Models for Visualization in Parallel Debuggers

Technical Report CSE-89-01

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

Sue Utter
Cornell National Supercomputer Facility
Cornell University
email: psu@cornellf.tn.cornell.edu

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

July, 1989

Models for Visualization in Parallel Debuggers

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

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

Abstract

The complexity of parallel programming has stimulated the development of a variety of debugging tools. This survey of recent research focuses on debugger visualization systems. The effectiveness of such systems is bounded by the degree to which their representations of run-time behavior correlate with the language structures used to incorporate parallelism, as well as the logical framework adopted by the programmer. Current visualization systems are compared with the conceptual models supported by parallel languages. Attention is drawn to the fact that debuggers are tied to specific models and that this association may restrict their usefulness and acceptability.

Keywords: parallel and distributed debuggers, program visualization, language support for parallelism

Introduction

The advent of parallelism has introduced a new range of problems that are well beyond the experience of traditional serial programmers. Established paradigms for program development are known to be inadequate, but no alternatives have as yet achieved widespread acceptability. The lack of consensus is particularly appreciable in the area of parallel debugging, where published treatments — both theoretical and methodological — reflect the individual experiences of tool developers. The result is a proliferation of unrelated and sometimes conflicting information (compare, for example, the viewpoints expressed in [1]–[7]). Descriptions of parallel debuggers are often so sketchy that few real conclusions can be drawn. Furthermore, many systems are still in the experimental stage, so no evidence has been presented regarding their suitability for real-world problems or a general audience.

A complete discussion of parallel debugging tools is clearly beyond the scope of this paper. Instead, we focus on an aspect that is critical to the user, but often downplayed in papers and reports: the way parallel execution is represented graphically by a debugger. The representational and interpretive concepts which form the basis for graphical displays, together with the screen configurations employed, are referred to collectively as the debugger's *visualization system.* Attention will be drawn to the fact that existing visualization systems are tied to specific conceptual models for parallelism, an association which may be undesirable in terms of both usefulness and acceptability.

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

Existing programming languages impose restrictions on the program development process by supporting only a limited number of conceptual models for parallelism [8]. The programmer must, in effect, reconfigure the mental model to fit the framework of the language at hand. This situation has impact on debugging technology. Graphical portrayals of execution must be consistent with that framework if the programmer is to make accurate inferences. When the representational scheme employed by a debugger requires that the user make yet another conceptual

---

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.

The authors would like to thank the developers who furnished original copies of their debugger output: Robert J. Fowler *et al.* of the University of Rochester; David Socha *et al.* of the University of Washington; Alfred A. Hough and Janice E. Cuny of the University of Massachusetts, Amherst; and M. Zimmermann *et al.* of the Ecole Polytechnique Federale de Lausanne.

transformation, the relationship between program code and run-time behavior is obscured rather than elucidated.

The effectiveness of a debugger's visualization system is bounded not only by the representational scheme it employs, but also by the degree to which program code actually reflects the programmer's logic. Although a human can and does apply knowledge of the algorithm in reasoning about program behavior, an automated tool must rely solely on details gleaned from analysis of the program text and its run-time behavior. The information provided by a debugger is effectively constrained to the implementation rather than the underlying problem. It is critical, therefore, that the "semantic gap" between concept and code be minimized.

In serial programming paradigms, this can be accomplished in two ways. First, formal software design methodologies structure the conceptualization process so that conversion to program code is more straightforward. Alternatively, the introduction of programming language constructs at a higher level of abstraction shifts transformation responsibilities from the programmer to the compiler. Unfortunately, neither alternative is available to the parallel programmer. Parallel design paradigms are still in an embryonic stage, while programming language support for parallelism is at a distressingly low level, reminiscent of the early days of Fortran [9, 10]. What's more, recent studies indicate that abstraction mechanisms themselves may compromise the correct functioning of parallel programs [11]. Given that the distance between problem logic and parallel code is unlikely to improve in the near future, it is particularly important that debuggers employ appropriate schemes for visualization.

Types of Visualization Systems

The developers of parallel debugging tools have elected to represent program execution in distinctive ways. Commercially available debuggers employ a *state-based* approach, probably because this represents a natural extension of serial debugging technology [3]. Serial execution can be traced by observing changes in "program state", as characterized by the contents of data storage and the flow of program control. This approach is extended to parallel execution by viewing a program as a collection of functional entities executing concurrently or intermittently. Where a serial debugger allows the animation of instructions or the examination of data within a single program, a parallel tool provides similar capabilities for a series of processes. This is usually accomplished by supporting multiple debugger windows, one per process [12, 13, 14, 15].

Even with serial debugging, it is difficult for the user to correlate changes in program state with his/her concept of what correct program behavior should be. This difficulty is compounded when the user must assimilate state changes occurring in each of a number of processes and synthesize some coherent notion of overall behavior. A few experimental systems have attempted to facilitate this conceptualization by abstracting the process unit. Rather than tracing state changes

locally within processes, these systems portray the global flow of execution among them. The process becomes the basic unit of observation; state changes include creation, deletion, execution, delay, and completion. Program structure is depicted by a hierarchical framework such as a call graph, where vertices represent processes and arcs denote execution or calling sequence. Execution behavior can be animated by altering the color or pattern of the vertices to reflect state changes through time. This type of visualization system is included in the Schedule environment developed by Dongarra and Sorensen [16] for writing portable parallel programs (see Figure 1). Here, the nodes represent subroutine-level processes and can have one of four states: clear nodes are subroutines awaiting the completion of subordinate nodes, lined nodes are ready to run but haven't yet been started by the system, hashed nodes are currently executing, and black nodes have reached completion. The visualization system of Zimmermann *et al.* [17, 18] provides a similar view of execution flow; in this case, however, other program components such as synchronization objects are included as part of the display. In Figure 2, for example, circular nodes represent processes, while various box shapes are used to signify monitors, modules, routines, and signals.

