Interactive Debugging and Performance Analysis of Massively Parallel Applications

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

Olav Hansen
Parsytec Computer GmbH, Postfach 370167
D-52035 Aachen, Germany

Abstract

In the field of high performance computing, massively parallel processing systems (MPPs) get more and more important. A rising number of complex applications is parallelized for execution on these machines. Still a significant portion of the time needed for parallelization is spent for the process of debugging and performance tuning. A main reason for this fact is the absence of adequate tools supporting this phase of program development.

In this article, we present a novel tool environment, consisting of a parallel debugger (DETOP), a performance analyzer (PATOP), and a common monitoring system for PowerPC-based parallel computers. The environment specifically addresses the topics of scalability, usability for dynamic, multithreaded programming models, minimal intrusion, debugging and tuning methodology and comfortable user interfaces. We derive requirements for tools monitoring the runtime behavior of parallel programs, present the concepts used to meet these requirements in our tool environment, and describe its implementation and its usage.

DETOP is based on the event-action paradigm and supports both data parallel codes and programs based on functional decomposition. Special features are provided for applications that dynamically create new threads or consist of multiple executables. PATOP supports a systematic search for performance bottlenecks in massively parallel applications using the concept of attributed measurements and distributed evaluation. Both tools are based on a common, distributed on-line monitoring system providing the necessary runtime information.

Index Terms: Parallel programming, performance analysis, debugging, distributed monitoring

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 massive parallelism to the solution of such problems. We consider the use of at least one hundred processors to be massively parallel. Today, several commercially available hardware platforms exist that offer massive parallelism, e.g. the GC/PowerPlus systems marketed by the German supercomputer vendor Parsytec. 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 and optimization process of programs that have already 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 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 or consist of multiple executables are not supported by current debugging tools.

In performance debugging the main deficiency concerns scalability issues. Current tools are not scalable in the way they collect and present performance data. In addition, they do not provide an adequate support of tracking down performance bottlenecks to their origins in a structured way.

We present our parallel debugger DETOP and our performance measurement tool PATOP which are available for Parsytec's PowerPC based parallel computing systems. PATOP especially supports measuring of programs on massively parallel systems. DETOP supports debugging of applications containing functional and data parallelism. Applications may consist of multiple threads and multiple program modules.

Prototypes of PATOP [BHL90] [Han94] and DETOP [BW94] have been developed at Lehrstuhl für Rechnertechnik und Rechnerorganisation, Technische Universität München (LRR–TUM) within the German Science Foundation special research grant SFB 342 and the ESPRIT projects HAMLET and PREPARE. Within a cooperation between TUM and Parsytec, the tools have been improved and adopted to the Parsytec PowerPC based parallel systems. While TUM has ported the tools and the underlying monitoring system, Parsytec has implemented the necessary debugging support in the operating system kernel. So PATOP and DETOP are also an excellent example for the successful cooperation and transfer of technology between university research laboratories and industry.

In Section 2 we will introduce important requirements for tools monitoring the run-time behavior of massively parallel applications and give a brief overview on existing tools in the field. Section 3 presents the tools' features and user interfaces. Section 4 gives a short overview of the target system the tools were realized for and Section 5 describes the most important aspects of the implementation of the tools. Concluding remarks are given in Section 6.

2 Requirements

Tools that are used to monitor the runtime behavior of parallel programs, like debuggers and performance analyzers, must meet a couple of requirements in order to be of practical use. Meeting these requirements has been a primary design goal for DETOP and PATOP. In this section we discuss these requirements and how they are met by existing tools.

Support of methodical application

Tools like debuggers and performance analyzers are used for a specific purpose, namely to locate logical bugs or performance bugs in a program. In order to help users in this purpose, tools must provide and support a method to track down such bugs. A generally accepted method is the top-down approach: Initially, the system (or program) is watched at a very high and coarse grain 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, tools 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 performance values, 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. Thus, an intermediate level must enable the user 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.

The debugging system called Prism [SAB+94] is worth mentioning in this context. It provides the possibility to create semantically meaningful sets of processors. This allows the observation of erroneous programs at an abstract level before turning to source code.

The IPS-2 system [MCH+90] supports a stepwise refinement process in detecting and tracking down performance bottlenecks. It is possible to look at data for the complete system, individual nodes, processes, and procedures. Profile values can be produced to identify the processor states and nodes that cause performance losses. Values with high resolution in time can be produced to identify the program phases where performance losses occur. Finally, with process and procedure related values it is possible to detect the program parts that cause performance losses.

The W^3 search model [HM93] introduces three axes along which performance bottlenecks can be tracked down. The "why" axis gives a hierarchy of potential performance bottlenecks. Along this axis it can be examined whether a program execution shows the symptoms of any of the performance bottlenecks on the axis. The "where" axis gives the means to identify the resources (e.g. synchronization objects, code, discs) that are related to the performance bottleneck. The "when" axis gives the possibility to identify phases in the program execution when certain bottlenecks occur. Such phases can for example be defined by source code annotation or execution predicates. Examples for such predicates are "program enters function A" or "a synchronization delay is above a threshold". The model also includes an automated guidance that gives hints for further refinement steps on the different axes. The W^3 model gives a well structured picture of the axes along which performance bottlenecks have to be tracked down. It lacks, though, on a general concept on how to do the refinement along these axes in a series of small steps.

Clear and intuitive user interface

Several surveys (e.g. [PC94]) show that a main reason for tools not being widely used is the lack of user-friendliness. Usage of tools should be clear and intuitive. Graphical interfaces and strongly 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. curves and bar graphs for performance data, markers in the source code for the current point of execution or matrix diagrams to visualize arrays. In addition it must always be clear for the user what is displayed in a diagram. Fancy displays showing a wide range of different information usually are very impressive, but are often too complex for being of any practical use.

Some of the debugging tools, like ndb or ipd [Int91] are based on pure textual interfaces that are difficult to use for non-computer scientists. However, debuggers using graphical interfaces, e.g. Prism [SAB+94] are becoming more common. Prism additionally tries to filter identical messages from different nodes in order to reduce the amount of information. There is also a number of graphical interfaces built on top of text-based debuggers, e.g. xipd [RAA+93] or Panorama [MB93]. But, however, with graphical user interfaces arises the problem of cluttered displays when observing multiple processes [SFG94].

Many performance measurement tools like for example Paragraph [HE91] [RAA+93] offer a great variety of displays to show different aspects of performance behaviour. The main deficiency in user interfaces of these tools is their inadequate support of measurement specifications.

Support of parallel programming models

Parallel computers 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 that are able to execute multiple threads in parallel within a shared memory. Thus, 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.

For tool environments, three impacts result from this trend:

- Multiple threads per node complicate implementation of both debuggers and performance analyzers. For instance, it must be possible to set breakpoints for individual threads, even if the executed code is shared. Performance values measured for single threads must take account of thread scheduling.
- Parallel programs based on both data and task partitioning require the tools to provide 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 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. They also must allow to monitor such threads. For a debugger this implies that 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 also creates severe problems for most visualization strategies that require the number of threads and their topology to remain constant.

Current debuggers do not 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.

