

ALua: Flexibility for Parallel Programming*

C. Ururahy^{a †}, N. Rodriguez^{a ‡ §}, and R. Ierusalimschy^{a ¶}

^aDepartamento de Informática – PUC/Rio
22453-900 — Rio de Janeiro, Brazil

In this paper we present ALua, an event-driven communication mechanism for developing distributed parallel applications, based on the interpreted language Lua. We propose a dual programming model for parallel applications, where ALua acts as a gluing element, allowing precompiled program parts to run on different machines. We show, through examples, how three types of applications can benefit from the flexibility that derives from this model. We then present a study of ALua’s performance, by comparing execution times of two parallel applications written in ALua with their counterparts written in PVM. **keywords:** interpreted languages, parallel programming, rapid prototyping.

1. Introduction

The use of prototyping as a technique for program development is becoming more and more widespread. Interpreted languages are considered highly adequate for this type of programming, due to the degree of flexibility and interactivity they offer [1,2]. Interpreted languages have also been gaining popularity as *glue* languages; in this context, applications are split in two parts, a *kernel* and a *configuration*, usually written in two 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, usually written in an interpreted language, connects these components to give the final shape to the application [3,2]. With this design, we can build flexible applications, without compromising their performance.

The design of a programming language always involves a trade-off between performance and flexibility. Interpreted languages, such as Lua, Perl, or Tcl, are usually highly flexible; for instance, it is quite easy to modify an application without stopping it. Languages such as C or Fortran, on the other hand, stick to performance. They have more strict type systems, they need detailed information about memory use, and they are slower to compile and link; on the other hand, they can be more than an order of magnitude faster than an interpreted language. Using both kinds of languages in an application allows us to have the best of both worlds, as we can choose what must be fast and what must be flexible.

*Work supported by CNPq

† Author is supported by CNPq-Brasil.

‡ Corresponding author.

§ Author is supported by CNPq-Brasil/CAPES-Brazil.

¶ Author is supported by CNPq-Brasil/CAPES-Brazil.

In parallel programming, the need to deal with issues such as heterogeneity and fault tolerance, specially when dealing with distributed memory environments, has also led to the development of multi-lingual programming models [4]. Many researchers have also identified the need to separate concerns and requirements of the computation itself and of cooperation and communication between computational components. *Coordination models* [4,5] support this separation of concerns, often proposing distinct languages for programming these two types of activities. However, because of the emphasis that is conventionally placed on performance of parallel programs, interpreted languages have not received much attention as potential candidates for coordinating languages. Moreover, compiled languages can also provide static type checking on the bindings between components [6]. This is adequate for the context of distributed-memory parallel machines, in which information about the executing environment is generally available before program execution.

However, parallel programmers are currently facing new issues related to harnessing the computing power available through large-area networks. For instance, grid initiatives and projects [7,8] 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. Resource availability on a grid can suffer great variation over time and space. As discussed in [9], one important technique for dealing with those variations is to use *adaptive strategies*, allowing the program to react dynamically to changes in the environment. Besides, in general, programs that can benefit from running on a computational grid are long-running applications. Often, it does not make sense to have to stop the application and begin processing all over again just because the programmer launched the application with inadequate parameters or inadequate monitoring procedures. It should be possible to gather data about the execution and act upon the program while it is running [9].

In light of these requirements, the use of an interpreted language for coordinating a parallel application gains new importance. 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. On the other hand, given the unstable and dynamic configuration of grid environments, the performance of the coordination language, although important, is not so critical as in conventional machines.

In this work, we present ALua, a system for programming parallel applications in distributed-memory environments based on a dual programming-language model. ALua uses the extension language Lua [10] to coordinate the interaction between components written in C. Like other distributed environments, an application in ALua is composed by a group of processes running in multiple hosts and communicating through a network. Lua code handles all communication among processes (and therefore defines the architecture of the application), while C functions handle the CPU-intensive tasks in each process.

Another important characteristic of an interpreted language is the provision of a mechanism for execution of chunks of code created dynamically. In ALua, messages are chunks of code to be executed by the receiver. This provides a very simple, and yet very powerful, communication mechanism. There is only one communication primitive, *send*, that sends a chunk of code to another process. There is no equivalent to a *receive* primitive.

Instead, ALua uses an event-driven programming model, where the arrival of a message is handled as an event. This communication mechanism is quite flexible: A programmer can trivially use it for simple tasks, like invoking a remote function, but he can also use it for much more complex tasks, like remotely changing the algorithm that a process is running. In the context of long-running parallel applications, and with the availability of an interactive console, this is a powerful possibility, and allows the programmer to redefine dynamically the behavior of the application.

We first described ALua, in a simplified version, in [11], which presented implementations for several classical distributed algorithms (such as probe-echo, heart beat and filtering) in order to evaluate the event-driven communication paradigm. In [12] we concentrated on a specific class of distributed algorithms, for termination detection. The next step was to evaluate the performance and ease use of ALua in more real-sized applications [13].

In this paper, we discuss the use of ALua in distributed parallel applications. We show the flexibility that can be gained with ALua by dynamically injecting new code into an application. The examples show how this mechanism can be useful both for changing parts of the application, for prototyping purposes, and for introducing monitoring facilities in long-running applications. We also perform some experiments to evaluate the performance penalty we incur by using an interpreted language. Since we are using an interpreted language, a significant loss of performance was foreseen. However, by keeping the main processing kernel in a compiled language, we obtained unexpectedly good results.

In the next section we describe the current implementation of ALua. Section 3 shows some small example programs, and how to implement simple communication patterns with ALua. Section 4 discusses more complex examples, and shows the flexibility achieved with ALua. Then, in Section 5, we present two applications previously written in C and PVM and compare execution times of the ALua versions and their compiled counterparts. Finally, in the last two sections we discuss related work and draw some conclusions.

2. The ALua System

In ALua, a program is composed of processes called *agents*, running on one or more physical machines. These processes communicate using an asynchronous *send* operation. Each agent contains a Lua interpreter and an event loop, which manages network and user-interface events. Figure 1 shows the structure of an agent. Each event contains a piece of Lua code. The event loop continually receives events, and sends its contents (the Lua code) to the Lua interpreter for immediate execution.

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 agents (`send(receiver, msg)`), or run a program (`dofile("progname")`).

Network events correspond to messages received from other processes. The `send` operation sends a piece of Lua code to be executed in another process. ALua has no explicit operation for receiving a message. The receiver's event loop will automatically execute the received code. The result is an event-driven programming model, compatible with the character of interpreted languages: not very secure, but highly flexible. As we will see, it

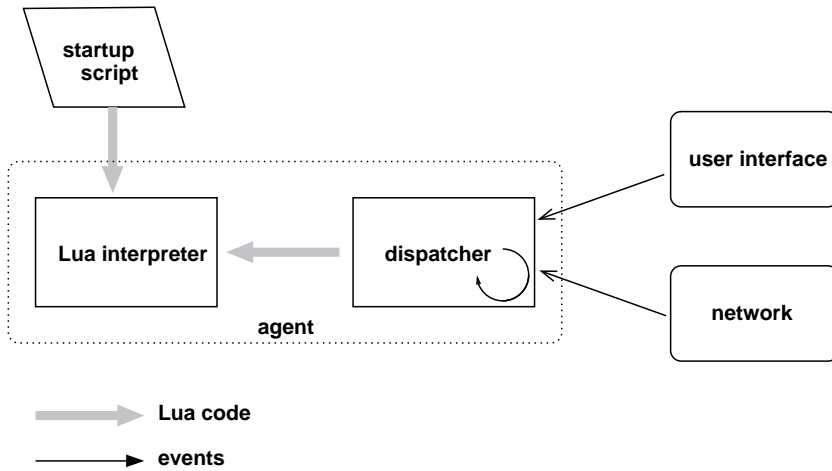


Figure 1. The Structure of an Agent.

is very easy to implement several typical distributed tasks with this mechanism, such as to call remote procedures or to inspect and modify remote variables.

