ALua: An Event-Driven Communication Mechanism for Parallel and Distributed Programming

Cristina Ururahy
e-mail: ururahy@inf.puc-rio.br

Noemi Rodriguez
e-mail: noemi@inf.puc-rio.br

Roberto Ierusalimschy
e-mail: roberto@inf.puc-rio.br

PUC-RioInf.MCC30/00 August, 2000

Abstract: This paper presents ALua, an event-driven communication mechanism based on the interpreted language Lua. This mechanism takes advantage of the interpreted nature of Lua to define messages as chunks of code which are executed by the receiver. The event-driven distributed programming model is discussed and demonstrated with a set of examples. To illustrate the flexibility of the mechanism, the paper describes the implementation of an asynchronous agents team and a parallel programming paradigm. The applicability of ALua in real-sized parallel applications is studied by comparing the implementation of two parallel applications written in ALua with their compiled counterparts.

Keywords: distributed systems, parallel applications, interpreted languages, event-driven communication.

Resumo: Este trabalho apresenta ALua, um mecanismo de comunicação dirigida a eventos, baseado na linguagem interpretada Lua. Esse mecanismo tira proveito da natureza interpretada de Lua para definir mensagens como blocos de código, os quais são executados pelo processo receptor. O modelo de programação distribuída dirigida a eventos é discutido e demonstrado através de um conjunto de exemplos. Para ilustrar a flexibilidade do mecanismo, o artigo descreve a implementação de um time de agentes assíncronos e de um paradigma de programação paralela. A aplicabilidade de ALua em aplicações paralelas de tamanho real é estudada através da comparação da implementação de duas aplicações paralelas, desenvolvidas tanto em ALua quanto em uma linguagem compilada.

Palavras-chave: sistemas distribuídos, aplicações paralelas, linguagens interpretadas, comunicação dirigida a eventos.

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. Interpreted languages have also been gaining popularity in the *extension language* paradigm; in this context, applications 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 C++. The configuration part, usually written in an interpreted language, connects these classes and objects to give the final shape to the application [CIS92, Ous98]. Such configuration languages are also called *extension* languages.

This work explores these ideas in the context of distributed and parallel programming. It presents ALua, a communication mechanism based on Lua [IFC96], an extension language developed at PUC-Rio. This mechanism takes advantage of the interpreted nature of Lua to define messages as chunks of code which are executed by the receiver. Furthermore, ALua uses an event-driven programming model; the arrival of a message is an event which can be handled in the same way as a keystroke or a mouse selection. Section 2 presents ALua and section 3 contains a set of examples which illustrate its flexibility and some typical programming patterns.

The ALua mechanism was originally proposed in a simplified form in [CRI95]. In that work, several classical distributed algorithms, such as probe/echo, heart beat and filtering, were implemented; the goal was to evaluate the event-driven approach by implementing in ALua distributed applications with previous, well known, implementations. With the goal of investigating not only ALua's expressive power but also what kind of flexibility could be obtained by the combination of event orientation with an interpreted language, one specific class of algorithms, for termination detection, was chosen for further study [RUIC96].

The next step was to evaluate the use of ALua in more real-sized applications. With this goal, two parallel applications previously developed with traditional compiled languages were re-implemented in ALua. The idea was both to analyze the ease of development with ALua and to estimate its cost in terms of execution time. Since we were using an interpreted language, a significant loss of performance was foreseen. However, using ALua as a configuration language, as described above, and keeping the main processing kernel in a compiled language allowed us to obtain unexpectedly good results. Section 4 describes these applications and the observed results. Finally, Section 5 discusses some related works and Section 6 presents final remarks.

2 ALua Mechanism

The ALua distributed programming mechanism is an event-driven extension of the interpreted language Lua [IFC96]. ALua, originally proposed in a simplified form in [CRI95], consists of a library of communication features.

In ALua, a program is composed of processes called *agents*, running on one or more physical machines, which communicate using an asynchronous *send* operation. Each agent contains a Lua interpreter, besides an event loop, which manages network and user interface events. The structure of an agent can be seen in Figure 1. Through the user interface, it is possible to provide chunks of Lua code, which will be interpreted by the agent. These chunks may contain all kinds of Lua commands, such as the execution of a file, or even the transmission of messages to other agents.

The arrival of a message is treated as an event, i.e., there is no operation available for the user to receive a message explicitly. As for other events, the programmer may register any function

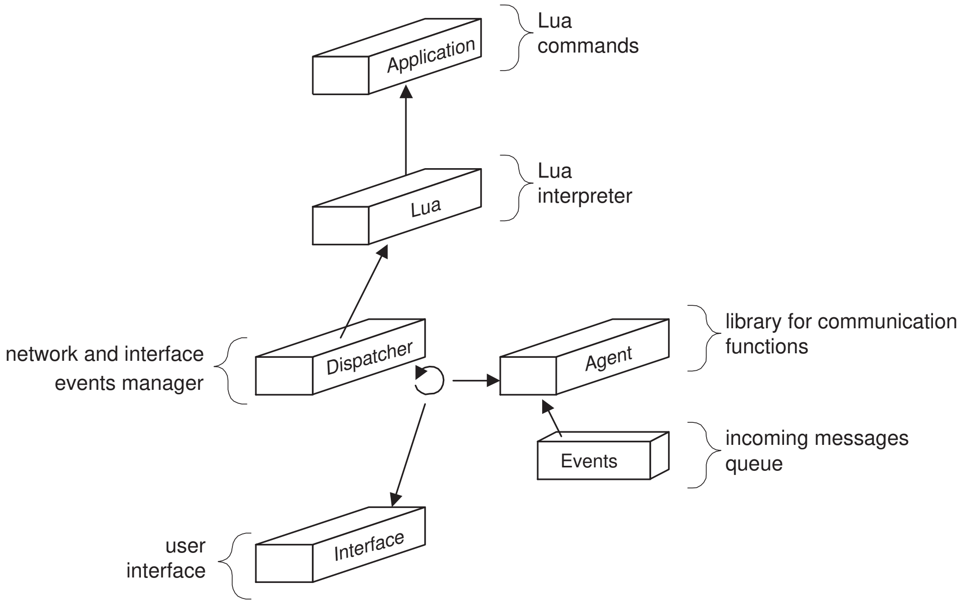


Figure 1: The Structure of an Agent.

as a handler for the arrival of a message. However, a default handler is defined, which looks at messages as if they were chunks of Lua code, and executes them. So, by default, received messages are executed by the interpreter of the agent which receives them. This results in an event-driven programming model compatible with the characteristics of interpreted languages, since the possible consequences of receiving a message can only be determined after it is received.

