

Grid environment for on-line application monitoring and performance analysis

Bartosz Baliś^a, Marian Bubak^{a,b,*}, Włodzimierz Funika^a, Roland Wismüller^c, Marcin Radecki^b,
Tomasz Szepieniec^b, Tomasz Arodź^b and Marcin Kurdziel^b

^a*Institute of Computer Science, AGH, Kraków, Poland*

^b*Academic Computer Centre CYFRONET AGH, Kraków, Poland*

^c*Universität Siegen, Siegen, Germany*

Abstract. This paper presents an application monitoring infrastructure developed within the CrossGrid project. The software is aimed at enabling performance measurements for the application developer and in this way facilitating the development of applications in the Grid environment. The application monitoring infrastructure is composed of a distributed monitoring system, the OCM-G, and a performance analysis tool called G-PM. The OCM-G is an on-line, grid-enabled, monitoring system, while G-PM is an advanced graphical tool which allows to evaluate and present the results of performance monitoring, to support optimization of the application execution. G-PM supports build-in standard metrics and user-defined metrics expressed in the Performance Measurement Specification Language (PMSL). Communication between the G-PM and the OCM-G is performed according to a well-defined protocol, OMIS (On-line Monitoring Interface Specification). In this paper, the architecture and features of the OCM-G and G-PM are described as well as an example of use of the monitoring infrastructure to visualize the status and communication in the application, to evaluate the performance, including discovering the reason of the performance flaw.

1. Introduction

The focus of the CrossGrid project [6] is to provide services and tools for Grid interactive applications. The development process of such applications requires specialized tools including performance analysis tools that would be supported by an application monitoring service. The need of such kind of tools is even stronger since contemporary grids provide a very limited access to the resources where an application is running. Typically, the results are not known until an execution is finished and practically only a very laconic status of the job is available.

From the perspective of the application developer the situation is difficult – on the one hand he/she should develop an application that performs well on various

types and configurations of resources available on the Grid; on the other hand, a crucial issue for improving an application – measurements of the performance are hardly available.

Having this in mind, within CrossGrid, we develop a monitoring environment for applications which is composed of a distributed monitoring system, the OCM-G and a performance analysis tool, G-PM. The purpose of this environment is to collect data about running application and enable the user to observe its performance in on-line mode, dynamically change measurements to support discovering reasons of performance problems, etc.

In this paper, we describe the architecture of the environment being discussed as well as explain the main features of the OMIS-based interface that is used for expressing monitoring requests. Next, we show examples of how the OCM-G and G-PM can be used for monitoring and performance evaluation of Grid applications. The examples include also construction of user-defined metrics expressed in the Performance

*Corresponding author: Marian Bubak, Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland. Tel.: +48 12 617 39 64; Fax: +48 12 633 80 54; E-mail: bubak@agh.edu.pl.

Measurement Specification Language (PMSL). Additionally, we provide a comparison with related work.

2. The application monitoring infrastructure

In this section we give an overview of our approach for a grid application monitoring infrastructure. An architecture and components with their functionality will be described in detail, not passing over a way of exchanging the monitoring data between the components that is based on a standardized protocol, OMIS, on which we focus below.

2.1. OMIS – A versatile monitoring interface

The interface specification between a tool and an application monitor (monitoring system) is extremely important, since this should allow to define monitoring activities easily and be extensible. On-line Monitoring Interface Specification (OMIS) [13] meets these requirements well.

The target system, as viewed by OMIS, forms a hierarchy of objects. In case of grid environments, the top-level objects are *sites* which contain *nodes* which in turn could be comprised of *processes*. The objects are identified by so called *tokens*.

OMIS defines three types of services: *information* services – to obtain information about objects, *manipulation* services – to manipulate objects, and *event* services – to detect events in the target system (especially inside applications), whose occurrence can trigger some defined *actions*.

By combination of services a *monitoring request*, belonging to one of two types can be formed: *unconditional*, which comprises one or more actions, and *conditional*, which is composed of one event service and one or more actions. In case of unconditional requests, the actions are executed immediately and only once, while in case of conditional requests the actions are executed whenever the associated event occurs, which can take place multiple times.

A very useful feature of OMIS, especially for the performance analysis, is the ability to create objects that store the performance values inside the monitoring system. These are so called *counters* and *integrators* which can be used for example to count the function calls or to sum the volume of data sent.

2.2. OCM-G – A distributed monitoring system

The OCM-G (OMIS-Compliant Monitoring system for Grid) is a scalable distributed system for on-line monitoring of interactive applications in the Grid environment.

The architecture of the system is shown in Fig. 1. It is comprised of several components that are distributed over the Grid to accompany the monitored application and to enable efficient gathering of monitoring data as well as to distribute monitoring requests. On the node level Local Monitors (LMs) contact monitored processes and obtain the requested data using various mechanisms such as `ptrace` and direct communication with processes. The latter is realized via a module (library) linked to the application. The library is called Application Module (AM) and is considered to be a part of the OCM-G monitoring infrastructure. The LMs from several nodes are connected to a Service Manager (SM) that is typically located one per Grid site. The role of this component is to distribute a monitoring request to the underlying LMs and to collect monitoring data from them. Moreover, SM improves the scalability of the OCM-G and enables monitoring in clusters using private IP addresses (in this case SM should be placed on the machine with public and private interfaces). On the top of the components hierarchy of the OCM-G, the Main Service Manager (MainSM) is placed. This is a single component that keeps connections to all SMs running in the monitoring system and acts as an entry point to the monitoring system for tools.

The OCM-G is designed to be a user-private monitoring system which means that all OCM-G components are running using the privileges of the user and only the user is empowered to use them. Authorization and authentication is based on the Grid Security Infrastructure [9].

A major part of performance data is obtained from instrumentation of communication libraries. The OCM-G is provided with a tool for automatic instrumentation of MPI libraries that is independent from an MPI implementation. By default, a static instrumentation is inactive. A selected range of instrumentation can be activated only on a request from the tool, when the application is running. In the same manner, probes – a manually inserted code for monitoring – can be activated and deactivated dynamically.