An important characteristic of ALua is that it treats each message as an atomic chunk of code. Because it adopts the event-driven paradigm, it handles each event to completion before starting the next one. This means that there is no internal concurrency in an ALua agent. The resulting programming model is similar to the asynchronous model described in [14], with events triggering atomic actions. In our experience this is not a hindrance. As we discuss in the next section, 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, an agent 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 [15], the lack of concurrency greatly simplifies many aspects of distributed programming, since there is no need for synchronization inside one agent.

When the ALua program creates an agent on a new machine, the system spawns an *ALua daemon* on that machine. Although conceptually ALua agents exchange messages directly between them, the ALua daemons in fact act as intermediates in this exchange (Figure 2), as in PVM [16]. When agent A, in host X, sends a message to agent B, in host Y, this message goes first to the ALua daemon running in X, then to the ALua daemon in host Y, and finally to agent B. Daemons communicate through a positive acknowledgment protocol implemented over UDP. Agents and daemons exchange messages using the Unix-Domain protocol.

Although ALua provides only one primitive communication function (`send`), the system provides other functions, both to support the creation and termination of processes, and to facilitate communication. Following is a list of the main functions used in this paper:

alua.spawn Spawns multiple agents. When all spawned agents are ready to receive

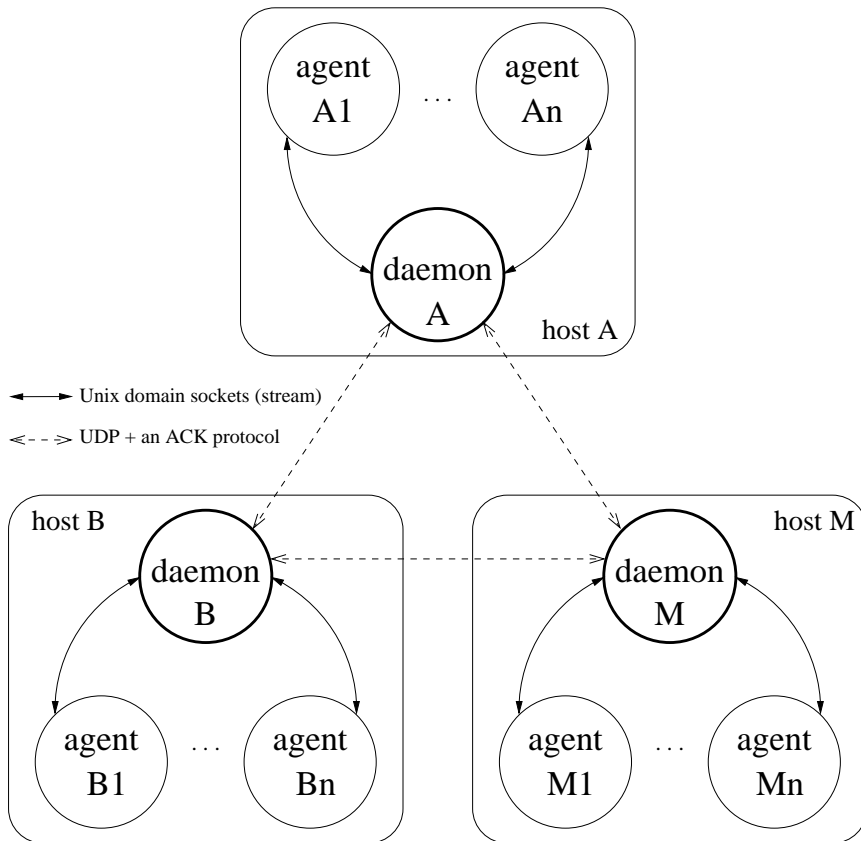


Figure 2. ALua Communication Model.

messages, ALua signals the callee by calling its *alua.spawn_completed* function.

alua.send Sends Lua code to an agent.

alua.mcast Sends Lua code to multiple agents.

alua.tostring Converts a Lua value into a string that represents this value in Lua, so that the value can be inserted in a message to another process.

alua.exit Terminates an agent's execution.

alua.exit_all Terminates execution of all spawned agents.

3. Programming Examples

In this section we present a set of examples to illustrate the basic functionality of ALua.

Our first example shows how to set the value of a variable in another agent:

```

n = 50
-- send variable 'n' to agent B
msg = format("n=%d", n)

```

```
alua.send("B", msg)
```

The `format` function is similar to `sprintf` in C, but instead of using a given buffer it creates and returns a new string (Lua has dynamic memory allocation and garbage collection). The result of this call to `format` is the string `"n=50"`; the last line of our example sends this string to agent B, that will eventually run this piece of code, setting its own variable `n` to 50.

It is not difficult to send more complex data with our mechanism. All we need is a way to serialize the data. ALua offers the `tostring` function to do that.

```
x = {1, 4, 9, 16}    -- table
-- set the value of 'x' in variable 'y' of agent B
msg = format("y=%s", alua.tostring(x))
alua.send("B", msg)
```

In this example, first we create a *table*⁶ with elements 1, 4, 9, and 16. The final value of `msg` in this example is `"y={1, 4, 9, 16}"`; when B runs this code chunk, it creates a table with the given elements and assigns it to `y`.

In the next example, an agent sends the definition of a function `addtb` to agent B; this function returns the sum of the elements of a table.

```
-- "[" and "]" are string delimiters
msg = [[ function addtb(t)
        local tot=0
        for i=1, getn(t) do
            tot = tot + t[i]
        end
        return tot
    end
]]

-- Sends the definition of function f to agent B.
alua.send("B", msg)
```

For the local agent, the whole function text is only a string. (Although we can also use `"..."` and `'...'` for denoting literal strings in Lua, the format `[[...]]` allows strings that span several lines.)

In Lua, functions are first-class values, and function names are regular global variables that happen to contain a function. Actually, we could write the previous message as

```
msg = [[ addtb = function (t)
        ...
    end
]]
```

⁶Tables in Lua are dynamic associative arrays; they are the basic data-structuring mechanism of the language. In this example, the table acts as a regular array with numeric indices.

Therefore, in our example, B may or may not have a previously defined function `addtb` in its environment. If the variable `addtb` is already defined in B when the message arrives, the message will redefine it.

A very common programming technique in ALua is to include, in the message sent to an agent, code that makes the agent send back a result. This creates the possibility of RPC-like calls, except that the caller is not blocked while the answer is pending. In the next example, we use this technique to determine the sum of the elements of a table `y` in agent B. First, agent A runs the next chunk:

```
msg = [[ alua.send("A", "print(" .. addtb(y) .. ")") ]]

alua.send("B", msg)
```

Agent B then receives the message

```
alua.send("A", "print(" .. addtb(y) .. ")")
```

(The `..` is the string concatenation operator in Lua.) Then, it evaluates the arguments to the `send` function. Assuming that table `y` in agent B is `{10, 20, 40}`, the result of

```
"print(" .. addtb(y) .. ")"
```

will be the string `"print(70)"`. This string is the message sent back to agent A, which will then run it and print the number 70.

