as well as any ongoing interactions that have not yet been reflected in the individual process states. Not only are storage space problems compounded, but the added memory traffic may cause further interference with normal execution sequencing.

Even the availability of transparent collection mechanisms and unlimited storage do not alleviate the problems in monitoring parallel program execution. The recording of global state information in a parallel system presupposes some general mechanism for coordinating the control of all component processors. Although this might be attainable in tightly-coupled multiprocessors, it is virtually impossible in loosely-coupled multiprocessors and multiple computer systems. The lack of a system-wide clock facility presents a primary obstacle. In loosely-coupled systems, each processor keeps a local clock, whose time may vary significantly from those of other processors within the system. Consequently, it becomes difficult or impossible to state that one event occurred before another in some global sense.

An alternative to trace-based monitoring is to allow the programmer to insert arbitrary breakpoints that trigger the interruption of normal execution sequencing so that program state may be examined interactively. This approach offers some respite from the storage overhead associated with full tracing by eliminating the need to record lengthy sequences of state changes. Instead, the current state is measured, analyzed, and displayed on the fly. It is important to note that the capability of exploring program state at specified points in no way alters the fact that monitoring is (at least conceptually) a passive process: the user has no real interaction with the executing program.

One difficulty with the migration of breakpointing techniques to parallel programs is that the concept of a breakpoint has no clear meaning. Should only the processor
encountering the breakpoint halt, or should all program activity be stopped? Halting a single processor reflects the state of only one individual process, not that of the program as a whole. On the other hand, the ability to halt all processors necessitates a global control mechanism that is capable of stopping and re-starting all processors simultaneously.

3.2. Modifying Program Behavior

Debuggers can provide direct control over run-time behavior through facilities that allow the programmer to halt execution, examine the current program state, change some aspect of that state, then restart execution. The most prevalent technique is the use of breakpoints to interrupt normal sequencing, followed by operations that alter the values of program variables. User input thus determines the progress and future states of the program.

Since program state is presented only on demand, this approach ameliorates the data collection and storage overhead of monitoring. As in breakpointing, the current state must be computed on the fly. The ability of the user to alter the displayed state, however, means that the debugger must be capable of controlling run-time behavior as well as observing its progress. Current serial debugger technology limits modifications to data storage areas; although data values may be altered at will, flow of control (i.e., the instruction stream) remains immutable.

Unfortunately, the concept of global data values loses some of its precision in the parallel environment. If, for example, two processes are racing to alter a shared variable, should user intervention apply only to the immediate value, or should it persist long enough to override the competing accesses? The problems inherent in parallel breakpoints affect this situation as well: should the user be informed that execution halted just after one access was made but just prior to another access, or perhaps be allowed to prolong execution for one more access so that the altered value will persist? A natural extension of this idea would be to allow the user to specify differential rates of execution for individual processes and/or processors, thereby affecting the outcome of race- and deadlock-prone situations. These and other issues tied to behavioral modifications have not been answered satisfactorily for parallel execution environments.

3.3. History of Parallel Debuggers

General opinion holds that the availability of commercial software for parallel processing seriously lags behind the introduction of new hardware. Nowhere is this more
evident than in the field of parallel debugging. The few companies currently offering such products have elected to adapt serial techniques to multiple-processor systems, rather than developing new approaches\textsuperscript{8,23,24,25,26,27}. Where a serial debugger allows the examination or manipulation of program state within a single instruction stream, the parallel tool provides analogous capabilities for a series of processes. This is typically accomplished through the use of process identification numbers as qualifiers on commands that otherwise duplicate standard serial debugger operations. In some cases, windowing mechanisms allow the simultaneous display of two or more instruction streams (each window effectively defines a debugger for one process), but there is no real attempt to correlate their activities.

Because this approach limits its support to the "extended serial" model of parallelism, it has a number of shortcomings. First, debuggers are typically invoked after the appearance of a bug, but the non-reproducibility of parallel programs gives rise to the possibility that the original error will not re-occur during the debugging run. Second, standard debugging control mechanisms are highly intrusive. By perturbing the normal execution sequence, they run the risk of masking the bug and/or introducing new errors. Furthermore, commercially available systems neglect support for viewing how the activity of one process influences that of others, at best providing minimal lists of messages or events (e.g., the debugger for Intel's iPSC\textsuperscript{R} hypercube, BBN's gist\textsuperscript{R}, and Encore's Parasight\textsuperscript{R}). Since all information is provided at a low level, it is difficult for the programmer to abstract collections of activities into logically meaningful events. This places the burden of conceptualizing global program behavior on the user — a task that can be awkward for serial programs, but is extremely cumbersome when multiple processes are involved.