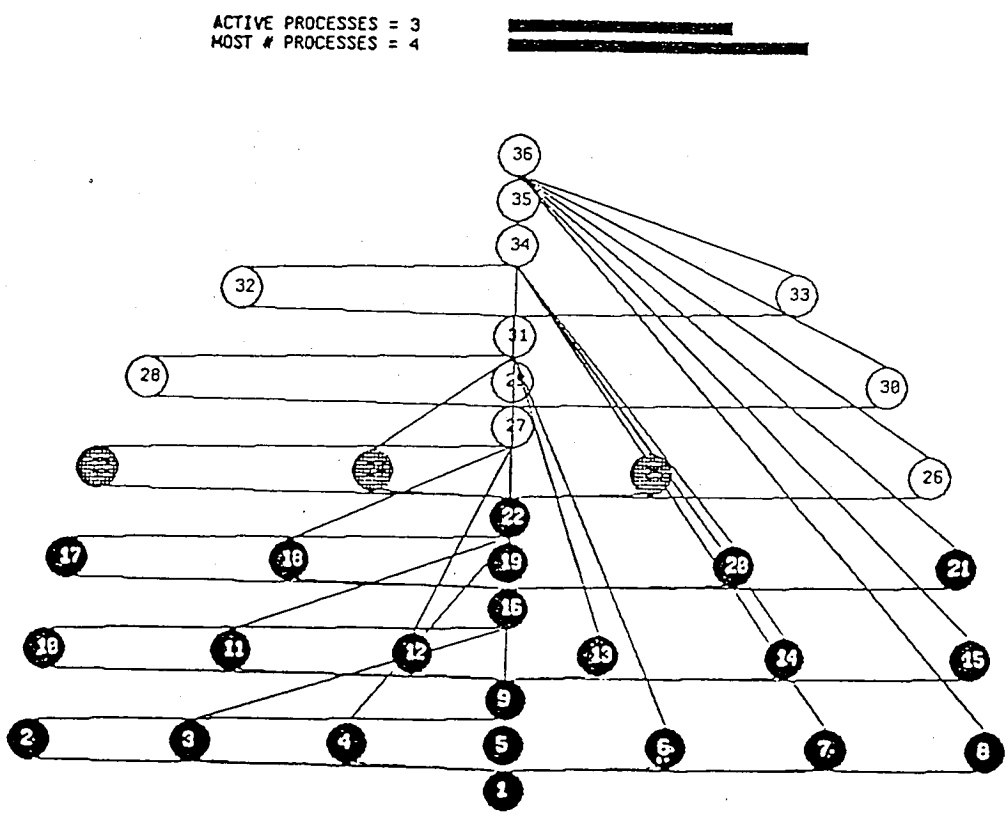

Figure 1. Subroutine-level process representation, Schedule [16]

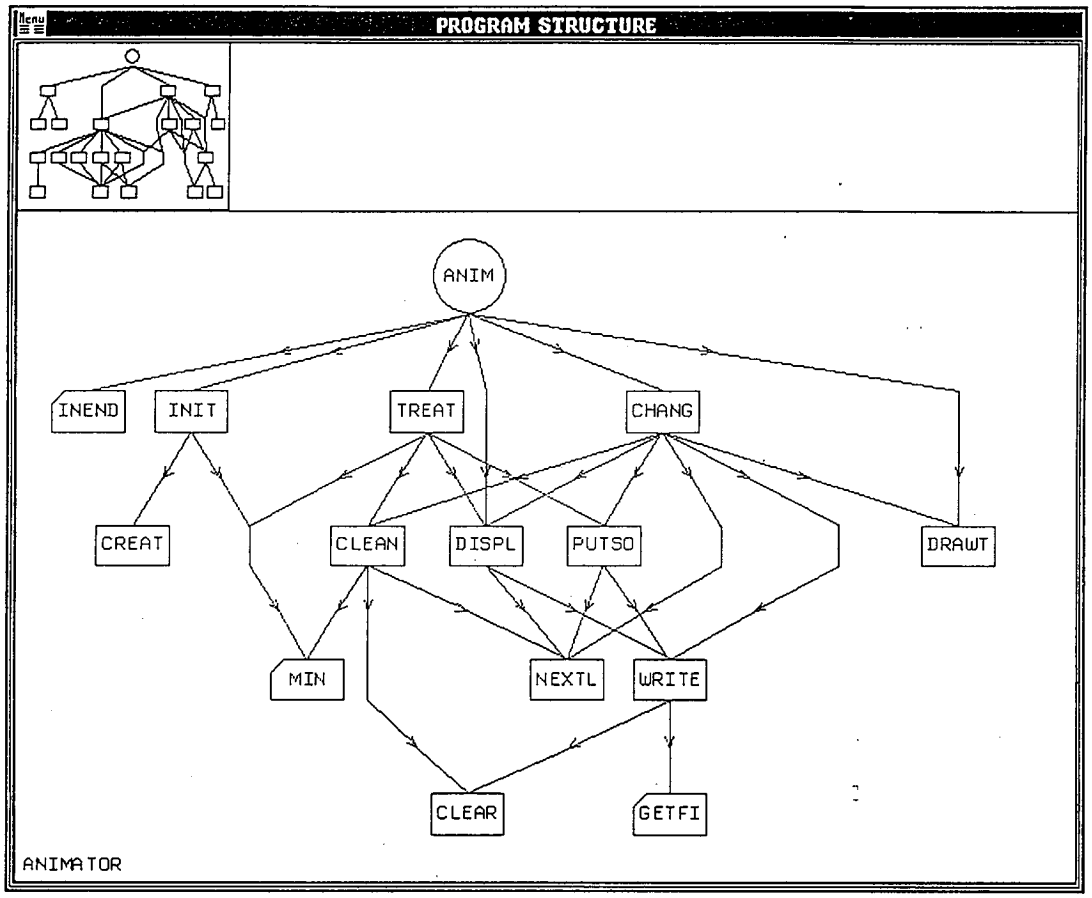

Figure 2. Program structure representation, Zimmermann et al. [17]

The problem with state-based visualization systems is that snapshots of individual and collective processor states do not adequately describe parallel behavior. If the notion of global state is to be useful, the occurrence of process interactions at the instant of the snapshot must also be taken into account. This introduces a new level of complexity, reflecting the fact that in parallel programs the sequencing of interactions can provoke additional errors. Thus, understanding the temporal relationships among interactions, both potential and observed, is fundamental to developing correct parallel programs. A series of experimental visualization systems have evolved from this realization. They may be characterized as *interaction-based*, in reference to their emphasis on this aspect of program behavior.

Processes interact in two ways: through communication, or the exchange of data information; and through synchronization, the exchange of control information. To date, debugger developers have focused on displaying communication interactions [19]. Time-process diagrams are employed to provide a static after-image of communications over time. In most cases, a time line is drawn along one dimen-

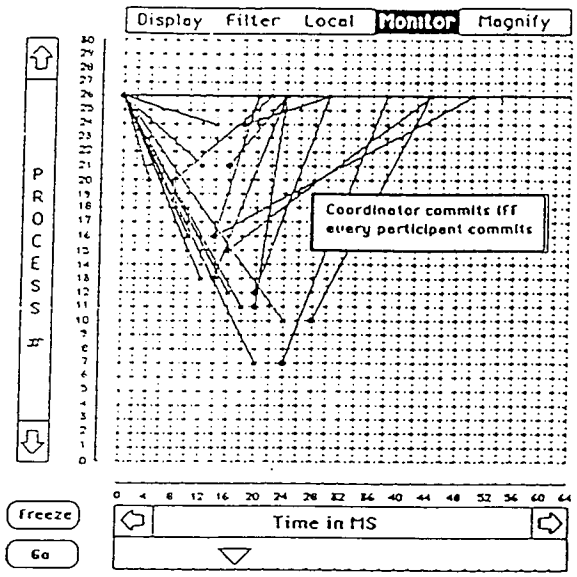

Figure 3. Process-time diagram, IDD [20]

sion, while individual processes are distributed across a second. Moving back and forth along the time line reveals the sequence of communications recorded during execution. Message traffic is depicted by lines connecting processes in pairs. One example is the display proposed by Harter *et al.* [20] for the IDD distributed debugger, shown in Figure 3. Here, processes are distributed along the vertical axis, while time increases from left to right along the horizontal. The Moviola system being developed at the University of Rochester by LeBlanc, Fowler, and Mellor-Crummey [21] uses essentially the same arrangement, with the axes reversed (Figure 4).

