Managing jobs with an interpreted language for dynamic adaptation

Anolan Milanés*
Pontifícia Universidade Católica do Rio de Janeiro
Computer Science Department
anolan@inf.puc-rio.br

Noemi Rodriguez
Pontifícia Universidade Católica do Rio de Janeiro
Computer Science Department
noemi@inf.puc-rio.br

Bruno Schulze
National Laboratory for Scientific Computing
Computer Science Dep, Brazil
schulze@lncc.br

ABSTRACT

In this paper we explore the advantages of using interpreted languages for building submitting engines for the grid. In particular, we discuss an example engine, developed using ALua, for submitting jobs in a cluster, which can be extended to a grid environment. We claim that the flexibility offered by interpreted languages justifies the problems related to the intrinsic loss of efficiency associated with this kind of languages. The focus of this work is on adaptation and ease of use.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems; D.3.2 [**Programming Languages**]: Language Classifications

General Terms

Languages, Management

Keywords

grid computing, job management, adaptation, interpreted languages

1. INTRODUCTION

Despite the fact that grid computing potential has been generally accepted, and the emergence of some standards (most of them *de facto* standards as is the case of the major grid initiative, the Globus Toolkit), working on grids is not an easy chore [3]. Computer grids involve, besides the well-known and studied distributed paradigm, a perception of unity in environments that can be quite heterogeneous as to platforms, software, and user skills. Besides, they introduce new problems of security and issues related to virtual organizations (permissions and effective collaboration, for instance). This has generated a quest for new infrastructures, middleware and applications.

We propose a lightweight tool for scheduling and managing jobs submitted through the grid. It is based on ALua, an extension of the Lua interpreted language for the execution of parallel and distributed applications. ALua is a lightweight and very portable system with a dual programming language model that allows building flexible application without compromising performance.

The rest of this paper is organized as follows: Section 2 provides the context of this work. Section 3 explains the motivations for the use of interpreted languages in the grid, and Section 4 reviews the ALua system. Section 5 describes the implementation and discusses some security issues. Section 6 presents some preliminary experiments. Finally, Section 7 summarizes the paper and discusses future work.

2. BACKGROUND

The Globus Toolkit [6] is a widely accepted *de-facto* standard middleware for Grid Computing. It provides basic components to implement resource management, data management, and information services. The resource management pillar includes the GRAM (Grid Resource Allocation Manager) Component [4], which provides support for job execution and management. GRAM includes the *gatekeeper*, that receives the job execution request, authenticates the client, maps him to a local account, and then instantiates a Job Manager to take care of the job and of the communications with the client. A general overview of GRAM is depicted in Figure 1, taken from [4].

GRAM parses job descriptions written by the user in Resource Specification Language (RSL) [17]. Globus does not provide a local scheduler (it uses fork by default to execute processes in the local host), but it offers interfaces and the gram-reporter packages for a group of local schedulers such as Condor [19], LSF [12] and OpenPBS [14], and allows adding new scheduler-specific job managers.

2.1 Schedulers and Job Managers

To avoid misunderstandings due to the frequent reuse of terms in grid computing, we will at this point define some