One of the unique features of the OCM-G is support for interoperability. This is a virtue of the OMIS interface implementation which provides that the tools are capable of interacting with the monitoring system

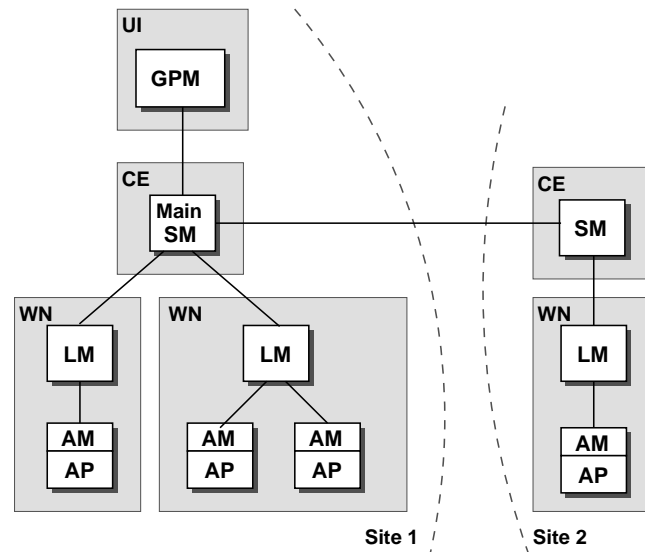


Fig. 1. Architecture of the G-PM and OCM-G.

and accessing the data cooperatively, thus several tools may operate on the same application in parallel. This opens the door to synergy of various tools working together, e.g. a performance analyzer with an automatic load balancer.

For a more detailed description of the monitoring services provided by the OCM-G and some design details concerning high efficiency, low intrusiveness, accuracy of measurements, transparency for the user, we refer the reader to [2].

2.3. G-PM – Grid-oriented performance evaluation tool

A Grid-oriented Performance Measurement tool (G-PM) is designed to support performance measurements of the application's execution and of the underlying grid infrastructure. The monitoring data are obtained from OCM-G via the OMIS-based protocol. The same interface is used also for requesting the data needed to measure the metrics that the user enabled. The measured values are displayed in the form of various graphs, such as bar graphs, pie charts, multi-curve plots, histograms or communication matrices.

The architecture of the G-PM is depicted in Fig. 1. It consists of the User Interface and Visualization Component (UIVC), the Performance Measurement Component (PMC) allowing for measurements based on standard metrics, like "data volume", "time spent in", and the High Level Analysis Component (HLAC), which provides a powerful facility for the user to define own

metrics and realize measurements, meaningful in the context of the investigated application.

The tool supports a wide set of built-in metrics. These metrics fall into two categories, i.e. *function-based* and *sampled* metrics. The function-based metrics are associated with the instrumented versions of specific functions. They are suitable for monitoring the behavior of applications with respect to various libraries they use. For example, the metrics based on instrumented versions of the MPI library enable the monitoring of application communication and overhead due to parallelization. The sampled metrics, on the other hand, are not related to any functions or events. They enable monitoring of such quantities as the CPU load on the node or memory used by the process.

The G-PM can be customized by the user or application developer by means of a dedicated Performance Measurement Specification Language PMSL [18]. The language allows for processing and combining the existing metrics into a form more suitable for a particular evaluation. Furthermore, in PMSL a new category of metrics is available i.e. *probe-based* metrics. These metrics are associated with a call to a function inserted into the application's code by the developer, i.e. the *probe*. The probes are typically used to signal the G-PM that the control flow of the application reached a code location specified by the developer. Furthermore, probes can pass additional information from within the application to the G-PM, e.g. values of some internal variables. An intuitive example of using a probe is a metric which measures a solver residue

in consecutive iterations. In this case a probe is inserted at the end of the main solver loop which, when invoked, sends the current residue value to the G-PM for processing and visualization.

2.4. Dealing with the monitoring environment

Being familiar with main characteristics of the OCM-G and G-PM, now we would like to explain the procedure of starting up the monitoring infrastructure. We focus on the user's actions that should be done to enable the monitoring of the application as well as the way the components are started thus forming the monitoring infrastructure.

From the user's perspective there are only two objects of interest: the application intended to run and the tool the user wants to obtain the monitored data with. The start-up of the environment was designed to be as easy as possible, however there are some special actions that the user needs to take for enabling application monitoring.

Firstly, the application should be prepared for monitoring by relinking it with additional libraries: a monitoring library, an instrumented communication library (MPI) and optionally with a static instrumentation of a probes code. Except for inserting the probes in the source code, the rest of necessary actions are done automatically by a script. Next, prior to starting the application, the OCM-G's MainSM should be spawned (step 1 – the number refers to Fig. 2). This component serves as a name service for OCM-G components and a gateway to the monitoring system for tools. The MainSM delivers the *connection string* which serves as a contact information for other components to establish communication with MainSM.

When MainSM is running, the application can be started (step 3) in the typical way, only two additional command-line parameters are required: the connection string obtained from MainSM (step 2) and an arbitrarily chosen application name. Using yet another parameter the user can make the application processes to run after registration in the monitoring service (step 5) or to wait until an explicit command from the user comes. This feature enables defining measurements and monitoring from the very beginning of an application's run.

While the application is starting up on the grid it executes a code from the Application Module. Inside there are instructions to fork off the local monitoring infrastructure (i.e. Local Monitors) (step 4). Next, the Local Monitors establish communication with Service Managers (step 10) or start them themselves if necessary.

Finally, a ready-to-use distributed monitoring system is formed. There are mechanisms that guarantee only one LM will be running on a single node and only one SM will operate for all nodes within a site. Details of the start-up process are shown in the Fig. 2.

In the last step, the G-PM can be started to make the monitoring environment complete. The user passes to it the connection string and the application's name, similarly as for the application (step 11). Based on this, the G-PM attaches to the application and obtains the list of processes with localization and the list of functions that helps the user define a measurement.

When the monitoring activity is finished, the whole infrastructure can easily be shut down with one command from the tool. Additionally, there are mechanisms for shutting-down the OCM-G's components in case when MainSM is closed for any reason, which guarantees a full clean-up in case of a crash. It is worth mentioning that shutting-down the components of the OCM-G does not affect the application.

