UCRL-CR-123018
B310550

Involving Users in the Design Cycle for Parallel Tools

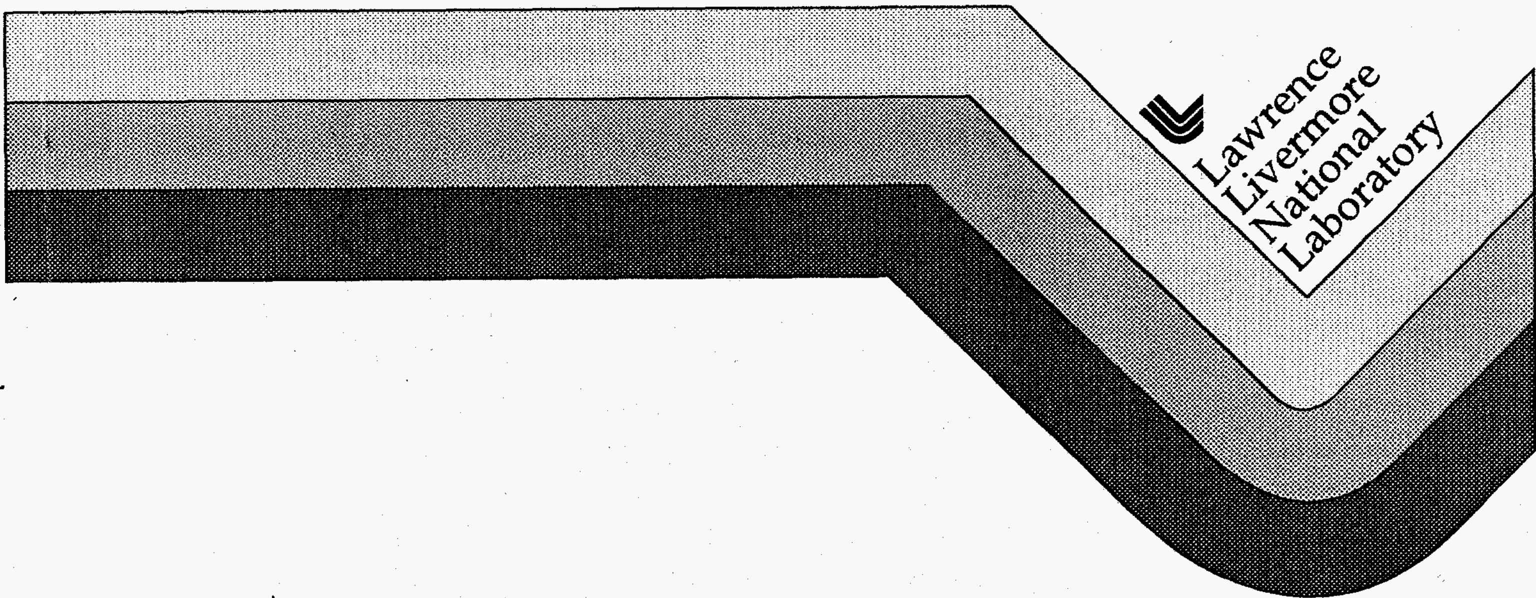
Cherri M. Pancake

RECEIVED
FEB 06 1996
OSTI

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

January 31, 1995



MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DT

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Involving Users in the Design Cycle for Parallel Tools

Project Report

Submitted to:
Sam Coleman
Lawrence Livermore National Laboratory

Submitted by:
Cherri M. Pancake
Department of Computer Science
Oregon State University

January 31, 1995

Summary

Parallel programmers do not use software tools, in spite of the fact that parallel application development is a difficult and time-consuming task that could benefit from tool support. It has become increasingly clear that the simple availability of elegant, powerful software tools employing the latest technology is not enough. Usability is the real key to success; users simply do not adopt tools that fail to respond to their needs.

Research in the area of usability engineering indicates that five design principles can have significant impact on parallel tool usability:

- tools must be based on demonstrable user requirements
- actively involve users throughout tool design
- minimize tool complexity to reduce the learning curve
- support the tool across multiple machine platforms to amortize the user's investment
- employ iterative refinement techniques to improve tool usability

Those principles served as the starting point for a Parallel Tools Consortium project to develop a tool that will help users determine the final state of a program that crashes or is terminated forcibly. Carried out over a period of ten months, the project involved the collaboration of tool researchers, and implementors, and users. This report describes how user-centered design techniques were applied to ensure that the tool would provide simple, intuitive support for the programmer's task.

Users were recruited for the project working group so that they could have direct input to design decisions, even in the earliest sets of user trials. Additional feedback was acquired from a broader user base, in three distinct phases. These were spaced out over a period of six months so that feedback could be analyzed, then applied to refine the tool before the next trial. In some cases, user input kept us from investing substantial effort in features that would not have been used or appreciated. In others, feedback showed us where our conceptions of usefulness did not quite align with those of the user community.

There is no doubt that the LCB tool, as it exists today, is dramatically different from the initial design conception. According to the users who have experimented with it, those differences have greatly improved its usability.

Introduction

Although the "build it and they will come" mentality has dominated the design of scientific software for some time, it is increasingly clear that this attitude is responsible for the failure of many software systems [28]. Software users are no longer willing to put up with products that are difficult to learn or use [10]. This is a positive direction. As William Howell, Executive Director of Science for the American Psychological Association writes, "Much of the improvement in software is attributable to research and knowledge supplied by research in human cognition and behavior, expertise that the computer scientists who designed the earlier systems never realized they needed until consumer discontent became impossible to ignore" [11].

