Integrating PVaniM into WAMM for Monitoring Meta-Applications

R. Baraglia, M. Cosso, D. Laforenza, M. Nicosia

CNUCE - Institute of the Italian National Research Council
Via S. Maria, 36 - I56100 Pisa (Italy)
Tel. +39-50-593111 - Fax +39-50-904052
e-mails: R.Baraglia@cnuce.cnr.it, D.Laforenza@cnuce.cnr.it

Abstract. Metacomputing is one of the most interesting evolutions of Parallel Processing. A complete environment for metacomputing should have tools for monitoring applications that can gather information both on the applications being executed and on the processors that they are executed on. Such data can be used to manage statistics, for debugging, and for tuning meta-applications. This paper describes an integration between WAMM, a visual interface for the configuration and management of a metacomputer, and PVaniM, a system that provides support for displaying the behaviour of PVM applications.

1 Introduction

WAMM (Wide Area Metacomputing Manager) [2, 3] is a graphic interface based on OSF/Motif and PVM [4, 5, 6], developed by the Parallel Processing Research Group at CNUCE, Pisa, in 1995[1]. WAMM is the first step towards the development of a complete tool for the management of a metacomputer [1] based on PVM. Like others of its kind [7, 8] besides having functionalities that facilitate the user in defining and managing a virtual machine, this tool helps to automate some activities that would otherwise be carried out manually by the programmer. This paper describes the integration of monitoring functionalities in WAMM, so as to allow an on-line or post-mortem analysis of the behaviour of a meta-application during its execution. The main characteristics of WAMM are: geographical view of the system, configuration of the virtual machine, remote commands, remote compilation, task control and monitoring functionalities.

2 Monitoring applications in WAMM

Previous versions of WAMM enabled users to get trace data generated with tracing mechanisms supplied by PVM. The record events could thus be received by the interface, but they weren't managed in any way. To add monitoring to WAMM would have required a tool for analysing the trace data received. Since the PVM tracing mechanisms don't carry out any buffering of the data generated,

[1] For more information see at: `http://miles.cnuce.cnr.it`

such data are quite intrusive with regard to the execution of the application. Consequently, our solution doesn't use trace data generated by PVM, instead it uses data produced by a library that we created [15] on the basis of the PVaniM library. Moreover, since it would have been very costly to develop a tool from scratch for monitoring and displaying the behaviour of the applications, we decided to expand the functionalities of WAMM by integrating an existing monitoring tool. We examined the main tools [8, 9, 10, 11, 12, 13, 14] and opted for PVaniM [10]. Not only did we extend the functionalities of WAMM but we also analysed in detail the chosen tool and tried to eliminate some of its limitations in terms of our requirements.

3 PVaniM

PVaniM is a system that supports on-line and post-mortem displays of the behaviour of PVM applications, written in C, C++, or Fortran. PVaniM consists of a tool library plus two display tools (`pvanimOL` for on-line analysis and `pvanim` for post-mortem analysis). It was designed by Brad Topol and John T. Stasko of the Georgia Institute of Technology and by Vaidy Sunderam of Emory University. The main features of the system are: separation of on-line and post-mortem display functionalities, use of displays on external loads, support for interactive steering, overhead introduced by monitoring that can be controlled by the user, support for I/O and support for the traces defined by the user.

4 WAMM-PVaniM integration: implementation aspects

Integrating WAMM with an existing monitoring tool on the one hand overcame the problem of having to develop a new tool, yet on the other hand entailed a detailed study of PVaniM. This enabled us to eliminate some of the limitations that we had found in PVaniM, while leaving both the modular structure of WAMM and the general functioning of PVaniM as they were. For example, we decided not to modify the display techniques used by PVaniM. In order to avoid delays in producing displays, the component in charge of display doesn't have to calculate the relationships between the information it receives and the tasks to which such information refers. This means that the process that deals with display only has to concern itself with graphically representing data, assuming that the data relating to a particular task occupy the same position in the preset ad hoc data structures, whichever task sends this information. The correctness of the display phase obviously thus depends on the tasks of the application, since it is at this level that the association needs to be maintained between the data and the tasks to which such data refer. One of the main implementational choices that makes our solution differ from the one adopted by PVaniM was in determining who takes charge of this association and the decision about when to make it visible, in a univocal way, to all the application tasks.

