DEBUGGER VISUALIZATIONS
FOR SHARED-MEMORY MULTIPROCESSORS

Technical Report CSE-91-04

Cherri M. Pancake, Sue Utter

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

March 1991
Debugger Visualizations
for Shared-Memory Multiprocessors

Cherri M. Pancake
(Visiting Scientist, Cornell Theory Center)
Department of Computer Science and Engineering
Auburn University
Auburn, Alabama 36849
(e-mail) pancake@ducvax.auburn.edu

Sue Utter
Technology Integration Group
Cornell National Supercomputer Facility
Cornell University
Ithaca, New York 14853
(e-mail) psu@cornellf.tn.cornell.edu

Abstract

The production of error-free code has always been challenging, but the incorporation of parallelism makes the task truly formidable. Although the idea of exploiting multiple processors in order to decrease wall-clock time is simple conceptually, understanding the complexity of parallel behavior is not. This paper reviews recent developments in parallel debuggers, focusing on techniques for program behavior visualization. The exploitation of graphical techniques enhances the partnership between user and tool, making better use of human creative and analytic capabilities.

Two general visualization approaches are contrasted. The first characterizes behavior by the flow of execution control, portraying changes in process state or sequences of lower-level events. The second employs the framework of data structures to describe patterns of access to shared data. Each approach is appropriate in certain debugging situations. Experience suggests that parallel debuggers need a variety of techniques and levels of presentation if they are to provide comprehensive support for applications programmers.

Keywords: parallel debuggers, parallel debugging, program behavior, behavior visualization, visualization
Introduction\(^1\)

Converting a serial application to parallelism complicates debugging activities by orders of magnitude. Most scientific programmers discover this the hard way, when a purportedly correct serial program is found to generate inexplicable results after a simple, circum-spect parallel construct has been introduced. What should be a relatively minor error, easy to pinpoint and eliminate, requires hours or days to correct. The frustrations of parallel debugging and the scarcity of appropriate tools discourage many scientists from exploiting parallelism\([1, 2]\).

What makes parallel debugging so difficult, and what can software tools do to help? From the programmer's viewpoint, the most annoying problem is that results are often inconsistent from one program run to another. This is due to critical differences between serial and parallel execution environments. First, the order in which program statements execute is only partially predictable for a parallel program. Subtle discrepancies in processor timing or startup overhead may cause identical sequences of instructions to execute faster or slower as they are scheduled on multiple processors. If the system is not dedicated (i.e., executing only one user program at a time), the number of processors allocated to the program may vary from run to run, or even at different points in a single execution. Another key difference derives from allowing multiple processors to share access to common memory locations. Memory contention or the need to create and manage multiple copies of variables

\(^1\)A portion of this research was conducted using the Cornell National Supercomputer Facility, a resource of the Center for Theory and Simulation in Science and Engineering (Cornell Theory Center), which receives major funding from the National Science Foundation and IBM Corporation, in addition to support from New York State and members of the Corporate Research Institute. Funding was provided by the National Science Foundation and Auburn University Teaching-Grant-in-Aid 90-204. Extensions to the prototype implementation of PF-View are being supported by IBM as a Joint Development Project.
can result in performance variations that come and go in response to minor changes in scheduling. It can also provoke race conditions or other access anomalies when the variables are not properly protected. Due to the interaction of all these factors, parallel programs can appear to function correctly for extended periods, then mysteriously produce errors.

It is the non-determinism of parallel environments — independent processors carrying out interrelated activities at unpredictable rates — that complicates the detection and correction of program errors. Simply understanding the basis for behavioral inconsistencies, however, is not enough. Specialized tools are needed which can describe program execution and elucidate the effects of non-determinism on behavior. They must also compensate for some of the idiosyncracies of the parallel environment. For example, one of the most frustrating aspects of parallel debugging is that incorrect results may not be reproducible. If an error cannot be provoked consistently, it is difficult to determine when, or if, it has been repaired. The first challenge for debugger developers is to provide as much stability as possible in an inherently unstable environment.

A second challenge, no less important, concerns how execution information should be presented to the user. Most parallel programs are lengthy and compute-intensive (otherwise there would be little point in incurring the overhead of parallelism). Tracking their runtime behavior involves massive quantities of information, since multiple copies of instruction sequences must be tracked across processors and perhaps memory locations. In addition, although programmers might prefer to ignore the background activities carried out by the operating system, a certain amount of this information will be critical to establish exactly what happened during program execution.

