GRAPHICAL ANIMATION OF PARALLEL FORTRAN PROGRAMS

Technical Report CSE-91-07

C. Pancake, S. Utter

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

May 1991

Graphical Animation of Parallel Fortran Programs

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

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

Abstract

A tool for visualizing the behavior of parallel Fortran programs is described. `PF-View` builds on the functionality already provided by IBM's Parallel Fortran Trace Facility, extending its usefulness by postprocessing the textual trace information to derive a graphical framework on which program behavior is animated. Its design capitalizes on formal and informal studies of how programmers go about parallelizing scientific applications, providing an intuitive representation of program execution. Detailed information on lock/event synchronization and performance analysis, while quickly available through additional windows, is hidden from the casual user to maximize ease and simplicity. Most importantly, the program as the user knows it — in the form of hierarchically-organized source code — is constantly available and automatically correlated with the animation to underscore visually the relationship between source statements and run-time actions.

Keywords: parallel debugging, parallel Fortran, parallel traces, program behavior, behavior visualization, visualization

Introduction[1]

Scientific 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 designed to facilitate a "cyclic" approach to error detection and correction [1, 2, 3]. 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. Interactive breakpoint-style debuggers and tracing monitors, which record information on run-time occurrences without allowing the user to alter program behavior, are widely available.

The situation is less encouraging for the parallel programmer. Technical difficulties — primarily due to the intrusive effects of debugging activities on program execution — hinder traditional breakpoint-style debuggers. Moreover, this approach to debugging may not be particularly appropriate in parallel systems, since it requires that the programmer juggle mental images of concurrently executing entities. Surveys of recent research in parallel debugging tools indicate a general movement away from the arbitrary observation activities of breakpoint-driven debuggers and toward the more structured trace-based systems [4, 5, 6]

Debugging tools which present execution data in purely textual form must rely on the user to assimilate the information and derive an overall model of program behavior. Alternatively, the sequence of recorded events can be analyzed and displayed in some more meaningful form.

[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. Project funding was provided by the National Science Foundation and an Auburn University Teaching-Grant-in-Aid. Extensions to the prototype implementation of PF-View are being supported by IBM as a Joint Development Study.

Program behavior visualization refers to the use of graphical techniques to represent execution behavior. Visualization techniques offer a means of more clearly relating trace information to the user's mental representation, so the detection and isolation of errors is facilitated. Another advantage of exploiting graphics is their potential for conveying large quantities of information. Well designed graphical displays can incorporate substantial amounts of detail without sacrificing intelligibility. Unfortunately, parallel debuggers with integrated graphics capabilities are still in the experimental stage [4]. The technological problems of monitoring large programs and composing graphical representations within real-time constraints have not yet been resolved.

The tool described here takes advantage of the functionality already provided by a text-based trace facility and extends its capabilities by postprocessing the trace information to build a graphical framework on which program behavior can be animated. IBM's Parallel Fortran Trace Facility (PF-Trace) is a monitoring system which can record the sequencing of run-time events for programs of arbitrary size and complexity [7]. While the information supplied by PF-Trace can be useful in both debugging and performance tuning, the messages it generates are obscure and extremely difficult to interpret correctly (see Figure 1). This is not just because of the confusing format of trace file records. To use a trace effectively, the programmer must first identify the records of interest and determine their meaning, locate the subprogram and statement number cross-references, then manually search the compiler source listing to find which statements triggered the message. Add to this the volume of trace records generated by an average-size scientific program (often on the order of several megabytes), and it is no surprise that PF-Trace is under-rated and under-utilized by the user community.

Our animation tool, called `PF-View`, automates the interpretation and correlation tasks which make the current PF-Trace product so unpalatable to the programmer. Postprocessing is employed to analyze the PF-Trace output, filter out the considerable quantity of data not

```
00000008 3 0 S0000.W0000 PPORIG   PARALLEL PROGRAM PROC 06 XA
00000008 3 0 S0000.W0000 PTORIG   FROM SYSTEM CODE TASK S0000 ADDR 00CBE000
00000008 3 0 S0000.W0000 PTCALL   FROM SYSTEM CODE TASK S0000.W0000
00000009 3 0 S0000.W0000 PTPARM   FROM PPORIG.0000 FILE TALL CHAR MMSS 99999999
00000009 3 0 S0000.W0000 PPPARM   FROM SYSTEM CODE SPIN QD 00000006 BW 00000000
00000009 3 0 S0000.W0000 PPPARM   FROM SYSTEM CODE SPIN TS 00000050 TW 00000000
00000009 3 0 S0000.W0000 PPPARM   FROM SYSTEM CODE SPIN LC 00000004 LW 00000001
00005832 3 0 S0000.W0000 DYCOMM   NAME COM1   ADDR 3CC21000.016E45A8
00010340 5 0 S0000.W0000 PDFORK   FROM RANDOM.0023 PARM 00 06 000041 001 001000
00010340 5 0 S0000.W0000 PDFORK   FROM RANDOM.0023 TASK S0000.X0001
00010342 5 0 S0000.X0001 PDINIT   FROM RANDOM.0023
00010381 1 5 S0000.W0000 PDFORK   FROM RANDOM.0023 TASK S0000.X0002
00010382 1 5 S0000.X0002 PDINIT   FROM RANDOM.0023
00010388 5 0 S0000.X0001 PLLOCK   FROM PDLOOP.0039 LOCK FINAL
00010393 5 0 S0000.X0001 PLFREE   FROM PDLOOP.0041 LOCK FINAL
00010436 3 4 S0000.W0000 PDFORK   FROM RANDOM.0023 TASK S0000.X0003
00010437 3 4 S0000.X0003 PDINIT   FROM RANDOM.0023
00010479 5 3 S0000.W0000 PDFORK   FROM RANDOM.0023 TASK S0000.X0004
00010481 5 3 S0000.X0004 PDINIT   FROM RANDOM.0023
00010541 2 2 S0000.W0000 PDFORK   FROM RANDOM.0023 TASK S0000.X0005
00010542 2 2 S0000.X0005 PDINIT   FROM RANDOM.0023
00010614 4 1 S0000.W0000 PDFORK   FROM RANDOM.0023 TASK S0000.X0006
00010614 4 1 S0000.X0006 PDINIT   FROM RANDOM.0023
00010623 5 0 S0000.X0002 PLLOCK   FROM PDLOOP.0039 LOCK FINAL
00010670 5 0 S0000.X0002 PLFREE   FROM PDLOOP.0041 LOCK FINAL
00010732 5 0 S0000.X0004 PLLOCK   FROM PDLOOP.0039 LOCK FINAL
00010735 3 2 S0000.X0005 PLLOCK   FROM PDLOOP.0039 WAIT X0004
00010800 2 1 S0000.X0003 PLLOCK   FROM PDLOOP.0039 WAIT X0005
00010831 5 0 S0000.X0004 PLFREE   FROM PDLOOP.0041 POST X0005
00010854 4 3 S0000.X0005 PLFREE   FROM PDLOOP.0041 POST X0003
00010910 2 3 S0000.X0003 PLFREE   FROM PDLOOP.0041 LOCK FINAL
00010945 4 0 S0000.X0006 PLLOCK   FROM PDLOOP.0039 LOCK FINAL
00010976 4 0 S0000.X0006 PLFREE   FROM PDLOOP.0041 LOCK FINAL
00011117 5 3 S0000.X0004 PDTERM   FROM RANDOM.0126 MORE W0000
00011125 3 4 S0000.X0003 PDTERM   FROM RANDOM.0126 MORE W0000
00011130 2 2 S0000.X0005 PDTERM   FROM RANDOM.0126 MORE W0000
00011158 4 1 S0000.X0006 PDTERM   FROM RANDOM.0126 MORE W0000
00011258 3 5 S0000.X0002 PDTERM   FROM RANDOM.0126 MORE W0000
00011294 3 0 S0000.X0001 PDTERM   FROM RANDOM.0126 POST W0000
00011294 3 0 S0000.W0000 PDJOIN   FROM RANDOM.0126 WAIT
00011294 3 0 S0000.W0000 PDJOIN   FROM RANDOM.0026 DONE
```

