Referências Bibliográficas

[1] **ALua: Programação Paralela e Distribuída em Lua**. Website. http://alua.inf.puc-rio.br. 7

[2] **ejabberd – High-performance Enterprise Instant Messaging**. Website. http://www.process-one.net/en/ejabberd. 3.4

[3] **Yaws**. Website. http://yaws.hyber.org. 3.4

[4] AGHA, G.. **Actors: a model of concurrent computation in distributed systems**. MIT Press, Cambridge, MA, USA, 1986. 3.5

[5] ANDREWS, G. R.; SCHNEIDER, F. B.. **Concepts and Notations for Concurrent Programming**. ACM Comput. Surv., 15(1):3–43, 1983. 2

[6] ARMSTRONG, J.. **Erlang — a Survey of the Language and its Industrial Applications**. In: INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog, p. 16–18, Hino, Tokyo, Japan, 1996. 3.4

[7] ARMSTRONG, J.. **Programming Erlang**. Pragmatic Bookshelf, City, 2007. 3.4

[8] BENTON, N.; CARDELLI, L. ; FOURNET, C.. **Modern Concurrency Abstractions for C#**. In: ECOOP '02: PROCEEDINGS OF THE 16TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, p. 415–440, London, UK, 2002. Springer-Verlag. 3.2

[9] CAROMEL, D.; MATEU, L. ; TANTER, E.. **Sequential Object Monitors**. In: Odersky, M., editor, ECOOP 2004 - OBJECT-ORIENTED PROGRAMMING,18TH EUROPEAN CONFERENCE, volumen 3086 de **Lecture Notes in Computer Science**, p. 316–340, Oslo, Norway, 2004. Springer-Verlag. 3.1

[10] CHRYSANTHAKOPOULOS, G.; SINGH, S.. **An Asynchronous Messaging Library for C#**. Electronic Article, 2005. Synchronization and Concurrency in Object-Oriented Languages (SCOOL), OOPSLA 2005 Workshop, San Diego, California, USA. 3.3

[11] DIJKSTRA, E. W.. **Guarded commands, nondeterminacy and formal derivation of programs**. Commun. ACM, 18(8):453–457, 1975. 2

[12] DIJKSTRA, E. W.. **The structure of "THE" - multiprogramming system**. Commun. ACM, 26(1):49–52, 1983. 2

[13] FOURNET, C.; GONTHIER, G.. **The Join Calculus: A Language for Distributed Mobile Programming**. In: APPLIED SEMANTICS, INTERNATIONAL SUMMER SCHOOL, APPSEM 2000, CAMINHA, PORTUGAL, SEPTEMBER 9-15, 2000, ADVANCED LECTURES, p. 268–332, London, UK, 2002. Springer-Verlag. 3.2

[14] HALLER, P.; ODERSKY, M.. **Event-based Programming without Inversion of Control**. In: JMLC, p. 4–22, 2006. 3.5

[15] HALLER, P.; ODERSKY, M.. **Actors that Unify Threads and Events**. In: INTERNATIONAL CONFERENCE ON COORDINATION MODELS AND LANGUAGES, Lecture Notes in Computer Science (LNCS), 2007. 3.5

[16] HANSEN, P. B.. **A Programming Methodology for Operating System Design**. In: IFIP CONGRESS, p. 394–397, 1974. 2

[17] HERLIHY, M.; LUCHANGCO, V.; MARTIN, P. ; MOIR, M.. **Nonblocking memory management support for dynamic-sized data structures**. ACM Trans. Comput. Syst., 23(2):146–196, 2005. 2

[18] HOARE, C. A. R.. **Towards a theory of parallel programming**. Operating System Techniques, p. 61–71, 1972. 2

[19] HOARE, C. A. R.. **Monitors: an operating system structuring concept.** Technical report, Stanford, CA, USA, 1973. 2

[20] IEEE. **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, 1995. 2

[21] IERUSALIMSCHY, R.. **Programming in Lua, Second Edition**. Lua.org, 2006. 5.1

[22] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; FILHO, W. C.. **Lua – an Extensible Extension Language**. Software – Practice and Experience, 26(6):635–652, 1996. 1

[23] JOHANSSON, E.; SAGONAS, K. ; WILHELMSSON, J.. **Heap architectures for concurrent languages using message passing**. SIGPLAN Not., 38(2 supplement):88–99, 2003. 6.4

[24] LEE, E. A.. **The Problem with Threads**. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006. 2

[25] NEHAB, D.. **Luasocket: Network support for the Lua language**. Website, 2007. http://www.tecgraf.puc-rio.br/luasocket. 5.5

[26] NYSTROM, J. H.; TRINDER, P. W. ; KING, D. J.. **Are High-level Languages Suitable for Robust Telecoms Software?** In: Winther, R.; Gran, B. A. ; Dahll, G., editors, SAFECOMP, volumen 3688 de **Lecture Notes in Computer Science**, p. 275–288. Springer, 2005. 7

[27] OUSTERHOUT, J.. **Why Threads Are a Bad Idea (for most purposes)**. Presentation given at the 1996 Usenix Annual Technical Conference, January, January 1996. 2

[28] SERRANO, M.; BOUSSINOT, F. ; SERPETTE, B.. **Scheme Fair Threads**. In: PPDP '04: PROCEEDINGS OF THE 6TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING, p. 203–214, New York, NY, USA, 2004. ACM. 4

[29] SUTTER, H.. **A Fundamental Turn Toward Concurrency in Software**. Dr. Dobb's Journal, 30(3), 2005. 1