A number of recent, experimental systems have attempted to alleviate some of the limitations associated with "extended serial" debuggers. The most common improvements are the addition of facilities for examining interprocess communications\textsuperscript{28,29} and the implementation of logical clock mechanisms to yield more consistent global control\textsuperscript{30}. In addition, there has been a general movement away from the arbitrary observation activities of breakpoint-driven monitors and toward the more structured trace-based systems.

As noted previously, monitors for parallel programs introduce a variety of technical problems, primarily due to the lack of global clock and control mechanisms. Specific strategies for managing these issues comprise the bulk of current research in parallel debugging. Proposed solutions include the use of external hardware monitors\textsuperscript{31}, the implementation of logical clocks\textsuperscript{32}, the limiting of debugger activities to block boundaries where interprocess synchronization occurs\textsuperscript{33}, and the recording of order-of-access information on shared variables in lieu of the actual values assigned\textsuperscript{34}.

10
A common tactic for reducing trace size is to increase the granularity of the information recorded. Rather than viewing execution as a series of program state transitions at the level of individual statements, most recent systems attempt to elide sequences of statements into behaviorally relevant units. In general, the concept of a primitive unit is associated with process interaction. Another new technique, designed to facilitate the application of cyclic debugging methods in a parallel environment, strikes a compromise between the need for full execution information and the intrusiveness of monitoring mechanisms. The term "replay" has been coined for this approach, which monitors a program run with minimal interference and subsequently re-executes the program under the guidance of the monitor record and identical inputs. This assures an equivalent execution sequence, thereby allowing the use of more intrusive — but also more robust — information-gathering techniques during re-execution. There also appears to be a growing interest in the concept of continuous monitoring systems. Many developers feel that any effective solution to the problems of parallel debugging will involve a trace-based system with the capacity to monitor program execution on a continuing basis, even during production runs. Such measures can normalize the associated overhead from run to run, yet still guarantee that the events leading to any execution error are captured.

The general availability of high quality graphics has added impetus to the increasing popularity of trace-based debuggers. The history produced by a monitor need not be presented to the programmer in its original form. Debuggers that exhibit trace data in textual form, by listing variable values or call sequences, rely on the user to extrapolate the information needed to derive a mental model of program behavior. Alternatively, the sequence of program events recorded by the monitor can be analyzed and displayed in some more meaningful form. This concept is referred to as program visualization, defined by Baeker as "the use of the technology of interactive graphics and the crafts of graphic design, typography, animation, and cinematography to enhance the presentation and understanding of computer programs." By using visualization techniques to provide graphical views of program behavior, a debugger can present trace information in a fashion more analogous to the user's mental representation. Another advantage of exploiting graphics is that a larger quantity of information can be conveyed. As Raeder points out, it is "common knowledge that the human mind is strongly visually oriented and that people acquire information at a significantly higher rate by discovering relationships in complex pictures than by reading text." Well designed graphical displays can integrate substantial amounts of detail without sacrificing intelligibility.

Numerous psychological studies have demonstrated that as problem complex-
ity increases, so does the importance of the manner in which the problem is presented. It follows that the process of understanding the behavior of a complex parallel program can be facilitated by a visualization system which provides unified graphical representations of program behavior. The remainder of this discussion is devoted to recent developments in visualization techniques for parallel debuggers. Examples are drawn from a survey of some three hundred articles, conference papers, and technical reports.42

4. New Methods for Visualization

The visualization techniques employed by parallel debuggers can be characterized by the perspectives from which they represent program execution. Three basic approaches may be distinguished. They will be referred to as functional, structural, and behavioral, in allusion to the corresponding models of systems analysis: functional portrayals of a parallel system emphasize its nature as a set of entities and their relevant tasks; structural portrayals describe how the system is put together, focusing on interfaces and the flow of information between units; while behavioral portrayals are concerned primarily with how the system as a whole responds to specific conditions or inputs.

4.1. Functional Visualization Schemes

The simplest view of a parallel program is as a collection of entities (processes), each performing some sequence of activities which can be carried out as an integral unit. From this perspective, the principal concern in debugging is function: each process must correctly execute its specified duty. Commercially available parallel debuggers provide the means for inspecting the activities of processes on an individual basis, thereby supporting a functional view of program behavior. The state-based information collected by these systems is generally displayed as text, in spite of any use they may make of graphical techniques to support windows or multiple instruction streams. The user must correlate changes in function, as reflected by progress through program statements, with his/her concept of what correct program behavior should be. This is awkward even with a single window (i.e., serial program), and the difficulties are compounded when changes occurring in a number of processes must be assimilated and synthesized into some coherent notion of overall behavior.