The next example illustrates another useful programming technique in ALua, borrowed from other event-oriented architectures, which consists of structuring an agent as a state machine. Let us suppose we want agent A to send a message to agents B and C, and to terminate its own execution when it is sure that both B and C have received the message. In a conventional message-passing system, we could do roughly this:

```
-- Agent A
msg_received = 0
msg=[[ alua.send("A", "msg_received=msg_received + 1") ]]
alua.mcast({"B", "C"}, msg)

-- empty loop to wait replies
while msg_received ~= 2 do end

print("The End.")
alua.exit()
```

However, in ALua this code would halt the application. Because ALua has no internal concurrency, the code would block in the loop, and the agent would never handle the incoming events. In an event-driven paradigm, such as in ALua, we could recast this example as below:

```
-- Agent A
function First_Answer ()
```

```

    Answer = Second_Answer
end

function Second_Answer ()
    print("The End.")
    alua.exit()
end

Answer = First_Answer

msg=[[ alua.send("A","Answer()") ]]

alua.mcast({"B", "C"}, msg)

```

In this example, the variable `Answer` represents the state of agent A; it starts waiting for the first answer. After agent A sends the message to agents B and C, it waits until an event arrives. Agents B and C will both receive the message

```
alua.send("A","Answer()")
```

When the first message arrives at A, from either B or C, it triggers function `First_Answer`, so that A goes to a new state, waiting for the second answer. When the second message arrives, it triggers `Second_Answer`, which terminates A.

4. Flexibility

This section discusses the flexibility that we can gain from using an interpreted and event-driven communication model. In contrast with the previous examples, which intended to illustrate how to program common communication tasks in ALua, the designs described in this section are typically hard to replicate in conventional programming systems.

4.1. Self Replicating Message

The first example deals with broadcasting an action in a logical pipeline configuration, in which each node does not have knowledge about the entire configuration.

We assume that a variable `nexthost`, at each agent, contains the name of the next agent in the pipeline. The action we want to broadcast is

```
alua.send("A", "print(..n..)")
```

which will cause each agent to send the value of its own variable `n` back to A asking A to print it. When the system starts, each agent knows nothing about the need for transmitting a message to the next agent, so the message itself must be responsible for its retransmission.

Figure 3 shows a partial solution, where the message is retransmitted only once (from B to C). Note that this message will never reach agent D.

Extending this same solution to deal with a larger number of retransmissions would require an arbitrarily large chunk of code to be transmitted, and would only work if

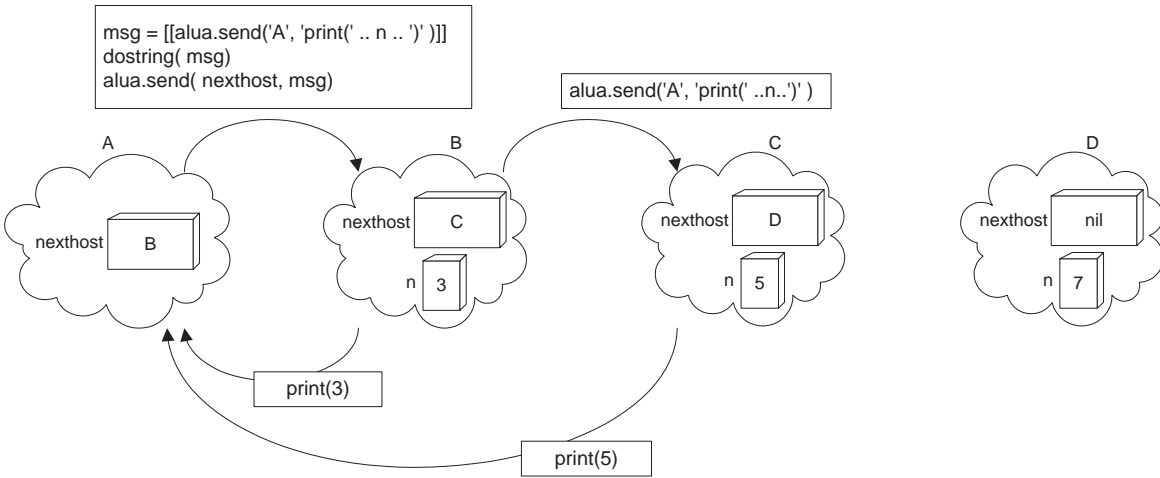


Figure 3. Semi-self-replicating Message

the original sender knew the exact number of agents in the pipeline. Instead, we use a self-replicating message, as shown in Figure 4.⁷

4.2. Tuple Multiplication

[18] discusses five parallel programming *paradigms*, each of them applicable to a set of algorithms with some common structure. As an experiment with ALua, we implemented a sequential and a parallel version for some of these paradigms. Each of these implementations consists of two parts: an application-independent skeleton, and some application-dependent functions. For each paradigm, we have two skeletons, one for a sequential implementation and other for a parallel implementation.

This programming example illustrates how ALua can be useful for testing and prototyping. For each application, the same application-dependent functions are used both in the sequential and in the parallel version of the algorithm. Therefore, the programmer can match different skeletons to different applications dynamically: She can, for instance, load the sequential version to test and debug the application specific code, and then load the parallel skeleton and execute it on the same application. Here we will discuss one of these paradigms, called *Tuple Multiplication* in [18], and show how to use it to implement two algorithms: matrix multiplication and *all-pairs shortest path*.

Tuple multiplication computes an $n \times n$ matrix c as the “product” of two matrices a and b . The matrix elements are obtained by applying a function f to every ordered pair consisting of an n -tuple of a (a row) and an n -tuple of b (a column), that is $c_{ij} = f(a_{i,\cdot}, b_{\cdot,j})$.

This structure captures well the commonality between matrix multiplication and a matrix-based algorithm for computing shortest paths between all pairs of nodes in a directed graph [19]. In the case of matrix multiplication, the tuple elements are rows and columns, respectively, and function f is the dot product. The case of *all-pairs shortest*

⁷We leave the understanding of this example as an exercise to the reader. The reference [17] has an interesting discussion about self-replicating programs.

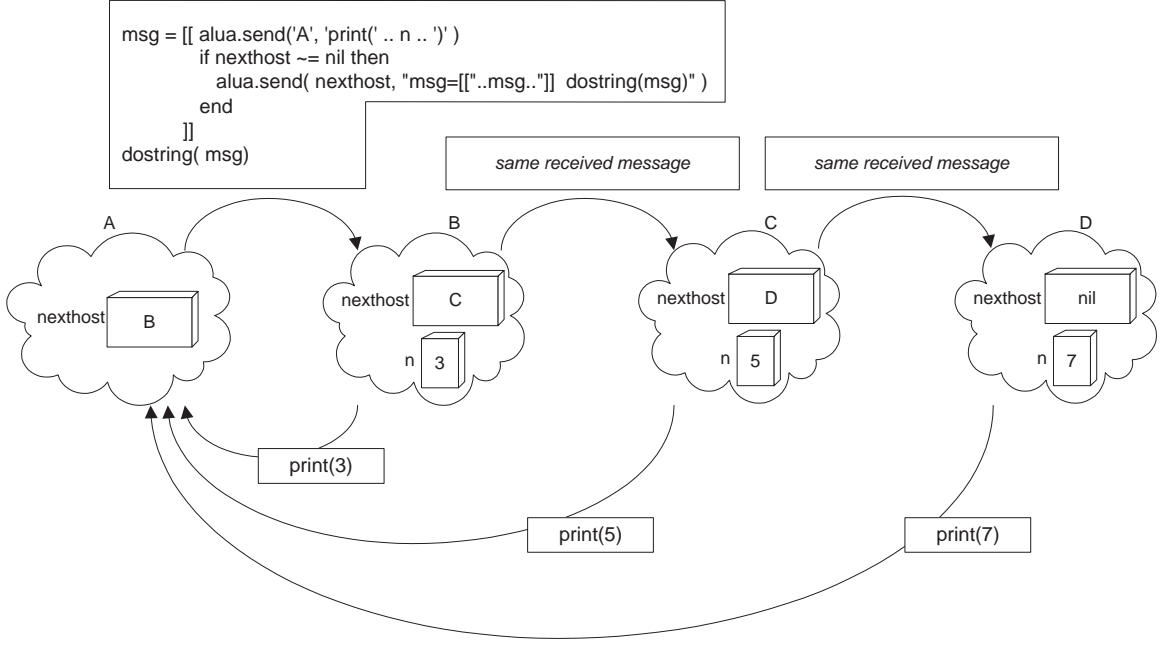


Figure 4. Self-replicating Message

path requires that we first review the algorithm.