[30] SUTTER, H.; LARUS, J.. **Software and the concurrency revolution**. ACM Queue, 3(7):54–62, 2005. 1

[31] URURAHY, C. D.; DE LA ROCQUE RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Alua: flexibility for parallel programming**. Computer Languages, 28(2):155–180, 2002. 5.5

[32] VARELA, C.; AGHA, G.. **Programming dynamically reconfigurable open systems with SALSA**. SIGPLAN Not., 36(12):20–34, 2001. 3.5

[33] VON BEHREN, R.; CONDIT, J. ; BREWER, E.. **Why events are a bad idea (for high-concurrency servers)**. In: HOTOS'03: PROCEEDINGS OF THE 9TH CONFERENCE ON HOT TOPICS IN OPERATING SYSTEMS, p. 4–4, Berkeley, CA, USA, 2003. USENIX Association. 3.5

[34] ZHANG, Y.. **Non-blocking Synchronization: Algorithms and Performance Evaluation**. PhD thesis, Chalmers University of Technology, 2003. 2

A
API da Biblioteca para Programação Concorrente em Lua

```
**************
* PARENT API *
**************

-- Create a new lua process
-- Returns true if sucessful or nil, error_message if failed
luaproc.newproc( <string lua_code> )

-- Create a new worker (pthread)
-- Returns true if sucessful or nil, error_message if failed
luaproc.createworker( <void> )

-- Destroy a  worker (pthread)
-- Returns true if sucessful or nil, error_message if failed
luaproc.destroyworker( <void> )

-- Synchronize workers (pthreads) and exit after
-- all lua processes have ended
-- No return, finishes execution.
luaproc.exit( <void> )

-- Set maximum lua processes that should be
-- recycled (default = 0)
-- Returns true if sucessful or nil, error_message if failed
luaproc.recycle( <int maxrecycle> )

************************************************************
* CHILD API                                                *
* Available only to processes spawned                      *
* with luaproc.newproc                                     *
************************************************************
```

```
-- Create a new lua process
-- Returns true if sucessful or nil, error_message if failed
luaproc.newproc( <string lua_code> )

-- Create a new worker (pthread)
-- Returns true if sucessful or nil, error_message if failed
luaproc.createworker( <void> )

-- Destroy a worker (pthread)
-- Returns true if sucessful or nil, error_message if failed
luaproc.destroyworker( <void> )

-- Send a message on a channel
-- Returns true if sucessful or nil, error_message if failed
-- Results in blocking if there is no matching receive
luaproc.send( <string channel_name>, <string msg1>,
[string msg2], [string msg3], ... )

-- Receive a message on a channel
-- Returns message string(s) if sucessful or
-- nil, error_message if failed
-- Results in blocking if there is no matching send
-- and the asynchronous flag is not set (nil) or set to false
luaproc.receive( <string channel_name>, [boolean asynchronous] )

-- Create a new channel
-- Returns true if sucessful or nil, error_message if failed
luaproc.newchannel( <string channel_name> )

-- Destroy a channel
-- Returns true if sucessful or nil, error_message if failed
luaproc.delchannel( <string channel_name> )

-- Set maximum lua processes that should be
-- recycled (default = 0)
-- Returns true if sucessful or nil, error_message if failed
luaproc.recycle( <int maxrecycle> )

<> = mandatory arguments
```

```
[] = optional arguments
```

B
Exemplo Simples de Serviço de Escalonamento Remoto

B.1
Servidor

```
1   require "socket"
2   require "luaproc"
3
4   -- server host/ip
5   shost = arg[1] or "127.0.0.1"
6
7   -- server port
8   sport = arg[2] or 3133
9
10  -- create a TCP socket and bind it to the specified
11  -- (or local) host at specified (or default) port
12  local server = assert( socket.bind( shost, sport ))
13
14  -- print a message informing what's up
15  print( "[luaproc scheduler listening on " ..
16    shost .. " port " .. sport .. "]" )
17
18  -- loop forever waiting for job submissions
19  while 1 do
20
21    -- wait for a connection from any client
22    local client = server:accept()
23
24    local cip, cport = client:getpeername( )
25    print( "-> connection from " .. cip .. ":" .. cport )
26
27    -- receive lua code
28    local codestr, err = client:receive( "*a" )
```

```
29
30    -- if there was no error, create new luaproc
31    if not err and codestr then
32      print( "-> creating new lua process" )
33      luaproc.newproc( codestr )
34    end
35
36    -- done with client, close the object
37    client:close()
38
39  end
40
41  luaproc.exit()
```

B.2 Cliente

```
1   require "socket"
2   require "luaproc"
3
4   -- file containing lua code
5   if ( not arg[1] ) then
6     print( "usage: " .. arg[0] ..
7       " <code.lua> [hostname|ip] [port]" )
8     return
9   end
10  codefilename = arg[1]
11
12  -- open and read file with lua code
13  codefh, err  = io.open( codefilename )
14  if ( err ) then
15    print( "error reading file " ..
16      codefilename .. " -> " .. err )
17    return
18  end
19  codestr = codefh:read( "*a" )
20  codefh:close()
21
22  -- hostname/ip to connect to
23  host = arg[2] or "127.0.0.1"
```

```
24
25  -- port to connect to
26  port = arg[3] or 3133
27
28  -- create a TCP socket and connect it to the specified
29  -- (or local) host at specified (or default) port
30  local client = assert( socket.connect( host, port ))
31
32  -- print a message informing what's up
33  print( "-> connected to " .. host .. ":" .. port )
34
35  -- send lua code
36  local lastbyte, err = client:send( codestr )
37
38  if ( err ) then
39    print( "error sending Lua code -> " .. err )
40  else
41    print( "-> lua code sent" )
42  end
43
44  -- done with client, close the object
45  client:close()
```

