Visualization of Parallel Program Execution *

Peter Braun[1] and Roland Wismüller[2]

[1] Siemens AG, ZFE T SN6,
Otto-Hahn-Ring 6, 81739 München

[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik, TU München, 80290 München

Abstract

The paper describes a tool for debugging parallel programs by visualization and animation of their execution behavior. The visualization and animation tool VISTOP (VISualization TOol for Parallel Systems) has originally been developed for a programming library called MMK in a PhD-thesis [Bra94] as part of a tool environment for programming distributed memory multiprocessors. VISTOP supports the interactive on-line visualization of message passing programs based on various views, in particular, a process graph based concurrency view for detecting synchronization and communication bugs.

The paper presents the features of this original version of VISTOP, the process of porting the tool to the PVM programming model, the new features of the PVM version and some of the problems that occurred during the port.

1 Introduction

While the development of parallel hardware advances very rapidly with systems consisting of hundreds or thousands of processing elements, the development and improvement of techniques and tools for debugging and performance analysis are lagging behind modern hardware technology. This is one of the main obstacles of a widespread usage of these systems.

New methods to program high performance computer systems such as HPF (High Performance Fortran) try to facilitate their use, however, the majority of parallel systems with fully distributed resources are still programmed with an explicit parallel programming model. Programming libraries such as PVM provide a model of communicating sequential processes which exchange information solely by messages. The interaction of many independent processes causes additional difficulties and bugs in parallel programs such as communication errors or performance bottlenecks which occur in addition to problems within the sequential parts of the program. The understanding of the execution behavior of many concurrent processes is very difficult and requires adequate software tools for debugging and performance analysis. However, current parallel debuggers operate at a very low level of abstraction and cannot provide enough insight into the overall execution of parallel programs. This is one of the main reasons why current tools are very often not accepted by the programmers of parallel systems [Pan93].

The following section gives a brief description of the visualization tool VISTOP for the MMK programming model. This version is currently being ported in order to support PVM applications. The changes to be made and the problems that arise are described in Section 3.

2 The Visualizer VISTOP for the MMK Programming Model

The current version of the visualization tool VISTOP (VISualization TOol for Parallel Programs) animates message passing programs developed with the parallel programming library MMK [BL90], which is similar to PVM or MPI. The basic idea of VISTOP is analogous to the classical approach to debugging sequential programs, which involves repeatedly stopping the program during execution, examining the state, and then continuing the execution. This is also the approach used in most commercially available debuggers for parallel and distributed systems.

*This work is partly funded by the German Science Foundation, Contract: SFB 342, TP A1

VISTOP tries to overcome the limitations of these extensions of traditional low-level break-point debuggers for sequential programs by providing a global view of the program state at the level of abstraction of the programmer. A global perspective makes it much easier to detect the reason for communication errors such as deadlocks or races. Main features of VISTOP compared to other approaches are [Bra94]:

- Most visualization systems are post mortem systems which analyze the program execution off-line after the program has terminated. However, debugging is an interactive process where the user examines the state of the system in order to decide which information is needed next. This approach can only be supported effectively by on-line tools. For instance, VISTOP can be used in conjunction with our parallel debugger DETOP [BW94] in order to inspect values of variables or to define break-points. On-line use reduces the amount of information that has to be gathered from the application as only the information that is requested by the user must be collected.

- Many distributed systems are missing a global time base. Thus, most visualization tools that rely on time stamped trace events cannot guarantee to show a "correct" representation of the program execution. This is not acceptable for a tool that should serve to increase the understanding of the program execution. VISTOP is based on a formal time model which uses causal relationships to enforce that the visualization is consistent with causality [SM92].

- The presentation is based on the semantics of the programming model. It shows the communication protocols and communication errors, i.e. the tool can be used during the debugging phase of a program and does not require that the program must terminate correctly.

- VISTOP supports a dynamic process model which makes the tool applicable to a wide range of applications such as distributed database or traffic control systems.

The system provides three views that show different aspects of a parallel computation. The *concurrency view* is based on a hierarchical process graph and shows communication and synchronization events. The *object creation graph* presents dynamic object creation. The *system view* shows the distribution of the program objects on a parallel system.

2.1 Concurrency View

In the concurrency view, the program objects are arranged in a two-dimensional area. The basic presentation metaphor of the view is a dynamic, hierarchical object graph. The nodes of the graph form little icons which can represent either objects of the programming model or subgraphs of the program (see figure 1).