This paper reflects the emergence of a new focus of research in parallel debuggers: the use of graphical techniques to portray the behavior of parallel programs. In particular, it deals with visualization strategies that have been developed for shared-memory multiprocessors (information on distributed-memory systems will be found in [3, 4]). The examples are drawn from a survey of more than four hundred publications on parallel and distributed
debuggers[5]. The first section defines debugger visualization systems and establishes their requirements in terms of typical debugging activities. The next two sections describe specific techniques for portraying information on execution flow and data access patterns. These are followed by a summary of the visualization support needed for a comprehensive debugging environment.

Visualization Requirements for Parallel Debuggers

Visualization, in its broadest sense, refers to the use of computer graphics and specialized representational techniques to convey information. For parallel debugging, we are concerned with program visualization, or graphical techniques that enhance the understanding of computer programs. Even more specifically, a debugger must address the problems of program behavior visualization, since its chief goal is to provide insight into the sequence of events that occur as the program executes on a specific machine.²

The advantage of visual representations is that they offer a means of managing the complexity of performance data. Well designed graphical displays can integrate substantial amounts of detail without sacrificing intelligibility. They also capitalize on the fact that humans are visually oriented, and especially adept at recognizing visual patterns (in some cases, graphical models make it possible to uncover solutions which defy conventional analytical techniques). Empirical studies have made it clear that the manner in which problems are presented assumes more importance as complexity increases, and that appropriate graphical representations significantly enhance the problem-solving process[6, 7, 8]. It follows that behavioral visualizations can facilitate debugging by helping users understand the complex interrelationships of parallel programs.

The problems are similar to those of scientific visualization systems, particularly the

²For this discussion, it is immaterial whether visualization occurs during execution or through post-mortem analysis.
need to interpret enormous quantities of data in order to arrive at some useful representation. There are significant differences as well. Because scientific programs model physical systems, there is some "natural" representation against which the visualization can be compared to determine its effectiveness. Visualization focusses on how to abstract the appropriate display from the raw data. For a parallel debugger, the situation is almost the opposite. The recorded data may be quite close to the physical portrayal of an executing system (changing data values, instruction counters, synchronization activities, etc.). To understand and correct the source of an error, however, the programmer needs to observe the execution of the problem being solved, not just the program code. The debugger must capture the problem-solving strategy at work[9].

Parallel debugger developers have begun to explore how execution events should be reformulated for presentation to the programmer. Debuggers that simply extend the scope of serial debuggers, showing code and data operations on a per-processor basis (e.g., Encore's Parasight, Sequent's pdbx and other parallel versions of UNIX's dbx), are of little use because they require too much effort. The user must assimilate details from many instruction streams in order to extrapolate some notion of overall behavior. Debugging tools are needed precisely because it is difficult to recognize anomalous behavior in a parallel program. Therefore, this discussion is limited to tools which synthesize some general view of program execution.

Debugger visualization systems employ both static and animated techniques for representing program behavior. Unlike the computation × time relationships of serial execution, parallelism requires consideration of computation × processors × time. In static displays, a "time line" occupies one screen dimension, providing a graphical history with explicit temporal relationships. Animated displays, on the other hand, mimic the elapse of time by projecting a series of frames, or snapshots of program behavior. There are tradeoffs to both approaches, as discussed below.

The simplest way to consider program behavior is in terms of transformations performed on input data to generate output data. This viewpoint is convenient for describing the effects
of non-determinism in parallel programs, since it reduces errors to one of two kinds: (1) performing a transformation at the wrong time, in relation to overall program sequencing; or (2) performing the transformation on the wrong data value. (The third possibility, performing the wrong transformation, is a logical error that presumably could have been detected in the serial program.) The next two sections describe visualization techniques which facilitate the detection of each kind of error.

Visualizing Execution Flow

To date, most debugger visualization systems have emphasized the control flow aspects of parallel execution. Although the level of information presented and the graphical representations employed vary considerably from one implementation to another, the systems can be categorized according to whether they portray execution in terms of process state changes or process events. Each approach has been demonstrated to be effective in certain debugging contexts[10, 3].

The representation of program behavior as a series of state changes reflects the fact that a parallel program is a collection of processes executing on physical processors. Each process undergoes a cycle of creation, initiation, termination, and destruction; by comparing their progress, the programmer can infer the progress of the program as a whole. This approach makes most sense when processes correspond clearly to program structure - i.e., with subroutine-level parallelism. The programmer has modularized the work to be performed into subprogram units, some of which can be invoked at the same time or in multiple incarnations. The primary information to be conveyed by the debugger is the sequence in which the subroutines actually begin and complete, so animation is used to portray the elapse of time.

