Enhancing OCM to Support MPI Applications

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

Abstract. The OCM is a universally usable, distributed on-line monitoring system currently implemented for the PVM programming library. Due to the growing use of MPI in parallel programming, there is a need to enhance the OCM in order to support MPI applications development. This paper presents approaches to solve the problems of the `mpich`-oriented start-up mechanism and profiling MPI library calls in the OCM. While the existing transport layer for the OCM internal communication is preserved, a number of new MPI-related services are added.

1 Introduction

The ever growing interest in parallel programming with message passing led to the establishment of MPI as its standard. While there are a number of tools which support MPI applications development, most of them suffer from poor portability and interoperability. The emergence of the *On-line Monitoring Interface Specification* (OMIS) [1] is intended to relieve the problem with providing a well-defined mechanism of interaction between a monitoring system and tools, e.g. for debugging, performance monitoring, load balancing, etc.

For supporting a typical tool functionality, OMIS defines three categories of services: information, manipulation, and event services. To prove the concept, an *OMIS-Compliant Monitoring system* (OCM) has been built using the basic set of services and a PVM-related extension [2]. OCM comprises a set of local monitors, which provide the complete monitoring functionality on *each single node*, and Node Distribution Unit (NDU) which is responsible for managing the local monitors' activities within the *entire configuration*. Due to the substantial differences between PVM and MPI w.r.t. the handling of application processes, the following problems have to be solved when extending the OCM functionality to support MPI applications development: *start-up* mechanism of OCM, realization of *information services*, and *profiling* of MPI library calls. `mpich` was used as a relatively mature, popular and publicly available implementation of MPI.

2 Start-up Mechanism of OCM and MPI-Related Information Services

As defined in OMIS, the information services should gain some information about the monitored MPI parallel application: a *list of nodes* where the program runs, *local UNIX PID*, *rank* in MPI_COMM_WORLD, and *path* to the program's executable. This information can be accessed by the tool using the `mpi_get_nodelist()` and `mpi_get_proclist()` services realized within the MPI-related extension of OCM. In contrast to PVM which provides `pvm_config()` and `pvm_tasks()` to obtain the above information, in `mpich` the parallel program to be monitored must be started in the *ptrace mode* to get this information via the system call `ptrace()`, which allows to read the values of variables from the address space of the application's master process. The data needed for the information services are contained in the table `MPIR_proctable` (a variable provided by `mpich`) and the *PID* specification file which is created by `mpirun` for the `p4 device` [3].

`mpich` provides the `mpirun` start-up script, which being called with a parameter like `gdb`, `ddd`, `tv` or `xgdb` debuggers, will run the corresponding debugger. Thus, by having modified the start-up script (e.g., with adding OCM to the list of debuggers used for `mpich`), it has been possible to start the OCM. When `mpirun` starts an application in this way, it forces the slave processes into an endless loop inside the `MPI_Init()` function [3]. This provides the time necessary for the OCM to obtain the information about the process IDs and to capture control over all of the application processes. The application's execution can then be forced to leave the loop inside `MPI_Init()` and to resume by changing the value of a loop control variable using `ptrace()`.

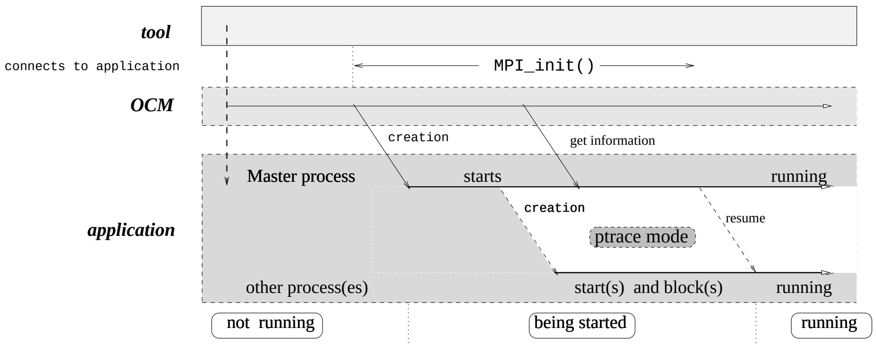

Fig. 1. Scheme of the start-up mechanism of OCM

The above mechanism is used by a new function - `mpi_config()` (used within OCM only), whose purpose is similar to that of `pvm_config()` in the context of the tool. Thus, we managed, on one hand, to provide a basis for the MPI-related information services, on the other hand, to minimize changes in the existing OCM services. Fig. 1 illustrates the start-up mechanism. First, the master process is created and started, then the other processes follow and get blocked, afterwards the program is enabled to be `ptraced`. OCM obtains the indispensable data from the master process. Fig. 2 shows how OCM may be used to control

