Applying Human Factors to the Design of Performance Tools

Cherri M. Pancake

Oregon State University, Department of Computer Science, Corvallis, OR 97331, USA
`pancake@cs.orst.edu`
`http://www.cs.orst.edu/~pancake`

Abstract. After two decades of research in parallel tools for performance tuning, why are users still dissatisfied? This paper outlines the human factors issues that determine how performance tools are perceived by users. The information provides insight into why current performance visualizations are not as well received as they should be — and what must be done in order to develop tools that are more closely aligned to user needs and preferences. Specific mechanisms are suggested for improving three aspects of performance visualizations: how the user explores the performance space, how the user compares different aspects of program behavior, and how the user navigates through complex source code.

1 The Performance Tuning Problem

For over two decades, a great deal of research effort has been directed at tools for improving the performance of parallel applications. Significant progress has been made, as can be seen by comparing some of the surveys of parallel tools from that time period [22, 23, 2]. Why, then, are parallel performance tools still so hard to "sell" to the user community?

Part of the problem is that parallel tools can be extremely difficult to implement. The tool developer must copy with an inherently unstable execution environment, where it may be impossible to reproduce program events or timing relationships. Monitoring and other tool activities, intended to observe program behavior, in fact perturb that behavior, sometimes causing errors or performance problems to appear or disappear in unpredictable ways. Further, the notoriously short lifetime of most parallel computers means that there is an extremely small "window of opportunity." Tool development must often begin before the hardware or operating system is stable, but must become available very soon after the computers are first deployed in order to acquire any significant user base [20].

Technological challenges are just one aspect of the problem, however. The most common complaints about parallel performance tools concern their lack of usability [17, 21]. They are criticized for being too hard to learn, too complex to use in most programming tasks, and unsuitable for the size and structure of real-world parallel applications. Users are skeptical about how much value tools can really provide in typical program development situations. They are reluctant to invest time in learning to apply tools effectively because they are uncertain that the investment will pay off.

The situation is paradoxical. As we have established elsewhere, users do not turn to parallel processing unless they need to increase the size, complexity, resolution or speed of their applications.[19] This means that all parallel users are concerned to some degree with application performance. They should be predisposed in favor of any tools that could help automate the processes involved in measuring, analyzing, and improving performance. Instead, they lament the lack of suitable tools.

The fact is that a tool is viewed as just that, a *tool*, something that should facilitate a human process and enable the human to achieve his/her intended goal. Tools are not used in isolation, nor are they appreciated as standalone elements. A tool is only perceived as valuable when it clearly assists the user to accomplish the tasks to which it is applied. Thus, a tool's propensity for being misapplied or its failure to support the kinds of tasks users wish to perform constitute serious impediments to tool acceptance. This is where human factors come into play.

Human factors refers to the characteristics, capabilities, and limitations of human beings and how these affect our use of technology. *Human factors engineering*, then, seeks to improve system effectiveness and safety by explicitly addressing human factors in the design of new systems. (For a general overview of human factors applied to software development, see [5].)

This paper outlines some of the basic human factors that influence the usability of parallel tools. In particular, we examine the human factors environment within which a user attempts to tune the performance of a parallel application. This is followed by a discussion of how current tools support, or fail to support, the human factors involved. The information provides insight into why current performance visualizations are not as well received as they should be — and what must be done in order to develop tools that are more closely aligned to user needs and preferences. Specific mechanisms are suggested for improving three aspects of performance visualizations: how the user explores the performance space, how the user compares different aspects of program behavior, and how the user navigates through complex source code.

2 How Users Approach Performance Tuning

The best way to understand what kinds of tool support are important is to consider how application developers actually go about the process of performance tuning. Over the past decade, we have carried out *in situ* studies of hundreds of scientists and engineers involved in developing parallel applications.[17] Based on this work, we propose that the tuning process has two dimensions, distinguished here as the conceptual framework and the task structure.

2.1 Conceptual Framework — What the User Seeks

As noted earlier, all application developers are concerned, at least on a superficial level, with program performance. Typically, performance measurement and analysis activities are interspersed with periods of code development and debugging.[18] During a performance tuning interval, the user consciously or unconsciously poses a series of questions that address different aspects of the application's behavior.

Those questions establish a *conceptual framework*, or a series of subgoals that must be met in order to accomplish the more general goal of improving performance. Any dependencies among the subgoals establish the order in which they will have to be accomplished. The following is an example of a typical conceptual framework:

1. *Identification:* Is there a performance problem? What are the symptoms, and how serious are they?
2. *Localization:* At what point in execution is performance going wrong? What is causing the problem to occur?
3. *Repair:* What about the application must be changed to fix the problem? [Perform the repair.]
4. *Verification:* Did the "fix" improve performance? [If not, optionally undo the repair, then go back to (2).]
5. *Validation:* Is there still a performance problem? [If so, return to (1).]

Subgoals are indicated in italics, and the ordering and repetition of particular subgoals has also been shown explicitly. In real life, we have observed that users rarely stop to think about the steps they are following or the rationale behind them. That is, the process is largely intuitive in nature.

