The NANOS Resource Manager in the eNANOS Project

Ivan Rodero, Julita Corbalan

Computer Architecture Department
Technical University of Catalonia (UPC)
Jordi Girona 1-3, Modul D6, 08034 Barcelona, Spain
{irodero, juli}@ac.upc.edu

Abstract

In this paper we present the NANOS Resource Manager that we have used within the eNANOS project. In particular, we describe its main functionalities and how we have integrated it into the eNANOS architecture. We also describe the extended functionalities of the NANOS Resource Manager that perform dynamic processor allocation in coordination with the eNANOS jobs scheduler. The target of our coordinated scheduling strategy is to manage multilevel parallel applications on clusters of SMPs architectures efficiently.

1 Introduction

Traditional operating systems provide a reduced set of functionalities to manage the processor allocation to applications. These mechanisms are usually hidden to the user domain: to the regular applications and other components of the systems that are not executed in system mode such as the queuing systems. Although the numerous efforts devoted to the research on the area of cluster scheduling, there is a lack in the mechanisms for the efficient processor allocation still such as the dynamic allocation of threads to the processors.

To perform these tasks, the *NANOS Resource Manager* (NANOS-RM) has been developed on top of traditional systems providing this extended functionality without modifying the native systems [1]. As a local processor scheduler, the NANOS-RM main goal is to efficiently distribute a set of processors between the applications under its control. It provides an API to coordinate with the different components of the system: parallel applications, queuing system, monitoring system, and performance

analysis system. It also includes a set of predefined processor scheduling policies that are based on space-sharing approaches to distribute the processors. The responsibility of the processor distribution is shared by the NANOS-RM and the applications.

In this paper we focus on the cluster and processor scheduling strategies that we have at the two bottom levels of our architecture. As is described in [6], the *eNANOS Scheduler* implements the cluster scheduling based on co-allocation policies, and the *NANOS-RM* performs the processor scheduling for SMP nodes.

Figure 1 shows the eNANOS architecture focusing on the components at its lower scheduling layer. It also shows some details and the APIs concerning the coordination between the applications, processor scheduler, and cluster scheduler. The NANOS-RM obtains the applications performance directly, and it provides the performance information of a node to the eNANOS Scheduler that collects the performance information of each node. This mechanism helps the cluster scheduler to improve its cluster scheduling strategies.