3. Examples of use

In this section, we present how the user can make use of the OCM-G and G-PM to observe the status and to get some generic data about an application running on the Grid. The examples are illustrated with the cases of real-world applications. The last part of the section contains a use-case on how to examine the application's performance and to determine potential sources of problems.

3.1. Grid application at a first glance

The OCM-G and G-PM can be easily used to determine some generic information about an execution of the application, e.g., to which nodes the application's processes have been submitted, if they are really running or whether the application makes progress well or not. Some data about the advance of computation can be valuable for the user who wants to estimate when the application would finish or for the developer to check the influence of the introduced improvements. Assuming that the typical application includes a main loop in which some computations are done, it is easy to make a progress indicator based on the number of loop iterations. At the end of the loop, we can insert a *probe*, e.g. `loop_ended()` and instruct the OCM-G to count probe calls in each process. This allows us to determine roughly the progress of an application and being given

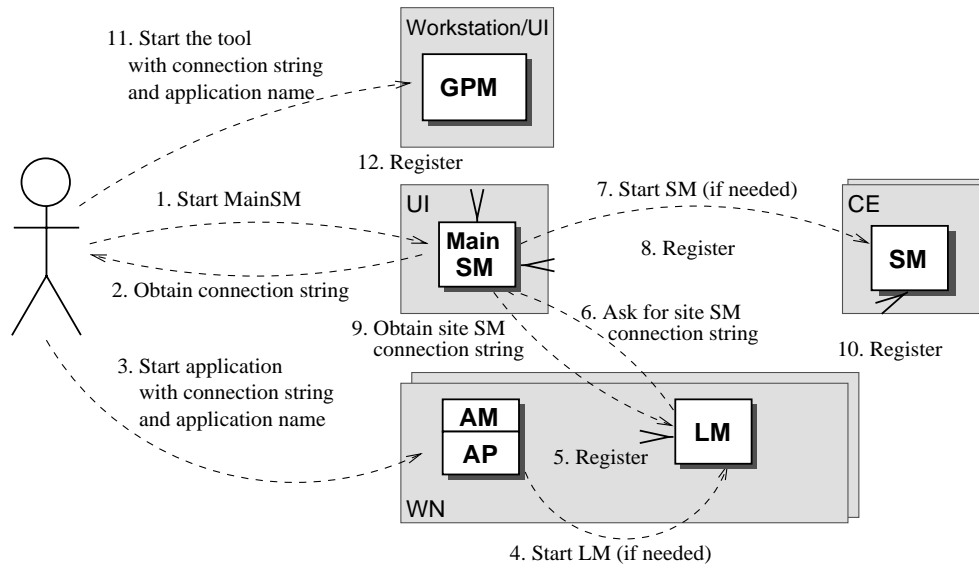


Fig. 2. Start-up of monitoring infrastructure.

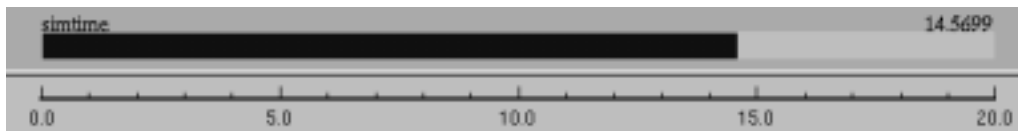


Fig. 3. Bar graph showing the application's progress (probe executions).

a total number of loop iterations, we can estimate the time when the application will terminate its execution.

There are also simpler ways to estimate the progress. We can even manage without inserting any probe, if only the main loop includes functions that are executed in each iteration. Typically, good candidates are these from a collective communication class like `MPI_Bcast()`. The OCM-G can be instructed to create a counter which will be automatically incremented whenever the program reaches a particular function call. A value of this counter can be periodically retrieved and then be plotted with a bar graph as an indicator of the execution progress.

A little more sophisticated example shows how the probes can be used to define an application's *simulation time*. In the application, a probe at the end of the time loop is inserted. The probe receives the simulated time as an additional parameter. A very simple use of this probe is a user-defined metric that just determines the application's progress, i.e. the current simulation time. Such a metric can be defined by the user via G-PM's Performance Metrics Specification Language PMSL in the following way:

```

Simulation_time(Process p,
VirtualTime vt) {
    PROBE loop_stop(Process,
        VirtualTime, double);
    double simtime;
    RETURN simtime AT loop_stop(p,
        vt, simtime);
}
  
```

This specification basically states that the value of the metric is a parameter `simtime` provided by the execution of probe `loop_stop()`. The value then can be visualized by G-PM, e.g. as a bar graph shown in Fig. 3, which acts as an on-line application progress bar.

When a metric defined in PMSL is requested to be measured by G-PM, its specification provided by the user is automatically translated into proper OMIS requests for the OCM-G. In the example, the OCM-G is instructed to monitor executions of the `loop_stop` probe and send G-PM the value of the `simtime` parameter. For a detailed discussion of the translation process, please refer to [18].

Another important information about a message passing application is a communication topology ex-