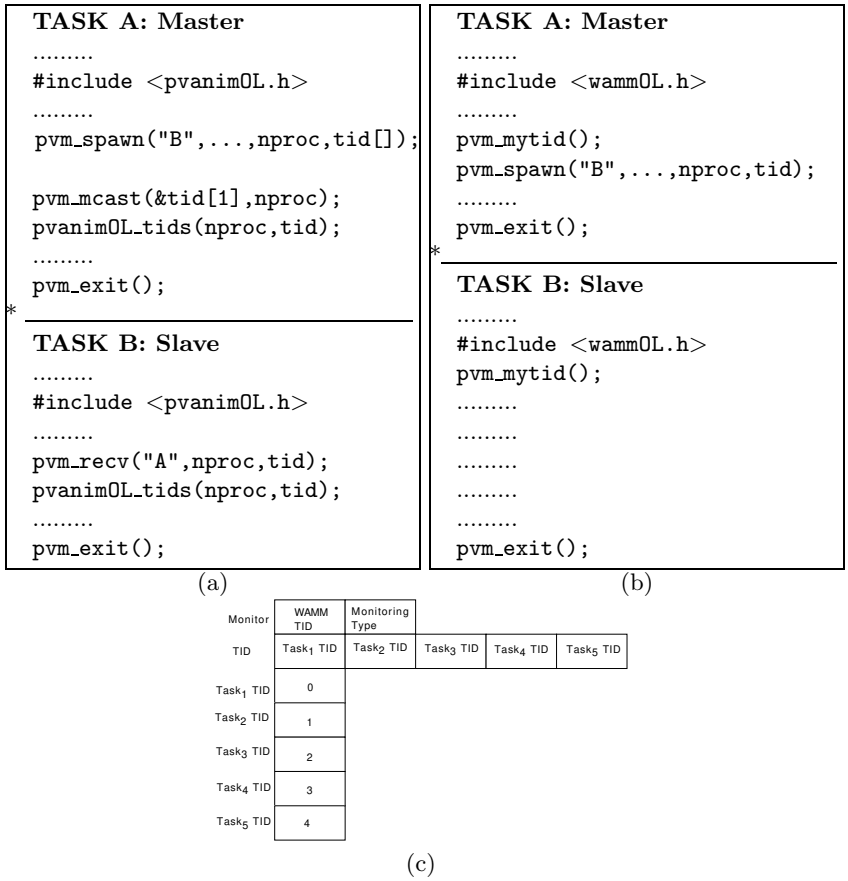

Fig. 1. Monitoring of an application with PVaniM (a) or WAMM (b), and snapshot of the database managed by the PVM master demon (c).

4.1 Use of PVaniM: implications on the instrumentation of the applications

In PVaniM the development of the task-position association is entrusted to the master process of the application, which via the issue of the array containing the TIDs of all the tasks activated by it (slaves), indicates to all the other tasks what the established order is. Before the monitoring phase begins, each application task has to invoke the `pvanimOL_tids()` routine (see at Fig. 1(a)) so that all the data structures needed for monitoring are suitably organised. This means that, before monitoring begins, each task has to be aware of the global number of application tasks along with their identifiers. As shown in Fig. 1(a), these implementational choices mean that:

– to instrument an PVaniM application the programmer has to modify the applications by inserting `pvm_mcast()`, `pvm_recv()` and `pvanimOL_tids()`;
– if, after invoking the first `pvm_spawn()`, task A (master) needs to continue with other calculations and then invokes a new `pvm_spawn()` to activate other tasks, for the latter to be able to be monitored the function `pvm_mcast()` needs to be executed (and thus the invocation of the `pvanimOL_tids()`) only after the second `pvm_spawn()`, thus delaying the beginning of the monitoring. Consequently any communications carried out by the two `pvm_spawn()`'s cannot be monitored;
– it is impossible to trace processes that were activated by task B.

4.2 Use of WAMM: implications on the instrumentation of the applications

To reduce modifications to the instrumented code Fig. 1(b), in our solution the association between the various tasks and the number they will be identified with for the next display is made using a database created and updated by the PVM master demon, which allows some data to be shared among each task of the virtual machine. The storing of the TIDs of all the processes to monitor, within the database, doesn't mean that all the PVM tasks have to know a priori the TIDs of the processes to monitor, since this information can be determined at run-time, when the first communication takes place between a task and a new partner. Moreover, using the database avoids having to manage problems of mutual exclusion, since access to the database is regulated by the master demon which acts as a server. The database maintained by the PVM master demon (see Fig.1(c)), allows whole data to be stored, and these data are grouped into classes. In our solution we use the following classes:

– **monitor**: this is used by the `wamm`[2] process to insert its own TID and the type of monitoring to carry out chosen by the user before the execution of the application is activated;
– **TID**: this is used by all the processes that take part in the monitoring. When they are activated the various tasks insert the value of their TID in the first free position of this class; the position in which they are inserted will represent the index with which they will be identified by the monitor and all the other tasks.
– $Task_i$ **TID** : where $Task_i$ indicates the i-th task involved in the monitoring. Each of those classes is used to obtain, through a single database access, the $Task_i$ index that will be used by `wamm` and all other tasks.
The use of classes has several advantages: the class *monitor* enables any task to find out, at execution time, whether monitoring has to be carried out or not, the type of monitoring to carry out and the value of the TID of the task to which trace data have to be sent. This means that the user doesn't have to recompile the application should he/she wish to execute it with a

[2] `wamm` is the user interface of `WAMM`

different type of monitoring or with no monitoring at all. On the other hand, the combined use of the other classes, allows the task-position association described earlier to be made (Fig.1(c)).