Time-process information can also be animated through a series of displays, each depicting an instant in program execution. The Radar debugger, developed by LeBlanc and Robbins [22] for a message-passing system, uses animation to display the direction of process communications as well as their sequencing (Figure 5). Processes are represented as boxes with input and output ports; the number of queued messages is shown with each box. Hough and Cuny [23, 24] capitalized on the observation that some parallel programs exhibit recurring or uniform patterns of interprocess communications. Their Belvedere debugger allows the user to specify the spatial arrangement of processes in order to model the communication topology most likely to reflect such logical patterns. Figure 6 shows a sample frame from an interaction animation, arranged to represent a hypercube topology. Highlighted arrows represent SENDs, highlighted ports indicate messages received via GETs, and multiple arrowheads portray message queueing at a port.

The second type of interaction — synchronization — permits one process to influence the activity of another by the exchange of control information in the form of semaphores, events, barriers, rendezvous, and so forth. These mechanisms impose constraints on the global ordering of events and can be used to coordinate

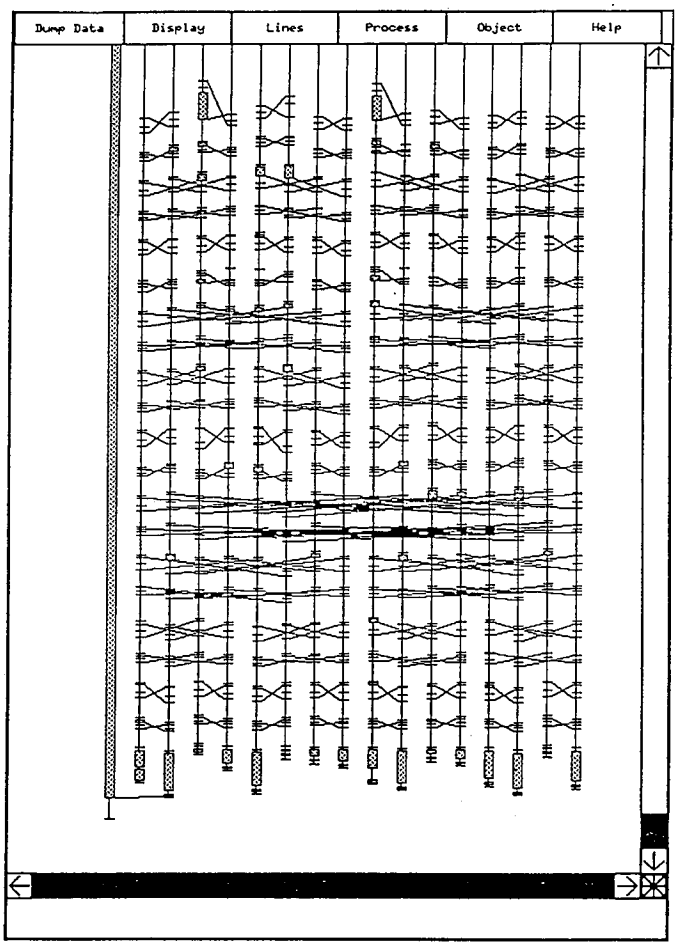

Figure 4. Process-time diagram, Moviola [21]

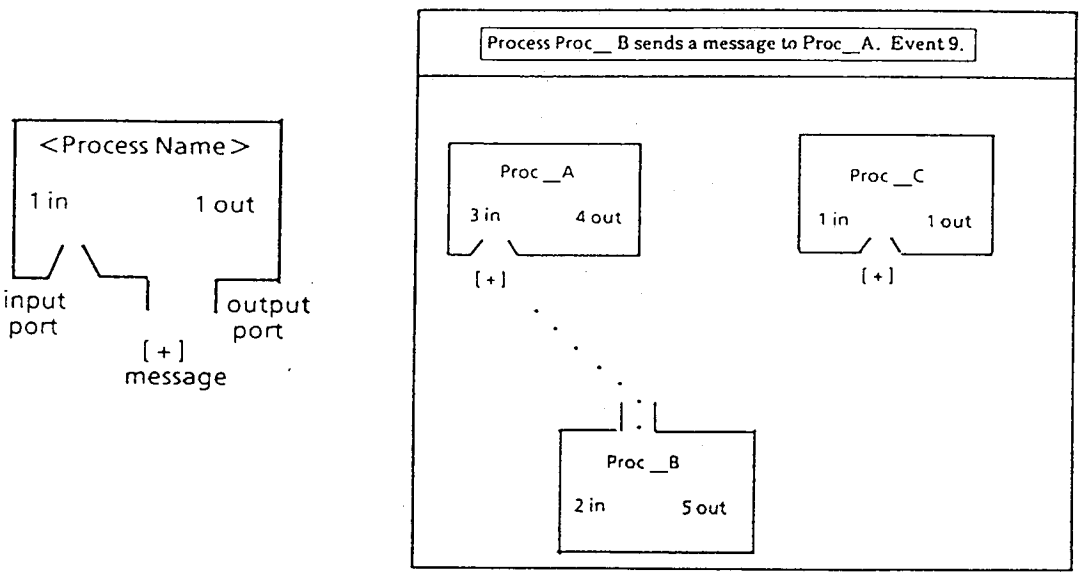

Figure 5. Animation frame, Radar [22]

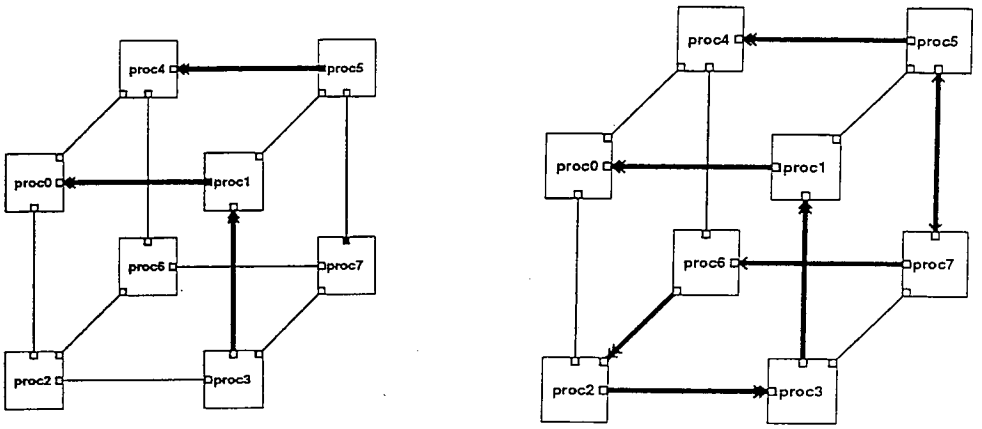

Figure 6. Animation frames, Belvedere [23]