2.2 Task Structure — How the User Seeks It

While the user may not recognize the underlying conceptual framework, he/she carries it out through a series of deliberate actions. This is the *task structure*, a sequence of individual activities that the user plans and executes to achieve a particular subgoal.

Problem stabilization. The subgoals of identification, verification, and validation are generally accomplished using a single task structure, which we call *problem stabilization*. The performance of the application is benchmarked to obtain some sort of timing information, either overall execution time for the application or a series of timings reflecting the duration of individual phases of the application. Execution will be timed repeatedly in order to obtain a series of "representative" data — that is, reflecting inputs, parameter settings, system load, etc., that are considered typical of the application's intended use.

The timings must then be compared to determine how consistent they are. The performance data are also examined in terms of what the user anticipated (e.g., the computational kernel may be expected to occupy roughly 80% of the total execution time, to complete in about 70 minutes, or to achieve roughly 10% of aggregate peak speed). From the consistency of these timings, the user infers whether the performance is acceptable. Note that this really is a process of inference, since there is no simple way of determining whether an arbitrary application's performance is "good" or "bad," and even less way of determining *a priori* how much effort will be needed to improve upon that performance.

Search space reduction. The subgoal of localization involves a second set of tasks, which we will call *search space reduction.* In a typical scenario, the user first makes an educated guess about which aspect of the application's execution (e.g., memory use, communications, load balance) is responsible for the perceived behavior. That hypothesis is generally based on intuition and recollections of previous experiences with performance tuning, and is therefore highly individual. The user then tests the hypothesis by adding hand-coded instrumentation (or using a performance monitoring tool) in an attempt to isolate where the problem occurs during program execution. The objective is to narrow down which region(s) of the source code is responsible for the problem (e.g., in the part of the program where cells exchange values with their neighbors). It should be noted that this may be a very rough approximation, and may later turn out to be erroneous.

Selective modification. The subgoal of repair is achieved with a third set of tasks, applied in iterative cycles, that we'll call *selective modification.* Focusing attention on one potential problem location at a time, the application developer uses manual inspection to study the code and attempt to determine what is causing the problem. The difficulty — and the effectiveness — of this step is highly dependent on the user's experience level, since it is largely a matter of intuition and judgement. The code is modified in an attempt to alleviate the performance problem, and the user proceeds to the verification stage. If the code change is deemed to be successful, attention is then focused on the next problem location. If, on the other hand, performance does not appear to have improved, the user is likely to un-do the changes and try again.

The task set is called selective because we have found few instances where experienced users attempt simultaneous changes in different regions of the program. When asked why, they say they have learned that it is difficult to accurately assess and balance the effects of multiple, possibly counteracting "improvements." It is perceived as more productive to narrow their focus to small problems that can be solved incrementally, rather than attempting to develop and apply a more global strategy.

2.3 Implications for Performance Tools

An understanding of typical approaches to performance tuning explains much of the frustration that users feel about current tools support. Most tools begin by popping up a series of windows, each revealind detailed information about a particular performance metric. Yet users prefer to begin by gaining an in-the-large perspective on application behavior (where it is spending its time, major hotspots for memory use, etc). A fine-grained summary really does nothing to help the user assess whether performance is generally "good" or "bad," or if tuning efforts are likely to pay off. As more than one user has commented, "Don't tool developers know how awful it is to bring up a tool and have it try to show me all 100,000 communications that took place?" [17] For problem stabilization, the user needs to be able to *explore* the total program space at a higher level, looking for evidence of anomalies.

Current tools are somewhat better when applied to search space reduction. Many allow the user to view different aspects of program behavior, such as memory use, communications traffic, CPU utilization, etc. However, they fail to support the core activity — deciding which aspect is responsible for poor performance. What is needed here is support for *comparing* different metrics. Such comparisons (e.g., the fact that CPU utilization drops in the same places that memory use soars, but appears unrelated to message traffic)would yield meaningful suggestions for directing programmer effort. It is also essential to provide source-code clickback or some other mechanism for identifying which source statements correspond to a particular pattern of behavior.

In terms of selective modification, current tools are singularly lacking in support. What users need are mechanisms for *navigating* through complex source code hierarchies and pinpointing the regions of code that have similar performance behavior. Existing tools do not even provide mechanisms for flagging portions of code as interesting so that they can be revisited or compared with others. Users typically resort to printed source listings and highlighting pens to carry out the activities associated with this set of tasks. Further, current tools do not provide any guidance about what type of code modifications are called for. As users are apt to describe it, tools "don't tell me anything I can actually *use* to go back and fix my code."[17]

Finally, users can be alienated by what a tool does, as well as by what it fails to do. Application developers are trying to explore the behavior of their code, not the behavior of the tool. When a tool shows particular information without indicating why that information is shown or how it might be used to tune application performance, for example, users quickly lose patience. The single most persistent complaint we have recorded is that tools "keep popping up windows I don't need in places I don't like."

3 Applying Human Factors to Tool Design

The human factors issues just discussed can be applied directly to the design of parallel performance tools, specifically in designing graphical representations of performance data. This section discusses how that can be accomplished.

