DETOP – An Interactive Debugger for PowerPC Based Multicomputers

Michael Oberhuber and Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik der Technischen Universität München
D-80290 München, Germany

Abstract

Debugging still is one of the most time consuming phases in programming high performance parallel computers. Parallel debuggers can greatly support users in this process and thus reduce time needed for program development. However, most debuggers either don't provide easy-to-use interfaces and therefore are rejected by lots of potential users, or are usable only for a subclass of applications supported by the machine's programming model. In this article we define design goals for parallel debuggers and present the user interface and implementation of our parallel debugger DETOP. DETOP provides a graphical interface that is simple to use even for unexperienced users and supports not only static data parallel codes, but also dynamic programs and applications based on functional decomposition. DETOP is also an excellent example of cooperation and transfer of technology between universities and industry.

1. Introduction

Developments in the field of supercomputing during the last couple of years showed that new classes of applications (The Grand Challenges) gain in importance that demand new dimensions in computing power. These demands can only be saturated by applying parallelism to the solution of such problems. Today several commercially available hardware platforms exist that offer massive parallelism, e.g. the GC/PowerPlus systems. Although improvements on the hardware side are still desirable, e.g., lower communication latencies, the main problems have to be solved in developing applications for such systems. Parallelism increases the efforts that are necessary for developing programs that work correctly and that utilize the theoretical performance capabilities.

The state of the art in programming large scale parallel applications are programs with explicitly stated parallelism. The parallelism is either defined in a programming model with explicit message passing or by giving data distributions like those in High Performance Fortran. To be more efficient the development of correct and efficient parallel programs has to be supported in all phases of the program design and coding. In this article, however, we focus on the support of the debugging process of programs that have been coded and are being executed on parallel hardware platforms. Existing tools in this field have deficiencies that limit their usability to a subclass of the existing programs. Easy to use debugging tools are limited to single threaded programs or to programs that have a static process structure. Programs that dynamically create new processes and consist of multiple executables are not supported by current debugging tools.

We present our parallel debugger DETOP which is now available for Parsytec PowerPC based parallel computing systems (PowerXplorer and GC/PowerPlus) running the PARIX operating system. DETOP supports debugging of applications containing both functional and data parallelism. Applications may consist of multiple threads and multiple program modules. Prototypes of DETOP [1] were developed at Lehrstuhl für Rechnertechnik und Rechnerorganisation, Munich University of Technology (TUM) within the German Science Foundation special research grant SFB 342. Within a cooperation between TUM and Parsytec, the debugger has been improved and adopted to the Parsytec PowerPC based parallel systems. While TUM has ported the debugger and the underlying monitoring system, Parsytec has implemented the necessary debugging support in the nanokernel that is the basis of PARIX on the PowerPC computers. So DETOP is also an excellent example for a successful cooperation and transfer of technology between university research laboratories and industry.

In the next section we give a brief overview on existing debugging tools and will also state their deficiencies. In section 3 we present the primary design goals we wanted to meet. Section 4 introduces the user interface and command set of DETOP, while the next section gives some details on the implementation. A short conclusion and our future plans are presented in section 6.

2. State of the Art

To support the process of detecting and locating bugs in a program, a debugger must provide means to watch the internal execution of that program. There are two different approaches how to gather, analyze and present run time information: The off-line and the on-line approach.

Off-line debugging. Most parallel computing platforms provide tools that allow to record and view event traces, e.g. ParaGraph [2]. Though mostly aimed towards performance analysis, these tools can, to some extent, also be used for correctness debugging. However, the way in which event traces are visualized is not suitable for massively parallel or dynamic systems.

Ariadne [3] uses event abstractions to deal with the difficulty of presenting huge amounts of events to the user. Instead of visualizing the raw event streams, Ariadne allows the user to define abstract events and to ask if and where these abstract events occur. DARTS [4] even goes further and transforms event streams into Prolog fact bases. In this way, the trace can be automatically related to the program's source code. Furthermore, Prolog can be used as a very powerful query language. Although these tools avoid the problem of displaying huge event traces, they still need a global event trace that cannot be gathered in a scalable way. Furthermore, they only allow to monitor a very restricted set of events, usually only communication, so the information is often too course grained to track down the cause of a bug in the source code.

On-line debugging. On-line debuggers allow a more detailed view on program execution. They provide functions to stop threads at given locations in the source code, to view or modify the programs data structures, to determine the source code position where threads are stopped and to execute threads in a controlled fashion, e.g. in single steps. Usually, the user is allowed to specify a group of nodes, processes or threads. A command is then executed independently for each element in that group.

