The use of interpreted languages for implementing parallel algorithms on distributed systems

Noemi Rodriguez
noemi@inf.puc-rio.br

Cristina Ururahy
ururahy@inf.puc-rio.br

Roberto Ierusalimschy
roberto@inf.puc-rio.br

Renato Cerqueira
rcerq@inf.puc-rio.br

PUC-RioInf.MCC13/96 March, 1996

Abstract: This paper proposes the investigation of the role of interpreted languages as a tool for the development of parallel programs in distributed environments. Besides, it argues that an event-driven approach simplifies many aspects of concurrent programming, since processes are never blocked on communication primitives.

These ideas have been experimented with through the implementation of several solutions to the classical problem of termination detection. This implementation has used a programming environment based on the extension language Lua.

Keywords: Agents, Event-driven Programming, Concurrent Programming, Termination Detection Algorithms, Interpreted Languages

Resumo: Este artigo investiga o uso de linguagens interpretadas como uma ferramenta para o desenvolvimento de programas paralelos em ambientes distribuídos. Além disto, ele apresenta como uma abordagem orientada a eventos pode simplificar muitos dos aspectos de programação concorrente, já que processos nunca são bloqueados no uso das primitivas de comunicação.

Essas idéias foram experimentadas através da implementação de várias soluções para o problema clássico de detecção do término de processamentos distribuídos. Esta implementação usou um ambiente de programação baseado na linguagem de extensão Lua.

Palavras-chave: Agentes, Programação Orientada a Evetos, Programação Concorrente, Algoritmos para Detecção do Término de Processos Distribuídos, Linguagens Interpretadas

This work has been sponsored by CNPq and CAPES.

1 Introduction

The implementation of parallel programs on clusters of workstations has become quite popular in the last few years. Besides the fact that this architecture has a much lower cost than a parallel machine, it is already available at most academic institutions.

Parallel programs have traditionally been implemented on distributed systems using conventional programming languages, such as Fortran or C, plus a library with support for communication, such as, for instance, PVM [Sun90]. More recently, some effort has been put into developing parallel programming languages. Some of these, as Linda [CG89] and CC++ [CK93], are in reality extensions to existing sequential languages. Others, such as Orca [BST89] and SR [AO93], are languages designed from scratch for parallel programming.

Usually, these different development platforms are compared as to application execution time or parallel efficiency. However, as pointed out in [Fos95], when measuring the performance of a parallel application or programming tool, different metrics may need to be taken into consideration. The ease of construction of an application and the time a programmer spends to implement it may in some cases be more important than the final execution time.

The assumption that programmer effort is an important metric in many cases is adopted by several authors. For instance, the rationale behind the development of parallel programming languages is that it will be easier for a programmer to design, implement, and debug a program written in such languages than when using a sequential language with library calls. Also, the use of rapid prototyping as a technique for program development is becoming more and more widespread [Bro95], and can be expected to extend to parallel programming as well. Some areas of distributed programming, such as WWW services, are already trading efficiency for flexibility through the use of languages such as Perl [Sch93] and Java [jav95]. Of special interest for such services is the facility of exchanging code between different machines, offered by these languages.

Working under this assumption, this paper discusses some advantages of using high level interpreted programming languages for the development of parallel programs. We argue that the use of this kind of language as a basis for parallel programming tools should be further investigated. As support for our argument, we show an example of use of an interpreted language in a classical application in parallel distributed programming, namely, that of detecting program termination. Two different algorithms for termination detection are implemented in Lua, an extension language developed at PUC-Rio [IFC96].

This paper is organized as follows. Section 2 contains a discussion of the major features of interpreted languages and a short description of the Lua programming language. Section 3 describes our implementation examples. In Section 4, this implementation using Lua is compared to implementations using a traditional language, C, and using a distributed programming language, SR. Finally, the last section contains concluding remarks.

