Moonlight Chords

Anolan Milanés[1], **Roberto S. Bigonha**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

{anolan, bigonha}@dcc.ufmg.br

Abstract. *Concurrency is a fundamental trend 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. 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 [Lee 2006]. The difficulty to design and write correct concurrent programs following that model comes from the conjunction of the shared memory model with the preemptive execution of multiple threads, which may produce unpredictable executions. When the execution of the processes interferes, programmers need to resort to synchronization mechanisms in order to prune the spectrum of possible histories to the desired ones.

There are various approaches to cope with concurrency, some proposing alternatives to multithreading either based on non-preemption or avoiding shared memory. In any case, the idea consists in discarding the multithreading as programming model, maybe building above it another model, deterministic and reliable. New asynchronous concurrency abstractions have been proposed for multithreading based languages like Polyphonic C$^{\#}$ [Benton et al. 2004]. One of those mechanisms is called *Chords*. A chord is a synchronization construction that allows coordinating events. It is composed of a header and a body. The header reflects the association of the body with a set of methods defined in the header: the execution of the body is deferred until all the invocations declared on the chord header are issued. Methods can be synchronous or asynchronous. Synchronous methods block until the chord is enabled (all the methods in the header have been called), asynchronous methods return immediately. A particular method may appear in several headers; if several chords are enabled, in theory an unspecified chord is selected for execution.

Advantages of Chords include explicit declarative concurrency support and encapsulation of the underlined platform. That makes them a suitable construction for local concurrency and also distributed computing, while avoiding the problems related to lock-based programming. Chords are an object oriented version of the join patterns from the join-calculus process calculus [Fournet and Gonthier 2000]. They have been implemented in Polyphonic C# and its successor Cω [Benton et al. 2004], also in MC# [Guzev and Serdyuk 2003] and Parallel C# [Guzev 2008] that extend Polyphonic C#. There are also Java implementations, based either in the modification

of the JVM (Join Java [Itzstein and Jasiunas 2003] or source-to-source transformation (JChords [Vale e Pace 2009]). This work studies the issues related to the project and implementation of chords through the construction of a chord library in Lua, available at [MB10 2010].

2. Brief introduction to Lua

Lua [Ierusalimschy et al. 2007] is an interpreted, procedural and dynamically-typed programming language. It is based on prototypes and features garbage collection. Tables are the language's 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 suspend 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
```

3. Proposal

Before initiating the implementation of our chord library, there are some aspects that must be defined:

1. Parameters substitution. How to behave on collision? Parameter substitution is done by name. That leads to the restriction that methods on the same header cannot have the same parameter names. On collision the behaviour is unspecified.
2. Since on the invocation of a method its parameters must be saved until the guard succeeds, when will actual parameters be evaluated? What will happen if they are modified in the meantime? In Lua, closures and coroutines are first class values. This means that the chord body can receive closures and coroutines as parameters. Also, Lua provides lexical scope and, as such, closures encapsulate the non-local variables to the function. Thus, if the parameters of the chord body include a coroutine or if the value of its non-local variables change, the result returned by the chord body may be different when executed on the moment a message is issued or after the chord is finally enabled. As result, for instance, a chord could try to execute a coroutine that was active at the moment of the call, but now is already dead. However, it seems that this problem comes naturally from separating the invocation of a method/function of its execution having concurrency in a stateful

language with shared memory. A locking solution would not do better, because it could happen that the execution of a method/function delays while there are others accessing a common resource, which side-effects could include modifying a function non-local variable or executing a coroutine it received as argument. Our implementation executes the chord body when the guard is satisfied, using the parameters whose references were saved at the time of the invocation.

3. Synchronization methods are not mandatory in a header. In case all methods are asynchronous, should they be executed on a new coroutine (following the Polyphonic C#/Cω solution)? Our implementation assumes that solution to preserve the asynchronous specification. This solution, however, leads to another problem, as we shall explain in Section 5.
4. The chord body is executed on the synchronous process that issued the call (if any), thus it must block. As noted by Benton et alii [Benton et al. 2004], permiting more that one synchronous process on a chord would allow the construction of a *rendezvous* like synchronization. However, as they explain and we show in the Example 2, (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.

Our proposal allows coordinating synchronous and asynchronous methods. Asynchronous methods return immediately, while the synchronous method block until a guard is satisfied. After a chord is enabled, the body is executed on the thread of the synchronous method, which is unique for each chord. At least the synchronous method must execute on a different coroutine (otherwise, all processes would deadlock). In order to analyze the requirements of our implementation, we study a set of use cases we present in the sequel.

4. Use cases

The *producer-consumers problem with limited buffer* (or *bounded-buffer problem*) describes two types of processes, producers and consumers, sharing a common, fixed-size buffer with N positions. Producers generate data and put it into the buffer and consumers get it from the buffer, removing it. Producers cannot put data on a full buffer and consumers cannot read data from an empty buffer. In our example, in case a producer finds the buffer full, just like when a consumer finds it empty, it will block. The code in listing 2 shows a solution for the producers-consumers problem based on chords. This implementation consists on two chords that express the synchronization requirements stating that producers produce only when there is an empty slot (line 3) and consumers consume only when there is a full slot (line 2). Producers and consumers should block when the guards are not satisfied, thus `put()` and `get()` methods are declared *synchronous*. The system is initialized by sending an `empty()` message. Note that, while this implementation works for a single-slot buffer, a N-slot buffer can be simply implemented by invoking N times the method `empty()`. It is equivalent to creating N slots. Every method call will match only a chord, thus the behaviour will still be correct (the addition of `empty()` and `full()` messages equals the size of the buffer).

Listing 2. Producers/consumers