C
Disparador de Processos Lua com Sincronização

```
1   require "luaproc"
2
3   if ( not arg[1] ) then
4     print( "usage: lua " ..arg[0].. " <num_procs>" )
5     return
6   end
7
8   -- create additional worker thread
9   luaproc.createworker()
10
11  -- create master process
12  luaproc.newproc( [=[
13
14  -- total processes to spawn
15  nproc = ]=] .. arg[1] .. [=[
16
17  -- create a channel for each process
18  for i = 1, nproc, 1
19  do
20    luaproc.newchannel( "channel" .. i )
21  end
22
23  -- spawn processes
24  for i = 1, nproc, 1
25  do
26    luaproc.newproc( [[
27      -- receive message from master process
28      luaproc.receive( "channel]] .. i .. [[" )
29    ]] )
30  end
31
```

```
32  -- send a message to each process
33  for i = 1, nproc, 1
34  do
35    luaproc.send( "channel" .. i, "die" )
36  end
37
38  ]=] )
39
40  -- wait until all processes have ended
41  luaproc.exit()
```

D
Envio e Recebimento Simples de Mensagens

```
1   require "luaproc"
2
3   if (( not arg[1] ) or ( not arg[2] )) then
4     print( "usage: lua " .. arg[0] ..
5       " <transmissions> <msg_file>" )
6     return
7   end
8
9   -- open and read msg file
10  msgfile, err  = io.open( arg[2] )
11  if ( err ) then
12    print( "error reading file " .. arg[2] ..
13      " -> " .. err )
14    return
15  end
16  msg = msgfile:read( "*a" )
17  msgfile:close()
18
19  -- scrub msg string
20  msg = string.gsub( msg, "(%W)", "%%%1" )
21
22  -- create additional worker thread
23  luaproc.createworker()
24
25  -- create master process
26  luaproc.newproc( [=[
27
28  -- create a channel
29  luaproc.newchannel( 'achannel' )
30
31  -- create receiver process
```

```
32  luaproc.newproc( [[
33    -- receive messages
34    for turn = 1,]=] .. arg[1] .. [=[, 1 do
35      msg = luaproc.receive( 'achannel' )
36    end
37  ]] )
38
39  -- send messages
40  for turn = 1,]=] .. arg[1] .. [=[, 1 do
41    luaproc.send( 'achannel', [[]=] .. msg .. [=[]] )
42  end
43  ]=] )
44
45  luaproc.exit()
```

E
Anel de Mensagens

```
1   require "luaproc"
2
3   if (( not arg[1] ) or ( not arg[2] )) then
4     print( "usage: lua " .. arg[0] ..
5       " <number_of_peers> <number_of_messages>" )
6     return
7   end
8
9   -- create additional worker thread
10  luaproc.createworker()
11
12  -- create master process
13  luaproc.newproc( [=[
14
15  -- total peers in ring
16  totalpeers = ]=] .. arg[1] .. [=[
17
18  -- total messages to cycle the ring
19  totalmsgs = ]=] .. arg[2] .. [=[
20
21  -- create a channel for each peer
22  for i = 1, totalpeers, 1
23  do
24    luaproc.newchannel( 'peer' .. i )
25  end
26  luaproc.newchannel( 'master' )
27
28  -- create peers
29  for i = 1, totalpeers, 1
30  do
31    luaproc.newproc( [[
```

```
32      me   = "]] .. i .. [["
33      next = "]] .. ((i % totalpeers) + 1) .. [["
34      -- send initial synchronization msg
35      luaproc.send( 'master', 'peer ' ..
36        me .. ' is alive' )
37      -- send message to neighbour peer
38      for turn = 1,]] .. totalmsgs .. [[, 1 do
39        msg = luaproc.receive( 'peer' .. me )
40        luaproc.send( 'peer' .. next, msg )
41      end
42      -- first peer must notify ring is over
43      if ( me == '1' ) then
44        msg = luaproc.receive( 'peer' .. me )
45        luaproc.send( 'master', 'ring over' )
46      end
47    ]] )
48  end
49
50  -- synchronize peers
51  for i = 1, totalpeers, 1
52  do
53    luaproc.receive( 'master' )
54  end
55
56  -- send initial msg
57  luaproc.send( 'peer1', '1' )
58
59  -- receive final msg
60  luaproc.receive( 'master' )
61  ]=] )
62
63  luaproc.exit()
```

F
Disparador de Processos Lua sem Sincronização

```
1   require "luaproc"
2
3   if ( not arg[1] ) then
4     print( "usage: lua " .. arg[0] ..
5       " <recycle_limit> <num_procs>" )
6     return
7   end
8
9   -- create additional worker thread
10  luaproc.createworker()
11
12  luaproc.recycle( arg[1] )
13
14  for i = 1, arg[2], 1
15  do
16    luaproc.newproc( [[
17      print( 'hello from luaproc ]] .. i .. [[' )
18    ]] )
19  end
20
21  luaproc.exit()
```