3.1 How Graphical Representations Relate to Task Structure

Most parallel tools have adopted graphical representations in order to portray performance data or other tool information. In many other types of software, graphics have been shown to be useful for a variety of reasons (cf. [10, 28, 1, 4]). In addition to making large quantities of quantitative data coherent, effective graphics encourage the eye to compare and contrast elements, revealing patterns or exposing anomalies in the data that would not be discernible if the representations were numeric or textual. Graphical techniques are also capable of revealing information at varying levels or detail, capitalizing on human familiarity with how the appearance of physical objects changes when they are seen from different distances. Graphics can facilitate the problem-solving process by providing memorable, easily manipulated symbols representing a wide range of concepts, even highly encapsulated information.

The taxonomy of visualization goals proposed by [13] is useful in clarifying how graphics can be exploited by software tools. The authors define the types of activities that can be facilitated through appropriate visualizations:

– **Identify**: establish the collective characteristics by which an object is distinctly recognizable.
– **Locate**: determine specific position, either absolute or relative.
– **Distinguish**: recognize as different or distinct (independent of being able to identify the object).
– **Categorize**: place in specifically defined divisions within a classification scheme.
– **Cluster**: join into (conceptual or physical) groups of the same, similar, or related type.
– **Rank**: give an object order or position with respect to other objects of like type.
– **Compare**: examine so as to notice similarities and differences (independent of ordering).
– **Associate**: link or join in a relationship.
– **Correlate**: establish a direct causal, complementary, parallel, or reciprocal connection.

All of these activities are important in performance tuning, where the user must make sense of large quantities of interrelated performance measurements, draw conclusions about their relative importance, and infer the effects that will occur when behavior is modified.

From the standpoint of parallel performance tools, visualization offers three particular advantages. First, it provides a way to manage the voluminous and complex data associated with parallel program execution. Second, it can capitalize on the user's pattern recognition capabilities. Third, it can assist the user to explore and "interpret'" program behavior by providing alternate views, reflecting different aspects or levels of behavior.

It is not easy to implement graphical displays that exploit these capabilities, however. Consider the problem of visualizing a large and complex set of performance data. The most straightforward techniques result in a series of windows, each showing a different portion of the data,[8] but as previously described, this is overwhelming to most users. The problem is that this type of representation simply uses graphics to expose the dimensions of performance behavior. In applying human factors to tools design, the key is to concentrate on what users need to do. We must structure the presentation of information so that it specifically supports user tasks.

Recall that our discussion of the user task structure indicated three primary requirements:

1. Support for exploring the total performance space. This is particularly important for problem stabilization activities.
2. Support for comparing different aspects of program behavior. This is needed primarily during search space reduction.
3. Support for navigating through complex source code. While navigation is useful during all phases of performance tuning, it is particularly germane to selective modification.

In the sections which follow, heuristics are suggested for integrating each of these features into the kinds of graphical representations used by parallel performance tools. Recall that the goal is to make the operations, and the information revealed through those operations, fit more "naturally" into the user's task structure.

3.2 Facilitating Exploration

Exploration requires mechanisms that allow the user to view program performance from a high level, while still being able to identify anomalous behavior. Three heuristic techniques particularly lend themselves to the exploration of performance visualizations.

1. Add filtering capabilities to zoom operations. Zooming operations are commonly employed to accommodate scalability in performance visualizations. When the program is lengthy or complex, a performance visualization can grow arbitrarily large in size; it cannot be expected to fit in its entirety on the screen. Zooming operations allow the user to increase the apparent magnification of an image, so that the window is filled by what was formerly a small area of the image. The converse operation, panning, reduces the size of the image, making it possible to fit a larger area within the window boundaries.

Human factors considerations indicate that zooming operations should be much more powerful than simple magnification. Instead, they should mimic the human's change in perspective as he/she moves closer to an object by showing more information with each successive zoom (rather than just a magnified version of the same information). Consider, for example, the System Performance Visualization Tool (SPV), developed by Intel for its Paragon series of distributed-memory multiprocessors.[12] The results of its zooming operations are shown in Figure 1. At the upper right is the "zoomed out" view, which portrays the entire system at the level of nodes and interconnets. The nodes are colored to indicate the general level of activity on each CPU. Zooming in on the display (middle window) focuses on a smaller number of nodes and shows specific figures for CPU utilization as well as the amount of traffic with neighboring nodes. Zooming in again (lower right) focuses attention on a particular CPU, revealing details such as memory bus and communication processor usage. This was found to be a very intuitive mechanism for supporting both high-level, overall views of the system and the ability to see a detailed perspective when behavior seemed to be anomalous.

