On-line Tool Support for Parallel Applications

Marian Bubak[1,2], Włodzimierz Funika[1], Bartosz Baliś[1],
and Roland Wismüller[3]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
[3] LRR-TUM – Technische Universität München, D-80290 München, Germany
{bubak,funika}@uci.agh.edu.pl, balis@icsr.agh.edu.pl, wismuell@in.tum.de
*p*hone: (+48 12) 617 39 64, *f*ax: (+48 12) 633 80 54 *p*hone: (+49 89) 289-28243

Abstract. The paper presents a motivation and issues concerning the development and adaptation of on-line tools to an OMIS Compliant Monitoring system (OCM). It covers enhancements needed both at low-level parts of the monitoring environment and at the user interface level in order to achieve full tool support for parallel applications. The tools' operation should be based on the monitoring of parallel applications developed with different programming models, offering a wide range of tools capable of cooperating with each other. The paper presents the recent development of the OCM-based environment of on-line tools and outlines perspectives for further research.
Keywords: parallel programming, monitoring systems, interoperability.

1 Introduction

For support of the development of parallel message-passing applications there have been designed a number of run-time tools [1]. These tools allow to observe and manipulate the behavior of an executing application and thus support performance analysis, visualization, debugging, etc. They are intended to provide needed support in all the phases of program development. During recent years, a large number of tools have emerged. Most of them follow the off-line trace based approach, where the gathered data can only be analysed after the application terminated, representative examples being Vampir, Pablo, and ParaGraph. The on-line tools are much fewer and, moreover, fewer are the tools which feature a well defined interface to a monitoring facility, what could provide prerequisites for interoperability. The on-line tools mainly provide a specialized tool functionality (e.g. ParaDyn/DynInst[10]) or support for particular parallel environments (e.g. TAU [13]).

Among the initiatives to define universal interfaces between tools and monitoring systems, there are DAMS[6], and DPCL[12]. However, DAMS does currently not address performance monitoring, while DPCL does not offer direct support for message passing environments like PVM [7] and MPI [11]. The *On-line Monitoring Interface Specification* (OMIS) [9] is aimed at building a tool environment which can be extended in the *supported parallel environments* dimension as well as in the *provided tools* dimension. OMIS defines a standard interface to provide a unified access to a wide set of services for tool development, which are divided into three categories: those for retrieving information about a program's execution, for controlling the execution, and for detecting specific events, in order to start proper actions.

Based on the compliance with the OMIS specification and its implementation, OCM[1], a performance analysis tool, PATOP is enabled to function in message passing programming environments, interoperate with other on-line monitoring based, OMIS-compliant tools like the DETOP debugger [16], etc. The OCM, originally developed to monitor PVM applications, has been enhanced to support development of MPI applications. The next natural step was to adapt the mentioned tools to the latter programming model. In this paper, we follow the issues concerning relevant changes to the tools, the OCM monitoring system, the enhancement of their functionality as well as provide an outlook at the interoperability of OCM-based tools.

2 OMIS as Basis for Building a Tool Environment

OMIS is intended to meet the goal of building a standardized interface between application development support tools and parallel programming environments, which addresses 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 analyzers [3, 4]. This implies that a monitoring system compliant with OMIS needs to be capable 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).

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*, for short *NDU*) responsible for distributing requests to the *local monitors* and 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.

At the beginning, the use of OMIS for parallel programming was rather constrained. On the one hand, an OMIS compliant monitoring system, the OCM was available to monitor PVM programs only, on the other hand, there already existed a number of powerful tools, most of which did not support parallel applications using PVM or other popular message passing mechanisms. The design we will report on below thus had two goals: to extend the applicability of the OCM to other parallel programming facilities, especially MPI, and to extend the OMIS-based tool environment, specifically w.r.t. performance analysis.

The ever growing use of MPI motivated us to port and enhance the OCM's functionality so it could also support the development of MPI applications. When

[1] OCM stands for the OMIS Compliant Monitoring system

porting, 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 PVM and MPI applications. 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 start-up of all of the OCM's components. The usage of OCM-based tools should be as near to sequential on-line tools as possible, this implies that one can start a tool without an already running program. Thus, the tool starts the OCM, which, in its turn, starts the application. As opposed to PVM, in MPI, which features the lack of virtual machine, the hosts to be monitored are known only after an application has been started. Thus, the start-up must proceed in a sequence reversed w.r.t. PVM: first, an application is created, then a monitor is started on each node which is used by the application. In order to keep the interface consistent with the PVM version and to enable a restart of an application, an MPI specific service into the OCM has been introduced, with which all the processes of an MPI application are created.

