Exploring Lua for Concurrent Programming

Alexandre Skyrme[1], **Noemi Rodriguez**[1,2], **Roberto Ierusalimschy**[1]

[1]Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rua Marquês de São Vicente, 225 – CEP 22.451-900 – Rio de Janeiro – RJ – Brazil

[2]Rede Nacional de Ensino e Pesquisa (RNP)
Rua Lauro Müller, 116 sala 3902 – CEP 22.290-906 – Rio de Janeiro – RJ – Brazil

{askyrme,noemi,roberto}@inf.puc-rio.br

Abstract. *The popularization of multi-core processors and of technologies such as hyper-threading demonstrates a fundamental change in the way processors have been evolving and also increases interest in concurrent programming, particularly as a means to improve software performance. However, concurrent programming is still considered complex, mostly due to difficulties in using the available programming models, which have been subject to recurring criticism. The increased interest in concurrency and the lack of proper models to support it stimulates the development of proposals aimed at providing alternative models for concurrent programming. In this paper, we explore some of Lua's facilities to devise such a model, based on user threads and message passing. We also demonstrate why Lua was particularly well suited for this objective, describe the main characteristics of our model and present a library developed to implement it, along with results of a performance evaluation.*

1. Introduction

Regardless of its growing importance, concurrent programming is still mostly based on dated models. Constructions like semaphores [Dijkstra 1983], conditional critical regions [Hoare 1972], guards [Dijkstra 1975], and monitors [Hansen 1974, Hoare 1973] were all originally designed for operating systems and are admittedly complex for higher-level programming. Moreover, they do not scale well for massive concurrency. This scenario has stimulated the proposal of alternative models for concurrent programming, such as Erlang [Armstrong 1996], Polyphonic C# [Benton et al. 2002], Sequential Object Monitors [Caromel et al. 2004], and the Concurrency and Coordination Runtime [Chrysanthakopoulos and Singh 2005].

Since 2003 the Lua programming language [Ierusalimschy et al. 1996, Ierusalimschy et al. 2006, Ierusalimschy et al. 2007] features coroutines, which enables collaborative multithreading. However, a common criticism for coroutines is that they cannot explore hardware parallelism, such as provided by multi-core processors. In 2006, Ierusalimschy [Ierusalimschy 2006] proposed the use of multiple, independent states in Lua to implement *Lua processes*, based on some form of message passing. In this paper we advance that proposal, building a complete library for concurrent programming in Lua based on message passing over channels. As we will see, the resulting library showed encouraging performance results even when running hundreds of thousands of simultaneous processes.

The rest of this paper is organized as follows. In section 2 we point out some of the downsides of multithreading and present the devised model for concurrent programming in Lua. In section 3 we describe how we implemented this model and in Section 4 we present some results of a performance evaluation of the implementation. Finally, in Section 5, we draw some conclusions.

2. Concurrent Programming in Lua

Programming with preemptive multithreading and shared memory demands synchronization constructions to ensure mutual exclusion and conditional synchronization [Andrews and Schneider 1983]. Unfortunately, the synchronization burden, the difficulty to debug code, and the lack of determinism during execution makes development with preemptive multithreading and shared memory admittedly complex [Lee 2006].

Moreover, as argued by Ousterhout [Ousterhout 1996], the criticism to multithreading is not limited to development complexity. It is often difficult to obtain good performance when using preemptive multithreading with shared memory. A too coarse locking reduces the opportunities for concurrency, while a fine-grained locking may add too much overhead to the program. Due to these difficulties, many standard libraries are not *thread-safe*, that is, cannot ensure proper behavior of their functions during simultaneous execution by multiple threads, hampering software from exploring this kind of multithreading. Problems with performance are greatly increased when scaling up to massive multithreading.

These issues have encouraged the development of alternative models for concurrent programming. In this work we explore a model based on execution threads with no shared memory, which use message passing for synchronization and communication. We implemented this concurrency model for Lua through a library called `luaproc`. Next, we describe the model details and the API provided by this library.

Because they have independent resources, we call each thread in the library a *Lua process*. We create Lua processes through calls to the `luaproc.newproc` function. As user threads, Lua processes are entities scheduled exclusively through a scheduler which runs in user space, without direct relation to operating system processes or other kernel scheduled entities.

