Performance Measurement, Debugging and Load Balancing for Metacomputing

Marian Bubak[1,2], Włodzimierz Funika[1], Dariusz Żbik[1], Dick van Albada[3], Kamil Iskra[3], Peter Sloot[3], Roland Wismüller[4], Krzysztof Sowa-Piekło[5]

[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] *Research Institute Computer Science, Universiteit van Amsterdam, The Netherlands*

[4] *LRR-TUM – Technische Universität München, D-80290 München, Germany*

[5] *Institute for Software Science, University of Vienna, Austria*

bubak@uci.agh.edu.pl

Abstract

The paper presents experience with building tools for the development of parallel applications and for load balancing using a library of migrating sockets. Based on this experience, design ideas for a tool environment which supports metacomputer applications are presented. These ideas are related to aspects of tools development for the Globus system.

1 Introduction

When developing a program for modern computer architectures, it is crucial to have tools, which support the design, debugging, testing, and evaluation phases of program development. However, building parallel tools which meet requirements imposed by a metacomputer system and users is a tough task. On the other hand, a computer system should enable an efficient and transparent use of its resources. Based on our experience in the development of utilities for parallel distributed computing, we present design ideas for a tool environment that supports performance measurement, debugging and load balancing of metacomputer (grid) applications [1].

The performance measurement and debugging tools are based on existing utilities for parallel distributed computing, which are built on the OCM – a flexible distributed monitoring system compliant to the Online Monitoring Interface Specification OMIS[2]. For grid computing, the OCM capabilities will be extended by defining a set of new objects and services. Efficient load balancing may be realized by extending the well-known Dynamite resource management system [3] with a new migrating sockets library.

This library is intended to make it possible to re-establish a new connection between migrating processes and to ensure that during migration no socket connections and data are lost.

The principal idea of OMIS [2] is to define a standardized interface between tools and monitoring systems. This results in a set of tools which fill the gap between the multiplicity of tools and the scarcity of really portable monitoring systems. Other benefits include *interoperability* of tools as well as the OMIS's capability of *extending* the range of possible services so that its functionality can be adapted to specific features of a particular software or hardware environment. The tools, primarily built on proper monitoring facilities, have been transferred onto the infrastructure of the OCM based on OMIS, due to the reasons above. Further work on porting these tools to grid computations will make use of the same OCM, which needs, however, to be enhanced. Below we consider the evolution, structure, and features of the tools as well as changes in the monitoring functionality when incorporating our tools into an existing metacomputing infrastructure such as the Globus system [1].

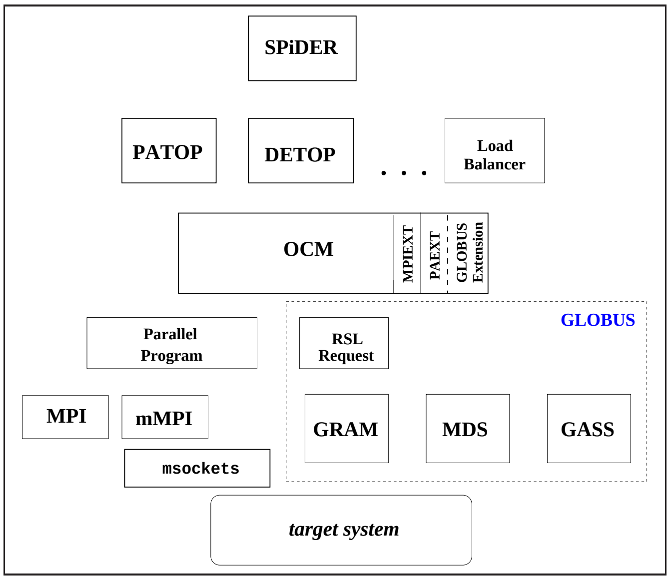

Figure 1: Architecture of the parallel tool environment

In order to utilize distributed computing power more efficiently it is essential to allow process migration from one, heavily loaded host to another, idle host, in a way that also allows the original host to be serviced (i.e. rebooted) without a need to break computations. One of the systems developed to support such a dynamic load balancing is Dynamite, a Dynamic Task migration environment based on PVM [3]. As a rule, any parallel computation requires some communication and synchronization. In the migration environment, it needs to handle the communication between processes, in spite of the migration. It means that the communication libraries have to be rewritten.