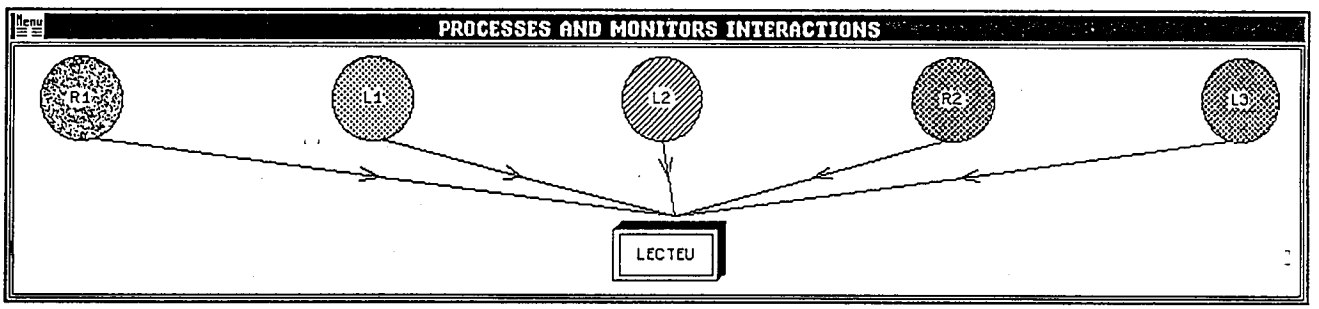

Figure 7. Synchronization via a monitor, Zimmermann et al. [17]

access to shared data or to control the sequencing of process activities. Although synchronization plays a critical role in many parallel programs, contributing a new source of errors as well as degrading performance, only one documented visualization system provides direct support for observing synchronization patterns. The tool, developed by Zimmermann *et al.* [17, 18], is based on a programming language that makes use of monitors and signals to implement synchronization. As shown in Figure 7, processes are represented by circles, while monitors occur as boxes; again, patterning serves to indicate the current state of each process.

Support for Conceptual Models

Just as debugger developers choose to represent program execution in a variety of ways, programming language designers select among alternative approaches in supporting parallelism. Where visualization systems reflect the primary focus in portraying the dynamic behavior of parallel processes — as a sequence of state changes or of process interactions — language support corresponds to the way in which program elements are decomposed for execution in parallel.

Programming languages support parallelism in three major ways: (1) by incorporating parallel constructs as integral parts of the language definition; (2) by adding parallel extensions to an existing serial language; or (3) by providing high-level interfaces to parallel routines stored in runtime libraries. Our treatment ignores the issue of exactly how parallel features are incorporated, focusing instead on the way parallelism is modeled.*

The specific parallel features provided by a programming language reflect the language designer's concept of parallelism and how this might best be integrated into high-level programs. A survey of currently available languages reveals that in most cases, direct support is limited to a single conceptual model, with subsidiary features added to achieve some degree of flexibility [8]. Furthermore, although syntax and naming conventions differ significantly from one language to another, the range of conceptual models supported is quite limited. The models can be divided into two general categories according to the structural focus employed to incorporate parallelism: task-oriented or data-oriented. Analogies will be drawn between the models provided by programming languages and those currently supported in debugger visualization systems. (A third focus, object-oriented, is reflected in some experimental languages; they are considered in a section on alternative models.)

Task-Oriented Models

The earliest models for parallelism were task-oriented, reflecting the viewpoint of operating systems designers who saw parallelism as the concurrent execution of functionally distinct elements. In this category, parallelization is achieved by partitioning the tasks to be performed into a collection of serial processes, each possessing a unique thread of control. Interaction among these entities is implemented when necessary by heterogeneous mechanisms for communication and synchronization, but their role is ancillary to that of processing.

The simplest task-oriented model is one of *independent processes* (Figure 8). It implements a restricted form of parallelism by establishing a set of processes that execute independently and then self-terminate. Child processes are spawned implicitly when an appropriate bracketing element is encountered in the program syntax; the closing bracket implicitly defines a synchronizing barrier at which the parent process waits. Only a few languages (e.g., Algol 68, Smalltalk) support this early model, which was superceded by the concept of cooperating processes.

Cooperating processes form the basis of the most widely supported model (Figure 9). In this scheme, a pool of autonomous processes execute indefinitely, cooperating to share a set of common resources. Interprocess cooperation typically

* In the cases described, parallel features are explicit; that is, the programmer must specify when concurrent execution is appropriate. Parallelization is said to be implicit when the compiler is capable of recognizing potentially concurrent portions of a serial program and generating parallel code. Vectorization, which generates parallel code for predefined array operations only, represents a similar approach. These alternatives are not considered, since they remove parallelism from the domain of the programmer.

