Visualizing PVM Executions

Thomas Kunz and David J. Taylor
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

June 26, 1995

Abstract

Because of the complexity of distributed applications, understanding their behaviour is a challenging task. Frequently, tools that provide graphical visualizations based on process-time diagrams are provided to facilitate this understanding task. While a wide variety of such tools exist, few address the complexity problem directly. In contrast, our research has focused on the problem of visualizing complex distributed executions. Our visualization tool, Poet, provides abstraction facilities in both the process and time dimension to reduce the apparent complexity of a distributed execution. Because of its emphasis on target-system independence, adapting Poet for PVM 3.3, utilizing the tracing facility provided in this version of PVM, was relatively straightforward. This paper summarizes the necessary steps to adapt Poet to PVM and presents a few simple visualizations. We also discuss the use of event abstraction to provide execution visualizations that match the programming model provided by PVM, in particular with respect to group-communication primitives.

1 Introduction

To facilitate reasoning about the execution of massively parallel and distributed applications, we are developing a visualization tool that displays application execution using process-time diagrams. These displays are based on the notion that each process consists of a totally ordered sequence of primitive events, each representing some activity performed by a process and considered to take place at an instant in time. Typically, the lowest level of observed behaviour consists of events representing process interactions, such as sending and receiving messages and process creation and termination.

Our tool, Poet [11, 12], displays processes and events using two-dimensional process-time diagrams. The placement of events along the time axis is based on either their occurrence in real time or their relationship to other events in the partial order introduced by Lamport [9]. Each display mode has value: for example, partial-order displays are useful for debugging, while real-time displays are useful for performance analysis.

Massively parallel and distributed executions typically contain many processes and primitive events. To assist in understanding such executions, abstract visualizations are provided in which processes are grouped into process clusters and primitive events are grouped into abstract events. Such abstractions are either derived automatically [4, 7, 8] or created manually by a user [5, 12]. Poet allows a user to navigate the resulting hierarchy of abstract views, to collect increasingly detailed information for smaller parts of the execution, for example.

Currently, Poet runs in a variety of target environments, such as OSF DCE, Hermes, ABC++, μC++, and SR. This paper describes how we adapted Poet to PVM 3.3. The basic event model in Poet allows only one-to-one relations. To support PVM features such as multicast and group communication, event abstraction was used. Such activities are modeled as multiple primitive events, grouped into abstract events, and displayed as single entities.

The paper is organized as follows. The next section presents examples of the basic visualizations provided by Poet. Section 3 outlines how we adapted Poet to PVM, utilizing the tracing facility of version 3.3. Section 4 discusses the use of event abstraction to provide visualizations that match the programming model provided by PVM, in particular group-communication and synchronization primitives. Finally, Section 5 summarizes the current status of our implementation and describes possible future work.

2 Basic Visualizations

Poet is based on a very simple and general model of behaviour of distributed applications [14]. The model contains two basic concepts: traces and events. A trace represents some entity with sequential behaviour and events represent activities of interest. Each event is assumed to be associated with a single trace. In addition, events may be connected to each other, as event pairs, and pairings may be synchronous or asynchronous.

For each specific target system, a mapping between these concepts and the entities in the target environment must be defined. In the case of PVM, traces correspond to tasks and events represent the execution of a subset of the PVM library functions. In particular, we are interested in all functions that correlate tasks with one another, e.g., message sends and receives, group-communication and synchronization functions, and task creation/termination. Functions whose invocations have only local (e.g., internal to a task) effects, are ignored. The rationale here is that the visualizations focus on the relationship between tasks, see also [1, 6]. Once a user understands these relationships, he/she can examine the local (sequential) computation for each task using traditional approaches.

In some cases, we model the execution of a PVM function with two rather than one primitive events, one event representing the invocation and a second event representing the termination of the function. This provides useful insight into the duration of a function. Unlike XPVM, where each function execution is displayed as an interval, we limit ourselves to the following functions: `spawn`, `barrier`, and `receive`. For these functions, information about their duration might provide valuable insight into the performance of a PVM execution. They help to answer questions such as: how long is task `xyz` blocked until it receives a message? Do some tasks spend a much longer time waiting for a barrier synchronization to occur than other tasks? For many other PVM functions,

such as `send`, the prime determinant of their duration is their implementation within the PVM library and the performance of the system software they utilize and therefore beyond the scope of an application programmer. In these cases, we abstract away from the duration. The events representing the function execution are collected at the invocation of the function for the various `send` primitives and at the termination of a function in all other cases.