The MMK programming model provides three predefined object types. *Tasks* are the active components which perform the calculations of the program. They communicate via *mailboxes* and synchronize via *semaphores*. The communication protocol can be adjusted by specifying the buffer size of the mailboxes and by the selection of timeout intervals of the communication functions. For example, a synchronous communication between two task objects similar to the ADA rendezvous semantics can be specified by setting the buffer size to 0 and the timeout interval to unlimited.

Usually, programming objects are displayed by a small icon which shows a characteristic symbol for the object type and the identifier of the object. The size of the icon can be adjusted by zooming. In addition, the icons change their shape to show the current state of the object. This feature is used to visualize the communication protocol and error conditions. Additional information about objects can be displayed in a so-called "detail view" which can be obtained by selecting an object with the left mouse button. For instance, the detail view of a mailbox object shows the message buffer and the task objects which were sending the messages which allows to detect communication errors that are caused by messages from wrong senders.

The layout of the object graph should ideally reflect the communication structure of the parallel application in order to make the display most effective. The arrangement of the program object can be assigned by the user accordingly by selecting and moving the objects. Communication and synchronization events are indicated by arrows. A thin arrow characterizes a non-blocking communication or synchronization. Blocking library calls are visualized by thick arrows (see figure 1).

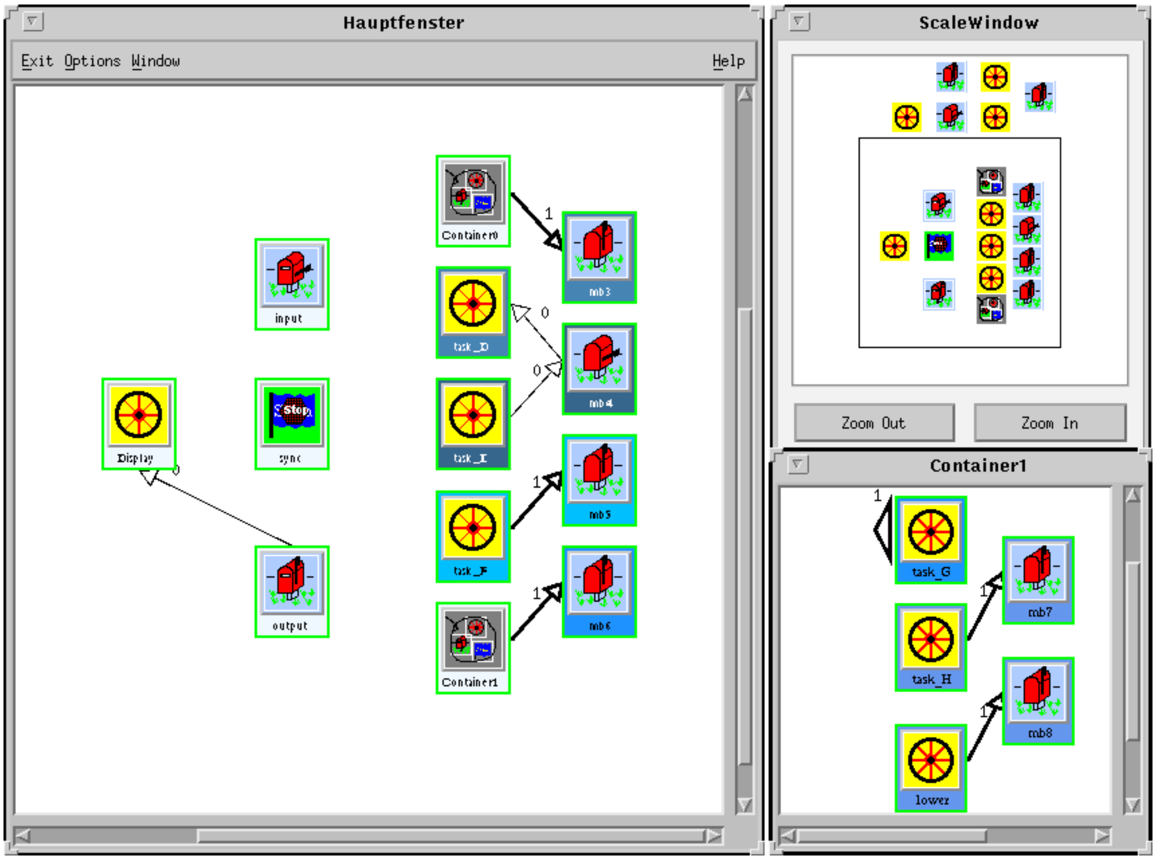

Figure 1: VISTOP Concurrency View.

