On-Line Monitoring Support in PVM and MPI ⋆

Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Informatik, Technische Universität München, D-80290 München, Germany
email: wismuell@in.tum.de

Abstract. PVM and MPI have often been compared regarding their functionality and performance from the users' point of view. In this paper, however, we will present some comparison from a tool developer's point of view. We will show how a supporting infrastructure – the OMIS compliant monitoring system OCM – can be implemented for both PVM and MPI. Thus, we will put light onto those features of PVM and MPI that support the implementation of on-line tools.

1 Introduction

In the community of parallel computer users, there is a broad agreement that there should be tool environments that are uniform across different hardware platforms and programming libraries [SMP95]. A closer analysis shows that the fact that we usually don't have such environments is mainly caused by the tools' *monitoring systems*, which are responsible for observing and possibly manipulating the run-time behavior of an application program (called the *target program* or *target application*). While monitoring systems for off-line tools need only interact with the parallel programming library and therefore are portable across hardware platforms, those supporting on-line tools also have to interact with operating system and hardware. Thus, these systems are typically not portable.

At LRR-TUM, we tried to solve this problem by defining a common interface for on-line monitoring systems that is independent of the supported hardware and parallel programming library and is usable for a large variety of different on-line tools. The OMIS project (On-line Monitoring Interface Specification) has shown that it is possible to identify a large set of services that are independent of the target platform, so we only need small, well-defined extensions to allow for different programming models etc.

We achieved this by using an abstract object based model of a parallel system. In this model, the system basically consists of a set of compute nodes (which may have multiple processors) and a set of processes distributed across these nodes. In addition, processes may consist of several threads. In message passing systems we have two additional objects, namely messages and message queues. This abstract model allows to invoke most monitoring services without having

⋆ This work is partly funded by *Deutsche Forschungsgemeinschaft*, Special Research Grant SFB 342, Subproject A1.

to know whether we actually have a PVM process on a SUN workstation node, or an MPI process on a CRAY. Although the implementation of these services might differ, the interface is the same, so tools do not need to be adapted. Thus, a set of OMIS-based tools is available for a new platform as soon as one monitoring system is available.

Tools interact with the monitoring system by specifying requests, which consist of an optional event definition and a list of actions. The action list is executed whenever the monitoring system detects the specified event. For example, the following request defines a simple breakpoint, which stops each target process when it reaches address `0x1234`:

```
thread_reached_address([], 0x1234): thread_stop([$proc])
```

We can not go further into the details of OMIS here; the interested reader is referred to [LWSB97] and [LW97].

In addition to specifying the interface, we implemented the OMIS compliant monitoring system OCM [WTL98] for PVM on networks of workstations, which is currently being ported to MPI. In the rest of this paper, we will first present the coarse structure of the OCM. We will then focus on the necessary interactions between the OCM and the programming library, i.e. PVM or MPI. In this way, we will put light onto those features of PVM and MPI that support the implementation of on-line tools.

2 The OMIS Compliant Monitoring System OCM

As the OCM must monitor a distributed system, it must itself be distributed, i.e. we need local monitoring components on each node. We decided to use one additional process per node, which enables the monitoring system to react on events, independently of the state of the target program. Operating system support for debugging also requires the use of an additional process.

Because each monitor process can operate on its local node only, but OMIS allows tools to issue requests relating to multiple nodes, we need another component, called Node Distribution Unit (NDU), that analyzes each request and splits it into pieces that can be processed locally [WTL98]. The NDU also assembles the partial results and returns them to the tool. As shown in Fig. 1, the NDU is currently implemented as a central process, which communicates with the tools and the local monitor processes via message passing.

The communication uses buffered send and interrupt-driven receive operations. We need interrupt-driven receives (or comparable concepts such as Active Messages) to avoid blocking of the monitor processes. It would result in the inability to timely react on occurrences of other important events. A polling scheme is not feasible, as it consumes too much CPU time.