An important characteristic of ALua is that each received message is treated as an atomic chunk of code. Since it adopts the event-driven paradigm, each event is handled to completion before a new one is examined; this means there is no concurrency in each ALua agent. In our experience this has not proved to be a hindrance. As discussed in the next section, the use of the event-driven paradigm, as of any other programming paradigm, implies in a program structure different from the one used in a conventional programming language. Event handlers, including the handler of a received message, must typically be small, non-blocking chunks of code. If larger actions are required, an agent can always resort to sending a message to itself, as a means of breaking up the code to be executed in non-atomic parts and allowing other messages to be received. On the other hand, as pointed out in [Bic95], the lack of concurrency greatly simplifies many aspects of distributed programming, since there is no need for synchronization inside one agent.

In order to improve performance, an ALua daemon process is automatically spawned in every host where an ALua agent is running (Figure 2). Actually, each agent's communication is restricted to its daemon. This agent-daemon communication uses the Unix Domain Sockets [Ste90] which can be twice as fast as a TCP socket. The Unix domain protocol performs client-server communication on a sigle host, using stream sockets (similar to TCP).

Communication between different hosts is made daemon to daemon and uses the transport protocol UDP. Besides, this daemon to daemon communication follows an ACK-protocol to garantee that messages will not be lost. For instance, if agent A, in host X, wants to send a message to agent B, in host Y, this message will go from A to daemon-X (via Unix Domain), from daemon-X to daemon-Y (via UDP + ACK protocol) and finally from daemon-Y to B (via Unix Domain). If agent B were in host X, than the path would be from A to daemon-X and

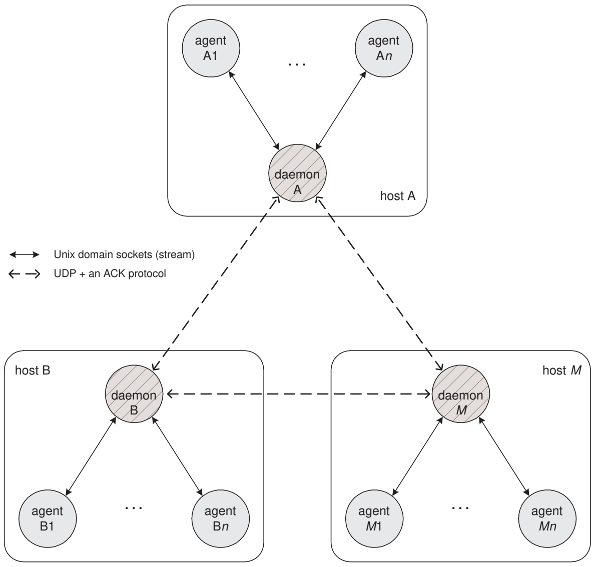


Figure 2: ALua Communication Model.

from daemon-X to B (both via Unix Domain).

Among the functions provided by the ALua library, the following ones are the most used:

- *alua.spawn*
  Spawns n agents. When the spawned agents are ready to receive messages, function *alua.spawn_completed* is called at the local agent. Although the latter function has a default definition, it is usually redefined in the application program.

- *alua.send*
  Sends Lua code to the specified agent.

- *alua.mcast*
  Sends Lua code to one or more specified agents.

- *alua.exit*
  Terminates an agent's execution.

- *alua.exit_all*
  Terminates execution of its "children" (all previously spawned agents). The agent where this command is executed still running until *alua.exit()* is called in its environment or until its "father" calls *alua.exit_all*.

3 Programming Examples

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

The first, very simple, example, shown in Figure 3, sets the value of variables `n` and `y` in agent B's environment.

Notice that any type of value can be sent to an agent[1]. In this example, an agent defines a table named `x`, with 4 elements, where each element is its index's square. Tables are Lua's

[1] Except the ones which describe `userdata` types, used as references to C objects, or functions.

```
n = 50
x = { 1, 4, 9, 16}    -- table

-- Send variables n and x to agent B.
-- In B, x will be named y.

msg = format("n=%d; y=%s", n, alua.tostring(x))
alua.send("B", msg)
```

Figure 3: Setting variables in a remote agent.

basic data structuring constructor. Tables are in fact dynamic associative arrays that directly implement a multitude of data types, like ordinary arrays, records, sets and bags. Table `x` will be sent to agent B, but it will be named `y` in B.

In the next example, shown in Figure 4, an agent sends the definition of a function `addtb` to agent B. This function returns the sum of the elements of a table. The symbols '[[' and ']]' are one kind of string delimiters used in Lua. Although the symbols `"` and `'` can also be used for delimiting strings, the symbols presented in this example allow not only strings that span several lines, but also nested delimiters.

In this example, the sender may not have function `addtb` defined in its own environment; if this is the case, even after the message is sent, it will still be true. On the other hand, B may or may not have a previously defined function `addtb` in its environment. Since functions are first-class values in Lua, if function `addtb` is already defined in B when the message arrives, `addtb` is then redefined.

```
msg = [[ function addtb( t)          -- The symbols "[[" and "]]" are
            local i=1                -- string delimiters.
            local tot=0

            while t[i] do
               tot = tot + t[i]
               i = i + 1
            end
            return tot
         end
      ]]

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

Figure 4: Defining function `addtb` in a remote agent.

One very common programming technique in ALua is to include an answer in the code to be executed at the destination. This creates the possibility of RPC-like calls, except the originator is not blocked while the answer is pending. In the example in Figure 5, this technique is used to determine the sum of the elements in table `y` in agents B and C.

In this example, if the elements of table `y` in agent B are as shown in figure 3, B will send back a message to A, with the command `print(30)`. The code `"print(" ..addtb(y)..  ")"` causes the value of `addtb(y)` to be evaluated in B's environment and then concatenated to the