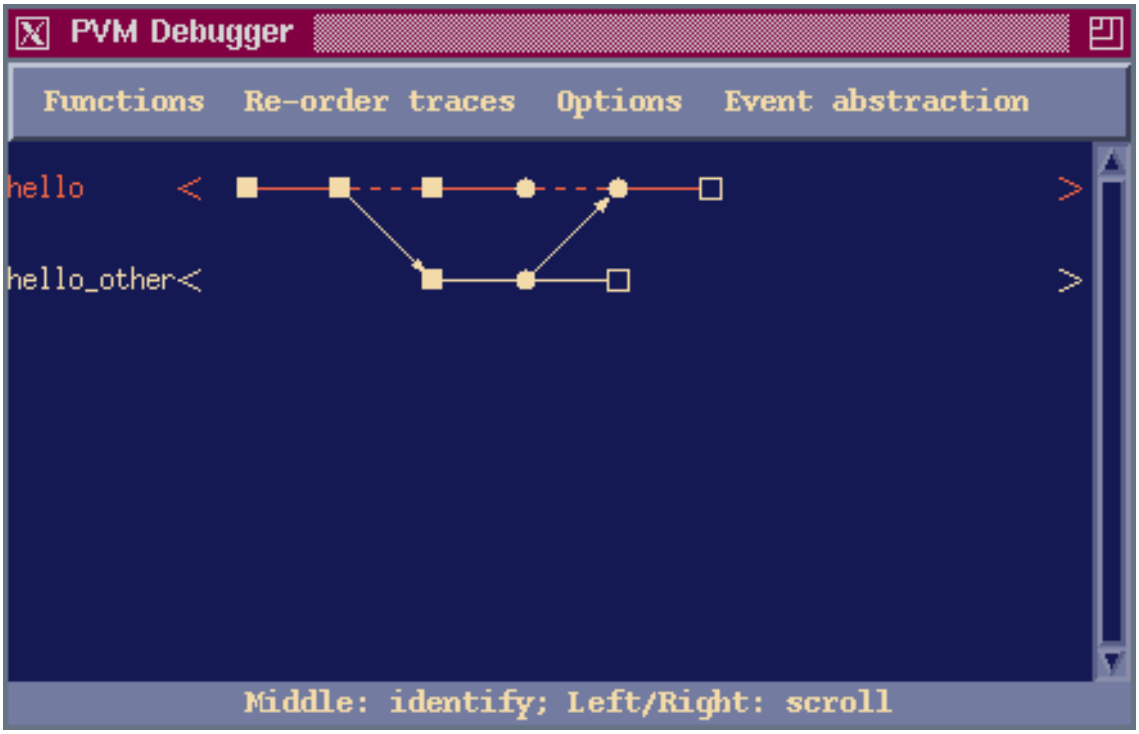

Figure 1: Visualization of `hello`, using a partial-order display

Both display types offered by Poet use the same basic symbols. Each task is represented by one horizontal line, symbols placed on the appropriate line represent events. Time flows from left to right, and a scrollbar allows for scrolling in the vertical (process) dimension. (Scrolling in the time dimension is more complex and depends on the display type; it has also been implemented by the tool.) The lines are preceded by the task name. Events that are paired (such as sending and receiving a message or spawning a task and the start of a task execution) are connected by a sloping arrow. Sloping arrows are used to represent the asynchronous nature of these event pairings, synchronous pairings would be visualized by vertical arrows. Different symbols are used to distinguish different types of events. Filled squares indicate task-creation events, open squares depict task-termination events. Filled circles represent asynchronous message sends and receives (both the blocking for a message and the actual receipt of a message). The task lines are drawn in three different states, approximating the task state. Before a task starts (indicated by a filled square) and after its termination (the open square), the line is invisible. After executing a blocking

receive primitive, the task is blocked until it receives a message. This is represented by a dashed line. In all other cases, a task is assumed to be active, displayed by a solid line.

Figure 1 shows one of the visualizations for an execution of the `hello` PVM example that is part of the source distribution. In this simple distributed application, the first task, `hello`, starts execution, depicted by a filled square. Since this is the first task in the system, it has no parent, e.g., this start event is not connected to a `spawn` event in another process. It then creates (or spawns, in PVM terminology) a second task, named `hello_other` (the second and third filled squares) and blocks on a message receive, depicted by a filled circle. The trace line is drawn with dashes to visually indicate that the task is blocked. The second task starts execution, depicted by a filled square. Since this task was created by another task, the start event is connected to the corresponding `spawn` event, the second filled square in the first task. The second task then sends a message to the first task (the filled circle) and exists (the open square). The first task prints, upon receipt of this message (the second filled circle, connected to the send/filled circle in the second task), the ubiquitous `hello, world from xxx` message on the screen, where `xxx` is the name of the machine on which `hello_other` executed, and exits too.

