Monitoring PVM Programs Using the DAMS Approach

José C. Cunha and Vitor Duarte

Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia
Departamento de Informática *
2825 Monte de Caparica, Portugal
{jcc,vad}@di.fct.unl.pt

Abstract. Monitoring tools are fundamental components of a development environment as they provide basic support for performance evaluation, debugging, and program visualization. We describe our experiments with several monitoring tools for PVM, namely XPVM, developed at ORNL. Tape/PVM, developed at IMAG Lab, and DAMS, developed at UNL. These tools are compared and their use is described to support instrumentation and monitoring of a high level distributed language, PVM-Prolog, an extension to Prolog that provides an interface to PVM. This language is being used for the implementation of multi-agent systems, and it provides support for heterogeneous programs, built from C and Prolog components that communicate using PVM.

1 Introduction

In order to support the monitoring of parallel and distributed applications many tools have been developed in the recent past for distinct platforms and programming environments. In many cases PVM [2] is being used as the basic programming layer that provides an adequate degree of portability, architecture and operating system independence, and rapid prototyping. Such increased use was accompanied by efforts to provide monitoring support for PVM programs. Among the existing tools there are a large diversity of supported functionalities, often offered in a very monolithic way. Basic monitoring support such as code instrumentation and event generation, buffering, transfer, storage, and presentation, is tightly coupled with some given mode of operation, and it is not easy or possible to adapt or modify. For example, XPVM [5] includes built-in support for execution tracing, a PVM control console, and a display tool.

In this paper we discuss an alternative approach, which is based on a software architecture, called DAMS [3], that provides an infrastructure for the implementation of monitoring, debugging, and resource management that can be adapted according to user and application requirements. We describe our experimentation with the above approaches that we have used to instrument and monitor

* Thanks to B. Moscão, J. Vieira, D. Pereira. To EU COPERNICUS SEPP(CIPA-C193-0251), HPCTI(CP-93-5383), the Portuguese CIENCIA, PRAXIS XXI PRO-LOPPE and SETNA-ParComp, and DEC EERP PADIPRO(P-005).
programs in the PVM-Prolog language [7]. This allowed us to compare those tools in a practical setting and exercise important aspects such as information gathering, buffering, transfer, processing, and archiving, as well as information analysis, interpretation, and visualization.

The paper is organized as follows. In section 2 we present an overview of the main requirements for a monitoring tool. In Section 3 we describe the DAMS approach. Section 4 discusses our experimentation with the monitoring of PVM-Prolog programs, using XPVM, Tape/PVM, and DAMS, and compare such experiments. In Section 5 we comment related work, and finally we present some conclusions.

2 PVM Monitoring Tools

The goal of the monitoring activity is to gather runtime information about a target computation. This information may be used by tools at distinct levels of a computing system and for very different purposes such as program visualization, program testing and debugging, and performance evaluation.

Event Identification and Generation. Ideally, one would like to have a high-level user interface for the specification of the events to be monitored. This depends on the type of instrumentation that is applied into the target program (into the source code, into the library code, into the object code, or at the kernel level code). A significant example of an advanced tool that supports automatic probe insertion and dynamic instrumentation of parallel programs, including support for PVM applications, is Paradyn [8]. In XPVM [5] the instrumentation of the PVM library is supported so that the only requirement is for the user to select the PVM calls that should be monitored. Tools like Tape/PVM and PGPVM are also based on the instrumentation of the PVM library and allow to trace specific classes of PVM calls or specific program source modules. Such tools additionally support the specification of user-defined events, but this must be explicitly programmed by the user. PVaniM [10] also supports on-line monitoring based in periodic sampling of some values. The sampling rate is defined by the user as a compromise between the update rate of the visualizations and the application perturbation.

