OMIS-Compliant Monitoring System for MPI applications

Marian Bubak[a] and Włodzimierz Funika[b] and Radosław Gembarowski[c] and Roland Wismüller[d]

[a] (1)Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059, Kraków, Poland
(2)Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
bubak@uci.agh.edu.pl

[b]Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059, Kraków, Poland
funika@uci.agh.edu.pl

[c]Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059, Kraków, Poland
gembarow@icsr.agh.edu.pl

[d]LRR-TUM, Technische Universität München, D-80290 München, Germany
wismuell@in.tum.de

Abstract

The OCM is a distributed on-line monitoring system implemented according to the OMIS interface specification. The currently available implementation of the OCM was intended for programs using PVM. At present, there is a strong need to adapt the OCM to support MPI applications development due to the growing use of the MPI standard in parallel programming. The paper presents the issues needed to be solved when porting OCM to MPI, namely related to the mechanisms of the `mpich`-oriented start-up and MPI library calls profiling. While the existing internal OCM communication is preserved, adapting the OCM for MPI programs needs the addition of a number of new MPI specific services.

1 Introduction

MPI is a widely accepted standard for message passing programming. To support the development of MPI applications there have been designed a number of run-time tools [1]. Unfortunately, the on-line tools among them suffer from poor interoperability and portability. The On-line Monitoring Interface Specification (OMIS) [2] is aimed at relieving this situation. OMIS defines a standard interface to provide a unified access to a wide set of services for tool development. In OMIS, there are three categories of services which allow to retrieve information about the execution of a paralell program, to control the execution, and to detect specific events during execution. Other initiatives are undertaken to build distributed monitoring systems based on different assumptions, e.g., DAMS (distributed monitoring and control layer) [3] and PARMON (environment for monitoring of large clusters) [4].

The main feature of OMIS is its ability to support large number of tools on a wide range of platforms under different programming environments. This implies, e.g., that a debugging tool which can co-operate with a performance analyzer in a particular parallel environment can be easily transferred into another environment while preserving its ability of co-operation. Thus a high level of portability and interoperability, so crucial in run-time tools, is provided. The OMIS-Compliant Monitoring system (OCM) [5] is a monitoring system implemented in line with OMIS.

The OCM has originally been implemented for the PVM programming library and can be used to monitor PVM parallel programs only [6]. The growing use of MPI motivated our involvement in the porting and enhancement of the OCM's functionality so it could support the development of MPI applications as well.

The paper presents approaches to solve the problems referring to the `mpich`-oriented start-up mechanism [7] and monitoring of MPI library calls in the OCM. While the existing transport layer for the OCM internal communication is preserved, a number of new MPI specific services had to be added. A preliminary description of related concepts has been presented in [8].

2 A tour around the OCM

OMIS is intended to meet the goal of building a standardized interface between application development support tools and parallel programming environments. Such an interface must address the issues of versatility, portability, extensibility and independence of the underlying parallel environment. OMIS is not restricted to a single kind of tools, especially it supports both debuggers and performance analysers [9, 10]. This implies that a monitoring system compliant with OMIS needs to be able to control processes from the very beginning of their execution (crucial for debugging) as well as to monitor the parallel programming library calls efficiently (crucial for performance analysis).

Generally, OMIS defines:

1. three categories of monitoring-related services, as mentioned before: information services, manipulation services, and event services,
2. a basic set of services which are available on any platform,
3. an extension interface that allows to add new services supporting the operation of tools in particular environments,
4. a uniform string-based syntax for event-driven requests to the services defined,
5. data structures for service replies,
6. six C-level function calls that build the interface to the monitoring system.