```
msg = [[ msg = "print(" ..addtb(y).. ")"  -- The operator '..' is a string concatenator.
         alua.send("A", msg)
      ]]

alua.mcast({"B", "C"}, msg)               -- Sends the message to both
                                          -- agents in the table: B and C.
```

Figure 5: Sending a message with request for an answer.

other strings (Concatenation automatically forces conversion of numerical values to strings.).

The next example, shown in Figure 6, illustrates another common programming technique in ALua, which consists of structuring an agent as a state machine. The idea is that A must send a message to agents B and C and terminate execution when it is sure that they have received the message.

In a conventional message-passing system, this would be done roughly as shown in Figure 6a. However, in ALua this would halt the application, since it would be blocked in a loop, waiting for the variable `msg_received` to be incremented and would not succeed in treating the messages that actually increment this variable.

In an event-driven mechanism, such as ALua, this example could be written as shown in Figure 6b. After agent A sends the messages it sleeps until the next event arrives. Agents B and C will receive the message:

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

In this way, A will execute function `Answer` when each of the messages sent by B and C is received. When the first message from either B or C arrives, it triggers function `Answer`, causing A to go to a "received one message" state. When the second message arrives and triggers `Answer` again, A will go to a "received all messages" state.

In a classical *send/receive* model, agent A could use the `send` primitive to send the message to both agents B and C and then block explicitly on two receive statements until the answering messages from B and C arrived. Applications in a *send/receive* model can be easily translated to an event-driven model as state machines.

3.1 Self Replicating Message

The next example deals with broadcasting a message in a logical pipeline configuration, where a variable called `nexthost` indicates, at each agent, the name of the next agent in the pipeline. The message which is to be broadcasted is the string

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

which, when received, will cause each agent to send the value of variable `n` back to A. Suppose that each agent knows nothing about the need for transmitting a message to the next agent, and that we want the message itself to contain the necessary code for its retransmission.

Figure 7 shows a partial solution, where the message is retransmitted only once (from B to C).

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 the original sender knew the exact number of agents in the pipeline.

```
-- Agent A

...

msg_received = 0

msg=[[ msg = "msg_received=msg_received + 1"
       alua.send("A", msg)
    ]]