Low intrusion

Computation and communication overhead introduced by debuggers and performance analyzers must be kept as low as possible. Delays created by monitoring the program's behavior can lead to performance values that differ significantly from those of an undisturbed program. In contrast to sequential programs, where it is possible to compensate for these delays by an automated data analysis, the effects of monitoring overhead on parallel programs cannot be determined in a general way. Delays due to monitoring may even change the logical behavior of parallel programs (*probe effect*[Gai86]).

In order to keep these delays as low as possible, a tool should only monitor those events that are of current interest for the user. Usually this is only possible by an on-line mode of operation.

Besides the overhead in time, also the overhead in space is an important factor, since most MPPs do not 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

Many performance bottlenecks and also some logical errors do not appear when a program is scaled down. Therefore, it is 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. Thus, 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 analysis, for instance, to compute mean values of performance data or 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. Couch [Cou93] defines a display to be scalable, if its clarity, meaning, and size are independent of the number of processing elements involved in the observed program's execution. He shows that scalable displays can be achieved by displaying categorical statistics about system behavior.

The performance measurement tools for parallel programs that have the widest distribution are Paragraph [HE91] [RAA+93] and the tools belonging to the EXPRESS environment [Par92]. Paragraph and the EXPRESS tools offer scalable displays that give overview information on the system performance, and they also offer displays to link local performance behavior to the program parts that cause them. Their deficiency with respect to the presentation of performance behavior is that they do not offer a way of small steps to go from overview information to detailed information. Their implementation is not scalable because they are trace-based and work with a single trace file for all nodes.

With IPS-2, trace data can be stored and processed in a distributed way when IPS-2 is used on a network of workstations that have local disks. IPS-2 is in principle able to handle any number of nodes. Global performance effects can be tracked down in space and time by stepwise refinement. This scheme is usable up to a certain extent; however, when looking at long program runs, where the locality of performance problems changes, stepwise refinement seems to be too much overhead for localizing aspects of a program's performance behavior. Also, the implementation of IPS-2 is only scalable to a certain extent because trace data can be stored and processed in a distributed way; it is limited by a centralized processing scheme for combining values from different locations.

3 The Tools

3.1 DETOP

The process of debugging covers two different aspects: The first is to watch the internal execution of a program in order to detect whether it shows some erroneous behavior. If there is some incorrect behavior, the second step is to locate the cause of the error. These aspects are best supported by a top-down approach as stated in Section 2. At the highest level, a coarse grain overview on the program's execution is useful, showing the existing processes and their interaction from a global point of view. A medium level provides detailed information on the states of single processes and on communication behavior, e.g. the contents of message queues. In order to track down the cause of an error, the lowest level must allow watching the execution of single processes at the level of the source code, e.g. to determine the executed statements and the contents of variables or data structures.

DETOP is based on four basic concepts to efficiently support this top-down approach to debugging: on-line operation, event-action paradigm, asynchronous debugger interface and multiple debugging contexts. Moreover does it provide a powerful graphical user interface and special features for dealing with highly dynamic process models.

On-line operation

Debugging is a very intuitive and interactive process, since the programmer approaches the cause of an error step by step, based on previously gathered information. The type of information that is needed to locate a bug is not yet known when the program is compiled or started. Therefore, off-line approaches are of limited use, since they require that the type of information to be traced is defined at the latest when the program is started. Since tracing all possibly interesting data is prohibitively expensive, off-line tools usually don't support the finest granularity stated above, that is essential for locating bugs.

We therefore use an on-line distributed monitoring system that selectively retrieves the currently needed information and also allows to influence the running program. The monitoring system operates at the machine language level; the necessary translation to source language level is performed by the debugger's host part. Due to selective monitoring and since no symbol tables have to be stored in the monitoring system, its usage of the parallel processor's resources, like CPU time, communication bandwidth and memory, is very low.

Event-action paradigm

Interactive debugging is usually characterized by stopping the monitored program at interesting points and inspecting its state. This is reflected in the fact that most debuggers operate in two modes: Either the program is executing while the debugger monitors breakpoints or the program is stopped while the debugger is used to inspect program state. However, this scheme requires an unnecessary amount of user interaction, especially when only a coarse grain overview on the program's execution is desired, and also yields a large delay, since the whole program is stopped for long periods of time. Programmers therefore often use `print` statements for coarse grain debugging that provide the necessary information at relevant program points without the need for repeated interaction. However, changing the instrumentation is very expensive due to the necessary recompilation, so this scheme is rather unproductive.

The event-action scheme avoids this deficiency. It allows to define points of interest in a program's execution by specifying events. Events are not limited to textual points in a program, e.g. source lines or procedure entries, as usual breakpoints are, but also include, for example, creation of new threads, termination of threads, exceptions and sending or receiving of messages. Additional parameters allow to restrict the events to specific threads or communication links. Each event may be associated with arbitrary actions like displaying the current position in the program text, printing variables or visualizing a data structure. Of course, stopping the program, a single processing node or a single thread is also possible.

Two different forms of output, appending to an output protocol and updating in-place, allow both to generate event traces and to create displays showing the program's last state. In combination with data visualization, the latter mode can be used to create animations. Since the update can be triggered by any event, the places in the source code where the visualization is to be updated can be defined on-line without changing the program's source code.

Actions may also enable or disable the monitoring of other events, so the event-action scheme provides a natural way of specifying complex combinations of events. Furthermore, since the actions may regard processing nodes different from the one where the triggering event occurred, monitoring of global events is possible, as well as global actions. In order to achieve minimal intrusion and maximum scalability, we don't follow the usual scheme where the monitoring system sends each event occurrence to the debugger that in turn starts the action, but translate actions to machine level and distribute them among the monitors at event definition time. Events are then monitored in a fully distributed fashion. Once an event is detected, the monitoring system autonomously performs the actions and sends the results to the debugger asynchronously, without need for stopping the application.

Asynchronous debugger interface

In order to fully support the event-action scheme, our debugger is able to receive, process and display the results of an action in an asynchronous way. The same scheme is also used for unconditional debugger commands, so the interface is completely asynchronous and mode-less. The user neither has to stop the whole application in order to invoke debugger commands, nor is the debugger's interface blocked while waiting for commands to finish. On the one hand, this simplifies the debugger's usage, on the other hand, it allows commands concerning different processing nodes to be executed in parallel. In addition, an asynchronous interface is needed to avoid deadlocks, e.g. when single-stepping a thread over a receive call where the sending thread is currently stopped.

Multiple debugging contexts

As stated above, a debugger is used both to get a coarse grain overview on a program's execution and for detailed inspection of program parts. Multiple debugging contexts allow to clearly separate data at different levels of granularity. Each context consists of a subset of the application's threads, events monitored in these threads and the associated actions. In our implementation, each context is represented by a debugger window showing the threads, a source code listing and the output of events and actions. Multiple contexts are also profitable to debug different parts of an application without intermixing the results.

