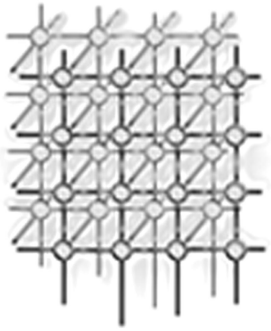
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat: Pract. Exper. 2003; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

Programming and coordinating grid environments and applications



C. Ururahy*,† Noemi Rodriguez†

Computer Science Department – PUC-Rio,
Rua Marquês de São Vicente 225, 22453-900, Rio de Janeiro-RJ, Brazil

SUMMARY

The heterogeneous and dynamic nature of Grid environments place new demands on models and paradigms for parallel programming. In this work we discuss how ALua, a programming system based on a dual programming language model, can help the programmer to develop applications for this environment, monitoring the state of resources and controlling the application so that it adapts to changes in this state.

KEY WORDS: distributed systems; parallel applications; Grid computing; interpreted languages; event-driven communication; middleware; coordination; dynamic adaptation.

1. INTRODUCTION

Parallel programmers are currently facing new issues related to harnessing the computing power available through large-area networks. Grid initiatives and projects [1, 2] seek to create a distributed computing infrastructure for highly demanding scientific and engineering applications. Grids present a highly heterogeneous and dynamic configuration, in contrast to conventional parallel machines. One important technique for dealing with these variations is to use *adaptive strategies*, allowing the program to react dynamically to changes in the environment [3]. Besides, it often does not make sense to have to stop an application and begin processing all over again just because it was launched with an inadequate configuration.

*Correspondence to: Computer Science Department – PUC-Rio,
Rua Marquês de São Vicente 225, 22453-900, Rio de Janeiro-RJ, Brazil
†E-mail: {ururahy, noemi}@inf.puc-rio.br
Contract/grant sponsor: CNPq-Brazil

These issues place new requirements on parallel programming, shifting its traditionally tight focus on performance issues. The use of an adequate programming language for coordinating a parallel application gains new importance. Coordination models [4] support the separation of concerns of the computation itself from those of cooperation and communication between computational components, often proposing distinct languages for programming these two activities.

In this paper, we discuss ALua, a distributed programming system based on C and Lua, an interpreted language, and the flexibility that this system can bring to Grid environments. One of the important characteristics of an interpreted language is allowing for interactivity: With an interpreted coordination language, the programmer may use a "coordination console" to monitor and control the application. Besides, facilities such as dynamic typing and reflexivity allow us to create high-level interfaces for libraries which are powerful but very complex to use in traditional compiled languages.

In the next section we present the ALua model. Section 3 presents the work we have been doing using ALua for Grid environments and applications and discusses how this approach can simplify monitoring and adaptation. Finally, in Section 4 we draw our conclusions.

2. ALUA

The ALua proposal is to use a dual programming model for parallel applications, where ALua acts as a linking element, allowing pre-compiled parts of the program to be executed in different computers. In this context, applications are divided in two parts, kernel and configuration, usually written in different languages. The kernel implements the basic components of the system and is usually written in a compiled statically typed language, such as C or C++. The configuration part, that is usually written in an interpreted language, connects these components defining the final shape of the application [5]. Using this model, we can build flexible applications without compromising their performance [6].

ALua [6] is an event-driven communication model for parallel distributed applications, based on the interpreted language Lua [7]. An important feature of an interpreted language is the support for executing dynamically created chunks of code. In ALua, messages are chunks of code that will be executed by the recipient. This provides a very simple yet very powerful communication mechanism [6]. There is only one asynchronous communication primitive in ALua, `send`, that sends a chunk of code to another process. There is no equivalent to a `receive` primitive. Each process has a Lua interpreter and an event loop, that manages network and user-interface events. The arrival of a message is treated as a network event. This communication model is very flexible: A programmer can use it to perform simple tasks, such as calling a remote function, but she can also use it for much more complex tasks, such as remotely changing the algorithm a process is executing. In the context of long-term parallel applications, and using an interactive console, this is a powerful possibility that allows the programmer to redefine the application behavior dynamically.

