State Based Visualization of PVM Applications*

Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik, Technische Universität München, D-80290 München, Germany
Tel.: +49-89-289-28164, Fax: +49-89-289-28232
email: wismuell@informatik.tu-muenchen.de

Abstract. Understanding the dynamic behavior of parallel programs is a critical issue both for debugging and for optimization. A visualization tool displaying an animated sequence of the global states the program runs through offers valuable support for this process. The paper presents the features and the implementation of VISTOP, a state based visualizer for PVM applications. It supports program flow visualization based on various views and uses an event ordering algorithm to ensure consistent visualization without requiring a global clock.

1 Introduction

The complex interaction of a large number of components makes programming parallel and distributed systems a complicated and error-prone task. The dynamics supported by programming libraries like PVM can make this situation even worse, because the execution of programs where tasks (or other objects) are created and deleted during runtime can be hard to comprehend. Breakpoint-based debuggers, which are also available for parallel programs now (e.g. P2D2 [4]) offer programmers valuable help in understanding program behavior and in locating bugs. However, they basically allow only the inspection of the local states of single processes. But especially for finding bugs related to communication or synchronization, the global program state is of much more importance, e.g. which tasks are currently waiting for messages, which are sending, which groups do exist and what tasks do they contain? Although debuggers can somehow answer these questions, getting the answer is a rather complex job for the user. In addition, debuggers do not provide a coarse grain overview on a program's execution, e.g. its communication structure.

A solution to these problems is to complement debuggers with a visualization system that shows consistent global states of an application and animates state changes. Such a tool offers both a global overview and – by recording relevant information – also an animation of the program's coarse grain execution flow.

In the following, we present the concepts and some implementation issues of the visualization system VISTOP that animates communication and synchronization as well as dynamic creation, deletion, and group membership of tasks. The visualizer is an integral part of a larger environment called THE TOOL-SET for PVM [12], which among others will also include a parallel debugger.

* Partly funded by the German Science Foundation, Contract: SFB 342, TP A1

2 Visualizing the Execution of PVM Programs

The idea of providing tools visualizing the communication and synchronization behavior of parallel programs is not new. For PVM, a couple of such tools already do exist. Some examples are XPVM [5, 2], ParaGraph [7, 6], HeNCE [1], and Trapper [14]. However, these tools do not adequately address several problems, whose solution has been a primary design goal of VISTOP.

First, most of these systems only display an event stream, usually as a space-time diagram. From such a representation, it is, however, difficult to get information on a program's global state, e.g. the tasks blocked in receive requests, members of groups, contents of the tasks' receive queues, or information on barriers. VISTOP computes an internal model of a program's global state from the stream of events by including knowledge about the semantics of PVM. For instance, the members of a group are computed from "joingroup" and "leavegroup" events. Therefore, VISTOP can display global states instead of events. The model can also detect certain errors, e.g. if a message is unpacked incorrectly.

A second issue is consistency of visualization. Event visualization tools usually display events in the order in which they are received by the tool. However, due to local buffering and network delays, this strategy can lead to a visualization violating causality. VISTOP addresses this problem by sorting the received events in such a way that causal relationships are satisfied and consistency with the modeled state is guaranteed.

Finally, existing visualizers offer only limited scalability. Space-time diagrams use one line for each PVM task; some tools do not even allow these lines to be reordered. VISTOP allows to freely select the objects to be visualized and to rearrange them in the display area. It also supports a hierarchical visualization at different levels of abstraction by allowing tasks to be grouped together into a hierarchy of container objects according to the application's logical structure.

