Enhancing the Functionality of Performance Measurement Tools for Message Passing Environments

Marian Bubak[1,2], Włodzimierz Funika[1], Kamil Iskra[1], Radosław Maruszewski[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,iskra}@uci.agh.edu.pl, wismuell@in.tum.de
phone: (+48 12) 617 39 64, *fax:* (+48 12) 633 80 54

Abstract. The paper presents the concept of and insight into enhancing the portability and functionality of two performance measurement tools – PATOP and TATOO – in order to adapt them to work with message passing applications. We discuss the concepts of porting the tools to the OCM monitoring environment, the structure of the modified tools and the extensions made, as well as implementation issues.

1 Introduction

Performance measurement tools for message passing applications are intended to help in finding and fixing bottlenecks either via straightforward analysis of the state of application processes' execution and the communication between processes, or by modelling the location, time and cause of the bottleneck [1]. During recent years, a considerable number of performance tools have been made available [2]. Unfortunately, as a rule off-line tools feature the following principal disadvantages: potentially vast volumes of performance data, poor guideness and insufficient flexibility. In turn, existing on-line tools are rather difficult to port to parallel environments which are implementations of the *de facto* standards, such as `PVM` [3] and `MPI` [4]. Moreover, making two or more on-line tools to analyse or visualize the behaviour of the same application concurrently is virtually impossible.

Our efforts are aimed at development of performance measurement tools suitable for monitoring parallel applications that use the message passing paradigm, ensuring that the tools are portable to all current and future message passing libraries. Currently we are interested in monitoring `PVM` and `MPI` applications. In addition, we aim at enabling the interoperation of a tool with other program development support tools, like debuggers, visualisers etc., thus allowing to run such tools concurrently, while operating on the same instance of parallel application.

In order to realize these goals we decided to port and enhance the functionality of two existing performance measurement tools, `PATOP` [5] and `TATOO` [6], using the On-line Monitoring Interface Specification `OMIS` [7] and the `OCM` (OMIS Compliant Monitoring system) for low-level monitoring. The layer of abstraction provided by OMIS is used to gain the necessary flexibility, portability and interoperability,

In the subsequent sections we will discuss the functionality of the existing tools and the modifications made to support message passing programs. Preliminary concepts have been presented in [8].

2 PATOP and TATOO on Top of the OCM

PATOP and TATOO, two tools for on-line (PATOP) and off-line (TATOO) performance analysis, have originally been developed at LRR-TUM for the analysis of applications running under the PARIX environment on Parsytec systems. Performance visualisation is provided using an X11/Motif user interface. Due to almost identical user interfaces in both PATOP and TATOO, users do not have to learn twice how to use them. Both tools provide reasonably rich sets of available measurements and display diagrams. They support measurement and visualisation at various levels of detail: the whole system, individual nodes, particular processes or user functions. The available measurements comprise *busy* and *sojourn time*, *delay* in sending and receiving, *data volume* sent and received. Performance data may be presented in a wide range of ways, as bar graphs, multicurves, color bars, curve/matrix graphs, distrigrams, and matrix diagrams.

OMIS [7] has been designed to define a standardised interface between runtime tools and the parallel programming environments, which is independent of the parallel environment, while providing a universal, portable and extensible API. OMIS includes the definition of a basic set of monitoring services divided into three groups: *information* services, *manipulation* services, *event* services. The basic set can be augmented with extension services to support requests specific to a particular purpose or environment. The OCM is a reference implementation of OMIS, which provides support for monitoring message passing applications under PVM (a version for MPI is about to be issued). The OCM is a distributed monitoring environment, with a central Node Distribution Unit and local monitoring processes (one per node), communicating with the monitored processes via shared memory. It is worth mentioning that an alternative approach which provides a software architecture for the implementation of monitoring is presented in [9].