*Figure 1.*Excerpt from an IBM Parallel Fortran trace file

germane to applications programming, and apply abstraction techniques that correlate the low-level events recorded during execution with high-level source program constructs. Visualization techniques are then applied to construct a graphical representation corresponding to program structure as it is known and manipulated by the programmer. This graphical portrayal of run-time behavior is animated so the user can watch execution "as it happened." Multiple views, in the form of pop-up windows, reflect changing needs during the program life cycle.

A particularly valuable enhancement is the provision of a source code window cross-correlated with the animation. This visually underscores the relationship between source code (where behavior is specified by the programmer) and run-time actions (where that behavior is observed), allowing the user quickly and accurately to link program results with the responsible source code. Another unique feature is `PF-View`'s hierarchical approach, which organizes complex behavioral and source program information for quick access and comprehension. Finally, the tool provides the user with an unusual variety of mechanisms for controlling the sequence of animation.

The `PF-View` system is described in five sections. The first establishes an operational context by outlining the features of the IBM Parallel Fortran environment. This is followed by an overview of the tool. Techniques used in visualizing program behavior and cross-correlating that behavior to the programmer's source code are presented in separate sections. A concluding section summarizes `PF-View`'s features and compares it to other debugging tools.

IBM's Parallel Fortran Facilities

IBM expanded its sequential VS Fortran environment to support parallelism on the 3090 series of vector multiprocessors, which make use of up to six tightly-coupled CPUs. The Parallel Fortran (PF) system includes language extensions, new compiler services, and com-

prehensive run-time support [7]. The compiler employs static analysis to detect DO loops whose iterations are independent and thus safe to parallelize. Through compiler options and directives, the programmer may combine four modes of processing: scalar, vector, parallel, and vector parallel. Explicit parallelism, at both medium and coarse granularity, is also available via new language constructs.

Two constructs support medium-grained parallelism. PARALLEL LOOP is an extension to the iterative DO loop, permitting loop iterations to proceed concurrently. Borrowing from the case statement formats of serial languages, the PARALLEL CASES construct specifies distinct blocks of code which may execute in parallel.[2] Both constructs are *structured* in the sense that the notation clearly delimits the start and end of parallel activity. The closing delimiter activates implicit barrier synchronization; explicit synchronization is still required to assure mutual exclusion for access to shared variables. DO FIRST LOCK and DO FINAL LOCK extensions to a PARALLEL LOOP provide automatic critical section support. The programmer also may define LOCK variables, managed via supplemental run-time library routines. Moreover, within a PARALLEL CASES, the programmer may exercise limited control over the order in which CASE blocks execute through the WAITING FOR CASES clause. Examples of the two constructs appear in Figures 2 and 3.

While structured constructs offer the advantages of readability and added semantic content, they are not always adequate to express parallelism. In such cases, the programmer may specify that replicates of subroutines execute concurrently. This requires explicit task creation, scheduling, and termination. Binary and counting semaphores (called events), locks, and explicit barriers provide flexible control over task coordination. Since the compiler abrogates all responsibility for task management in the presence of these extensions, however, the mechanisms for coarse-grained parallelism are potentially quite dangerous. In practice, most Parallel Fortran errors are due to improper task coordination or failure to

[2]In spite of its CASE-like appearance, the structure is equivalent to COBEGIN...COEND.

```
PARALLEL LOOP stmt[,] index=start,end[,incr]
   [PRIVATE var[,var ...]] ...
   [DO FIRST [LOCK]
       initialization_code]
   [DO EVERY]
       body_code
   [DO FINAL [LOCK]
       termination_code]
stmt CONTINUE
```

(a) construct syntax

```
      PARALLEL LOOP 10 I=1,2000
         A(I) = ...
         C(I) = ...
         IF (C(I).LT.0.0) STOP LOOP
         B(I) = ...
10    CONTINUE
```

```
      PARALLEL LOOP 20 I=1,50
         PRIVATE SLOCAL
         DO FIRST
            SLOCAL = 0.0
         DO EVERY
            ...
            SLOCAL = SLOCAL + ...
         DO FINAL LOCK
            SUM = SUM+SLOCAL
20    CONTINUE
```

(b) examples

Figure 2. PARALLEL LOOP construct

```
PARALLEL CASES
[PRIVATE var[,var ...]] ...
CASE [case_num][WAITING FOR CASES(n1[,n2 ....])]
   case_code
...
END CASES
```

(a) construct syntax

```
      PARALLEL CASES
      CASE
         ...
      CASE 2
         DO 10 I=1,2000
10          A(I) = ...
      CASE
         ...
      CASE 4, WAITING FOR CASES(2)
         DO 20 J=1,650
20          C(J) = A(J*2) ...
      END CASES
```

(b) example

Figure 3. PARALLEL CASES construct

enforce mutual exclusion [8].