*Also associated to LNCC-Brazil

MGC '05 November 28- December 2, 2005 Grenoble , France

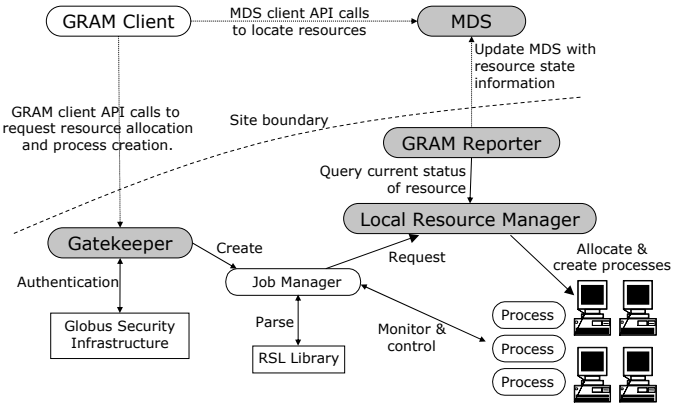

Figure 1: GRAM Overview

expressions that will be used in this paper. We will call *Job Scheduler* a program that searches for the resources necessary to execute the tasks composing a job. A *Resource Manager* is in charge of the ordering, submitting, monitoring and controlling the jobs. A job is a unit of submission. Every job is composed of one or more tasks, and may define communication or precedence relationships among these tasks.

Local job schedulers such as LSF (Load Sharing Facility), Sun Grid Engine (SGE), OpenPBS (Portable Batch System) and Condor allocate resources from a networked pool of computers to jobs submitted. They provide different sets of features, which include checkpointing and process migration, authentication and authorization, daemon fault recovery, staging, and dynamic load balancing. As the grid environment is usually characterized by changing conditions, there is a need of some kind of adaptation. Adaptability in those systems is mainly based on job migration. They set the QoS parameters on submitting, and look for optimal ways to redistribute the tasks upon environmental changes. Condor, for instance, was developed for scavenging idle computer cycles. It can be configured to kill a guest job when a computer stops being idle, reinitiating in another host.

3. SCRIPTING IN THE GRID

The benefits of using an interpreted language for coordinating applications while maintaining their core in a compiled language have long been discussed in distributed programming [18] . However, in high performance computing, probably due to the focus on efficiency, emphasis has been on ignoring multilingual programming. In our view, this will change with the growth of Computer Grids.

Parallel programmers have traditionally placed very strong emphasis on the performance of the tools they use. This has caused most of them to choose simple programming environments based on a single conventional programming language, such as C and FORTRAN, and a communication library, such as MPI. In environments such as a computer cluster — maybe the most popular parallel system in the nineties — this choice works well. Clusters are typically highly homogeneous environments. Issues like load balance and fault tolerance do not play a significant role in cluster programming, because the programmer often has complete control over the system while his application is running.

However, the scenario is quite different with computer grids. Grids provide an economically convenient alternative for processor-hungry applications: different institutions can place parts of their computing resources in a pool from which many may benefit. This pool of resources becomes a virtual machine spanning different administrative and geographical domains. In this scenario, the programmer has no control over the global availability of resources, which tends to suffer great variability. Under such circumstances, coordinating the use of resources becomes much more complex than in a local, homogeneous environment. Besides, in general, programs that can benefit from running on a computational grid are long-running applications. It often does not make sense to stop the application and launch it again in order to improve resource usage. It becomes important to be able to act upon the application while it is running [1].

The use of an interpreted scripting language as a coordination tool can bring several advantages in this setting. One of the important characteristics of an interpreted language is interactivity: with an interpreted coordination language, the programmer may use a console to control the application. New chunks of code can be added dynamically, and existing functions can be redefined, thus allowing the programmer to tune the application without having to restart it. This provides a new dimension of adaptation, in which not only tasks can be redistributed according to execution conditions, but the code they are running can also be modified, either because of execution conditions or as a consequence of partial application results. In [21] we discuss an example of such facilities over the implementation of an A-Team (asynchronous agent team) application. In that case, the program itself was based on the dual Lua/C model, keeping the processing core in C and communication in Lua. We show how ALua allows us to monitor the application and to redefine some of its parts on the fly, for instance adding a log file, with information about transmitted messages.

Scripting languages typically allows the programmer to code complex tasks with small effort: the programmer can, with a few lines of code, insert new monitoring and logging facilities in a running application. Programs implemented using interpreted languages are also highly portable. In environments like the Grid, where the destination of the execution is usually unknown, portability is an important issue.

