Experiments with multitasking and multithreading in LUATEX

Parallelism & Concurrency

> “A system is said concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously”
> [Breshears, 2009, p. 2])

Parallelism: Implicit & Explicit

- 8 processors p_0, p_1,...p_7;
- `for(int i=0; i<8;i++){a[i]=2*i;}`
- each `a[i]=2*i` on processor p_i;
- Speedup $S = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{8}{1} = 8$ (800% !)

Implicit parallelism

This kind of parallelism is easy to understand and it's *implicit* in the `for` cycle. Theoretically is the perfect solution: if *all* the code can run in parallel the end user see a speedup almost equal the number of the processors (the underling OS still needs at least a processor too) *without* change anything.

Implicit parallelism

OpenMP: (*Open MultiProcessing*) is a `C` library that offers some parallel constructs by means of the `#pragma` directives of `C`, so if a compiler doesn't support these directives it's still able to compile the code.

```
#pragma omp parallel for
for (i = 0; i < 8; i++){a[i]=2*i;}
```

'Decorate the code'.
It works under Linux/OS X/Windows (gcc, icc, Visual C; clang almost finished)

Nice but...

Implicit parallelism

Amdahl's law:

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{1}{(1 - F_e) + F_e / S_e}$$

T_{seq} execution time for the sequential version

T_{par} execution time for the parallel version of the program

F_e is the fraction of the original time in the sequential execution that can be converted in a parallel one

S_e the speed up that can be obtained if *all* the sequential code can be converted into a parallel one [Hennessy and Patterson, 2012, p. 46])

Implicit parallelism

8 processor: speed up max 100%.
To have a speed up $S = 4$ (i.e. 50% of the max speed up) is enough to rewrite 50% of the code ?
The Amdahl's law:

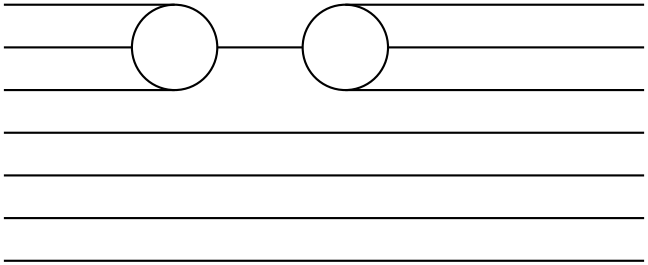
$$4 = \frac{1}{(1 - F_e) + F_e/8} \Longrightarrow F_e = 6/7 \approx 86\%$$

We have to rewrite 86% of the code: if we rewrite a 50% we have a speed up $S = 1.77$ i.e $\approx 22\%$ of the max speed up.

The Amdahl's law **is not linear on code !**

Implicit parallelism

Implicit parallesim is fragile: just a single (short) bottleneck degrades the performance:



Very likely you are always *under* 100% of max speed up.

Explicit parallelism & concurrency

The user creates and manages the tasks.

Pros: Very likely you are always *over* 100% of speed of the previous *sequential* implementation. One can manage parallelism & concurrency, the last one being more flexible

Cons: Hard to understand, both at low level (libraries) and at abstract level (synchronization)

We will see only explicit parallelism & concurrency.

Processes, threads and processors

- *task*: generic term to an activity;
- *process*: a task in the context of an OS, i.e. its data structures. A process is specific for a CPU/OS;
- *multistasking*: an OS that can manage more than a single process. It can share a single CPU among different processes, or manage several CPUs with several processes;
- *multiprocessor*: a system with several CPUs. In a SMP (*Symmetric MultiProcessor*) architecture, all CPUs are peers and a process can migrate from a CPU to another one;
- *thread*: a subtask of a process. A process can have several threads; every threads inside a process share the same address space of memory. A thread is a *light task* (≈1/10).