In distributed executions, one basic relation between primitive events is the $\rightarrow$ ("happened before") relation introduced by Lamport [9], and originally defined for IPC events in asynchronous systems only. This relation can easily be extended to cover process-creation and process-termination events as well as synchronous message passing [2]. The $\rightarrow$ relation induces a partial order on the set of events and is of particular importance for the analysis of distributed applications: an event a can only influence another event b if it happens before that event. Two events cannot influence each other when they are concurrent, that is, unrelated by the $\rightarrow$ relation. The display shown in Figure 1 is built based on this relation. A primitive event a happens before another primitive event b if and only if there exists a directed path from a to b, following message arrows and process lines in the time direction. Displays built on this relation are sometimes called *logical-time* displays and are a useful tool for functional debugging (e.g., where one is concerned with correctness of an application) and to reason about causal relations in distributed executions.

To support performance analysis and debugging, Poet also provides visualizations based on real-time. Figure 2 gives an example, using the `hello` application again. Since we are collecting events from potentially different machines and we cannot assume the existence of a global, synchronized clock, Poet provides a built-in real-time-adjustment algorithm. Since we do not, in general, have any control over the clocks used in the target environment, we decided to adjust the real times assigned to events after they were reported to Poet.

If clocks were perfectly synchronized, real times would always be consistent with the partial order, but real times obtained from actual clocks may not have this property. In particular, "tachyons" might exist, that is, pairs of events such that the "sending" event appears to have occurred *after* the corresponding "receive" event. Such event pairs point to local clocks that are out of synchronization, and serve as the basis of the time adjustment, the details of which are described in [13]. The result of the time adjustment is that the real-time is consistent with the $\rightarrow$ relation, i.e., $a \rightarrow b \Rightarrow T(a) < T(b)$, where $T(a)$ denotes the real-time assigned to event a after adjustment.

In the real-time display, time again flows from left to right. Events are placed along the time axis according to the (adjusted) time of their occurrence. Tick marks at the bottom of the display

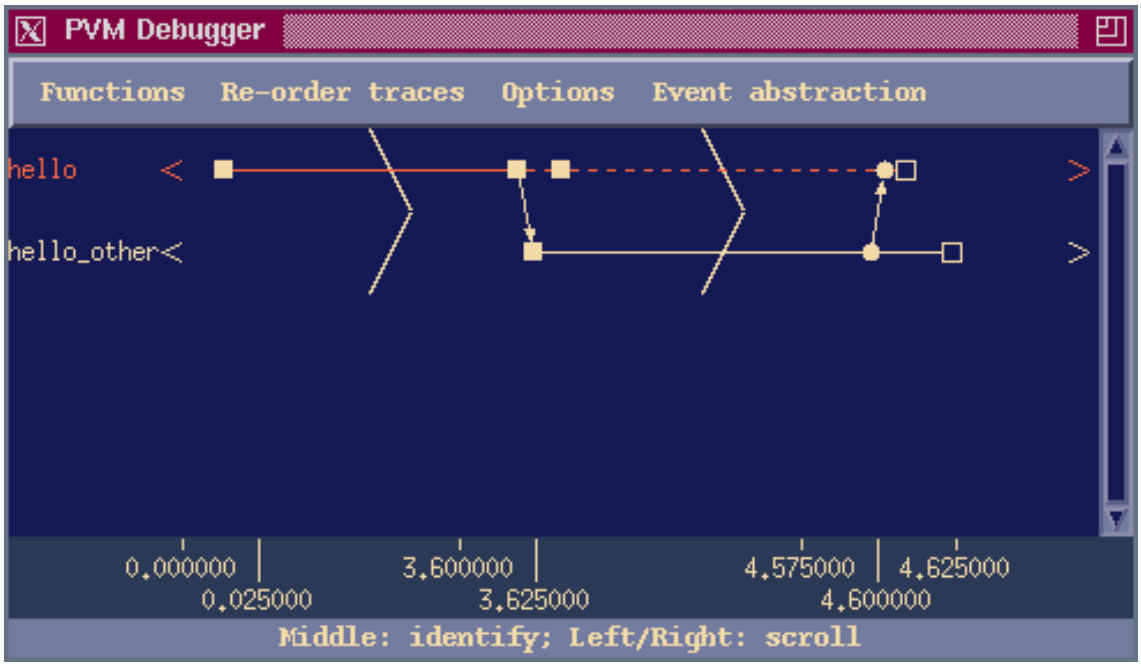

Figure 2: Visualization of `hello`, using a real-time display

provide the user with some indication of the display scale, which can be modified within certain limits. To economize on display space, large gaps (long time periods that contain no events) are replaced with "cut" marks, two of which are shown in Figure 2. Also, Figure 2 seems to miss one event, compared to Figure 1, the "ready-to-receive" event. For the given display scale, this event overlaps with the third filled square. By adjusting the display scale to a finer granularity, they can be separated on the display.