The PF-Trace monitoring system included with the run-time environment can be invoked at execution time without recompilation. It reports the occurrence of a variety of predefined events, in the form of time-stamped records written to a trace file. Trace records are generated even when a program terminates prematurely, thus furnishing a history of the events leading up to the abend. They reflect the start and end of program execution; origination and termination of tasks; assignment and completion of task work; task waits at barriers; allocation and sharing of common blocks among tasks; start and end of implicit parallel DO loop, PARALLEL LOOP, and PARALLEL CASES; and use of locks and events [7]. A mechanism is also provided to record user-defined messages.

Because even small PF programs are capable of generating trace files large enough to fill user disk space, a filtering mechanism controls which events are recorded and which ignored. Filtering levels may be set via a command-line option or by library calls from within the program. They reflect the organization of run-time monitoring routines rather than source-level constructs, however, so even a judicious use of filters produces many more records than the applications programmer needs.

Figure 2b shows a simple PARALLEL LOOP where partial array summations are performed in parallel. Each participating task must compete for access to the critical section (DO FINAL LOCK) protecting the reduction variable. The programmer might wish to determine which task holds the lock at a particular point and which, if any, tasks await their turns. The trace output for the construct is presented in Figure 1 (the first eight lines report program startup; the remaining records correspond to the loop's execution as three tasks). It includes the information needed to establish the presence or absence of lock contention, but even an experienced PF-Trace user finds it necessary to confer with the source code, sketch lines connecting the records belonging to each task, and perhaps perform timestamp arithmetic in order to draw any conclusions.

With programs of normal complexity, the task of ferreting out useful information is tedious and error-prone. Some frequent PF-Trace users automate portions of the process by writing trace file analyzers that match up construct-delimiting records and extrapolate timing information. Figure 4 shows an example generated by `Tracemap`, a postprocessor written by astronomer Alan Karp, of IBM's Palo Alto Scientific Center. Although this is an improvement over the original PF-Trace output, it is still difficult to follow and requires that the user manually compare the trace information with the source code listing to establish the relationship between program structure and run-time behavior.

Organization of PF-View

In general, tools for debugging parallel programs rely on a quasi-chronological reporting of execution events to describe program behavior. Whether analyzed on-the-fly or recorded for subsequent processing, events are described with only a minimal amount of detail. Program structure — as the programmer knows it — is reflected only through indirect links (e.g., program module name, statement or line number, message identifier, user-defined event name) to the source code which triggered the events. Establishing which segment of source code was responsible for a particular event therefore requires analysis of both the execution history and the original source program.

Unlike other trace-based tools, which leave this task to the programmer, `PF-View` employs a postprocessing step to correlate program structure with run-time behavior automatically. In the records generated by PF-Trace, the link between execution event and source code takes the form of a string naming a program file, plus an integer representing not the source file line, but a compiler-generated "internal statement number" (ISN).

As diagrammed in Figure 5, `PF-View`'s postprocessor analyzes the information and compares it with the structure of the source program, as recorded in the compiler listing file. The *source analysis phase* utilizes the compiler listing rather than the original source so it

```
TRACEMAP OF TASKS FOR TRACE FILE PIPROG3 TRACE F.

Trace analysis:   6  Procs, XA  Mode

 Time(sec)     ISN        Info
  0.077980                      +    ROOT
  6.902231 MAIN...0024          |      6.824251-1
  6.902231 MAIN...0024 --> W0001 |-->+ WORKER
  6.905472 MAIN...0024 --> W0002 |-->|---+ WORKER
  6.908179 MAIN...0024 --> W0003 |-->|---|---+ WORKER
  6.909206 MAIN...0027     S0001 P   |   |   |
  6.919304 MAIN...0033     S0001 e   |   |   |
  6.921688 WORKER.0007     S0001 e   W   |   |
  6.925690 WORKER.0007     S0001 e   |   |   W
  6.956210 WORKER.0007     S0001 e   |   W   |
  6.956680 WORKER.0007     S0001 |   |   |   |   0.037376-p
  9.106052 MAIN...0034     00010 *   |   |   |
  9.125842 WORKER.0015     S0001 |   L   |   |
  9.126779 WORKER.0018           |   |   |   |   2.224548-1
  9.126779 WORKER.0018 --> X0001 |   |-->|---|---+ PCFORK
  9.177822 WORKER.0015     W0001 |   |   |   l   |
  9.178819 MAIN...0035     S0002 w   |   |   l   |
  9.179973 WORKER.0018 --> X0002 w   |-->|---l---|---+ PCFORK
  9.192091 WORKER.0029           w   |   |   l   w   |
  9.193196 WORKER.0018 --> X0003 w   |-->|---l---w---|---+ PCFORK
  9.194136 WORKER.0018 <-- X0003 w   |<--|---l---w---|---+   0.000940-C
  9.217434 WORKER.0028     X0001 w   |   |   l   w   P
  9.217978 WORKER.0018 <-- X0002 w   |<--|---l---w---+   0.038005-C
  9.218463 WORKER.0029           w   |   |   l   |   0.026372-w
  9.254865 WORKER.0018 <-- X0001 w   |<--|---l---+   0.128086-C
  9.255335 WORKER.0034           w   c   |   l
  9.255789 WORKER.0034           w   |   |   l
  9.256785 WORKER.0036     S0001 w   F   |   |   0.130943-L   0.078963-l
  9.257765 WORKER.0038     S0002 w   p   |   |
  9.294703 MAIN...0024 <-- W0001 w<--+   |   |   2.392472-T
  9.557944 WORKER.0015     W0003 w       l   |
  9.626201 WORKER.0036     S0001 w       |   F   0.369416-L   0.068257-l
  9.627065 WORKER.0038     S0002 w       |   p
 11.031148 MAIN...0024 <-- W0003 w<------|---+   4.122969-T
 11.151220 WORKER.0034           w       |
 11.152207 WORKER.0036     S0001 w       F   1.526006-L
 11.155044 MAIN...0038           t       |
 11.155451 MAIN...0038           |       |   0.000407-t
 11.155567 MAIN...0024 <-- W0002 |<------+   4.250095-T
```