G
Disparador de Processos Lua com Retardos

```
1   require "luaproc"
2
3   if ( not arg[1] ) then
4     print( "usage: lua " ..arg[0].. " <num_procs>" )
5     return
6   end
7
8   -- create additional worker thread
9   luaproc.createworker()
10
11  -- create master process
12  luaproc.newproc( [=[
13
14  -- delay function
15  function delay()
16    --do nothing for a while
17    print( "----- delay start -----" )
18    for i = 1, 500000000, 1 do end
19    print( "----- delay end   -----" )
20  end
21
22  -- total processes to spawn
23  nproc = ]=] .. arg[1] .. [=[
24
25  -- delay to measure memory consumption
26  -- before any creation
27  delay()
28
29  -- create a channel for each process
30  for i = 1, nproc, 1
31  do
```

```
32    luaproc.newchannel( "channel" .. i )
33  end
34
35  -- delay to measure memory consumption
36  -- after creating channels
37  delay()
38
39  -- spawn processes
40  for i = 1, nproc, 1
41  do
42    luaproc.newproc( [[
43      -- receive message from master process
44      luaproc.receive( "channel]] .. i .. [[" )
45    ]] )
46  end
47
48  -- delay to measure memory consumption
49  -- after lua processes (and channels)
50  -- have been created
51  delay()
52
53  -- send a message to each process
54  for i = 1, nproc, 1
55  do
56    luaproc.send( "channel" .. i, "die" )
57  end
58
59  ]=] )
60
61  -- wait until all processes have ended
62  luaproc.exit()
```

H
Busca de Cadeias de Caracteres

H.1
Versão Processos Lua

H.1.1
Inicializador

```
1   require "luaproc"
2
3   -- scheduler worker threads
4   schedwt = 6
5
6   -- string searching worker
7   -- lua processes
8   workers = schedwt - 1
9   if ( workers <= 0 ) then
10    workers = 1
11  end
12
13
14  -- check for patterns filename
15  -- check for search target filenames
16  if (( arg[1] == nil ) or ( arg[2] == nil )) then
17    print( "usage: " .. arg[0] ..
18      " <patterns_file> <target_file(s)>" )
19    return
20  end
21
22  -- create scheduler worker threads
23  -- (one worker is already created
24  -- by default)
25  for i = 1, (schedwt - 1), 1
26  do
```

```
27    luaproc.createworker()
28  end
29
30  -- read search target files into single string
31  targetstr = table.concat( arg, ";", 2 ) .. ";"
32
33  -- initialize the environment:
34  --  * create a channel for the master (coordinator)
35  --    process
36  --  * create a channel for result transmission
37  --  * create channels for each worker
38  --  * spawn the master process
39  --  * send a message to the master with the patterns
40  --    filename
41  --  * send a message to the master with the search
42  --    target filename(s)
43  --  * send a message to the master with the number of
44  --    workers available
45  luaproc.newproc( [=[
46    luaproc.newchannel( "master" )
47    luaproc.newchannel( "result" )
48    for i = 1, ]=] .. workers .. [=[, 1
49    do
50      luaproc.newchannel( "worker" .. i )
51    end
52    luaproc.newproc( "loadfile( 'master.lua' )()" )
53    for i = 1, ]=] .. workers .. [=[, 1
54    do
55      luaproc.newproc( "loadfile( 'worker.lua' )()(" ..
56        i .. ")" )
57    end
58    luaproc.send( "master", "]=] .. arg[1] .. [=[" )
59    luaproc.send( "master", "]=] .. targetstr .. [=[" )
60    luaproc.send( "master", "]=] .. workers .. [=[" )
61  ]=] )
62
63  luaproc.exit()
```

H.1.2 Mestre

```
1   require "io"
2   require "os"
3   require "string"
4   require "table"
5
6   print( "master -> ready " )
7
8   -- write matches to file
9   outfile = io.open( "matches.txt", "w" )
10
11  -- receive patterns filename, search target filename(s)
12  -- and number of available workers
13  fpatterns  = luaproc.receive( "master" )
14  fsearch    = luaproc.receive( "master" )
15  numworkers = luaproc.receive( "master" )
16
17  print( "master -> read patterns from file " .. fpatterns )
18  print( "master -> target files = " .. fsearch  )
19  print( "master -> available workers = " .. numworkers  )
20
21  searchfiles = {}
22  for f in string.gmatch( fsearch, "([^;]*);" ) do
23    print( "master -> search file " .. f )
24    table.insert( searchfiles, f )
25  end
26
27  -- open patterns file
28  fh, err = io.open( fpatterns, "r" )
29  if ( not fh ) then
30    print( "master -> error opening file - " .. err )
31    return
32  end
33
34  -- read patterns into a table
35  patterns = {}
36  numlines = 0
37  for line in fh:lines( )
```

```
38  do
39    nomagicline = string.gsub( line, "(%W)", "%%%1" )
40    table.insert( patterns, nomagicline )
41    numlines = numlines + 1
42  end
43
44  fh:close( )
45
46  -- generate a code to indicate to workers that
47  -- all files have been processed
48  workdone = os.tmpname()
49
50  -- send work done code to workers
51  for i = 1, numworkers, 1
52  do
53    luaproc.send( "worker" .. i, workdone )
54    print( "master -> sent work done code to worker " .. i )
55  end
56
57  -- send number of patterns read to workers
58  for i = 1, numworkers, 1
59  do
60    luaproc.send( "worker" .. i, numlines )
61    print( "master -> sent pattern count to worker " .. i )
62  end
63
64  -- send each pattern to each worker
65  for i = 1, numworkers, 1
66  do
67    for j = 1, numlines, 1
68    do
69      luaproc.send( "worker" .. i, patterns[j] )
70    end
71    print( "master -> sent patterns to worker " .. i )
72  end
73
74  -- calculate initial job count
75  if ( tonumber( numworkers ) < table.getn( searchfiles )) then
76    initialjobs = numworkers
```

```
77  else
78    initialjobs = table.getn( searchfiles )
79  end
80
81  -- sequentially distribute initial jobs
82  for i = 1, initialjobs, 1
83  do
84    luaproc.send( "worker" .. i, searchfiles[i] )
85    print( "master -> sent initial filename to worker " .. i )
86  end
87
88  -- distribute remaining jobs according to worker demand
89  for i = (initialjobs + 1), table.getn( searchfiles ), 1
90  do
91    workerdone = luaproc.receive( "master" )
92    matches = luaproc.receive( "result" )
93    print( "master -> received results from worker " ..
94      workerdone )
95    if ( matches ~= "" ) then
96      print( "master -> write " .. string.len( matches ) ..
97        " bytes to results file" )
98      outfile:write( matches .. "\n" )
99      outfile:flush( )
100   else
101     print( "master -> empty result set" )
102   end
103   luaproc.send( "worker" .. workerdone, searchfiles[i] )
104   print( "master -> sent search filename to worker " ..
105     workerdone )
106 end
107
108 -- receive remaining results
109 for i = 1, initialjobs, 1
110 do
111   workerdone = luaproc.receive( "master" )
112   matches = luaproc.receive( "result" )
113   print( "master -> received results from worker " ..
114     workerdone )
115   if ( matches ~= "" ) then
```

```
116       outfile:write( matches .. "\n" )
117       outfile:flush( )
118     end
119   end
120
121   -- notify workers that all work is done
122   for i = 1, numworkers, 1
123   do
124     print( "master -> sending work done code to worker " .. i )
125     luaproc.send( "worker" .. i, workdone )
126     print( "master -> sent work done code to worker " .. i )
127   end
128
129   -- remove temporary file whose name was
130   -- used as work done code
131   os.remove( workdone )
132
133   print( "master -> done" )
134
135   -- close results file
136   outfile:close( )
```