Figure 1 illustrates a classic example of state-change animation, from the Schedule Trace Analysis Facility[11]. A “data dependence graph” provides the visual framework. Although
this looks like a subroutine call graph, it is in a sense inverted; arcs reflect control dependencies, which means the node at the top of the graph cannot execute until all others have completed. Also, the vertices denote processes, so concurrent invocations of the same subroutine appear as multiple nodes. As execution progresses, the appearance of each node changes to indicate its current state: black nodes have reached completion, hashed nodes are currently executing, lined nodes are ready to run but haven’t yet been started by the system, and clear nodes are awaiting the completion of other nodes upon which they depend.

Because Schedule functions in both Alliant-FX and Cray-2 environments, dynamic process creation is also possible. In this particular example (from a symmetric tridiagonal eigenvalue problem), additional processes may be spawned depending on the number of roots to be found at each step. Small, square nodes are added to the display as this occurs. The graphical distinction between static and dynamic process creation allows the user to con-
trast the program’s potential for parallelism with the degree of parallelism actually recorded during execution. Here, for example, node 5 (third from the left in the third tier) had the same potential for spawning as its companions on that level, but no additional processes were created because a run-time calculation determined that the amount of work did not warrant them.

The dynamic call-graph tool provided in the Faust parallel programming environment[12] is similar, although nodes come and go on the screen rather than changing color. In addition to the summary window representing overall program execution, the display provides windows indicating which portion of the tree is executed by each physical processor (on an Alliant-FX or Cedar multiprocessor).

It is possible to portray more than simple process state with this type of representational scheme. GMAT’s Stategraph tool[13], for example, embellishes the tree with textual cues reflecting the status of locks, events, and barriers on the target Alliant-FX or Cray-X/MP (Figure 2). The information reveals precisely why a processor is in a particular state. An optional popup window for each node provides more detailed information, including a list of recent events.

The main problem with state-change portrayals is that they do not provide the user with an accurate sense of context. The animation is either event-driven, with a new frame displayed each time a node changes state, or time-driven, with frames generated after the elapse of a specified time interval. Since the previous state is overwritten with the updated information, there is no explicit record of program history. The user must keep mental track of what transpired during several frames in order to recognize patterns of behavior.

Other visualization schemes are based on process events rather than state changes. An event is a low-level occurrence reflecting the need to coordinate the activities of multiple processes. Events may be defined by the language, the operating system, and/or the user, but typically include synchronizations such as fork/join operations, barriers, and semaphores. State change portrayals make use of events to determine process state. Event visualizations
Figure 2: Frame from GMAT's Stategraph animation

report them directly, describing the sequence that occurred during execution. Because behavior is reported in more detail, event visualizations are somewhat more flexible, applying to loop-level as well as subroutine-level parallelism.

The most common representation of process event information is in a chronological summary referred to as a process/time diagram. A time line occupies one dimension of the screen, while individual processes are distributed along the second. Moving along the time line for each process reveals the sequence of events it experienced during execution. For example, the output from BBN's gist performance analyzer[15] is shown in Figure 3. Each horizontal
Figure 3: Output from BBN's gist
line corresponds to a single processor on the BBN Butterfly, while the small boxes represent events. This static, after-the-fact portrayal is designed to facilitate the identification of event pattern anomalies, such as the "stragglers" indicated in the figure.

One disadvantage of the process/time approach is that there are no visual cues as to what source code statements precipitated each event. The IMPACT (Integrated, Multiprocessor Performance-Analysis and Characterization) event display tool included with Faust[12] addresses this problem. The programmer creates an event definition file establishing which events should be portrayed and associating icons with the events. As a result, it is possible to identify conclusively the source of each event in the summary (Figure 4). The tradeoffs are
obvious: event-specific portrayals offer more information than generalized summaries, but make it harder to recognize event patterns, and consequently harder to identify anomalies.

Like state-change portrayals, time-process diagrams can be event-driven or time-driven, depending on whether the units along the time axis represent gaps between events (i.e., logical time) or physical units of time. They can also be animated. GMAT's Timeline tool[13] simulates execution by filling in a process/time diagram as time elapses. Events are shown as boxes superimposed on the vertical time line for each processor (Figure 5). In this case, information on process state is also integrated into the visualization: the number
Figure 6: Animation frame from PF-View, showing code window and execution history window

and density of vertical lines reflect state, while the spawning of processes is shown by heavy horizontal lines.