As shown in Fig. 1, the OCM is a distributed monitoring environment with a local monitor process on each node of the target system. In addition, there is a central component (the Node Distribution Unit, NDU) responsible for distributing requests to the local monitors and

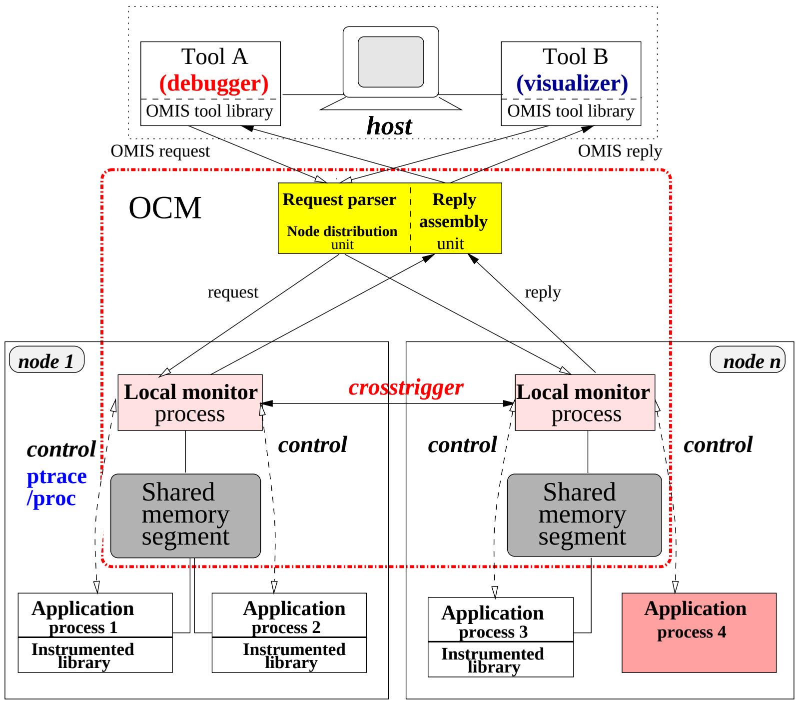

Figure 1: Structure of OCM-based monitoring

re-assembling their replies. The local monitor processes control the application processes via operating system interfaces (`ptrace` and `/proc`) and communicate with additional parts of the monitoring system contained in instrumented libraries, using a shared memory segment.

One of the main points in the design of the OCM concerns the way the user can handle both the execution of an application program with the OCM and the components of the proper OCM. First of all, the OCM usage should be as near to sequential on-line tools as possible, this implies that one can start a tool without an already running program with a command line like

```
tool <tool_parameters> <application_name> <application_parameters>
```

The tool then starts the OCM, which in turn starts the application. Another requirement is that a tool should support a simple 'rerun' of the application, without a need to start the tool anew. However, due to significant differences in PVM and MPI mechanisms [11], in MPI these general features may be realized in a way different from that for PVM, including a different sequence of start-ups as shown below. When porting the OCM to MPI, our main concern was to preserve the user interface designed for PVM as much as possible. This would ensure that the user could easily switch between the PVM and MPI applications.

For PVM, due to the "virtual machine" that exists independently of any application program, it is possible to realize two things: on one hand to start a monitor process on each host, on the other hand to create/rerun the application processes: the OCM just has to start the initial process, which will explicitly spawn all other processes. For MPI, there is no virtual machine – the hosts to be monitored are known only after an application has been started. Thus, the start-up must proceed in a reversed sequence: first, an application is created, then a monitor is started on each node which is used by the application. In order to support a consistent interface with the PVM version and to enable a restart of an application, a MPI specific service has been introduced which creates all processes of an MPI application. The first invocation of this service after the start-up of the OCM just continues the already existing application, while subsequent invocations restart it using the same host configuration.

The considerations above imply that the bulk of activity about the modification of the OCM to support MPI applications will focus on the program (re)start and initialization mechanisms within the OCM, application and a tool as well as the ways of transferring the control and application-related data between them.

3 Modifications in the OCM to support the MPI applications

As stated above, the following mechanisms are to be taken into consideration when enhancing the OCM in order to support MPI applications development:

- starting-up MPI programs,
- initialization procedure of parallel programs,
- detection of MPI functions calls.

The first and most important thing an on-line monitoring system must know about an application to be monitored is:

▷ what are the hosts the application is running on?

▷ which processes on these hosts belong to the application?