2.2 Hierarchical Abstraction

Hierarchical abstraction can be used at every level of computation to control complexity and make the construction of larger systems more manageable. While the method is widely applied in textual programming (e.g. functions, procedures, modules) and in design tools such as CASE tools, currently no existing visualization tool exploits this technique.

In VISTOP hierarchical abstraction is realized by “container objects”. Each container consists of two parts, an icon and an associated concurrency window. The icon represents the group of program objects located in its concurrency window or one of its descendants. Program and container objects can be moved between different windows by drag and drop. Thus, container can be used to form a hierarchical program graph by distributing the program objects in an appropriate way.

In order to control presentation complexity of the display during visualization all concurrency windows can be closed and opened by selecting the respective container. Figure 1 shows the main window of the concurrency view that contains two container objects `Container0` and `Container1`. Only the concurrency window of `Container1` is visible whereas `Container0` is closed.

Communications between objects outside and objects inside a container are displayed by an arrow from the outside object to the container (see communication between `Container0` and mailbox `mb3`). Communication events within a container are only displayed in the concurrency window associated with the container, i.e. if the window is closed, the event is invisible (see communication between `task_H` and `mb7`). Communication between members of a container with communication partners outside are displayed by an arrow head (see `task_G`). The window that contains representations of both communicating objects shows the communication partner which is `mb6` in the example.