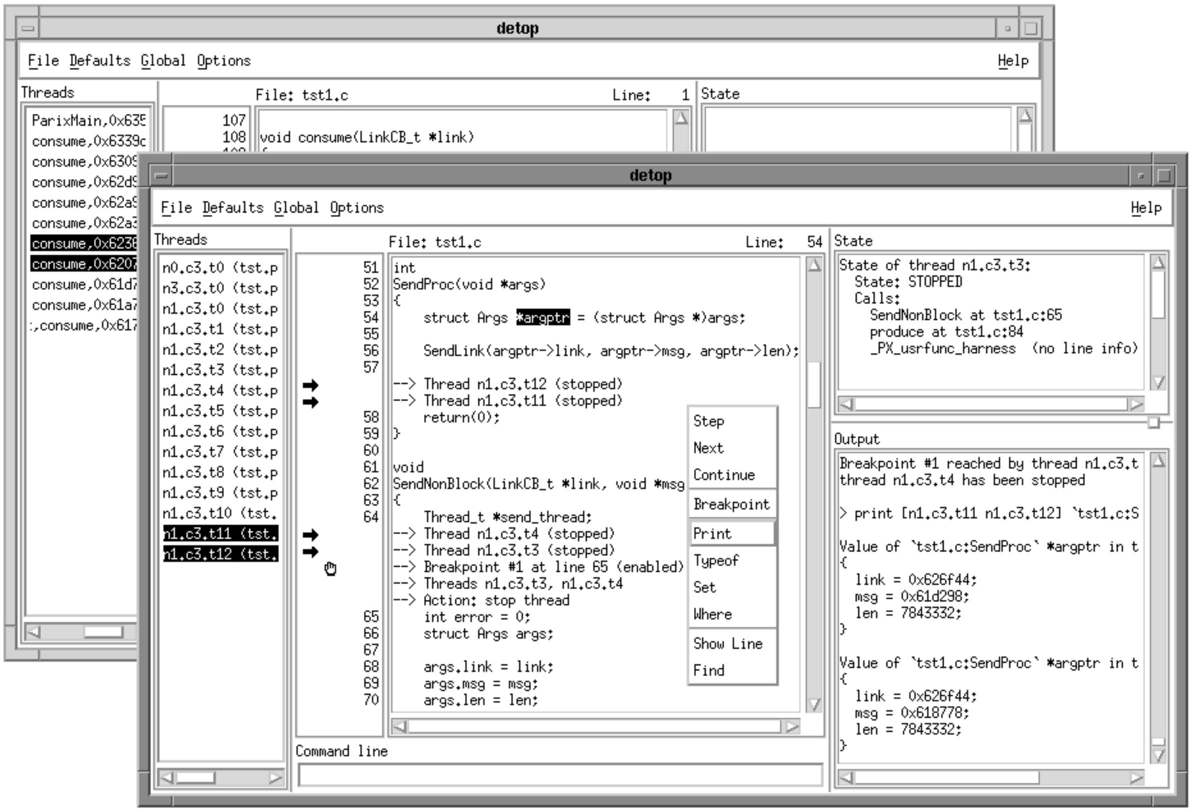

Figure 1: Two DETOP windows containing different debugging contexts. The context is defined by the thread list in the left window part. The center window part shows a listing with active breakpoint definitions and current thread positions. The right window part contains the output of debugging commands like queries of variable values or thread states.

Graphical user interface

DETOP provides a graphical user interface whose main component is a debugger window as shown in Fig. 1. For data parallel applications where all threads execute the same code, a single window is often sufficient and offers the most convenient and comprehensive way for debugging a parallel program. However, in order to support other paradigms of parallel programming, especially functional decomposition where threads execute totally different codes, 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 to debug threads with different codes in applications based on functional decomposition. Multiple windows also allow to use one window for a coarse grain 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 source code positions 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 the following 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 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. When 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 both supports C and Fortran 77.

On the left side, the debugger window shows the list of threads currently associated with that window. 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. The set of threads associated with a debugger window may be changed at any time using a separate window that also allows to view all currently existing threads.

DETOP provides two different forms of command output. Information may either be appended to an output 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.
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.

Dynamic thread creation

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. 1). 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, thread termination and creation, 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. Thus, even a dynamically created thread can be debugged from the very beginning of its execution.

3.2 PATOP

For the optimization of a program it is necessary to look at possibilities to eliminate sources of performance loss. In the parallelization process of a program, the increase of computational effort is inevitable because the parallel execution has to be organized and recalculating a value on a node is sometimes cheaper than transferring it from other nodes. For the optimization these computations are only relevant as far as they can be reduced by using more efficient computation schemes. This does not differ from the optimization of other sequential program parts.

On-line performance data collection

Massively parallel computing systems are nearly exclusively used for a class of applications that consists of long running programs repeating the same execution cycles over and over again. In these applications, only those performance bottlenecks are of interest that appear regularly or continuously and thus have a noticeable impact on the execution time of the program. This allows to operate PATOP in an on-line mode. The on-line mode of operation makes it possible to produce data at any time only for those measurements that are displayed in the user interface. This technique reduces the amount of measurement data to be handled to a fraction of the data that would have to be recorded to operate PATOP in an off-line mode. Additional data reduction is achieved by producing average values according to the resolution specified for the displays. This also leads to a significant reduction of measurement data compared to complete execution traces.

Reasons for performance loss

The main reasons for performance loss are idle processor states. It is also not possible to eliminate them completely because usually every algorithm has inherent sequential parts. The actions that are necessary to eliminate them depend on the circumstances that cause them. If idle processor states are caused by access delays to remote data or by conflicts in accessing remote data they can be reduced by changing the access scheme of the responsible system component or by increasing the processing speed for requests. Such access loss can also be reduced by decreasing the access distance between a system component where the data is stored and the node where it is needed. When idle processor states are caused by an overloaded communication network, they can only be eliminated when considering all communication that is done by the program. Communication has to be coordinated and reduced. Single processors may also become idle due to improperly balanced work. In this case, the performance can only be increased by a improved static load distribution or by methods for dynamic load balancing.

Detecting optimization points

A strategy to detect inefficiently coded sequential program parts is to look at the computation time required for the execution of a program's procedures and to examine the coding and usage of the procedures that