Some of these tools, like ndb or ipd [5] are based on pure textual interfaces that are difficult to use for non-computer scientists. However, debuggers using graphical interfaces, e.g. Prism [6] are becoming more common. There is also a number of graphical interfaces built on top of text-based debuggers, e.g. xipd [7] or Panorama [8]. These tools usually provide visualization of arrays and thread states to ease debugging of complex applications.

However, these advanced features are only usable for data parallel or SIMD-style programs. Current debuggers don't support applications that create or delete threads dynamically or are composed of multiple executables due to dynamically loaded codes. Most debuggers even require a one-to-one mapping between threads and processing nodes, i.e. they support only one thread per node, due to the way breakpoints are implemented and since the visualization of thread states is often based on the fixed hardware topology.

3. Primary Design Goals

Tools that are used to monitor the run-time behavior of parallel programs, like debuggers, must meet a couple of requirements in order to be of practical use. To meet these requirements has been a primary design goal for DETOP.

Ease of use. Several surveys (e.g. [9]) show that one reason for tools not being widely used is the lack of user-friendliness. Usage of tools should be clear and intuitive. Graphical interfaces and strictly guided user interaction can simplify the handling for unexperienced users. However, the interface must also provide a quick (and maybe less intuitive) access to all functions for expert users. Another aspect is the way how gathered data is presented. Information should be displayed in a simple, well known manner, e.g. markers in the source code for the current point of execution or matrix diagrams to visualize arrays.

The most important point is, however, to support a systematic way of usage. In order to give active help to users, debuggers must provide and support a method to track down bugs in the software. A generally accepted method is the top-down approach: Initially, the system (or program) is watched at a very high and course-grained level. When unintended behavior is detected, parts of the system are observed at a lower and more detailed level. This process is repeated until the bug has been located. In order to support this approach, debuggers for parallel systems must provide views at different levels of abstraction, ranging from a global view of the whole system to individual threads, procedures, statements or data structures.

The global view should provide information on the system as a whole and on the interaction between system components. For debuggers this implies the need for global event detection, global breakpoints and monitoring of communication and synchronization. A global view is also supported by methods allowing to specify subsystems, e.g. sets of nodes or threads, by properties like thread state or executed code, rather than explicit enumeration.

Besides a global view, tools must also provide information on levels that are detailed enough to locate a bug. So on an intermediate level, the user must be able to inspect single threads or messages; the finest level should provide a link to the source code, i.e. procedures, source lines and data structures.

General usability. Parallel computing systems usually provide an operating system and compilers that allow to use several programming languages and several strategies for parallelization. A tool environment for such a computing system should therefore support all these languages and parallelization techniques.

Many sophisticated debuggers only support C, although most scientific applications are written in Fortran. Likewise, most tools are only usable for data parallel programs with exactly one process on each of the parallel computer's nodes. This restriction had been reasonable when most systems didn't offer efficient support for multiple processes on a node. However, modern computer architectures and operating systems not only allow efficient scheduling of multiple threads per node, but also dynamic creation of new threads and even dynamic loading of new code. Furthermore, there is a trend towards multiprocessor nodes able to execute multiple threads in parallel within a shared memory. So task partitioning is more widely used in order to overlap communication delays and also as a means of structuring software. Dynamic loading of new code is often used to include dusty deck sequential programs into a parallel wrapper. This trend results in three impacts:

- Multiple threads per node require a more sophisticated implementation of debuggers. For instance, it must be possible to set breakpoints for individual threads, even if the executed code is shared.

- Parallel programs based on both data and task partitioning require more flexible user interfaces. In order to achieve clarity and scalability, it should be possible to group together threads in the data parallel part of the program that execute the same code and behave in a very similar way. However, the tools must also allow to view those parts of the program separately that perform different tasks, e.g. by using several windows.

- Threads that are created or even loaded dynamically induce further requirements on debugging tools: They must provide identifiers for these threads that are more meaningful to the user than a pointer to some structure in the operating system. A debugger must allow to monitor such threads, i.e. it must be able to stop a dynamically created thread before it starts execution. In order to support source level debugging, the debugger must also be able to determine the thread's binary file and load its symbol table, if necessary. Dynamic creation of threads creates severe problems for most visualization strategies that require the number of threads and their topology to remain constant.