Communication between Lua processes occurs exclusively through message passing. On the one hand, communication with message passing can be slower when compared to shared memory. On the other hand, the lack of shared memory avoids the performance and complexity penalties associated to shared-memory synchronization primitives. Besides, programs can use the same communication model for processes within the same machine and for processes in a distributed environment.

As their own names imply, the `luaproc.send` function sends messages and the `luaproc.receive` function receives messages. Message addressing is based on *channels*. Channels must be explicitly created by the `luaproc.newchannel` function and destroyed by the `luaproc.delchannel` function. A channel is an entity on its own, without any direct relation to Lua processes. Each channel is named by a string, which must be specified as a parameter to the `luaproc.newchannel` function. Each process may send to and receive from any channel, as long as the process knows the channel name. Thus, it suffices to know a channel name in order to use it.

Each message carries a tuple of atomic Lua values: strings, number, or booleans. More complex types must be encoded in some form. For instance, it is easy in Lua to serialize data [Ierusalimschy 2006], that is to convert it into a stream of bytes or characters, in order to save it in a file, send it through a network connection or, in this case, send it in a message. Structured values can easily be encoded as a piece of Lua code (as a string) that, when executed, reconstructs that value in the receiver.

The `luaproc.send` operation is synchronous: it returns only after another Lua process has received its message on the targeted channel or if the channel does not exist. Otherwise the sending Lua process is blocked until one of these two conditions happen.

The `luaproc.receive` function, on the other hand, can be either synchronous or asynchronous, depending on a parameter. A synchronous receive behaves similarly to a synchronous send: it only returns after matching with a send operation on that channel, or if the channel does not exist. The asynchronous receive operation, in contrast, always returns immediately; its result indicates whether it got any message.

The reason we opted for blocking on send operations is that this provides a simpler, more deterministic, programming model. When a call to `luaproc.receive` returns successfully, it is possible to assert that the message was received. Additionally, non-blocking sends increase implementation complexity. Of course, the programmer can still send messages asynchronously. One easy way to do that is to create a new Lua process with the sole purpose of sending a message. Because the creation of Lua processes is an asynchronous operation, control is immediately returned to the creator process and any occasional communication block affects only the newly created Lua process. Another option is to create a "queue process" to enqueue messages to a channel.

3. Model Implementation

In its standard configuration, Lua includes concurrent programming support through the use of coroutines. Each coroutine represents a different execution flow in user space. Execution control relies on a cooperative model and can be accomplished through calls to the `coroutine.yield` and `coroutine.resume` functions. Calls to the `coroutine.yield` function suspend the coroutine's execution, while calls to the `coroutine.resume` resume it. Once a coroutine starts running, it runs until it finishes or yields.

The API that Lua offers to C includes a function to create coroutines, as well as functions to suspend and resume their execution. This facility, allied to the flexibility offered by the API for interaction with the Lua interpreter from C code and to the dissociation between coroutines and kernel threads, makes Lua particularly well suited for the exploration of alternative models for concurrent programming.

Another important facility offered by Lua is the possibility of multiple Lua states. The entire API that Lua offers to C operates over an abstract type called a *Lua state*. By creating multiple Lua states, a C program can have multiple Lua programs that are completely independent.

Our library, `luaproc`, uses Lua states and coroutines to implement Lua processes. Each process runs as an exclusive coroutine inside its own Lua state. These processes are run by *workers*, which are kernel threads implemented with the POSIX Threads library (pthreads) [IEEE 1995]. There is no fixed relationship between workers and Lua pro-

cesses. Each worker repeatedly gets a process from the ready queue and runs it until it finishes or blocks. Even though we use kernel threads, there is no memory shared among Lua processes, because each has its own Lua state.

The following sub-sections present a more detailed description of our library's implementation and characteristics.

3.1. Lua Processes

Using Lua code from within C code is normally preceded by the creation of a Lua state, represented in C by a variable of type lua_State. A Lua state defines the interpreter's state and keeps track of functions and global variables, among other information related to the interpreter.

Once Lua code has been loaded in a Lua state, it is possible to control its execution through functions provided by the C API for Lua. Control takes place as if the Lua code was executed as a coroutine. Therefore, even if the Lua code does not include explicit calls to Lua's standard coroutine handling functions, it is possible to suspend and resume its execution through C functions. This feature is essential to allow control over Lua processes execution.