Figure 4. Excerpt of output from Tracemap postprocessor

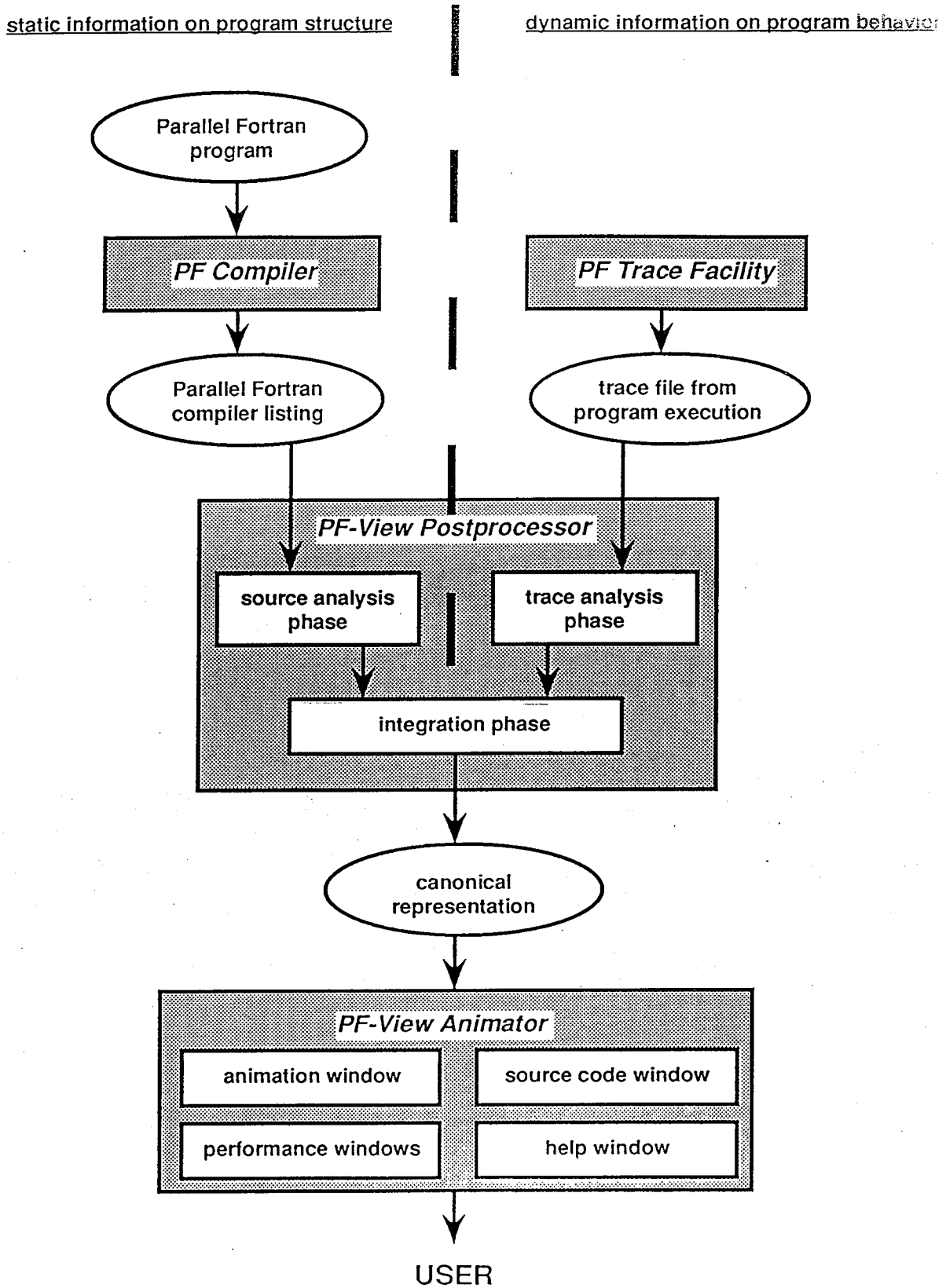


Figure 5. General structure of PF-View

can establish ISNs and recognize any loops that were automatically parallelized. In a separate *trace analysis phase*, the event records generated during execution are examined to create a behavioral profile. Pattern analysis filters system-level events that are irrelevant to the programmer and groups together sequences of low-level events corresponding to single source-level constructs. An *integration phase* then cross-references the analysis information in order to establish the causal relationships between source code and run-time behavior. Since `PF-View` uses hierarchical organization to provide the user with multiple views of program dynamics, this involves the construction of multi-level indices. The product is a canonical representation used to drive the graphical animation.

Animating Program Behavior

The most problematical aspect of parallel traces is the complexity of the information they encompass. `PF-View` approaches this problem by organizing run-time data to reflect human factors studies of debugging activities. Experiments with serial programmers suggest that they adopt an "ease into it" strategy [1]. This involves putting off complex issues as long as possible by first formulating a high-level, abstracted view of program behavior. Various tests are performed at this level in an effort to isolate the source of error. Once a clue to the error's locality has been generated, the programmer can concentrate efforts there, effectively "funneling in" on the problem [9]. The procedure mimics the top-down approach to program development and is based on a hierarchical view of program behavior: the programmer starts by observing overall behavior, then successively narrows the focus to particular areas of interest.