```
1 lchords = require "lchords"
2 lchords.join("sync get()", "full(val)")(function(val) lchords.empty(); return val; end)
3 lchords.join("sync put(val)", "empty()")(function(val) lchords.full(val) end)
4 lchords.empty()
```

As noted in [de Moura 2004], the producers-consumers problem is a pattern for many different problems. Thus, problems following the producers-consumers pattern can also be implemented using this mechanism.

The *readers-writers* problem describes a system where many processes may access the shared memory concurrently, either for reading and writing the same location. Only a process can access the location when it is a writer. Several readers are allowed to access at the same time. The code in listing 3 implements a readers-writers lock with readers preference. When a writer wishes to write it asks for an `ExclusiveLockAdq()`.

Listing 3. A readers-writers lock

```
1 lchords = require "lchords"
2   lchords.join("sync ExclusiveLockAdq()", "idle()")()
3   local function ExclusiveLockRel() lchords.idle() end
4   lchords.join("sync SharedLockAdq()", "idle()")(function() lchords.s(1) end)
5   lchords.join("sync SharedLockAdq()", "s(n)") (function(n) lchords.s(n+1) end)
6   lchords.join("SharedLockRel()", "s(n)")(
7       function(n) if n==1 then lchords.idle() else lchords.s(n-1) end end)
8 lchords.idle()
```

The lock is only granted when there is no other process accessing the buffer. This is controlled with the message `idle()`: when there are no accesses to the buffer, it is idle. The chord that controls writers access to the buffer is shown in line 2. When the writer finishes accessing the buffer, it calls the `ExclusiveLockRelease()` method to send a new `idle()` message, allowing access to the buffer. The readers ask for a `SharedLock()` to access the buffer which, as described in the chord in line 4, is only granted when the buffer is `idle()`. As shown, the private acquisition of the lock by readers and writers is controlled by means of consuming an `idle()` message. Since the access can be shared by several readers, the idle message is only sent when there are no more readers. This is expressed by the chord in line 7, where the message s(n) describe the state of the readers (number of readers processing) anytime. While the readers are still accessing the system, the counter in `s(n)` is incremented (line 5).

5. Implementation

Our library provides a solely function: a *join*. Chords can be specified by declaring a header and a body using the syntax *join (method list) (body)*, where *method list* is a list of method names separated by commas. The modifier *sync* can be used to indicate that the method will execute synchronously.

All the information necessary to control the chords are saved in tables `allChords` and `allMsgs`. Table `allChords` contains the list of chords and for every chord, the function body, the name of the synchronous method and a table whose keys are the messages specified in the chord header. Table `allMsgs` is indexed with the name of the messages and contains the number of active invocations, the list of arguments for every invocation and the parameter names. Basically, the library works by checking if any chord where the message was defined is enabled. If there is one, the body is executed with the parameters of some invocation. The code is basically divided in three parts or functions:

- the `join` function receives a list of methods and returns a function to allow the syntax `join(method list)(function)`. The parameter of the returned function is the chord body. The declared methods are created in the chord namespace as functions that receive a variable number of parameters (in Lua is written as *(...)*) and execute the matching operation we explained before.