2 Interpreted Languages

There is increasing demand for customizable applications. As applications became more complex, customization with simple parameters became impossible: users now want to make configuration decisions at execution time; users also want to write macros and scripts to increase productivity [Rya90, Fra91, Ous90]. This demand is changing the structure of programs. Nowadays, many programs are split in two parts, a *kernel* and a *configuration*, written in two different languages. The kernel implements the basic classes and objects of the system, and is usually written in a compiled, statically typed language, like C or Modula-2. The configuration part, usually written in an interpreted, flexible language, connects these classes and objects to give the final shape to the application [CIS92]. Such configuration languages are also called *extension languages*.

It is important to note that the requirements on extension languages are different from those on general purpose programming languages. Conventional languages, like Fortran and C, put heavy emphasis on efficiency and static verifications, and therefore are usually compiled. Extension languages, on the other hand, emphasize flexibility and expressiveness; for these requirements, an interpreter is more convenient.

Besides their use as extension languages, interpreted languages are also useful for rapid prototyping [Bro95]. Rapid prototyping is being increasingly used in development of new systems, mainly as a tool for requirement acquisition.

2.1 Lua

Lua is a language specifically designed to be used as an extension language [IFC95, IFC96]. As such, Lua has no concept of main program, being instead always called from a *host* program, usually written in C. This program interacts with the Lua interpreter via an API which provides functions to execute pieces of Lua code, to manipulate global variables, and the like.

Lua is small, portable (it is being used in platforms ranging from PC-DOS to CRAY), has a simple syntax and a simple semantics. And it is flexible.

Such flexibility has been achieved through some unusual mechanisms that make the language highly extensible. Among these mechanisms, we emphasize:

Associative arrays (called *tables* in Lua) are a strong unifying data constructor. Dynamic associative arrays directly implement a multitude of data types, like ordinary arrays, records, sets, and bags. They also lever the data description power of the language. Moreover, being implemented as hash tables, they allow more efficient algorithms than other unifying constructors like strings or lists. Tables in Lua are dynamically created objects with an identity. This greatly simplifies the use of tables as objects, and the addition of object-oriented facilities.

Reflexive facilities allow the language a kind of "self manipulation" [Kir92]. Particularly, Lua provides functions to execute a given piece of code, and to dynamically create new functions as well. In a distributed environment, these facilities allow code to be sent from one process to another. Reflexive facilities also help producing highly polymorphic code. Many operations that must be supplied as primitives in other systems, or coded individually for each new type, can be programmed in a single generic form in Lua. Examples are cloning objects and linearization.

C libraries can be easily incorporated to the interpreter, adding new functionalities to the language.

Of special consequence to this work is the *Tche* environment [CI95, CRI95]. Tche is an application which creates an interface between Lua and the operating system, through an event-driven approach. It consists of two components: a basic library and a main loop.

The basic library adds to the language a plethora of graphical functions (which are of no concern in this work), plus only one function for communication, called `send`. This function has a conventional behaviour, asynchronously sending a string to a specified recipient. The system has no symmetrical function to receive a message; instead, messages are received as special *events*.

```
function queryresult (x)
  ...
end

message = "send(A_id, 'queryresult(' .. var .. ')')"
send (B_id, message)
```

Figure 1: A query for the value of variable `var`.

The event loop is responsible for dispatching the OS events to appropriate handlers, written in Lua. Again there is a plethora of interface events, like mouse movements, clicks, keyboard events, etc, and one special communication event. The handler function for this event is called every time the process receives a message, with the message and the sender as its arguments. The default implementation for this handler function is to *execute* the message, assuming that the message is a piece of Lua code. This behaviour gives great flexibility to the communication mechanism, allowing the sender to modify global variables, to call functions, and even to define new functions in the receiver's environment. As an example, in order to query the value of a variable in process B, a process A can execute the code shown in Figure 1. Process A will send to B the message[1]

```
send(Aid, 'queryresult(' .. var .. ')')
```

When B receives this message, it will execute it. Assuming `var` has value `1`, process B will send back to A the message

```
queryresult(1)
```

Finally, upon receipt of this message, the function `queryresult` in process A will be called, signaling the end of the query.