The PF-View system[14] also reports on both process state and event occurrences. It employs information gathered by IBM's Parallel Fortran Trace Facility to re-play program execution as a series of changes projected on a hierarchical history reflecting the structure of the source program(Figure 6). PF-View offers a number of options which allow the user to control the speed and sequencing of animation. Single-step mode progresses through the execution history, one parallel construct at a time or at the level of individual events. Reverse
execution, context-sensitive repeat, and time-driven mechanisms provide added flexibility. Animation can alternate freely between event- and time-driven modes. The behavioral information is also cross-correlated with a source program display. The user can select an area in the source code and have animation begin or continue at the corresponding point in the execution history, or select a location/event in the history and automatically determine which source statements caused it.

**Visualizing Data Access Patterns**

Arrays and the transformations performed on them are the foundation for most current scientific and engineering applications. Several new visualization techniques have evolved which portray program behavior from the vantage of this basic data structure. In MatVu[18], for example, two-dimensional arrays are depicted as grids of cells, each representing a single element (Figure 7). A logarithmically scaled color table is mapped to the magnitude of the values stored in the matrix. During animation, color changes in individual cells reflect updates to the values.

Matrix visualizations capitalize on familiarity. For many scientific applications, this representational technique captures the user's mental model of the program — regular manipulation of array elements. By observing the patterns of change, the programmer can deduce how the computation is proceeding. Deviation from the expected pattern signals a problem. The task then becomes one of pinpointing the source of the error. Unfortunately, matrix visualization may be of little use for this activity.

In serial debugging, a transformational error is localized by examining data values via source code instrumentation or with the help of a debugger. Aberrant values serve to localize the faulty portion of the computation. As described previously, parallelism complicates the situation by introducing non-determinism. Errors may be caused by the inadvertent sharing of data among processors, or by inconsistencies in the sequence in which data is accessed by
Figure 7: Animation frames from MatVu, showing ranges of matrix values.
Figure 8: Output from Moviola for the upper triangulation step of Gaussian elimination

multiple processors. For these bugs, it is critical that the user be able to ascertain which
processors access the data over the course of execution.

Moviola[16] uses a process/time portrayal to describe accesses to shared variables on
the BBN Butterfly (Figure 8). Here, time increases from top to bottom along the vertical
axis, and processes are distributed horizontally. The "events" in this visualization are lock
synchronizations, or reads and writes to shared variables. The diagonal lines connecting
pairs of processes reflect lock access dependences. Boxes indicate that the process had to
wait for the access it needed.

A problem which arises with this type of visualization is that programmers on shared-
memory systems tend to make extensive use of shared data. The complexity of data accesses
can result in displays that are virtually unintelligible. The SHMAP system[17] deals with
this problem by portraying accesses within the framework of individual matrices rather
than in terms of processes. As in MatVu, each matrix is depicted as a grid of cells. In
this case, however, animation colors are mapped to physical processors rather than value
ranges. When a cell changes color, it means that the corresponding processor has accessed the memory location. SHMAP also makes use of color intensity to provide a sense of context — the color gradually fades away as the animation progresses, reflecting the elapse of time. A drawback is that each matrix is represented by twin displays, one tracking read accesses and the other, write accesses; the user must shift focus back and forth to detect sequencing anomalies.

**Conclusions**

Each of the visualizations described is particularly appropriate for certain debugging activities. Practical experiences with parallel debuggers suggest that it is important to support a variety of techniques and several levels of presentation. Although developers have proposed designs combining control flow and data access portrayals[10, 19, 20, 21], no current systems provide this kind of flexibility. GMAT and PF-View do offer complementary views of execution control flow, reflecting both process state and event occurrences. Both tools suffer from the fact that they furnish no information on data access patterns, although PF-View does show lock synchronization as a process state change.

It is also unfortunate that most debuggers rely exclusively on information gathered at runtime. A major concern for the parallel programmer is what kind of behavior might be expected during future program runs. Static program analysis should not be neglected, since it provides important clues on what patterns of behavior can be expected. A new extension to the ParaScope tool[22] proposes to combine static dependence analysis with dynamic techniques in order to report sources of schedule-dependent behavior. The idea is to be able to warn the programmer that certain code segments might result in inconsistent behavior; alternatively, the tool would validate that all executions using that input will produce identical output.

Developing new visualization techniques is difficult, largely because our experience base
is so limited. Many parallel programmers have never used a debugging tool, while those who have were circumscribed by the narrow scope of existing implementations. It is clear that program behavior visualization is a promising direction for parallel debugger research. The exploitation of graphical techniques enhances the partnership between user and tool, making better use of human creative and analytic capabilities. Only by broadening our approach to encompass a variety of representational schemes, however, can we hope to discover the most effective ways to portray the complexity of parallel execution.

References


