Lua Chords

April 27, 2011

Abstract

Concurrency is a fundamental trend in modern programming. Languages should provide mechanisms for programmers to write correct and predictable programs that can take advantage of the computational power of current architectures. Chords is a high level synchronization construction adequate for multithreaded environments. This work studies chords through the implementation of a chords library in Lua.

1 Introduction

Concurrency is a fundamental trend in modern computer programming. However, multithreaded programming, the de-facto standard model for concurrent programming, is known to be hard and error-prone [8]. The difficulty to write correct multithreaded programs comes from the combination of resource sharing in shared memory environments with mutable state, with the preemptive scheduling of threads. Since the programmer is not able to control the sequence of interleavings nor the moment the threads are preempted, there will be a very large range of different ways in which the threads can run, each one producing potentially different results due to the interference with the execution of the other threads. To produce predictable results under that scenario, programmers need to resort to synchronization mechanisms in order to prune the spectrum of possible execution histories to the desired ones.

There are various approaches to cope with concurrency so as to produce correct programs. One of them consists basically in avoiding using threads directly. Instead, new programming models, deterministic and reliable, can be built above threads. In this direction, new concurrency abstractions have been proposed for multithreading-based languages. One of those mechanisms is called *Chords*, introduced in Polyphonic C# [1].

A chord is a synchronization construct that allows coordinating events. A chord associates the execution of a function (the chord body) with the invocation of a set of messages specified in the chord header: after those messages are invoked, the chord is enabled and the body is fired (executed). Those messages can be synchronous or asynchronous. Asynchronous methods return immediately, while synchronous methods block until all header methods are called.

That is, if a chord is described as:

```
sync send([args]) & ready([args]) {func([params])}
```

it is expected that after send and ready are invoked, the function `func` will execute in the thread where the call to send was issued (the synchronous method).

The effect is tantamount to the creation of a (one-shot) continuation on the execution of the synchronous method that is executed on the space of the synchronous process when the chord is enabled.

The work described in this paper has been developed using the Lua language [6]. Despite its many advantages, the cooperative multithreading supported by Lua is not suitable for execution in multicored environments. We look for models to take advantage of these environments while enabling to write correct programs. Chords are an object oriented version of the join patterns from the join-calculus process calculus [3]. They are closely related with Petri Nets [12, 14] and constraint programming [16]. Besides Polyphonic C# and its successors Cω [1], MC# [5] and Parallel C# [4], there are also Java implementations of chords, based either in the modification of the JVM (Join Java [7]) or in source-to-source transformation (JChords [17]). Other chord implementations are present in VODKA [14], Concurrent Basic [15], the boost-join library [9] and JErlang [13].

The advantages of Chords include:

(i) allowing to state explicitly the rules of synchronization,

(ii) implicit synchronization control by means of enforcing the stated synchronization rules and

(iii) abstraction: hiding from the programmer the details of implementing locking, suspension and restart of the computations involved. The creation of the threads is also abstracted from the programmer: a new thread is created when a chord is enabled, given that all its methods are asynchronous.

(iv) composable,

(v) avoiding deadlock, as discussed in 2.2.1,

(vi) adequate for local and distributed settings. Chords permit to synchronise methods that can be invoked either on the local machine or from another host through remote method calls.

That makes Chords an attractive construction for local concurrency and also distributed computing. However, the particularities of programming using a message-passing style, along with the performance costs involved in the runtime operation, still hinder their success.

In this paper we describe LChords [11], a library to provide an infrastructure for chord-based programming in Lua. The contributions of this work include:

- A description of an implementation of a chord library in Lua to study the advantages of this model and its feasibility;
- a proposal for separating basic mechanisms from customizable user policies.

2 Lua chords

The advantages of using chords as the concurrency mechanism during the design of a language include better integration, extended syntax and the ability to detect syntax errors at compile time. However, building a library allows for faster

prototyping while conducting our study, thus we have chosen to build a chords library instead of creating a new language. To that end, we have developed an implementation of a chords library in Lua we have named LChords [11]. This section provides an introduction to Lua. Then we present the LChords library.

2.1 Brief introduction to Lua

Lua [6] is an interpreted, procedural and dynamically-typed programming language. It is based on prototypes and features garbage collection. Tables are the language single data structuring mechanism and implement associative arrays, indexed by any value of the language except *nil*.