VISTOP visualizes a program's execution as a sequence of snapshots where each snapshot shows different views of the program's global state. A control panel allows forward or backward stepping through the sequence of snapshots, and a continuous play mode with selectable speed. The transitions between different snapshots are animated in order to accentuate the state change. Currently, the visualizer provides three different views of program state:

1. The **concurrency view** displays a dynamic, hierarchical task graph showing communication and synchronization between tasks. The nodes of this graph represent objects of the programming model (tasks, barriers) or container objects (subgraphs of the program). Arcs denote interaction between objects. The left side of Fig. 1 shows a screen dump of this view that is described in more detail below.
2. The **object creation view** displays the dynamic creation graph of tasks spawned and terminated during runtime. It consists of trees, whose root nodes are the initial tasks spawned by the PVM console. Other nodes represent tasks spawned during run-time; the parent entry defines the spawning task. An example of this view is shown in the upper right part of Fig. 1.
3. The **system view** shows the distribution of tasks over the different hosts. The display is organized as a matrix where each column contains all objects located on one host.

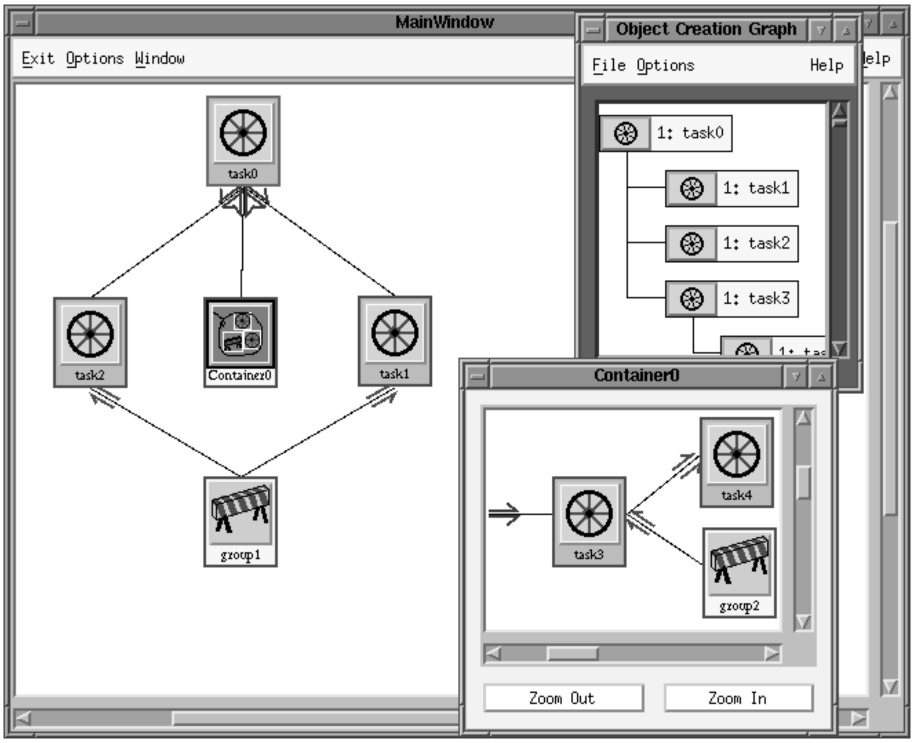

Fig. 1. Concurrency view and object creation view

We will now present the visualizer's concurrency view in more detail. The visualization patterns used in this view are depicted in Fig. 2. When visualizing a snapshot of a program's execution, each existing PVM task and each active barrier is represented by an icon, which may be slightly modified in color or shape to display some of the object's local state information (e.g. timeout or error conditions). Interactions are visualized as lines drawn between the objects involved. Additional arrow heads in different shapes and colors on either side of the line specify the kind of interaction: A green (filled) arrow head denotes an interaction that does not block the active partner, while a red (hollow) arrow head denotes a blocking one. The direction of the arrow head indicates the direction of the (requested) information transfer, while its position marks the active partner. In the middle left of Fig. 2, **task1** is sending to **task0**, since there is a filled arrow head attached to **task1** and directed towards **task0**. Broadcasts and multicasts are visualized in an intuitive way by showing that a task sends to multiple tasks at the same time (see **task0** in Fig. 1). The lower left of Fig. 2 shows the visualization pattern when **task4** waits for a message from **task3**. The same pattern is also used for barrier synchronization. In this case, the icon of **task3** is replaced by a barrier symbol, which is created when the first task in a group calls **pvm_barrier()** and is destroyed again when the last task leaves this call.