As described in [19], the following matrix-based algorithm may be used to determine shortest paths. The algorithm constructs a sequence of distance matrices D^r , where each entry $d_{i,j}^r$ represents the minimum path weight from i to j that passes in at most r intermediate nodes. The initial matrix D^0 (containing paths with no intermediate nodes) is constructed directly from the adjacency matrix:

$$d_{i,j}^0 = \begin{cases} 0 & \text{if } i = j \\ \text{weight of edge } (i,j) & \text{if it exists} \\ \infty & \text{otherwise} \end{cases}$$

Thus, at each step $r + 1$, we substitute the current path weight between i and j , which passes through at most r nodes, for the shortest weight of a path between i and j which traverses at most $r + 1$ nodes:

$$d_{i,j}^{r+1} = \min_{1 \leq k \leq n} (d_{i,k}^r + d_{k,j}^1)$$

This algorithm takes $n - 1$ iterations to construct the *all-pairs shortest path* matrix, containing the minimum weights of paths traversing at most $n - 1$ nodes. Since the computation of each $d_{i,j}$ in iteration r requires n comparisons, this algorithm computes shortest paths in $O(n^4)$ time.

However, as our goal is to compute only the final D^{n-1} , we do not need to compute all intermediate D^r for $1 \leq r \leq n - 1$. Instead, we can compute D^{n-1} with $\log(n - 1)$ steps by combining D^r with itself at each iteration:

$$d_{i,j}^{2r} = \min_{1 \leq k \leq n} (d_{i,k}^r + d_{k,j}^r)$$

```

function multiply(a, b, n)
  local c = {}
  for i = 1, n do
    c[i] = {}      -- create an array
    for j = 1, n do
      c[i][j] = f(a[i], b[j])
    end
  end
  return c
end

```

Figure 5. Tuple Multiplication paradigm: Sequential skeleton.

With this improvement, the running time for the algorithm is $O(n^3 \log n)$.

We now return to the tuple-multiplication paradigm. The paradigm describes the computation of a matrix as a *product* of two other matrices through the application of a function f on pairs of n -tuples.

Figure 5 presents the sequential skeleton for tuple multiplication. Function `multiply` receives two matrices and their dimensions, computes the resulting matrix and displays the result. For simplicity, we assume that both matrices are square.

Figure 6 shows the specific code for the matrix multiplication problem. It defines a single function, `f`, which is invoked by `multiply`. This function receives one row from the first matrix and one column from the second one, and returns one resulting matrix element.

Figure 7 shows the specific code for the all-pairs shortest paths problem. Function `allpaths` creates the initial distance matrix $n \times n$ of a graph with n nodes and calls function `multiply` $\log n$ times to compute the resulting matrix. Again, function `f` is invoked by `multiply`.

Figure 8 shows a skeleton for a parallel paradigm. The master agent executes the parallel version of the `multiply` function, distributing the computation of rows among all available agents. Each agent computes at least `qmin` rows and at most `qmax` rows of the resulting matrix. Each agent receives only the necessary rows of the first matrix, but all of them receive the complete second matrix.

Each of the agents repeatedly executes function `node` (which they got from the master) to compute rows of the final matrix. The agent sends each row to the master as soon as it finishes the computation. Because communication in ALua is asynchronous, the master uses the callback function `finish` to signal to the application that it has obtained a final result.

In matrix multiplication, function `finish` simply exhibits the resulting matrix. However, when computing all-pairs shortest paths, it is necessary to make successive multiplications before the final result is obtained. Therefore, function `finish` calls function

```

function matrix(a, b, n)
    c = multiply(a,
                transpose(b),
                n)
end

function finish (a)
    show(a)
end

function f(ai, bj)
    local n = getn(ai) -- get size
    local cij = 0
    for k = 1, n do
        cij = cij + ai[k]*bj[k]
    end
    return cij
end

```

Figure 6. Matrix multiplication - application specific code.

```

function allpaths(a, n)
    d = a
    m = 1
    while (m < n) do
        d = multiply(d,
                    transpose(d),
                    n)
        m = 2 * m
    end
end

function f(ai, bj)
    local n = getn(ai)
    local cij = infinity
    for k=1,n do
        cij = min(cij,
                  sum(ai[k], bj[k]))
    end
    return cij
end

function finish (a)
    if m < n then -- only for the parallel version
        d = multiply(a, transpose(a), n)
        m = 2 * m
    else
        show_graph(a)
    end
end

```

Figure 7. All-pairs shortest paths - application specific code.

```

function multiply(a, b, dim)
  local bstr = alua.tostring(b)
  n = dim
  for k=1,nmax do    -- nmax is the number of nodes
    -- frow is the first row to be calculated in node k
    -- lrow is the last row to be calculated in node k
    -- Both depend on qmin and qmax (see the text)
    msg = format([[column=%s row={}]], bstr)
    for i=frow,lrow do
      msg = msg .. format(" row[%d]=%s",i, alua.tostring(a[i]))
    end
    msg = msg .. format(" n=%d node(%d, %d)", n, frow, lrow)
    alua.send(Procs[k], msg)
  end
  master_i = 0
  master_c = {}
end

function master_receive(row, i)
  master_c[i] = row
  master_i = master_i + 1
  if master_i == n then    -- result is ready
    finish(master_c)
  end
end

-- This string is sent to slave agents when they are started.
node_code=[[
  function node(r, s)      -- 1 <= r <= s <= n
    for i=r,s do
      c = {}
      for j=1,n do
        c[j] = f(row[i], column[j])
      end
      msg = format("master_receive(%s,%d)", alua.tostring(c), i)
      alua.send("master", msg)
    end
  end
end
]]

```

Figure 8. Tuple Multiplication paradigm: Parallel skeleton.

multiply again after each non-final iteration, and calls function `show_graph` to exhibit the shortest paths matrix when the final result is obtained.

As we said before, the beauty of this approach is that you only have to change the skeleton to change the paradigm.

4.3. An Asynchronous Agents Team

Another interesting example of ALua's flexibility is the implementation of an asynchronous agents team (A-Team) [20] for the Set Covering Problem [21]. The complete program is rather complex, so in this section we will present only a rough sketch of it.

An A-Team is composed by *agents* (autonomous entities working in a cooperative way) and shared memories, creating super-agents with a cyclic data flow and no control flow. The main features of the A-Team architecture are:

- autonomous agents
An autonomous agent makes its own choice when selecting its input data and resource allocation policy. Because autonomous agents are completely independent of each other, new agents can be added to or removed from the system without notifying other agents or a manager.
- asynchronous communication
Agents can read and write data to shared memories without any kind of synchronization among them. There are no logical constraints on shared memory accesses, except for data integrity. Allied to their autonomy, this feature allows agents to work full-time in parallel.
- cyclic data flow
Agents can retrieve, modify and store data in shared memories continuously. Such cyclic data flow allows continuous iteration and feedback among the agents.

Experience with this paradigm indicates that the cooperation among agents tends to generate synergy, i.e., the result produced by them, when seen together, can be better than the sum of the results obtained independently (there is a greater chance of finding a solution close to optimal) [20,21]. These experiences also suggest that an A-Team is a scale-efficient organization, i.e., its performance gets better when new components (such as agents or memories) are introduced in the system.

We started from a previous implementation of A-Teams written in C, which uses the communication package DPSK+P [22], based on a shared objects architecture. The system (described in [21]) is composed by two *servers* that act as solution repositories, representing the shared memory of the A-Team; two *initialization agents* that initialize the repositories; and nine *worker agents* that implement different refining algorithms for the solutions stored in the repositories.