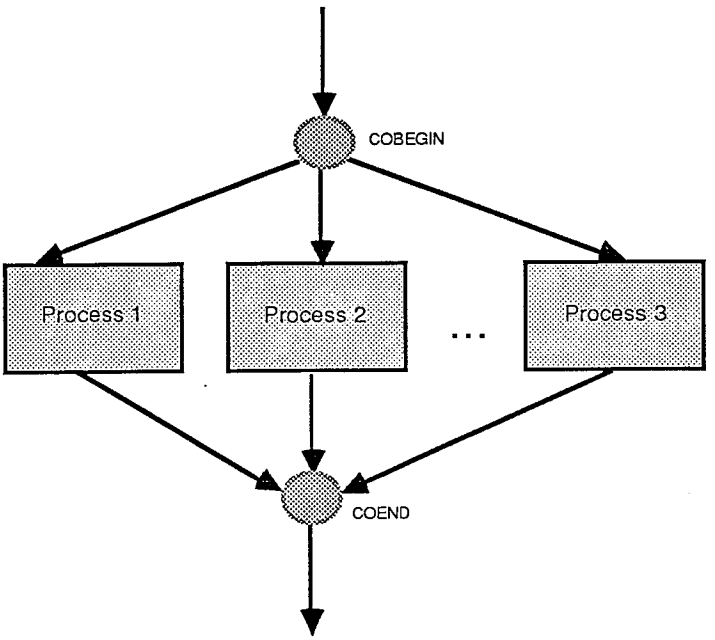

Figure 8. Independent processes model

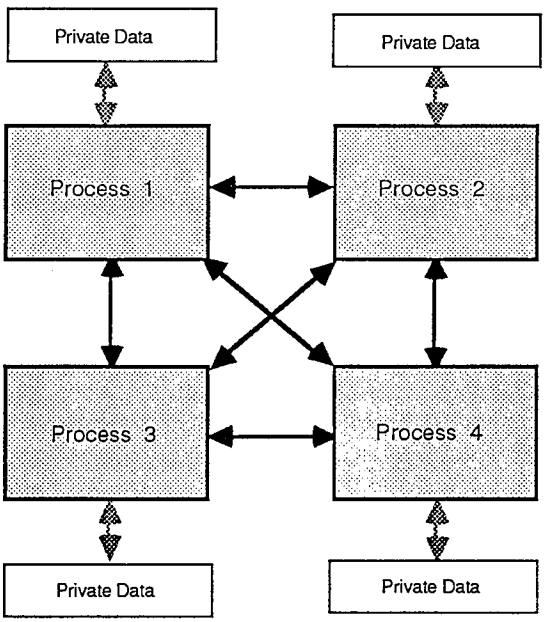

Figure 9. Cooperating processes model

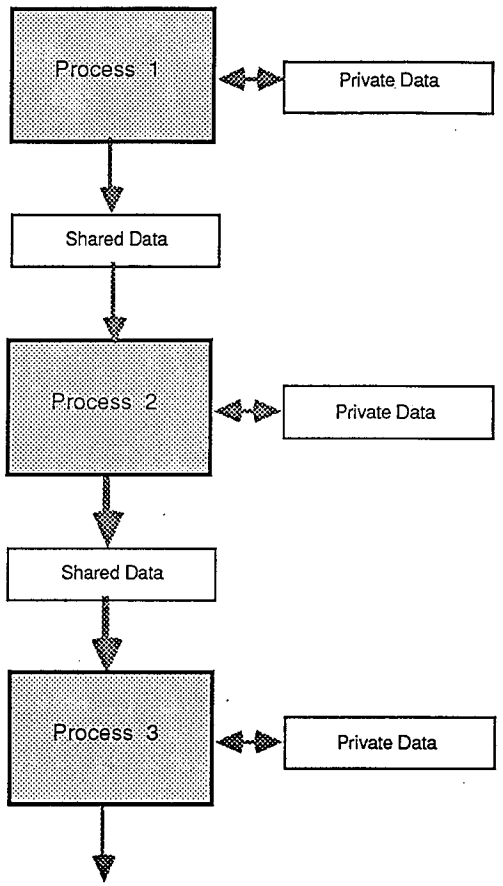

Figure 10. Pipeline model

requires two-way communication between pairs of processes, implemented directly by message passing or indirectly through the sharing of mailboxes or other storage areas. Processes are defined explicitly and created either statically or dynamically; each instance is given a unique identifier to facilitate interaction. One process cannot be said to strictly dominate another and there is no real mechanism for global control of activities. This model serves as the basis for parallelism in several programming languages, including Occam, Ada, and PL/I.

In an alternate model, processes are arranged to form a *pipeline*, or sequence of producer-consumer associations (Figure 10). The output of one process serves as input to the next, so interaction is limited to immediately adjacent units. A certain degree of synchronization is achieved by assuring that consumers do not attempt to process non-existent input and that producers do not over-write output which has not been consumed. This is best supported by one-way communication (e.g., ports, channels, signals), but shared flags provide an alternate, indirect form of synchronization. Therefore, the pipeline model can be accommodated even by programming languages with only minimal concurrency mechanisms.

More general demands for data sharing led to the development of the *moni-*

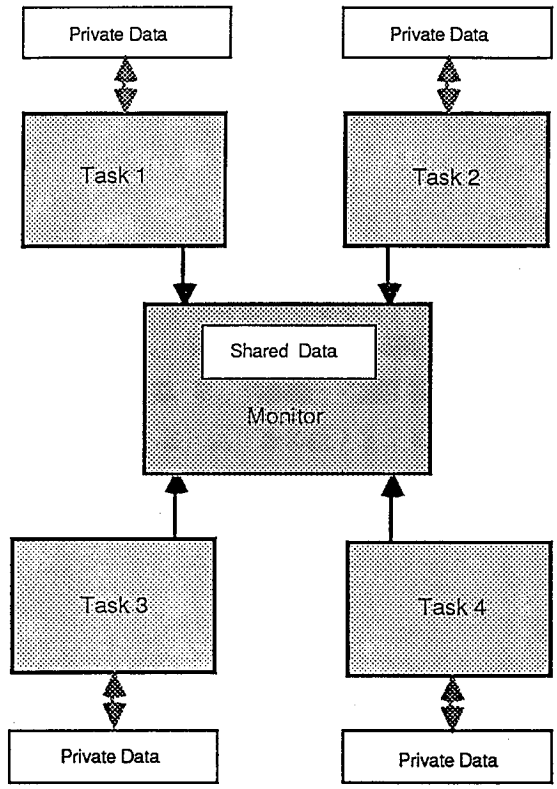

Figure 11. Monitor/task model

tor/task model (Figure 11). Here, one process is distinguished as the monitor, or guardian; its role is to arbitrate the contention that occurs when other processes simultaneously attempt to access a shared resource. The monitor ensures mutual exclusion by granting access to just one contender at a time. Unlike task processes, the monitor is a passive entity, activated only when it is invoked by one or more tasks. A number of languages were designed to support this model directly (e.g., Concurrent Pascal, Mesa, Modula and Modula2, Concurrent Euclid).

Debugger Support for Task-Oriented Models

For task-oriented parallelism, the motivating factor in devising a programming strategy is function: the programmer decomposes the problem in terms of sequences of activities that can be carried out as integral units, referred to here as tasks. A single predetermined task is assigned to each process (of course, two or more processes may perform identical activities). Although the order of execution within a process is deterministic, the global execution of multiple processes is nondeterministic. The principal concern in debugging this type of program is to insure that the function assigned to each process, and summarily to the program as a whole, is performed correctly. State-based techniques are therefore the foundation of debugger support for task-oriented models.

If task-oriented parallelism had not progressed beyond the independent pro-

cesses model, available commercial debuggers would provide adequate support, since the ability to examine individual threads of control within the pool of concurrently executing processes would suffice. The other conceptual models, however, allow the activity of one process to influence the others. Processes are seen as functional entities that may be created dynamically, undergo dormant periods, and be terminated independently. The resulting nondeterminism of global program activities can lead to so-called "schedule-dependent" bugs [25], which occur only when particular global execution paths are taken. To be effective, debugger support must be extended to provide the capacity for examining overall flow of execution control (as in the experimental Schedule system).

Extension of the independent processes model to include cooperation among processes necessitates further additions to debugger support. This has been reflected in the development of interaction-based systems. In languages where cooperation is through message-passing, the ability to observe communication patterns is desirable, while for interaction via shared storage the patterns of synchronization controlling the order of access are more important. Such requirements do not replace the more general need to analyze changes within individual processes — they add to it. Unfortunately, with the exception of the Zimmermann *et al.* system, no documented debuggers provide both state-based and interaction-based features.

The pipeline and monitor/task models represent particular approaches to cooperation among processes and, as such, could benefit from more specialized debugger features. The use of Belvedere-like techniques for arranging processes in a display to reflect the pipeline topology, for example, would significantly enhance the presentation of communication patterns for the pipeline model. In this sense, the visualization system proposed by Zimmermann *et al.* is a prime example of the advantages to be gained by tailoring a debugger to a particular conceptual model. The capability it provides for inspecting the synchronization of accesses to monitor resources is extremely useful; this specialization, however, is achieved at the cost of limited applicability.

Data-Oriented Models

Data-oriented models provide a contrasting approach, in which parallelism occurs as the simultaneous performance of an operation on multiple data elements. Whenever the size of the data warrants parallelization, activities are replicated for application across data subsets. In effect, program execution is viewed as a single thread of control which temporarily diverges into parallel action sequences that later converge. The methods used to distribute data and synchronize convergence become crucial factors.