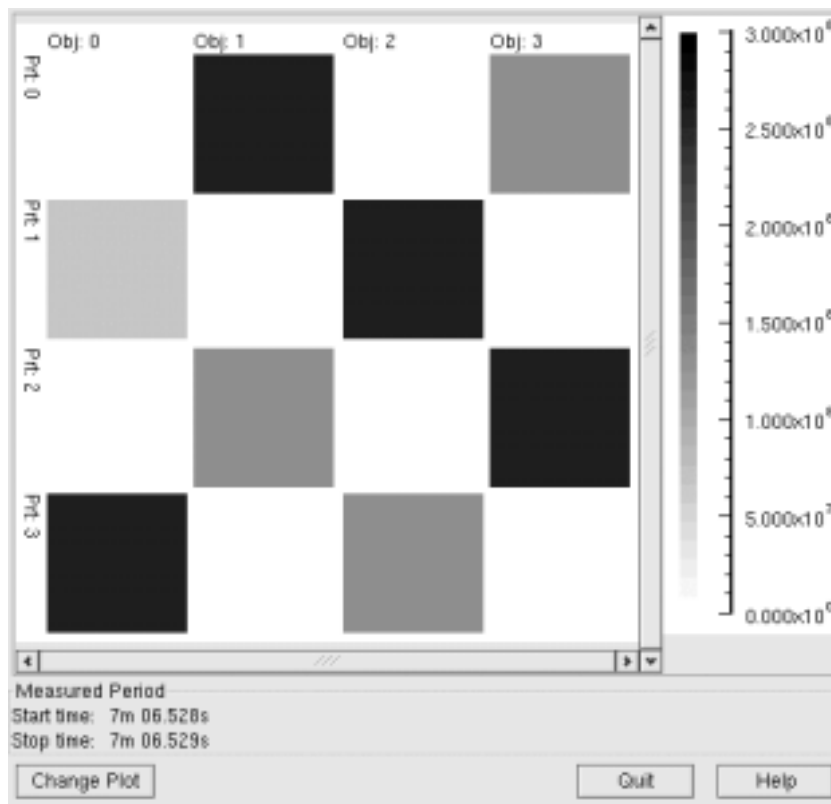


Fig. 4. Matrix diagram – communication pattern of the application.

plaining which processes communicate with which ones. Particularly, if the user must deal with the application that was written some time ago, or even worse, by the authors who cannot provide support. Dealing with the performance of a program without having a detailed knowledge about its structure and communication patterns is a hard task.

The G-PM provides standard metrics related to data volumes sent using different MPI calls. Additionally, with these metrics we can specify *partner objects*. A partner object in case of `MPI_Send()` means the process the message is sent to, in `MPI_Recv()` it would be the process the message was sent from. By use of partner objects we can distinguish between messages sent in each process-to-process link.

The result of measurements with partner objects is shown in Fig. 4. The chart is *matrix diagram* type, it presents the details on communication volumes sent between all communicating processes, where the intensity of communication is expressed by the extent of box grayness. This diagram is an example of monitoring a bidirectional ring application. Here, we have four processes, each of them communicates with two

neighbors. We can see that the communication in the “forward” direction (from a lower rank to a higher one) is more intensive than in the “backward” direction. It is also noticeable that the process with rank “0” sends “forward” a bit less data than others. A communication pattern diagram could discover problems if there were processes which do not communicate with others at all.

3.2. Performance evaluation

Beside providing generic information on the application’s execution as shown in the previous paragraph, the OCM-G/G-PM enable the user to examine a wide spectrum of parameters to evaluate the application’s performance, for example communication delays or CPU usage. Moreover, when a performance problem is found, the monitoring infrastructure provides means to drill down the affected code to find the reasons of flaw.

A good starting point to assess the application’s performance is to check whether the communication delays are uniform. In Fig. 5 we can observe a distribution of receive delays over percentage ranges. There is a group of two processes with delays at the beginning

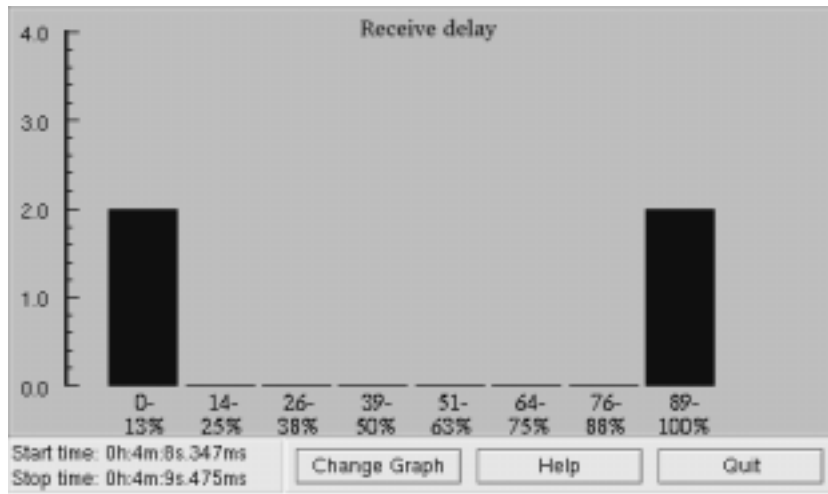


Fig. 5. Histogram of delays in MPLRecv().

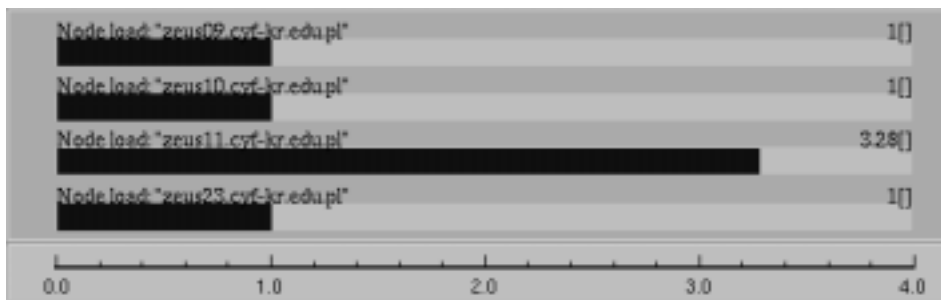


Fig. 6. Machine load during the ring application execution.

of the lowest range, but there is also a group at the other end of the highest range which may indicate a performance flaw.

We put forth a hypothesis that this can be caused by worse execution conditions which can be checked by monitoring the load on the machines where processes are executed. The results are depicted in Fig. 6 where we can observe that one of the nodes has a significantly higher load than the others. This causes that one process is not keeping up with the others and sending the data later than the partners expect being the reason for higher communication delays.

The last example in this section presents a comparison of an application's execution on several grid sites with a variable configuration of processors and network connectivity in the CrossGrid testbed. The application at hand was a development version of the blood flow simulation kernel [7] developed within the CrossGrid project. In Fig. 7 an application's progress at a given site is presented. Each curve corresponds to one site, on the y-axis there is the number of iterations of the

main loop, while the x-axis is a time line. All applications were suspended after registration in the OCM-G and then resumed in the same moment by the OCM-G, so we can compare it directly. At the beginning, the application executes quite well, but it reaches a *transition point* after which the execution speed (iterations per time unit) significantly decreases. It could be natural (resulting from an application structure itself), but one thing should make us worry – one site does not exhibit such a behavior (loki01.ific.uv.es). We are going to solve the riddle in the next section dedicated to finding performance flaws.