Given a specific instance of the set-covering problem, the typical pattern of behavior of an A-Team agent is to request a solution from the solutions repository, refine this solution, send the new solution back to the repository, and start over. One repository stores dual solutions, the other stores primal solutions. Both repositories simply answer agents' requests. Depending on the input data, this application can keep running for some days before reaching an interesting result.

Over this implementation, we replaced the communication package DPSK+P by ALua. The original system was composed by 15 programs, with 34 modules, where only 18 were directly related to communication activities. We rewrote these 18 modules to use ALua, and kept unchanged the other modules. Our new implementation keeps a Lua console available, so we can enter new commands to the system at any time.

The use of ALua allows us to dynamically reconfigure the system, consult the repository, or even redefine functions. For instance, the following code shows how we can redefine the behavior of function `alua.send` on the fly to instrument the application, creating a log file of the communication messages.

```
old_send = alua.send

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

alua.send = new_send
```

After this redefinition, function `alua.send` will execute the code

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

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

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

upon receiving the message.

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

This example illustrates how the interpreted nature of ALua can be useful for interactively controlling applications with large execution times. The A-team application can take hours to complete: it is extremely interesting for the programmer to be able to change its behavior without having to wait for a new execution. The ALua programmer can use an interactive console to inject monitoring code, as discussed above, or to redefine the behavior of one or more of the agents, allowing parts of the program to be redefined according to partial, observed results.

5. Performance Study

This section describes experiments we conducted to find out the cost we pay for using an interpreted language. We re-implemented in ALua two parallel applications previously

written in C and PVM: an N-body simulator and a tool for distributed volumetric visualization. Keeping in line with the idea of Lua as an extension language, we split the applications in two parts. We wrote the communication functions in ALua, and kept the processing kernel in C.

The original applications were based on a conventional *send/receive* model. To implement them in Lua, we had to modify them so as to use ALua's event-driven model. Most of the modifications were very simple, limited to changing iterative control to the state machine model discussed in Section 3.

These two applications were chosen because they implemented quite different communication patterns, thus allowing us to compare ALua's performance under different conditions. The N-Body simulator uses a peer to peer structure, while the volumetric visualization is closer to a master-workers application.

5.1. The N-Body Simulation Problem

The N-Body simulation problem studies the evolution of an N-body system under the influence of Newtonian gravitational attraction. The bodies consist of a mass, position and initial speed, and are distributed over a finite physical domain.

Our implementation of the N-Body Simulation Problem was based on an existing implementation of the *Barnes-Hut* algorithm [23] written in C and PVM [24].

In the parallel implementation, the physical domain is divided into N regions, and each processor is responsible for the particles in one of the regions. The simulation goes on for a number of iterations; in each iteration each agent computes the forces among its particles, collects information about all particles in the system and redistributes them (as a result of the gravitational forces, particles may migrate from one region to another).

At the end of each iteration each agent collects information about all others, so as to determine particle redistribution. This results in a large amount of communication between agents. The N-Body Simulation Problem also deals with a large data structure, which is exchanged among all agents many times during its execution.

The application is composed of a master agent that starts the worker agents, reads the initial particle distribution, sends it to the first worker and starts its execution. In order to improve performance, some functions, such as the ones that exchange particles information and perform particles redistribution, are written in C, although the main skeleton is written in Lua.

Lua functions add flexibility to the application because they can be redefined at runtime. For instance, we can change function `begin_worker` to display a message for each iteration, by using the master's console to send to each worker a new `begin_worker` definition:

```
> msg = [[ old = begin_worker
           function begin_worker()
             print(iteration)
             old()
           end
         ]]
> alua.mcast(worker_tids, msg)
```

5.2. Distributed Volumetric Visualization

The other application we studied in this experiment was the so called Distributed Volumetric Visualization, based on [25]. The problem consists of visualizing a set of tri-dimensional data, using the *ray casting* optimization technique [26,27].

Figure 9 shows the structure of the application. A *master* agent divides the image to be computed into a number of regions, or sub-images, each region corresponding to a task. These tasks are distributed (dynamically or statically) among N *worker* agents. After all tasks have completed, a *visualizer* agent exhibits the complete image, placing each sub-image in the correct position.

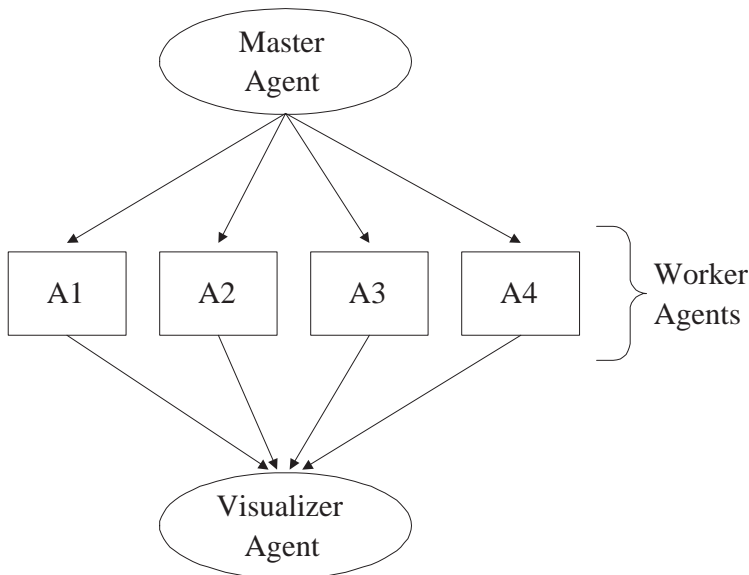


Figure 9. Distributed Volumetric Visualization - structure.

In this application the communication among the agents is restricted to exchanges between the master and each worker, and between each worker and the visualizer. On the other hand, if workers finish computation at approximately the same time, a communication bottleneck may occur.

5.3. Experimental Results

To evaluate the efficiency of these applications, we compared the execution times of both implementations (PVM and ALua) under the same execution conditions. We tested both applications on a cluster of 32 Linux workstations connected by an isolated ethernet network.

For the N-Body Simulation Problem, we carried out 48 experiments, each of them with 5 replications. We took three factors into consideration in the design of the experiments: the number of particles (512, 1024, 2048 and 4096), the number of processors (2, 4, 8 and 16), and the number of iterations (1, 10 and 20).

Table 1 shows the execution times for each experiment. We divided the execution times for experiments with 10 and 20 iterations by 10 and 20 respectively, to keep them comparable to the experiments with 1 iteration. The table also shows the standard deviation between replications of each experiment and the ratio between execution times with ALua and PVM.

In the worst case (1024 particles, 16 processors and 20 iterations) ALua was 4 times slower than PVM while in the best case (2048 particles, 2 processors and 20 iterations) ALua was 2 times faster than PVM. An interesting point in table 1 is the difference in the deviations observed for ALUA and PVM.

For the Distributed Volumetric Visualization application, we conducted 18 experiments, each of them replicated 5 times. In this case, the factors we took into consideration were the partition method (blocks, pixels and scanlines), the number of processors (4, 8 and 16) and the distribution of tasks, which can be either static or dynamic. In the static case, the total number of sub-images is divided by the number of workers and each worker receives its complete workload in the beginning of the algorithm. In the dynamic case, distribution is demand-driven, i.e., each worker initially receives one sub-image and requests a new task each time it completes the previous one.

Table 2 presents the execution times for the pixel and scanline partition methods, with dynamic and static distribution, using 4, 8 and 16 agents. Standard deviation and ratios are again shown. In the worst case (dynamic pixel with 8 processors) ALua was 11 times slower than PVM while in the best case (static scanline with 4 processors) ALua was 1.2 times faster than PVM.