Creating an elegant, powerful piece of software does not guarantee that it will be used. Usability, the ultimate key to software adoption, encompasses a variety of factors, including how easy the software is to learn, its efficiency for advanced users, how easy it is to remember even for infrequent users, its "forgiveness" of user errors, and how pleasant it is to use. These are human-oriented factors, requiring that the software interface designer know and understand the target users, the set of tasks they will want to perform, and the logical models they will use in applying the software to those tasks [23, 21, 3]. The central tenet is "know and involve the users" -- with emphasis on the plurality, heterogeneity, and even cultural diversity of users [17, 22].

Unfortunately, very few computer scientists have any formal training or expertise in cognitive psychology, ethnology, or even the subdiscipline of computer science known as HCI (human-computer interaction). One consequence is that only software targeted at mass markets like home computing shows real evidence of being designed to please the consumer. For example, metaphors and symbols offer a powerful means of conveying relationships and actions to users of varying levels of expertise. Users are familiar with the rectangular desktop as a symbol for a workspace, or a trashcan to represent the delete function; similarly, a rolodex or a file folder conveys meaning without added clutter in an environment based on a desktop metaphor. Most metaphors, symbols, and icons have been developed for a general home or business environment, however. No comprehensive metaphors have been developed for dealing with the scientific research community; in fact, there are few success stories even in the more long-standing, data-oriented disciplines, such as library science [32].

The current reality is that there is remarkably little understanding of human factors requirements for software, particularly software targeted at scientists [7, 9]. The problem is glaring in the area of tool support for parallel programming [29, 30], and it has had serious repercussions on tool usability.

Current State of Tool (Dis)Use

When tool developers were brought together with tool users at the 1993 DARPA/NSF Workshop on Parallel Computing Systems, they concluded that "A lot of smart people are developing parallel tools that smart users just won't use." Surveys and interviews have found that most users prefer hand-coded instrumentation over current parallel tool offerings. Kuehn estimated that some 99% of the technical programmers at his institution rely on PRINT statements in spite of the availability of debuggers and performance tools [14]. Our studies have revealed rates almost as high in a variety of research and industrial settings [31]. Even within so-called "tool-disposed" groups, some 40% avoid parallel tools [25]. Among people who do use tools, a surprisingly large number develop in-house or personal tools rather than use those that are available commercially or in the public domain [25, 31].

Parallel tools have the potential for greatly assisting the application development process, and are probably more important to parallel than to serial programmers [29, 18, 31]. Hand-coded instrumentation is a reliable (albeit tedious) method for gathering the information needed to debug or tune serial programs, but I/O in the parallel environment can seriously perturb timing relationships, masking errors and performance bottlenecks or provoking new ones. Technological factors, such as the time delay involved in offloading the contents of I/O buffers and problems in obtaining reliable global timestamps, can result in improperly ordered or lossy information. The complexity of parallel computers and their susceptibility to small variations in the run-time environment also suggest that mechanisms for gathering information need to be of an accuracy and efficiency beyond the scope of most application programmers.

Why, then, are parallel tools so under-utilized? It can be argued that there is some tradition of antipathy to tools, and that these are unpopular among serial programmers, too. However, a recent survey contrasted tool use among comparable populations of parallel and serial users. Serial programmers were found to be significantly more likely to employ tools than were parallel programmers, while almost 80% of those with vector programming experience reported tool use [4]. In a number of recent workshops sponsored by such diverse groups as ARPA, NSF, ONR, NASA, ACM, and HPCCI, users have complained that tool developers are more concerned with conducting interesting research or adding ''bells-and-whistles'' than with creating tools that will actually be used. Although this attitude is unnecessarily cynical, it does reflect the current perception that parallel tools are developed under supply-push rather than demand-pull economics [26]. That is, tools are not being designed to match user requirements and user strategies for developing parallel applications.

> Tool developers do not ask users the "right" questions. If a tool designed for a particular programming task isn't applied to that activity, the developer asks the user why it's not being used. What needs to be asked instead is how the user does go about the task, why it is performed in that way, and what he/she does to simplify or streamline the effort [25:40].

The dominant motivation for current tool design appears to be the exposing of available technology, whose intrinsic value is obvious to tool researchers and implementors. Unfortunately, it is not clear to the user community that such technology can be applied easily or effectively to their program development strategies. For the most part, they decline to make the attempt.

How can technical programmers be encouraged to use parallel tools? This is largely a human factors issue. Scientists and engineers are interested in software tools to the extent that they contribute to scientific research activities, not as an end in themselves. If a tool requires a lengthy training period, is the source of repeated frustration, or fails to yield useful results, it simply will not be used. Yet this is precisely the situation with parallel tools. Studies indicate that

> [Parallel] tool information often is presented in a fashion that reflects the tool's organization, rather than the logical patterns employed by users. The information sought by the user may be available only indirectly, via multiple operations or through assimilation from multiple sources. At best, the tool is considered clumsy and hard to learn; at worst, the user assumes it cannot provide the desired information.... [T]he audience for parallel tools is made up of scientists, engineers, and other technical programmers. They do not approach programming in the same way as their computer science counterparts, nor are they tolerant of tools that are complicated or non-intuitive [25:40].