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

while msg_received ~= 2 do
end

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

(6a) Halting application

```
-- Agent A

function Answer()
   msg_received = msg_received + 1
   if msg_received = 2 then
      print("The End.")
      alua.exit()
   end
end
...

msg_received = 0

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

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

(6b) Correct application

Figure 6: State Machine Example.

To avoid these problems, we can use a self-replicating message, as shown in Figure 8. We leave the understanding of this example as an exercise to the reader.

3.2 A Complete Example

The following example is an implementation of a parallel programming paradigm for the tuple multiplication problem [BH95]. The sole purpose of this example is to present the structure of a complete ALua program; it is not intended as a good solution for the specific problems discussed.

This example is based in [BH95], where five parallel programming paradigms are discussed. Three of these paradigms have been implemented in ALua, each of them with two sample applications: Divide and Conquer, with Fast Fourier Transform and Quick Sort as applications, Monte Carlo, with Simulated Annealing and Primality Test, and Tuple Multiplication, with the Matrix Multiplication and the All-Pairs Shortest Paths as special cases, which will be discussed in this section.

In a previous work, we implemented a sequential and a parallel version for each of the paradigms. Each version is an application-independent skeleton containing invocations of application-dependent functions. The same set of application-dependent functions can be used in the sequential and in the parallel version. This architeture allows the programmer to match different skeletons to different applications dynamically: he 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.

For sequential algorithms it is well-known that matrix multiplication is similar to the problem of finding the *shortest paths* between every pair of nodes in a directed graph [CLR90].

In tuple multiplication, the product of two n-tuples a and b is an nxn matrix c. The matrix elements are obtained by applying the same function f to every ordered pair consisting of an element of a and an element of b, that is $c_{ij} = f(a_i, b_j)$.

In the case of matrix multiplication, the tuple elements are rows and columns, respectively,

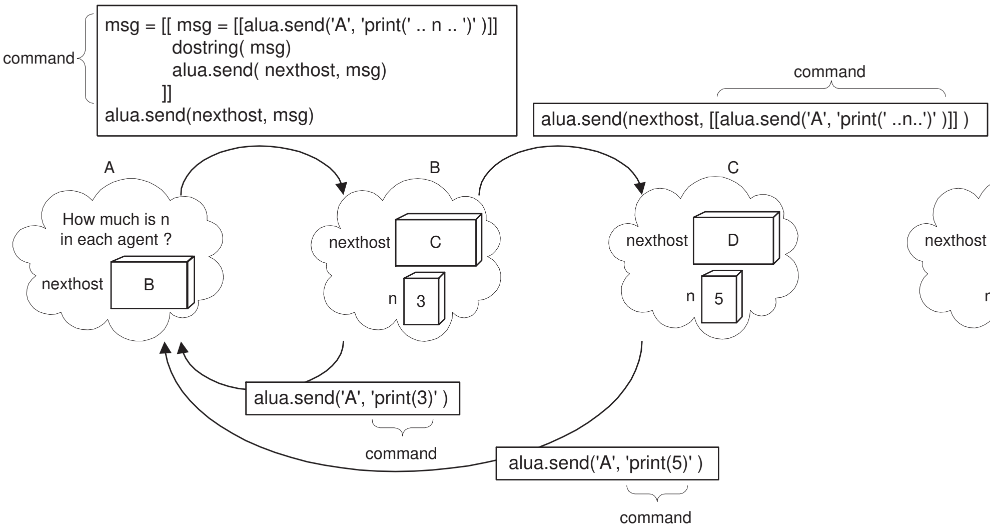


Figure 7: Semi-self-replicating Message

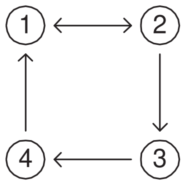
and every function value is the dot product of a row and a column. The input to the shortest paths problem is the adjacency matrix of a graph with n nodes. The output is a distance matrix computed by a sequence of tuple multiplications. In every multiplication, tuple a consists of the n rows of the adjacency matrix, while tuple b consists of the n columns of the distance matrix. The function value $f(a_i, b_j)$ defines the shortest path length found so far between nodes i and j of the graph.

Figure 9 shows the sequential skeleton for tuple multiplication [2]. Function `multiply` will receive two matrices and their dimensions, compute a resulting matrix and display the result. For simplicity, we suppose both matrices are square.

The specific code for the matrix multiplication problem is shown in Figure 10. Function `matrix` creates two matrices and calls function `multiply`, which depends on the chosen paradigm (sequential or parallel). Function `multiply` invokes function `f` (that must be defined in the application specific code), which receives one row from the first matrix and one column from the second one and returns one resulting matrix element.

Figure 11 shows the specific code for the all-pairs shortest paths problem. The all-pairs shortest paths application compute the shortest paths between every pair of nodes in a directed graph, i.e., how many nodes should be traversed to reach one node from another. Function `allpaths` creates an adjacency matrix nxn of a graph with n nodes and calls function `multiply` (paradigm dependent) $n-1$ times in order to calculate the resulting matrix. Function `f` will be invoked by `multiply`.

To illustrate this application, consider the following directed graph with nodes 1, 2, 3 and 4.



[2]The complete codes are listed in the appendix.

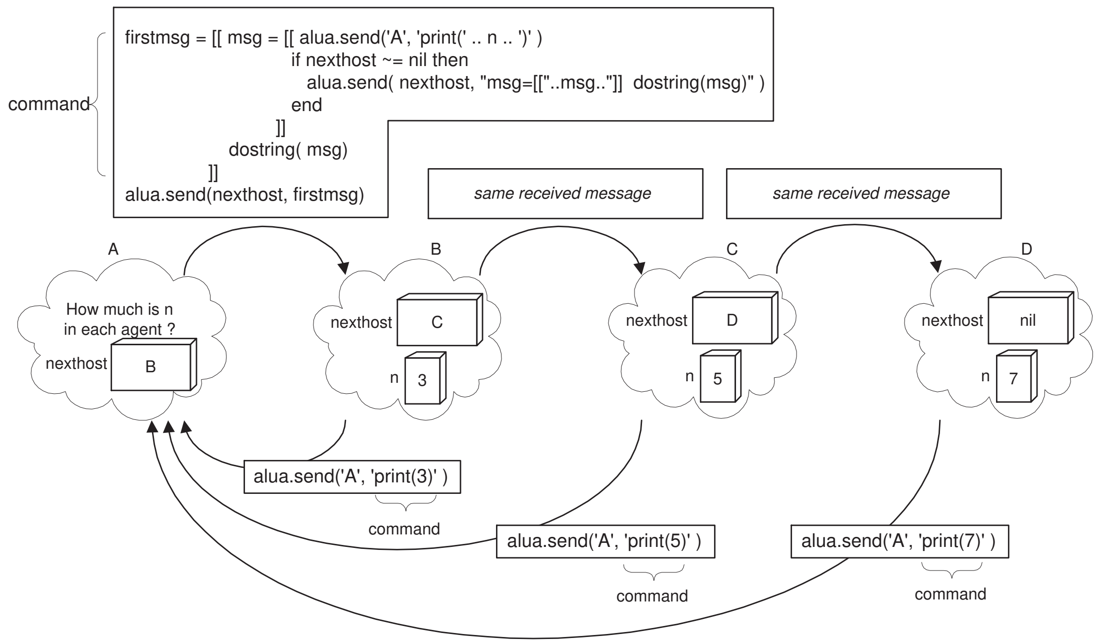


Figure 8: Self-replicating Message

This graph is represented as the matrix a below, where every edge has length 1. Entries (i,j) with value 1 indicate the existence of an edge from node i to node j and with value ∞ indicate there is no edge. Entries (i,i) are set to 0.

from\to	1	2	3	4
1	0	1	∞	∞
2	1	0	1	∞
3	∞	∞	0	1
4	1	∞	∞	0

The goal of this problem is to compute an nxn distance matrix d. A matrix element d_{ij} defines the shotest path from node i to node j. If there is no path d_{ij} is ∞.

Following the shortest path from one node to another, each of the $n-1$ edges may be visited, but no more than once. (Otherwise, the path is not the shortest one.) Consequently, the shortest path, if any, always has fewer than n edges.

A sequence of distance matrices will be computed: d^1, d^2, ..., d^{n-1}. The first matrix defines all shortest paths of 1 (or 0) edges, which is the adjacency matrix ($d = a$). The second matrix defines all shortest paths of 2 (or fewer) edges ($d^2 = d$ x d) and so on. The $n-1^{th}$ matrix defines all shortest paths of $n-1$ (or fewer) edges. Since every shortest path has fewer than n edges, the $n-1^{th}$ matrix is the distance matrix we are looking for. (That is why function `multiply` is called $n-1$ times.)

Note that, in Figure 11, function `f` is quite different from the one in Figure 10. In this case, instead of calculating the dot product of a row and a column, it computes the minimum value of the sums of each pair. The minimum value corresponds to the smallest distance between each pair of nodes (minimum number of edges to be traversed).

The resulting matrix of this example would be:

```
function multiply(a, b, n)
   local i, j, c

   i = 1
   c = {}
   while i <= n do
      j = 1
      c[i]={}
      while j <= n do
         c[i][j] = f(a[i], b[j])
         j = j + 1
      end
      i = i + 1
   end
   return c
end
```

Figure 9: Tuple Multiplication paradigm: Sequential skeleton.

```
function matrix(dim)
   local a, b, c

   n = dim
   < Here goes the code to create >
   < matrices a and b.            >
   c = multiply(a, transpose(b), n)
   if c then
      show(c)
   end
end
```

```
function f( ai, bj)
   local cij = 0
   local k = 1
   while k <= n do
      cij = cij + ai[k]*bj[k]
      k = k + 1
   end
   return cij
end
```

Figure 10: Matrix multiplication - application specific code.

from\to	1	2	3	4
1	0	1	2	3
2	1	0	1	2
3	2	3	0	1
4	1	2	3	0

To illustrate the meaning of this matrix, let us consider the entry (3,2), its value indicates that it is necessary to traverse at least 3 edges to reach node 2 from node 3. Looking at the graph, it is easy to see that these edges are: 3–4, 4–1 and 1–2.

The parallel version of this paradigm is shown in Figure 12. 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 whole second matrix.

Each of the agents executes function `node`, which repeatedly computes rows of the final matrix. Each row is sent to the master as soon as it has been computed.