Closures and coroutines are first-class values in Lua. Lua coroutines are lines of execution with their own stack and instruction pointer, sharing global data with other coroutines. In contrast to traditional threads (for instance, Posix threads), coroutines are collaborative: a running coroutine suspends execution only when it explicitly requests to do so. Lua coroutines are asymmetric. They are controlled through calls to the *coroutine* module. A coroutine is defined through an invocation of *create* with a function as parameter. The created coroutine can be (re)initiated by invoking *resume*, and executes until it suspends itself explicitly calling *yield*. The first value returned by *coroutine.resume* is *true* or *false* indicating whether the coroutine was resumed successfully, further results are the values passed optionally to *yield*.

Listing 1: Example using Lua coroutines

```
1 function func()
2   coroutine.yield("Now I'm yielding")
3 end
4
5 co = coroutine.create(func)
6 print(coroutine.resume(co)) --> true     Now I'm yielding
7 print(coroutine.status(co)) --> suspended
8 coroutine.resume(co)
9 print(coroutine.status(co)) --> dead
```

2.2 LChords by example

The intention, when we developed our chord library, was to implement in Lua the chord semantics as described in Polyphonic C# [1], trying to surpass the limitations involved in building a library instead of compiling a new language. When appropriate, other extensions to chords are evaluated.

Our implementation of LChords allows coordinating synchronous and asynchronous messages. The distinction between synchronous or asynchronous in LChords define the way methods behave after a match, that is, after all the messages are received: asynchronous methods return immediately, while the (at most one) synchronous method block until the chord is enabled. Note that, in consequence, any number of asynchronous methods can be called on the same coroutine even if they take part of the same chord header. The chord will be eventually fired after all the methods in the header are invoked. That is not valid for synchronous methods since, when invoked, they block the execution of the coroutine, preventing the invocation of the rest of the methods necessary to fire the execution of the chord and thus, causing the starvation of that coroutine.

After the invocation of a method, the value of the arguments of the call must be saved until the rule is satisfied. There is no guarantee that the value of the arguments is not modified from the moment the call is issued until the event of the execution of the chord body. When a chord is enabled, its body is executed on the thread of the synchronous method, if any. In our implementation, we restrict that each chord can declare at most one synchronous method. If no synchronous method was defined in chord, a new thread (coroutine) shall be created in order to execute the body of the chord.

The chords platform is in charge of creating new messages, so they can be used across the program as regular functions. In LChord, messages behave like but are not actually functions, but rather synchronization signals carrying information. That is, they are not directly executed, they are used to fire the execution of the chord body using the arguments of the calls as parameters. Messages may appear in various patterns simultaneously, thus they must be created as object of the system.

Finally, LChord allows to define sets of joins working together, like the *joint* construction in the boost-join library [9]. This favors encapsulation, allows to group messages in namespaces and to specify diferent policies for every chord set, like for instance, a scheduling strategy. Besides, sets can be composed in order to create more complex structures.

The API of LChords can be described as follows:

- `new` returns a set of chords. This function can optionally receive the policy to be used for matching.
- `msgs` allows to define the messages of a chord set. It receives a string containing the name of every message to be created, plus a "sync" identifier when the message is synchronous.
- `join` creates a chord. It receives a list of messages with their parameters as strings, and the function that constitutes the body of the chord.

To illustrate how the library can be used, we describe the solution for a well known multi-process synchronization problem: the producers-consumers problem, as programmed using LChords.

To define a buffer for the producers-consumers problem, we need to load the lchord library:

```
local lchord = require ''lchord''
```

then we initialize a new set of chords we called buffer:

```
local buffer = lchord.new ( policy )
```

Now we can define the valid messages in buffer. We need the message `put` to send a value to the buffer, and the message `empty` to mimic the availability of slots inside the buffer (a message per slot). We create message `get` in order to receive the value saved in the buffer and a message `full` to indicate that the buffer has full slots. Messages `put` and `get` are declared as synchronous because they are locked until the condition (empty or full, respectively) be satisfied.

```
buffer:msgs ( 'sync get', 'sync put', 'full' , 'empty' )
```

