

Design and Implementation of a General-Purpose API of Progress and Performance Indicators

Ivan Rodero, Francesc Guim, Julita Corbalan, and Jesús Labarta

Barcelona Supercomputing Center,
Technical University of Catalonia,
Jordi Girona 29, 08034 Barcelona, Spain
E-mail: {ivan.rodero, francesc.guim, julita.corbalan, jesus.labarta}@bsc.es

In High Performance Computing centers, queuing systems are used by the users to access and manage the HPC resources through a set of interfaces. After job submission, users lose control of the job and they only have a very restricted set of interfaces for accessing data concerning the performance and progress of job events. In this paper we present a general-purpose API to implement progress and performance indicators of individual applications. The API is generic and it is designed to be used at different levels, from the operating system to a grid portal. We also present two additional components built on top of the API and their use in the HPC-Europa portal. Furthermore, we discuss how to use the proposed API and tools in the eNANOS project to implement scheduling policies based on dynamic load balancing techniques and self tuning in run time.

1 Introduction

In High Performance Computing (HPC) centers, queuing systems are used by the users to access the HPC resources. They provide interfaces that allow users to submit jobs, track the jobs during their execution and carry out actions on the jobs (i.e. cancel or resume). For example, in LoadLeveler the `llsubmit` command is used to submit an LL script to the system. Once the job is queued, and the scheduler decides to start it, it is mapped to the resources by the corresponding resource manager.

After job submission, users lose control of the job and they only dispose of a very restricted set of interfaces for accessing data concerning the performance, or progress or job events. In this situation, the queuing system only provides a list of the submitted jobs and some information about them, such as the job status or the running time. Although this is the scenario in almost all the HPC centers, there is information about the jobs that have been submitted that is missing but required by the users. For example, they want to know when their application will start running, once started, when it will finish, how much time remains, and if their application is performing well enough.

If experienced users could obtain such information during the job run time, they would be able to take decisions that would improve system behavior. For example, if an application is not achieving the expected performance, the user can decide to adjust some parameters on run time, or even resubmit this application with new parameters. In the worst case, if an application achieves low performance during its execution, it is cancelled by the system because of a wall clock limit timeout and thus consumes resources unnecessarily.

In this paper we present a general-purpose API which can implement progress and provide performance indicators of individual applications. The API is generic and it is designed to be used at different levels, from the operating system to a grid portal. The

design can also be extended, and the development of new services on top the API are easy to develop. The implementation is done through a lightweight library to avoid important overheads and starvation with the running application. We also present two additional components built on top of the API and explain how they are in the HPC-Europa portal, which is a production testbed composed of HPC centers. The API and the additional tools can be used in both sequential and parallel applications (MPI, OpenMP and mixed MPI+OpenMP). Furthermore, we discuss how to use the proposed API and tools in the eNANOS project¹¹ to implement scheduling policies based on dynamic load balancing techniques and self tuning in run time, to improve the behavior of the applications and optimize the use of resources.

In the following sections we describe the proposed API, some tools developed on top of the API, its design and implementation, some uses of these tools, and we discuss the benefits of using them in production systems.

2 Related Work

In both literature and production systems we can find several mechanisms which allow users to monitor their applications. Firstly, we can find mechanisms offered by operating system, such as the /proc file system in Unix and Linux systems, or the typical ps command. These mechanisms allow the user with information about what is happening during the execution of the application, basically in terms of time and resources used. In addition, several monitoring systems have been developed in different areas. The Performance API (PAPI) project⁹ specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. Ganglia³ is a scalable distributed monitoring system for high-performance computing systems, typically for clusters. Other monitoring tools, such as Mercury¹ or NWS¹⁴ are designed to satisfy requirements of grid performance monitoring providing monitoring data with the metrics.

We have found some similarities between the presented work in this paper and other monitoring APIs such as SAGA⁴. However, these efforts do not exactly fit our objectives and requirements, since with these existing monitoring tools it is difficult to give users a real idea of the progress and behavior of their applications, while for us this is essential. In our approach we try to provide a more generic tool, open, lightweight, not OS-dependent, and easy to use and to extend in different contexts (from a single node to a grid environment).

3 Architecture and Design

The API has 9 main methods which provide information about the progress of the application and application performance. Furthermore, its generic design allows the developer to publish any kind of information concerning the job/application, independent of its nature and content. The API specification is presented below.