A number of experimental systems have attempted to facilitate this conceptualization by abstracting the process unit. Rather than tracing function locally within processes, these systems portray the global flow of execution among them. The process replaces the
statement as the basic unit of observation, so state now reflects process creation, deletion, execution, delay, and completion. Functional dependencies are shown explicitly through some form of "process genealogy." This is often depicted as a hierarchical framework similar to a call graph, where vertices denote processes and arcs indicate control dependencies. Execution behavior is then animated by altering the color or pattern of the vertices to depict state changes over time.

This type of visualization system is included in the Schedule Trace Analysis Facility developed by Dongarra and Sorensen\textsuperscript{43} at the Argonne National Laboratory (Figure 3). Nodes represent subroutine-level processes and can have one of four states: clear nodes are subroutines awaiting the completion of subordinate nodes, lined nodes are ready to run but haven't yet been started by the system, hashed nodes are currently executing, and black nodes have reached completion. Stategraph\textsuperscript{44}, developed for the GMAT system at Livermore National Laboratory, extends the functional representation by adding textual abbreviations to indicate why each process has been suspended. Another debugger, developed by Zimmermann et al.\textsuperscript{45,46} at the École Polytechnique Federale de Lausanne, provides a related view of execution flow (Figure 4). This tool is based on a programming languages that makes use of monitors to enforce synchronization, so a special graphical cue (the box icon) identifies monitor processes.

4.2. Structural Visualization Schemes

The problem with functional visualization is that snapshots of individual and collective process activity do not adequately describe parallel behavior. If the notion of global state is to be useful, the occurrence of process interactions must also be taken into account. A series of experimental visualization systems have evolved from this realization. Such systems provide a means of inspecting the interconnection of processes via flow of information and thus may be characterized as providing a structural view of program execution.

Processes interact in two ways: through communication, or the exchange of data information; and through synchronization, the exchange of control information. To date, debugger developers have focused on displaying communication interactions. The simplest presentation of communications data is in the form of textual messages, ordered by occurrence. This is analogous to traces of control flow or data accesses in traditional debugging systems, and suffers from the same problems of intelligibility. As in other textual traces, it is possible to use indentation to express causal relationships. The Traveler parallel debugger developed by Manning\textsuperscript{47} employs such a nesting scheme to indicate when further
Figure 3. Subroutine-level process representation from the Schedule Trace Analysis Facility\textsuperscript{43}.

Figure 4. Process state display from Zimmermann et al.'s visualization system\textsuperscript{45}.

message traffic is provoked by some initial communication.

Another way of presenting an after-image of communications over time is through the use of time-process diagrams. A time line is drawn along one dimension, while individual processes are distributed across a second. Moving back and forth along the time line reveals the sequence of communications recorded during execution. Message traffic is portrayed by lines connecting pairs of processes. One example is the display proposed by Harter et al.\textsuperscript{48}
Figure 5. Process-time diagram from IDD\textsuperscript{48}.

at the University of Colorado for the IDD distributed debugger, shown in Figure 5. Here, processes are distributed along the vertical axis, while time increases from left to right along the horizontal. The Moviola system being developed at the University of Rochester by LeBlanc et al.\textsuperscript{49} uses essentially the same arrangement, with the axes reversed (Figure 6).

The PPEM developed by Brandis and Thakkar\textsuperscript{50} at the Oregon Graduate Center expands the time-process concept to include indications of the direction and type of communications flow. The information is presented in a “causality history diagram,” illustrated in Figure 7. Upward arrows indicate outgoing messages, downward arrows signal message receipt, and text is incorporated to denote user-defined message types as well as destination addresses.

Communications information can also be animated through a series of displays, each depicting an instant during program execution. The Radar debugger, developed by LeBlanc and Robbins\textsuperscript{51} at the Georgia Institute of Technology, uses animation to portray the direction and sequencing of communications in a more abstract form (Figure 8). Processes are represented as boxes with input and output ports; the number of queued messages is shown at each port. As a message is passed from one process to another, a dotted line appears across the screen. Hough and Cuny\textsuperscript{52,53} of the University
of Massachusetts capitalized on the observation that some parallel programs exhibit recurring or uniform patterns of interprocess communications. Their Belvedere debugger allows the user to specify the spatial arrangement of processes in order to model the communication topology most likely to reflect such logical patterns. Figure 9 shows a sample frame from an interaction animation, with processes arranged to represent a hypercube topology. Highlighted arrows represent outgoing messages, highlighted ports indicate messages received, and multiple arrowheads portray message queueing at a port.