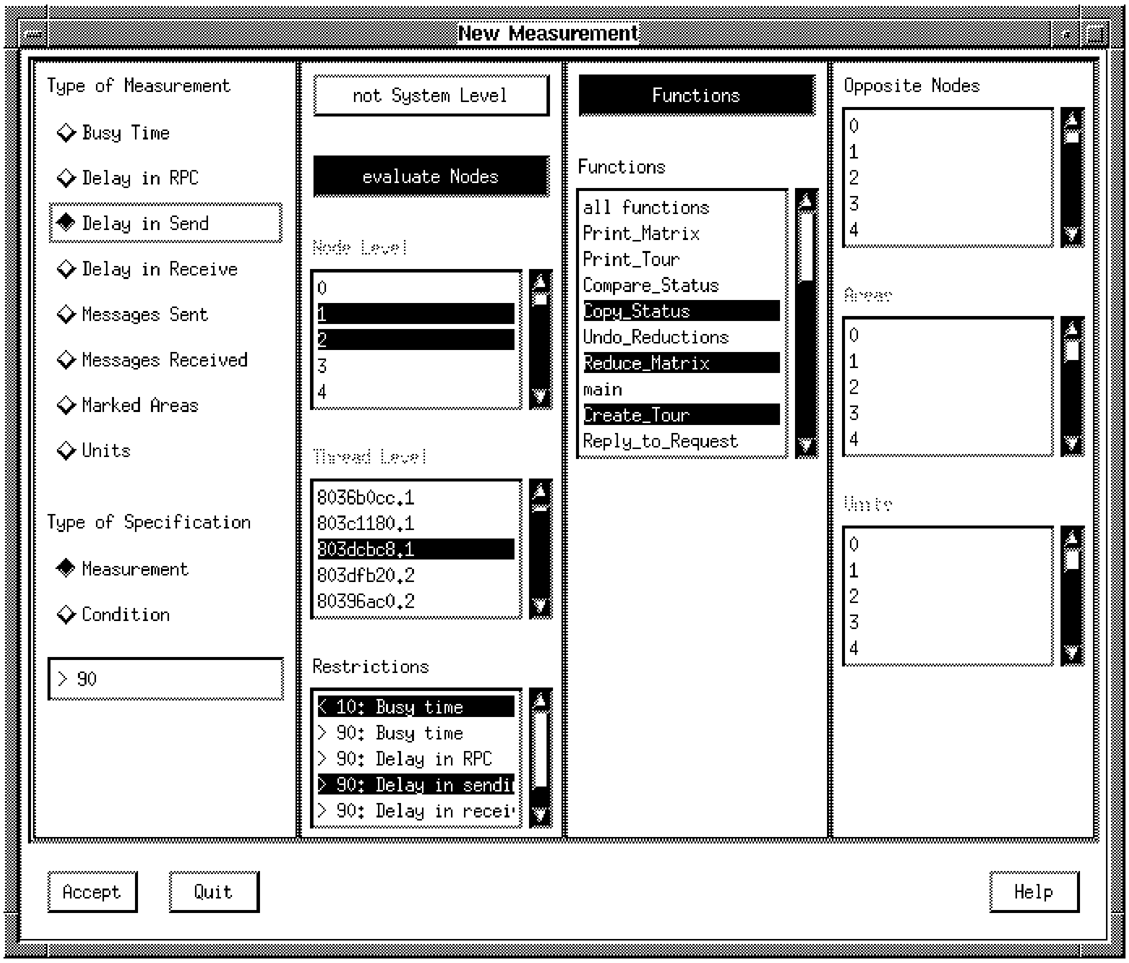

Figure 2: The PATOP measurement definition window. Each measurement is defined as a cross product of the first three columns. The first column is used for the definition of the measurement type (Busy time, communication delay,...) and the purpose of the definition (attribute or complete measurement). The second column defines the system relation (complete system, attributed node set, processor node, thread). The third column defines the program relation (complete program or single function). Use of the fourth column varies with measurement type.

require the largest amount of computation time. This strategy is the same as for sequential programs.

The other categories of performance loss only occur in parallel program execution, and they usually also have greater optimization potential. All these categories of performance loss are caused by idle processor states. Therefore, a measurement of the system utilization shows their existence. For a distinction between the categories of performance loss it has to be investigated where the execution streams are blocked and how much load there is on the communication network. If the communication network is heavily loaded and the execution streams are blocked in different kinds of communication-related system calls, this shows loss due to conflicts in accesses to the communication network. Otherwise, it has to be examined in what kind of system calls execution streams are blocked. If there is a small degree of parallelism in the program, execution streams are blocked in inter-process communication calls. In case the program is delayed by remote accesses, the execution streams will be blocked in remote access calls. Load balancing problems make themselves conspicuous by an uneven distribution of the idle time over the processors. However, the same symptom may also be created by delays in accessing remote data. To distinguish between these alternatives, it is necessary to determine the program parts where processes are blocked, and to consider their semantics.

Figure 2 shows the measurement definition window of PATOP. It supports the measurement of CPU utilization, communication delays, delays in external I/O, data transfer rates, and delays in parts of the program that are defined by user inserted statements. All measurements can be done in relation to the complete program or individual procedures. This gives the possibility to link measurement values to the program parts that cause them.

For the first analysis steps, global performance like overall CPU utilization has to be looked at. PATOP presents global performance figures in a scalable way either as global sums or as node distribution values. The latter give categorical statistics about the distribution of values measured on single nodes over a set of value ranges. It is also possible to present overall CPU utilization as matrix values where each node contributes one value to an element of the matrix. Such matrix values can not be displayed in a way that is completely scalable but displays scale up to quite large numbers of nodes, and users like them because they are less abstract than distribution values.

Attributed measurements

Starting from global system performance information, more detailed levels give the possibility to link aspects of the global behavior to individual system components and then to the program parts that are responsible for them. Based on this knowledge, the optimization of the program can be started. Couch defines the capability of linking categorical information to individual system components and program parts as the subview mapping property [Cou93]. A display has the subview mapping property, if there is a procedure that can map a rectangular subregion of the display back to the subset of the input data used to render it. Important for the scalability is, that the process of linking global performance behavior to individual system components and program parts can be done in a reasonable number of small steps. A step in this context consists of the examination of some performance information and has to provide the information needed for the next step. A step is small, if it can either be done by looking at a scalable display or the number of processing elements that have to be considered is independent of the system size. Through the concept of attributed measurements, distribution values get the subview mapping property. Nodes contributing to a value range in a distribution profile can be identified in a single specification step, and measurements can be focused on them. This is also possible, if the set of nodes that contributes to a value range varies with time. The focusing is done by specifying behavioral attributes as thresholds to results of measurements. An example for a measurement that is focused through a behavioral attribute is:

> The average delay of processes in external I/O operations on all nodes that are more than 50% idle.

The behavioral attribute is evaluated for each measurement interval and only those nodes are considered in the measurement that fulfill the condition of the behavioral attribute. Figure 3 shows an example of such an attributed measurement. It is possible to combine more than one condition to one attribute. In any case, the measurement values can be produced by the monitor in a data parallel scheme as described later.

Displays

To present measurement values, PATOP offers a set of displays that emphasize different aspects of these values. PATOP always shows values averaged over a specified interval of time. This interval, and thus the resolution in time, can be configured individually for each measurement and may be between $50ms$ and the complete program execution time. The graphical presentations offered by PATOP are based on the following display types:

1. Bargraph displays show a single value per measurement. This is either the one most recently recorded or the average of all values recorded so far.

2. Multicurve displays show values for different measurements over a time axis as coordinate on the y-axis.

3. Matrix displays show system related matrix values. The value for each node is coded as a color and displayed at coordinates in the matrix that represent the node.

4. Distribution displays show the distribution of performance values. For a set of value ranges, they indicate the number of nodes that have values within these ranges.