The integration of PATOP with the OCM involves an additional layer, ULIBS to ease interfacing PATOP to the OCM. The task of ULIBS is to transform high-level measurement specifications to event/action relations accepted by the OCM and to transform back the results. In Fig. 1 the structure of the on-line monitoring environment is presented with monitoring a sample parallel application. In order to enable the monitoring of a parallel application by the OCM, some parts of the OCM must belong to the target system. To provide run-time tools with

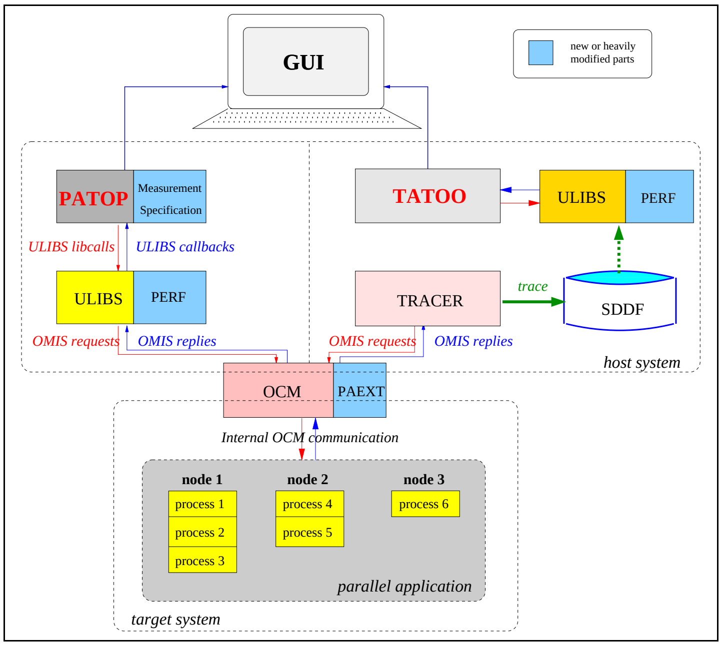

Fig. 1. Operation of PATOP and TATOO on top of the OCM

full functionality, such as the monitoring of communication library calls, parts of OCM must be linked to the application being monitored. The PATOP tool runs on a host machine and communicates with the OCM using the abstraction layer, ULIBS.

The structure of the TATOO tool is shown on the right side of Fig. 1. Performance data are stored in local trace buffers within the monitored application processes. When the application terminates, or a trace buffer is almost full, the OCM reads its contents. The trace data is then sent to TRACER – a standard OMIS client distributed with the OCM. TRACER saves the data to a trace file in the SDDF format. The TATOO tool reads this trace file, using a special, off-line version of ULIBS. Adapting PATOP and TATOO to PVM and MPI required the definition of a number of new services for performance analysis and a modification of the ULIBS functionality.

3 Gathering of Performance Data Using 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 the necessary support for obtaining such information. There are three principal ways of making this data available to the performance analysis tools:

1. Using *direct communication* — whenever an interesting event is detected, the OCM executes the services that have been associated with that event. The services send their results (i.e. information on the event or the state of the application when the event occured) to the tool. Fig. 2 shows the flow of control and data in this case.
2. Using a *trace* — when a relevant event occurs, some information on the event is stored in a trace buffer local to the application process. It is later read by the OCM on request. The tracing mechanism is available via an OMIS extension included in the OCM. Fig. 3 shows the flow of control and data in this approach. It is also valid for the integrator/counter approach described below.
3. Using *counters* and *integrators* — the most important information on each event (e.g. the message length for a send event) is summarized in counters and integrators local to the application process and read by the OCM on request. The mechanism supporting the counters and integrators is part of PAEXT — a performance analysis extension to the OCM, which is outlined in Section 4.