3 Event Collection and Processing

The design of Poet follows a client-server architecture. In a minimal configuration, it consists of a disk-server process and two clients of the disk server: the debug-session process and the checkpoint process.

The debug-session process is responsible for direct interaction with the user. It obtains user input via the keyboard and mouse, and generates appropriate displays in response. Most of the complex algorithms for the visualization tool are in this process, including those for display building.

The disk server is responsible for communication with the processes making up the target application. Event data is received from event-collection "hooks" embedded in the target environment and placed in a disk file, with minor transformation to make it more convenient for later use.

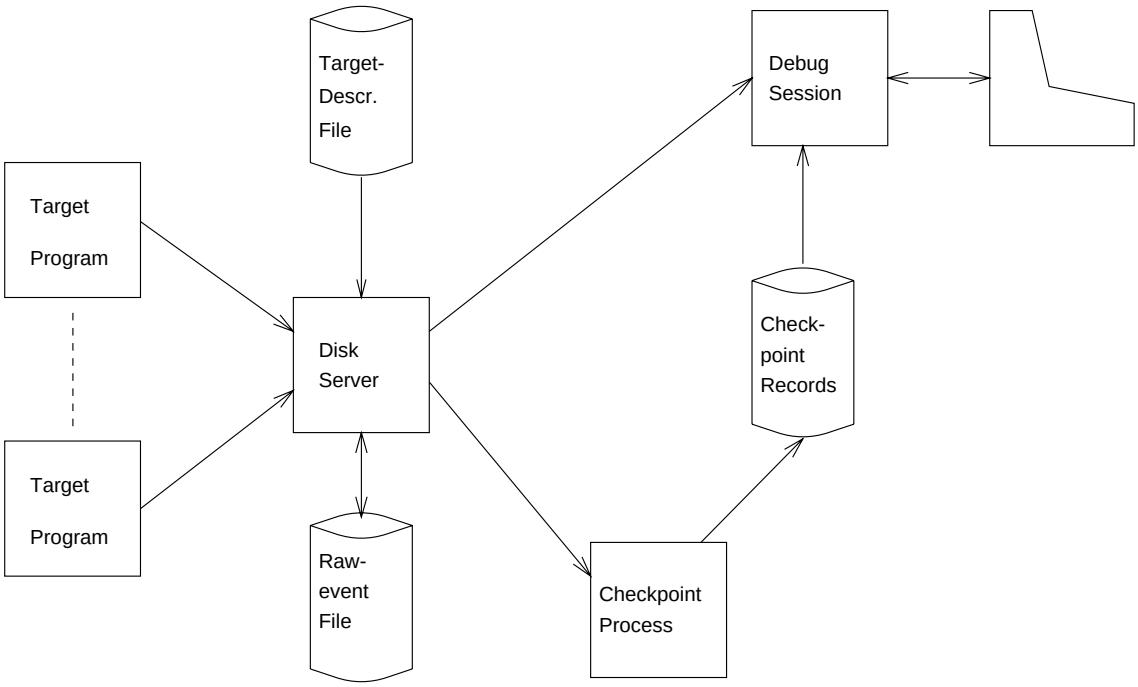

Figure 3: Architecture of Poet

The semantics of the event data reported from each of the various target systems supported are described in a separate target-description file that is read by the disk server. The event traces are self-identifying in that they start with an identifier that uniquely characterizes the target system, allowing the disk server to read in the correct target description file. Clients of the disk server (the other two processes) request event data as required, using a well-defined protocol.

The checkpoint process exists solely to improve performance. To determine the $\rightarrow$ relation efficiently, logical timestamps are associated with events [2, 9, 10]. Because timestamps are $O(N)$ in size if N processes exist, storing a timestamp with each event is not feasible. Thus, timestamps must be computed as needed, but the calculation of a timestamp, in principle, requires tracing the execution history of the target application from its beginning to the point at which the specified event occurs. To make calculation of arbitrary timestamps reasonably efficient, the checkpoint process simply timestamps all available event data and periodically writes out a checkpoint containing the internal state of the timestamp algorithm. The debug-session process can then find a checkpoint preceding an event that is to be timestamped and need only run the timestamp algorithm forward from that checkpoint. Communication between the debug-session and checkpoint processes takes place only through the file of checkpoint data, whereas communication between the disk-server and its clients involves explicit message passing.

The design, consisting of a single server and multiple clients, allows easy addition of further processes as necessary. For example, the debug-session process can be instantiated several times to provide

multiple simultaneous views of the execution. Clients have also been written to make "annotations" to the raw event data, such as marking events participating in data races, and converting run-time information such as a thread address into a more meaningful symbolic form by debug-time use of the symbol tables stored in object files.