An important property of the above protocol is that it is non blocking. The function `send` is, as mentioned above, asynchronous, and since receiving is carried on through an event driven mechanism, a process is never blocked. Another important property is that each event is handled to completion before the system starts handling the next event.

3 The Implementation of Termination Detection Algorithms using Lua

3.1 The Problem of Termination Detection

The importance of the problem of detecting that a distributed computation has terminated is widely recognized [Ray88, BA90, Tel94]. Some distributed programs terminate explicitly, but it is very common to structure a program in parallel worker processes which become active when they receive messages with some workload from each other, and passive when this workload has finished. In such programs, the problem of detecting that the whole computation has terminated may be non-trivial. As in many cases where global knowledge is necessary, it is not sufficient to simply enquire if each node has finished its work, since a message in transit may change back to active the state of a process which was previously observed as passive.

[1] The operator `..` denotes string concatenation in Lua.

Several algorithms for termination detection are available in the literature. For the purpose of this work, two of these algorithms were chosen to be implemented. The choice did not take into account the efficiency or number of messages exchanged, but focused instead on the diversity of the solutions and their relative simplicity. Since we are using these algorithms only as programming examples, we will not give great attention to some details. For a comprehensive discussion about these algorithms see [Ray88, Mat87, BA90]

In most termination detection algorithms, a detection procedure is started by one process (which may or not be one of the worker processes), which after receiving the results of this procedure, is able to determine whether termination has occurred. This process will be called the *initiator*. The messages exchanged between processes to carry out the detection procedure are called *control* messages, as opposed to messages exchanged due to the execution of the parallel program itself, called *basic* messages.

The algorithms used in this work assume that the processes involved in the parallel computation execute each at a different network node, and that the communication links between these nodes create a connected graph. Each process is essentially passive, becoming active only when it receives a basic message. At this moment, it may send a number of messages to its neighbours. The receipt of a basic message and the resulting transmissions is an atomic operation, ie, no other messages are received between the receipt of a message and the transmission of messages sent as its consequence.

It is interesting to notice how well the event driven paradigm fits into this model. As pointed out in the previous section, Lua works exactly this way, always processing an event to completion before receiving the next one.

3.1.1 The Four Counter Algorithm

To find out whether any messages are still in transit, one very obvious solution is for each process to count the number of sent and received messages. If the initiator could demand from all processes the values of these counters for a given instant of absolute time, then it would be possible for it to safely determine the number of messages in transit. If S is the accumulated value of the counters of sent messages and R the accumulated value of counters of received messages, then the program would be terminated if $R = S$. However, because of different local times and different intervals for the control messages to reach their destinations, the assumption of absolute time is not feasible.

One solution to this problem, described in [Mat87] as the *sceptic* algorithm, is for the initiator to ask for the values of the counters two times in a row. After receiving the values of one round and checking that $R = S$, the initiator immediately starts a second round and obtains new accumulated values R' and S'. It is shown that if $R = S = R' = S'$, then the system was terminated after the first round.

To avoid having two rounds of messages for accumulating counters, Mattern proposed a variant of the procedure delineated above, using a spanning tree approach. One way to build the spanning tree is using a Probe/echo algorithm; this is the solution we have adopted here.

Probe/echo algorithms [And91] use a parallel graph traversal method to implicitly create a spanning tree for the node graph. These algorithms may be considered the concurrent analog of depth first search algorithms, and may be used for a variety of applications, such as broadcast of a message, collecting information, and so on. Roughly, they work as follows. A *probe* is a message sent by a node to its successors, an *echo* is a subsequent answer. When a node receives a probe from a neighbour, it send probes to all its other neighbours, and proceeds to wait for their answers. When it receives all the echos it expects, the node can send an echo to the neighbour which first sent it the probe. In graphs with cycles, a node may receive more than one probe. In this case, it can answer all but the first probe immediately, with an empty message.