```
function allpaths(dim)
   local a, b, d

   n = dim

   < Here goes the code to create graph a.>

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

   if d then
      show_graph(d)
   end
end
```

```
function f( ai, bj)
   local cij = infinity
   local k = 1
   while k <= n do
      cij = min(cij, sum(ai[k], bj[k]))
      k = k + 1
   end
   return cij
end

function show(a)   -- used by parallel
   if m < n then   -- version only
      d = multiply(a, transpose(a), n)
      m = 2 * m
   else
      show_graph(a)
   end
end
```

Figure 11: All-pairs shortest paths - application specific code.

Since the communication in ALua is asynchronous, the parallel version makes use of the callback function `show` (see line 47 in Figure 12) to indicate that the result is ready.

For the matrix multiplication example, function `show` simply exhibits the resulting matrix. But, for the all-pairs shortest paths, it is necessary to make successive multiplications before the final result is obtained. Because of this, function `show` calls function `multiply` again in case the final result is not reached, and calls function `show_graph` to exhibit the resulting matrix (graph) when the final result is ready.

3.3 An Asynchronous Agents Team

Another example of ALua's use is an implementation of an asynchronous agents team (A-Team) [Sou93], which consists of a set of algorithms to solve the Set Covering Problem [Lon95]. This example takes a great advantage of the flexibility provided by ALua. Since this example is quite complex, its details won't be discussed here.

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

- autonomous agents
  An autonomous agent makes it own choice when selecting its input data and resource allocation policy. Since autonomous agents are completely independent of each other, new agents can be added to the system and others removed from it without any notification to other agents or to some manager.

- asynchronous communication
  Agents can read and write data in shared memories without any kind of synchronization among them. There is no logical dependency to access shared memories, except for data integrity. The agents autonomy together with the asynchronous communication allows agents to work full-time in parallel.

- ciclic data flow
  Agents can retrieve, modify and store data in shared memories continuously. A ciclic data flow allows a continuous agent's modifications flow making the iteration and feedback among the agents possible.

The use of this paradigm [Sou93, Lon95] indicates that the cooperation among the agents tends to reach 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 the optimal one to the problem addressed by the A-Team. 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 to the system.

Originally, the A-Team discussed here was implemented in C, with the communication package DPSK+P [CS92], based in a shared objects architecture. This communication package was replaced by ALua.

The system is composed by two servers, acting as solution repositories, representing the shared memory of the A-Team [Lon95]; two initialization agents, initializing the repositories; and nine worker agents, implementing refining algorithms for the solutions stored in the repositories.

The initialization and worker agents send messages to the solution repositories, that process these messages and may generate results for the agents. One of the repositories stores dual solutions for an instance of the Set Covering Problem, while the other stores primal solutions for the same instance. The typical behaviour of an A-Team agent is: request a solution from the solutions repository, refine this solution, send the new solution back to the repository and start over. On the other hand, a solution repository simply answers agents' requests.

Depending on the input data, this application can keep running for some days before reaching an interesting result. When this is the case, the use of ALua makes it possible to dynamically reconfigure the system, consult the repository, or even redefine functions. For instance, it was possible to redefine the behaviour of the ALua communication functions, on the fly, in order to create a log file of the communication messages, helping to debug the application. In the same way, it was possible to restore the original functions.

```
function multiply(a, b, dim)
   local bstr = alua.tostring(b)

   n = dim
   < nmax is the number of nodes >
   k = 1
   while k <= nmax do
      < 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 in the text).  >

      msg = format([[column=%s row={}]], bstr)

      i = frow
      while (i <= lrow) do
         msg = msg .. format(" row[%d]=%s",i, alua.tostring(a[i]))
         i = i + 1
      end
      msg = msg .. format(" n=%d node(%d, %d)", n, frow, lrow)
      alua.send( Procs[k], msg)
      k = k + 1
   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
      show(master_c)
   end
end

-- This string will be sent to slave agents when they are started.
node_code=[[
   function node( r, s)        -- 1 <= r <= s <= n
      i = r
      while i <= s do
         local j = 1

         c = {}
         while j <= n do
            c[j] = f( row[i], column[j])
            j = j + 1
         end
         msg = format("master_receive(%s,%d)", alua.tostring(c), i)
         alua.send("master", msg)
         i = i + 1
      end
   end
]]
```

Figure 12: Tuple Multiplication paradigm: Parallel skeleton.

4 Case Studies

The goal of the work described in this section was to verify if ALua was robust enough for use in large applications and find out the cost one is paying for using an interpreted language. Two parallel applications previously written in C and PVM [Sun90] were chosen for implementation in ALua: an N-body simulator and a tool for distributed volumetric visualization.

Keeping in line with the idea of Lua as an extension language, the applications were split into two parts. The communication part of the application was written in ALua; the other part, containing the processing kernel, was written in C.

4.1 The N-Body Simulation Problem

The implementation of a solution for the N-Body Simulation Problem was based on the *Barnes-Hut* algorithm [BH86]. More specifically, our implementation was based on [FG93].

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

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).

This application involves a large amount of communication among all the agents in the system, because in each iteration all agents collect information about all others, so as to determine how to redistribute their particles. The N-Body Simulation Problem also deals with a large data structure, that is exchanged among all agents many times during its execution.

Figure 13 shows the complete Lua code for the master agent. Although the body of this code is written in Lua, functions `init_particles`, `read_init_dist` and `send_particles` are completely implemented in C. The master agent starts all the workers it will need in the simulation, reads the initial distribution, sends it to the first worker and starts its execution.

Figure 14 shows the complete Lua code for the workers. In order to achieve a better performance, some of the functions, such as `orb_decompose`, `exchange_tree` and `compute_boundaries` again are written in C, although the main skeleton is written in Lua.

Lua functions can add an additional flexibility to the application because they can be redefined during runtime. For instance, function `begin_worker` (in Figure 14) can be changed to display a message for each iteration. In order to do so, in the master's console one could send a new `begin_worker` definition for each worker:

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

In the interpreter, \\ allows the user to insert a new line without evaluating the previous one.

```
function main()
  dofile(Directory .. infile)     -- read initialization parameters --
  alua.spawn(nworkers)
end

function alua.spawn_completed( hosts)
  worker_tids = hosts

  alua.send_agentid(hosts, hosts) -- We'll use a complete graph of nodes (workers)
                                  -- It means: Everybody knows everybody.

  < Here goes the code to send initialization info, variables and workers' ids to all workers.>

  nbody=init_particles()
                                  -- read initial distribution  --
  read_init_dist( nbody, nparticles, Directory .. init_file)

  < Here goes the code to send specific worker's code to them.>

                                  -- send particles to worker 0 --
  send_particles(nbody, worker_tids[1], 1, "INIT_DATA", "nparticles =recv_particles(nbody)")

  alua.send(worker_tids[1], "worker()") -- start execution on worker 0
end

main()
```

Figure 13: N-Body Problem: Master.