The second type of interaction — synchronization — permits one process to influence the activity of another by the exchange of control information in the form of semaphores,
events, barriers, rendezvous, and so forth. These mechanisms impose constraints on the global ordering of events and can be used to coordinate access to shared data or to control the sequencing of process activities. Although synchronization plays a critical
role in many parallel programs, contributing a new source of errors as well as degrading performance, only a few documented visualization systems provide direct support for observing synchronization patterns.

One such system, mtdbx, from Griffin et al. at the Los Alamos National Laboratory, employs a character-based time-process diagram to give information on events, barriers, and access to critical sections. In Figure 10, for example, parentheses enclose the portion of time that a process was in its critical section, an S indicates the spawning of a subprocess, a Q that the process has terminated, a P that it has posted an event, and a W that it is awaiting a signal. Related information is provided by Zimmermann et al.'s system, described previously, which animates the sequence of synchronizing events (Figure 11).

While the view of program behavior provided by these visualization systems is predominantly structural, some also offer a limited degree of insight into functionality by
differentiating between active and suspended processes. In the cases of Moviola and PPEM, the wait states portrayed in the displays reflect the implicitly synchronous nature of message traffic. The mtdbx trace, on the other hand, reflects the suspension/resumption imposed at explicit synchronization points. The intermingling of both functional and structural views can be especially helpful in detecting deadlock situations and for performance evaluation.

4.3. Behavioral Visualization Schemes

Execution can also be viewed from a third and higher perspective. The behavioral approach reveals how the program responds in a global, algorithmic sense to particular scheduling schemes and sets of input variables. The advantages of this approach derive from its relationship to the cycle of program development. The process of implementing a computational solution to a particular problem involves a series of conceptual restructurings: the programmer’s abstract model for problem solution must be reformulated as an algorithmic solution, then transformed into program code, and finally translated into executable code. Each restructuring is a source of potential error and distortion.

In debugging, the sequence of transformations is performed in reverse, starting from the level of program code. Debuggers typically supply clues about errors in terms of the program code: the succession of interprocess messages or events, the call sequence, the state of a process when an error occurs, etc. This level of information aids in investigating anomalies introduced during the transformation from algorithm to program code. It is the programmer, however, who must examine program statements, relate them to the
appropriate portion of the algorithmic solution, and extrapolate the conceptual abstractions necessary to infer the cause of the error.

By furnishing a view of program execution from a behavioral perspective, a debugger can elevate the level of exploration to encompass the initial transformation from abstract model to algorithm. This tactic has been adopted by algorithm animation systems\textsuperscript{56,57}, which use graphical techniques to provide a logical view of program behavior. Execution is portrayed in terms of high-level operations on abstract data structures. Since comprehensibility is the objective of these visualization systems, iconic representations are arranged on the screen to maximize the clarity of the algorithm's actions.

The Voyeur system, developed by Socha \textit{et al.}\textsuperscript{58} at the University of Washington, is the first debugging tool to support abstract representations of parallel programs. In Figure 12, for example, a load-balancing algorithm is animated as a display of fishes and sharks across a grid. The ALADDIN\textsuperscript{39} system will offer similar capabilities, although the parallel version is not yet complete. Abstraction techniques such as these can significantly improve the effectiveness of visualization systems, since the user no longer needs to integrate isolated components of program behavior in order to infer the overall effect. Their major drawback is the question of how or if debugging tools might extract sufficiently abstract information from program code to provide meaningful displays. Even the simplest documented examples required extensive input from the user prior to execution.

5. Visualization Systems from the User Perspective

The goal of a debugger visualization system is to allow the inspection of relevant aspects of program behavior to facilitate the detection and elimination of errors. Just how successful existing visualization systems are in attaining this goal can only be surmised. Ultimately, their effectiveness must be judged on the basis of user experiences, which tend to be sporadic and often go unreported in the literature. Nevertheless, a series of observations can be made concerning the tradeoffs of various approaches, supplemented where possible with user comments on debugger implementations.

Visualization systems can be evaluated on two levels. Edward Hill\textsuperscript{60} observed that "a drawing acts as a reflection of the visual mind. On its surface we can probe, test, and develop the workings of our peculiar vision." To be truly effective, a visualization system must provide displays appropriate to the creative exploration of parallel program behavior. The informative value of a system's displays therefore comprises the first evaluation criterion. Critical issues include the graphical mechanisms used to portray program execution and
Figure 12. Animation of a load-balancing algorithm from Voyeur.techniques for reducing the complexity of behavioral information. The second evaluation criterion reflects the ease with which users interact with the system. Here, the key factors are how well the visualization system relates to the user's conceptual model of program structure and the mechanics of the user interface.