Minimal intrusion. Computation and communication overhead introduced by the debugger must be kept as low as possible. Delays created by monitoring the program can lead to significant changes in its timing behavior and may even change its logical behavior (*probe effect*). In order to keep these delays as low as possible, a debugger should only monitor those events that are of current interest for the user. Usually this is only practical by an on-line mode of operation.

Besides the overhead in time, also the overhead in space is an important factor, since most multicomputer systems don't provide virtual memory. Therefore only a very small part of a tool (the monitoring system) should run on the same nodes as the monitored program. Otherwise, the program is likely to fail due to insufficient free memory. Even with virtual memory, exceeding physical memory size should be strictly avoided, since it results in a dramatic change of program behavior.

Scalability. Sometimes logical errors do not appear when a program is applied to small test input sets. Therefore it may be necessary to analyze its behavior in real application. This refers to the system size the program is executed on as well as the input data set. So tools are required to be scalable in both the collection and the presentation of data.

By scalable data collection we mean that the amount of data exchanged between tool and monitoring system must not grow with the system size, i.e. the number of nodes or threads. To achieve this goal, the monitoring system must be able to perform an autonomous distributed data acquisition and analysis, for instance to detect global events and execute the associated actions. The monitoring system must also be able to efficiently distribute commands and collect and combine the individual results. The tool may then send a single command and receive a single result, independent of system size.

Scalable data representation includes summarizing of information, focusing on relevant parts of data, hierarchical views and presentation of analyzed, more abstract data.

4. The Parallel Debugger DETOP

DETOP provides a graphical user interface whose main component is a debugger window as shown in Fig. 1. For data parallel applications, a single window is often sufficient and offers the most convenient and comprehensive way to debug all threads of the program. However, in order to support also other paradigms of parallel programming, especially functional decomposition, DETOP has been designed to allow multiple debugger windows. Each of these windows may be associated with an arbitrary set of threads. So for data parallel programs a single window can be used for all threads, while different windows can be used for the different parts of applications based on functional decomposition. Multiple windows also make possible to use one window for a course grained overview of the whole application, while another one is used for detailed inspection on one selected thread.

The center of each window is a listing of the threads' source code. The current locations of the threads associated with this window and the breakpoints set for these threads are indicated in the listing by small icons and additional descriptions inserted into the source code (see Fig. 1). The listing is also used for fast and intuitive command input. We use an input scheme that is object oriented in some sense: The user may select an object in the listing, e.g. a line, a variable name or a breakpoint marker. Based on the type of this object, he will get a