3 Interactions Between the OCM and PVM/MPI

In the following, we will identify the interactions necessary between the OCM and the PVM [GBD+94] and MPI [For95,For97] environments. Although we

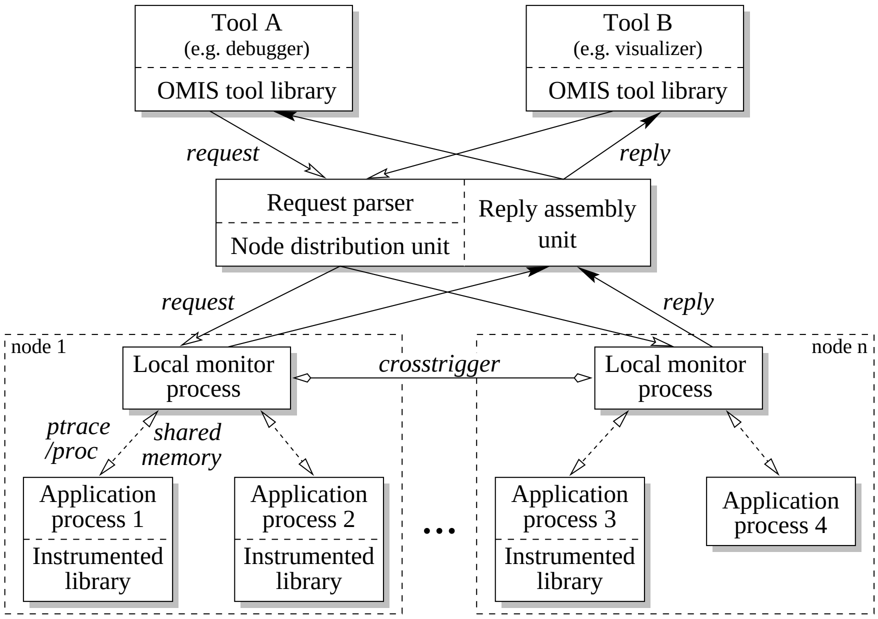

Fig. 1. Coarse Structure of the OCM

will only talk about the OCM, the results apply to on-line monitoring systems in general. Interaction between the OCM and the programming library[1] is mainly necessary for three activities:

1. for start-up and initialization,
2. for acquiring the state of the target program and the programming library,
3. for the detection of events related to the programming library.

In addition, the OCM may be able to just *use* the programming library, e.g. for its internal communication.

Since it turns out that the MPI specification does not support these interactions sufficiently, we have to use implementation specific interfaces. In the following descriptions we will refer to `mpich` version 1.1 [GLDS96].

3.1 Start-up and Initialization

When an on-line tool is started, we have to distinguish between two different scenarios: (1) the tool starts the target program (via the OCM), or (2) the tool (and the OCM) connects to an already running program.

[1] In the following discussions, the term “programming library” also includes the accompanying run-time environment, e.g. the PVM daemons.

Starting an application. In principle, starting the target program can be achieved by calling `pvm_spawn` or `MPI_Comm_spawn_multiple` (MPI-2 only) from within the OCM. However, this leads to a couple of possible problems: First, the OCM itself must be a PVM/MPI application, which may be undesirable[2]. Second, the behavior of the target application may change, since the processes now have a parent process within PVM or MPI, which may not be the case with the normal startup. For MPI-1, this kind of startup is not possible at all, since there is no dynamic process creation.

Thus, in the OCM for PVM, we start the target program the same way as a UNIX shell does, i.e. a monitor process `fork`s a child process that `exec`s the proper executable file. This process then eventually registers as a PVM task and dynamically creates other PVM tasks (see Section 3.3).

For MPI, establishing the connection between a monitoring system and its target application must be integrated into the `mpirun` program. In `mpich`, `mpirun` already contains an option for starting the `totalview` debugger [Dol]. This option can be used for the OCM, too, as it allows the monitoring system to `fork/exec` one process (the master process). `mpirun` then starts all other (slave) processes in a way that blocks their execution in `MPI_Init` until the monitoring system has achieved control over them.

Connecting to an application. The situation after the start-up is very similar to the one, where the OCM must attach to a running application. We will therefore not distinguish between these cases in the following discussion. The most important information now needed by the OCM is, which components (i.e. nodes and processes) must be monitored. Since we should not force the user of a tool to specify this information in detail, it must be retrieved from the programming library. The minimum information required consists of:

1. A list of all nodes that can be used by the target application. For workstation nodes, the minimum information needed is their host name.
2. A list of all processes in the target application. For each process, the minimum information needed is: its processing node, its local UNIX PID, the full (local) path to its executable file, and its global ID assigned by the programming library (e.g. the PVM TID or the rank in `MPI_COMM_WORLD`).

Any further information can be acquired via operating system services when host name and PID are known. Unfortunately, the path to the executable cannot be safely determined in this way, but it is needed e.g. to access symbol tables. The process ID assigned by the programming library is not really necessary for monitoring, but is important for the user of a tool in order to identify processes.