5.1. Portraying the Dimensionality of Program Behavior

Visualization systems employ static or animated techniques for representing program behavior over time. The nature of the output medium restricts the expression of this behavior to two dimensions. Unlike the work × time relationships of serial execution, however, parallelism requires consideration of work × time × processors. The effective
portrayal of behavior within these limits is problematical.

In static displays, the depiction of time occupies one screen dimension, so processor topology can be reflected only through a meaningful placement of processes across the second. That this can be done effectively is illustrated in the odd-even merge sort example described for Moviola.49 A correct execution sequence results in the butterfly network of crisscrosses shown in Figure 13a. An examination of the execution history from an incorrect run reveals an erroneous exchange pattern; in Figure 13b, the final exchange should be between the first two pairs of processes rather than the observed swap between the second and fourth pairs.

The use of animation to portray program behavior allows the logical relationships between processes to be depicted spatially using the two dimensions of the screen. Temporal relationships are represented via the dimension of elapsed time, allowing events to be seen "as they happen". This natural rendition of time adds extra depth to the display by allowing for more expressive use of the screen. The advantages of such a system can be seen in the Belvedere example (Figure 10), where the graphical display of process relationships reflects the conceptual model of the program.

The elimination of the time dimension from the screen buffers the effects of program complexity on display intelligibility, so animations can also present more detailed information. In Moviola's static presentations, interprocess communications occur as lines connecting processors. This simplistic view suffices to describe implicit communications through shared variables, but no the distinct phases which occur during synchronous communications. Although the static diagram of the PPEM system accomplishes multi-phase portrayals through the addition of text (Figure 7), the information is difficult to assimilate. The transition to a animated form of representation allows the presentation of a more intuitive picture. In the Radar (Figure 8) and Belvedere (Figure 9) examples, each phase of the communications process can be distinguished easily.

The natural representation of time provided by animated displays also introduces some problems. A principal difficulty is how to control the timing of display updates in some germane fashion. While the most straightforward method is to choose intervals equal or directly proportional to the actual times recorded for events, this is not feasible in most cases. Parallel program runs tend to be lengthy and the activities observed by debuggers occur only sporadically. The simple scaling of time intervals thus results in displays with long periods of inactivity punctuated by irregular bursts of complex updates. Some debugger designers have opted to distort the time dimension by introducing special representations for "blank" time intervals. Others allow the user to vary the animation rate, speeding
Figure 13. Time-process display of a merge sort from Moviola\textsuperscript{49}: (a) correct exchange sequencing; (b) closeup of erroneous final exchange.
it up for those portions of the program which are of little interest, and slowing it down to observe activities more carefully. Either approach has the disadvantage of destroying the proportionality of timing between events. The user must subjectively quantify the observed intervals to recognize timing-dependent phenomena. This is particularly critical for performance assessments, where an accurate timescale is needed to identify hotspots, bottlenecks, and load balancing requirements.

Another difficulty with animated displays is that the user must mentally keep track of what transpired several frames before or after the occurrence of a critical event in order to get a global idea of behavior. One measure suggested by Hough and Cuny\textsuperscript{52} to alleviate this is “traced animation,” where events remain on the screen through successive frames, thus generating an enduring record. Unfortunately, their approach is only useful until the same sequence starts to repeat, at which time the prior trace is overwritten — a situation unobservable by the user. Brewer \textit{et al.}\textsuperscript{61} of the Argonne National Laboratory developed an alternate approach in their MAPI/MAPA system. The user is free to vary the display refresh rate as well as the speed of the animation, thereby determining the period of time during which an activity persists on the screen. In effect, this provides a dynamic “window” onto the program’s execution history. It should be noted that static displays do not suffer from the same problem, since they portray a global view of program behavior over time.