A lot of parallel computing environments use TCP or UDP through the socket interface as a communication mechanism. We have developed a library which offers a programmer an interface compatible with the standard BSD socket, and allows migration of processes with open sockets. The draft architecture of the tool environment with Globus-bound modules and Globus elements used to support the operation of the Globus-oriented tools is shown in Fig. 1.

2 Parallel Tools Based on OMIS

2.1 Performance Measurement Tool

The concept of a new performance measurement environment being the goal of our research evolved through a number of stages. *Primarily*, we focused on an off-line performance analysis tool, i.e. the collection of performance data based on an a-priori set of evaluation goals, followed by a performance visualization phase [4]. The latter allows to correlate the data obtained – statistics, usage profiling, and time-lined diagrams – to the program's source code. The *second stage*, aimed at transforming the off-line tool into an on-line one, including enhancement of the monitoring subsystem (MS), built using OMIS [5]. This implied building an intermediate component between MS and tools we called *request controlling subsystem*.

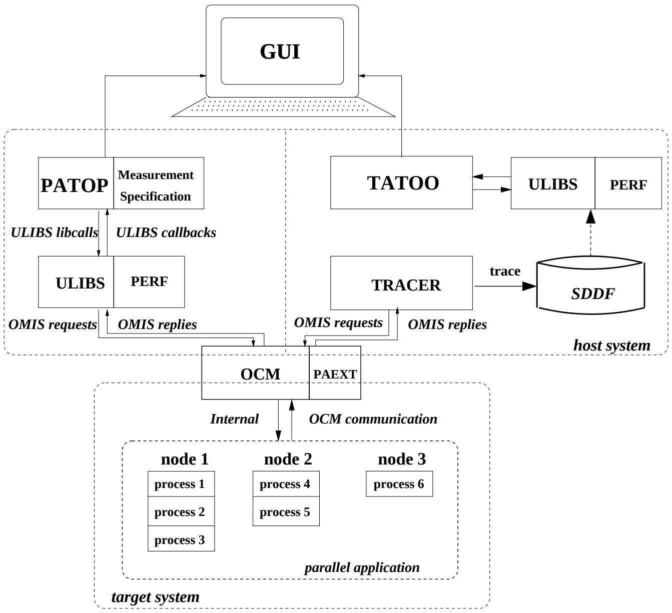

Figure 2: Operation of PATOP and TATOO on top of the OCM.

In order to fully use the OMIS's functionality, the *third* stage comprised a substitution of the existing MS for a comprehensive OMIS-compliant Monitoring System

(OCM), extending it to monitor MPI applications. The OCM consists of a central Node Distribution Unit and local monitoring processes (one per node), which handle the objects via shared memory. Part of the previously developed visualization mechanisms was incorporated into the existing performance measurement tools, PATOP and TATOO. Porting PATOP to the OCM involved an additional layer, called 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 [5] and reversely (Fig. 2).

2.2 Symbolic Debugging

SPiDER[7, 6] – an advanced symbolic debugging system for HPF, F90, and C parallel programs – makes it possible to control and monitor program processes at the source code level. A multiple process view of the program allows the user to either monitor and control a single process or to inspect the program as a whole. For HPF programs SPiDER provides a support for inspecting distributed data structures which are viewed as a single global entity, i.e., the user can inspect and modify a section or individual elements of distributed arrays without a need to specify which processor the elements reside on. As for HPF programs, distribution of data has a crucial impact on the efficiency of computations, the debugger provides advanced capabilities for examining these distributions. The sophisticated visualization of the system provides a graphical representation of array distribution with convenient navigation within distributed data arrays and logical processor arrays. It emphasizes mapping relationships between data and processors by using colors and highlighting elements residing on a selected processor.

Moreover, the data distribution steering capabilities allow to dynamically change the distribution of the array on stopping the program at a breakpoint. Visualization of values by assigning colors to array elements according to their values complements the view of the program's data. In addition, SPiDER can store up to seven snapshots of the array content and visualize differences between content of the given array at different states of the program. For complex applications, where the distribution of a given array changes multiple times during its execution, the user can see an animated replay of a redistribution sequence by using a distribution animator facility. With this facility the user can observe at each redistribution how array elements migrate from one set of processors to another.

SPiDER has a multilayered architecture (Fig. 3). At the machine level there is a Base Debugging Subsystem (BDS), built on the parallel debugger DETOP[8], providing a collection of conventional debugging functions. On top of BDS functionality, a Programming Model Dependent Debugging Subsystem provides more elaborate, higher level debugging functions, presenting the program execution and data at the HPF source code level. Finally, at the user level there is a Visualization Subsystem which consists of a GUI and the Graphical Data Distribution Tool (GDDT)[9] intended for distributed data visualization.