The most important data the monitoring system must know about an application to be monitored is: `which hosts` the application is running on, and `which processes` on these hosts belong to the application. So, *how the necessary data can be obtained?* For MPI applications using `mpich` [8], the data concerning the application can be retrieved during its startup in the following way: Once the application is started with a proper command line flag passed to `mpirun`, the latter will start a single monitor process. A procedure following the initial step is used to obtain information from an internal data structure with the UNIX `ptrace` system call or `proc` file system, when the processes created by the master process are stopped by an endless loop inside `MPI_Init()`. So, it is the first (master) local monitor that starts up the MPI program and distributes explicitly the information about the running program among the components of the OCM. The above information on the application gets also accessible for tools through an MPI specific information service provided by the OCM. For a detailed description of the start-up procedure, the reader is referred to [2].

For performance analysis goals, the OCM gathers information about MPI function calls via instrumentation of the parallel library, what consists in building a wrapper for each library function. To ease making wrapper modules, a kit of universal tools for automatic generation of the individual wrappers has been developed.

3 Operation of PATOP on Top of the OCM for MPI

In general, performance measurement tools for parallel applications are intended to help in finding and fixing bottlenecks by carrying out various measurements [10]. In order to provide *on-line* performance analysis facilities for the *de facto* message passing standards `PVM` [7] and `MPI` [11], an existing tool, PATOP [16] was used. Skipping details, (see [3]) we outline the functionality of the tool and the changes to the OCM and the tool, which are necessary to make them work together for PVM applications (very shortly) and next for MPI programs.

PATOP has originally been developed for the on-line performance analysis of applications running under PARIX on Parsytec systems. The tool provides a reasonably rich set of available performance measurements and display diagrams at various levels of detail. The integration of PATOP with the OCM involved an additional layer, ULIBS. In the case of PATOP, the task of ULIBS is to transform high-level measurement specifications from PATOP to event/action relations acceptable for the OCM and to transform back the results. Both the OCM and ULIBS are extended with modules specific to performance analysis, PAEXT and PERF, respectively.

3.1 Gathering Performance Data with the OCM

In order to be able to provide performance analysis of a parallel program, tools need to acquire information about the program such as the amount of time spent on waiting for a message from another process, or the message size. The OCM provides all needed support for obtaining such information. To make this data available to the performance analysis tools, there are provided three principal approaches: *direct communication* - immediate sending data on having detected an interesting event, *tracing* — storing information in a trace buffer whenever a relevant event occurs, *counters/integrators* - summarizing the most important information on each event in counters and integrators for `integer` and `float` values, respectively. The mechanism for the counters and integrators is provided by an extension to the OCM, PAEXT, in which two new types of OMIS objects are defined, *counters* and *integrators*.

When choosing a appropriate way of gathering performance data, it is indispensable to take into account the demand for a particular kind of performance data and the cost of gaining it, the most flexible of them but expensive being the direct communication, whereas the least intrusive but also much less informative being the counters/integrators approach.

The involvement of ULIBS, which had been developed prior to the OCM, as an intermediate abstraction level implied its modification to comply with the OMIS interface. Now, ULIBS provides the following functionality for tools: *asynchronous* requests and callbacks, configuration and current state *info* (nodes, processes), *starting/stopping* processes, *debugger-oriented* mechanisms like symbol tables, variable info, breakpoints, single stepping. Whereas the interface of ULIBS basically remained unchanged w.r.t. to the original lsversion, its implementation had to be completely redesigned for the use with the OCM.

3.2 Adaptation of the Tool Environment to MPI

Extending PATOP, which is in the top layer of the tool environment, with a capability to measure performance of MPI applications implied some indispensable extensions to the existing components of the environment. Fig. 1 shows the general architecture of the environment with MPI-specific parts and performance analysis modules exposed.