Now, messages can be accessed inside the namespace `buffer`, as `buffer.get`, `buffer.full`, etc. To express the two synchronization rules enforced in the problem we write:

```
buffer:join (buffer.put) 'val' '&' (buffer.empty) ' ' {
      function (val)  buffer.full(val) end
}
buffer:join (buffer.get)' ' '&' (buffer.full) "val" {
      function (val) buffer.empty(); return val end
}
```

Now we create the coroutines, register them in the scheduler and start it. Messages empty and full do not need a separate coroutine to execute: full will execute in the space of the coroutine that invoked the `put` message, in turn, `empty` executes in the space of `get`. The first messages `empty`, those that we need to initialize the slots in the buffer, may execute in the main coroutine before the scheduler is started (otherwise it would never be scheduled). In the example implement a two-place buffer by sending the message `empty` twice.

```
buffer.empty()
buffer.empty()

local Get1 = coroutine.create(buffer.get)
local Get2 = coroutine.create(buffer.get)
local Put1 = coroutine.create(buffer.put)
local Put2 = coroutine.create(buffer.put)

buffer.scheduler:register(Put1, 35)
buffer.scheduler:register(Put2, 21)
buffer.scheduler:register(Get1)
buffer.scheduler:register(Get2)

buffer.scheduler:start()
```

This solution for the producers-consumers problem has the advantage of allowing to create buffers of any size, just by sending the desired number of `empty` messages to create the slots. Non interference among the coroutines is guaranteed because the information is not shared but transferred as arguments when the messages are invoked.

In what follows, we describe a more interesting example: a chord-based solution to the Dijsktra's dining philosophers problem. Solving this problem using chords is attractive because avoid deadlock by design: a necessary condition for a deadlock to occur is that resources be taken one at a time and are only released after the end of execution. In this case, it does not happen, since the resources are taken all at once or none. Two forks are grabbed, or none.

2.2.1 The dining philosophers by LChords

There is a group of philosophers (the number is irrelevant here, if more than one) performing continually the same actions: eating and thinking. They need two forks to eat but those forks are shared: one with the left neighbour in the table, and the other with the right neighbour.

Since each philosopher perform the same actions, and to avoid repeating the same code for all philosophers, we encapsulate their behaviour in a class we named Philosopher. Initially we can define the class Philosopher as:

```
Philosopher:new = function(name, leftFork, rightFork)
    thisPhil = {}
    thisPhil.name = name
    thisPhil.leftFork = leftFork
    thisPhil.rightFork = rightFork
    return thisPhil
end
```

where `leftFork` and `rightFork` are the forks the philosopher needs to be able to eat.

As the philosopher needs both forks to eat, we create a chord to specify the synchronization rule that enables the philosopher to eat after the invocation of those messages plus a `takeForks` message indicating that the philosopher is hungry. Note that we need a different `takeForks` message for every philosopher. We achieve this encoding its creation as part of the class and creating a message which name is the result of concatenating the string `takeForks` with the name of the philosopher we are instantiating:

```
Philosopher:msgs ('sync takeForks'..name)
thisPhil.takeForks = Philosopher["takeForks"..name]
```

Now we can define synchronization rules made by invocations to the methods `leftFork` and `rightFork` respect to each philosopher:

```
Philosopher:join (leftFork)'' '&' (rightFork)'' '&'
      (thisPhil.takeForks)'self' {
    function(self)
         print("Phil ", self.name, " is eating with", self.leftFork, self.rightFork)
         coroutine.yield() -- I'll let another to progress
     end
}
```

Messages `leftFork` and `rightFork` indicate that the forks are available. To initiate the execution, the forks must be available, thus we make a call correspondent to every fork.

```
leftFork() --We have to send this message to start
```

Note that the forks are messages shared with the left and right neighbour of each philosopher, respectively. Thus, we need to define a message for every fork, in this case we have four philosophers:

```
Philosopher:msgs ('fork0', 'fork1', 'fork2', 'fork3')
```

Now we can initialize a philosopher, specifying its name and its corresponding forks:

```
local marx = Philosopher:new ('Marx', Philosopher.fork0, Philosopher.fork1)
local engels = Philosopher:new ('Engels', Philosopher.fork1, Philosopher.fork2)
...
```

Now that our class is done, we register the coroutine that will execute the philosophers' code, and then we start the scheduler:

```
for _,phil in ipairs {marx, engels, hegel, maqui} do
        local coro = coroutine.create(phil.run)
        Philosopher.scheduler:register (coro, phil)
end

Philosopher.scheduler:start()
```

The example shows a feature of LChords that is allowing the dynamic creation of messages and chords. Note that it is possible to declare new messages dynamically, as it was required in order to have a different `takeForks` message for every Philosopher. After that message was created, we needed to define the synchronization conditions that allow to control when the apetite of the philosopher can be satisfied. On the other side, the fork messages where created on the beginning of the program and are shared by the philosophers.