With PVM, it is trivial to get this information, provided that at least one monitor process is a PVM task. In this case, it can just call `pvm_config` to get the necessary information on the nodes, and `pvm_tasks` for the process infor-

[2] We will show some of the resulting problems later.

mation[3]. However, we can see that precisely because `pvm_tasks` is available, a monitoring system should not contain any PVM tasks: If it does, it might modify the behavior of the target application, as it may depend on the result of `pvm_tasks`. In addition, the PVM TIDs may change w.r.t. an unmonitored run. This can be a serious problem when e.g. TIDs have been stored in a log file and a user tries to debug a suspicious process in a second (monitored) run.

The MPI standard does not specify functions to retrieve the information listed above. As a consequence, it is impossible to build on-line tools for MPI that can be used with all MPI implementations. However, `mpich` provides a special interface allowing to acquire the configuration data in the following way:

1. When invoked with the debugging option, `mpirun` starts the master process under control of the monitoring system. The monitor now must place a breakpoint on a special function (`MPIR_Breakpoint`) and start the process.
2. The master process calls `MPI_Init`, which creates all other processes. For each created process, `MPI_Init` adds a data block to a process list, containing the name of the host where the process has been started, its local PID, and the path to its executable. In addition, detailed information on MPI communicators is preserved.
3. When the start-up has completed, the master process calls `MPIR_Breakpoint` and thus triggers the breakpoint set in step 1.
4. Using symbol table information of the master process, the monitoring system can now read the process and communicator lists. However, it is still not trivial to obtain a mapping between processes and their rank in `MPI_COMM_WORLD`.

3.2 Acquisition of State Information

There are two different kinds of state information stored in the programming library that are of interest for on-line tools:

1. The target application's configuration, i.e. nodes and processes. We have already discussed this above.
2. The state of the communication subsystem, i.e. message queues, blocked processes etc.

Neither PVM nor MPI/`mpich` offers sufficient support for the last item, so the only solution is to extract this information from internal and undocumented data structures in the programming library. Of course, this is not a satisfactory solution, since in this way the monitoring system becomes extremely sensitive to changes in the programming library's implementation.

3.3 Event Detection

Library Calls. Most events related to the programming library can be detected by monitoring the entry into and the exit from library functions. A common

[3] There is actually a slight difficulty in getting the full path names, however, it is beyond the scope of this paper to discuss the solution.

technique is to insert *hooks*, i.e. small pieces of monitoring code that are executed immediately before and after the actual library function. A major concern is how to insert hooks without having to recompile the programming library or the target application.

In MPI, an elegant method is used to achieve this: The MPI standard requires that each MPI function provides two entry points. One of these entry points has the usual name starting with `MPI_`, the other one uses the prefix `PMPI_`. With a few additional requirements, which are outlined in the standard, this so-called profiling interface allows to easily incorporate hooks by just linking the target application against an additional library. This library contains functions starting with `MPI_`, which perform some monitoring work and call the original MPI routines via their second entry point `PMPI_`. The application code now calls these new functions instead of the original ones.

In PVM, no such mechanism exists. Thus, we have to use *binary wrapping* [CM92] to achieve the same effect. The basic idea is to create differently named entry points in an existing binary version of a library by changing the names in its linker symbol table. We can then proceed as explained above.

Process creation. This event that cannot be detected in the way outlined above. By instrumenting e.g. `pvm_spawn`, we can only detect that (1) a process wants to create other processes (hook at the function's beginning), or (2) it successfully created other processes (hook at the end). However, we cannot catch the new processes: at point (1), they are not yet existent, while at point (2) they already started execution and may even have terminated. However, being able to monitor new processes from their very beginning is essential for debugging.

PVM therefore defines the *tasker* interface [BDG+95]. A tasker is a PVM task responsible for starting new processes on its local node. Thus, it can prepare the new process in a way that keeps it from running until the monitoring system is ready to control it. Although the tasker interface offers the only way to immediately obtain control over new processes without having to modify PVM, its use leads into some trouble: First, all monitor processes now have to be PVM tasks, since we need a tasker on every node. As stated in Section 3.1, this may influence the behavior of the target application. Second, when a tasker terminates, its child processes are killed. This means that although we can dynamically attach the OCM to a target application, we cannot simply detach it again. Finally, when the tasker is informed by the PVM system, the new task does not have a TID yet, which leads to an assignment problem inside the OCM.