Event Buffering, Transfer, and Processing. Local event buffering allows to maintain the collected events in the local memory of each node, so that the monitoring communication overhead can be kept as low as possible. Possibly, event transfer to a central monitoring and storage node can take place in the end of the execution. However, there is the problem of local memory overflow so periodic flushing of events may become necessary during runtime. The amount of collected information can also be restricted to user-defined events. Concerning the user, an operation mode supporting on-line event transfer may be required to allow further event interpretation and visualization to proceed during runtime.
In XPVM, the frequent event transfer to the central XPVM daemon using PVM messages is related to the typical use of XPVM for on-line program visualization. Alternatively, XPVM allows the user to select a post-mortem visualization mode but this doesn’t disable the on-line event transfer and visualization. In Tape/PVM and PGPVM, post-mortem event transfer is the only option. Besides sampling, PVaniM event generation can also rely on tracing. Such options are under user control, as well as the specification of off-line or on-line event processing and visualization.

**Event Storage and Presentation.** Event storage in a central monitoring node is required in order to support further event processing, analysis and interpretation. Some related issues concern the formats of the event traces, their efficient storage, manipulation and access by other tools, and the management of causally consistent event orderings [11]. In XPVM, event storage in a trace file is performed in a meta-format (SDDF) allowing flexibility in event processing. The events are reordered if necessary to preserve the causality between send and receive events. In Tape/PVM, event storage takes place in the end of the execution, in a specific format. There is a monitoring starting phase that performs a first physical clock synchronization, allowing more “consistent” timestamp information on the event records, although it can take some time. An upper limit can be imposed by the user. An ending phase performs a final adjustment of the timestamps. By relying on off-line event transfer and storage, and this type of synchronization, Tape/PVM can compensate the intrusion effect. In PVaniM, a separate command reorders the events in the end to preserve the causality in the final trace.

### 2.1 Flexibility of the Monitoring Architecture

A monitoring system should have a flexible internal architecture, allowing easy configuration, and adaptation to specific low-level platforms. Concerning PVM monitoring tools, hardware and operating system heterogeneity are typically supported, even if most of the tools depend on the specific PVM version being used. Additionally, concerning easy configuration and tailoring to the application needs, most of the tools are not so flexible because their architectures are more or less based on the PVM internal mechanisms. Also, the user sees them as monolithic blocks with a closed set of functionalities. For example XPVM provides built-in PVM control console, trace execution, and on-line display. There are few possibilities of changing its internal event processing strategies.

Similar considerations would apply to the other mentioned tools. Their re-configuration to meet new functionalities would require great knowledge and internal changes to their architectures.

### 2.2 Tool Interfacing and Integration

Efforts on the definition of more uniform and standard monitoring interfaces are essential towards the successful support of environments that integrate many
high level tools. In order to handle the distinct and evolving requirements posed by such tools, a monitoring layer should provide well-defined interfaces, and support easy definition of new services. This goal motivates the search for an extensible set of basic mechanisms for event handling, including accessing, filtering, searching, which can be met for example by object-oriented interfaces to manipulate meta-formats like SDDF[1].

Concerning tool interfacing, several requirements should be satisfied:

1. On-line interaction of the monitoring tool and an external observer such as the detected events are passed on to the observer, according to several possible modes of operation. On-line interaction between an external controller or observer such as a debugger, and the target computation.

2. Support of dynamic attachment and detachment of observer and controller tools. Support of multiple tools, acting upon the same ongoing computation, and requiring coordination and synchronization.

Concerning the PVM monitoring tools under analysis, namely XPVM, Tape/PVM, PGPVM, PVaniM, tool interaction can only be achieved through the trace file, so this always implies an off-line processing and interpretation of events. None of the above requirements is satisfied.

3 The DAMS

The DAMS system (Distributed Applications Monitoring System) supports the monitoring and control of applications consisting of distributed Target Processes. It has the software architecture shown in Figure 1.