Another facility associated with interpreted languages is a more flexible type system, with features such as dynamic typing, functions as first-order values, and support for closures. These features are important in building abstractions that help the programmer deal with low-level mechanisms at the level of detail that he needs [16]. The cognitive weight of learning to use libraries and services needed to control Grid environments, such as the ones offered by Globus, may be well above what some application programmers can take; however, these libraries and services will probably be much easier to learn if seen at a higher level of abstraction.

On the other hand, all this flexibility calls for a self-disciplined programming style to allow for the base code to be easily maintainable.

4. ALUA

ALua [21] is an event-oriented extension of the interpreted language Lua [9] for the development of distributed applications. ALua inheritates from Lua the ease of integration with C code, thus supporting the use of a dual programming mode that takes advantages of both worlds: computation-

ally hard tasks are typically executed in C, and Lua is used as the programming language for communication, collaboration, and dynamic adaptation.

ALua processes communicate by exchanging asynchronous messages, consisting of chunks of Lua code. There is a single communication primitive, *alua.send*. The arrival of a message is handled as an event, and the handler for this event executes the received chunk of code. All the messages are executed atomically: every event is handled to completion, eliminating the need for concurrency management. ALua *applications* are environments within which processes can exchange messages(a little like communicators in MPI).

ALua's implementation is based on daemons that run on every machine, mediating the communication among processes. In version 4, ALua has been improved by migrating all the code to Lua, modifying the API and adding the possibility of different applications to coexist in a set of daemons. As a consequence, new processes can join an existing application and exchange messages with running processes, and a process may belong to various applications. The installation depends only on the luasocket and luaposix libraries (both extremely portable, although not yet supported by Windows).

ALua's application in grid environments has been explored before. [20] reports an experience integrating ALua with Globus for the monitoring of grid resources and the dynamic adaptation of the application.

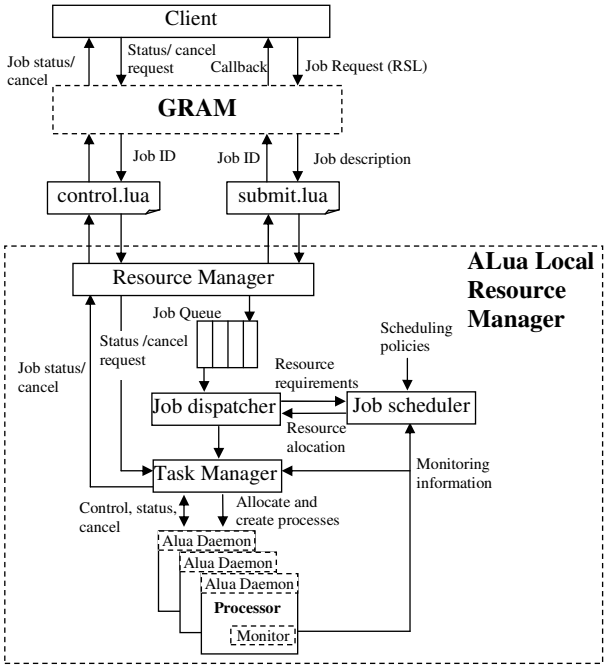

Figure 2: Architecture

5. IMPLEMENTATION

The motivation of our implementation was the need of submitting applications through the grid to clusters and machines where the *Globus Toolkit* for some reason is not installed. For instance, in clusters with virtual addressing we would not be allowed to use *Globus*. Alternative solutions as *Condor* can be too heavy and/or difficult to install and configure for the job we need them to do. Besides, instability in the load and frequent exclusive access requests made adaptability a requirement.

Our goal was to develop a mechanism for the allocation of resources for computational jobs submitted through the grid. The requisites for our system would be the portability and simplicity of the installation and configuration, that it be lightweight and highly flexible without compromising efficiency, and high availability (no need to stop the system for reconfiguration). For the moment, no support will be provided for interactive jobs.