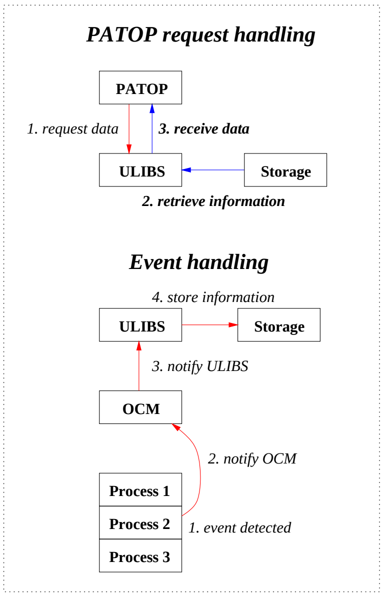

Fig. 2. Direct communication mechanism

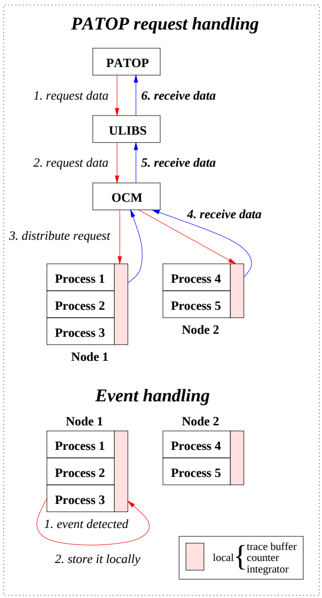

Fig. 3. Tracing and counting/integrating mechanisms

When choosing the proper way of gathering performance data it is indispensable to take into consideration the demand for a particular kind of performance data and the cost of gaining it. The *direct communication* allows to acquire not only the full information on each event, but also information on the state of the application when the event occurred. However, it also has the greatest cost, as a message needs to be sent for each event occurrence. In the *tracing* and *counter/integrator* approach, communication is needed only infrequently when the displays of PATOP are updated. On the other hand, less information can be retrieved. While *tracing* at least allows to store detailed information on each single event, *counters and integrators* only provide summary information. But of course, the data volume that needs to be transfered is therefore minimal. Table 1 summarises the pros and cons of each approach. For each measurement,

	Direct communication	Tracing	Counters/integrators
Available range of performance data	basically unlimited	wide	only summary information
Acces to state info	yes	no	no
Event filtering	yes (in ULIBS)	yes (in ULIBS)	no
Monitoring overhead	high	medium	low
Processing in ULIBS	high	high	low

Table 1. Comparison of three ways for gathering performance data

we use the *least intrusive* approach that still meets our goals. E.g., to carry out a system-wide analysis of the message volume sent, all the necessary support can be provided with counters with the least possible intrusion. On the other hand, when limiting the analysis e.g. to certain source functions or to the messages sent to individual destination processes only, we have to use traces (all the events are stored in the trace, together with the destination processes' IDs; the uninteresting ones are later filtered out in ULIBS). The current set of event services defined by OMIS do not efficiently support these more sophisticated performance measurements. To overcome this difficulty, new specialised event services are needed. Such services can be added to the OCM using the OMIS extension interface.

4 PAEXT - a New OCM Extension

For the purposes of performance analysis using the counter/integrator approach, a new extension (PAEXT) to the OCM has been defined. In PAEXT two additional types of tokens, i.e., OMIS object identifiers are defined:

`pa_c_` and `pa_cl_` for *counters*. These are plain `integer` variables that can be used for measuring values like message count and message volume.

`pa_i_` and `pa_il_` for *integrators*. These are used for storing `floating` values, but are more complicated than that. They actually store an *integral* of a function which is constant over some interval. Internally, the integrator consists of three variables: `value` — the value of the function, `time` — the beginning of the interval, and `integral` — the integral value at the beginning of the interval. The integrators can be used for measuring values like the message receive delay.

These objects can be either *local* (`pa_cl_` and `pa_il_`), i.e. there is a separate instance for each process of the application, or *global* (`pa_c_` and `pa_i_`), i.e. one instance for the whole application (for performance reasons they are implemented as local objects on each node with a global sum performed on read in the OCM). Table 2 lists the operations defined for local objects. The same operations also exist for global ones. In the case of adding a new interval to an integrator, `time`

Service	Meaning
token `pa_counter_local_create ( token* objlist)`	create counters
void `pa_counter_local_delete ( token* counter_list)`	delete counters
void `pa_counter_local_increment ( token counter, integer increment)`	increment the value of the counter
integer `pa_counter_local_read ( token* counter_list, integer clear)`	get values of counters
token `pa_integrator_local_create ( token* objlist)`	create integrators
void `pa_counter_integrator_delete ( token* integrator_list)`	delete integrators
void `pa_integrator_local_increment ( token integrator, floating increment, floating time)`	increment the value of the integrator
floating `pa_integrator_local_read ( token* integrator_list, floating time, integer clear)`	get values of integrators