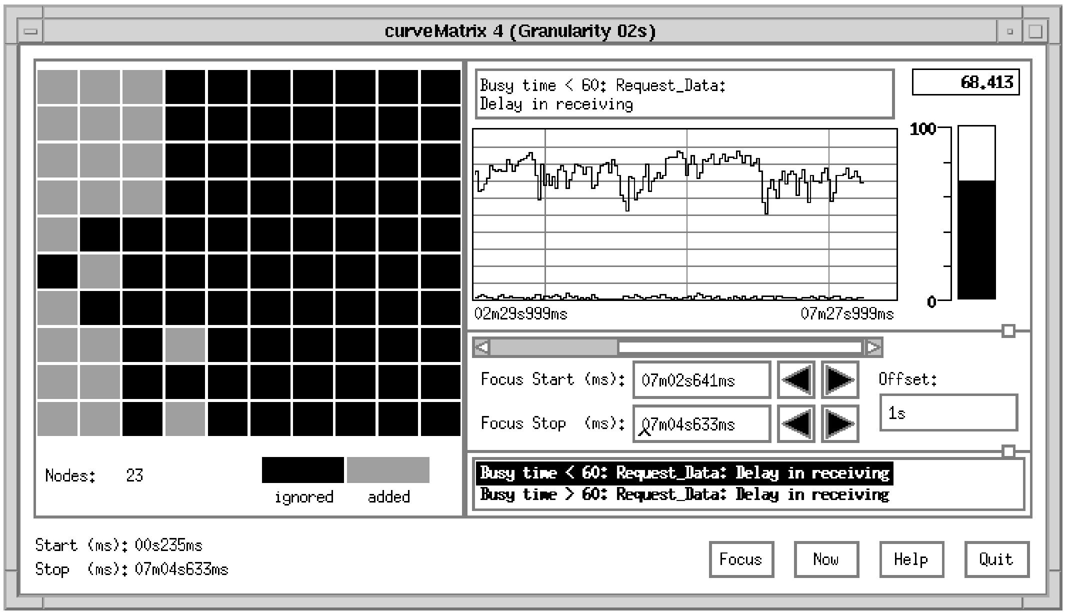

Figure 3: Display showing attributed measurement. The gray boxes in the matrix show the set of processor nodes on which the attribute (Busy time < 60 %) is currently valid. The upper curve shows the average delay of all processes in receive calls within the user defined program routine "Request_Data". The lower curve shows the same measurement for the complementary set of nodes (Busy time > 60 %).

5. The curve-matrix display shows values for attributed measurements. In addition to a multicurve display showing the measurement values over a time axis, it also contains a matrix with the set of nodes that contributed to the last value (see Figure 3). This diagram is not mandatory for displaying attributed measurements, but it is supposed to illustrate their operation.

4 Target Architecture

Before we discuss the implementation of DETOP and PATOP in more detail, we will briefly present the current target architecture and its operating system environment, since some of its features have influenced the tools' design.

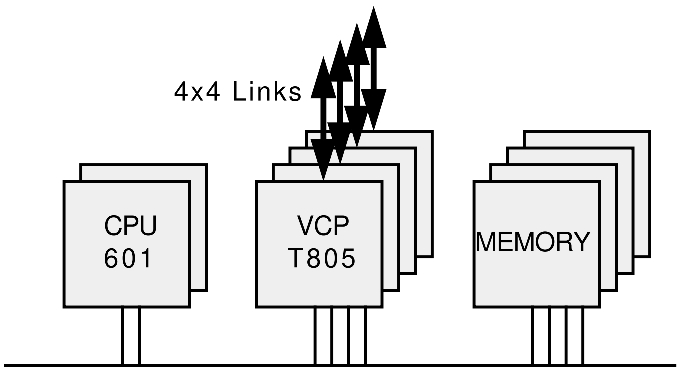

Figure 4: Node Architecture of GC/PowerPlus

DETOP and PATOP have been implemented for the new generation of Parsytec parallel computing systems, based on PowerPC and Transputer processors. They support the PARIX message passing interface that is an integral part of these computers. The Parsytec GC/PowerPlus is a distributed memory parallel system that scales up to 512 nodes. Each node consists of two PowerPC 601 computing processors and four INMOS T805 communication processors. The processors on a node communicate through shared memory. The inter node communication structure is a two dimensional grid where each T805 has one link in each of the four directions. It is possible to connect a parallel file system to the boundary of the grid structure. Host workstations are also connected through Transputer links to the open ends of the grid. The PowerXplorer has a simpler node architecture with one PowerPC 601 and one INMOS T805 per node. This system can be scaled up to 32 nodes.

Both systems run the parallel operating environment PARIX. PARIX is a message passing library that offers inter- and intra-node communication mechanisms that are based on virtual links connecting two computing nodes. In addition, PARIX provides a subset of POSIX system calls. System calls that cannot be handled locally are forwarded to a host workstation or to the parallel file system through an RPC mechanism. PARIX also offers features to dynamically create threads, i.e. lightweight processes running in a common address space on each node, and communication links. It is also possible to load and start additional executables from application threads; this can be done independently on each node. The GC/PowerPlus and PowerXplorer systems can also be programmed using the forthcoming standard programming models MPI and PVM. Both the MPI and PVM library are implemented on top of PARIX.

For thread scheduling and inter-node communication PARIX is based on a Nano Kernel. On the GC/PowerPlus, the Nano Kernel uses a common process queue for scheduling threads on both computation processors. Scheduling only takes 3 to 7 μs . The Nano Kernel also transfers the execution of inter-node communication to the available communication processors. Forwarding of messages within the physical grid structure is then autonomously handled by these processors. To increase the communication bandwidth, links can be configured in a way that a single message passing call is executed by multiple communication processors in parallel. The measured communication bandwidth between a neighboring pair of nodes in the GC/PowerPlus is about 3.3 MBytes/s, the measured setup time is about 160 μs [Prz95].

Communication with processes executing outside the parallel computer, e.g. on its front end, is not directly supported by PARIX. Instead, standard UNIX mechanisms can be used, e.g. TCP/IP sockets, since the corresponding system calls are transparently forwarded to the front end by PARIX. However, as has been measured on a PowerXplorer, the communication bandwidth is only about 370 KBytes/s.

The feature of this target system that had the largest impact on our tool design is the multi-threaded, very dynamic programming model of PARIX. Much effort has been necessary to support this programming model adequately (e.g. thread-safe implementation of breakpoints, timing routines that account for thread scheduling, support for dynamically created threads and dynamically loaded codes).

Also, some restrictions of the GC/PowerPlus and PowerXplorer systems had an impact on the tools' design. The coarse grain architecture of these systems, i.e. a MPP system attached to a host workstation doing all user interactions, is mirrored in the tools' architecture, where there is a monitoring system on the parallel processors and an interactive component on the front end or a remote workstation (see Fig. 5). Since the amount of main memory on each node is comparatively small, we decided to keep the monitors as small as possible. Likewise, since the bandwidth between the nodes and the front-end is much lower than the internal bandwidth, we tried to reduce the amount of data exchanged between monitoring system and the tools' host parts by a distributed data preprocessing. In addition, we introduced an asynchronous evaluation scheme for results to avoid blocking the tools due to communication delays.

However, the techniques implemented for this target system are still extremely useful when the limitations are removed. Even on a machine with larger memory, the monitoring system should be small, since e.g. scientific applications often tend to operate at the memory limit. Also, distributed preprocessing of results is still necessary on a machine with very high external communication bandwidth, since the host otherwise is not able to process and display all the data in an acceptable amount of time. Therefore, we do not expect that the basic realization of the tool-set will change when porting it to more powerful systems.