The simplest and most widely used data-oriented model is the *concurrent loop* (Figure 12). Like the classical indexed loop structure of serial computing, the concurrent loop consists of a sequential body enclosed by bracketing elements. The sequence of operations specified within the loop body is carried out on a series of logical processors corresponding in number to the loop iterations. Many

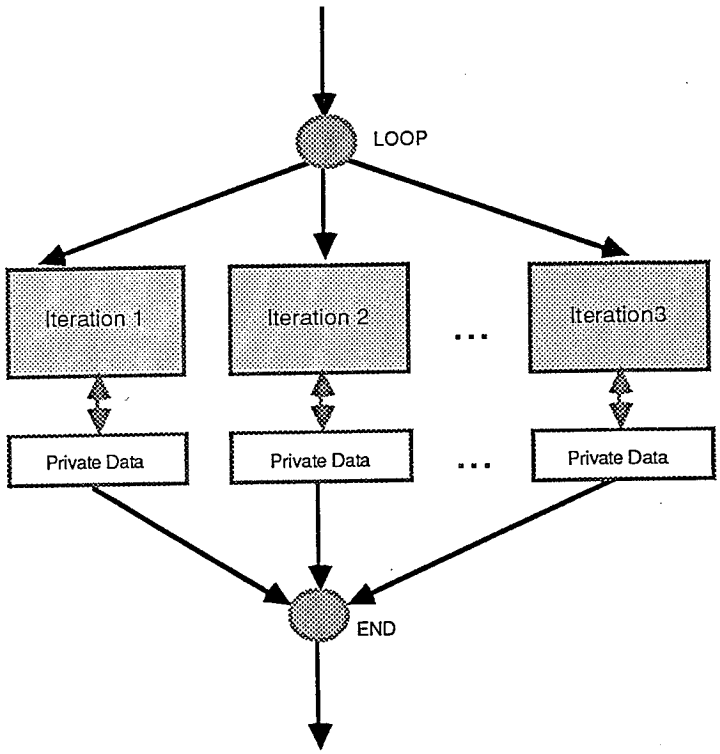

Figure 12. Concurrent looop model

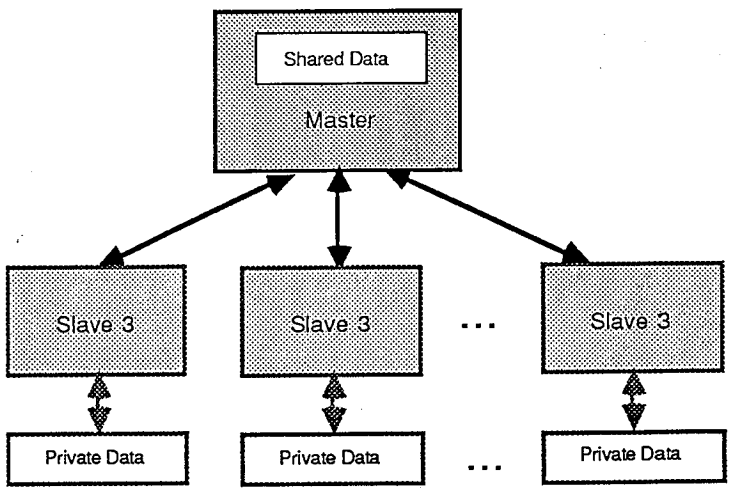

Figure 13. Master/slave model

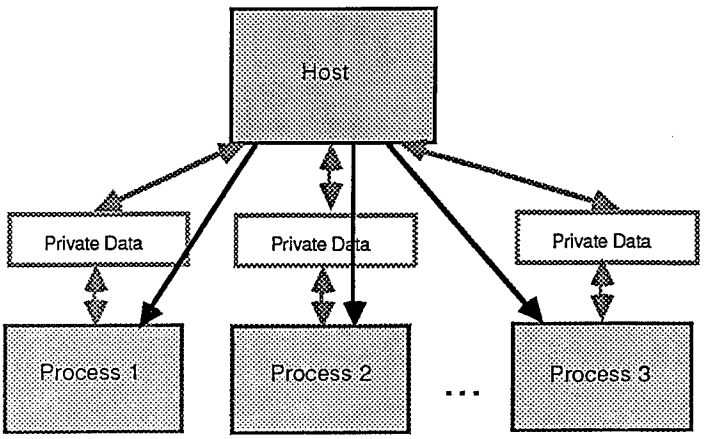

Figure 14. Domain decomposition model

serial programming languages which have been extended for vector or parallel architectures, including Parallel Pascal and most parallel Fortrans, support the concurrent loop model. Some versions allow the occurrence of data dependencies within the loop body, automatically compensating by scheduling delays in the activation of certain processors (this technique is often called "pipelining," in reference to the model already described).

The *master/slave* model is a more generalized version of the concurrent loop (Figure 13). This model is anthropomorphic: one logical processor, identified as the "master," is clearly superior to the others, his "slaves." In effect the master owns all logical processors and decides when and how to use each. When parallelization is appropriate, the master activates the desired number of slaves, then waits for them to complete their work before continuing. Typically, the master is responsible for balancing the load among available processors as well as managing all communications. A variant of the master/slave model uses a queue manager to keep track of activities to be performed. As each logical processor finishes a task, it reports the results to the master and requests a new task; since tasks are distributed round-robin style as processors become available, load balancing is implicit. Barriers are often used to ensure that all slaves have completed before the master resumes activity. Several parallel Fortrans (e.g., BBN Butterfly, Encore Multimax, the Parallel Computing Forum's proposed standard) include syntactic structures or multitasking library routines appropriate to this model.

While the master/slave and concurrent loop models reflect the data-oriented parallelism inherent in vectors and matrices, the so-called *domain decomposition* model corresponds to other topological distributions of data such as multidimensional arrays or trees (Figure 14). A divide-and-conquer approach is taken, with each processor again executing the same operations on its own data. In this case, however, the data cannot be assumed to occupy consecutive locations in storage and must therefore be distributed across the network of processors prior to begin-

ning the operations. A similar step is needed at the end of processing to accumulate the results; alternatively, all processors are halted as soon as any solution is found. One process, often distinguished as the "host" (corresponding to the concept of the root of a tree), is responsible for dividing the data, propagating it to the appropriate logical processors, receiving the reduced solution, and coordinating communication. Data distribution is typically achieved via explicit broadcast and reduction constructs. The clearest support for domain decomposition is found in parallel versions of functional and logic languages such as Parlog, Concurrent Prolog, *Lisp, and MultiLisp, but some procedural languages provide related features (e.g., Parallel Pascal, iPSC parallel C library).

Debugger Support for Data-Oriented Models

With data-oriented models, developing a programming strategy is by domain rather than functional decomposition: data, not tasks, are partitioned and distributed across logical processors. The parallel threads are ephemeral, varying in number according to the requirements of data subsets. The mechanisms for data distribution and access become more important than the definition of activities *per se*. Interaction-based debuggers thus form the basis for current debugging support of data-oriented models.

Because some data-oriented models are analogous to task-oriented versions, certain state-based techniques can provide a minimal amount of supplemental aid. The state-based systems which simply extend serial debugging by allowing multiple windows for observing state changes are of little help, since they are effective only for noting changes in data value and do not provide clues as to which process effected the change. State-based debuggers which illustrate global flow of execution are also of limited use, due to the altered role of nondeterminism in data-oriented models. Here, nondeterminism occurs when blocks of data are processed at the local level, while major program operations proceed deterministically.

The situation is different for interaction-based debugging systems. Most of these were developed for distributed memory machines, where data must be explicitly parceled out among processors. Interaction-based debuggers therefore provide at least a crude approximation of interprocess communication patterns, which in many applications are indicative of whether or not the program is proceeding correctly. Of course, the simple fact that communication patterns are as expected does not insure program correctness; it is critical that the proper values be transmitted at each point. For this reason, most interaction-based systems also allow the user to examine message content.