Table 2. Local *counter* and *integrator* specific services.

marks the end of the previous interval and the beginning of the new one, and `increment` is the value of the function in the new interval (relative to the previous value). In the read operation, if `clear` is non-0, the value is reset after being read. As an example, to measure the total delay in `pvm_recv()`, the following services can be defined:

– event services (the integrator's `value` is initially 0):

```
thread_has_started_lib_call([], "pvm_recv"):
   pa_integrator_global_increment(pa_i_1, 1, $time)
thread_has_ended_lib_call([],"pvm_recv"):
   pa_integrator_global_increment(pa_i_1, -1, $time)
```

the `pa_i_1` token is associated firstly with the start `time` of each `pvm_recv` through `1` flag, and finally passing `-1` as the measurement flag makes the time spent in the function to be incremented on the exit from the function.
– once in a while the current integral containing the time elapsed can be obtained from the `pa_i_1` integrator token, and then the integrator gets reset:
`:pa_integrator_global_read([pa_i_1], 0, 1)`

5 Modifications of the ULIBS Library

ULIBS is a library of high-level performance measurement and debugging functions. It defines a level of abstraction between run-time tools and the OCM. It has been developed before the OCM for use within the PARIX parallel environment, and was later adapted to the OCM. ULIBS provides the following functionality to the 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. *Performance analysis* support in ULIBS comprises several functions:

– define a new measurement based on a specification passed as argument. Available kinds of measurement include *global sum*, *matrix*, *distribution*, *restriction* and *event trace*. Data to be measured includes *busy* and *sojourn time*, message passing *delays*, and *data volume*.
– start/stop the gathering of performance data,
– get performance data over the period of time passed as an argument; call an asynchronous callback when done,
– delete a measurement.

While the interface of these functions basically remained unchanged w.r.t. to PARIX version, their implementations had to be completely redesigned for the use with the OCM. As outlined above, in contrast to the highly specialized PARIX monitoring system [5], the OCM does currently not provide direct support for the more sophisticated measurement specifications, like the ability to limit a measurement to some source-level functions. These need to be emulated in ULIBS for the time being, resulting in higher overhead. More sophisticated support for performance analysis from OCM is underway.

6 Concluding Remarks

In this paper, we have presented porting two performance measurement tools – PATOP and TATOO – to the OCM environment and the enhancement of their functionality. As a result we have got relatively universal tools which may interoperate with other tools (e.g. debuggers). In the final version of the paper, quantitative data on the performance monitoring overhead will be given. The tools may be used for the effective user-controlled search of performance bottlenecks in message passing programs. The extensions made to the OCM and

ULIBS also enable to detect performance problems with further, e.g. automatic tools [10] on top of OCM.

Acknowledgements. This research was carried out within the German-Polish collaboration.

References

1. 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
2. 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.
3. Geist, A., et al.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts (1994)
4. 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/`
5. Wismüller, R., Oberhuber, M., Krammer, J. and Hansen, O.: Interactive Debugging and Performance Analysis of Massively Parallel Applications, *Parallel Computing*, **22** (1996) 415-442
`http://wwwbode.in.tum.de/~wismuell/pub/pc95.ps.gz`
6. Borgeest, R. and Dimke, B. and Hansen, O.: A Trace Based Performance Evaluation Tool for Parallel Real Time Systems, *Parallel Computing*, **21 4** (1995) 551-564.
7. 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`
8. Bubak, M., Funika, W., Iskra, K., Maruszewski, R., and Wismüller, 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**, Springer, 1999, pp. 1270-1273.
9. 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, Lecture Notes in Computer Science **1401**, Springer, 1998, pp. 708-717.
10. Espinosa, A., Margelef, T., and Luque, E.: Automatic Detection of PVM Program Performance Problems. 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. 19-26.