The user should not be forced to specify this information explicitly. Instead it should be retrieved by the monitoring system automatically. The PVM version of the OCM could use existing PVM library calls that return this information. Such calls do not exist in MPI. So how can the OCM get the necessary data?

For MPI applications using `mpich`, the data concerning the application can be retrieved during its startup in the following way: When the application is started with a proper command line flag passed to `mpirun`, `mpirun` will start a single monitor process. Later the initial (master) process (with global rank 0) is created and controlled . When the master process executes the `MPI_init()` function, `mpich` creates all other processes and stores the relevant information in an internal data structure. When done, `MPI_init()` calls a special routine `MPIR_Breakpoint`, where the monitor may place a breakpoint to stop the master process. It can then read the necessary information from the following `mpich` variables, using the UNIX `ptrace`[1]:

- `MPIR_proctable` - pointer to an array of `MPIR_PROCDESC` structures, which are defined as:

[1] An alternative method to control the application's execution is to use `/proc` interface

```
typedef struct {
    char *host_name;        host name
    char *executable_name;  path of the program's executable
    int pid;                system process identifier
} MPIR_PROCDESC;
```

- `MPIR_proctable_size` - size of the array,

The processes created by the master process are stopped by the `totalview` interface to `mpich` [12] using an endless loop inside `MPI_Init`. So there is time to create the other monitor processes on each host used by the application, which then gain control over the applicaiton processes on their local host.

A number of problems stem from a sort of functional asymmetry between the local monitors. The initial (master) local monitor is must start up the MPI program and must distribute the data about the running program to the other local monitors and NDU. This implies that the first local monitor has to provide a larger functionality.

While the PVM oriented version of the OCM can access the application-related data via PVM library calls on each node, this is not possible with MPI, so the data must be explicitly distributed among nodes. In case `mpich` is used, the information retrieved from the master process is then distributed to all local monitor processes by NDU (Fig.2). This information is also accessible to tools via an MPI specific information service.

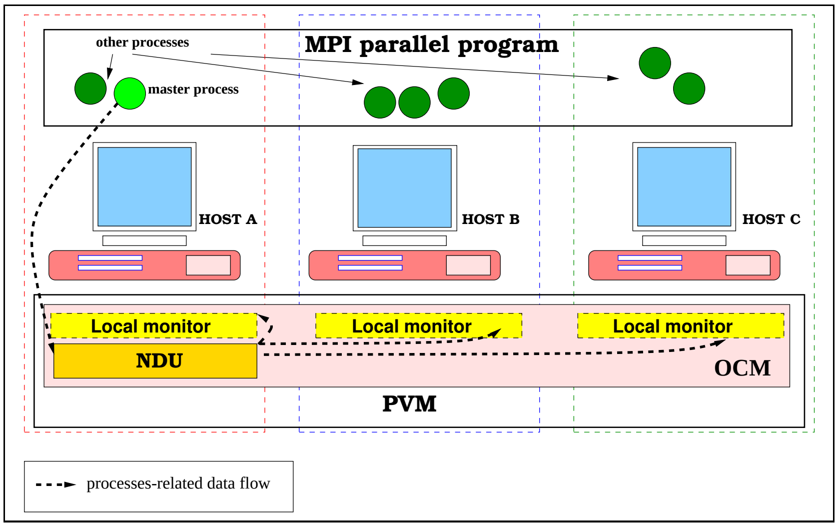

Figure 2: Distribution of application-related information

In practice, the data of the program to be monitored can be accessed via a `PI` prefixed file which is generated by `mpich` when starting the program with `mpirun` script or defined by the user (Fig. 3). This file being an element of the `p4` device serves also as a set of input arguments for the MPI program. The `PI` file comprises the data describing the paths to the program's executables, the architecture of the hosts, the number of processes on the hosts.