Processes, threads and processors

- *socket*: component that provide mechanical and electrical connections between a Central Processor and the motherboad;

- *core*: a CPU of a Central Processor. A CP can have more than a single core (dual-core, quad-core...);

- *Simultaneous MultiThreading (SMT)*: technique that maps a thread into a virtual CPU to maximize the use of a core. Intel© *Hyper-Threading Technology HTT™* a proprietary implementation of SMT.
  The HTT™ is a hardware feature and need support from the OS (Linux/OS X/Windows support HTT)

Processes, threads and processors

As examples, the computers used for the paper have the motherboard K55V from ASUSTeK COMPUTER INC. has a single socket that hosts a Intel© CORE™i7-3610QM at 2.30GHz with 4 cores which implements the HTT, offering 8 virtual CPU; the motherboard T101MT always from ASUSTeK COMPUTER INC. has a single socket with a Intel© ATOM™ N450 at 1.66GHz with 1 core that still implements the HTT, offering 2 virtual CPU. The first is on a notebook running Linux and the second one on a netbook running Microsoft Windows 7 Home Premium and both the CP are soldered on the motherboard (so effectively there is not socket). Both OSs support SMP and SMT (see (Kerrisk, 2010) for Linux and (Russinovich et al., 2012b) and (Russinovich et al., 2012a) for Windows 7), but on Windows the SMP is not enabled because there is only one core.

The era of the single processor is ended.

Processes, threads and processors

Multitasking can be:

- *cooperative*: the task yields the control to other tasks (Lua cooroutines);
- *preemptive*: the OS can stop the execution of a task, usually a after a fixed amount of time (*quantum* or *time slice*), and transfert the control to another task. Linux/OS X/Windows are multitasking preemptive OSs.

Threads

Under UNIX©: POSIX Threads (Pthreads), IEEE standard 1003.1

http://pubs.opengroup.org/onlinepubs/9699919799

- Linux has an almost full support for Pthreads and the API are described in (Kerrisk, 2010) ch. 28 to 33;
- OS X has also full support for Pthreads (http://www.opengroup.org/openbrand/register/brand3591.htm)
- for Windows the Pthreads win32 (active from ≈10 years) offers an almost complete implementation https://sourceware.org/pthreads-win32/

A good tutorial is also https://computing.llnl.gov/tutorials/pthreads .

API is specified in a set of C header files ⟶ A wrapper for LuaTEX with SWIG.

Threads

First tentative 1/3

```
/* core.i */
%module core
%{
#include "pthread/sched.h"
#include "pthread/semaphore.h"
#include "pthread/pthread.h"
%}
%include "pthread/sched.h";
%include "pthread/semaphore.h"
%include "pthread/pthread.h"
}

$>swig -importall -I/usr/include
 -I/usr/lib/gcc/x86_64-linux-gnu/4.7/include/
 -lua core.i
$>gcc  -fpic -I./pthread -pthread \
     -c core_wrap.c -o core_wrap.o
$>gcc -Wall -shared -pthread core_wrap.o \
     -llua5.2 -lpthread  -o core.so
```

Threads

First tentative 2/3

```
local _core= package.loadlib("./core.so","luaopen_core")
if not(_core) then
  print("error loading _core")
  return 1
end
local pthread = _core()
for k,v in pairs(pthread) do
  print(k,v)
end
```

Threads

First tentative 3/3

```
pthread_atfork       function: 0x7f7d1f1dbbd0
pthread_attr_destroy function: 0x7f7d1f1e5860
pthread_attr_init    function: 0x7f7d1f1e59a0
pthread_barrier_destroy     function: 0x7f7d1f1dcb60
pthread_barrier_init function: 0x7f7d1f1dcca0
pthread_barrier_wait function: 0x7f7d1f1dca20
pthread_create       function: 0x7f7d1f1e96d0
pthread_detach       function: 0x7f7d1f1e5ae0
pthread_equal function: 0x7f7d1f1d9c80
pthread_exit  function: 0x7f7d1f1e9a20
:
:
```

Success — enough to continue.

Threads

Thread safety

A data/code/function is thread safe if it can used by more threads concurrently without errors in the computation.

Is LuaTEX thread safe ?