H.1.3 Trabalhador

```
1   require "io"
2   require "string"
3   require "table"
4
5   function worker( id )
6
7     print( "worker " .. id .. " -> ready" )
8
9     -- receive the work done code
10    workdone = luaproc.receive( "worker" .. id )
11    print( "worker " .. id .. " -> work done code is " ..
12      workdone )
13
14    -- receive the total number of patterns
15    numpatterns = luaproc.receive( "worker" .. id )
```

```
16    print( "worker " .. id .. " -> total patterns = " ..
17      numpatterns )
18
19    -- receive patterns into a table
20    patterns = {}
21    for i = 1, numpatterns, 1
22    do
23      table.insert( patterns,
24        luaproc.receive( "worker" .. id ))
25      print( "worker " .. id .. " -> received pattern " .. i )
26    end
27
28    -- receive search target filename
29    filename = luaproc.receive( "worker" .. id )
30
31    -- work until work done code is received
32    while( filename ~= workdone ) do
33
34      print( "worker " .. id .. " -> received filename " ..
35        filename )
36
37      -- initially matches table is empty
38      matches = {}
39
40      -- open search target file
41      fh, err = io.open( filename, "r" )
42      if ( not fh ) then
43        print( "worker " .. id .. " -> " ..err )
44        print( "worker " .. id .. " -> error opening file" )
45        print( "worker " .. id .. " -> aborting" )
46        return
47      end
48
49      print( "worker " .. id .. " -> start searching in " ..
50        filename )
51
52      -- check for pattern matches
53      for line in fh:lines( )
54      do
```

```
55        for _,p in pairs( patterns )
56        do
57          if ( string.match( line, p )) then
58            table.insert( matches, line )
59          end
60        end
61      end
62
63      fh:close( )
64
65      print( "worker " .. id .. " -> end searching in file " ..
66        filename )
67      print( "worker " .. id .. " -> found " ..
68        table.getn( matches ) .. " matches" )
69
70      -- send this worker's id to master
71      luaproc.send( "master", id )
72
73      -- send this results to master
74      luaproc.send( "result", table.concat( matches, "\n" ))
75
76      -- receive another search target filename
77      filename = luaproc.receive( "worker" .. id )
78
79    end
80
81    print( "worker " .. id .. " -> done" )
82
83  end
84
85  return worker
```

H.2
Versão Lua Simples

```
1   -- check for patterns filename
2   -- check for search target filenames
3   if (( arg[1] == nil ) or ( arg[2] == nil )) then
4     print( "usage: " .. arg[0] ..
5       " <patterns_file> <target_file(s)>" )
```

```
6     return
7   end
8
9   -- open patterns file
10  fh, err = io.open( arg[1], "r" )
11  if ( not fh ) then
12    print( "[master] error opening file - " .. err )
13    return
14  end
15
16  print( "started execution" )
17
18  -- read patterns into a table
19  patterns = {}
20  for line in fh:lines( )
21  do
22    nomagicline = string.gsub( line, "(%W)", "%%%1" )
23    table.insert( patterns, nomagicline )
24  end
25
26  fh:close( )
27
28  outfile = io.open( "matches_single.txt", "w" )
29
30  for i = 2, table.getn( arg ), 1
31  do
32    fh, err = io.open( arg[i], "r" )
33    if ( not fh ) then
34      print( "error opening file - " .. err )
35      return
36    end
37    print( "opened file " .. arg[i] )
38    matches = {}
39    for line in fh:lines( )
40    do
41      for _,p in pairs( patterns )
42      do
43        if ( string.match( line, p )) then
44          table.insert( matches, line )
```

```
45          end
46        end
47      end
48      fh:close( )
49      if ( #matches > 0 ) then
50        outfile:write( table.concat( matches, "\n" ) .. "\n" )
51        outfile:flush( )
52      end
53    end
54
55    outfile:close( )
56
57    print( "finished execution" )
```