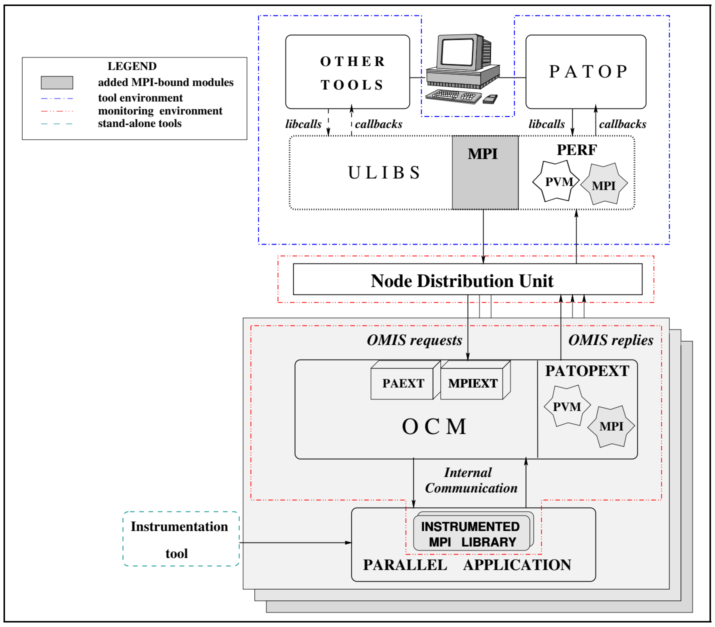

Fig. 1. Architecture of the monitoring environment

Adapting PATOP to MPI required some new modules to be added to the environment. The OCM was extended with a new MPI-bound extension, MPIEXT. The PATOPEXT module of the OCM was extended with an MPI-specific part. The instrumented MPI library is intended to be linked to a parallel application. A new module for MPI was added to ULIBS, while the PERF module within ULIBS was extended with an MPI-specific part. This part contains information relevant to the performance measurements definitions. These definitions are requests to the monitoring system for monitoring parallel library calls. In its turn, the PERF module needed to be extended by the measurement definitions which depend on the MPI communication subroutines semantics.

In case of MPI's collective communication, in practice, it is not always possible to distinguish between the sender and the receiver. Therefore, two new measurements have been added to PATOP, i.e. those related to the time and volume of a collective communication call. These measurements are expressed in terms of the OMIS requests similarly to those related to point-to-point communication. Since collective communication always involves a *communicator*, connected with a group of processes, it would be convenient to be able to apply a global measurement to a collection of processes involved, instead of selecting the processes one-by-one. Therefore, the tool environment was enhanced to provide information on communicators created within an application, which is gathered by means of appropriate instrumentation of the MPI communicator-bound subroutines.

3.3 Monitoring Overhead Test

In the closing stage of the adaptation of PATOP to MPI, the perturbation induced by the tool environment into the execution of a program was measured with the *ping-pong* benchmark provided within `mpich`. In the benchmark, two processes exchange messages using blocking send/receive subroutines `MPI_Send()` and `MPI_Recv()`. In the example, 10000 messages, 10000 bytes each were sent from process one to process two and back. That gives a total number of nearly 200 MegaBytes transmitted. Three test runs were performed:

1. Uninstrumented application. In this case, the application linked with an uninstrumented MPI library was executed.
2. Instrumented application. Similarly to test No. 1, but the instrumented library was used to test the overhead of inactive instrumentation code.
3. Application + PATOP. The application was monitored with PATOP using the *messages sent* and *messages received* performance measurements.

In each test, the *wall clock* time was measured using the `MPI_Wtime()` subroutine. Thus, the overhead comprises the direct overhead caused by the instrumentation and indirect overhead caused by the monitoring processes and PATOP. The table below shows a summary of the results obtained on two SUN workstations (300 MHz UltraSparc II), connected via fast Ethernet (100 MBit/s). The indicated error interval is $\pm\sigma$, i.e. the values' standard deviation.

Test	Mean exec. time $[s]$	Abs. overhead per event $[\mu s]$	Relative overhead [%]
Uninstrumented application	41.77 ± 0.19	–	–
Instrumented application	41.62 ± 0.14	–	–
Application + PATOP	43.54 ± 0.32	88.5 ± 18.5	4.2 ± 0.89

Within the limits of statistical variation the execution times of an uninstrumented application and an application with inactive instrumentation code are the same[2]. On the other hand, the overhead of full monitoring is more significant - the application was running approximately 4.2% slower. However, by executing PATOP and the OCM's NDU process on a separate node, this overhead could be further reduced.