In addition to task and barrier objects, users can create container objects to realize a

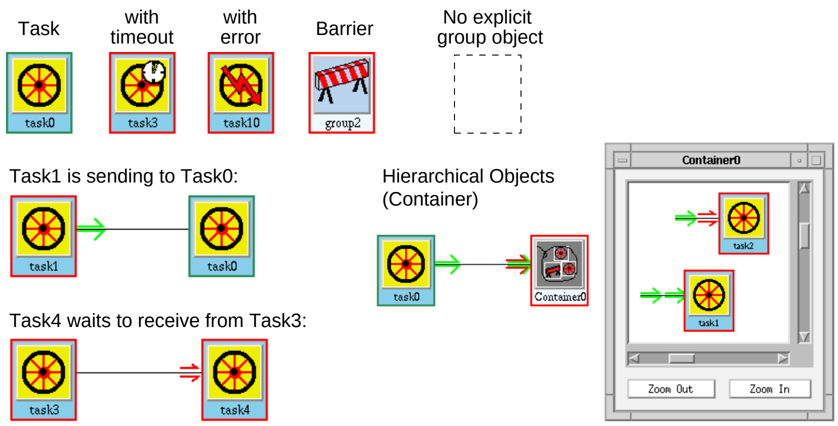

Fig. 2. Visualization patterns in the concurrency view

hierarchical visualization. A container consists of an icon and an associated concurrency window showing what is happening inside it. Objects can be moved into the container by simply dragging icons into the container window. The container's icon then represents the sum of all objects located in the container itself or a nested one. Interactions between a container and other objects are combined by overlaying the corresponding arrow heads, providing a comprehensive summary. Details about the individual interactions can be derived by inspecting the container's concurrency window. In the lower right of Fig. 2, **task0** broadcasts to **task1** and **task2**, both in the container, while **task1** receives a message from **task0**. **task2** didn't start receiving yet and is still waiting for a message from **task0**. By closing and opening concurrency windows associated with containers, the complexity of the visualization can be controlled. If a window is closed, events totally inside the container will be invisible and will therefore also be ignored when determining the next snapshot to be visualized.

The concurrency view does not include explicit objects representing groups. In principle, group objects could have been realized as containers, which combine all tasks in a group. However, containers in VISTOP form a strict hierarchy of tasks, while in PVM tasks can belong to multiple groups. Thus we decided to provide an additional view, called group manager. It displays a list of all groups existing at the current snapshot, where a single group, a union, or an intersection of groups can be selected from. All objects that do not belong to the specified group(s) will then loose parts of their coloring, thus allowing to quickly determine group memberships.

3 Implementation Issues

3.1 Structure of VISTOP

VISTOP is implemented as a hierarchy of three layers: A **data acquisition layer** responsible for gathering events, a **modeling layer** computing consistent global states from this event stream, and a **visualization layer**, which finally displays different views of the computed global state. In the following, we will provide some more information on the first two layers and the issue of sorting events to achieve a consistent visualization.

3.2 Data acquisition

The task of this layer is to monitor relevant events in an executing application, to store these events, and to forward them to the modeling layer in the correct order. In the final release of VISTOP, this layer will make use of an OMIS compliant on-line monitoring system. OMIS is an emerging On-line Monitoring Interface Specification defined in a cooperation between our laboratory and Emory University [13, 11]. The monitoring system detects the relevant events in each PVM task, buffers these events in local trace buffers and forwards the buffers to the tool either periodically or upon request. It also provides mechanisms for other tools, e.g. a parallel debugger, to attach to the application concurrently to the visualizer. In such a case, services of the monitoring system can inform the visualizer on relevant modifications done by other tools. Thus, the visualizer can instruct the monitoring system to flush its trace buffers when the application is stopped by a debugger, and can correctly display its state at the breakpoint. Furthermore, the on-line monitoring system offers services to obtain state information of an application. This allows the visualizer to be started at some breakpoint, which requires to initialize its state model with the application's current state.