5 Tool Realization

As already stated before, the tools are realized in a distributed way (see Figure 5). On the nodes of the parallel system there resides a distributed monitoring system. It contains the functions necessary to observe

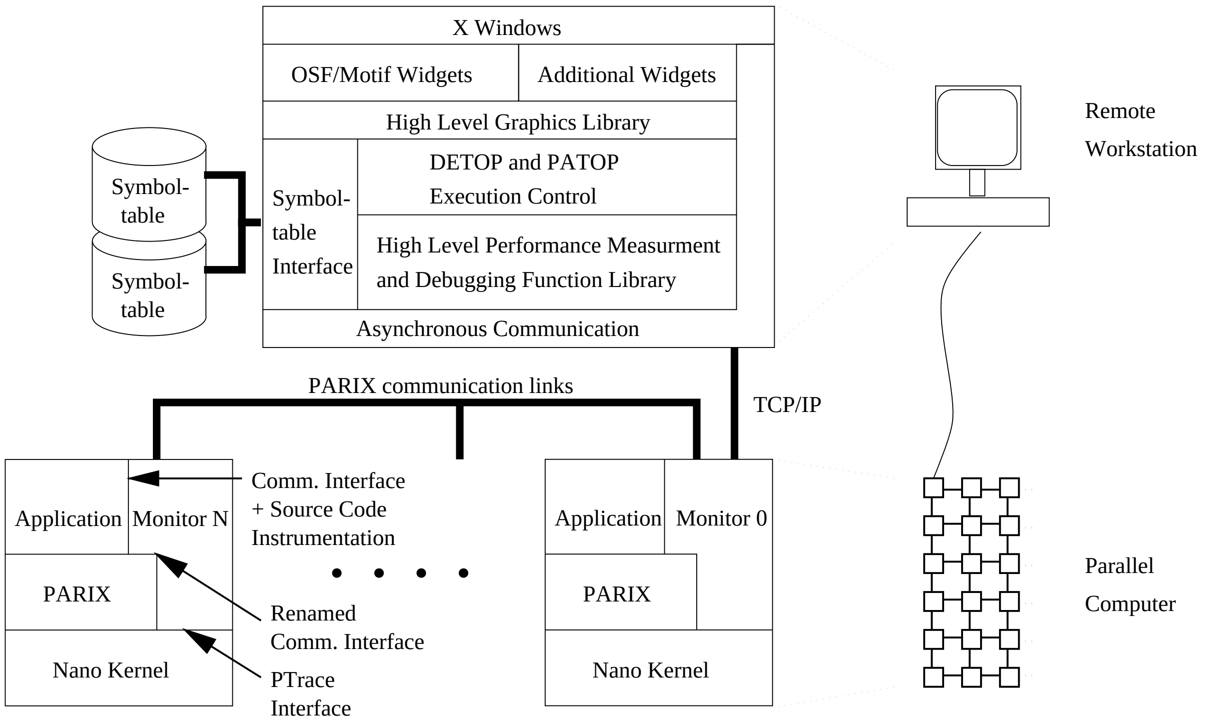

Figure 5: Embedding of the distributed monitor into the target system and structure of the host components of DETOP and PATOP

and control the execution of the parallel application. It is kept as small as possible in order to leave enough memory space for the application. The monitor functions are kept rather low level in order to disturb the execution of the application as little as possible. The host part of the tools provides the user with a sophisticated graphical interface.

5.1 Host Components

Figure 5 shows that the overall structure of both tools is the same. There is one dedicated module for each tool that defines its functions. The other modules are shared by both tools and contain common functions as well as functions only used by one tool.

The graphical user interface is based on the X Window system. It uses the OSF/Motif widget set and some additional widgets for the graphical representation of measurement values in PATOP. On top of the widget interface, a high level graphical programming interface is defined that contains C++ classes for the window types the tools use in their user interfaces.

A set of utility routines translates between the functions as they are offered by the distributed monitoring system and the high level debugging and performance measurement functions that are offered by the graphical user interface. For a translation between source code level and machine code level, symbol information as provided by the symbol table interface is used (see below). For the monitor communication and the asynchronous handling of monitor replies, an asynchronous communication library is used (see section 5.2).

Symbol table management

When a program is compiled for debugging, compilers generate 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. Although 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 given procedure. There is a couple of different symbol table formats

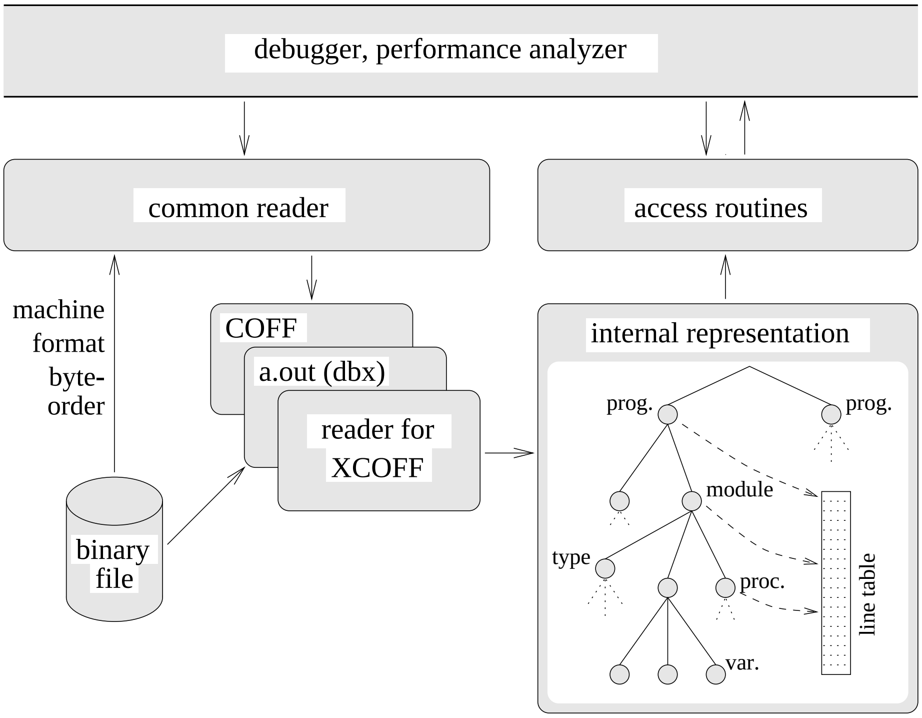

Figure 6: Structure of symbol table management library

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 to support both applications where code is loaded dynamically, i.e. applications consisting of more than one executable, and 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. 6 shows the structure of this symbol table manager. A common reader 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. the list of all procedures displayed in the PATOP measurement definition window.

5.2 Asynchronous Communication

Since massively parallel computers often are located at computing centers, users typically have only remote access to these machines. If the tools had to be executed on the parallel computer's front end, all the graphical output would have to be redirected to the local workstation. Thus, using them remotely would be very annoying due to the network delays. We therefore decided to use TCP/IP sockets for the communication

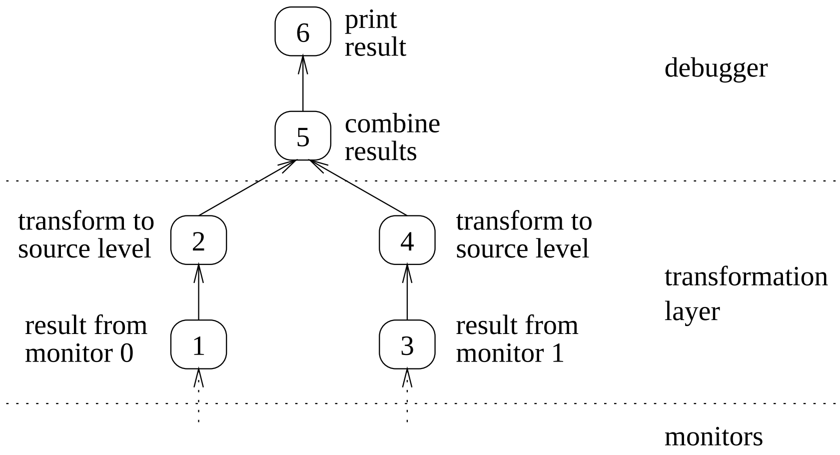

Figure 7: Example callback structure for a debugger request. Results are received from monitor 0 (callback 1) and monitor 1 (callback 3). Transformations for both results are made separately (callbacks 2 and 4). When results from both monitors have been transformed they are combined by callback 5 and printed by callback 6.

between the monitoring system and the tools[1]. The monitor on node 0, which is also the root of the monitoring system's communication tree, acts as a server transforming between TCP/IP and the MPP's internal communication. Therefore, the tools can be started locally on any workstation having network access to the MPP without the need for redirecting a large amount of graphics data[2]. Since the amount of data exchanged between tool and monitor is rather low and communication is asynchronous, usage is still comfortable even over a large distance[3].