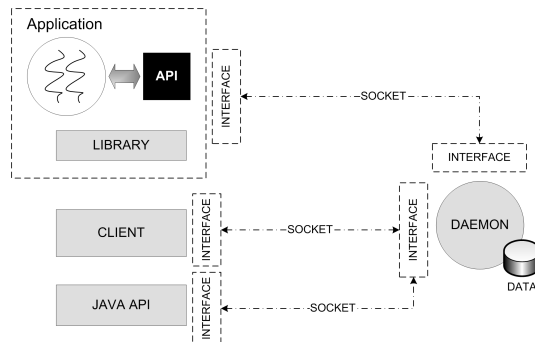


Figure 1. Overall architecture

```

int CONNECT (char *appl_name)
/* Establishes a connection with the daemon for a particular application*/
int DISCONNECT (void)
/* Closes a connection with the daemon*/
pi_id INIT_PROG_INDICATOR (char *name, pi_desc *description)
/* Initializes a progress indicator with the specified name & description*/
int INC_PROG_INDICATOR (pi_id id , pi_value *value)
/* Increments a progress indicator with the specified value*/
int DEC_PROG_INDICATOR (pi_id id, pi_value *value)
/* Decrements a progress indicator with the specified value*/
int SET_PROG_INDICATOR (pi_id id, pi_value *value)
/* Sets the value of a progress indicator with the specified value*/
int RESET_PROG_INDICATOR (pi_id id)
/* Sets the default value of a progress indicator*/
int GET_PROG_INDICATOR (pi_id id, pi_value *value)
/* Returns the value of the specified progress indicator by reference*/
int REL_PROG_INDICATOR (pi_id id)
/* Releases a progress indicator. The identifier can be reused*/

```

We have also included the `CONNECT_PID` method, which takes into account the PID of the application. This method is specially useful when the real user of the progress and performance API is an application itself (a good example is the implementation of the tool constructed on top of the `/proc` file system). Its interface is shown below.

```

int CONNECT_PID(char *appl_name, pid_t identifier)
/* Establishes a connection for a particular application & PID */

```

We present the architecture of the whole system in Figure 1. The application uses the API and it is linked to a library which implements the API and provides an interface to connect the application to a daemon which manages the rest of the components. The daemon is the main component of the system and, basically, it is a multi-threaded socket server that gives service to the clients, to the library and to other external tools developed on top of the API. Apart from managing all the connections, it stores all the data and provide an interface to access it.

We have implemented a client interface that can be used through a command line tool, and can also be used through an administration tool to start, stop and manage the main

functionality of the daemon. Moreover, the system provides a Java API that allow external Java applications to access the system functionality. This API is specially useful to implement services that use the progress and performance system in higher layers such as web and grid services (an example of this approach is presented later with the HPC-Europa portal). All the interface communications are implemented with sockets to ensure extensibility and flexibility; furthermore, it enhances efficiency.

The main data structures used in the whole system are shown below. For simplicity and efficiency, a progress or performance indicator is defined as a generic metric with some restrictions in terms of data types and the information that can include.

```

typedef int pi_id;
typedef enum {ABSOLUTE=1, RELATIVE} pi_type;
typedef enum {INTEGER=1, STRING, DOUBLE, LONG} TOD;
typedef union{
    int int_value;
    char str_value[BUF_SIZE]; /* e.g. BUF_SIZE=256*/
    double double_value;
    long long long_value;
} pi_value;
typedef struct _pi_desc{
    pi_value init_value;
    pi_type type;
    TOD tod;
} pi_desc;

```

The “pi_id” is defined as an integer type and identifies the progress indicators. The “pi_type” define the kind of progress indicator and include both relative and absolute indicators. An example of an absolute indicator is the number of completed iterations of a loop and an example of a relative indicator is the percentage of completed iterations of this loop. “TOD”, that is the acronym of Type Of Data, defines the kind of the progress indicator data. The “pi_value” stores the value of the progress indicator depending on the data type specified in the “tod” structure. The full description of a progress indicator is composed of the type of indicator (pi_type), its type of data (TOD), and an initial value (for example 0 or an empty string).

The constructor of the metric class of the SAGA monitoring model⁴ is shown below. The metric specification of SAGA seems even more generic than the API of the progress indicators because it includes some more attributes and uses the String data type for each of those attributes. However, as our daemon returns the information in XML format, we are able to provide more complex information. Thus, when the client requests information about a given indicator, the daemon provides information about the metric, plus some extra but useful information, for instance the application process ID or the hostname.

```

CONSTRUCTOR ( in string name,
              in string desc,
              in string mode,
              in string unit,
              in string type,
              in string value,
              out metric metric);

```

The XML schema of the data obtained from the daemon with the client interfaces is shown in Figure 2. It is composed of a set of connections (PI_connection). Each connection has a name to identify it, the name of the host in which the application is running (the