4 Interoperability within the Tool Environment

The term *interoperability*, in the context of monitoring, mainly refers to online tools, and means their capability to run concurrently when applied to the same application [15]. Moreover, a cooperation between tools is supposed to be possible to provide additional functionality to the tool environment. For example, when applied to a long-time running parallel application, PATOP can be used to monitor and visualize the application execution. When in PATOP's performance

[2] In fact, the mean execution time for the instrumented application by chance even was slightly smaller than the mean execution time of the uninstrumented application.

displays one detects an unexpected behavior of the application, DETOP could be helpful to suspend the application's execution and to check relevant variables, and afterwards, to resume the application's execution.

The first basic requirement for interoperability concerns the capability of different tools to run concurrently. In case of tools coming from different vendors, *structural conflicts* between different portions of the monitoring systems may occur, which may even prevent tools from running concurrently. As multiple tools may request an operation on the same object at the same time, there must be provided an infrastructure to handle multiple requests. If tools do not form a monolithic environment, the interoperability of tools based on different monitoring systems is hardly attainable due to likely conflicts, e.g. those on exclusively accessible objects [14]. Further problems may occur at the user level and manifest in *logical conflicts*. E.g., if a debugger and a analyzer work concurrently, and a process is stopped by the former, the latter might not show this fact on its performance displays unless it is notified of the event. This could result in inconsistent performance visualization, what relates to *consistency problems*.

Originally, the OCM was designed to provide some coordination in accessing shared objects by multiple tools [15]. This concerns the mutual exclusivity of requests referring to a single object, the distribution mechanism of requests operating on more than one node, locking requests for their execution time.

To provide a management of the environment, a new OCM-oriented tool, OCTET (OCM-based Tool Environment top-level Tool) has been developed, which is intended to work on top of the tool environment [5]. OCTET performs two tasks: the first one is to start-up the tool environment, while the second task is to provide tools with information to resolve consistency problems. For setting up parameters like the name of a parallel environment (PVM or MPI), paths to the application and tools' executables, number of processes to be run (in case of MPI only), the tool provides a simple user interface.

On the startup of the tool environment, OCTET starts the OCM, application and tools[3]. The tools obtain a list of the application processes' tokens, attach to each of the application processes, and finally, get information on the environment to enable their possible interactions.

The cooperation of PATOP and DETOP can reveal incorrect behavior which occurs in two cases, depending on which tool started an application. If this was PATOP, which kept retrieving performance data from the OCM after the application had been suspended by DETOP, this could not comply with the user's expectations that PATOP should hang up monitoring when the application is being suspended. A solution may be in allowing PATOP to be notified on DETOP's activities. On the other hand, if the application is started by DETOP, PATOP may not start performance monitoring. Again, a notification that the application processes have changed their states is necessary.

Fortunately, based on the OMIS specification it is possible to "program" an action of PATOP on each event, which is related to a change in the state of a process, with *conditional requests* to the OCM [9]. The process' identifier is actu-

[3] Currently, in case of an MPI application, this itself is started prior to the OCM.

ally passed to the appropriate callback function mechanism, which is activated on every occurrence of the event. The callback function performs appropriate actions to stop or resume the measurements.

Questions may arise about interactions of the tools:

1. A tool, e.g. PATOP can program reactions to various scenarios of tools' co-operation. *So, how can PATOP learn of the actual configuration of the tool environment?*
2. *Which part of the software should be responsible for sending the above requests?* When answering we should bear in mind that inserting an additional code into a tool would affect the principle of the independence of tools.

To address the problems, each tool was provided with a specific library, in which every probable scenario of tools' cooperation would be handled [15]. The library is implemented as part of ULIBS and designed so that it does not affect the implementation of tools involved. With this implementation, the interoperability modules within the tools have to be provided with information on what tools are running within the environment. This information can be provided by the OCTET tool. For example, if OCTET knows that it is going to run two tools, e.g. PATOP and DETOP, it can inform each tool that the other tool is running. This information would actually be processed within ULIBS and passed to the interoperability module, what results in issuing appropriate conditional requests as mentioned above.

5 Concluding Remarks

Our work was focused on extending the range of supported parallel environments as well as on enhancing the set of provided on-line tools and their functionality available to the user. To accomplish this, on the one hand, the OCM has been enhanced to support a set of general purpose performance analysis requests, provided by PAEXT, on the other hand, it was possible to add a tool-specific requests (e.g. those for PATOP, embedded in PATOPEXT). In its turn, a tool, i.e. PATOP, was modified to be capable to support MPI, by adding measurements related to MPI-specific communication.