Each Lua process is comprised by an independent Lua state, where the process code is loaded during process creation. The independence between Lua states ensures the lack of shared memory between Lua processes and helps to enforce message passing as a means for interprocess communication. The remaining structure used to implement Lua processes is compact and has few members other than the process Lua state. Among relevant structure members are the process execution state (idle, ready, blocked or finished) and the number of arguments that must be used when resuming its execution in case it is blocked. No unique process identifier (PID) is included since there is no fixed relation between workers and processes.

Even though the creation of a Lua state is a cheap operation, loading all standard Lua libraries can take more than ten times the time required to create a state [Ierusalimschy 2006]. Thus, to reduce the cost of creating Lua processes, only the basic standard library and our own library are automatically loaded into each new Lua process. The remaining standard libraries (io, os, table, string, math, and debug) are pre-registered and can be loaded with a standard call to Lua's `require` function.

Our library also offers a facility to recycle Lua processes, which is optionally activated through a call to the `luaproc.recycle` function. Recycling consists in reusing states from finished Lua processes to execute new processes. Instead of being destroyed after finishing its execution, a state can be stored for reuse. Consequently, creation of a Lua process can be done by simply loading new Lua code in a recycled state, thus eliminating the costs of creating a new state and loading libraries.

3.2. Scheduler

The scheduler is automatically initialized when our concurrent programming library is loaded. During its initialization, which occurs in the context of the operating system thread responsible for executing the code that loads our library, a worker is created.

The scheduler manages a single ready queue (FIFO), which holds Lua processes ready for execution. The scheduler itself is responsible for adding newly created Lua

processes to the end of the ready queue. Workers execute the Lua code associated with each Lua process.

Workers are kernel threads managed with the POSIX Threads library (pthreads), which perform the following cycle: it retrieves the first Lua process from the ready queue; executes the Lua code associated with the process until it finishes, blocks or yields; and takes appropriate measures depending on execution outcome. Creation and destruction of workers in execution time is supported through the API functions `luaproc.createworker` and `luaproc.destroyworker`.

If the execution of a Lua process ends because the Lua code related to the process has finished normally, the worker closes the corresponding Lua state and destroys the process. If, during the execution of a Lua process, a call is made to the standard Lua function `coroutine.yield`, the worker simply reinserts the process at the end of the ready queue. This suspends the process execution and allows other processes to execute, which is the expected behavior of a yield. If the execution of a Lua process results in an unexpected error, the worker prints an error message, closes the corresponding Lua state and destroys the process.

Since there is only a single ready queue, all workers must get Lua processes from the same queue. This implies that shared memory synchronization primitives must be used to serialize access and manipulation of the queue. To that matter, conditional variables and mutual exclusion were used, as they are both supported by the POSIX Threads library (pthreads).

3.3. Inter-process Communication

Lua uses a virtual stack to pass values to and from C. Each element in this stack represents a Lua value. Calls from Lua to functions implemented in C use the virtual stack to pass function arguments. Likewise, these C functions use the virtual stack to pass results back to Lua. Therefore, passing messages in our library simply implies copying data from the sender's virtual stack to the receiver's virtual stack.

3.4. Blocking Strategy

In our library, a Lua process can only have its execution blocked in two distinct situations:

1. when it calls the synchronous receive function with a channel where there are no processes waiting to send, that is, when an attempt to receive a message occurs without a previous corresponding attempt to send to the same channel;
2. when it calls the send function with a channel where there are no processes waiting to receive, that is, when an attempt to send a message occurs without a previous corresponding attempt to receive from the same channel.

When a Lua process blocks, the worker adds it to the corresponding channel's queue and gets another process from the ready queue in order to run it. A blocked Lua process is unblocked only if there is a matching call on the same channel or if the channel where it is blocked is destroyed. When such a matching call happens, the same worker that is executing the process that made the call removes the blocked process from the channel queue, copies message data between virtual stacks, and places the unblocked process at the end of the ready queue.

To keep track of Lua processes that are blocked trying to communicate, each channel has two distinct queues (FIFO): one holds processes blocked when trying to send messages to the channel and another holds processes blocked when trying to receive messages from the channel. At most one of these queues will not be empty at any given time, otherwise the processes from each queue could match.

3.5. A Sample Application

In this section we present, in listing 1, the source code of a sample "hello world" application developed with our library.