Asynchronous communication is also needed for the replies of actions triggered by events that may occur at any time. The tools asynchronously receive monitor replies by defining the communication socket as an additional input source for the X-window system. Thus, the tools can simultaneously wait for user input and results sent by the monitoring system. As stated earlier, this communication scheme is used whenever possible to avoid blocking the graphical interface.

However, asynchronous receipt of messages is not sufficient, since the data sent by the monitoring system must also be processed and displayed in an asynchronous way. This requires some form of callback functions. In the simple form used in the X-window system, the mechanism is not flexible enough, so we have implemented a more powerful scheme using the concept of data-triggered evaluation: Functions within the transformation layer that require communication with the monitoring system will not block until the result is available, but immediately return a *result-id*. A callback function may then be associated with one or more *result-id*s. Adding the callback leads to a new *id* representing the callback's result, so the process can be repeated. In this way, a data flow graph is constructed for each request. Figure 7 shows an example. Each inner node represents a callback, the leaves represent the receipt of a result from the monitoring system. Once a result is received, the graph is evaluated in a data driven manner, i.e. a callback is executed when its input data is available. For each callback different modes can be selected:

- Callbacks may be either dynamic or static, i.e. the graph is either deleted after evaluation or initialized for processing further results. The latter mode is used for the results of actions repeatedly triggered by an event.

- The callback can be executed when all its input parameters are available or whenever one of them is available. Again, the latter mode is used for event processing when there are alternative sources for an event.

[1] For the GC/PowerPlus and PowerXplorer systems, this is in fact the only feasible way of communication between a program on the parallel computer and a program outside of it.

[2] A user-defined key ensures that access to the monitoring system is secure.

[3] At the PTools General Meeting in 1995, we actually have demonstrated a debugging session, where DETOP was executed in California, while the parallel program was running in Germany; it worked very well!

- Finally, a callback may also be synchronous, i.e. adding the callback will block the caller until the callback has been executed, just like a usual function call. Thus, the same interface can be used for both asynchronous and synchronous evaluation.

The callback mechanism allows to combine a strictly layered software design with asynchronous evaluation. Therefore, it is not only used in the host layers, but also within the monitoring system, e.g. to evaluate and combine performance data.

5.3 Distributed Monitoring System

The basic part of both programming tools is the monitoring system. Its task is to provide the tools with adequate information. Since the sort of data a performance analyzer needs differs from the demands of a debugger, the question arises whether to use a single monitoring system for both tools or one for each purpose. The bodies of different monitors would be the same and only some data retrieving functions would differ. The answer is therefore to use one universal and flexible monitoring system. In case of parallel systems the monitors are distributed over the corresponding computing nodes. But to have one monitor on each node does not necessarily mean to have a distributed monitor. The monitors need to interact in order to form a common monitoring system. Communication is necessary for several reasons. First, there is a need for global commands that affect all involved processing nodes. Since the connection between the host and the parallel machine is usually much slower than the inter-node connection, it is more efficient to send only one message to a dedicated node, where the message is replicated and sent to all nodes using a simple spanning tree for distribution. Besides this kind of broadcast, a multicast communication between monitors is provided. The same way commands are distributed over a set of nodes global data is gathered. 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.

The monitoring system is, of course, highly dependent on the underlying operating system. Where the tools operate at source level, the monitor deals with machine addresses. To get necessary information about the processes and to be able to modify their data, the operating system must provide a corresponding interface. In case of this implementation, information offered by PARIX and the lower level Nano Kernel are used. The Nano Kernel provides an adapted PTrace interface, which is presented below. The monitoring system itself is running at PARIX level, resulting in the structure shown in Fig. 5.

The PTrace interface of the Nano Kernel is similar to the standard Unix PTrace. However, it is modified according to the dynamic and multi-threaded process model. 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 signaled 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.

For debugging, the main instrumentation technique used by the monitor is dynamic object code instrumentation via the PTrace interface. For the acquisition of performance data, additional techniques are used: Process related execution statistics, like a process' CPU time, are collected within the Nano Kernel during process scheduling and can be accessed via an extension of the PTrace interface. Measurements concerning communication are obtained through an instrumentation of the according routines in the PARIX library. Since common operating system services are carried out on a host system and initiated by sending an appropriate request, the same instrumentation can also be used to measure the time spent in these services. They can be distinguished from ordinary communications by their partner. For those measurements that acquire data about the time spent in user functions, sampling is used. The sampling interval length is $10ms$. During each sample, the current instruction pointer of the interrupted process is compared with the code boundaries of watched functions. Then the stack is searched back to its beginning, comparing the return addresses within the stack frames with the code boundaries of the watched functions. Apart from this sampling technique, the user has direct access to the monitoring system through source code instrumentation. He can mark parts of the application code by inserting monitor calls at the beginning and at the end of the area he is interested in. Additionally, it is possible to visualize application data by calling another monitor function.

The monitoring interface

To achieve the goal of a universal monitoring system, a programmable monitoring interface is used. Our approach separates the syntax from the semantics of the command interface [Tre94]. This means that there is a common, well defined syntax allowing arbitrary commands that provide the semantics. In that way it is possible to offer a complete and orthogonal interface for both tools. To take advantage of these properties, the commands can be composed in different ways, so the total number of basic 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 positions where they are currently stopped, it is possible to combine a command called view with a command called inspect. view determines the identifiers of all threads and pipes its output to inspect, that locates the threads' current program counter value. Using our syntax, this request looks like

```
{view['T']{inspect[?]}}
```

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. To get the same effect 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 a monitoring system, i.e. to avoid disturbance and to hold the monitor as small as possible.

In summary, the programmable monitoring interface results in simple adaption to new demands and a powerful and flexible command set using only a small number of basic commands.

Event–action scheme

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 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 provides a programmable event–action management allowing each tool, e.g. the debugger, to choose the event handlers and the actions it needs for its purposes.

If a tool requests the monitoring system to send a notification when an event occurs, the monitoring system autonomously performs the filtering and preprocessing of the information. Each monitor itself decides whether the current event is actually important. Furthermore, it reduces the amount of information sent to the tool as far as possible.