- the TEX part is not thread safe; several global vars (by design);
- the Lua state is not thread safe (it cannot be shared between threads); the source code has the lock/unlock functions currently nop;

Trivial question: LuaTEX is a program which, when executed, gives a process with a *single* thread. So *it's thread safe by definition*.

Threads

Thread safety

(Ierusalimschy, 2013) p. 251: each Lua function receives a pointer to *a* Lua state and uses exclusively this state, and this 'implementation makes Lua reentrant and ready to be used in multithreaded code' (p 251).

Thus ((Ierusalimschy, 2013) sec. 31.2): each thread creates its own Lua state and this interpreter executes a chunk of Lua code (Lua-pre-thread).

Threads

Example: setup (1/4)

```
local _core= package.loadlib("./core.so","luaopen_core")
if not(_core) then print("error loading _core") os.exit(1) end
local clock = os.clock
function sleep(n)  – seconds
  local t0 = clock() – **W A R N I N G**
  while clock() - t0 <= n do end
end
local pthread = _core()
attr  = pthread.new_pthread_attr_t_p()
pthread.pthread_attr_init(attr)
pthread.pthread_attr_setdetachstate(attr,
          pthread.PTHREAD_CREATE_JOINABLE)
t0    = pthread.new_pthread_t_p()
t1    = pthread.new_pthread_t_p()
t2    = pthread.new_pthread_t_p()
```

Threads

Example: prepare threads (2/4)

```
local code_base =
[=[
  local clock = os.clock
  function sleep(n)  – seconds
   local t0 = clock() – **W A R N I N G**
   while clock() - t0 <= n do end
  end
  local s,s1
  local th="%s"
  for i=1,%s do
   local s=os.date()
   local s1=th.."<"..tostring(s).." "..tostring(i)..">"
   io.write(s1,"\n")
   sleep(%s)
  end
]=]
local code0=string.format(code_base,"1","10","0.4")
local code1=string.format(code_base,"2","10","0.3")
local code2=string.format(code_base,"3","10","0.4")
```

Threads

Example: create and run (3/4)

```
local data = nil
local rc

rc = pthread.swiglib_pthread_create(t0,attr,-1,
                          code0,string.len(code0),nil)
rc = pthread.swiglib_pthread_create(t1,attr,-1,
                          code1,string.len(code1),nil)
rc = pthread.swiglib_pthread_create(t2,attr,-1,
                          code2,string.len(code2),nil)
pthread.pthread_attr_destroy(attr);

pthread.pthread_join(pthread.pthread_t_p_value(t0),data)
pthread.pthread_join(pthread.pthread_t_p_value(t1),data)
pthread.pthread_join(pthread.pthread_t_p_value(t2),data)
print("end")
```

Threads

Example: output (4/4)

```
2<Fri Sep 13 18:07:54 2013 1>
3<Fri Sep 13 18:07:54 2013 1>
1<Fri Sep 13 18:07:54 2013 1>
2<Fri Sep 13 18:07:54 2013 2>
3<Fri Sep 13 18:07:54 2013 2>
1<Fri Sep 13 18:07:54 2013 2>
2<Fri Sep 13 18:07:54 2013 3>
3<Fri Sep 13 18:07:54 2013 3>1<Fri Sep 13 18:07:54 2013 3>

2<Fri Sep 13 18:07:54 2013 4>
1<Fri Sep 13 18:07:54 2013 4>
3<Fri Sep 13 18:07:54 2013 4>
2<Fri Sep 13 18:07:54 2013 5>
2<Fri Sep 13 18:07:54 2013 6>
3<Fri Sep 13 18:07:54 2013 5>
1<Fri Sep 13 18:07:54 2013 5>
2<Fri Sep 13 18:07:54 2013 7>
3<Fri Sep 13 18:07:54 2013 6>
```

Threads

Problems/*Solutions*:

- a thread is not a general C or Lua function / *write your own wrapper (`pthread_create` is still present)*
- string messages is a poor mechanism / *extend the wrapper with SWIG*
- we cannot use the typesetting engine of LuaTEX / *write a pure Lua of the functionality needed*
- create a Lua state is slow / *you can choose the standard modules*
- there can be some not thread safe functions hidden in standard modules / *read the documentations, the source code...*

Threads

First conclusion:

- simple experiments work (Linux/Windows 32 bit);
- Lua-per-thread useful to manage external resurces;

But:

- still to check OS X/windows 64 bit;
- hard to control, many low level details, C p.o.v vs. Lua p.o.v, no ConTEXt Lua document.