When adapting Poet to PVM, one of our goals was to avoid changes to the PVM source and to allow PVM executions to be visualized without prior re-compilation. We thus opted to utilize the PVM 3.3 tracing facility. In a nutshell, PVM executions are spawned with a certain number of tracing flags set (to detect the execution of those PVM functions that we associate with primitive events). This trace information is pre-processed by a separate PVM task, the `collector`, which transforms the event stream into the format expected by Poet. Also, to enable real-time displays, the physical time associated with each event occurrence is the time reported by the PVM tracing facility, not the time that `collector` pre-processed the event. Figure 4 shows this architecture.

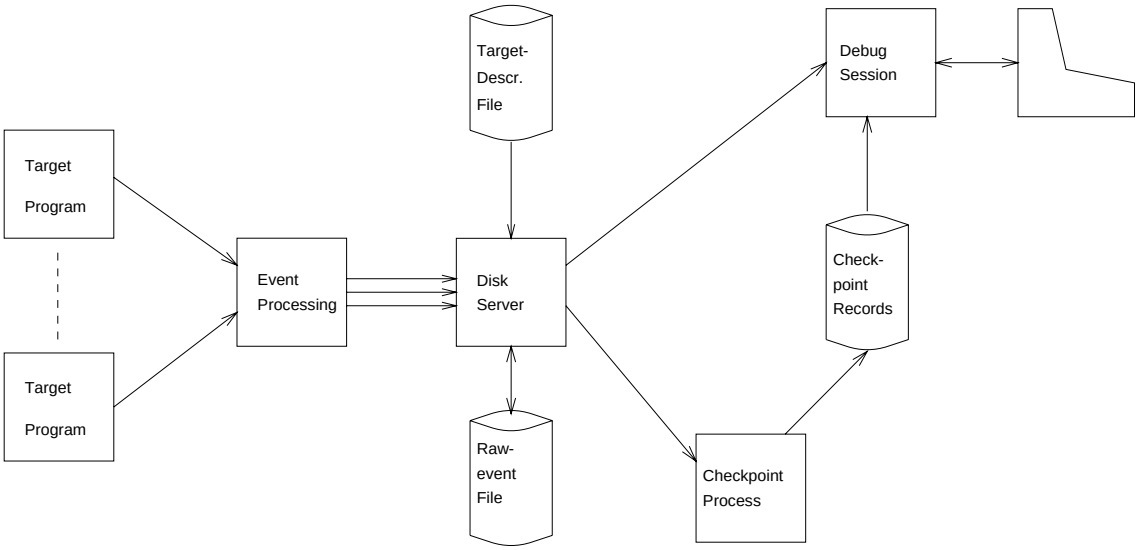

Figure 4: Combining PVM and Poet

While this approach seems straightforward, two comments are in order. First, the `collector` task needs to establish n connections to the disk-server, one for each host in the parallel virtual machine. The real-time adjustment described above depends on the fact that primitive events that are timestamped with potentially out-of-sync real-time are reported over different event streams. Events reported in the same event stream are supposedly timestamped by the same (local) clock, and therefore need no adjustment.

Second, the introduction of a separate process adds overhead to Poet. Alternatively, we could have inserted our own event-collection "hooks" into the PVM library or included the pre-processing functionality in the disk-server process. However, this additional overhead is well worth its price. First, the introduction of `collector` allows the execution of PVM applications without changes to the PVM system or re-compilation of the source. Second, the overhead of pre-processing the event

stream is negligible compared to the overhead of tracing a PVM execution in the first place, certainly in version 3.3. Others report a tracing overhead of up to 1000% [3]; our own measurements show that the pre-processing time is always only a small fraction of the tracing overhead. Finally, this design allows us to separate the event storage and display functionality from the event-collection mechanism. In other tools, for example XPVM, event collection, storage, and display are all combined in one process. The resulting PVM task is very resource-intensive, and competes heavily with the actual application for CPU resources (when performing on-line visualization). In contrast, reporting the event stream to the disk server is done via TCP/IP connections, so the resource-intensive processes could be migrated to a host that is not part of PVM, minimizing the resulting perturbation.

4 Advanced Visualizations

The visualizations described so far are fairly simple, following the execution model described above closely. In fact, in most of our target environments, the mapping of our basic model to executions in that environment is straightforward. In particular, process-creation, interprocess-communication, and synchronization primitives are all strictly one-to-one. In PVM, however, additional primitives exist that introduce one-to-many and many-to-many relations. The task-creation primitive, for example, allows the spawning of n instances of a task instead of only one instance at a time. Communication primitives such as `mcast` and `bcast` relate one send event to a number of receive events, and group operations such as `barrier`, `scatter`, `gather`, and `reduce` relate n tasks to each other. Figure 5 depicts a sample execution of the `spmd` example distributed with the PVM source, which utilizes two of these more advanced primitives. The display is built based on the $\rightarrow$ relation.