Today's tools may be technologically sophisticated, but they lack the critical ingredient: usability.

The Basis for Tool Usability

The concept that usability should be the driving factor in software design and implementation is not particularly new; it has appeared in the literature under the guises of usability engineering, user-centered design, and iterative design [23, 17, 24, 2]. Definitions of usability vary, but typically involve the notion that usability is some combined function involving both the operative aspects of the software itself, and the training or expertise embodied in the user (e.g., [1]). There is no firm consensus on what methodology is most appropriate for achieving usability, nor on the frequency with which users should be involved in design decisions (cf. [12, 13]). What is clear is that usability can only be accomplished with the active participation of actual users.

It is instructive to apply the lessons from experimental usability engineering to the subject of parallel tools. First and foremost is the notion that meaningful, useful software is driven by concrete

needs. Such requirements can be identified only by soliciting input directly from the user community. Specifically, a tool will not be useful unless it facilitates tasks that the user already does, and that are time-consuming, tedious, or error-prone when performed manually. If, instead, a tool's design is driven by the kinds of support that tool developers are ready or able to provide, it will miss the mark.

Ease-of-learning is a key factor for attracting users. The time a user invests to learn a tool will not be warranted unless it can be amortized across many applications of the tool. In addition, lack of regular use may force the user to re-learn a tool many times over. Tool complexity is therefore a dis-incentive, not just for new users, but also for those who have not used the tool for a period of time. The short lifespan of most parallel computers exacerbates this problem. Like it or not, most parallel programmers will end up migrating their applications across several machine platforms over the course of time. The investment in learning a tool will probably not be warranted unless the tool is supported on more than one platform, and behaves in a consistent way across platforms.

Once a tool is familiar to the user, other usability factors begin to dominate. An important concept of user-centered design is that *usefulness* and *ease-of-use* can be ensured only if users are actively incorporated into the software design cycle. Since the implicit goal of any tool is to increase user productivity, *throughput* is also important. This measure reflects the time and effort required to accomplish the specific tasks to which the user applies the tool, and includes the negative influence of frequent errors or the difficulty of making corrections. Tool users are the only ones who will have the insight needed to accurately identify features which represent potential sources of confusion.

The tradition of soliciting user feedback only during the very early and very late stages of development is not adequate for assessing and improving usability. During early stages, the design is too amorphous for a user to grasp completely, while during late phases such as alpha testing, the software structure has already been solidified in ways that may impede usability. Different types of usability problems will be caught and corrected at different points in the design cycle. Moreover, it is important to observe at least some users on a sustained basis. The introduction of any computerized tool does more than replace a sequence of manual operations by automated ones [15, 20, 8]. Only by observing and analyzing how users interact with the tool as they become familiar with it can designers confront the anthropological and sociological issues at stake.

In summary, the basis for tool usability lies in how well and how easily a tool responds to user needs --- something that can only be determined with the help of actual users.

Involving Users in Design

The most accepted way of involving users in the design of software is through a process of iterative refinement [10, 22]. As shown in Figure 1, representative groups of users are exposed to the software at various stages in development, in a variety of testing and interviewing situations. The user

feedback obtained is then used to refine the design, with the result serving as input to the next iteration of the design process. This allows the developer to tune and tailor the tool in response to user reactions.