Listing 1. A simple 'hello world" application with Lua processes.

```
-- load our concurrent programming library
require "luaproc"

-- create an additional worker
luaproc.createworker()

-- create a new lua process
luaproc.newproc( [==[
    -- create a new channel
    luaproc.newchannel( "achannel" )
    -- create a new lua process
    luaproc.newproc( [=[
        -- send a message to the channel
        luaproc.send( "achannel", "hello world" )
    ]=] )
    -- create a new lua process
    luaproc.newproc( [=[
        -- receive a message from the channel
        msg = luaproc.receive( "achannel" )
        -- print the received message
        print( msg )
    ]=] )
]==] )

-- wait until all lua processes
-- have finished before exiting
luaproc.exit()
```

As we can see, the program begins by loading our library with the standard Lua `require` function. Then, it creates an additional worker and a main Lua process that will hold the remaining of our application. This main Lua process creates a channel and two additional Lua processes. While one of these processes sends a message on the channel, the other one receives the message and then prints it. We ensure that our application will not exit before all Lua processes have completed their execution by calling the `luaproc.exit` function, which simply prevents workers from exiting while there are unfinished Lua processes.

4. Performance Evaluation

In this section we describe some experiments we made to evaluate the performance of `luaproc`. All tests, unless stated otherwise, were conducted on a computer running the Linux operating system, with an AMD Athlon 64 X2 dual-core 3600+ processor with 3 GB of RAM. The chosen Linux distribution was Ubuntu 7.10 (Gutsy Gibbon), with standard kernel 2.6.22-14-generic #1 SMP and Native POSIX Threads library (NPTL) 2.6.1. All tests used two workers in order to exploit both processor cores and stimulate parallelism.

The tests ran on an unprivileged user account on the operating system. Each test was executed at least three times and the results presented on this section correspond to the values' arithmetic mean. Execution times were measured with Linux's Bourne Again shell (bash) time command.

We also carried out some comparative tests to evaluate our library against Erlang [Armstrong 2007]. Despite syntax heterogeneity and implementation differences, notably Erlang's built-in support for most of the functionalities we implement through a library, these tests represent an important benchmark. Moreover, further analysis of their results could result in future improvements to our library.

Erlang offers three different execution modes: interpreted code (`escript`), compiled code with symmetric multiprocessing (SMP) support enabled (`erl -smp`), and compiled code with SMP support disabled (`erl`). In all our tests there was just a slight variation in execution time between compiled code with SMP support enabled and compiled code with SMP support disabled. In fact, our tests consistently showed slightly worse (higher) execution times when SMP support was enabled. Furthermore, SMP support is disabled by default for compiled code and not supported for interpreted code. For these reasons, we choose only to present times for interpreted code and compiled code with SMP support disabled.

Further comparative tests were also carried out in order to evaluate our library against a traditional kernel multithreading with shared memory model by using the POSIX Threads library (pthreads) [Skyrme 2008]. We believe this comparison is not as interesting as the one with Erlang, since the pthreads library does not include built-in message passing primitives and is not known to offer scalability that allows for massive concurrency. Therefore, due to space constraints, we opted not to present them in this work.

4.1. Process Creation

In this simple test we measure execution time to create increasing number of Lua processes. First, a main Lua process which will host the remainder of our application's code is created. Then, from within the main Lua process, the same number of Lua processes and communication channels are created, as if each Lua process had its own channel. The spawned Lua processes simply wait for a message from the main Lua process, which is only sent after all of them have been spawned, and then finish their execution.

We reproduce a similar test with Erlang. Just as in the Lua processes test, a certain number of Erlang processes are created and wait for a message, sent by a main process, before finishing their execution. The main difference is that in Erlang there is no need (nor support) to create communication channels, since messages are addressed using process identifiers (PIDs).