I
Disparador de Pthreads sem Sincronização

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <pthread.h>
4
5   /* thread main */
6   void *thread_main( void *args ) {
7     /* say hello */
8     printf( "hello from thread %ld\n", (long )args + 1 );
9     /* exit */
10    pthread_exit( NULL );
11  }
12
13  /* main program main */
14  int main( int argc, char **argv ) {
15
16    long int i;
17    int maxthreads;
18    pthread_t athread;
19    pthread_attr_t tattr;
20
21    if ( argc < 2 ) {
22      fprintf( stderr, "usage: %s <maxthreads>\n", argv[0] );
23      return -1;
24    }
25
26    /* created threads detached so
27    resources are freed immediately
28    after thread exit */
29    pthread_attr_init( &tattr );
30    pthread_attr_setdetachstate( &tattr,
31      PTHREAD_CREATE_DETACHED );
```

```
32
33      /* convert command line arg to int */
34      maxthreads = atoi( argv[1] );
35
36      /* create threads */
37      for ( i = 0; i < maxthreads; i++ ) {
38        if ( pthread_create( &athread, &tattr, thread_main,
39          (void *)i ) != 0 ) {
40          fprintf( stderr, "error creating thread %ld.\n", i );
41          return -1;
42        }
43      }
44
45      /* wait for threads and exit */
46      pthread_exit( NULL );
47
48      return 0;
49
50    }
```

J
Disparador de Pthreads com Sincronização

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <pthread.h>
4   #include <unistd.h>
5
6   /* globals */
7   pthread_cond_t cond_main = PTHREAD_COND_INITIALIZER;
8   pthread_cond_t cond_threads = PTHREAD_COND_INITIALIZER;
9   pthread_mutex_t mutex_main = PTHREAD_MUTEX_INITIALIZER;
10  pthread_mutex_t mutex_threads = PTHREAD_MUTEX_INITIALIZER;
11  int threadcount = 0;
12
13  /* thread argument struct */
14  typedef struct thread_arg_st {
15    int tid;
16    int maxthreads;
17  } thread_args;
18
19  /* thread main routine */
20  void *thread_main( void *args ) {
21    thread_args *targ = (thread_args *)args;
22    /* wait until all threads have been created */
23    pthread_mutex_lock( &mutex_threads );
24    threadcount++;
25    if ( threadcount >= targ->maxthreads ) {
26      printf( "all threads have been created, sleeping...\n" );
27      fflush( stdout );
28      pthread_cond_signal( &cond_main );
29    }
30    while ( threadcount < targ->maxthreads ) {
31      pthread_cond_wait( &cond_threads, &mutex_threads);
```

```
32      }
33      pthread_mutex_unlock( &mutex_threads );
34      /* exit */
35      pthread_exit( NULL );
36    }
37
38    /* main program main routine */
39    int main( int argc, char **argv ) {
40
41      int i;
42      int max;
43      pthread_t athread;
44      thread_args *targ;
45
46      if ( argc < 2 ) {
47        fprintf( stderr, "usage: %s <maxthreads>\n", argv[0] );
48        return -1;
49      }
50
51      max = atoi( argv[1] );
52      targ = (thread_args *)malloc( max * sizeof( thread_args ));
53
54      /* initial sleep */
55      printf( "initial sleep\n" );
56      fflush( stdout );
57      usleep( 5000000 );
58
59      /* create threads */
60      for ( i = 0; i < max; i++ ) {
61        targ[i].tid = (i + 1);
62        targ[i].maxthreads = max;
63        if ( pthread_create( &athread, NULL, thread_main,
64          &targ[i] ) != 0 ) {
65          fprintf( stderr, "error creating thread %d.\n", i );
66          return -1;
67        }
68      }
69
70      /* wait until all threads have been created */
```

```
71      pthread_mutex_lock( &mutex_threads );
72      pthread_cond_wait( &cond_main, &mutex_threads);
73      pthread_mutex_unlock( &mutex_threads );
74
75      /* after thread creation sleep */
76      usleep( 5000000 );
77
78      /* notify all threads that they can exit */
79      pthread_mutex_lock( &mutex_threads );
80      pthread_cond_broadcast( &cond_threads );
81      pthread_mutex_unlock( &mutex_threads );
82
83      pthread_exit( NULL );
84
85      free( targ );
86
87      return 0;
88
89  }
```

K
Disparador de Processos Erlang

K.1
Versão Interpretada

```
1   % Processes
2   %  Create N processes then destroy them
3   %
4   %  (Adapted from Programming Erlang - Software
5   %   for a Concurrent World - Joe Armstrong -
6   %   Chapter 8 - page 141)
7
8   main( N ) ->
9     % convert command line argument to integer
10    [Nhead|_] = N,
11    {Max,_} = string:to_integer( Nhead ),
12    % spawn N processes which will wait for a message
13    L = for( 1, Max,
14      fun() -> spawn( fun() -> wait() end ) end ),
15    % send a message to each process in order to finish it
16    lists:foreach( fun( Pid ) -> Pid ! die end, L ).
17
18  % child processes code
19  % wait for a message then
20  % exit after receiving it
21  wait() ->
22    receive
23      die -> void
24    end.
25
26  % for recursion construction
27  for( E, E, F ) -> [ F( ) ];
28  for( B, E, F ) -> [ F( ) | for( B + 1, E, F )].
```