Since some events are asynchronous in nature, the interface between the monitoring system and the tool is also asynchronous, i.e. if a tool defines an event and associated actions, the monitoring system will immediately return only an acknowledge. The action's results are then sent to the tool whenever the event occurs. This is very important when considering dynamic process models where lightweight processes (threads) or heavy processes (new programs) can be created or terminated at any time. 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.

Global events

A specific problem of parallel programs concerns global events [HW88] i.e. an event definition refers to more than one node. Global events can only be observed by the whole monitoring system, because they occur 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 [Lam78], e.g. vector time-stamps [Fid88]. But for on-line debugging we prefer a more pragmatic way that is more efficient and, as practical experience shows, also sufficient. We do not regard causal relationships but simply try to inform the corresponding nodes as fast as possible. This is realized by the event–action management sending a command to the corresponding node. That means, no special mechanisms are necessary to deal with this inter–monitor communication. Dedicated action functions is all we need.

A typical example of a global event is composed of some communication sub–events on different nodes. This might be formulated like "trigger an event if send operations occurred on nodes one, three and five **and** if a receive operation on node four finished". In this example, the observation is performed on each node and the necessary information triggered by sub–events is exchanged using the event–action mechanism.

Performance data collection

For PATOP it is important to gather the enormous amount of data in a clever manner even on a hundreds of nodes. To achieve this, all measurement values are combined in a data parallel fashion. As already mentioned a simple spanning tree is used for this approach. Each node computes its own position in this tree by using information from the operating system about the current grid of nodes. Therefore, each monitor knows on one hand for how many children's answers it has to wait. On the other hand it is able to determine the parent node. The parsing unit is extended to handle these result messages. That means there is again a predefined syntax of these messages, but the contents depends on the measurement. The gathering process starts at the leaves of the tree and the last node, the root, sends the final result to the tool. The intermediate nodes process the received values depending on the sort of measurement and send the result to the parent (see Figure 8). In all cases, the communication is always reduced to one value for each node. This value is a fixed size structure for a global sum, a matrix, a distribution, a value resulting from an attributed measurement. From the root node the data is transferred to the performance visualization component of PATOP. No further postprocessing has to be done outside of the node monitors, and still the additional overhead produced by the processing of values within the monitoring system is very small.

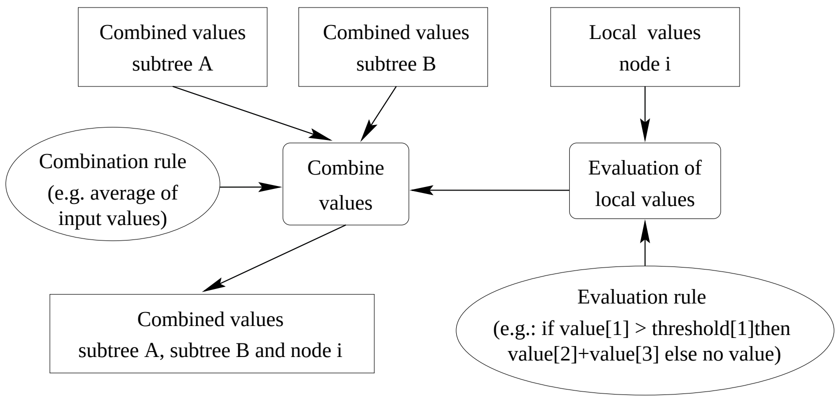

Figure 8: Distributed construction of measurement values. The example shows the construction of an attributed measurement: reading of local measurement values, evaluation of local measurements according to the evaluation rule, combination of local values with values received from child nodes in the spanning tree, and transfer of the result value to the parent node in the spanning tree.

6 Conclusion and Future Work

The article introduced a tool environment for debugging and tuning large scale parallel applications that has been implemented for PowerPC based MPP systems. We have presented the basic concepts, user interface and implementation of the tool-set and also gave examples, how these tools can effectively be used to optimize and debug applications whose structures are typical for those appearing in scientific high performance computing.

The tool-set is available since December 1994 on a commercial basis. Since this date, a large number of installations have been made (roughly about a hundred). The user response and our own practical experience show that DETOP and PATOP are widely accepted by users and offer valuable help in the development of correct and efficient parallel software. Further enhancements will include the following topics:

- advanced statistical evaluation of performance values,
- automatic or semi-automatic detection and localization of performance bottlenecks,
- comfortable debugging of communication by providing special breakpoints,
- visualization of regular and irregular data structures and procedure call trees,
- support for parallel programs written in C++.

In addition, we will adapt the tool environment to other programming models, including High Performance Fortran (HPF) and PVM. The HPF version will especially address the problem of mapping runtime information back to the source code. The PVM implementation will be done both for networks of workstations (being part of THE TOOL-SET for PVM [LWB+95]) and a special version of PVM running on the GC/PowerPlus. The main issues to be addressed in this work are heterogeneity and the perturbation by multi-user operation.

References

[BHL90] T. Bemmerl, O. Hansen, and T. Ludwig. PATOP for Performance Tuning of Parallel Programs. In H. Burkhart, editor, *Proceedings of the CONPAR 90 - VAPP IV Conference*, volume 457 of *Lecture Notes in Computer Science*, pages 840 – 851, Zürich, Schweiz, September 1990. Springer-Verlag.

[BW94] 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.

[Cou93] A. L. Couch. Categories and Context in Scalable Execution Visualization. *Journal of Parallel and Distributed Processing*, 18:129 – 146, June 1993.

[Fid88] 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, February 1988.

[Gai86] J. Gait. A Probe Effect in Concurrent Programs. *Software–Practice and Experience*, 16(3):225–233, March 1986.

[Han94] 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, München, April 1994. Springer-Verlag.

[HE91] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

[HM93] J.K. Hollingsworth and B.P. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[HW88] 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.

[Int91] Intel Corporation. *iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual*, April 1991.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LWB+95] T. Ludwig, R. Wismüller, R. Borgeest, S. Lamberts, C. Röder, G. Stellner, and A. Bode. The Tool-set – An Integrated Tool Environment for PVM. In *Proceedings of EuorPVM'95 Short Papers*, Lyon, France, September 1995. Ecole Normale Supérieure de Lyon. Technical Report 95-02.

[MB93] 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.

[MCH+90] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206 – 217, April 1990.

[Par92] ParaSoft Corporation, 2500 E. Foothill Blvd., Pasadena CA 91107. *EXPRESS - Building Parallel and Distributed Programs, Version 3.2*, 1992.

[PC94] 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.

[Prz95] B. Przywara. Standard Benchmarks on Parsytec GC/PowerPlus Parallel Computer. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 18–34. IOS Press, Amsterdam, May 1995.

[RAA+93] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith. The Paragon Performance Monitoring Environment. In *Proceedings of Supercomputing '93*, pages 850–859, Portland, Or., November 1993. IEEE.

[SAB+94] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. A Scalable Debugger for Massivly Parallel Message-Passing Programs. In *Proceedings of SHPCC'94, Scalable High Performance Computing Conference*, pages 825–832, Knoxville, Tennessee, May 1994.

[SFG94] P. E. Sorel, M. G. Fernandez, and S. Ghosh. A Dynamic Debugger for Asynchronous Distributed Algorithms. *IEEE Software*, (1):69–76, January 1994.

[Tre94] T. Treml. *Monitoring Paralleler Programme.* PhD thesis, Technische Universität München, 1994.