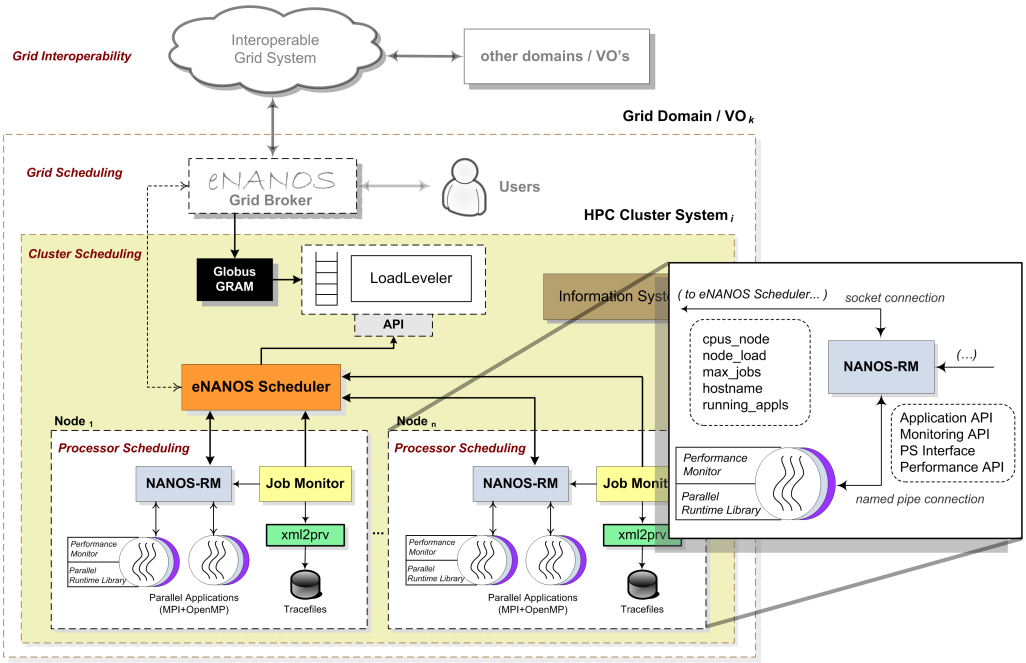

Figure 1: Low level support detail of the eNANOS architecture

In eNANOS, the scheduling decisions are communicated to the other components in the system and are taken based on direct information. That is why we provide a well defined set of APIs between the levels to both facilitate the scheduling decisions (for instance specific allocations) and get detailed information in run time (for instance the real performance reached by a job). As can be seen in the highlighted window of figure 1 we provide some APIs to coordinate various components inside a node: the parallel applications, the performance analysis tools, its own monitoring system, and some runtime libraries that provides several functionalities such as parallelism management. Moreover, we provide other APIs to coordinate different NANOS-RMs with the eNANOS Scheduler. They provide static resource information such as the number of CPUs, and dynamic information such as the maximum number of jobs that a node can run simultaneously without degrading the applications performance (also known as multi-programming level).

In the rest of this paper we present the architeture of the NANOS-RM and the details of the different NANOS-RM's functionalities.

2 Overall Architecture

The main responsibilities of the NANOS-RM are: collecting the applications requirements, applying the scheduling policy and sending the processor distribution to the applications. We assume that the parallel applications are executed in the context of a runtime (OpenMP, MPI) that provides the required functionality to manage their parallelism. The implementation is done by modifying the run time including, in the appropriate places, calls to the NANOS-RM API to adjust the processor allocation according to the NANOS-RM decisions. Moreover, it requires a reduced set of functionalities from the native system that make it portable among different operating systems. The most important ones are:

- A shared memory mechanism that is usually provided by the majority of unix-based operating systems.
- A binding mechanism between processes and processors. In general it is provided by AIX, IRIX, Linux as well as other operating systems.

Although the current implementation is done on top of AIX, it can be easily ported to other operating systems such as Linux however, the efficiency of the NANOS-RM will depend on the level of control provided by the native operating system.

Internally, the NANOS-RM is implemented with several components as is shown in figure 2: main execution thread, scheduler thread, application support API, job scheduler support API and monitoring support API. The main thread initializes the data structures