```
local join = function(...)
local found, args
local idx = #allChords + 1
allChords[idx] = {}
allChords[idx]["msg"] = {}
for i = 1, select("#", ...) do -- preparing the stage
  local msg = select(i, ...)
  msg, found = string.gsub (msg, patternSync, "%1")
  msg, args = string.match(msg,patternMsg)
  allChords[idx]["msg"][msg]=true --The msg is in this chord
  if found ~=0 then allChords[idx].sync = msg end
  allMsgs[msg]={["called"] = 0,["args"] = {}}
  allMsgs[msg]["names"] = {args:gsub (patternArgs,"%1")} --Name of the parameters
  lchords[msg] = function(...)
        return faux(msg, ...)
  end
end
return function(func)
      allChords[idx].func = func
end
end
```

- The matching function looks for enabled chords. While the method is synchronous in some chord and no chord was enabled, it must block (yields). If the enabled chord contains only asynchronous methods, a new coroutine is initiated.

```
local function faux(msg, ...)
    allMsgs[msg].called = allMsgs[msg].called + 1
    table.insert(allMsgs[msg].args, {...} ) --saving args
    local toreturn = true
    for i, chord in ipairs(allChords) do
          if chord["sync"]==msg then
  toreturn = false
  if chkMatch(chord) == true then return execBody(chord) end
          elseif chord["sync"]==nil and chord.msg[msg]==true then --we all async, check for match
  if chkMatch(chord) == true then -- all async and chord matched
     local f = coroutine.wrap(execBody)  -- we have a match
     f(chord)
     return
  end
    end
    end
    if toreturn==true then return else coroutine.yield(); return faux(msg, ...) end
 end
```

- The function that makes the appropriate arrangements for the parameters and executes the body. The original Lua does not provide information about the function arguments before the function is executed. To solve this problem we can use a C function that retrieves that information, or then to receive the function as a string and extract the argument names using pattern matching. Here we are using a Lua version (LuaNua [Milanés et al. 2010]) that provides internal information about the structure of the values.

```
local function execBody(chord)
   local args = {}  -- parameters list
   if chord.func then
      local params = debug.content(debug.content(chord.func).p).locvars or {}
      for i = 1, #params do
          if params[i].startpc==0 then params[i]=params[i].varname else params[i]=nil end
      end
      for msg in pairs(chord.msg) do
    allMsgs[msg].called = allMsgs[msg].called - 1
    idx = next(allMsgs[msg].args) --choose args from list
    for i, v in ipairs(allMsgs[msg].args[idx] or {}) do --
        args[lookup(allMsgs[msg].names[i], params)] = v
    end
    allMsgs[msg].args[idx] = nil
      end
      return chord.func(unpack(args))
   end
end
```

This implementation poses some restrictions:

- At most one method per chord can be declared synchronous (see Section 3).

- The body of a chord with only asynchronous methods on the header, does not have a return statement or it is empty.
- Method on the same header cannot have the same name.
- All the formal arguments in a chord header have distinct names (if the names also appear as body function parameters), otherwise the behaviour is undetermined.
- In all chords, the same method is invoked using the same parameters and order (this restriction belongs to the current implementation and is simple to eliminate).

A problem that this version of the implementation is not able to resolve is how to execute a body of a chord when the header contains only asynchronous methods. In this case, on the invocation of the last needed method to enable the chord, the library must execute the function in another coroutine to avoid delaying the asynchronous method. However, Lua coroutines are asymmetric, that means that the method will not be able to return until the coroutine yields or returns. The method should return immediately and leave the execution of the body for an external scheduler or coroutine available, it should be aware of. We are analyzing the modifications to the syntax to allow for that behaviour, currently the function is executed on a new coroutine initiated by that method.

6. Analysis

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 [Benton et al. 2004] 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.

Since our library is based on coroutines, we further analyze the advantages and disadvantages of offering chords in Lua as a concurrency mechanism compared to simple coroutine mechanisms. The criteria of the analysis include performance, abstraction level, error-proneness, readability and expression power.

Performance: The performance of aplications implemented using coroutines directly will be better than using chords, since we are building chords over coroutines. However, we consider that using chords does not represent a serious performance overhead. The reason is that in our implementation (i) the structure that keeps the information regarding every chord is filled initially when the join is called (that is, just once per chord) (ii) the matching operations consist in indexing the table `AllMsgs` with the message name to increment its call counter and to retrieve the list of chords containing the message last invoked. Then, we check if the invocation has enabled a chord, by traversing the table corresponding to the chord to check the messages counters. Since those lists are unlikely to have more than a few records, those operations should be inexpensive. As noted elsewhere, the most expensive operation related with this mechanism comes at the time of executing the body of a chord composed of asynchronous methods, because a new coroutine will have to be created and resumed.

Abstraction: Abstraction is one of the benefits of chords: after the chord is declared and the synchronization rules are stated, the synchronization of the methods specified in the header is controlled by the chord. The fact that it is build over coroutines or other model is hidden from the user.

Error-proneness: Writing a chord is somewhat different to the usual locking, but after the chord is defined the programmer can forget about it. Coroutine calls must be inserted all across the code, thus the tendency to oblivion is expected to be higher.

Readability: Chords allow to explicitly define and understand the synchronization rules of a system. In coroutine based programming, the synchronization is encoded implicitly inside the program, requiring to analyze the code correspondent to every coroutine to understand how the system works.

Expression power: The listing 4 shows the code for the producers-consumers problem based on coroutines. This implementation has two main advantages: the pro-

Listing 4. Implementation of producers/consumers with coroutines [de Moura 2004]