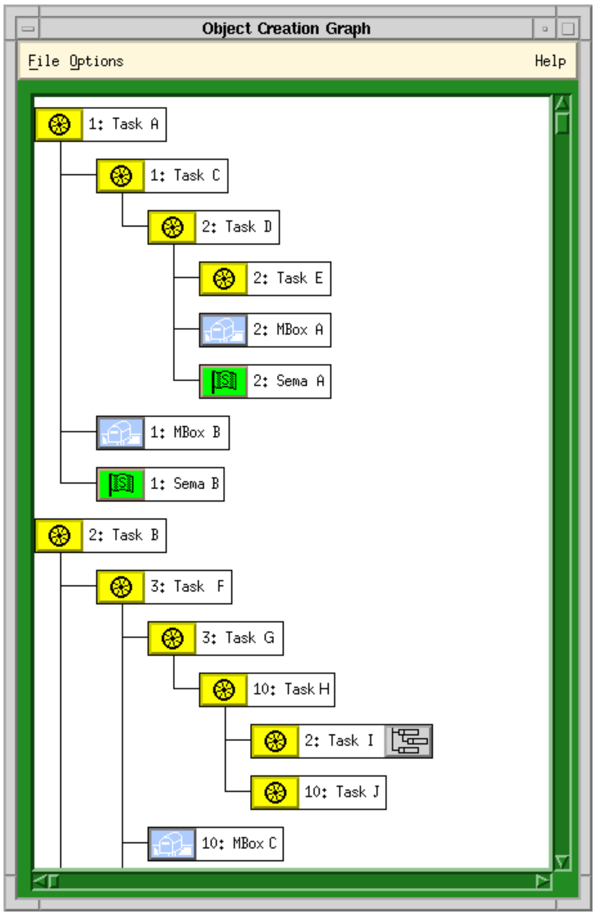

Figure 2: VISTOP Object Creation View

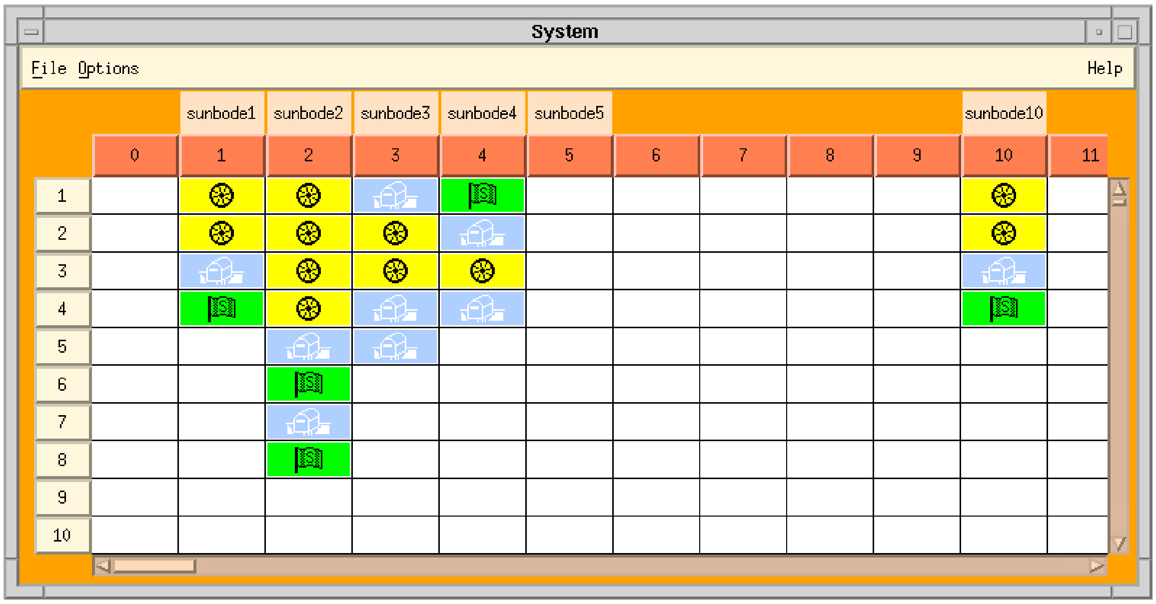

Figure 3: VISTOP System View

2.3 Object Creation View

This view displays the dynamic object creation chain of programming objects created and/or deleted frequently as it happens for example in database systems or tree search algorithms. The objects are represented in form of a graph consisting of at least one tree. An example for an object tree can be seen in figure 2. Each of the tasks that are declared statically form the root of a separate tree. The parent entry shows the task that created the child object dynamically. By selecting a node of the graph, additional information about the object is displayed (state of the object, module where the object was created, . . .).

When a new object is created, this is animated in the object creation view as follows. First, the position of the object in the tree is determined. All objects below this position move down one line. A "turtle" appears below the task which creates the new task and draws a line from the parent to the new object. Now the turtle grows and finally the icon for the new object appears. The deletion of an object is animated in a way analogously to the creation.

2.4 System View

This visualization component shows the distribution of the objects of an application onto different processing nodes. Different processing nodes may be located on different machines, when a program is distributed in a network of UNIX workstations. The display is organized as a matrix. Each column contains all objects of one node. An object may be displayed as an icon that shows its type, its name or its object identifier. By selecting an object with the mouse, additional information about the object can be obtained. The VISTOP system view is displayed in figure 3.

This view can visualize migration of objects in applications which perform migration with the automatic load balancing facility of the MMK. When objects are created or deleted dynamically, this view shows the location of the objects and their overall distribution in the system. Again, the events displayed in this view are animated to show the dynamic behavior of the underlying application.

3 Porting VISTOP to PVM

The application of the MMK version of VISTOP proved that the visualizer is a valuable tool for understanding and debugging parallel programs. For PVM, however, no tool of comparable functionality existed. Therefore, we decided to port VISTOP to the PVM programming model. This port is one part of a much larger project called "THE TOOL-SET for PVM" [LWB+95], where a set of integrated tools for the development and execution of PVM programs is being developed.

3.1 Structure of VISTOP

In order to show, which parts of the visualizer are affected by the port to another programming model, we will now briefly present the structure of VISTOP. It consists of three major parts (see Fig. 4):

1. The **data acquisition** is responsible for detecting relevant events in the execution of a program and for feeding the events to the modeling layer in a correct order. Events are either read on-line from a monitoring system, or off-line from a pre-recorded trace file. In any case, since the target systems usually don't provide an accurate global time, events from different processes must be sorted according to the causality ("happened before") relation [Lam78].

2. The **modeling layer** computes consistent global states from the event stream generated by the acquisition layer. This computation also takes into account the semantics of the programming model, so the computed state can provide information that is not directly contained in the events. For instance, the model can decide whether or not a receive call blocks its caller, since it computes the queue of receivable messages from the event stream. If the queue is empty, the task must be blocked. In the same manner, error conditions, e.g. receiving a message with wrong data type or length can be detected by the model.

3. The **visualization layer** finally visualizes different views of the global state computed by the model. It also controls the two other layers.