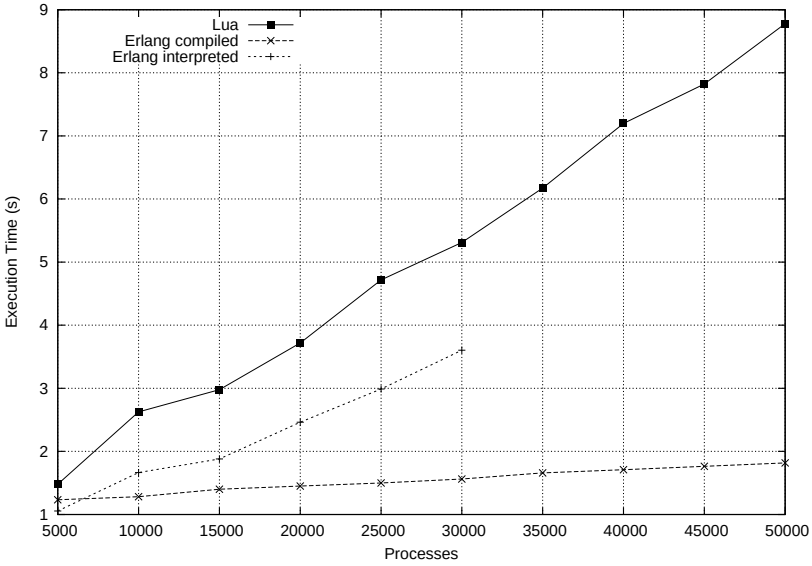

Figure 1. Execution times to create n processes in Lua and Erlang.

Figure 1 shows the total execution times for creating increasing quantities of Lua processes, along with the total execution times for creating, both using interpreted code and compiled code, Erlang processes. Erlang's interpreter (`escript`) limits process creation to 30,000 processes, which explains why the corresponding line in the figure reaches an upper bound at that point.

As we can observe in the figure, execution time increases almost linearly with the number of processes, both in Lua and in Erlang.

4.2. Communication

Message passing is the intended way for Lua processes to communicate and synchronize, therefore it is important to evaluate how it performs. In this test we sequentially send and receive messages of different sizes and measure execution time. First, the message contents are read from a file composed of copies of the same string separated by newlines. Then, a main Lua process that will host the remainder of our application's code is created. Next, from within the main Lua process, a communication channel is created and a new Lua process, whose sole purpose is to receive messages, is spawned. Finally, the main Lua process sends the same message sequentially, 1,000 times, to the second Lua process.

We conduct a similar test using Erlang. Just as in the previous test, the main difference between our Lua code and our Erlang code is the lack of need to create communication channels in the later. Apart from that, Erlang code also differs slightly since a few additional messages must be sent in order to inform the receiver process identifier (PID) to the sender process and to ensure proper synchronization.

Figure 2 shows the total execution times for sending and receiving messages of increasing sizes using our library and Erlang. Once again, we present both the results for interpreted and compiled Erlang code.

As we can see in the figure, our library presents good communication performance, with execution times below 0.1s to send messages with up to 10,000 bytes. Erlang,

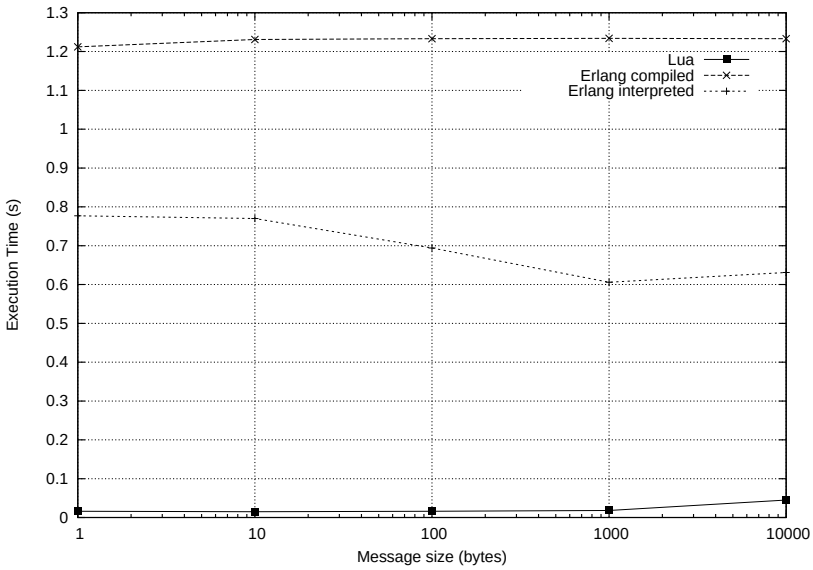

Figure 2. Execution times for sequentially sending 1,000 messages of increasing sizes.

in turn, presents better performance when interpreted, rather than compiled. It presents almost constant execution times when compiled, which suggests it relies on an O(1) operation to perform message passing, such as copying a pointer that points to a shared memory address that holds message data.