4.3 Further remarks

To collect the information relating to the processors that make up the metacomputer, PVaniM uses an `olslave` process (that executes the Unix command `uptime`) which is activated by each application task when `pvanimOL_tids()` is invoked. If several tasks are allocated on the same machine and such a situation is quite frequent, we would have several slave processes (`olslave`'s) needlessly producing the same information, and that would increase the load of the machines themselves. We avoid this by creating one `olslave` for each machine in the the metacomputer. In fact, the `PVMTasker` processes generated by WAMM are given the task of activating the slaves in such a way as to have one copy on each machine that is part of the metacomputer. We thus manage to reduce the intrusion of the tool on the execution of the application. In addition, although it is a functionality that is scheduled for future versions, the current version of PVaniM doesn't allow several copies of the same application to be monitored, whereas this is possible with WAMM. Finally, PVaniM doesn't allow one to choose just on-line analysis or just post-mortem analysis. This is in fact a useful alternative and is implemented in WAMM.

5 Some performance issues

Figures 2(a) and 2(b) respectively show the average completion time and the elapsed times obtained by the various executions of a sample application[3], described in [5], as a function of the type of monitoring required. From the tests executed on a small cluster of workstations, our solution shows an improvement in performance over PVaniM of around 10%.

Moreover, as highlighted by Fig. 2(a), the execution that was carried out just in *Sampling* modality is much more expensive than execution just in *Tracing*. This is because in the first case, the data collected are managed and displayed, whereas in the second case, the data are only collected in a temporary memory buffer that is downloaded on the secondary memory only when the buffer is saturated, without any kind of processing taking place. The improvement in performance of WAMM over PVaniM derives above all, in our opinion, from the fact that for the collection of data regarding the load on the machines we use less slave processes (`oslave`'s) that execute the `uptime` command. Figure 2(b) also shows that the times obtained in the execution of the application via WAMM (Sampling+Tracing) are quite different and this gives rise to a very wavy plot, while the curve for PVaniM is quite constant. This is mainly, we think, due to the fact that access to the database managed by the PVM demon is centralised.

[3] One-Dimensional Heat Equation

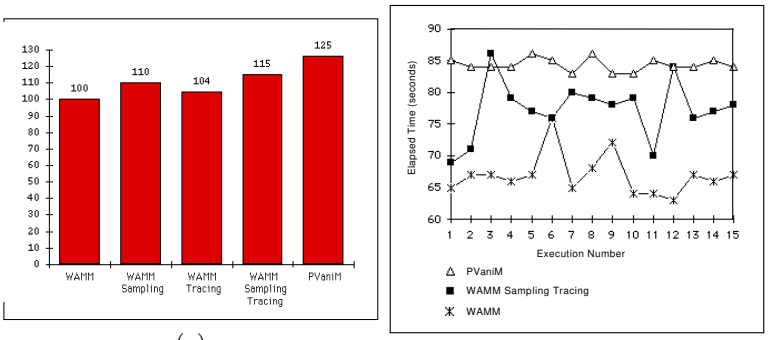

Fig. 2. Comparison of the average completion time (a) and elapsed times (b) of the test application as a function of the type of monitoring required.

This implies that the various tasks could conflict with each other when trying to access the database and due to the serialisation of the accesses, this causes a delay which in any case is not particularly expensive.

In our opinion, the choice of the test application and, in particular, its communication pattern, doesn't seem to be related to the better performance obtained by WAMM over PVaniM. Just to quantify the cost of using the database, assume that each execution of a `pvm_insert()` or `pvm_lookup()` function is approximately equivalent to two `pvm_send()`'s. The cost of our solution in the worst case, i.e. for an application consisting of n PVM tasks each communicating with all the others, is: two `pvm_insert()` for each task (equal to $4n$ `pvm_send()`'s) and $2(n(n-1)/2)$ `pvm_lookup()` (equal to $2n^2 - 2n$ `pvm_send()`'s), that is a total of $2n^2+2n$ `pvm_send()`'s. PVaniM has in any case a cost equal to *n-1* `pvm_send()`'s. The test application adopted consisted of a *master* and five *slaves*; each task communicates with the *master* process and with only two adjacent nodes. Thus the cost of WAMM in relation to the use of the database is 60 `pvm_send()`'s, whereas PVaniM only carries out 5 `pvm_send()`'s.