`PF-View` supports this process by furnishing a multi-level hierarchy of animation. At the highest level, animation occurs as a series of graphical changes projected onto components of a structured framework representing the program as a whole (Figure 6). This appears as a series of iconic *execution units* — corresponding to the program's parallel constructs and

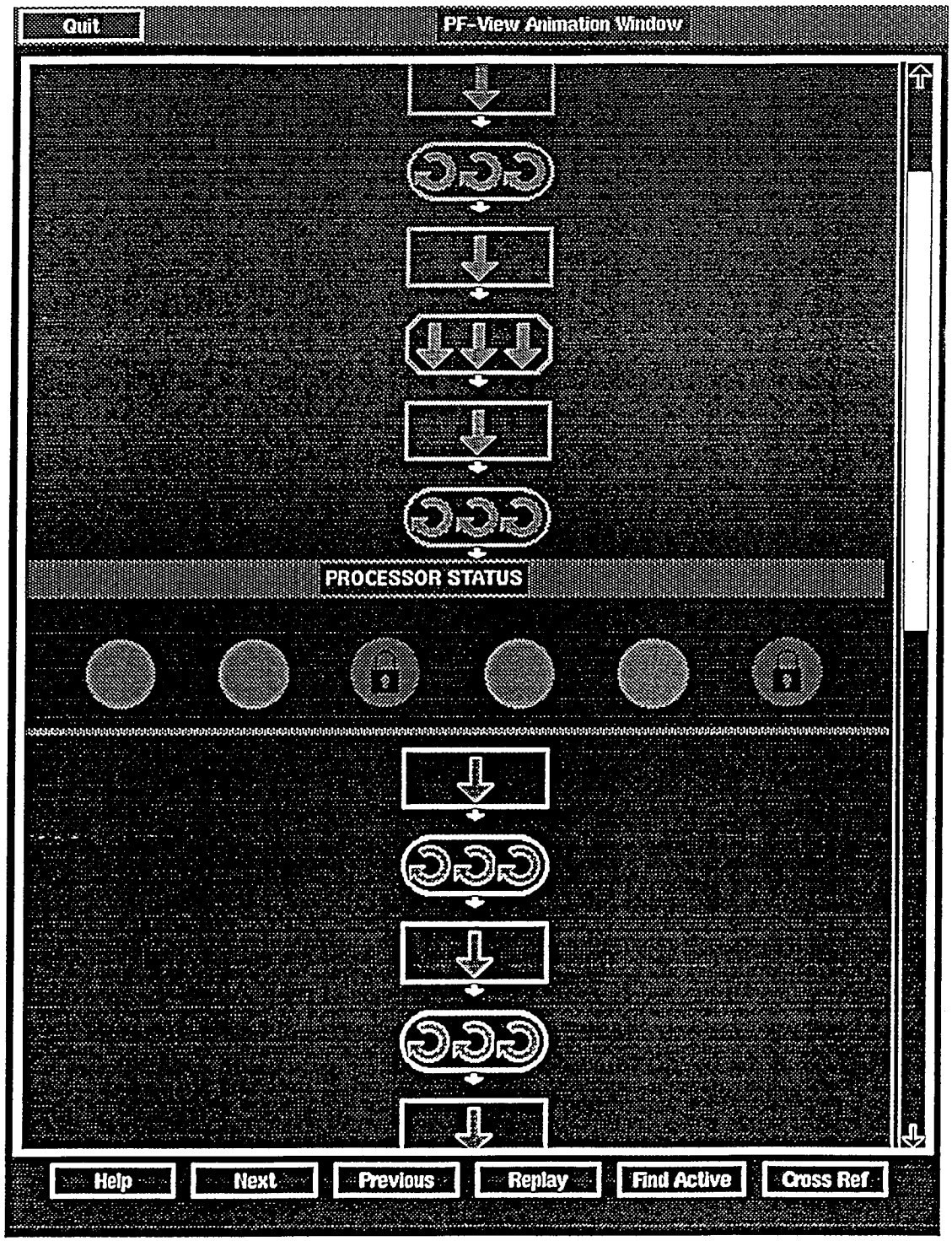
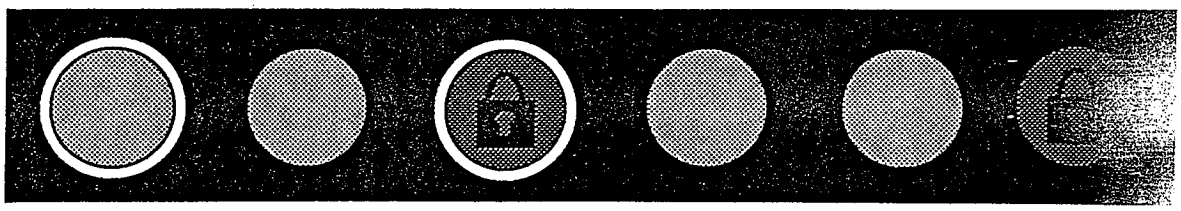


Figure 6. PF-View's execution history window

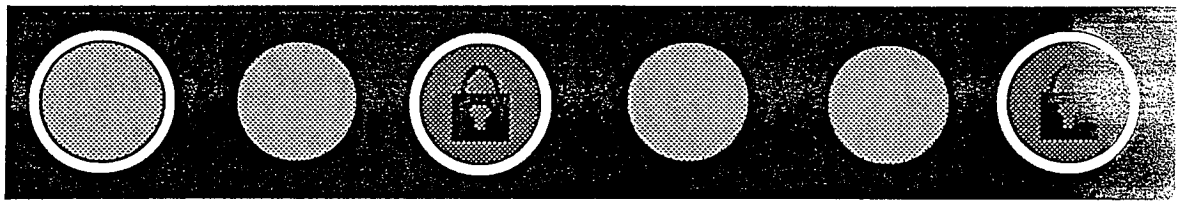
the serial code that separates them — linked linearly to reflect the flow of execution from one unit to another. Serial sections of code appear as rectangles containing single arrows, PARALLEL LOOPs or automatically parallelized DOs appear as ovals with multiple circular arrows, and PARALLEL CASES are polygons with multiple arrows. (Parallel subroutine invocations are not supported in the prototype, but will be added in the full implementation.) When animation is carried out on a color monitor, each run-time occurrence is reflected by a color change; here, they appear as shades of grey.

Execution is animated by stepping through the history window. Each run-time occurrence is reflected by a change in icon color and intensity. Units which have not yet executed are white, the currently executing icon or icons are green, and completed units appear in a dimmed grey. The user watch the entire execution at the global (program) level, or may elect to observe selected units at a greater level of detail. When an execution unit is "selected" with a mouse click, the animation hierarchy is expanded automatically so that individual processors participating in the construct appear.