The user interface is a console, where the user can enter arbitrary Lua commands. Each line the user types generates an event. Through simple commands the user can inspect

variables (`print(var)`), change variable values (`var = exp`), send messages to other processes (`send(receiver, msg)`), or even run a program (`dofile("progname")`).

An important characteristic of ALua is that it treats each message as an atomic chunk of code, handling each event to completion before dealing with the next one. This means that there is no internal concurrency in an ALua process. In our experience this is not a hindrance. As we discuss in [6], the use of the event-driven paradigm, as of any other programming paradigm, leads programmers to create specific program structures. Messages must typically be small, non-blocking chunks of code. If an application requires larger actions, a process can always resort to sending a message to itself, as a means of breaking up its code in non-atomic parts, therefore allowing other messages to be received in between. On the other hand, as pointed out in [8], the lack of concurrency greatly simplifies many aspects of distributed programming, since there is no need for synchronization inside one process.

In [6] we show a few examples that illustrate useful programming techniques in ALua, borrowed from other event-oriented architectures.

3. THE ALUA MODEL: FLEXIBILITY FOR THE GRID

Among the main Grid technologies, the mechanisms that the *Globus toolkit* [2] offers stand out not only for being used in several sites, but also for the independence of its services, what allows us to select only the services that are relevant to the Grid we want to use. But, at the same time that Globus is very popular for the amount and independence of the services it offers, it becomes very complex to use so many different services and libraries all together.

In this work, we use ALua for monitoring and developing parallel applications for the Grid. ALua is not only a distributed parallel programming model for the Grid, but also a tool for developing, monitoring and adapting Grid applications and monitoring the Grid itself. In the works we developed so far, the ALua model showed to be highly flexible, bringing advantages not only to the final shape of the applications, but also to their development process. ALua allows, for example, the rapid development of application prototypes, and many times there is no need to replace this prototype, because we do not observe a considerable performance loss in the application.

In the Grid computing context, a tool like ALua becomes very important, once the Grid has a dynamic configuration in its definition. The ALua model can be used to perform a bunch of different tasks. For example, the interpreted nature of ALua can be useful for interactively controlling applications with large execution times. The following code shows how the programmer can use the console to redefine the behavior of function `alua.send` on the fly to instrument the application, creating a log file of the communication messages. In Lua, functions are first-class values, and function names are regular global variables that happen to contain a function.

```
old_send = alua.send
function new_send(to, msg)
   write(alua.mytid .. " sending mesg to " .. to)        -- write log file
   msg = format("write(alua.mytid..' received mesg from %s');", alua.mytid) .. msg
   old_send(to, msg)
```

```
end
alua.send = new_send
```

(The .. is the string concatenation operator in Lua.) After this redefinition, function `alua.send` will execute the code

```
write(alua.mytid .. " sending mesg to " .. to)
```

every time it sends something, and it will instruct the receiver to execute

```
write(alua.mytid .. ' received mesg from <SENDER'S TID>')
```

upon receiving the message.

If we run the above code in only one process, only its messages will be logged. To keep track of all messages in the system, we can broadcast the above code to all processes. To restore the original function, we simply send the message `alua.send = old_send` to each process.

We have integrated ALua with existing Grid communication and resource management mechanisms so that we can dynamically analyze the behavior of applications and the Grid. Through an ALua console we can monitor the Grid nodes that are part of a specific computation, as well as their resources. Also, we can send code that can change the behavior of a node, adapting an application dynamically, without the need to define what this adaptation will be in advance.

For the monitoring and adaptation infrastructure, we use Globus services for resource allocation management, security infrastructure and directories. With the meta directories service (MDS) [9], we can find out not only the nodes available for executing an application, but also their characteristics and resources. The idea is to make these services available to the ALua programmer, so that she can use the infrastructure that is already available in the Grid platform in a much more flexible and dynamic way, in the applications as well as in a management console. We believe that accessing these services through ALua, together with the flexibility the ALua model provides, greatly simplifies a programmer's life.