From the point of view of each visited node in our parallel computation model, the Probe/echo algorithm creates two phases, a "down" phase characterized by the receipt of a first control message (which is propagated to all the neighbours), and a "up" phase characterized by the receipt of the last echo from its neighbouring nodes. These two phases are used as the two waves necessary for the implementation of the sceptic algorithm.

3.1.2 The Coloured Ring Method

In this method, a ring structure is imposed on the processes for the purpose of detection. This ring has no relation to the topology of the graph formed by the communication links. The processes are arranged in the ring in an order P_0,...,P_{n-1}, and communication on this ring occurs between each pair of processes P_i and P_{i-1}, and between P_0 and P_{n-1}. The algorithm, as described in [Ray88], assumes that communication is instantaneous and no messages are lost. We will see later how the first of these requirements has been relaxed in our implementation.

Supposing P_0 is the initiator, the detection procedure is started by P_0 launching a token which circulates on the ring. Each process passes the token on only when it is passive. In a first approximation, when P_0 receives the token back, it will know all processes are passive. However, a process may pass the token on and later be reactivated by a message from some process which has not yet been visited by the token. To avoid this, a colour is given to each process and to the circulating token, and the following procedure is adopted.

All processes and the token are initially white. If a process P_i sends any message to a process P_j such that $i < j$, then P_i turns black. This is to indicate that P_i may have reactivated a process which had already been visited by the token. When retransmitting the token, each process acts as follows. If the process is white, the token is passed on with its colour unchanged. If the process is black, a black token is transmitted. In any case, the process becomes white after retransmitting the token.

Given these rules, if P_0 receives back a white token and it is also white, termination is detected.

In order to relax the requirement of instantaneous communication, a further rule has been added to the set above. When a process sends any messages, it remains black until the receipt of all these messages has been acknowledged. After this, it will remain black or not according to the rules above.

3.2 Implementation in Lua

The distributed program model which was described in Section 3.1 has been implemented in the following way. Each network node is represented by a Lua/Tche process running on a different physical machine. Communication links are simulated by providing each node with information about which machines are its neighbours. One particular machine runs the initiator code. This code, besides controlling the termination detection algorithm as described above, is also responsible for sending to all other nodes part of the code they should execute.

The code executed on all other nodes is structured in two parts. One of these parts contains the communication skeleton which is the same for any detection algorithm used. A second part of the code comprehends the functions which are dependent on the chosen termination detection procedure. This is the part of the code sent by the initiator. It has been called the *server* code, to emphasize the idea that it implements a specific service, the detection. With this structure,

```
   function Info(argx)
   -- Process has received a basic message
      MReceived = MReceived+1
      InfoS(argx)   -- tell server that a basic message has arrived
      >> here goes the code to process basic message <<
      >> This code must set variables NMessages, with the number of  <<
      >> basic messages to be sent, and create the list TargetProcs, <<
      >> with the processes to which messages will be sent <<
      MSent=MSent+NMessages
      SendS(TargetProcs)   -- tell server that messages have been sent
   end

   function Init()
      MSent=0
      MReceived=0
   end

-- Main Body
>> initialization of variables and communication links <<
```

Figure 2: Basic communication skeleton

it has been possible to implement a function, called *SetAlgorithm*, which allows the user to interactively chose the detection algorithm to be used in the next execution round.

The server code consists of a set of functions which are called by the basic communication skeleton when sending or receiving messages, at "key" moments of operation. This set comprises the following functions:

- `InfoS(sender)`, called when the process receives a basic message from `sender`;
- `SendS(targetprocs)`, called to signal that each process in the list `targetprocs` will receive a basic message;

besides some other auxiliary functions, not worth mentioning here. Besides these, the server also defines a function `Message`, which each process invokes in another processes in order to implement the termination algorithm. The different implementations of these functions are discussed in the following sections.

The basic communication skeleton which runs at each node is shown in Figure 2. Function `Info` is called every time the process receives a basic message. It increments the received messages counter, and calls the server function `InfoS`. Then it executes the code to carry on the parallel program, updates variable `MSent` with the total number of messages sent, and tells the sever which messages have been sent, through function `SendS`.