We can observe that the parallelism in both applications did not result in any speedup either in PVM or in ALua. The poor results for parallelism have no impact on our evaluation of ALua as a communication package. They probably indicate an inadequate granularity (computation/communication ratio) for these applications in the specific execution platform, which means we are comparing ALua and PVM under relatively heavy communication requirements; this makes the evaluation only the more interesting.

Figure 10 presents graphs with the ALua/PVM execution time ratios. In the N-Body simulation, ALua is in average around two times slower than PVM. As the number of processors grow, behavior of ALua seems to get worse. In the 16-processor cases, each processor is receiving information from 15 other processors at the end of each iteration, possibly leading to delays in acknowledgments and unnecessary retransmissions. Also, the relative weight of the communication mechanism increases as granularity of computation decreases.

ALua results for the visualization application were worse, specially for the pixel-based partition method: in this case, ALua was in average 9 times slower than PVM. This application does not seem, in fact, to have the characteristics that would recommend ALua's use. A large number of small messages is sent to a single process, as workers complete their allotted tasks and send results to the visualizer agent. In the scanline partition method, ALua did much better: it was in average less than 3 times slower than PVM. The scanline partition seems to reflect a more adequate communication/computation ratio: a comparison of execution times for the PVM application using both methods shows us the weight of communication in the pixel-based partition method.

# part			ALua		PVM		ALua/PVM	
			time (s)	deviation	time (s)	deviation		
512	2	1	0.44	0.00	0.42	0.07	1.05	
		10	0.13	0.00	0.14	0.02	0.97	
		20	0.12	0.00	0.15	0.03	0.80	
	4	1	0.79	0.00	0.50	0.09	1.58	
		10	0.24	0.01	0.17	0.03	1.42	
		20	0.21	0.00	0.19	0.02	1.11	
	8	1	1.45	0.01	0.57	0.13	2.54	
		10	0.45	0.00	0.28	0.04	1.62	
		20	0.40	0.00	0.25	0.03	1.60	
	16	1	3.26	0.07	0.79	0.07	4.13	
		10	1.54	0.05	0.40	0.03	3.89	
		20	1.40	0.05	0.37	0.02	3.83	
	1024	2	1	0.61	0.00	0.56	0.08	1.09
			10	0.28	0.01	0.33	0.03	0.86
			20	0.28	0.01	0.27	0.03	1.04
4		1	0.97	0.01	0.81	0.06	1.20	
		10	0.45	0.01	0.38	0.06	1.18	
		20	0.44	0.01	0.36	0.04	1.24	
8		1	1.60	0.01	0.97	0.18	1.65	
		10	0.71	0.01	0.49	0.05	1.45	
		20	0.67	0.01	0.50	0.05	1.35	
16		1	3.38	0.02	1.28	0.14	2.64	
		10	1.84	0.03	0.65	0.04	2.82	
		20	3.97	0.03	0.93	0.06	4.28	*
2048		2	1	0.96	0.02	0.78	0.13	1.23
			10	0.39	0.00	0.56	0.10	0.69
			20	0.36	0.00	0.72	0.09	0.51
	4	1	1.41	0.02	1.11	0.14	1.27	
		10	0.54	0.01	0.65	0.21	0.83	
		20	0.51	0.01	0.64	0.06	0.81	
	8	1	2.10	0.03	1.60	0.17	1.31	
		10	0.82	0.03	0.68	0.03	1.22	
		20	0.79	0.02	0.65	0.05	1.20	
	16	1	3.55	0.10	2.14	0.17	1.66	
		10	2.22	0.22	0.79	0.03	2.80	
		20	1.97	0.08	0.85	0.14	2.32	
	4096	2	1	1.98	0.01	1.44	0.07	1.38
			10	1.31	0.01	1.29	0.02	1.02
			20	1.31	0.01	1.12	0.04	1.18
4		1	2.57	0.02	1.83	0.11	1.40	
		10	1.66	0.01	1.57	0.22	1.06	
		20	1.73	0.02	1.56	0.28	1.11	
8		1	3.23	0.06	2.60	0.30	1.24	
		10	2.00	0.02	1.58	0.14	1.26	
		20	1.98	0.02	1.68	0.18	1.17	
16		1	4.82	0.07	3.49	0.33	1.38	
		10	3.46	0.04	1.89	0.07	1.84	
		20	5.03	0.07	2.15	0.08	2.34	

Table 1

The N-Body problem: execution time in *s*.

		ALua		PVM		ALua/PVM		
		time (s)	deviation	time (s)	deviation			
Pixel	Dynamic	4	143.76	0.45	13.27	0.55	10.83	
		8	150.39	0.50	13.31	0.18	11.30	*
		16	148.27	0.31	13.88	0.19	10.68	
	Static	4	81.95	0.00	9.94	1.77	8.24	
		8	41.08	0.02	4.34	0.03	9.47	
		16	20.75	0.03	4.69	0.05	4.42	
Scanline	Dynamic	4	1.08	0.05	0.19	0.00	5.68	
		8	1.17	0.04	0.26	0.03	4.50	
		16	1.23	0.04	0.44	0.04	2.80	
	Static	4	0.15	0.02	0.18	0.02	0.83	*
		8	0.17	0.03	0.20	0.03	0.85	
		16	0.39	0.05	0.29	0.02	1.34	

Table 2

The Distributed Volumetric Visualization problem: execution time in *s*.

As a whole, the experimental results lead us to believe that ALua is a viable tool for development and prototyping. The use of rapid prototyping as a technique for program development is becoming more and more widespread [28], and can be expected to extend parallel programming as well. Also, the gain in flexibility achieved with ALua can in some cases compensate the performance losses. It is worth noting that just turning off the optimization option in a C compiler can result in a slowdown of four or five times in the execution of the resulting program. Nevertheless, people frequently turn off optimizations to speed up compile time. With ALua, compile time is zero.

We believe we can also make some adjustments to get better performance from ALua. We have conducted some preliminary experiments in keeping parts of the communication in C, in situations in which large quantities of data have to be transferred and there is no need for the code exchange facility. Besides, our current implementation of reliable communication between daemons is rather straightforward (for instance, the protocol uses a sliding window of size 1). Refinements of this daemon-to daemon communication protocol will probably lead to an improvement in performance.

6. Related Work

Osterhout, in his seminal paper [29], was one of the first authors to advocate the use of a “hard” language integrated with a scripting language (which he named an *embeddable language*). In that paper, he also introduced the idea of using plain code as a message format for communication among processes. However, as far as we know, he did not apply this idea to distributed or parallel algorithms (although he used it for communication among widgets in the Tk GUI toolkit).