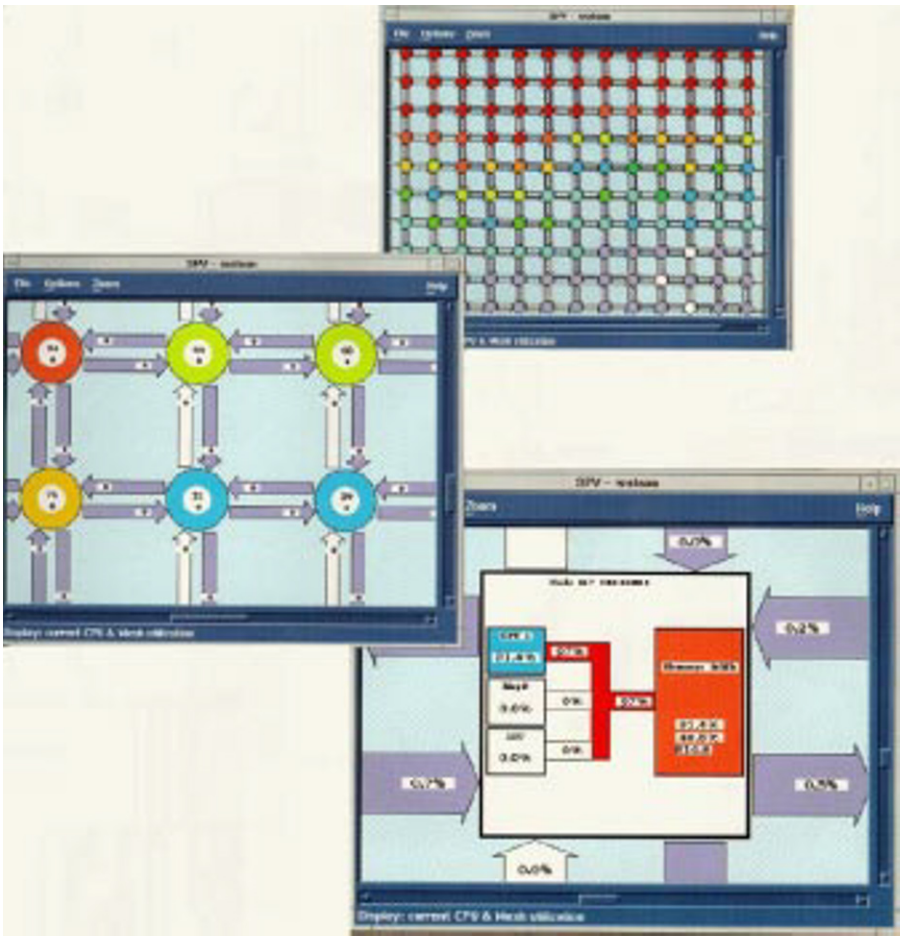

Fig. 1. Effects of zoom operation in **SPV** [12]; each successive zoom reveals more detail about a smaller portion of the system.

2. Make data labels more adaptive. Another problem with exploration is that of providing the user with adequate textual labels. Color, shape, and other graphical characteristics can be very useful for encoding many dimensions of information into a single visualization. With no textual labels, however, the user must being with a mental picture of what he/she is looking for. For example, in the middle display of Figure 1, the user must know ahead of time what the numbers in the circles refer to and recognize the difference between numbers in two different positions (the upper figure refers to compute processor utilization and the lower to communication processor utilization). A label would clarify this.

The problem with textual labels is that they occupy considerable screen real estate and are difficult to position so as not to overlap one another. However, recent human factors studies have shown that by displaying only a small subset of labels within the immediate vicinity of the cursor, it is possible to facilitate exploration of complex displays without overwhelming the user with unwanted detail.[7] In our example, when the user moves the cursor over a node, labels would appear to indicate the meaning of each number. The labels would disappear as the cursor moved to another location.

3. Make it possible to focus attention selectively. Another technique that human factors studies have demonstrated to be effective for exploration is the use of deformation-based focusing, commonly called "fisheye" focus.[24] With this mechanism, the user can magnify just a selective area of a large display — most typically a circular area around the cursor position. For example, Figure 2 shows a callgraph display from the Lightweight Corefile Browser (LCB),[16] a Parallel Tools Consortium project, to which a fisheye distortion has been applied. Only the node labels closest to the cursor position are completely visible.

A recent human factors study proposes that distortion focusing could be even more effective if multiple focal points are used in the same display.[27] In the SPV displays, for example, this would make it possible, to see close-ups of two different nodes simultaneously.

3.3 Facilitating Comparison

As noted previously, much of the user's time is typically spent comparing performance information from different executions, different portions of the same execution, or different types of information about the same area of execution. To facilitate these activities, tool visualizations should underscore the differences in performance behavior so that they are more readily identified and understood by the user.

4. Support display cloning. Current tools do not provide minimal support for comparing across program executions or different portions during a single execution. It is up to the user to bring up two instantiations of the tool and experiment with the settings of both of them simultaneously. In fact, existing tools do little to facilitate comparison of different metrics or types of performance data. Typically, the user may only view one particular region of the program and one metric at a time. **CXperf**, a performance analysis tool developed by Hewlett Packard for their Exemplar series, does allow the user to combine arbitrary pairs of metrics and view them with respect to different granularities of source code (routines, basic blocks).[9] Figure 3 illustrates the display when the user chooses to view CPU time x thread x source routine. Note, however, that to compare