3.3. Finding performance flaws

It could be said that the usefulness of performance analysis tools can be proved by checking to what extent is possible to detect and help to fix performance flaws with these tools. In this section we follow through an investigation of a problem with the blood simulation application mentioned in the previous paragraph. Such

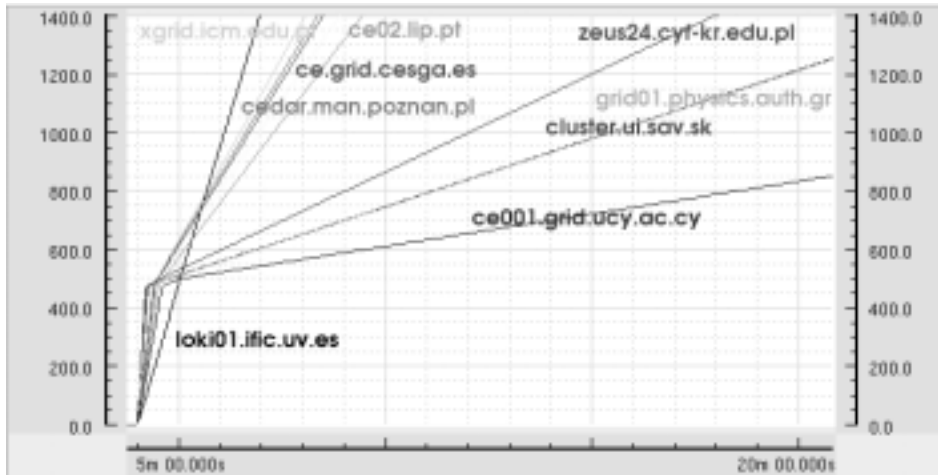


Fig. 7. Performance of the blood stream application at several grid sites.

finding a cause of the problem can be greatly facilitated by the possibility to narrow the area of search in the code. The G-PM/OCM-G supports such an approach. Having access to the application source code, the user can insert OCM-G probes into a part that potentially does not perform well and relying upon the probes the user can build measurements which work only in the scope of a selected code. Such a procedure can be applied iteratively to narrow the area of performance flaw search.

As we can see in Fig. 7, our tested application runs well until about five hundred iterations is performed, when the time of one loop iteration noticeably increases. There are several methods (functions) called in the loop and our goal is to figure out which one is responsible for consuming considerably more time. To find a candidate, we divided the loop code into three parts and inserted probes at the beginning and the end of each one as shown below.

```
for(i = 0; i <= ge.niter; i++) { ...
    start1(i);
    ge.body_force_x (local_dp);
    /*"lbee.h" */
    start2(i);
    ge.propagation (); /*"lbee.h" */
    start3(i);
    ge.bounce_back_links();
    ge.coll(ge.tau, i);
    end3(i);
... }
```

In the G-PM we created a user-defined metric in the PMSL language. This metric states that each

time the probes are executed a difference between the call time of `start3()` and `end3()` will be returned, which means that the measured value will contain just a wall clock time the program spent in between the probes. Such a metric was also defined for the remaining probes.

```
Time_two_probes3(Process p,
VirtualTime t) {
    PROBE start3(Process,
VirtualTime);
    PROBE end3(Process,
VirtualTime);
    Value time;
    time = Time(NOW) AT end3(p,t)
        - Time(NOW) AT start3(p,t);
    RETURN time;
}
```

The results of such a measurement expressed as a time percentage of three parts within the loop show that the application spends most of the time in the third part (ca. 75%), i.e. between `start3()` and `end3()` which contains two method calls. Repeating this step again we determine that it is the `ge.coll()` that consumes a major part of time.

Knowing this we can continue to drill down in the `ge.coll()` method or we can use an alternative method. We can obtain some parameters from the application meaningful in the application's context and influence the amount of computation. In this case application computes interactions between objects (collisions), so we can measure how many of them was computed in each iteration. Figure 8 shows the wall

clock time needed for each execution of the affected function (bottom picture). At some point the execution time grows by a factor of 10. The upper plot shows the total number of executions of the innermost loop body, and the number of collisions with fluid and solids which are computed in each iteration. They take turns conditionally and the collision with fluid is much more CPU-consuming than with a solid. As we can see in the upper figure the number of collisions are constant, so the amount of computation remains on the same level, still not discovering the cause of bad performance.

In Fig. 9 we see that a longer execution time is not an effect of scheduling (e.g. a less priority to the job, or background load etc.). Again, the lower curve is the execution time of our function, the upper is the (total) CPU time of the process. The curve does not flatten (it even gets a bit steeper at the transition point), so it means that the process is still running at full speed. In this example, one can also observe a significant variance in the execution time, which may indicate a cause in the hardware.

In fact, just in hardware there is the source of the problems. It turned out that after several hundreds of iterations the application starts to process *Not a Number* values. In contrast to normal floating point operations, the NaNs are very badly handled by Intel Pentium processors causing a decrease in the application's performance. Athlon processors do not show any difference in operations on NaNs. This also explains why on one site the application did not exhibit a performance problem. This case is depicted as a completely straight line in Fig. 7 (site *loki01.ific.uv.es*).

4. Related work

In this section we would like to focus our attention on other's activities in a field of grid application monitoring. However, we do not intend to provide exhaustive analysis. For an extended report on this subject, please refer to [19,20].

To explain the background of the approach under discussion, we should note that our work on application monitoring begins with the first implementation of an OMIS-compliant monitor – the OCM that supported cluster environments and message passing PVM and MPI applications. Also several tools were implemented on top of OMIS/OCM: PATOP for performance analysis, DETOP for debugging, and others. The development of the OCM-G and G-PM is a direct continuation of the previous work, started in 2001 with a first proposal of an application monitoring system for the Grid [3].

4.1. Grid application monitoring systems

There are several existing efforts to enable monitoring of grid applications such as Autopilot [17] in the GRaDS project [10], GRM/R-GMA [16] in the DataGrid project [8], and GRM/Mercury [15] in the GridLab project [11].