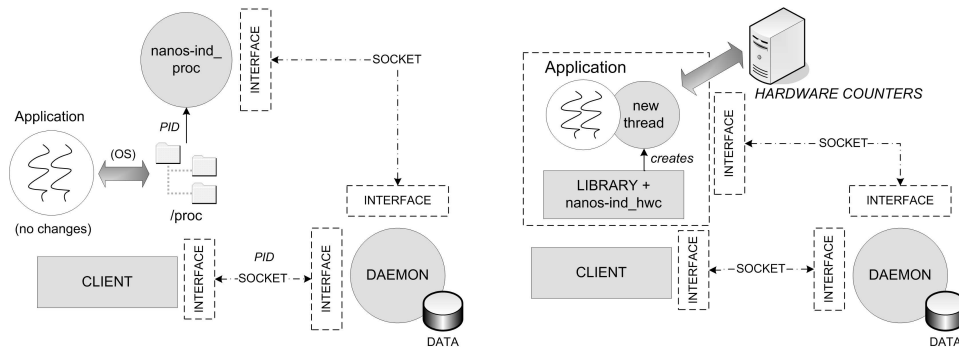


Figure 3. Schema of the "/proc" module (left), and schema of the hardware counters module (right)

and manage the hardware counters, and for AIX we have used PMAPI. Some examples of metrics that can be implemented with the performance indicators are the number of MIPS or MFLOPS that the application is achieving (absolute value) or the percentage of MFLOPS (respect the maximum possible in the resource) that the application is achieving. The hardware counters required in PAPI to calculate these metrics are: PAPI_FP_OPS (number of floating-point operations), PAPI_TOT_INS (total number of instructions) and PAPI_TOT_CYC (total number of cycles). In the case of MPI+OpenMP parallel applications, we monitor the MPI processes from the MPI master process, and we use collective metrics for the OpenMP threads. The basic schema of this component is shown in Figure 3.

5 The HPC-Europa testbed: a case of study

In addition to developing some tools on top the API, we have implemented a Java API to allow grid services to access the API progress indicators. The methods of this API are shown below.

```
String getDaemonInfo (String daemonHostname)
/* Gets all the information from the daemon in the specified host */
String getDaemonInfoPid (String daemonHostname, int appl_pid)
/* Gets the connections and indicators of the specified application */
String getDaemonInfoConnection (String daemonHostname, String connectName)
/* Gets the information of the connection with the specified name */
String getDaemonInfoIndicator (String daemonHostname, String indicatorName)
/* Gets the information of the indicators with the specified name */
```

We have used the Java API in the HPC-Europa single point of access portal. One major activity of the HPC-Europa project is to build a portal that provides a uniform and intuitive user interface to access and use resources from different HPC centers in Europe⁷. As most of the HPC centers have deployed their own site-specific HPC and grid infrastructure, there are currently five different systems that provide a job submission and basic monitoring functionality in the HPC-Europa infrastructure: eNANOS¹⁰, GRIA⁵, GRMS⁶, JOSH⁸, and UNICORE¹³.

The monitoring functionality is done with the "Generic Monitoring Portlet" as is shown in Figure 4. In this figure we can see 3 submitted jobs, two of them running in the same