The effectiveness of both static and animated displays derive in large measure from their success in demonstrating the patterns of process interactions. For static displays, this involves arranging process lines across one dimension so that the appropriate behavioral patterns are emphasized visually. Any success, however, appears to be more a matter of chance than of technique, since the placement algorithms employed by current debuggers are extremely limited. The generation of meaningful animations can be even more difficult because the arrangement of processes within the plane of the screen must also stimulate user recall as the display changes. Hough and Cuny\textsuperscript{53} illustrate the problem with an example. During a grdsort program, swaps are done by row, column, and diagonal exchanges. Figure 14a shows a snapshot taken during the animation of such a program; the occurrence of unrelated events graphically masks the important interactions so that no coherent pattern emerges as the user watches the transitions. If execution is viewed from the perspective of a single process (rather than the global system state), a much more expressive display results; as seen in Figure 14b, the logical patterns can now be discerned. Unfortunately, the availability of features controlling display viewpoints requires that the user not only experiment with the spatial arrangement of processes, but also view execution from many perspectives in order to achieve a useful representation.
Figure 14. Animation frames from Belvedere\textsuperscript{52}: (a) extraneous information masks the fact that a communication is missing; (b) viewing execution from the perspective of processor e22 reveals the problem.

5.2. Reducing Display Complexity

The extraction of relevant information for display is another critical concern in the design of visualization systems. The monitoring of parallel programs quickly results in enormous quantities of trace data. If presented in its entirety, an execution history can easily compound the problems of debugging. The issues of trace simplification, however, have not been addressed in any generally acceptable manner. Experience has shown that the cyclic debugging process involves putting off complex issues as long as possible\textsuperscript{1}. The programmer formulates a high-level, abstracted view of program behavior. It is at this level that tests are performed in an attempt to stabilize and localize the error (see Figure 2). Once clues to the error's locality have been recognized, the programmer can concentrate repair efforts there, effectively "funneling in" on the error\textsuperscript{62}. The process mimics the top-down approach to program development, which is often recommended for reducing program complexity.

If parallel debuggers are to support this strategy effectively, they must allow the programmer to observe overall behavior, then successively narrow the focus as hypotheses concerning the cause of the error are refined and tested. Display structures should permit a hierarchical examination, starting with global system behavior, followed by a subset of processes, then an individual process. To do this, the programmer must be allowed to step freely from one level to the next. The use of "zoom" or "pop-up" facilities to window
program details achieves this very effectively, as demonstrated in the GMAT tools. Such features do not change the content of the display; they simply manipulate which portion will be seen and at what scale, allowing the user to enlarge areas of interest or to shrink them for an overall view.

A complementary approach would be to incorporate mechanisms permitting the user to determine the display's content as well as its scale. Filtering mechanisms can be used to eliminate particular actions or program components from the user's view of program behavior, but they are available only rarely \cite{48,49,63}. The clustering approach proposed by some authors \cite{64,65} would provide the user with a basis for combining sequences of actions into meaningful higher-level abstractions that are customized for each target program. Since the user would bear the responsibility of devising appropriate abstractions, it is uncertain how effective such a strategy might be.

5.3. Supporting Conceptual Models

Whatever the basis for program visualization, the user must be able to correlate display information with the program code in order to identify and eliminate errors. To date, there has been little attempt on the part of debugger developers to facilitate correlation activities, and few mechanisms exist for coordinating monitored program events with the appropriate statements in the program. At best, the user can interactively view the statement where a breakpoint occurred — it is his/her responsibility to infer which program actions caused the anomaly revealed at that breakpoint. The ability to step interactively "backwards" from a breakpoint, re-playing each program action in reverse, would be extremely useful. Although this is a standard feature in many serial debuggers, it has not been implemented successfully by any parallel debuggers \cite{66}.

As mentioned previously, the programmer must also make a mental transformation from program code to conceptual problem solution in order to hypothesize the cause of the error and devise an appropriate repair. Consequently, it is important that debuggers accommodate the conceptual models for parallelism supported by common programming languages. These can be categorized as task-oriented or data-oriented, according to the structural focus employed to incorporate parallelism \cite{67}; the classification is analogous to the use of functional decomposition or domain decomposition as the basis for program development.

The earliest models for parallelism were task-oriented, reflecting the viewpoint of operating systems designers, who saw parallelism as the concurrent execution of distinct
programs. Parallelization is achieved by partitioning the tasks to be performed into a collection of sequential processes that will be carried out concurrently in relative autonomy. The cooperating processes, pipeline, and monitor/task models are common examples from this category. For task-oriented parallelism, the motivating factor in devising a programming strategy is function: the programmer decomposes the problem in terms of sequences of activities that can be carried out as integral units. Therefore, the principle concern in debugging this type of program is to insure that the function assigned to each process, and summarily to the program as a whole, is performed correctly. Functionally-oriented visualization techniques are the foundation of debugger support for task-oriented models.