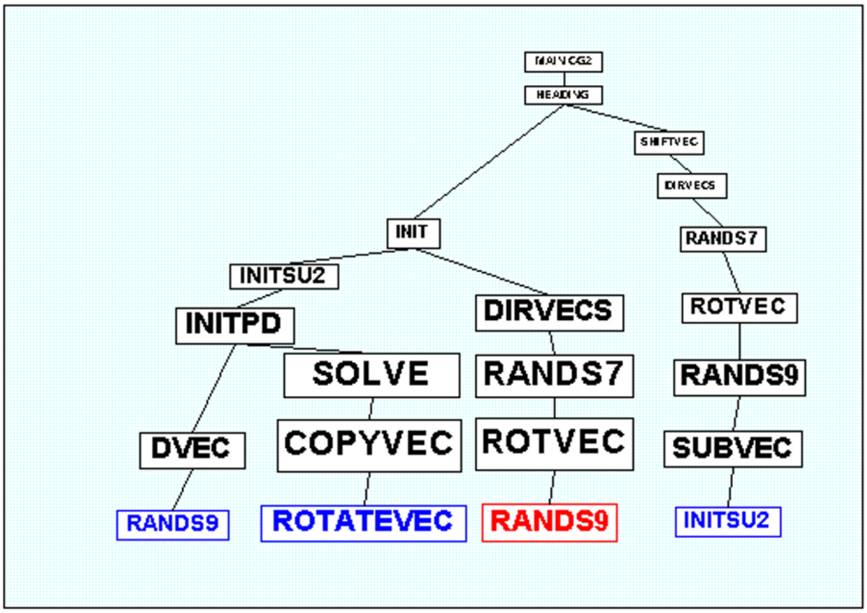

Fig. 2. Effects of deformation focus (fisheye) on a callgraph display from **LCB** [16]; objects near the cursor are magnified, while others remain obscure.

the memory usage or instruction counts for those same threads and routines, the user must switch to a different display. It becomes very difficult to determine cause-and-effect relationships, such as the possible impact of communications congestion on CPU utilization. If the user could clone the window, then change from CPU utilization to communications and the metric, it would be very easy to determine if the two behaviors were interrelated.

5. Use color to enhance discrimination. All too often, parallel tools appear to choose colors based on the ease with which they can be specified. Yet human factors specialists have known for decades that poorly chosen colors actually hinder the user's ability to find targets or recognize patterns (see [3] for a survey of studies on the impact of color on human performance). Given that the visualizations used in performance tools are likely to containvery dense information, coloring is particularly important.

Unfortunately, it is not possible for this paper to show clear examples of how color can be modified to improve usability. We have prepared Web pages listing colors that can be discriminated well on most types of screens and using even fairly simple color models (like that supported by Web browsers). Since color perception is strongly affected by the background and surrounding colors, lists are shown for display against both white and black backgrounds. (See **http://www.cs.orst.edu/p̃ancake/colors.html** and **colors2.html**.) There

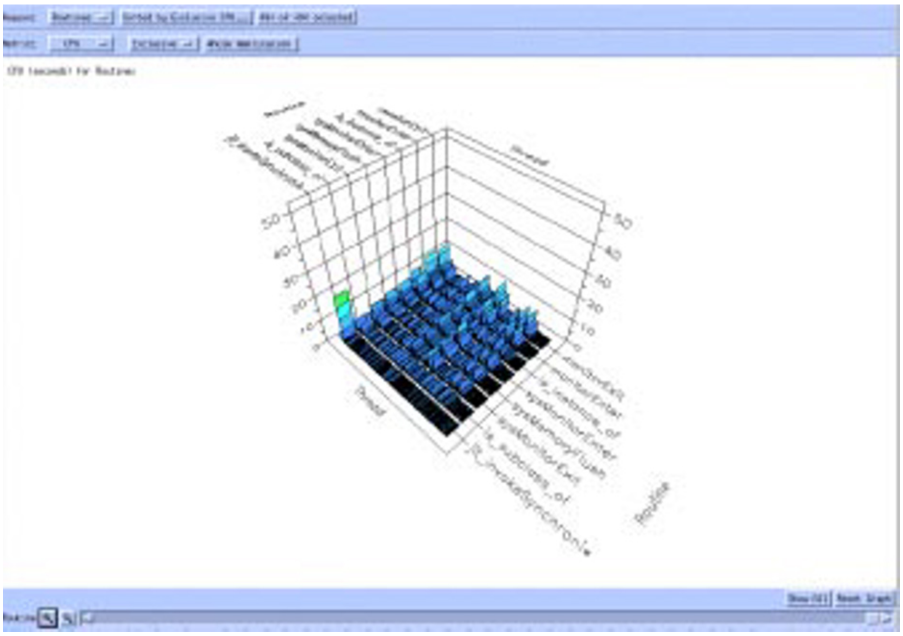

Fig. 3. CXperf [9] display plotting CPU time x thread x code region. Different combinations of metric/location information can be shown, but just one at a time.

are also a number of references that provide guidance on how to use color more effectively; two particularly clear ones are [14] and [26].

6. Make it easier to bring up related information. Current tools are also lacking in their capability for recognizing when different types of information might be related. Consider a typical set of metrics: CPU utilization, communications traffic, memory traffic, etc. Rather than simply presenting data as it is requested by the user, the tool could pre-analyze the range of values along each performance dimension. Then, when a user is examining an area where CPU utilization is high, it would be possible to for him/her to request to see those areas where some other metric is "high."