```
function produtor()
   return coroutine.wrap(function()
                            while true do
                                 item = produz()
                                 coroutine.yield(item)
                            end
                         end)
end

function consumidor(prod)
    while true do
       local item = prod()
       consome(item)
    end
 end
```

ducer does not need to know the consumer, and the consumer uses the producer as a function, thus it is transparent if the producer is a coroutine or not. On the other hand, the implementation based on chords requires that both the producer and the consumer are executed inside coroutines, although they are not aware of the other's condition, or even, of its existence. The advantage of chords over the coroutine implementation is on the clarity the rules are declared. Also, this example is consumer driven, while the example shown for chords the processes run independently ("buffer-driven"): as soon as the buffer is full the consumer can consume, as soon as it is empty, the producer can produce. Also, to modify the example to implement a bounded buffer is simpler in the chord example in 2. The chord implementation is very compact, while the coroutine based is also short.

7. 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 advantages and issues related with this mechanism. We have concluded that the main advantage of chords is in its capacity for declarative specification of synchronization. In this subject, readability, abstraction and error-proneness are better than using coroutines directly. The expressiveness of this construction is satisfactory, while performance should not be mostly affected. The implementation was simplified thanks to Lua features such as first-order functions and dynamic typing.

Future works in this direction include a thoughtful performance analysis and the codification with chords of a real application currently implemented using a state machine. The library can also be extended in order to allow for a syntax where join state-

ments could be used just like methods in a header definition. Also, quantifiers allowing, for instance, to specify that a particular number of times a message should be received in order to free the guard. Timeouts is another issue that could be considered, and policies to indicate how to conduct matching. We also intend to explore the advantages of that and other mechanisms can bring for the construction of truly concurrent systems, that is, based on preemptive multithreading.

Acknowledgements

This work was partially funded by CNPq Brasil.

References

[Benton et al. 2004] Benton, N., Cardelli, L., and Fournet, C. (2004). Modern concurrency abstractions for c#. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):769–804.

[de Moura 2004] de Moura, A. L. (2004). *Revisitando co-rotinas*. PhD thesis, Pontifícia Universidade Católica-PUC-Rio.

[Fournet and Gonthier 2000] Fournet, C. and Gonthier, G. (2000). 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.

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

[Guzev and Serdyuk 2003] Guzev, V. and Serdyuk, Y. (2003). 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.

[Ierusalimschy et al. 2007] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007). 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. ACM.

[Itzstein and Jasiunas 2003] Itzstein, S. and Jasiunas, M. (2003). 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.

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

[MB10 2010] MB10 (2010). Lua chords. Available at https://homepages.dcc.ufmg.br/~anolan/research/lchords:start. Visited on 06/09/2010.

[Milanés et al. 2010] Milanés, A., Rodriguez, N., and Ierusalimschy, R. (2010). Reflection-based language support for the heterogeneous capture and restoration of running computations. preprint (2010).

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