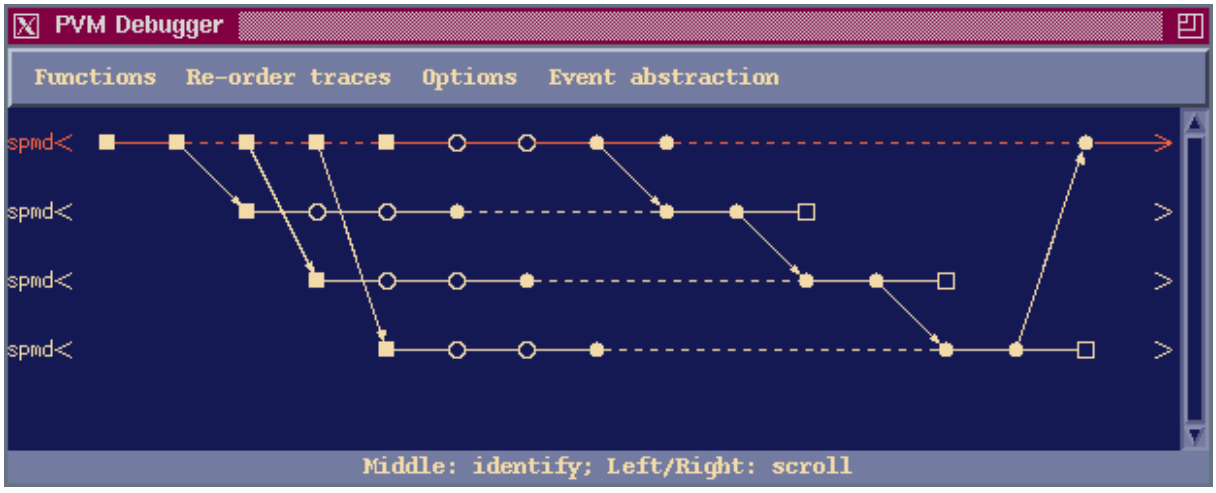

Figure 5: Spmd execution, first attempt

The first task spawns three additional copies of itself, using one invocation of the `spawn` command.

All four tasks join a group and perform a barrier synchronization. After the barrier is passed, one message is passed around the set of tasks, establishing a logical ring.

To visualize these advanced primitives, three approaches were considered. A first solution would be to visualize them as internal events in each task, without explicit visual connection. This is, for example, the approach taken by XPVM to visualize all group-related primitives. A `barrier` operation, for example, might be depicted by n primitive events, two for each task involved in the barrier synchronization. (As pointed out earlier, `barrier` operations are one of the few selected functions where we trace both barrier entry and barrier exit to support performance analysis and debugging.) An example is given in Figure 5, where barrier entry and exit are depicted by open circles. This solution, however, is very unsatisfactory. As discussed above, the primary purpose of our visualization is to help a user to understand how tasks related to each other. Clearly, the four tasks interact through the barrier operation, which should also be explicitly indicated by the display.

A second solution, feasible for primitives that result in one-to-many relations such as `mcast`, `bcast`, or `spawn`, is to model them as a number of one-to-one relations. For example, a multicast to five different tasks could be visualized as five consecutive sends from the same task to each of the five tasks involved in the multicast. Figure 5 shows how the single `spawn` command is replicated three times to visualize the creation of three additional tasks. This solution is similarly not very appealing, for two reasons. First, it is not obvious how to extend this idea to primitives that introduce many-to-many relations, such as a barrier synchronization or a `reduce` operation. Second, such a visualization blurs the abstraction level. Ideally, the visualization should work at the abstraction level provided by the programming model. In the above example, the spawn operation is performed by *one* primitive, so the representation should indicate this by the use of *one* symbol. Otherwise, it becomes indistinguishable from three consecutive spawns, even though this might be an important distinction for understanding the application logic.

The third approach, therefore, addresses the shortcomings of the first two approaches by explicitly indicating the resulting interprocess relations, be they one-to-one, one-to-many, or many-to-many, working on the abstraction level provided by the PVM programming model. To achieve these goals, we utilize the concept of *event abstraction* [4, 5]. An abstract event is a set of more primitive events. Our previous research has identified how to handle abstract events with a certain internal structure. We can define a $\rightarrow$ relation on event sets that contain both primitive and abstract events, we know how to timestamp abstract events to determine the $\rightarrow$ relation efficiently, and how to build displays that contain abstract events, represented by one symbol. Therefore, to visualize more complex interactions, we proceed as follows. First, we capture and pre-process primitive events as before, modeling events that introduce one-to-many or many-to-many relations as n independent primitive events or n one-to-one relations, following our basic model. While breaking down complex interactions this way, we also retain information about which primitive events belong together, modeling them as an abstract event. The disk server is informed about the existence of these abstract events and in turn directs the debug-session process to display the execution at the proper abstraction level. The resulting architecture is shown in Figure 6.