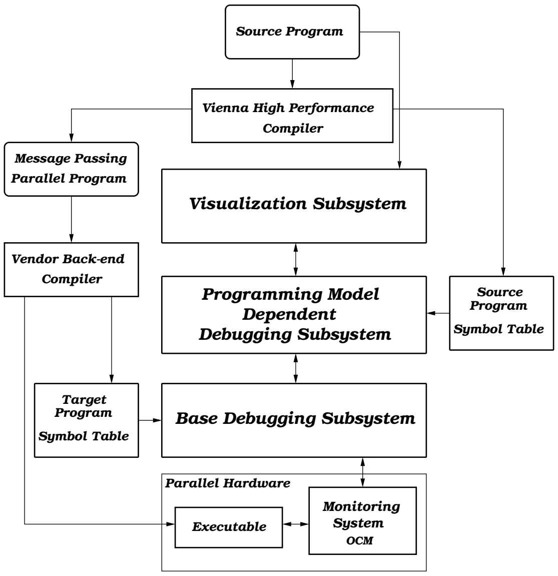

Figure 3: Structure of the SPiDER debugger.

2.3 Grid Implications for OMIS-based Tools

The main requirement put on such tools for metacomputing remains the same as for parallel and distributed computing: to present the execution of the program and its data in terms of the source code. This will be accomplished by extending the OCM capabilities by defining a set of new objects and services, which will cover the features of grid computations.

As the scope of HPC steadily expands from single parallel systems to highly distributed and large sets of heterogeneous systems, we are considering the changes in the functionality and interfaces, which the tool infrastructure under discussion must undergo in order to be adapted to grid computations. At the current stage we aim at monitoring Globus-oriented applications while using Globus services to support the monitoring environment operation [1].

When extending the operation of the OCM for grid computations, the following aspects should be considered. The set of monitorable objects within a grid should comprise the elements of an application being monitored as well as a set of resources the application is interested in. The application's execution may comprise a set of jobs, subjobs, processes and threads. The set of grid resources is represented by a number of services, which can be evaluated by access or service time. A tool system must be able to attach to a detach from running application: whereas in the original OCM this is possible via UNIX mechanisms, in a grid computation, this should be done via "standard" Globus services, e.g., GRAM. The same applies to starting/stopping, suspending/resuming an application with a tool, which is feasible with Globus elements. In order to provide a proper analysis of the resource requirements of a grid application, the OCM will be equipped with a Resource Specification Language parser.

While in conventional message passing applications the communication within the OCM environment is organized by means of PVM, this may be insufficient or even impossible in grid computations. They may lack the proper PVM infrastructure, which implies the necessity to introduce another protocol, e.g. a socket-based one for inter-OCM communication. The naming policy within the OCM infrastructure should be enhanced due to a different collection of object abstractions compared to the single parallel system, e.g. jobs, subjobs, etc. Monitoring information on a grid computation should be collected with the grid services, via the OCM.

Extending tools should take into consideration the following aspects: The volume of the data supplied, and its highly distributed, complex, high-dimensional character require the use of new data representation techniques, data clustering, and multi-level data transformations in order to present application and performance data in intuitive ways [1]. For the execution control there is required an extensive and easy to use process grouping capability that enables controlling and inspection of the processes by mapping their actual state to the programming model being used.

3 Socket Connection for Migrating Processes

3.1 Extensions Required for Socket Migration

To easily migrate processes, the new socket library (called `msocket`), which replaces the system one, should enable establishing new connections regardless of the number of migrating processes. All connections established before the migration should be kept, and data must not be lost. A parallel application with processes intended to migrate during runtime should not need to be developed in any special manner and all modifications to communication libraries have to be transparent. Packets coming to the migrating process may be redirected to a new process location using a daemon [3] or with a piece of code called *mirror* left at the previous location (computing node) [10]. The `msocket` library is based on these both concepts.

After process migration, all connections are redirected to the new process location and a mirror is created. The mirror captures and redirects only the packets on-the-fly during a process migration, and when, after process migration, connections through the mirror are flushed out, the mirror is removed.

On each compute node there is one daemon which propagates the redirect requests and resolves addresses.

In order to localize resources unambiguously, the application uses virtual addresses which are mapped onto the real IP addresses. The user application[1] (parallel programming evironments) always uses the virtual port and IP numbers, whereas the system TCP/IP stack uses only the real addresses.