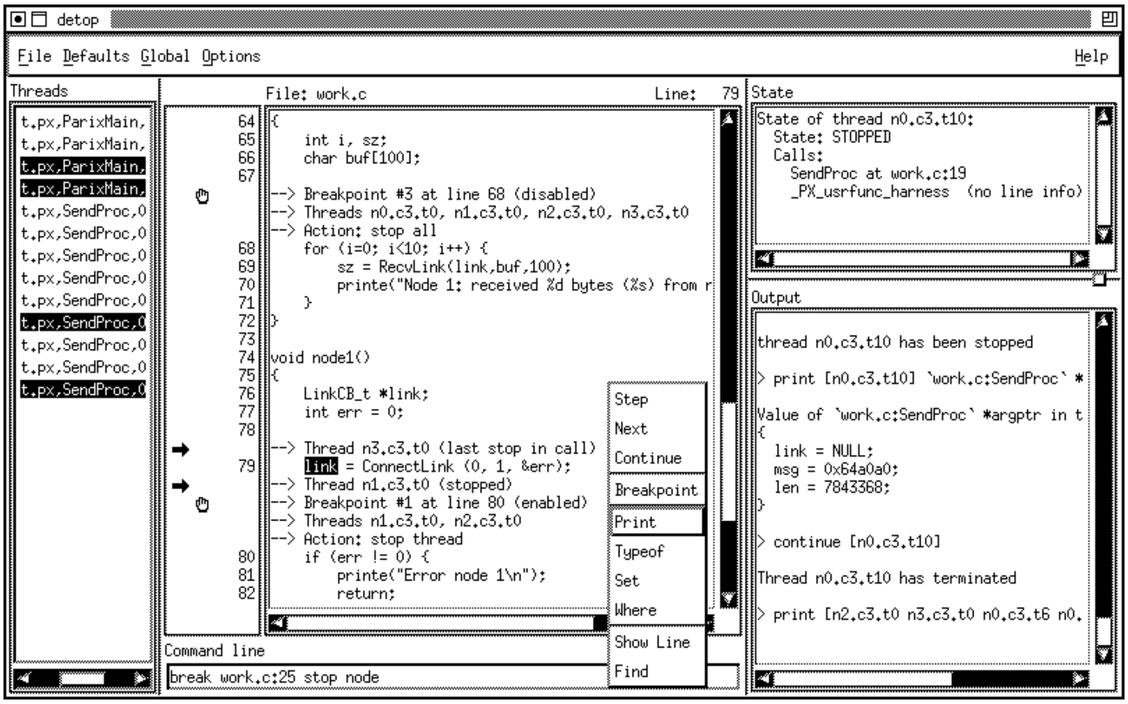

Figure 1: DETOP debugger window

popup menu offering commands appropriate for the selected object. So if a line or variable is selected, the menu will allow to set a breakpoint or to print the type or contents of the variable. If a breakpoint marker has been selected, the menu will offer items to delete or modify this breakpoint. In contrast to other debuggers providing graphical input, commands use both the selected piece of text and the position of this text in the source code. So if a variable `i` is selected in a procedure `foo`, the `print` command will print the value of `i` in this procedure, even if it is not on top of the call stack. This is what users expect intuitively, so usage is greatly simplified. In addition to menu input there is also a command line for more experienced users and complex commands. Command line syntax supports both C and Fortran 77.

On the left side, the debugger window shows the list of threads currently associated with that window. The list may be changed at any time using a separate window (see Fig. 2) that allows to view all currently existing threads and to assemble the new selection. In this list a subset of threads may be selected. All commands entered in the debugger window will then be applied to all selected threads. In addition, there are also global commands that always affect the whole application, e.g. a global stop.

DETOP provides two different forms of command output. Output may either be appended to a protocol or updated in-place. The first mode is used for all unconditional commands, but may also be used to generate traces of events or other data. The second mode is, for example, used to display the state and procedure traceback of threads. Each time a thread is stopped, the state display (in the upper right of Fig. 1) is updated and shows the current stopping point.

The debugging commands offered by DETOP fall into six major groups:

1. *Execution control:* It is possible to start, stop and single step any thread at any time. In addition, there is also a global start and stop.

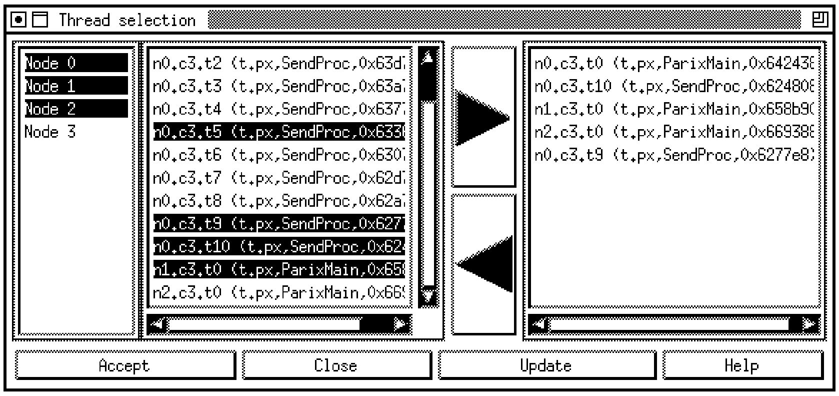

Figure 2: DETOP thread selection window

2. *Inspection of program state:* These commands allow to retrieve information on the program's current state, e.g. the list of all existing threads, the point of execution and procedure traceback of threads and the types and values of variables or expressions.

3. *Modification of program state:* Currently only modification of variables is supported.

4. *Events and actions (breakpoints):* Breakpoints may be set on source code lines or procedure entries for an arbitrary set of threads. A breakpoint consists of an execution event and an associated stop action. The event is raised whenever one of the selected threads reaches the given position in the source code. The stop action can either stop the thread that raised the event, the thread's processing node or the whole application. The latter modes are essential in order to obtain consistent views of shared variables or the program's global state.

   In addition, there are several events that are permanently monitored by DETOP, e.g. exceptions or termination of a thread. It is also possible to define a special breakpoint that will stop dynamically created threads immediately before they start executing.

5. *Miscellaneous:* There are also commands to display a source file, to set defaults, e.g. a default action for breakpoints, and to configure the graphical interface.

6. *Help:* DETOP also provides full context sensitive on-line help.