The Four Counter Algorithm

In order to implement this termination detection algorithm, the server only has to implement the function `Message`. Nevertheless, the other server functions (`InfoS`, `SendS`, etc) are defined with an empty body, to allow the use of the same skeleton.

As already explained, these server functions are installed on the processes by the initiator. To start the detection procedure, the initiator sends a message like

```
"Message(0, 0, 0, 0)"
```

that is, it calls the function `Message` on an arbitrary process. After that, this function will send messages with the generic format `Message(R, S, R', S')` to all the neighbourhood, which in turn will execute the function `Message`, carrying on the algorithm.

The Coloured Ring Algorithm

In the implementation of this algorithm, several server functions were effectively necessary.

Function `InfoS` sends an acknowledgement of the received message to the process which sent the basic message. This is necessary in order to relax the need for instantaneous communication, as discussed in Section 3.1.2.

`SendS` verifies if any of the processes in the list *targetprocs* may already have been visited by the control wave. If this is the case, it makes the current process black. This function also registers the number of messages which have been sent. The current process will in any case remain black until a corresponding number of acknowledgements is received.

Function `Message`, as before, is invoked by messages in the control wave. To start the detection procedure, the initiator, which is process P_0, sends a message like

```
"Message(0)"
```

representing a white token, to process P_{n-1}. When executing `Message`, each process P_i will send to process P_{i-1} the message `"Message(0)"` or `"Message(1)"`, according to the rules described in Section 3.1.2. When the token reaches P_1, this process will send the initiator a message with a function call which tests whether termination has occurred or not.

4 Comparison with other implementations

The algorithms described above have also been implemented in C and SR, as an experience for the comparison of different programming platforms.

The implementation in C used BSD sockets, which provide basic facilities for sending and receiving messages through untyped streams of bytes.

SR [AO93] is an imperative language for distributed and parallel programming. The basic ideas in SR are virtual machines, resources, and synchronous and asynchronous communication. *Virtual machines* provide a global address space which may be shared between several processes. One or more virtual machines may be started on the same physical machine. Processes in different virtual machines may use several communication models, namely, remote procedure call, rendezvous, dynamic process creation with input parameters, and asynchronous message passing. The last two models were used in the implementation of the termination detection algorithms. Asynchronous message passing is based on typed channels. Finally, *resources* are in some way analog to C++ classes; a resource is a template from which different instances may be created.

In both algorithms, the number of lines of code in the C implementation is about twice that in the SR and Lua implementations. The program in Lua is the one requiring the fewer number of lines, but the difference to SR is not significant, specially considering the obvious difference resulting from the need for declarations.

In terms of implementation effort, the advantage of C is that the programmer needs to learn no new concepts. The facilities for communication provided by sockets are very simple and require little understanding of new concepts. However, once the new programming languages were mastered, the implementation effort needed for C implementation proved to be much bigger than for the other languages.

The implementation effort which was needed for Lua and SR proved again to be roughly similar. This was quite interesting, since the design of Lua did not at any moment take the goal of distributed programming into account. However, there was one major drawback introduced by this lack of specific support for process control: it was not possible to implement in Lua exactly the same model used in C and SR. In C and SR, each process, after deciding which messages would be sent, created a new process to handle the transmission of the messages itself. In this new process, a random delay was introduced before the transmission of each message, thus emulating arbitrary transmission delays. This model could not be implemented in the current version of Tche/Lua due to the restriction that only one process can be running on each physical machine (there is no provision for the identification of specific processes in a message).

5 Final Remarks

Besides the algorithms described in this paper, we have used Lua to implement a number of other distributed algorithms, such as heartbeat and filter algorithms. This experience is described in [CRI95], which also describes the extensions made to Lua to implement inter-process communication. The results of the experiment described in this previous work and in the present one show the importance of studying interpreted languages as a tool for parallel programming in distributed systems.