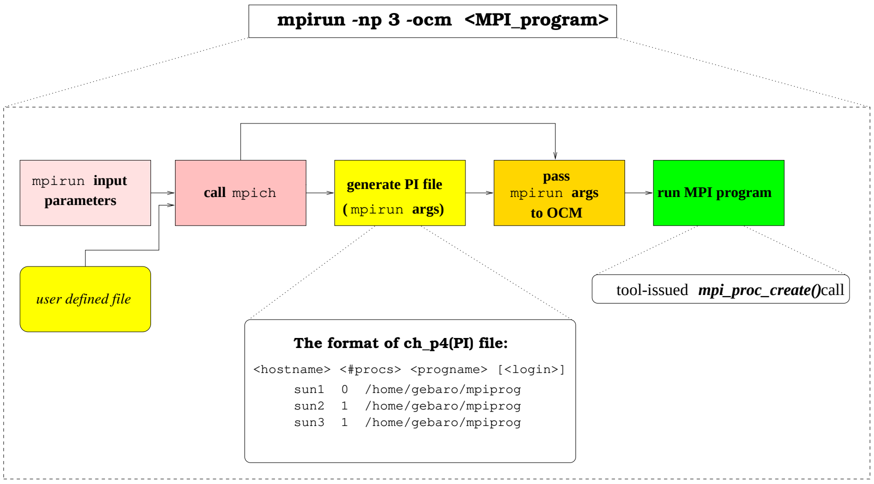

Figure 3: `mpirun` script execution

The process with number 0 is identified as the master process of the program. As the automatically-generated `PI` file and the user-defined `PI` file delivers the paths to the program's executables if the user starts another MPI program, there should be called another `mpirun` script.

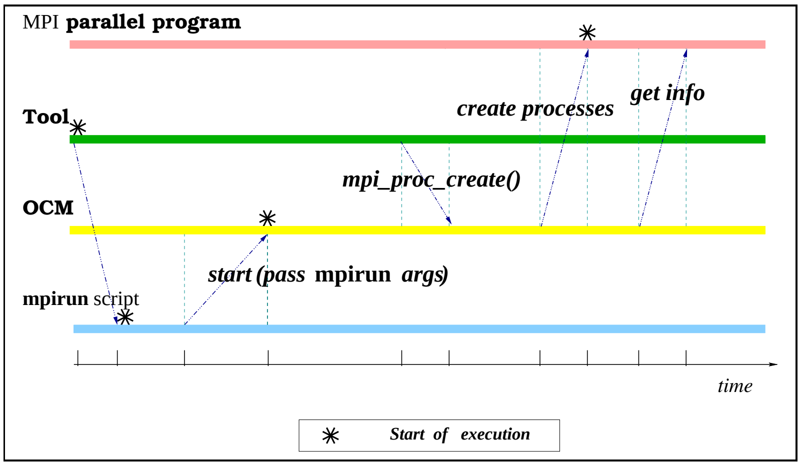

Figure 4: Time-lined relationships of tool with OCM and application

The time-lined relationships of the OCM – tool – MPI program evolve as shown in Fig. 4. The tool has to invoke the `mpirun` script with the '-ocm' option to start the OCM. The `mpirun` starts the OCM as a debugger process and passes the application's name and parameters to it. When the tool requests `mpi_proc_create()`, the OCM starts the application 'master' process and gets information about the application. The user can manipulate and control the execution of the MPI program with the tool which issues OMIS requests. The OCM plays the role of a server, while the tool is a client.

In order to support performance measurement of parallel programs using the MPI library, it is necessary that the OCM can detect the entry into and the exit from MPI functions. Adding adequate wrappers at the beginning and the end of the profiling version of the `MPI` functions allows to transfer the relevant data about entry into and exit from `MPI` functions to the `OCM`. This wrapping of MPI functions is supported by an instrumenting tool. From a list of functions for which wrappers are to be generated, the prototype declarations of these functions, and a list of wrapper definitions, it automatically creates the instrumented MPI library.