Since support for dynamic applications has been a major design goal, there are also special features that ease debugging of programs that dynamically create new threads. Threads are identified by a combination of a global identifier and additional information related to the source code. The global identifier consists of the node number, a context number and a thread number. The context number is used to distinguish between threads having different executable files. The thread number starts at 0 for the initial thread and is incremented for each dynamically created one. Additional information is used to make identification more easy for the user:

- For applications that dynamically load new code, the name of the executable file is given for each thread.

- The name of the thread's root function is given in order to support applications based on functional decomposition.
- Finally, the pointer to the thread's control block, as returned by the `CreateThread` library call is shown. If the pointer has been stored in some program variable, the corresponding thread can be identified in this way.

Threads are always selected from lists displaying the global identifiers and the additional information (see Fig. 2). Once a thread has been selected for debugging, its symbol table is automatically loaded into the debugger, if it is not already available. Thus, only the symbol tables needed for the current debugging session are loaded; there is no need for the user to explicitly specify the symbol tables to be used.

DETOP also supports two additional event types that are essential for debugging dynamic applications. Termination of debugged threads is permanently monitored. When a thread terminates, a message is printed and the thread is removed from all thread lists. A special option requests the debugger to monitor also creation of new threads. When a new thread is created, a message containing the thread's identifier is printed, and the thread is stopped immediately before it starts executing. It is then possible to select this thread for debugging, to set breakpoints in this thread and to start it. So even a dynamically created thread can be debugged from the very beginning of its execution.

5. Implementation

The debugger is built in a strictly hierarchical manner in order to ease portability and maintenance. The debugger's basis is a distributed monitoring system that consists of one monitor on each of the parallel computer's nodes. DETOP itself can be executed on any workstation allowing a network access to the parallel computer. The host part is again divided into two layers: the graphical user interface and an underlying transformation layer. The main tasks of this layer are transformation between source language level and machine level, i.e. symbol table management, distribution of commands to the appropriate monitors and the management of events and actions.

The transformation layer is connected to a communication server via TCP/IP. This server, which is part of the monitor on Node 0 for the GC/PowerPlus and PowerXplorer implementation, acts as a bridge between the TCP/IP network and the monitoring system's internal communication network. Communication between monitor system and debugger is completely asynchronous, so in contrast to sequential and most other parallel debuggers neither the debugger interface is blocked while a monitor is busy on some command, nor it is necessary to interrupt the whole application in order to issue debugger commands.

In the following, we will discuss three aspects of DETOP's implementation, the monitoring system, symbol table management and the user interface in more detail.

5.1. Monitoring system

The task of the monitoring system is to provide the tools with adequate informations. Since monitoring is not only required for debugging a universal and flexible monitoring system is used. All other members of the TOPSYS tool family [10], like the performance analyzer PATOP [11] and the visualizing tool VISTOP [12] are also based on this monitoring system.

As mentioned above the monitors are distributed over the corresponding computing nodes. But to have one monitor on each node does not mean to have a distributed monitor. The monitors need to interact in order to form a common monitoring system for several common tasks. Communication is necessary for a couple of different reasons. First, there is a need for global commands that affect all involved nodes. Since the connection between the host and the target is usually much slower than the inter node connection, it is more efficient to send only one message to a dedicated node (Node 0 in our case), where the message is replicated and sent to all other nodes using a spanning tree for distribution. The same way global data is gathered. Besides this kind of broadcast, a multicast communication between monitors is provided. Second, the communication is not limited to spread and gather data but it is also used to exchange information for recognizing distributed events and triggering distributed actions.

To achieve the goal of a universal monitoring system, a programmable monitoring interface has been implemented. Our approach separates the syntax from the semantics of the command interface [13]. This means that there is only a defined syntax allowing arbitrary commands that provide the semantics. In this way it is possible to offer a complete and orthogonal interface for several tools. To take advantage of these properties, the commands can be combined in different ways, so the total number of commands is quite low. Commands can be combined in a parallel way, i.e. they are executed independently, and as a pipeline, i.e. part of a command's input is taken from the output of a previous command. For example, in order to determine all threads on one node and the position where they are currently stopped, it is possible to combine a command called view with a command called inspect. view delivers all threads and pipes its output to inspect, which locates the current position. Using our syntax it looks like

$$\{\ \mathsf{view}\texttt{['T']}\{\ \mathsf{inspect}\texttt{[?]}\}\}$$

The parameter 'T' asks view to return the list of existing threads, the question mark symbolizes that inspect expects the output of view as its input parameter. Without such a pipelining support, it would either be necessary to call both commands subsequently and therefore causing a lot of communication traffic between the tool and the monitor, or to create a special command that overlaps already existing ones. Both alternatives are in contrast to the general objectives of our monitoring system, i.e. to avoid disturbance and to keep the monitor as small as possible.