Although the current version (Globus Toolkit 4) has an attractive group of features, most of them related to web services, we chose the distribution 2.4 (Pre-WS), thinking in efficiency. To interface between our local scheduler with the Globus GRAM job manager we wrote a Perl module as suggested in [7]. It allows submitting and managing jobs through the grid using the Globus commands. Figure 2 shows the system architecture.

Users who want their job to be managed by the ALua Job Manager just need to submit a regular Globus RSL descriptor and specify ALua as the job manager. The ALua Resource Manager, upon receiving a request, will enqueue it until it is time for execution, and will return to the user an ID that can be used to control the job, that is, to investigate the state of the job or cancel it. Using ALua allows new processes to join an enqueued or already executing application. This means that the running processes and the new process can freely communicate with each other.

When it is the turn of our job to be executed, the Resource Manager will create a process (that will be called a Task Manager) to take care of this particular job. This Task Manager process spawns the number of processes specified in the RSL descriptor on the hosts chosen by the Job Scheduler, which selects the best available resources based on monitoring information and pre-defined policies. The Task Manager also controls the state of the created processes and the load in the nodes where they are running, so it is possible to specify Service Level Parameters that should be respected during execution. In this way, the behavior of the system can range from scavenging to a guarantee of the quality of service for the user. When a node's load increases and it does not satisfy requirements anymore, all the job managers controlling processes in that node whose expectations would be violated are notified so they can cancel their respective processes and ask to the scheduler for a new host to re-initiate them. In conclusion, we have a job manager hierarchy where a job dispatcher takes the jobs from the queue and creates a new job manager to watch the job until its end. In case of a request for the creation of a new process (or processes), the corresponding job manager asks the Job Scheduler for an available host in which to create the process.

The control of the job and of the load parameters is based on the information gathered by the monitoring system. It is implemented by the collection agents and the notification engine. The collection agents are ALua programs running in every execution host. They collect information about CPU load (and can be easily extended to any other collectable parameter) at programmable intervals. The information is sent (using a push model) to the scheduler who analyzes the data during the resources selection process, and to the job managers, so they can be aware of the state of the execution

environment where their respective processes are running. The interested consumers (e.g., the scheduler and the job managers) subscribe to the collection agents of their interest. In order to avoid unnecessary communication, the engine has been designed so that new information will be sent only in case of state changes. The granularity of those changes can also be specified, and can be altered at run time.

For the notification engine we used LuaMonitor, an implementation whose original version, developed for CORBA, is described in [5]. Lua Monitor is an extensible monitoring mechanism written in Lua. It is based on the concept of properties, aspects and observers. *Properties* are the parameters to be observed: for instance, in our case we defined the CPU load as a property. An *aspect* of a property allows observers to watch not only the value, but also its behavior, for instance along time (increase, dramatic change, etc). The function that evaluates an aspect can be changed at run-time, and new aspects can be created, so the requirements can be dynamically adapted. The observer specifies the property/aspect of interest and a callback so that it can be notified when a certain aspect (or aspects) of a property becomes true.

It is very easy to subscribe to a monitor using this framework, as shown in this line of code:

```
LuaMonitor:attachEventObserver({
notifyEvent=function(self, event)
        alua.send("taskmanager","monitor_alarm()");
end},"CPUIncrease","$CPU:Increasing")
```

This code registers a function as callback for an event, which in this case is the increase of CPU load. In the example shown, if the event occurs, a message will be sent to the *taskmanager* process to execute function monitor_alarm(). It allows for a customizable response to events taking place in processes that the observer process (the Task Manager in this case) is controlling. The notification function could be changed at any time, and so could the parameters that generate this event, allowing for adaptation in reaction to changes in the environment without the need of reinitiating the system.

5.1 Security issues

A major reason against the use of interpreted languages for infrastructure tools are some security issues intrinsic to the flexibility that they offer.

The ALua system currently does not offer support for authentication. The idea here is to exploit the security framework offered by the Globus Toolkit for sign-on. However, the problem of authenticating messages still remains. With no authentication for the messages received and executed by the daemons, it is possible for a user inside the cluster to execute (unknowingly) a script on behalf of another user. We will address this matter in future work.