![DAMS Architecture](image)

**Fig. 1. DAMS Architecture**

DAMS is organized as a set of Local Manager Processes (LM), one on each physical node. They supervise the local processes and the communication with
the central Service Manager Process (SM). The SM process does not directly handle the application commands. These commands are forwarded to Service Modules that are responsible for their interpretation, according to each service (e.g. debugging, profiling or resource management). On each node, Local Managers act as intermediate between the SM and the Driver processes, by passing commands to the Drivers and sending the replies back to the SM. The Drivers apply the client commands to each Target Process.

DAMS provides well-defined interfaces between the client tools, the SM, the LM processes, and the Drivers, in the form of an internal message-based protocol which is architecture and system independent. It can be mapped onto different underlying communication platforms such as PVM, MPI or TCP/IP sockets. Each DAMS implementation assures there is no interference between the internal DAMS messages and the messages exchanged by the target processes.

DAMS includes a basic built-in set of commands so that the SM can launch, interact and destroy configurations of LM and Driver processes, and the Drivers can be dynamically attached to target processes. Asynchronous event notification is also supported so that each client tool can define an handler to be invoked on event occurrence. This can be used for tool synchronization e.g. involving a debugger and a visualization tool. DAMS does not include any other built-in services. Each service must be explicitly added by specifying a pair (Service Module, Driver). Actually, each Service Module contains the functions defining a specific service, and this code is linked with the Service Manager on system configuration. During execution, each existing service can be accessed by the client tools through an interface library that passes the client requests to the corresponding SM. Multiple concurrent client tools can coexist in a DAMS configuration. The current prototype runs on top of PVM, on a heterogeneous Ethernet LAN with Linux/PC’s, and a FFDI-based cluster of OSF/1 DEC Alpha processors. It allows the exploitation of the Alpha Ethernet links for the communications between DAMS managers, at the same time as the target application can use the FDDI connections. This can contribute to a reduction in the monitoring intrusion.

4 Monitoring PVM-Prolog Programs

PVM-Prolog [7] is a distributed extension to Prolog that provides a complete interface to PVM. The interface was already used to implement an agent-based language [9]. Reasoning models based on distributed agents are also being developed for diagnosis applications, and this experimentation requires a flexible development and execution environment. Besides supporting rapid prototype development, PVM-Prolog allows to bridge the gap between such high level abstractions and actual PVM execution:

**PVM Interface.** This is based on a set of built-in Prolog predicates for the spawning of independent Prolog evaluators (as PVM tasks), in local and remote processors. This includes the passing of top goals to such tasks and PVM-like communication of Prolog terms.
**Language heterogeneity.** A lower level set of Prolog predicates gives access to all PVM functions, namely to spawn PVM tasks that can execute Prolog or any kind of executable files. This also supports the communication between PVM-Prolog and C/PVM tasks, by packing, passing and converting Prolog terms to C structures.

**Multi-threading.** A multi-threaded model supports internal concurrency with user-level threads. Each thread solves an individual goal within a PVM-Prolog task. Communication between threads is supported by shared queues using specific predicates.

User-defined Prolog and thread predicates, and PVM-like predicates are supported by C code. The latter require argument conversion and passing to the C/PVM functions. Some predicates for spawning and communication have distinct parameters than the PVM functions. For example, there is a predicate to spawn a Prolog task and pass a Prolog goal as an argument, and the implementation must launch the Prolog task as an executable file, and pass the PVM arguments.

Several experiments were performed in order to support monitoring of PVM-Prolog programs. One goal was to trace events related to user-defined Prolog predicates; PVM predicates; and thread predicates. Another goal was to access and visualize the trace by using existing off-line and on-line visualization packages at the same abstraction level as defined by the application and not only at the PVM level.