In comparison to Figure 4, we have added a new process, the `abstractor`, to Poet. This process has a dual "personality": it is a PVM task, receiving information from the `collector` task about

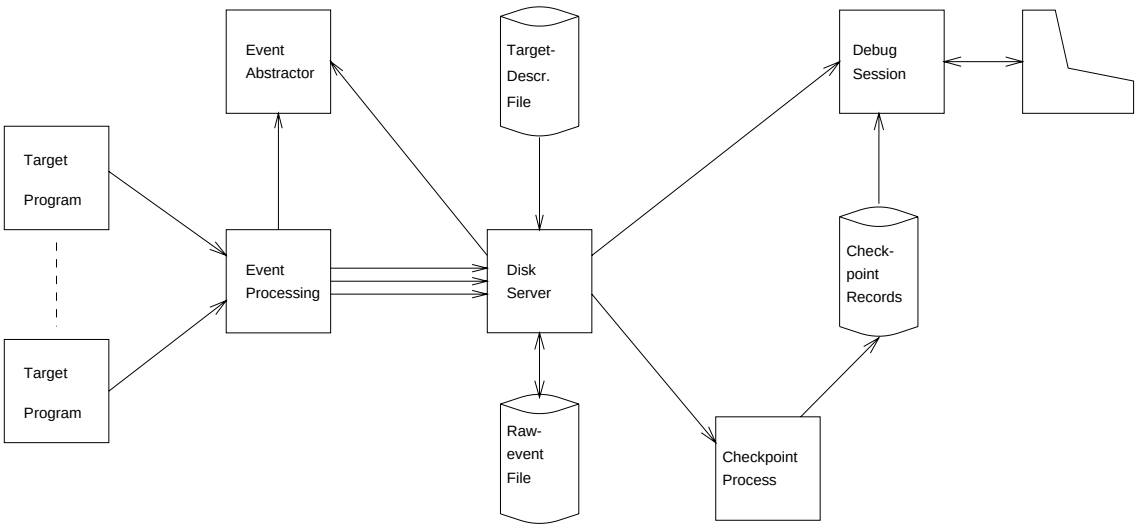

Figure 6: Introducing event abstraction

abstract events, and at the same time a disk-server client, using the disk-server protocol to inform the disk server about the creation of new abstract events. The detection and creation of abstract events is fully automatic, the membership of primitive events being determined by the semantics of the communication primitives. Separating the `abstractor` from the `collector` was done for two reasons. First, the necessary processing overhead in the `abstractor` is separated from the primitive event preprocessing. This becomes particularly important when we consider complex distributed executions which generate a large number of primitive events, where efficient pre-processing and forwarding of events to the disk server is of prime importance. Second, a separate `abstractor` can be instantiated as a client process of the disk-server, enabling it to utilize the disk-server protocol. This protocol already includes requests to create, delete, and modify abstract events, as part of the user interface offered by the debug-session process. The `collector`, on the other hand, is not a disk-server client and can therefore not utilize the existing facilities directly.

Figure 7 presents a visualization of the same execution shown in Figure 5, this time utilizing event abstraction for the depiction of the more advanced PVM primitive. In general, abstract events are visualized by a rectangle stretching over a number of trace lines. The intersection between the rectangle and the trace line is highlighted when primitive events from this task are members of the abstract event. Figure 7 contains two abstract events. The first abstract event consists of the three primitive spawn events in the first task, collapsing them into a single symbol on the display. This clearly indicates that the three tasks are created as a consequence of the execution of *one* primitive in the source of the first task. The eight primitive events representing the barrier synchronization (the open circles in Figure 5) form the second abstract event, relating all the processes involved in the barrier synchronization to each other. In summary, Figure 7 represents a much more "truthful" visualization of the application execution than Figure 5, providing a visualization at the abstraction level offered by PVM's programming model.