```
nbody = makenbody()

function worker()
  < Init tree structure and variables.           >
  < distribute particles using the ORB algorithm.>
  mytree, nbody, myparticles = orb_decompose(mytree,nbody,nparticles)
  iteration = 0
  begin_worker()
end

function begin_worker()
  if iteration < MAX_ITERATIONS then
     form_tree(mytree, nbody, myparticles) -- construct local tree: ORB  --
     -- exchange information among workers: broadcast algo      --
     if nworkers > 1 then
        exchange_tree(root, iteration)
     else
        resume_worker()
     end
  else
     finish_worker()
  end
end

function resume_worker()
     compute_boundaries(root, dim)
     recompute_boundaries(root, dim)
     compute_forces(root, mytree, TIMESTEP, TOL, SOFT)
     sendbody = emptynbody()                  -- pool particles         --
     myparticles, nbody, sendbody = collect_particles(mytree, nbody, sendbody)

     -- There is a need to synchronize the processes here so that       --
     -- runaway messages are not mixed with exchange ones.              --
     sync_to_runaway()
end

function worker_runaway()
      local newparticles                      -- send runaway particles --
      newparticles, nbody = runaway(root, sendbody, XMAX/100, nbody)
      myparticles = myparticles + newparticles
      cleanup(root, mytree, dim, mytid-1)     -- clean up               --
end