The address translation from virtual to real ones needs a server that maintains relations between the virtual and real socket addresses and the time of the last use of

[1] except some specialized programs like proxy server

each address, which allows to remove unused addresses. The address server has to be a centralized part of the system because each address should be unique. To simplify the `msocket` library, there is a local address server on each node.

The `msocket` library wraps all calls within the user code to the TCP/IP stack so no changes in the system IP stack are required. The user code can not use TCP/IP stack directly. Checkpointing and restoring of a process is done with the Dynamite modules [3].

3.2 Description of the `msocket` operation

In Fig. 4 a connection scheme between two processes and the mirror is presented. The process `p2` has been moved from the node `n2` to `n3`. The process `p1` has two real sockets `r1` and `r1'` mapped onto one virtual socket `v1`. The process `p1` flushes the connection through the real socket `r1`. The process `p2'` also uses two sockets: one of them to connect directly to the process `p1` and the second one to get data from the mirror. As the `msocket` redirects a connection after migration of the partner process, it is necessary to keep two sockets for one connection.

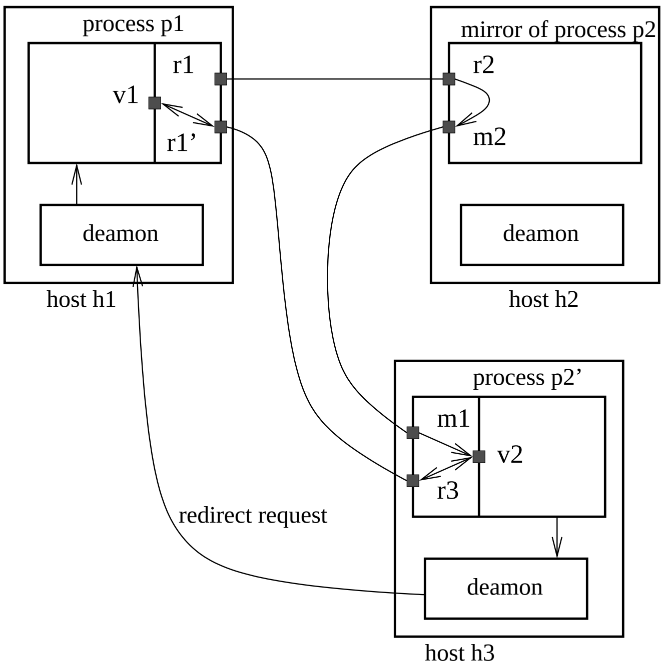

Figure 4: Data flow after migration.

A mirror is started when a process's state is being saved before migration. The migrated process starts work at the new node and connects to the mirror. The mirror works in a loop, it reads data from sockets inherited from the parent process and sends this data to the new process location. When a remote process (i.e. `p1` in Figure 4) redirects the connection (creates `r1'`) the process notifies the mirror and closes the old socket (`r1`). Data are redirected to a new process location through multiplexed connection. When a remote process flushes a connection (the send buffer is empty), the remote process closes the connection. The mirror is removed when the process which had left the mirror, has completed its execution, or all connections which were served by the mirror have been terminated.

A daemon on each node forces processes to redirect their connections. When a process migrates, it informs peers about its new location via daemons at the peers' hosts. When a program is aware of the IP and port numbers of the remote host, it can establish communication with the process through TCP or UDP protocol [11]. These numbers are passed to the socket library and then, in the same form, to the system network stack [12].

To inform the server of the client's virtual addresses, `msocket` sends the client's virtual IP, port to the server when a connection is being established. For TCP connections, this structure is sent only once while for UDP it has to be sent with each packet.

After the process migration each application's read request is a bit complex. If the mirror process (using the old process's connection) has not flushed the connection yet the `msocket` reads data from the mirror. Reading from the mirror can deliver data to other socket than that the application request wanted to read (because the connection between the mirror and the process is multiplexed). This sort of data will be stored in a buffer associated with the socket–recipient of the data. Whenever the mirror finishes serving the connection (the buffer is flushed and the peer process closes the old real socket) the application request performs a read from the buffer and afterwards from the new socket.

During the connection redirect, when the process creates new socket, i.e. the `r1'` socket in Figure 4, it is possible only from the `p1` process addressing space. There is no synchronization between the migration of the process `p2` and the process `p1` life so redirection activities are asynchronous and could be done from a signal handler. In this example the daemon at node `n1` uses the `kill()`[2] system call to force redirection.