Tasks

ZeroMQ (ZMQ): C library for Message passing — exchange data between processes, or threads inside processes.

It's currently used by CERN to update their Controls Middleware system software previously based on the CORBA middleware (Dworak et al., 2012);

> "We are migrating an infrastructure with 3500 active servers and 1500 active clients from CORBA to ZMQ. We only provide the library, our users implement the clients/servers on top of it and deploy were/how they want. We also have many combinations of hardware/OS: low level front-ends, middle tier servers and workstations."

(see http://comments.gmane.org/gmane.network.zeromq.devel/18437)

Tasks

It's a generalization of the unix socket: tcp socket (`tcp://<ipv4>:<port>`), thread socket (`inproc://<name>`).

Typical patterns are:

- Request-reply: connects a set of clients to a set of services.
- Publish-subscribe: connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- Push-pull: connects nodes in a fan-out / fan-in pattern that can have multiple steps, and loops. This is a parallel task distribution and collection pattern.
- Exclusive pair: connects two sockets in an exclusive pair. This is a low-level pattern for specific, advanced use cases and still experimental.

(see http://en.wikipedia.org/wiki/Messaging_pattern.)

Tasks

In LuaTEX we can have different processes by running a program in different shells or by spawning process from the same shell (not exactly the same), as in

```
os.execute(
string.format("start 'worker' /b luatex.exe  'server.lua' '%s'
2>&1",arg))

os.execute(
string.format("luatex 'server.lua' '%s'  2>&1 &",arg))
```

Tasks

SWIG wrapper for ZeroMQ

```
%module core
%{
#include "zeromq/zmq.h"
%}
%include "zeromq/zmq_utils.h" ;
%include "zeromq/zmq.h" ;

LUAINC52=/usr/include/lua5.2
LIBS="-lpthread -lzmq"
CFLAGS="-g  -O2 -Wall -I./zeromq -pthread"
swig  -lua core.i
rm -vf core_wrap.o
gcc   -O2 -fpic -I./zeromq -I$LUAINC52 \
      -c core_wrap.c  -o core_wrap.o
rm -vf core.so
gcc -Wall -shared -O2 -Wl,-rpath,'$ORIGIN/.'\
    $CFLAGS \
    core_wrap.o  \
    -llua5.2 $LIBS \
    -o core.so
```

Tasks

ZeroMQ is a good library to build you own concurrent application: It has no (http, ftp ...) servers, but you can use it to build them.

An idea for ConTEXt: use for a job a pool of identical MKIV instances coordinated with ZeroMQ on a tcp socket. For example the typesetting of chapters could be done with a push-pull pattern.

Pros: you can use the full power of MKIV, the processors you want, it's robust.

Cons: it's slow compared to threads.

Tasks

PUSH-PULL process version:

```
$ time ./luatex ex008-PUSH_PULL-process.lua \
   >/dev/null
real   0m12.120s
user   0m0.196s
sys    0m0.852s
```

PUSH-PULL thread version:

```
$ time ./luatex ex008-PUSH_PULL-joined.lua \
   >/dev/null
real   0m0.712s
user   0m0.064s
sys    0m4.952s
```

i.e. approximately 17 faster.