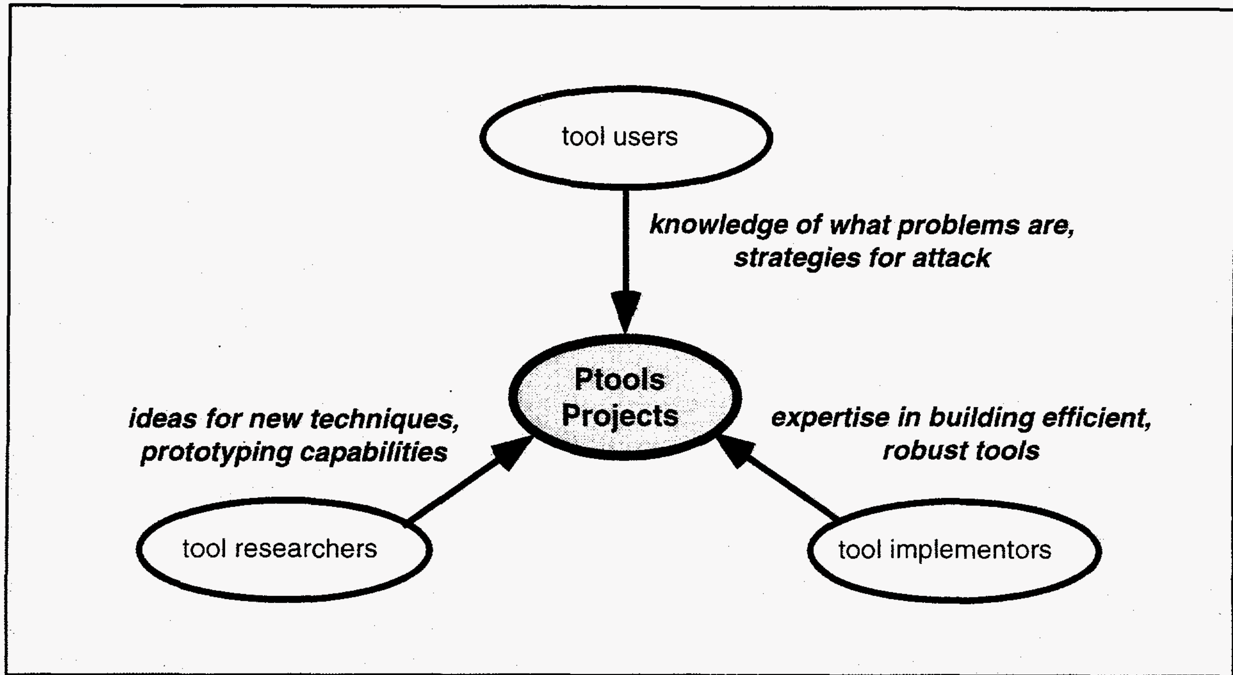


Figure 1. Usability engineering applied to tool design

The last few years have yielded considerable progress in tool technology and in efforts to define standards for parallel languages and operating environments. At the same time, the user community has accumulated a considerable experience base in parallel programming. If tool developers and users collaborate, it should be possible to leverage these results in order to arrive at tools that not only do things users want to do, but also are implementable across a range of parallel and clustered systems. The parallel tools community is a varied one [26]. The largest concentration of tool researchers is currently at academic institutions, although some will be found in federally funded and industry research centers. Implementors of tool products typically work in the industrial sector. Users, on the other hand, tend to be clustered in the national laboratories, with increasing numbers in the in technical areas of industry, such as seismic imaging, pharmaceutical engineering, and aerospace or automotive engineering.

Forging a collaboration between such diverse and professionally segregated groups is not a simple task. Yet each group brings expertise that is not duplicated in the others. Historically, tool researchers have provided the impetus for technological advances, yielding new theories, paradigms, techniques, and representational models. The implementor group is the repository of most expertise in developing robust, efficient implementations for specific real-world parallel computers. The third group, users, forms the customer base for parallel tools and thereby establishes the basic requirements. They also embody more

There has been virtually no research in user interface requirements among scientific programmers [29, 30], so we did not have a pool of information upon which to draw. However, a recent analysis of network usage in the oceanography community [9] underscored the fact that a generic "scientist user" model is too simple to describe requirements for network software. More attention should be paid to how scientists in the disciplines actually work -- what their social structure looks like and what kinds of access they need to access what kinds of resources. Choices should be based on grounded knowledge of patterns within specific disciplines. With that in mind, the original working group members solicited the participation of other user organizations, including the National Center for Atmospheric Research, BioNumerik Pharmaceuticals, the U. S. Navy's Ocean Surveillance Center, and the San Diego Supercomputing Center. Each of these turned out to be instrumental in identifying particular problem areas. Furthermore, the group relied on the evidence that one participant had gathered, in a series of earlier surveys and user-site interviews [31], concerning the patterns with which parallel applications are developed by scientific programmers.

How the Lightweight Corefile Browser Addresses Usability Requirements

The Lightweight Corefile Browser (LCB) was one of the first collaborative projects adopted by the Parallel Tools Consortium. The idea for LCB originated among the user community. In a number of discussion sessions (some organized by us on behalf of HPC vendors, but others in open forums such as Supercomputing '93 and Intel User Group meetings), users complained that their chief concern was the fact that there is no current support for answering a simple -- and crucial -- question:

Why did my program crash?

They emphasized that, in particular, they need help in answering this question without the need to execute their program again (as would be the case, for example, with most interactive debuggers).

In sessions where users were asked to prioritize the urgency of their needs for tool support, they ranked this requirement as number one. They elaborated on the need, revealing that they wanted to be able to answer a range of more specific questions, including:

- Which process was the culprit?
- Why did the culprit fail?
- How far did the culprit get before it failed?
- How did the culprit arrive at that point?
- Was the culprit the only process executing that routine? If not, which others did?
- How deeply into the source program had each process executed?
- Which other routines were actively executing when the program terminated?

They also indicated that the answers needed to be phrased in terms of source code routine names and line numbers, rather than machine addresses.

In one session (a roundtable discussion at Supercomputing '93), it was pointed out by an industry developer from the audience that such information was already available, embedded in corefiles. Users made it clear that corefile mechanisms were inadequate, for several reasons. First, corefiles are not available on all systems; even when available, they aren't always produced (for example, when the user forcibly terminates a program that appears to be deadlocked or caught in a loop). Second, the corefile mechanism varies widely, sometimes generating data for just the failing processor, sometimes for a subset of processors, sometimes for all processors. Third, corefiles are wasteful of storage space, occupying as much as several Gigabytes for a full core image of numerous processors. This seems totally unnecessary to users, who want access to only a small amount of key information: the current value of the program counter, the contents of invocation stack frames, and an error or reason code. Fourth, it is difficult or impossible for a user to extract the information desired from corefiles. Moreover, the user has no real alternative. Hand-coded instrumentation, in particular, fails, since the user is unable to predict the point of failure, and since the data most needed is often left in the buffers rather than being flushed to disk files.

The Ptools project addressed these issues by proposing a graphical tool. It was designed to provide a simple and convenient way of examining the final state of a parallel program that terminates abruptly (e.g., one that crashes, or that the user is forced to terminate when it appears to hang or to behave in some undesirable way). The goal was to furnish the programmer with a global, high-level view of the program's dynamic calling structure. Such a view could be constructed from a minimal amount of data recorded by the operating system or some other run-time monitor (e.g., an interactive debugger). That data is referred to as a "lightweight corefile," but it need not be stored in a physical file.[1] An implicit goal of the effort was to demonstrate that collaborations of users with developers would in fact yield tools that are more responsive to user needs. Our institution, Oregon State, was charged with coordinating that collaboration. The remainder of this report discusses how we sought input from user, applied that input to the design process, and verified the tool's usability.