4.3. Parallelism

The use of multiple workers in multi-processed environments allows for parallel execution of Lua processes. In this test we explore our library's parallelization potential by implementing a parallel string search application. We also implement a serial version of the same application, using only standard Lua libraries, to evaluate if there is any significant performance cost when our library is used.

The parallel version of the application is divided in three modules. The first module is responsible for initializing the application: it creates workers and communication channels, spawns a coordinator Lua process and several searcher Lua processes, and then sends messages to the coordinator Lua process with the names of the file that holds the patterns and the target to be searched.

The second module is responsible for coordinating job distribution and for centralizing results. It reads the patterns from a file, sends them to the searchers and then starts to progressively distribute target file names to searchers. It also receives results from searchers and notifies them when all target files have been searched.

The third module is the searcher, that is, it is responsible for searching target files for patterns. Each searcher receives a single file name at a time and only sends back its results to the coordinator after processing the whole file. The results sent are composed by the lines of the target file that matched any of the patterns.

This test was exceptionally carried out on a computer with four AMD Opteron dual-core 2.2 GHz processors, for a total of eight processor cores, and 32 GB of RAM. Its

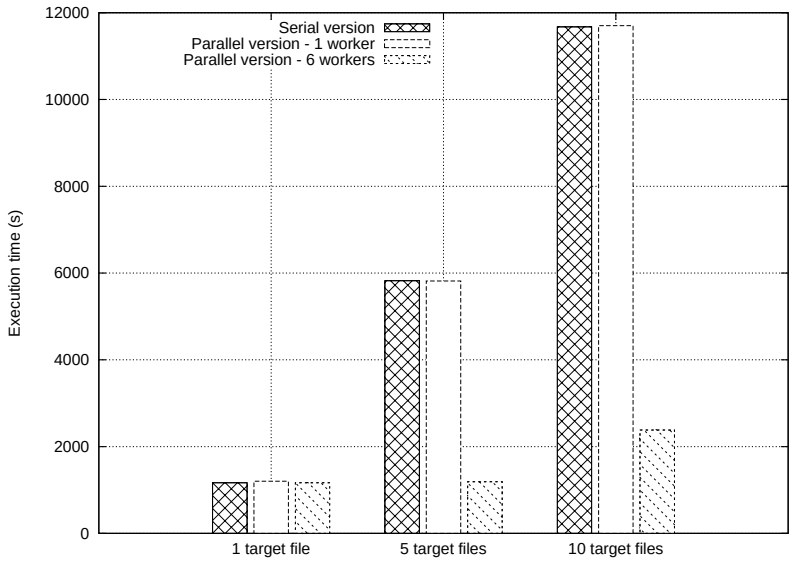

Figure 3. Execution times for serial and parallel string search.

operating system was also Linux, but this time with the CentOS 5.1 distribution, standard kernel 2.6.18-53.1.6.el5xen #1 SMP and Native POSIX Threads library (NPTL) 2.5.

Initially, six workers were used to run the parallel version of the application in order to stimulate parallelism and reduce concurrency in the execution of one coordinator and five searcher Lua processes. Next, still using the parallel version of the application, just a single worker was used, in order to allow for a more balanced comparison with the serial version. The pattern file used throughout the test was the same and it contained 25 lines, with one string per line. The target files were copies of a single file, which included 6,605,423 lines and 2,147,483,849 bytes (around 2 GB). Results are shown in figure 3.

The results indicate that exploitation of parallelism on multi-processed environments, as expected, can result in proportional reductions in execution time. As we can observe, when using the serial version of the application or the parallel version with a single worker (kernel thread) execution time increases almost linearly with the number of target files. On the other hand, when the parallel version runs with six workers, execution times for one or five target files is almost the same, which strongly suggests that while one worker acted as the coordinator, the other five workers acted as searchers and processed the five target files in parallel. Still regarding the parallel version with six workers, it is worth noticing that, once again as expected, execution time increases linearly when the number of target files increase from five to ten.

Finally, results also show an almost insignificant difference in execution times between the serial version of the application, which uses only standard Lua libraries, and the parallel version, which uses our library, when it is run with a single worker.

5. Conclusion

In this work we presented an alternative model for concurrent programming in Lua. The model is characterized by using message passing, as opposed to shared memory, as the only inter-process communication method. Its implementation uses kernel threads as

workers so that multiple processes can run in parallel.