In summary, the programmable monitoring interface results in

- a simple adaption to new demands and
- a powerful and flexible command set using only a small number of basic commands.

Another important point for monitoring parallel programs is to recognize events and to initiate the corresponding event handling. The handling may also imply the execution of certain actions defined for this kind of event. The programmability of our monitor interface

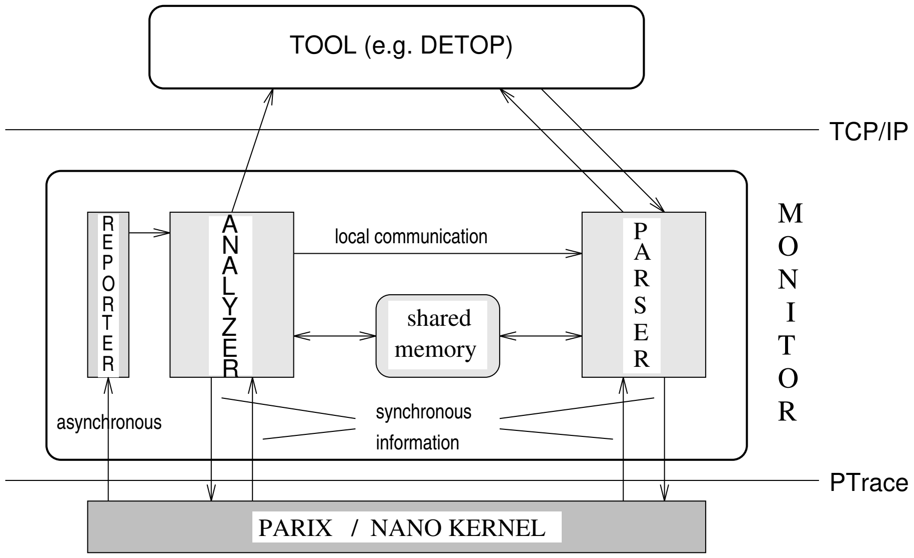

Figure 3: Monitor on one node

also covers the aspect of event–action management. This means, there is no fixed event–action relation, but each available action function and event handler can be assigned to an event. Additionally, it is possible to use any monitoring command as an action, since the interface is the same. An example is to set a breakpoint at some address in a thread when a certain communication event occurs (not necessarily on the thread's node). In this way, our monitoring system's programmable event–action management allows each tool, e.g. a debugger, to choose the event handlers and the actions it needs for its purposes.

Since some events are asynchronous in nature the interface between the monitoring system and the tool should reflect this fact, i.e. the instrumentation by the tool and the feedback are decoupled. This is especially important in considering dynamic process models where light weight processes (in general called threads) or heavy processes (new programs) are created or terminate. If there is a request for these events by the tool, the monitor must signal them without any delay, however the tool must not be blocked due to busy waiting.

A specific problem of parallel programs concerns global events [14]. Global events can only be observed by the monitoring system as a whole, because they are a combination of simple events occuring on different computing nodes. This implies also a communication between the monitoring entities. With global events arises the problem of global time. The most common approach to solve this problem is to use logical clocks [15], e.g. vector time-stamps [16]. But for on-line debugging we prefer a more pragmatic way that is much more efficient and, as practical experience shows, sufficient too. We do not use causal relationships but simply try to inform the corresponding nodes as fast as possible. This mechanism is realized by the event–action management sending a command to the corresponding node as described above.

The implementation of our monitoring system is multithreaded. There are two main threads, called Analyzer and Parser. Figure 3 shows the main structure of a monitor. The task of the Parser is to scan, parse and interpret commands. The analyzer is responsible for the event handling and for initiating actions. A third thread called Reporter is used to preprocess and filter asynchronous events. On Node 0 there are two additional threads dealing with host communication. These threads are able to receive and send messages independently. Using a multithreaded design guarantees short response time and asynchronous event handling. It is especially effective on GC/PowerPlus nodes where two processors provide real parallelism on one node.

The monitor's interface to the operating system is similar to the Unix PTrace interface. However, it is modified according to the dynamic and multithreaded process model of PARIX. The standard interface allows to monitor system calls, to read and write the code and data segments of any thread and to determine and modify a thread's state. In addition the PARIX PTrace allows to watch dynamic events, like creation or termination of threads. These events are signalled in an asynchronous way, so the monitor is not blocked in the PTrace call but is free to perform other operations until the event occurs.