One of our first contributions to the project was to define the basic requirements described above; this was carried out primarily in the user sessions that gave rise to the project concept. For our own guidance, we also established and prioritized specific objectives associated with each of the four usability factors outlined previously. Table 1 presents the objectives that were identified as important if the tool was to be adopted quickly and widely. Since there are no existing tools of this nature, it was absolutely essential that our solution be easy to learn. We interpreted that as meaning the conceptual model presented to the user should encapsulate an intuitive notion of what it means to say that a parallel program is at a specific point in its execution. Any names or messages we used to report that state should be consistent with user terminology, rather than computer science terminology (e.g., "call graph" rather than "stack frame records").

---

[1] Information on how LCB is designed to interact with the underlying system is provided elsewhere, together with a user guide for the tool [LCB].

The second priority was ease-of-use. Since debugging activities tend to be concentrated in sporadic intervals [30, 31, 14], we could not rely on learned familiarity. Essentially, we must assume that users would not retain knowledge of tool operations from one invocation to another. This also meant that user errors would be likely, and that we must provide mechanisms for allowing the user to recover from any mistakes [16].

Factor	*Objectives*
Ease of learning	• provide intuitive conceptual model • make terminology and operations consistent
Ease of use	• allow infrequent users to return to tool without re-learning • fast recoverability from any possible user errors
Usefulness	• help user understand how to apply tool to new task scenarios
Throughput	• reduce likely frequency of errors • tool must be efficient enough to increase user productivity

Table 1. Prioritization of usability objectives for LCB project

Usefulness was established as being of somewhat less priority, since users had already made it clear that they felt in desperate need of tool support of this type. We felt that the most important consideration for this factor was that users be able to understand how to apply the tool to a new task. For example, if a user understood how to invoke LCB to see which process caused a program crash, it should be simply for him/her to apply the tool to a different problem, such as determining where a deadlock had occurred.

Finally, throughput was relegated to the lowest priority of the four factors. It was still important, however, that the tool itself not detract from user throughput by being error-prone. We also stipulated that efficiency -- in terms of the amount of time required to invoke the tool and apply it to a task -- must be reasonable, or the tool would not be used.

Applying User-Centered Techniques in Designing a Parallel Tool

In the early stages of design, the primary activity was to identify an appropriate conceptual model (representation) that would map well to the user's mental model of the dynamic structure of a parallel

program. Our previous work in this area had established that scientific programmers relied heavily on the concept of a "call graph" in conceptualizing both static and dynamic program structure [27]. That research had culminated in the development of a graphical representation, the Program Phase Tree (PPT), for animating the progress of parallel program execution. A PPT was created using a graph editing tool [5] so that the user could modify a simple call graph to reflect his/her logical model of program structure. Users seemed to immediately understand the tree-like representation, and they commented favorably on the ability to observe the program state in terms of familiar program names and relationships. We ultimately dropped the project, however, because users did not favor having to learn a graph editing tool simply as a preparatory step for what they really wanted to see: information on the program's dynamic behavior. Furthermore, the PPT representation relied on colored lines to portray the activity of each process, a technique which did not scale well beyond sixteen processes.

LCB project personnel also reviewed a variety of other tools to observe existing methods of representing dynamic structure. Summarized in [19], these were found to consist largely of stack tracebacks that employed precisely the type of cryptic hexadecimal notations users had complained about. After discussing the possibilities informally among the working group, it was decided to prepare an initial "paper prototype" that was based on the PPT, but attempted to overcome its problems with scalability and the user effort required to define the structure.

The pre-prototype design, drafted at Oregon State, was presented publicly at the Parallel Tools Consortium General Meeting in June of 1994. A breakout session devoted to LCB was attended by some 30 persons, most of them representing the user community. First, we went over the user requirements on which the project was based. In response to issues raised by some of the tool developers active in the working group, users were asked to verify that source routine name and line number were enough data to identify the location of a process. Users confirmed that they did not need to see information on data storage, argument values, loop counters, or anything else as they made a "first cut" at determining why the program had malfunctioned. They wanted to know the reason for failure and the relative location of processes, but nothing more.

Users were then asked to look at pictures of the prototype design, discuss what the representation conveyed to them, what types of operations they would expect to carry out, what changes would make the information more obvious to them, and so forth. The first reaction was startling in its simplicity: in many cases of program failure, there's really no need for anything as complex as a graphical display. Users were firm in stating that for many errors, simply knowing the location where the culprit process faulted is sufficient for them to correct the problem. This idea had not occurred to any of us on the working group, but it clearly had profound implications for ease-of-use and throughput.

Further discussion elicited the fact that it was imperative we provide a text-based, command-line version of the tool in addition to a graphical version. There were several reasons for this. First was the consideration of efficiency and simplicity; why spend the time bringing up a graphical tool, when a

command-line tool could return a small amount of key data in just a fraction of that time? Second, in some situations -- such as dial-up access from home or a hotel -- graphics capabilities are simply not available, but a command-line is. Finally, it was pointed out that many users simply do not like graphical tools, and if we insisted on graphics we would lose that potential audience.