We made an experiment in which we linked the Lua library with a few libraries from Globus: GRAM and DUROC (Globus Resource Allocation Management), and LDAP (used in the Meta Directories Service). We then built the ALuaMonitor, a monitoring mechanism, based on previous work [10]. A monitor is a program that has a *timer* and that from time to time verifies the state of a list of properties of user defined resources. For each monitored property, the user can register a callback, that is responsible for performing the adaptation itself. It can be written in Lua or C and can use other libraries, such as MPI. We used the LDAP binding to Lua to query the dynamic information that the MDS provides. This binding is more than just a mapping of LDAP functions to Lua, but rather a flexible high-level interface to LDAP. The following code shows how to define a property that indicates what percentage of CPU is available in the last 1, 5 and 15 minutes. The ALuaMonitor obtains this information from MDS.

```
1 ldap_obj = LDAP:new({server = "server.par.inf.puc-rio.br"})
2
3 lookupF = function(self) -- Calculates the value of the property.
4                          -- In this case, it is a table containing
5                          -- the three desired values.
```

```
 6    ldap_obj.filter="Mds-Device-Group-name=processors"
 7
 8    entries = ldap_obj:search({"Mds-Cpu-Free-1minX100", "Mds-Cpu-Free-5minX100",
 9                               "Mds-Cpu-Free-15minX100"})
10
11    -- The attributes Mds-Cpu-Free-* on the first entry
12    -- will be the property's value.
13    local currval = {entries[1]["Mds-Cpu-Free-1minX100"],
14                     entries[1]["Mds-Cpu-Free-5minX100"],
15                     entries[1]["Mds-Cpu-Free-15minX100"]}
16    return currval
17 end
18 -- The value of this property will be updated every 30 seconds!
19 prop = ALuaMonitor:defineProperty("CPU", lookupF, 30)
```

The property value can be retrieved at any time through function `prop:getValue()`.†

For each property we can define different *aspects*. In the following code we show how to define the *Decreasing* aspect of the *CPU* property we created previously. This aspect indicates whether the amount of free CPU is getting smaller over time. In this example, we considered that this happens if the amount of free CPU has decreased by more than 10% in the last 14 minutes.

```
aspectF = function(self, currval, monitor)
   local cons = currval[3] - currval[1]  -- (15 min - 1 min)
   if cons > 10 then
      return "yes"
   else
      return "no"
   end
end
prop:defineAspect("Decreasing", aspectF)
```

To obtain this aspect's value we can use `prop:getAspectValue("Decreasing")`.

Besides defining properties and aspects, we can also create observers for different properties and their aspects. In the following code we create an observer based on the *CPU* property and on its *Decreasing* aspect. The observer's callback function (called *notifyEvent*) will be called in case the amount of free CPU in the last minute is less than 75% and has decreased by more than 10% in the last 14 minutes. The third parameter to function `attachEventObserver` should be a string, that represents a boolean expression, defining a condition that will be checked every time the property value is updated. In the boolean expression, property values can be expressed as `$<property name>` and will be interpreted as `<property>:getValue()` when the expression is evaluated. Similarly, the aspect values can be expressed as `$<property`

†Tables in Lua, represented as {...}, are dynamic associative arrays; they are the basic data-structuring mechanism of the language. In lines 8 and 13–15, the table acts as a regular array with numeric indices. In Lua, functions are first-class values and can be assigned to table fields as in `tb.f = function(...) ... end`. `tb:f(...)` is a syntax sugar for `tb.f(tb, ...)`.

name>:<aspect name> and will be interpreted as <property>:getAspectValue(<aspect name>).

```
observer = { notifyEvent = function(self, event)
                             << code for adapting the application >>
                             alua.send(<<targetProcess>>, <<adaptingCode>>)
                           end}
ALuaMonitor:attachEventObserver(observer, "CPUDecrease",
                                "$CPU[1] < 75 and $CPU:Decreasing == 'yes'")
```

Using the LuaDUROC binding, that allows processes to be spawned in a Grid as part of an application, together with the ALuaMonitor and ALua, we built a simple application that uses this monitoring mechanism to adapt itself to changes in the execution environment. We used a monitor as the one we presented earlier and simulated a CPU intensive usage that triggers the application adaptation.