4 Concluding Remarks

The OCM designed to support the development of parallel applications has originally been implemented using PVM as the parallel programming library. While enhancing the OCM functionality to support MPI applications, a number of problems was to be coped with, namely, how to:

- start-up the application program
- realize an MPI specific information service
- distribute the information on the MPI application to the local monitors
- detect calls to MPI library functions

The experience gained when porting the OCM to MPI showed that the dependence of the OCM on the underlying PVM has significantly influenced the adaptation of the OCM mechanism to the new environment features in order to make the user interface similar to that for PVM applications. This implication is very important for the design of monitoring systems in the future.

Acknowledgements

The Authors are grateful Mr. Krzysztof Sowa for helpful comments and to Mr. Bartosz Baliś for his collaboration. This research was done within the German–Polish cooperation.

References

[1] Browne, S.: Cross-Platform Parallel Debbuging and Performance Tools. In: Alexandrov, V., Dongarra, J., (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 5th European PVM/MPI Users' Group Meeting, Liverpool, UK, September 7-9, 1998, Lecture Notes in Computer Science 1497, Springer, 1998, pp. 257-264.

[2] Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A.: OMIS – On-line Monitoring Interface Specification (Version 2.0). LRR-TUM Research Report Series, vol. 9, Shaker Verlag, Aachen, Germany, ISBN 3-8265-3035-7, 1997.

[3] Cunha, J., Lourenço, Vieira, J., Moscão, B., and Pereira, D.: A Framework to Support Parallel and Distributed Debbuging. In: Sloot, P., Bubak, M., Hertzberger, B., (eds.): Proc. Int. Conf. High Performance Computing and Networking, Amsterdam, April 21-23, 1998, 708-717, Lecture Notes in Computer Science 1401, Springer, 1998.

[4] Rajkumar, B., Krishna Mohan, K.M., Gopal, B.: PARMON: A Comprehensive Cluster Monitoring System. In: Proceedings of High Performance Computing on Hewlett-Packard Systems, Zurich, Switzerland, 298 – 311 ETH Zürich, 1998;
HPCC System Software PARMON Group: PARMON User Manual, C-DAC,1998
`http://cdacb.ernet.in`

[5] Wismüller, R., Trinitis, J., Ludwig, T.: A Universal Infrastructure for the Run-time Monitoring of Parallel and Distributed Applications. In: EuroPAR'98, Parallel Processing, Lecture Notes in Computer Science, vol. 1470, 173-180, Southampton, UK, September 1998, Springer 1998.

[6] Wismüller, R.: On-Line Monitoring Support in PVM and MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. EuroPVM/MPI'98, Lecture Notes in Computer Science, vol. 1497, 312-319, Liverpool, UK, September 1998, Springer, 1998.

[7] Gropp, W., Lusk, E.: User's Guide for `mpich`, a Portable Implemention of MPI. ANL/MCS-TM-ANL-96/6, 1996.

[8] Bubak, M., Funika, W., Gembarowski, R., Hodurek, P., and Wismüller, R.: Enhancing OCM to Support MPI Applications. In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): Proc. Int. Conf. High Performance Computing and Networking, Amsterdam, April 12-14, 1999, 1274-1277, Lecture Notes in Computer Science 1593, Springer, 1999.

[9] Bubak, M., Funika, W., Iskra, K., and Maruszewski, R.: OCM-Based Tools for Performance Monitoring of Message Passing Applications. In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): Proc. Int. Conf. High Performance Computing and Networking, Amsterdam, April 12-14, 1999, Lecture Notes in Computer Science 1593, 1270-1273, Springer, 1999.

[10] Bubak, M., Funika, W., Młynarczyk, G., Sowa, K., and Wismüller, R.: Symbol Table Management in an HPF Debugger. In Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): Proc. Int. Conf. High Performance Computing and Networking, Amsterdam, April 12-14, 1999, 1278-1281, Lecture Notes in Computer Science 1593, Springer, 1999.

[11] Gropp, W., Lusk, E.: Why are PVM and MPI So Different? In: Bubak, M., Dongarra, J., Wasniewski, J. (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 4th European PVM/MPI Users' Group Meeting, Cracow, Poland, November 3-5, 1997, Lecture Notes in Computer Science 1332, 3-10, Springer, 1997.

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