From this description, it is clear that most of the work has to be done in the two lower layers. Since we have a new programming library and also new target hardware, we need a different monitoring system. Also the event processing has to be adapted, since events and parameters have changed, and there are new interactions (e.g. barrier

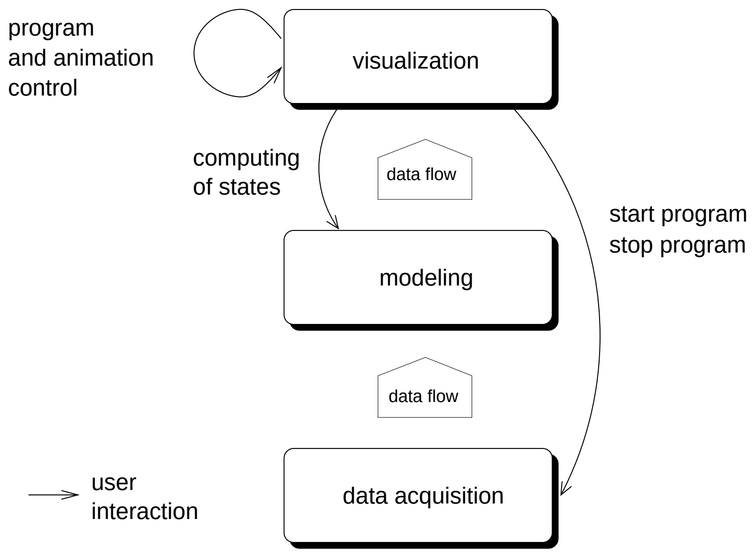

Figure 4: Structure of VISTOP

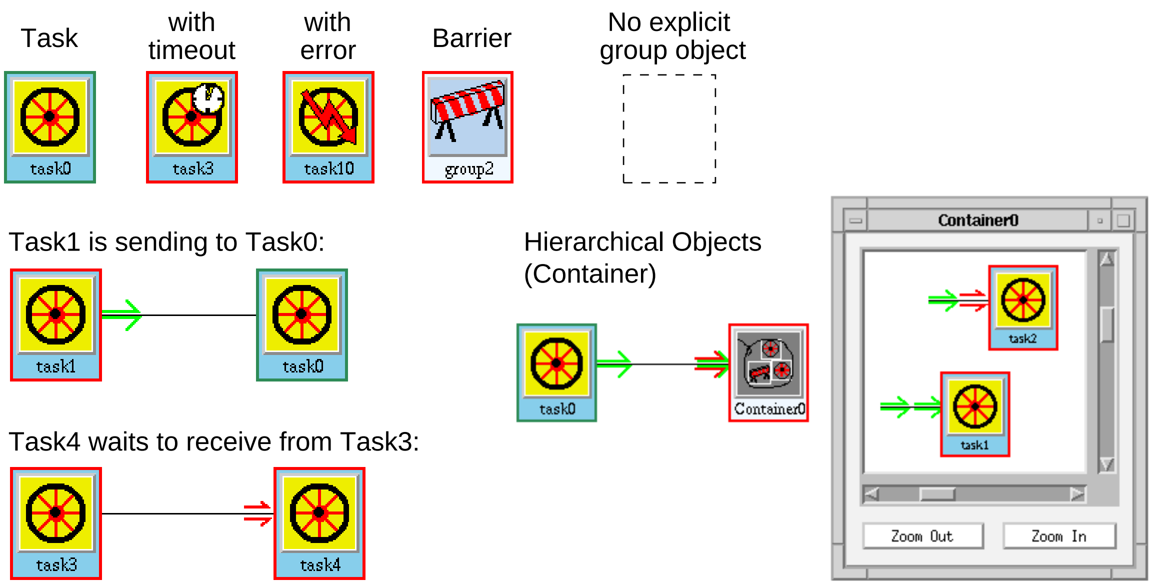

Figure 5: New visualization patterns

synchronization) that influence the computation of a causal event ordering. The modeling layer requires major modifications, since the semantics of PVM is very different from the semantics of the MMK programming model. Therefore, the way how global states are computed from events is also different. Finally, visualization patterns for the new object and interaction types in PVM have to be incorporated into the visualization layer. However, most of its features can be reused.

3.2 Visualization Patterns

In contrast to the MMK, there are no mailboxes and no semaphores in PVM; communication is performed directly between tasks. For synchronization, there is a new object, the barrier. In addition, PVM offers process groups and new communication patterns: broadcast in a group, multicast to a list of tasks and receiving messages without specifying the sender.