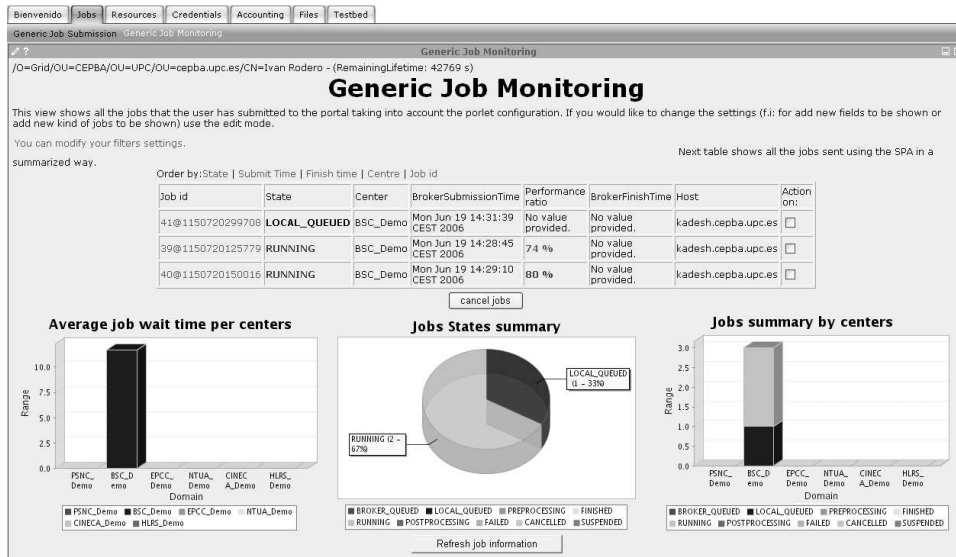


Figure 4. Monitoring functionality of the HPC-Europa SPA portal

machine. For each of these jobs, the monitoring portlet provides a set of data in which a “performance ratio” is included. This ratio indicates the percentage of MFLOPS the application is achieving in relation to the maximum MFLOPS the application can achieve in this machine. This value gives the user an idea of how its applications are behaving. In case of the example shown in Figure 4, if the scheduler starts running another application in the same machine and the machine does not have any more free resources, the performance ratio of the running applications would decrease more and more, because the machine would be probably overloaded.

6 Discussion

In this paper we have discussed the lack of information on the progress and performance of the applications in a HPC system. We have also discussed the suitability of a progress and performance system which gives the user an idea of how its applications are progressing and helps to improve the performance and the use of resources of the HPC systems. In an attempt to improve the system, we have presented a generic API of progress and performance indicators and some tools which can be implemented on top of this API. We have also explained how we have used this API and tools in the HPC-Europa single point of access portal.

In addition to using the progress indicators API as a monitoring tool, we have introduced this API inside the eNANOS system¹¹ to improve its scheduling policies. The idea is using the progress indicators to perform the grid scheduling guided by the performance and progress indicators. Therefore, the job scheduling can be done based on dynamic load balancing techniques at the grid level. Furthermore, we have integrated the progress and

performance system into our grid brokering system¹⁰ and also with other systems such as GRMS⁶ in the framework of the CoreGRID project² as described in¹².

Future work will focus on improving the scheduling policies, using the progress indicators API more deeply with the current implementation. We believe that the progress indicators API can improve current scheduling policies by combining the information about the progress of the applications with the information obtained from our prediction system and the run-time information about load balance. Specifically, we believe that the dynamic adjustment of MPI+OpenMP applications on run-time can be improved with this additional information.

Finally, we are working to improve the current modules of the progress indicators API and to implement new functionalities. Our main objective is to make it possible for the user to avoid adding any line of code in the original application or recompiling the application to obtain the performance indicators obtained from the hardware counters. We are working on dynamic linkage for sequential applications and the usage of PMPI for MPI applications.

Acknowledgments

This work has been partially supported by the HPC-Europa European Union project under contract 506079, by the CoreGRID European Network of Excellence (FP6-004265) and by the Spanish Ministry of Science and Technology under contract TIN2004-07739-C02-01.

References

1. Z. Balaton, G. Gombis, *Resource and job monitoring in the grid*, Proceedings of the Euro-Par, pp. 404-411, 2003.
2. CoreGRID Web Site. <http://www.coregrid.net>
3. Ganglia Web Site. <http://ganglia.sourceforge.net>
4. T. Goodale, S. Jha, T. Kielmann, A. Merzky, J. Shalf, C. Smith, *A Simple API for Grid Applications (SAGA)*, OGF GWD-R.72, SAGA-CORE WG, September 8, 2006.
5. GRIA project Web Site. <http://www.gria.org>
6. Grid Resource Management System (GRMS) Web Site. <http://www.gridlab.org/grms>
7. HPC-Europa Web Site. <http://www.hpc-europa.org>
8. JOSH Web Site. <http://gridengine.sunsource.net/josh.html>
9. PAPI Web Site. <http://icl.cs.utk.edu/papi>
10. I. Rodero, J. Corbalan, R. M. Badia, J. Labarta, *eNANOS Grid Resource Broker*, EGC 2005, LNCS 3470, Amsterdam, The Netherlands, pp. 111-121, 14-16 February, 2005.
11. I. Rodero, F. Guim, J. Corbalan, J. Labarta, *eNANOS: Coordinated Scheduling in Grid Environments*, ParCo 2005, Málaga, Spain, pp. 81-88, 12-16 September, 2005.
12. I. Rodero, F. Guim, J. Corbalan, A. Oleksiak, K. Kurowski, J. Nabrzyski, *Integration of the eNANOS Execution Framework with GRMS for Grid Purposes*, CoreGRID Integration Workshop, Krakow, Poland, pp. 81-92, October 19-20, 2005.
13. UNICORE Web Site. <http://www.unicore.org>
14. R. Wolski, N. T. Spring, J. Hayes, *The network weather service : a distributed resource performance forecasting service for metacomputing*, Future Generation Computer Systems **15(5-6)**, 757-768 (1999).