The call graph display was generally received with favor, but users questioned the ability of a display labeled with textual routine names to scale up to hundreds of nodes. They suggested that we also provide some sort of zoomed-out version that would make it possible to capture the entire calling structure (or most of it) within the visible portion of the window. It should be possible to zoom in and out quickly, though, so that the user could examine details, check the general surroundings, then look at another detailed area. Another suggestion was that we make the overview graph the initial display as the tool was first invoked. Our initial design specified that the culprit display (showing the calling sequence for the routine where failure occurred) be the default, assuming that the user was most concerned with this information. The reasoning was that user want to see the overall state of the whole program when the tool comes up, not just the small portion of state attributable to the culprit process. Also, a culprit need not be present; in such cases, it only makes sense to show the overview.

We also used the opportunity to informally canvas users on issues that have impact on the design of other displays. The terms (notably "culprit process", "reason for failure", "call graph", and "lightweight corefile") did not appear to present any problems of ambiguity or obscurity. In fact, users were much less concerned than the developers about one issue.. On different systems, the "culprit" might be known as a node, a processor, a process, a task, a thread, or some combination thereof. Where we felt it would be confusing for different incarnations of the tool to show "0:0" (for node/process ID), "34657" (task ID), "1534 t1" (process/task ID), etc., users were indifferent because they (a) typically use just one system at a time, and (b) are already used to its method of representing execution. The use of color-coded "buckets" to represent relative level of activity in each routine (e.g., one color for routines where 1-4 processors are active, another for 5-10) also turned out to be less complex that we had anticipated; users commented that an approximate notion of relative activity was more than sufficient, and they understood that they would be able to see the exact count on demand.

All in all, the session was useful on several counts. First, it confirmed that we had correctly interpreted user requirements for the tool, in terms of what kind of information it should present. It underscored the importance of presenting data at more than one level of detail (call graph vs. overview). It also revealed that our notions of default/principal views was not necessarily that of our users. It eliminated work that we had expected to be difficult (e.g., ways to normalize variations in the ways that process-IDs are represented, chromographic display of precise count on process activity). Finally, we walked away knowing that we were on track, and understanding what our next steps should be. We had also managed to enlist promises of support from several of the user organizations, who joined the working group. The next two months were spent implementing the command-line version and a prototype of the graphical version, incorporating all of the recommendations from the first round of interactions with users.

Prototypes of the command-line and graphical interfaces were tested at the National Center for Atmospheric Research in August of 1994. The availability of a prototype with which users could interact, but which was still easy to modify, proved to be of inestimable value. We prepared users[2] for the trials by explaining that the tool was intended to be invoked after a program had crashed or been terminated explicitly by the programmer, and that the act of terminating the program had "resulted in a minimal-sized corefile, which must be named as input file on the command line." Each user was then asked to apply the tool to determine where and why the program had crashed, verbalizing what he/she thought was going on (a procedure known as protocol analysis [6]). We observed their interaction, took notes, and questioned them only for purposes of clarification.

Other than a few simple suggestions about format, the users were satisfied with the command-line version. In fact, virtually all of them made a point of complimenting us on providing this simple, fast procedure for obtaining the most rudimentary information. One tester confessed that he would probably use the command-line version in preference to the graphical one, even if the latter could give him more information.

For the graphical version, the first reaction was to the layout of the call graph itself. For both the overview and call graph displays, we had opted to display the tree horizontally (i.e., with the root, or main procedure of the program, at the left-hand side of the window). Users liked the ability to change the orientation back and forth from horizontal to vertical, but they uniformly recommended that we make vertical layout the default because it offered a more intuitive reflection of a call graph ("that's the way we draw it").

One issue that had preoccupied us during the prototype implementation was the fact that a given program routine might occur multiple times in the graph. Although we collapse call chains containing precisely the same sequence of routines (or common prefixes), a routine that has been invoked from two different locations will appear as two nodes. This was a reflection of the graph display software, which requires acyclic graphs. We were particularly concerned with the possibility of deeply nested recursions, which would result in long repeated strings of nodes -- as opposed to, say, a node with an arrow indicating a cycle. The response of the users was surprising. First, they simply assumed that routines would be repeated and did not necessarily understand that there was any alternative for this. Second, when they comprehended how this point would effect the representation of recursive invocations, they stated that they actually preferred to see a long chain (even if it were 50 nodes long!) because that visually reinforced the idea that the routine had deep recursion (which was likely to be an error).

Users also commented favorably on the fact that the cursor changed shape whenever it entered a window (the overview, culprit view, and call graph view) where things could be selected. All of them

[2] These included several "meta-users" (former users who now help other users enable their applications) from NCAR, as well as one user from an industrial site.

noticed the fact that the message area instructed them to click on a node for more information. They liked this mechanism for indicating that additional operations were possible.

Another feature they commented on was the fact that the message area presented them with "hints" about the node the cursor was positioned over. In the overview, for example, the message indicated precisely how many processes were active in that routine. In the culprit view, the message changes to indicate the line number where the corresponding procedure call occurred. Users seemed to have no trouble at all interpreting the purpose or the meaning, and they liked the context-sensitive nature of the messages.

Interestingly, they objected most vocally to some of the Motif-conforming aspects of the prototype. For example, they found the quit-confirmation dialog to be extremely annoying. In general, they objected to any time that Motif policies dictated the addition of an extra dialog or keystroke.