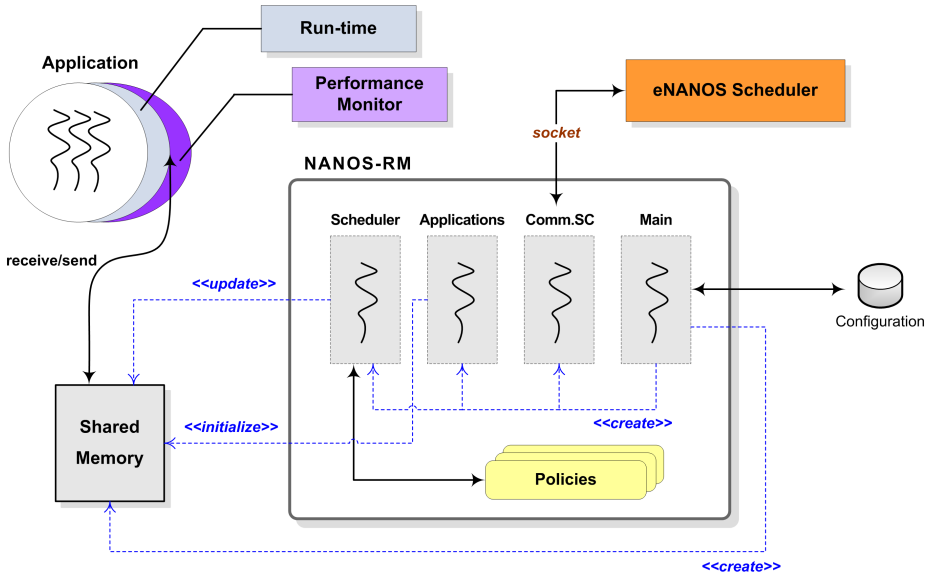

Figure 2: NANOS-RM overall architecture

based on the configuration and creates the rest of the working threads. It works in an iterative way (asynchronously) and performs the following basic tasks: processor scheduling, system load control, and dynamic detection of multilevel applications. The processor scheduling is performed by the scheduler thread which periodically wakes up, collects the available information regarding the applications request and performance, applies the scheduling policy and informs the applications. The other components implement communication with applications, with the job scheduler and with the monitoring tool. The initialization and configuration is done by the command-line interface.

3 Processor allocation policies

The scheduling policies supported by the NANOS-RM are based on dynamic CPU allocation and they are composed of two phases. Since these scheduling policies are oriented to MPI+OpenMP applications they are multilevel. They also support one level of parallelism (regular MPI or OpenMP). The first scheduling phase is performed between the applications and implements a FIFO policy: one application has N MPI processes and requires a minimum of N processors. The second phase is performed between the processes and implements two possible policies: Equipartition (EQUIP)

[4] and Dynamic Processor Balancing (DPB) [2]. Equipartition starts the application execution allocating the required number of CPUs and distributes them equally among the application processes. Dynamic Processor Balancing tries to balance the load between the MPI processes of an application.

4 Job model

Figure 3 shows the job model used by the NANOS-RM scheduler. A job is composed of a set of processes, and one of them is marked as the master. However, when the job information is collected by the eNANOS job scheduler [6] from different NANOS-RMs we use a set of master identifiers because the job can be mapped into different nodes. Thus, the applications have one master process for each node. Moreover, the NANOS-RM is in charge of managing other job information such as the total number of requested CPUs that is computed from the process data. If a job is composed of more than one process, the job is considered to be an MPI+OpenMP one. Otherwise, it is considered to be a regular OpenMP job.

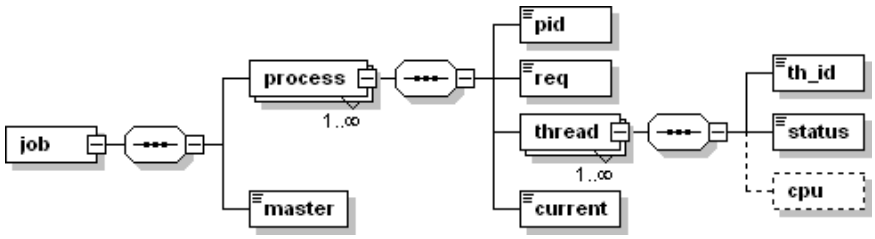

Figure 3: NANOS-RM job schema

5 Applications

The NANOS-RM supports MPI and MPI+OpenMP applications. The source code of the applications does not require modification; it just needs to be re-linked with the libraries provided with the NANOS package. These libraries include functionalities to connect/disconnect to the NANOS-RM, ask automatically for processors, adapt the application parallelism to the NANOS-RM allocations and inform about the application performance.

6 Performance monitor

The philosophy of the eNANOS project includes automatic measurement of the applications performance. Moreover, the performance library that is provided with the NANOS package automatically measures the load imbalance of the MPI+OpenMP applications, and the scheduling policies are implemented using this performance information.

Most scientific applications are iterative which means that they apply the same algorithm several times to the same data set. In particular, the data is processed repeatedly until the number of iterations reaches a specific value or until the value of a parameter reaches a particular value (for instance, when the error converges to certain value). Using this iterative behavior of the applications, the profiling library is able to obtain the accumulated value of the relevant computation and communication periods.