These differences must be reflected in the patterns that are used to visualize communication and synchronization in the concurrency view. The new patterns are depicted in Fig. 5: The task icons are identical to that of the MMK version. In addition, there is a new symbol for barriers. It is created when the first task in a group calls `pvm_barrier()` and is destroyed when the last task leaves the synchronization call.

There are no explicit objects visualizing groups in the concurrency view. A first idea has been to visualize each group by a container object that contains all the tasks in that group. However, containers in VISTOP form a strict hierarchy of tasks, i.e. a task can only be in one container, while in PVM, a task can belong to more than one group. Thus we decided to provide an additional view, called group manager. It displays the list of all groups that exist at the current point in the animation, and allows to select a single group, a union, or an intersection of groups. When this is done, all visualized objects that do not belong to the specified group (or union/intersection) loose parts of their coloring, as shown in Fig. 6. In this way, users can quickly determine the group membership of visualized objects.

Since PVM uses direct communication between tasks and offers some new communication structures, also the way of visualizing interaction between objects has been modified, as shown in Fig. 5. Instead of drawing thin or thick arrows between objects, now a simple line is drawn, denoting that there is some interaction. Additional arrow heads in different 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 interaction. The direction of the arrow head indicates the direction of the (requested) information transfer, while its position marks the active partner of an interaction. For example, in the middle left of Fig. 5, `task1` is sending to `task0`, since there is a filled arrow head attached to `task1` that is directed towards `task0`. The lower left of the figure shows the visualization pattern when `task4` waits for a message from `task3`. The new visualization strategy can also show multiple interactions, e.g. between a task and a container object. In the lower right of Fig. 5, `task0` sends a broadcast 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`. Broadcasts and multicasts are visualized in an intuitive way by showing that a task sends to multiple tasks at the same time (see `task0B` in Fig. 6).

In addition to these changes in the concurrency view, the detail view of task objects has been adapted to the task information available with the PVM programming model (see Fig. 7).

3.3 Data Acquisition and Modeling

Although the long term goal for our work is to provide the visualizer VISTOP for PVM based on an on-line monitoring system, the first version uses XPVM for data acquisition. XPVM records a trace of events in a PVM application and is able to store this trace in an SDDF file [GKP94, BDG+95]. Using XPVM for data acquisition allows VISTOP can be used for PVM before our own monitoring system is ported. However, we currently loose the interactive control over the application's execution. Therefore in a next step, the visualizer will make use of the monitoring system developed in the OMIS project [LWSB95] and will then also become part of THE TOOL-SET.

The modeling of global states from a sequence of events turned out to be much more complicated for PVM than it was for the MMK programming model. The main reason for this fact is the absence of an event that indicates the insertion of a message into a task's message queue. Such an event is not provided by the built-in trace facility of XPVM. For the modeling of global states, this has three consequences:

1. The message 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.

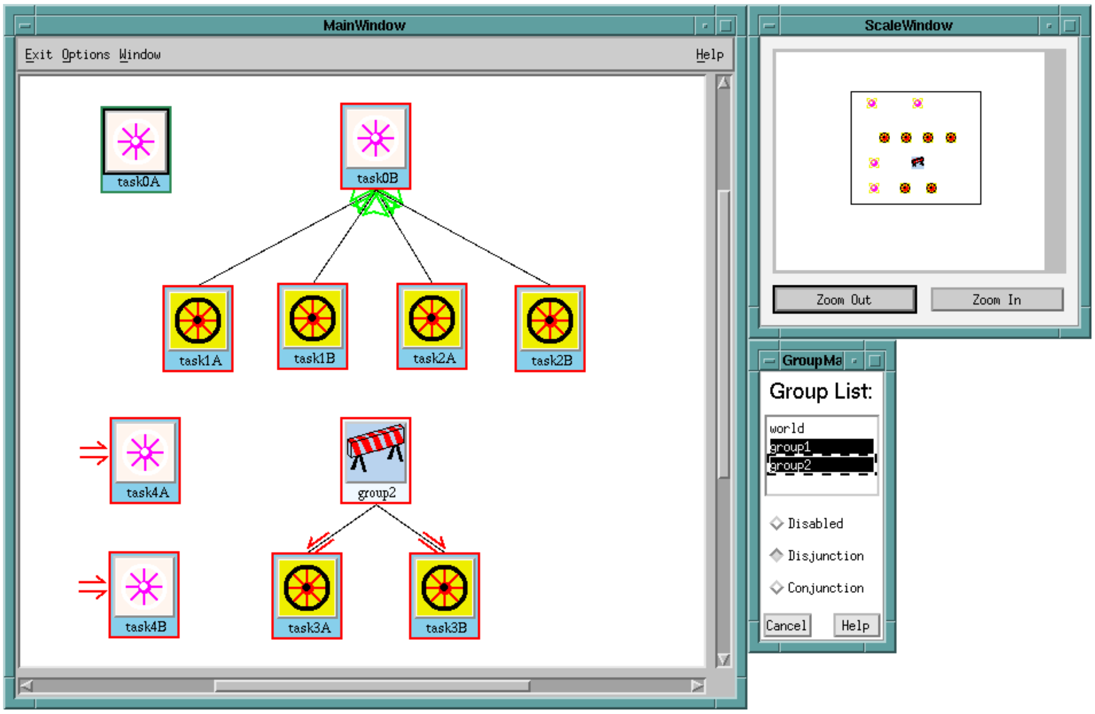

Figure 6: New concurrency view

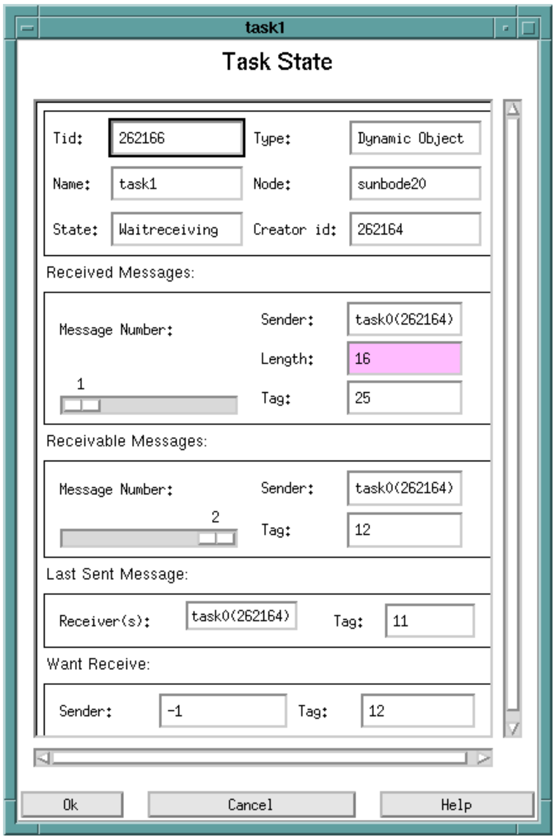

Figure 7: Detail view of a task

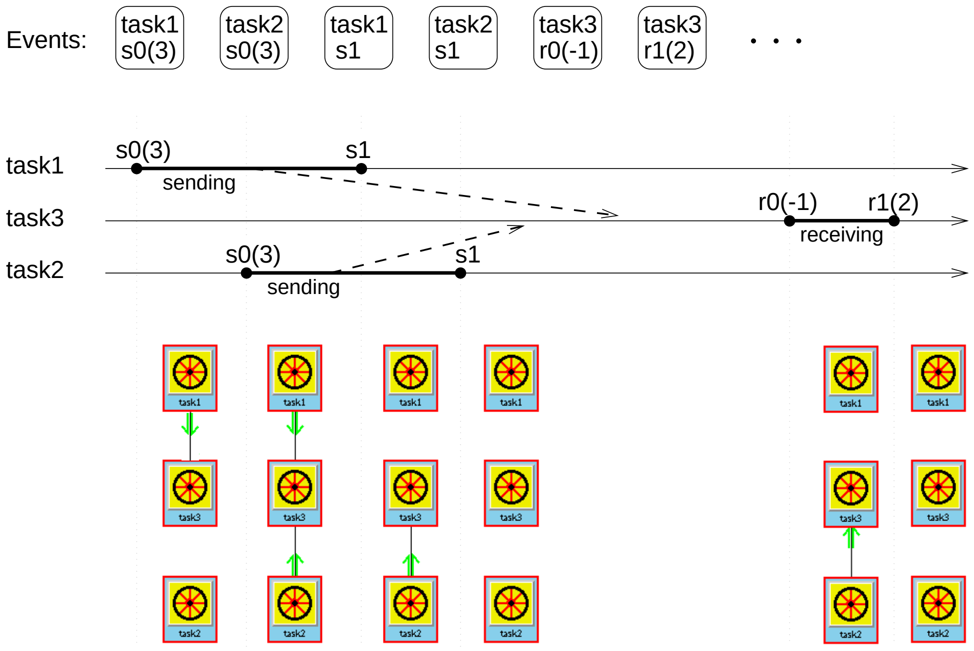

Figure 8: Processing of communication operations

2. The VISTOP animation is designed to show when a task blocks in a receive call due to an empty message queue. Since the point in time, when a message is put into the message queue of a task is unknown, it is not possible to decide whether the task blocks in a receive call or not. As the current solution, we assume that a message is inserted into the receiver's queue when the sender finishes its send operation.

3. It is impossible to easily visualize receive operations in the usual way, if no sender is specified in the receive call, i.e. a wild-card is used. Since the order of messages in the task's message queue cannot be modeled properly, it is not known in this situation, which task is the sender of the message that is being received.

Fig. 8 illustrates the modeling process and the last problem just mentioned. The first line shows a simplified event stream that is the input for the model. `s0`/`s1` denote the start/end of a send call, `r0`/`r1` the start/end of a receive operation. The numbers in parentheses indicate the destination or source task, -1 denotes a wild-card. The space-time diagram below shall indicate the result of the modeling process: a sequence of global states is computed. Finally, as shown in the bottom half of the figure, each global state is visualized by a graphical pattern.

For the first four states, the modeling and visualization process is quite straightforward. However, a problem arises when event `r0(-1)` is beefing processed. The global state after this event is the following: "`task3` is receiving a message from `task2`"[1] However, according to the description above, the model cannot determine the sender of the message being received until it sees the event `r1(2)` that always contains the actual sender. So in order to visualize correct global states, the receive operation is shown not before the modeling layer has processed the end-of-receipt event. Then an artificial event is introduced that causes the model and the visualization to return to a state where the task finished receiving. So in effect, receive operations are shown one step too late, but a simple analysis shown that at least the causality relation cannot be violated by this delay.