As on the previous occasion, we took advantage of the opportunity to query users about planned additions to the prototype. We had developed a skeletal window for the routine-search capability, but had not yet implemented the feature. Users were asked what they thought would happen when the tool searched for the occurrence of a name, and they clearly agreed that this would cause the display to warp to the next node corresponding to that routine. They also confirmed our idea that the search should be circular (i.e., when the last occurrence is found, search cycles back to the first occurrence).

We specifically requested user input on the list sorting algorithms. For the routine search dialog, they confirmed that the routine name list should be ordered alphabetically, with file names visible only when absolutely necessary to disambiguate source routines with shared names. We also questioned the order that would be most appropriate for the popup windows, associated with each node, listing the IDs of each active process and the line number at which it halted. Recognizing the fact that the process IDs themselves were arbitrary in the sense that the programmer cannot control them, we had intended to list the entries in numerical order by line number, so that scrolling through the list would reflect scrolling through source locations. Users suggested a significant modification: entries should appear grouped by line number, in descending order of frequency (i.e., most active locations first). However, they imposed the exception that if the culprit were active in this routine, that item should appear first, regardless of how many other locations were more active. This was an unanticipated strategy, but it reflected an interesting logic similar to that shown by the users in navigating through the displays. They seem to prefer to look at the most active locations first, then proceed to those of lesser activity.

Finally, we asked users under what circumstances they would want to be able to switch to another lightweight corefile during a single session with the browser tool. None of them felt this was a worthwhile feature.

It was particularly gratifying to hear several users state that they were amazed to find that we had managed to keep the tool really simple -- and that although they were not normally tool users, they thought they would find this one useful. As in the early case, we applied the feedback from the trials to

streamline the design, then completed the prototype and began refining it into an alpha release for distribution to other user sites.

Alpha versions of the tool were tested at Supercomputing '94, Intel, Meiko, Livermore National Laboratory, and San Diego Supercomputing Center in October-November of 1994. The alpha trials essentially reaffirmed the design decisions. Users of a variety of experience levels all interacted easily with the tool, with little or no guidance. In fact, they were able to suggest some strategies for extending LCB's usefulness.

One suggestion was that we allow the color "buckets" to vary according to user needs. Our reaction was to over-complicate what users were asking for. We interpreted their request as meaning that they wanted some interactive feature for setting the number of buckets and the bounds for each. After more discussion, however, we found that they did not want nearly that much flexibility, and they became worried we might make the tool too complex by adding "bells and whistles". Since the user knows how many processes were involved in the execution, and since this is likely to be a similar quantity for successive program executions, all they really need is a static way of indicating in advance the bucket bounds. A simple resource setting would suffice to indicate, for example, that instead of assigning buckets to 1-25, 26-50, 51-75, and 76-100 processes, the user would rather see 1-4, 5-10, 11-20, and 21-100.

At this stage, it also became apparent that we had under-estimated some of the possible applications for the call graph representation. John May of LLNL hooked LCB into the TotalView interactive debugger, as an alternative way of showing program state at any arbitrary break in program execution. Although there is no "culprit" in this case, the features for observing relative degree of activity or navigating through the call graph are certainly applicable. Since his initial effort, several of the industrial members of Ptools have expressed interest in using the LCB visualization for situations calling for displays of global program state. This has resulted in a new requirement that is currently being implemented: addition of redrawing capabilities so that the visual representation can be changed (in response to updated state information) without the user having to dismiss the window and re-invoke it.

Conclusions

There is no doubt that LCB, as it exists today, is dramatically different from the initial design conception. Most of the ideas for change were suggested by users, and have been reaffirmed in later trials with other users. In some cases, user input kept us from investing substantial effort in features that would not have been used or appreciated. In others, feedback showed us where our conceptions of usefulness did not quite align with those of the user community.

Of the lessons we learned, the most important is that tool developers should take it very seriously when users ask for simple support. Since developers since a more flexible, more complex system, we tend

to be ready to interpret requests as complicated, when in fact they are for very crude or rudimentary information. Simplicity offers a win/win situation: easier for users to learn and apply, and easier for developers to implement. Tool designers must be careful not to let opportunities for simplicity escape our notice.

Second, it is important to involve users from the very outset of tool design. If the developer can furnish them with realistic estimates of the level of effort required to implement certain types of features, users can be quite adept at prioritizing their needs. Even more important, they can furnish specifics on exactly what and how much information they require to perform the target tasks. This allows the developer to focus on how to make key information available, rather than on discovering what information might be key.

Third, no decision whatsoever should be considered final until users have reaffirmed its importance and appropriateness. If user feedback is contradictory, either the population sampled has been too small or a new solution needs to be devised that incorporates features from the opposing parties. Developing and showing multiple iterations of tools to the users is positive in several ways -- not the least of which is that it reinforces their contribution and is the best way to motivate continued user involvement. What users don't appreciate is telling a developer that something is very hard to use, then seeing that feature remain unchanged.

Last but not least, developers should be careful to question themselves at every step. Is this feature really wanted by the users, or am I getting carried away? So-called "bells and whistles" will turn away testers as well as the end users. Hugh Caffey of BioNumerik Pharmaceuticals expressed it succinctly:

> Have you ever heard a user complain because a tool is too simple?

References

[1] L. J. Arthur, *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons, 1985.