To allow migrating processes to connect to a program which uses a system socket library, a proxy server is called for and the real IP and port numbers are used.

4 Concluding Remarks

Due to a recent interest in grid computations there arises a problem of tool support for development of applications, while preserving and improving the functionality of the existing parallel tools. The tool infrastructure discussed in the paper is built on top of a versatile monitoring system, the OCM, being an implementation of a common tools/monitoring system interface OMIS. This allows a considerable saving of efforts needed to develop, on one hand, a tool environment, on the other hand, a monitoring

[2]daemon has `root UID` or all migrating processes have the same `UID`

system to provide monitoring activities proper to the particular tool requirements. Grid computations being supported by appropriate software environments imply substantial changes in the low level modules, which until now used standard UNIX facilities as well as in the functionality of the tools. In particular, this refers to the operation of manipulation and information services provided by the OCM, which, in a grid environment, will be supported by available Globus services, e.g. MDS, GRAM or GASS. The upper OCM layer, NDU, should be able to parse a resource request expressed by RSL strings. Adding proper event services is connected mainly with instrumentation and the operation of the upper layer of the OCM.

Now we are working on a prototype implementation of *mpich* using the socket library. This research is aimed at validating whether the design above is realizable, while a possible success will show under which conditions this is feasable. Our migrating socket library is low-level, being usable for any parallel and distributed application where communication is realized through sockets. It is possible to modify parallel environments like PVM or MPI to use `msocket` library instead of the system ones, and, of course, with Globus.

Acknowledgements. We would like to thank Prof. Jacek Mościński for valuable discussions and to Dr. Jarek Nabrzyski for his suggestions. Our sincere thanks also go to Mr. Jacek Niwicki for the encouragement. This research was done in the framework of the Polish-Dutch-German-Austrian collaboration and it was supported in part by the KBN grant 8 T11C 006 15.

References

[1] Kesselman, C. and Foster, I. (eds.): *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[2] 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.

[3] van Albada, G.D., Clinckemaillie, J., Emmen, A.H.L., Gehring, J., Heinz, O., van der Linden, F., Overeinder, B.J., Reinefeld, A., and Sloot, P.M.A.: Dynamite – Blasting Obstacles to Parallel Cluster Computing. In: Sloot, P.M.A., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 12-14, 1999, Springer, Lecture Notes in Computer Science 1593, (1999) 300-310.

[4] Bubak, M., Funika, W., Iskra, K., Maruszewski, R., and Mościński, J.: Off-line and On-line Performance Monitoring Tools. In: *HiPer'98 Conference – High Performance Computing on Hewlett-Packard Systems*, ETH Zurich, Switzerland, October 14 - 16, (1998), 200-206.

[5] 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.), *Proceedings of 6th European PVM/MPI Users' Group Meeting*, Barcelona, Spain, September 1999, Lecture Notes in Computer Science 1697 (1999) 67-74, Springer.

[6] Brezany, P., Czerwiński, P., Koppler, R., Sowa, K., Volkert, J.: Advanced Visualization and Data Distribution Steering in an HPF Parallelization Environment. In *Proc. of Parallel Computing ParCo99 Conference*, Delft, The Netherlands, August 1999.

[7] Brezany, P., Grabner, S., Sowa, K., Wismüller, R.: DeHiFo – An Advanced HPF Debugging System. In *7th Euromicro Workshop on Parallel and Distributed Processing, PDP'99*, Madeira, Portugal, February 1999, 226 – 232, IEEE CS Press.

[8] Wismüller, R., Oberhuber, M., Krammer, J. and Hansen, O.: Interactive Debugging and Performance Analysis of Massively Parallel Applications. *Parallel Computing*, 22 (1996) 415-442.

[9] Koppler, R., Grabner, S., and Volkert, J.: Visualization of Distributed Data Structures for HPF-like Languages. *Scientific Programming*, spec. issue *High Performance Fortran Comes of Age*, 6 (1997) 115–126.

[10] Zandy, V.C., Miller, B.P. and Livny, M.: Process Hijacking. Computer Sciences Department University of Wisconsin, February 1999, http://www.cs.wisc.edu/paradyn/papers/

[11] Stevens, W.R.: *UNIX®Network Programming.* Prentice Hall, Inc. Englewood Cliffs, New Jersey USA, 1990.

[12] Stevens, W.R.: *TCP/IP Illustrated, Volume 1 – The Protocols.* Addison-Wesley, 1994.