Each task-oriented model represents a particular approach to cooperation among processes and, as such, could benefit from specialized debugger features. The incorporation of Belvedere-like facilities for arranging the display to reflect a pipeline or hypercube pattern of interaction, for example, enhances the representational content significantly. From another perspective, the monitor-based visualization system proposed by Zimmermann et al. (Figure 11) shows the advantages to be gained by tailoring a debugger to particular conceptual models. Its capability for revealing accesses to monitor resources is accurate and easy to use; such specialization, however, is achieved at the cost of limited applicability.

Data-oriented models provide a contrasting approach, in which parallelism occurs as the simultaneous performance of a common operation on multiple data elements. Whenever the size of the data warrants parallelization, activities are replicated for application across data subsets. In effect, program execution is viewed as a single thread of control which temporarily diverges into parallel action sequences that later converge. The most common examples are the concurrent loop and master/slave models, which reflect the data parallelism inherent in vectors. The so-called domain decomposition model corresponds to other topological distributions of data, such as multidimensional arrays or trees. With data-oriented models, developing a programming strategy is by domain rather than functional decomposition: data, not tasks, are partitioned and distributed across logical processors. Structurally-oriented visualization schemes thus form the basis for current debugging support of these models.

Most structurally-oriented debuggers were developed for distributed memory machines, where data must be explicitly parceled out among processors. Consequently, they provide at least a crude approximation of interprocess communication patterns. Representing the communication patterns for programs on shared memory machines poses a distinct set of problems. Each write operation to shared memory constitutes a potential communication
to all other processes. Since the overhead for interprocess communication is significantly less for shared versus distributed memory machines, this type of interaction is likely to occur with some frequency, with the result that time-process displays quickly become unintelligible. One solution\textsuperscript{64} is to limit the portrayal of run-time behavior to storage access patterns. Another approach, used in GMAT's Timeline\textsuperscript{44} tool, is to attribute synchronization events on the time-process display with textual keys distinguishing, for example, masters from slaves and processes waiting on locks from those at a barrier.

5.4. Interface Mechanics

While the primary goal of any debugger visualization system is to furnish informative views of program execution, its effectiveness is limited when the user cannot interact easily with the tool. The most common problem has already been alluded to: parallel debugging systems place the responsibility for building a meaningful display on the user, who must encode specialized commands as part of the debugger interface. This not only introduces a new source of program errors, but also makes display appropriateness highly dependent on the user's expertise and familiarity with the system.

One step in simplifying the user interface is to remove some of the burden for specifying program characteristics such as the number of processes, type of communication, the amount and regularity of communications, and the types of synchronization available. The GMAT tools represent one approach, employing fixed algorithms to determine the placement of elements on the screen. The difficulty here is that use of the display area can be quite ineffectual, with some portions becoming congested while others remain blank. It is also possible for the configuration to grow so large that portions of it are totally inaccessible to the user. Perhaps the solution, at least for post-mortem tools, lies in the fact that much of the relevant information can be gleaned from the history file. If a preliminary characterization of the parallel environment can be constructed automatically, missing information can be requested from the user via intelligent queries. Unfortunately, the problem of analyzing process interactions to determine optimal placement has been shown to be NP-complete\textsuperscript{68} and few visualization systems incorporate facilities of this nature. The tool developed by Zimmermann et al. is an example. It pre-analyzes the number of connections between various program components so they may be arranged on the screen to minimize arc intersections; however, there is no indication of how largescale or complex a program might be supported automatically, nor what (if any) control the user has over the display format.
On the whole, existing interface mechanics are clumsy and only marginally acceptable. Even the production-level commercial tools require extensive keyboard input and offer few of the amenities users have come to expect, such as online help facilities, macro or function key definition capabilities, and menu support. Debuggers based on commercial graphical interface packages (e.g., SunView® or X Window®) are the exception; they generally take advantage of “canned” routines to provide mouse control, pop-up menus and data screens, dynamic histograms, and a variety of visual cues that encourage user experimentation.

6. Conclusions

Past history indicates that the motivating factor in the design of most software tools is ease of implementation, not ease of use. This approach has been tolerated in the serial programming environment primarily because users have learned to develop their own methods for programming effectively, resorting to the use of tools only when absolutely necessary. In parallel environments, however, inherent problems of complexity and nondeterminism force the user to rely on programming aids. Tools for parallel program development must be useful, not just easy to implement.

There is no doubt that parallelism presents the programmer with new conceptual challenges. For this reason, it is especially important that the cognitive processes involved in program development be used to guide as well as to measure the progress of tool design. The current state of parallel debugger technology reflects the fact that the user interface was neglected as commercial developers coped with the challenge of implementing even basic debugging services. Nowhere is this more evident than in the area of visualization systems. Debuggers available to the general user community offer at best only minimal graphical capabilities. Yet many techniques which could enhance debugger visualization systems have already been developed in other areas of computer science.