The goal of the application monitoring environment in the GRaDS project is to enable an adaptive environment in which the system parameters are adjusted at run-time to sustain a high performance of the running application. Performance information about both the application and the system is combined and a performance optimization action is automatically taken (e.g., I/O buffers are resized) when a performance loss situation is discovered. Thus, the goal of the GRaDS/Autopilot environment is rather different than ours. It is oriented towards automatic steering rather than providing feedback to the user so that he can see how his application performs and perhaps discover bottlenecks, weaknesses in the algorithm, bugs, etc.

In the GRM/R-GMA environment, the GRM monitor is used to collect traces about a running application. The trace data is published in the R-GMA [14], and used by tools to visualize the application behavior. However, the R-GMA infrastructure, based on java servlets technology, is rather heavy-weight and introduces a relatively large overhead. Additionally, the use of traces probably prevents from achieving low-latency and low intrusion at the same time: either traces are rarely collected and the latency increases or they are frequently collected and the intrusion is higher. This makes this solution appropriate for batch-oriented applications, where a post-mortem analysis is suitable, rather than interactive ones.

In GridLab, a similar concept as in DataGrid is used, though the R-GMA is replaced by a more efficient infrastructure – the Mercury monitor.

4.2. Performance tools for grid applications

The user assesses the usability of tools, based on the extent of provided functionality and ease of use: the user is interested in a combination of powerful performance analysis of the grid infrastructure with analysis of the application at multiple abstraction levels, all through possibly the same/similar graphical visualization scheme.

The existing tools developed for support of grids, such as NetworkWeatherService (NWS) [29] or Globus Monitoring and Discovery Service (MDS) [30], are

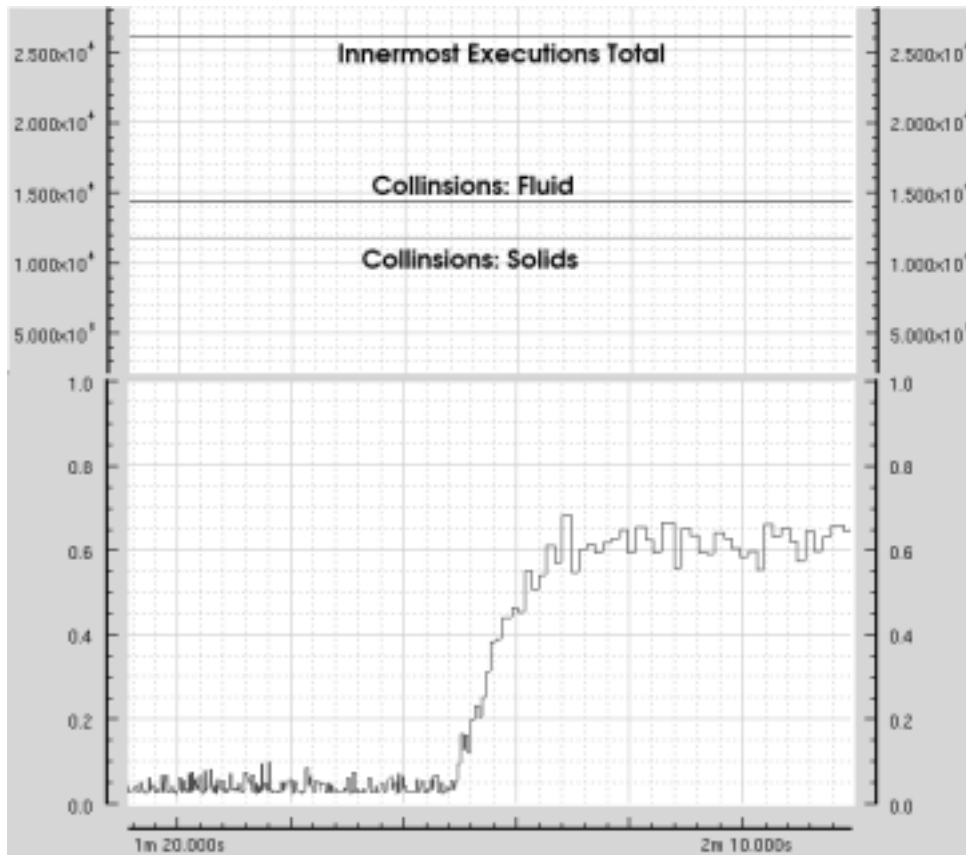


Fig. 8. Amount of computation (upper) vs. wall clock time of `ge.coll()` method (lower).

limited to monitoring the grid infrastructure. The NetLogger-based GridMapper [31] combines the infrastructure information with limited event-based information from applications, but the visualization of the data is oriented towards a geographical location of events. Thus, the tool is intended for control and monitoring of the state of the whole grid rather than for analyzing performance of a single application. An ensemble of tools: SvPablo, Autopilot and Virtue, while combining both the infrastructure and application data, are mainly designed for non-specialists.

Another feature of tools is the mode in which performance data is gathered. Existing tools use either relational database (DataGrid's R-GMA) or LDAP (Globus MDS), which can be then queried. In case of tools like NWS or MDS, changes in gathered information require a change to the configuration and cannot be done instantly. Paraver [21] and Pablo [22] are two of the very few tools that support a user-configurable data analysis. While Paraver uses a menu-driven approach, Pablo is based on graphical programming. However,

in both cases, only an off-line processing of trace data is supported.

There are tools that support configurable metrics, but these are used to simplify the internal implementation only. It is not possible for the user to specify own metrics, not only because there is no user interface for it, but mainly because of a low level of specifications, i.e. due to a high dependence on an implementation, defining an own metric is too complex for the user. An example is the Paradyn tool [23], which uses a metrics definition language called MDL [24] to define all the on-line metrics it allows to measure. Similarly, being targeted at the off-line examination of trace files, the EXPERT performance analysis tool [26] supports configurable metrics by using a language named EARL [25]. High-level specification languages have also been proposed for automatic bottleneck detection, where they are used to describe performance properties, which are similar, but not really identical to (user-defined) metrics. The most prominent examples are ASL [27] and JavaPSL [28]. At the moment, ASL has not yet been implemented and JavaPSL is implemented only for off-line evaluation.