The lack of shared memory eliminates the need to control access to data shared among execution flows and critical regions, which simplifies development and reduces the probability of inconsistencies which can result in data corruption and execution failures. The predictability of blocked communication, in turn, facilitates debugging and increases determinism of execution flows.

The Lua programming language, despite not being specifically developed for concurrent programming, demonstrated enough flexibility to allow satisfactory implementation of the presented model for concurrent programming. Additionally, it provided adequate performance and scalability to our library, as can be observed through the results presented in Section 4.

Although our library is intended to be used locally only, that is, in individual computers, it is easy to extend it to support execution of Lua processes in a distributed environment. We have already successfully developed and experienced with a very simple client-server application that uses LuaSocket [Nehab 2007], an extension library that adds network support to Lua, to allow for remote creation of Lua processes.

The use of the POSIX Threads library (pthreads) as a means to benefit from kernel threads allowed for the exploitation of parallelism intermediated by the underlying operating system. Nevertheless, paradoxically, it also resulted in a significant increase in development complexity, mostly due to the need to handle typical obstacles related to using preemptive multithreading with shared memory.

The difficulties we experienced while developing the library to implement the presented model ratify criticism to preemptive multithreading with shared memory and reinforce the necessity for alternative models for concurrent programming. The limitations of preemptive multithreading with shared memory, in particular the complexity in development, create difficulties even when it is used just as a building block to structure alternative models.

This work does not exhaust the investigation of the presented model for concurrent programming in Lua, nor the exploration of new alternatives for concurrent programming. Our library could be further improved by new functionalities and it could be further evaluated by development of more complex, or so-called "real–world", applications combined with a more extensive performance evaluation. In addition, the usability of our library, which we intuitively believe to be better than other libraries, still lacks proper testing. Nevertheless, the results presented in this work represent an important step towards allowing other contributing efforts to be undertaken.

References

Andrews, G. R. and Schneider, F. B. (1983). Concepts and Notations for Concurrent Programming. *ACM Comput. Surv.*, 15(1):3–43.

Armstrong, J. (1996). Erlang — a Survey of the Language and its Industrial Applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan.

Armstrong, J. (2007). *Programming Erlang*. Pragmatic Bookshelf, City.

Benton, N., Cardelli, L., and Fournet, C. (2002). Modern Concurrency Abstractions for C#. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 415–440, London, UK. Springer-Verlag.

Caromel, D., Mateu, L., and Tanter, E. (2004). Sequential Object Monitors. In Odersky, M., editor, *ECOOP 2004 - Object-Oriented Programming,18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 316–340, Oslo, Norway. Springer-Verlag.

Chrysanthakopoulos, G. and Singh, S. (2005). An Asynchronous Messaging Library for C#. Electronic Article. Synchronization and Concurrency in Object-Oriented Languages (SCOOL), OOPSLA 2005 Workshop, San Diego, California, USA.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457.

Dijkstra, E. W. (1983). The structure of "THE" - multiprogramming system. *Commun. ACM*, 26(1):49–52.

Hansen, P. B. (1974). A Programming Methodology for Operating System Design. In *IFIP Congress*, pages 394–397.

Hoare, C. A. R. (1972). Towards a theory of parallel programming. *Operating System Techniques*, pages 61–71.

Hoare, C. A. R. (1973). Monitors: an operating system structuring concept. Technical report, Stanford University, Stanford, CA, USA.

IEEE (1995). *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA.

Ierusalimschy, R. (2006). *Programming in Lua*. Lua.Org, Second edition.

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

Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2006). *Lua Reference Manual 5.1*. Lua.Org.

Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007). The evolution of Lua. In *Third ACM SIGPLAN Conference on History of Programming Languages*, pages 2–1–2–26, San Diego, CA.

Lee, E. A. (2006). The Problem with Threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

Nehab, D. (2007). Luasocket: Network support for the Lua language. Website. http://www.tecgraf.puc-rio.br/luasocket.

Ousterhout, J. (1996). Why Threads Are a Bad Idea (for most purposes). *Presentation given at the 1996 Usenix Annual Technical Conference, January*.

Skyrme, A. (2008). An Alternative Model for Concurrent Programming in Lua. Master's thesis, Pontifical Catholic University of Rio de Janeiro (PUC–Rio).