However, as the monitoring system is not fully implemented yet, the data acquisition layer currently works with the trace facility integrated into the PVM library since PVM 3.3. It allows to obtain a trace of PVM library calls usually containing one event entry for the call and its input parameters and one for the return and the result parameters. For portability reasons, we do not receive the traced events directly from the PVM system, but read the files written by XPVM, which use the standardized SDDF trace format. However, the traces do not fully support the needs of state oriented visualization tools.

There are currently two major limitations we had to overcome. First, the event indicating the execution of a broadcast only specifies the group where the broadcast is sent to. But for a correct visualization it is important to know to which tasks the message has actually been sent. We therefore have slightly modified the PVM library, so that this information is included in the event trace.

Another limitation is that the PVM tracing mechanism does not record an event when a message is inserted into a task's receive queue. For the modeling of global states, this has three consequences:

1. The VISTOP animation is designed to show whether a task blocks in a receive call due to an empty message queue. Without the event, this is not possible, since we do not know the point in time, when a message is put into the queue. As a current solution, we assume that messages are inserted as soon as the sender finishes the send operation.
2. The receive queue of a task cannot be computed accurately. All the model can provide is a *set* of receivable messages. It cannot determine the order in which messages are stored in the queue[2].

[2] Modeling the queue of receivable messages of a task is also complicated by PVM's direct message routing scheme. However, discussing this topic is beyond the scope of this paper.

3. It is impossible to visualize receive operations in the desired way, if no sender is specified in **pvm_recv()**, i.e. the wild-card is used. Since the receive queue cannot be modeled properly, it is then unknown, which task is the sender of the message being received. We currently overcome this problem by delaying the visualization of receive operations until we have seen the event generated by the end of the receive call. This event always contains the sender task.

In addition to event gathering, the data acquisition layer must also ensure that the event stream is forwarded to the modeling layer in an order consistent with causality and the modeled state. Since this topic touches both layers, we delay its discussion until we have presented the modeling layer in more detail.

3.3 Modeling Layer

The modeling layer computes information on the global state of the monitored application from the sequence of events received from the data acquisition layer. It is implemented as an extended state machine, which computes a new state s' from the current state s when given an event e and a visualization direction d: $s' = f(s, e, d)$. The state transition function f obeys the rules $f(f(s, e, \mathrm{forward}), e, \mathrm{backward}) = s$ and $f(f(s, e, \mathrm{backward}), e, \mathrm{forward}) = s$, which ensure that forward and backward animation will be consistent to each other. Being able to incrementally compute states in both directions frees the modeling layer from storing the whole history of snapshots and greatly reduces memory consumption.

The application's global state is modeled by a collection of C++ objects, representing hosts, tasks, groups, and barriers. Host objects only contain the host's name; task objects store a task's id, its name, a reference to the host, task state (e.g. sending, computing, ...), its parent's id, a list of groups where the task is member of, and lists containing information on currently receivable messages and the messages that have been sent and received since the program has been started. Message information includes the sender task, the tag, length, and the sequence of data types packed into the message. The messages' contents are, however, not stored. Group objects provide information on the group's name and the list of member tasks. Barrier objects finally contain a reference to the proper group, the number of tasks that must synchronize, and the list of tasks currently waiting at the barrier. All information contained in these objects is available to the user in a detailed view which is popped up by clicking at the object's graphical representation in a visualization window. Interactions between objects are internally represented by a fifth kind of objects containing the interaction type (e.g. send, broadcast, barrier wait) and references to the interacting partners.

In addition to computing state information and passing it to the visualization layer, the modeling layer also performs extensive error checking. For instance, it can check the correct usage of barriers, examines whether messages are unpacked correctly, and can provide detailed information on why a PVM library call has failed.