function worker_runaway_received()
      iteration = iteration + 1
      begin_worker()
end
```

Figure 14: N-Body Problem: Workers.

4.2 Distributed Volumetric Visualization

The other application in this experiment was the so called Distributed Volumetric Visualization and was based on [Sei97]. The problem consists of visualizing a set of tri-dimensional data. This application uses the *ray casting* optimization technique [Lev88a, Lev88b].

The structure of the application is shown in Figure 15. A *master* agent divides the image to be computed into a number of regions, or sub-images, each one of them corresponding to a task. These tasks are distributed (dynamically or statically) among N *worker* agents. After all tasks have been completed, a *visualizer* agent exhibits the complete image, placing each sub-image in the correct position.

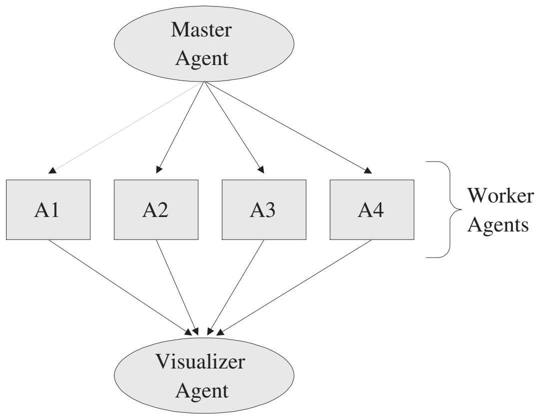


Figure 15: Distributed Volumetric Visualization - structure.

In this application the communication among the agents is restricted to communication between the master and each worker, and between each worker and the visualizer.

4.3 Experimental Results

In order to implement these applications in Lua, some modifications to the original algorithms were necessary, since they were based on a *send/receive* model and ALua deals with an event-driven model. Most of the necessary changes were very simple, limited to change some iterations to a model similar to a state machine (see Figure 6).

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

For the N-Body Simulation Problem, 48 experiments were carried out, each of them with 5 replications. Three factors were taken into consideration in the design of the experiments: the number of particles (512, 1,024, 2,048 and 4,096), 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 512 and 2,048 particles, using 2, 4, 8 and 16 agents, with 1, 10 and 20 iterations. The execution times for the 10 and 20 iterations experiments were divided by ten and twenty respectively, in order to be compared to the 1 iteration ones.

In the worst case (512 particles, 16 processors and 20 iterations) ALua was 4 times slower than PVM while in the best case (2,048 particles, 2 processors and 20 iterations) ALua was 1.22 times faster then PVM. In the average case, ALua was 1.76 times slower.

For the Distributed Volumetric Visualization, 18 experiments were done, each of them replicated 5 times. In this case, the factors taken into consideration were the partition method

N-Body		512				2048			
		2	4	8	16	2	4	8	16
	1	0.43	0.51	0.59	0.84	0.91	1.31	1.67	2.17
PVM	10	0.13	0.19	0.27	0.43	0.60	0.82	0.82	0.83
	20	0.15	0.17	0.23	0.38	0.72	0.72	0.69	0.74
	1	0.49	0.83	1.49	3.34	1.33	1.78	2.46	4.18
ALua	10	0.17	0.29	0.50	1.59	0.63	0.84	1.10	2.16
	20	0.15	0.26	0.44	1.50	0.59	0.82	1.07	2.03

Table 1: The N-Body problem: execution time in s.

(blocks, pixels and scanlines), the number of processors (4, 8 and 16) and the distribution of tasks. The distribution of tasks 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. 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, dynamic and static distribution, using 4, 8 and 16 agents. 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. In the average case, ALua was 4.70 times slower.

	Scanline				Pixels			
	Dynamic		Static		Dynamic		Static	
	ALua	PVM	ALua	PVM	ALua	PVM	ALua	PVM
4	1.08	0.19	0.15	0.18	143.76	13.27	81.95	9.94
8	1.17	0.26	0.17	0.20	150.39	13.31	41.08	4.34
16	1.23	0.44	0.39	0.29	148.27	13.88	20.75	4.69

Table 2: The Distributed Volumetric Visualization problem: execution time in s.

Although no extensive performance analysis was conducted, we believe these results are important, since they show numbers of the same order of magnitude for the PVM and ALua implementations. The efficiency loss in the ALua implementations must be viewed in context: a mechanism such as ALua is adequate for prototyping and exploration of alternatives. With this in mind, a factor of five (in average) in execution time can not be a very high price to pay. 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.

5 Related Work

The non-deterministic behavior of an ALua program, which remains blocked until a message is received, and then executes the code dictated by the received message, is in some ways similar to the behavior exhibited by Ada's `select` statement and SR's `in` statement. Since Ada and SR are statically typed languages, the set of messages that can be received by these statements and the code to be executed upon their receipt is defined at compile-time. The interpreted nature of Lua allows the message to carry the code which is to be executed. One interesting feature of both Ada and SR is that the programmer can state conditions which must hold true for a certain message to be received. This can be partially simulated in Lua by the use of the state machine pattern discussed in section 3; however, since no restrictions are imposed on the transmitted code, it is impossible to achieve the same degree of control. This lack of control is coherent with the dynamic nature of the proposed mechanism; in this line, one could create programming patterns where the desired conditions would be tested by the transmitted code itself.

[Bic95] discusses the *Autonomous Objects* paradigm for message passing, which encompasses systems where, as in ALua, messages have been elevated from being simple carriers of data to a higher form where behavioral information may be carried along with the message for interpretation/execution at the receiving node. The author proposes a classification according to the ability of a system to serve as what he calls a *coordination language*, by what he means the possibility of activating independent programs resident on various nodes throughout the network. It is interesting to discuss ALua's potential in this respect, which is in fact directly inherited from Lua.

ALua agents can be classified in the third category proposed in [Bic95] (the one with more support for coordination), where, in addition to being able to spawn or to communicate with other processes, autonomous objects can also invoke precompiled functions, which are loaded and executed as part of the currently running process. Because of the facilities available for its interfacing with C, a great number of libraries are available for Lua. As two examples of special interest in the context of ALua, we may cite LuaOrb, which allows dynamic access to CORBA servers [ICR98], and LuaMan, which provides support for network management [RLMS98].

6 Final Remarks

In this paper we have described ALua and reported a series of experiments which were conducted for its evaluation.

The works described in sections 3.2 and 3.3 were intended to evaluate the flexibility of the mechanism. Besides the ease with which different proposed algorithms could be implemented, one important result of these works was to show the support the mechanism provides for testing and prototyping. In the implementation of Tuple programming paradigm, a single framework based on a chosen communication model was built, allowing for experimenting different implementations, with no need to test and debug the basic communication pattern (parallel and sequential) for each application.

The results described in section 4, although preliminary, are stimulating, since they indicate that the cost of using a tool such as ALua for parallel applications can be relatively low. It is important to remember that, 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. We believe 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.

The ALua platform has been in use at PUC-Rio for some time now, both in project development and in distributed programming courses. One project currently under development is a distributed network management system. The goal of this project is to experiment with distributed management architectures, which are becoming more and more important as the size of the managed networks grows [WBSS98]. In the system we are building, a master ALua process delegates to several helper processes the task of polling variables selected by the user and maintaining statistics on their values (In this context, the use of the word *agent* is unfortunate, since it clashes with network management *agents*).

Versions of ALua for SunOS and for Linux can be downloaded from `http://mutantes.les.inf.puc-rio.br/~ururahy/alua/`.

References

[BH86] J. Barnes and P. Hut. A hierarchical o(n log n) force calculation algorithm. *Nature*, 324:446–449, 1986.

[BH95] Per Brinch Hansen. *Studies in Computational Science: Parallel Programming Paradigms.* Prentice-Hall, 1995.

[Bic95] L. F. Bic. Distributed computing using autonomous objects. In *Proceedings of the 5th IEEE Computer Science Workshop on Future Trends of Distributed Computing Systems*, 1995.

[CIS92] D. Cowan, R. Ierusalimschy, and T. Stepien. Programming environments for end-users. In *12th World Computer Congress*, volume 3, pages 54–60, Madrid, September 1992. IFIP.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[CRI95] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. An experiment with event-driven distributed programming (in portuguese). In *PANEL95 — XXI Conferência Latino Americana de Informática*, pages 225–236, Canela, Brazil, 1995. SBC.

[CS92] E. Cardozo and S. Sichman. *DPSK+P User's Manual - C++ Interface, version 1.0.* FEE/UNICAMP, October 1992.

[FG93] M. Franklin and V. Govidan. The N-Body Problem: Distributed system load balancing and performance evaluation. In *Proceedings of the 6th International Conference on Parallel and Distributed Computing Systems*, Louisville, KY, October 1993. PDCS.

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

[ICR98] R. Ierusalimschy, R. Cerqueira, and N. Rodriguez. Using reflexivity to interface with CORBA. In *International Conference on Computer Languages 1998*, Chicago, 1998. IEEE.

[IFC96] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Lev88a] M. Levoy. Display of surface from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[Lev88b] M. Levoy. Efficient ray tracing of volume data. *ACM Transaction on Graphics*, 9:29–37, 1988.

[Lon95] Humberto Longo. Aplicação de *A-Teams* ao problema de recobrimento. Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP, 1995.

[Ous98] John Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[RLMS98] N. Rodriguez, M. Lima, A Moura, and M. Stanton. A platform for the development of extensible management applications. In *INET'98*, Geneva, Switzerland, july 1998.

[RUIC96] N. Rodriguez, C. Ururahy, R. Ierusalimschy, and R. Cerqueira. The use of interpreted languages for implementing parallel algorithms on distributed systems. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing — Second International Euro-Par Conference*, pages 597–600, Volume I, Lyon, France, August 1996. Springer-Verlag. (LNCS 1123).

[Sei97] R. B. Seixas. *Optimization Techniques for Volumetric Visualization (in portuguese)*. PhD thesis, Dep. Informática, PUC-Rio, Rio de Janeiro, Brazil, 1997.

[Sou93] Pedro Souza. *Asynchronous Organizations for Multi-Algorithm Problems*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1993.

[Ste90] W. Stevens. *UNIX Network Programming*. Prentice-Hall, 1990.

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

[WBSS98] S. Waldbusser, A. Bierman, B. Stewart, and R. Stuart. Distributed management framework, August 1998. Internet Draft draft-ietf-disman-framework-01.txt.

A Tuple Multiplication paradigm: Sequential skeleton

```
function multiply(a, b, n)
   local i, j, c

   i = 1
   c = {}
   while i <= n do
      j = 1
      c[i]={}
      while j <= n do
         c[i][j] = f(a[i], b[j])
         j = j + 1
      end
      i = i + 1
   end
   return c
end
```

B Matrix multiplication - application specific code

```
function matrix(dim)
   local a, b, c

   n = dim
   a = create_matrix(n)
   b = create_matrix(n)
   print("Matrix A")
   show(a)
   print("Matrix B")
   show(b)
   print("Matrix C = AxB")
   c = multiply(a, transpose(b), n)
   if c then
      show(c)
   end
end

function f( ai, bj)
   local cij = 0
   local k = 1
   while k <= n do
      cij = cij + ai[k]*bj[k]
      k = k + 1
   end
   return cij
end
```

C All-pairs shortest paths - application specific code