This example also shows an advantage of using chords: it is possible to avoid deadlock by means of guaranteeing that there is no way a resource will be taken unless all the needed resources are available. It is known that a necessary condition for deadlock is the gradual acquisition of resources so that they are not

released until the task is completed. That is typical of lock-based implementations of the problem of the dinning philosophers where a fork is taken and then the other. In chords it should not happen since messages are only consumed when the chord is enabled, and then they are all consumed once. An exception is when there is a composition of joins and a circular dependency among resources taking part in the composition or another joins.

The complete program is shown in listing 2 .

Listing 2: A dinning philosphers implementation

```
local lchord = require "lchord"
local math = require "math"

Philosopher = lchord.new()
Philosopher:msgs ('fork0','fork1','fork2','fork3')
Philosopher.new = function(self, name, leftFork, rightFork)
    thisPhil = {}
    thisPhil.name = name
    thisPhil.leftFork = leftFork
    thisPhil.rightFork = rightFork
    Philosopher:msgs ('sync takeForks'..name)
    thisPhil.takeForks = Philosopher["takeForks"..name]
    Philosopher:join (leftFork)'' '&' (rightFork)'' '&' (thisPhil.takeForks)'self' {
        function(self)
            print("Phil ", self.name, " is eating with",self.leftFork,self.rightFork)
      coroutine.yield()
        end
    }
    thisPhil.releaseForks = function (self)
        print(" I am ",self.name," releasing forks",self.leftFork, self.rightFork);
        self.leftFork(); self.rightFork();
        print(" I am ",self.name," forks released ",self.leftFork, self.rightFork);
  coroutine.yield()
    end
    thisPhil.think = function(self)
        print(" I am ",self.name, ", going to sleep");
        Philosopher.scheduler:sleep(math.random(5));
    end
    thisPhil.eat = function(self)
        print ("I am "..self.name.." I'll try to eat with",self.leftFork, self.rightFork)
        self:takeForks()
        self:releaseForks()
    end
    thisPhil.run = function(self)
        while true do self:think(); self:eat() end
    end
    thisPhil.leftFork() --We have to send this message to start
    return thisPhil
end
local marx = Philosopher:new ('Marx', Philosopher.fork0, Philosopher.fork1)
local engels = Philosopher:new ('Engels', Philosopher.fork1, Philosopher.fork2)
local hegel = Philosopher:new ('Hegel', Philosopher.fork2, Philosopher.fork3)
local maqui = Philosopher:new ('Maquiavelo', Philosopher.fork3, Philosopher.fork0)
for _,phil in ipairs {marx, engels, hegel, maqui} do
  local coro = coroutine.create(phil.run)
  Philosopher.scheduler:register (coro, phil)
end
Philosopher.scheduler:start()
```

2.3 Discussion

Several aspects of LChord deserve a through discussion. One of them refers to how the arguments of the message invocations are associated with the formal parameters of the chord body. Plociniczak and Eisenbach [13] suggested that declaring formal parameters could be unnecessary. However, since the chord body is actually a function and, in Lua, functions are first class values that

can be defined anywhere on the code, we found this idea would be error-prone. Thus, in LChords, the association between real parameters on the declaration of the messages at the header and actual parameters in the body of the chord, is based on matching the names of the parameters. In consequence, the restriction, posed in the asynchronous join calculus [3], that the methods on the same header cannot have the same names, and formal parameters are pairwise disjoint, is in place also here. On names collision, the actual value of these parameters is unspecified.

When a message is called, the arguments of the call are saved until the chord is enabled. Since Lua is dynamically typed, there are no limits or in kind or in number to the arguments of an invocation. Any type of value is valid. If the amount of values is higher than the values definition, it will be ignored, if lower, the rest of the values are considered nil.