3.4 Consistent Ordering of Events

The modeling layer computes state information from the sequence of events passed from the data acquisition layer. For a correct visualization it is therefore important to pass the events in a correct order. However, the data acquisition layer may receive events generated on different hosts in an incorrect order due to event buffering done by the hosts or due to network delays. It is impossible to order these events correctly by using their time stamps, since their resolution is usually not good enough, and clocks on different hosts cannot be synchronized with the required accuracy. However, the order of events must obey the causality (or "happened before") relation [10], i.e. an event e_1 must be prior to an event e_2 if there is a causal dependency from e_1 to e_2 (written as $e_1 \rightarrow e_2$). As an example, consider e_1 to be the sending of a message and e_2 to be the receipt of this message on a different host. Even if the data acquisition layer receives e_2 before e_1 due to buffer or network delays, it must forward e_1 to the modeling layer first, because a message cannot be received before it has been sent. In the current implementation, we store incoming events in different buffers for each host used by the application, assuming that the monitoring system ensures correct event ordering within a single host. Whenever the next event is requested by the modeling layer, we have to determine the buffer from which to read that event, thus incrementally merging the local buffers into a consistent global sequence.

The problem of merging local event traces to a global one according to causal relationships is well known; there are also several solutions yet (e.g. [8, 9]). However, with a state based visualization approach it turns out that just considering causal dependencies between events is not sufficient for achieving a consistent visualization. As an example, consider the situation depicted in the space-time diagram in Fig. 3. Note that there are no causal dependencies (in the sense of "happened before") between events in different tasks. However, it is obviously incorrect to order these events as e_1, e_2, e_3, e_4, because the broadcast actually sent the message to task t_1, but with this sequence t_1 would not be in the group when the broadcast is executed. In contrast, the orders e_2, e_1, e_3, e_4 and e_1, e_3, e_2, e_4 are both correct.

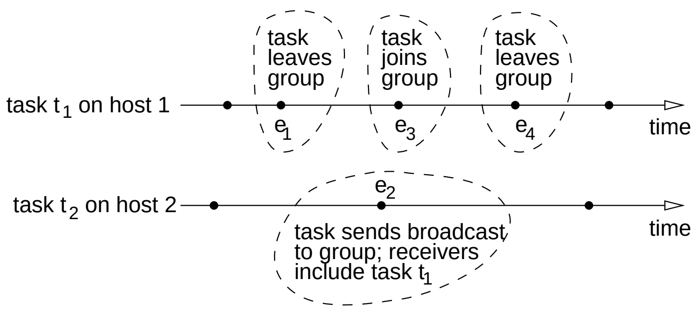

Fig. 3. Example illustrating consistency constraints between events and modeled state

This implies that in addition to dependencies between events, we have constraints imposed by the current state in the model when selecting the next event to process. Therefore, we use an extended sorting algorithm that works as follows: The input are n event buffers, one for each host. We read and remove the next event e_i to be processed from buffer i, only if the following conditions are met:

1. there is no event e_j in an other buffer j with $e_j \rightarrow e_i$ (causality)
2. e_i can happen in the currently modeled state of the application (consistency)

By construction, this strategy will never result in an inconsistent visualization, however, it can lead to deadlock or starvation, i.e. cases where no further events can be selected from one or more event buffers. In our example, according to this strategy we could select three events in the sequence e_1, e_3, e_4. If t_1 never joins the group again, we can not select event e_2 anymore and therefore also no succeeding event in the buffer of t_2. We therefore use a heuristics guiding the selection based on the following considerations:

1. If there is an event e_i that cannot be processed due to condition 2, the modeled state differs from the state where e_i actually occurred in the program execution.
2. In such a situation, we favor events that make this difference smaller.
3. Events that increase the difference are processed only if there are no other selectable events. When considering such an event (say e_k), we perform a look-ahead search in the event-buffers to make sure that the "negative effect" of e_k on the modeled state will be undone by a later event in the buffer. We process e_k only if this search is successful.
4. If there are two events e_i and e_j that both could be processed, but e_j would change the state in a way that inhibits subsequent processing of e_i due to condition 2, e_i is processed first.