The **Paradyn**[15] performance monitoring tools from University of Illinois operate in something of this vein, searching through different regions of execution for evidence of bottlenecks. Only a single type of information is presented to the user, however; the tool suppresses what it has determined about other areas that might have performance problems. Human factors studies of other settings indicate that the ability to gain quick access to possibly related information can play a major factor in improving human performance and increasing the user's sense of command over the software.[6]

3.4 Facilitating Navigation

7. Provide a context for navigation. Given the volume of information available through parallel performance tools, it's all too easy for the user to become lost in large displays. Other types of electronic displays have found it useful to maintain some sort of overview representation on the screen at all times, so that the user can keep some sense of where he/she is in terms of the overall information space.

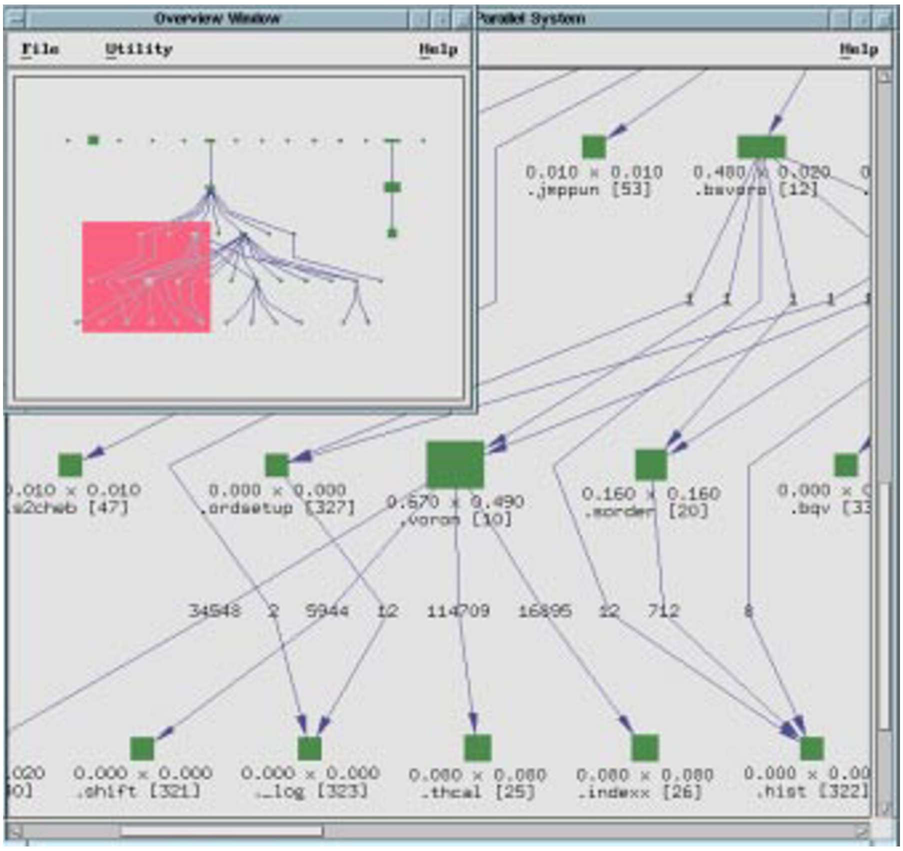

Fig. 4. Xprofiler [11] uses thumb-nail images to help the user navigate through large amounts of performance data.

One performance tool does this through the use of a thumb-nail sketch. **Xprofiler** [11], developed for IBM's SP/2 computers, maintains a very small-scale image that is highlighted to show which portion of the overall program graph is being viewed. As may be seen in Figure 4, the user can quickly grasp a sense of where the detailed information fits into the overall view. An alternate way of providing a sense of context is portrayed in Figure 5. Here, the call graph representation from **LCB** has been augmented with a miniaturized representation of the source code. The user can control which area is zoomed in by sliding the rectangular box that is superimposed on the miniaturized code, displayed along the left side of the screen. The display at the right shows a zoomed-in area of execution, revealing the actual calling structure.