Another issue is related to the fact that actual parameters can mutate from the invocation of the message to the moment when the body is executed. In Lua, some values are immutable, like numbers or strings, while other values, like tables or coroutines, have an internal mutable state that can be changed by the application. In our implementation, arguments are saved at the time of the invocation. What will happen if they are modified in the meantime? For example, a coroutine may be resumed and then terminate changing it state to `dead`. Therefore, it is possible for a chord to try to execute a coroutine that is active at the moment of the call, but is dead when the body of the chord is finally activated.

This comes naturally from separating the invocation of a method/function from its execution having concurrency in a stateful language where processes (coroutines) are sharing references. In fact, a locking solution would not do better: the execution of a method/function requiring mutual exclusion must delay while there is another process accessing a common resource, which side-effects could include modifying the value of common variables. Then, when the method/function is unlocked, it should verify if the data to be handled is still valid. In Lua this can potentially arise for any composite value, since those values are references instead of native values. Saving a deep copy of the values instead of a reference would be an alternative, though a quite expensive one. Another consists in forcing inmutability (See [2]), which would otherwise require compiler support. Our implementation assumes this fact as part of its semantics, the programmer should be aware of.

Concerning the invocation of synchronous methods, the process (coroutine) where they are executing must block until the chord is enabled. Then, when the join is matched, the chord body is executed on the space of the coroutine where the call was issued. As noted by Benton et al. [1], allowing more that one synchronous process on a chord would permit the construction of a *rendezvous* like synchronization. However, as they explain (i) it is possible to construct such mechanism combining single-synchronous-method based chords (ii) the choice of the thread where the body is executed would influence the result of the execution . The current implementation of LChords considers the presence of at most one synchronous method.

On the other hand, synchronization methods are not mandatory in a header. When all methods in a header are asynchronous, our implementation assumes the same decision made in Polyphonic C#/Cω, that is, a new computation (in this case, a new coroutine) is created to preserve the asynchronous specifica-

tion. This, however, leads to another problem. When the chord is enabled, the library must execute the body in a new coroutine to avoid delaying the last asynchronous method. However, Lua coroutines are asymmetric. Different from symmetric coroutines, that provide a `transfer` function allowing to transfer control to any coroutine, asymmetric coroutines always return to the coroutine that made the `resume`. That means that the method will not be able to return until the initiated coroutine yields or returns. The general idea of chords over cooperative multithreading calls for the need of a scheduler: when a synchronous method blocks, the control must return to another non-blocked coroutine. An asynchronous call in a chord with no synchronous messages should return immediately and leave the execution of the body to a scheduler he should be aware of. That scheduler does not need to be necessarily provided as part of the platform.

Computations may need to return results. Coroutines may return results either when yield or the execution is ended, then the results are received by the coroutine that issued the resume, in this case, the scheduler. For the user to collect the results of a coroutine, a method must be provided by scheduler to that end.

It is known that the coexistence of inheritance and concurrency in concurrent object oriented languages originate a group of problems collectively called as inheritance anomaly [10].

The join-calculus and the implementation guarantees that if there is a match, some chord will be enabled. However, since there is not ordering guarantees (that is, any enabled chord will be selected for execution), a blocked process could starve. Avoiding starvation requires defining fair scheduling policies. On the other hand, contrary to what may seem, it is still possible to write chorded programs suffering from deadlock. Lets take the following example:

```
sync put(v) & empty() {full(v)}
sync get() & full(v) {empty()}
```

According to this code, the continuation `full` will only be executed after the `put` and `empty` calls are issued. Similarly, `empty` will be called after a `full` and `get` messages are received. That is, a system of two processes invoking `get` and `put` cannot proceed if `full` or `empty` were not previously fired. We can see there is a circular dependency, and both `get` and `put` are synchronous, thus they will block until the respective chord is enabled, that is, forever. If synchronous methods must be used, this problem could be solved incrementing the chord with a timeout, to guarantee finite await time, and some way to inform that an error occurred. Support for timeouts is already present in other languages with concurrency support like Ada and Java.