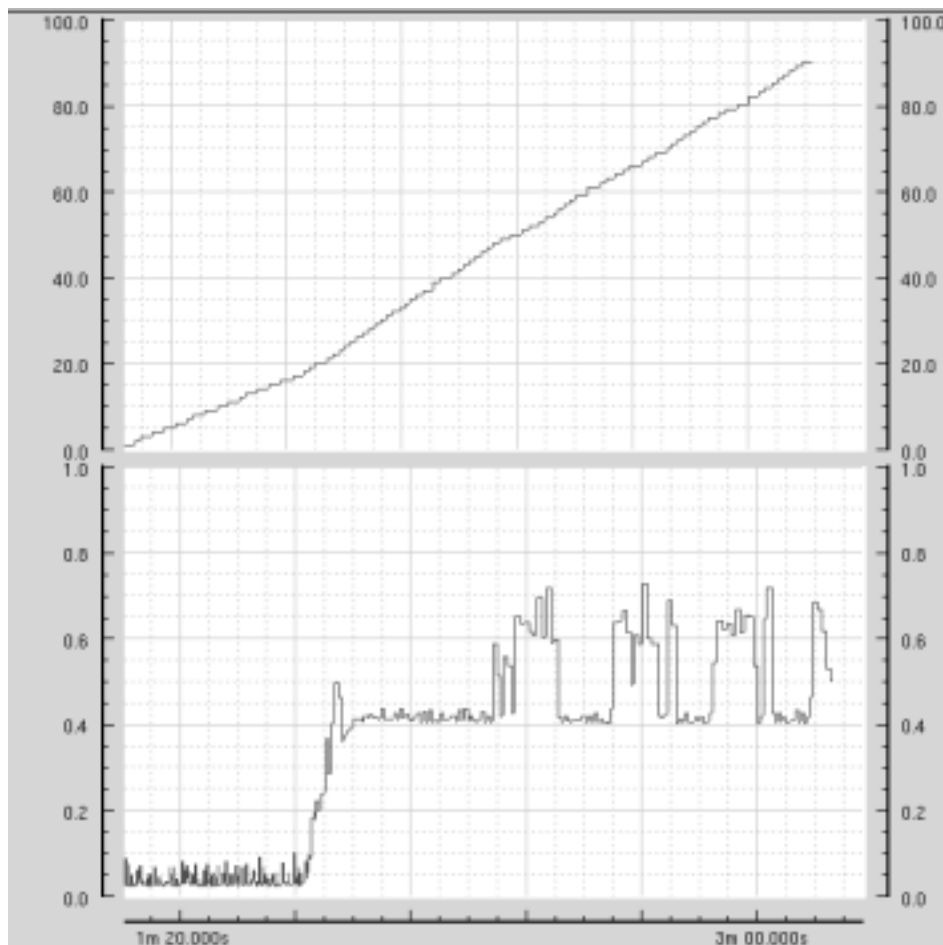


Fig. 9. Progress of the blood stream application (upper) vs. wall clock time of `ge.coll()` method (lower).

5. Conclusions

We have presented a grid monitoring infrastructure for interactive applications being developed within CrossGrid. Its two main components were described: the OCM-G – a grid-enabled, distributed monitoring system and the performance evaluation tool G-PM. Also a general view how the user interacts with the monitoring system was shown.

A use of a wide range of monitoring techniques with a variety of metrics and visualisation diagrams was illustrated with a number of examples. This facilities enable the user to achieve a required goal of monitoring. To sum up, the OCM-G/G-PM could be useful not only for the typical application user but also for the grid application developers who wish to improve the performance of their application.

When compared to other grid monitoring and performance analysis tools, our approach noticeably provides

unique features. As probably the most important one we should mention the possibility to control the process of monitoring in the runtime and the ability to create user-defined metrics, beside a wide range of standard ones, which makes the G-PM tool and the underlying monitoring infrastructure, the OCM-G, widely extendible by new ways of getting insight into the application behaviour.

The future of the OCM-G/G-PM is determined by the evolution of grids which inevitably goes towards web-services and Java technology. Although support for Java applications seems to be troublesome, there are on-going efforts to migrate the OCM-G/G-PM to these platforms [32,33].

Acknowledgements

This work was partly funded by the European Commission in the framework of IST projects CrossGrid,

GRIDSTART and K-WfGrid as well as by the Polish State Committee for Scientific Research, SPUB-M 112/E-356/SPB/5.PR UE/DZ 224/2002-2004. We are also grateful to the CrossGrid Integration Team led by Jesus Marco, Harald Kornmayer, and Jorge Gomes for their invaluable help.