On the other hand, currently, jobs are executed with the same rights as those of the Resource Manager process. This does not allow for distinguishing among different users submitting their jobs to the same Manager and for enforcing their individual resource limitations. One solution would be to create a different daemon per user on each executing node, but this does not scale well and is not compatible with centralized scheduling. A better solution would be to have a single daemon per node, executing as a special user, creating user environments [11] with the correct rights for each submitted job.

5.2 Scheduling on the Grid

The tool we developed allows for the execution of jobs submitted through the grid and the publication of the data in the MDS, but there is still a lot of work to do for achieving a real integration with the Grid environment. In this initial stage we are, there is neither meta-scheduling nor communication with processes outside the ALua site. ALua processes should to be able to communicate transparently with the other processes executing tasks belonging to the same job, with independence of the locality and local resource manager of the remote site. This will lead to the inclusion of global addressing and of mechanisms to allow for site-to-site effective and secure communications (probably using the Globus XIO API).

We also intend to add some facilities for dynamic interaction with the application itself. Using Lua support tools, we can use information from `.h` application header files to make some global variables automatically available in a surrounding ALua environment. This would allow programmers to, through a console, monitor the state of running applications, and eventually interact with them, by changing parameters or resource allocation decisions.

6. EXPERIMENTAL RESULTS

We have performed a set of experiments to analyze the overhead generated by our tool compared to the execution using remote command execution (*rsh*). We also measured the overhead caused by the monitoring system. For the experiments we used the heuristic proposed in [15] for the traveling tournament problem. The tasks are independent; there is no communication among them. Only one process was allocated per processor. The experiments were performed on a set of Intel Pentium II CPUs with 398 Mhz and 280 MB of RAM.

Table 1: Performance results

Instance	Remote execution	ALua	ALua with monitors
nl14	295.34	295.56	295.72
nl16	2291.54	2293.668	2295.352

Table 1 shows timing in seconds for the execution of the algorithm with two instances. The results of the table show that the execution using remote command execution was only slightly faster than with ALua.

We can also observe that the expected increase on the computational time when the monitoring system is operating is negligible (about 0.05 percent of the computational time). We concluded that our tool satisfy our requirements of low computational overhead.

7. FINAL REMARKS

Interpreted languages are often applied in the grid to simplify operations like process automation, error checking, sophisticated analysis and display applications, as in GrADS [8], which offers a proprietary script language, and Chimera Grid Tools [2], which contains a library of Tcl language. Another application of interpreted languages in computer grids are the Commodity grid Kits (CoG Kits) [22].

They aim to provide wrappers between Globus and particular commodity frameworks. Examples are the Perl Cog Kit [13] (for Perl) and pyGlobus [10] (for Python).

We have no knowledge of a scheduler implemented using interpreted languages. Our tool allows for deploying parallel and distributed applications in grid clusters (even with virtual IPs) without much overhead. Computationally expensive procedures could be compiled and accessed directly from ALua, allowing for efficiency and flexibility. Other advantages of this approach include the rapid development intrinsic to the Lua language and the simplicity of installation and configuration of ALua. These factors allow us to easily experiment with different policies and configurations.

The system allows for opportunistic computing and also for dedicated clusters. Adaptation is achieved by modifying the number of processes composing an application or by job migration. The job migration is initiated after detecting a performance degradation surpassing the pre-defined expectations. However, another level of adaptation can be carried out by modifying the running application itself, dynamically inserting new code in running processes or even new processes.

The flexibility offered by our approach allows, for instance, including at run time a new metric in the scheduler's view of the system state. This may imply adding a new property to the monitor to extract the parameter, subscribing to be notified when the parameter changes and changing the scheduler function to take in account that new information. It can be simply done by:

1. creating a new process and joining it to the Job Manager application;

2. sending a message to every monitor with the new sensor function, the property registration request and the notification request;

3. sending a message to the scheduler containing the function that implements the new algorithm that takes into account the included parameter. This could be written in Lua or encapsulated in a dynamic library.