For `mpich`, process creation doesn't seem to be an issue at all, as there is no dynamic process creation in MPI-1. However, this is not true. During the start-up phase, the monitoring system gets immediate control over the master process only (see Section 3.1). From the viewpoint of the monitoring system, the slave processes are created dynamically. In order to prevent them from running too far, `mpirun` starts them with a special option that results in an endless loop inside `MPI_Init`. The monitoring system can free a process again by changing a variable inside that process. A drawback of this method is that it is impossible to monitor activities prior to the call to MPI_Init. Thus, applications should call MPI_Init as the very first statement in their main function[4].

3.4 Using PVM/MPI Within the OCM

In principle, PVM and MPI may also be used to implement the communication inside the OCM. Unfortunately, neither of the two offers the interrupt-driven (or handler-driven) receives required by the OCM. However, if communication is implemented using TCP sockets, the sockets can be manipulated, such that a signal handler is called when a message arrives. The OCM then can call available non-blocking receive functions to read the message. Accessing the socket is easy in PVM, since there is a library call (pvm_getfds) returning its file descriptor. In MPI, there is no such call.

There is another stringent reason why the OCM can't use MPI for its communication: When a tool is started, it must be able to connect to the OCM, even when the OCM is already running. However, even with MPI-2, it is impossible for an independent process to join an already running MPI application.

3.5 Consequences for the OCM

As a consequence of the above discussions, the OCM for PVM *must* be a PVM application (due to the use of pvm_config, pvm_tasks, and the tasker interface) and also *can* be a PVM application (since new processes can connect to a running PVM application). Despite the disadvantages already mentioned, the OCM for PVM is implemented this way.

On the other hand, for the MPI version of the OCM there is neither a necessity nor a possibility to use MPI for internal purposes. All the information on the target application can be retrieved without the need for the monitors being MPI processes. MPI also does not offer some essential features needed for the communication within the OCM. This is not astonishing, since MPI has been designed purely for parallel applications, while the OCM is a typical distributed application with accordingly different requirements.

We have therefore designed a tiny communication interface specifically adapted to the needs of the OCM. It has been implemented both on top of PVM, and (for the MPI version of the OCM) by using plain TCP/IP socket connections.

4 Conclusion

As a conclusion, we can state that both PVM and mpich offer sufficient support for building on-line monitoring systems and tools. In a direct comparison, the PVM support wins, however, we also have shown some problematic details. mpich allows to retrieve all necessary information, too, but the interface is more complex and a bit cumbersome. On the other hand, MPI and mpich support features that are missing in PVM, e.g. the profiling interface.

[4] However, in C++ programs constructors may be invoked even before main.

In summary, the experience with the OCM has shown that parallel programming libraries should at least provide the following features to support on-line monitoring: A list of nodes (with host names), a list of processes (with node, PID, global ID, path to executable), a means to intercept process creation, and a second entry point for each library function. In the ideal case, a monitoring system should be able to use this functionality, even if it is *not* an application of this programming library.

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*, 9(2), 1995.

[CM92] J. Cargille and B. P. Miller. Binary Wrapping: A Technique for Instrumenting Object Code. *ACM SIGPLAN Notices*, 27(6):17-18, June 1992.

[Dol] Dolphin Interconnect Solutions, Inc. TotalView Overview. WWW page. `http://www.dolphinics.com/tw/tvover.htm`.

[For95] The MPI Forum. MPI: A Message-Passing Interface Standard, Version 1.1. Technical report, Univ. of Tennessee, Knoxville, TN, June 1995. `http://www.mpi-forum.org/docs/mpi-11.ps.Z`.

[For97] The MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, Univ. of Tennessee, Knoxville, TN, July 1997. `http://www.mpi-forum.org/docs/mpi-20.ps.Z`.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical report, Argonne National Lab, July 1996. `http://www.mcs.anl.gov/mpi/mpicharticle/paper.html`.

[LW97] T. Ludwig and R. Wismüller. OMIS 2.0 – A Universal Interface for Monitoring Systems. In M. Bubak et al., editors, *Proc. 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, pages 267-276, Crakow, Poland, November 1997. Springer Verlag. `http://www.in.tum.de/~wismuell/pub/europvm97b.ps.gz`.

[LWSB97] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. `http://wwwbode.informatik.tu-muenchen.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`.

[SMP95] T. Sterling, P. Messina, and J. Pool. Findings of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments. Technical report, Center of Excellence in Space Data and Information Sciences, NASA Goddard Space Flight Center, Greenbelt, Maryland, 1995. `http://cesdis.gsfc.nasa.gov/PAS2/findings.html`.

[WTL98] R. Wismüller, J. Trinitis, and T. Ludwig. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR, USA, August 1998. To appear.