5.2. Symbol table management

When a program is compiled for debugging, compilers emit a symbol table that is stored in the program's executable file. The symbol table is used to link machine addresses to source code items, e.g. variables, procedures or lines in a source file. Though this information is mainly needed for correctness debugging, it is also important for performance analysis, if performance values have to be related to the source code, e.g. by restricting a measurement to a some procedure. There is a couple of different symbol table formats and also a variety of extensions for different source languages. In order to keep our tools portable, we have developed a symbol table management library that can read various formats and offers a uniform interface for accessing symbol table information that is independent of format and source language. Since the library is able to manage several symbol tables at the same time, it allows us to support both applications where code is loaded dynamically, i.e. applications consisting of more than one executable, as well as programs on heterogeneous systems, where there are different executables for the different types of node processors. Mixed language programs, e.g. written in C and Fortran are also supported.

Fig. 4 shows the structure of this symbol table manager. A common reader first opens the executable file and determines the target machine, its byte order and the symbol table format used. Next, a format-specific reader is called building up an internal representation that is independent of format and programming language. The internal representation is a tree-like structure where each node represents a symbol. The interface to access this structure acts like a pattern matcher. Given a start node representing a context, e.g. a source file or procedure, all symbols matching the given specification are returned. Specification consists of symbol name or address, symbol class (e.g. procedure, variable, type) and subclass (e.g. global symbol, parameter), where wildcards are possible for each component. Various search types allow to look for all descendent symbols, for local symbols only, or for symbols within scope of the program structure represented by the start node. Thus, the interface provides a uniform support for both searching specific symbols and retrieving the set of all symbols with a given class, e.g. a list of all procedures in an application.

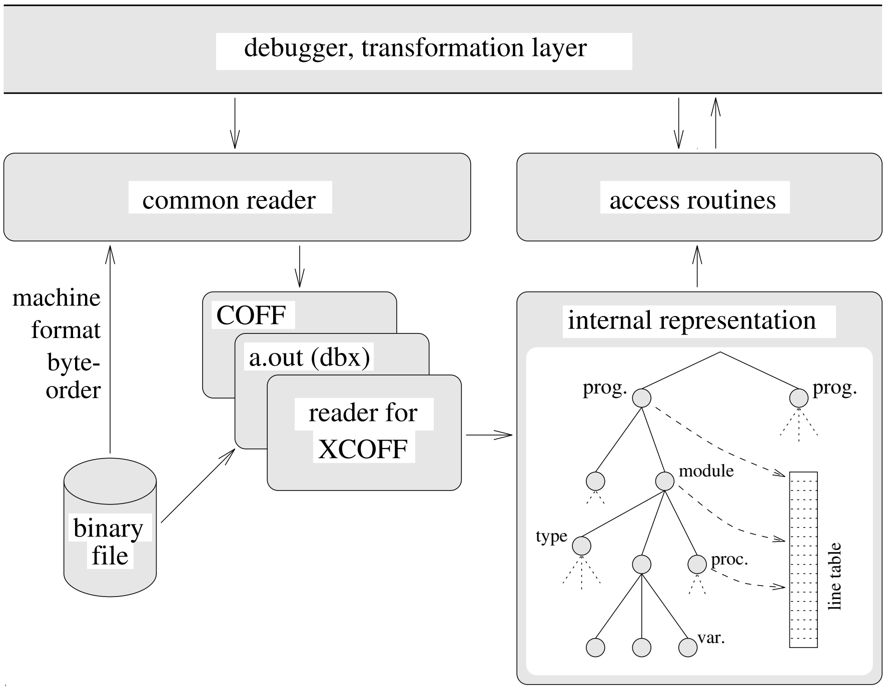

Figure 4: Structure of symbol table management library

5.3. User interface

The user interface has been built on top of X windows using the Motif widget set. We have used C++ and the Widget Creation Library (Wcl) for rapid prototyping of the debugger window. Wcl allows to specify the complete hierarchy of widgets together with all the needed resources and callbacks in a single resource file. This file is interpreted by a special Wcl function creating the debugger window. So the window's layout and the binding between buttons or menu items and debugger functions can be modified without need to recompile the debugger.