As expected, the solution adopted in WAMM generaly leads to greater communication, and this confirms that the improvement in performance of WAMM over PVaniM is mainly due to the reduced number of `oslave` processes.

6 Conclusions

In order to extend WAMM with monitoring functionalities, in this work we have analysed the methodologies and difficulties connected with the design and development of tools for monitoring applications in a metacomputing environment. To be able to carry out the implementational part of our work, we analysed some of the most common monitors for PVM applications. We examined the main tools for analysing the behaviour of PVM applications. These evaluation enabled us to decide on PVaniM as being the most suitable tool for integra-

tion with WAMM. It is worth pointing out that our work was not limited to connecting in some way PVaniM monitoring functions with those supported by WAMM. In fact we went much further by removing what we considered to be the drawbacks of PVaniM. The resulting tool was tested on a cluster of workstations using a benchmark application that allowed us to compare our solution with the one adopted by PVaniM. The results obtained proved that our solution, as compared with the one used in PVaniM, ensures a shorter average completion time of the application under observation.

7 Acknowledgments

We would like to thank CNUCE Institute of the Italian National Research Council for allowing us to use their computing facilities. Very special thanks to Brad Topol for all his kindness and expertise in promptly answering all our queries about PVaniM.

References

1. L. Smarr, C.E. Catlett. Metacomputing. Communications of the ACM, June 1992, Vol. 35, No. 6 (45- 52).
2. R. Baraglia, G. Faieta, M. Formica, D. Laforenza. WAMM: A Visual Interface for Managing Metacomputers. EuroPVM'95, Ecole Normale Supérieure de Lyon, Lyon, France, September 14-15, 1995, 137–142.
3. R. Baraglia R., G. Faieta, M. M. Formica, D. Laforenza. Experiences with a Wide Area Network Metacomputing Management Tool using IBM SP-2 Parallel System. Concurrency: Practice and Experience, J. Wiley & Sons, Inc., Vol.9(3), 1997, pp.223-239.
4. V.S. Sunderam. PVM: a Framework for Parallel Distributed Computing. Concurrency: Practice and Experience, 2(4):315–339, December 1990.
5. A.L. Beguelin, J.J. Dongarra, G.A. Geist, W. Jiang, R. Mancheck, V.S. Sunderam. PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.
6. A.L. Beguelin, J.J. Dongarra, G.A. Geist, R. Mancheck, V.S. Sunderam, W. Jiang. PVM3 Users' guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Lab, May 1994.
7. J.E. Devaney, R. Lipman, M. Lo, W.F. Mitchell, M.Edwards, C.W. Clark The Parallel Applications Development Environment (PADE), User's Manual. PADE-Major Release 1.4, November 21, 1995.
8. J.A. Kohl, G.A. Geist. XPVM 1.0 Users' Guide. Technical Report ORNL/TM-12981. Computer Science and Mathematical Division, Oak Ridge National Laboratory, Oak Ridge, TN, April 1995.
9. M. T. Heath and J. E. Finger. ParaGraph: a tool for visualizing performance of parallel programs. Oak Ridge National Lab, Oak Ridge, TN, 1994.
10. B. Topol and J. Stasko, V. Sunderam. PVaniM 2.0:Online and Postmortem Visualization Support for PVM, June,1996.
11. B. Topol and J. Stasko, A. Alund. PGPVM Performance Visualization Support for PVM, February,1995.

12. E. Maillet. TAPE/PVM an efficient performance monitor for PVM applications - User's guide. Available via FTP: ftp.imag.fr, /imag/APACHE/TAPE, June, 1995.
13. A. Beguelin, J. Dongarra, A. Geist, V. Sunderam. Visualization and Dubugging in a Heterogeneous Environment. IEEE Computer, vol.26, n. 6, 88–95, June,1993.
14. P H. Worley. A New PICL Trace File Format. Oak Ridge National Lab, ORNL/TM-12125, Oak Ridge, TN, USA, September,1992.
15. M. Cosso, M. Nicosia. Il monitoraggio delle meta-applicazioni. Estensioni di una interfaccia per la gestione di metacalcolatore, con funzionalitá di monitoraggio. Master Thesis, Dept. of Computer Science, University of Pisa, May, 1997.

This article was processed using the LaTeX macro package with LLNCS style