K.2
Versão Compilada

```
1  % Processes
2  %  Create N processes then destroy them
3  %
4  %  (Adapted from Programming Erlang - Software
5  %   for a Concurrent World - Joe Armstrong -
6  %   Chapter 8 - page 141)
7  -module( cprocesses ).
8  -export( [start/1] ).
9
10 start( [MaxAtom] ) ->
11   % convert command line argument to integer
12   Max = erlang:list_to_integer(
13     erlang:atom_to_list( MaxAtom )),
14   % spawn N processes which will wait for a message
15   L = for( 1, Max, fun() ->
16     spawn( fun() -> wait() end ) end ),
17   % send a message to each process in order to finish it
18   lists:foreach( fun( Pid ) -> Pid ! die end, L ).
19
20 % child processes code
21 % wait for a message then
22 % exit after receiving it
23 wait() ->
24   receive
25     die -> void
26   end.
27
28 % for recursion construction
29 for( E, E, F ) -> [ F( ) ];
30 for( B, E, F ) -> [ F( ) | for( B + 1, E, F )].
```

L
Envio e Recebimento Simples de Mensagens em Erlang

L.1
Versão Interpretada

```
1   % SendRecv
2   %  Read message from file F[1] and
3   %  transmit it a number of times (F[2])
4   %
5
6   main( F ) ->
7     % parse command line arguments
8     [Fhead,MsgFilename|_] = F,
9     {Transmissions,_} = string:to_integer( Fhead ),
10    io:format( "message file = ~p, transmissions = ~p~n",
11      [MsgFilename, Transmissions] ),
12    % spawn sender and receiver processes
13    SenderPid = spawn( fun() -> sender() end ),
14    ReceiverPid = spawn( fun() -> receiver() end ),
15    % send parent id to receiver process
16    ReceiverPid ! { parent, self() },
17    % send transmission count to receiver process
18    ReceiverPid ! { tcount, Transmissions },
19    % send transmission count to sender process
20    SenderPid ! { tcount, Transmissions },
21    % send receiver pid to sender process
22    SenderPid ! { recvpid, ReceiverPid },
23    % send msg filename to sender process
24    SenderPid ! { msgfile, MsgFilename },
25    % wait for final message from receiver
26    receive
27      die -> void
28    end,
```

```
29    io:format( "finished~n", [] ).
30
31  % sender process
32  sender() ->
33    % receive transmission count from parent
34    receive
35      { tcount, TransmissionCount } -> void
36    end,
37    % receive receiver pid from parent
38    receive
39      { recvpid, RecvPid } -> void
40    end,
41    % receive message filename from parent
42    receive
43      { msgfile, MsgFile } -> void
44    end,
45    % read message file contents
46    {_,MsgData} = file:read_file( MsgFile ),
47    % send contents to receiver process
48    for( 1, TransmissionCount, fun() ->
49      RecvPid ! { msg, MsgData }
50    end ).
51
52  % receiver process
53  receiver() ->
54    % receive parent id from parent
55    receive
56      { parent, ParentId } -> void
57    end,
58    % receive transmission count from parent
59    receive
60      { tcount, TransmissionCount } -> void
61    end,
62    % receive message from sender
63    for( 1, TransmissionCount, fun() ->
64      receive
65        { msg, _ } -> void
66      end
67    end ),
```

```
68      ParentId ! die.
69
70    % for recursion construction
71    for( E, E, F ) -> [ F( ) ];
72    for( B, E, F ) -> [ F( ) | for( B + 1, E, F )].
```

L.2
Versão Compilada

```
1     % SendRecv
2     %  Read message from file FileAtom and
3     %  transmit it a number of times
4     %  (TransmissionAtom)
5     %
6     -module( csendrecv ).
7     -export( [start/1] ).
8
9     start( [TransmissionsAtom,MsgFile] ) ->
10      % parse command line arguments
11      Transmissions = erlang:list_to_integer(
12        erlang:atom_to_list( TransmissionsAtom )),
13      MsgFilename = MsgFile,
14      io:format( "message file = ~p, transmissions = ~p~n",
15        [MsgFilename, Transmissions] ),
16      % spawn sender and receiver processes
17      SenderPid = spawn( fun() -> sender() end ),
18      ReceiverPid = spawn( fun() -> receiver() end ),
19      % send parent id to receiver process
20      ReceiverPid ! { parent, self() },
21      % send transmission count to receiver process
22      ReceiverPid ! { tcount, Transmissions },
23      % send transmission count to sender process
24      SenderPid ! { tcount, Transmissions },
25      % send receiver pid to sender process
26      SenderPid ! { recvpid, ReceiverPid },
27      % send msg filename to sender process
28      SenderPid ! { msgfile, MsgFilename },
29      % wait for final message from receiver
30      receive
31        die -> void
```

```
32      end,
33      io:format( "finished~n", [] ).
34
35    % sender process
36    sender() ->
37      % receive transmission count from parent
38      receive
39        { tcount, TransmissionCount } -> void
40      end,
41      % receive receiver pid from parent
42      receive
43        { recvpid, RecvPid } -> void
44      end,
45      % receive message filename from parent
46      receive
47        { msgfile, MsgFile } -> void
48      end,
49      % read message file contents
50      {_,MsgData} = file:read_file( MsgFile ),
51      % send contents to receiver process
52      for( 1, TransmissionCount, fun() ->
53        RecvPid ! { msg, MsgData }
54      end ).
55
56    % receiver process
57    receiver() ->
58      % receive parent id from parent
59      receive
60        { parent, ParentId } -> void
61      end,
62      % receive transmission count from parent
63      receive
64        { tcount, TransmissionCount } -> void
65      end,
66      % receive message from sender
67      for( 1, TransmissionCount, fun() ->
68        receive
69          { msg, _ } -> void
70        end
```

```
71      end ),
72      ParentId ! die.
73
74    % for recursion construction
75    for( E, E, F ) -> [ F( ) ];
76    for( B, E, F ) -> [ F( ) | for( B + 1, E, F )].
```