For example, the animation frame in Figure 6 portrays six processors cooperatively executing a PARALLEL LOOP (once again, shape connotes the source construct, with circular icons for parallel loop tasks, diamonds for tasks executing PARALLEL CASES, etc.). As tasks progress from idle to executing to completed states, forced waits may be imposed by locks, by WAITING FOR CASES synchronization, or at join barriers. These situations situations are depicted by special symbols superimposed on the suspended task icons. In Figure 6, the third and sixth tasks have suspended as they tried to gain control of locks. If the user requests additional information, all processes involved in the lock contention are displayed in highlighted form. Figure 7 illustrates the two possibilities when the user clicks on the leftmost blocked task from Figure 6. In Figure 7a, the lock task 3 wants is currently held by task 1; although task 6 is also waiting, the display indicates that it is waiting for some other lock. Figure 7b depicts the effects of contention for the same lock by tasks 3 and 6.



(a) only one task awaits access to the lock held by task 1



(b) two tasks are suspended waiting for the lock

Figure 7. Graphical representations of lock contention

PF-View also supports multiple animation modes. In *event mode*, the user controls the rate at which the animation is updated via a next-event button. This mode is particularly useful during debugging, when the sequencing of events may be more critical than actual timing relationships. The programmer is free to "reverse" execution at any point, stepping backward through the events which led up to a particular situation. In addition, "instant replay" features allow the user to repeat the animation of a sequence of events related to a particular source program structure.

Alternatively, in *time mode* frames are generated automatically according to trace time-stamps, allowing events to be observed "as they happened"; the user may set the rate to be equal or proportional to the recorded times. Although this mechanism provides a natural representation of execution, it also introduces problems. Parallel runs tend to be lengthy and the activities recorded by monitoring facilities 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. `PF-View` therefore allows the user to change the animation mode interactively, using time-driven mode to speed through portions of the program which are

of little interest, then slowing the rate or single-stepping in order to follow other activities more carefully.

Although not yet implemented in the prototype version of PF-View, a variety of optional timing statistics will accompany the animation displays. The most commonly needed statistics, elapsed time (percentage of total execution time animated up to the point of the query) and processor utilization (number of active processors), have been designed to be sensitive to the query's context. During high-level animation, for example, elapsed time refers to a percentage of total program time, whereas the same query in a low-level animation reflects the percentage of time which has elapsed in the execution of the unit under scrutiny.

Summary statistics will be available as well. In high-level animation, the amount of time between start and completion of a code block will be given both in actual time units and as a percentage of the total execution time for the program. Another breakdown will reflect the proportions of time processors spent executing user code, in system overhead, and idle. For parallel execution units, tables will summarize average values for various task activities: execution time; wait time at lock, WAITING FOR CASES and join barrier sychronization; number of times a task accessed a lock without waiting; and number of times lock contention caused suspension. These statistics are particularly useful during performance tuning activities, since they provide insight on where improvement efforts might be most effective.

Relating Run-Time Behavior to Source Code

During animation, PF-View displays source code alongside the behavioral history, in a scrollable *code window* (Figure 8). Parallel scientific applications are often thousands of lines long and may encompass many modules. Presenting source code as a monolithic piece of text can be very ineffective, since the viewing area provides only a glimpse of the whole and does little to reveal general program structure. The user must frequently scroll back