2.4 The need for synchronization mechanisms in Lua: a case for a coroutine-targeted implemention

Lua coroutines provide means for cooperative multithreading in Lua [?]. One advantage of cooperative multithreading is that the programmer can avoid special synchronization mechanisms by carefully choosing the points in the code where each thread yields the execution making their synchronization implicit. However, sometimes it might be necessary to yield at points in the code that might result race conditions. For example, consider the code below:

```
function ForwardTo(data, host, port)
```

```
    local sock = ReuseSockTo(host, port)
    local sent, err, lastpos = 0
    while true do
      sent, err, lastpos = sock:send(data, lastpos+1)
      if not sent and err == "timeout" then
        yield("waitUntilWrite", sock)
      else
        break
      end
    end
  end
```

Function `ForwardTo` sends a byte stream over a socket by repeatedly sending chunks of the stream as big as allowed by the underlying network infrastructure. Whenever the `send` operation on line ?? does not completes due to a timeout, the function yields to the scheduler requesting that it remains suspended until the socket is ready to be written again. This yield must be done in order to improve performance by allowing other threads to run while the current threads waits for the socket to be ready. Ideally, this is not a good point to yield because during this yield other threads can gain the right to execute and write into the same socket interfering with the stream being sent. As a result, there is the need for some synchronization mechanism to avoid that more than one thread writes to the same socket concurrently.

Using LChords we can implement a synchronization mechanisms as illustrated below, where we create a chord to send data through a socket. The `start` message is used only to enable the sender to accept one send request through the message `send`. Whenever the data is entirely sent, the body of the chord calls message `start` again to allow other send request to be processed. Message `send` can be either synchronous or asynchronous depending whether the programmer wants that a call to message `send` returns immediately or only after the data is completely sent over the socket.

```
sender = lchord.new()
sender:msgs('start', 'sync send')
sender:join(sender.start) '&' (sender.send) 'data' 'host' 'port' {
  function(data, host, port)
    ForwardTo(data, host, port)
    sender:start()
  end
}
sender:start()

sender:send("Hello, World!", "some_host", 12345)
```

Arguably, it is much more simpler to provide multithread synchronization mechanisms in a cooperative model than in a preemptive model. However, we implemented our LChords library using a cooperative multithreading model over coroutines as means to experiment with the concept of chords to implement synchronization mechanisms for cooperative threads.

3 Implementation

This section describes the issues we have resolved to provide such behaviour as a Lua library.

There is a clear separation of what is done at "configuration time" and at runtime. Because transactions executed at runtime hurt performance, it is wise to transfer as much as possible of those operations to the "configuration" stage.

At the configuration stage, the messages and queues are created, both join patterns and their continuations are registered, and the arguments of the messages are associated with the continuations parameters.

Then, at runtime, for every invocation it must be verified if some pattern is already satisfied (that is, if the invocation just issued has enabled some join). In that case, the continuation correspondent to the matched pattern will be executed using as parameters the arguments stored for every (enqueued onto the invocations queue) invocation involved in the selected join, consequently removed (popped of the invocations queue). Of course, that process will hurt performance, thus the implementation must invest in optimizing as much as possible the runtime procedure. It is also fundamental that the framework guarantees atomicity during matching, message queue operations and job scheduling [1].

3.1 Configuration stage

The configuration stage starts with the creation of the chord set. A chord set is a table that contains the methods exported by the API, that is, methods `msgs` and `join`, along with a group of private fields, namely, the mask of bits of the set, the list of messages, the list of joins and the scheduler.

When the `msgs` function is called, a new entry in inserted in the messages list. That entry contains an index, the name of the message that was received as a parameter during the creation of the message, whether this message is synchronous, and a reference to the queue where the arguments of the calls to this message are saved. The index of the message indicates the *bit* correspondent to this message in the set bitmask. Every set has a mask of bits indicating that there are pending calls at the correspondent message queue, in order to enhance the performance during matching. The list of messages is indexed by the reference to the function inpersonating the message/method, that function is actually a closure that encapsulates a self-reference: when invoked, the call to this message is enqueued, the bitmask is set, and the match process starts to check for an enabled join.

```
local whenCalled = function (set, f, ...)
                   local msg = set.__msgL[f]

                   msg.queue:enqueue(...)
                   set.__bitMask:set(msg.index)
                   set.scheduler:match (set, f)
end
```

Finally, a new key is created in the set table in order to access this message inside the set namespace, as `set.msg`.