Several groups have used Tcl for distributed programming. Tcl-DP is a Tcl extension for distributed programming [30]. However, its goal is to make it easy to program socket-based client-server applications, and not to support higher-level communication models. In another direction, the Agent Tcl project [31] at Dartmouth College uses Tcl as a basis for an agent system. An agent migrating from one machine to another could in some ways be compared to ALua’s capability of sending code in messages; however, agents arriving at a new machine execute in a new environment, whereas in ALua, when a process receives a message it executes this message in its own environment, with access to existing variables

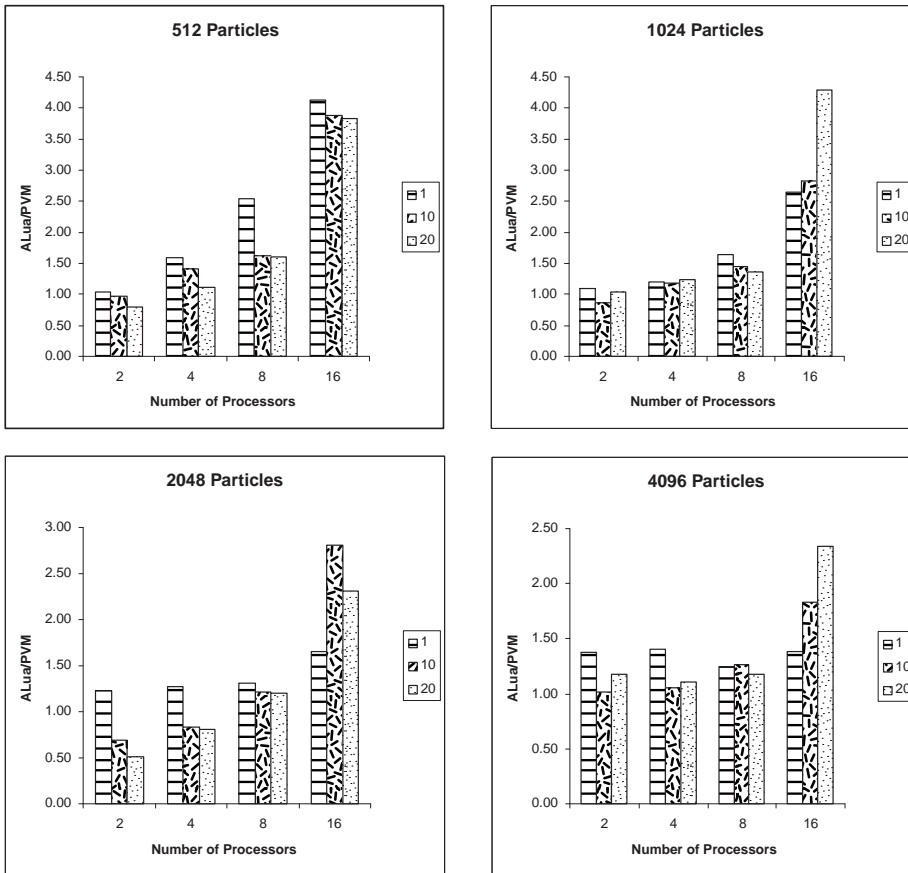


Figure 10. ALua/PVM ratio for the N-Body simulation

and functions. Again, since agents can exchange messages, ALua’s communication model could be emulated; however, that is not the goal of the work. The TACOMA project [32] focuses on operating system support for agents written in a variety of programming languages, including Tcl.

Another drawback of systems using Tcl is performance. “The Great Computer Language Shootout” [33], a comprehensive benchmark among dozens of languages, reports Lua performance being typically two to five times faster than the corresponding Tcl programs.

In as much as we propose the use of ALua to create a flexible communication infrastructure to be combined with compiled program parts for greater efficiency, we can view ALua as a *coordination* language. Papadopoulos and Arbab’s survey [4] classifies coordination models into two broad categories: *data-driven* and *control-driven*. The main idea in this classification is that in data-driven coordination, the state of the computation at any time is defined both by the values of the data being exchanged and by the actual configuration of the coordinated components, while in control-driven coordination the actual values of the data being manipulated are not involved in defining the state of the computation. Supposedly, control-driven models would allow the separation between programming and

configuration concerns to be clearer.

According to the above definition, ALua would be classified as a data-driven coordination language, since the information received in messages can contain code and therefore may redefine component parts and interaction patterns. However, in contrast to typical examples of the data-driven paradigm, such as Linda [5], the ALua model is not based on adding coordination primitives to an existing language: it proposes the use of a full-fledged programming language as a glue between distributed, compiled components. This allows the programmer to completely separate basic component programming from configuration, if he so desires. However, because Lua is a complete programming language and because it has a very flexible interface with C, different levels of integration between the gluing code and the “hard computation” can be chosen by the programmer. We believe this choice is an important facility for the programmer. As discussed in [34], separation of coordination and computation code can be very difficult to maintain when dynamic coordination facilities are required. The same issue is illustrated by the mechanisms for dynamic instantiation in Darwin [6]. Although Darwin’s configuration language, Regis, includes an elegant dynamic instantiation mechanism for recursive structures, the dynamic instantiation of arbitrary modules must be programmed directly in the computation code.

Maybe the ALua model can best be compared to the *MESSENGERS* paradigm introduced by Bic [35]. This paradigm is based on *Messenger* objects, which can navigate among network nodes, performing navigation and computation actions expressed using C. Messengers are compiled to intermediate code that can be dynamically moved across machines. The *messenger* library includes functions for invoking separately, precompiled C functions, allowing it to integrate independently developed code.

The *MESSENGERS* paradigm is described in the framework of a discussion about message passing algorithms. [35] uses the expression ‘*autonomous objects*’ to denote paradigms where messages have been elevated from being simple carriers of data to a higher form, such that some behavioral information can be carried by each message and interpreted by the receiver. The paper further classifies such models along two axes: navigational autonomy and dynamic composition. Navigational autonomy refers to the degree to which a message can include decisions about its own destiny (other than its immediate recipients). ALua allows the programmer to build systems where messages contain the whole behavior of a system, and the nodes are mere message interpreters. (The self-replicating code in section 4.1 follows this approach.) Dynamic composition represents the extent to which autonomous objects can activate independent programs and carry new functional behavior to different nodes. ALua (or, more exactly, Lua) can invoke other processes, can load and call functions written in Lua, and can call functions written in C previously linked to the interpreter. Moreover, Lua also has a non-standard library that allows it to load pre-compiled C libraries dynamically [36].

Another language for connecting distributed applications is Glish [37]. Although not typically cited in this context, Glish can easily be viewed as a coordination language. Like ALua, Glish adopts an event-oriented approach, and it is bilingual. However, unlike ALua, where each agent can be bilingual (with a core written in C and a communication layer written in Lua), in Glish there is one single *master* agent written in Glish, that coordinates the work of slaves written entirely in C. As the slaves are coded in a compiled language, they can handle only a fixed set of messages, with pre-defined content types.

7. Final Remarks

Interpreted languages have been gaining importance over the last years, as programmers realize the virtues of these languages, such as flexibility and support for rapid prototyping. However, such languages are seldom used in parallel applications. As parallel applications increasingly use heterogeneous large-area networks, they also need more flexibility and adaptability. In this paper, we argued that interpreted languages can provide such flexibility to parallel applications, with an acceptable performance penalty.

In section 4 we evaluated the flexibility that ALua can bring to parallel programming. Besides the ease with which different proposed algorithms could be implemented, one important result of that section was to show the support that ALua provides for testing and prototyping. In the implementation of the Tuple Multiplication programming paradigm, we were able to build one single framework based on a chosen communication model, allowing for experimenting different implementations, with no need for testing and debugging the basic communication pattern (parallel and sequential) for each application. In discussing the A-Team application, we showed how ALua can be useful for long-running applications. The facility of an interactive console that the programmer can use to monitor and control an application dynamically is an important tool for this scenario. Because of its dynamic features, ALua also fits well in the context of distributed multimedia applications [38].

ALua's dual programming model, which allows the communication (and coordination) code to be written in ALua and the computation code to be written in a compiled language, allows us to have this gain in flexibility without paying too high a performance price. The results we described in section 5, although preliminary, are stimulating, as they indicate that the costs of using ALua for parallel applications can be relatively low. These costs are quite consistent with the costs of using an interpreted language in sequential applications [33]. It is important to remember that, as pointed out in [39], when measuring the performance of a parallel application or programming tool, we need to consider different metrics. The ease of construction of an application and the time a programmer spends to implement it are frequently more important than the final execution time.

Some areas of distributed programming, such as WWW services, are already trading efficiency for flexibility through the use of languages such as Perl and Java. We believe that the facilities for testing and prototyping in ALua make it an interesting development environment, even if in final versions the program is translated to a compiled language.