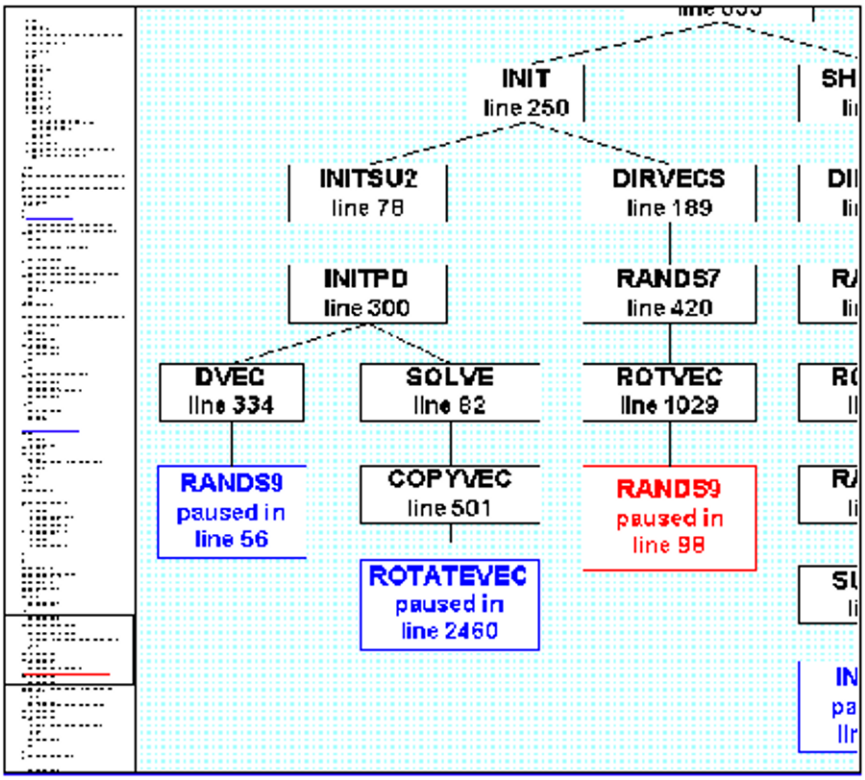

Fig. 5. Example of how navigation in **LCB** could be improved by adding a minitiarized program listing.

8. Show hierarchical relationships explicitly. Most performance tool developers have come to recognize that they must provide links from execution data back to the source code routines or lines where the behavior occurred. However, the typical source-code-clickback mechanism simply pops up a window that has scrolled automatically to the statement or the first line of the corresponding routine. This is not sufficient for most real-world applications, which tend to be very large, organized in complex ways, and developed by whole teams of people over long periods of time. Scrolling to a message-sending line, for example, may provide little clue as to where in the program logic a problem has arisen.

Here, too, other human factors work indicate a strategy for facilitating user interaction with performance tools. Shneiderman's work with "treemaps" (e.g., [25]) indicate that rectangular maps can be very effective in representing hierarchical structures. A containment relationship applies; that is, information that is subordinate to other information is embedded within its superordinate's area. Figure 5 applies this concept to the representation of the source code structure from Figure 4. Note that very little space is needed to represent the call tree, but no information is sacrificed. Like trees, treemaps require that the informa-

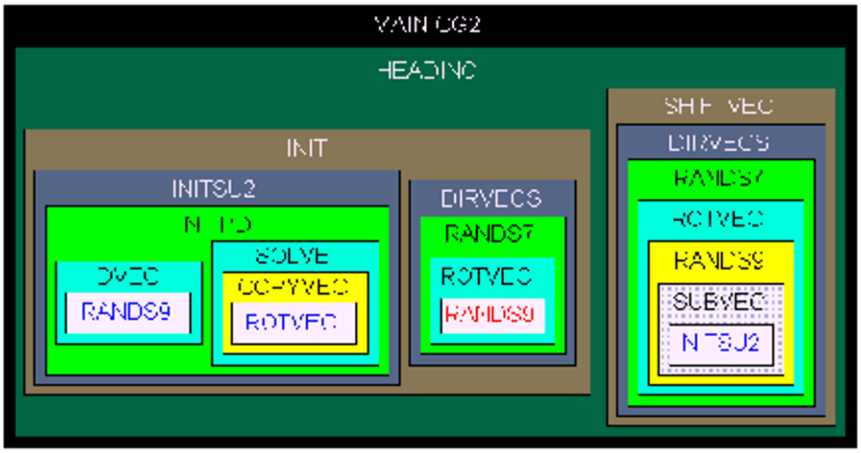

Fig. 6. Example of how navigation in **LCB** could be improved by making the hierarchical relationships of the source code explicit.

tion to be represented contain no cycles or recursion. The approach could also be combined with zooming or fisheye techniques described earlier, in order to provide control over the amount of detail portrayed.

9. Support user-inserted "landmarks". A final navigational aid is the addition of mechanisms whereby a user can "mark," or visually flag, regions of the display. This does not affect the data in any way; it simply makes it easier for the user to return to an area for later comparison or study. Consider the displays shown in Figures 3 and 4, or any of the traditional time-line diagrams so common in performance tools. The user ultimately navigates through a considerable information space, so returning to a point typically requires a great deal of searching. If the tool allowed the user to superimpose visual landmarks (such as small colored flags) directly on the display, it would be very simple to re-locate those positions.

4 Conclusions

Parallel tools suffer in comparison to the usability of software on personal computers because the resources available for development are much greater in the desktop world and the problems to be solved are much less complex. Many usability issues remain unresolved for parallel tools, which are often the implementation of an untried solution. As a result, the ways in which performance tools can be applied effectively during application development remain obscure to the user community.

The developers of parallel tools appear unaware of many basic human factors studies which could help them improve the usability of their visualizations and interaction mechanisms. Nine examples were presented in this paper, with descriptions of how each would improve a typical performance tool scenario.

User acceptance of parallel tools will not increase appreciably until such tools are usable within the framework of typical application development strategies. This will require that tool developers come to an understanding of how experienced users go about developing and tuning large-scale applications. The goals of this effort should be:

– to identify successful user strategies in developing real applications;
– to devise ways to apply knowledge of those strategies so that tool functionality can be presented in an intuitive, usable, and familiar manner; and
– to use this functionality in the development of new tools.