mpirun -np 3 -ocm mpi_program	call `mpirun` with OCM
	mpi_program consists of three processes
tool	call tool, which uses OMIS functions
:mpi_get_nodelist()	return the list of nodes
:node_attach([n_1,n_2,n_3])	attach to nodes
:mpi_proc_create()	create all processes of the application
:mpi_get_proclist()	return the list of processes
:proc_attach([p_1,p_2,p_3])	attach to processes
:thread_continue()	MPI program runs

Fig. 2. Example of the OCM functions calls for MPI program

an MPI program. First, the MPI program is started with OCM and `mpirun` generates a file with a description of processes of the MPI program. Next, the user can call the tool which uses OMIS functions, and the tool connects to OCM. When the tool calls the services `mpi_proc_create` and `thread_continue`, OCM creates the processes and runs the parallel program.

3 MPI Library Calls Detection

In the context of the OMIS requirement that the monitoring system should recognize the application events related to the programming library, it is important to be able to detect the entry into and the exit from library functions and to read the values of their input and output parameters. A common method to do this is inserting so called *hooks* at the beginning and the end of all programming library functions. The hooks can inform the monitoring system whenever these events occur. While in PVM programmers are forced to do binary wrapping to detect library calls, in MPI this is realized with the *Profiling Interface* mechanism that implies that all the MPI functions, whose names normally start with a `MPI_` prefix, are accessible with a `PMPI_` prefix, too. When implementing the detection of the library functions with this mechanism, the profiling versions of MPI functions are written in the style as in Fig. 3. The routines for generating hooks are created only once, and then wrapping them around the `PMPI_` calls is done automatically using a wrapper generator with adding a specific code to each function. To use the profiling library, the application has to be linked in a *special way* - the wrapper object code must be linked before the profiling MPI library which, in turn, should be placed before the standard MPI library.

```
int MPI_Function()
{ ...
    if ( OCM_HOOK_IS_ACTIVE(...) )  {   /* is hook active ? */
        ocm_evmgmt_hook_exec(...);      /* inform OCM about entry
        ...                                into library function */
    }
    retcode = PMPI_Function();          /* call original function
                                           using PMPI_ entry */
    if ( OCM_HOOK_IS_ACTIVE(...) )  {   /* is hook active ? */
        ocm_evmgmt_hook_exec(...);      /* inform OCM about exit
        ...                                from library function */
    }
    return retcode;
}
```

Fig. 3. Profiling version of MPI function

Now the OCM can activate hooks for specific functions and receive notifications about the detected library function calls from the profiling library.

4 Conclusion

While enhancing OCM, which up to now worked only with PVM, to support MPI applications as well, two substantial problems were to be coped with: (1) due to differences in starting up applications in PVM and MPI, the *start-up* mechanism was to be modified, (2) to preserve consistency with the PVM *information service*, a new MPI information service was developed. The enhancement presented was carried out for the `mpich` implementation of MPI. To extend the solutions above to support other MPI implementations, it will be necessary to perform their case studies.

Acknowledgments. This research was done in the framework of the German–Polish collaboration.

References

1. Ludwig, T., Wismüller, R., Sunderam, V., and Bode. A. OMIS – On-line Monitoring Interface Specification (Version 2.0), vol.**9** of LRR-TUM Research Report Series. Shaker Verlag, Aachen, Germany, (1997), ISBN 3-8265-3035-7.
2. Wismüller, R., Trinitis, J., Ludwig, T. A Universal Infrastructure for the Runtime 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 Verlag.
3. 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 Verlag.