[2] J. M. Carroll and M. B. Rosson, "Human-Computer Interaction Scenarios as a Design Representation," *Proc. HICSS-23*, 1990, pp. 555-561.

[3] J. M. Carroll, J. C. Thomas and A. Mahotra, "Presentation and Representation in Design Problem-Solving," *British Journal of Psychology*, Vol, 71, 1980, pp. 143-153.

[4] C. R. Cook, C. M. Pancake, and R. Walpole, "Are Expectations for Parallelism Too High? A Survey of Potential Parallel Users," *Proc. Supercomputing '94*, 1994, pp. 126-133.

[5] S. De la Chica, "A Graphical Tools for Customizing the Display of Acyclic Graphs," M. S. Thesis, Department of Computer Science and Engineering, Auburn University, 1993.

[6] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data*, MIT Press, 1984.

[7] J. C. French, A. K. Jones and J. I. Pfaltz, eds., "Multidisciplinary interfaces (Panel Report)," *Proceedings Workshop on Scientific Database Management: Panel Reports and Supporting Material*, NSF sponsored workshop held at School of Engineering and Applied Science, University of Virginia, 1990, pp. 2-12.

[8] D. L. Heppe, W. H. Edmondson and R. Spence, "Helping Both the Novice and Advanced User in Menu-Driven Information Retrieval Systems," *Proceedings Conference of the British Computer Society*, Sept. 1985, pp. 95-100.

[9] B. Hesse, L. Sproull, S. Kiesler, and J. Walsh, "Returns to Science: Computer Networks in Oceanography," *Communications of the ACM*, Vol. 36, No. 8, 1993, pp. 90-101.

[10] K. Holtzblatt and H. Beyer, "Making Customer Centered Design Work for Teams," *Communications of the ACM*, Vol. 36, No. 10, Oct. 1993, pp. 93-103.

[11] W. Howell, "How Social Scientists Can Contribute to the Information Revolution, *Chronicle of Higher Education*, June 8, p. A40.

[12] R. Jeffries, J. R. Miller, C. Wharton and K. M. Uyeda, "User Interface Evaluation in the Real World: A Comparison of Four Techniques," *Proc. CHI'91*, 1991, pp. 119-124.

[13] C-M. Karat, R. Campbell and T. Fiegel, "Comparison of Empirical Testing and Walkthrough Methods in User Interface Evaluation," *Proc. CHI'92*, 1992, pp. 397-404.

[14] J. Kuehn, "NCAR User Perspective," *Proc. 1992 Supercomputing Debugger Workshop*, Jan 1993.

[15] M. Kyng, "Designing for Cooperation: Cooperating in Design," *Communications of the ACM*, Vol. 34, No. 12, December 1991, pp. 64-73.

[16] C. Lewis and D. A. Norman, "Designing for Error," in *Readings in Human-Computer Interaction*, ed. R. M. Baecker and W. A. S. Buxton, Morgan Kaufmann, 1983, pp. 627-638.

[17] A. Marcus, "Human Communications Issues in Advanced UIs," *Communications of the ACM*, Vol. 36, No. 4, April 1993, pp. 101-109.

[18] J. R. McGraw and T. S. Axelrod, "Exploiting Multiprocessors: Issues and Options," in *Programming Parallel Processors*, R. G. Babb, ed., Addison-Wesley, 1988, pp. 7-26.

[19] M. Muddarangegowda and C. M. Pancake, *Lightweight Corefile Browser*," Technical Report 94-80-17, Department of Computer Science, Oregon State University, 1994.

[20] National Research Council, *Information Technology and the Conduct of Research: The User's View*, National Academy Press, 1992.

[21] J. Neilsen, "Non-Command User Interfaces," *Communications of the ACM*, Apr. 1994, pp. 83-98.

[22] J. Neilsen, "Iterative User-Interface Design," *Computer*, Nov. 1993, pp. 32-40.

[23] J. Neilsen, "The Usability Engineering Life Cycle," *Computer*, March 1992, pp. 12-22.

[24] D. A. Norman, "Cognitive Engineering," in *User Centered System Design*, D. A. Norman and S. W. Draper, eds., Erlbaum Associates, 1986, pp. 31-62.

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

[26] C. M. Pancake, "A Collaborative Effort in Parallel Tool Design," in *Environments and Tools for Parallel Scientific Computing*, J. Dongarra and B. Tourancheau, eds., SIAM, 1994, pp. 112-118.

[27] C. M. Pancake, "Customizable Portrayals of Dynamic Program Structure," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993, pp. 64-74.

[28] C. M. Pancake, "Why is There Such a Mis-Match between User Requirements and Parallel Tools?" keynote address, ARPA/NSF Workshop on Instrumentation of Parallel Computing Systems, Keystone, CO, Mar. 1993.

[29] C. M. Pancake, "Software Support for Parallel Computing: Where Are We Headed?" *Communications of the ACM*, Vol. 34, No. 11, Nov. 1991, pp. 52-64.

[30] C. M. Pancake and D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Researchers?" *IEEE Computer*, Vol. 23, No. 12, Dec. 1990, pp. 13-23.

[31] C. M. Pancake, *et al.*: results of user surveys conducted on behalf of Intel Supercomputer Systems Division, IBM Corporation, and CONVEX Computer Corporation, 1989-1993.

[32] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd ed., Addison-Wesley, 1992.