The context sensitive popup-menus are implemented in the following way: The source code listing and the debugger's output are organized as a list of text objects. Each text object is a correlated piece of text with a specific type, e.g. a source line, a breakpoint marker or the output of a command, together with its internal representation as a data structure and additional information. Each text object has an associated callback function that is invoked when the user has selected a piece of text inside this object and presses the mouse button to get the popup-menu. The callbacks will then generate a menu containing commands useful for the selected object type. In addition, the selected piece of text and its position within the text object is passed to the callback. Together with the data structure associated with the text object, this information is used to build the parameters for commands invoked via the popup-menu.

6. Conclusion and Future Work

The paper presented DETOP, a debugger for parallel programs providing support not only for static data parallel programs, but also for dynamic applications and programs based on functional partitioning. The debugger is distributed by Parsytec for their GC/Powerplus and PowerXplorer systems since Dec. 1994. It is complemented by the performance analysis tool PATOP [11], also developed by our research group, to provide a comfortable programming environment for these parallel machines.

Our future plans are to extend the capabilities of DETOP for debugging communication between threads, e.g. by allowing breakpoints to be set on links. A thread sending to or receiving from the link will then be stopped before it completes communication, so it is possible to watch communication on a link. This type of breakpoint also allows to stop the receiver of a message, so processing of that message can be debugged. In order to implement this feature, the monitoring system needs more support from the PARIX programming library and the underlying nanokernel. This support will be provided by Parsytec in a future cooperation.

Other issues are support for C++ and visualization of data structures and program state within the debugger. Finally we will port both DETOP and PATOP to the PVM and MPI parallel programming models. Implementation is planned both for versions running on networks of workstations and for a version running on top of PARIX.

References

[1] T. Bemmerl and R. Wismüller. On-line Distributed Debugging on Scalable Multiprocessor Architectures. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking, Volume II: Networking and Tools*, volume 797 of *Lecture Notes in Computer Science*, pages 394–400, München, April 1994. Springer-Verlag.

[2] Michael T. Heath. Recent Developments and Case Studies in Performance Visualization using ParaGraph. In G. Kotsis and G. Haring, editors, *Proc. of Workshop on Monitoring and Visualization of Parallel Processing Systems*, pages 175 – 200, Moravany nad Váhom, CSFR, October 1992. Elsevier, Amsterdam, 1993.

[3] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple. The Ariadne Debugger: Scalabe Application of Event-Based Abstraction. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–95, San Diego, California, May 1993. ACM.

[4] M. Timmerman, F. Gielen, and P. Lambrix. High Level Tools for the Debugging of Real Time Multiprocessor Systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 151–157, San Diego, California, May 1993. ACM.

[5] D. Breazeal, K. Callaghan, and W. D. Smith. IPD: A Debugger for Parallel Heterogeneous Systems. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 216–218, Santa Cruz, CA, May 1991.

[6] D. Allen, R. Bowker, K. Jourdenais, J. Simons, S. Sistare, and R. Title. Data Visualization and Performance Analysis in the Prism Programming Environment. In *IFIP Working Conference on Programming Environments for Parallel Computing*, Edinburgh, April 1992.

[7] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith. The paragon performance monitoring environment. In *Proc. of Supercomputing '93*, pages 850–859, Portland, Or., November 1993. IEEE.

[8] J. May and F. Berman. Panorama: A Portable, Extensible Parallel Debugger. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 96–106, San Diego, California, May 1993. ACM.

[9] C. Pancake and C. Cook. What Users Need in Parallel Tool Support: Survey Results and Analysis. Technical Report CSTR 94-80-3, Oregon State University, Dept. of Computer Science, Corvallis, OR, June 1994.

[10] T.Bemmerl and A. Bode. An integrated tool environment for programming distributed memory multiprocessors. *Distributed Memory Computing*, 487 of LNCS:130 – 142, 1991.

[11] O. Hansen. A tool for optimizing programs on massively parallel computer architectures. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking, Volume II: Networking and Tools*, volume 797 of *Lecture Notes in Computer Science*, pages 350 – 356, Munich, April 1994. Springer-Verlag.

[12] T. Bemmerl and P. Braun. Visualization of Message Passing Parallel Programs with the TOPSYS Parallel Programming Environment. *Journal of Parallel and Distributed Computing (Special Issue on Tools and Methods for Visualization of Parallel Systems and Computations*, 18(2):118 – 128, June 1993.

[13] Thomas Treml. *Monitoring Paralleler Programme*. PhD thesis, Munich University of Technology, 1994.

[14] D. Haban and W. Weigel. Global Events and Global Breakpoints in Distributed Systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, volume II, Sortware Track, pages 166–175, 1988.

[15] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] C. J. Fidge. Timestamps in message–passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Februar 1988.