Providing representations of communication patterns for programs on shared memory machines poses a distinct set of problems. Each WRITE operation to shared memory constitutes a potential communication to all other processes. Since the overhead for interprocess communication is significantly less for shared versus distributed memory machines, this type of interaction is likely to occur with some frequency, with the result that time-process displays quickly become unintelligible.

Data distribution on shared memory machines can also be implemented via process interactions that employ synchronization to constrain access to shared data structures. The observation of this type of synchronization would be of considerable value during debugging. It is therefore unfortunate that only one of the documented systems (that of Zimmermann *et al.*) supports the direct viewing of synchronization information.

Alternative Models

No discussion of support for conceptual models would be complete without consideration of recent developments in object-oriented techniques for parallel programmers. The languages and debugging tools which support this approach are still experimental, having evolved from the object-oriented design methodologies of software engineering. Object-oriented paradigms blur the distinction between data and manipulation. Program entities, or objects, correspond to the components of a modularly decomposed system. Unlike the results of functional or domain decomposition, entities do not uniformly represent subsets of activities or data. What is important is the establishing of an orderly system of interaction among independent, self-contained entities; the question of whether an object contains data, operations, or both is immaterial. In effect, the program becomes a system of black boxes, each emitting and receiving messages. A number of parallel programming languages supporting this approach have been proposed, including ABCL, Concurrent Smalltalk, Act3, and the Linda languages.

A related approach has been adopted in some recent tools for algorithm animation. The Voyeur system, developed by Socha *et al.* [26], provides several alternatives for viewing parallel program behavior in abstracted form. In Figure 15, for example, a load-balancing algorithm is animated as a display of fishes and sharks across a grid. The Aladdin system [27] offers similar capabilities, but the parallel version is not yet complete. Abstraction techniques such as these can significantly improve the effectiveness of visualization systems, since the user no longer needs to integrate isolated components of program behavior in order to infer the overall effect.

There are two major drawbacks associated with object-oriented models. First, their use requires a radical departure from traditional methods of incremental program development. Object-oriented design methods were developed in response to the need for building large software systems in an orderly and reliable fashion. It is uncertain whether they will be able to achieve general acceptability. Second, it is not yet clear how or if debugging tools might extract sufficiently abstract information from program code to provide meaningful displays. Even the simplest documented examples required extensive input from the user prior to execution.

Conclusions

It is clearly possible to implement a variety of conceptual models using any parallel programming language. A clever programmer can build a monitor system

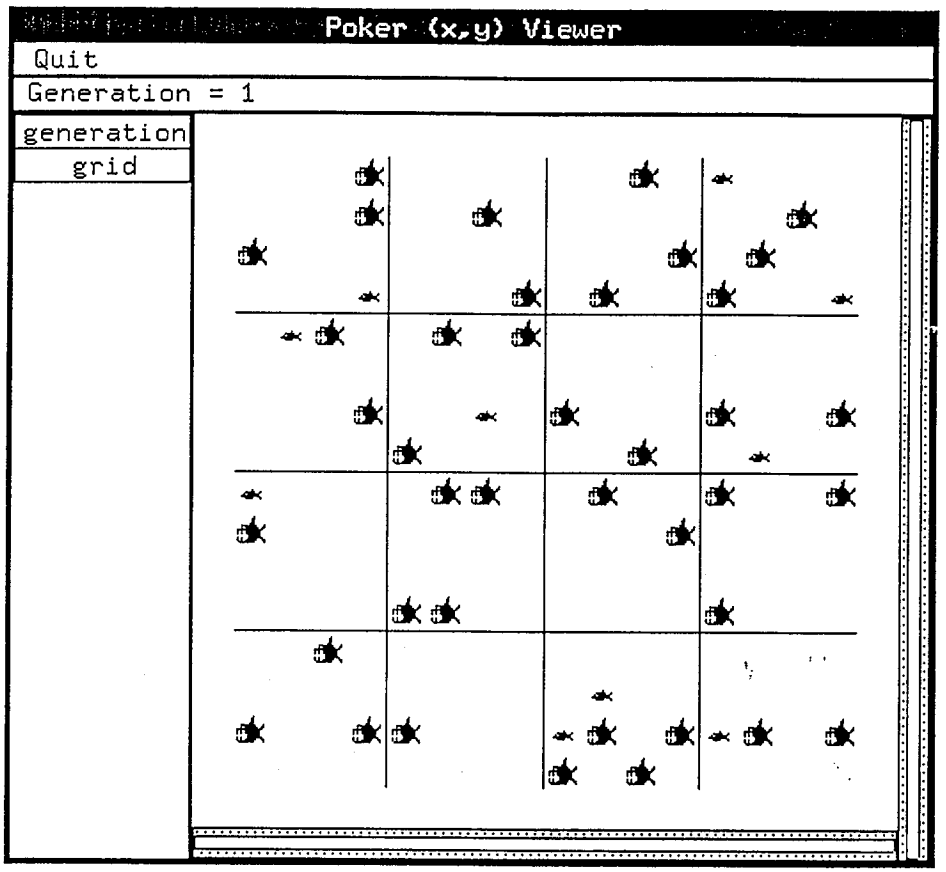

Figure 15. Algorithm animation frame, Voyeur [26]

in parallel Fortran or a concurrent loop model in Ada, even though the languages provide no direct facilities for their implementation. Program development becomes significantly easier and more reliable, however, when the language does support the desired model. A close correlation between concept and code allows the programmer to take full advantage of the compiler's error-checking capabilities. At the same time, comprehension improves since program structure more clearly reflects problem logic.

There is an analogous relationship between the models supported by a debugger and the language in which the program is written. Again, it is possible to employ an unrelated visualization system successfully to analyze and modify program behavior. Nevertheless, this approach is tedious and unreliable, since it needlessly shifts reponsibilities from the tool to the user. A programmer who developes a data-oriented application, for example, is concerned with the logical consequences of matrix manipulations or tree traversals. By restricting program observation to the sequence of write operations to shared memory, the debugger forces him/her to assimilate a great deal of extraneous data in order to extrapolate those aspects of behavior which reveal the underlying concepts.

Debugging activities would clearly be facilitated if the representational framework could be correlated more closely to programmer logic. The so-called "event-based" monitors take this approach, allowing the user to construct a higher-level view of system behavior by filtering program actions or clustering them into abstract events [2, 21, 28, 29, 30, 31]. Unfortunately, the data is generated in the form of a textual trace rather than a graphical display, so the user must still perform a series of conceptual transformations to extract the relevant information.

Another problem facing the user is that display decisions are rarely automated. Current parallel debugger technology relies on user-supplied information to determine the number, type, and placement of visual elements on the screen. The production of lucid graphical representations requires a great deal of effort and typically involves the use of specialized command languages. In most cases, the usefulness of the program visualization is highly dependent on how well the user understands the relationship between debugger features and program structure. Furthermore, the need to encode this sort of "debugger interface" is yet another program transformation, introducing new potential for error [19]. Debugger acceptability could be improved, particularly among novices, by the provision of such features as default debugging modes, interactive support of display reconfiguration, and libraries of examples demonstrating the effective instrumentation of common language-supported models.