A simple experiment: plotting the zeros of a polynomial

Easy to explain:

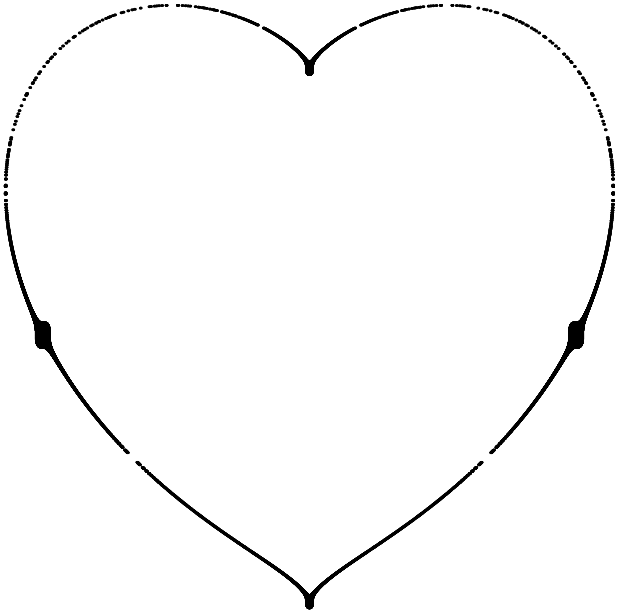
1. a parallel evaluation of a polynomial $P[X, Y]$ is trivial (?) ;
2. $\forall x_0 : -|M| \leq x_0 \leq |M|$ a thread T_{x_0} check if $\forall y : -|M| \leq y \leq |M|$, $P[x_0, y] \leq \epsilon$;
3. $\forall y_0 : -|M| \leq y_0 \leq |M|$ a thread T_{y_0} check if $\forall x : -|M| \leq x \leq |M|$, $P[x, y_o] \leq \epsilon$;
4. plot the result with MetaPost;

Plotting the zeros of a polynomial

Serial version:

```
\starttext
\startluacode
local M  = 4*1024 –256 – even
local Min, Max = -1.5,1.5
local data = {}
local P = function(X,Y) return (X^2+9/4*0^2+Y^2-1)^3-X^2*Y^3-9/80*0^2*Y^3 end
local eps = 0.0001
local M  = 4*1024 –256 – even
local _F = string.format
context("\\startMPpage")
context("pickup pencircle scaled 1;")
for i=1,M do
 local v_i =((i-1)*(Max-Min)/(M)) +Min
 for j=1,M do
   local v_j = ((j-1)*(Max-Min)/(M)) +Min
   if math.abs(P(v_i,v_j))<eps then context(_F("draw (%s,%s) scaled 100;",
v_i,v_j)) end
   if math.abs(P(v_j,v_i))<eps then context(_F("draw (%s,%s) scaled 100;",
v_j,v_i)) end
 end
end
context("\\stopMPpage")
\stopluacode
\stoptext
```

Plotting the zeros of a polynomial



“tenth anniversary”

Plotting the zeros of a polynomial

Parallel version 1/5:

```
local _core= package.loadlib("./core.so","luaopen_core")
if not(_core) then print("error loading _core") os.exit(1) end


local _helpers= package.loadlib("./helpers/core.so","luaopen_core")
if not(_helpers) then print("error loading _core") os.exit(1) end
local helpers=_helpers()


local pthread = _core()
attr  = pthread.new_pthread_attr_t_p()
pthread.pthread_attr_init(attr)
pthread.pthread_attr_setdetachstate(attr, pthread.PTHREAD_CREATE_JOINABLE)


local tx,ty =  {},{}
local M  = 4*1024
for i=1,M do
 tx[i] = pthread.new_pthread_t_p()
 ty[i] = pthread.new_pthread_t_p()
end
```

Plotting the zeros of a polynomial

Parallel version 2/5:

```
local code_base =
[=[
   local _core= package.loadlib("./core.so","luaopen_core")
   if not(_core) then print("error loading _core") os.exit(1) end
   local pthread = _core()


  local _helpers= package.loadlib("./helpers/core.so","luaopen_core")
  if not(_helpers) then print("error loading _core") os.exit(1) end
  local helpers=_helpers()


  local P=function(X,Y) return %s end
  local y_axis,X0 = %s ,%s
  local x_axis,Y0 = %s, %s
  local Min, Max = %s , %s
  local M,id = %s , %s
  local eps =%s


  local array = nil
  if swiglib_pthread_data~=nil then
    array = pthread.lightuserdata_touserdata_double_p(swiglib_pthread_data)
  end
  local data={}
```