We consider the standard mechanism that MPI defines to instrument the applications. It consist of providing a new interface that is called before the real MPI interface [5]. Figure 4 shows how the application is instrumented using the standard MPI profiling mechanism. When a MPI call is invoked from the application, the library measures the spent time on the call and adds this value to the total amount of spent time on MPI calls. The iterative structure of the application is detected using a Dynamic Periodicity Detector (DPD) library [3]. DPD is invoked from the instrumented MPI call using the PMPI mechanism. Its input is a value obtained from a composition of the MPI primitive type (send, receive, etc.), the destination process and the buffer address. With this value the DPD detects the pattern of the application periodic behavior. When a period is detected, the profiling library keeps track of the time spent in executing the whole period. These two values, MPI time and execution time, are averaged with the values obtained from some periods and are passed to the resource manager to feed the allocation policy. More details of this mechanism can be found in [2].

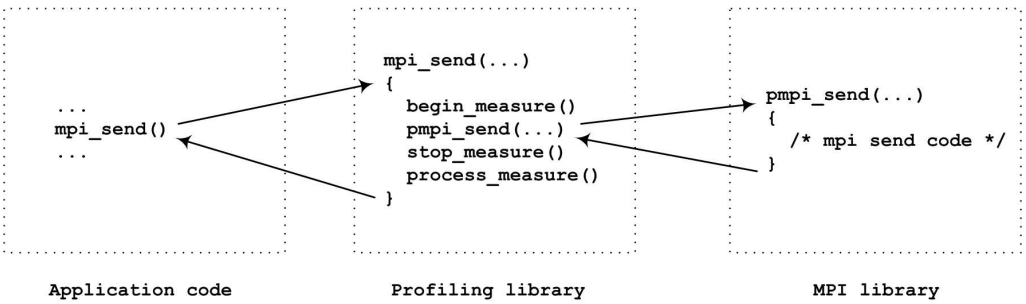

Figure 4: NANOS-RM profiling mechanism

7 NANOS-PS

The NANOS-PS is a tool included in the NANOS-RM package that allows users or applications to obtain information about the status of the jobs, as well as some basic information regarding the NANOS-RM details. The NANOS-PS can be queried through a command-line client or through an API. The NANOS-RM information includes detailed information of scheduling decisions such as the allocations between CPUs and threads/processes. Moreover, it includes the monitoring data of the running applications following the job model presented previously in figure 3.

8 Application API

From the point of view of the application, we are assuming that the run time libraries (of OpenMP or MPI) have been modified to include the following methods. However, it is also possible to invoke these methods directly from the applications through the API shown in listing 1.

```
int  ConnectRM(int master_jobid)
void DisconnectRM()
int  CPUSCurrent()
int  CPUSRequest(int cpus)
void NewThread(int pid,int vpid)
int  MustRelease(int id)
int  ReleaseCPU(int vpid)
void WakeUpThreads()
```

Listing 1: NANOS-RM application API

ConnectRM connects with the NANOS-RM. It returns 0 if the connection is established successfully and -1 if there is any error. The *master_jobid* is the master process identifier of the job in the case of MPI+OpenMP applications. Note that to invoke the rest of the methods it is necessary to establish a connection previously, if there is no established connection an error is thrown. *DisconnectRM* disconnects the application to the NANOS-RM. *CPUSCurrent* returns the current number of allocated CPUs by the applications that are running in the node at the moment that the method is invoked. *CPUSRequest* requests the indicated number of CPUs (*cpus*) for the application. It waits until the next scheduling cycle and returns the number of allocated CPUs to the application. *NewThread* informs the NANOS-RM that the application has obtained a new kernel thread. The *pid* is the physical identifier of the new thread and *vpid*

is the logical identifier (which can be from *0* to *threads_number-1*). *MustRelease* returns 1 if the logical identifier of the thread (which is specified by *id*) is marked to be released. This situation happens when the NANOS-RM reduces the number of allocated CPUs of an application. *ReleaseCPU* blocks the thread that is specified by its logical identifier (*vpid*). In order to invoke this method, an application must execute the *MustRelease* method previously to check whether the thread must be released. *WakeUpThreads* unblocks all the threads that are marked to be resumed. This situation happens when the NANOS-RM increases the number of allocated CPUs of an application.