One of the main concerns within this work was interoperability of on-line tools for parallel programming support, which is a key feature for building a powerful, easy-to-adapt tool environment. The OCM monitoring system provides a basis to meet these requirements. Tools adapted to the OCM are enabled to run concurrently and operate on the same object, based on introduction of a high level tool. Due to this, a definition of tools' interactions, which leads to effective tool cooperation, is possible without intrusion into their implementation.

Future work will be concentrated on further extending of the range of parallel environments supported by the tools and on the problem of direct interactions between the tools already involved and those to be added. As for the new environments, our main goal is providing support for OpenMP applications. Special attention will be paid to a well defined naming policy for loops and barriers, based on information from the compiler. Further development in the context of interoperability will be focused on extending the role of OCTET in "programming" tools' interactions.

Acknowledgements. This work has been carried out within the Polish-German collaboration and supported, in part, by KBN under grant 8 T11C 006 15.

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. Bubak, M., Funika, W., Gembarowski, R., and Wismüller, R.: OMIS–Compliant Monitoring System for MPI Applications. In: R. Wyrzykowski, B. Mochnacki, H. Piech, J. Szopa (Eds.), PPAM'99 - The 3th International Conference on Parallel Processing and Applied Mathematics, Kazimierz Dolny, Poland, 14 - 17 September 1999, pp. 378-386, IMiI Czestochowa (1999).
3. Bubak, M., Funika, W., Iskra, K., Maruszewski, R., and Wismüller, R.: Enhancing the Functionality of Performance Measurement Tools for Message Passing Applications. In: Dongarra, J., Luque, E., Margalef, T., (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface. Proceedings of 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999, Lecture Notes in Computer Science **1697**, Springer, 1999. pp. 67-74.
4. 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.
5. Bubak, M., Funika, W., Balis, B., and Wismller, R. Interoperability of OCM-based On-line Tools. In: Dongarra, J., Kacsuk, P., and Podhorszki, N., (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface. *Proc. 7th European PVM/MPI Users Group Meeting*, Lecture Notes in Computer Science **1908**, pages 242-249, Balatonfred, Hungary, September 2000. Springer
6. 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.
7. Geist, A., et al.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts (1994)
8. Gropp, W., Lusk, E.: User's Guide for `mpich`, a Portable Implemention of MPI. ANL/MCS-TM-ANL-96/6, 1996.
9. Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A.: OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, (1997)
`http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`
10. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., and Newhall, T.: The Paradyn Parallel Performance Measurement Tool, *IEEE Computer*, vol. 28, No. 11, November, 1995, pp. 37-46

11. MPI: A Message Passing Interface Standard. In: Int. Journal of Supercomputer Applications, **8** (1994); Message Passing Interface Forum: MPI-2: Extensions to the Message Passing Interface, July 12, (1997) `http://www.mpi-forum.org/docs/`
12. Pase, D. Dynamic Probe Class Library: tutorial and reference guide, Version 0.1. Technical Report, IBM Corp., Poughkeepsie, NY, June 1998. `http://www.ptools.org/projects/dpcl/tutref.ps`
13. Shende, S.,Malony, A.D., Cuny, J., Lindlan, K., Beckman, P., and Karmesin, S.: Portable Profiling and Tracing for Parallel Scientific Applications using C++.In: Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134-145, Aug. 1998.
14. Trinitis, J., Sunderam, V., Ludwig, T., and Wismüller, R.: Interoperability Support in Distributed On-line Monitoring Systems. In: M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, High Performance Computing and Networking, 8th International Conference, HPCN Europe 2000, volume 1823 of Lecture Notes in Computer Science, Amsterdam, The Netherlands, May 2000. Springer.
15. Wismüller, R.: Interoperability Support in the Distributed Monitoring System OCM. In R. Wyrzykowski et al., editor, Proc. 3rd International Conference on Parallel Processing and Applied Mathematics - PPAM'99, pages 77-91, Kazimierz Dolny, Poland, September 1999, Technical University of Czestochowa, Poland.
16. Wismüller, R., Oberhuber, M., Krammer, J. and Hansen, O.: Interactive Debugging and Performance Analysis of Massively Parallel Applications. *Parallel Computing*, **22**(3), (1996), 415-442 `http://wwwbode.in.tum.de/~wismuell/pub/pc95.ps.gz`
17. Wismüller, R., Trinitis, J., and Ludwig T.: OCM - A Monitoring System for Interoperable Tools. In: Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98, Welches, OR, USA, August 1998.