M
Anel de Mensagens em Erlang

M.1
Versão Interpretada

```
1   % Ring
2   %  Spawn P[1] peers and cycle
3   %  P[2] messages around sequentially
4   %
5   %  usage: escript ring.erl <peers> <messages>
6   %
7
8   main( P ) ->
9     % convert command line arguments to integers
10    [Pfirst,Psecond|_] = P,
11    {Peers,_} = string:to_integer( Pfirst ),
12    {Messages,_} = string:to_integer( Psecond ),
13    % get parent (self) id
14    ParentId = self(),
15    io:format( "-master- ring start (~p x ~p)~n",
16      [Peers, Messages] ),
17    % spawn peers
18    FirstPeer = spawn( fun() ->
19      peer( 1, Peers, Messages, ParentId )
20    end ),
21    % send first peer id to last peer
22    receive
23      { lastpeer, LastPeer } -> void
24    end,
25    LastPeer ! { parent, FirstPeer },
26    % send a message to the ring
27    FirstPeer ! { ringmsg, 1 },
28    % ring end
```

```
29    receive
30      { lastpeer, ringover } -> void
31    end,
32    io:format( "-master- ring end~n", [] ).
33
34  % recursive peer code
35  % last peer
36  peer( Peers, Peers, Messages, Parent ) ->
37    % last peer should receive first peer's id
38    Parent ! { lastpeer, self() },
39    receive
40      { parent, NextPeer } -> void
41    end,
42    % execute ring
43    ring( 1, Messages, NextPeer ),
44    % notify parent that ring is over
45    Parent ! { lastpeer, ringover };
46
47  % spawn peers and execute ring
48  peer( PeerCounter, Peers, Messages, Parent ) ->
49    NextPeer = spawn( fun() ->
50      peer( PeerCounter + 1, Peers, Messages, Parent )
51    end ),
52    ring( 1, Messages, NextPeer ).
53
54  % recursive ring code
55  % forward last message and return
56  ring( Messages, Messages, NextPeer ) ->
57    receive
58      { ringmsg, Msg } -> void
59    end,
60    NextPeer ! { ringmsg, Msg };
61
62  % forward messages and increase counter
63  ring( MessageCounter, Messages, NextPeer ) ->
64    receive
65      { ringmsg, Msg } -> void
66    end,
67    NextPeer ! { ringmsg, Msg },
```

```
68      ring( MessageCounter + 1, Messages, NextPeer ).
```

M.2 Versão Compilada

```
1   % Ring
2   %  Spawn PeersAtom peers and cycle
3   %  MessagesAtom messages around sequentially
4   %
5   %  usage: escript ring.erl <peers> <messages>
6   %
7   -module( cnewring ).
8   -export( [start/1] ).
9
10  start( [PeersAtom,MessagesAtom] ) ->
11    % convert command line arguments to integers
12    Peers = erlang:list_to_integer(
13      erlang:atom_to_list( PeersAtom )),
14    Messages = erlang:list_to_integer(
15      erlang:atom_to_list( MessagesAtom )),
16    % get parent (self) id
17    ParentId = self(),
18    io:format( "-master- ring start (~p x ~p)~n",
19      [Peers, Messages] ),
20    % spawn peers
21    FirstPeer = spawn( fun() ->
22      peer( 1, Peers, Messages, ParentId )
23    end ),
24    % send first peer id to last peer
25    receive
26      { lastpeer, LastPeer } -> void
27    end,
28    LastPeer ! { parent, FirstPeer },
29    % send a message to the ring
30    FirstPeer ! { ringmsg, 1 },
31    % ring end
32    receive
33      { lastpeer, ringover } -> void
34    end,
35    io:format( "-master- ring end~n", [] ).
```

```
36
37  % recursive peer code
38  % last peer
39  peer( Peers, Peers, Messages, Parent ) ->
40    % last peer should receive first peer's id
41    Parent ! { lastpeer, self() },
42    receive
43      { parent, NextPeer } -> void
44    end,
45    % execute ring
46    ring( 1, Messages, NextPeer ),
47    % notify parent that ring is over
48    Parent ! { lastpeer, ringover };
49
50  % spawn peers and execute ring
51  peer( PeerCounter, Peers, Messages, Parent ) ->
52    NextPeer = spawn( fun() ->
53      peer( PeerCounter + 1, Peers, Messages, Parent )
54    end ),
55    ring( 1, Messages, NextPeer ).
56
57  % recursive ring code
58  % forward last message and return
59  ring( Messages, Messages, NextPeer ) ->
60    receive
61      { ringmsg, Msg } -> void
62    end,
63    NextPeer ! { ringmsg, Msg };
64
65  % forward messages and increase counter
66  ring( MessageCounter, Messages, NextPeer ) ->
67    receive
68      { ringmsg, Msg } -> void
69    end,
70    NextPeer ! { ringmsg, Msg },
71    ring( MessageCounter + 1, Messages, NextPeer ).
```