Provisions must be taken to guarantee that only users with administrator privileges would be able to perform such simples but powerful operations.

The major disadvantage of our proposal is the compromise between flexibility and security. We plan on modifying ALua to include messaging authentication.

As mentioned before, the presented work is still in progress, and is part of a larger project that investigates what may be gained from the flexibility an interpreted language offers in grid environments. We intend to complete the integration of the scheduling tool with the grid, as discussed in Section 5.2. We are also creating an API to allow for the execution of MPI programs inside ALua clusters, and developing better monitoring facilities and fine-grain management. On a longer perspective, we are looking for ways to link our work with application scheduling, for building application-aware resource managers able to optimize the resources allocation.

8. ACKNOWLEDGMENTS

This work has been partially funded by CNPq and PCI/LNCC. We would like to thank Pedro Bastos for the invaluable assistance.

9. REFERENCES

[1] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *The International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[2] Chimera grid tools. http://rotorcraft.arc.nasa.gov/cfd/CFD4/CGT/man.html.

[3] J. Chin and P. V. Coveney. Towards tractable toolkits for the grid: a plea for lightweight, usable middleware. Technical Report UKeS-2004-01, National e-Science Centre, February 2004.

[4] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, London, UK, 1998. Springer-Verlag.

[5] A. L. de Moura, C. D. Ururahy, R. Cerqueira, and N. Rodriguez. Dynamic support for distributed auto-adaptive applications. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 451–458, Washington, DC, USA, 2002. IEEE Computer Society.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[7] GRAM Job Manager Scheduler Tutorial. http://www-unix.globus.org/api/c-globus-2.4/globus_gram_job_manager/html/globus_gram_job_manager_interface_tutorial.htm.

[8] Grid Analysis and Display System (GrADS). http://grads.iges.org/grads/grads.html.

[9] R. Ierusalimschy, L. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[10] K. R. Jackson. pyGlobus: A Python interface to the Globus Toolkit$^{\text{tm}}$. *Concurrency and Computation: Practice and Experience*, 14(13-15):1075–1083, 2002.

[11] K. Keahey, K. Doering, and I. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 34–42, Washington, DC, USA, 2004. IEEE Computer Society.

[12] LSF home page. www.lsf.org.

[13] S. Mock, M. Thomas, M. Dahan, K. Mueller, C. Mills, and G. von Laszewski. The perl commodity grid toolkit. *Concurrency and Computation: Practice and Experience*, 14(13-15):1085–1095, 2002.

[14] Portable Batch System home page. http://www.openpbs.org/.

[15] C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, 2005. to appear.

[16] S. Rossetto and N. Rodriguez. Integrating remote invocations with asynchronism and cooperative multitasking. In *Third International Workshop on High-level Parallel Programming and Applications*

(HLPP 2005), Warwick, UK, 2005.

[17] Resource Language Definition. http://www-fp.globus.org/gram/rsl_spec1.html.

[18] J. Schneider and O. Nierstrasz. Components, scripts and glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[19] D. Thain, T. Tannenbaum, and M. Livny. *Condor and the Grid*, pages 299–335. John Wiley, 2003.

[20] C. Ururahy and N. Rodriguez. Programming and coordinating grid environments and applications. *Concurrency and Computation: Practice and Experience*, 16(5):543–549, 2004.

[21] C. Ururahy, N. Rodriguez, and R. Ierusalimschy. Alua: flexibility for parallel programming. *Computer Languages*, 28(2):155180, December 2002.

[22] G. von Laszewski, I. T. Foster, and J. Gawor. Cog kits: a bridge between commodity distributed computing and high-performance grids. In *Java Grande*, pages 97–106, 2000.