References

- [1] B. Baliś, M. Bubak, T. Szepieniec, R. Wismüller and M. Radecki, *OCM-G – Grid Application Monitoring System: Towards the First Prototype*, Proc. Cracow Grid Workshop 2002, Krakow, December 2002.
- [2] B. Baliś, M. Bubak, W. Funika, T. Szepieniec, R. Wismüller and M. Radecki, in: *Monitoring Grid Applications with Grid-enabled OMIS Monitor*, F. Fernandez Rivera, M. Bubak, A. Gomez Tato and R. Doallo, eds, Proc. First European Across Grids Conference, Santiago de Compostela, Spain, February 2003. LNCS 2970, Springer, 2004.
- [3] M. Bubak, W. Funika, B. Baliś and R. Wismüller, *Concept For Grid Application Monitoring*, in Proceedings of the PPAM 2001 Conference, vol. 2328 of Lecture Notes in Computer Science, Naleczow, Poland, September 2001. Springer, pp. 307–314.
- [4] M. Bubak, W. Funika and R. Wismüller, *The CrossGrid Performance Analysis Tool for Interactive Grid Applications*, Proc. EuroPVM/MPI 2002, Linz, September 2002.
- [5] M. Bubak, W. Funika, B. Baliś and R. Wismüller, On-line OCM-based Tool Support for Parallel Applications, in: *Annual Review of Scalable Computing*, (vol. 3), (chapter 2), Y. Chung and Kwong, eds, World Scientific Publishing Co. and Singapore University Press, Singapore, 2001, pp. 32–62.
- [6] The CrossGrid Project (IST-2001-32243): <http://www.eu-crossgrid.org>.
- [7] CrossGrid biomedical application's Web page, <http://www.eu-crossgrid.org/biomedical.htm>.
- [8] The DataGrid Project: <http://www.eu-datagrid.org>.
- [9] I. Foster, C. Kesselman, G. Tsudik and S. Tuecke, *A Security Architecture for Computational Grids*, in: Proc. 5th ACM Conference on Computer and Communications Security Conference, 1998, pp. 83–92.
- [10] The GrADS Project: <http://hipersoft.cs.rice.edu/grads>.
- [11] The GridLab Project: <http://www.gridlab.org>.
- [12] P. Kacsuk, *Parallel Program Development and Execution in the Grid*, Proc. PARELEC 2002, International conference on parallel computing in electrical engineering, Warsaw, 2002, pp. 131–138.
- [13] T. Ludwig, R. Wismüller, V. Sunderam and A. Bode, *OMIS – On-line Monitoring Interface Specification (Version 2.0)*. Shaker Verlag, Aachen, (vol. 9), LRR-TUM Research Report Series, 1997, <http://www.bode.in.tum.de/omis/>.
- [14] R-GMA: A Grid Information and Monitoring System, http://www.gridpp.ac.uk/abstracts/AllHands_RGMA.pdf.
- [15] N. Podhorszki, Z. Balaton and G. Gombas, *Monitoring Message-Passing Parallel Applications in the Grid with GRM and Mercury Monitor*, in: Proc. 2nd European Across Grids Conference, Nicosia, CY, To appear in Lecture Notes in Computer Science, Springer Verlag, 28–30 Jan. 2004.
- [16] N. Podhorszki and P. Kacsuk, *Monitoring Message Passing Applications in the Grid with GRM and R-GMA Proceedings of EuroPVM/MPI'2003*, Venice, Italy, 2003. Springer 2003.
- [17] J.S. Vetter and D.A. Reed, Real-time Monitoring, Adaptive Control and Interactive Steering of Computational Grids, *The International Journal of High Performance Computing Applications* **14** (2000), 357–366.
- [18] R. Wismüller, M. Bubak, W. Funika, T. Arodź and M. Kurdziel, *Support for User-Defined Metrics in the Online Performance Analysis Tool G-PM*, in: Proc. 2nd European Across Grids Conference, Nicosia, CY, 28–30 Jan. 2004, LNCS 3165, Springer Verlag.
- [19] M. Gerndt, Performance Tools for the Grid: State of the Art and Future. APART White Paper. Research Report Series, Vol. 30. LRR, Technische Universität München. Shaker Verlag, 2004.
- [20] S. Zaniolas and R. Sakellariou, A Taxonomy of Grid Monitoring Systems, *Journal of Future Generation Computer Systems*, to appear.
- [21] European Center for Parallelism of Barcelona. Paraver. Web page, <http://www.cepba.upc.es/paraver/>.
- [22] University of Illinois. Pablo Performance Analysis Environment: Data Analysis. Web page, <http://www-pablo.cs.uiuc.edu/Project/Pablo/PabloDataAnalysis.htm>
- [23] B.P. Miller et al., The Paradyn Parallel Performance Measurement Tools, *IEEE Computer* **28**(11) (Nov. 1995), 37–46, <http://www.cs.wisc.edu/paradyn/papers/overview.ps.gz>.
- [24] J.R. Hollingsworth, B.P. Miller, M.J.R. Goncalves, Z. Xu, O. Naim and L. Zheng, *MDL: A Language and Compiler for Dynamic Program Instrumentation*, in Proc. International Conference on Parallel Architectures and Compilation Techniques, San Francisco, CA, USA, Nov. 1997, ftp://grilled.cs.wisc.edu/technical_papers/mdl.ps.gz.
- [25] F. Wolf and B. Mohr, EARL – A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs, in: *Proc. of the 7th International Conference on High- Performance Computing and Networking (HPCN 99)*, A. Hoekstra and B. Hertzberger, eds, Amsterdam, The Netherlands, 1999, pp. 503–512.
- [26] F. Wolf and B. Mohr, Automatic Performance Analysis of MPI Applications Based on Event Traces, in: *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference*, A. Bode, T. Ludwig, W. Karl and R. Wismüller, eds, volume 1900 of Lecture Notes in Computer Science, Munich, Germany, Aug. 2000, pp. 123–132, Springer-Verlag.
- [27] T. Fahringer, M. Gerndt, G. Riley and J.L. Träff, Knowledge Specification for Automatic Performance Analysis. Technical report, ESPRIT IV Working Group on Automatic Performance Analysis, Nov. 1999, Web page, <http://www.fz-juelich.de/apart-1/reports/wp2-asl.ps.gz>.
- [28] T. Fahringer and C. Seragiotto, *Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL*, in: 9th IEEE High-Performance Networking and Computing Conference, SC'2001, Denver, CO, Nov. 2001.
- [29] R. Wolski, N. Spring and J. Hayes, The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, *Future Generation Computer Systems* **15** (1999), 757–768.
- [30] X. Zhang, J. Freschl and J. Schopf, *Performance Study of Monitoring and Information Services for Distributed Systems*, Proceedings of HPDC, August 2003. Web page, <http://www-unix.mcs.anl.gov/schopf/Pubs/xuehaijeff-hpdc2003.pdf>.
- [31] W. Allcock, J. Bester, J. Bresnahan, I. Foster, J. Gawor, J.A. Insley, J.M. Link and M.E. Papka, *GridMapper: A Tool for Visualization of the Behavior of Large-Scale Distributed Systems*, Proceedings of High Performance Distributed Com-

- puting 11 (HPDC-11), Edinburgh, Scotland, 2002. Web page, <http://www-unix.mcs.anl.gov/fl/publications/hpdc11-gridmapper.pdf>.
- [32] W. Funika, M. Bubak, M. Smętek and R. Wismüller, *Monitoring System for Distributed Java Applications*, in: Proc. International Conference on Computational Science 2004, Krakow, 6–9 June 2004, Part III, pp. 472–479, LNCS 3038 Springer, 2004.
- [33] B. Baliś, M. Bubak and M. Węgiel, *Adaptation of Legacy Software to Grid Services*, in: Proc. International Conference on Computational Science 2004, Krakow, 6–9 June 2004, Part III, LNCS 3038, Springer, 2004, pp. 26–33.