As a longer term solution, we will try to provide the crucial event "a message is inserted into a tasks message queue" with the PVM on-line monitoring system developed in the OMIS project. However, this will most probably require some modifications in the PVM daemons. We will therefore try to further cooperate with the PVM development team in order to avoid the necessity of a special PVM version for the usage of our tools.

[1] Even if the messages have the same tag and arrive at `task3` in the order suggested by the dashed arrows, the task can still receive `task2`'s message first, if there is a direct connection between `task1` and `task3`. In this case, PVM will always first ask the daemon for a receivable message before it checks the direct connections.

4 Conclusion

The application of VISTOP for the MMK programming model has shown that visualization can increase the understanding of parallel programs. Hierarchical abstraction is quite helpful to increase the scalability of visualization tools for functional debugging of parallel programs. The concept is easy to understand and can be applied to any parallel program.

The visualizer is now also available for the PVM programming environment in a first version. It currently animates SDDF traces generated by the XPVM tool. Future work includes the development of a PVM on-line monitor that will acquire the data needed for VISTOP, and an integration of the visualizer into THE TOOL-SET.

References

[BDG+95] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Recent Enhancements to PVM. *International Journal of Supercomputing Applications and High Performance Computing*, 1995.

[BL90] T. Bemmerl and T. Ludwig. MMK - A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing. In H. Burkhart, editor, *Proc. of CONPAR90 VAPP IV*, pages 744–755, Zürich, Schweiz, 1990. Springer-Verlag.

[Bra94] Peter Braun. *Visualisierung des Ablaufverhaltens paralleler Programme*. PhD thesis, Technische Universität München, 1994.

[BW94] T. Bemmerl and R. Wismüller. On-line Distributed Debugging on Scalable Multiprocess or Architectures. In *High-Performance Computing and Networking*, volume II, pages 394–400, München, April 1994. Springer.

[GKP94] A. Geist, J. Kohl, and P. Papadopoulos. Visualization, Debugging, and Performance in PVM. In *Proc. Visualization and Debugging Workshop*, October 1994.

[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.

[LWSB95] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS – On-line Monitoring Interface Specification. Internal report, Technische Universität München, 1995.

[Pan93] Cherri M. Pancake. Customizable portrayals of program structure. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 64 – 74, San Diego, Ca, 16. – 18. Mai 1993.

[SM92] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB124-15/92, Universität Kaiserslautern, 1992.