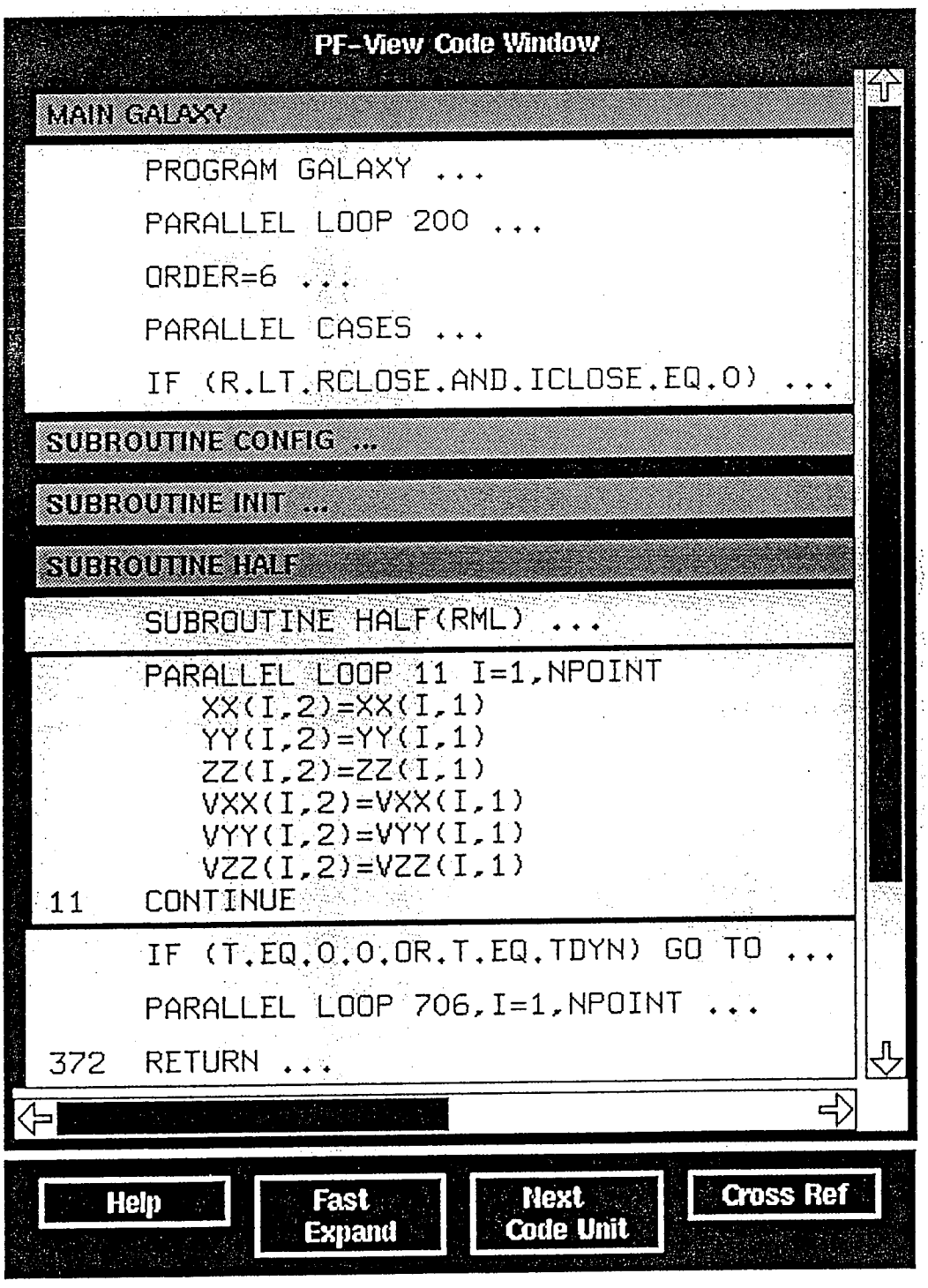


Figure 8. PF-View's source code window

and forth, looking for points of reference. Therefore, PF-View again employs hierarchical techniques to reflect the code organization established by the programmer. At the highest level are program modules: the main program and its subroutines and functions. Each of these is composed of blocks of statements (corresponding to the parallel and serial execution units of the animation window), which in turn are composed of individual statements. As in the animation window, upper levels may be expanded to reveal underlying code, then compressed again to conserve space.

When the code window is first created, it lists the main routine and all subprograms. Each module name is followed by an ellipsis to indicate that more code is hidden beneath the visible line. By clicking on a name, the user can expand it to reveal the execution units belonging to that module; the ellipsis after the name disappears to indicate that it has been expanded. Similarly, clicking on an execution unit reveals the individual statements it includes. Clicking on an expanded unit or module name reverses the process — the components disappear and the ellipsis shows again.

The correlation of program behavior (in the animation window) with the source which caused it (in the code window) is implemented through cross-referencing features. These are specified in one window but trigger changes in the other. For example, the user may select an event in the animation window and use the *Cross Ref* option to identify the code that precipitated it. This causes the code window to automatically scroll to the statements responsible for the run-time event. In Figure 9, the user has stepped the animation to the point where task 3 has suspended due to a lock wait. Selecting the cross-ref option for task 3 identifies the call to the PLLOCK routine where lock access was attempted, and scrolls the code window to that location.

When a cross-ref is requested, the enclosing subprogram is expanded automatically if needed. When the event corresponds to an individual statement, as in the example, the enclosing execution unit will also be expanded. Other code segments remain unaltered to

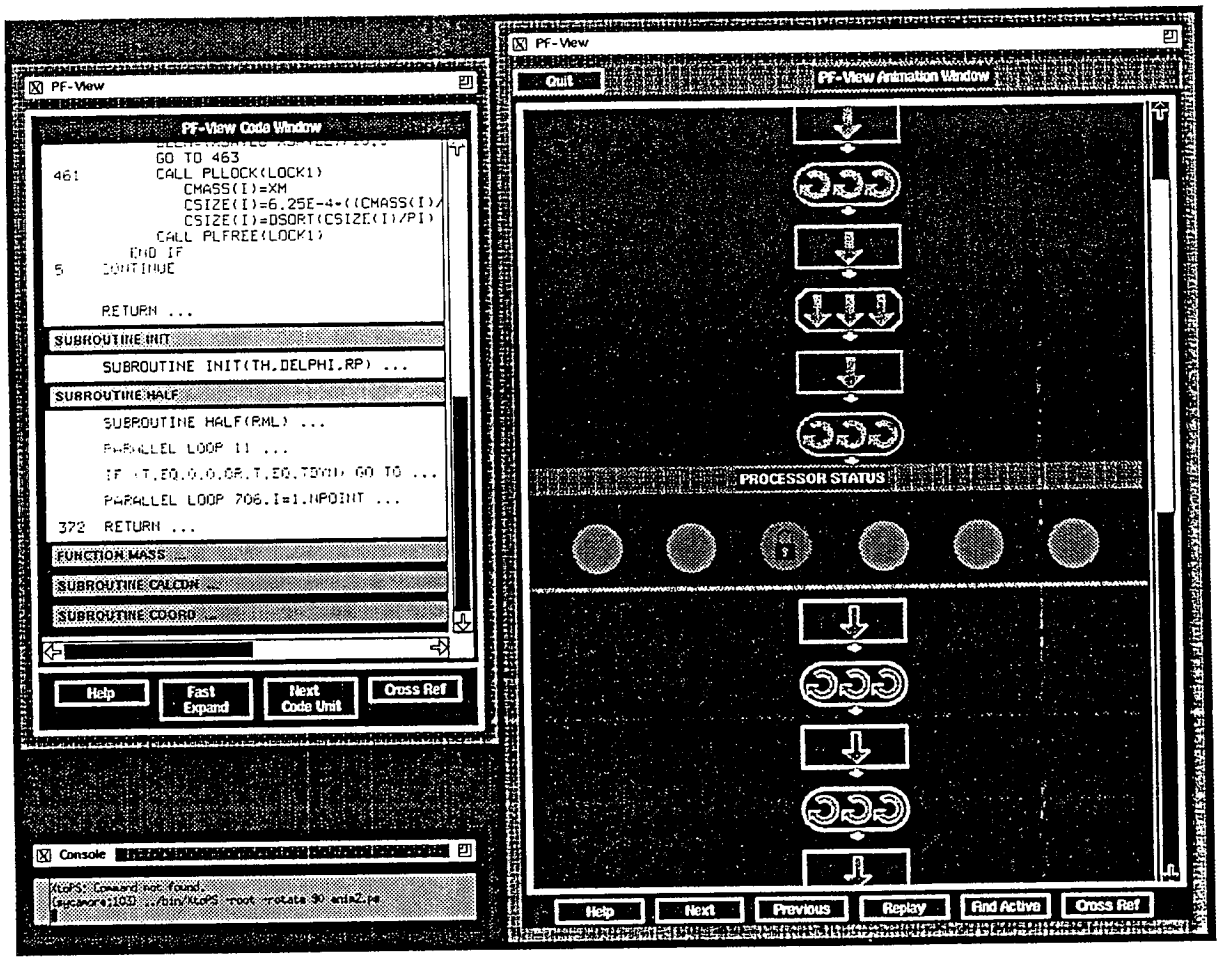


Figure 9. Sample PF-View session

facilitate user orientation.