The heuristics ensures that the sorting algorithm will never select events that would result in starvation or deadlock. In our example, it leads to the consistent event sequence e_2, e_1, e_3, e_4. In cases where the heuristics leaves multiple candidates to choose from, which must be concurrent events for which no uniquely defined ordering exists, we use the events' time stamps for selecting the next one being processed. If the events' timely separation is greater than the clock skew between different hosts, this ensures that the events are displayed in the order in which they have been monitored.

4 Conclusion and Future Work

VISTOP is a state based visualization system that offers a consistent global view on the execution of PVM applications. Experience with a previous version of this tool [3] for an other message passing library has proved that state based visualization offers valuable help in understanding the dynamic behavior of parallel programs, debugging message passing and synchronization errors, and even for locating performance bottlenecks.

VISTOP is a part of THE TOOL-SET for PVM [12], which also will include other tools, such as a parallel file system, checkpointing and load balancing facilities, a parallel debugger, a performance analyzer and a deterministic execution controller. Future work on the visualizer includes extensions for the support of THE TOOL-SET's automatic components, i.e. an animation of I/O operations using the parallel file system, and task migrations performed by the load balancer. In addition, we will support a tightly integrated usage of VISTOP with the debugger, the deterministic execution controller and the checkpointer. Using these tools together will allow to start the debugger on any state snapshot selected in the visualizer. This will be achieved by automatically restarting the application from the last available checkpoint prior to the selected snapshot and by using the deterministic execution controller to ensure that the selected state is reached even in the presence of nondeterminism due to message races. In this way debugging large programs with high execution times will be efficiently supported.

References

1. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and K. Moore. HeNCE: A Heterogeneous Network Computing Environment. Technical report UT-CS-93-205, Univ. of Tennessee, Computer Science Department, Aug. 1993.
2. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Recent Enhancements to PVM. *Intl. J. of Supercomputing Applications and High Performance Computing*, 1995.
3. A. Bode and P. Braun. Monitoring and Visualization in TOPSYS. In *Proc. Workshop on Monitoring and Visualization of Parallel Processing Systems*, pages 97 – 118, Moravany nad Váhom, CSFR, Oct. 1992.
4. D. Cheng and R. Hood. A Portable Debugger for Parallel and Distributed Programs. In *Proc. Supercomputing'94*, pages 723–732, Washington D.C., Nov. 1994. IEEE.
5. A. Geist, J. Kohl, and P. Papadopoulos. Visualization, Debugging, and Performance in PVM. In *Proc. Visualization and Debugging Workshop*, Oct. 1994.
6. M. T. Heath. Recent Developments and Case Studies in Performance Visualization using ParaGraph. In *Proc. Workshop on Monitoring and Visualization of Parallel Processing Systems*, pages 175 – 200, Moravany nad Váhom, CSFR, Oct. 1992.
7. M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5), Sept. 1991.
8. D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Analyzing Traces with Anonymous Synchronization. In *Proc. Intl. Conf. on Parallel Processing*, pages 70–77, Aug. 1990.
9. D. Kimelman and D. Zernik. On-the-Fly Topological Sort – A Basis for Interactive Debugging and Live Visualization of Parallel Programs. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 12–20, San Diego, CA, May 1993.
10. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.
11. T. Ludwig, M. Oberhuber, and R. Wismüller. An Open Monitoring System for Parallel and Distributed Programs. To appear in Proceedings of EuroPar'96, Lyon, France, Aug. 1996.
12. 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 *Proc. EuroPVM'95 Short Papers*, Lyon, France, Sept. 1995. ENS Lyon. Technical Report 95-02. `http://wwwbode.informatik.tu-muenchen.de/~wismuell/publications/europvm95.ps.gz`.
13. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS – On-line Monitoring Interface Specification. Technical Report TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Feb. 1996. `http://wwwbode.informatik.tu-muenchen.de/~omis/HTML/OMIS/Version-1.0/version-1.0.ps.gz`.
14. C. Scheidler and L. Schäfers. TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications. In *Proc. PARLE'93, Parallel Architectures and Languages Europe*, volume 694 of *LNCS*, pages 403–413, München, June 1993.