In the fields of system performance and scientific visualization, for example, much recent research has been motivated by the need to portray complex information concisely and effectively. Three-dimensional graphics are now used routinely to describe system performance or to interpret large volumes of data. To date, however, there has been no attempt to incorporate such techniques in debugger visualization systems (in spite of the fact that screen dimensionality has been a chronic stumbling block in debugger display design).
Figure 15. Three-dimensional summary of interprocessor communication from Hyperview\textsuperscript{69}.

The graphics produced by the Hyperview system, developed by Malony and Reed\textsuperscript{69} at the University of Illinois, demonstrates the similarities between the visualization of parallel system performance and program behavior characterization. Figure 15 depicts message traffic among processors in a hypercube, graphically summarizing the effects when row partitioning is used to implement the simplex algorithm for linear optimization. Such techniques are obvious alternatives to classical time-process diagrams, which also present post-mortem summaries of interprocess communications. An example from the area of scientific visualization, where data derived from computer simulation of various scientific and engineering models is displayed graphically for interpretive purposes, is presented in Figure 16. The subject is the simulation of a ring injection of particles into a region exhibiting transitional flow\textsuperscript{70}. Behavioral portrayals such as this could also be of considerable use during debugging.

Of course, the addition of sophisticated graphical techniques to debuggers does not
Figure 16. Behavioral animation of fluid mechanics from a scientific visualization system.

guarantee improvement. The image that produces instantaneous recognition in one setting can obscure program behavior in another. Consider, for example, the MAPA tool already cited. Its graphical representations were designed to reveal access patterns for one- or two-dimensional arrays in Fortran programs. Color intensity is used in an extremely effective manner: as each access to an array element occurs, the corresponding position in the display turns dark (red for write access, blue for read), then gradually fades into the background. While the tool's visual cues are wholly appropriate for certain programs, they are of no help in detecting most parallel programming errors. Conversely, even application of the correct tool to an appropriate situation does not of itself guarantee a satisfactory outcome. An example of this was given in Section 5.1, where the effectiveness of Belvedere’s displays was shown to depend, not only on the arrangement of icons on the screen, but also on the proper selection of a communications viewpoint. If the user is to benefit from a debugger visualization system, the right tool must be used skillfully in an appropriate situation.

Some results of research in serial program development tools can also be applied to parallel visualization systems. In particular, the remarkably slow acceptance of serial debuggers — extending over a period of more than two decades — is testimony to the importance of convenience, flexibility, and appropriateness. Only in the past few years have user interfaces evolved to provide high-level mechanisms that are easy to learn and apply. Graphical displays with reconfigurable windows and single-key or mouse-driven interfaces have succeeded where line-oriented debuggers requiring lengthy typed commands could not. Even more crucial has been the conscious effort to integrate debugger capabilities with established patterns of human debugging behavior. There are lessons to be learned from the
experiences of serial debugger technology: the tool must accommodate itself to the needs of the user, not *vice versa*; and even the most advanced capabilities can be of little help when the user must struggle with interface mechanics.

In conclusion, although recent developments in parallel debuggers are promising, their future is still uncertain. Now that the rudimentary mechanisms for monitoring parallel execution are understood, it is time to accelerate the development of accurate and effective visualization systems. Parallel debuggers will have only limited appeal until they can portray program behavior in a manner which is suitable for real-world problems and a general user audience.
References


Although some sources differentiate between processes as the basic units of work scheduled by the operating system and tasks as the threads of execution control within a process, the distinction is not needed here.


The mechanisms used to monitor execution may be implemented in hardware (e.g., ICMs or dedicated “spy” processors), in software, or through a combination of hardware and software probes.


35 Jong-Deok Choi, Barton P. Miller, and Robert Netzer, Techniques for Debugging Parallel Programs with Flowback Analysis, Technical Report 786, Computer Sciences Department, University of Wisconsin, Madison (August 1988).


49 Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey, “An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-scale


66 Although the term "reverse execution" has been used with respect to parallel debuggers, it actually refers to a rollback mechanism which would allow the user to re-execute the program from a previous point. See Douglas Z. Pan and Mark A. Linton, "Supporting Reverse Execution of Parallel Programs," Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices, 24 (1): 124–129 (January 1989).


69 Allen D. Malony and Daniel A. Reed, Visualizing Parallel Computer System Performance, Technical Report UIUC-CDCS-R-88-1465, Department of Computer Science, University of Illinois at Urbana-Champaign (September 1988).