Plotting the zeros of a polynomial

Parallel version 3/5:

```
  if y_axis then
     for i=1,M do
        local v= ((i-1)*(Max-Min)/M) +Min
        if array~=nil then
          if math.abs(P(X0,v))<eps then helpers.double_array_setitem(array,i-1,v)
else helpers.double_array_setitem(array,i-1,2*Min) end
        end
        –data[#data+1]=P(X0,v)
        –print(i-1,X0,".y=",data[#data])
     end
     return
  end

  if x_axis then
     for i=1,M do
        local v= ((i-1)*(Max-Min)/M) +Min
        if array~=nil then
         if math.abs(P(v,Y0))<eps then helpers.double_array_setitem(array,i-1,v)
else helpers.double_array_setitem(array,i-1,2*Min) end
        end
        –data[#data+1]=P(v,Y0)
        –print(i-1,Y0,".x=",data[#data])
     end
     return
  end
]=]
```

Plotting the zeros of a polynomial

Parallel version 4/5:

```
local Min, Max = -1.5,1.5
local code_x, code_y , v= {},{},{}
local formula = "(X^2+9/4*Y^2+Z^2-1)^3-X^2*Z^3-9/80*Y^2*Z^3"
formula = string.gsub(formula,'Y','(0)')
formula = string.gsub(formula,'Z','Y')
local eps = "0.0001"

for i=1,M do
 v[i] =((i-1)*(Max-Min)/(M)) +Min
 code_x[i] = string.format(code_base,formula, "true",v[i],"false","nil",
Min,Max,M,i,eps)
 code_y[i] = string.format(code_base,formula, "false","nil","true",v[i],
Min,Max,M,i,eps)
end

local rc = nil
local array_x,array_y={},{}
for i=1,M do
 array_x[i]=helpers.new_double_array(M)
 array_y[i]=helpers.new_double_array(M)
 pthread.swiglib_pthread_create(tx[i],attr,-1,code_x[i], string.len(code_x[i]),
pthread.double_p_to_void_p(array_x[i]) )
 pthread.swiglib_pthread_create(ty[i],attr,-1,code_y[i], string.len(code_y[i]),
pthread.double_p_to_void_p(array_y[i]) )
end
pthread.pthread_attr_destroy(attr);
```

Plotting the zeros of a polynomial

Parallel version 5/5:

```
for i=1,M do
 pthread.pthread_join(pthread.pthread_t_p_value(tx[i]),nil)
 pthread.pthread_join(pthread.pthread_t_p_value(ty[i]),nil)
end

for i=1,M do
 local x = array_x[i]
 local y = array_y[i]
 for j=0,M-1 do
  if helpers.double_array_getitem(x,j) > 2*Min then
     print(v[i],helpers.double_array_getitem(x,j))
  end
  if helpers.double_array_getitem(y,j) > 2*Min then
    print(helpers.double_array_getitem(y,j),v[i])
  end
 end
end
```

Plotting the zeros of a polynomial

Results:

```
$time luatex  heart-serial.lua >zs
real    0m18.955s
user    0m18.869s
sys     0m0.068s

$ time luatex heart-parallel.lua >z
real    0m13.453s
user    0m53.327s
sys     0m14.441s
```

It seems that $S = \frac{18.955}{13.453} = \approx 1.41$ but...

Plotting the zeros of a polynomial

Surf:

```
surf version 1.0.6 - visualizing algebraic curves and algebraic
surfaces
Copyright (C) 1996-1997  Friedrich-Alexander-Universitaet
Erlangen-Nuernberg,
1997-2000 Johannes Gutenberg-Universitaet Mainz.
Authors: Stephan Endrass, Hans Huelf, Ruediger Oertel, Ralf
Schmitt,
Kai Schneider and Johannes Beigel.

For reporting bugs or getting news about latest developments,
please visit our homepage at http://surf.sourceforge.net/
```