### 4.1 Using Some PVM Monitoring Tools

**XPVM.** As the supported instrumentation is present in PVM there is no need to change the PVM-Prolog system or the application. The user selects PVM calls for monitoring through the XPVM GUI. Even if the user is only using the PVM-Prolog level, one must know about PVM so to understand the relation between the PVM calls presented by the XPVM and the PVM related predicates. Then the application must be launched from the XPVM console. One cannot attach XPVM to already running applications, without using the GUI. As XPVM runs synchronously with the application it introduces great perturbation but allows on-line visualization. In the end the user can make a post-mortem analysis.

**Tape/PVM.** We had to build an instrumented PVM-Prolog version. Code for the starting/ending phases of the Tape/PVM monitoring was added. A rewrite of the C code of each PVM-Prolog predicate was required, replacing each PVM call with its instrumented version as supplied by the Tape/PVM library. New event descriptions, corresponding to PVM-Prolog user-defined and thread predicates can be inserted in the trace file using a specific Tape function in specified points of the PVM-Prolog code. Selective monitoring of certain classes of PVM calls and modules of the source code (C) can be specified by the user. Interaction with other tools is only possible through the final trace file. An existing filter has been used to obtain a NPICL[12] format enabling visualization using Paragraph[4].
A Monitoring Service in DAMS. Based on the ability to include new Service Modules we implemented a monitoring service on the DAMS that is supported by a “Service Module” (SM), a “Driver” in each machine and a user console for controlling the service. A library that is linked with each target process supports the interaction with the driver, and the instrumentation of the application code.

The following components were implemented:

The SM manages the monitoring service, including functions to start the drivers in all the machines, detecting the end of the monitoring phase and collecting the traces from each driver;
The Driver stores the trace data generated by the instrumentation library in the local processes. The information is buffered locally until requested by the “Service Module”.
The Instrumentation library interacts with the local driver to connect the target process to the driver, to collect the event descriptions and to close the connection.
The Console allows the user to launch the monitoring system and its components in all the machines, and provides status information. During runtime, the user can request the already collected trace for each individual machine or process.

The PVM-Prolog predicates were instrumented to generate events corresponding to the start and end of each predicate call. The event description includes time-stamp, process identifier, identification of the call and information on the call parameters and result. The current monitoring service module asks for the trace in the end or by user request. Automatic on-line transfer is easily achieved. A conversion program enabled the generation of a NPLICL trace obtaining a global trace preserving a causality consistent event ordering, and allowing the use of ParaGraph for post-mortem program visualization.

5 Related Work

There are other monitoring systems for PVM that could have been tested. Some have a similar architecture to the Tape/PVM. They allow the monitoring of the PVM calls and use some traditional techniques for reducing the perturbation. Other systems, as Paradyn [8], represent a pioneering effort to separate the interfacing services and their implementation, as well as providing well-defined interfaces between separate modules. The On-line Monitoring Interface Specification (OMIS) [6] is another effort to provide a built-in standard interface. Its goal is to allow faster tool development and portability by decoupling the tools and the monitor. OMIS also allows adding new library services.

DAMS follows a more basic approach as it just provides the infrastructure and the basic mechanisms. Monitoring and debugging services can be replaced or modified freely, in part or all, according to the user and tool needs. Additionally it eases the implementation of these services in distinct platforms including heterogenous ones.
6  Conclusions and Ongoing Work

We briefly compared some PVM monitoring tools and described their use to support the monitoring of a distributed programming language. We have presented the main motivation to the DAMS approach, and showed how it was used to implement a monitoring service for that language. Due to its neutrality concerning both the client tools and the target processes, DAMS provides a flexible and easily extensible environment. It supports on-line interaction between the target application and the tools and dynamic attachment of multiple concurrent tools. DAMS is an ongoing project. Its open organization is allowing us to increase its functionalities in many directions. It has been used to support a distributed process-level debugging service [3], and it is being extended to support a thread-level debugger, and a thread visualization service. DAMS supports the monitoring and control of heterogeneous applications, with PVM and MPI components. A distributed debugger for PVM-Prolog is also being developed as a DAMS service.

References