9 Monitoring API

The monitoring API that is one of the mechanisms to provide dynamic data from the applications that are under the NANOS-RM control. This API provides information regarding the available resources and the current job status through the methods shown in listing 2.

```
int  Connect_JM2RM(char* hostname,int *fds)
void Disconnect_JM2RM(int *fds)
int  getStaticInfoSystem(int *fds,struct STATIC_INFO_system *info)
int  getDynamicInfoSystem(int *fds, struct DYN_INFO_system *info)
int  getPS(int *fds,JM_data *JM_info, int detail)
```

Listing 2: NANOS-RM job scheduler and monitoring APIs

Connect_JM2RM connects the monitoring system to the NANOS-RM, *Hostname* indicates the node where the NANOS-RM is running and *fds* is the connection handler which is an output parameter. The method returns 0 if the connection is established successfully and -1 if there is any error. Note that the connection handler is required by the rest of the methods, therefore it is necessary to establish a connection previously. *Disconnect_JM2RM* disconnects the monitoring system to the NANOS-RM. *fds* should be the connection handler that was returned by the *Connect_JM2RM* method. *getStaticInfoSystem* and *getDynamicInfoSystem* collects static and dynamic information of the system respectively. They return 0 if the method has been executed properly and -1 if there is any error. *fds* is the connection handler that was returned by the *Connect_JM2RM* method. *info* must be a valid memory address where to return the monitoring data.The static monitoring information follows the *STATIC_INFO_system* data structure and the dynamic information follows the *DYN_INFO_system* one. *getPS* collects information concerning the running jobs details. It returns 0 if the method has

been executed properly and -1 if there is any error. Since it allocates memory with *malloc*, it must be freed when it is not used. *Detail* indicates the detail level of the data that the NANOS-RM will provide. The valid values for *detail* are *RESUMED* and *DETAILED*.

10 Performance API

The performance API of the NANOS-RM is provided jointly with the application API. It has its own methods to send the information concerning the performance, but the connection and disconnection functionalities are provided by the application API. The performance API adds the two new functions that are described in listing 3.

```
void SetPercBalancingData(double percentage)
void SetIterTimeData(double time)
```

Listing 3: NANOS-RM performance API

SetPercBalancingData provides the percentage of time that the application has been run inside MPI functions. *SetIterTimeData* returns the spent time per iteration of the application's most external loop. We assume that applications are iterative.

11 Summary

In this paper we have presented the NANOS Resource Manager, we have described its main functionalities and how it has been integrated into the eNANOS project. Our main lines of future work are focused on improving the coordination with the eNANOS Scheduler [6] in order to enhance the scheduling strategies in clusters composed of SMP architectures.

Acknowedgements

This paper has been supported by the Spanish Ministry of Science and Education under contract TIN200760625C0201.

References

[1] J. Corbalan, Coordinated Scheduling and Dynamic Performance Analysis in Multiprocessor Systems, Ph.D. thesis, Computer Architecture Department, Technical University of Catalonia (UPC), Spain (July 2002).

[2] J. Corbalan, A. Duran, J. Labarta, Dynamic Load Balancing of MPI+OpenMP Applications, in: International Conference on Parallel Processing (ICPP), Montreal, Canada, 2004, pp. 195–202.

[3] F. Freitag, J. Corbalan, J. Labarta, A Dynamic Periodicity Detector: Application to Speedup Computation, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, USA, 2001.

[4] C. McCan, R. Vaswani, J. Zahorian, A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors, ACM Transactions on Computer Systems 11 (2) (1993) 146–178.

[5] The MPI Message-passing Interface Standard Web Site.
`http://www.mpi-forum.org`

[6] I. Rodero, J. Corbalan, Design and Implementation of the eNANOS Scheduler, UPC-DAC-RR-CAP-2008-20, Tech. rep., Computer Architecture Department, Technical University of Catalonia (2008).