Using Lua, we were able to build concise interfaces that are very light for the programmers who are using few resources, but still offer a lot of flexibility for the ones who need more complex tasks.

We applied the experiment we just described to control and reconfigure applications that use other communication libraries. Because of its popularity, we chose to use the mpich-g2 [11],an MPI implementation for the Grid.

We changed the structure of an MPI application so that it has an event loop that can receive any ALua or MPI message. An MPI message triggers the part of the original code responsible for the application loop step, while an ALua message can perform instrumentation and adaptation tasks. The monitor described earlier runs independently of the main application and the observer's `notifyEvent` function sends an ALua message to the master process. For the simplest version of this experiment, we exported the relevant configuration parameters in the C code to Lua so that the code sent in ALua messages could adjust them when needed. The degree of integration between Lua and the application kernel will determine the flexibility level and the adaptation possibilities that may be achieved.

4. FINAL REMARKS

We have discussed on-going work using ALua's flexibility to allow the programmer to monitor and control resource usage on the Grid. Although the tools available in the Globus toolkit provide an extensive set of facilities, their integrated use demands a lot of effort.

The idea of our work is somewhat similar to that of the Java CoG Kit [12], in that both propose mappings between Globus services and a specific programming model. As we said before, the goal of this mapping is not just making these libraries available to Lua, but rather to provide a flexible high-level interface to them. We can say we implemented a Lua CoG Kit, that also offers support for rapid prototyping of grid applications. In this line, the LuaOrb system [13] offers mappings between the Lua language and CORBA, COM and Java components, providing interoperability among these components. We believe that integrating both ALua with LuaOrb will allow the application developer to explore grid services (such as

resource management, security, and resource finding) while she develops high-level components using the tools that she finds most appropriate.

The goal of our work is to provide flexible, interactive, and uniform programming interfaces for dealing with different aspects of distributed computing. In the case of the Grid, the ALua interface allows the programmer to deal with the dynamic and heterogeneous nature of the environment and make effective use of the available resources.

REFERENCES

1. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.
2. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.
3. G. Allen, T. Dramlitsch, I. Foster, T. Goodale, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Cactus-G toolkit: Supporting efficient execution in heterogeneous distributed computing environments. In *Proceedings of 4th Globus Retreat*, Pittsburgh, PA, 2000.
4. G. Papadopoulos and F. Arbab. Coordination models and languages. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 46, pages 329–400. Academic Press, Aug. 1998.
5. J. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
6. C. Ururahy, N. Rodriguez, and R. Ierusalimschy. ALua: flexibility for parallel programming. *Computer Languages*, 28(2):155–180, Dec. 2002.
7. R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
8. L. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8):55–61, Aug. 1996.
9. Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *Proceedings of 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, 2001.
10. A. L. de Moura, C. Ururahy, R. Cerqueira, and N. Rodriguez. Dynamic support for distributed auto-adaptive applications. In *Proceedings of AOPDCS (held in conjunction with IEEE ICDCS 2002)*, pages 451–456, Vienna, Austria, July 2002.
11. I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of 1998 SC Conference*, Nov. 1998.
12. Gregor von Laszewski, Ian Foster, and Jarek Gawor. Cog kits: a bridge between commodity distributed computing and high-performance grids. In *Java Grande*, pages 97–106, 2000.
13. R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. In *International Symposium on Distributed Objects (DOA'99)*, Edinburgh, Scotland, 1999. IEEE Press.