Tools that do not mesh well with user goals and task structures are not considered by users to be worth the time that must be invested to learn them. We will not see tools appreciated properly until they can address the human factors issues associated with parallel performance tuning.

References

[1] Card, S., J. Mackinlay and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think*, Morgan Kaufman, 1999.

[2] Casavant, T. L., "Tools and Methods for Visualization of Parallel Systems and Computations: Gues Editor's Introduction," *Journal of Parallel and Distributed Computing*, 1993. 18 (2): 103–104.

[3] Christ, R. E., "Review and Analysis of Color Coding Research for Visual Displays," *Human Factors*, 1975, 17 (6): 542-570.

[4] Cleveland, W., *Visualizing Data*, Hobart Press, 1993.

[5] Curtis, B., *Human Factors in Software Development: A Tutorial*, 2nd Edition, IEEE Computer Society Press, Washington, DC, 1985.

[6] Czerwinski, M., et al., "Visualizing Implicit Queries for Information Management and Retrieval," *Proceedings ACM Conference on Human Factors in Computing Systems (CHI'99)*, 1999, pp. 560–567.

[7] Fekete, J-D. and C. Palisant, "Excentric Labeling: Dynamic Neighborhood Labeling for Data Visualization," *Proceedings ACM Conference on Human Factors in Computing Systems (CHI'99)*, 1999, pp. 512–519.

[8] Heath, M. T., A. D. Malony and D. T. Rover, "The Visual Display of Parallel Performance Data," *IEEE Computer*, 1995, 28 (11): 21–28.

[9] Hewlett-Packard Corporation, *CXperf User's Guide*, Publication B6323-96001, available online at
`http://docs.hp.com:80/dynaweb/hpux11/dtdcen1a/0449/@Generic_BookView`, 1998.

[10] Horton, W., *Illustrating Computer Documentation: The Art of Presenting Information Graphically on Paper and Online*, John Wiley & Sons, 1991.

[11] IBM Corporation, *IBM AIX Parallel Environment: Operation and Use*, IBM Corporation publication SH26-7231, 1996.

[12] Intel Corporation, *System Performance Visualization Tool User's Guide*, Intel Corporation, publication 312889-001, 1993.

[13] Keller, P. R. and M. M. Keller, *Visual Cues: Practical Data Visualization*, IEEE Computer Society Press, 1993.

[14] Marcus, A., *Graphic Design for Electronic Documents and User Interfaces*, ACM Press, Tutorial Series, 1992.

[15] Miller, B. P. *et al.*, "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer*, 1995, 28 (11): 37–46.

[16] Muddarangegowda, M. and C. M. Pancake, "Basing Tool Design on User Feedback: The Lightweight Corefile Browser," Technical Report, Oregon State University, available online at
`http://www.CS.ORST.EDU/ pancake/papers/lcb/lcb.html`, 1995.

[17] Pancake, C. M., unpublished interview notes from field studies and group interviews at user sites in the U.S., 1987–1999.

[18] Pancake, C. M. and D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Researchers?" *IEEE Computer*, 1990, 23 (12): 13–23.

[19] Pancake, C. M. and C. Cook, "What Users Need in Parallel Tool Support: Survey Results and Analysis," *Proc. Scalable High Performance Computing Conference*, 1994, pp. 40–47.

[20] Pancake, C. M., "Establishing Standards for HPC Systems Software and Tools," *NHSE Review*, 1997, 2 (1). Available online at `nhse.cs.rice.edu/NHSEreview`.

[21] Pancake, C. M. "Exploiting Visualization and Direct Manipulation to Make Parallel Tools More Communicative," in *Applied Parallel Computing*, ed. B. Katstrom et al., Springer Verlag, Berlin, 1998, pp. 400–417.

[22] Pancake, C. M., M. L. Simmons and J. C. Yan, "Guest Editor's Introduction: Performance Evaluation Tools for Parallel and Distributed Systems," *IEEE Computer*, 1995, 28 (11): 16–19.

[23] Pancake, C. M., M. L. Simmons and J. C. Yan, "Guest Editor's Introduction: Performance Evaluation Tools for Parallel and Distributed Systems," *IEEE Parallel and Distributed Technology*, 1995, 3 (4): 14–19.

[24] Sarkar, M. and M. H. Brown, "Graphical Fisheye Views of Graphs," *Proceedings 1997 IEEE Symposium on Visual Languages*, 1997, pp. 76–83.

[25] Shneiderman, B., "Tree Visualization with Tree-maps: A 2-D Space-Filling Approach," *ACM Transactions on Graphics*, 1992, 11 (1): 92–99.

[26] Thorell, L. G. and W. J. Smith, *Using Computer Color Effectively, An Illustrated Reference*, Prentice Hall, 1990.

[27] Toyoda, M. and E. Shibayama, "Hyper Mochi Sheet: A Predictive Focusing Interface for Navigating and Editing Nested Networks through a Multi-focus Distortion-Oriented View," *Proceedings ACM Conference on Human Factors in Computing Systems (CHI'99)*, 1999, pp. 504–510.

[28] Tufte, E. R., *Visual Explanations: Images and Quantities, Evidence and Narrative*, Graphics Press, 1997.