Doubtless, the ability to transmit code along with messages was one of the most important features introduced by using Lua. As mentioned in the introduction, this ability is already being explored in the context of WWW-like applications. It is interesting to note that one of the major problems of using "imported" code in these applications, which is the threat presented to security, is much simpler to deal with in the context of distributed parallel programming.

One interesting use of this ability has been shown by the architecture used in our implementation. A framework based on a chosen communication model for a parallel program was built, allowing for experimenting different implementations, with no need to test and debug the basic communication pattern for each variant.

Some parallel programs are structured in a model where the different workers do not carry out complementary parts of the global task. Instead, each worker uses a different strategy to search for a global solution, and eventually announces the others may stop their search, if a terminating condition is reached [PR95]. In this kind of program, it could be very interesting to have a scheduler which sends the code to be executed through communication channels.

The specific system which we used, Lua/Tche, was not specifically designed for distributed programming and, on this account, suffers some limitations relating to process control. Nevertheless, it was possible to achieve great simplicity in these implementations. This can be partially credited to the event-driven comunication model adopted in Tche. On one hand, this model requires each event to be handled to completion before dispatching the next one. On the other hand, processes are never blocked. We feel that it is worthwhile to further investigate pros and cons of this model.

References

[And91] G. Andrews. Paradigms for process interaction in distributed programs. *Computing Surveys*, 23(1), 1991.

[AO93] G. Andrews and R. Olsson. *The SR Programming Language*. Benjamin Cummings, 1993.

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.

[Bro95] F. P. Brooks. *The Mythical Man-Month: essays on software enginnering*. Addison-Wesley, anniversary edition, 1995.

[BST89] Henri Bal, Jennifer Steiner, and Andrew Tanenbaum. Programming languages for distributed computing systems. *Computing Surveys*, 21(3), 1989.

[CG89] N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3), 1989.

[CI95] A. Carregal and R. Ierusalimschy. Tche — a visual environment for the Lua language. In *VIII Simpósio Brasileiro de Computação Gráfica*, pages 227–232, São Carlos, 1995.

[CIS92] D. Cowan, R. Ierusalimschy, and T. Stepien. Programming environments for end-users. In *12th World Computer Congress*, pages 54–60 Vol. A–14, Madrid, Sep 1992. IFIP.

[CK93] K. Chandy and C. Kesselman. A declarative concurrent object oriented notation. In *Research Disrections in Concurrent Object Oriented Programming*. MIT Press, 1993.

[CRI95] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. Uma experiência em programação distribuída dirigida por eventos. In *PANEL95 — XXI Conferência Latino Americana de Informática*, pages 225–236, Canela, 1995.

[Fos95] I. Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[Fra91] N. Franks. Adding an extension language to your software. *Dr. Dobb's Journal*, 16(9):34–43, 1991.

[IFC95] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Reference manual of the programming language Lua version 2.1. Monografias em Ciência da Computação 08/95, PUC-Rio, Rio de Janeiro, Brazil, 1995. (available by ftp at `ftp.inf.puc-rio.br/pub/docs/techreports`).

[IFC96] R. Ierusalimschy, L. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6), 1996.

[jav95] Javabook, 1995. available through ftp at `ftp://java.sun.com` in directory `/docs/JavaBook.ps.tar.Z`.

[Kir92] G. N. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems*. PhD thesis, University of St. Andrews, St. Andrews, UK, 1992.

[Mat87] F. Mattern. Algorithms for ditributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[Ous90] J. Ousterhout. Tcl: an embeddable command language. In *Proc. of the Winter 1990 USENIX Conference*. USENIX Association, 1990.

[PR95] S. C. Porto and C. C. Ribeiro. Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints. *Journal of Heuristics*, 1(2), 1995.

[Ray88] Michel Raynal. *Distributed Algorithms and Protocolos.* Wiley, 1988.

[Rya90] B. Ryan. Scripts unbounded. *Byte*, 15(8):235–240, 1990.

[Sch93] Randall Schwartz. *Learning perl.* O'Reilly and Associates, 1993.

[Sun90] V. Sunderman. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, December 1990.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.