Cross-referencing may also be carried out in the other direction by selecting a unit within the code window. This action causes the corresponding parallel events in the animation window be executed next (since the mapping from code to event is one-to-many, replay actually continues at the next instance of the construct in the execution history). With this mechanism, the user may watch the propagation of an error by restarting animation from a known point. Code-to-animation cross-referencing also provides an effective way to interactively filter the events seen during animation. For example, the user may view just the execution of one parallel construct which is suspected of containing a bug or a performance bottleneck.

Comparison with Other Work

The program behavior visualization techniques incorporated in `PF-View` were developed in response to the experiences of other tool developers. A survey of more than four hundred technical reports, journal articles, conference papers, and doctoral theses revealed a consensus that graphical techniques were critical to the success of any parallel debugger interface [4, 11]. Furthermore, it became obvious that the visualization requirements of applications programmers rarely overlapped with those of systems designers or tool implementors. Consequently, `PF-View` is targeted to a specific user audience: scientific applications programmers. In particular, the tool's design addresses the problems associated with learning to use a program visualization interface, summarized in [5] and [10].

For example, several alternatives for managing the complexity of behavioral information were explored. The most common mechanisms are zooming, panning, and scrolling (e.g., [12, 13, 14]), which permit the user to change the focus of display interactively. A second technique is to filter the information by allowing the user to specify that only certain types of events or those associated with particular processes be displayed [12, 13, 14]. Cluster-

ing provides an abstraction mechanism whereby related events may be grouped together for display as a single, higher-level occurrence; however, it is provided only rarely and requires considerable programmer expertise to be effective [14, 15]. The disadvantage of all three approaches is that the visualization session begins at maximum complexity, displaying all known information, and is only reduced to manageable proportions by user effort. In PF-View, we circumvent the problem: (1) by identifying what events the applications programmer will want to see and applying a pre-visualization filter to remove all others; (2) by clustering together low-level events to correspond with the more abstract occurrences defined by source code constructs; and (3) by employing a hierarchical presentation schema to minimize the amount of display detail.

The decision to correlate behavioral information with original source constructs was made after observing the disproportionate amount of time required to do it manually. None of the tools surveyed in [4] provides this type of support, although a couple of recent commercial products "animate" copies of the source code with a highlighting bar.[3] In most behavior visualizations, process state is shown with no indication whatsoever of why state changes might have occurred; in many cases, it is not even possible to determine if process icons represent replications of the same subroutine or distinct code segments (e.g., [13, 16, 12]). Others offer event definition capabilities which permit the user to generate his or her own links to the source program, but the procedure requires added effort and expertise, and the user is still responsible for manually cross-referencing the display and program. [14] A major contribution of the PF-View project is that it has established how easy it can be to automate the tedious process of code/behavior correlation.

PF-View's design capitalizes on formal and informal studies of how programmers go about parallelizing scientific applications. It provides an intuitive representation of program execution that can be manipulated in a variety of ways, all extremely easy to learn. Even users

[3]This technique has been common for several years in serial debuggers for PCs.

with little experience of graphical interfaces and debugging tools quickly grasp the concepts of stepping through program execution, expanding and contracting the hierarchical displays, and using the cross-reference function to correlate source code with behavior. Detailed information on lock/event synchronization and performance analysis, while quickly available through additional windows, is hidden from the casual user to maximize ease and simplicity. Most importantly, the program as the user knows it — in the form of hierarchically-organized source code — is constantly available and automatically correlated with run-time actions. `PF-View`'s graphical animation techniques make the raw functionality of the IBM Parallel Trace Facility usable and accessible to the scientific programmer.

References

[1] John D. Gould, "Some Psychological Evidence on How People Debug Computer Programs," *International Journal of Man-Machine Studies*, 7: 151–182, 1975.

[2] John D. Gould and Paul Drongowski, "An Exploratory Study of Computer Program Debugging," *Human Factors*, 16 (3): 258–277, 1974.

[3] Randall Rustin, editor, *Debugging Techniques in Large Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[4] Cherri M. Pancake and Paula Sue Utter, "A Bibliography of Parallel Debugging Tools — 1990 Edition," *ACM SIGPLAN Notices*, 26 (1): 21–37, January 1991.

[5] Paula Sue Utter and Cherri M. Pancake, "Advances in Parallel Debuggers: New Approaches to Visualization," Technical Report CTC-89-TR-18, Cornell University, Cornell Theory Center, July 1989, (to appear in *Advances in Parallel and Distributed Processing*).

[6] Charles E. McDowell and David P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, 21 (4): 593–622, 1989.

[7] IBM Corporation, *Parallel Fortran Language and Library Reference*, IBM Corporation, Kingston NY, publication SC23-0431-0, March 1988.

[8] Cherri M. Pancake and Donna Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?" *IEEE Computer*, 23: 12: 13–23 (Dec 1990).

[9] Ben Shneiderman and Richard Mayer, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," *International Journal of Computer and Information Sciences*, 8 (3): 219–238, 1979.

[10] Cherri M. Pancake and Sue Utter, "Models for Visualization in Parallel Debuggers," *Proceedings of Supercomputing '89*, pp. 627–636, November 1989.

[11] Cherri M. Pancake and Sue Utter, "Debugger Visualizations for Shared-Memory Multiprocessors," Technical Report CTC-89-TR-18, Cornell University, Cornell Theory Center, March 1991.

[12] Robert J. Fowler, Thomas J. LeBlanc and John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," *Proceedings of the ACM SIGSOFT/SIGPLAN Workshop on Parallel and Distributed Debugging*, published in *ACM SIGPLAN Notices*, 24: 1: 163–173, January 1989.

[13] J. J. Dongarra and D. C. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," in Jamieson, *et al.*, eds., *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, Massachusetts, 1987.

[14] Vincent A. Guarna, Jr., *et al.*, "Faust: An Integrated Environment for Parallel Programming," *IEEE Software*, 6: 4: 20–27, July 1989.

[15] Alfred A. Hough and Janice E. Cuny. "Perspective Views: A Technique for Enhancing Parallel Program Visualization," *Proceedings of 1990 International Conference on Parallel Processing*, pp. II.124–132, August 1990.

[16] Mark K. Seager *et al.*, *Graphical Multiprocessing Analysis Tool (GMAT)*, Technical Report UCID-21345, Lawrence Livermore National Laboratory, March 1988.