The list of joins is incremented with calls to the `join` function. The `join` arguments are parsed recursively in order to extract the messages composing the join and their arguments, along with the function to be executed when the chord is enabled. That function and the mapping between arguments of the calls to the parameters of the function are saved into the fields of the join. The need to extract the parameter names of inactive functions was one of the motivations for choosing the new version of Lua 5.2 (at this moment, still an alpha version). Another reason was the availability of bitwise operations. Every join pattern is coded as a bit mask in order to enhance the performance of the matching procedure, just as described in [1], that mask is permanent during the whole execution.

Now the scheduler can be started.

3.2 Runtime stage

When a message is invoked, the call is enqueued, the bit mask is set, and the match function is called in order to test for a match.

The match is retrieved by the chosen policy. If no match is found, asynchronous messages just return. Synchronous messages, instead, must block until the chord is enabled, thus their coroutine is put to sleep: it is moved into a list for sleeping coroutines and yielded. When the join is enabled, that coroutine will be resumed in order to execute the continuation of the join. Since all the parameters of the execution will be available only after the join is enabled, the coroutine receives them as `resume` arguments. Then the function can be executed:

```
table.insert(self.sleeping [msg], coro)
local func, params = coroutine.yield()
func (table.unpack(params))
```

When a match is found, a cleanup procedure is triggered. Every call involved in the match are dequeued, and the call arguments replaced in the function call. Arguments shortages and surpluses are ignored. Then, if the queues are already empty, the correspondent bit in the set bitmask must be cleared. If the join matched has no synchronous messages, then a new coroutine is registered on the scheduler in order to execute the continuation, and the message that triggered the match returns. To the contrary, if there is a synchronous message in the join and that is the one that executes in the current coroutine, the function is just executed. Would otherwise, the synchronous method of the join is blocked waiting for a match, thus we must move it from the *sleeping* table and registering it back in the scheduler with the function and the parameters of the continuation. The scheduler will resume it using the function and the parameters of the continuations as arguments, that will be returned by the `yield` function the coroutine was executing (see 3.2).

Aditionally, the scheduler provides functions for registering coroutines, returning results and timing functions. Timing functions are necessary when the scheduler is idle to avoid a busy wait, and to allow functions to sleep, for instance, in the example of the Dining Philosophers. For example, the function sleep yields if the timeout has not been reached yet.

```
sleep = function (self, timeout)
  local StartTime = os.time()
  while self.time(StartTime) < timeout do
    coroutine.yield()
  end
end,
```

3.3 Custom Policies

If several Polyphonic chords are enabled, in theory an unspecified chord is selected for execution, while in fact, the policy is first match–first run. We have studied ways for the framework to allow the programmer to define custom policies. This section discusses various examples of frequently used policies to determine what the framework should provide in order to make this possible.

3.3.1 A sched_first_match policy

The first_match_policy is a policy frequently provided in practical chord implementations. Following this policy the program returns the first join found enabled. Provided that the platform makes available an *iterator* that returns every matched join in any order (that non-determinism is offered for free by Lua hash tables), we just need to make calls to the iterator until the result is false or the end of the table was reached (a nil is returned).

```
local default = function (set)
        for status, join in iterFact(set) do
            if status==true then return join end
        end
        return nil
  end
```

3.3.2 Implementing a sched_round_robin policy

In a sched_round_robin policy, the selected join must be the one least recently enabled. To implement that policy, the user-programmer can increment the joins with fields to allow for custom matching:

```
local round_robin = function (set)
        local ret = nil
        for status, join in iterFact(set) do
            if status==true then
                if join._visited == nil then
                        join._visited = os.time()
                        return join
                else
                        if ret == nil then
                                ret = join
                        else
                                ret = ret._visited > join._visited and join or ret
                        end
                end
            end
        end
        if ret then ret._visited = os.time() end
        return ret
end
```

3.3.3 Implementing a sched_sync_first policy

Another policies to be considered include:

1. sched_longest_match: that is, order by the number of messages contained in a join;
2. sched_sync_first: joins containing sync messages have higher priority;
3. sched_pri_first_match, sched_pri_longest_match, sched_pri_round_robin: priorities based policies.

4 Analysis

We have noted in those examples the main advantage of programming using chords, namely: making synchronization rules explicit. We can also see how the characteristics of Lua, like dynamic typing and cooperative concurrency support, made the implementation easier in some aspects.