```
function allpaths(dim)
   local a, b, d
```

```
   n = dim
   a = create_graph(n)
   m = n
   print("Initial Graph")
   show_graph(a)

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

   print("Distance Matrix")
   if d then
      show_graph(d)
   end
end

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

function show(a)        -- It will be used by the parallel version only
   if m < n then
      d = multiply(a, transpose(a), n)
      m = 2 * m
   else
      show_graph(a)
   end
end
```

D Tuple Multiplication paradigm: Parallel skeleton

```
function multiply(a, b, dim)
   local k, qmin, qmax, msg, nmax
   local bstr = alua.tostring(b)

   n = dim
   if n < nnodes then
      nmax = n
   else
      nmax = nnodes
   end
   qmin = floor(n/nmax)
   qmax = qmin + 1
   long = mod(n, nmax)

```

```
   k = 1
   while k <= nmax do
      local frow, lrow
      local i

      if k <= long then
         frow  = (((k-1)*qmax)+1)
         lrow  = (k*qmax)
      else
         frow  = (((k-1)*qmin)+long+1)
         lrow  = ((k*qmin)+long)
      end

      msg = format([[column=%s row={}]], bstr)

      i = frow
      while (i <= lrow) do
         msg = msg .. format(" row[%d]=%s",i, alua.tostring(a[i]))
         i = i + 1
      end
      msg = msg .. format(" n=%d node(%d, %d)", n, frow, lrow)
      alua.send( Procs[k], msg)
      k = k + 1
   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
      show(master_c)
   end
end

-- This string will be sent to slave agents when they are started.
node_code=[[
   function node( r, s)
      -- 1 <= r <= s <= n
      local i, c

      i = r
      while i <= s do
         local j = 1

         c = {}
         while j <= n do
            c[j] = f( row[i], column[j])
            j = j + 1
         end
         msg = format("master_receive(%s,%d)", alua.tostring(c), i)
         alua.send("master", msg)
         i = i + 1
      end
```

```
  end
]]
```

E N-Body Problem: Master

```
infile = "input"                -- input filename           --
init_file = "initialdist"       -- initial dist filemane    --

function main()
        -- master_tid           -- master task id           --
        -- worker_tids[MAXNODES]   -- worker task ids       --
        -- inp                  -- input file descriptor    --
        -- nbody                -- initial particle list    --

  dofile(Directory .. infile)   -- read parameters          --

  alua.spawn(nworkers)

  master_tid = alua.mytid       -- enrol in LUA             --
end

function alua.spawn_completed( hosts)
  local msg
                                -- send initialization info --
  worker_tids = hosts

  alua.send_agentid(hosts, hosts) -- We'll use a complete graph of nodes (workers)
                                  -- It means: Everybody knows everybody.

  msg =        format("nparticles= %d ",     nparticles)
  msg = msg .. format("nworkers= %d ",       nworkers)
  msg = msg .. format("MAX_ITERATIONS= %d ", MAX_ITERATIONS)
  msg = msg .. format("TIMESTEP= %d ",       TIMESTEP)
  msg = msg .. format("SOFT= %d ",           SOFT)
  msg = msg .. format("TOL= %d ",            TOL)
  msg = msg .. format("MEASURE_TIME= %d ",   MEASURE_TIME)
  msg = msg .. format("HOW_OFTEN= %d ",      HOW_OFTEN)
                                -- send worker task ids also         --
  msg = msg .. format("worker_tids= %s", alua.tostring(worker_tids))

  alua.mcast(worker_tids, msg)  -- multicast initialization message  --

  nbody=init_particles()
                                -- read initial distribution         --
  read_init_dist( nbody, nparticles, Directory .. init_file)

  msg = format("Directory= '%s' ", Directory)
  msg = msg .. "dofile(Directory..'worker.lua')"
  alua.mcast(worker_tids, msg)  -- Get worker code.

                                -- send particles to worker 0        --
  send_particles(nbody, worker_tids[1], 1, "INIT_DATA",
                       "nparticles =recv_particles(nbody)")
```

```
  alua.send(worker_tids[1], "worker()") -- start execution on worker 0
end

function master_end()
  alua.exit_all()               -- leave LUA Agent          --
  exit()
end

workers_finished = 0

main()
```

F N-Body Problem: Workers

```
nbody = makenbody()

function worker()
  local msg

  mytid = alua.mytid                    -- enroll in com
  hostname = worker_tids[alua.mytid]
  master_tid = alua.myparent            -- get master tid

  dim = log2(nworkers)                  -- dimension variable used in
                                        -- tree construction
  mynum = mytid - 1

  root = makeroot()                     -- initialize tree structure
  mytree = root
                                        -- distribute particles using
                                        -- the ORB algorithm
  mytree, nbody, myparticles = orb_decompose(mytree,nbody,nparticles)

  iteration = 0
  begin_worker()
end

function begin_worker()
  if iteration < MAX_ITERATIONS then
     form_tree(mytree, nbody, myparticles) -- construct local tree: ORB
                                           -- exchange information among
                                           -- workers: broadcast algo
     if nworkers > 1 then
        exchange_tree(root, iteration)
     else
        resume_worker()
     end
  else
     finish_worker()
  end
end

function resume_worker()
     compute_boundaries(root, dim)
```

```
     recompute_boundaries(root, dim)

     compute_forces(root, mytree, TIMESTEP, TOL, SOFT)

     sendbody = emptynbody()               -- pool particles
     myparticles, nbody, sendbody = collect_particles(mytree, nbody, sendbody)

     -- There is a need to synchronize the processes here so that       --
     -- runaway messages are not mixed with exchange ones.              --
     sync_to_runaway()
end

function worker_runaway()
      local newparticles
                                           -- send runaway particles
      newparticles, nbody = runaway(root, sendbody, XMAX/100, nbody)
      myparticles = myparticles + newparticles

      cleanup(root, mytree, dim, mytid-1)  -- clean up
end

function worker_runaway_received()
      iteration = iteration + 1
      begin_worker()
end

function finish_worker()
   -- end of simulation. Send finish signal to master
   signal_finish( master_tid)
end

function worker_exit()
   -- end of simulation. Send particle distribution to master
   send_dump_info( nbody, master_tid, mynum)

alua.exit()
end
```