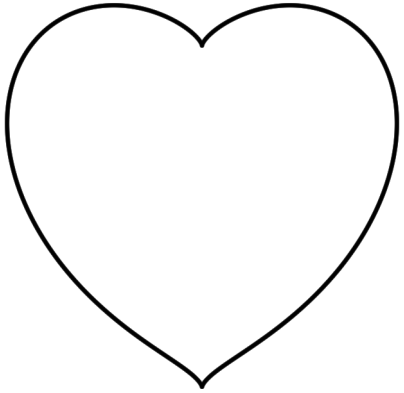
Plotting the zeros of a polynomial

Surf:

```
width =800;
height=800;
curve_red  =255;
curve_green=255;
curve_blue =255;
curve_width=2;
curve_gamma=3;
curve =  (x^2+9/4*0^2+y^2-1)^3-x^2*y^3-9/80*0^2*y^3;
scale_x=0.2;
scale_y=0.2;
color_file_format = ppm;
clear_screen;
draw_curve;
filename="heart" ;
save_color_image;
```

Plotting the zeros of a polynomial

Surf:



Plotting the zeros of a polynomial

Surf:

```
$ time surf heart.pic && convert -negate -density 300x300
heart.ppm heart.pdf

************** executing heart.pic
executing script...
saving color image...

real0m0.084s
user0m0.076s
sys0m0.004s
```

This serial version is 160 times **faster** than the “trivial” parallel version...

Conclusion

These are experiments, so don't take them as examples. They show that something of concrete is possible without change the codebase of LuaTEX.

On the otherside:

1. Concurrency is hard to control even theoretically;
2. It demands unusual skills;
3. Libs introduce dependencies. It's not a problem for a private application: It's a big problem for a community wide (open) distribution. Probably a solution is not deploy libs, but promote pattern a tools to build solutions with libs (Swiglib project).

Current code (Linux x86_64 only) is under the section experimental at https://swiglib.foundry.supelec.fr

Short bibliography

Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc..

Hennessy, J. L. and Patterson, D. A. (2012). *Computer Architecture — A Quantitative Approach*. 5th edition Morgan Kaufmann.

Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st edition San Francisco, CA, USA: No Starch Press.

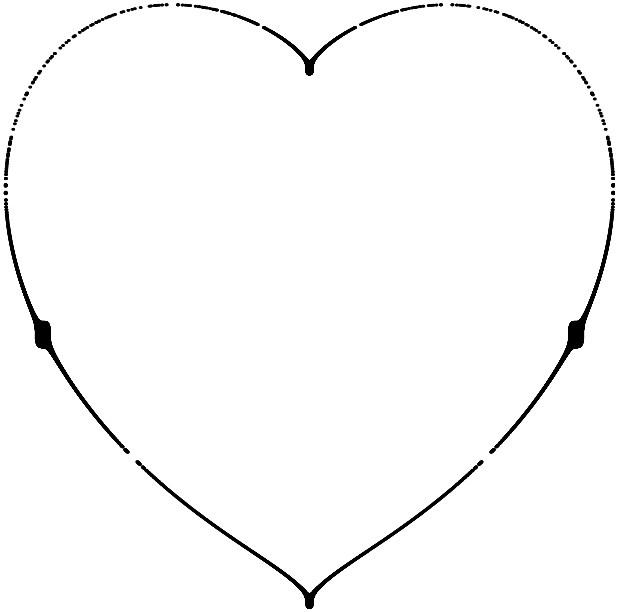
Russinovich, M. E., Solomon, D. A. and Ionescu, A. (2012b). *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. 6th edition Microsoft Press.

Russinovich, M. E., Solomon, D. A. and Ionescu, A. (2012a). *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7 (Windows Internals)*. Microsoft Press.

Ierusalimschy, R. (2013). *Programming in Lua, Third Edition*. Lua.Org.

Dworak, A., Ehm, F., Charrue, P. and Sliwinski, W. (2012). The new cern controls middleware. *Journal of Physics: Conference Series, 396*(1), 012017.

That’s all folks !
Thank you !



“tenth anniversary”