4.1 The need for a multithreaded implementation

Actually, the real need for chords and the worst problems it could face come in a (really) concurrent environment, when race conditions and other concurrent hazards are in place. While Lua's coroutine approach for concurrency works very well for monoprocessor architectures, the fact is that currently it cannot exploit fully the nowadays ubiquitous multicored architectures.

Several proposals can be found in literature to solve this problem, all of them with there strengts and weaknesses. (cite) The remainder of this paper discusses our implementation of a concurrent Lua based on the proposed chord library and the challenges it represents compared to the coroutine version.

5 Final remarks

Concurrency is an urgent issue in modern programming. Languages should provide mechanisms allowing programmers to write correct and predictable programs that can take advantage of the computational power provided by current architectures. This work studies chords, a high level synchronization construction adequate for multithreaded environments. Through the implementation of a chord library in Lua, we have analyzed the issues related with this mechanism. The implementation took advantage of several Lua features such as first-order functions and dynamic typing. We have concluded that the main advantage of chords is in its capacity for declarative specification of synchronization. On the other hand, we noted some problems that show up when using chords, and we studied how some of those problems can be overcome.

The reader can notice that our implementation enjoys the simplicity of the cooperative multithreading model, turning unnecessary to use locking mechanisms "under the hoods". As Benton et alii [1] comment, implementing chords require atomicity to decide if a chord was enabled, to pop pending calls and when scheduling the chord body for execution. In Lua, atomicity is guaranteed since it is the programmer who explicitly give the control up by placing calls to *yield* on its program.

As a future work, given the differences among the various implementations, we want to know: What are the roots for those mismatches? Do they rely on the particular language characteristics? What should be defined as the minimal chord implementation and what can be left for the user/programmer to decide?

Acknowledgements

This work was partially funded by CNPq Brasil.

References

[1] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):769–804, 2004.

[2] Jan Schäfer Christian Haack, Erik Poll and Aleksy Schubert. Immutable Objects for a Java-Like Language. In *Proc. European Symposium On Pro-*

gramming 2007, volume 4421 of *Springer Lecture Notes in Computer Science*, pages 347–362, 2007.

[3] Cedric Fournet and Georges Gonthier. The Join Calculus: a language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pages 268–332. Springer-Verlag, 2000.

[4] Vadim Guzev. Parallel C#: the usage of chords and higher-order functions in the design of parallel programming languages, 2008. http://parallelcsharp.com/docs/conferences/PDPTA08/PDPTA08.pdf.

[5] Vadim Guzev and Yury Serdyuk. Asynchronous Parallel Programming Language Based on the Microsoft .NET Platform. In *PaCT 2003 : parallel computing technologies*, volume 2763/2003 of *Lecture Notes in Computer Science*, pages 236–243. Springer, 2003.

[6] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA, 2007. ACM.

[7] Stewart Itzstein and Mark Jasiunas. On implementing high level concurrency in Java. In *In Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference*, Lecture Notes in Computer Science, pages 151–165. Springer, 2003.

[8] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[9] Yigong Liu. Join - asynchronous message coordination and concurrency library. Available at http://code.google.com/p/join-library/, 2007. Last visited on 13 january 2011.

[10] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. pages 107–150, 1993.

[11] A. Milanés and R. S. Bigonha. Moonlight Chords. 4th Workshop on Languages and Tools for Multithreaded, Parallel and Distributed Programming (LTPD 2010), 2010.

[12] Martin Odersky. Functional nets. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

[13] Hubert Plociniczak and Susan Eisenbach. Jerlang: Erlang with joins. In Dave Clarke and Gul A. Agha, editors, *COORDINATION*, volume 6116 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2010.

[14] Tiark Rompf. Design and implementation of a programming language for concurrent interactive systems. Master's thesis, 2007.

[15] Claudio V. Russo. Join patterns for visual basic. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 53–72, New York, NY, USA, 2008. ACM.

[16] Martin Sulzmann and Edmund S.L. Lam. Haskell - Join - Rules. In Olaf Chitil, editor, *IFL '07: 19th Intl. Symp. Implementation and Application of Functional Languages*, pages 195–210, September 2007.

[17] Sergio Vale e Pace. Programação concorrente baseada em acordes para plataforma Java. Master's thesis, Departamento de Ciência da Computação, DCC/UFMG, 2009.