Finally, experience supports the notion that program development benefits most when different conceptual models are applied to different types of problems [32, 33]. The results of human factors research also encourage the notion that a close link be maintained between the conceptual models used in developing a program and the way program features are presented for examination during debugging [34]. Therefore the issues of flexibility and adaptability must be raised with respect to the visualization systems currently employed by debugging tools. By limiting the execution viewpoint to state-based or interaction-based approaches, a debugger restricts its direct applicability to programs developed using specific decompositional methods and certain types of languages. Yet few researchers have acknowledged the importance of supporting alternative mechanisms for visualization, and fewer still have taken steps to accommodate such variations in their designs. If parallel debuggers are to achieve widespread acceptability, efforts must be directed to the development of more flexible visualization systems.

References

[1] Hector Garcia-Molina, Frank Germano, Jr., and Walter H. Kohler. "Debugging a Distributed Computing System." *IEEE Transactions on Software Engineering*, SE-10 (1984): 2: 210-219.

[2] Peter Bates and Jack C. Wileden. "High-level Debugging of Distributed Systems: The Behavioral Abstraction Approach." *Journal of Systems and Software*, 3 (1983): 225-264.

[3] Charles E. McDowell, David P. Helmbold, and Anil K. Sahai. *A Survey of Debugging Tools for Concurrent Programs.* Technical Report UCSC-CRL-87-22, Baskin Center for Computer Engineering and Information Sciences, University of California, Santa Cruz, 1987.

[4] D. Bhatt and M. Schroeder. "Comprehensive Approach to Instrumentation for Experimentation in a Distributed Computing Environment." *Proceedings of the Third International Conference on Distributed Computing Systems*, 1982, pp. 330-340.

[5] Jack J. Dongarra and Danny C. Sorenson. "A Portable Environment for Debugging Parallel FORTRAN Programs." *Parallel Computing*, 5 (1987): 175-186.

[6] Thomas J. LeBlanc and John M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay." *IEEE Transactions on Computers*, C-36 (1987): 4: 471-482.

[7] Barton Miller and Thomas LeBlanc, eds. *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 1988. The papers and session summaries were reprinted in *SIGPLAN Notices*, January 1989; the extended abstracts were omitted.

[8] Cherri M. Pancake. "Conceptual Models in Parallel Programming Languages." *Proceedings ACM Southeast Region 26th Annual Conference*, 1989, pp. 369-375.

[9] Cherri M. Pancake and Donna Bergmark, "Parallel Programming Languages: Do They Meet the Needs of Scientists and Engineers?" under review for publication.

[10] Alan Karp. "Programming for Parallelism." *IEEE Computer*, May 1987, pp. 43-57.

[11] R. C. B. Cooper and K. G. Hamilton. "Preserving Abstraction in Concurrent Programming." *IEEE Transactions on Software Engineering*, 14 (1988): 2: 258-263.

[12] Evan Adams and Steven S. Muchnick. "Dbxtool: A Window-based Symbolic Debugger for Sun Workstations." *Software – Practice and Experience*, 16 (1986): 7: 653-669.

[13] Sequent Corporation. *PDBX Parallel Debugger for Sequent Systems.* Sequent Computer Systems Technical Publications, Beaverton, OR, 1986.

[14] Intel Corporation. *iPSC Concurrent Debugger Manual.* Intel Scientific Computer Corporation Document 310613, Santa Clara, CA, 1987.

[15] James H. Griffin, Harvey J. Wasserman, and Lauren P. McGavran. "A Debugger for Parallel Processes." Technical Report LA-UR-88-1174, Los Alamos National Laboratory, Los Alamos, NM, April 1988.

[16] Jack J. Dongarra and Danny C. Sorensen. "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs." Technical Memorandum 86, Argonne National Laboratory, Mathematics and Computer Science Division, November 1986.

[17] M. Zimmermann, F. Perrenoud, and A. Schiper. "Understanding Concurrent Programming through Program Animation." *Proceedings of the Nineteenth ACM SIGCSE Conference*, February 1988.

[18] M. Zimmermann, F. Perrenoud, and A. Schiper. "Graphical Animation of Concurrent Programs" (extended abstract). *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, May 1989, pp. 342-344.

[19] Sue Utter. "Enhancing the Usability of Parallel Debuggers." Paper presented at the Hawaii International Conference on System Sciences, January, 1989.

[20] Paul K. Harter, Jr., Dennis M. Heimbigner, and Roger King. "IDD: An Interactive Distributed Debugger." *Proceedings of the Fifth International Conference on Distributed Programs*, 1985, pp. 498-506.

[21] 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 SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging. SIGPLAN Notices*, January 1989, pp. 163-173.

[22] Richard J. LeBlanc and Arnold D. Robbins. "Event-driven Monitoring of Distributed Programs." *Proceedings of the Fifth International Conference on Distributed Computing Systems*, 1985, pp. 515-522.

[23] Alfred A. Hough and Janice E. Cuny. "Belvedere: Prototype of a Pattern-oriented Debugger for Highly Parallel Computation." *Proceedings of the 1987 International Conference on Parallel Processing*, 1987, pp. 735-738.

[24] Alfred A. Hough and Janice E. Cuny. "Initial Experiences with a Pattern-Oriented Parallel Debugger," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging. SIGPLAN Notices*, January 1989, pp. 195-205.

[25] David Callahan et al. "Parallel Programming Support in ParaScope." In *Parallel Computing in Science and Engineering*, ed. R. Dierstein *et al.*. Springer-Verlag, New York, 1987, pp. 91-106.

[26] David Socha, Mary L. Bailey, and David Notkin. "Voyeur: Graphical Views of Parallel Programs." *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging. SIGPLAN Notices*, January 1989, pp. 206–215.

[27] Esa Helttula et al. "Graphical Specification of Algorithm Animations with ALADDIN." *Proceedings of the Twenty-second Annual Hawaii International Conference on System Sciences*, 1989, pp. 892-910.

[28] Peter C. Bates, Jack C. Wileden, and Victor R. Lesser. "A Debugging Tool for Distributed Systems." In *Second Annual Phoenix Conference on Computers and Communications*, 1983, pp. 311-315.

[29] F. Baiardi, N. De Francesco, and G. Vaglini. "Development of a Debugger for a Concurrent Language." *IEEE Transactions on Software Engineering*, SE-12 (1986): 4: 547-553.

[30] Edward T. Smith. "Debugging Tools for Message-based, Communicating Processes." *Proceedings of the Fourth International Conference on Distributed Computing Systems*, 1984, pp. 303-310.

[31] Miguel E. Garcia and W. Joseph Berman. "An Approach to Concurrent Systems Debugging." *Proceedings of the Fifth International Conference on Distributed Computing Systems*, 1985, pp. 507-514.

[32] Thomas J. LeBlanc *et al.* "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *ACM SIGPLAN Parallel Programming: Experience with Applications, Languages, and Systems*, July 1988, pp. 164-170.

[33] D. Gentner and A. Stevens, eds. *Mental Models.* Erlbaum Associates, Hillsdale, NY, 1983.

[34] Robert W. Holt *et al. Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation into Computer Debugging.* Technical Report TR-78-A21, U. S. Army Research Institute for the Behavioral and Social Sciences, 1978.