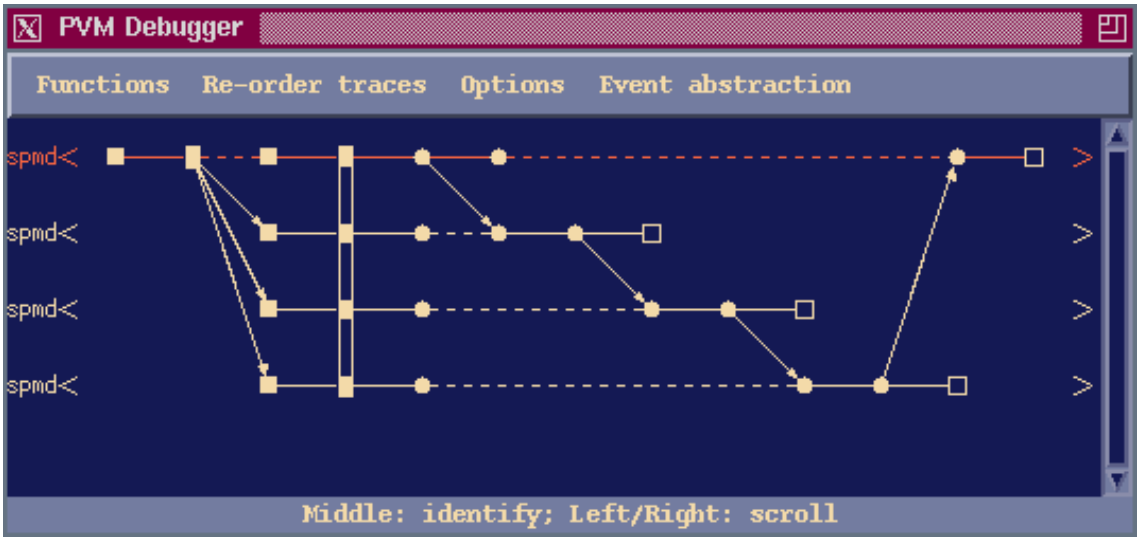

Figure 7: Introducing event abstraction

5 Conclusions

Our research effort has centered around the creation of tools to visualize the execution of parallel and distributed applications. Particular emphasis has been placed on keeping the tool portable or target-system independent and on providing mechanisms to visualize executions at various abstraction levels to facilitate the understanding task for complex applications. This paper has discussed our adaptation of Poet to PVM and provided examples of a few relatively simple visualizations. Poet provides visualizations based either on the notion of logical time, which are suited for reasoning about the functionality of an execution, or on the notion of real-time, which supports performance analysis and debugging.

PVM provides a number of primitives that extend traditional message-passing environments, such as group communication. We briefly discussed some approaches to visualizing these primitives and their respective shortcomings. Rather than inventing new visualizations, we employed the already available event-abstraction facility to provide adequate displays of an execution. The resulting architecture was outlined and an example visualization shown.

Poet has been under development for a number of years and is relatively stable and mature. The adaptation to PVM is nearing completion and we are currently conducting experiments to examine the performance of Poet, particularly in contrast to XPVM 1.0.3 and the new XPVM 1.1.0, as well as issues of scale. To this end, we have obtained a number of non-trivial PVM applications from various sources.

One aspect of our research is the automatic derivation of abstraction hierarchies to manage the vast amount of information involved in understanding the execution of complex applications. Tools to

derive such abstraction hierarchies are described in [4, 7, 8]. Poet is capable of using the resulting abstraction hierarchies to display the execution at various abstraction levels, allowing a user to navigate the abstraction hierarchy. We are currently working on adapting the existing tools to PVM and are studying ways to improve them.

References

[1] James C. Browne, Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual programming and debugging for parallel computing. *IEEE Parallel & Distributed Technology*, 3(1):75–83, 1995.

[2] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[3] J. A. Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. Technical report, Computer Science & Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 1995.

[4] Thomas Kunz. Reverse engineering distributed applications: an event abstraction tool. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):303–323, September 1994.

[5] Thomas Kunz. Visualizing abstract events. In *Proceedings of the 1994 CAS Conference*, pages 334–343, Toronto, Ontario, Canada, November 1994. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

[6] Thomas Kunz and James P. Black. Abstract debugging of distributed applications. In Karsten M. Decker and René M. Rehmann, editors, *Proceedings of the IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 353 – 358. Birkhäuser Verlag, Basel, Switzerland, July 1994. ISBN 3-7643-5090-3.

[7] Thomas Kunz and James P. Black. Understanding the behaviour of distributed applications through reverse-engineering. *Distributed Systems Engineering Journal*, 1(6):345–353, December 1994.

[8] Thomas Kunz and James P. Black. Using automatic process clustering for design recovery and distributed debugging, 1995. To appear in IEEE Transactions on Software Engineering.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, September 1994.

[11] David J. Taylor. A prototype debugger for Hermes. In *Proceedings of the 1992 CAS Conference, Volume I*, pages 29–42, Toronto, Ont., Canada, November 1992. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

[12] David J. Taylor. The use of process clustering in distributed–system event displays. In *Proceedings of the 1993 CAS Conference*, pages 505–512, Toronto, Ont., Canada, October 1993. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

[13] David J. Taylor and Michael H. Coffin. Integrating real-time and partial-order information in event-data displays. In *Proceedings of the 1994 CAS Conference*, pages 157–165, Toronto, Ontario, Canada, November 1994. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

[14] David J. Taylor, Thomas Kunz, and James P. Black. Achieving target-system independence in event visualisation. Submitted for publication, CASCON'95, 17pp in ms.