One issue that readily comes to mind when discussing a system that allows code to be sent across the network and executed upon receipt is that of security. Here, as often is the case, flexibility is at odds with safety. The ALua system can be customized to be more secure using facilities native to Lua. Functions considered potentially harmful (for instance, all output -generating functions) can be redefined and replaced for dummies in an initialization script as part of ALua's configuration.

Many distributed programming environments are tailored for a specific interaction paradigm, such as client-server or peer-to-peer. ALua does not impose a specific programming pattern. This is one of the sources of its flexibility, but on the other hand may be considered a source of programmer confusion. However, an ALua program will be hard to follow only in cases when the needs of the program are complex; moreover,

common interaction patterns, such as RPC for client-server interactions, can easily be encapsulated and provided in library functions.

Event-driven systems have been gaining importance over the last years. It is now common for applications to be written in an interface-driven style, where the control flow is directed by user interactions. On the other hand, the use of distribution is another important trend in application development. Systems such as ALua bring these two trends together, allowing control flow to be directed not only by user interactions but also by network events.

Versions of ALua for SunOS and for Linux can be downloaded from

<http://www.tecgraf.puc-rio.br/~ururahy/alua/>

REFERENCES

1. J. Ousterhout, Scripting: Higher level programming for the 21st century, *IEEE Computer* 31 (3).
2. B. Gates, VBA and COM, *Byte* 23 (3) (1998) 70–72.
3. D. Cowan, R. Ierusalimschy, T. Stepien, Programming environments for end-users, in: 12th World Computer Congress, Vol. 3, IFIP, Madrid, 1992, pp. 54–60.
4. G. Papadopoulos, F. Arbab, Coordination models and languages, in: *Advances in Computers*, Vol. 46, Academic Press, 1998, pp. 329–400.
5. N. Carriero, D. Gelernter, Linda in context, *Communications of the ACM* 32 (4) (1989) 444–458.
6. J. Magee, N. Dulay, J. Kramer, Structured parallel and distributed programs, *IEEE Software Engineering Journal* (1996) 73–82.
7. I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: Enabling scalable virtual organizations, *Intl. J. Supercomputer Applications* To be published.
8. I. Foster, C. Kesselman, Globus: A metacomputing infrastructure toolkit, *The International Journal of Supercomputer Applications and High Performance Computing* 11 (2) (1997) 115–128.
URL citeseer.nj.nec.com/foster96globu.html
9. G. Allen, T. Dramlitsch, I. Foster, T. Goodale, N. Karonis, M. Ripeanu, E. Seidel, B. Toonen, Cactus-G toolkit: Supporting efficient execution in heterogeneous distributed computing environments, in: *Proceedings of 4th Globus Retreat*, Pittsburgh, PA, 2000.
10. R. Ierusalimschy, L. H. Figueiredo, W. Celes, Lua—an extensible extension language, *Software: Practice and Experience* 26 (6) (1996) 635–652.
11. R. Cerqueira, N. Rodriguez, R. Ierusalimschy, An experiment with event-driven distributed programming (in portuguese), in: *PANEL95 — XXI Conferência Latino Americana de Informática*, SBC, Canela, Brazil, 1995, pp. 225–236.
12. N. Rodriguez, C. Ururahy, R. Ierusalimschy, R. Cerqueira, The use of interpreted languages for implementing parallel algorithms on distributed systems, in: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), *Euro-Par’96 Parallel Processing — Second International Euro-Par Conference*, Springer-Verlag, Lyon, France, 1996, pp. 597–600, Volume I, (LNCS 1123).
13. C. Ururahy, N. Rodriguez, ALua: An event-driven communication mechanism for

- parallel and distributed programming, in: Proceedings of ISCA 12th International Conference on Parallel and Distributed Computing Systems (PDCS-99), Fort Lauderdale, USA, 1999, pp. 108–113.
14. V. Barbosa, An Introduction to Distributed Algorithms, MIT Press, 1996.
 15. L. Bic, M. Fukuda, M. Dillencourt, Distributed computing using autonomous objects, IEEE Computer 29 (8) (1996) 55–61.
 16. V. Sunderman, PVM: A framework for parallel distributed computing, Concurrency: Practice and Experience 2 (4) (1990) 315–339.
 17. C. Kaplan, The search for self-documenting code, <http://www.cs.washington.edu/homes/csk/paper/> (1994).
 18. P. B. Hansen, Studies in Computational Science: Parallel Programming Paradigms, Prentice-Hall, 1995.
 19. T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
 20. P. Souza, Asynchronous organizations for multi-algorithm problems, Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA (1993).
 21. H. Longo, Aplicação de *A-Teams* ao problema de recobrimento, Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP (1995).
 22. E. Cardozo, S. Sichman, DPSK+P User's Manual – C++ Interface, version 1.0, FEE/UNICAMP (Oct. 1992).
 23. J. Barnes, P. Hut, A hierarchical $O(n \log n)$ force calculation algorithm, Nature 324 (1986) 446–449.
 24. M. Franklin, V. Govindan, The N-Body Problem: Distributed system load balancing and performance evaluation, in: Proceedings of the 6th International Conference on Parallel and Distributed Computing Systems, PDCS, Louisville, KY, 1993.
 25. R. B. Seixas, Optimization techniques for volumetric visualization (in portuguese), Ph.D. thesis, Dep. Informática, PUC-Rio, Rio de Janeiro, Brazil (1997).
 26. M. Levoy., Display of surface from volume data, IEEE Computer Graphics and Applications 8 (3) (1988) 29–37.
 27. M. Levoy., Efficient ray tracing of volume data, ACM Transaction on Graphics 9 (1988) 29–37.
 28. F. P. Brooks, The Mythical Man-Month: essays on software engineering, Addison Wesley, anniversary edition, 1995.
 29. J. Ousterhout, Tcl: An embeddable command language, in: Proceedings of the USENIX Winter 1990 Technical Conference, USENIX Association, Berkeley, CA, 1990, pp. 133–146.
 30. B. Smith, L. Rowe, S. Yen, Tcl distributed programming, in: 1st Tcl/Tk Workshop, 1993, pp. 50–51.
 31. R. Gray, Agent Tcl: A transportable agent system, in: Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland, 1995. URL <http://agent.cs.dartmouth.edu/papers/gray:agenttcl.ps.Z>
 32. D. Johansen, R. van Renesse, F. Schneider, Operating System Support for Mobile

- Agents, in: Proceedings of the Fifth Workshop Hot Topics in Operating Systems (HotOS), Washington, USA, 1995, pp. 42–45.
33. D. Bagley, The great computer language shootout, <http://www.bagley.org/~doug/shootout/> (2001).
 34. M. Schumacher, F. Chantemargue, B. Hirsbrunner, The STL++ coordination language: A base for implementing distributed multi-agent applications, in: Proceedings of the Third International Conference on Coordination Models and Languages, Springer-Verlag, 1999, pp. 399–414.
 35. L. Bic, M. Fukuda, M. Dillencourt, Distributed computing using autonomous objects, IEEE COMPUTER 29 (8) (1996) 55–61.
 36. R. Borges, Dynamic library loading facilities for the Lua language, <http://www.tecgraf.puc-rio.br/~rborges/loadlib/> (1998).
 37. V. Paxson, C. Saltmarsh, Glish: a user-level software bus for loosely-coupled distributed systems, in: 1993 Winter USENIX Technical Conference, 1993, pp. 141–156.
 38. L. Leite, R. Alves, G. Lemos, T. Batista, DynaVideo – a dynamic video distribution service, in: 6th Eurographics Workshop on Multimedia (EG Multimedia 2001), Springer-Wien, Manchester, UK, 2001, pp. 95–106.
 39. I. Foster, Designing and Building Parallel Programs, Addison Wesley, 1995.